Designing Functional Implementations of Graph Algorithms

Dipl.-Math. Nikita Danilenko

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2015

1. Gutachter:	Prof. Dr. Rudolf Berghammer Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Priv.-Doz. Dr. Frank Huch Christian-Albrechts-Universität zu Kiel

Datum der mündlichen Prüfung: 11. Dezember 2015

Zusammenfassung

Diese Dissertation beschäftigt sich mit der Entwicklung von Algorithmen auf Graphen in funktionalen Programmiersprachen am Beispiel von Haskell. Klassische Graphalgorithmen werden üblicherweise in imperativen Programmiersprachen beschrieben und analysiert. Imperative Programmiersprachen eignen sich besonders, um Programmabläufe zu beschreiben, in welchen die Reihenfolge der Operationen entscheidend ist. Dies betrifft insbesondere die schrittweise, in der Regel destruktive Veränderung von Objekten. Solche Iterationen kommen häufig im Falle von Optimierungsproblemen auf Graphen vor. In der funktionalen Programmierung abstrahiert man von einer festen Berechnungsreihenfolge und beschreibt Problemlösungen als Kompositionen von Teillösungen. Ferner sind funktionale Programmiersprachen referentiell transparent, sodass destruktive Veränderungen nur bedingt möglich sind.

Die Entwicklung rein funktionaler Graphalgorithmen setzt bei der Zerlegung der bestehenden Probleme in einfachere Probleme an. Oftmals können Lösungen dieser Teilprobleme auch in anderen Situationen eingesetzt werden. Darüber hinaus erlaubt es diese Kompositionalität, einzelne Funktionen mit wenig Aufwand durch beispielsweise effizientere oder verständlichere Fassungen auszutauschen.

Als Zwischenschritt in der Entwicklung wird in dieser Dissertation ein algebraischer Ansatz verwendet, welcher zu großen Teilen auf der Relationenalgebra im Sinne von Tarski beruht. Es gibt zwei wichtige Vorteile dieses Ansatzes. Der erste Vorteil ist die Formalität der entstehenden Spezifikationen. Durch die Einschränkung auf wenige Operationen und Strukturen sind Lösungen weniger fehleranfällig und können zusätzlich mit Hilfe von Beweisassistenten verifiziert werden. Der zweite Vorteil ergibt sich daraus, dass die Spezifikation ausführbar ist, sobald die notwendigen Basisoperationen implementiert sind. Die Grundidee des Ansatzes besteht also darin, eine (möglicherweise umgangssprachliche) Lösung zunächst in eine rein algebraische zu übertragen und diese dann zu implementieren. In vielen Fällen sind die entstehenden Ausdrücke trotz ihrer Formalität leserlich, weil die algebraischen Operationen natürlichsprachige Interpretationen zulassen.

In dieser Dissertation werden Grundlagen einer möglichen Implementierung des oben erwähnten algebraischen Ansatzes in der funktionalen Programmiersprache Haskell behandelt. Ausgehend hiervon werden exemplarisch einige Probleme der Graphentheorie gelöst. Beispielsweise werden kardinalitätsmaximale Matchings und kardinalitätsminimale Knotenüberdeckungen in bipartiten Graphen untersucht. Darüber hinaus werden Fragestellungen aus dem Bereich der sozialen Netzwerke betrachtet und bekannte Lösungen mit den vorgestellten Mitteln umgesetzt. Schließlich werden Optimierungen der vorgestellten Implementierungen und weitere Probleme, welche mit den obigen Methoden gelöst werden können, diskutiert.

Abstract

This dissertation deals with the development of graph algorithms in functional programming, using the example of Haskell. Classic graph algorithms are usually presented and analysed in imperative programming languages. Imperative programming languages are well-suited for the description of a program flow, in which the order in which the operations are performed is important. One common example of such a description is the successive, typically destructive modification of objects. This kind of iteration often occurs in the context of graph algorithms that deal with a certain kind of optimisation. In functional programming, the order of execution is abstracted and problem solutions are described as compositions of intermediate solutions. Additionally, functional programming languages are referentially transparent and thus destructive updates of objects are discouraged.

The development of purely functional graph algorithms begins with the decomposition of a given problem into simpler problems. In many cases the solutions of these partial problems can be used to solve different problems as well. What is more, this compositionality allows exchanging functions for more efficient or more comprehensible versions with little effort.

An algebraic approach with a particular focus on relation algebra as defined by Tarski is used as an intermediate step in this dissertation. This approach comes with two important advantages. The first one is the formality of the resulting specifications. Since one is restricted to a small set of operations and structures, the resulting solutions are less error-prone and can be verified using proof assistants. The second advantage is that the specification is executable, once the necessary operations are implemented. The basic idea of the above approach is to take a (possibly informal) solution, translate it into a purely algebraic one, and then implement the latter. Despite their formality, the resulting expressions are still readable, because the algebraic operations have intuitive interpretations.

This dissertation presents the basics of the algebraic approach in the functional programming language Haskell. Using this foundation, some exemplary graph-theoretic problems are solved in the presented framework. These solutions include those for the maximum matching problem and the minimum vertex cover problem in bipartite graphs. Furthermore, problems from the field of social network analysis are considered and it is shown, how established solutions can be implemented with the above means. Finally, optimisations of the presented implementations are discussed and pointers are provided to further problems that can be solved using the above methods.

Acknowledgements

I am grateful to my supervisor Rudolf Berghammer for having me in his research group. Thank you for giving me the freedom to research on my own, guidance and pointers on virtually any topic, and for many (heated) discussions over the years. Many thanks to Insa Stucke for being a good colleague and for always being open for debate.

Hats off to Jan Christiansen for his insatiable drive for deeper understanding, which greatly improved my scientific presentation. Thanks to Fabian Reck and Björn Peemöller for all kinds of technical insights into programming languages. I thank Sandra Dylus for discussing the presentation of several parts of this work and for helping me with all kinds of technical issues. Furthermore, thanks to all participants of the weekly workshop of Rudolf Berghammer and Michael Hanus for giving me the opportunity to present my work in progress and for sparking additional ideas; the members of this workshop consist of all the previously mentioned persons, as well as Stefan Bolus, Sebastian Fischer, Sebastian Hanowski, Frank Huch, Hans Langmaack, Henning Schnoor, and Jan Rasmus Tikovsky.

I greatly appreciate the help of Anja Hildebrandt, Sandra Dylus, Insa Stucke, Jan Christiansen, Sebastian Preugschat, and Yannik Potdevin, who have read drafts of this work and provided honest and insightful feedback. I thank all (former) colleagues from my floor, which, in addition to those already mentioned above include Maike Bradler, Jane Eitzen, Mike Gabriel, Linda Haberland, Ulrike Pollakowski, Ina Pfannschmidt, Renate Staecker, Aenne Straßner and Carolina Wandt; thank you for providing a very pleasant working environment.

Finally, I am very thankful for the opportunity to teach computer science and mathematics to students in practical courses and to the many students I have taught during my time at the university. The countless questions concerning many different topics have greatly increased my understanding of both basic and sophisticated concepts.

Contents

1 Introduction **1**

2 Preliminaries **7**
2.1 Organisational Remarks 7
2.2 Notations and Prerequisites 7
2.3 Graphs and Their Algebraic Representation 10
2.4 Relations and Matrices 17
2.5 Haskell and Functional Programming 22
2.6 Graph Implementations 24

3 Kleene Closures – A Case Study **27**
3.1 Introduction 27
3.2 Algebraic Preliminaries 28
3.3 A Functional Approach 33
3.4 Application to Square Matrices 39
3.5 A Functional Implementation 44
3.6 Alternative Implementations 53
3.6.1 Arrays 53
3.6.2 Lists Revisited 54
3.6.3 Blockwise Computation 54
3.7 Complexity and Comparison 55
3.7.1 Kleene Closure Complexity 55
3.7.2 The Setting of Random Tests 56
3.7.3 Kleene Closure Measurements 57
3.8 Related Work and Discussion 60

4 Graph Algebra and its Generalisation **63**
4.1 Graphs, Vertices and their Algebraic Model 63
4.2 Rearranged Multiplication and its Implementation 67
4.2.1 Abstracted Sum and Sum Generation 69
4.2.2 Scalar Multiplication and Construction 71
4.2.3 Generalised Vector-Matrix Multiplication 72
4.2.4 Requirements on Vector and Matrix Representations 73
4.3 Applications 74
4.3.1 Number-like multiplication 75
4.3.2 Discrete Successors 77
4.3.3 Check for Successors 78

4.3.4 Extending Walks by One Step 79
4.3.5 Outgoing Values and Transposition 81
4.4 Multiplication Sequences and Reachability 85
4.5 Finding Disjoint Shortest Paths 90
4.6 Type Classes for Set Operations 94
4.7 Correctness Proofs 98
4.8 Related Work and Discussion 105

5 Bipartite Maximum Matching **107**
5.1 Specification and Solution 107
5.2 Searching for Augmenting Paths 111
5.3 The Bipartite Maximum Matching Function 117
5.4 Maximum Matching with Disjoint Paths 121
5.5 Testing Symmetry and Bipartiteness 123
5.5.1 Testing Symmetry 124
5.5.2 Testing Bipartiteness: A Simple Approach 124
5.5.3 Testing Bipartiteness: An Efficient Approach 126

6 Bipartite Minimum Vertex Cover **133**
6.1 Specification and Solution 133
6.2 Proof of Kőnig's Theorem 136
6.3 The Bipartite Minimum Vertex Cover Function 151
6.4 The Non-Bipartite Case 152

7 Maximum Flows and Minimum Cuts in Networks **155**
7.1 Specification and Solution 155
7.2 The Maximum Flow Function 161
7.3 Minimum Cuts in Networks 168
7.4 Related Work and Discussion 169

8 Algebraic Problems **171**
8.1 Approval Voting 171
8.2 Balanced Graphs and Clustered Graphs 179
8.3 The Centrality Problem 184
8.4 Related Work and Discussion 190

9 Conclusion **193**
9.1 Summary 193
9.2 Future Work 196
9.2.1 Functional Programming 196
9.2.2 Functional Logic Programming 202

Bibliography **205**

A Proofs **213**
A.1 Proofs from Chapter 3 213
A.2 Proofs from Chapter 5 218
A.3 Proofs from Chapter 7 222

Nomenclature **225**

Index **239**

Chapter 1

Introduction

In this thesis we consider graph algorithms in the functional programming language Haskell [Mar09]. When it comes to graph algorithms, estimates of their complexities, and hints for improvements of the provided implementations, authors usually provide sequentially structured code [Bat94; Din06; Edm65; HK73; KT06], which is similar to actual code in imperative languages like Pascal or C. The idea behind these algorithms is to provide an executable means as to *how* to find the solution of a certain problem. Functional languages, not just Haskell in particular, abstract from this view of problem solutions and attempt to solve problems by exactly specifying *what* to compute. Clearly, even in functional programming an actual implementation is important, but typical applications try to separate and modularise the components of a solution, such that intermediate components are both easily exchangeable and applicable in other cases.

Although it is obviously possible to simply translate a sequential, imperative program into a functional one, this comes with two downsides. First, the resulting code would probably be considered not good practice by most functional programmers, due to lack of abstraction and modularity. Second, more importantly, some constructs from imperative programming come with additional costs in functional languages, when implemented directly. One prominent example are arrays: in imperative programming these data structures are ubiquitous and provide constant time read and write access to all of their entries. In functional programming, such data structures are usually not supported directly, because they break the *referential transparency*: using the same function with the same arguments always yields the same result. In Haskell there are several types of arrays and those that in fact allow constant time read and write access reside in a special type of monad, which encapsulates a local world and thus allows the updates locally without breaking the referential transparency globally. Arrays that can be used without monads provide fast access, but only very slow update functions, so that applications that are based on repeated array updates are likely to become more complex, when implemented without a proper functional refactoring.

In the context of graph algorithms one often finds references to other algorithms like a path search with a given strategy, but at the same time, these references only indicate the general similarity, because the actual algorithm needs to be rewritten to fit the given purpose. A functional solution to such a problem is to add an additional argument to the function that contains the actual difference in the concrete application

and to implement the function itself generically, while possibly relaxing the restricted type as well. For example, an implementation of a function

$$sortAscending :: [Integer] \rightarrow [Integer]$$

that sorts a list of integers in ascending order varies from the function

$$sortDescending :: [Integer] \rightarrow [Integer]$$

that sorts a list of integers in descending order only in the fact that one uses the comparison function $(\leqslant)$ in the first case and $(\geqslant)$ in the second case. One simple abstraction of this similarity is to define a function

$$sortBy :: (Integer \rightarrow Integer \rightarrow Bool) \rightarrow [Integer] \rightarrow [Integer]$$

which sorts a list of integers using the abstract comparison function that is now supplied as an argument. Then one has

$$\begin{aligned} sortAscending &= sortBy\ (\leqslant) \\ sortDescending &= sortBy\ (\geqslant) \end{aligned}$$

which is both simpler and more general, because we can now sort lists with respect to other orders, too. As a final refactoring step one can then realise that the function is actually more general than expected and abstract it to

$$sortBy :: (\alpha \rightarrow \alpha \rightarrow Bool) \rightarrow [\alpha] \rightarrow [\alpha]$$

which can deal with arbitrary data types and not just integers. Such abstractions are common practice in functional programming and often provide insights into the actual computational structures that are employed in particular applications. This technique can be applied to graph theory as well. For example, it is a well-known fact in graph theory that the algorithm for finding all vertices that are reachable from a given vertex depends on the data structure in which the successor candidates are maintained: using a stack results in a depth-first search, while using a queue yields a breadth-first search. The different data types can be abstracted into a type class and then specified in a similar fashion as above for the sorting functions.

Graph algorithms in functional programming have been considered before on several occasions [KW91; KL95; Joh98; RL99; Erw01; Ber11; Dol13]. The ingenious approach to depth-first search by King and Launchbury [KL95] showed that graph algorithms in functional languages can benefit from local, imperative features and thus become as asymptotically complex as their imperative counterparts. Most importantly, King and Launchbury [KL95] use these features in precisely one function, while all applications are written in the usual, compositional style. The work of Erwig [Erw01] deals with a flexible approach to graphs, which involves graph modification by adding vertices or edges and is based upon an inductive approach to graphs. This approach is particularly interesting for various reasons. From the perspective of graph theory, many inductive constructions and proofs can be translated into this framework. From a functional programming angle, inductive structures are well-known and familiar, so that graphs behave similar to other structures in the language.

From an implementational point of view, some algorithms, most notably the depth-first search and the breadth-first search become particularly simple, because the inductive decomposition of graphs allows elegant implementations of both algorithms without the need to consider previously visited vertices. This comes at the price of an additional logarithmic factor in the asymptotic complexity estimates, when compared to the corresponding imperative algorithms. The results of Kashiwagi and Wise [KW91] and Dolan [Dol13] provide elegant implementations of graphs represented as (densely filled) matrices, where Dolan considers those algorithms in particular that are based on the computation of a certain Kleene algebra operation. All functions presented in these sources are purely functional, but the density of the matrices leads to high space consumptions and overestimates the number of edges by the square of the number of vertices asymptotically. Johnsson [Joh98] considers certain graph algorithms that are implemented by constructing recursively defined lazy arrays with an externally defined primitive. Finally, Berghammer [Ber11] shows how a variety of graph problems that are expressible in terms of relation algebra can be implemented in Haskell using a purely functional implementation of the transitive closure function.

One particular computation scheme that is applicable to a variety of graph problems is not found in any of the previously mentioned related works, namely the computation of successors of a set of vertices, while at the same time collecting some information along the way. This kind of computation is well-known in the theoretical contexts of semirings or Kleene algebras [Con71; Koz90; Koz94]. Matrices over these algebraic structures, which belong to the same category of algebraic structures again, can be used to model graphs in terms of their adjacency matrices. Sets of vertices can then be modelled using vectors and successors can be computed in terms of a multiplication of these vectors with the adjacency matrix. Depending on the chosen underlying structure, the result vector contains certain additional information, like extended paths or the minimum edge weight along a given path. In relation algebra, the use of (relational) vectors and the multiplication of these vectors with relations is a well-established tool for the computation of successors, reachable vertices and other set-based results. Since relation algebra is known to be a great tool for reasoning and calculating in graphs [SS93], many results from this field can be incorporated once the necessary framework is established.

We study this type of vector-matrix multiplications in detail and abstract it as far as possible. The result of this abstraction is a simple higher-order function, which we use for the solution of numerous problems. One important observation in this abstraction is that the relational context, namely the actual successors, is maintained in most cases. This context allows us to use relational means to reason about (intermediate) results and functions. In the course of this approach we derive purely (relation) algebraic solutions for certain problems. The great benefit of this algebraically flavoured approach is the fact that once the required algebraic operations are implemented, algebraic specifications usually become executable. Additionally, functions that are

based upon these operations and other high-level interface functions are automatically independent of the actual underlying implementation. Overall, the resulting code is rather declarative and in many cases abstract compositions of certain algorithms in theory can be realised as an actual function composition.

An additional benefit of the algebraic approach is that despite the restriction of the operations to a small set, these operations are often human-readable, so that algebraic specifications can be understood with little effort, while at the same time being more concise than informal definitions. Conversely, in many cases informal definitions allow direct translations into algebraic statements by simply replacing the terminology, for instance replacing a statement about the successors of a vertex set by a multiplication of a vector with a matrix. We prove several results of this type, some of which have not been proved using only algebraic means before.

Once we have established the basic components of our framework, we use it for the solution of some classical problems from graph theory. While some of the resulting functions have a prototypical look-and-feel, we discuss modifications and optimisations as well. The main goal of this dissertation is to show how graph algorithms can be designed in Haskell using some algebraic means and concepts from functional programming in general. We do not aim for maximal efficiency, but discuss some complexities informally along the way. One of the main reasons is that in cases in which our approach is less efficient than an imperative one, the inefficiency can be reduced (or removed entirely) by using different data structures. Since we wish to present an abstract functional approach to graph algorithms, we use prototypical data types for the sake of presentation and mention more efficient choices later.

The dissertation itself is structured as follows.

- ▻ Chapter 2 provides the main definitions and algebraic structures that are used in this text. Additionally, it provides some organisational remarks including notations used in the remainder of this text and some Haskell preliminaries.

- ▻ In Chapter 3 we derive a purely functional implementation for the computation of the Kleene closure of matrices over Kleene algebras. Since this is a classical example of a non-trivial graph algorithm, we use the results of this chapter as a starting point for later implementations.

- ▻ Chapter 4 presents the abstraction of the vector-matrix multiplication and some simple tools for the construction of such multiplications. We provide several examples in this chapter, some of which are non-trivial. Also, we discuss a simple reachability scheme that takes a vector-matrix multiplication and computes reachability layers with additional information. Finally, we discuss how the framework can be abstracted and possible reasoning about functions in this framework.

- ▻ Using the tools from Chapter 4 we consider the bipartite maximum matching problem in Chapter 5. We present a canonic algorithm and a more efficient one.

Additionally, we consider the problem of checking whether a graph is bipartite. Aside from a pure predicate for this test, we also implement a more efficient function, which actually computes a bipartition in case the graph is bipartite.

- In Chapter 6 we solve the bipartite minimum vertex cover problem functionally. While the actual solution in relation algebra and the corresponding implementation in Haskell are comparatively simple, the purely relational proof of the correctness of the construction is quite technical. We present the proof, since it has not been published in the version given in Chapter 6 before.
- Chapter 7 deals with maximum flows and minimum cuts in networks. We discuss the relation to the maximum matching problem and outline the well-known solution of the latter in terms of the former.
- In Chapter 8 we consider some problems that can be expressed with purely algebraic means. These problems include a special type of manipulation in certain voting games and the computation of the betweenness centrality of a vertex.
- We conclude this thesis by a short summary of our results and a discussion of possible topics for future research.

Finally, we omit some proofs along the way due to their length or technicality. For the sake of completeness we provide the most important ones in Appendix A.

Chapter 2

Preliminaries

2.1 Organisational Remarks

Before we begin, we would like to make some remarks concerning the structure of this document. Every major part is subdivided into chapters and sections. All mathematical components like definitions and theorems are located at the third level, and these components are numbered sequentially in every enumeration level. Thus if we reference Proposition 3.7.5, said proposition can be found in chapter 3, section 7 and is the fifth mathematical component in this section. Some proofs are located in the appendix, where the chapter number is changed to an uppercase letter.

Additionally, we maintain an equation labelling that is simple to reference and to recover in the text. All equations that are mentioned in between mathematical statements or proofs are numbered with the number of the last mathematical statement before that equation and a consecutive roman number. This way Equation (2.2.5.ii) is the second equation that is mentioned after the mathematical component with the number 2.2.5, but before the component 2.2.6. Similarly, auxiliary equations in proofs are numbered using the number of the mathematical statement that is being proved followed by a consecutive letter. This means that Equation (5.2.3.a) is the first equation in the proof of the mathematical statement 5.2.3.

In this dissertation we provide several definitions, notations, and function implementations. To aid the reader there are three glossaries, which show where the respective term has been defined. In the nomenclature (page 225ff.) we list two groups: mathematical notations and Haskell functions. Both groups contain a short description of the corresponding entry and a page reference to the notation or function definition. The index (page 239f.) lists where mathematical terms are defined.

2.2 Notations and Prerequisites

In this section we briefly summarise all (possibly non-standard) notations we use throughout this book.

▻ The set $\mathbb{N}$ is the set of natural numbers containing 0. The sets $\mathbb{Z}, \mathbb{Q}$ and $\mathbb{R}$ denote the integers, the rational numbers, and the real numbers respectively. For every

$X \in \{ \mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R} \}$, every $b \in X$ and every $\preccurlyeq \subseteq X \times X$ we write

$$X_{\preccurlyeq b} := \{ x \in X \mid x \preccurlyeq b \} \ .$$

Thus $\mathbb{N}_{<n}$ is the set of all natural numbers that are strictly smaller that n, while $\mathbb{R}_{>0}$ is the set of the positive real numbers.

- Let A, B, C be sets, $S \subseteq A \times B$ and $f : S \to C$. Suppose that we have $a \in A$ and $b \in B$ such that $(a, b) \in S$. We then write $f(a, b)$ instead of $f((a, b))$ for simplicity.
- If A, B, C are sets and $f : A \times B \to C$, we use

$$f(a_0, -) : B \to C \ , \quad b \mapsto f(a_0, b) \ ,$$
$$f(-, b_0) : A \to C \ , \quad a \mapsto f(a, b_0)$$

to denote the partial applications of f to $a_0 \in A$ and $b_0 \in B$ respectively.

- We use the mathematical λ-notation for function definition. Thus we always write

$$f : A \to B \ , \quad a \mapsto E$$

where E is some expression. Occasionally, we omit the type of the function, if it can be reconstructed from the context.

- For all sets S, R, C we denote with $S^{R \times C}$ the set of all $R \times C$ matrices over S, where a matrix $a \in S^{R \times C}$ is a mapping

$$a : R \times C \to S \ .$$

We use the family notation for matrices and omit the index brackets, which is to say that we write

$$a_{r,c} := a_{(r,c)} := a(r, c)$$

for all $r \in R$ and $c \in C$. In the special case that $R = \mathbb{N}_{<n}$ and $C = \mathbb{N}_{<m}$ for some $n, m \in \mathbb{N}$, we write $S^{n \times m}$ instead of $S^{\mathbb{N}_{<n} \times \mathbb{N}_{<m}}$ and say that the matrix has n rows and m columns. For every $r \in R$ we define

$$a_r : C \to R \ , \quad c \mapsto a_{r,c}$$

and call it the *r-th row of a*. Note that this is merely a partial application of a to r.

- Let A be a set and $n \in \mathbb{N}$. Then every $x \in A^n$ is actually a function $x : \mathbb{N}_{<n} \to A$. We call x a *vector of length n over A*. If n and A are clear from the context, we write

$$x = i \mapsto x_i \ .$$

When necessary, we write vectors in list notation

$$x = (x_0, \ldots, x_{n-1}) \ .$$

The positions in this notation denote the value of x at that position. For every $a, b \in \mathbb{N}_{<n}$ such that $a \leqslant b$ we define

$$(x_k)_{k=a}^{b} : \mathbb{N}_{<b-a+1} \to A \ , \quad j \mapsto x_{j+a} \ .$$

Less formally, we have $(x_k)_{k=a}^{b} = (x_a, \ldots, x_b)$. For simplicity, we consider A^n and $A^{1\times n}$ to be the same set. This is to say that every vector is also a $1 \times n$-matrix.

▻ When explicitly accessing fixed indices of vectors, we use the projection functions. Let A, B be a sets. Then for all $j \in A$ we have

$$\mathrm{pr}_j : B^A \to B\ , \quad f \mapsto f(j)\ .$$

While this definition obviously depends on the sets A, B, we overload it to avoid unnecessary clutter. Since vectors are functions, too, the projection functions are applicable to vectors as well. In particular, for every $n \in \mathbb{N}$, every $k \in \mathbb{N}_{<n}$ and every $v \in A^n$ we have $\mathrm{pr}_k(v) = v(k) = v_k$.

▻ The symbol $\iff$ means (meta-)logical equivalence, while the symbol $\Leftrightarrow$ is the logical connective. Similarly, $\Longrightarrow$ and $\Longleftarrow$ are the metalogic symbols, while $\Rightarrow$ and $\Leftarrow$ are their syntactic counterparts.

▻ The *type* of a (homogeneous) algebraic structure is the list of the arities of its constants and functions in their order of appearance. For example the type of a unital ring $(R, +, \cdot, 0, 1)$ is the list $(2, 2, 0, 0)$.

▻ A *lattice* is an algebraic structure $(L, \sqcup, \sqcap)$ of the type $(2, 2)$, where $\sqcup$ and $\sqcap$ are both associative, commutative and satisfy the absorption laws. The *lattice order* is defined as

$$\sqsubseteq := \{\ (a, b) \in L \times L \mid a \sqcup b = b\ \}\ .$$

A lattice is called *distributive* if and only if the operations distribute over one another. It is well-known [DP02] that one distributivity law, e.g. that $\sqcap$ distributes over $\sqcup$, implies the other one as well. A lattice is called *complete* if and only if every subset A of the lattice has a supremum (denoted $\bigsqcup A$) and an infimum (denoted $\bigsqcap A$). An algebraic stucture $(L, \sqcup, \sqcap, \bot, \top)$ of the type $(2, 2, 0, 0)$ is called *complementary lattice* if and only if $(L, \sqcup, \sqcap)$ is a lattice and for each $x \in L$ there is a $y \in L$ such that

$$x \sqcup y = \top \quad \wedge \quad x \sqcap y = \bot\ .$$

For every complementary lattice we define

$$^{-L} := \{\ (x, y) \in L \times L \mid x \sqcup y = \top \wedge x \sqcap y = \bot\ \}\ .$$

The relation $^{-L}$ is total by definition. A complementary lattice $(L, \sqcup, \sqcap, \bot, \top)$ is called a *Boolean algebra* if and only if $(L, \sqcup, \sqcap)$ is distributive. Davey and Priestley [DP02] show that in any Boolean algebra the complement of any element is unique and thus the relation $^{-L}$ is a function. We usually omit the index and write $\overline{x}$ instead of $\overline{x}^L$.

▻ On several occasions we require a special set containing exactly one element. For this purpose we choose an object $\square$ and set $\mathbb{1} := \{\ \square\ \}$.

- ▻ For every set S we define
$$S^* := \bigcup_{n \in \mathbb{N}} S^n \,,$$
which is the set of all finite lists (or sequences) over S. This nomenclature clashes with the star closure of a matrix, which we denote by A^* as well in Chapter 3. However, the actual meaning can be easily reconstructed from the context, particularly because lists and closures of matrices do not appear in the same context in the remainder of the text. We use the symbol $+\!\!+$ to denote list concatenation. For every $s \in S^*$ the set
$$\left\{ k \in \mathbb{N} \;\middle|\; s \in S^k \right\}$$
contains exactly one element. Thus the function
$$|-| : S^* \to \mathbb{N}\,, \quad s \mapsto \min \left\{ k \in \mathbb{N} \;\middle|\; s \in S^k \right\}$$
is well-defined. For every $s \in S^*$ we call $|s|$ *the length of* s.

- ▻ When we use the term "relation" synonymously with "binary relation".

- ▻ We use an informal Landau notation, where we usually write $\mathcal{O}(\text{expression with } x)$ rather than more formally $\mathcal{O}(x \mapsto \text{expression with } x)$. Additionally, we define for functions $f : \mathbb{N}^2 \to \mathbb{R}_{\geqslant 0}$
$$\mathcal{O}(f) := \left\{ g \in (\mathbb{R}_{\geqslant 0})^{\mathbb{N}^2} \;\middle|\; \exists\, c \in \mathbb{R}_{>0} : \exists\, k_0 \in \mathbb{N} : \forall\, k,l \in \mathbb{N}_{\geqslant k_0} : g(k,l) \leqslant c \cdot f(k,l) \right\}$$
and thus use the same notation for binary functions. Occasionally, we reference the implicit constants in the Landau notation, which are the corresponding c in the definition.

2.3 Graphs and Their Algebraic Representation

In this section we define the graph structure and consider representations thereof in terms of an algebraic framework.

2.3.1 Definition (Graph)**.**
Let V be a non-empty set and $E \subseteq V \times V$. Then the pair (V, E) is called a *(directed) graph*. The elements of V are called *vertices* and those of E are called *(directed) edges* or *arcs*. Given a set L and an $f : E \to L$, the triple (V, E, f) is called an *L-labelled graph*. For every $e \in E$ the value $f(e)$ is called the *label of e* and f is called *label function*. ⫽

Alternatively, one could require the label set to belong to the structure of a labelled graph, such that a labelled graph is a quadruple (V, E, L, f). We do not use this definition, because on many occasions one considers an arbitrary graph with a fixed label set (cf. Convention 2.3.9 and the graphs in Section 8.3). Using our notation we can simply fix a concrete set L and assume a given L-labelled graph (V, E, f). This is

more technical in the other definition, because we have to introduce V, E, f and then require that (V, E, L, f) is a labelled graph, which reduces legibility.

We make the following observations.

(1) The edge set E is a (homogeneous) relation and thus all of relational reasoning applies to this set. In particular, we can use the relational operations of multiplication (e.g. $E \cdot E$) and transposition. Additionally, we can use relational terminology to describe graph properties, which we elaborate shortly. Usually, we do not distinguish between $G = (V, E)$ and E and apply the relational terminology to the graph itself.

(2) When we say "graph" we always mean a directed graph. Normally, one distinguishes directed graphs from undirected ones, where "undirected" means that $E \subseteq \{ \{ v, w \} \mid v, w \in V \}$. Note that in the case of undirected graphs in this sense, edges may contain a single vertex (instead of two). We model undirected graphs in the above sense as symmetric graphs, that is graphs, which satisfy the property

$$\forall v, w \in V : (v, w) \in E \Rightarrow (w, v) \in E .$$

(3) Every graph can be viewed as a $\mathbb{1}$-labelled graph using the unique mapping $f : E \to \mathbb{1}$ as a label function ($\mathbb{1}$ is a terminal object).

Many graph problems can be expressed and solved through the use of relational modelling. For example, this technique has been applied to a large variety of problems in the textbook of Schmidt and Ströhlein [SS93]. The resulting expressions are usually compact and even human-readable. Consider for example an undirected graph $G_{\text{classic}} = (V, E_{\text{classic}})$ in its classic representation, i.e. $E_{\text{classic}} \subseteq \{ \{ v, w \} \mid v, w \in V \}$. An edge set $M \subseteq E_{\text{classic}}$ is called a *matching* if and only if M is loop-free and the following condition holds:

$$\forall x, y, z \in V : \{ x, y \} \in M \wedge \{ x, z \} \in M \Rightarrow y = z .$$

At first glance, the latter condition is somewhat similar to the functionality of a relation. When expressing the above relationally, we first rewrite

$$E_{\text{relational}} := \{ (x, y) \in V \times V \mid \exists\, e \in E_{\text{classic}} : x, y \in e \} .$$

Clearly, $E_{\text{relational}}$ is a symmetric relation, because sets are invariant to the order in which their elements are listed. The same is true for the relational version of every subset of E_{classic}. With this consideration, we can rephrase the above condition in terms of the the following formula:

$$\forall x, y, z \in V : (x, y) \in N \wedge (x, z) \in N \Rightarrow y = z .$$

Now the above condition is actually just the functionality of N. Relational means allow to express the symmetry and functionality of N even more compactly, namely as $N^{\top} = N$ (symmetry) and $N \cdot N \subseteq \mathsf{I}$ (functionality), where $^{\top}$ denotes the converse

(transposition) of a relation, $\cdot$ is the relational composition and I is the identity relation. The fact that N is loop-free can be expressed in terms of the identity relation I, namely as $N \subseteq \bar{\mathsf{I}}$, which states that every pair contained in N has different components. To summarise, in relational terms an edge set $N \subseteq E_{\text{relational}}$ is called a matching, if and only if it is symmetric and functional, which is to say that

$$N = N^{\top} \quad \wedge \quad N \cdot N \subseteq \mathsf{I} \quad \wedge \quad N \subseteq \bar{\mathsf{I}}$$

holds. Not only is this last condition more compact than the original one, but it also allows application of a wide variety of classic (!) results. For example, it is well known that the intersection of two functional relations is again functional, that intersecting symmetric relations yields a symmetric relation, and that if two relations are contained in a third one, so is their intersection. Thus the intersection of two matchings is also a matching.

When dealing with non-trivially labelled graphs, the relational model becomes less convenient — one can view the label function as a relation itself or use more complex relations to model edge labels. However, there is another, more practical approach to arbitrary edge labels that makes use of a matrix representation of graphs as matrices over a certain algebraic structure, typically a semiring or a Kleene algebra. These matrices are special cases of the following definition.

2.3.2 Definition (Adjacency matrix)**.**
Let L be a set, $G = (V, E, f)$ be an L-labelled graph and Z an object such that $\mathsf{Z} \notin L$. Then we define the *adjacency matrix of* G as

$$A_G : V \times V \to L \cup \{ \mathsf{Z} \} \ , \quad (v, w) \mapsto \begin{cases} f(v, w) & : (v, w) \in E \\ \mathsf{Z} & : \text{otherwise} \ . \end{cases}$$

Up to the choice of Z the matrix A_G uniquely represents G. ⫽

The intuition behind this representation is that some fixed object Z represents the fact that there is no edge between two vertices and otherwise every edge is labelled with a given value. While we call A_G the adjacency matrix of a given graph, it is actually just a mapping from $V \times V$ to $L \cup \{ \mathsf{Z} \}$ and every matrix representation of A_G in the usual table form requires an enumeration of V.

In the above definition, edge values are arbitrary values of some set L. Practical applications usually rely on additional semantics of the edge values, like maximal capacity (in routing) or length (in the computation of paths with a given length). One particularly versatile structure for these semantics is a semiring.

2.3.3 Definition (Semiring)**.**
Let S be a non-empty set. An algebraic structure $(S, +, \cdot, 0, 1)$ of the type $(2, 2, 0, 0)$ is called *semiring* if and only if all of the following hold:

(SR1) $(S, +, 0)$ is a commutative monoid.

(SR2) $(S, \cdot, 1)$ is a monoid.

(SR3) $\cdot$ distributes over $+$ from both sides.

(SR4) 0 is annihilating with respect to $\cdot$.

The semiring is called *idempotent*, if and only if the addition is idempotent, i.e. we have that the following holds:

$$\forall s \in S : s + s = s \ .$$

For simplicity of notation we assume that $\cdot$ binds more strongly than $+$. ⫽

Before we continue, let us have a look at some common examples of semirings.

2.3.4 Example (Semiring).

(1) Every ring is a semiring, but only the trivial ring is an idempotent semiring. We discuss the second statement shortly.

(2) For every set X the structures $\left(2^X, \cup, \cap, \varnothing, X\right)$ and $\left(2^X, \cap, \cup, X, \varnothing\right)$ are idempotent semirings.

(3) More generally, every lattice $(L, \sqcup, \sqcap)$ with a least element $\bot$ and a greatest element $\top$ constitutes the semirings $(L, \sqcup, \sqcap, \bot, \top)$ as well as $(L, \sqcap, \sqcup, \top, \bot)$. In particular, $(\mathbb{B}, \vee, \wedge, \mathsf{F}, \mathsf{T})$ is a semiring.

(4) The structure $(\mathbb{R}_{\geqslant 0} \cup \{ \infty \}, \min, +, \infty, 0)$ is a semiring. It is known by the names *tropical semiring* and the *min-plus semiring*, cf. the textbook of Heidergott, Olsder, and Woude [HOW06]. The choice of $\mathbb{R}_{\geqslant 0}$ and ∞ is somewhat arbitrary and can be generalised. We discuss this fact when we deal with the implementation. ⫽

Semirings capture the essence of computations that consist of choices (addition) and sequential compositions (multiplication). It is important to note that idempotent choices never have non-trivial invertible elements. In fact, suppose $(S, +, \cdot, 0, 1)$ is an idempotent semiring and we have $a, b \in S$ such that $a + b = 0$. Then we find that

$$a = a + 0 = a + (a + b) = (a + a) + b = a + b = 0 \ ,$$

which in turn yields $0 = a + b = 0 + b = b$. This fact can be restated as follows: in idempotent semirings the non-zero elements are closed under addition. Above, we noted that only the trivial ring is an idempotent semiring. This is simple to see with the previous remark, because the additive structure of a ring is a group, but since the only element of an idempotent semiring that has an additive inverse is 0, we find that the additive group of the ring is the trivial group and thus the ring is trivial, too.

Another important note about idempotent semirings is that they allow a natural definition of an order on their elements.

2.3.5 Definition (The order of an idempotent semiring)**.**
Let $(S, +, \cdot, 0, 1)$ be an idempotent semiring. Then we define

$$\leqslant_S := \{ (s, t) \in S \times S \mid s + t = t \} \ .$$

The relation $\leqslant_S$ is called *(idempotent) semiring order*. When there is no risk of confusion, we will omit the index S.

The anticipatory name is justified by the following simple lemma, which is well-known [Koz94] and included only for the sake of completeness.

2.3.6 Lemma (The idempotent semiring order is an order)**.**
Let $(S, +, \cdot, 0, 1)$ be an idempotent semiring. Then $\leqslant_S$ is an order and 0 is the least element with respect to $\leqslant_S$. Also, $+$ and $\cdot$ are monotonic in both components with respect to $\leqslant_S$.

Proof. Reflexivity. Let $s \in S$. Then we have $s + s = s$, since S is idempotent and thus by Definition 2.3.5 we have $s \leqslant_S s$.
Transitivity. Let $s, t, u \in S$ such that $s \leqslant_S t$ and $t \leqslant_S u$. Then we get

$$\begin{array}{cl} & s + u \\ = & \wr\ t \leqslant_S u \text{ yields } t + u = u\ \wr \\ & s + (t + u) \\ = & \wr\ \text{addition is associative}\ \wr \\ & (s + t) + u \\ = & \wr\ s \leqslant_S t \text{ and thus } s + t = t\ \wr \\ & t + u \\ = & \wr\ \text{again } t + u = u, \text{ since } t \leqslant_S u\ \wr \\ & u \ . \end{array}$$

Thus we have $s \leqslant_S u$.
Antisymmetry. Let $s, t \in S$ such that $s \leqslant_S t$ and $t \leqslant_S s$. Then we find

$$\begin{array}{cl} & t \\ = & \wr\ s \leqslant_S t \text{ thus } s + t = t \text{ by Definition 2.3.5}\ \wr \\ & t + s \\ = & \wr\ \text{addition is commutative by Definition 2.3.3.}(SR1)\ \wr \\ & s + t \\ = & \wr\ t \leqslant_S s \text{ thus } t + s = s \text{ by Definition 2.3.5}\ \wr \\ & s \ . \end{array}$$

Least element. Let $s \in S$. Then $0 + s = s$ and thus $0 \leqslant_S s$.
Monotonicity of addition. Let $s, t, u \in S$ such that $s \leqslant_S t$. Then we have

$$\begin{array}{cl} & (s + u) + (t + u) \\ = & \wr\ \text{associativity and commutativity of } +\ \wr \end{array}$$

$$\begin{aligned}
& s+t+u+u \\
= \quad & \wr \text{ addition is idempotent } \wr \\
& s+t+u \\
= \quad & \wr \text{ since } s \leqslant_S t \text{, we have } s+t=t \wr \\
& t+u\ .
\end{aligned}$$

By definition of $\leqslant_S$ we get that $s+u \leqslant_S t+u$ holds. Since addition is commutative, we get the monotonicity in the second component directly from the just shown monotonicity in the first component.
Monotonicity of multiplication. Let $s, t, u \in S$ such that $s \leqslant_S t$. Then we get

$$\begin{aligned}
& s \cdot u + t \cdot u \\
= \quad & \wr \text{ multiplication is distributive by Definition 2.3.3.}\textit{(SR3)} \wr \\
& (s+t) \cdot u \\
= \quad & \wr \text{ since } s \leqslant_S t \text{ we have } s+t=t \wr \\
& t \cdot u\ .
\end{aligned}$$

Thus we get $s \cdot u \leqslant_S t \cdot u$. The second distributive law yields the monotonicity of the multiplication in the second component in an analogous fashion. ⫽

There are many more useful semirings and we will use some of these in the remaining text. More importantly, semirings allow typical constructions like products and coproducts. These constructions can be used to combine several different computations into one. We apply this technique in the search for a special type of path when we deal with the flow problem in Chapter 7. One particularly interesting construction is that of square matrices. As is the case with many other algebraic structures, square matrices over a semiring are again a semiring.

2.3.7 Theorem (Square matrices over a semiring form a semiring)**.**
Let $(S, +, \cdot, 0, 1)$ be a semiring and $n \in \mathbb{N}$. Then the structure $(S^{n \times n}, \boxplus, \boxtimes, \mathbb{0}_n, \mathbb{1}_n)$ is a semiring, where $\boxplus$ is the pointwise addition of matrices, $\boxtimes$ is matrix multiplication, $\mathbb{0}_n$ is the zero matrix and $\mathbb{1}_n$ is the identity matrix. ⫽

We omit the proof for this theorem, since it is a simple, but lengthy case of dotting the i's and crossing the t's. For the simplicity of representation we establish the following convention.

2.3.8 Convention (Matrix semiring notation)**.**
Let $(S, +, \cdot, 0, 1)$ be a semiring and $n \in \mathbb{N}$. Then we use the same symbols for the operations and constants in $S^{n \times n}$ as for S itself. Occasionally, we add an index n to these objects for clarity. For instance, 1_n is the multiplicative unit in $S^{n \times n}$. ⫽

When dealing with graphs whose edges are labelled with values from a semiring, we usually assume that 0 means "no edge". This is a common design decision,

because in semiring terms edges with a value of 0 carry no information: choosing between a zero-labelled edge and any other edge yields the latter and composing a zero-labelled edge with any other edge yields another zero-labelled edge. Incidentally, in most cases a 0 value's interpretation is also consistent with "no edge", for instance, if numbers represent the capacity of an edge, then an edge with an edge label 0 simply does not have any capacity. While this might just mean that its capacity has been exceeded, algorithmically this fact behaves the same as there being no edge.

Thus the value 0 has a special meaning in the case of a semiring and we use it as a Z-value in terms of labelled graphs.

2.3.9 Convention (Labelling with semiring values)**.**
Let $(S, +, \cdot, 0, 1)$ be a semiring and $G = (V, E)$ a graph. Suppose that $f : E \to S \setminus \{ 0 \}$. For simplicity, we call the tuple (V, E, f) an S-labelled graph and when representing the graph as an adjacency matrix we always choose $\mathsf{Z} := 0$. If no explicit function f is given, we choose the function $f_1 : E \to S \setminus \{ 0 \}, \; e \mapsto 1$. //

In summary, we have seen how graphs can be viewed in algebraic terms: either as a relation or more generally as an adjacency matrix. Just as with the relational counterpart, adjacency matrices over algebraic structures allow the application of the calculus of (linear) algebra and many graph problems can be rephrased in terms of bases, eigenvectors or matrix multiplications. We will introduce these representations when necessary. More importantly, graphs can be seen as elements of semirings themselves (namely the semiring of square matrices), thus allowing an approach to graph problems which is completely free of vertices or edges, but depends only on algebraic interactions between certain semiring elements.

However, we will occasionally require the notion of paths, which we model as sequences of vertices.

2.3.10 Definition (Walk, path and cycle)**.**
Let $G = (V, E)$ be a graph. Let $p \in V^*$ and $v, w \in V$. Then p is called a

$$\begin{aligned}
\textit{walk in } G \;&:\Longleftrightarrow\; \forall\, i \in \mathbb{N}_{<|p|-1} : (p_i, p_{i+1}) \in E \,, \\
\textit{walk in } G \textit{ from } v \textit{ to } w \;&:\Longleftrightarrow\; p \text{ is a walk} \wedge p \neq () \wedge p_0 = v \wedge p_{|p|-1} = w \,, \\
\textit{path in } G \textit{ (from } v \textit{ to } w\textit{)} \;&:\Longleftrightarrow\; p \text{ is a walk in } G \text{ (from } v \text{ to } w) \wedge p \text{ is injective} \,, \\
\textit{cycle in } G \;&:\Longleftrightarrow\; p \neq () \wedge p \text{ is a walk in } G \wedge p_0 = p_{|p|-1} \,.
\end{aligned}$$

A cycle $c \in V^*$ is called *odd*, if and only if $|c|$ is even and it is called *even* if and only if $|c|$ is odd. //

By this definition, walks are sequences of vertices, in which immediate successors in the sequence are successors in the graph. Note that walks can be empty and thus paths can be empty as well. This marginal case is allowed for consistency, because it is sometimes convenient to begin a path construction with the empty path. A path is a walk in which no vertex occurs twice and a cycle is a non-empty walk that ends

where it starts. While it may seem counterintuitive that cycles are called odd if and only if they have even length, the motivation is merely that the parity of the cycle is based upon the number of edges in the cycle, rather than the number of vertices. Clearly, the number of edges in a walk is one less than its length and thus an odd cycle has an odd number of edges.

2.4 Relations and Matrices

In this section we recall some definitions as well as well-known results concerning relations and matrices. The general idea is to express as many graph properties as possible in terms of algebraic means. For unlabelled graphs, it is known that relation algebra is a convenient tool, while for labelled graphs one can often use the algebra $S^{n \times n}$ for some fitting structure S.

We have already hinted at the fact that relations provide an elegant means to deal with many types of graph properties and problems. Interestingly, most of these properties can be expressed without ever dealing with the question whether a given pair is contained in a relation or not. This style allows a concise algebraic reasoning and the restriction to a small set of algebraic rules requires a rigorous approach to proofs and derivations. These algebraic rules for relations can be summarised in terms of a categorical approach. We use a similar definition as Furusawa and Kahl [FK98], but restrict ourselves to sets for simplicity.

2.4.1 Definition (Relation algebra)**.**
An abstract relation algebra is a small category $(\mathfrak{T}, \mathcal{R}(,), \cdot, \mathsf{I})$ such that all of the following axioms hold, where we write $r : X \leftrightarrow Y$ instead of $r \in \mathcal{R}(X, Y)$.

(RA1) For all $A, B \in \mathfrak{T}$ there are operations $\cup, \cap : \mathcal{R}(A, B) \times \mathcal{R}(A, B) \to \mathcal{R}(A, B)$, and constants $\mathsf{O}, \mathsf{L} \in \mathcal{R}(A, B)$ such that $\mathsf{O} \neq \mathsf{L}$ and

$$(\mathcal{R}(A, B), \cup, \cap, \mathsf{O}, \mathsf{L})$$

is a complete, atomic Boolean algebra. Recall that this definition also provides a complement operation $\bar{\ }$ as we have discussed in Section 2.2. Elements of $\mathcal{R}(A, B)$ are called (abstract) relations and $A \leftrightarrow B$ is called their *type*. When there is a risk of ambiguity, we add indices to the constants, e.g. $\mathsf{L}_{A,B}$.

(RA2) There is an operation $^\top$ such that for all $A, B \in \mathfrak{T}$ and all $r : A \leftrightarrow B$ we have $r^\top : B \leftrightarrow A$.

(RA3) For all $A, B, C \in \mathfrak{T}$ and all $q : A \leftrightarrow B$, $r : B \leftrightarrow C$ and $s : A \leftrightarrow C$ we have the so-called Schröder equivalences:

$$q \cdot r \subseteq s \iff q^\top \cdot \bar{s} \subseteq \bar{r} \iff \bar{s} \cdot r^\top \subseteq \bar{q} \ ,$$

where $\bar{\ }$ is the complement function mentioned above in *(RA1)*.

(RA4) For all $A, B, C, D \in \mathfrak{T}$ and all $r : A \leftrightarrow B$ we have the Tarski rule

$$r \neq \mathsf{O} \iff \mathsf{L}_{C,A} \cdot r \cdot \mathsf{L}_{B,D} = \mathsf{L}_{C,D} \, .$$

The most important example of an abstract relation algebra is the relation algebra of concrete relations. Let $\mathfrak{T}$ be a set of non-empty sets, $\mathcal{R}(A, B) := \mathcal{P}(A \times B)$, $\cdot$ be the relational composition, and

$$\mathsf{I}_A := \{ (x, y) \in A \times A \mid x = y \}$$

for all $A, B \in \mathfrak{T}$. Then $(\mathfrak{T}, \mathcal{R}(,), \cdot, \mathsf{I})$ is a relation algebra, where the abstract operations on $\mathcal{R}(A, B)$ are the set operations with the same symbolic names (e.g. $\cup$ denotes the union of sets). When possible, we use abstract rules for relational proofs only. This way, all results remain valid in case of other relation algebras, too, while we use only concrete relations for our applications.

From the definition of a relation algebra one can derive two particularly important properties of the greatest element. Suppose that $(\mathfrak{T}, \mathcal{R}(,), \cdot, \mathsf{I})$ is a relation algebra and let $X, Y, Z \in \mathfrak{T}$. Then we have the following equalities:

$$(\mathsf{L}_{X,Y})^\top = \mathsf{L}_{Y,X} \, , \qquad (2.4.1.i)$$

$$\mathsf{L}_{X,Y} \cdot \mathsf{L}_{Y,Z} = \mathsf{L}_{X,Z} \, . \qquad (2.4.1.ii)$$

The first property relies on the fact that transposition is an involution and the second one is a consequence of the Tarski rule [Ber08]. In the remaining text we will assume a basic familiarity with relational reasoning. When providing arguments for certain transformations or inequalities, we will typically provide a name for the employed property, but not cite any reference directly. For more detail on these properties we refer to the textbooks of Berghammer [Ber08] and Schmidt and Ströhlein [SS93].

Relation algebra allows an elegant description of relational properties like totality or transitivity with algebraic means only. The basic idea for such descriptions is to take the property for concrete relations as a formula in first-order or second-order logic and to derive a point-free representation from this formula. Then one can use this point-free representation as a definition in the abstract case, well-knowing that it does indeed describe the usual property in the concrete case. We summarise some of these properties.

2.4.2 Definition (Relational properties)**.**
Let $(\mathfrak{T}, \mathcal{R}(,), \cdot, \mathsf{I})$ be a relation algebra, $X, Y \in \mathfrak{T}$ and $r \in \mathcal{R}(X, Y)$. Then r is called

$$\begin{aligned}
\textit{total} &:\iff \forall\, Z \in \mathfrak{T} : r \cdot \mathsf{L}_{Y,Z} = \mathsf{L}_{X,Z} \, , \\
\textit{surjective} &:\iff r^\top \text{ total} \, , \\
&\iff \forall\, Z \in \mathfrak{T} : r^\top \cdot \mathsf{L}_{X,Z} = \mathsf{L}_{Y,Z} \\
&\iff \forall\, Z \in \mathfrak{T} : \mathsf{L}_{Z,X} \cdot r = \mathsf{L}_{Z,Y} \, , \\
\textit{univalent} &:\iff r^\top \cdot r \subseteq \mathsf{I} \, , \\
\textit{injective} &:\iff r^\top \text{ injective}
\end{aligned}$$

$$\iff r \cdot r^\top \subseteq \mathsf{I} \; . \qquad /\!\!/$$

To check that a relation $r \in \mathcal{R}(X, Y)$ is total it is sufficient to check that there exists a $Z \in \mathfrak{T}$ such that $r \cdot \mathsf{L}_{Y,Z} = \mathsf{L}_{X,Z}$ holds. Indeed, suppose that this is the case and let $Z' \in \mathfrak{T}$. Then we get

$$r \cdot \mathsf{L}_{Y,Z'} = r \cdot \mathsf{L}_{Y,Z} \cdot \mathsf{L}_{Z,Z'} = \mathsf{L}_{X,Z} \cdot \mathsf{L}_{Z,Z'} = \mathsf{L}_{X,Z'} \; ,$$

where the first and last transformations are due to Equation (2.4.1.ii).

There are two special types of relations which are particularly useful when it comes to modelling sets algebraically, called vectors and points.

2.4.3 Definition (Vector, point)**.**
Let $(\mathfrak{T}, \mathcal{R}(,), \cdot, \mathsf{I})$ be a relation algebra, $X, Y \in \mathfrak{T}$ and $r \in \mathcal{R}(X, Y)$. Then we call r

$$\begin{aligned} \textit{vector} &:\iff \mathsf{L}_{X,X} \cdot r = r \; , \\ \textit{point} &:\iff r \text{ is a univalent, total vector} \; . \end{aligned} \qquad /\!\!/$$

This is one of the two common definitions in the literature (used by Kawahara [Kaw06], for example). In this approach vectors are considered to be row vectors. Similarly, one can consider vectors to be column vectors [Ber08; SS93]. These different definitions are compatible: if in r is a row vector, then $r^\top$ is a column vector and the same is true for points. If $r : X \leftrightarrow Y$ is a concrete relation and a row vector, then every column of r contains the same value.

To model sets algebraically, one needs to transform the set into a relation. One of the well-known ways to do that is to consider for a set S the relation

$$S_{\mathbb{1}} := \mathbb{1} \times S \; ,$$

which is essentially the same as S, save for the fact that every $s \in S$ is replaced with $(\square, s)$. Similarly, given a vector $r : D \leftrightarrow S$, we can consider this relation to be a subset of S, because $\mathrm{pr}_2(r) \subseteq S$ holds. We then obtain the following properties:

$$\begin{aligned} \mathrm{pr}_2(S_{\mathbb{1}}) &= S \; , \\ \mathsf{L}_{D,\mathbb{1}} \cdot \mathrm{pr}_2(r)_{\mathbb{1}} &= r \; , \end{aligned}$$

where the first one is trivially true, while the second one depends on the fact that r is a vector. Thus it is possible to reconstruct a set from the vector that represents the set and also to reconstruct a vector from the set it represents. Due to these properties we will not distinguish between sets S and their vector representation $S_{\mathbb{1}}$ in the remainder of the text. Whenever we consider a set as a vector, we usually write it in a lower case letter. Thus if $s \subseteq X$ denotes a set and $r : X \leftrightarrow Y$ is a relation, we write

$$s \cdot r$$

instead of

$$\mathrm{pr}_2(s_{\mathbb{1}} \cdot r)$$

when we consider $s \cdot r$ as a set. This identification is for convenience only, because it is simple to inline this definition in every particular case.

On several occasions we use matrices over a given structure. While arguments which deal with matrix indices are typically simple, they are also sowewhat tedious, because of the interaction of many indices. For the sake of simplicity we make use of standard unit vectors and standard unit matrices, which allow a more algebraic approach to matrix indices.

2.4.4 Definition (Standard unit vectors and matrices)**.**
Let $(S, +, \cdot, 0, 1)$ be a semiring and $n \in \mathbb{N}$. For every $i \in \mathbb{N}_{<n}$ we define:

$$\mathsf{e}^n(i) : \mathbb{N}_{<n} \to S \ , \quad j \mapsto \delta_{i,j} \ ,$$

where δ is the Kronecker delta and call the vector $\mathsf{e}^n(i) \in S^n$ the *i-th standard unit vector (of dimension n)*. Similarly, for all $m \in \mathbb{N}$ and all $j \in \mathbb{N}_{<m}$ we define the (i, j)*-th standard unit matrix of the size* (n, m) as

$$\mathsf{e}^{n,m}(i, j) := (\mathsf{e}^n(i))^\top \cdot \mathsf{e}^m(j) \ ,$$

where we consider $\mathsf{e}^k(i)$ to be a $1 \times k$-matrix for each $k \in \{ n, m \}$ and $\cdot$ is matrix multiplication. Whenever the dimensions are reconstructible from the context, we omit the indices for both standard unit vectors and matrices. ⫽

Unit vectors are a well-known algebraic tool. Conveniently, they can be used to describe many matrix properties in a more algebraic fashion than by simply accessing all indices of a matrix. Some of these descriptions are summarised in the following proposition.

2.4.5 Proposition (Standard unit vectors)**.**
Let $(S, +, \cdot, 0, 1)$ *be a semiring and* $n \in \mathbb{N}$.

(1) For all $i \in \mathbb{N}_{<n}$*, all* $m \in \mathbb{N}$ *and all* $a \in S^{n \times m}$ *we have*

$$a_i = \mathsf{e}^n(i) \cdot a \ .$$

(2) For all $i \in \mathbb{N}_{<n}$*, all* $m \in \mathbb{N}$*, all* $j \in \mathbb{N}_{<m}$ *and all* $a \in S^{n \times m}$ *we have*

$$(a_{i,j}) = \mathsf{e}^n(i) \cdot a \cdot \mathsf{e}^m(j)^\top \ .$$

In particular, for all $i, j \in \mathbb{N}_{<n}$ *we have*

$$\mathsf{e}^n(i) \cdot \mathsf{e}^n(j)^\top = \mathsf{e}^n(i) \cdot 1_n \cdot \mathsf{e}^n(j)^\top = ((1_n)_{i,j}) = (\delta_{i,j}) \ ,$$

where δ *is the Kronecker delta.* ⫽

In the remainder of this book we will make ample use of the scalar multiplication of matrices with a semiring element. Since we consider vectors to be a special type of matrices, this operation is applicable to vectors as well. For every monoid $(S, \cdot, 1)$ and all $m, n \in \mathbb{N}$ we define the scalar multiplication as

$$\bullet : S \times S^{m \times n} \to S^{m \times n} \ , \quad (s, a) \mapsto ((i, j) \mapsto s \cdot a_{i,j}) \ .$$

Clearly, this definition depends on the monoid operation, but is actually independent of the actual matrix size. Since the matrix size is not reflected in the actual definition, we use the same symbol to denote scalar multiplications of different matrix sizes. The following properties of the scalar multiplication are of utmost importance.

2.4.6 Proposition (Properties of scalar multiplication)**.**
Let $(S, +, \cdot, 0, 1)$ be a semiring, and $l, m, n \in \mathbb{N}$.

(1) For all $s \in S$, all $a \in S^{l \times m}$ and all $b \in S^{m \times n}$ we have

$$s \bullet (a \cdot b) = (s \bullet a) \cdot b.$$

In particular, for all $s \in S$ and all $a \in S^{l \times m}$ we have

$$s \bullet a = s \bullet (1 \cdot a) = (s \bullet 1) \cdot a \ .$$

Additionally, for all $s, t \in S$ and all $a \in S^{l \times m}$ we have

$$s \bullet (t \bullet a) = (s \cdot t) \bullet a \ .$$

(2) For all $s \in S$ and all $a \in S^{l \times m}$ we have

$$\big(\forall\, i \in \mathbb{N}_{<l} : \forall\, j \in \mathbb{N}_{<m} : s \cdot a_{i,j} = a_{i,j} \cdot s\big) \Longrightarrow s \bullet a = a \cdot (s \bullet 1) \ .$$

In particular, we have for all $s \in S$, all $a \in S^{l \times m}$ and all $b \in S^{m \times n}$ that

$$\big(\forall\, i \in \mathbb{N}_{<l} : \forall\, j \in \mathbb{N}_{<m} : s \cdot a_{i,j} = a_{i,j} \cdot s\big) \Longrightarrow s \bullet (a \cdot b) = a \cdot (s \bullet b) \ .$$

For unit vectors we get that for all $s \in S$ and all $i \in \mathbb{N}_{<l}$

$$(s) \cdot \mathsf{e}(i) = s \bullet \mathsf{e}(i) \quad \text{and} \quad \mathsf{e}(i)^\top \cdot (s) = s \bullet \mathsf{e}(i)^\top \ .$$

(3) For all $s \in S$ and all $v \in S^l$ we have

$$(s) \cdot v = (s \bullet (1)) \cdot v = s \bullet ((1) \cdot v) = s \bullet v \ .$$

In particular, for all $s, t \in S$ and all $a \in S^{l \times m}$ we have

$$(s \cdot t) \bullet a = s \bullet (t \bullet a) = (s) \cdot (t \bullet a) \ .$$

//

The above proposition yields the associativity of the scalar multiplication in a certain sense. This associativity is the reason why we usually omit brackets and write $s \bullet a \cdot b$, because both $(s \bullet a) \cdot b$ and $s \bullet (a \cdot b)$ denote the same value. We omit the proof, because the statements are well-known.

One particularly important type of matrices (or relations, more generally) are so-called partial identities. In the concrete case of (Boolean) matrices these are those matrices that have non-zero values at most at their main diagonal and each of these values is less or equal than 1, where 1 is the multiplicative unit of an underlying semiring. As it turns out, this descriptive, but component-based definition can be replaced with a more general definition, which at the same time is simpler and more convenient.

2.4.7 Definition (Partial identity)**.**
Let $(S, +, \cdot, 0, 1)$ be an idempotent semiring. We then define for every $T \subseteq S$

$$\mathsf{PI}_S(T) := \{ t \in T \mid t \leqslant_S 1 \} \ .$$

The elements of $\mathsf{PI}_S(T)$ are called *partial identities*. We omit the subscript S when there is only one semiring present. ⫽

In the special case that $\mathsf{PI}(S) = \{ 0, 1 \}$, the matrices in $\mathsf{PI}(S^{n \times n})$ consist of zeroes and ones only, where all ones are located on the main diagonal. Such matrices are well-suited for the representation of subsets of $\mathbb{N}_{<n}$, as we will see later in Section 3.3.

2.5 Haskell and Functional Programming

All code presented in this thesis (except from some exemplary functions in Section 9.2.2) is presented in Haskell [Mar09]. Haskell is a non-strict, functional language with strong static typing. We refrain from giving an introduction into the language itself, because many excellent introductions already exist [Chr12; OSG08; Lip11]. We assume a basic familiarity with Haskell, which includes data types, higher-order concepts and the concept of type classes as well as the standard type classes like *Ord*, *Monoid* and *Functor*.

The (standard) Haskell libraries that we use are available online at `http://hackage.haskell.org` and usually contain extensive documentation and, in many cases, examples. The libraries can be searched using the Haskell library search engine Hoogle [Mit13], which is available online at `https://www.haskell.org/hoogle`. It is particularly useful for finding single functions. For instance, in Section 4.3 we use the function *groupBy* without mentioning its type or what it does exactly. Entering `groupBy` into Hoogle finds various functions with this name, one of which is polymorphic and has the type

$$groupBy :: (\alpha \to \alpha \to Bool) \to [\alpha] \to [[\alpha]] \ .$$

A corresponding link leads to the Haskell module *Data.List*, where the documentation shows a brief description, the related function *group* and a link to the actual implementation.

Haskell packages are collections of Haskell modules, where the latter are simply files that contain Haskell code. The usual nomenclature for modules is *X.Y.Z* and so forth, where *X* denotes the most general description of the module functionality like *Data*, the *Y* is a more concise specification like *IntMap* and the *Z* value shows possible variations like *Lazy* or *Strict*. While this is a greatly simplified example, in most cases we reference only this kind of modules, saying something like "The Haskell module *Data.List* provides a function *groupBy*" or "The function *fromList* is defined in the module *Data.IntMap*".

As for the actual implementations of certain functions we usually use an implementation that is simplified using standard techniques, but still legible. For example,

we write

$$\begin{aligned}&\mathit{scaleList} :: \mathit{Integer} \to [\mathit{Integer}] \to [\mathit{Integer}]\\&\mathit{scaleList}\ x\ \mathit{xs} = \mathit{map}\ (x\cdot)\ \mathit{xs}\end{aligned}$$

rather than η-reducing the above expression to the shorter version

$$\mathit{scaleList} = \mathit{map} \circ (\cdot)$$

which is less intuitive to read. However, we use the latter definition occasionally, in particular in cases where the actual definition is presented only for the sake of completeness and not because the function definition is important.

We discuss two different fold strategies in Section 3.3 and Section 3.5 in some detail. For that reason we now briefly discuss the underlying functions and their implementations in Haskell. The first function is the so-called left-fold *foldl*, which can be defined in Haskell as follows.

$$\begin{aligned}&\mathit{foldl} :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta\\&\mathit{foldl}\ _\ e\ [\,] \qquad\quad = e\\&\mathit{foldl}\ f\ \ e\ (x : \mathit{xs}) = \mathit{foldl}\ f\ \ (f\ e\ x)\ \mathit{xs}\end{aligned}$$

The idea behind folding a list from the left is to traverse the list left to right and to use the previously computed value as the new accumulator. The "left" in the name refers to the fact that applications of f are associated to the left, for example $\mathit{foldl}\ f\ e\ [a, b, c] = f\ (f\ (f\ e\ a)\ b)\ c$. However, note that since Haskell is non-strict, the function f does not compute any value until it is actually demanded. This can lead to a large stack of unevaluated applications of f, which is costly to maintain. Another list folding strategy is given by the right-fold function *foldr*.

$$\begin{aligned}&\mathit{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta\\&\mathit{foldr}\ _\ e\ [\,] \qquad\quad = e\\&\mathit{foldr}\ f\ \ e\ (x : \mathit{xs}) = f\ x\ (\mathit{foldr}\ f\ e\ \mathit{xs})\end{aligned}$$

This function also traverses a list from left to right, but instead of using previously computed values, it uses values that will be computed later. The "right" in the name refers to the fact that now f associates to the right and we have $\mathit{foldr}\ f\ e\ [a, b, c] = f\ a\ (f\ b\ (f\ c\ e))$. It is known that left-folds can be expressed in terms of right-folds, but not vice versa [Hut99]. In fact, right-folds are computationally more powerful, because they *can* yield values for infinite lists, while left-folds never terminate on such lists. One particular example is the well-known function $\mathit{and} :: [\mathit{Bool}] \to \mathit{Bool}$, which computes the conjunction of a list of Boolean values. When defined as a left-fold $\mathit{and} = \mathit{foldl}\ (\wedge)\ \mathit{True}$, this function will not terminate on infinite lists. Using a right-fold on the other hand $\mathit{and} = \mathit{foldr}\ (\wedge)\ \mathit{True}$ on a list that contains at least one occurrence of *False* will also yield *False*, because $(\wedge)$ does not evaluate its second argument, if the first one is *False*.

The presented code is intended to be executable. We have implemented a polished version of this code in our *gwaf* library [Dan15]. Many of the functions presented in

this thesis are present in the library as more parametric versions or even type class functions. However, the general ideas behind the implementations are precisely those we are about to discuss.

Whenever it comes to reasoning about Haskell code we take an informal approach and use equational reasoning in terms of the code itself. The reason for this informality is the fact that the semantics of a functional language for pure functions is essentially the syntax of the functions up to the consideration of possible errors. This kind of reasoning is very common in functional programming and is often considered a benefit of this paradigm, because the semantics depend on the (function) definitions only, rather than on certain memory transformations. Also, we sometimes mention the complexity of a function, but rarely use the Landau notation for concise statements. This is due to the fact that complexity in lazy functional languages is not compositional [San95]. For example, if $ack :: Integer \rightarrow Integer \rightarrow Integer$ denotes the Ackermann function, the computation of the list

$$map\ (\lambda i \rightarrow ack\ i\ i)\ [1\,..\,n]$$

requires a lot of effort even for very small n like $n = 10$, while the computation of its length

$$length\ (map\ (\lambda i \rightarrow ack\ i\ i)\ [1\,..\,n])$$

is linear in the list length, which is n and thus can be computed with little effort even for large n. Whenever we mention the complexity of a function, we take this difficulty into account and consider only the actually computed information under the assumption that the function result is fully evaluated.

On some occasions we present measurements of execution times and memory consumption. All of these measurements were taken on a Dell Latitude E6320 notebook with an Intel Core i5-2520M CPU (2×2.5 GHz, with two logical cores per physical core) with 8 GB of DDR3 RAM, running Ubuntu 12.04 and using GHC 7.6.3.

The basic concepts presented in this dissertation are not restricted to Haskell. For the most part, we make ample use of higher-order functions, which are ubiquitous in functional programming in general. On some occasions, we present code that is explicitly based upon non-strictness. This is crucial when computing a possibly infinite intermediate structure, like the one we present in Section 4.5. However, it is well-known that non-strictness can be simulated in strict languages [Pau96; WTM98]. Thus the particular choice of the non-strict functional language Haskell is not a restriction and the resulting code can be translated to other (possibly strict) languages like ML.

2.6 Graph Implementations

Using some kind of graph representation to solve certain specific problems is a common practice in computer science. In fact, one usually learns a good deal about

different graph implementations in the course of the first (bachelor's) year. However, these implementations are mostly suited for a single problem or a small class of problems and may result in poor efficiency in terms of complexity and implementation when used for another problem class. In general, it is difficult to choose a graph representation which

(a) has a good asymptotic complexity for a large number of problems,

(b) allows simple solutions for said problems and

(c) lives up to the theoretical complexity in terms of the implicit constants in the big-O notation being of moderate size.

Typically, this problem results in a gap between the actual problem solution and its implementation, which is due to certain preconditions not being met initially, many tedious corner cases or simply the fact that the chosen data structures are missing the required interface functions.

One elegant attempt to solve this kind of problems is treating graphs as an *abstract data type* (ADT), whose concrete definition can be supplied depending on the problem at hand [RL99]. Assuming that the interface functions of the ADT are sufficiently well-designed, one can then provide a solution in terms of the ADT functions and choose different concrete implementations to obtain both, an elegant implementation as well as a good complexity. In Haskell such an approach can be implemented in terms of type classes, which specify interface function that needs to be implemented for any particular instance.

We take an intermediate approach for the sake of simplicity and presentation. When it comes to the implementation of graph functions, we usually provide a complete definition. In doing so, we use as many general purpose functions from our data types as possible and use the actual definitions as little as possible. This approach allows us to exchange the data types by different ones, which provide the same functions, but with possibly different implementations and not change anything in the actual implementation. We briefly discuss an abstraction with type classes in Section 4.6 and hint at possible variations in locally used data types in Section 4.2.1.

Chapter 3

Kleene Closures – A Case Study

In this chapter we consider a single algorithm, namely the computation of the Kleene closure of a matrix. Said algorithm is a well-known, classical result and we use it as a basis for abstracting the components, which are necessary for the implementation of further algorithms. Large portions of this chapter have been previously published at the LOPSTR 2014 conference [Dan14b].

3.1 Introduction

The Kleene closure is a well-established computational pattern with numerous applications, e.g. in regular expressions or (generalised) reachability. While the closure operation can be defined in any Kleene algebra, the Kleene algebra of square matrices over a Kleene algebra is of particular interest. Once a closure operation for square matrices is defined, it can be used to solve classical matrix problems like matrix multiplication or matrix inversion. Additionally, it can be applied to a wide variety of problems from different fields that have a natural representation in terms of matrices. Examples of such problems are reachability in graphs (Boolean matrices), the Dijkstra algorithm (matrices over the so-called tropical Kleene algebra (cf. Example 3.2.2.(3)) or the CYK parsing algorithm (matrices over rules). Finally, we show in Section 3.4 that every closure can be expressed in terms of a matrix closure (of a matrix with a fixed size), which is to say that using matrices is not a restriction, but simply a general view for many different problems, while allowing all of the usual algebraic means associated with matrices.

The vast amount of problems captured by the computational scheme of the Kleene closure has led to a lot of research in this area, which includes several implementations in different programming languages. These are usually provided as imperative (pseudo-)code, but at the time of this writing there has been little development of a functional program for the computation of the Kleene closure, which is not merely a (direct) translation of a given imperative algorithm into a functional language. One can argue that not every program that is written in a functional language is in fact what can be considered a functional program, because an arbitrary program may lack the usual compositional structure. This subtle distinction is particularly important when dealing with algorithms. Algorithms usually have a sequential look-and-feel[1]

[1] A formal definition of "algorithm" is an intricate matter, cf. Blass and Gurevich [BG03].

that allows following a set of instructions step by step. Such a construct fits well in the context of imperative languages. Functional programs on the other hand are typically inherently compositional and not necessarily computed in sequence. This conceptual difference indicates that an algorithmic solution of a problem may not be suited for a functional implementation.

In this chapter we generalise an approach taken to compute a specific instance of the Kleene closure by Berghammer [Ber11] to the general case and present a purely functional program that can be used to compute the said closure. Our functions are prototypical by design, but the modularity of their components can be easily used to improve upon the implementation. Our approach is structured as follows.

- ▷ We recall the definition and some basic properties of Kleene algebra in Section 3.2. and provide a definition of the Kleene closure that employs an auxiliary function.
- ▷ In Section 3.3 we provide an auxiliary function that can be used to compute the Kleene closure. Using algebraic reasoning we derive a recursive variant of this function.
- ▷ The function from Section 3.3 is studied in the special case of square matrices over a Kleene algebra in Section 3.4.
- ▷ We implement the obtained recursion in Haskell in Section 3.5, where we additionally employ Kleene algebra laws to improve performance.
- ▷ We discuss three alternative implementations in some detail in Section 3.6 and present comparative measurements of all functions in Section 3.7.

At the time this book is written, such a derivation is novel. The code in this chapter varies significantly from the one published at the LOPSTR 2014 conference [Dan14b]. In particular, we exchanged the data types towards those we used at the TFP 2014 workshop [Dan14c] to simplify the presentation in this book. A version of the (less parametric) LOPSTR 2014 code is available at `https://github.com/nikitaDanilenko/functionalKleene` and as a component of our *gwaf* library [Dan15].

3.2 Algebraic Preliminaries

In the following we will deal with Kleene algebras according to the definition given by Kozen [Koz94]. All definitions and consequences in this section are mentioned by Kozen in the above source and the non-elementary results are proved as well; we include all of these for the sake of completeness only. We begin with the definition of a Kleene algebra, which can be split into two parts: the notion of an idempotent semiring and the concept of the star closure.

3.2.1 Definition (Kleene algebra)**.**
Let K be a non-empty set. An algebraic structure $(K, +, \cdot, ^\star, 0, 1)$ of the type $(2, 2, 1, 0, 0)$ is called a *Kleene algebra* if and only if all of the following conditions hold:

(KA1) $(K, +, \cdot, 0, 1)$ is an idempotent semiring.

(KA2) For all $a \in K$ we have $1 + a \cdot a^\star \leqslant a^\star$ and $1 + a^\star \cdot a \leqslant a^\star$.

(KA3) For all $a, b \in K$ the following implications hold:

(i) $b \cdot a \leqslant b \Rightarrow b \cdot a^\star \leqslant b$,
(ii) $a \cdot b \leqslant b \Rightarrow a^\star \cdot b \leqslant b$.

The order which is used in *(KA2)* and *(KA3)* is the idempotent semiring order from Definition 2.3.5. Additionally, we define

$$^+ : K \to K\,, \quad a \mapsto a \cdot a^\star$$

and call this function the *Kleene closure*. We define that * and $^+$ bind more strongly than multiplication (which binds more strongly than addition by Definition 2.3.3). ⫽

First and foremost, any Kleene algebra is an idempotent semiring. In addition to composition and choice we have another operation that can be thought of as *iteration*, namely *. By Definition 3.2.1.*(KA2)* we have that for all $a \in K$ the value $a^\star$ is a pre-fixpoint (i.e. contracted element) of the function $f_a : K \to K$, $k \mapsto 1 + a \cdot k$, because $f_a(a^\star) = 1 + a \cdot a^\star \leqslant a^\star$ holds.

3.2.2 Example (Kleene algebra).

(1) Let $(\mathfrak{T}, \mathcal{R}(,), \cdot, \mathsf{I})$ be a relation algebra and $T \in \mathfrak{T}$. We define * to be the reflexive-transitive closure of relations, which is to say that we have

$$\begin{aligned} ^\star : \mathcal{R}(T,T) &\to \mathcal{R}(T,T)\,, \\ r &\mapsto \bigcap \{\, s \in \mathcal{R}(T,T) \mid s \text{ reflexive} \wedge s \text{ transitive} \wedge r \subseteq s \,\}\ . \end{aligned}$$

Then $(\mathcal{R}(T,T), \cup, \cdot, ^\star, \mathsf{O}, \mathsf{I})$ is a Kleene algebra. In particular, this is true for concrete relations.

(2) $(\mathbb{B}, \vee, \wedge, ^\star, \mathsf{F}, \mathsf{T})$, where $^\star : \mathbb{B} \to \mathbb{B}$, $b \mapsto \mathsf{T}$, is a Kleene algebra.

(3) Let

$$^\star : \mathbb{R}_{\geqslant 0} \cup \{\, \infty \,\} \to \mathbb{R}_{\geqslant 0} \cup \{\, \infty \,\}\ , \quad b \mapsto 1\ .$$

Then $(\mathbb{R}_{\geqslant 0} \cup \{\, \infty \,\}, \min, +, ^\star, \infty, 0)$ is a Kleene algebra, called the *tropical Kleene algebra* (cf. Example 2.3.4.(4)). ⫽

Kleene algebras are a well-understood structure and numerous properties of these algebras are known[2]. In many cases, the abstract rules for Kleene algebras are known

[2] We refer to the textbook by Conway [Con71] for an introduction and to the work of Kozen [Koz90] for an overview of related structures.

from the special case of a single Kleene algebra and then generalised to the abstract case. For example, it is simple to see that for concrete relations $a, b \in \mathcal{P}(X \times X)$ one finds that

$$a \cdot (b \cdot a)^\star = (a \cdot b)^\star \cdot a$$

holds. The very same rule remains true in the case of abstract relations and then again in the general case of a Kleene algebra. In the following proposition we summarise those few properties, which we need in the remainder of this chapter. All of these properties are known and proofs have been provided by Kozen [Koz94].

3.2.3 Proposition (Properties of the star closure)**.**
Let $(K, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra. Then the following hold:

(1) $\forall\, a \in K : 1 + a \cdot a^\star = a^\star = 1 + a^\star \cdot a.$ *(fixpoint)*

(2) $\forall\, a, b \in K : 1 + a \cdot b = b \Rightarrow a^\star \leqslant b.$ *(least fixpoint)*

(3) $\forall\, a, b \in K : (a + b)^\star = a^\star \cdot (b \cdot a^\star)^\star.$ *(decomposition)*

(4) $\forall\, a, b, x \in K : a \cdot x = x \cdot b \Rightarrow a^\star \cdot x = x \cdot b^\star.$ *(weak commutativity)*

(5) $\forall\, a \in K : 1 + a^+ = a^\star = (1 + a)^+.$ ⫽

Now that we are equipped with these important tools, we can easily compute the star closures (and thus also the Kleene closures) of the constants in a Kleene algebra.

3.2.4 Corollary (Closures of constants)**.**
Let $(K, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra. Then we have:

(1) $0^\star = 1$ *and* $0^+ = 0.$

(2) $1^\star = 1$ *and* $1^+ = 1.$

Proof. Statement (1). We calculate as follows:

$$\begin{aligned} & 0^\star \\ = \ & \wr \text{ Proposition 3.2.3.(1) } \wr \\ & 1 + 0 \cdot 0^\star \\ = \ & \wr \text{ 0 is annihilating by Definition 2.3.3.}\textit{(SR4)} \wr \\ & 1 + 0 \\ = \ & \wr \text{ 0 is neutral with respect to addition by Definition 2.3.3.}\textit{(SR1)} \wr \\ & 1\ . \end{aligned}$$

By definition we also get

$$\begin{aligned} & 0^+ \\ = \ & \wr \text{ definition of the Kleene closure, Definition 3.2.1 } \wr \end{aligned}$$

$$
\begin{aligned}
& 0 \cdot 0^{\star} \\
= \; & \wr \; 0 \text{ is annihilating by Definition 2.3.3.}\textit{(SR4)} \; \wr \\
& 0 \; .
\end{aligned}
$$

Statement (2). First, we get

$$
\begin{aligned}
& 1 + 1 \cdot 1 \\
= \; & \wr \; 1 \text{ is neutral with respect to multiplication by Definition 2.3.3.}\textit{(SR2)} \; \wr \\
& 1 + 1 \\
= \; & \wr \; \text{addition is idempotent by Definition 3.2.1.}\textit{(KA1)} \; \wr \\
& 1 \; .
\end{aligned}
$$

Thus by Proposition 3.2.3.(2) we find that $1^{\star} \leqslant 1$. On the other hand we have

$$
\begin{aligned}
& 1 \\
= \; & \wr \; 0 \text{ is neutral with respect to addition} \; \wr \\
& 1 + 0 \\
\leqslant \; & \wr \; 0 \leqslant 1 \cdot 1^{\star} \text{ and addition is monotonic, both by Lemma 2.3.6} \; \wr \\
& 1 + 1 \cdot 1^{\star} \\
= \; & \wr \; \text{Proposition 3.2.3.(1)} \; \wr \\
& 1^{\star} \; .
\end{aligned}
$$

The Kleene closure is now simply $1^{+} = 1 \cdot 1^{\star} = 1 \cdot 1 = 1$ by the above and the neutrality of 1 with respect to multiplication. ⫽

Just as it was the case with the semiring structure, the set of square matrices of a fixed size over a Kleene algebra is again a Kleene algebra.

3.2.5 Theorem ($K^{n \times n}$ as a Kleene algebra)**.**
Let $(K, +, \cdot, ^{\star}, 0, 1)$ be a Kleene algebra. Then there is a function $^{*} : K^{n \times n} \to K^{n \times n}$ such that $(K^{n \times n}, +, \cdot, ^{*}, 0_n, 1_n,)$ is a Kleene algebra, where we use Convention 2.3.8 for the operations and constants in $K^{n \times n}$. ⫽

Note that we use * to refer to the star closure operation in K, while * is the special case of the closure operation in $K^{n \times n}$. The statement of the theorem can be split in two parts according to the definition of Kleene algebras (Definition 3.2.1). The first part is showing that $(K^{n \times n}, +, \cdot, 0_n, 1_n)$ is an idempotent semiring, which is true by Theorem 2.3.7. The second part is providing the function * and verifying that it satisfies the properties required by Definition 3.2.1.*(KA2)* and Definition 3.2.1.*(KA3)*. Throughout the literature one usually finds two particular definitions, which we mention here for the purpose of comparison. Let $a \in K^{n \times n}$.

The first definition states that it is possible to compute matrices $a^{(0)}, \ldots, a^{(n)}$ such that $a^{(0)} := a$, and for all $i, j, k \in \mathbb{N}_{<n}$ one has

$$
a^{(k+1)}_{i,j} := a^{(k)}_{i,j} + a^{(k)}_{i,k} \cdot \left(a^{(k)}_{k,k} \right)^{\star} \cdot a^{(k)}_{k,j} \; . \tag{3.2.5.i}
$$

With these definitions one obtains that $a^{(n)} = a^+$ holds. The star operation $\star$ of K in Equation (3.2.5.i) is applied to the Kleene algebra element $a^{(k)}_{k,k}$, which is put in brackets for legibility and does not denote a 1×1-matrix. This definition provides two ways of computing a^*: either as $a^* = 1 + a^+$ or as $a^* = (1 + a)^+$, by Proposition 3.2.3.(5).

The second approach is based upon an inductive definition, in which one recursively decomposes a given matrix into smaller components. For every $k \in K$ one defines the star closure of the 1×1-matrix (k) as

$$(k)^* := (k^\star) \ .$$

For any matrix $a \in K^{n \times n}$, where $n \geqslant 2$, one then chooses $l, m \in \mathbb{N}_{>0}$ such that $n = l + m$ and splits a into submatrices

$$a = \begin{pmatrix} p & q \\ r & s \end{pmatrix} \ ,$$

where $p \in K^{l \times l}$, $s \in K^{m \times m}$, and q, r have corresponding dimensions. Assuming that the Kleene closure of matrices of the sizes l and m is already known, one can then compute $x := (p + q \cdot s^* \cdot r)^*$ and set

$$a^* := \begin{pmatrix} x & x \cdot q \cdot s^* \\ s^* \cdot r \cdot x & s^* + s^* \cdot r \cdot x \cdot q \cdot s^* \end{pmatrix} \ . \qquad (3.2.5.ii)$$

Since all of the matrices used in this definition have strictly smaller dimensions than a, these computations can be used recursively for the computation of a^*, thus turning the above assumption into a recursive definition.

Both definitions are elegant in different ways: the first one is easy to translate in graph-theoretic terms and easy to implement in an imperative language, while the second one has a foundation in automata theory. Still, both definitions are rather algorithms (the second one can even be considered a non-deterministic one, because the actual decomposition does not matter), since they describe a sequence of steps that lead to $a^\star$. From a complexity point of view the second definition provides an additional challenge, since it contains matrix multiplication. It is not apparent at the first glance, what the exact complexity is, while the first one is clearly cubic in n.

As for the non-determinism in the second definition, one usually chooses a fixed decomposition structure. For instance, in complexity estimates one usually decomposes the dimension n around

$$\left\lceil \frac{n}{2} \right\rceil = \begin{cases} \frac{n}{2} & : n \text{ even} \\ \frac{n+1}{2} & : \text{otherwise} \ . \end{cases}$$

Such a decomposition allows an elegant computation of the complexity of the second definition. We refer to the textbook of Pettorossi [Pet13] for more details. In a functional programming language, one could represent the matrix as a collection of rows. While this is a natural choice, splitting containers at a certain position can come with some overhead in Haskell. For example, computing the size of an *IntMap* from *Data.IntMap* takes linear effort in the size of the *IntMap*, and splitting the *IntMap*

takes an additional logarithmic effort. The data type *Data.Set* on the other hand allows constant time size computations (and logarithmic splitting), but is unsuited to represent vectors, because only (totally) ordered elements can be contained in such sets. However, these additional computations can be easily avoided. Instead of explicitly splitting containers, one may use an implicit decomposition that is particularly simple for the employed data structure. For instance, Dolan [Dol13] in fact represents matrices as a list of lists and an efficient decomposition is obtained by choosing $l = 1$ and $m = n - 1$. This choice is implicit, because a given non-empty list is split using pattern matching

$$xs = h : t \,,$$

which is a constant time operation. Unfortunately, this decomposition results in large constants in the complexity estimates, since the computational structure does not work well with non-strictness.

Above, we essentially stated that the non-determinism can be avoided by a fixed decomposition choice. Clearly, it is also possible to employ actual non-determinism using a logic or functional logic language to split matrices non-deterministically. We will not examine such an implementation in this section, but we discuss non-determinism in graph algorithms in Section 9.2.2.

3.3 A Functional Approach

In this section we develop a purely functional version of the Kleene closure. This version originates from a theoretical observation, but is suitable for an implementation as we show later. Our approach is a direct (and proper) generalisation of the methods employed by Berghammer [Ber11] to develop a functional version of the Warshall algorithm, which is a special case of the Kleene closure algorithm. For the sake of simplicity we use a slightly more sophisticated structure than a Kleene algebra, namely a so-called Kleene algebra with tests. Tests are Kleene algebra elements that display the same properties as elements of a Boolean algebra.

While it may seem that this choice constitutes a proper restriction, it turns out not to be the case: every Kleene algebra can be considered a trivial Kleene algebra with tests. Thus all theoretical results hold in case of Kleene algebras, too. When it comes to practical applications for arbitrary Kleene algebras, this consideration turns out to be too weak for a useful definition. Still, the Kleene algebra of square matrices over a Kleene algebra, which is a particularly interesting application, is in fact a non-trivial Kleene algebra with tests. Thus our results are not only of theoretical interest. What is more, every Kleene closure computation can be expressed as the computation of the Kleene closure of a matrix as we will see shortly. Thus our choice of Kleene algebras with tests is not a restriction, but merely a slight detour.

We use a similar definition and notation for Kleene algebras with tests as has been presented by Kozen and Smith [KS96].

3.3.1 Definition (Kleene algebra with tests)**.**
A Kleene algebra with tests (KAT for short) is a structure $(K, B, +, \cdot, ^\star, 0, 1)$ such that all of the following conditions hold:

(KAT1) $(K, +, \cdot, ^\star, 0, 1)$ is a Kleene algebra.

(KAT2) $B \subseteq K$.

(KAT3) $(B, +_B, \cdot_B, 0, 1)$ is a Boolean algebra, where $+_B, \cdot_B$ are the operations of K restricted to B.

We usually omit the indices and use the Kleene algebra symbols for the respective operations in the Boolean algebra. ⫽

The concept of abstract tests is designed to provide an elegant means to express Boolean conditions about Kleene algebra elements in terms of Kleene algebra elements. This abstraction allows a unified, algebraic approach to what is usually accomplished with Hoare triples [Hoa83]. Hoare triples, usually denoted $\{ \varphi \} \, e \, \{ \psi \}$, consist of a precondition φ, an expression e and a postcondition ψ and have the following semantics: given the validity of φ, the execution of the expression e results in the validity of the postcondition ψ. These semantics are often used in program verification, which explains the term "execution" above (the expression e is typically an assignment in an imperative language). However, there is a distinction between *conditions* and *expressions*. The conditions come from some logic (Boolean logic, temporal logic or similar) and the expressions are elements of a language structure (assignments, or applications). Kleene algebras with tests allow treating such constructs using a single algebra and a special subalgebra. For example, an *if-then-else* assignment can be written as

$$ite(b, x, y) := b \cdot x + \overline{b} \cdot y \ ,$$

where $b \in \{ 0, 1 \}$, $x, y \in K$ and $\overline{b}$ denotes the complement of b. This construct is similar to the Shannon decomposition of Boolean functions, but is more general since the values x, y can be arbitrary Kleene algebra elements.

We proceed with some common examples of Kleene algebras with tests.

3.3.2 Example (Kleene algebra with tests).

(1) Let $(K, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra. Then $(K, \{ 0, 1 \}, +, \cdot, ^\star, 0, 1)$ is a Kleene algebra with tests.

(2) Let $(\mathfrak{T}, \mathcal{R}\,(,)\,, \cdot, \mathsf{I})$ be a relation algebra, $T \in \mathfrak{T}$, and set

$$P := \{ r \in \mathcal{R}\,(T, T) \mid r \subseteq \mathsf{I} \} = \mathsf{PI}_{\mathcal{R}(T,T)}(\mathcal{R}\,(T, T)) \ .$$

Then $(\mathcal{R}\,(T, T)\,, P, \cup, \cdot, ^\star, \mathsf{O}, \mathsf{I})$ is a Kleene algebra with tests, where * denotes the reflexive-transitive closure (cf. Example 3.2.2.(1)).

(3) Let $(K, B, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra with tests, $n \in \mathbb{N}$, and set

$$P := \{ a \in B^{n\times n} \mid a \leqslant_n 1_n \} = \mathsf{PI}_{K^{n\times n}}\left(B^{n\times n}\right) ,$$

where $\leqslant_n$ is the idempotent semiring order in the matrix semiring. Then the structure $(K^{n\times n}, P, +, \cdot, ^*, 0, 1)$ is a Kleene algebra with tests[3].

(4) Let $(K, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra, $n \in \mathbb{N}$, and set

$$P := \left\{ a \in K^{n\times n} \;\middle|\; \exists\, S \in \mathcal{P}(\mathbb{N}_{<n}) : a = \sum_{i \in S} \mathsf{e}(i,i) \right\} .$$

Then $(K^{n\times n}, P, +, \cdot, ^*, 0, 1)$ is a Kleene algebra with tests. //

For the sake of completeness we provide a proof of all of the above statements in Appendix A.1.4. Note that the statement in Example 3.3.2.(4) is true for every Kleene algebra, not just in Kleene algebras with tests. Also note that the Boolean algebras in Example 3.3.2.(2), Example 3.3.2.(3) and Example 3.3.2.(4) have a very similar structure, which is summarised in Lemma A.1.3. Since we are aiming for a function that computes the matrix closure, we see that the choice of a Kleene algebra with tests is not a restriction.

Now suppose that $(K, B, +, \cdot, ^\star, 0, 1)$ is a Kleene algebra with tests. We then consider the following function:

$$\tau : K \times B \to K , \quad (a, b) \mapsto a \cdot (b \cdot a)^\star .$$

This function is a translation of the function in relational terms by Berghammer [Ber11]. The key idea behind this function is to describe how to compute the Kleene closure of any element, by first computing the Kleene closure of some other elements. The value $b \cdot a$ can be thought of as a restriction of a to b and thus the idea behind the computation is to compute the closure $(b \cdot a)^\star$, which is likely to be simpler than $a^\star$, and then use it in the remaining term.

Just as its relational version, τ has the following properties for all $a \in K$:

$$
\begin{aligned}
& \tau(a, 0) \\
= \;& \{\!\!\{ \text{ Definition of } \tau \text{ } \}\!\!\} \\
& a \cdot (0 \cdot a)^\star \\
= \;& \{\!\!\{ \; 0 \cdot a = 0 \text{ since } 0 \text{ is annihilating by Definition 2.3.3.}\textit{(SR4)} \; \}\!\!\} \\
& a \cdot (0^\star) \\
= \;& \{\!\!\{ \; 0^\star = 1 \text{ by Corollary 3.2.4.(1) } \}\!\!\} \\
& a \cdot 1 \\
= \;& \{\!\!\{ \; 1 \text{ is neutral with respect to multiplication by Definition 2.3.3.}\textit{(SR2)} \; \}\!\!\} \\
& a
\end{aligned}
\tag{3.3.2.i}
$$

[3] We use Convention 2.3.8 for the operations and constants in $K^{n\times n}$.

and similarly

$$
\begin{aligned}
& \tau(a, 1) \\
= \quad & \wr \text{ Definition of } \tau \ \wr \\
& a \cdot (1 \cdot a)^\star \\
= \quad & \wr \text{ 1 is neutral with respect to multiplication by Definition 2.3.3.}\textit{(SR2)} \ \wr \\
& a \cdot a^\star \\
= \quad & \wr \text{ Definition of the Kleene closure (Definition 3.2.1) } \wr \\
& a^+ \ . \qquad\qquad (3.3.2.\text{ii})
\end{aligned}
$$

To deal with tests between 0 and 1 we take a similar approach as Berghammer [Ber11] and study the recursion properties of τ. We observe that for all $a \in K$ and all $b_1, b_2 \in B$ we get the following chain of equations[4]:

$$
\begin{aligned}
& \tau(a, b_1 + b_2) \\
= \quad & \wr \text{ definition of } \tau \ \wr \\
& a \cdot ((b_1 + b_2) \cdot a)^\star \\
= \quad & \wr \text{ by distributivity, Definition 2.3.3.}\textit{(SR3)} \ \wr \\
& a \cdot (b_1 \cdot a + b_2 \cdot a)^\star \\
= \quad & \wr \text{ by Proposition 3.2.3.(3) } \wr \\
& a \cdot \left((b_1 \cdot a)^\star \cdot ((b_2 \cdot a) \cdot (b_1 \cdot a)^\star)^\star\right) \\
= \quad & \wr \text{ by associativity, Definition 2.3.3.}\textit{(SR2)} \ \wr \\
& (a \cdot (b_1 \cdot a)^\star) \cdot (b_2 \cdot (a \cdot (b_1 \cdot a)^\star))^\star \\
= \quad & \wr \text{ definition of } \tau \ \wr \\
& \tau(a, b_1) \cdot (b_2 \cdot \tau(a, b_1))^\star \\
= \quad & \wr \text{ definition of } \tau. \ \wr \\
& \tau(\tau(a, b_1), b_2) \ .
\end{aligned}
$$

In summary we get for all $a \in K$ and all $b_1, b_2 \in B$

$$\tau(a, b_1 + b_2) = \tau(a, b_1) \cdot (b_2 \cdot \tau(a, b_1))^\star = \tau(\tau(a, b_1), b_2) \ . \qquad (3.3.2.\text{iii})$$

All three equations (Equations (3.3.2.i), (3.3.2.ii), and (3.3.2.iii)) are generalisations of the relational counterparts derived by Berghammer [Ber11], both in terms of the decomposition of the underlying set as well as the algebraic structure.

Now let us consider the above recursion in case that there exists a finite subset $B' \subseteq B$, such that[5] $\sum_{b \in B'} b = 1$. Since B' is finite, there exists an $m \in \mathbb{N}$ and a bijection $\beta : \mathbb{N}_{<m} \to B$. Then we can use the above formula and compute $a_0 := a$ and $a_{i+1} := \tau(a_i, \beta(i))$ for all $i \in \mathbb{N}_{<m}$. A straightforward induction shows that we get the

[4]Save for the generalised arguments, this computation is the one performed by Berghammer [Ber11].

[5]Such a set, where $b < 1$ for all $b \in B'$ holds, does not necessarily exist. However, if B is finite, we can simply choose the atoms of B (i.e. the upper neighbours of 0) as B'.

following property of this construction:

$$\forall\, k \in \mathbb{N}_{\leqslant m} : a_k = \tau \left(a, \sum_{j=0}^{k-1} \beta(j) \right) .$$

This property allows a non-recursive computation of the list $(a_i)_{i=0}^{m}$. Additionally, it can be used to compute the Kleene closure of a, because

$$a_m = \tau \left(a, \sum_{j=0}^{m-1} \beta(j) \right) = \tau \left(a, \sum_{b \in B'} b \right) = \tau\,(a, 1) = a^{+}$$

holds, where the last equality holds by Equation (3.3.2.ii). Note that the construction of the auxiliary list $(a_i)_{i=0}^{m}$ is structurally very similar to the construction given in Equation (3.2.5.i). The key difference is that the above statement is true in any Kleene algebra (with tests) and does not depend on matrices.

The computational paradigm of Equation (3.3.2.iii) is similar to a left-fold, because each successive value is computed from the previous one (cf. Section 2.5). Note that this property does not depend on a particular order of traversal of the partition elements, but is intrinsic to the actual computation. In the above computation we split the sum into its first summand and the rest, while we could have taken the last element and the corresponding rest as well without changing the result. The order of traversal is given by the function β and is thus irrelevant to the actual computation. Depending on the actual problem, one might even parametrise over the traversal order and choose different orders for different elements.

Left-folds are favoured over right-folds in strict languages since they are usually more efficient (tail-call). In a lazy setting tail-calls can become more complex, because the accumulation parameter is not evaluated until needed, while its construction can become increasingly more complex. Additionally, right-folds are usually favoured in a lazy setting, because they often can yield partial results, while left-folds are not suited for this task (we have discussed this briefly in Section 2.5 using the example of the function *and*). The reason for that is that the outer-most call in a left-fold is the left-fold function itself, rather than an operation that may provide partial information about the accumulator. In particular, the accumulation parameter cannot be accessed until the structure has been folded completely. Depending on the application, this may have a strong effect on the memory consumption of a right-folded variant of a function when compared to the left-folded one.

Since we are looking for a solution in a lazy functional language, we may need to transform the above recursion into a right-fold[6]. To achieve that we will determine the Kleene closure for a specific subset of K, namely

$$\{\, b \cdot a \mid b \in B \wedge a \in K \,\}$$

[6]It is a well-known fact [Hut99] that any left-fold can be expressed in terms of a right-fold, but not vice versa. However, this new expression does not remedy the above mentioned problems with left-folds. This is why our aim is a non-generic transformation.

and then apply τ in a recursive fashion as described above, assuming that there is a decomposition of 1 as a sum of finitely many tests. By Equation (3.3.2.iii) the knowledge of * on the above set is enough to compute the Kleene closure of any element of the Kleene algebra. This approach is natural in the sense that given an element $b \in B$ and an $a \in K$ the value $b \cdot a$ can be thought of as a restriction of a to b. The recursion equation for τ, Equation (3.3.2.iii), states that we can reduce the restriction step by step until we restrict to 1, which constitutes no restriction at all, since $1 \cdot a = a$. To actually arrive at this last value in an inductive fashion, one should choose proper tests for the decomposition of 1, which is to say tests that are strictly smaller than 1. However, this is obviously not necessary in theory.

Let us summarise the result of this section in the following theorem.

3.3.3 Theorem (Recursion properties of τ).
Let $(K, B, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra with tests and let

$$\tau : K \times B \to K , \quad (a, b) \mapsto a \cdot (b \cdot a)^\star .$$

Then the following hold:

(1) For all $a \in K$ we have $\tau(a, 0) = a$.

(2) For all $a \in K$ we have $\tau(a, 1) = a^+$.

(3) For all $a \in K$ and all $b, c \in B$ we have

$$\tau(a, b + c) = \tau(a, b) \cdot (c \cdot \tau(a, b))^\star .$$

(4) For all $a \in K$, all $m \in \mathbb{N}$, all $b \in B^m$ and all $i \in \mathbb{N}_{<m}$ setting $n_i := \sum_{j=0, j \neq i}^{m-1} b_j$ we find that

$$\tau\left(a, \sum_{j=0}^{m-1} b_j\right) = \tau\left(a, n_i\right) \cdot \left(b_i \cdot \tau(a, n_i)\right)^\star .$$

Proof. The statements (1), (2) and (3) are just repetitions of the Equations (3.3.2.i), (3.3.2.ii) and (3.3.2.iii) respectively.
Statement (4). We have that $b_i + n_i = \sum_{j=0}^{m-1} b_j$. Thus the result is an immediate consequence of Statement (3). ⫽

Note that while the statement in Theorem 3.3.3.(4) looks like we subtract the value b_i, this is not necessarily the case. Since the addition in a Kleene algebra is idempotent, the same result holds true even if we replace n_i by the full sum $\sum_{j=0}^{m-1} b_j$, because

$$b_i + \sum_{j=0}^{m-1} b_j = \sum_{j=0}^{m-1} b_j .$$

One intention of the statement is to reduce the number of summands and thus the number of recursion steps, when it comes to an implementation. Clearly, an efficient implementation requires a choice of b that does not contain the same value

twice, because otherwise an unnecessary computation is performed. However, in the theoretical framework, this does not matter at all.

The second intention of the statement is to provide a good starting point for a computation of a^+ for a given $a \in K$. We achieve this by choosing a fitting injective[7] $b \in B^m$ such that

$$\sum_{j=0}^{m-1} b_j = 1 \ .$$

Since $\tau(a,1) = a^+$, we can use the discussed statement to split off the summands of the sum until no summands are left. Then we use $\tau(a,0) = a$ and finish the recursion. The main task is to find a $b \in B^m$ such that for all $i \in \mathbb{N}_{<m}$ the computation of $(b_i \cdot \tau(a, n_i))^\star$ is simple in the sense that no further recursion of τ is required[8].

3.4 Application to Square Matrices

In this section we use the closure function we have just developed to obtain a closure function for square matrices. While the generic approach is clearly applicable in the particular case of the Kleene algebra of square matrices, it is not obvious that it can be transformed into a variant, which can be computed in terms of a right-fold.

As we have mentioned before in Example 3.3.2.(4), given a Kleene algebra K and an $n \in \mathbb{N}$ the set

$$B := \left\{ a \in K^{n\times n} \;\middle|\; \exists\, S \in \mathcal{P}(\mathbb{N}_{<n}) : a = \sum_{i \in S} \mathsf{e}(i,i) \right\} = \mathsf{PI}_{K^{n\times n}}\left(\{\, 0,1 \,\}^{n\times n} \right)$$

is a Boolean algebra (this statement is proved in Proof A.1.4). Thus we can apply the previous result to the Kleene algebra with tests $(K^{n\times n}, B, +, \cdot, ^*, 0, 1)$. The key idea is to decompose 1 in $K^{n\times n}$ into a finite number of tests. One particular decomposition is obtained by setting $b_i := \mathsf{e}(i,i)$ for all $i \in \mathbb{N}_{<n}$. The prerequisite on this decomposition, which we have discussed before Theorem 3.3.3, is that the closure of $b_i \cdot a$ (where $a \in K^{n\times n}$) should be computable without any additional closures in $K^{n\times n}$. We now show that this is the case. To that end we first need three auxiliary lemmas.

The first lemma deals with the star closure of homothetic matrices, which are scalar multiples of the identity matrix. These matrices are closed under the closure operation, which allows a simple computation of their closure.

3.4.1 Lemma (Homothetic closure)**.**
Let $(K, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra and $n \in \mathbb{N}$. Let $\varphi : K \to K^{n\times n}$, $c \mapsto c \bullet 1_n$, where $\bullet : K \times K^{n\times n} \to K^{n\times n}$ is the scalar multiplication of a matrix. Then φ is a Kleene algebra homomorphism. In particular, we have the following rule:

$$\forall\, a \in K : a^\star \bullet 1_n = (a \bullet 1_n)^* \ ,$$

[7] Recall that $b \in B^n$ is actually a function $b : \mathbb{N}_{<n} \to B$, as we discussed in Section 2.2.

[8] Otherwise, we might end up in an infinite loop.

*where * denotes the star closure of matrices.*

Proof. Clearly, $\varphi(0_K) = 0_n$ and $\varphi(1_K) = 1_n$ by the very definition of scalar multiplication. The additivity and multiplicativity of φ are immediate consequences of its definition as well. The only difficulty is the star operation. Let $a \in K$. Then we have

$$\begin{aligned}
& 1_n + \varphi(a) \cdot \varphi(a^\star) \\
= \quad & \wr \text{ since } \varphi(1) = 1_n \text{ and } \varphi \text{ is multiplicative } \int \\
& \varphi(1) + \varphi(a \cdot a^\star) \\
= \quad & \wr \text{ additivity of } \varphi \int \\
& \varphi(1 + a \cdot a^\star) \\
= \quad & \wr \text{ by Proposition 3.2.3.(1) } \int \\
& \varphi(a^\star) \ .
\end{aligned}$$

By Proposition 3.2.3.(2) this provides $\varphi(a)^* \leqslant \varphi(a^\star)$. Thus $\varphi(a)^*$ has non-zero entries at most along its main diagonal. Let $i \in \mathbb{N}_{<n}$. Then we have

$$\begin{aligned}
& (\varphi(a)^*)_{i,i} \\
= \quad & \wr \text{ by Proposition 3.2.3.(1) } \int \\
& (1_n + \varphi(a) \cdot \varphi(a)^*)_{i,i} \\
= \quad & \wr \text{ additivity of the index function, definition of } 1_n \text{ and } \varphi \int \\
& 1 + ((a \bullet 1) \cdot \varphi(a)^*)_{i,i} \\
= \quad & \wr \text{ definition of } \cdot\text{, which is matrix multiplication in this case } \int \\
& 1 + \sum_{j=1}^{n} (a \bullet 1)_{i,j} \cdot (\varphi(a)^*)_{j,i} \\
= \quad & \wr\ a \bullet 1 \text{ is zero outside its main diagonal and } a \text{ along it } \int \\
& 1 + a \cdot (\varphi(a)^*)_{i,i} \ .
\end{aligned}$$

Again, Proposition 3.2.3.(2) yields $\varphi(a^\star)_{i,i} = a^\star \leqslant (\varphi(a)^*)_{i,i}$. Thus $\varphi(a^\star) \leqslant \varphi(a)^*$ and since $\leqslant$ is an order, we conclude $\varphi(a^\star) = \varphi(a)^*$. ⫽

While not particularly deep, the result is still interesting, because it allows to compute every star closure in terms of a matrix closure. Obviously, the more general matrix closure can have a higher complexity, but from a merely theoretical view we get that computing matrix closures is as general as computing arbitrary closures in Kleene algebras.

Next, we compute the closure of Kleene algebra elements whose square is a multiple of the element itself. The following lemma generalises the statement of Proposition 3.2.3.(1), which is obtained using the lemma with $c = v$.

3.4.2 Lemma (Generalised idempotent closure).
Let $(K, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra and $c, v \in K$ such that $v^2 = c \cdot v$. Then we have $v^\star = 1 + c^\star \cdot v$.

Proof. Set $a := c$, $b := v$ and $x := v$. Then we have $a \cdot x = c \cdot v = v \cdot v = x \cdot b$, hence by Proposition 3.2.3.(4) we get $c^\star \cdot v = a^\star \cdot x = x \cdot b^\star = v \cdot v^\star$, which by Proposition 3.2.3.(1) yields $v^\star = 1 + v \cdot v^\star = 1 + c^\star \cdot v$. ⫽

The final step is to note two properties of matrices of the form $\mathsf{e}(k,k) \cdot a$.

3.4.3 Lemma (Properties of restrictions to atoms).
Let $(S, +, \cdot, 0, 1)$ be a semiring and $n \in \mathbb{N}$. Then the following hold:

(1) For all $a \in S^{n\times n}$ we have

$$(\mathsf{e}(k,k) \cdot a)^2 = (a_{k,k} \bullet 1) \cdot \mathsf{e}(k,k) \cdot a \ .$$

(2) For all $a, b \in S^{n\times n}$, all $s \in S$ and all $j, k \in \mathbb{N}_{<n}$ we have

$$(a \cdot (s \bullet \mathsf{e}(k,k)) \cdot b)_j = \left(a_{j,k} \cdot s\right) \bullet b_k \ .$$

Proof. Statement (1). Let $a \in K^{n\times n}$, $k \in \mathbb{N}_{<n}$. First, we get

$$\begin{aligned}
& \mathsf{e}(k)^\top \cdot \mathsf{e}(k) \cdot a \cdot \mathsf{e}(k)^\top \\
= \ & \wr \text{ properties of standard unit vectors, Proposition 2.4.5.(2) } \wr \\
& \mathsf{e}(k)^\top \cdot (a_{k,k}) \\
= \ & \wr \text{ by Proposition 2.4.6.(2) } \wr \\
& a_{k,k} \bullet \mathsf{e}(k)^\top \\
= \ & \wr \text{ by Proposition 2.4.6.(1) } \wr \\
& (a_{k,k} \bullet 1) \cdot \mathsf{e}(k)^\top \ .
\end{aligned} \tag{3.4.3.a}$$

Thus we have

$$\begin{aligned}
& (\mathsf{e}(k,k) \cdot a)^2 \\
= \ & \wr \text{ definition of } \mathsf{e}(k,k)\text{, Definition 2.4.4 } \wr \\
& \mathsf{e}(k)^\top \cdot \mathsf{e}(k) \cdot a \cdot \mathsf{e}(k)^\top \cdot \mathsf{e}(k) \cdot a \\
= \ & \wr \text{ Equation (3.4.3.a) } \wr \\
& (a_{k,k} \bullet 1) \cdot \mathsf{e}(k)^\top \cdot \mathsf{e}(k) \cdot a \\
= \ & \wr \text{ definition of } \mathsf{e}(k,k)\text{, Definition 2.4.4 } \wr \\
& (a_{k,k} \bullet 1) \cdot \mathsf{e}(k,k) \cdot a \ .
\end{aligned}$$

Statement (2). Let $a, b \in S^{n\times n}$, $s \in S$ and $j, k \in \mathbb{N}_{<n}$. Then we have

$$\begin{aligned}
& (a \cdot (s \bullet \mathsf{e}(k,k) \cdot b))_j \\
= \ & \wr \text{ row access via standard unit vectors, Proposition 2.4.5.(1) } \wr \\
& \mathsf{e}(j) \cdot a \cdot (s \bullet \mathsf{e}(k,k) \cdot b) \\
= \ & \wr \text{ definition of } \mathsf{e}(k,k)\text{, Definition 2.4.4 } \wr \\
& \mathsf{e}(j) \cdot a \cdot \left(s \bullet \mathsf{e}(k)^\top \cdot \mathsf{e}(k) \cdot b\right)
\end{aligned}$$

$$
\begin{aligned}
&= \quad \wr \text{ by Proposition 2.4.6.(2) } \wr \\
&\quad \mathsf{e}(j) \cdot a \cdot \left(\mathsf{e}(k)^\top \cdot (s \bullet \mathsf{e}(k) \cdot b)\right) \\
&= \quad \wr \text{ associativity of matrix multiplication } \wr \\
&\quad \mathsf{e}(j) \cdot a \cdot \mathsf{e}(k)^\top \cdot (s \bullet \mathsf{e}(k) \cdot b) \\
&= \quad \wr \text{ by Proposition 2.4.5.(2) and Proposition 2.4.5.(1) } \wr \\
&\quad (a_{j,k}) \cdot (s \bullet b_k) \\
&= \quad \wr \text{ by Proposition 2.4.6.(3) } \wr \\
&\quad (a_{j,k} \cdot s) \bullet b_k \ .
\end{aligned}
$$

We can now use the above tools to first compute the Kleene closures of matrices in the set

$$\left\{ \mathsf{e}(k,k) \cdot a \;\middle|\; k \in \mathbb{N}_{<n} \wedge a \in K^{n \times n} \right\} ,$$

which are matrices that are restricted to individual rows. This result can be used to obtain the Kleene closure of a general matrix using the function τ from Section 3.3.

3.4.4 Theorem (Kleene closure of matrices)**.**
Let $(K, +, \cdot, ^\star, 0, 1)$ be a Kleene algebra, $n \in \mathbb{N}$ and $B \subseteq K^{n \times n}$ be a Boolean algebra such that

$$\{ \mathsf{e}(k,k) \mid k \in \mathbb{N}_{<n} \} \subseteq B \ .$$

Let

$$\tau : K^{n \times n} \times B \to K^{n \times n} , \quad (a,b) \mapsto a \cdot (b \cdot a)^* \ .$$

Then the following hold:

(1) For all $a \in K^{n \times n}$ and all $k \in \mathbb{N}_{<n}$ we have

$$(\mathsf{e}(k,k) \cdot a)^* = 1_n + (a_{k,k})^\star \bullet (\mathsf{e}(k,k) \cdot a) \ .$$

(2) For all $a \in K^{n \times n}$, all $b \in B$ and all $k \in \mathbb{N}_{<n}$ we have

$$\tau\left(a, \mathsf{e}(k,k) + b\right) = \tau(a,b) + \tau(a,b) \cdot \left((\tau(a,b)_{k,k})^\star \bullet \mathsf{e}(k,k)\right) \cdot \tau(a,b) \ .$$

(3) For all $a \in K^{n \times n}$, all $b \in B$ and all $k, j \in \mathbb{N}_{<n}$ we have

$$\tau\left(a, \mathsf{e}(k,k) + b\right)_j = \tau(a,b)_j + \left(\tau(a,b)_{j,k} \cdot (\tau(a,b)_{k,k})^\star\right) \bullet \tau(a,b)_k \ .$$

Proof. Statement (1). Let $a \in K^{n \times n}$, $k \in \mathbb{N}_{<n}$. By Lemma 3.4.3.(1) we have

$$(\mathsf{e}(k,k) \cdot a)^2 = (a_{k,k} \bullet 1) \cdot \mathsf{e}(k,k) \cdot a \ .$$

Now we can apply Lemma 3.4.2 to calculate as follows:

$$
\begin{aligned}
&\quad (\mathsf{e}(k,k) \cdot a)^* \\
&= \quad \wr \text{ by Lemma 3.4.2 } \wr \\
&\quad 1 + (a_{k,k} \bullet 1)^* \cdot \mathsf{e}(k,k) \cdot a \\
&= \quad \wr \text{ by Lemma 3.4.1 } \wr
\end{aligned}
$$

$$\begin{aligned}
& 1 + ((a_{k,k})^\star \bullet 1) \cdot \mathsf{e}(k,k) \cdot a \\
= \;& \wr \text{ property of scalar multiplication, Proposition 2.4.6.(1) } \int \\
& 1 + (a_{k,k})^\star \bullet \mathsf{e}(k,k) \cdot a \;.
\end{aligned}$$

Statement (2). Let $a \in K^{n\times n}$, $b \in B$ and $k \in \mathbb{N}_{<n}$. Then we have

$$\begin{aligned}
& \tau(a, \mathsf{e}(k,k) + b) \\
= \;& \wr \text{ addition is commutative } \int \\
& \tau(a, b + \mathsf{e}(k,k)) \\
= \;& \wr \text{ Theorem 3.3.3.(3) } \int \\
& \tau(a,b) \cdot (\mathsf{e}(k,k) \cdot \tau(a,b))^* \\
= \;& \wr \text{ by Theorem 3.4.4.(1) } \int \\
& \tau(a,b) \cdot (1 + (\tau(a,b)_{k,k})^\star \bullet (\mathsf{e}(k,k) \cdot \tau(a,b))) \\
= \;& \wr \text{ distributivity and Proposition 2.4.6.(1) } \int \\
& \tau(a,b) + \tau(a,b) \cdot ((\tau(a,b)_{k,k})^\star \bullet \mathsf{e}(k,k)) \cdot \tau(a,b) \;.
\end{aligned}$$

Statement (3). Now let $a \in K^{n\times n}$, $b \in B$ and $j, k \in \mathbb{N}_{<n}$. Then we calculate:

$$\begin{aligned}
& \tau(a, \mathsf{e}(k,k) + b)_j \\
= \;& \wr \text{ by Theorem 3.4.4.(2) } \int \\
& \big(\tau(a,b) + \tau(a,b) \cdot \big((\tau(a,b)_{k,k})^\star \bullet \mathsf{e}(k,k)\big) \cdot \tau(a,b)\big)_j \\
= \;& \wr \text{ additivity of the index function } \int \\
& \tau(a,b)_j + \big(\tau(a,b) \cdot \big((\tau(a,b)_{k,k})^\star \bullet \mathsf{e}(k,k)\big) \cdot \tau(a,b)\big)_j \\
= \;& \wr \text{ by Lemma 3.4.3.(2) } \int \\
& \tau(a,b)_j + \big(\tau(a,b)_{j,k} \cdot (\tau(a,b)_{k,k})^\star\big) \bullet \tau(a,b)_k \;.
\end{aligned}$$

//

Assuming that we have already computed the value $\tau(a,b)$ for a given $a \in K^{n\times n}$ and a test $b \in B$, we can compute $\tau(a, \mathsf{e}(k,k) + b)$ without any additional applications of τ. Recall that $\tau(a, 1_n) = a^+$ and thus we can use the decomposition

$$1_n = \sum_{i=1}^{n} \mathsf{e}(i,i)$$

and the recursion steps of the previous theorem to compute a^+. Such a decomposition is possible in the chosen Boolean algebra, because it contains all standard unit matrices and thus also all possible sums of these matrices. It is important to note that a Boolean algebra as required by Theorem 3.4.4 always exists in the matrix Kleene algebra $K^{n\times n}$, because we can choose

$$B = \left\{ a \in K^{n\times n} \;\middle|\; \exists\, S \in \mathcal{P}(\mathbb{N}_{<n}) : a = \sum_{i\in S} \mathsf{e}(i,i) \right\} .$$

This set is in fact a Boolean algebra, as we have already mentioned at the beginning

of this section. Clearly, B satisfies the condition

$$\{\, \mathsf{e}(k,k) \mid k \in \mathbb{N}_{<n} \,\} \subseteq B \;.$$

Furthermore, B is a minimal Boolean algebra with respect to inclusion that satisfies this property, because any other Boolean algebra that contains $\{\, \mathsf{e}(k,k) \mid k \in \mathbb{N}_{<n} \,\}$ also contains all finite sums of these elements, but this is exactly the set B.

Note that while Theorem 3.4.4.(2) uses matrix multiplication which is an a-priori cubic operation in the matrix dimension, the statement in Theorem 3.4.4.(3) uses only vector operations which are linear in the length of the vector. Additionally, the computational structure is now rather a right-fold, because the outermost operation is a vector addition and no longer an application of τ. This fact constitutes an advantage in Kleene algebras whose addition (and thus also the addition of vectors over these Kleene algebras) is sufficiently lazy. Indeed, if $+$ can provide *partial* information before evaluating both of its arguments, partial information about the Kleene closure may be obtained as well. A particularly interesting case is the case of Boolean matrices, where the addition is the logical disjunction. Once the first argument of the disjunction is *True*, the second one (and all subsequent ones, too) do not need to be evaluated, because the result of the overall addition is *True*. While this is an extreme case, the actual benefits depend on the application as well. If the application requires partial information about the Kleene closure only and the addition of the given Kleene algebra is sufficiently lazy, the computation of the complete Kleene closure can be avoided.

3.5 A Functional Implementation

In this section we implement the Kleene closure function for square matrices. As we have mentioned before, a large portion of the theoretical framework we have presented is applicable in the general setting of a Kleene algebra with tests. The results of the previous section indicate that specific matrix operations can lead to a more efficient implementation. However, there are other reasons for our specialisation, which we discuss in a moment.

We begin with an implementation of semirings and Kleene algebras. Typically, algebraic structures are encoded in terms of type classes and we choose this way, too.

```
class Semiring σ where
  (⊕), (⊛)      :: σ → σ → σ
  zero, one     :: σ
  isZero, isOne :: σ → Bool

class Semiring σ ⇒ IdempotentSemiring σ

class IdempotentSemiring κ ⇒ KleeneAlgebra κ where
  star :: κ → κ
```

Additionally, we require instances of these type classes to satisfy the corresponding algebraic laws. Such an approach requires a user to check the necessary conditions, which may or may not be neglected in an application[9]. The idempotence is provided by the trivial definition

```
instance IdempotentSemiring S
```

for a given data type S. Again, this is a safe-guard that a user should provide this trivial instance only if the addition is in fact idempotent.

The predicates *isZero*, *isOne* are not explicit parts of the algebraic definition. In a theoretical setting we can use an abstract notion of object equality for arbitrary objects, while in practice this equality may be undecidable. We require equality checks for constants only, since these are sufficient for a simple optimisation (otherwise one can require *KleeneAlgebra* to be a subclass of *Eq* thus allowing arbitrary equality checks).

For the general case of Kleene algebras with tests we use another type class.

```
class IdempotentSemiring κ ⇒ KAT κ where
  isSimple     :: κ → Bool
  simpleKleene :: κ → κ
  sparseTests  :: [κ]
```

The intended semantics are the following:

(1) The list *sparseTests* is a list of tests such that their sum is *one*.

(2) The predicate *isSimple* and the (possibly partial) function *simpleKleene* provide the property that the Kleene closure of those $a :: \kappa$ that satisfy[10] *isSimple a* can be computed with the function *simpleKleene*.

Note that *KAT* is not a subclass of *KleeneAlgebra*, but only of *IdempotentSemiring*. The reason for this definition is that *KAT*s provide a means to compute the Kleene closure (and thus also the star closure) generically. If *KAT* was a subclass of *KleeneAlgebra*, we could not provide instances by using the generic *KAT* Kleene closure function as the definition of *star*. This implementation deviates from the definition of a Kleene algebra with tests. However, every Kleene algebra with tests can be made an instance of the above type class by taking the trivial tests, only. Suppose that K is a concrete Kleene algebra and we have already provided an instance declaration for *IdempotentSemiring*. Then we can add the following code to obtain a *KAT*.

```
instance KAT K where
  isSimple x     = isZero x ∨ isOne x
```

[9] To ensure that the requirements are met, one can use proof assistants like Coq [BC04] to make certain functions applicable only once the preconditions are checked (i.e. proved). However, in Haskell one usually requires certain laws implicitly, say for *Monad* or *Functor*, which is why we do not explore the mentioned approach further.

[10] This is a simplified notation, because the data type *Bool* is not the same as the set of Boolean values $\mathbb{B}$. We use this notation only to avoid stating that *isSimple a* is semantically equivalent to *True*.

$$simpleKleene = id$$
$$sparseTests \quad = [zero, one]$$

The idea behind this implementation is to use Corollary 3.2.4, which yields that $0^+ = 0$ and $1^+ = 1$ and to define only the constants as simple. With the additional assumption that the following holds

$$\forall\, a :: \kappa \,.\, \forall\, t \leftarrow sparseTests \,.\, isSimple\ (t \circledast a)\ ,$$

we can implement the results from Theorem 3.3.3 in the following fashion.

$$katStar :: KAT\ \kappa \Rightarrow \kappa \rightarrow \kappa$$
$$katStar\ a$$
$$\quad |\ isSimple\ a = simpleKleene\ a$$
$$\quad |\ otherwise \ \ = one \oplus katPlus\ a$$
$$katPlus :: KAT\ \kappa \Rightarrow \kappa \rightarrow \kappa$$
$$katPlus\ a = \tau\ a\ sparseTests$$
$$\tau :: KAT\ \kappa \Rightarrow \kappa \rightarrow [\kappa] \rightarrow \kappa$$
$$\tau\ a\ [\,] \qquad = a$$
$$\tau\ a\ (t : ts) = x \circledast katStar\ (t \circledast x)\ \textbf{where}\ x = \tau\ a\ ts$$

The only variation is that we do not explicitly decompose *one* by splitting off summands, but use a list of tests, whose sum is *one* by the above preconditions and simply split off the head of the list. This provides a simple decomposition regardless of the actual addition function.

The above code is a straightforward translation of the results in Theorem 3.3.3 in Haskell. However, it is a non-trivial task to provide an instance declaration of *IdempotentSemiring* for square matrices. The reason for this difficulty is that one needs to take the matrix size into account, which can be accomplished in several ways (e.g. checking the size by hand or encoding the size in the type[11]), but cannot be omitted. The reason for this necessity is that without fixed matrix sizes given an (idempotent) semiring S we are considering the set

$$\mathbb{S} := \bigcup \{\, S^{n \times n} \mid n \in \mathbb{N} \,\}\ .$$

To define an instance of the semiring type class, we need a value $\mathbb{1}$ and then setting $\mathbb{S}' := \mathbb{S} \cup \{\, \mathbb{1} \,\}$ a function $\otimes : \mathbb{S}' \times \mathbb{S}' \rightarrow \mathbb{S}'$ such that $(\mathbb{S}', \otimes, \mathbb{1})$ is a monoid (among other things!).

Clearly, one would expect that $\otimes$ is not just any operation, but has something in common with the usual matrix multiplication. For example, one might consider the requirement that the restriction of $\otimes$ to any fixed matrix algebra $K^{n \times n}$ is in fact the classical matrix multiplication. Also, a natural requirement is that invertible matrices

[11] The first method is tedious and introduces unnecessary incompatibilities. The second method can be realised using a simulation of dependent types in Haskell, similar to the one provided by McBride [McB02].

with respect to matrix multiplication are still invertible with respect to $\otimes$. In other words, the following conditions are required for all $n \in \mathbb{N}$:

(1) The restriction $\otimes|_{K^{n\times n} \times K^{n\times n}}$ is the usual matrix multiplication.

(2) Every $a \in K^{n\times n}$ that is invertible with respect to matrix multiplication is also invertible with respect to $\otimes$.

Unfortunately, there is no such operation (and constant). In fact, assume we have found $\otimes$ and $\mathbb{1}$ as desired. Now let $n, m \in \mathbb{N}$. Let 1_n be the identity matrix of $K^{n\times n}$ and 1_m the identity matrix of $K^{m\times m}$. Then we find:

$$1_n = 1_n \cdot 1_n = 1_n \otimes 1_n = \mathbb{1} = 1_m \otimes 1_m = 1_m \cdot 1_m = 1_m \ .$$

In particular, we get $n = m$ and thus there is at most one natural number, which is obviously false.

For every $n \in \mathbb{N}_{>1}$ the set of invertible matrices of $S^{n\times n}$ is non-trivial[12]. Thus the invertibility condition is not just covering corner cases. The implementation of Dolan [Dol13] omits said condition and provides a very elegant solution in which only certain diagonal matrices are invertible. This setting is achieved by distinguishing between scalar matrices (i.e. $c \bullet 1$ for some $c \in K$) and non-scalar matrices. The former matrices do not have a fixed size, while the latter do. Missing positions for addition and multiplication are simply considered to be 0 (which is a known algebraic trick). However, non-scalar matrices are closed under multiplication in this implementation and thus never invertible.

We have seen in Theorem 3.3.3 that a rowwise approach appears to be computationally better suited for an implementation in Haskell. Additionally, this approach comes with the benefit of not depending on a semiring instance for matrices, but merely on some algebraic rules concerning vectors. This is why we choose (row) vectors as a basis for our implementation. For now, we consider the following, simplified representation.

```
type Arc α = (Int, α)
newtype Vec α  = Vec {unVec :: [Arc α]}
newtype Mat α = Mat {matrix :: Vec (Vec α)}
```

The intuition behind *Arc α* is that an arc points to its *Int* value (first component) and is labelled with its *α* value (second component). This intuition is particularly descriptive in case the matrix represents a graph: then an arc is the target of an edge together with its label. A *Vec α* is a wrapped lists of arcs and a matrix is a wrapped vector of vectors (similar to the mathematical notation). For example, the graph from Figure 3.1 is represented as follows in Haskell.

```
exampleGraph :: Mat Char
exampleGraph = Mat (Vec [(0, Vec [(1, 'g'), (3, 'r')]),
```

[12]For instance, the matrix $a := \sum_{k=1}^{n} e(k, n+1-k)$ is always its own inverse.

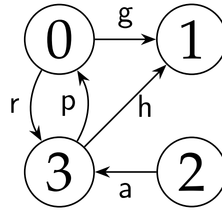


Figure 3.1. An example graph.

$(1, Vec\ [\,]),$
$(2, Vec\ [(3, \texttt{'a'})]),$
$(3, Vec\ [(0, \texttt{'p'}), (1, \texttt{'h'})])])$

For convenience we define a *Functor* instance for vectors as follows[13].

instance *Functor Vec* **where**
  $fmap\ f = Vec \circ map\ (second\ f) \circ unVec$

We assume and maintain the condition that *Vec*s are sorted in ascending order of their first components and that the second components are non-zero (in case of a semiring instance for α). This condition is not guaranteed by the above *Functor* instance, because it is more general, and thus has to be dealt with manually. To avoid repetition and to improve abstraction we define the empty vector and a test, whether a given vector is empty or not.

$emptyVec :: Vec\ \alpha$
$emptyVec = Vec\ [\,]$

$isEmptyVec :: Vec\ \alpha \to Bool$
$isEmptyVec = null \circ unVec$

With these conditions we can implement an indexing operation for rows. First, we observe that on association lists that are ordered in ascending order of their indices, a look-up function can be defined more efficiently than on unordered lists.

$orderedLookup :: Ord\ \kappa \Rightarrow \kappa \to [(\kappa, \alpha)] \to Maybe\ \alpha$
$orderedLookup\ _\ \ [\,] = Nothing$
$orderedLookup\ pos\ ((k, x) : kxs)\ |\ k \equiv pos = Just\ x$
$\quad |\ k < pos = orderedLookup\ pos\ kxs$
$\quad |\ otherwise = Nothing$

Since vectors may not contain a specified key, we first define a parametric index access function that takes a default value and then instantiate it to the case of vectors over semirings.

$withAt :: \alpha \to Vec\ \alpha \to Int \to \alpha$
$withAt\ def\ vec\ pos = fromMaybe\ def\ (orderedLookup\ pos\ (unVec\ vec))$

[13]The function *second* is located in *Control.Arrow* and above we have $second\ f = \lambda(i, x) \to (i, f\ x)$.

$(!) :: \mathit{Semiring}\ \sigma \Rightarrow \mathit{Vec}\ \sigma \to \mathit{Int} \to \sigma$
$(!) = \mathit{withAt}\ \mathit{zero}$

Also, we can access rows of a matrix in a very similar fashion using the empty vector as a default value.

$(!!!) :: \mathit{Mat}\ \alpha \to \mathit{Int} \to \mathit{Vec}\ \alpha$
$(!!!) = \mathit{withAt}\ \mathit{emptyVec} \circ \mathit{matrix}$

The conditions on matrices are that a matrix $a \in K^{n\times n}$ is represented by $\mathit{aMat} :: \mathit{Mat}\ \kappa$ such that $a_i = \mathit{aMat}\ !!!\ i$, and $a_{i,j} = (\mathit{aMat}\ !!!\ i)\ !\ j$ for all $i, j \in \mathbb{N}_{<n}$. This implementation is canonic and convenient in the sense that index access is very similar to its mathematical counterpart. Similar implementations have been used before by Berghammer [Ber11] and Danilenko [Dan12].

Recall that Theorem 3.4.4 requires a scalar multiplication and an addition for rows. Note that in our implementation vectors are a special kind of matrices only up to isomorphism. To avoid this technicality, we use different functions for the scalar multiplication of matrices and vectors: $(\bullet_{\mathsf{v}})$ denotes the scalar multiplication of vectors, while $(\bullet_{\mathsf{m}})$ is the scalar multiplication of matrices. A straightforward implementation is the following one.

$\mathit{filterVec} :: (\alpha \to \mathit{Bool}) \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ \alpha$
$\mathit{filterVec}\ p = \mathit{Vec} \circ \mathit{filter}\ (p \circ \mathit{snd}) \circ \mathit{unVec}$

$\mathit{removeZeroes} :: \mathit{Semiring}\ \sigma \Rightarrow \mathit{Vec}\ \sigma \to \mathit{Vec}\ \sigma$
$\mathit{removeZeroes} = \mathit{filterVec}\ (\mathit{not} \circ \mathit{isZero})$

$(\bullet'_{\mathsf{v}}) :: \mathit{Semiring}\ \sigma \Rightarrow \sigma \to \mathit{Vec}\ \sigma \to \mathit{Vec}\ \sigma$
$s \bullet'_{\mathsf{v}} \mathit{vec} = \mathit{removeZeroes}\ (\mathit{fmap}\ (\lambda v \to s \circledast v)\ \mathit{vec})$

The essential part is that every value is multiplied with the given scalar, which is accomplished by the *fmap* expression. The reason for the additional application of *filterVec* is that in semirings we may have zero divisors (i.e. elements that are non-zero, whose product is zero). Since we wish to maintain the property that all values in the *Vec* are non-zero, we need to manually remove potentially introduced zeroes. However, there is room for some canonic improvement, because in case of the constants *zero*, *one* we already know the result of the scalar multiplication beforehand.

$(\bullet_{\mathsf{v}}) :: \mathit{Semiring}\ \sigma \Rightarrow \sigma \to \mathit{Vec}\ \sigma \to \mathit{Vec}\ \sigma$
$s \bullet_{\mathsf{v}} \mathit{vec}$
$\quad \mid \mathit{isZero}\ s \quad = \mathit{emptyVec}$
$\quad \mid \mathit{isOne}\ s \quad = \mathit{vec}$
$\quad \mid \mathit{otherwise} = \mathit{removeZeroes}\ (\mathit{fmap}\ (s\circledast)\ \mathit{vec})$

As for the addition of vectors, we note that we do not need the tidying step *removeZeroes* when adding vectors with entries from an idempotent semiring, because, as we have shown after Example 2.3.4, in an idempotent semiring the non-zero values are closed under addition. Since we required the first components of a *Vec* to be

sorted in ascending order, we can use a simple merging technique to compute the sum of two vectors. Again, we distinguish the addition of vectors, which is $(+_\mathsf{v})$, from the matrix addition $(+_\mathsf{m})$.

```
(+v) :: IdempotentSemiring σ ⇒ Vec σ → Vec σ → Vec σ
v +v w = Vec (add (unVec v) (unVec w)) where
  add []              y  = y
  add x               [] = x
  add x@((i,v) : ivs) y@((j,w) : jws)
    | i ≡ j           = (i, v ⊕ w) : (add ivs jws)
    | i < j           = (i, v)     : (add ivs y)
    | otherwise       = (j, w)     : (add x   jws)
```

Recall that the function τ from Theorem 3.3.3 depends on a decomposition of 1 into tests. In Theorem 3.4.4 we noted that the tests

$$\{\, e(i,i) \mid i \in \mathbb{N}_{<n} \,\}$$

allow a simple computation of $(e(i,i) \cdot a)^*$. Observe that every such test (and every sum of a finite number of such tests) is uniquely determined by a subset of $\mathbb{N}_{<n}$. To be perfectly accurate, setting

$$B := \left\{ a \in K^{n \times n} \,\middle|\, \exists\, S \in \mathcal{P}(\mathbb{N}_{<n}) : a = \sum_{i \in S} \mathsf{e}(i,i) \right\}$$

we get that the function

$$f : \mathcal{P}(\mathbb{N}_{<n}) \to B\,, \quad S \mapsto \sum_{i \in S} \mathsf{e}(i,i)$$

is a Boolean algebra isomorphism from the Boolean algebra $(\mathcal{P}(\mathbb{N}_{<n}), \cup, \cap, \emptyset, \mathbb{N}_{<n})$ to $(B, +, \cdot, 0, 1)$. Since Theorem 3.4.4 can be expressed without the explicit need for standard unit matrices, but merely some index operations, we essentially replace any application $\tau(a, f(S))$ by $\tau(a, S)$. Due to the fact that f is an isomorphism, this is basically a renaming operation, but representing a subset of $\mathbb{N}_{<n}$ is simpler than representing a partial identity matrix, because we can simply use sorted lists for the former.

To gather the indices of a matrix, we use the following function, which takes a list and produces a list of its indices.

```
rowIndices :: Mat α → [Int]
rowIndices = map fst ∘ unVec ∘ matrix
```

We can then implement the Kleene closure of a matrix in a similar fashion as in the general case in the beginning of this section[14].

```
kleeneClosureMatrix :: KleeneAlgebra κ ⇒ Mat κ → Mat κ
kleeneClosureMatrix a = τ a (rowIndices a)
```

[14]In the applicative terminology of McBride and Paterson [MP08] this is $\tau \circledast$ *rowIndices*.

We now implement a slightly modified version of the function τ from Theorem 3.4.4 in an inductive fashion. First, we have $\tau(a, 0) = a$ for all $a \in K^{n \times n}$ by Theorem 3.3.3.(1). The constant 0 is now represented by the empty list, because $f(\emptyset) = 0$ holds. Thus the base case can be implemented as follows.

$$\tau :: \mathit{KleeneAlgebra}\ \kappa \Rightarrow \mathit{Mat}\ \kappa \rightarrow [\mathit{Int}] \rightarrow \mathit{Mat}\ \kappa$$
$$\tau\ a\ [\,] = a$$

For the complex case, Theorem 3.4.4.(3) states that we can compute $\tau\,(a, f(\{\,k\,\} \cup S))$ as some function applied to $\tau(a, f(S))$. Thus we write

$$\tau\ a\ (k : s) = \mathit{newMat}\ k\ (\tau\ a\ s)$$

and thus delay the actual computation in the auxiliary function *newMat* which implements the vector operations from Theorem 3.4.4.(3).

$$\mathit{newMat} :: \mathit{KleeneAlgebra}\ \kappa \Rightarrow \mathit{Int} \rightarrow \mathit{Mat}\ \kappa \rightarrow \mathit{Mat}\ \kappa$$
$$\mathit{newMat}\ k\ a = \mathit{Mat}\ (\mathit{fmap}\ (\lambda a_j \rightarrow (a_j\,!\,k) \circledast \mathit{star}\ (a_k\,!\,k) \bullet_{\mathsf{v}} a_k +_{\mathsf{v}} a_j)\ (\mathit{matrix}\ a))$$
$$\quad \mathbf{where}\ a_k = a\ !!!\ k$$

Essentially, this is exactly the equation from Theorem 3.4.4.(3). The only modification is that we swapped the arguments of $(+_{\mathsf{v}})$. This transformation is valid, because the commutativity of $(\oplus)$ makes $(+_{\mathsf{v}})$ commutative as well. For right-fold operations it is generally better to add new information to the recursive result from the side that is evaluated first, because this allows a sufficiently lazy folding operation to extract this information before the recursive computation. Experiments have shown that this algebraic modification improves the running time to require about 2/3 of the time required by the original version.

Note that the complete definition of τ is

$$\tau\ a\ [\,] \quad\quad = a$$
$$\tau\ a\ (k : s) = \mathit{newMat}\ k\ (\tau\ a\ s)$$

which by the universal property of *foldr* (as discussed by Hutton [Hut99]) can be rewritten as

$$\tau\ a = \mathit{foldr}\ \mathit{newMat}\ a$$

that in turn can be η-reduced to

$$\tau = \mathit{foldr}\ \mathit{newMat}$$

for maximum simplicity.

Both versions, the generic one from the beginning of this section and the specialised one from above, require lists of tests (or objects that are interpreted as tests) which they traverse. In both cases the list is traversed left-to-right and in case of the specialised version the resulting computation is a proper right-fold. However, the actual order of the tests in the list itself is an implicit parameter and the implementation does not depend on a particular order. In case of the specialised function

above, we used the natural order $[0\,..\,n-1]$ to represent the tests $[e(1,1),\ldots,e(n,n)]$ for simplicity. Still, any other order yields the same result. This freedom of choice of the order is reminiscent of the freedom to choose a row (or column) for Gaussian elimination or to compute a determinant of a matrix[15]. In both cases the chosen order usually depends on the matrix at hand rather than a fixed one. For example, when computing determinants using the Laplace expansion, one typically chooses a row (or column) with many zeroes first, which may not be the first row (or column). Similarly, for Gaussian elimination if the i-th row of a matrix has more zeroes than the first row, it is simpler to eliminate the variable x_i before x_1. As a further example, in a reachability-based application one can interpret the row indices of the matrix as vertices of a graph. Then the breadth-first order (shortest path) or depth-first order (any path) of the vertices represent viable order alternatives, too.

To accommodate this freedom of the order choice, we can simply rewrite the actual function that computes the Kleene closure to be parametrised over the index list as well, which in turn is usually obtained from a given matrix.

```
kleeneClosureMatrixWith :: KleeneAlgebra κ ⇒ (Mat κ → [Int]) → Mat κ → Mat κ
kleeneClosureMatrixWith order a = τ a (order a)
```

Conveniently, this modification does not concern τ itself, but only its application.

On a further modification note, lists may not be the best suited data structure for storing tests. Instead, a tree-like (or heap-like) structure might be used, which is then passed to the Kleene closure function. The tree itself can then either be flattened into a list or traversed directly. In both cases the actual order is encoded in the structure of the tree rather than written in a list. We can use a type class (similar to a Java iterator)

```
class Structure σ where
  isEmpty :: σ α → Bool
  next :: σ α → (α, σ α)
```

such that *isEmpty s* is true if and only if there are no elements in the structure and *next* splits a non-empty structure into an element and the remaining structure. We then redefine τ as

```
τ :: (KleeneAlgebra κ, Structure σ) ⇒ κ → σ κ → κ
τ s a | isEmpty s = a
      | otherwise  = newMat k (τ a s′)
     where (k, s′) = next s
```

thus replacing the pattern matching by an explicit case distinction whether the structure is non-empty and a call to *next* in the positive case. Thus the function *next*, which is supplied by the type class instance implicitly encodes the traversal order of the structure.

[15] In fact, both examples are computationally similar to the Kleene closure [Leh77; AAM03].

3.6 Alternative Implementations

In this section we present and discuss different implementations of the Kleene closure function. The driving question behind this presentation is whether the approach is worth the effort, which is to say, whether the previously presented, slightly technical approach really provides an advantage as compared to a more straightforward one. To that end we consider three alternative implementations in Haskell. We avoid comparisons with implementations in different languages for the sake of direct comparability of the implementations.

3.6.1 Arrays

As we have stated before in Section 3.2, typical implementations in imperative languages make use of arrays to implement a Kleene closure algorithm in terms of Equation (3.2.5.i). Haskell also provides several types of arrays (most importantly mutable and immutable ones) which can be used for an implementation as well, because they provide fast access to their indices. For the sake of simplicity, we use the most basic array type, which allows a simple and pure (i.e. free of side effects) code. Careful observation of Equation (3.2.5.i) shows that we do not need to modify existing arrays (which is very costly for pure arrays), but rather only build new ones. Due to this observation there is (probably) no particular gain in using mutable arrays.

A second observation concerning Equation (3.2.5.i) is that we do not need to keep $n+1$ arrays in memory, but only two at any given time. This observation is due to the fact that the array $a^{(k+1)}$ is constructed from $a^{(k)}$ alone, making previous indices obsolete, so that they can be garbage-collected. The fact that older arrays can be discarded is somewhat similar to in-place updates, but no actual modification takes place and thus the function is still referentially transparent.

We use a straightforward implementation of matrices in terms of arrays by indexing the arrays with pairs of indices, such that for all $i, j \in \mathbb{N}_{<n}$ the matrix value $a_{i,j}$ is represented by $arr\,!\,(i,j)$, where $(!)$ denotes the built-in array query function.

$$\mathbf{newtype}\ \mathit{ArrayMat}\ \alpha = \mathit{AM}\ \{\mathit{unAM} :: \mathit{Array}\ (\mathit{Int}, \mathit{Int})\ \alpha\}$$

For the actual closure function we repeatedly construct a new array from an old one. The local function *newValue* is a copy of Equation (3.2.5.i) and the function *newArray* creates a complete array using the implementation of said equation.

```
kleeneClosureArray :: KleeneAlgebra κ ⇒ ArrayMat κ → ArrayMat κ
kleeneClosureArray a0 = AM (foldl newArray (unAM a0) [0 . . n]) where
  newArray arr k     = listArray bnds (map (newValue arr k) (range bnds))
  newValue a k (i, j) = ((a ! (i, k)) ⊛ star (a ! (k, k)) ⊛ (a ! (k, j))) ⊕ a ! (i, j)
  bnds               = bounds a
  n                  = snd (snd bnds)
```

Here the *foldl* essentially acts as a `for`-loop.

3.6.2 Lists Revisited

Another observation concerning Equation (3.2.5.i) is that it contains the index j at the same relative position. This allows the following rephrasing:

$$\forall\, i,k \in \mathbb{N}_{<n} : a_i^{(k+1)} = a_i^{(k)} + \left(a_{i,k}^{(k)} \cdot \left(a_{k,k}^{(k)}\right)^{\star}\right) \bullet a_k^{(k)} \;.$$

Just as we have done before, the arguments of $+_{\mathsf{v}}$ can be swapped due to the commutativity of $+_{\mathsf{v}}$. This equation is strikingly similar to our recursion for τ. However, the computational direction is different: while the recursion for τ delegates the *remainder* of the computation into a recursive application of τ, the above equation shows how to compute the *next* step.

We can use the specification of the above equation for a third implementation, where we reuse the function *newMat* from Section 3.5.

```
kleeneClosureLeft :: KleeneAlgebra κ ⇒ Mat κ → Mat κ
kleeneClosureLeft a = foldl (flip newMat) a (rowIndices a)
```

Note however that the indices of the matrix are used in exactly the reversed order as in the function *kleeneClosure*.

3.6.3 Blockwise Computation

In the more recent work, Dolan [Dol13] uses (a flipped version of) Equation (3.2.5.ii) for an implementation of the *star* closure of a matrix. The complete implementation is provided in the above article and we used this code with only very minor variations[16]. Matrices are represented by the following data type

```
data Matrix α = Matrix [[α]] | Scalar α
```

and are either a list of the complete rows (including possible zeroes in the matrix) or a scalar matrix, which is represented by the scalar alone. Rows contain the values at all positions in their order of appearance and the scalar matrices do not have a fixed dimension. In this implementation there are no explicit indices, which makes it difficult to remove unnecessary zeroes (otherwise a list of a length strictly smaller than n can represent a variety of rows and there is no way of distinguishing these possibilities). Even if indices were introduced, one would either have to split off the last element of the list instead of the first one as in the implementation by Dolan or reindex the corresponding matrices. Both options come with additional complexity, which increases the implicit constants in the Landau estimate for the complexity of the star closure in this implementation. Additional zeroes on the other hand require more space.

The Kleene closure function for matrices can be realised as follows

[16]Our modifications mostly deal with a slightly different type class for Kleene algebras and an additional type class, which simplifies testing. These modifications have very little effect on the run time behaviour let alone on the complexity of the code.

$$\mathit{kleeneClosureBlockwise} :: \mathit{KleeneAlgebra}\ \kappa \Rightarrow \mathit{Matrix}\ \kappa \to \mathit{Matrix}\ \kappa$$
$$\mathit{kleeneClosureBlockwise}\ a = a \circledast \mathit{closure}\ a$$

where *closure* is the function from Dolan [Dol13], which computes a^* and $(\circledast)$ is the matrix multiplication function provided by the Kleene algebra instance for matrices given in the same source.

3.7 Complexity and Comparison

In this section we present the results of an empiric comparison of the methods from the previous sections. All closure functions we have presented so far have cubic complexities in the dimension of the matrix, which we discuss in a moment, but there is no clear indication as to how large the implicit constants in the Landau notation are. We analyse the running times of the different functions by generating random matrices and measuring the results. Conveniently, Haskell's random data mechanism allows the creation of the same random data, which leads to precise results, because different functions can run on the very same data set.

3.7.1 Kleene Closure Complexity

Under the assumption that we wish to fully evaluate the result of the Kleene closure using our function *kleeneClosure*, we get a cubic complexity estimate in the dimension of the matrix. Suppose that said dimension is $n \in \mathbb{N}$. Since

$$\mathit{kleeneClosure} \equiv \tau \circledast \mathit{rowIndices} \equiv \lambda a \to \mathit{foldr}\ \mathit{newMat}\ a\ (\mathit{rowIndices}\ a)\ ,$$

we know that the Kleene closure computation of an $n \times n$-matrix a requires n calls of the function *newMat*, since *rowIndices a* has n elements[17]. The function *newMat k a* is an application of *map*, which additionally uses the query function (!!!). The length of the list the *fmap* function is applied to is n and for every element in this list, there are two calls to the function (!) which are linear in n, one call to $(\bullet_{\mathsf{v}})$, which is also linear in n and, finally, one call to $(+_{\mathsf{v}})$, which is linear in the size of both its arguments, but since both arguments are lists of length at most n, this application is also linear in n. Thus the function *newMat* is quadratic in n, which gives an overall cubic complexity. We have not factored in the complexity of the semiring operations $(\oplus)$, $(\circledast)$ and *star* because they do not depend on n and are thus constant time functions (in n). However, they may still be significant costs associated with these operations depending on the concrete semiring.

The complexity of *kleeneClosureLeft* is also cubic in n, because only the folding direction is different, but all employed functions are the same as in the function *kleeneClosure*. Our last implementation *kleeneClosureArray* is canonically cubic in n, because it constructs n arrays with each array containing size n^2 elements to represent

[17] We have $\mathit{rowIndices} = \mathit{map}\ \mathit{fst} \circ \mathit{unVec} \circ \mathit{matrix}$, where *unVec* and *matrix* are constant time operations.

an $n \times n$-matrix. Finally, the complexity of *kleeneClosureBlockwise* is computed by Dolan [Dol13] to be cubic in the dimension of the matrix.

3.7.2 The Setting of Random Tests

Since Haskell is a pure language, the creation of random data is free of side effects. In GHC 7.6.3 it is realised by creating pseudo-random data from a seed. Data types that allow random data are instances of the type class *Random*, which, in particular, comes with the function

$$random :: RandomGen\ \gamma \Rightarrow \gamma \to (\alpha, \gamma)$$

that has the following semantics: given a random generator g (cf. the type class *RandomGen* from the package `random` for more detail) it creates a random value of the type α and a new generator. The key idea is that the application *random gen*, where *gen* is some random generator, *always yields the same result*. This concept makes random experiments easily repeatable. The pseudo-random generation is based upon the method of L'Ecuyer [LEc88]. In our implementation we use only those random generators, which can be created from a single *Int* value with the function *mkStdGen* :: *Int* → *StdGen*, where *StdGen* is an instance of *RandomGen* provided by *System.Random*.

We implemented a random matrix generation function *randomMatrix* based upon shuffling. Given a density $d \in [0, 1]$, which describes the percentage of non-zero elements in the matrix, and a size n we calculate the number of non-zero positions in the matrix as $p = \lfloor d \cdot n^2 \rfloor$. We then create p random elements and fill the remaining $n^2 - p$ positions in the matrix with zeroes. At this point a matrix with dimension n is represented by a list with n^2 elements. Then, this list is shuffled using the shuffle function *shuffle′* from the `random-shuffle` package by Kiselyov [Kis12]. This technique is known to be uniformly distributed[18]. Finally, we remove the zero positions and regroup the matrix according to our specification from Section 3.5.

For the actual tests we have generated three random generators from the random number generator generated from the number 42. We then fixed a function we wanted to test (*testFunction*), a matrix size (*size*), a density[19] (*dens*) and computed

$$testFunction\ (randomMatrix\ gen\ size\ dens)\ ,$$

[18] The default implementation in the package `random` in version 1.0.1.1 (which we used) is known to have a strong correlation at the first value of subsequent random generators [CP13; Ste15]. However, neither do we use subsequent random generators, nor merely their first values, but use a stream of random values generated by a single random generator. Since our interest is in some empirical data and not necessarily truly independent results, the correlated implementation is sufficient for our purpose.

[19] The highest density we use is 0.1, which corresponds to 10% of existing entries. This value may seem small, but in a square matrix with dimension 1000 we already get $100,000$ entries. Aside from the fact that such matrices require lots of space, they are also very likely to have transitive closures which are filled at every position. Depending on the underlying semiring this computation may then in fact be simpler (e.g. in the Boolean semiring) than in the general case.

for each of the three generators (*gen*) mentioned above[20], thus obtaining three random matrices per each pair of a density and a size. In the following we always use arithmetic mean of these three measurements. Note that in all cases the actual creation of the random matrix is performed anew, because the sharing mechanism of the GHC would otherwise compute the random matrix once and then share this value with any subsequent uses. Also, the creation of the random matrix counts towards the total time and space usage to simulate pre-processed input rather than monolithic computations. Finally, we evaluate the result of all calls completely using the built-in function *deepseq* from *Control.DeepSeq*, but do not inspect this result any further.

3.7.3 Kleene Closure Measurements

The data in Table 3.1a, Table 3.1b, Table 3.2a, and Table 3.2b show the average (avg) and the maximum (max) space consumptions in megabytes and the running time (sec) in seconds. The letters are abbreviations for *a*(rray), *b*(lock), *l*(eft) and *r*(ight) and refer to the respective closure function. The computation that is denoted with *b* uses the function *kleeneClosureBlockwise* from Section 3.6.3, while b^* computes the blockwise star closure only. It is a known fact[21] that matrix multiplication is as complex as the star closure computation. Thus there is no asymptotic difference between the two different closure computations. The value "$-$" denotes an out-of-memory exception that occurs using standard settings. All measurements were obtained using RTS options after a compilation with the optimisation flag $-O2$.

The values involving the tropical Kleene algebra are different from the ones presented in Danilenko [Dan14b], because we used *Tropical Int* in said work, which is technically not a Kleene algebra (adding large *Int* values may result in a negative number, which violates the distributivity law in said structure). In this work we have used *Tropical Integer* instead, which is a Kleene algebra. Since *Integer*s are unbounded, they can require more space and the operations are not as fast as those for *Int* values.

We observe that the respective right-fold variant of the Kleene closure is almost always faster and never uses more space than all other versions. Additionally, the row-based functions (*l* and *r*) are always able to deal with the largest input and the right-fold function is typically faster and requires less space than the left-fold function. Another observation is that the block version behaves very similar to the array version in terms of time and space consumption, being worse than the array variant for larger or denser graphs. This is not surprising, because the array variant simply computes several arrays, while the block variant contains several recursive calls and matrix multiplications, which have a more complex structure. In all cases

[20] Our actual random matrix function takes an additional argument, which controls the range of the random data.

[21] We refer to the textbook of Pettorossi [Pet13] for a proof.

Table 3.1

(a) Evaluation in the tropical Kleene algebra.

$d \setminus n$		100					250					500					750					1000				
		a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r
0.001	avg	5	5	5	1	1	22	43	43	1	1	129	–	–	4	4	–	–	–	8	8	–	–	–	14	13
	max	19	17	16	1	1	181	242	251	2	2	1460	–	–	9	6	–	–	–	15	13	–	–	–	34	23
	sec	0.2	0.3	0.2	0.1	0.1	4.2	7.4	5.6	0.1	0.1	43.4	–	–	0.7	0.7	–	–	–	1.8	1.9	–	–	–	3.7	3.5
0.01	avg	5	5	5	1	1	22	29	38	6	5	139	–	–	10	8	–	–	–	2	2	–	–	–	3	2
	max	18	17	16	1	1	177	219	201	12	13	1576	–	–	38	34	–	–	–	70	53	–	–	–	130	107
	sec	0.2	0.3	0.3	0.1	0.1	6.8	7.1	8.7	2.9	2.4	81.6	–	–	57.8	55.9	–	–	–	247.0	216.0	–	–	–	642.1	577.3
0.025	avg	5	5	5	1	1	22	43	44	5	4	139	–	–	3	2	–	–	–	3	2	–	–	–	1	1
	max	19	16	17	2	2	184	251	247	8	6	1579	–	–	34	29	–	–	–	83	71	–	–	–	152	129
	sec	0.2	0.3	0.4	0.1	0.1	9.4	8.3	9.2	7.9	6.6	92.1	–	–	83.2	77.5	–	–	–	318.3	285.1	–	–	–	705.0	643.5
0.05	avg	5	5	5	1	1	22	36	45	4	4	–	–	–	4	2	–	–	–	3	3	–	–	–	3	1
	max	20	16	16	2	1	181	232	265	9	8	–	–	–	38	33	–	–	–	86	76	–	–	–	156	138
	sec	0.3	0.3	0.4	0.3	0.2	10.2	9.7	11.1	10.5	9.1	–	–	–	87.1	79.1	–	–	–	321.2	287.8	–	–	–	695.4	631.2
0.1	avg	4	5	5	1	1	29	45	46	5	4	–	–	–	4	2	–	–	–	3	4	–	–	–	2	4
	max	16	17	17	2	1	226	266	275	9	9	–	–	–	39	35	–	–	–	88	80	–	–	–	157	144
	sec	0.4	0.4	0.5	0.4	0.3	9.4	8.8	10.0	9.5	8.7	–	–	–	85.8	77.4	–	–	–	287.9	263.8	–	–	–	692.9	645.6

(b) Evaluation in the Boolean Kleene algebra.

$d \setminus n$		100					250					500					750					1000				
		a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r
0.001	avg	5	5	5	1	1	22	44	43	1	1	129	–	–	4	4	–	–	–	8	8	–	–	–	14	13
	max	19	17	15	1	1	181	242	250	2	2	1461	–	–	10	6	–	–	–	15	13	–	–	–	34	23
	sec	0.2	0.3	0.2	0.1	0.1	4.0	7.1	5.4	0.1	0.1	40.5	–	–	0.6	0.6	–	–	–	1.7	1.7	–	–	–	3.3	3.3
0.01	avg	5	5	5	1	1	22	31	38	8	8	135	–	–	95	90	–	–	–	3	5	–	–	–	–	2
	max	18	17	15	1	1	177	220	201	40	44	1528	–	–	536	524	–	–	–	948	235	–	–	–	–	776
	sec	0.2	0.3	0.3	0.1	0.1	3.3	5.8	6.6	0.8	0.9	20.8	–	–	17.4	16.3	–	–	–	67.9	60.1	–	–	–	–	163.7
0.025	avg	5	5	5	1	1	23	45	44	8	7	139	–	–	2	1	–	–	–	–	6	–	–	–	–	–
	max	18	16	16	3	3	185	243	248	37	26	1574	–	–	336	110	–	–	–	–	429	–	–	–	–	–
	sec	0.1	0.3	0.3	0.1	0.1	2.1	5.8	6.5	2.3	1.8	16.3	–	–	23.5	21.7	–	–	–	–	76.9	–	–	–	–	–
0.05	avg	5	5	5	2	1	22	47	43	4	3	154	–	–	3	2	–	–	–	–	4	–	–	–	–	–
	max	20	16	16	4	2	181	236	253	44	15	1668	–	–	379	145	–	–	–	–	520	–	–	–	–	–
	sec	0.1	0.2	0.3	0.1	0.1	1.9	5.7	5.8	2.8	2.4	16.0	–	–	25.7	22.8	–	–	–	–	85.5	–	–	–	–	–
0.1	avg	3	5	5	1	1	26	50	45	4	3	–	–	–	4	2	–	–	–	–	1	–	–	–	–	–
	max	11	16	16	4	2	211	257	267	49	21	–	–	–	394	169	–	–	–	–	572	–	–	–	–	–
	sec	0.1	0.2	0.2	0.1	0.1	1.9	6.3	6.0	3.1	2.7	–	–	–	27.0	24.2	–	–	–	–	84.4	–	–	–	–	–

Table 3.2

(a) Evaluation in the product Kleene algebra "tropical × Boolean".

$d \setminus n$		100					250					500					750					1000				
		a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r
0.001	avg	14	13	13	1	1	125	126	130	1	1	–	–	–	3	4	–	–	–	7	7	–	–	–	12	13
	max	65	60	60	1	1	1107	940	981	2	2	–	–	–	6	6	–	–	–	13	13	–	–	–	23	23
	sec	0.9	1.1	1.0	0.1	0.1	20.9	20.6	20.5	0.1	0.1	–	–	–	0.7	0.7	–	–	–	1.8	1.8	–	–	–	3.4	3.6
0.01	avg	13	13	14	1	1	96	138	133	8	4	–	–	–	35	27	–	–	–	–	–	–	–	–	–	–
	max	60	65	70	1	1	649	889	941	21	14	–	–	–	675	428	–	–	–	–	–	–	–	–	–	–
	sec	0.9	1.1	1.2	0.1	0.1	19.6	23.2	24.5	3.2	2.5	–	–	–	84.2	77.4	–	–	–	–	–	–	–	–	–	–
0.025	avg	12	12	13	1	1	125	131	134	21	16	–	–	–	–	19	–	–	–	–	–	–	–	–	–	–
	max	48	60	64	2	2	680	987	1018	120	70	–	–	–	–	754	–	–	–	–	–	–	–	–	–	–
	sec	0.9	1.1	1.2	0.1	0.1	24.7	22.8	25.9	11.6	10.2	–	–	–	–	116.8	–	–	–	–	–	–	–	–	–	–
0.05	avg	13	12	13	3	2	132	173	172	17	10	–	–	–	–	7	–	–	–	–	–	–	–	–	–	–
	max	40	60	60	7	4	758	1222	1141	190	101	–	–	–	–	1025	–	–	–	–	–	–	–	–	–	–
	sec	1.1	1.1	1.3	0.5	0.4	25.9	27.1	29.8	14.6	13.2	–	–	–	–	120.2	–	–	–	–	–	–	–	–	–	–
0.1	avg	16	15	16	3	2	130	137	139	13	9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	max	47	65	75	13	7	805	1044	1063	225	149	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	sec	1.2	1.4	1.5	0.7	0.6	25.1	23.7	26.3	15.0	13.7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

(b) Evaluation in the *Balance* Kleene algebra, as discussed later in Section 8.3.

$d \setminus n$		100					250					500					750					1000				
		a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r	a	b^*	b	l	r
0.001	avg	5	5	5	1	1	22	43	43	1	1	129	–	–	4	4	–	–	–	8	8	–	–	–	14	13
	max	19	17	15	1	1	181	242	250	2	2	1461	–	–	9	6	–	–	–	15	13	–	–	–	34	23
	sec	0.2	0.3	0.2	0.1	0.1	4.1	7.2	5.5	0.1	0.1	43.7	–	–	0.7	0.7	–	–	–	2.0	2.1	–	–	–	3.9	3.8
0.01	avg	5	5	5	1	1	22	32	38	4	4	135	–	–	14	12	–	–	–	2	2	–	–	–	1	1
	max	19	17	16	1	1	177	220	202	9	8	1528	–	–	21	19	–	–	–	38	31	–	–	–	75	63
	sec	0.2	0.3	0.3	0.1	0.1	5.5	6.2	6.9	0.9	0.8	66.3	–	–	24.6	22.7	–	–	–	112.2	99.1	–	–	–	307.8	283.9
0.025	avg	5	5	5	1	1	23	42	38	3	2	139	–	–	2	1	–	–	–	3	1	–	–	–	2	1
	max	18	16	16	1	1	185	231	229	5	4	1574	–	–	20	18	–	–	–	49	43	–	–	–	91	80
	sec	0.2	0.3	0.3	0.1	0.1	6.3	6.5	7.1	2.9	2.5	70.4	–	–	38.1	34.3	–	–	–	140.3	144.7	–	–	–	362.5	323.4
0.05	avg	5	5	5	1	1	22	42	43	3	2	154	–	–	3	2	–	–	–	5	2	–	–	–	1	3
	max	20	16	16	1	1	181	241	254	6	5	1668	–	–	23	20	–	–	–	53	48	–	–	–	96	87
	sec	0.2	0.3	0.3	0.1	0.1	6.6	6.4	7.2	3.9	3.5	73.7	–	–	42.3	39.3	–	–	–	160.1	147.8	–	–	–	367.6	336.8
0.1	avg	3	5	5	1	1	26	44	45	3	3	–	–	–	4	3	–	–	–	1	3	–	–	–	2	4
	max	11	16	16	1	1	211	257	267	6	6	–	–	–	25	22	–	–	–	56	50	–	–	–	99	90
	sec	0.2	0.3	0.3	0.2	0.1	6.9	6.6	7.3	4.4	4.1	–	–	–	42.9	40.0	–	–	–	175.3	152.3	–	–	–	385.0	345.2

the row-based functions are the only ones that provide results for graphs with more than 500 vertices.

Occasionally, (tail-recursive) functions in a non-strict setting can be improved using *strictness annotations* which force (partial) evaluation of an argument before it is used. We have experimented with this technique in case of the above implementations and have found that it yields little to no improvement in case of the left-fold variant, but noticeable (yet constant) improvements in the right-fold version. This outcome is particularly interesting, because usually strict left-folds provide a better alternative than strict right-folds. Since strictness annotations are not purely algebraic means, we did not explore the implications any further. Still, these empiric results indicate that a row-based approach, particularly one which can be transformed into a right-fold, is a good implementation choice, which also justifies the slightly larger theoretical overhead.

3.8 Related Work and Discussion

Aside from the two implementations in terms of arrays and in terms of the block matrices we have given above, there are many other approaches to a functional version of Kleene's algorithm to compute either the Kleene closure or the star closure over a given Kleene algebra. O'Connor [OCo11] provides another array-based implementation that uses Equation (3.2.5.i). We have not included a comparison with this version because it is structurally very similar to our array version. The special case of transitive closures of concrete relations (which are Kleene closures over the Boolean Kleene algebra) is treated by Johnsson [Joh98] using a monadic abstraction of lazy arrays, which are implemented efficiently externally. The work of Berghammer [Ber11], which we have used as a foundation, presents another implementation of the transitive closure of relations. This version is significantly faster and less space consuming than ours, because it uses only lists of (bounded) integers and not association lists with arbitrary values. However, it is restricted to Boolean matrices only and our implementation can be considered a direct generalisation to arbitrary Kleene algebras. Similarly, the *fgl* library [EM14] provides a function that computes the transitive closure of a graph, while ignoring the edge labels, thus computing the Kleene closure over the Boolean Kleene algebra only.

In the case of the works of Dolan [Dol13], O'Connor [OCo11] and Johnsson [Joh98] there is no derivation of the correctness of the implementation and all three depend rather rigidly on the chosen data types. While we also made an implementation choice, this was only to provide a complete implementation. It is rather simple to generalise our implementation to a collection of rows from a list of rows and possibly to generalise association lists to more efficient data types like *Data.IntMap*. Essentially, every representation of matrices that supports the notion of rows and the possibility to add rows and multiply them by a scalar is suited for our implementation. In

particular, it is easily possible to provide row access functions to all non-row-based implementations from before, but the reverse direction is less straightforward.

While we have shown a strongly list-based implementation, the actual application is written in terms of provided interface functions rather than a hands-on use of the employed data types. These interface functions (and constants, like $emptyVec :: Vec\ \alpha$) are easy to replace. *Data.IntMap* provides many convenient functions (which are often available under the same names in other variants of finite maps), which can be used to implement all of our utility functions. We might assume that we choose

$$\textbf{newtype}\ Vec\ \alpha = Vec\ \{toIntMap :: IntMap\ \alpha\}$$
$$unVec :: Vec\ \alpha \to [Arc\ \alpha]$$
$$unVec = toList$$
$$mkVec :: [Arc\ \alpha] \to Vec\ \alpha$$
$$mkVec = fromList$$

as a vector representation, where the functions *toList* and *fromList* are functions that are already provided by the *IntMap* data type. With this new data type we can define the addition by using the *unionWith* function that is also provided by the *IntMap* data type to obtain the following implementation.

$$(+_{\mathsf{v}}) :: Vec\ \alpha \to Vec\ \alpha \to Vec\ \alpha$$
$$v +_{\mathsf{v}} w = Vec\ (unionWith\ (\oplus)\ (toIntMap\ v)\ (toIntMap\ w))$$

Since *IntMap*s already provide a *Functor* instance, we can reuse our implementation of $(\bullet_{\mathsf{v}})$ by simply replacing *filterVec* with *Data.IntMap.filter* and *emptyVec* with *Data.IntMap.empty*. Note that both replacements simply provide a more efficient implementation: the original ones that use *unVec* and *mkVec* still yield the correct result, but use intermediate structures that are not necessary. The next logical abstraction step is to define type classes with certain functions to deal with the exchange of implementations in a generic fashion. We discuss this step in more detail in Section 4.6.

Finally, we have shown that the choice of the less general non-trivial Kleene algebras with tests is not a proper restriction, particularly because the Kleene algebra of square matrices of a fixed size is a Kleene algebra with tests without any further requirements. Thus our approach can be used to compute the Kleene closure of a matrix over every Kleene algebra and thus also in every Kleene algebra, too, as we have seen in Lemma 3.4.1. While we have dealt with the case of finite matrices only, the right-fold variant is also suited to provide partial results in the infinite case as well, as long as sufficient information about $x + y$ can be obtained from x alone. This possibility constitutes a proper improvement over all other versions from Section 3.6, because all other versions are based on left-folds and left-folds always diverge on infinite lists.

Chapter 4

Graph Algebra and its Generalisation

In this chapter we consider a computation scheme, which is derived from the notion of a vector-matrix multiplication and is applicable in a variety of functions, including many path-based ones. Although the original notion of a vector-matrix multiplication is purely algebraic, we consider a generalisation of the multiplication pattern to achieve a more flexible framework, which, incidentally, is also simpler to implement. Large portions of this chapter have been published before at the TFP 2014 Symposium [Dan14c].

4.1 Graphs, Vertices and their Algebraic Model

Graph algorithms essentially deal with graphs, vertices or sets of vertices and paths between vertices. Clearly, there are typically more additional components involved (e.g. Boolean terms, weight functions or tuples of vertices), but they usually either are side conditions on the solution of a certain problem or can be composed from more basic components as the above mentioned paths or sets of vertices. For simplicity of presentation, we deal only with sets of vertices (a single vertex v is represented by the singleton set $\{ v \}$) and paths between sets of vertices (paths between two vertices s, t can be computed as the paths between the vertex sets $\{ s \}$ and $\{ t \}$).

There are two common ways to represent sets of vertices in algebraic terms — using either vectors or partial identities. Both terms can be described in a purely relational setting without resorting to linear-algebraic notation. A similar result holds in the general case, too, which we discuss in a moment.

4.1.1 Definition (Sets as vectors)**.**
Let $G = (V, E)$ be a finite graph, $n := |V|$ and $(S, +, \cdot, 0, 1)$ be a semiring. Since $n = |V|$, there is a bijective mapping $v : \mathbb{N}_{<n} \to V$. We define

$$\mathsf{asVector} : 2^V \to S^n , \quad A \mapsto \left(i \mapsto \begin{cases} 1 & : v(i) \in A \\ 0 & : \text{otherwise.} \end{cases} \right)$$

Due to the property

$$2^V \cong \{ 0, 1 \}^V \subseteq S^V \cong S^n .$$

the function $\mathsf{asVector}$ is essentially merely an inclusion embedding. Similarly, we define

$$\mathsf{asSet} : S^n \to 2^V , \quad s \mapsto \{ v(i) \mid i \in \mathbb{N}_{<n} \land s_i \neq 0 \} . \qquad /\!\!/$$

Simply put, we have
$$\mathsf{asVector}(A) = \chi_{v^{-1}(A)} ,$$
where $\chi_{v^{-1}(A)}$ denotes the characteristic function of $v^{-1}(A)$ in $\mathbb{N}_{<n}$. For example, consider a graph G with $|V| = 5$ and the subset $A := \{ v_0, v_2, v_3 \}$. Then we have $\mathsf{asVector}(A) = (1, 0, 1, 1, 0)$, which is to say that exactly those positions are filled with ones in the result, which correspond to the indices of the vertices in the set. Obviously, the mapping asVector also depends on S and more importantly on the actual bijection $v : \mathbb{N}_{<n} \to V$. However, we omit these implicit operands mainly for the following reasons.

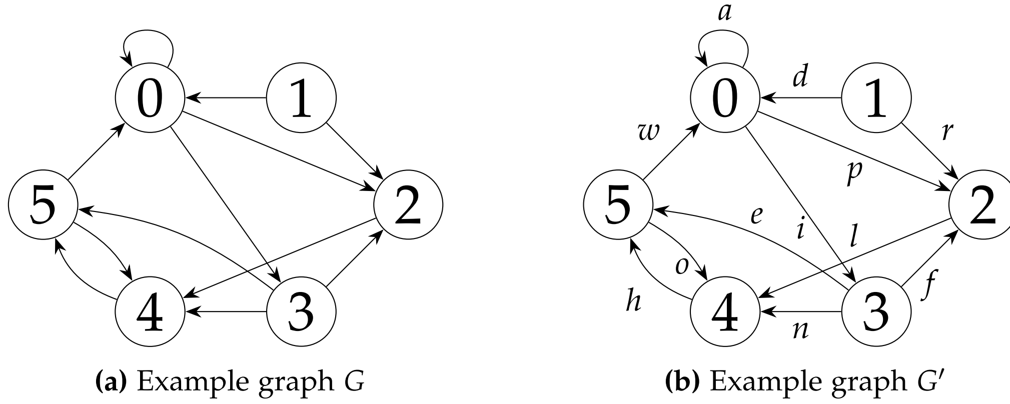
(1) Every semiring contains an element 0 and an element 1. Thus the representations provided by asVector are essentially the same, but differ only in terms of their actual meaning. Whenever there is a risk of ambiguity, we will index these constants like 0_S or $1_{S'}$.

(2) The enumeration order of V is assumed to be fixed. In particular, we never change the order mid-way. Whenever $V = \mathbb{N}_{<n}$ for a given $n \in \mathbb{N}$, we implicitly assume the enumeration to be $v = \mathrm{id}_{\mathbb{N}_{<n}}$.

Vectors are a convenient algebraic tool and many well-known properties of linear-algebraic vectors carry over to vectors over semirings, too, as we have already summarised in part in Proposition 2.4.5 and Proposition 2.4.6. However, vectors are an *extrinsic* representation: they are not elements of the matrix semiring (unless $n \leqslant 1$). The representation in terms of partial identities on the other hand is *intrinsic* and more general, because it is not limited to the matrix semiring.

As it turns out, the set $\{ 0, 1 \}^n$ is isomorphic to $\mathsf{PI}(\{ 0, 1 \}^{n \times n})$, where $0, 1$ are the constants of a given idempotent semiring and PI refers to the partial identities of a semiring as defined in Definition 2.4.7. In order to verify that the structures are isomorphic, we define the functions
$$\varphi_1 : \mathcal{P}(\mathbb{N}_{<n}) \to \{ 0, 1 \}^n , \quad S \mapsto \sum_{i \in S} \mathsf{e}^n(i)$$
as well as
$$\varphi_2 : \mathcal{P}(\mathbb{N}_{<n}) \to \mathsf{PI}\left(\{ 0, 1 \}^{n \times n}\right) , \quad S \mapsto \sum_{i \in S} \mathsf{e}^{n,n}(i, i) ,$$
where we use the standard unit vectors and matrices from Definition 2.4.4. It is simple to verify that both functions are semiring isomorphisms[1] and thus the function
$$\varphi_2 \circ \varphi_1^{-1} : \{ 0, 1 \}^n \to \mathsf{PI}\left(\{ 0, 1 \}^{n \times n}\right)$$
is an isomorphism.

[1] To be more precise, they are isomorphisms from $(\mathcal{P}(\mathbb{N}_{<n}), \cup, \cap, \varnothing, \mathbb{N}_{<n})$ to the respective structure with pointwise addition and multiplication. The constants in the image sets are the images of the constants of $\mathcal{P}(\mathbb{N}_{<n})$ under φ_1 and φ_2 respectively.



(a) Example graph G

(b) Example graph G'

Figure 4.1. Structurally identical graphs.

The representation of sets as vectors has interesting applications in the context of graph theory. Consider for example the graph G from Figure 4.1(a) and in that graph the set of vertices $X := \{ 0, 2, 3 \}$. One typical task in graph theory is then to compute the set of successors of X, which is precisely the set $\{ 0, 2, 3, 4, 5 \}$. A more sophisticated task is to compute the successors and label each successor with the number of times it is reached from X, which is the set

$$\{ (0,1), (2,2), (3,1), (4,2), (5,1) \} ,$$

where each first component denotes the successor and the respective second component denotes the number of times it is reached.

It is a known fact[2] that both problems can be solved in the same algebraic fashion: first, the graph and the set are translated into the adjacency matrix A_G (cf. Definition 2.3.2) and the vector $\mathsf{asVector}(X)$ over some semiring and second, the multiplication $\mathsf{asVector}(X) \cdot A_G$ is performed. The above values are

$$A_G = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \mathsf{asVector}(X) = (1,0,1,1,0,0) .$$

The matrix multiplication in this term depends on the chosen semiring. In the case of the Boolean semiring, we get

$$\mathsf{asVector}(X) \cdot A_G = (1,0,1,1,1,1)$$

and $\mathsf{asSet}((1,0,1,1,1,1)) = \{ 0, 2, 3, 4, 5 \}$, which is exactly the set of successors of X. We can perform the very same multiplication over the semiring of natural numbers

[2]The particularly important case of unlabelled successors is described in detail by Schmidt and Ströhlein [SS93] and by Berghammer [Ber08].

$(\mathbb{N}, +, \cdot, 0, 1)$ to obtain the vector

$$\mathsf{asVector}(X) \cdot A_G = (1, 0, 2, 1, 2, 1) \; .$$

The value at each index shows how often the vertex with said index has been reached. In particular, we also get $\mathsf{asSet}(\mathsf{asVector}(X) \cdot A_G) = \{\, 0, 2, 3, 4, 5 \,\}$ as above. There are many more problems that can be solved as seen above using a fitting semiring and thus exchanging the meaning of $+$, $\cdot$, 0 and 1. In particular, since products and coproducts of semirings are again semirings, one can combine several seemingly different computations into one using a corresponding semiring construction. We will see a natural example of such a combination in Section 7.2.

In many cases the edges of a graph carry some additional information (e.g. capacities or just names). It is important to note that this information may not be of interest for some particular simple problem, but only for a more complex one. For instance, the graph G' from Figure 4.1(b) is structurally identical to G, but every edge also has a label. Since these labels are characters, a sensible interpretation is to consider them to be single letter words in the semiring of regular expressions $(A^*, |, \cdot, _, \varepsilon)$, where A is the latin alphabet, $|$ is the alternative of two regular expressions, $\cdot$ is their concatenation, "_" denotes no word and ε is the empty word. In this interpretation the matrix A_G becomes (cf. Convention 2.3.9)

$$A_G = \begin{pmatrix} a & _ & p & i & _ & _ \\ d & _ & r & _ & _ & _ \\ _ & _ & _ & _ & l & _ \\ _ & _ & f & _ & n & e \\ _ & _ & _ & _ & _ & h \\ w & _ & _ & _ & o & _ \end{pmatrix}$$

and the computation $\mathsf{asVector}(X) \cdot A_G$ now yields

$$\mathsf{asVector}(X) \cdot A_G = (a, _, f|p, i, l|n, e) \; .$$

While we can reconstruct the set of successors from this vector again using asSet, it seems unnecessary to perform the semiring additions and multiplications along the way, especially because the equality of two regular expressions is difficult to compute[3], which makes the decision $v_i \neq 0$ in this semiring possibly non-trivial. There are several ways to remedy this difficulty; for example, we might transform the above adjacency matrix to its counterpart in the Boolean semiring or come up with a simple check for constants in the regular expression semiring. Still, both solutions shroud the simple fact that the edge labels are irrelevant in this example. Instead of finding a suitable semiring or mapping the edge labels to a better suited semiring, we take another approach and consider the actual computation that is necessary for the solution of the given problem and to parametrise it in an algebraically flavoured fashion. We discuss this approach shortly.

[3]Two regular expressions are called equal, if and only if the languages they describe are equal.

The remainder of this chapter is structured as follows.

- ▻ Section 4.2 deals with vector-matrix multiplications, the necessary abstraction and the implementation of this abstraction. We briefly recall our implementation from Section 3.5.
- ▻ We provide several applications of the framework of Section 4.2 in Section 4.3. The correctness of the applications is discussed informally, a more formal approach is given in Section 4.7.
- ▻ In Section 4.4 we provide an implementation of a simple reachability function and provide a non-trivial application thereof in Section 4.5.
- ▻ We elaborate on a set oriented abstraction of the employed scheme in terms of type classes in Section 4.6.

At the time of this writing, our approach was novel. Our work slightly overlaps with the work of Dolan [Dol13], but both our goals and abstractions are different, which we discuss at corresponding places. A polished version of all the code in this chapter, as well as the implementation of all functions which are discussed only, is available at `https://www.github.com/nikitaDanilenko/vmm`. This chapter is the main foundation of our *gwaf* library [Dan15].

4.2 Rearranged Multiplication and its Implementation

In this section we study the multiplication of a vector with a matrix over some given algebraic structure and generalise its underlying scheme to the non-algebraic case. In our notation the multiplication of a vector with a matrix is a special case of the general matrix multiplication, however, matrix multiplication is also a special case of several vector-matrix multiplications as we discuss later.

Let $(S, +, \cdot)$ be an algebraic structure of the type $(2, 2)$, where $+$ is associative and $n, m \in \mathbb{N}$. For the moment we do not impose any further rules on this structure, because no rules are necessary for the plain definition of the multiplication of a vector with a matrix. Typically, this multiplication is defined by a pointwise description of the result vector as follows

$$\odot : S^n \times S^{n \times m} \to S^m \ , \quad (v, A) \mapsto \left(i \mapsto \sum_{k=0}^{n-1} v_k \cdot A_{k,i} \right) \ .$$

This definition can be simplified by using the rows of a matrix as follows:

$$(v, A) \mapsto \sum_{k=0}^{n-1} v_k \bullet A_k \ ,$$

where $\bullet$ is the scalar multiplication of a vector (cf. Section 3.4, considering vectors as matrices) and the sum is the pointwise addition of vectors. To see that the above

definitions in fact describe the same function, we first note that the pointwise addition of vectors $w_1, w_2 \in S^m$ is defined as

$$w_1 + w_2 = i \mapsto (w_1)_i + (w_2)_i \ .$$

Using this definition, a straightforward induction shows that for all $l \in \mathbb{N}$ and $ws \in (S^m)^l$ we have

$$\sum_{j=0}^{l-1} ws_j = i \mapsto \sum_{j=0}^{l-1} (ws_j)_i \ ,$$

where the left sum denotes the generalisation of the binary vector addition and the right one is the generalisation of the addition in S. With this rule we get

$$\begin{aligned}
& \sum_{k=0}^{n-1} v_k \bullet A_k \\
= \ & \wr \text{ above rule with } ws := j \mapsto (v_j \bullet A_j) \ \wr \\
& i \mapsto \sum_{k=0}^{n-1} (v_k \bullet A_k)_i \\
= \ & \wr \text{ pointwise definition of } \bullet \ \wr \\
& i \mapsto \sum_{k=0}^{n-1} v_k \cdot A_{k,i} \ ,
\end{aligned}$$

which proves that both definitions yield the same result. Note that at first glance, the computational scheme in both computations appears to be the same. However, there is a subtle difference, in particular when it comes to an implementation. Let us demonstrate both computations in an example. Consider the following vector and matrix over $(\mathbb{N}, +, \cdot)$:

$$v := (2, 0, 1) \text{ and } A := \begin{pmatrix} 0 & 1 & 1 \\ 2 & 3 & 5 \\ 0 & 8 & 0 \end{pmatrix} \ .$$

The original computation then looks as follows:

$$\begin{aligned}
v \odot A &= (2 \cdot 0 + 0 \cdot 2 + 1 \cdot 0, 2 \cdot 1 + 0 \cdot 3 + 1 \cdot 8, 2 \cdot 1 + 0 \cdot 5 + 1 \cdot 0) \\
&= (0 + 0 + 0, 2 + 0 + 8, 2 + 0 + 0) \\
&= (0, 10, 2) \ .
\end{aligned}$$

The rearranged multiplication yields the following steps:

$$\begin{aligned}
v \odot A &= 2 \bullet (0, 1, 1) + 0 \bullet (2, 3, 5) + 1 \bullet (0, 8, 0) \\
&= (0, 2, 2) + (0, 0, 0) + (0, 8, 0) \\
&= (0, 10, 2) \ .
\end{aligned}$$

We observe the following differences:

(1) The original version needs to query exact matrix positions, while the rearranged

variant only requires access to the rows of the matrix and some algebraic functions on rows.

(2) The rearranged computation is more declarative than the original one, because the result is described as the sum of vectors rather than a vector with specific components. That is to say, we have more of a function composition rather than a list comprehension.

(3) The sum-of-vectors approach in the rearranged version can produce arbitrary values, while the original version always yields a vector of a fixed length.

(4) The latter computation can make use of algebraic rules, as we have already seen in Section 3.5. In case of a semiring we have $1 \bullet_{\mathsf{v}} \mathit{vec} = \mathit{vec}$ and $0 \bullet_{\mathsf{v}} \mathit{vec} = \mathit{emptyVec}$, where both are constant time operations, while the original version actually performs the multiplications with these constants in every component.

The rearrangement of the vector-matrix multiplication is essentially based upon a "vector sum" and a "scalar multiplication". The quotation marks indicate that while the underlying abstractions are those of these two operations, they can be abstracted to a more general definition.

4.2.1 Abstracted Sum and Sum Generation

The (finite) vector sum takes a list of vectors as input and produces a vector. However, in the general case, the result may be any value of an arbitrary type ρ. Similarly, the input of the sum is not necessarily a list of vectors, but a list of values of a fixed type. Thus the most general case of a sum has the type $[\iota] \to \rho$.

In many applications the sum is a folded version of a binary addition. The addition of vectors depends on an associative operation, but not on a neutral element of this operation. This is because the neutral element of the addition is (typically) the value zero and our prerequisite on vectors from Section 3.5 states that vectors do not contain zero values. Since vectors in our implementation are (a special case of) integer-to-something maps, one usually refers to the addition as union. We can employ a very similar merging technique as we have done for $(+_{\mathsf{v}})$ in Section 3.5 to define a function *unionWith* that applies a supplied function in case of index equality and adds the arguments to the result vector if the indices differ[4]. The actual sum function is then simply an application of a right-fold.

$\mathit{unionWith} :: (\alpha \to \alpha \to \alpha) \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ \alpha$

$\mathit{bigUnionWith} :: (\alpha \to a \to \alpha) \to [\mathit{Vec}\ \alpha] \to \mathit{Vec}\ \alpha$
$\mathit{bigUnionWith}\ \mathit{op} = \mathit{foldr}\ (\mathit{unionWith}\ \mathit{op})\ \mathit{emptyVec}$

Below we list some simple example applications.

[4]The general merging technique is provided in Section 4.6.

```
ghci> unionWith const (Vec [(0,'c'),(2,'a'),(3,'r')]) (mkVec [(1,'h'),(2,'i')])
Vec [(0,'c'),(1,'h'),(2,'a'),(3,'r')]
ghci> unionWith (+) (Vec [(0,2),(2,1),(3,7)]) (mkVec [(1,3),(2,4)])
Vec [(0,2),(1,3),(2,5),(3,7)]
```

The merge-based approach has the simple property that *unionWith op* is associative if and only if *op* is associative.

Merging operations have complexities that are linear in the sum of the size of both its arguments[5]. With this in mind, let us assume that we have a list of vectors $vs :: [Vec\ \alpha]$ and $n \in \mathbb{N}$ such that the indices of all vectors in *vs* are contained in $\mathbb{N}_{<n}$. Then assuming that $op :: \alpha \to \alpha \to a$ is a constant time operation, it takes $\Theta(n \cdot length\ vs)$ operations to fully evaluate *bigUnionWith op vs*. Depending on the length of *vs* and the actual application, it may be more efficient to first write all values in an intermediate container, for instance a *Data.IntMap*, and then transform the result back into a list. The *IntMap* data structure is a special type of tree and allows queries and updates, whose complexity is logarithmic in the key size. For example, we might use the following definition, where $empty :: IntMap\ \alpha$ is the empty *IntMap* and

```
insertWith :: (α → α → α) → Int → α → IntMap α → IntMap α
```

is a function that inserts a value at a specified *Int* position. If the position is not yet filled, the value is the supplied value, otherwise the value is the supplied operation applied to the supplied value and the value that is already present in the map. Both, *empty* and *insertWith*, are provided by the *IntMap* implementation in the Haskell module *Data.IntMap*. Finally, the function $intMapToVec :: IntMap\ \alpha \to Vec\ \alpha$ is supposed to transform an *IntMap* into a vector according to our preconditions[6].

```
bigUnionWith' :: (α → α → α) → [Vec α] → Vec α
bigUnionWith' op =
  intMapToVec ∘ foldr (uncurry (insertWith op)) empty ∘ concatMap unVec
```

This implementation requires $sum\ (map\ (length \circ unVec)\ vs)$ insertions into the initially empty map and every insertion has a logarithmic complexity in n. Thus the overall complexity is $\mathcal{O}(sum\ (map\ (length \circ unVec)\ vs) \cdot \log_2(n))$. There are cases in which each of these two versions is more efficient than the other: for two vectors of length n we have at most $2n$ operations with *bigUnionWith*, but $2n \cdot \log_2(n)$ with *bigUnionWith'*. If on the other hand *vs* is a list of n pairwise disjoint singleton vectors

$$vs = [Vec\ [(n-i, i)] \mid i \leftarrow [0\,..\,n-1]]\ ,$$

[5] To be more precise, one needs $length\ xs + length\ ys - length\ (xs \cap ys)$ iteration steps, *xs* and *ys* are the ordered index lists of both components and $(\cap)$ denotes list intersection. In case of index equality one needs to take possible costs of the supplied addition operation into account as well.

[6] One possible implementation is $intMapToVec = Vec \circ toAscList$, where *Data.IntMap.toAscList* maps an *IntMap* to an association list that is sorted in ascending order of its indices.

then *bigUnionWith* requires a number of operations that is quadratic in n, while *bigUnionWith′* takes only $n \cdot \log_2(n)$ operations.

Instead of an immutable intermediate structure, one might also take an approach with mutable arrays, which allow fast read and write access to array positions in a special type of monad[7] at the cost of an array initialisation. This approach removes the logarithmic factor in the estimate for *bigUnionWith′*, but only if the vectors in *vs* provide enough positions so that the array initialisation, which is linear in the size of the array, does not create too much overhead. In the singleton example we obtain a complexity of $2n$ operations, which is (almost certainly) better than $n \cdot \log_2(n)$ operations, but in the example of two vectors of length n, we get $3n$ operations, which is worse than the merge-based version.

We keep these basic three possibilities in mind, but use *bigUnionWith* for now for the sake of simplicity. The differences in the implementations are discussed in more detail in Section 4.6.

4.2.2 Scalar Multiplication and Construction

As we have already seen before in Section 3.5 the mathematical type of scalar multiplication corresponds to the Haskell type $\sigma \to Vec\ \sigma \to Vec\ \sigma$. In our rearrangement we do not require the result of a scalar multiplication to be a row, which is why we generalise the type to $Int \to \sigma \to Vec\ \tau \to \iota$. The $\sigma \to Vec\ \tau \to \iota$ part is straightforward and the additional *Int* argument denotes the act of a vertex on its adjacency list, which is explicit in abstract notation, but is usually lost in concrete applications. To be more precise, the term $v_k \bullet A_k$ contains a reference to the vertex k, but once v_k and A_k are substituted with actual values (cf. our rearrangement example from the beginning of this section) there is no such reference left.

A heterogeneous version of scalar multiplication in the mathematical sense is a special case of the above abstraction, where $\iota = Vec\ \omega$ for some type ω. In many applications such a version is of particular interest, and we provide a generator function for this mathematical case[8].

$$\begin{aligned}&sMultWith :: (Int \to \sigma \to \tau \to \iota) \to Int \to \sigma \to Vec\ \tau \to Vec\ \iota\\&sMultWith\ mult\ i = fmap \circ mult\ i\end{aligned}$$

Note that this definition does not depend on vectors, but merely on a data type that has a *Functor* instance. The above definition of *sMultWith* is a particularly simple example of a case where switching the representation of vectors is possible with very little effort.

[7] For example, the *STArray* from the module *Data.Array.ST* performs all operations in the state transformer monad *ST* from the module *Control.Monad.ST*

[8] The η-expanded definition reads $sMultWith\ mult\ i\ x\ v = fmap\ (\lambda y \to mult\ i\ x\ y)\ v$.

4.2.3 Generalised Vector-Matrix Multiplication

Recall that the rearranged vector-matrix multiplication applies a sum of the type $[\iota] \to \rho$ to an intermediate list of values that are created by applying a scalar multiplication of the type $\mathit{Int} \to \sigma \to \mathit{Vec}\ \tau \to \iota$ to the values in the vector (type σ) and the rows of the matrix (type $\mathit{Vec}\ \tau$). Since a matrix is essentially a special type of vector, we could simply query both the vector and the matrix at the same position and combine the results. In a functional setting this is not always a practical choice due to the possibly high costs associated with query operations. Our implementation has linear query operations, which yield a quadratic complexity for creating the intermediate list. We can do better by replacing the association lists with a map structure to obtain a complexity of $\mathcal{O}(n \cdot \log_2(n))$, where n is the length of the vector (and the number of rows in the matrix). An even more natural choice is to provide a "zipping" operation on vectors (or rather association lists) that applies a given operation in case of index equality and discards the values otherwise. From a set-theoretical point of view such a function resembles an intersection, which justifies the name of the following function, where the suffix L hints at the result being a list.

$$\mathit{intersectionWithKeyL} :: (\mathit{Int} \to \alpha \to \beta \to \gamma) \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ \beta \to [\gamma]$$

We provide an implementation of *intersectionWithKeyL* in Section 4.6. Haskell data structures for the representation of maps typically come equipped with a more homogeneous function providing the above functionality, but yielding a map rather than a list. This is obviously only a minor difference and such a function is easily implemented in terms of the above one as follows.

$$\begin{aligned}&\mathit{intersectionWithKey} :: (\mathit{Int} \to \alpha \to \beta \to \gamma) \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ \beta \to \mathit{Vec}\ \gamma\\&\mathit{intersectionWithKey}\ \mathit{op}\ v\ w = \mathit{Vec}\ (\mathit{intersectionWithKeyL}\ (\lambda i\ x\ y \to (i, \mathit{op}\ i\ x\ y))\ v\ w)\end{aligned}$$

We can now implement the abstracted version of the multiplication of a vector with a matrix as follows.

$$\begin{aligned}&\mathit{vecMatMult} :: ([\iota] \to \rho) \to (\mathit{Int} \to \sigma \to \mathit{Vec}\ \tau \to \iota) \to \mathit{Vec}\ \sigma \to \mathit{Mat}\ \tau \to \rho\\&\mathit{veMatMult}\ \mathit{sum}\ \mathit{sMult}\ v\ m = \mathit{sum}\ (\mathit{intersectionWithKeyL}\ \mathit{sMult}\ v\ (\mathit{matrix}\ m))\end{aligned}$$

This function is the key component in most applications in the remainder of this text. Although the function is surprisingly simple in its structure, it already reveals a key insight into the nature of (abstract) vector-matrix multiplication. The term inside the sum produces a list of intermediate values which are then consumed by the supplied *sum* function, possibly discarding values along the way. We may thus consider vector-matrix multiplication to be an instance of a generate-then-prune paradigm[9]. While it may seem that there is very little pruning involved in a practical application, the amount of actual discarding depends on the concrete problem. We will shortly see examples of cases where values can be discarded quickly, but also different examples in which all values matter for the final result.

[9] Another common name of this paradigm is "branch-and-bound".

It is important to note that the above implementation does not depend on any properties of the employed types $\sigma, \tau, \iota, \rho$. This heterogeneous type choice is a major difference to the semiring-based approach, because our scalar multiplication and sum functions are not necessarily derived from an underlying algebraic structure, but can be arbitrary computations. One particular consequence of this general approach is that both functions are heterogeneous in the sense that the result type may be different from any part of the input type, which is not possible in case of semiring operations. For example, semiring multiplication takes two semiring values and produces yet another semiring value. Thus a scalar multiplication of a vector over a semiring always yields a vector over the same semiring. In many applications this kind of homogeneity is far less natural than the more heterogeneous version $\sigma \to \mathit{Vec}\ \tau \to \mathit{Vec}\ \sigma$, which resembles a structural act[10]. Clearly, this generalisation is still a special case of our setting.

On a final note, our approach has the benefit of being somewhat independent of the vector structure. In particular, we do not explicitly access matrix or vector positions and thus a parametric abstraction of the data types is rather simple.

4.2.4 Requirements on Vector and Matrix Representations

In Section 3.5 we have already mentioned some conditions about *Vec* and *Mat* values with respect to the vectors and matrices they represent. Now, in the more general case of non-semiring labels, we need to adjust these requirements accordingly. To that end, we recall Definition 2.3.2 and Convention 2.3.9.

The connection between the Haskell values and the actual vectors and matrices that are represented by these values can be summarised as follows. Let α be a type that is interpreted by the set A and Z be an element such that $\mathsf{Z} \notin A$.

(1) For every $n \in \mathbb{N}$ a vector $v \in (A \cup \{\,\mathsf{Z}\,\})^n$ is represented by a value $\mathit{vec} :: \mathit{Vec}\ \alpha$ if and only if we have

$$\forall\, i \in \mathbb{Z} : \mathit{withAt}\ \mathsf{Z}\ \mathit{vec}\ i = \begin{cases} v_i & : i \in \mathbb{N}_{<n} \wedge v_i \neq \mathsf{Z} \\ \mathsf{Z} & : \text{otherwise}\ . \end{cases}$$

This is to say that a vector is represented by some kind of associative structure that is filled in exactly the non-Z positions. The function *withAt* has been defined in Section 3.5 and returns a vector position if it is present in the vector and the supplied failure value otherwise.

(2) For every $n, m \in \mathbb{N}$ a matrix $m \in (A \cup \{\,\mathsf{Z}\,\})^{n \times m}$ is represented by the value

[10]Structural acts are functions of the type $\sigma \to \alpha \to \alpha$ satisfying certain properties depending on the algebraic structure of σ. These functions constitute a well-known algebraic generalisation of structural operations of the type $\sigma \to \sigma \to \sigma$. In particular, structural operations are special cases of structural acts. We refer to the elegant textbook of Aluffi [Alu09] for more detail.

$mat :: Mat\ \alpha$ if and only if the following holds:

$$\forall\, i \in \mathbb{Z} : mat\ !!!\ i = \begin{cases} \mathsf{representation}(m_i) & : i \in \mathbb{N}_{<n} \\ emptyVec & : \text{otherwise} \end{cases},$$

where $\mathsf{representation}(v)$ is the Haskell representation of the vector v and the function (!!!) has been defined in Section 3.5 and yields the corresponding row of a matrix. Additionally, the following should hold

$$size\ (matrix\ mat) = n\ ,$$

where $size :: Vec\ \alpha \rightarrow Int$ denotes the number of filled positions in a vector. This latter part is important for the reconstruction of the number of matrix rows from the matrix.

These conditions essentially depend only on an indexing function *withAt*, because (!!!) is defined in terms of this function and the size requirement. We did not mention the previously required condition that the elements of the association list of a $Vec\ \alpha$ element are sorted in ascending order of their indices. However, this requirement is stated for convenience only, because it allows us to provide a complete implementation. Instead of association lists one can employ other structures that provide a similar functionality, for example *Data.IntMap*. When exchanging the structure, we now only have to deal with a proper implementation of *withAt* and the requirements above remain valid, because they do not depend on an actual structure, but are rather rules that need to hold in the general case of the existence of a *withAt* function.

There is one caveat concerning the above definition and that is a multiplication in a semiring that has zero divisors. By Convention 2.3.9 we have that in a semiring we always assume $\mathsf{Z} = 0$ and that all values that occur in a vector are non-zero. If a semiring has zero-divisors, one cannot simply use our function *sMultWith* from Section 4.2.2, because simply mapping the multiplication function over a vector may introduce zeroes. Instead, one needs to apply an additional filter operation to remove newly computed zeroes. We consider this problem in Section 4.3.1.

4.3 Applications

In this section we apply the abstraction we have developed in the previous section to cover many examples of an algebraic approach to graph problems. Many of these examples can be expressed in terms of semirings, in particular those of Section 4.3.4. However, the semiring approach is rather technical and requires a tedious verification of the semiring laws, as well as a proper implementation. Also, the semirings required for similar problems as in Section 4.3.4 are typically still quite different, while our abstraction allows us to observe similarities directly, because the functions used in the generalised vector-matrix multiplication are strikingly similar.

For now we provide only informal arguments for the correctness of our implementation. We discuss a proof scheme in Section 4.7 and phrase our arguments in a fashion that hints at a rephrasing in more formal terms.

We use the following conversion functions for simplicity. The function *mkVec* takes a list of arcs and transforms it into a vector by sorting the arcs with respect to their keys and keeping only the first occurrence of a key[11]. Based upon this function we define the function *toVecFrom* which takes a function f from vertices to values and a list of vertices. It applies the function $v \to (v, f\ v)$ to every vertex in the list and then turns the resulting list into a vector using *mkVec*. A special case of this function is the function *toVecWith*, which takes a single value a instead of a function and labels every vertex in the list with the value a. Finally, the function *toVec* is a particularly simple case, that maps a list of vertices to a vector, where every vertex is labelled with ().

```
type Vertex = Int
mkVec :: [Arc α] → Vec α
mkVec = Vec ∘ map head ∘ groupBy ((≡) ‘on‘ fst) ∘ sortBy (comparing fst)
toVecFrom :: (Vertex → α) → [Vertex] → Vec α
toVecFrom f = mkVec ∘ map (id &&& f)
toVecWith :: α → [Vertex] → Vec α
toVecWith = toVecFrom ∘ const
toVec :: [Vertex] → Vec ()
toVec = toVecWith ()
```

Similarly, we can transform a given vector into a list of vertices.

```
fromVec :: Vec α → [Vertex]
fromVec = map fst ∘ unVec
```

Both functions depend on an association list structure of a vector. Since association lists are the most basic form of an integer-to-something mapping, most implementations of these mappings provide conversion functions from and to association lists. Thus the specialised functions *unVec* and *Vec* can be replaced by more general versions called *toAscList* and *fromAscList*, respectively.

4.3.1 Number-like multiplication

One particularly simple instance of the vector-matrix multiplication scheme is the usual multiplication of a vector with a matrix over a semiring. We have already defined a type class for semirings in Section 3.5, which we can use. A straightforward implementation is the following one.

[11] On sorted lists the call *map head* ∘ *group* is more efficient than a *nub*, which is a built-in function for the removal of duplicates. All functions in this definition are defined in the standard Haskell modules and can be found using Hoogle [Mit13].

$(\odot) :: \mathit{Semiring}\ \sigma \Rightarrow \mathit{Vec}\ \sigma \rightarrow \mathit{Mat}\ \sigma \rightarrow \mathit{Vec}\ \sigma$
$(\odot) = \mathit{vecMatMult}\ (\mathit{bigUnionWith}\ (\oplus))\ (\mathit{sMultWith}\ (\lambda_\ x\ y \rightarrow x \circledast y))$

Recall that in general semirings the result of both the multiplication and the addition can be zero, without any of the operands being zero[12]. We can remedy this problem by reusing our function that removes zeroes from a vector and optimising the scalar multiplication as we have done before in Section 3.5.

$(\odot') :: \mathit{Semiring}\ \sigma \Rightarrow \mathit{Vec}\ \sigma \rightarrow \mathit{Mat}\ \sigma \rightarrow \mathit{Vec}\ \sigma$
$(\odot') = \mathit{vecMatMult}\ (\mathit{removeZeroes} \circ \mathit{bigUnionWith}\ (\oplus))\ (\bullet_{\mathsf{v}})$

In particular, we can apply this definition to any kind of numbers.

```
newtype Number n = Number {unNumber :: n}
  deriving (Eq, Num, Show)
instance Show n ⇒ Show (Number n) where
  show = show ∘ unNumber
instance (Eq n, Num n) ⇒ Semiring (Number n) where
  (⊕)    = (+)
  (⊛)    = (·)
  zero   = 0
  one    = 1
  isZero = (0 ≡)
  isOne  = (1 ≡)
```

For the remainder of this chapter we assume that the graphs G and G' from Figure 4.1(a) and Figure 4.1(b) are represented by the constants $g :: \mathit{Mat}\ (\mathit{Number\ Int})$ and $g' :: \mathit{Mat\ Char}$ respectively, where all edge labels in g are *Number* 1. We then get the following results using the GHCi interpreter.

```
ghci> toVecWith 1 [2,3,0] ⊙ g
Vec [(0,1),(2,2),(3,1),(4,2),(5,1)]
ghci> mkVec [(5,1),(2,−1)] ⊙ g
Vec [(0,1),(4,0)]
ghci> mkVec [(5,1),(2,−1)] ⊙′ g
Vec [(0,1)]
```

All three examples are usual multiplications of a vector with a matrix. The first application is exactly the example from Section 4.1, where we multiply the vector $(1,0,1,1,0,0)$ with the matrix A_G using the usual numeric operations for addition and multiplication. In the second example the vector $(0,0,-1,0,0,1)$ is multiplied with the matrix A_G, which introduces a zero at the fourth position. Using $(\odot')$ with the same example removes this zero.

[12] This is different from the situation in *idempotent* semirings, which we have discussed in Section 3.5.

4.3.2 Discrete Successors

In every graph, especially in graphs with non-trivial labels, one may need to compute the set of (discrete) successors of a set of vertices. This task is easily accomplished with our tools. The resulting function takes a set of vertices, which is represented by a list, and a graph. After the list has been transformed to a vector, a special multiplication is performed and the vector information is discarded.

$$(\otimes_{\rightarrow}) :: [\mathit{Vertex}] \rightarrow \mathit{Mat}\ \alpha \rightarrow [\mathit{Vertex}]$$
$$v \otimes_{\rightarrow} m = \mathit{fromVec}\ (\mathit{toVec}\ v \otimes m)$$

The multiplication $(\otimes)$ is a heterogeneous version of the Boolean vector-matrix multiplication. The sum takes the first occurrence of an index. The scalar multiplication takes a vertex, its value, and its adjacency list and returns the (labelled) successors of the vertex. The latter is a constant-time operation, because the adjacency list is exactly the list of labelled successors. We implement this multiplication as follows.

$$\mathit{leftmostUnion} :: [\mathit{Vec}\ \alpha] \rightarrow \mathit{Vec}\ \alpha$$
$$\mathit{leftmostUnion} = \mathit{bigUnionWith}\ \mathit{const}$$
$$(\otimes) :: \mathit{Vec}\ \sigma \rightarrow \mathit{Mat}\ \alpha \rightarrow \mathit{Vec}\ \alpha$$
$$(\otimes) = \mathit{vecMatMult}\ \mathit{leftmostUnion}\ (\lambda_\ _\ \mathit{row} \rightarrow \mathit{row})$$

The multiplication ignores the information in the supplied vector and only uses its indices for its result. Since the values in the graph do not matter, the resulting function works regardless of the graph labels. Below are some example applications of the resulting $(\otimes_{\rightarrow})$ function.

ghci> $[0,3,2] \otimes_{\rightarrow} g$
$[0,2,3,4,5]$
ghci> $[0,3,2] \otimes_{\rightarrow} g'$
$[0,2,3,4,5]$

The multiplication $(\otimes)$ is an instance of the generate-then-prune paradigm, because many possible arcs leading to a successor are collapsed into one. We benefit from Haskell's non-strictness, because while the sum expands into a stack of *foldr*-applications, the operation performed on the same index is *x* '*const*' _, which discards any further results once any result has been computed.

It is possible to obtain a similar result using a semiring where non-zero values are closed under addition (for instance an idempotent semiring). We can then add *one* instead of () to every vertex and use the previously defined multiplication $(\odot')$.

$$(\otimes^{S}) :: \mathit{Semiring}\ \sigma \Rightarrow [\mathit{Vertex}] \rightarrow \mathit{Mat}\ \sigma \rightarrow [\mathit{Vertex}]$$
$$v \otimes^{S} m = \mathit{fromVec}\ (\mathit{toVecWith}\ \mathit{one}\ v \odot' m)$$

There are two caveats in this approach. First, the condition that non-zero values are closed under addition may not be neglected, because otherwise arcs leading to the same vertex may cancel each other out, leaving no arc leading to the vertex. Since

not all semirings satisfy this condition, this approach constitutes a proper restriction regarding the general definition of $(\otimes_{\rightarrow})$. Second, the discarding effect of *const* requires that the addition $(\oplus)$ in the semiring satisfies the property

$$one \oplus _ = one \; .$$

This is the case in the Boolean semiring and the Haskell implementation reflects this as well. However, this is not necessarily true in general.

Finally, we note that *const* is associative. To make *const* the addition in a semiring, we require a value *zero* that satisfies $zero \text{ `} const \text{` } x = x$ due to the semiring axioms. Unfortunately, this element breaks the discarding feature of *const*, because now we no longer have $x \text{ `} const \text{` } _ = x$. Clearly, the above is already true for a *Monoid* instance, where the monoid operation is basically the function *const*. Haskell provides a *Monoid* instance for the data type *First* in *Data.Monoid*. The monoid operation inspects its first argument and if it is the neutral element, the second argument is returned, otherwise it returns the first argument. Since the first argument is not inspected in case of *const*, it is less strict than the monoid operation for *First*. In our approach we do not need to take the above problems into account. Since we do not use a semiring instance, neither the multiplication, nor *zero*, nor *one* are used, which is particularly convenient. Thus relaxing the semiring laws leads to a simplified implementation, which in particular focusses on exactly the necessary components.

4.3.3 Check for Successors

A somewhat trivial instance of the generalised vector-matrix multiplication scheme is the check, whether a given set (or list) of vertices has any successors. Obviously, this is equivalent to the question, whether at least one of the adjacency lists of the vertices in the given list is non-empty. We can transform the existential quantification into an iterated disjunction, which corresponds to the function $or :: [Bool] \rightarrow Bool$ in Haskell. The scalar multiplication then simply needs to check, whether the supplied row is non-empty. The following implementation is based upon precisely this test.

$$hasSuccsMult :: Vertex \rightarrow \alpha \rightarrow Vec\ \beta \rightarrow Bool$$
$$hasSuccsMult\ _\ _ = not \circ isEmptyVec$$

$$(\odot_?) :: Vec\ \alpha \rightarrow Mat\ \beta \rightarrow Bool$$
$$(\odot_?) = vecMatMult\ or\ hasSuccsMult$$

$$hasSuccs :: [Vertex] \rightarrow Mat\ \alpha \rightarrow Bool$$
$$hasSuccs\ vs\ g = toVec\ vs \odot_? g$$

Clearly, we could solve the problem in a variety of different other ways as well. In particular, we could compute the set of discrete successors using the function $(\otimes_{\rightarrow})$ from Section 4.3.2 and then check whether this set is empty or not. One important benefit of the above approach is that we can avoid the implicit use of the function *bigUnionWith*, which in turn avoids unnecessary computations.

4.3.4 Extending Walks by One Step

One particularly important application of vector-matrix multiplication is the computation of walks. The multiplication of a vector with a matrix always computes (in some sense) the successors of all vertices in the vector (i.e. indices that are filled with information). This step can be repeated several times to obtain walks leading from an initial set to a target set. In order to achieve that, one usually labels the successors with some information that can be interpreted after every single multiplication.

We now deal with two special instances of the general description above. The first one can be used to reach vertices, while at the same time finding a single walk that leads to these vertices from the starting set. The basic idea is to simply assume that in any given vector every vertex is already labelled with a walk that leads to this vertex. What we want is to compute the successors of the vertices in the vector and to label them with a walk leading there.

The specification hints at the fact that a particular choice is not important, so that we can reuse our function *leftmostUnion* to choose the left-most occurrence of an index. We use the data type *Seq* for walks from the module *Data.Sequence* which allows to efficiently add elements to the end of the structure using the function $(\triangleright) :: Seq\ \alpha \to \alpha \to Seq\ \alpha$. This data type is based upon so-called finger trees, which were introduced by Hinze and Paterson [HP06]. The general multiplication pattern is similar to the ones we used before.

$$\begin{aligned}&\textbf{type}\ Walk = Seq\ Vertex\\&(\odot_{\sim}) :: Vec\ Walk \to Mat\ \alpha \to Vec\ Walk\\&(\odot_{\sim}) = vecMatMult\ leftmostUnion\ walkMult\end{aligned}$$

The function *walkMult* now takes a vertex $v :: Vertex$ and a walk $w :: Walk$ such that w is a walk that leads to v and the adjacency list of v and returns a vector, where each successor is labelled with a walk leading to said successor. Since w is a walk that leads to v, the walk $w \triangleright v$ is a walk that leads to every successor of v. We use this in our implementation.

$$\begin{aligned}&walkMult :: Vertex \to Walk \to Vec\ \alpha \to Vec\ Walk\\&walkMult = sMultWith\ (\lambda v\ walk\ _ \to walk \triangleright v)\end{aligned}$$

Using the multiplication $(\odot_{\sim})$ and the vector $v = toVecWith\ \langle\rangle\ [0,3,2] :: Vec\ Walk$, we obtain the following examples. We use the $\langle\ldots\rangle$-notation to denote walks and assume that the *Show* instance for vectors lists all filled positions in the form $(key \mid value)$.

ghci> $v \odot_{\sim} g$
$(0 \mid \langle 0\rangle)\ (2 \mid \langle 0\rangle)\ (3 \mid \langle 0\rangle)\ (4 \mid \langle 2\rangle)\ (5 \mid \langle 3\rangle)$
ghci> $v \odot_{\sim} g'$
$(0 \mid \langle 0\rangle)\ (2 \mid \langle 0\rangle)\ (3 \mid \langle 0\rangle)\ (4 \mid \langle 2\rangle)\ (5 \mid \langle 3\rangle)$
ghci> $v \odot_{\sim} g \odot_{\sim} g'$
$(0 \mid \langle 0,0\rangle)\ (2 \mid \langle 0,0\rangle)\ (3 \mid \langle 0,0\rangle)\ (4 \mid \langle 0,2\rangle)\ (5 \mid \langle 0,3\rangle)$

Note that the first two calls yield the same result, although the underlying graphs have different types. This polymorphism is particularly interesting for the third example, because we multiply the original vector by a graph of one type and then the result of this multiplication by a graph of another type using the very same vector-matrix multiplication $(\odot_{\sim})$ due to its heterogeneous and polymorphic type.

In a similar fashion it is also possible to have vectors carry some information about a list of walks and then to extend all these walks. It is a known fact that this problem can be solved in the so-called Kleene algebra of walks, but we provide a more hands-on implementation[13]. The problem specification is similar to the one from above, but this time we use a list of walks that all lead to the given vertex as labels in the vector. Our goal is to label the successors of all these vertices with a list of walks, such that

(1) each walk in the list leads to the vertex that is labelled with the list

(2) every walk in the list is one of the previously existing walks extended by exactly one step.

The resulting function is again an instance of the general vector-matrix multiplication.

$(\odot_{\approx}) :: \mathit{Vec}\ [\mathit{Walk}] \to \mathit{Mat}\ \alpha \to \mathit{Vec}\ [\mathit{Walk}]$
$(\odot_{\approx}) = \mathit{vecMatMult}\ \mathit{allUnion}\ \mathit{walksMult}$

We note that the set of lists of walks leading to one fixed vertex is closed under concatenation. That is to say that if $ws_1, ws_2 :: [\mathit{Walk}]$ are lists of walks that lead to a certain vertex, then so is their concatenation $ws_1 +\!\!+ ws_2$. Thus the sum function should concatenate all labels at the same index.

$\mathit{allUnion} :: [\mathit{Vec}\ [\alpha]] \to \mathit{Vec}\ [\alpha]$
$\mathit{allUnion} = \mathit{bigUnionWith}\ (+\!\!+)$

The actual extension by exactly one step works almost precisely as before: if $v :: \mathit{Vertex}$ is a vertex and $ps :: [\mathit{Walk}]$ is a list of walks that lead to v, then for every successor w of v the list of walks $\mathit{map}\ (\rhd v)\ ps$ is a list of walks that leads to w. The implementation below is simply a Haskell version of this procedure.

$\mathit{walksMult} :: \mathit{Vertex} \to [\mathit{Walk}] \to \mathit{Vec}\ [\alpha] \to \mathit{Vec}\ [\mathit{Walk}]$
$\mathit{walksMult} = \mathit{sMultWith}\ (\lambda v\ ws\ _ \to \mathit{map}\ (\rhd v)\ ws)$

Note that we can clearly see that the one-walk-extension $(\odot_{\sim})$ is very similar to the all-walk-extension $(\odot_{\approx})$. In particular, the scalar multiplication used for $(\odot_{\approx})$ is merely an extension of the one we used for $(\odot_{\sim})$, adjusted to work on lists of walks instead of a single walk. It is somewhat technical to obtain this similarity in the

[13]To use the purely algebraic multiplication, we need a semiring instance for the type $[\mathit{Walk}]$. The steps required for this instance are somewhat technical and more bulky than what is necessary in this example.

purely algebraic case, particularly because we use the vector *indices* for the new labels. While still possible, the similarity is more difficult to see.

Let us have a look at the example from before, adjusted to the case of a list of walks, where we use the vector $v = \mathit{toVecWith}\ [\langle\rangle]\ [0,2,3]$. The respective calls now yield the following results.

ghci> $v \odot_{\approx} g$
$(0 \mid [\langle 0\rangle])\ (2 \mid [\langle 0\rangle, \langle 3\rangle])\ (3 \mid [\langle 0\rangle])\ (4 \mid [\langle 2\rangle, \langle 3\rangle])\ (5 \mid [\langle 3\rangle])$
ghci> $v \odot_{\approx} g'$
$(0 \mid [\langle 0\rangle])\ (2 \mid [\langle 0\rangle, \langle 3\rangle])\ (3 \mid [\langle 0\rangle])\ (4 \mid [\langle 2\rangle, \langle 3\rangle])\ (5 \mid [\langle 3\rangle])$
ghci> $v \odot_{\approx} g \odot_{\approx} g'$
$(0 \mid [\langle 0,0\rangle, \langle 3,5\rangle])\ (2 \mid [\langle 0,0\rangle, \langle 0,3\rangle])\ (3 \mid [\langle 0,0\rangle])$
$(4 \mid [\langle 0,2\rangle, \langle 3,2\rangle, \langle 0,3\rangle, \langle 3,5\rangle])\ (5 \mid [\langle [0,3]\rangle, \langle [2,4]\rangle, \langle [3,4]\rangle])$

Just as with $(\odot_{\sim})$ the multiplication is applicable to graphs with different labels and can be used in sequence with differently labelled graphs.

4.3.5 Outgoing Values and Transposition

So far most of our multiplications have not used the values that are actually stored in the matrix. Such an approach is often useful when one needs to solve a discrete problem in a richer context, like finding a discrete path in a labelled graph, for example. Let us now consider a problem, where the matrix information is necessary.

Given a vector, where the values are lists of arcs $[\mathit{Arc}\ \alpha]$, we wish to compute its successor vector such that every vertex in the result vector is labelled with a new list of arcs that satisfies the following properties for every vertex $v :: \mathit{Vertex}$ in the original vector and every successor w in the result vector:

(1) If v is labelled with $\mathit{arcs} :: [\mathit{Arc}\ \alpha]$, then the label of w contains the list *arcs*.

(2) The label of w contains the arc (v, lab), where *lab* is the label of the arc that leads to w in the vector.

Intuitively, we want to collect all outgoing values from a given starting point. The resulting multiplication is, once again, an instance of the general scheme. We note that the labels we are interested in are closed under concatenation, which allows us to reuse our function *allUnion* from Section 4.3.4 as follows.

$(\odot_{\mathrm{out}}) :: \mathit{Vec}\ [\mathit{Arc}\ \alpha] \rightarrow \mathit{Mat}\ \alpha \rightarrow \mathit{Vec}\ [\mathit{Arc}\ \alpha]$
$(\odot_{\mathrm{out}}) = \mathit{vecMatMult}\ \mathit{allUnion}\ \mathit{outMult}$

To implement *outMult*, we simply look at the properties we wish to obtain. Using the scalar multiplication generator *sMultWith*, we need to provide a function that takes a vertex $v :: \mathit{Vertex}$, a list of arcs $\mathit{ovs} :: [\mathit{Arc}\ \alpha]$ that is used to rescale the adjacency list of v, and a value $a :: \alpha$ that denotes the value in the adjacency matrix, the arc from

v is leading to. The result type is another list of arcs that contains *ovs* and also the value (v, a), because the arc (w, a) denotes an arc that leads to w and is labelled with a. Now the implementation is simple.

$$\begin{aligned} &\mathit{outMult} :: \mathit{Vertex} \to [\mathit{Arc}\ \alpha] \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ [\mathit{Arc}\ \alpha] \\ &\mathit{outMult} = \mathit{sMultWith}\ (\lambda v\ \mathit{ovs}\ a \to (v, a) : \mathit{ovs}) \end{aligned}$$

Let us have a look at some examples of this multiplication.

```
ghci> toVecWith [ ] [0,2,3] ⊙out g
  (0 | [(0,1)])
  (2 | [(0,1),(3,1)])
  (3 | [(0,1)])
  (4 | [(2,1),(3,1)])
  (5 | [(3,1)])
```

This example is somewhat simple: the vertices that are located in the arcs of the label denote the predecessors of the target vertex, while the labels denote the edge label of the graph. Since the graph is labelled with *Number* 1 only, all edge labels are exactly the same and the result is essentially the one of the multiplication $(\odot_{\approx})$, except the concrete notation. Now let us consider a labelled graph.

```
ghci> toVecWith [ ] [0,2,3] ⊙out g'
  (0 | [(0,'a')])
  (2 | [(0,'p'),(3,'f')])
  (3 | [(0,'l')])
  (4 | [(2,'l'),(3,'n')])
  (5 | [(3,'e')])
ghci> mkVec [(0,[ ]),(2,[(7,'x')]),(3,[(7,'x'),(8,'y')])] ⊙out g'
  (0 | [(0,'a')])
  (2 | [(0,'p'),(3,'f'),(7,'x'),(8,'y')])
  (3 | [(0,'l')])
  (4 | [(2,'l'),(7,'x'),(3,'n'),(7,'x'),(8,'y')])
  (5 | [(3,'e'),(7,'x'),(8,'y')])
```

The first application shows that we collect the predecessors of the target vertices, just as above, and we also see that we additionally collect the labels leading from the predecessors to the target vertices. In case of the second application, the additional information is simply copied. In particular, we do not remove multiple occurrences of the arc (7, 'x') in the label of the target vertex 4, nor do we sort the results.

Despite the apparently unstructured behaviour of the function $(\odot_{\text{out}})$, it is useful in a very curious application, namely the transposition of square matrices. Since we can collect all predecessors and their respective values that lead to the target, we can simply take all possible vertices and compute precisely that. The result for every vertex is the list of arcs that lead to this vertex. This result is not necessarily a matrix according to our specification, because not every vertex needs to be contained in the

result vector. To adjust the result to satisfy the matrix conditions we can use a union with a sufficiently large vector that is labelled with an empty list of arcs.

$$preTranspose :: Vec\ [Arc\ \alpha] \to Vec\ [Arc\ \alpha] \to Mat\ \alpha \to Vec\ [Arc\ \alpha]$$
$$preTranspose\ vs\ cols\ mat = (vs \odot_{\text{out}} mat) \cup_{|} cols$$

$$(\cup_{|}) :: Vec\ \alpha \to Vec\ \alpha \to Vec\ \alpha$$
$$(\cup_{|}) = unionWith\ const$$

We use the function $(\cup_{|})$, which is the *left-biased union* and thus takes the left-most value in case an index occurs more than once. Clearly, we have that

$$foldr\ (\cup_{|})\ emptyVec = leftmostUnion\ .$$

Recall that a graph is essentially a vector of vectors and thus we can transform the result of *preTranspose* into a graph. The actual implementation of the transposition depends on whether the matrix we are transposing is a square matrix or not. This dependency is due to the fact that the sufficiently large vector we have mentioned above needs to compensate possibly missing positions and thus needs to have exactly as many entries as the number of columns in the matrix. If the matrix is a square matrix, it is simple to add possibly missing positions as follows.

$$transposeSquare :: Mat\ \alpha \to Mat\ \alpha$$
$$transposeSquare\ mat = Mat\ (fmap\ Vec\ (preTranspose\ vs\ vs\ mat))\ \textbf{where}$$
$$\quad vs = verticesWith\ [\,]\ mat$$

$$verticesWith :: \alpha \to Mat\ \beta \to Vec\ \alpha$$
$$verticesWith\ x = fmap\ (const\ x) \circ matrix$$

The function *verticesWith* simply collects the vertices of a graph in a vector, while labelling every vertex with the supplied value. Note that the result is in fact a vector with exactly as many entries as there are rows in the matrix, because of the size requirement that we discussed in Section 4.2.4. We obtain the transposition by applying *preTranspose* to the vertices of the graph and the graph itself and then transforming it into a matrix. Note that we use *fmap Vec*, which simply turns every list of arcs in the result of *preTranspose* into a vector. While it seems that we may break our requirement on vectors and should use *fmap mkVec* instead, this is not the case. We prove this statement later in Section 4.7, but for now let us see how this transposition works in an example. Consider the matrix

$$A := \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 3 \\ 0 & 4 & 0 \end{pmatrix}\ ,$$

which is represented as

$$a :: Mat\ Int$$
$$a = Mat\ (Vec\ [(0, Vec\ [(1,1),(2,2)]),$$
$$\qquad\qquad (1, Vec\ [(2,3)]),$$

$$(2, \mathit{Vec}\ [(1,4)])])$$

in our implementation. Set $vs = \mathit{verticesWith}\ [\,]\ a = \mathit{Vec}\ [(0,[\,]),(1,[\,]),(2,[\,])]$. The main computation is the following one, where we avoid pretty-printing for clarity of presentation:

$$\begin{aligned}
& vs \odot_{\mathrm{out}} a \\
={}& \mathit{bigUnionWith}\ (+\!\!+)\ (\mathit{intersectionWithKeyL}\ \mathit{outMult}\ vs\ a) \\
={}& \mathit{bigUnionWith}\ (+\!\!+) \begin{bmatrix} \mathit{Vec}\big[(1,[(0,1)]),(2,[(0,2)])\big], \\ \mathit{Vec}\big[(2,[(1,3)])\big], \\ \mathit{Vec}\big[(1,[(2,4)])\big] \end{bmatrix} \\
={}& \mathit{Vec}\ \big[(1,[(0,1)]),(2,[(0,2)])\big] \cup_{(+\!\!+)} \Big(\mathit{Vec}\ \big[(2,[(1,3)])\big] \cup_{(+\!\!+)} \mathit{Vec}\ \big[(1,[(2,4)])\big]\Big) \\
={}& \mathit{Vec}\ [(1,[(0,1),(2,4)]),(2,[(0,2),(1,3)])]\ .
\end{aligned}$$

Observe that this is almost the transposed matrix A, but its first row, which is to say the adjacency list of 0, is missing. The union $(\cup_{\mathsf{l}})$ is applied to the above result and vs and we get

$$\begin{aligned}
(vs \odot_{\mathrm{out}} a) \cup_{\mathsf{l}} vs = \mathit{Vec}\ [&(0,[\,]), \\
&(1,[(0,1),(2,4)]), \\
&(2,[(0,2),(1,3)])]\ ,
\end{aligned}$$

which is now in fact the representation of the matrix $A^{\top}$ (without the *Mat* constructor). Essentially, the scalar multiplication maps the original matrix entries to a special notation. The sum, which traverses the rows from top to bottom, adds these values by simply appending them to each other. The special notation takes care of the fact that rows are indexed properly and thus we obtain the required order without any sorting steps. Obviously, this is just an illustrating argument and we deal with a proper proof later in Section 4.7.

Now that we have seen how to implement the transposition of square matrices, it is simple to obtain a function that computes the transposition of non-square matrices as well. However, we still need to know how many columns a matrix has. This information cannot be extracted from our reduced representation, because it requires that there is at least one entry in the last column, which may not be true. We thus use an additional parameter for the non-square transposition function that provides the number of columns in the matrix.

```
transposeNonSquare :: Int → Mat α → Mat α
transposeNonSquare c mat = Mat (fmap Vec (preTranspose vs cols ∩l cols)) where
  vs   = verticesWith [] mat
  cols = toVecWith [] [0 .. c − 1]
(∩l) :: Vec α → Vec β → Vec α
(∩l) = intersectionWithKey (λ_ x _ → x)
```

The basic strategy is the same, but instead of using the same vector to correct possibly missing adjacency lists, we use another one that has the right number of entries that are all $[\,]$. There is no guarantee that the first argument of *transposeNonSquare* is the number of columns of the matrix. We have that for a matrix $m :: Mat\ \alpha$ which represents a matrix $M \in A^{r \times c}$ the result of *transposeNonSquare cNum m* is the transposition of the matrix M considered as a matrix from $A^{r \times cNum}$. In particular, this yields the following properties.

(1) If $cNum = c$, then *transposeNonSquare m* is the representation of $M^{\top}$.

(2) If $cNum < c$, then *transposeNonSquare m* is the representation of $(M_{cNum-})^{\top}$, where M_{cNum-} is the matrix M restricted to its first *cNum* columns.

(3) If $cNum > c$, then *transposeNonSquare m* is the representation of $(M_{cNum+})^{\top}$, where M_{cNum+} is the matrix M extended by $cNum - c$ columns.

In all cases the result of *transposeNonSquare* is indeed a matrix according to our requirements from Section 4.2.4.

4.4 Multiplication Sequences and Reachability

While some of our multiplications from Section 4.3 have quite heterogeneous types that barely resemble any actual vector-matrix multiplication, most of the multiplications have a type like

$$(\odot) :: Vec\ \sigma \to Mat\ \tau \to Vec\ \sigma\ ,$$

where σ is a type that contains some information we are interested in and τ is the type of labels used in the graph. This kind of multiplications is reminiscent of structural acts, as we have already hinted at before in Section 4.2. Such types allow repeated applications: if we have a matrix $m :: Mat\ \tau$ and a vector $v :: Vec\ \sigma$, we can compute $v \odot m$ and use this result in a further multiplication with m, namely $(v \odot m) \odot m$, where we can omit the brackets[14] and write $v \odot m \odot m$. We have presented small examples of this repeated application in Section 4.3.4. One particularly simple example is to successively compute the successors of a list of vertices, beginning with an initial list. For example, consider the vector $v = [1]$ and the graph $graph :: Mat\ Char$, which denotes the graph from Figure 4.1(b). We can then perform the following computations.

ghci> $v \otimes_{\rightarrow} graph$
$[0,2]$
ghci> $v \otimes_{\rightarrow} graph \otimes_{\rightarrow} graph$
$[0,2,3,4]$

[14]The Haskell 2010 Report, Section 4.4.2 [Mar09] states that an operator without a fixity annotation automatically has the highest precedence and associates to the left.

```
ghci> v ⊗→ graph ⊗→ graph ⊗→ graph
[0,2,3,4,5]
```

This naïve sequence repeats several operations every time a multiplication is applied. For instance, we compute the successors of the vertex 0 two times in the final example. In order to avoid this repetition, we can implement a simple reachability scheme that is based upon vector-matrix multiplication.

We can use relation algebra to specify what we are computing in case of simple reachability. To that end let $(\mathfrak{T}, \mathcal{R}\,(,)\,, \cdot, \mathsf{I})$ be a relation algebra, $D, T \in \mathfrak{T}$, $n \in \mathbb{N}_{>0}$ and $r : \mathbb{N}_{<n} \to \mathcal{R}\,(T, T)$. We consider for every $i \in \mathbb{N}_{<n}$ the relation $r(i)$ to be some graph. Now let $s \in \mathcal{R}\,(D, T)$ be a (relation-algebraic) vector. The vector s can be interpreted as a set of vertices as we have discussed in Section 2.4. We then compute a sequence of vectors defined as follows:

$$\begin{aligned} \mathsf{step}_0 &:= s\ , \\ \mathsf{step}_{k+1} &:= \mathsf{step}_k \cdot \left(\prod_{i=0}^{n-1} r(i) \right) \cap \overline{\bigcup_{i \in \mathbb{N}_{<k+1}} \mathsf{step}_i} \quad \text{for all } k \in \mathbb{N}\ . \end{aligned}$$

Then we have the following result:

$$\bigcup_{k=0}^{|V|-1} \mathsf{step}_k = s \cdot \left(\prod_{i=0}^{n-1} r(i) \right)^{\star}\ . \tag{4.4.i}$$

The first step is simply the set of vertices we begin with. Each successive step is obtained by going along edges in $r(0)$ through $r(n-1)$ and finally removing (intersecting with the complement) all vertices we have already reached in a previous step. Actually, the number of steps that is necessary to compute the reachability is usually less than $|V| - 1$, because once a reachability step is empty, all subsequent steps are empty as well. However, the bound is tight, because in the graph $(\{0, 1\}, \{(0, 1)\})$ there are two vertices and two reachability steps from the set $\{0\}$.

Note that the product of the relations is irrelevant in the above definition, we could have used a single relation. The reason we did not is modularity: in an actual program, we would not want to compute $v \cdot (A \cdot B)$, because matrix multiplication of square matrices is cubic in the number of rows of the matrices. Instead, computing $(v \cdot A) \cdot B$, which is the same value, comes with only quadratic complexity in the number of rows of the matrices, but requires the knowledge of both A and B.

When abstracting from the relational case, we observe that in the expression $(v \cdot A) \cdot B$ the "$\cdot$" denotes vector-matrix multiplication. Thus the only additional component we need to implement the above specification is a complement function. In relational terms, one typically considers *absolute* complements, which is to say that $r \cup \overline{r} = \mathsf{L}$ and $r \cap \overline{r} = \mathsf{O}$ holds for all relations r. In order to implement this type of complement in case of vectors over a structure requires the dimension of the vector, which is the n in the expression $v \in S^n$. With our convention about vectors we cannot access said n, since there is no way of telling whether a position is missing because it

is filled with a zero or because it is larger than the intended dimension. However, we can use the notion of the *relative* complements, which correspond to set difference in case of sets. Clearly, for all sets $A, B \in \mathcal{P}(X)$ we have $A \cap \overline{B} = A \setminus B$, where the complement is taken with respect to X. We can use this variant of complements to implement the function

$$(\setminus) :: Vec\ \alpha \to Vec\ \beta \to Vec\ \alpha$$

for relative complements in a similar fashion as we did for the union and intersection of vectors. The heterogeneous type of $(\setminus)$ hints at the fact that we ignore the values in the second argument and only remove those indices from the first argument that are filled in the second argument.

With this function at hand it is simple to come up with a function that is parametrised over the actual vector-matrix multiplication and yields the reachability steps we have defined above.

```
stepsWith :: (Vec α → Mat β → Vec α) → Vec α → [Mat β] → [Vec α]
stepsWith _   r [] = [r]
stepsWith mul r gs = go r (verticesWith () (head gs)) where
  go v w | isEmptyVec v = []
         | otherwise     = v : go v' w' where
                             w' = w \ v
                             v' = foldl mul v gs ∩| w'
```

In case the supplied multiplication maintains the relational context[15], which is to say that indices of the result vector are exactly the successors of the indices in the argument vector, we have the following result about the list computed above. Suppose that *steps* is the result list of *stepsWith*. For every $i \in \mathbb{N}_{<length\ steps}$ the vector *steps* !! i corresponds to the set of vertices reachable from *steps* !! 0 in exactly i steps, by walking through all supplied graphs exactly once per step. Each vertex is additionally labelled with some information collected by the supplied vector-matrix multiplication. This list resembles the list of steps computed during a *breadth-first search* (BFS), but without the knowledge about the actual order in which the vertices were visited. The above function incorporates a minor improvement, because once an empty reachability step is reached, the local function *go* returns the empty list and thus ends the search for further reachability steps. The current step is maintained in the local variable v. For simplicity, we use the indices of the first matrix only as the vertex set. Alternatively, one could also take the union of all indices. The latter approach comes with some additional overhead, because it requires an iterated union operation.

The original task was to compute those vertices that are reachable from a start set along a shortest path. We can compute these vertices using Equation (4.4.i), which states that we need to combine the vectors in the list provided by *stepsWith*. Since this

[15]This may not hold for arbitrary multiplications. For example, using a semiring with zero divisors, certain successors can have a zero value in the result vector of the given multiplication.

lists contains distinct reachability steps, all vectors in this list are pairwise disjoint. Assuming a multiplication that maintains the relational context, we can reuse our function *leftmostUnion* from Section 4.3.2.

$$\begin{array}{l} \mathit{reachableWith} :: (\mathit{Vec}\ \alpha \to \mathit{Mat}\ \beta \to \mathit{Vec}\ \alpha) \to \mathit{Vec}\ \alpha \to [\mathit{Mat}\ \beta] \to \mathit{Vec}\ \alpha \\ \mathit{reachableWith}\ \mathit{mult}\ \mathit{start}\ \mathit{gs} = \mathit{leftmostUnion}\ (\mathit{stepsWith}\ \mathit{mult}\ \mathit{start}\ \mathit{gs}) \end{array}$$

One useful application of having the reachability steps at hand is the following pattern for the computation of shortest paths between two vectors. The key idea is to use Equation (4.4.i) to compute the first reachability step from the source vector s that has a non-empty intersection with the target vector t. We then have the following equalities:

$$\begin{aligned} & s \cdot \left(\prod_{i=0}^{n-1} r(i)\right)^{\star} \cap t \\ = \quad & \wr \text{ by Equation (4.4.i) } \wr \\ & \left(\bigcup_{k=0}^{|V|-1} \mathsf{step}_k\right) \cap t \\ = \quad & \wr \text{ distributive law } \wr \\ & \bigcup_{k=0}^{|V|-1} (\mathsf{step}_k \cap t) \ . \end{aligned} \tag{4.4.ii}$$

We use Equation (4.4.ii) for the implementation as follows.

$$\begin{array}{l} \mathit{shortestWith} :: (\mathit{Vec}\ \alpha \to \mathit{Mat}\ \beta \to \mathit{Vec}\ \alpha) \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ \beta \to [\mathit{Mat}\ \beta] \to \mathit{Vec}\ \alpha \\ \mathit{shortestWith}\ \mathit{mul}\ \mathit{start}\ \mathit{end}\ \mathit{gs} = \\ \quad \mathit{head}\ (\mathit{dropWhile}\ \mathit{isEmptyVec}\ (\mathit{map}\ (\cap_{|}\ \mathit{end})\ (\mathit{stepsWith}\ \mathit{mul}\ \mathit{start}\ \mathit{gs})) +\!\!+ [\mathit{emptyVec}]) \end{array}$$

The call *shortestWith mul start end gs* finds those vertices in *end* which are reachable from *start* along shortest paths through *gs* and labels these vertices with the information collected by *mul*, where *mul* is again a multiplication that maintains the relational context. To achieve that we first compute the reachability steps, and intersect every element with the target set. Then we drop the results as long as they are empty, add the empty vector to the very end of this intermediate list and return the head of this list. While it looks as though we add unnecessary complexity using $+\!\!+[\mathit{emptyVec}]$, this is not the case. If the list returned by the *dropWhile* call is not empty, computing its head does not require the evaluation of $+\!\!+[\mathit{emptyVec}]$ due to non-strictness. If said list is empty, then we compute $[\,] +\!\!+ [\mathit{emptyVec}]$ which is a constant time operation.

Many graph algorithms that are phrased in imperative pseudo-code contain references to BFS-like schemes, where typically an actual BFS is modified to fit the necessary purpose. For example, one can label visited vertices with a list of the predecessors that lead to this vertex. Essentially, this labelling is what our multiplication $(\odot_{\sim})$ is accomplishing, because $(\odot_{\sim})$ clearly maintains the relational

context. Thus using *reachableWith* $(\odot_{\sim})$ has a very similar effect as described above, but the modification is obtained through parametrisation instead of rewriting.

It is reasonable to assume that a multiplication $(\circledast)$ computes the product $v \circledast m$ in $\mathcal{O}(\mathit{size}\ v \cdot \mathit{dim}\ m)$ steps, where *dim m* is the number of rows of a square matrix m, because this is the complexity of a naïve implementation as well. Assuming such a multiplication, the reachability function above is quadratic in *dim m*. While this complexity is not optimal, it is still better than a version that uses the reflexive-transitive closure of a relation explicitly. For every relation A Equation (4.4.i) states that we compute $v \cdot A^*$ without computing A^* first. Computing the star closure with the means of Chapter 3 is cubic in the number of rows of the matrix, regardless of the actual version. Hence, the overall complexity of first computing A^* followed by a vector-matrix multiplication with v has a worst-case cubic complexity in the dimension of the matrix. This is due to the fact that a vector-matrix multiplication can depend on all values in a matrix and thus a complete evaluation of A^* is indeed required in general. An additional benefit of our implementation is that the traversal of a list of matrices in (cyclic) sequence merely increases the implicit constants in the quadratic term $\mathcal{O}\left((\mathit{dim}\ m)^2\right)$, while the star closure approach requires a relational multiplication and thus increases the implicit constants in the term $\mathcal{O}\left((\mathit{dim}\ m)^3\right)$, at least in the canonic version. Structurally, the difference between the two approaches corresponds to the difference between computing $v \cdot (A \cdot B)^*$ with a single function and first computing $(A \cdot B)^*$, followed by a multiplication with v. Note that even with non-strict evaluation it is still more efficient to compute $(v \cdot A) \cdot B$ rather than $v \cdot (A \cdot B)$ because the vector-matrix multiplication may depend on all values in $A \cdot B$.

Interestingly, we can improve our result further using a sum function that writes all values in an intermediate structure combining the results with a supplied function as we have discussed in Section 4.2.1. Suppose we have a vector v with k non-zero entries that denotes a subset of the vertices of a graph represented by a square matrix m with n rows. Then the vector-matrix multiplication first requires an intersection, which is linear in n, because the vector has a length of at most n. After the intersection, we have an intermediate list of k vectors *mvs*. Assuming a constant-time combination function, the modified sum of these vectors writes every element in every one of the vectors in an intermediate structure, which takes $\mathcal{O}\left(\mathit{sum}\ (\mathit{map}\ \mathit{size}\ \mathit{mvs}) \cdot \log_2(n)\right)$ steps for building the intermediate structure and an additional number of steps to transform said structure back into a result vector. However, this additional number is at most *sum* (*map size mvs*), because every filled position in the result vector is contained in at least one vector in *mvs* and thus the additional operation simply increases the constant in the $\mathcal{O}$-term above. Since all vectors in the result list of *stepsWith* are pairwise disjoint, each adjacency list occurs at most once in an intermediate list *mvs*. Thus the overall complexity of *stepsWith* is $\mathcal{O}\left(|E| \cdot \log_2(n)\right)$, because the sum of the lengths of all adjacency lists is exactly the number of edges $|E|$ in the graph. While slightly worse than the imperative implementation that takes $\mathcal{O}\left(|E| + n\right)$ operations, we still benefit from a modular and purely functional implementation.

4.5 Finding Disjoint Shortest Paths

In this section we show how our multiplication scheme combined with the reachability function from Section 4.4 can be applied to solve the non-trivial problem of finding a maximal set of pairwise disjoint shortest paths between two sets of vertices in a graph. By "maximal" we mean a maximal element with respect to inclusion and paths are called disjoint if and only if their respective vertex sets are disjoint. Our solution depends on the pruning scheme of King and Launchbury [KL95], which in turn relies on Haskell's non-strictness. The original solution of this problem by Hopcroft and Karp [HK73] can be split into two parts:

(1) a BFS on the graph determines the vertices that are reachable from the first set

(2) a DFS that finds paths between the two sets and removes all vertices along these paths from the graph, until no paths remain.

The technique we present is based upon the algorithm given by Dinitz [Din06] and constitutes a variation of the original algorithm.

Suppose that we have a reachability forest, such that every vertex that is contained in a tree in that forest occurs in a shortest path from the first set to the second one. With this prerequisite, all we need to do is to perform a depth-first search on this forest and to collect the path along the way. Actually, the situation is even better, because we know that every tree in the forest contains at most one path that is disjoint with all other paths we have found so far, because a forest is a collection of trees and trees in Haskell have a unique root.

Now the task is to obtain this forest. In order to do that we use a similar approach as King and Launchbury [KL95] and use the definitions of trees and forests from *Data.Tree*.

```
type Forest α = [Tree α]
data Tree α   = Node α (Forest α)
```

The above type of trees is sometimes called *rose trees* and allows arbitrary branching. Rose trees are particularly well-suited to collect unevaluated computations. To obtain a forest as required, we define a multiplication that extends forests by one step.

Let $v :: \mathit{Vertex}$ be a vertex, and consider a forest $f_v :: \mathit{Forest\ Vertex}$, such that every shortest path from the starting set to v is contained in f_v. Now let w be a successor of v. We wish to construct a forest $f_w :: \mathit{Forest\ Vertex}$ such that every shortest path from the starting set to w is contained in f_w. To achieve that we can consider all predecessors of w, compute the forests with the above property for every predecessor and then simply take

$$f_w = [\mathit{Node}\ x\ f_x \mid x \leftarrow \mathit{predecessors}\ w]$$

as the desired forest. Instead of this backwards construction, we can use a forward construction, too. To that end we label every successor of a vertex x with a trivial

forest that consists of exactly the tree *Node x* f_x and collect all such forests into one using our function *allUnion* from Section 4.3.4.

$(\odot_{\Cup}) :: Vec\ (Forest\ Vertex) \rightarrow Mat\ \alpha \rightarrow Vec\ (Forest\ Vertex)$
$(\odot_{\Cup}) = vecMatMult\ allUnion\ forestMult$

$forestMult :: Vertex \rightarrow Forest\ Vertex \rightarrow Vec\ \alpha \rightarrow Vec\ (Forest\ Vertex)$
$forestMult = sMultWith\ (\lambda v\ forest\ _ \rightarrow [Node\ v\ forest])$

Note that $(\odot_{\Cup})$ maintains the relational context (cf. Section 4.4). While it may seem inefficient to compute the concatenation of singleton lists, in this case this comes with only constant overhead. The reason is that all our implementations for the *bigUnionWith* function, where $allUnion = bigUnionWith\ (+\!\!+)$, are based on right-folds and since concatenation is linear in the size of its *first* argument, it is a constant time operation[16] in case of the call $[x] +\!\!+ xs$.

This new multiplication is then used as a parameter for our reachability function *shortestWith* from Section 4.4. Note that the start vector may contain a forest from a previous computation.

$reachForest :: Vec\ (Forest\ Vertex) \rightarrow Vec\ \beta \rightarrow [Mat\ \gamma] \rightarrow Vec\ (Forest\ Vertex)$
$reachForest = shortestWith\ (\odot_{\Cup})$

Let us have a look at an example. Consider the following example graph in Figure 4.2. There are several different maximal sets of pairwise disjoint paths from $\{0,1\}$ to $\{7,8\}$ in this graph. For example, $\{(0,3,6,8)\}$ is such a set, as is the set $\{(0,2,5,7),(1,3,6,8)\}$. The function *reachForest* computes the intermediate steps

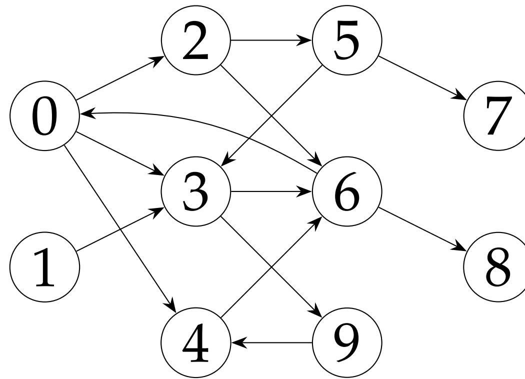


Figure 4.2. A graph with several maximal disjoint path sets from left to right.

shown in Figure 4.3, where the first component is the vertex in the layer and the second component is the reachability forest that leads to this vertex from the start vector. We observe that every vertex that is contained in any intermediate forest is located on a shortest path from the vertex set $\{0,1\}$ to $\{7,8\}$. Note that subtrees can occur more than once, as can be seen in case of the second step, which is why the resulting

[16] There are two pattern matches performed on the first argument. The complete computation is $[x] +\!\!+ xs = (x : []) +\!\!+ xs = x : ([] +\!\!+ xs) = x : xs$.

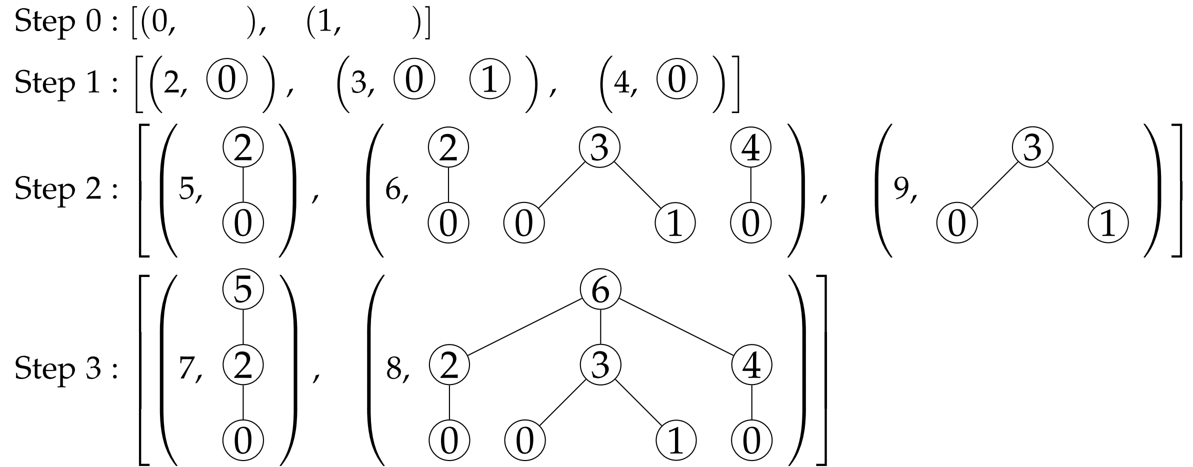


Figure 4.3. Stepwise reachability forests in the example graph

forests need to be pruned with a suited strategy. The basic technique for this pruning operation is very similar to the one presented by King and Launchbury [KL95]. We use a monadic set interface *SetM* that provides the following functions.

$include :: Vertex \to SetM\ ()$	adds a vertex to the monadic set
$contains :: Vertex \to SetM\ Bool$	checks whether a vertex is contained in the set
$runNew :: Int \to SetM\ \alpha \to \alpha$	creates a new set and computes its effect

The actual implementation is interchangeable and the module *Data.Graph* provides two instances: one with constant-time update arrays and another with logarithmic-time update trees.

For simplicity of presentation we represent paths using the same data type that we used for walks in Section 4.3.4.

```
type Path = Walk
```

While there is no guarantee that walks are actually paths, we use the *Path* synonym only in the context of shortest walks, which are also shortest paths.

Now suppose that (i, f) is an element in the result vector of the *reachForest* function. As we have already stated above, the forest f may contain at most one path that is of interest, because every other path has the same final vertex i. In other words, there might not be a path along unvisited vertices through a given tree in f, which indicates that the result type of the pruning operation should be *SetM* (*Maybe Path*). Instead of combining the two monads *SetM* and *Maybe* manually, we use a monad transformer [LHJ95] called *MaybeT* for a less convoluted solution.

```
chop :: Forest Vertex → MaybeT SetM Path
chop []                = mzero
chop (Node v ts : fs) = do b ← lift (contains v)
```

$$
\begin{array}{l}
\quad\quad\mathbf{if}\ b\ \mathbf{then}\ \mathit{chop}\ \mathit{fs} \\
\quad\quad\quad\mathbf{else\ do}\ \mathit{lift}\ (\mathit{include}\ v) \\
\quad\quad\quad\quad\quad\quad \mathit{fmap}\ (\rhd v)\ \mathit{candidate}\ \text{`}\mathit{mplus}\text{`}\ \mathit{chop}\ \mathit{fs} \\
\mathbf{where}\ \mathit{candidate}\ \mid \mathit{null}\ \mathit{ts} \quad\ \ = \mathit{return}\ \mathit{empty} \\
\quad\quad\quad\quad\quad\quad\quad\ \mid \mathit{otherwise} = \mathit{chop}\ \mathit{ts}
\end{array}
$$

If there is no tree left, there is no path left in the current forest and the value *mzero* is returned, which corresponds to the lifted version of *Nothing*. Otherwise there is a left-most tree *Node v ts* in the forest. Now if the root vertex v of this tree has been visited previously, we continue with the remaining forest. If the root vertex has not been visited so far, we visit the root vertex and compute a path candidate in the subforest of this vertex *ts*. The candidate is then extended by its final vertex, which is v. The *mplus* function returns the left-most occurrence of a non-failing value (similar to the *mplus* instance for *Maybe*). Thus if the candidate is in fact a path, the extended path is returned, otherwise the search for a path continues in the remaining forest. Finally, the candidate is simple to compute: if the subforest of the root node is empty, we have reached the bottom of the forest and the path candidate is the empty path; otherwise the candidate is the result of the recursive call to *chop*.

Let us now implement the actual function that computes a maximal set of pairwise disjoint shortest paths. Recall that the result of *reachForest* returns a vector where every vertex in this vector is labelled with the shortest reachability forest that leads to this vertex from the starting set. Thus we need to apply the function $\mathit{chop} \circ \mathit{uncurry}\ \mathit{Node}$ to every pair (i, f) that is contained in this result vector. Then we need to collect the result of every such call, which is simply *map runMaybeT*, where *runMaybeT* returns the content of the *MaybeT* container. We then *sequence* these results and compute the monadic effect with *runNew n*, where n is the number of vertices in the graph. Finally, we remove possible *Nothing* values using the function $\mathit{catMaybes} :: [\mathit{Maybe}\ \alpha] \to [\alpha]$, which removes all occurrences of *Nothing* in the argument list and unwraps all *Just* values. We optimise the above construction as follows:

$$
\begin{array}{ll}
& \mathit{sequence} \circ \mathit{map}\ \mathit{runMaybeT} \circ \mathit{map}\ (\mathit{chop} \circ \mathit{uncurry}\ \mathit{Node}) \\
= & \wr\ \mathit{map}\ \text{is a homomorphism}\ \int \\
& \mathit{sequence} \circ \mathit{map}\ (\mathit{runMaybeT} \circ \mathit{chop} \circ \mathit{uncurry}\ \mathit{Node}) \\
= & \wr\ \text{property of}\ \mathit{mapM}\ \text{on lists:}\ \mathit{sequence} \circ \mathit{map}\ f = \mathit{mapM}\ f\ \int \\
& \mathit{mapM}\ (\mathit{runMaybeT} \circ \mathit{chop} \circ \mathit{uncurry}\ \mathit{Node})\ .
\end{array}
$$

The main function is then a combination of the above ideas.

$$
\begin{array}{l}
\mathit{disjointPaths} :: \mathit{Int} \to \mathit{Vec}\ (\mathit{Forest}\ \mathit{Vertex}) \to \mathit{Vec}\ \beta \to [\mathit{Mat}\ \gamma] \to [\mathit{Path}] \\
\mathit{disjointPaths}\ n\ \mathit{start}\ \mathit{end}\ \mathit{gs} = \mathit{catMaybes}\ (\mathit{process}\ (\mathit{reachForest}\ \mathit{start}\ \mathit{end}\ \mathit{gs}))\ \mathbf{where} \\
\quad \mathit{process} = \mathit{runNew}\ n \circ \mathit{mapM}\ (\mathit{runMaybeT} \circ \mathit{chop} \circ \mathit{return} \circ \mathit{uncurry}\ \mathit{Node}) \circ \mathit{unVec}
\end{array}
$$

Assuming that the graph from Figure 4.2 is represented by the value $\mathit{graph} :: \mathit{Graph}\ ()$ in Haskell, we get the following result.

```
ghci> disjointPaths 10 (toVecWith [] [0,1]) (toVec [7,8]) [graph]
[⟨0,2,5,7⟩,⟨1,3,6,8⟩]
```

Despite some additional technicality and the length of this implementation, the solution to the disjoint path problem is still reasonably simple. In particular, it is still compositional and depends on the reachability function from Section 4.4 and a certain multiplication. Additionally, exchanging a path from a vertex set to another by a maximal set of pairwise disjoint paths is very simple, due to the modularity that comes with the purely functional approach. We will see examples of this in Chapter 5 when comparing the versions of Section 5.3 and Section 5.4.

4.6 Type Classes for Set Operations

In this section we pick up on the idea of generalising the set operations using type classes, which we have already mentioned before on several occasions. So far we have used some operations on vectors that resembled set operations like *unionWith* or *intersectionWithKey* from Section 4.2. We did not provide the implementation for these functions, but stated that all of them can be implemented in essentially the same fashion. In fact, we can define the following generalised merging operation on arbitrarily sorted lists [Dan12]. The idea is to abstract all components of an actual merging operation into the first six arguments. Supplying the first six arguments to *mergeWith* provides a function that takes two lists (the seventh and eighth arguments) and produces a result. These two lists are referred to in the comments below.

```
mergeWith :: (α → β → Ordering)   comparison function
          → ([β] → γ)             action in case the first list is empty
          → ([α] → γ)             action in case the second list is empty
          → (α → β → γ → γ)       operation in the LT case
          → (α → β → γ → γ)       operation in the EQ case
          → (α → β → γ → γ)       operation in the GT case
          → [α] → [β] → γ
mergeWith cmp lEmpty rEmpty lt eq gt = merge where
  merge []         ys         = lEmpty ys
  merge xs         []         = rEmpty xs
  merge l@(x : xs) m@(y : ys) = case cmp x y of
                                  LT → lt x y (merge xs m)
                                  EQ → eq x y (merge xs ys)
                                  GT → gt x y (merge l  ys)
```

The actual set operations can be easily defined in terms of this general pattern and some auxiliary functions.

```
opCons :: (α → β → γ) → α → β → [γ] → [γ]
opCons op x y l = op x y : l
```

$thirdArg :: \alpha \to \beta \to \gamma \to \gamma$
$thirdArg\ _\ _\ x = x$

$cmpFst :: Ord\ \kappa \Rightarrow (\kappa, \alpha) \to (\kappa, \beta) \to Ordering$
$cmpFst\ (i, _)\ (j, _) = compare\ i\ j$

For the union operation we note that in case one of the lists is empty, the other one is returned and in case of non-empty lists we add all values in ascending sequence omitting duplicates to the result list. Thus an implementation can look as follows.

$unionWith :: (\alpha \to \alpha \to \alpha) \to Vec\ \alpha \to Vec\ \alpha \to Vec\ \alpha$
$unionWith\ f\ v\ w = Vec\ (merge\ (unVec\ v)\ (unVec\ w))$ **where**
$\quad merge = mergeWith\ cmpFst\ id\ id\ (opCons\ const)$
$\qquad (opCons\ (\lambda(i, x)\ (_, y) \to (i, f\ x\ y)))$
$\qquad (opCons\ (flip\ const))$

The intersection operation discards values in all cases other than the non-empty equality case. We can implement it in the following fashion.

$intersectionWithKeyL :: (\alpha \to \beta \to \gamma) \to Vec\ \alpha \to Vec\ \beta \to [\gamma]$
$intersectionWithKeyL\ f\ v\ w = merge\ (unVec\ v)\ (unVec\ w)$ **where**
$\quad merge = mergeWith\ cmpFst\ (const\ [\,])\ (const\ [\,])$
$\qquad thirdArg$
$\qquad (opCons\ (\lambda(i, x)\ (_, y) \to (i, f\ i\ x\ y)))$
$\qquad thirdArg$

Other set operations like the (symmetric) difference of sets can be implemented in a similar fashion. With these concrete implementations at hand we can reason about their properties and complexities. For example, one can show that given an associative operation $op :: \alpha \to \alpha \to \alpha$ the corresponding vector operation $unionWith\ op :: Vec\ \alpha \to Vec\ \alpha \to Vec\ \alpha$ is associative as well. However, the direct proof is rather technical and unpleasant, which is why we omit it for now and discuss it in Section 4.7 again.

These merging operations as well as their abstraction are not new. For example, *Data.IntMap.Lazy* provides the function *mergeWithKey*, which is a special case of our *mergeWith* function that works on integer maps. However, merging operations have linear complexities in the sum of the sizes of both their arguments and this can be less efficient for repeated applications than a non-merging variant as we have discussed in Section 4.2.1. We can abstract the components of these merging operations using type classes, which denote that a structure supports the notion of a union or an intersection. For example, the union of two structures can be linear in the sum of their sizes. In case of two different structures, say one that can be traversed[17] and one that supports insertions at keys that are logarithmic in the size of the key, it may

[17]This kind of traversal is only a special case of the more general approach defined in terms of the *Traversable* type class that provides a function $traverse :: (Applicative\ \varphi, Traversable\ \tau) \Rightarrow (\alpha \to \varphi\ \beta) \to \tau\ \alpha \to \varphi\ (\tau\ \beta)$. In our case the traversal is essentially an *fmap* over the first structure, which in turn can be expressed in terms of *traverse* and the identity functor.

be better to traverse the first structure while adding the values to the second one. We use this motivation for a heterogeneous approach to set union and define the corresponding type class as follows[18].

class *Unionable* τ ι **where**
  $cupWith :: (\alpha \to \alpha \to \alpha) \to \tau\ \alpha \to \iota\ \alpha \to \iota\ \alpha$
  $cup :: \tau\ \alpha \to \iota\ \alpha \to \iota\ \alpha$
  $cup = cupWith\ const$
  $bigCupWith :: Foldable\ \varphi \Rightarrow (\alpha \to \alpha \to \alpha) \to \iota\ \alpha \to \varphi\ (\tau\ \alpha) \to \iota\ \alpha$
  $bigCupWith\ op = Foldable.foldr \circ cupWith$

The function *cupWith* is the abstraction of the *unionWith* function and the additional functions are merely defined for convenience. We can easily populate this new type class with two interesting inhabitants.

instance *Unionable Vec Vec* **where**
  $cupWith = unionWith$

instance *Unionable Vec IntMap* **where**
  $cupWith\ op\ v\ im = foldr\ (uncurry\ (insertWith\ op))\ im\ (unVec\ v)$

The first instance simply allows *Vec*s to be added exactly as before and has the complexity

$$\mathcal{O}(|v| + |w|)\ ,$$

where v, w are the vectors that are unified. The second one is an example of the insertion strategy, because all values in the vector are inserted in the *IntMap*. In this case, the first vector is traversed, while the second one is queried, which yields a complexity of

$$\mathcal{O}\left(|v| \cdot \log_2(|w|)\right)\ ,$$

where v is the first and w is the second argument of the union. We can also define a homogeneous union for *IntMap*s, taking either the function *Data.IntMap.Lazy.unionWith* as *cupWith* or by copying the approach from the heterogeneous instance above. The actual complexities depend on the particular instance. The benefit of this approach is that these complexities are exposed and one can obtain an overall complexity estimate based upon the data types used in a given application. Recall, however, that the above estimates are only true if one considers a full evaluation of the result as we have mentioned in Section 2.5, because Haskell is a non-strict language and the result of any function is evaluated only as much as necessary for subsequent applications.

We can proceed for the intersection in a similar fashion. In the homogeneous case of two vectors, we can use the function *intersectionWithKey*. If we have a traversable structure and one that supports fast access to values at its keys, we can traverse the first structure while querying the second one. Abstracting this idea leads to the following type class.

[18] To allow type classes that have more than one parameter we need to use the language extension *MultiParamTypeClasses*.

```
class Intersectable τ κ where
  capWithKey :: (Int → α → β → γ) → τ α → κ β → τ γ
  cap :: τ α → κ β → τ α
  cap = capWithKey (const const)
```

Again, we can provide two canonic instances.

```
instance Intersectable Vec Vec where
  capWithKey = intersectionWithKey
instance Intersectable Vec IntMap where
  capWithKey f vec im =
    Vec (mapMaybe (λ(i, x) → fmap (f i x) (IntMap.lookup i im)) (unVec vec))
```

In the second instance the function *mapMaybe* :: ($\alpha \to$ *Maybe* β) $\to [\alpha] \to [\beta]$ maps the supplied function over the list and keeps only the *Just* values. Just as we have seen in the case of the parametrised union, we can also define an instance for the intersection of *IntMap*s with themselves. The first intersection is linear in the sum of the sizes of both arguments, the second intersection has a complexity of

$$\mathcal{O}\left(|vec| \cdot \log_2(|im|)\right) ,$$

where $| - |$ denotes the size of a structure. This latter complexity can be improved further using arrays. Consider the following vector implementation in terms of arrays.

```
newtype VecArray α = VecArray {unVecArray :: Array Int (Maybe α)}
```

The case that we have a key that is filled with *Nothing* means that there is no entry at that key in the vector. Then we can define the intersection quite similarly as for the *IntMap*.

```
instance Intersectable Vec VecArray where
  capWithKey f vec arr =
    Vec (mapMaybe (λ(i, x) → fmap (f i x) (unVecArray arr ! i)) (unVec vec))
```

Essentially, this is the same definition as above, but we replaced the *lookup* operation for integer maps with the query operation of arrays. The latter is a constant time function and thus the complexity of this intersection is merely

$$\mathcal{O}(|vec|)$$

and thus linear in the size of the first argument only. However, we cannot achieve a similar result for the union, because the modification of an array is linear in the size of the array and the size of *VecArray*s is always the dimension of the vector, which is different from the case of *Vec*s, whose size is the number of filled positions.

Both of the above type classes can be generalised even further. For example, the idea of traversing a structure, querying a structure or inserting something into a structure has nothing to do with unions or intersection. Suppose that we define a type class for the generalisation of *mapMaybe* and one for lookups.

$$
\begin{array}{l}
\textbf{class } Functor\ \varphi \Rightarrow KeyMaybeFunctor\ \varphi\ \textbf{where} \\
\quad fmapMaybeWithKey :: (Int \to \alpha \to Maybe\ \beta) \to \varphi\ \alpha \to \varphi\ \beta \\
\textbf{class } Lookup\ \lambda\ \textbf{where} \\
\quad at :: \lambda\ \alpha \to Int \to Maybe\ \alpha
\end{array}
$$

We assume that *fmapMaybeWithKey* maps every value and its implicit key to the result of the supplied function and removes it in case it is *Nothing*. The function *at* is assumed to return the value at a given key, if the key is present in the structure, and *Nothing* otherwise. With these two functions at hand, we can define a generic intersection function.

$$
\begin{array}{l}
genericCapWithKey :: (KeyMaybeFunctor\ \varphi, Lookup\ \lambda) \\
\qquad\qquad\qquad \Rightarrow (Int \to \alpha \to \beta \to \gamma) \to \varphi\ \alpha \to \lambda\ \beta \to \varphi\ \gamma \\
genericCapWithKey\ f\ t\ l = fmapMaybeWithKey\ (\lambda i\ a \to fmap\ (f\ i\ a)\ (l\ \text{`}at\text{`}\ i))\ t
\end{array}
$$

Note that this is actually the generalised definition of the latter two *capWithKey* functions we have described. A similar construction is possible in case of the set union and the set difference, but we omit the details to avoid clutter.

The approach with different type classes for different operations is convenient in the sense that complexity bottlenecks are exposed and a good separation of concerns is achieved. To be able to reason about type class functions, one has to define a set of rules, the user of these type classes needs to follow in order to maintain certain properties. Such rules need to be carefully designed and proved for any new class instance. For example, it is reasonable to assume that *fmapMaybeWithKey* is compatible with the *fmap* function from *Functor* in the sense that the following equality holds for all $f :: \alpha \to \beta$:

$$fmap\ f = fmapMaybeWithKey\ (const\ (Just \circ f))\ .$$

Similarly, one needs to define proper laws for every new type class and possibly for interactions between type classes as well. Examples of these interactions include the connection between the *Unionable* and *Intersectable* type classes (absorption laws) or the combination of the *KeyMaybeFunctor* and the *Lookup* type classes (query after deletion). These type classes and their corresponding laws come at a price. First of all, one needs to check these properties somehow and second, one has to define type class instance for newly added data types. Both, the proofs of the type class laws and the actual instance declarations, are too lengthy to be presented in this work. We refer to our *gwaf* library [Dan15] for details, where we define type classes for all necessary set operations and state rules for these as well.

4.7 Correctness Proofs

While developing new multiplications and discussing properties thereof, we have spent only little time with reasoning about the correctness of the implementation. In

this section we provide a brief outline of how to obtain correctness proofs about the multiplications from Section 4.3 and show that our implementation of the transposition function is in fact correct.

There are two main ingredients that allow relatively simple arguments about the vector-matrix multiplications we have defined.

(1) The connection between the implementation of vectors and corresponding operations and their mathematical counterpart.

(2) The fact that pure Haskell functions are essentially mathematical functions[19].

The first property is guaranteed by the choice of implementation. In the provided implementation, we can check the properties by hand, because we know the exact implementation. In a more abstract approach with type classes as discussed in Section 4.6 the type class rules need to provide this connection, which in turn needs to be checked for every new instance. The second property is based on a somewhat informal approach to the semantics of Haskell, which is usually taken when dealing with program properties.

Recall that our requirements on vectors from Section 4.2.4 state that we can transform a vector $v \in (A \cup \{ \mathsf{Z} \})^n$ into an element $vec :: Vec\ \alpha$ by removing all Z-values and placing the remaining positions in an associative structure. Knowing the size of the represented vector (which is the n in $v \in (A \cup \{ \mathsf{Z} \})^n$), we can also perform a backward transformation and obtain a mathematical vector from a $vec :: Vec\ \alpha$. The latter transformation allows simple reasoning about vector properties in mathematical terms. Also, this transformation can be used to define the mathematical counterpart of the generalised vector-matrix multiplication. Having this definition at hand, we can use the definition we used in the implementation, translate it into the purely mathematical setting and reason about it mathematically.

The generalised vector-matrix multiplication depends on a sum function and a scalar multiplication function. We simplify our approach to those cases, where the sum function is actually a folded binary addition and the scalar multiplication is obtained by mapping a certain function over every component of a vector. The reason for this simplification is that these preconditions are sufficient for some algebraic rules independent of the supplied functions, while the general case strongly depends on the supplied functions, because the structure of the result is not known in general.

We begin with two definitions that extend functions that do not operate on Z-values to variants that can deal with such values. For simplicity we abbreviate

$$A_{\mathsf{Z}} := A \cup \{ \mathsf{Z} \}$$

and for every $t \in \mathbb{N}$ we set

$$\mathsf{Z}_t : \mathbb{N}_{<t} \to A_{\mathsf{Z}} , \quad i \mapsto \mathsf{Z} .$$

[19]They are not exactly the same, because one needs to consider the case of failures (like non-termination), which cannot occur in the mathematical context.

Additionally, we use the curried function notation in this section for convenience. Thus every function

$$f : \prod_{i=1}^{k} A_i \to R$$

will now be written as

$$f : A_1 \to \ldots \to A_k \to R \ .$$

Due to the Curry isomorphism between $C^{A \times B}$ and $(C^B)^A$ this is merely another representation of functions. Just as in Haskell we then write function application by juxtaposition and use brackets only where necessary. We assume that function application binds more strongly than any other operator. For example, we have that $f\ x\ y = f(x)(y)$, which is the curried notation for $f(x, y)$. Due to the binding convention we also have

$$f\ x\ (y+1) = f(x)(y+1) \quad \text{and} \quad f\ x\ y+1 = f(x)(y)+1 \ .$$

4.7.1 Definition (Extended addition and extended multiplication)**.**

(1) Let A be a set and Z be an element such that $\mathsf{Z} \notin A$ and $p : A \to A \to A_{\mathsf{Z}}$. Then we define

$$\mathsf{exA}(p, \mathsf{Z}) : A_{\mathsf{Z}} \to A_{\mathsf{Z}} \to A_{\mathsf{Z}} \ , \quad a\ b \mapsto \begin{cases} p\ a\ b & : a \neq \mathsf{Z} \wedge b \neq \mathsf{Z} \\ b & : a = \mathsf{Z} \\ a & : \text{otherwise} \ . \end{cases}$$

The name exA is a mnemonic for "extended addition".

(2) Let A, B, C be sets, $\mathsf{Z}_A, \mathsf{Z}_B$ be objects such that $\mathsf{Z}_A \notin A$ and $\mathsf{Z}_B \notin B$. Let $\mathsf{Z}_C \in C$ and $\mu : \mathbb{Z} \to A \to B \to C$. Then we define

$$\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, \mathsf{Z}_C) : \mathbb{Z} \to A_{\mathsf{Z}_A} \to B_{\mathsf{Z}_B} \to C \ , \quad i\ a\ b \mapsto \begin{cases} \mu\ i\ a\ b & : a \neq \mathsf{Z}_A \wedge b \neq \mathsf{Z}_B \\ \mathsf{Z}_C & : \text{otherwise} \ . \end{cases}$$

The name exM is a mnemonic for "extended multiplication". //

The function $\mathsf{exA}(p, \mathsf{Z})$ introduces a new value Z that is neutral to the underlying binary operation p. The function $\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, \mathsf{Z}_C)$ adds a the value Z_A to A and the value Z_B to B and makes both these values behave as an annihilating element with respect to the underlying multiplication in the sense that using one of these values as an argument for $\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, \mathsf{Z}_C)$ produces the abstract zero element Z_C. Note that in both cases the original operation may produce zero values as well.

Next we define how to lift a binary operation to a binary operation on vectors and a ternary multiplication to a ternary scalar multiplication in the sense of Section 4.2.2.

4.7.2 Definition (Vector addition and multiplication)**.**

(1) Let A be a set, $n \in \mathbb{N}$ and $p : A \to A \to A$. Then we define

$$+_p : A^n \to A^n \to A^n \ , \quad a\ b \mapsto \left(j \mapsto p\ a_j\ b_j\right) \ .$$

(2) Let A, B, C be sets, $n \in \mathbb{N}$ and $\mu : \mathbb{Z} \to A \to B \to C$. Then we define

$$\mathsf{sm}(\mu) : \mathbb{Z} \to A \to B^n \to C^n \ , \quad i\ a\ b \mapsto \left(j \mapsto \mu\ i\ a\ b_j\right) \ .$$ ⫽

As a final preparation step we define the folded version of a binary function. For reasons of simplicity, we define this function on lists rather than on vectors. This is simply due to the fact that vectors have a fixed length, while lists do not.

4.7.3 Definition (Folded addition).
Let A be a set, $e \in A$ and $p : A \to A \to A$. Then we define the sum function $\sum_{p,e} : A^* \to A$ inductively as follows:

$$\sum_{p,e}[] = e \ ,$$

$$\sum_{p,e}(a : as) = p\ a\ \left(\sum_{p,e} as\right) \quad \text{for all } a \in A \text{ and all } as \in A^* \ ,$$

where we use the Haskell notation [] and : for the list constructors. ⫽

By the universal property of right-folds [Hut99] we get that

$$\sum_{p,e} = \mathsf{foldr}\ p\ e \ ,$$

where foldr is the mathematical version of the *foldr* function. Note that the function p is not necessarily associative and thus we cannot rearrange the parentheses in the definition of the $\sum_{p,e}$ function in general. Still, the folded sum has certain algebraic features, despite the relaxed requirements. In particular, the j-th component of the sum of vectors is the sum of all j-ths components of the added vectors.

4.7.4 Proposition (Projections of folded sums).
Let A be a set, Z an object such that $\mathsf{Z} \notin A$, $p : A \to A \to A_{\mathsf{Z}}$ and $n \in \mathbb{N}$. Then for every $j \in \mathbb{N}_{<n}$ we have

$$\mathrm{pr}_j \circ \sum_{(+_{\mathsf{exA}(p,\mathsf{Z})}),\mathsf{Z}_n} = \sum_{\mathsf{exA}(p,\mathsf{Z}),\mathsf{Z}} \circ \mathsf{map}\ \mathrm{pr}_j \ .$$

The parentheses around $+_{\mathsf{exA}(p,\mathsf{Z})}$ in the expression on the left-hand side of the equation are added only for legibility purposes.

Proof. We could prove this statement by a somewhat technical induction. Instead, we use the concept of *Free Theorems* [Wad89] which provide statements about polymorphic functions from their type alone[20]. The two functions involved above are foldr and map. The free theorem for foldr states[21] that given functions $g : K \to L \to L$, $h : M \to N \to N$, $a : K \to M$ and $b : L \to N$ such that

$$\forall\, k \in K : \forall\, l \in L : b\ (g\ k\ l) = h\ (a\ k)\ (b\ l) \tag{4.7.4.a}$$

holds, we get for all $l_0 \in L$ that

$$b \circ \mathsf{foldr}\ \ g\ l_0 = \mathsf{foldr}\ \ h\ (b\ l_0) \circ \mathsf{map}\ \ a \ .$$

[20]This technique is not only useful, but can be fully automated. The web interface at `http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi` provides an implementation.

[21]Save for the names of the sets this statement is the one given by Wadler [Wad89].

Now let $K := L := (A_{\mathsf{Z}})^n$, $M := N := A_{\mathsf{Z}}$, $a := b := \mathrm{pr}_j$ and $g := +_{\mathsf{exA}(p,\mathsf{Z})}$, $h := \mathsf{exA}(p, \mathsf{Z})$. We need to check Equation (4.7.4.a), which after inlining states that

$$\forall\, k, l \in (A^n)_{\mathsf{Z}_n} : \mathrm{pr}_j \left(k +_{\mathsf{exA}(p,\mathsf{Z})} l\right) = \mathsf{exA}(p, \mathsf{Z}) \left(\mathrm{pr}_j\, k\right) \left(\mathrm{pr}_j\, l\right) .$$

Let $k, l \in (A^n)_{\mathsf{Z}^n}$. Then we find

$$\begin{aligned}
&\mathrm{pr}_j \left(k +_{\mathsf{exA}(p,\mathsf{Z})} l\right)\\
=\;&\wr \text{ by Definition 4.7.2.(1) and infix notation for } +_{\mathsf{exA}(p,\mathsf{Z})} \,\wr\\
&\mathrm{pr}_j \left(i \mapsto \mathsf{exA}(p, \mathsf{Z})\; k_i\; l_i\right)\\
=\;&\wr \text{ definition of the projection function } \wr\\
&\mathsf{exA}(p, \mathsf{Z})\; (k_j)\; (l_j)\\
=\;&\wr \text{ the } j\text{-th component is the application of the } j\text{-th projection } \wr\\
&\mathsf{exA}(p, \mathsf{Z}) \left(\mathrm{pr}_j\, k\right) \left(\mathrm{pr}_j\, l\right) .
\end{aligned}$$

Thus the condition in Equation (4.7.4.a) is satisfied and we get

$$\mathrm{pr}_j \circ \mathsf{foldr} \left(+_{\mathsf{exA}(p,\mathsf{Z})}\right) (\mathsf{Z}_n) = \mathsf{foldr}\,(\mathsf{exA}(p, \mathsf{Z}))\,(\mathrm{pr}_j\; \mathsf{Z}_n) \circ \mathsf{map}\;\, \mathrm{pr}_j$$

by the free theorem for foldr. Since $\mathrm{pr}_j\; \mathsf{Z}_n = \mathsf{Z}$, this is exactly what we claimed. ⫽

Now we can finally define the mathematical version of the vector-matrix multiplication that we have discussed in Section 4.2.3.

4.7.5 Definition (Vector-matrix multiplication)**.**
Let A, B, C be sets, $\mathsf{Z}_A, \mathsf{Z}_B, \mathsf{Z}_C$ be objects such that $\mathsf{Z}_A \notin A$, $\mathsf{Z}_B \notin B$ and $\mathsf{Z}_C \notin C$. Let $p : C \to C \to C_{\mathsf{Z}_C}$, $\mu : \mathbb{Z} \to A \to B \to C_{\mathsf{Z}_C}$ and $k, l \in \mathbb{N}$. Set $(\mathsf{Z}_C)_l : \mathbb{N}_{<l} \to C_{\mathsf{Z}_C}\;,\; i \mapsto \mathsf{Z}_C$
Then we define

$$\begin{aligned}
\odot_{p,\mu} &: \left(A_{\mathsf{Z}_A}\right)^k \to \left(B_{\mathsf{Z}_B}\right)^{k\times l} \to \left(C_{\mathsf{Z}_C}\right)^l \;,\\
&\quad v\; m \mapsto \sum\nolimits_{\left(+_{\mathsf{exA}(p,\mathsf{Z}_C)}\right),(\mathsf{Z}_C)_l} \left[\, \mathsf{sm}(\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, \mathsf{Z}_C))\; i\; a\; m_i \;\middle|\; i \leftarrow [0, \ldots, k-1] \right] ,
\end{aligned}$$

where the list notation means the same as Haskell's list comprehension notation. That is to say, the above list can be read as

$$[\mathrm{mult}\; 0\; a\;\; (m_0)\,,\; \ldots,\; \mathrm{mult}\; (k-1)\; a\;\; (m_{k-1})]\;,$$

where $\mathrm{mult} = \mathsf{sm}(\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, \mathsf{Z}_C))$. Again, the parentheses around $+_{\mathsf{exA}(p,\mathsf{Z}_C)}$ are added only for the sake of legibility and we omit them in future applications. ⫽

Note that this is essentially the very definition we used in our implementation. The main difference is that we now use an explicit intersection, by simultaneously traversing the vector and the matrix, which, due to the purely mathematical representation, now have the same length in terms of index areas. While we required some technical definitions for the extensions of the addition function and the function for scalar multiplication, the resulting vector-matrix multiplication is still rather simple.

Let us now consider the multiplication $(\odot_{\text{out}})$ from Section 4.3.5 and its application to transposition. We have defined

$$(\odot_{\text{out}}) :: \mathit{Vec}\ [\mathit{Arc}\ \lambda] \to \mathit{Mat}\ \lambda \to \mathit{Vec}\ [\mathit{Arc}\ \lambda]$$
$$(\odot_{\text{out}}) = \mathit{vecMatMult}\ \mathit{allUnion}\ \mathit{outMult}$$

which after inlining the functions *allUnion* and *outMult* reads as follows.

$$(\odot_{\text{out}}) = \mathit{vecMatMult}\ (\mathit{foldr}\ (\mathit{unionWith}\ (+\!\!+))\ \mathit{emptyVec})\ (\mathit{sMultWith}\ (\lambda i\ \mathit{ovs}\ a \to (i,a) : \mathit{ovs}))$$

The key ingredients are the $(+\!\!+)$ function, which is lifted to vectors and then folded over lists of vectors, and the multiplication, which is extended to a scalar multiplication. These functions can be modelled mathematically as follows. Let us assume that the type λ has the mathematical interpretation L. We define $A := (\mathbb{Z} \times L)^*$, $B := L$ and $C := (\mathbb{Z} \times L)^+$. Additionally, let $\mathsf{Z}_A, \mathsf{Z}_B$ be objects such that $\mathsf{Z}_A \notin A$, $\mathsf{Z}_B \notin B$ and $\mathsf{Z}_C := []$. We then translate the Haskell functions above to the mathematical notation:

$$p : C \to C \to C_{\mathsf{Z}_C}\ , \quad p = (+\!\!+)\ ,$$
$$\mu : \mathbb{Z} \to A \to B \to C_{\mathsf{Z}_C}\ , \quad i\ \mathit{ovs}\ l \mapsto (i,l) : \mathit{ovs}\ .$$

Then the function $(\odot_{\text{out}})$ is mathematically represented by $\odot_{p,\mu}$. Since we used the function to define transposition, let us have a look at this application. In Section 4.3.5 we have stated that the result vector carries in each component the necessary information to easily build the transposed graph from this information. Let $k, l \in \mathbb{N}$ and $m \in (B_{\mathsf{Z}_B})^{k \times l}$ be a matrix. To compute the transposition of m we have used a vector that had as many entries as there are rows in m and every value at a given position was $[]$. In our current notation this is exactly the vector $(\mathsf{Z}_C)_k = []_k$. Now let $j \in \mathbb{N}_{<k}$. We calculate the j-th component of $[]_k \odot_{p,\mu} m$ as follows:

$$\begin{aligned}
& \left([]_k \odot_{p,\mu} m\right)_j \\
= {} & \wr \text{ explicit projection } \int \\
& \mathrm{pr}_j\left([]_k \odot_{p,\mu} m\right) \\
= {} & \wr \text{ Definition 4.7.5 and inlining } \mathsf{Z}_C = [] \int \\
& \mathrm{pr}_j\left(\sum\nolimits_{+_{\mathsf{exA}(p,[])},[]_k} \left[\mathsf{sm}\left(\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, [])\right)\ i\ \left([]_k\right)_i\ m_i \;\middle|\; i \leftarrow [0, \ldots, k-1]\right]\right) \\
= {} & \wr \text{ definition of } []_k \int \\
& \mathrm{pr}_j\left(\sum\nolimits_{+_{\mathsf{exA}(p,[])},[]_k} \left[\mathsf{sm}\left(\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, [])\right)\ i\ []\ m_i \;\middle|\; i \leftarrow [0, \ldots, k-1]\right]\right) \\
= {} & \wr \text{ by Definition 4.7.2.(2) } \int \\
& \mathrm{pr}_j\left(\sum\nolimits_{+_{\mathsf{exA}(p,[])},[]_k} \left[\left(t \mapsto \mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, [])\ i\ []\ (m_i)_t\right) \;\middle|\; i \leftarrow [0, \ldots, k-1]\right]\right) \\
= {} & \wr \text{ since } (m_i)_t = m_{i,t} \int
\end{aligned}$$

$$
\begin{aligned}
& \mathrm{pr}_j \left(\sum\nolimits_{+_{\mathsf{exA}(p,[])},[]_k} \left[\; (t \mapsto \mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, [])\; i\; []\; m_{i,t}) \;\middle|\; i \leftarrow [0, \dots, k-1] \right] \right) \\
= \; & \wr \text{ by Proposition 4.7.4 } \int \\
& \sum\nolimits_{\mathsf{exA}(p,[]),[]} \mathsf{map}\; \mathrm{pr}_j \left[\; (t \mapsto \mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, [])\; i\; []\; m_{i,t}) \;\middle|\; i \leftarrow [0, \dots, k-1] \right] \\
= \; & \wr \text{ since } \mathsf{map}\; g\; [\, f\; x \mid x \leftarrow xs \,] = [\, g\; (f\; x) \mid x \leftarrow xs \,] \int \\
& \sum\nolimits_{\mathsf{exA}(p,[]),[]} \left[\mathrm{pr}_j \; (t \mapsto \mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, [])\; i\; []\; m_{i,t}) \;\middle|\; i \leftarrow [0, \dots, k-1] \right] \\
= \; & \wr \text{ definition of } \mathrm{pr}_j \int \\
& \sum\nolimits_{\mathsf{exA}(p,[]),[]} \left[\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, [])\; i\; []\; m_{i,j} \;\middle|\; i \leftarrow [0, \dots, k-1] \right] \\
= \; & \wr \text{ by Definition 4.7.1.(2) and } [] \neq \mathsf{Z}_A \int \\
& \sum\nolimits_{\mathsf{exA}(p,[]),[]} [\, \mathsf{if}\; m_{i,j} \neq \mathsf{Z}_B \;\mathsf{then}\; \mu\; i\; []\; m_{i,j} \;\mathsf{else}\; [] \mid i \leftarrow [0, \dots, k-1] \,] \\
= \; & \wr \text{ definition of } \mu \text{ and } x : [\,] = [x] \int \\
& \sum\nolimits_{\mathsf{exA}(p,[]),[]} [\, \mathsf{if}\; m_{i,j} \neq \mathsf{Z}_B \;\mathsf{then}\; [(i, m_{i,j})] \;\mathsf{else}\; [] \mid i \leftarrow [0, \dots, k-1] \,] \\
= \; & \wr \; \mathsf{exA}(p, []) = +\!\!+ :: (\mathbb{Z} \times A)^* \to (\mathbb{Z} \times A)^* \to (\mathbb{Z} \times A)^* \int \\
& \sum\nolimits_{+\!\!+,[]} [\, \mathsf{if}\; m_{i,j} \neq \mathsf{Z}_B \;\mathsf{then}\; [(i, m_{i,j})] \;\mathsf{else}\; [] \mid i \leftarrow [0, \dots, k-1] \,] \\
= \; & \wr \text{ by Definition 4.7.3 we have } \sum\nolimits_{+\!\!+,[]} = \mathsf{foldr}\; (+\!\!+)\; [] \int \\
& \mathsf{foldr}\; (+\!\!+)\; [] [\, \mathsf{if}\; m_{i,j} \neq \mathsf{Z}_B \;\mathsf{then}\; [(i, m_{i,j})] \;\mathsf{else}\; [] \mid i \leftarrow [0, \dots, k-1] \,] \\
= \; & \wr \; \mathsf{foldr}\; (+\!\!+)\; [] \text{ is the concat function } \int \\
& \mathsf{concat}\; [\, \mathsf{if}\; m_{i,j} \neq \mathsf{Z}_B \;\mathsf{then}\; [(i, m_{i,j})] \;\mathsf{else}\; [] \mid i \leftarrow [0, \dots, k-1] \,] \\
= \; & \wr \text{ concatenating empty lists has no effect } \int \\
& \mathsf{concat}\; [\, [(i, m_{i,j})] \mid i \leftarrow [0, \dots, k-1] \wedge m_{i,j} \neq \mathsf{Z}_B \,] \\
= \; & \wr \text{ concatenating singleton lists } \int \\
& \left[\; (i, m_{i,j}) \;\middle|\; i \leftarrow [0, \dots, k-1] \wedge m_{i,j} \neq \mathsf{Z}_B \right] .
\end{aligned}
$$

Thus $([]_k \odot_{p,\mu} m)_j$ is an association list of indices i, such that $m_{i,j} \neq \mathsf{Z}_B$ or, in other words, those indices i that have an entry at position $m_{i,j}$. But this list is exactly the reduced[22] version of the list $[\, (i, m_{i,j}) \mid i \leftarrow [0, \dots, k-1] \,]$, which in turn is simply the j-th column of m with additional indices. Note that the list is in fact already sorted in ascending order of the indices. Thus wrapping every list in $[]_k \odot_{p,\mu} m$ in a *Vec* wrapper and interpreting the resulting vector as a matrix is in fact the transposition of m in the mathematical sense.

In our implementation we used a finishing step, where we added a certain vector to the above result. The reason why this is necessary in the application, but not necessary here is the difference between the two scalar multiplications. The scalar multiplication in the implementation removes zero values, while in the approach

[22] "Reduced" means "no designated zeroes occur" in this context.

above, it does not. Thus, as expected, every vector in the intermediate list has length l and every index in the range from 0 to $l-1$ is explicitly filled with a value, even if it is the zero value.

The above computation and the underlying definitions are clearly technical. In particular, one has to carefully distinguish the zero values in the different sets. For instance, in the above case we have taken $\mathsf{Z}_C = []$, but Z_A and Z_B were newly introduced elements. We did this for the sake of convenience only, because $\mathsf{exA}(+\!\!+, [])$ is the same as $(+\!\!+)$ on a different set. We could have chosen a new value for Z_C as well, which would have been ignored by $\mathsf{exA}(+\!\!+, \mathsf{Z}_C)$ and thus by $\sum_{\mathsf{exA}(p,\mathsf{Z}_C),\mathsf{Z}_C}$ as well. However, despite the technicality, we could in fact argue with purely equational reasoning and some more or less simple rules concerning some functions. This kind of proof has the advantage of an algebraic look-and-feel and in fact, many of the arguments in the above computation are motivated by well-known algebraic laws.

We have focussed on one example to show how to reason about results of a general vector-matrix multiplication. Similar arguments are possible for other applications, in particular those from Section 4.3, as well. The basic ingredients for this kind of algebraically flavoured reasoning is translating the Haskell approach to a proper mathematical model and then using known rules to reason about the latter. We have sketched above, how to accomplish this translation and reasoning. One important consequence of this translation is that there is only little difference between the Haskell approach and the mathematical one. This similarity comes with the great advantage that mathematical ideas are easy to implement and implementation features can be reconstructed mathematically. With this in mind, we can use all the properties of vectors and matrices for functional solutions for graph problems.

4.8 Related Work and Discussion

Our focus in this chapter was on graph-related problems that can be expressed in terms of immediate or iterated successors of a certain set of vertices. There are other works that deal with graph problems in this sense. The work of King and Launchbury [KL95] studies functions on graphs that can be expressed through applications of depth-first searches. While some of the functions deal with sets of vertices, there is no framework for an abstraction of the vector-matrix multiplication. The seminal work of Erwig [Erw01] deals with many graph functions, in particular breadth-first and depth-first search, but no abstraction for the collection of information along this search is available. The work of Kashiwagi and Wise [KW91] models graphs as matrices and sets of vertices as vectors over the Booleans, but this is accomplished in a dense setting. On the one hand this model allows very simple functions (for example, the addition of vectors is *zipWith* $(\vee)$), but at the same time comes at the cost of no distinction between $|E|$ and $|V|^2$, which leads to higher complexities. There is some slight similarity to the observation of Elliott [Ell12], but the applications go out

in two very different areas, since the above work focusses on algebraic applications. As we have mentioned before, there is a slight overlap with the work of Dolan [Dol13], but there are two key differences: Dolan deals with star algorithms on graphs and the graphs are modelled as matrices over Kleene algebras only. While star closures can be used to compute the vector of all reachable vertices from some vertex set v as $v \cdot E^\star$, the actual computation of $E^\star$ is cubic in the number of vertices. Thus the computation of all reachable vertices from v, namely $v \cdot E^\star$, has a cubic complexity as well[23]. We have also considered the computation of $v \cdot E^\star$, but in our approach this value is obtained from intermediate steps that do not contain an explicit computation of $E^\star$ and is at worst quadratic in the number of vertices. Also, only little of our approach depends on an underlying semiring and most functions are more general. Finally, the Haskell library *graphs* by Kmett [Kme] provides a very parametric view on graphs and differentiates between matrices in the mathematical sense and adjacency lists. This library provides some elegant functions, but, again, no computation of successors along with additional information is considered.

We have shown how to abstract the notion of vector-matrix multiplications in a rather convenient sense to solve graph problems. The approach is very generic and is essentially based on the notion of a vector sum and a scalar multiplication. Since many applications are based on very specific versions of the above functions, we have also provided generators that allow for a simple creation of the sum and the scalar multiplication. Additionally, we have provided several example applications, where the ones from Section 4.3.5 and Section 4.5 were non-trivial. Finally, we have discussed how one can abstract our concrete containers in terms of type classes and how to reason about the correctness of the code.

Our approach to solving this type of graph problems comes with an algebraic flavour, while at the same time being more general than in case of an actual semiring context. In particular, all semiring-based applications can be easily recovered in our approach. We focussed on those computational components that are actually necessary for the concrete problem at hand, while a semiring approach may be too demanding in terms of more rules and more constants. The non-trivial applications also demonstrate that even technical and seemingly more difficult problems can be solved rather simply through an application of a proper multiplication scheme.

[23] We have discussed these complexity issues in Section 4.4 in detail.

Chapter 5

Bipartite Maximum Matching

The maximum matching problem in bipartite graphs is a natural and well-studied problem. The results of this section have been previously published, while solved from two different perspectives: the purely functional one (published at the RAMiCS 2012 conference [Dan12]) and the functional logic one (published at WLP/WFLP 2014 workshop [Dan14a]). In this section we focus on the functional solution and discuss the functional logic one briefly in Section 9.2.2. The presented solution differs from the one in the original work both in terms of proofs and implementation, but only due to refactoring reasons, whereas the general ideas are exactly the same.

To the best of our knowledge our solution in Haskell was novel at the time of the writing of the first result [Dan12]. However, it is a well-known fact that the maximum matching problem can be solved using the maximum flow approach[1] [KT06] and Erwig and Miljenovic [EM14] provide an implementation of a maximum flow function in their Haskell library *fgl*. Thus it is possible to solve the maximum matching problem using the Haskell library *fgl* as well.

5.1 Specification and Solution

We begin with the definition of a matching.

5.1.1 Definition (Matching, matching number)**.**
Let $G = (V, E)$ be a finite symmetric graph and $M \subseteq E$. Then we define

$$M \text{ is a } \textit{matching} :\Longleftrightarrow M \subseteq \bar{\mathsf{I}} \wedge M = M^{\top} \wedge M \cdot M \subseteq \mathsf{I} \,.$$

We denote

$$\mathsf{matching}(G) := \max \left\{ |D| \;\middle|\; D \in 2^{E} \wedge D \text{ matching} \right\}$$

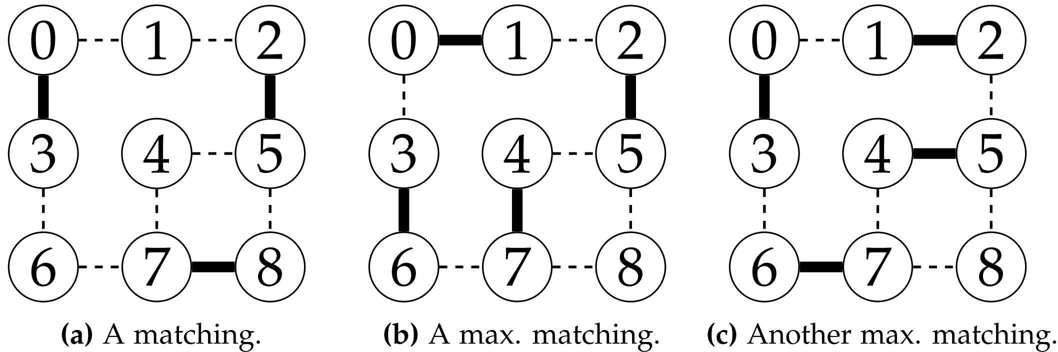
and call this number the *matching number of* G. M is called *maximum matching in* G if and only if $|M| = \mathsf{matching}(G)$ holds. ⫽

Thus a matching M is a loop-free, symmetric, and functional relation that is contained in E. This can be restated in a more illustrating fashion: each two edges in M that do not connect the same two vertices do not share a common vertex.

The matching number is well-defined, because the empty relation is obviously a matching in any graph. A maximum matching is a maximal element of the set of

[1] We discuss this briefly in Section 7.1.

all matchings in G with respect to cardinality. Clearly, maximum matchings are not necessarily unique, as is shown in Figure 5.1, where the bold lines represent edges that belong to the matching and the dashed ones represent non-matching edges.



(a) A matching. **(b)** A max. matching. **(c)** Another max. matching.

Figure 5.1. Matchings in a graph (undirected edges represent edges in a symmetric graph).

The maximum matching problem is of particular interest in case of *bipartite* graphs, which are graphs that allow a bipartition: a partition into two sets, such that all edges of the graph connect vertices from two different sets. Relationally, such a pair of sets can be described elegantly as follows.

5.1.2 Definition (Bipartition)**.**
Let $G = (V, E)$ be a graph and $a, b \subseteq V$. We define

$$(a, b) \text{ is a } \textit{bipartition of } G \;:\Longleftrightarrow\; a = \overline{b} \;\wedge\; (a \cdot E \cap a) \cup (b \cdot E \cap b) = \mathsf{O}\,. \qquad /\!\!/$$

The first condition is equivalent to

$$a \cup b = \mathsf{L} \quad \wedge \quad a \cap b = \mathsf{O}$$

and thus to the fact that $\{ a, b \}$ is a partition of $\mathsf{L}_{\mathbb{1},V}$. The second condition is equivalent to

$$a \cdot E \cap a = \mathsf{O} \quad \wedge \quad b \cdot E \cap b = \mathsf{O}\,,$$

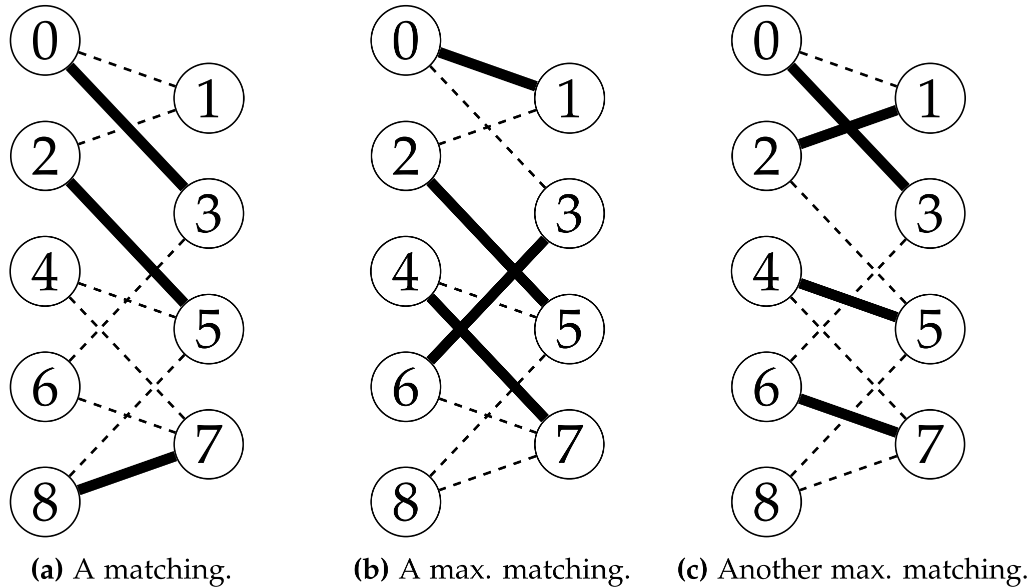
because the least upper bound of two lattice elements is O if and only if both elements are O. For every $s \in 2^V$ the set $s \cdot E$ denotes the successors of s in G and thus due to

$$s \cdot E \cap s = \mathsf{O} \iff s \cdot E \subseteq \overline{s}$$

the equality $s \cdot E \cap s = \mathsf{O}$ is equivalent to all successors of s being contained in $\overline{s}$. In the particular case of the partition $\{ a, b \}$ the above is equivalent to all successors of a being in b and all successors of b being in a, because $\overline{a} = b$. Thus the formula in Definition 5.1.2 in fact coincides with the usual definition of a bipartition.

In case of a bipartite graph one can think of a matching as a one-to-one assignment of objects from the first set to objects from the second one. Practical applications of this approach include the assignment of jobs to people, teachers to classes or, more classically, spouses to one another. The example matchings from Figure 5.1 are shown in the rearranged format in Figure 5.2, respectively. The graph above can be

bipartitioned into the sets $\{0,2,4,6,8\}$ and $\{1,3,5,7\}$. While this is easily checked in this special case, we discuss the general computation of a possible bipartition in Section 5.5. The condition that a graph is bipartite is known to be equivalent to the



(a) A matching. **(b)** A max. matching. **(c)** Another max. matching.

Figure 5.2. Matchings in a bipartitioned graph.

graph not containing any odd cycles by the theorem of Kőnig [Die00]. This condition can be rewritten as

$$E \cdot (E \cdot E)^{\star} \subseteq \bar{\mathsf{I}}$$

in relational terms. The above inclusion can be checked using the star closure operation and relational composition. For now, we will consider only bipartite graphs as input. However, in Section 5.5 we deal with efficiently checking whether a graph is symmetric and bipartite as well as finding a bipartition in case a bipartition exists. Note that any bipartite graph is also loop-free. Thus any edge set $M \subseteq E$ that is symmetric and functional is already a matching.

The key to the solution of the maximal matching problem in the general case is Berge's lemma [Ber57], which provides a characterisation of maximum matchings and also a construction for creating a larger matching from a non-maximum one. This lemma is based upon the existence of a special type of paths.

5.1.3 Definition (Uncovered, alternating, augmenting)**.**
Let $G = (V, E)$ be a finite, symmetric graph and $M \subseteq E$. We set

$$\mathsf{uncovered}(M) := \overline{\{\, w \in V \mid \exists\, v \in V : (v, w) \in M \,\}} = \overline{\mathsf{L}_{\mathbb{1},V} \cdot M}\ .$$

Given $A, B \subseteq E$, a sequence $p \in V^*$ is called *A-B-alternating walk* if and only if

$$\forall\, i \in \mathbb{N}_{<|p|-1} : (i \text{ even } \Rightarrow (p_i, p_{i+1}) \in A) \wedge (i \text{ odd } \Rightarrow (p_i, p_{i+1}) \in B)\ .$$

An A-B-alternating walk that is a path is called A-B-alternating path.

Now assume that M is a matching. A $p \in V^*$ is called *M-augmenting candidate*, if and only if both of the following hold:

(1) $|p| \geqslant 2 \wedge p_0, p_{|p|-1} \in \mathsf{uncovered}(M)$.

(2) p is $(E \setminus M)$-M-alternating.

An M-augmenting candidate that is a path is called M-augmenting path. ⫽

Note that alternating walks are in fact walks in a graph, because each subsequent pair of vertices along the path is contained in either A or B, which are both subsets of E. In less technical terms, an M-augmenting candidate is a walk that begins and ends in a vertex that is not covered by M and whose edges alternate between those in $E \setminus M$ and those in M. We do not call candidates "M-augmenting walks", because a candidate cannot necessarily be used to actually augment the given matching. An example for this is the graph in Figure 5.3, which we discuss in Section 5.2.

Let Δ denote the symmetric difference operation on sets[2]. For every path $p \in V^*$ let $E(p)$ denote the set of undirected edges along this path, which is to say that

$$E(p) = \left\{ (v,w) \in E \;\middle|\; \exists\, i \in \mathbb{N}_{<|p|} : (p_i, p_{i+1}) \in \{ (v,w), (w,v) \} \right\} .$$

With these notations we can state Berge's lemma as follows.

5.1.4 Lemma (Characterisation of maximum matchings, Berge (1957))**.**
Let $G = (V, E)$ be a finite, symmetric graph and $M \subseteq E$ be a matching. Then the following statements hold:

(1) If there are no M-augmenting paths in G, then M is a maximum matching in G.

(2) If there exists an augmenting path p in G, then $M \,\Delta\, E(p)$ is a matching and we have

$$|M \,\Delta\, E(p)| = 2 + |M| .$$

In particular, M is a maximum matching in G, if and only if there are no M-augmenting paths in G. ⫽

We have adjusted Berge's lemma to our setting, because the lemma itself is stated for undirected graphs. Since we model undirected graphs as symmetric directed ones, there are exactly twice as many directed edges as there are undirected ones.

In addition to the actual characterisation of a maximum matching, we also obtain a simple algorithm for finding such a matching. Beginning with the empty matching, we search for an augmenting path. If such a path exists, we create a matching with one edge more than the current one, otherwise we know that we have found a maximum matching. Since G is assumed to be finite, this procedure terminates after at most $|V|/2$ iterations, since each time we create a larger matching, two new vertices are covered. The main ingredient in this process is the search for augmenting paths, which can be realised rather simply with our means.

[2]That is $A \,\Delta\, B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$.

5.2 Searching for Augmenting Paths

We now aim for an algebraic description of the existence of augmenting paths. As it turns out, the resulting expression can be interpreted as an application of reachability from a given vertex set. We can thus use the technique (and functions) from Section 4.4 to compute this reachability. One important benefit of this approach is that we can use a special type of vector-matrix multiplication to check the existence of augmenting paths and return such a path in the positive case at the same time.

We begin with some properties of alternating walks and augmenting candidates. First, alternating paths have certain composition and decomposition properties: subpaths of alternating paths are alternating as well (but possibly in another order) and the concatenation of alternating paths is again alternating under certain conditions.

5.2.1 Proposition (Properties of alternating walks).
Let $G = (V, E)$ be a finite symmetric graph and $A, B \subseteq E$. Let $p, q \in V^*$ be A-B-alternating walks. Then the following hold:

(1) We have

$$\forall\, i, j \in \mathbb{N}_{<|p|} : i \leqslant j \Rightarrow (p_k)_{k=i}^{j} \text{ is } \begin{cases} A\text{-}B\text{-alternating} & : i \text{ even} \\ B\text{-}A\text{-alternating} & : i \text{ odd}\,. \end{cases}$$

(2) We have the following implication:

$$p = () \vee q = () \vee \Big(p \neq () \neq q \wedge |p| \text{ even} \wedge (p_{|p|-1}, q_0) \in B\Big)$$
$$\Longrightarrow p +\!\!+\, q \text{ is an } A\text{-}B\text{-alternating walk}\,.$$ ⫽

We prove this statement in Proof A.2.1. One key observation concerning augmenting candidates is that these are walks that traverse two different graphs in alternating sequence: the even edges are edges of $E \setminus M$, while the odd ones are contained in M, where even and odd refer to the positions in the edge sequence of a walk. Thus every augmenting candidate has even length and all of its vertices, except for the first and last ones, are covered by M.

5.2.2 Lemma (Properties of augmenting candidates).
Let $G = (V, E)$ be a finite, symmetric graph, $M \subseteq E$ be a matching and $p \in V^*$ be an M-augmenting candidate. Then the following hold:

(1) $|p|$ is even.

(2) $\forall\, i \in \mathbb{N}_{<|p|} : p_i \in \mathsf{uncovered}(M) \Leftrightarrow i \in \{\, 0, |p| - 1 \,\}$. ⫽

A proof of the above statements is provided in Proof A.2.2. This lemma has an important consequence in the case of bipartite graphs.

5.2.3 Lemma (Shortcuts in augmenting candidates)**.**
Let $G = (V, E)$ be a finite, symmetric and bipartite graph, $M \subseteq E$ be a matching and $p \in V^*$ be an M-augmenting candidate that is not a path. Then there exist $s, t \in \mathbb{N}_{<|p|}$ such that $0 < s < t < |p| - 1$, $p_s = p_t$ and

$$(p_k)_{k=0}^{s} +\!\!+ (p_k)_{k=t+1}^{|p|-1}$$

is an M-augmenting candidate. In particular, there exists an M-augmenting path in G. ⫽

We prove Lemma 5.2.3 in Proof A.2.3. The above statement is not true in graphs that are not bipartite. Consider for example the graph in Figure 5.3, where the bold lines indicate the edges of the matching M and the dashed lines are non-matching edges. The sequence $p = (0, 1, 2, 3, 4, 2, 1, 5)$ is an M-augmenting candidate, but not a

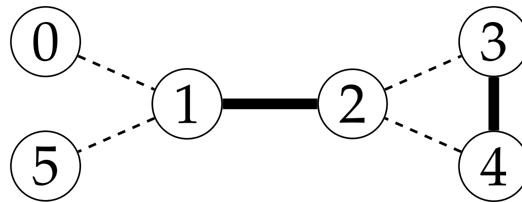


Figure 5.3. A matching in a non-bipartite graph.

path, because the vertices $1, 2$ occur twice. We have that $p_2 = 2 = p_5$, but the walk

$$(p_k)_{k=0}^{2} +\!\!+ (p_k)_{k=6}^{7} = (0, 1, 2, 1, 5)$$

is not alternating, since the edges $(1, 2)$ and $(2, 1)$ are both contained in M. We also have $p_1 = p_6$, but the path

$$(p_k)_{k=0}^{1} +\!\!+ (p_k)_{k=7}^{7} = (0, 1, 5)$$

is also non-alternating, because both of its edges are contained in $E \setminus M$. Thus it is not possible to create an augmenting path from this augmenting candidate. In fact, there are no augmenting paths in the above graph.

The previous results are not particularly deep or difficult, but tend to get rather technical. However, they can be used to obtain a relational specification, which we in turn can use to implement the path search using the means we have developed in Chapter 4. We state this characterisation as follows.

5.2.4 Proposition (Characterisation of the existence of augmenting paths)**.**
Let $G = (V, E)$ be a finite, symmetric graph and $M \subseteq E$ be a matching. Then the following statements are equivalent[3]*:*

(1) There exists an M-augmenting candidate in G.

(2) $\mathsf{uncovered}(M) \cdot ((E \setminus M) \cdot M)^{\star} \cdot (E \setminus M) \cap \mathsf{uncovered}(M) \neq \mathsf{O}$.

[3]The relational composition operation $\cdot$ is overloaded as described after Definition 2.4.3.

If, additionally, G is bipartite, then Statement (2) is also equivalent to the following statement:

(3) There exists an M-augmenting path in G.

Proof. Let $u := \mathsf{uncovered}(M)$. We prove the equivalence "(2) $\iff$ (1)" as follows:

$$u \cdot ((E \setminus M) \cdot M)^\star \cdot (E \setminus M) \cap u \neq \mathsf{O}$$
$\iff$ ≀ being non-empty ≀
$$\exists\, y : y \in u \cdot ((E \setminus M) \cdot M)^\star \cdot (E \setminus M) \cap u$$
$\iff$ ≀ definition of the intersection and abbreviating with quantifiers ≀
$$\exists\, y \in u : y \in u \cdot ((E \setminus M) \cdot M)^\star \cdot (E \setminus M)$$
$\iff$ ≀ definition of the relational composition ≀
$$\exists\, y \in u : \exists\, x \in V : x \in u \wedge (x, y) \in ((E \setminus M) \cdot M)^\star \cdot (E \setminus M)$$
$\iff$ ≀ since $u \subseteq V$ we have $x \in V \wedge x \in u \iff x \in u$ ≀
$$\exists\, y \in u : \exists\, x \in u : (x, y) \in ((E \setminus M) \cdot M)^\star \cdot (E \setminus M)$$
$\iff$ ≀ definition of relational composition ≀
$$\exists\, y \in u : \exists\, x \in u : \exists\, z \in V : (x, z) \in ((E \setminus M) \cdot M)^\star \wedge (z, y) \in E \setminus M$$
$\iff$ ≀ $(a, b) \in R^\star \iff \exists\, w \in V^+ : w$ is a walk in $R \wedge w_0 = a \wedge w_{|w|-1} = b$ ≀
$$\exists\, y \in u : \exists\, x \in u : \exists\, z \in V : (z, y) \in E \setminus M$$
$$\wedge \left(\exists\, w \in V^+ : w_0 = x \wedge w_{|w|-1} = z \wedge w \text{ is a walk in } (E \setminus M) \cdot M\right)$$
$\iff$ ≀ distributive law, w is not free in $(z, y) \in E \setminus M$ ≀
$$\exists\, y \in u : \exists\, x \in u : \exists\, z \in V : \exists\, w \in V^+ :$$
$$(z, y) \in E \setminus M \wedge w_0 = x \wedge w_{|w|-1} = z \wedge w \text{ is a walk in } (E \setminus M) \cdot M$$
$\iff$ ≀ rearranging quantifiers and the conjunction parts ≀
$$\exists\, w \in V^+ : \exists\, y \in u : \exists\, z \in V : \exists\, x \in u :$$
$$w_0 = x \wedge w_{|w|-1} = z \wedge (z, y) \in E \setminus M \wedge w \text{ is a walk in } (E \setminus M) \cdot M$$
$\iff$ ≀ distributive law, x is not free in the term after $w_0 = x$ ≀
$$\exists\, w \in V^+ : \exists\, y \in u : \exists\, z \in V : (\exists\, x \in u : w_0 = x)$$
$$\wedge\, w_{|w|-1} = z \wedge (z, y) \in E \setminus M \wedge w \text{ is a walk in } (E \setminus M) \cdot M$$
$\iff$ ≀ removing existential quantification by inlining, Equation (5.2.4.a) ≀
$$\exists\, w \in V^+ : \exists\, y \in u : \exists\, z \in V : w_0 \in u$$
$$\wedge\, w_{|w|-1} = z \wedge (z, y) \in E \setminus M \wedge w \text{ is a walk in } (E \setminus M) \cdot M$$
$\iff$ ≀ rearranging the parts of the conjunction ≀
$$\exists\, w \in V^+ : \exists\, y \in u : \exists\, z \in V : w_{|w|-1} = z \wedge (z, y) \in E \setminus M$$
$$\wedge\, w_0 \in u \wedge w \text{ is a walk in } (E \setminus M) \cdot M$$
$\iff$ ≀ distributive law, z is not free in the formula in the second line ≀
$$\exists\, w \in V^+ : \exists\, y \in u : \left(\exists\, z \in V : w_{|w|-1} = z \wedge (z, y) \in E \setminus M\right)$$

$$
\begin{aligned}
&\qquad \wedge\, w_0 \in u \wedge w \text{ is a walk in } (E \setminus M) \cdot M \\
\iff & \quad \wr \text{ removing existential quantification by inlining, Equation (5.2.4.a) } \wr \\
& \exists\, w \in V^+ : \exists\, y \in u : \left(w_{|w|-1}, y\right) \in E \setminus M \wedge w_0 \in u \wedge w \text{ is a walk in } (E \setminus M) \cdot M \\
\iff & \quad \wr \text{ extending } w \text{ by the edge } (w_{|w|-1}, y) \text{ and adjusting indices } \wr \\
& \exists\, s \in V^+ : |s| \geqslant 2 \wedge \Big(\exists\, y \in u : y = s_{|s|-1} \wedge \left(s_{|s|-2}, y\right) \in E \setminus M \\
&\qquad \wedge\, s_0 \in u \wedge (s_k)_{k=0}^{|s|-2} \text{ is a walk in } (E \setminus M) \cdot M\Big) \\
\iff & \quad \wr \text{ distributive law, } y \text{ is not free in the formula in the second line } \wr \\
& \exists\, s \in V^+ : |s| \geqslant 2 \wedge \left(\exists\, y \in u : y = s_{|s|-1} \wedge \left(s_{|s|-2}, y\right) \in E \setminus M\right) \\
&\qquad \wedge\, s_0 \in u \wedge (s_k)_{k=0}^{|s|-2} \text{ is a walk in } (E \setminus M) \cdot M \\
\iff & \quad \wr \text{ removing existential quantification by inlining } \wr \\
& \exists\, s \in V^+ : |s| \geqslant 2 \wedge s_{|s|-1} \in u \wedge \left(s_{|s|-2}, s_{|s|-1}\right) \in E \setminus M \\
&\qquad \wedge\, s_0 \in u \wedge (s_k)_{k=0}^{|s|-2} \text{ is a walk in } (E \setminus M) \cdot M \\
\iff & \quad \wr \text{ definition of walk, Definition 2.3.10 } \wr \\
& \exists\, s \in V^+ : |s| \geqslant 2 \wedge s_{|s|-1} \in u \wedge \left(s_{|s|-2}, s_{|s|-1}\right) \in E \setminus M \\
&\qquad \wedge\, s_0 \in u \wedge \left(\forall\, i \in \mathbb{N}_{<|s|-2} : (s_i, s_{i+1}) \in (E \setminus M) \cdot M\right) \\
\iff & \quad \wr \text{ definition of relational composition } \wr \\
& \exists\, s \in V^+ : |s| \geqslant 2 \wedge s_{|s|-1} \in u \wedge \left(s_{|s|-2}, s_{|s|-1}\right) \in E \setminus M \wedge s_0 \in u \\
&\qquad \wedge \left(\forall\, i \in \mathbb{N}_{<|s|-2} : \exists\, m_i \in V : (s_i, m_i) \in (E \setminus M) \wedge (m_i, s_{i+1}) \in M\right) \\
\iff & \quad \left\wr \begin{array}{l} M \text{ is functional and symmetric, thus a partial involution. Thus} \\ (m_i, s_{i+1}) \in M \iff (m_i, s_{i+1}) \in M \wedge m_i = M^{-1}(s_{i+1}) = M(s_{i+1}). \end{array} \right\wr \\
& \exists\, s \in V^+ : |s| \geqslant 2 \wedge s_{|s|-1} \in u \wedge \left(s_{|s|-2}, s_{|s|-1}\right) \in E \setminus M \wedge s_0 \in u \\
&\qquad \wedge \left(\forall\, i \in \mathbb{N}_{<|s|-2} : \exists\, m_i \in V : (s_i, m_i) \in E \setminus M \wedge (m_i, s_{i+1}) \in M \wedge m_i = M(s_{i+1})\right) \\
\iff & \quad \wr \text{ removing existential quantification by inlining } \wr \\
& \exists\, s \in V^+ : |s| \geqslant 2 \wedge s_{|s|-1} \in u \wedge \left(s_{|s|-2}, s_{|s|-1}\right) \in E \setminus M \wedge s_0 \in u \\
&\qquad \wedge \left(\forall\, i \in \mathbb{N}_{<|s|-2} : (s_i, M(s_{i+1})) \in E \setminus M \wedge (M(s_{i+1}), s_{i+1}) \in M\right) \\
\iff & \quad \wr \text{ see below, } (\star) \wr \\
& \exists\, r \in V^+ : |r| \geqslant 2 \wedge r_0 \in u \wedge r_{|r|-1} \in u \wedge r \text{ is } (E \setminus M)\text{-}M\text{-alternating} .
\end{aligned}
$$

The inlining transformation is based upon the fact that for every term t such that $x \notin \text{variables}(t)$ and every formula φ we have

$$\exists\, x : x = t \wedge \varphi \iff \varphi[x/t] , \tag{5.2.4.a}$$

where $\varphi[x/t]$ denotes the formula φ with every free occurrence of x replaced by t. This rule is a consequence of the coincidence lemma. Thus an existential quantification where the quantified variable is actually equal to some term can be replaced by inlining the term and removing the existential quantifier.

The equivalence marked with $(\star)$ can be proved as follows.

"$\Longrightarrow$": If $|s| = 2$, then setting $r := s$ is a path as desired. Otherwise $|s| > 2$ and setting

$$r := \left(\begin{cases} s_{\frac{i}{2}} & : i \text{ even} \\ M\left(s_{\frac{i+1}{2}}\right) & : \text{otherwise} \end{cases}\right)_{i=0}^{2(|s|-1)}$$

yields a desired path.
"$\Longleftarrow$": If r is such a path and $i \in \mathbb{N}_{<|r|-1}$ is odd, then $(r_i, r_{i+1}) \in M$ and thus $r_i = M(r_{i+1})$. Also, $\left(r_{|r|-2}, r_{|r|-1}\right) \in E \setminus M$, since it leads to the uncovered vertex $r_{|r|-1} \in u$. ⫽

This concludes the proof of "(2) $\iff$ (1)". Now suppose that G is bipartite. Then we get the following chain of equivalences:

$$
\begin{aligned}
& u \cdot ((E \setminus M) \cdot M)^\star \cdot (E \setminus M) \cap u \neq \mathsf{O} \\
\iff\ & \wr \text{ since } (1) \iff (2)\ \wr \\
& \exists\, r \in V^\star : |r| \geqslant 2 \wedge r_0 \in u \wedge r_{|r|-1} \in u \wedge r \text{ is } (E \setminus M)\text{-}M\text{-alternating} \\
\iff\ & \wr\ \text{"}\Longrightarrow\text{": By Lemma 5.2.3 there exists an augmenting path. "}\Longleftarrow\text{": Clearly, paths are also walks.}\ \wr \\
& \exists\, r \in V^\star : |r| \geqslant 2 \wedge r_0 \in u \wedge r_{|r|-1} \in u \wedge r \text{ is } (E \setminus M)\text{-}M\text{-alternating} \wedge r \text{ is a path}\,.
\end{aligned}
$$

Thus (2) $\iff$ (3) holds. ⫽

Note that the equivalence "(2) $\iff$ (3)" is not true in non-bipartite graphs, because the last equivalence in its proof is not true in this case. While the existence of an augmenting path clearly yields the existence of an augmenting walk, the converse is not true, as we have seen after Lemma 5.2.3. Thus in case of a non-bipartite graph only the direction (3) $\Longrightarrow$ (2) of Proposition 5.2.4 holds. While still useful, this statement only shows how to look for an augmenting path, but not whether such a path exists or not.

In algebraic terms, the set $\mathsf{uncovered}(M)$ can be represented by a vector and then the value $\mathsf{uncovered}(M) \cdot ((E \setminus M) \cdot M)^\star$ is computable as an instance of our reachability function *reachableWith* from Section 4.4. Then this intermediate term needs to be multiplied with $E \setminus M$ and intersected with $\mathsf{uncovered}(M)$ to get the necessary term

$$\mathsf{uncovered}(M) \cdot ((E \setminus M) \cdot M)^\star \cdot (E \setminus M) \cap \mathsf{uncovered}(M)\,.$$

We can improve the computation by explicitly avoiding the union of all reachability steps, which may require more computations than necessary. Recall that in every Kleene algebra K we have that

$$\forall\, a, b, x \in K : a \cdot x = x \cdot b \Rightarrow a^\star \cdot x = x \cdot b^\star$$

by Proposition 3.2.3.(4). In particular, for all $r, s \in K$ we can set $a := r \cdot s$, $b := s \cdot r$ and

$x := r$. Then we find
$$a \cdot x = (r \cdot s) \cdot r = r \cdot (s \cdot r) = x \cdot b$$
by the associativity of the multiplication. Thus the above equation yields
$$(r \cdot s)^\star \cdot r = a^\star \cdot x = x \cdot b^\star = r \cdot (s \cdot r)^\star \ .$$
Since relation algebras are Kleene algebras (cf. Example 3.2.2.(1)), we can apply this result to relations as well. In the above case we thus get that
$$((E \setminus M) \cdot M)^\star \cdot (E \setminus M) = (E \setminus M) \cdot (M \cdot (E \setminus M))^\star \ .$$
Thus the existence of an augmenting path is equivalent to the condition
$$\mathsf{uncovered}(M) \cdot (E \setminus M) \cdot (M \cdot (E \setminus M))^\star \cap \mathsf{uncovered}(M) \neq \mathsf{O} \ .$$
This rearrangement is now an instance of the pattern
$$v \cdot R^\star \cap w \ ,$$
which we have discussed in Section 4.4 (namely in Equation (4.4.ii)). Since we are interested in any path that leads from the initial set represented by the vector $\mathsf{uncovered}(M) \cdot (E \setminus M)$ to the set $\mathsf{uncovered}(M)$, we can use any shortest path as well. Using shortest paths comes with the benefit that we can reuse our function *shortestWith* from Section 4.4, which computes a shortest connection from a source vector to a target vector. Additionally, we already know that we can use a special type of multiplication to not only obtain the vertices that are actually reached, but also to collect some information along the way. In this case, we require a path that leads from the first set to the second one. This information can be collected with our function $(\odot_\sim)$ from Section 4.3.4. Note that the result of this information is in fact a path rather than just a walk, because M is a matching and *shortestWith* finds the shortest connection between two vectors. Thus the search for an augmenting path can be computed with the following function, because $(\odot_\sim)$ maintains the relational context as mentioned in Section 4.4 already.

```
augmentingPath :: Mat α → Mat α → Maybe Path
augmentingPath m eNotM = fmap (uncurry (flip (▷))) (maybeSome result) where
  result = shortestWith (⊙~) (fmap (const ⟨⟩) u ⊙~ eNotM) u [m, eNotM]
  u      = uncovered m
```

We assume that the parameter *m* represents the matching M and the parameter *eNotM* represents its relative complement $E \setminus M$. We will see why this is useful in Section 5.3. The local value *result* is the first non-empty intersection of a reachability step with the target vector if such a step exists and the constant *emptyVec* otherwise. If the result is empty, there is no augmenting path in the graph and Berge's lemma yields the maximality of *m*. Otherwise we obtain an augmenting path by simply reading the value at any index in *result*. The function *maybeSome* :: *Vec* $\alpha \to$ *Maybe* (*Arc* α) yields a value that is contained at some position in the vector in case the vector is non-empty and returns *Nothing* otherwise. One possible implementation of *maybeSome* is the

following one, where $listToMaybe :: [\alpha] \to Maybe\ \alpha$ is a Haskell function, such that $listToMaybe\ [\,] = Nothing$ and $listToMaybe\ (x : _) = Just\ x$.

$$maybeSome :: Vec\ \alpha \to Maybe\ (Arc\ \alpha)$$
$$maybeSome = listToMaybe \circ unVec$$

The final step $fmap\ (uncurry\ (flip\ (\rhd)))$ is applied to a $Maybe\ (Arc\ Path)$. If the argument is $Just\ (i, path)$ then the above function simply returns $Just\ (path \rhd i)$, which is the actual augmenting path. Otherwise it returns *Nothing*.

In the above implementation we still require the function *uncovered*, which can be implemented rather simply in the case of symmetric graphs, too. Due to symmetry, being uncovered means the same as having no successors and since the successors are contained in the adjacency list of a vertex, we can simply check whether the corresponding list is empty or not.

$$uncovered :: Mat\ \alpha \to Vec\ ()$$
$$uncovered = void \circ filterVec\ isEmptyVec \circ matrix$$

We simply unwrap the matrix vector and keep only those key-value pairs, where the second component (which is the adjacency list of a vertex) is empty. The function $void :: Functor\ \varphi \Rightarrow \varphi\ \alpha \to \varphi\ ()$ is a Haskell function that maps every value of a structure to $()$. The assumption of symmetry is implicit and has to be checked manually. We have now implemented all necessary functions to search for augmenting paths.

The actual function *augmentingPath* is surprisingly simple, considering that the problem of finding a special path as required is far from trivial. At the same time, it is very compositional in its structure. While we introduced two additional functions for the sake of simplicity, namely *maybeSome* and *uncovered*, these functions, as well as all other functions that were used in the implementation of *augmentingPath* are useful in their own right. This modular approach is common for functional programming, where the solution of a complex problem is usually obtained from the solutions to several simpler problems.

5.3 The Bipartite Maximum Matching Function

Now that we have dealt with the task of finding augmenting paths, we can implement a function that computes a maximum matching in a bipartite graph. To that end we first need a notion of the edges of a graph, to represent $E(p)$ from Lemma 5.1.4. For reasons of both simplicity and efficiency, we represent this set as an auxiliary graph, which we define as follows[4].

type $PlainVec \quad = IntMap\ ()$

[4]Instead of the inner *IntMap* () one can use the specialised data type *IntSet* from the package *containers*, which provides efficient operations on sets bounded integers.

```
type AuxGraph = IntMap PlainVec
```

The idea behind the *PlainVec* type is that it is simple to translate this structure into a special vector and at the same time we can efficiently modify values at certain keys[5]. We assume the same preconditions for *AuxGraph* as we did for *Mat*. In particular, an empty auxiliary graph with n vertices is an *IntMap*, where exactly the keys from $\mathbb{N}_{<n}$ are filled and all values at these keys are the empty *IntMap*.

```
intMapToVec :: IntMap α → Vec α
intMapToVec = Vec ∘ toAscList

toGraph :: AuxGraph → Mat ()
toGraph = Mat ∘ intMapToVec ∘ fmap intMapToVec
```

The empty auxiliary graph can be obtained from a given graph with the following function.

```
emptyAuxGraph :: Mat α → AuxGraph
emptyAuxGraph = fromAscList ∘ unVec ∘ fmap (const empty) ∘ matrix
```

This function unwraps the matrix, maps every value in the resulting vector to the empty *IntMap* and then transforms the vector into an *IntMap*.

Since the auxiliary graph supports fast update operations, we can easily implement a function that adds edges to a graph as follows.

```
addEdge :: Vertex → Vertex → AuxGraph → AuxGraph
addEdge v w = adjust (insert w ()) v

addEdgeSym :: Vertex → Vertex → AuxGraph → AuxGraph
addEdgeSym v w = addEdge w v ∘ addEdge v w
```

The function *adjust* :: $(\alpha \to \alpha) \to Int \to IntMap\ \alpha \to IntMap\ \alpha$ is a predefined Haskell operation on *IntMap*s and applies the given function to the value at the given key. Note that the above function is correct, because we assume that a sufficient range of keys is present in the map.

Now we can transform a path into an auxiliary graph. To that end, we first turn a path into a list of vertices. Zipping this list with its tail results in a list of pairs that denote exactly the (mathematical) edges along the path. These edges are then added to a supplied graph using the function *addEdge* from above.

```
pathToUndirectedGraph :: Path → AuxGraph → AuxGraph
pathToUndirectedGraph p graph = foldr (uncurry addEdgeSym) graph (zip vs (tail vs))
  where vs = toList p
```

The final preparation step is the implementation of the symmetric difference on graphs. Given two graphs, we can apply the function *unionWith* from Section 4.2.1 and Section 4.6 on the outer vector of the graph, to obtain all possible keys from

[5] If *IntMaps* are used for the representation of vectors as we have discussed in Section 4.6, we can omit this transformation and simply use the actual graph data type.

both graphs. The function that we supply to *unionWith* is then the actual symmetric difference on vectors, which can be implemented using a similar merging technique as in the case of *unionWith*. More modularly, we can use the function *mergeWith* and reuse the functions *cmpFst* and *thirdArg* from Section 4.6 for an implementation.

$$
\begin{array}{l}
\mathit{symDifference} :: \mathit{Vec}\ \alpha \to \mathit{Vec}\ \alpha \to \mathit{Vec}\ \alpha \\
\mathit{symDifference}\ (\mathit{Vec}\ v)\ (\mathit{Vec}\ w) = \mathit{Vec}\ (v\ \text{`}\mathit{symDiff}\text{'}\ w)\ \mathbf{where} \\
\quad \mathit{symDiff} = \mathit{mergeWith}\ \mathit{cmpFst}\ \mathit{id}\ \mathit{id}\ (\lambda x\ _\ l \to x : l)\ \mathit{thirdArg}\ (\lambda _\ y\ l \to y : l)
\end{array}
$$

$$
\begin{array}{l}
(\Delta) :: \mathit{Mat}\ \alpha \to \mathit{Mat}\ \alpha \to \mathit{Mat}\ \alpha \\
\mathit{Mat}\ m\ \Delta\ \mathit{Mat}\ n = \mathit{Mat}\ (\mathit{unionWith}\ \mathit{symDifference}\ m\ n)
\end{array}
$$

Our function *augmentingPath* from Section 5.2 takes both the matching and its relative complement in the graph as arguments. Seemingly, this requires a relative complement function on graphs, too. However, this is not the case. Let $(L, \sqcup, \sqcap, \bot, \top)$ be a Boolean algebra and

$$- : L \times L \to L,\ (a, b) \mapsto a \sqcap \overline{b}\ .$$

With this notion of the relative complement, we can define the symmetric union

$$\oplus : L \times L \to L,\ (a, b) \mapsto (a - b) \sqcup (b - a)\ .$$

Recall that the relative complement has the following properties for all $x, y, z \in L$:

$$x - (y - z) = (x - y) \sqcup (x \sqcap z)\ , \tag{5.3.i}$$

$$(x - y) - z = x - (y \sqcup z)\ . \tag{5.3.ii}$$

Both properties are easy to show using properties of Boolean algebras. Now let $a, b, c \in L$ such that $a \sqsubseteq c$. Then the following holds:

$$
\begin{array}{cl}
 & (a - b) \oplus c \\
= & \wr\ \text{definition of symmetric difference}\ \wr \\
 & ((a - b) - c) \sqcup (c - (a - b)) \\
= & \wr\ \text{by Equation (5.3.ii) and Equation (5.3.i)}\ \wr \\
 & (a - (b \sqcup c)) \sqcup ((c - a) \sqcup (c \sqcap b)) \\
= & \wr\ c \sqsubseteq a, \text{ thus } c - a = c \sqcap \overline{a} = \bot\ \wr \\
 & (a - (b \sqcup c)) \sqcup (c \sqcap b) \\
= & \wr\ \text{since } c \sqsubseteq a \text{ we have } c \sqcap a = c\ \wr \\
 & (a - (b \sqcup c)) \sqcup (c \sqcap a \sqcap b) \\
= & \wr\ \text{commutativity and associativity of } \sqcap\ \wr \\
 & (a - (b \sqcup c)) \sqcup (a \sqcap (b \sqcap c)) \\
= & \wr\ \text{by Equation (5.3.i)}\ \wr \\
 & a - ((b \sqcup c) - (b \sqcap c)) \\
= & \wr\ \text{property of the symmetric difference}\ \wr \\
 & a - (b \oplus c)\ .
\end{array}
$$

In the particular case of the graph $G = (V, E)$ we can consider the Boolean algebra $\left(2^V, \cup, \cap, \emptyset, V\right)$. Note that in this algebra we have that "$-$" is the relative complement $\setminus$ and $\oplus$ coincides with the symmetric difference Δ. Given a matching $M \subseteq E$ and an M-augmenting path p we thus have that $E(p) \subseteq E$, which yields

$$E \setminus (M \; \Delta \; E(p)) = (E \setminus M) \; \Delta \; E(p) \; .$$

Thus in every iteration step of the improvement algorithm, both M and $E \setminus M$ are modified in the exact same way. Hence, we do not require an explicit complement operation, because we begin with the empty matching, whose complement in E is E itself and in every iteration step change the matching and its complement using the symmetric difference only.

With these considerations, we can now implement the actual function that augments a matching, if it is not a maximum matching and returns *Nothing* otherwise.

```
augmentMatching :: AuxGraph → Mat () → Mat () → Maybe (Mat (), Mat ())
augmentMatching pg m cm = fmap f (augmentingPath m cm) where
  f path = (m Δ augPath, cm Δ augPath) where
    augPath = toGraph (pathToUndirectedGraph path pg)
```

The function that computes a maximum matching in a bipartite graph is then easy to implement using the function $maybe :: \beta \to (\alpha \to \beta) \to Maybe\ \alpha \to \beta$, which returns the first argument if the *Maybe* value is *Nothing* and returns the second argument applied to x if the *Maybe* value is *Just x*. We implement an intermediate function that returns the matching and its complement, because it is convenient in an application later.

```
maximumMatching :: Mat () → Mat ()
maximumMatching = fst ∘ maximumMatching'

maximumMatching' :: Mat () → (Mat (), Mat ())
maximumMatching' graph = go (emptyMat graph) graph where
  go m cm = maybe (m, cm) (uncurry go) (augmentMatching aux m cm)
  aux     = emptyAuxGraph graph

emptyMat :: Mat α → Mat β
emptyMat = Mat ∘ fmap (const emptyVec) ∘ matrix
```

The auxiliary function *emptyMat* creates an empty matrix with the same number of rows as its argument. Note that the function *maximumMatching* creates only one auxiliary graph *aux* and not one such graph per iteration. The main idea is that as long as *augmentMatching* does not return *Nothing*, we can augment the matching with another call to *augmentMatching*. If on the other hand the result of *augmentMatching* is *Nothing*, then we can return the current matching, because it is already a maximum matching.

The above implementation contains one trivial improvement, which is typically implemented by hand in imperative solutions. Instead of using the empty matching

as a starting point, one can easily implement a (greedy) function that computes a maximal matching, which is a matching that is maximal with respect to inclusion and then use this larger matching as a starting point. This improvement does not change the overall complexity of the algorithm, but improves the constants in the asymptotic analysis. In our case, we are always looking for shortest paths. If there exists a single edge that can be added to a matching, our *augmentMatching* function will in fact find such an edge, because it is a special case of an augmenting path.

5.4 Maximum Matching with Disjoint Paths

In the above approach, we search for a single augmenting path and thus every iteration step increases the size of the matching by one. Each path search has the same complexity as a breadth-first search and there are at most $|V|$ edges in any matching, but since every path search adds exactly two edges, there are at most $|V|/2$ iterations. In case of an imperative algorithm, this yields the complexity estimate $\mathcal{O}(|E| \cdot |V| + |V|)$, where the additive term $|V|$ can be omitted if the graph is non-empty. Our implementation has a cubic complexity in the number of vertices, because the search for a path is quadratic in the number of vertices. This can be optimised to obtain the complexity $\mathcal{O}(|E| \cdot |V| \cdot \log_2(|V|) + |V|)$ using different data structures as discussed in Section 4.6.

There is a faster algorithm for finding maximum matchings in bipartite graphs, which is known as the Hopcroft-Karp algorithm [HK73]. This algorithm is based upon one particularly elegant improvement: instead of searching for just a single shortest augmenting path, one searches for “as many shortest augmenting paths as possible”. Clearly, we cannot just take all shortest augmenting paths at once and use them to improve the matching, because the result may no longer be a matching. An example of this is the graph in Figure 5.4, where the bold edges are edges that belong to the matching M and the dashed ones do not belong to the matching. In this graph

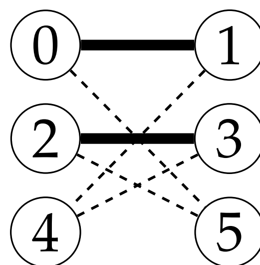


Figure 5.4. Only one path can be used for improvement

the paths $p_1 := (5,0,1,4)$ and $p_2 := (5,2,3,4)$ are both shortest augmenting paths, but the set

$$(M \,\Delta\, E(p_1)) \,\Delta\, E(p_2) = \{\, (0,5), (1,4), (2,5), (3,4), (4,1), (4,3), (5,0), (5,2) \,\}$$

is not a matching. This is due to the fact that the paths are not disjoint. However, using as many shortest *disjoint* paths as possible in every improvement step does not only maintain the matching property of the result, but it also improves the complexity of the algorithm. The Hopcroft-Karp algorithm does precisely that by searching for a set of shortest, pairwise disjoint augmenting paths that is maximal with respect to inclusion. Since it is possible to find such a set in the asymptotically same number of steps as finding a single path [HK73], the overall complexity is improved to $\mathcal{O}(|E| \cdot \sqrt{|V|} + |V|)$.

Recall that we have considered this very problem before in Section 4.5. We obtained the function

$$disjointPaths :: Int \to Vec\ (Forest\ Vertex) \to Vec\ \beta \to [Mat\ \gamma] \to [Path]$$

that takes the size of the graph (for convenience) and computes a list of shortest, pairwise disjoint paths from the source set to the target set, where the source set contains some vertices that are labelled with forests already. In our case, the path search begins after the first step, namely in $\mathsf{uncovered}(M) \cdot (E \setminus M)$. It is therefore convenient to first mark $\mathsf{uncovered}(M)$ with the empty forest at every position and then use the forest multiplication ($\odot_{\Cup}$) from Section 4.5 to collect the information how to get to the set $\mathsf{uncovered}(M) \cdot (E \setminus M)$ from $\mathsf{uncovered}(M)$. Note that the result will still be a set of pairwise disjoint paths, because the pruning takes place after the path search and thus the first step is also taken into account. We implement a function that searches for a maximal set of shortest, pairwise disjoint augmenting paths in a very similar fashion as we did before for a single augmenting path.

```
augmentingPaths :: Int → Mat α → Mat α → [Path]
augmentingPaths n m cm = disjointPaths n (u ⊙⋓ cm) u [m, cm]
  where u = fmap (const []) (uncovered m)
```

With this function at hand it is very simple to obtain a functional variant of the Hopcroft-Karp algorithm. All we need to do is to compute the size of the graph and replace the insertion of a single path into the auxiliary graph by the insertion of a list of paths. The latter task is easily implemented using a fold.

```
pathsToUndirectedGraph :: [Path] → AuxGraph → AuxGraph
pathsToUndirectedGraph ps graph = foldr pathToUndirectedGraph graph ps
```

This auxiliary function allows to augment a given matching as follows.

```
augmentMatchingHK ::
  Int → AuxGraph → Mat () → Mat () → Maybe (Mat (), Mat ())
augmentMatchingHK n pg m cm
   | null paths = Nothing
   | otherwise  = Just (m Δ augPaths, cm Δ augPaths)
  where paths    = augmentingPaths n m cm
        augPaths = toGraph (pathsToUndirectedGraph paths pg)
```

Note that except for the fact that we produce *Maybe* values manually, this is essentially the very same function as *augmentMatching*. Finally, the improved function for finding maximum matchings in finite, symmetric and bipartite graphs can be implemented as follows.

```
maximumMatchingHK :: Mat () → Mat ()
maximumMatchingHK = fst ∘ maximumMatchingHK'

maximumMatchingHK' :: Mat () → (Mat (), Mat ())
maximumMatchingHK' graph = go (emptyMat graph) graph where
  go m cm = maybe (m, cm) (uncurry go) (augmentMatchingHK n aux m cm)
  aux     = emptyAuxGraph graph
  n       = size aux
```

The function *size* :: *IntMap* α → *Int* is a predefined Haskell function, which returns the number of elements in the given *IntMap*. Again, the function *maximumMatchingHK* is essentially the same function as *maximumMatching*, except for some replacements and one additional local argument (the size of the graph). These simple replacements are a particularly good example of the modularity of a purely functional approach.

5.5 Testing Symmetry and Bipartiteness

While we searched for maximum matchings in Section 5.2 and Section 5.4, we have assumed the underlying graph to be symmetric and bipartite. Clearly, this is very convenient from an implementational point of view. However, it is also dangerous to expose functions that require certain preconditions, without some kind of verification, whether these conditions are satisfied or not. For the sake of convenience, one typically provides certain predicates that can be used to test the given conditions and allows the user to use the corresponding functions anyway. A more rigorous, but also more complex, approach is to explicitly wrap the actual functions in such tests. For instance, searching for maximum matchings in bipartite graphs can look as follows[6].

```
maximumMatchingSafe :: Mat () → Maybe (Mat ())
maximumMatchingSafe graph
  | isSymmetric graph ∧ isBipartite graph = Just (maximumMatching graph)
  | otherwise                             = Nothing
```

We focus on the latter approach and provide predicates for the necessary conditions only. This way we avoid unnecessary computations and provide an implementation that is particularly well-suited for rapid prototyping and interactive development.

[6] The error strategy can be improved in various ways. For instance, one can use error monads to describe the kind of error that occurred to let the user know, which condition exactly is violated.

5.5.1 Testing Symmetry

The first condition is the symmetry of a graph. Mathematically, a relation R is symmetric if and only if we have $R = R^{\top}$. In Section 4.3.5 we have defined a function for matrix transposition, which is based upon a special vector-matrix multiplication. While this provides a simple means to compute the transposition of a matrix, we still need a test for equality for matrices, which can be obtained from an equality test for vectors. Equality in Haskell is usually implemented as an instance of the *Eq* type class that provides the function $(\equiv)$, which is then exactly the predicate we are looking for.

The *Eq* instance for vectors is simple, because we can translate vectors into association lists and lists are already an instance of the *Eq* type class[7]. Similarly, the instance for matrices is simple as well, because matrices are vectors of vectors. Both instances can be obtained by simply adding **deriving** *Eq* after the definition of the respective data type.

Now the actual symmetry predicate is also rather simple. Technically, we should also check, whether the given matrix is a square matrix, but we omit this test for the sake of simplicity.

```
isSymmetric :: Eq α ⇒ Mat α → Bool
isSymmetric mat = mat ≡ transposeSquare mat
```

Note that this function has a quadratic complexity in the dimension of the given matrix, which is exactly the complexity for a symmetry test in an imperative implementation.

5.5.2 Testing Bipartiteness: A Simple Approach

There are two interesting approaches for testing the graph for being bipartite. The first one is by employing Kőnig's theorem that a graph is bipartite if and only if it does not contain an odd cycle. A little relational reasoning [Ber11] shows that this condition is equivalent to

$$E \cdot (E \cdot E)^{\star} \subseteq \bar{\mathsf{I}} \,. \tag{5.5.2.i}$$

This property can be checked once we have a matrix multiplication operation and an implementation of I at hand. The multiplication of two matrices over a semiring is simple, because it can be performed rowwise, where $(\odot')$ is the vector-matrix multiplication from Section 4.3.1.

```
times :: Semiring σ ⇒ Mat σ → Mat σ → Mat σ
times a b = Mat (fmap (⊙′ b) (matrix a))
```

Since the identity matrix needs to have a fixed size, we provide a function that takes a square matrix and returns the identity matrix of the same size.

[7] The same is true for many other associative structures like *IntMap*, thus we are not restricted to our particular vector implementation.

```
identityOf :: Semiring σ ⇒ Mat α → Mat σ
identityOf = Mat ∘ fmapWithKey (λi _ → mkVec [(i, one)]) ∘ matrix

fmapWithKey :: (Int → α → β) → Vec α → Vec β
fmapWithKey f = Vec ∘ map (λ(i, x) → (i, f i x)) ∘ unVec
```

The auxiliary function *fmapWithKey* is a generalisation of *fmap* and takes the key values into account[8].

In Chapter 3 we have developed a Kleene closure operation, but have not provided a star closure operation for matrices. This is easily amended once we have defined the addition of matrices, for which we reuse our function *unionWith* from Section 4.2.1.

```
unionMatWith :: (α → α → α) → Mat α → Mat α → Mat α
unionMatWith op a b = Mat (unionWith (unionWith op) (matrix a) (matrix b))

(⊞) :: Semiring σ ⇒ Mat σ → Mat σ → Mat σ
a ⊞ b = removeMatZeroes (unionMatWith (⊕) a b)

removeMatZeroes :: Semiring σ ⇒ Mat σ → Mat σ
removeMatZeroes = Mat ∘ fmap removeZeroes ∘ matrix
```

With these prerequisites the star closure can be expressed as $a^\star = 1 + a^+$ by Proposition 3.2.3.(5).

```
starClosure :: KleeneAlgebra κ ⇒ Mat κ → Mat κ
starClosure a = identityOf a ⊞ kleeneClosure a
```

Finally, Boolean algebra rules yield that

$$A \subseteq \overline{B} \iff A \cap B = \mathsf{O}\ .$$

Thus we can check whether $E \cdot (E \cdot E)^\star \cap \mathsf{I} = \mathsf{O}$. To that end we implement the emptiness test for matrices and the intersection of matrices as follows.

```
isEmptyMat :: Mat α → Bool
isEmptyMat = all (isEmptyVec ∘ snd) ∘ unVec ∘ matrix

intersectMatWithKey :: (Int → α → β → γ) → Mat α → Mat β → Mat γ
intersectMatWithKey op a b =
  Mat (intersectionWithKey (λ_ → intersectionWithKey op) (matrix a) (matrix b))

(⊓ₗ) :: Mat α → Mat β → Mat α
(⊓ₗ) = intersectMatWithKey (λ_ x _ → x)
```

Now the test for being bipartite can be implemented with these auxiliary functions. The resulting function is very similar to the one by Berghammer [Ber11].

```
isBipartite :: Mat α → Bool
isBipartite a = isEmptyMat (i ⊓ₗ (aBool ‘times‘ starClosure (aBool ‘times‘ aBool)))
  where aBool = toBoolMat a
        i     = identityOf a :: Mat Bool
```

[8] A similar function exists for *IntMaps* as well.

$$
\begin{aligned}
&\mathit{toBoolMat} :: \mathit{Mat}\ \alpha \to \mathit{Mat}\ \mathit{Bool}\\
&\mathit{toBoolMat} = \mathit{fmap}\ (\mathit{const}\ \mathit{one})\\
&\textbf{instance}\ \mathit{Functor}\ \mathit{Mat}\ \textbf{where}\\
&\quad \mathit{fmap}\ f = \mathit{Mat} \circ \mathit{fmap}\ (\mathit{fmap}\ f) \circ \mathit{matrix}
\end{aligned}
$$

The local variable *aBool* is a simplified version of the matrix, where all edge labels are mapped to the value *one* of the Boolean semiring.

Alternatively, since E is symmetric, we can use the Schröder equivalences to obtain the following equivalence:

$$
\begin{aligned}
& E \cdot (E \cdot E)^\star \subseteq \overline{\mathsf{I}}\\
\iff\ & \wr\ \text{Schröder equivalences, Definition 2.4.1.}\mathit{(RA3)}\ \wr\\
& \mathsf{I} \cdot ((E \cdot E)^\star)^\top \subseteq \overline{E}\\
\iff\ & \wr\ \text{since } E = E^\top \text{ we have } ((E \cdot E)^\star)^\top = (E \cdot E)^\star\ \wr\\
& \mathsf{I} \cdot (E \cdot E)^\star \subseteq \overline{E}\\
\iff\ & \wr\ \mathsf{I} \text{ is neutral with respect to multiplication}\ \wr\\
& (E \cdot E)^\star \subseteq \overline{E}\\
\iff\ & \wr\ \text{Boolean algebra rules}\ \wr\\
& (E \cdot E)^\star \cap E = \mathsf{O}\ . \qquad \text{(5.5.2.ii)}
\end{aligned}
$$

We can use this variant for an implementation as well.

$$
\begin{aligned}
&\mathit{isBipartite}' :: \mathit{Mat}\ () \to \mathit{Bool}\\
&\mathit{isBipartite}'\ m = \mathit{isEmptyMat}\ (\mathit{aBool}\ \sqcap_{\mathsf{I}}\ \mathit{starClosure}\ (\mathit{aBool}\ \text{`}\mathit{times}\text{`}\ \mathit{aBool}))\ \textbf{where}\\
&\quad \mathit{aBool} = \mathit{toBoolMat}\ m
\end{aligned}
$$

This implementation performs one multiplication less than the function *isBipartite*.

5.5.3 Testing Bipartiteness: An Efficient Approach

While the bipartiteness test from the previous section is based upon a purely relational characterisation and several auxiliary functions, it comes with two main disadvantages. First, the test is cubic in the number of vertices, because both the star closure and the matrix multiplication are cubic in the number of vertices. Second, and this is more crucial, this test only answers the question, whether a graph is bipartite, but does not provide a bipartition in the positive case. Fortunately, there is a known technique [KT06] that can both, test bipartiteness and find a bipartition at once. Additionally, this approach is at worst quadratic in the number of vertices.

First, we deal with testing whether a given pair of sets is a bipartition. We use Definition 5.1.2 for an implementation, but we omit the test for $a = \overline{b}$ for simplicity. In the actual application, this property is guaranteed otherwise.

$$
\begin{aligned}
&\mathit{isWeakBipartition} :: \mathit{Vec}\ () \to \mathit{Vec}\ () \to \mathit{Mat}\ () \to \mathit{Bool}\\
&\mathit{isWeakBipartition}\ a\ b\ e = \mathit{isEmptyVec}\ ((a \circledcirc e \cap_{\mathsf{I}} a) \cup_{\mathsf{I}} (b \circledcirc e \cap_{\mathsf{I}} b))
\end{aligned}
$$

The "weak" in *isWeakBipartition* refers to the fact that we do not check, whether the given vectors are in fact a partition of the vertex set. Now consider the case that G is connected and we have a vertex $v \in V$. Suppose that the relational point $p_v : \mathbb{1} \leftrightarrow V$ describes this vertex. Let $(s_i)_{i \in I}$ be the sequence of reachability steps from p_v through E as defined in Section 4.4, where $I = \mathbb{N}_{<k}$ for some suited $k \in \mathbb{N}$. Now let

$$I_{\text{even}} := \{ i \in I \mid i \text{ even} \} \quad \text{and} \quad I_{\text{odd}} := I \setminus I_{\text{even}} .$$

We define a specific partition by setting

$$a := \bigcup_{i \in I_{\text{even}}} s_i \quad \text{and} \quad b := \bigcup_{i \in I_{\text{odd}}} s_i . \tag{5.5.3.i}$$

Note that we have

$$\begin{aligned}
& a \cup b \\
= \;& \wr \text{ definition of } a \text{ and } b \text{ and combining indices } \wr \\
& \left(\bigcup_{i \in I_{\text{even}}} s_i \right) \cup \left(\bigcup_{i \in I_{\text{odd}}} s_i \right) \\
= \;& \wr \text{ combining indices (associativity and commutativity of unions) } \wr \\
& \bigcup_{i \in I} s_i \\
= \;& \wr \text{ property of the reachability steps } \wr \\
& p_v \cdot E^\star \\
= \;& \wr \text{ since } G \text{ is symmetric and connected, we have } E^\star = \mathsf{L} \wr \\
& p_v \cdot \mathsf{L} \\
= \;& \wr \text{ points are total, Definition 2.4.3 } \wr \\
& \mathsf{L} .
\end{aligned}$$

Also, we have $a \cap b = \mathsf{O}$, because reachability steps are pairwise disjoint. This yields the following property:

$$G \text{ is bipartite} \iff (a \cdot E \cap a) \cup (b \cdot E \cap b) = \mathsf{O} . \tag{5.5.3.ii}$$

For the direction "$\Longleftarrow$" we already know that $a \cup b = \mathsf{L}$ and $a \cap b = \mathsf{O}$ and thus $a = \overline{b}$. Since we assume $(a \cdot E \cap a) \cup (b \cdot E \cap b) = \mathsf{O}$, Definition 5.1.2 yields that (a, b) is a bipartition of G. The reverse direction is somewhat more technical and we provide a proof in Proof A.2.4. Note that Equation (5.5.3.ii) is a statement about the elements of a specific partition $\{ a, b \}$, namely the one defined above. The above equivalence does not hold for arbitrary partitions, because not every partition is a bipartition.

With the above characterisation we can implement the search for a bipartition for connected graphs. To achieve that we reuse our function *stepsWith* from Section 4.4, which computes the reachability steps.

$splitEvenOdd :: [\alpha] \to ([\alpha], [\alpha])$

$$\begin{aligned}
&\mathit{splitEvenOdd}\ xs = (\mathit{evens}\ xs, \mathit{odds}\ xs)\\
&\mathit{findBipartitionConnected} :: \mathit{Vertex} \to \mathit{Mat}\ () \to \mathit{Maybe}\ (\mathit{Vec}\ (), \mathit{Vec}\ ())\\
&\mathit{findBipartitionConnected}\ v\ g \mid \mathit{isWeakBipartition}\ a\ b = \mathit{Just}\ (a, b)\\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \qquad\qquad\quad = \mathit{Nothing}\\
&\quad \textbf{where}\ (a, b) = (\mathit{leftmostUnion} \times \mathit{leftmostUnion})\ (\mathit{splitEvenOdd}\ \mathit{steps})\\
&\qquad\qquad \mathit{steps}\ \ = \mathit{stepsWith}\ (\circledcirc)\ (\mathit{toVec}\ [v])\ [g]
\end{aligned}$$

The functions $\mathit{evens}, \mathit{odds} :: [\alpha] \to [\alpha]$ yield all values at the even and the odd positions in a list respectively. The function $(\circledcirc)$ is the discrete multiplication from Section 4.3.2, which ignores the values in the vector, but maintains the relational context. For functions the split operator $(\times)$ from *Control.Arrow* yields that

$$f \times g = \lambda(x, y) \to (f\ x, g\ y)\ .$$

In case of a non-connected graph, we consider every connected component C of G as a subgraph (V_C, E_C). For every connected component C of G let $\{ a_C, b_C \}$ be the partition as defined in Equation (5.5.3.i) for connected graphs. We define

$$a := \bigcup_{C \in \mathcal{C}} a_C \quad \text{and} \quad b := \bigcup_{C \in \mathcal{C}} b_C\ ,$$

where $\mathcal{C}$ denotes the set of all connected components of G. Then we get

$$\begin{aligned}
& a \cdot E \cap a\\
= \ & \wr\ \text{definition of } a\ \wr\\
& \left(\bigcup_{C \in \mathcal{C}} a_C\right) \cdot E \cap \left(\bigcup_{D \in \mathcal{C}} a_D\right)\\
= \ & \wr\ \text{relational multiplication is continuous with respect to } \cup\ \wr\\
& \left(\bigcup_{C \in \mathcal{C}} (a_C \cdot E)\right) \cap \left(\bigcup_{D \in \mathcal{C}} a_D\right)\\
= \ & \wr\ C \text{ is a connected component, thus } s \cdot E = s \cdot E_C \text{ for all } s \in 2^{V_C}\ \wr\\
& \left(\bigcup_{C \in \mathcal{C}} (a_C \cdot E_C)\right) \cap \left(\bigcup_{D \in \mathcal{C}} a_D\right)\\
= \ & \wr\ \text{distributive law in complete lattices}\ \wr\\
& \bigcup_{C \in \mathcal{C}} \bigcup_{D \in \mathcal{C}} (a_C \cdot E_C \cap a_D)\\
= \ & \wr\ C, D \in \mathcal{C} \wedge C \neq D \Longrightarrow \forall\, s \in 2^{V_C} : \forall\, t \in 2^{V_D} : s \cdot E_C \cap t = \mathsf{O}\ \wr\\
& \bigcup_{C \in \mathcal{C}} (a_C \cdot E_C \cap a_C)\ .
\end{aligned}$$

Similarly, we obtain the equality

$$b \cdot E \cap b = \bigcup_{C \in \mathcal{C}} (b_C \cdot E_C \cap b_C)\ .$$

Thus we have

$$
\begin{aligned}
& (a \cdot E \cap a) \cup (b \cdot E \cap b) \\
= \quad & \wr \text{ see above } \int \\
& \left(\bigcup_{C \in \mathcal{C}} (a_C \cdot E_C \cap a_C)\right) \cup \left(\bigcup_{C \in \mathcal{C}} (b_C \cdot E_C \cap b_C)\right) \\
= \quad & \wr \text{ rearrangement, due to commutativity and associativity of } \cup \int \\
& \bigcup_{C \in \mathcal{C}} (a_C \cdot E_C \cap a_C) \cup (b_C \cdot E_C \cap b_C) \ .
\end{aligned}
$$

In particular, this means that

$$(a \cdot E \cap a) \cup (b \cdot E \cap b) = \mathsf{O} \iff \forall\, C \in \mathcal{C} : (a_C \cdot E \cap a_C) \cup (b_C \cdot E \cap b_C) = \mathsf{O} \ .$$

Clearly, the set $\{\, a, b \,\}$ is a partition of V. Thus the above equivalence states that (a, b) is a bipartition of G if and only if for every connected component C we have that (a_C, b_C) is a bipartition of (V_C, E_C). Since $a = \overline{b}$ as noted above, the equality $(a \cdot E \cap a) \cup (b \cdot E \cap b) = \mathsf{O}$ yields that (a, b) is a bipartition of G and thus we get that G is bipartite. Furthermore, we get the following implication:

$$
\begin{aligned}
& (a \cdot E \cap a) \cup (b \cdot E \cap b) \neq \mathsf{O} \\
\iff \quad & \wr \text{ negation of the above equivalence } \int \\
& \exists\, C \in \mathcal{C} : (a_C \cdot E \cap a_C) \cup (b_C \cdot E \cap b_C) \neq \mathsf{O} \\
\iff \quad & \wr \text{ by Equation (5.5.3.ii) since } (V_C, E_C) \text{ is connected } \int \\
& \exists\, C \in \mathcal{C} : (V_C, E_C) \text{ is not bipartite} \\
\implies \quad & \wr \text{ by the Kőnig theorem there exists an odd cycle in } (V_C, E_C) \text{ and thus in } G \int \\
& G \text{ is not bipartite} \ .
\end{aligned}
$$

Thus we have the equivalence

$$G \text{ is bipartite} \iff (a \cdot E \cap a) \cup (b \cdot E \cap b) = \mathsf{O} \ .$$

In particular, we can first combine all bipartition candidates (the pairs (a_C, b_C)) as above and only then check the actual bipartiteness condition. This approach requires only two multiplications. Checking whether each connected component is bipartite on the other hand requires two multiplications per connected component.

We use the non-strictness of Haskell to obtain the reachability steps of all connected components. First, we compute the reachability steps from every single vertex and then remove equivalent results. An intermediate step in this procedure is a vector of all vertices of the graph, where each vertex is labelled with the list of the reachability steps from this vertex. This vector is then pruned as described below.

```
componentwise :: (Vertex → [Vec ()] → α) → Mat () → [α]
componentwise fun graph = runNew (length vs) (prune generated) where
  vs          = vertices graph
```

```
  generated = map (λv → (v, stepsWith (⊜) (toVec [v]) [graph])) vs
  prune []              = return []
  prune ((i, ls) : ilss) = do vis ← contains i
                              if vis
                                then prune ilss
                                else do includeAll (fromVec (leftmostUnion ls))
                                        fmap (fun i ls:) (prune ilss)
vertices :: Mat α → [Vertex]
vertices = rowIndices
includeAll :: [Vertex] → SetM ()
includeAll = mapM_ include
```

The function *vertices* returns the list of the vertices of a graph and the function *includeAll* applies the monadic set function *include* from Section 4.5 to every element of the list, ignoring the resulting effect. The actual function then first creates a list of pairs, where the first element is a vertex and the second one is the list of reachability steps starting with the vertex. Note that the steps are not computed until demanded. Then we create an empty set that maintains the list of all visited vertices and traverse the above list left-to-right. If we encounter a pair (v, ls) such that v is contained in the visited vertices, its list of reachability steps is ignored and we continue with the remaining vertices. If v is not contained in the visited set, we add (v, ls) to the result list and include all reachability steps in the set of visited vertices. This procedure is correct only for symmetric graphs, because in non-symmetric graphs the weakly connected components (connected components of the symmetric closure) can be strictly different from the strongly connected components (maximal subsets in which each two vertices are reachable from one another).

Now we can combine the previous functions into the following search for a bipartition.

```
findBipartition :: Mat () → Maybe (Vec (), Vec ())
findBipartition g | isWeakBipartition a b g = Just (a, b)
                  | otherwise = Nothing
  where (a, b)      = combine (unzip (componentwise (const splitEvenOdd) g))
        combine     = leftConcat × leftConcat
        leftConcat  = leftmostUnion ∘ concat
```

The componentwise reachability steps are collected and split into even and odd parts respectively. The result is a list of pairs, where each pair consists of the list of the even reachability steps in the first component and the odd reachability steps in the second component. This list is unzipped and the resulting lists are concatenated and unified, thus constituting a bipartition candidate for the complete graph. The actual test whether this candidate is indeed a bipartition is the same as in the function *findBipartitionConnected*.

Note that despite the somewhat technical derivation, the function *findBipartition* is rather simple. This simplicity is particularly due to the compositional approach, because we can use several pre-defined Haskell functions as well as functions we have defined before. An actual test for being bipartite is now very simple, because we only need to check, whether the result of the *findBipartition* function is a *Just* value, which is precisely what the function $isJust :: Maybe\ \alpha \rightarrow Bool$ from the Haskell module *Data.Maybe* does.

$isBipartiteFast :: Mat\ () \rightarrow Bool$
$isBipartiteFast = isJust \circ findBipartition$

Chapter 6

Bipartite Minimum Vertex Cover

In this chapter we consider the problem of finding a minimum set of vertices in a graph, such that all edges begin or end in at least one of these vertices. Many problems in graph theory have so-called *dual* problems, which can be solved in a very similar fashion as the problem itself, but instead of finding a maximal element with respect to some condition, one searches for a minimal element with respect to a similar condition. The minimum vertex cover (in bipartite graphs) is dual to the maximum matching problem from Chapter 5. In fact, the solution of the vertex cover problem can be based on the results about maximum matchings both in terms of theory and practice. The solution that we present is based upon Kőnig's theorem, which we state in Theorem 6.1.2. It uses the construction of a maximum matching for the construction of a minimum vertex cover and the characterisation of the existence of augmenting paths from Proposition 5.2.4 for the proof of the correctness of the construction.

We split this chapter into two main parts. The first part consists of Section 6.1 and Section 6.2, and contains the theoretical foundations that are necessary for the computation of a minimum vertex cover as well as a purely relation-algebraic proof of their correctness. The second part of this chapter consists of Section 6.3 and Section 6.4 and contains the actual implementation of a function that computes a minimum vertex cover in a bipartite graph as well as an implementation of an approximation algorithm in the general case.

At the time of this writing our proof of Kőnig's theorem as well as the algebraic implementation in Haskell have been novel. The former has since beed submitted for publication [BDHS16]. It is possible to implement a minimum vertex cover function using the Haskell library *fgl* of Erwig and Miljenovic [EM14]. To achieve that, one needs to implement the maximum matching function using the *fgl* library and then compute the star closure of a product graph, which is also technically possible in *fgl*. However, our approach benefits from our particular matching function from Section 5.4 and is based upon a relation-algebraic foundation.

6.1 Specification and Solution

A vertex cover is a set c of vertices such that for every arc in E either its starting point or its end point is contained in c. Some examples of vertex covers are shown in

Figure 6.1. From this informal specification one can derive the known relational one as follows[1]:

$$
\begin{aligned}
& c \text{ vertex cover} \\
\iff \quad & \wr \text{ specification above } \int \\
& \forall\, x, y \in V : (x,y) \in E \Rightarrow ((\square, x) \in c \vee (\square, y) \in c) \\
\iff \quad & \wr \text{ double negation, de Morgan's law and definition of complement } \int \\
& \forall\, x, y \in V : (x,y) \in E \Rightarrow \neg((\square, x) \in \overline{c} \wedge (\square, y) \in \overline{c}) \\
\iff \quad & \wr \text{ definition of transposition } \int \\
& \forall\, x, y \in V : (x,y) \in E \Rightarrow \neg \left((x, \square) \in \overline{c}^{\top} \wedge (\square, y) \in \overline{c}\right) \\
\iff \quad & \wr\ \mathbb{1} = \{\, \square \,\} \int \\
& \forall\, x, y \in V : (x,y) \in E \Rightarrow \neg \left(\exists\, o \in \mathbb{1} : (x, o) \in \overline{c}^{\top} \wedge (o, y) \in \overline{c}\right) \\
\iff \quad & \wr \text{ definition of relational composition } \int \\
& \forall\, x, y \in V : (x,y) \in E \Rightarrow \neg \left((x, y) \in \overline{c}^{\top} \cdot \overline{c}\right) \\
\iff \quad & \wr \text{ definition of complement } \int \\
& \forall\, x, y \in V : (x,y) \in E \Rightarrow (x, y) \in \overline{\overline{c}^{\top} \cdot \overline{c}} \\
\iff \quad & \wr \text{ definition of subset } \int \\
& E \subseteq \overline{\overline{c}^{\top} \cdot \overline{c}} \\
\iff \quad & \wr \text{ Boolean algebra: } x \sqsubseteq y \iff \overline{y} \sqsubseteq \overline{x} \int \\
& \overline{c}^{\top} \cdot \overline{c} \subseteq \overline{E} \\
\iff \quad & \wr \text{ Schröder equivalence; complement and transposition are involutory } \int \\
& \overline{c} \cdot E \subseteq c \,.
\end{aligned}
$$

We take this simplified expression as definition.

6.1.1 Definition (Vertex cover)**.**
Let $G = (V, E)$ be a graph and $c \subseteq E$. We define

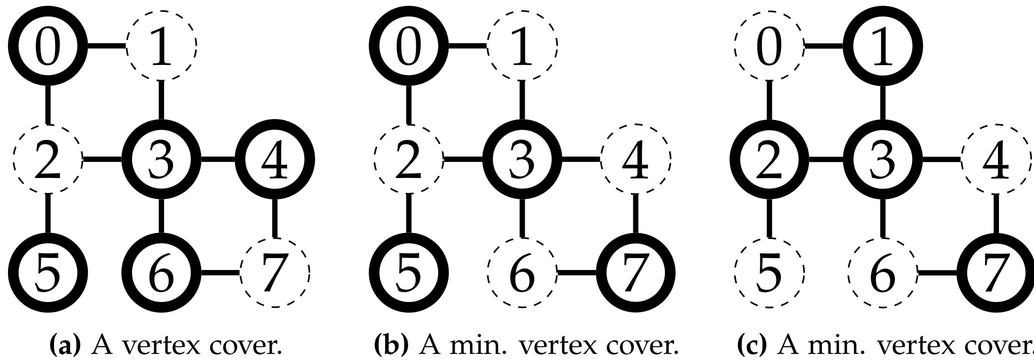
$$c \text{ is a } \textit{vertex cover of } G \ :\iff\ \overline{c} \cdot E \subseteq c \,.$$

We define
$$\mathsf{cover}(G) := \min \left\{\, |d| \;\middle|\; d \in 2^V \wedge d \text{ vertex cover of } G \,\right\}$$

and call this number *the vertex cover number of* G. c is called *minimum vertex cover of* G if and only $|c| = \mathsf{cover}(G)$. ⫽

Clearly, the vertex set itself is a vertex cover and thus vertex covers always exist. Minimum vertex covers are not necessarily unique, as is shown in Figure 6.1. In general graphs the task of finding a vertex cover is known to be NP-complete [Kar72].

[1] The same result for graphs such that $E \subseteq \overline{\mathrm{I}}$ is stated by Schmidt and Ströhlein [SS93].



(a) A vertex cover. **(b)** A min. vertex cover. **(c)** A min. vertex cover.

Figure 6.1. Vertex covers in a graph.

Fortunately, in case of a bipartite graph, the vertex cover problem has a complexity that is polynomial in the graph size.

The key idea originates from another theorem of Kőnig [Die00], which shows that in a bipartite graph the size of a maximum matching is equal to the size of a minimum vertex cover. The proof of this statement is constructive and is based upon the construction of a vertex cover that has the desired size. An additional argument then yields the minimality of the vertex cover.

We now first rephrase this statement.

6.1.2 Theorem (Vertex covers and matchings, Kőnig (1931)).
Let $G = (V, E)$ be a finite, symmetric graph. Then the following statements hold:

(1) $\mathsf{matching}(G) \leqslant 2 \cdot \mathsf{cover}(G)$.

(2) Suppose that G is bipartite and (a, b) is a bipartition of G. Let $M \subseteq E$ be a maximum matching in G. Set

$$r := (\mathsf{uncovered}(M) \cap a) \cdot ((E \setminus M) \cdot M)^\star \cdot (\mathsf{I} \cup (E \setminus M))$$

and

$$c := (a \cap \overline{r}) \cup (b \cap r)\ .$$

Then we have:

(i) c is a vertex cover of G.

(ii) $2 \cdot |c| = |M|$.

In particular, $\mathsf{matching}(G) = 2 \cdot \mathsf{cover}(G)$.

The vertex cover c in Theorem 6.1.2.(2) is a relational description of the informal definition given by Diestel [Die00]. Note that the proof is constructive and an actual vertex cover with half the size of a maximum matching is provided. This construction is particularly important, because we can use the relational definition of this vector for a rather simple implementation, as we will see in Section 6.3.

We have adjusted the statement of Kőnig's theorem to our setting, because the original version is a statement about undirected graphs. As we have discussed after Lemma 5.1.4, there are exactly twice as many directed edges as there are undirected ones, which explains the factor 2 in the above version of Kőnig's theorem.

Before we implement a function that computes a minimum vertex cover in a bipartite graph, we first provide a purely relation-algebraic proof of Kőnig's theorem in Section 6.2. The resulting implementation is then given in Section 6.3.

6.2 Proof of Kőnig's Theorem

In this section we provide a relation-algebraic proof of Theorem 6.1.2. A variant of this proof has been submitted for publication [BDHS16]. Mostly, Kőnig's theorem is reduced to the statement

$$\mathsf{matching}(G) = 2 \cdot \mathsf{cover}(G)$$

for bipartite graphs. This part is proved by Schmidt and Ströhlein [SS93], but without the actual definition of the vertex cover as in our case and with an informal computation of its cardinality. Kawahara [Kaw06] provides a proof of a theorem that is known to be equivalent to the above statement. The proof relies on properties of network flows and, again, does not deal with the concrete vertex cover from Kőnig's theorem. The proof of Kőnig's theorem by Diestel [Die00] is not relation-algebraic.

The proof presented in this section itself is purely relation-algebraic. However, a crucial component in this proof is the characterisation of the existence of augmenting paths from Proposition 5.2.4 and our proof of this result is not relation-algebraic. Also, we have not provided a relation-algebraic proof of the Berge lemma. This downside is amended by Berghammer, Danilenko, Höfner, and Stucke [BDHS16], where we provide a purely algebraic proof of the fact that in case of a bipartite graph we have the following inclusion for every maximum matching M:

$$\mathsf{uncovered}(M) \cdot (E \setminus M) \cdot (M \cdot (E \setminus M))^{\star} \subseteq \overline{\mathsf{uncovered}(M)} \,. \tag{6.2.1}$$

Since this inclusion is proved using only relation-algebraic means in the above source, the actual proof of the Kőnig theorem can be obtained with purely algebraic reasoning as well.

To be able to prove Kőnig's theorem, we need a notion of the cardinality of a relation. While this is clearly possible to manage by the usual, set-theoretic reasoning, we wish to maintain a purely algebraic proof. To achieve that, we use the axiomatisation of cardinality by Kawahara [Kaw06]. The idea of this axiomatisation is to provide axioms for the cardinality function that use only purely relational means.

6.2.2 Definition (Relational cardinality)**.**
Let $\mathcal{F} := \langle\, X \mid X \text{ finite, non-empty set} \,\rangle$ be the class of all finite and non-empty sets and let $\mathcal{C} := \mathcal{F} \times \mathcal{F}$. Let $(|-|_{X,Y})_{(X,Y)\in\mathcal{C}}$ be a family of mappings such that for all

$(X, Y) \in \mathcal{C}$ we have $|-|_{X,Y} : \mathcal{R}(X, Y) \to \mathbb{N}$. Then $(|-|_{X,Y})_{(X,Y)\in\mathcal{C}}$ is called *relational cardinality* if and only if all of the following conditions hold:

(C1) $\forall X, Y \in \mathcal{F} : \forall r \in \mathcal{R}(X, Y) : |r|_{X,Y} = 0 \Leftrightarrow r = \mathsf{O}$.

(C2) $|\mathsf{I}_{\mathbb{1}}|_{\mathbb{1},\mathbb{1}} = 1$.

(C3) $\forall X, Y \in \mathcal{F} : \forall r \in \mathcal{R}(X, Y) : |r|_{X,Y} = \left|r^{\top}\right|_{Y,X}$.

(C4) $\forall X, Y \in \mathcal{F} : \forall r, s \in \mathcal{R}(X, Y) : |r \cup s|_{X,Y} = |r|_{X,Y} + |s|_{X,Y} - |r \cap s|_{X,Y}$.

(C5) $\forall X, Y, Z \in \mathcal{F} : \forall r \in \mathcal{R}(X, Y) : \forall s \in \mathcal{R}(Z, Y) : \forall q \in \mathcal{R}(Z, X) :$

$$q^{\top} \cdot q \subseteq \mathsf{I} \Rightarrow \max\left\{ \left|r \cap q^{\top} \cdot s\right|_{X,Y}, \left|q \cap s \cdot r^{\top}\right|_{Z,X} \right\} \leqslant |q \cdot r \cap s|_{Z,Y} . \qquad \lrcorner\!\!/\!\!/$$

Definition 6.2.2.*(C1)* states that there is exactly one relation of cardinality zero in each lattice. Definition 6.2.2.*(C2)* is useful for computing the size of constants of a given type — for instance, one would naturally assume that the cardinality of $\mathsf{L}_{\{0,1\},\{0,1,2\}}$ is 6, which is indeed the case. By Definition 6.2.2.*(C3)* we know that cardinality is invariant under transposition and Definition 6.2.2.*(C4)* is the translation of the respective property of the concrete cardinality of finite sets to the relational setting. Finally, Definition 6.2.2.*(C5)* is referred to as *Dedekind rule for cardinalities* and states that the factors of the relational Dedekind rule have a smaller cardinality than the value that is decomposed by said rule.

The main result of Kawahara [Kaw06] is that there is a unique relational cardinality.

6.2.3 Theorem (Existence and uniqueness of the relational cardinality).
The set-theoretic cardinality of relations is the only relational cardinality. ⫽

Due to this theorem we use the usual notations $|r|$ to denote the cardinality of a relation, but we use only the axiomatisation from Definition 6.2.2 for proofs. Note that we omit indices for simplicity.

To get acquainted with relational cardinality, we first note some useful properties. All of the following statements are motivated by properties of the set-theoretic cardinality function. For example, if $f : X \to Y$ is a function and $A \subseteq X$, then $|f(A)| \leqslant |A|$ and if f is injective, then $|f(A)| = |A|$. Since we have $f(A) = A \cdot f$ in relational terms, we would expect that $|A \cdot f| \leqslant |A|$ holds as well. Indeed, this is the case.

6.2.4 Proposition (Properties of the relational cardinality).
The following hold:

(1) For all finite, non-empty sets X, Y and all $r, s : X \leftrightarrow Y$ we have

$$r \subseteq s \Longrightarrow |r| \leqslant |s| .$$

This statement is also mentioned by Kawahara [Kaw06] without a proof.

(2) *For all finite sets X, Y, Z, all $r : X \leftrightarrow Y$, and all $s : Z \leftrightarrow X$ we have*

$$r \text{ injective} \ \wedge\ s \subseteq \mathsf{L}_{Z,Y} \cdot r^{\top} \Longrightarrow |s| \leqslant |s \cdot r| \ .$$

(3) *For all finite, non-empty sets X, Y, Z, all $r : X \leftrightarrow Y$, and all $s : Z \leftrightarrow X$ we have*

$$r \text{ univalent} \Longrightarrow |s \cdot r| \leqslant |s| \ .$$

(4) *For all finite, non-empty sets X, Y, Z, all $r : X \leftrightarrow Y$, and all $s : Z \leftrightarrow X$ we have*

$$r \text{ univalent} \ \wedge\ r \text{ injective} \ \wedge\ s \subseteq \mathsf{L}_{Z,Y} \cdot r^{\top} \Longrightarrow |s| = |s \cdot r| \ .$$

(5) *For all finite, non-empty sets X, Y and all $r : X \leftrightarrow Y$ we have*

$$r \text{ univalent} \ \wedge\ r \text{ injective} \Longrightarrow |\mathsf{L}_{\mathbb{1},X} \cdot r| = |r| \ .$$

Proof. Statement (1). Let X, Y be finite, non-empty sets and $r, s : X \leftrightarrow Y$ such that $r \subseteq s$ holds. Then we get

$$\begin{aligned}
& |s| \\
= \ & \wr \text{ since } s = s \cap \mathsf{L} = s \cap (\overline{r} \cup r) = (s \cap \overline{r}) \cup (s \cap r) \ \int \\
& |(s \cap \overline{r}) \cup (s \cap r)| \\
= \ & \wr \text{ by Definition 6.2.2.}(C4) \ \int \\
& |s \cap \overline{r}| + |s \cap r| - |(s \cap \overline{r}) \cap (s \cap r)| \\
= \ & \wr \ (s \cap \overline{r}) \cap (s \cap r) = s \cap \overline{r} \cap r = s \cap \mathsf{O} = \mathsf{O} \ \int \\
& |s \cap \overline{r}| + |s \cap r| - |\mathsf{O}| \\
= \ & \wr \text{ by Definition 6.2.2.}(C1) \ \int \\
& |s \cap \overline{r}| + |s \cap r| \\
= \ & \wr \ r \subseteq s \iff r \cap s = r \ \int \\
& |s \cap \overline{r}| + |r| \\
\geqslant \ & \wr \ |s \cap \overline{r}| \in \mathbb{N} \ \int \\
& |r| \ .
\end{aligned}$$

Statement (2). Let X, Y, Z be finite, non-empty sets, $r : X \leftrightarrow Y$ and $s : Z \leftrightarrow X$. Suppose that r is injective and that $s \subseteq \mathsf{L}_{Z,Y} \cdot r^{\top}$ holds. Then we get

$$\begin{aligned}
& |s| \\
= \ & \wr \text{ by Definition 6.2.2.}(C3) \ \int \\
& \left|s^{\top}\right| \\
= \ & \wr \text{ since } s \subseteq \mathsf{L}_{Z,Y} \cdot r^{\top}, \text{ we have } s^{\top} \subseteq r^{\top^{\top}} \cdot \mathsf{L}_{Y,Z} \ \int \\
& \left|s^{\top} \cap r^{\top^{\top}} \cdot \mathsf{L}_{Y,Z}\right| \\
\leqslant \ & \wr \text{ by Definition 6.2.2.}(C5), \text{ with } q' = r^{\top}, r' = s^{\top}, s' = \mathsf{L}_{Y,Z}, \text{ since } q' \text{ is univalent } \ \int
\end{aligned}$$

$$
\begin{aligned}
& \left|r^\top \cdot s^\top \cap \mathsf{L}_{Y,Z}\right| \\
= \quad & \wr\ \mathsf{L}_{Y,Z} \text{ is the largest element } \int \\
& \left|r^\top \cdot s^\top\right| \\
= \quad & \wr\ \text{property of the transposition } \int \\
& \left|(s \cdot r)^\top\right| \\
= \quad & \wr\ \text{by Definition 6.2.2.}\mathit{(C3)} \int \\
& |s \cdot r| \ .
\end{aligned}
$$

Statement (3). Let X, Y, Z be finite, non-empty sets, $r : X \leftrightarrow Y$, $s : Z \leftrightarrow X$ and suppose that r is univalent. Then we get the following estimate:

$$
\begin{aligned}
& |s \cdot r| \\
= \quad & \wr\ \text{Definition 6.2.2.}\mathit{(C3)} \int \\
& \left|(s \cdot r)^\top\right| \\
= \quad & \wr\ \text{property of the transposition } \int \\
& \left|r^\top \cdot s^\top\right| \\
= \quad & \wr\ \text{intersection is idempotent } \int \\
& \left|r^\top \cdot s^\top \cap r^\top \cdot s^\top\right| \\
\leqslant \quad & \wr\ \text{by Definition 6.2.2.}\mathit{(C5)}\text{, with } q' = r,\ r' = r^\top \cdot s^\top,\ s' = s, \text{ since } q' \text{ is univalent } \int \\
& \left|r \cdot r^\top \cdot s^\top \cap s^\top\right| \\
\leqslant \quad & \wr\ \text{by Proposition 6.2.4.(1) } \int \\
& \left|s^\top\right| \\
= \quad & \wr\ \text{by Definition 6.2.2.}\mathit{(C3)} \int \\
& |s| \ .
\end{aligned}
$$

Statement (4) is obvious from Statement (2) and Statement (3).
Statement (5) is proved by Kawahara [Kaw06]. ⫽

With these properties we can already prove the first statement of Kőnig's theorem.

6.2.5 Proof (Proof of Theorem 6.1.2.(1)).

Proof. Let $M \subseteq E$ be a matching and $c \subseteq V$ be a vertex cover. Then we have

$$
\begin{aligned}
& \mathsf{L}_{\mathbb{1},V} \cdot M \\
= \quad & \wr\ \text{property of the complement } \int \\
& (c \cup \overline{c}) \cdot M
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \wr \text{ relational composition distributes over unions } \wr \\
&\quad c \cdot M \cup \overline{c} \cdot M \\
&\subseteq \quad \wr\ M \subseteq E \text{ and monotonocity of composition and union } \wr \\
&\quad c \cdot M \cup \overline{c} \cdot E \\
&\subseteq \quad \wr\ c \text{ is vertex cover, Definition 6.1.1 } \wr \\
&\quad c \cdot M \cup c\ . && \text{(6.2.5.a)}
\end{aligned}
$$

With this condition, we get the following inequalities:

$$
\begin{aligned}
&\quad |M| \\
&= \quad \wr\ M \text{ is matching, thus univalent and injective and Proposition 6.2.4.(5) } \wr \\
&\quad |\mathsf{L}_{\mathbb{1},V} \cdot M| \\
&\leqslant \quad \wr \text{ Equation (6.2.5.a) and Proposition 6.2.4.(1) } \wr \\
&\quad |c \cdot M \cup c| \\
&= \quad \wr \text{ by Definition 6.2.2.}(C4) \wr \\
&\quad |c \cdot M| + |c| - |c \cdot M \cap c| \\
&\leqslant \quad \wr \text{ property of the natural numbers } \wr \\
&\quad |c \cdot M| + |c| \\
&\leqslant \quad \wr\ M \text{ is univalent, Proposition 6.2.4.(3) } \wr \\
&\quad |c| + |c|\ .
\end{aligned}
$$

We thus have that $2 \cdot |c|$ is an upper bound of $\{\, |D| \mid D \in 2^E \wedge D \text{ matching} \,\}$ and thus

$$
\begin{aligned}
\mathsf{matching}(G) &= \max \left\{ |D| \;\middle|\; D \in 2^E \wedge D \text{ matching} \right\} \\
&\leqslant 2 \cdot |c|\ .
\end{aligned}
$$

But now $\mathsf{matching}(G)$ is a lower bound of the set $\{\, 2 \cdot |c| \mid c \in 2^V \wedge c \text{ vertex cover} \,\}$ and thus

$$
\begin{aligned}
\mathsf{matching}(G) &\leqslant \min \left\{ 2 \cdot |c| \;\middle|\; c \in 2^V \wedge c \text{ vertex cover} \right\} \\
&= 2 \cdot \min \left\{ |c| \;\middle|\; c \in 2^V \wedge c \text{ vertex cover} \right\} \\
&= 2 \cdot \mathsf{cover}(G)\ .
\end{aligned}
$$

The auxiliary statement that the cardinality of every matching is less or equal than twice the cardinality of a vertex cover is proved in a very similar fashion by [BHS16], save for the fact that the above formulation is explicitly true for *every* matching, not necessarily a maximum one.

For the remainder of this section we assume that $G = (V, E)$ is a bipartite graph with a bipartition (a, b). Furthermore, let $M \subseteq E$ be a matching and let us abbreviate

$$
N := E \setminus M\ , \tag{6.2.5.i}
$$

$$
u := \mathsf{uncovered}(M) = \overline{\mathsf{L}_{\mathbb{1},V} \cdot M}\ . \tag{6.2.5.ii}
$$

The name N is a mnemonic for *non-matching edges*. We proceed as in the theorem of Kőnig and define

$$r := (u \cap a) \cdot (N \cdot M)^{\star} \cdot (\mathsf{I} \cup N) \,, \tag{6.2.5.iii}$$

$$c := (a \cap \overline{r}) \cup (b \cap r) \,. \tag{6.2.5.iv}$$

Before we begin, let us first note what all these values are. The relation N is the complement of M in E and u are those vertices, which do not touch any edge in M. The more interesting value is r. This set can be described as the set of all those vertices that are reachable from $u \cap a$ via an N-M-alternating walk. Finally, the value c, which is the desired vertex cover, consists of two disjoint parts, one of which is contained in a and the other one in b. The part in a is not reachable from $u \cap a$ via an alternating walk and the part in b is reachable via such a walk.

First, we show that $\overline{c} \cdot E \subseteq c$ holds, which yields that c is a vertex cover. To that end we note that we have the following equality:

$$\overline{c} = (a \cap r) \cup (b \cap \overline{r}) \,. \tag{6.2.5.v}$$

This equality can be obtained as follows:

$$\begin{aligned}
& \overline{c} \\
= \;& \wr \text{ by Equation (6.2.5.iv) } \wr \\
& \overline{(a \cap \overline{r}) \cup (b \cap r)} \\
= \;& \wr \text{ de Morgan rule and the complement function is an involution } \wr \\
& (\overline{a} \cup r) \cap (\overline{b} \cup \overline{r}) \\
= \;& \wr \text{ distributive law } \wr \\
& (\overline{a} \cap \overline{b}) \cup (\overline{a} \cap \overline{r}) \cup (r \cap \overline{b}) \cup (r \cap \overline{r}) \\
= \;& \wr\; \overline{a} = b, \overline{b} = a, \text{ and } r \cap \overline{r} = \mathsf{O} \;\wr \\
& (b \cap a) \cup (b \cap \overline{r}) \cup (r \cap a) \\
= \;& \wr\; a \cap b = \mathsf{O} \text{ and commutativity of } \cup \text{ and } \cap \;\wr \\
& (a \cap r) \cup (b \cap \overline{r}) \,.
\end{aligned}$$

Intuitively speaking, every value in a that is reachable along an alternating walk that starts in $u \cap a$ should require an even number of edges, while every value in b that is reachable along such a walk should require an odd number of edges. This is due to the fact that (a, b) is a bipartition. The above observation allows a simple representation of $a \cap r$ and $b \cap r$. We note these properties, as well as an additional one in the following lemma.

6.2.6 Lemma (Alternating successors)**.**
The following properties hold:

(1) For all $w \subseteq u$ we have $w \cdot (N \cdot M)^{\star} \cdot M \subseteq w \cdot (N \cdot M)^{\star} \cdot N$.

(2) For all $w \subseteq u$ we have $w \cdot (N \cdot M)^\star \cdot E = w \cdot (N \cdot M)^\star \cdot N$.

(3) For all $w \subseteq a$ we have $w \cdot (N \cdot M)^\star \subseteq a$.

(4) For all $w \subseteq a$ we have $w \cdot (N \cdot M)^\star \cdot N \subseteq b$.

(5) $a \cap r = (u \cap a) \cdot (N \cdot M)^\star$.

(6) $b \cap r = (u \cap a) \cdot (N \cdot M)^\star \cdot N$.

Proof. Statement (1). First, we have

$$\begin{aligned} & \textbf{true} \\ \iff \quad & \wr \text{ reflexivity of } \subseteq \int \\ & \mathsf{L}_{\mathbb{1},V} \cdot M \subseteq \mathsf{L}_{\mathbb{1},V} \cdot M \\ \iff \quad & \wr \text{ Schröder equivalence and } M = M^\top \int \\ & \overline{\mathsf{L}_{\mathbb{1},V} \cdot M} \cdot M \subseteq \mathsf{O} \end{aligned}$$

and thus we have

$$u \cdot M = \overline{\mathsf{L}_{\mathbb{1},V} \cdot M} \cdot M = \mathsf{O}\ . \tag{6.2.6.a}$$

Now let $w \subseteq u$. With the above equation we obtain the following:

$$\begin{aligned} & w \cdot (N \cdot M)^\star \cdot M \\ = \quad & \wr \text{ fixpoint property of the star closure, Proposition 3.2.3.(1) } \int \\ & w \cdot (\mathsf{I} \cup (N \cdot M)^\star \cdot N \cdot M) \cdot M \\ = \quad & \wr \text{ composition distributes over unions, } \mathsf{I} \cdot M = M \int \\ & w \cdot (M \cup (N \cdot M)^\star \cdot N \cdot M \cdot M) \\ \subseteq \quad & \wr\ M \text{ is a matching, thus } M \cdot M \subseteq \mathsf{I} \text{ and monotonicity of composition } \int \\ & w \cdot (M \cup (N \cdot M)^\star \cdot N) \\ = \quad & \wr \text{ distributivity of the composition over unions } \int \\ & w \cdot M \cup w \cdot (N \cdot M)^\star \cdot N \\ = \quad & \wr\ w \subseteq u \text{, thus } w \cdot M \subseteq u \cdot M = \mathsf{O} \text{ by Equation (6.2.6.a) and } \mathsf{O} \cup x = x \int \\ & w \cdot (N \cdot M)^\star \cdot N\ . \end{aligned}$$

Statement (2). Let $w \subseteq u$. Then we have the following chain of inclusions:

$$\begin{aligned} & w \cdot (N \cdot M)^\star \cdot E \\ = \quad & \wr \text{ since } E = N \cup M \text{ by the definition of } N \text{, Equation (6.2.5.i) } \int \\ & w \cdot (N \cdot M)^\star \cdot (N \cup M) \\ = \quad & \wr \text{ composition distributes over unions } \int \\ & w \cdot (N \cdot M)^\star \cdot N \cup w \cdot (N \cdot M)^\star \cdot M \\ \subseteq \quad & \wr \text{ by Lemma 6.2.6.(1) and monotonicity of unions } \int \end{aligned}$$

$$
\begin{aligned}
& w \cdot (N \cdot M)^\star \cdot N \cup w \cdot (N \cdot M)^\star \cdot N \\
= \; & \wr \; \cup \text{ is idempotent } \wr \\
& w \cdot (N \cdot M)^\star \cdot N \\
\subseteq \; & \wr \text{ since } N \subseteq E \text{ and composition is monotonic } \wr \\
& w \cdot (N \cdot M)^\star \cdot E \; .
\end{aligned}
$$

Statement (3). For every $x \subseteq a$ we have the following inclusion:

$$
\begin{aligned}
& x \cdot (N \cdot M) \\
\subseteq \; & \wr \; x \subseteq a,\ N, M \subseteq E, \text{ monotonicity and associativity of the composition } \wr \\
& (a \cdot E) \cdot E \\
\subseteq \; & \wr \; (a, b) \text{ is a bipartition, thus } \mathbf{true} \iff a \cdot E \cap a = \mathsf{O} \iff a \cdot E \subseteq \overline{a} = b \; \wr \\
& b \cdot E \\
\subseteq \; & \wr \; (a, b) \text{ is a bipartition, thus } \mathbf{true} \iff b \cdot E \cap b = \mathsf{O} \iff b \cdot E \subseteq \overline{b} = a \; \wr \\
& a \; .
\end{aligned}
\tag{6.2.6.b}
$$

Now let $w \subseteq a$. Then we get

$$
\begin{aligned}
& w \cdot (N \cdot M)^\star \\
\subseteq \; & \wr \; w \subseteq a \text{ and monotonicity of the composition } \wr \\
& a \cdot (N \cdot M)^\star \\
\subseteq \; & \wr \text{ by Definition 3.2.1.}\textit{(KA3)}\text{, since } a \cdot (N \cdot M) \subseteq a \text{ by Equation (6.2.6.b) } \wr \\
& a \; .
\end{aligned}
$$

Statement (4). Let $w \subseteq a$. Then we have

$$
\begin{aligned}
& w \cdot (N \cdot M)^\star \cdot N \\
\subseteq \; & \wr \text{ by Lemma 6.2.6.(3) } \wr \\
& a \cdot N \\
\subseteq \; & \wr \; N \subseteq E \text{ and monotonicity of the composition } \wr \\
& a \cdot E \\
\subseteq \; & \wr \; (a, b) \text{ is a bipartition and thus } a \cdot E \subseteq \overline{a} = b \; \wr \\
& b \; .
\end{aligned}
$$

Statement (5). Since $u \cap a \subseteq a$, we have $(u \cap a) \cdot (N \cdot M)^\star \cdot N \subseteq b$ by Lemma 6.2.6.(4), which is equivalent to the equality

$$
\begin{aligned}
& \mathsf{O} \\
= \; & \wr \text{ Boolean algebra rules } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot N \cap \overline{b} \\
= \; & \wr \text{ since } (a, b) \text{ is a bipartition, we have } \overline{b} = a \; \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot N \cap a \; .
\end{aligned}
\tag{6.2.6.c}
$$

This yields the desired result as follows:

$$
\begin{aligned}
& a \cap r \\
= \quad & \wr \text{ definition of } r \ \wr \\
& a \cap (u \cap a) \cdot (N \cdot M)^\star \cdot (\mathsf{I} \cup N) \\
= \quad & \wr \text{ composition distributes over unions, } \mathsf{I} \text{ is neutral with respect to composition } \wr \\
& a \cap ((u \cap a) \cdot (N \cdot M)^\star \cup (u \cap a) \cdot (N \cdot M)^\star \cdot N) \\
= \quad & \wr \text{ distributive law } \wr \\
& (a \cap (u \cap a) \cdot (N \cdot M)^\star) \cup (a \cap (u \cap a) \cdot (N \cdot M)^\star \cdot N) \\
= \quad & \wr \text{ second intersection is } \mathsf{O} \text{ by Equation (6.2.6.c) } \wr \\
& a \cap (u \cap a) \cdot (N \cdot M)^\star \\
= \quad & \wr \ (u \cap a) \cdot (N \cdot M)^\star \subseteq a \text{ by Lemma 6.2.6.(3) } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \ .
\end{aligned}
$$

Statement (6). By Lemma 6.2.6.(3) we get $(u \cap a) \cdot (N \cdot M)^\star \subseteq a$, which is equivalent to the equality

$$
\begin{aligned}
& \mathsf{O} \\
= \quad & \wr \text{ Boolean algebra rules } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cap \overline{a} \\
= \quad & \wr \text{ since } (a, b) \text{ is a bipartition we have } \overline{a} = b \ \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cap b \ .
\end{aligned}
$$

We now calculate as before in Statement (5) to obtain:

$$
\begin{aligned}
& b \cap r \\
= \quad & \wr \text{ same computations as in the proof of Statement (5) } \wr \\
& (b \cap (u \cap a) \cdot (N \cdot M)^\star) \cup (b \cap (u \cap a) \cdot (N \cdot M)^\star \cdot N) \\
= \quad & \wr \text{ first intersection is } \mathsf{O}\text{, see above } \wr \\
& b \cap (u \cap a) \cdot (N \cdot M)^\star \cdot N \\
= \quad & \wr \ (u \cap a) \cdot (N \cdot M)^\star \cdot N \subseteq b \text{ by Lemma 6.2.6.(4) } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot N \ .
\end{aligned}
$$

With these prerequisites we can show that c is in fact a vertex cover. To show this, we first show that

$$(a \cap r) \cdot E \subseteq b \cap r \quad \text{and} \quad (b \cap \overline{r}) \cdot E \subseteq a \cap \overline{r}$$

hold. Then we can use Equation (6.2.5.v) to see that

$$\overline{c} \cdot E = (a \cap r) \cdot E \cup (b \cap \overline{r}) \cdot E \subseteq (b \cap r) \cup (a \cap \overline{r}) = c \ .$$

We elaborate these computations in the following proposition.

6.2.7 Proposition (c is a vertex cover)**.**
The following hold:

(1) $(a \cap r) \cdot E = b \cap r$.

(2) $(b \cap \overline{r}) \cdot E \subseteq a \cap \overline{r}$.

(3) c is a vertex cover. In particular this proves Theorem 6.1.2.(i).

Proof. Statement (1). We compute as follows:

$$\begin{aligned}
& (a \cap r) \cdot E \\
= & \quad \wr \text{ by Lemma 6.2.6.(5) } \wr \\
& (u \cap a) \cdot (N \cdot M)^{\star} \cdot E \\
= & \quad \wr \text{ by Lemma 6.2.6.(2) } \wr \\
& (u \cap a) \cdot (N \cdot M)^{\star} \cdot N \\
= & \quad \wr \text{ by Lemma 6.2.6.(6) } \wr \\
& b \cap r\ .
\end{aligned}$$

Statement (2). We first note the following equivalence:

$$\begin{aligned}
& (b \cap \overline{r}) \cdot E \subseteq a \cap \overline{r} \\
\iff & \quad \wr \text{ since } \overline{a} = b \text{ and } \overline{b} = a \ \wr \\
& (\overline{a} \cap \overline{r}) \cdot E \subseteq \overline{b} \cap \overline{r} \\
\iff & \quad \wr \text{ de Morgan rule and } E = E^{\top} \ \wr \\
& \overline{a \cup r} \cdot E^{\top} \subseteq \overline{b \cup r} \\
\iff & \quad \wr \text{ Schröder equivalence } \wr \\
& (b \cup r) \cdot E \subseteq a \cup r\ .
\end{aligned}$$

We show that $(b \cup r) \cdot E \subseteq a \cup r$ holds, which by the above argument yields the desired inclusion. We have:

$$\begin{aligned}
& (b \cup r) \cdot E \\
= & \quad \wr \text{ composition distributes over unions } \wr \\
& b \cdot E \cup r \cdot E \\
\subseteq & \quad \wr\ (a, b) \text{ is a bipartition, thus } \textbf{true} \iff b \cdot E \cap b = \mathsf{O} \iff b \cdot E \subseteq \overline{b} = a \ \wr \\
& a \cup r \cdot E \\
= & \quad \wr \text{ since } a = \overline{b} \text{ we have } (a \cap r) \cup (b \cap r) = (a \cup b) \cap r = \mathsf{L} \cap r = r \ \wr \\
& a \cup ((a \cap r) \cup (b \cap r)) \cdot E \\
= & \quad \wr \text{ composition distributes over unions } \wr \\
& a \cup (a \cap r) \cdot E \cup (b \cap r) \cdot E \\
= & \quad \wr \text{ by Proposition 6.2.7.(1) } \wr
\end{aligned}$$

$$
\begin{aligned}
& a \cup (b \cap r) \cup (b \cap r) \cdot E \\
\subseteq \quad & \{ \text{ since } b \cap r \subseteq b \text{ and composition is monotonic } \} \\
& a \cup (b \cap r) \cup b \cdot E \\
\subseteq \quad & \{ (a, b) \text{ is a bipartition, thus } \textbf{true} \iff b \cdot E \cap b = \mathsf{O} \iff b \cdot E \subseteq \overline{b} = a \} \\
& a \cup (b \cap r) \\
= \quad & \{ \text{ distributive law } \} \\
& (a \cup b) \cap (a \cup r) \\
= \quad & \{ \text{ since } (a, b) \text{ is a bipartition, we have } a \cup b = \mathsf{L} \text{ and } \mathsf{L} \text{ is the largest element } \} \\
& a \cup r \ .
\end{aligned}
$$

Statement (3). We have:

$$
\begin{aligned}
& \overline{c} \cdot E \\
= \quad & \{ \text{ by Equation (6.2.5.v) } \} \\
& ((a \cap r) \cup (b \cap \overline{r})) \cdot E \\
= \quad & \{ \text{ composition distributes over unions } \} \\
& (a \cap r) \cdot E \cup (b \cap \overline{r}) \cdot E \\
\subseteq \quad & \{ \text{ by Proposition 6.2.7.(1) } \} \\
& (b \cap r) \cup (b \cap \overline{r}) \cdot E \\
\subseteq \quad & \{ \text{ by Proposition 6.2.7.(2) } \} \\
& (b \cap r) \cup (a \cap \overline{r}) \\
= \quad & \{ \text{ commutativity of } \cup \text{ and definition of } c \text{, Equation (6.2.5.iv) } \} \\
& c \ .
\end{aligned}
$$

//

Note that the previous result is true for any matching, not just a maximum one. For the remainder of the section we assume that M is a maximum matching. This precondition is equivalent to a condition that is of crucial importance for the cardinality computation of c. We note the following chain of equivalences:

$$
\begin{aligned}
& \textbf{true} \\
\iff \quad & \{ \text{ assumption } \} \\
& M \text{ is a maximum matching} \\
\iff \quad & \{ \text{ by Lemma 5.1.4 } \} \\
& \text{there is no } M\text{-augmenting path} \\
\iff \quad & \{ \text{ by Proposition 5.2.4 } \} \\
& u \cdot (N \cdot M)^{\star} \cdot N \cap u = \mathsf{O} \\
\iff \quad & \{ \text{ Boolean algebra } \} \\
& u \cdot (N \cdot M)^{\star} \cdot N \subseteq \overline{u} \ .
\end{aligned}
$$

The last inclusion allows the following estimate:

$$
\begin{aligned}
& b \cap r \\
= \quad & \wr \text{ by Lemma 6.2.6.(6) } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot N \\
\subseteq \quad & \wr \text{ since } u \cap a \subseteq u \text{ and the monotonicity of the composition } \wr \\
& u \cdot (N \cdot M)^\star \cdot N \\
\subseteq \quad & \wr\ M \text{ is a maximum matching, see above } \wr \\
& \overline{u}\ . \qquad (6.2.7.i)
\end{aligned}
$$

Note that instead of using Proposition 5.2.4 to obtain the above estimate, we can also use Equation (6.2.1). The latter has the benefit that a purely relation-algebraic proof of this fact is provided by Berghammer, Danilenko, Höfner, and Stucke [BDHS16] as we have discussed in the beginning of this section.

It turns out that we also have $a \cap \overline{r} \subseteq \overline{u}$, which yields that

$$c = (a \cap \overline{r}) \cup (b \cap r) \subseteq \overline{u}$$

holds. We now prove this result.

6.2.8 Lemma (Matched vertices)**.**
We have $c \subseteq \overline{u} = \mathsf{L}_{\mathbb{1},V} \cdot M$.

Proof. First, we note the following estimate:

$$
\begin{aligned}
& u \cap a \\
= \quad & \wr\ \mathsf{I} \text{ is neutral with respect to } \cdot \wr \\
& (u \cap a) \cdot \mathsf{I} \\
\subseteq \quad & \wr\ \mathsf{I} \subseteq (N \cdot M)^\star \text{ by Definition 3.2.1.}\mathit{(KA2)} \text{ and the monotonicity of the union } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \\
\subseteq \quad & \wr\ \mathsf{I} \text{ is neutral with respect to } \cdot, \mathsf{I} \subseteq \mathsf{I} \cup N, \text{ and } \cdot \text{ is monotonic } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot (\mathsf{I} \cup N) \\
= \quad & \wr \text{ definition of } r, \text{ Equation (6.2.5.iii) } \wr \\
& r\ . \qquad (6.2.8.a)
\end{aligned}
$$

This yields the following chain of equivalences:

$$
\begin{aligned}
& \textbf{true} \\
\iff \quad & \wr \text{ by Equation (6.2.8.a) } \wr \\
& u \cap a \subseteq r \\
\iff \quad & \wr \text{ Boolean algebra: } a \sqcap b \sqsubseteq c \iff b \sqcap \overline{c} \sqsubseteq \overline{a} \wr \\
& a \cap \overline{r} \subseteq \overline{u} \qquad (6.2.8.b)
\end{aligned}
$$

Thus we find

$$
\begin{aligned}
& c \\
= \quad & \wr \text{ definition of } c\text{, Equation (6.2.5.iv) } \wr \\
& (a \cap \overline{r}) \cup (b \cap r) \\
\subseteq \quad & \wr \text{ by Equation (6.2.8.b) } \wr \\
& \overline{u} \cup (b \cap r) \\
\subseteq \quad & \wr \text{ by Equation (6.2.7.i) } \wr \\
& \overline{u} \cup \overline{u} \\
= \quad & \wr \text{ idempotence of the union } \wr \\
& \overline{u} \\
= \quad & \wr \text{ definition of } u\text{, Equation (6.2.5.ii), and the complement is an involution } \wr \\
& \mathsf{L}_{\mathbb{1},V} \cdot M \,.
\end{aligned}
$$

Recall that our goal is to show that

$$2 \cdot |c| = |M|$$

holds. To that end we will show that the vertices covered by M can be decomposed into c and $c \cdot M$ We have seen a similar result in Equation (6.2.5.a) during the proof of the first statement of Kőnig's theorem. In case of a maximum matching, we get not only that

$$\mathsf{L}_{\mathbb{1},V} \cdot M = c \cdot M \cup c \,,$$

but also that

$$c \cdot M \cap c = \mathsf{O} \,.$$

The latter condition will allow us to compute the cardinality of c with the tools of Proposition 6.2.4.

6.2.9 Proposition (Disjoint union)**.**
The following hold:

(1) $c \cdot M \cap c = \mathsf{O}$.

(2) $c \cdot M \cup c = \overline{u}$.

Proof. Statement (1). First, we get the following inclusion:

$$
\begin{aligned}
& r \cdot M \\
= \quad & \wr \text{ definition of } r\text{, Equation (6.2.5.iii) } \wr \\
& (u \cap a) \cdot (N \cdot M)^{\star} \cdot (\mathsf{I} \cup N) \cdot M \\
= \quad & \wr \text{ composition distributes over unions and } \mathsf{I} \cdot M = M \wr \\
& (u \cap a) \cdot (N \cdot M)^{\star} \cdot (M \cup N \cdot M) \\
= \quad & \wr \text{ composition distributes over unions } \wr
\end{aligned}
$$

$$
\begin{aligned}
& (u \cap a) \cdot (N \cdot M)^\star \cdot M \cup (u \cap a) \cdot (N \cdot M)^\star \cdot (N \cdot M) \\
= \quad & \wr \text{ by Lemma 6.2.6.(1), since } u \cap a \subseteq u \text{ } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot N \cup (u \cap a) \cdot (N \cdot M)^\star \cdot (N \cdot M) \\
\subseteq \quad & \wr \; x^\star \cdot x \leqslant 1 + x^\star \cdot x = x^\star \text{ holds in every Kleene algebra } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot N \cup (u \cap a) \cdot (N \cdot M)^\star \\
= \quad & \wr \; \mathsf{I} \text{ is the compositional identity, composition distributes over unions } \wr \\
& (u \cap a) \cdot (N \cdot M)^\star \cdot (N \cup \mathsf{I}) \\
= \quad & \wr \text{ commutativity of the union and definition of } r\text{, Equation (6.2.5.iii) } \wr \\
& r \, . \qquad\qquad (6.2.9.a)
\end{aligned}
$$

This yields

$$
\begin{aligned}
& \textbf{true} \\
\Longleftrightarrow \quad & \wr \text{ by Equation (6.2.9.a) } \wr \\
& r \cdot M \subseteq r \\
\Longleftrightarrow \quad & \wr \text{ Schröder equivalence and } M = M^\top \text{ } \wr \\
& \overline{r} \cdot M \subseteq \overline{r} \, . \qquad\qquad (6.2.9.b)
\end{aligned}
$$

We thus get:

$$
\begin{aligned}
& (a \cap \overline{r}) \cdot M \\
= \quad & \wr \; M \text{ is injective } \wr \\
& a \cdot M \cap \overline{r} \cdot M \\
\subseteq \quad & \wr \; a \cdot M \subseteq a \cdot E \subseteq \overline{a} = b\text{, since } (a, b) \text{ is a bipartition } \wr \\
& b \cap \overline{r} \cdot M \\
\subseteq \quad & \wr \text{ by Equation (6.2.9.b) } \wr \\
& b \cap \overline{r} \, . \qquad\qquad (6.2.9.c)
\end{aligned}
$$

Now we calculate as follows:

$$
\begin{aligned}
& c \cdot M \\
= \quad & \wr \text{ definition of } c\text{, Equation (6.2.5.iv) } \wr \\
& \big((a \cap \overline{r}) \cup (b \cap r)\big) \cdot M \\
= \quad & \wr \text{ composition distributes over unions } \wr \\
& (a \cap \overline{r}) \cdot M \cup (b \cap r) \cdot M \\
\subseteq \quad & \wr \text{ by Equation (6.2.9.c) } \wr \\
& (b \cap \overline{r}) \cup (b \cap r) \cdot M \\
= \quad & \wr \; M \text{ is injective } \wr \\
& (b \cap \overline{r}) \cup (b \cdot M \cap r \cdot M) \\
\subseteq \quad & \wr \text{ since } b \cdot M \subseteq b \cdot E \subseteq a \text{ and } r \cdot M \subseteq r \text{ by Equation (6.2.9.a) } \wr
\end{aligned}
$$

$$
\begin{aligned}
& (b \cap \overline{r}) \cup (a \cap r) \\
= \quad & \{ \text{ commutativity of } \cup \} \\
& (a \cap r) \cup (b \cap \overline{r}) \\
= \quad & \{ \text{ by Equation (6.2.5.v) } \} \\
& \overline{c} \ .
\end{aligned}
$$

Statement (2). We have the following chain of inclusions:

$$
\begin{aligned}
& \mathsf{L}_{\mathbb{1},V} \cdot M \\
\subseteq \quad & \{ \text{ by Equation (6.2.5.a), since } c \text{ is a vertex cover by Proposition 6.2.7.(3) } \} \\
& c \cdot M \cup c \\
\subseteq \quad & \{ \text{ by Lemma 6.2.8 } \} \\
& c \cdot M \cup \mathsf{L}_{\mathbb{1},V} \cdot M \\
\subseteq \quad & \{ \text{ since } c \subseteq \mathsf{L}_{\mathbb{1},V} \} \\
& \mathsf{L}_{\mathbb{1},V} \cdot M \cup \mathsf{L}_{\mathbb{1},V} \cdot M \\
= \quad & \{ \text{ idempotence of the union } \} \\
& \mathsf{L}_{\mathbb{1},V} \cdot M \ .
\end{aligned}
$$

This yields

$$\mathsf{L}_{\mathbb{1},V} \cdot M = c \cdot M \cup c$$

and since $\mathsf{L}_{\mathbb{1},V} \cdot M = \overline{u}$ by Equation (6.2.5.ii), we obtain the desired result. //

Now we can finally prove the last statement of Kőnig's theorem.

6.2.10 Proof (Proof of Theorem 6.1.2.(ii))**.**

Proof. Since M is symmetric, we get

$$c \subseteq \mathsf{L}_{\mathbb{1},V} \cdot M = \mathsf{L}_{\mathbb{1},V} \cdot M^{\top}$$

by Lemma 6.2.8. Additionally, M is univalent and injective, which by Proposition 6.2.4.(4) yields

$$|c \cdot M| = |c| \ .$$

We then have the following chain of equalities:

$$
\begin{aligned}
& |M| \\
= \quad & \{ \text{ by Proposition 6.2.4.(5), because } M \text{ is univalent and injective } \} \\
& |\mathsf{L}_{\mathbb{1},V} \cdot M| \\
= \quad & \{ \text{ by Proposition 6.2.9.(2) } \} \\
& |c \cdot M \cup c| \\
= \quad & \{ \text{ by Definition 6.2.2.} (C4) \} \\
& |c \cdot M| + |c| - |c \cdot M \cap c| \\
= \quad & \{ \text{ by Proposition 6.2.9.(1) } \}
\end{aligned}
$$

$$
\begin{aligned}
& |c \cdot M| + |c| - |\mathsf{O}| \\
= \quad & \{ \text{ by Definition 6.2.2.}(C1) \} \\
& |c \cdot M| + |c| \\
= \quad & \{ |c \cdot M| = |c|, \text{ see above } \} \\
& |c| + |c| \ .
\end{aligned}
$$

6.3 The Bipartite Minimum Vertex Cover Function

In this section we present an implementation of a function that computes a minimum vertex cover in a bipartite graph. Our maximum matching function in Section 5.3 did not check whether its argument is bipartite or not. This test was omitted due to the fact that the computation of a maximum matching does not depend on an actual bipartition of the graph, but on the mere assurance that there exists a bipartition. The construction of a minimum vertex cover in a bipartite graph from Theorem 6.1.2, however, does depend on a bipartition of the graph. We thus reuse our functions from Section 5.5 to obtain a bipartition and thus to ensure that the input graph is bipartite.

As we have stated before, Kőnig's theorem, Theorem 6.1.2, provides an explicit construction for a minimum vertex cover. We follow this definition as closely as possible. First, one computes a maximum matching M and its complement $E \setminus M$. This is precisely what our functions *maximumMatchingAndComplement* from Section 5.3 and *maximumMatchingAndComplementHK* from Section 5.4 compute. The second step in Kőnig's theorem is the computation of

$$r := (\mathsf{uncovered}(M) \cap a) \cdot ((E \setminus M) \cdot M)^{\star} \cdot (\mathsf{I} \cup (E \setminus M)) \ ,$$

where (a, b) is a bipartition of the graph. Recall that this is the set of those vertices that are reachable along an alternating path starting in $\mathsf{uncovered}(M) \cap a$. Given a bipartition, the matching and its complement, we can compute r by using our function *reachableWith* from Section 4.4 to achieve this in the following fashion.

```
alternatingReachable :: Mat () → Mat () → Vec () → Vec ()
alternatingReachable m cm a = s ∪| (s ⊜ cm) where
  s = reachableWith (⊜) (uncovered m ∩| a) [cm, m]
```

The result vector of the above function is essentially the definition of r, but we used the distributive law to avoid dealing with the identity matrix explicitly.

Finally, we can combine these functions to get a minimum vertex cover in case the graph is bipartite as follows.

```
minimumVertexCover :: Mat () → Maybe (Vec ())
minimumVertexCover graph = fmap makeCover (findBipartition graph) where
  makeCover (a, b) = (a \ r) ∪| (b ∩| r) where
```

$r = \mathit{uncurry\ alternatingReachable}\ (\mathit{maximumMatchingHK'}\ \mathit{graph})\ a$

We have reused the set functions $(\cup_{|})$ from Section 4.3.5 and $(\backslash)$, $(\cap_{|})$ from Section 4.4. Just as discussed in Section 4.4 we used the relative complement function $(\backslash)$ to compute $a \cap \overline{r} = a \backslash r$, which is possible without the notion of a greatest element. Suppose that the graph from Figure 6.1 is represented in Haskell by $\mathit{vcGraph} :: \mathit{Mat}\ ()$. We then get the following result.

```
ghci> minimumVertexCover vcGraph
Just ((0 | ()) (3 | ()) (5 | ()) (7 | ()))
```

The above example is exactly the vertex cover from Figure 6.1(b).

The above function *minimumVertexCover* is a particularly good example of the benefit that comes with (relation-)algebraic computations, because many of the underlying expressions are easily translated into our graph framework. Our implementation is actually an executable version of Kőnig's theorem, where the relational model allows a short and concise implementation. Note that the resulting function is as complex as computing a maximum matching. In fact, the computation of both a bipartition and r are at worst quadratic in the number of vertices of the graph, and all set operations are linear in the number of vertices. Thus all these complexities are subsumed by the complexity of the matching computation.

6.4 The Non-Bipartite Case

As we have mentioned already in the very beginning of this chapter, the general vertex cover problem is known to be NP-complete [Kar72]. In general graphs one can no longer use Kőnig's construction of a minimum vertex cover, because said construction depends on a bipartition of the graph (both in terms of its definition and its properties). While this only shows that *this* construction does not work in non-bipartite graphs, it still hints at the fact that the problem becomes more complex in the case of arbitrary graphs.

There exist approximation algorithms for vertex covers in general graphs. For example, the algorithm of Gavril and Yannakakis can be used to find a vertex cover that has at most twice as many vertices as a minimum vertex cover in polynomial time [CLRS01]. The idea behind this algorithm is to incrementally increase an initially empty set of vertices C by adding both endpoints of edges, as long as there are edges that do not touch any vertices in C. Once no more such edges are left, C is obviously a vertex cover. The cardinality estimate is based upon the fact that the edges added to the C constitute a matching. We refer to the textbook of Conway, Leiserson, Rivest, and Stein [CLRS01] for more detail.

The algorithm of Gavril and Yannakakis can be implemented very simply with our means. Recall that a vector $c : \mathbb{1} \leftrightarrow V$ satisfies the following equivalences:

$$c \text{ is a vertex cover}$$

$$\begin{aligned} \iff & \quad \wr \text{ by Definition 6.1.1 } \wr \\ & \overline{c} \cdot E \subseteq c \\ \iff & \quad \wr \text{ Boolean algebra rules } \wr \\ & \overline{c} \cdot E \cap \overline{c} = \mathsf{O} \,. \end{aligned}$$

In particular, this yields that

$$c \text{ is not a vertex cover} \iff \overline{c} \cdot E \cap \overline{c} \neq \mathsf{O} \,. \tag{6.4.i}$$

We can use this equivalence for an implementation as follows.

```
vertexCoverApprox :: Mat () → Vec ()
vertexCoverApprox e = approximate (verticesWith 0 e) where
  approximate notC | isEmptyVec cands = emptyVec
                   | otherwise        = new ∪| approximate (notC \ new)
                     where cands = notC ⊙← e ∩| notC
                           new   = toVec [x, y]
                           (x, y) = head (unVec cands)
(⊙←) :: Vec Vertex → Mat α → Vec Vertex
(⊙←) = vecMatMult leftmostUnion (sMultWith (λi _ _ → i))
```

In every iteration we check whether the right hand side of Equation (6.4.i) holds. If it does not, then the vertex set is already a vertex cover. If it does, we pick a vertex from the non-empty intersection. The newly defined vector-matrix multiplication $(\odot_{\leftarrow})$ marks all successors of a vertex set with some predecessor in the vertex set. Thus picking (w, v) from the above intersection yields a vertex and its predecessor. Since w is contained in *notC*, it is not yet covered by the the approximation result. Also, w is reached from *notC* via an edge in e, thus v is also not covered yet and we have found an edge in e that does not touch any vertex in the approximation result. We then add the vertices to the result of the approximation, remove them from its complement and continue the approximation.

The above function over-estimates the number of necessary vertices for a vertex cover. For instance, in the non-bipartite graph from Figure 5.3 the function *vertexCoverApprox* yields the vertex cover $C = \{0, 1, 2, 3\}$. Clearly, this is a vertex cover, but $C' := \{1, 2, 3\}$ is another vertex cover with a strictly smaller cardinality.

We omit the correctness proof of the above program. A purely relational version of both, the above algorithm and the proof of its correctness has been published by Berghammer, Höfner, and Stucke [BHS16].

Chapter 7

Maximum Flows and Minimum Cuts in Networks

The maximum flow problem and its dual, the minimum cut problem, are classical graph-theoretic problems. In both cases one considers a graph with edge weights and two distinct vertices called *source* and *sink*. A maximum flow is then a function that sends as many units as possible from the source to the sink, while at the same time being bounded from above by the edge weights and maintaining the so-called *Kirchhoff condition* that the amount of input flow equals the amount of the output flow in every vertex or, more intuitively: what goes in must come out. This problem is related to various resource allocation tasks including different kinds of traffic. The minimum cut problem on the other hand asks for a partition of the vertices into two sets, such that the first set contains the source, the second one contains the sink, and the sum of all capacities of the edges between these two sets is minimal.

Both problems have well-known solutions and we only deal with the implementation of these solutions in our framework. The implementation of the maximum flow function bears some similarity to the one that we have presented elsewhere [Dan14a]. However, in the present work we focus on a purely functional implementation, rather than a functional logic one.

The flow problem is also solved in Haskell by Erwig and Miljenovic [EM14]. The solution is based upon the inductive graph definition given by Erwig [Erw01]. The resulting function has some similarity to our implementation, for instance both functions consider the graph to be the actual flow function. We outline the differences in Section 7.4.

7.1 Specification and Solution

The maximum flow and minimum cut problems are related more closely than the matching and vertex cover problems, which is why we deal with both at the same time. The main link between these problems is the Ford-Fulkerson theorem, Theorem 7.1.4.

We begin with the necessary definitions and some examples. The problem we consider is formulated for a special type of graph called a network[1], which is

[1] In the literature this kind of graph is sometimes called an *oriented* network. Since we consider only this kind of graphs, we omit the additional modifier for simplicity.

essentially an asymmetric, weighted graph.

7.1.1 Definition (Network).
Let (V, E) be a finite graph such that $E \cap E^{\top} = \mathsf{O}$ holds and let $s, t \in V$ such that $s \neq t$ and $c : E \to \mathbb{Q}_{\geqslant 0}$. Then the quintuple $N := (V, E, s, t, c)$ is called *network with source s, sink t and capacity c*. ⫽

We proceed with the definition of flows and flow values.

7.1.2 Definition (Balance, flow, flow value).
Let $N = (V, E, s, t, c)$ be a network. For every function $g : E \to \mathbb{R}$ we define

$$\mathsf{bal}(g) : V \to \mathbb{R}\,, \quad v \mapsto \left(\sum_{w \in \{v\} \cdot E} g(v, w)\right) - \left(\sum_{w \in \{v\} \cdot E^{\top}} g(w, v)\right)$$

and call $\mathsf{bal}(g)$ the *balance of g*. A function $f : E \to \mathbb{R}_{\geqslant 0}$ is called a *flow in N* if and only if the following two conditions hold:

(1) $f \leqslant c$ (pointwise[2]) *(validity)*

(2) $\forall\, v \in V \setminus \{s, t\} : \mathsf{bal}(f)(v) = 0$. *(Kirchhoff law)*

If f is a flow, we define $|f| := \mathsf{bal}(f)(s)$ and call $|f|$ the *flow value of f*. We define

$$\mathsf{flow}(N) := \sup \{\, |h| \mid h : E \to \mathbb{Q}_{\geqslant 0} \text{ flow} \,\}$$

and call a flow f a *maximal flow in N* if and only if $|f| = \mathsf{flow}(N)$ holds. ⫽

The value $\mathsf{flow}(N)$ is well-defined. One can easily show that the set

$$F := \left\{ f \in (\mathbb{R}_{\geqslant 0})^E \;\middle|\; f \text{ flow} \right\}$$

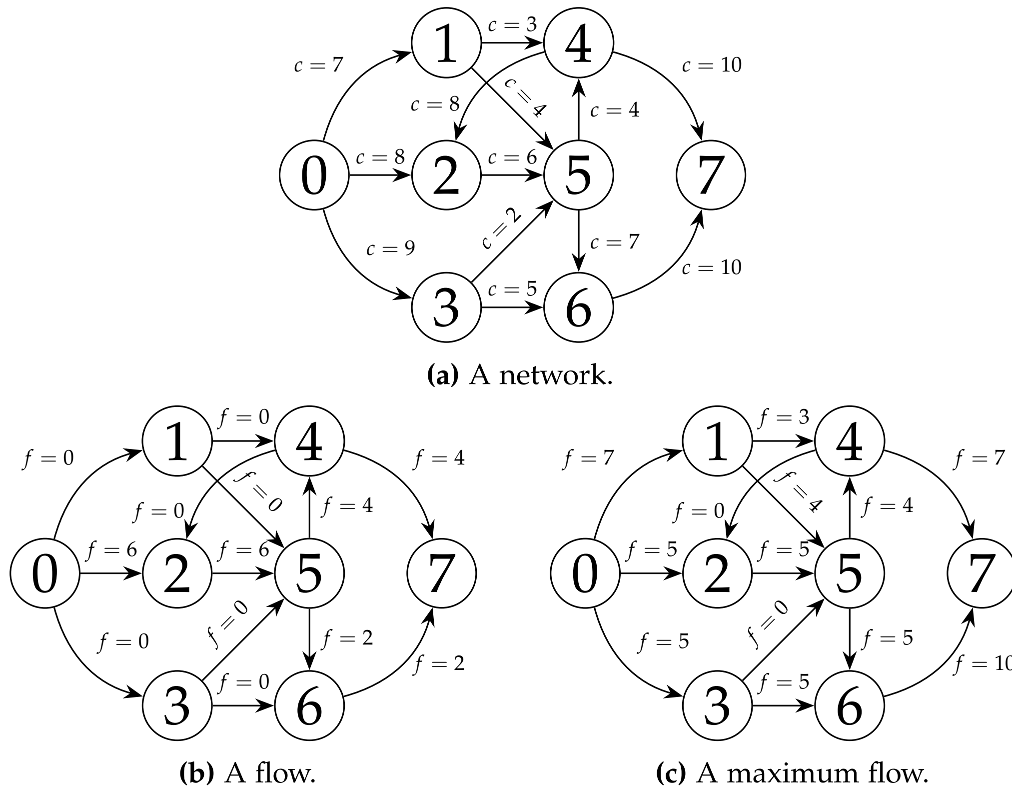
is a non-empty, topologically closed and bounded set in the finite-dimensional normed space $(\mathbb{R}^E, \|-\|_\infty)$, where $\|-\|_\infty$ is the supremum norm. Since $\mathbb{R}^E$ is isomorphic to $\mathbb{R}^{|E|}$, we get that F is compact. But then the function

$$\varphi : \mathbb{R}^E \to \mathbb{R}\,, \quad f \mapsto |f|$$

is clearly continuous and the Weierstraß theorem yields that the function takes on a maximum in F, which then in turn is a maximum flow. The above proof outline is a variation of a proof by Gallier [Gal11] and well-known in graph theory.

In Figure 7.1 we see an example network, as well as two flows in that network, one of which is a maximum flow. Since the maximality condition for maximum flows is required with respect to a non-injective function, maximum flows are not necessarily unique. Figure 7.2 shows an example of a network, in which there are two distinct maximum flows.

[2] In other words we have: $\forall\, e \in E : f(e) \leqslant c(e)$.



(a) A network.

(b) A flow.

(c) A maximum flow.

Figure 7.1. Flows in a network, where $s = 0$ and $t = 7$.

Just as was the case with the bipartite maximum matching problem, there exists an elegant characterisation of maximum flows. This result is due to Ford and Fulkerson and is based upon another type of augmenting path.

7.1.3 Definition (Residual capacity, residual network, augmenting path)**.**
Let $N = (V, E, s, t, c)$ be a network and $f : E \to \mathbb{R}_{\geqslant 0}$ be a flow in N. Set

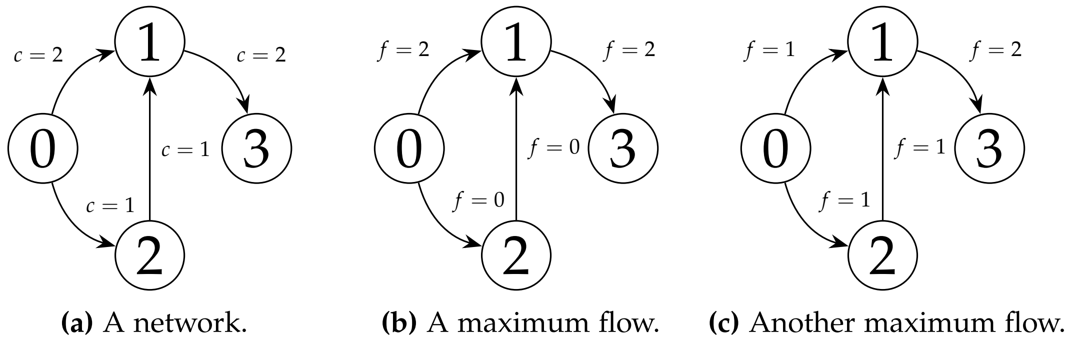
$$c_f : E \cup E^\top \to \mathbb{R}\,, \quad (v, w) \mapsto \begin{cases} c(v,w) - f(v,w) & : (v,w) \in E \\ f(w,v) & : \text{otherwise}\,. \end{cases}$$

The function c_f is called *residual capacity (with respect to f)* and is well-defined due to the asymmetry of E. We define

$$E_f := \left\{ e \in E \cup E^\top \;\middle|\; c_f(e) > 0 \right\}$$

and call $N_f := (V, E_f, s, t, c_f)$ the *residual network (with respect to f)*. A path in N_f from s to t is called f-augmenting path. ⫽

Intuitively, the residual network adds new edges, where there is already a non-zero flow and removes edges, for which the flow value is as high as possible, which is to say $f(e) = c(e)$ holds. The residual network and particularly augmenting paths can be used to either find that a flow is a maximum flow or to increase its flow value.



(a) A network. **(b)** A maximum flow. **(c)** Another maximum flow.

Figure 7.2. Maximum flows in a network, where $s = 0$ and $t = 3$.

For every path $p \in V^*$ let

$$A(p) := \left\{ e \in E \cup E^\top \;\middle|\; \exists\, i \in \mathbb{N}_{<|p|-1} : e = (p_i, p_{i+1}) \right\} .$$

The following characterisation is due to Ford and Fulkerson [FF62].

7.1.4 Theorem (Characterisation of maximum flows, Ford & Fulkerson 1962)**.**
Let $N = (V, E, s, t, c)$ be a network and $f : E \to \mathbb{R}_{\geqslant 0}$ a flow in N. Then the following hold:

(1) If there is no f-augmenting path, then f is a maximum flow.

(2) Let $p \in V^*$ be an f-augmenting path, and set $\varepsilon := \min\{ c_f(e) \mid e \in A(p) \}$. Then

$$f_p : E \to \mathbb{R}_{\geqslant 0}\,, \quad (v, w) \mapsto \begin{cases} f(v,w) + \varepsilon & : (v,w) \in A(p) \cap E \\ f(v,w) - \varepsilon & : (v,w) \in A(p)^\top \cap E \\ f(v,w) & : \textit{otherwise} \end{cases}$$

is a flow in N and we have $|f_p| = \varepsilon + |f| > |f|$.

In particular, f is a maximum flow, if and only if there is no f-augmenting path. ⫽

Note that this theorem is very similar to the Berge lemma, Lemma 5.1.4. This similarity is not a coincidence, since maximum matchings in bipartite graphs can be solved using a special flow. We discuss the corresponding construction in a moment. As before, the above theorem not only provides a simple test for checking, whether a flow is a maximum flow, but also yields an algorithmic procedure to find a maximum flow: beginning with the constant zero flow we can search for augmenting paths, compute the new flow and repeat this procedure until no paths are left. However, there are interesting restrictions to this simple algorithm, which is known as the Ford-Fulkerson algorithm. The first three results are due to Ford and Fulkerson [FF62] as well.

(1) If $c : E \to \mathbb{Q}_{\geqslant 0}$, then the above Ford-Fulkerson algorithm terminates. Its complexity is not necessarily polynomial in the graph size.

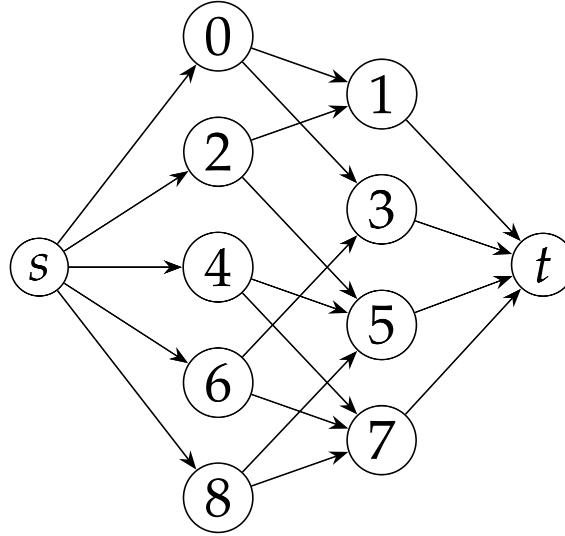


Figure 7.3. The network in case of the graph from Figure 5.2. All edges have capacity 1.

(2) If $c : E \to \mathbb{N}$, then for every maximum flow f we have $f(E) \subseteq \mathbb{N}$. In particular, the maximum flow has a natural value.

(3) In the general case, the procedure may not terminate.

(4) If a shortest path is chosen in every iteration, the algorithm terminates and is polynomial in the graph size.

The latter variation is known as the Edmonds-Karp algorithm [EK72]. We focus on this variant for two reasons: first, it is a polynomial algorithm and second, it is actually simpler to implement with our means.

It is a known fact that the bipartite maximum matching can be solved using a special flow instance [KT06]. Suppose that $G = (V, E)$ is a finite, symmetric graph with a bipartition (A, B). Now let s, t be objects such that $s, t \notin V$ and define $V_{\text{flow}} := V \cup \{ s, t \}$ and

$$E_{\text{flow}} := (\{ s \} \times A) \cup (E \cap (A \times B)) \cup (B \times \{ t \}) .$$

as well as $c : E_{\text{flow}} \to \mathbb{R}_{\geqslant 0}$, $e \mapsto 1$. Then $N := (V_{\text{flow}}, E_{\text{flow}}, s, t, c)$ is a network. A visualisation of this construction is given in Figure 7.3. Now let f be a maximum flow in the network N. Note that the flow in every edge is a natural number, thus $f(e) = 0$ or $f(e) = 1$ holds for all $e \in E_{\text{flow}}$. Then set

$$P := \{ e \in A \times B \mid f(e) = 1 \} = f^{-1}(\{ 1 \}) \cap (A \times B) .$$

Since $P \subseteq A \times B$ and (A, B) is a partition of V, we immediately find that

$$P \cdot P = \mathsf{O} \quad \text{and} \quad P^\top \cdot P^\top = \mathsf{O}$$

hold. We claim that we also have

$$P^\top \cdot P \subseteq \mathsf{I} \quad \text{and} \quad P \cdot P^\top \subseteq \mathsf{I} .$$

Let $x, y \in V$ such that $(x, y) \in P^\top \cdot P$. Then there exists a $z \in V$ such that $(x, z) \in P^\top$ and $(z, y) \in P$, thus $(z, x), (z, y) \in P$, which yields $x, y \in \{ z \} \cdot E_{\text{flow}}$. In particular, we have $z \in A$. Since f is a flow and $z \in A \subseteq V_{\text{flow}} \setminus \{ s, t \}$, we obtain

$$\begin{aligned}
& 0 \\
= \; & \wr \text{ Kirchhoff law } \wr \\
& \mathsf{bal}(f)(z) \\
= \; & \wr \text{ definition of } \mathsf{bal}\text{, Definition 7.1.2 } \wr \\
& \left(\sum_{w \in \{ z \} \cdot E_{\text{flow}}} f(z, w) \right) - \left(\sum_{w \in \{ z \} \cdot (E_{\text{flow}})^\top} f(w, z) \right) \\
= \; & \wr \text{ by construction: since } z \in A \text{ its only predecessor is } s \wr \\
& \left(\sum_{w \in \{ z \} \cdot E_{\text{flow}}} f(z, w) \right) - f(s, z) \; .
\end{aligned}$$

Thus we have the following inequality:

$$\sum_{w \in \{ x, y \}} f(z, w) \leqslant \sum_{w \in \{ z \} \cdot E_{\text{flow}}} f(z, w) = f(s, z) \leqslant 1 \; .$$

But since $(z, x), (z, y) \in P$, we have $f(z, x) = 1$ and $f(z, y) = 1$. Thus $x = y$ holds and we get $(x, y) \in \mathsf{I}$. A very similar argument yields $P \cdot P^\top \subseteq \mathsf{I}$.

Now set $M := P \cup P^\top$. Clearly, M is symmetric. Also, we have the inclusion

$$\begin{aligned}
& M \cdot M \\
= \; & \wr \; M = P \cup P^\top \text{ by definition } \wr \\
& \left(P \cup P^\top \right) \cdot \left(P \cup P^\top \right) \\
= \; & \wr \text{ composition distributes over unions } \wr \\
& P \cdot P \cup P^\top \cdot P \cup P \cdot P^\top \cup P^\top \cdot P^\top \\
= \; & \wr \; P \cdot P = \mathsf{O} \text{ and } P^\top \cdot P^\top = \mathsf{O} \; \wr \\
& P^\top \cdot P \cup P \cdot P^\top \\
\subseteq \; & \wr \; P^\top \cdot P \subseteq \mathsf{I},\; P \cdot P^\top \subseteq \mathsf{I}\text{, and union is monotonic } \wr \\
& \mathsf{I} \cup \mathsf{I} \\
= \; & \wr \text{ union is idempotent } \wr \\
& \mathsf{I} \, ,
\end{aligned}$$

which shows that M is a matching. The maximality of this matching is due to the maximality of f. We refer to the textbook of Kleinberg and Tardos [KT06] for details on the latter statement.

There are several reasons why we did not use this construction for the solution of the maximum matching problem in Chapter 5. First, this construction requires

a bipartition of the graph. Berge's lemma on the other hand is stated without an explicit bipartition and thus both our maximum matching functions can compute a maximum matching without having to compute a bipartition. Second, even though the construction of a bipartition is comparatively simple as we have shown in Section 5.5.3, the above construction is based upon a new graph. While this graph is not particularly difficult to compute with our means, it still comes with some overhead, which is not required in the original implementation. Finally, the above solution for the matching problem is *extrinsic*, since it requires an additional construction, while the solution based upon Berge's lemma is *intrinsic* and uses the graph alone.

7.2 The Maximum Flow Function

In this section we show how to find augmenting paths in residual graphs and how to construct a flow with a larger value. We have presented the underlying technique before [Dan14a] and now merely adjust said technique to our framework.

For the remainder of this section let $N = (V, E, s, t, c)$ be a network. We slightly modify our view on the capacity and flow functions. Originally, both are functions from E to $\mathbb{R}_{\geqslant 0}$. For simplicity, we extend their domain to $V \times V$, such that these extensions yield 0 for all values in $(V \times V) \setminus E$. For all functions $g, h : V \times V \to \mathbb{R}$ let

$$g \sqcap_{\mathsf{r}} h : V \times V \to \mathbb{R}\ , \quad e \mapsto \begin{cases} h(e) & : g(e) \neq \mathsf{Z} \\ 0 & : \text{otherwise}\ . \end{cases}$$

The function $\sqcap_{\mathsf{r}}$ can be thought of as the left-forgetful intersection of functions, because it returns the value of the second function if the first one is non-zero and zero otherwise. Let us denote by $\oplus, \ominus$ the pointwise addition and subtraction of real-valued functions respectively. Recall that since we consider both flows and capacities to be functions from $V \times V$ to $\mathbb{R}$, we can use matrix notation for these functions and use the scalar multiplication function $\bullet$, which we have defined in Section 2.4. Similarly, we can use the matrix transposition, which we denote[3] by "transpose", so that for all $g : V \times V \to \mathbb{R}$ we get

$$\text{transpose}(g) : V \times V \to \mathbb{R}\ , \quad (y, x) \mapsto g(x, y)\ .$$

Observe that we get the following result:

$$\forall\, f \in (\mathbb{R}_{\geqslant 0})^E : f \text{ flow} \Rightarrow c_f = (c \ominus f) \oplus \text{transpose}(f)\ . \tag{7.2.i}$$

In fact, let $f : E \to \mathbb{R}_{\geqslant 0}$ be a flow and let $v, w \in V$. Then the asymmetry of E yields

[3]Relational transposition can be considered to be a transposition of a Boolean matrix. However, since we consider matrices to be special functions, we have that every matrix is actually a set $\{\, ((x,y), a(x,y)) \mid x \in X \wedge y \in Y \,\}$, where $a : X \times Y \to Z$ is a function. The transposition of such a set, when considered as a relation, is $\{\, (a(x,y), (x,y)) \mid x \in X \wedge y \in Y \,\}$, which is different from the transposed matrix $\{\, ((y,x), a(x,y)) \mid x \in X \wedge y \in Y \,\}$.

that at most one of the values $f(v,w)$ and $f(w,v)$ is non-zero. We thus compute

$$\begin{aligned}
&((c \ominus f) \oplus \text{transpose}(f))(v,w) \\
= \;& \wr \text{ pointwise definition of } \oplus, \ominus \; \wr \\
&(c(v,w) - f(v,w) + \text{transpose}(f)(v,w) \\
= \;& \wr \text{ definition of transpose } \wr \\
&(c(v,w) - f(v,w)) + f(w,v) \\
= \;& \wr \text{ at most one of } f(v,w) \text{ and } f(w,v) \text{ is non-zero } \wr \\
&\begin{cases} c(v,w) - f(v,w) & : (v,w) \in E \\ f(w,v) & : (v,w) \in E^{\top} \\ 0 & : \text{otherwise} \end{cases} \\
= \;& \wr \text{ extension of the functions to } V \times V \; \wr \\
&c_f(v,w) \; .
\end{aligned}$$

Next, suppose that $f : E \to \mathbb{R}_{\geqslant 0}$ is a flow and that p is an f-augmenting path and set

$$\varepsilon := \min \left\{ c_f(e) \;\middle|\; e \in A(p) \right\} .$$

We define

$$\sigma_p : V \times V \to \mathbb{R} \; , \quad e \mapsto \begin{cases} 1 & : e \in A(p) \\ 0 & : \text{otherwise} \; . \end{cases}$$

Then σ_p is the characteristic function of $A(p)$ in $V \times V$. Finally, let

$$u_p := \varepsilon \bullet \left(c \sqcap_{\mathsf{r}} (\sigma_p \ominus \text{transpose}(\sigma_p))\right) \; . \tag{7.2.ii}$$

Intuitively, u_p is the update function that can be used to improve the flow f to the flow f_p in the Ford-Fulkerson theorem, Theorem 7.1.4, but in point-free notation. We claim the following auxiliary statements, which we prove in Proof A.3.1:

$$\forall\, v,w \in V : c(v,w) \neq 0 \wedge (v,w) \in A(p) \Leftrightarrow (v,w) \in A(p) \cap E \; , \tag{7.2.iii}$$

$$\forall\, v,w \in V : c(v,w) \neq 0 \wedge (v,w) \in A(p)^{\top} \Leftrightarrow (v,w) \in A(p)^{\top} \cap E \; . \tag{7.2.iv}$$

With these statements at hand we can reason as follows. Let $v,w \in V$. Then we get

$$\begin{aligned}
&u_p(v,w) \\
= \;& \wr \text{ definition of } u_p \text{, Equation (7.2.ii) } \wr \\
&\Big(\varepsilon \bullet \left(c \sqcap_{\mathsf{r}} (\sigma_p \ominus \text{transpose}(\sigma_p))\right)\Big)(v,w) \\
= \;& \wr \text{ definition of } \bullet \; \wr \\
&\varepsilon \cdot \Big(\left(c \sqcap_{\mathsf{r}} (\sigma_p \ominus \text{transpose}(\sigma_p))\right)(v,w)\Big) \\
= \;& \wr \text{ definition of } \sqcap_{\mathsf{r}} \; \wr \\
&\varepsilon \cdot \begin{cases} (\sigma_p \ominus \text{transpose}(\sigma_p))(v,w) & : c(v,w) \neq 0 \\ 0 & : \text{otherwise} \end{cases}
\end{aligned}$$

$$= \quad \wr \text{ pointwise definition of } \ominus \text{ and definition of transpose } \wr$$
$$\varepsilon \cdot \begin{cases} \sigma_p(v,w) - \sigma_p(w,v) & : c(v,w) \neq 0 \\ 0 & : \text{otherwise} \end{cases}$$
$$= \quad \wr\ p \text{ is a path, thus at most one of } (v,w) \in A(p) \text{ and } (w,v) \in A(p) \text{ holds } \wr$$
$$\varepsilon \cdot \begin{cases} 1 & : c(v,w) \neq 0 \wedge (v,w) \in A(p) \\ -1 & : c(v,w) \neq 0 \wedge (w,v) \in A(p) \\ 0 & : \text{otherwise} \end{cases}$$
$$= \quad \wr \text{ Equation (7.2.iii) and Equation (7.2.iv) } \wr$$
$$\varepsilon \cdot \begin{cases} 1 & : (v,w) \in A(p) \cap E \\ -1 & : (v,w) \in A(p)^\top \cap E \\ 0 & : \text{otherwise .} \end{cases}$$

Thus we obtain the following statements[4] about f_p and c_{f_p}:

$$f_p = f \oplus u_p \tag{7.2.v}$$
$$c_{f_p} = (c_f \ominus u_p) \oplus \text{transpose}(u_p) \ . \tag{7.2.vi}$$

The statement of Equation (7.2.v) is a trivial consequence of the very definition of f_p in Theorem 7.1.4. The statement of Equation (7.2.vi) can be shown as follows:

$$c_{f_p}$$
$$= \quad \wr\ f_p \text{ is a flow by Theorem 7.1.4 and thus we can use Equation (7.2.i) } \wr$$
$$(c \ominus f_p) \oplus \text{transpose}\left(f_p\right)$$
$$= \quad \wr \text{ property of } f_p \text{ by Equation (7.2.v) } \wr$$
$$\left(c \ominus (f \oplus u_p)\right) \oplus \text{transpose}(f \oplus u_p)$$
$$= \quad \wr \text{ transposition is additive } \wr$$
$$\left(c \ominus (f \oplus u_p)\right) \oplus \text{transpose}(f) \oplus \text{transpose}(u_p)$$
$$= \quad \wr \text{ pointwise definitions and group arithmetics } \wr$$
$$\left((c \ominus f) \ominus u_p\right) \oplus \text{transpose}(f) \oplus \text{transpose}(u_p)$$
$$= \quad \wr \text{ pointwise definitions, addition is commutative, and } x - y = x + \text{inverse}(y) \wr$$
$$\Big(\left((c \ominus f) \oplus \text{transpose}(f)\right) \ominus u_p\Big) \oplus \text{transpose}(u_p)$$
$$= \quad \wr \text{ property of the residual capacity, Equation (7.2.i) } \wr$$
$$(c_f \ominus u_p) \oplus \text{transpose}(u_p) \ .$$

We make use of Equation (7.2.v) and Equation (7.2.vi) to implement a flow augmentation function. The extension of the functions from E to $V \times V$ is used explicitly, because now we can use the graph itself as the capacity function in our implementa-

[4] Again, we use the point-free notation for simplicity and abstraction.

tion. For this purpose we assume that the graph is labelled with values, where each edge value denotes the capacity of that edge. We define a data type for networks for simplicity, which is basically a triple consisting of the capacity matrix, the source, and the sink.

```
data Network v = Network {capacity :: Mat (Number v), source, sink :: Vertex}
```

Before we proceed with the implementation of the flow augmentation and the adjustment of the residual capacity, we need a function that yields an augmenting path in case such a path exists. For the actual application we need not only the path, but also the minimum value of the edge labels along that path. We can accomplish both at the same time. First, we define a data type for the tropical semiring (cf. Example 2.3.4.(4)).

```
data Tropical ω = Min | Max | Weight {weight :: ω}
  deriving Eq

instance Ord ω ⇒ Ord (Tropical ω) where
  compare Min Min = EQ
  compare Min _   = LT
  compare _   Min = GT
  compare Max Max = EQ
  compare _   Max = LT
  compare Max _   = GT
  compare a   b   = comparing weight a b
```

This somewhat technical definition effectively adds two new values to an ordered data type such that *Min* is smaller than all other values and *Max* is larger than all other values. The Haskell function

$$comparing :: Ord\ \beta \Rightarrow (\alpha \to \beta) \to \alpha \to \alpha \to Ordering$$

from the module *Data.Ord* takes a function that maps to a data type that is an instance of the *Ord* type class and compares each two values from the domain of this function by applying the function to both and comparing the results[5]. If in addition to an *Ord* instance, the data type ω in an instance of the Haskell type class *Monoid*, one can define a semiring instance for *Tropical* ω. This definition requires the following expansion property

$$\forall x, y :: \omega\ .\ x \leqslant x\ \text{`mappend`}\ y\ .$$

If this property is satisfied, we can define a semiring instance in the following fashion.

```
instance (Ord ω, Monoid ω) ⇒ Semiring (Tropical ω) where
  zero = Max
  one  = Min
  (⊕) = min
```

[5] This function is Haskell's version of a pullback order. Note that the resulting relation is always a preorder and is an order if and only if the first argument of *comparing* is injective.

$$
\begin{aligned}
&Max \circledast _ &&= Max \\
&_ \quad \circledast Max &&= Max \\
&Min \circledast x &&= x \\
&x \quad \circledast Min &&= x \\
&a \quad \circledast b &&= Weight\ (weight\ a\ \text{`}mappend\text{`}\ weight\ b)
\end{aligned}
$$

The constants are *Max* and *Min* and the addition is the minimum of two values. As for the multiplication, the first two rules yield that *Max* is annihilating, the second two rules make *Min* neutral with respect to $(\circledast)$ and the final rule defines the multiplication to be the monoid operation on all original values.

We can now search for a shortest augmenting path using a special vector-matrix multiplication, which collects both, the path and the sum of all values along this path at the same time. In the special case of the tropical semiring this sum is then exactly the minimum of the values along the path.

$$
\begin{aligned}
&(\odot_{\sim,\sum}) :: Semiring\ \sigma \Rightarrow Vec\ (Path, \sigma) \rightarrow Mat\ \sigma \rightarrow Vec\ (Path, \sigma) \\
&(\odot_{\sim,\sum}) = vecMatMult\ leftmostUnion\ (sMultWith\ (\lambda i\ (p, m)\ y \rightarrow (p \rhd i, y \oplus m)))
\end{aligned}
$$

This function is essentially the vector-matrix multiplication over the product semiring of the path semiring and another semiring. The main difference is that the matrix contains only the values from the second semiring and no values from the first one. Clearly, $(\odot_{\sim,\sum})$ maintains the relational context (cf. Section 4.4). We provide the following instances of our *Number* data type from Section 4.3.1 to present some examples.

instance *Num v* $\Rightarrow$ *Monoid* (*Number v*) **where**
  mempty $= 0$
  mappend $= (+)$

instance *Ord v* $\Rightarrow$ *Ord* (*Number v*) **where**
  compare x y $=$ *compare* (*unNumber x*) (*unNumber y*)

Now suppose that the graph from Figure 7.1(a) is implemented in the constant *cap* :: *Mat* (*Number Double*). Then we get the following results for example applications, where we omit the record notation in the output for better legibility.

ghci> *shortestWith* $(\odot_{\sim,\sum})$ (*toVecWith* $(\langle\rangle, zero)$ [0]) (*toVec* [7]) [*cap*]
$(7 \mid (\langle 0,1,4 \rangle, Weight\ (Number\ 20.0))$

ghci> *shortestWith* $(\odot_{\sim,\sum})$ (*toVecWith* $(\langle\rangle, zero)$ [0]) (*toVec* [7]) [*fmap Weight cap*]
$(7 \mid (\langle 0,1,4 \rangle, Weight\ (Number\ 3.0))$

ghci> *shortestWith* $(\odot_{\sim,\sum})$ (*toVecWith* $(\langle\rangle, zero)$ [4]) (*toVec* [6]) [*cap*]
$(6 \mid (\langle 4,2,5 \rangle, Weight\ (Number\ 21.0))$

ghci> *shortestWith* $(\odot_{\sim,\sum})$ (*toVecWith* $(\langle\rangle, zero)$ [4]) (*toVec* [6]) [*fmap Weight cap*]
$(6 \mid (\langle 4,2,5 \rangle, Weight\ (Number\ 6.0))$

In the above examples we are searching for a path from a start vertex to a target

vertex and the semiring sum of all values along this path. The first and the third case yield the natural sum of the numbers, because the semiring instance of *Number* is defined this way. In the second and fourth example, the matrix values are mapped to the tropical semiring and the addition in this semiring is the minimum.

Using *shortestWith* $(\odot_{\sim,\sum})$, where *shortestWith* is defined in Section 4.4, we can find both an augmenting path and the minimum along this path with a single path search. We implement this search as follows, where the auxiliary function $maybeSome :: Vec\ \alpha \rightarrow Maybe\ (Arc\ \alpha)$ is defined in Section 5.2 and the functions *toVecWith* and *toVec* are defined in Section 4.3.

```
fAugmentingPath :: (Num ν, Ord ν) ⇒
  Mat (Number ν) → Vertex → Vertex → Maybe (Path, Number ν)
fAugmentingPath cf s t = fmap finish (maybeSome result) where
  finish (i, (p, m)) = (p ▷ i, weight m))
  result             = shortestWith (⊙~,∑) start end [fmap Weight cf]
  start              = toVecWith (⟨⟩, zero) [s]
  end                = toVec [t]
```

This function is very similar to the function *augmentingPath* from Section 5.2. In fact, the main differences are that we have different start and target sets and a slightly different result modification. With this function we can define the flow augmentation function as follows, where we use Equation (7.2.v) and Equation (7.2.vi) to compute the improved flow and the new residual capacity.

```
augmentFlow :: (Ord ν, Num ν) ⇒ Network ν → Mat (Number ν) → Mat (Number ν)
   → Maybe (Mat (Number ν), Mat (Number ν))
augmentFlow n f cf = fmap improve (fAugmentingPath cf (source n) (sink n)) where
  improve (p, ε) = (f ⊞ u_p, (c_f ⊟ u_p) ⊞ transposeSquare u_p) where
    u_p = c ⊓_r (σ_p ⊟ transposeSquare σ_p)
    σ_p = pathToGraphWith (λ_ _ → ε) p c
    c   = capacity n

(⊟) :: (Eq ν, Num ν) ⇒ Mat (Number ν) → Mat (Number ν) → Mat (Number ν)
a ⊟ b = a ⊞ fmap ((−1)⊛) b

(⊓_r) :: Mat α → Mat β → Mat β
(⊓_r) = intersectMatWithKey (λ_ _ x → x)
```

The *improve* function in the function *augmentFlow* takes a path and a value (the minimum along this path) and computes the augmented flow and the modified residual capacity. The functions (⊞) for the addition of matrices and *intersectMatWithKey* have been defined in Section 5.5.2 already. The implementation of (⊟) is based upon the group-theoretic definition of the subtraction, which is defined as the addition of the first argument and the inverse of the second one. The function *pathToGraphWith* is a directed variant of a similar function that we used in the case of matchings and can be defined as follows.

$$
\begin{aligned}
&\mathit{pathToGraphWith} :: (\mathit{Vertex} \to \mathit{Vertex} \to \alpha) \to \mathit{Path} \to \mathit{Mat}\ \beta \to \mathit{Mat}\ \alpha \\
&\mathit{pathToGraphWith}\ f\ p\ g = \mathit{Mat}\ (\mathit{mkVec}\ \mathit{pre} \cup_{\mathsf{l}} \mathit{correct})\ \mathbf{where} \\
&\quad \mathit{pre} \quad\ \ = \mathit{zipWith}\ (\lambda i\ j \to (i, \mathit{mkVec}\ [(j, f\ i\ j)]))\ \mathit{ps}\ (\mathit{tail}\ \mathit{ps}) \\
&\quad \mathit{ps} \quad\ \ \ = \mathit{toList}\ p \\
&\quad \mathit{correct} = \mathit{fmap}\ (\mathit{const}\ \mathit{emptyVec})\ (\mathit{matrix}\ g)
\end{aligned}
$$

It takes a labelling function, a path and a graph and creates a matrix, which consists of only those edges that are located on the path, such that every edge is labelled with the label that is obtained by applying the supplied function to both endpoints of the edge. The additional matrix argument provides the size of the result graph.

We now have everything we need to implement the maximum flow function. The resulting implementation is very similar to the one for matchings in Section 5.3. Using the previously defined functions we implement the function that computes a maximum flow in a network as follows.

$$
\begin{aligned}
&\mathit{maximumFlow} :: (\mathit{Num}\ \nu, \mathit{Ord}\ \nu) \Rightarrow \mathit{Network}\ \nu \to \mathit{Mat}\ (\mathit{Number}\ \nu) \\
&\mathit{maximumFlow} = \mathit{fst} \circ \mathit{maximumFlow}' \\
&\mathit{maximumFlow}' :: (\mathit{Num}\ \nu, \mathit{Ord}\ \nu) \Rightarrow \\
&\quad \mathit{Network}\ \nu \to (\mathit{Mat}\ (\mathit{Number}\ \nu), \mathit{Mat}\ (\mathit{Number}\ \nu)) \\
&\mathit{maximumFlow}'\ \mathit{net} = \mathit{go}\ (\mathit{emptyMat}\ c)\ c\ \mathbf{where} \\
&\quad \mathit{go}\ f\ \mathit{cf} = \mathit{maybe}\ (f, \mathit{cf})\ (\mathit{uncurry}\ \mathit{go})\ (\mathit{augmentFlow}\ \mathit{net}\ f\ \mathit{cf}) \\
&\quad c \qquad\ = \mathit{capacity}\ \mathit{net}
\end{aligned}
$$

We begin with the constant zero flow and the capacity function. In each iteration we increase the flow value and compute the new residual capacity according to Theorem 7.1.4. An example call on the network from Figure 7.1(a) yields the following result.

```
ghci> maximumFlow network
0 : (1 | Number 7.0) (2 | Number 5.0) (3 | Number 5.0)
1 : (4 | Number 3.0) (5 | Number 4.0)
2 : (5 | Number 5.0)
3 : (6 | Number 5.0)
4 : (7 | Number 7.0)
5 : (4 | Number 4.0) (6 | Number 5.0)
6 : (7 | Number 10.0)
7 :
```

This flow is exactly the maximum flow from Figure 7.1(c).

Despite the more technical structure of the algorithm, as well as not only purely relational arguments, the resulting function *maximumFlow* is rather simple and composed from small, independent components. Additionally, it is polynomial in the graph size. It is known [CLRS01] that the number of flow improvement steps in the Edmonds-Karp algorithm is bounded by $\mathcal{O}(|V| \cdot |E|)$. In each improvement step a path search is performed, which in our case takes at most $\mathcal{O}\left(|V|^2\right)$ operations. The

matrix operations in each improvement step have a complexity of $\mathcal{O}(|V|^2)$ and thus the overall complexity is $\mathcal{O}(|V|^3 \cdot |E|)$. Using different intermediate structures as discussed in Section 4.6 we can improve this complexity to $\mathcal{O}(|V| \cdot |E|^2 \cdot \log_2(|V|))$, which is different from the imperative complexity only by the logarithmic factor.

7.3 Minimum Cuts in Networks

The dual problem for maximum flows are minimum cuts. As we have seen in Chapter 6 it is possible to use the solution of the matching problem to solve the vertex cover problem. The same is true in case of the cut problem: it can be solved using a maximum flow. Since this problem is much simpler to solve than the vertex cover problem, we combine the theoretical basis, the examples and the implementation in this section. We begin with the definition of a cut.

7.3.1 Definition (Cut).
Let $N = (V, E, s, t, c)$ be a network. Let $\mathcal{S} := \{ S \subseteq V \mid s \in S \wedge t \in V \setminus S \}$. Every $S \in \mathcal{S}$ is called a *cut*. We define

$$\Xi : 2^V \to 2^E , \quad S \mapsto \{ (v, w) \in E \mid v \in S \wedge w \in V \setminus S \} = E \cap S^\top \cdot \overline{S}$$

and

$$\|-\|_c : 2^E \to \mathbb{R}_{\geqslant 0} , \quad S \mapsto \sum_{e \in S} c(e) .$$

For every $S \in \mathcal{S}$ the value $\|\Xi(S)\|_c$ is called the *capacity of the cut* S. We set

$$\mathsf{cut}(N) := \min \{ \|\Xi(T)\|_c \mid T \in \mathcal{S} \}$$

and call a cut S a *minimum cut* if and only if $\|\Xi(S)\|_c = \mathsf{cut}(N)$ holds. //

Since there is only a finite number of cuts, the value $\mathsf{cut}(N)$ is well-defined and there exists a cut with minimal capacity. Our definition differs slightly from the one in the literature [Die00], because cuts are usually defined as pairs (C, D) such that $\{ C, D \}$ is a partition of V. Clearly, this is equivalent to $D = \overline{C}$ and since $\overline{C}$ can be obtained from V and C, we omit D altogether. Note that among the $2^{|V|}$ subsets of V there are $2^{|V|-2}$ cuts. For example, in our example network from Figure 7.1(a) we have, among others, the cuts $\{ 0 \}$, $\{ 0, 1, 2, 3, 4 \}$ and $\{ 0, 4, 5, 6 \}$. The actual difficulty is to find the minimum cut amongst the rather large set of all cuts. In the above example, $\{ 0, 1, 2, 3, 5, 6 \}$ is a minimum cut.

The Ford-Fulkerson theorem as presented by Diestel [Die00] states that in a network the value of a maximum flow is equal to the capacity of a minimum cut, which is why said theorem is also known as the "max-flow min-cut theorem". The proof is constructive: let $N = (V, E, s, t, c)$ be a network and f a maximum flow in N. Then

$$S := \{ s \} \cdot (E_f)^\star$$

is a minimum cut. The proof consists of two parts: showing that S is a cut and that it has minimum capacity. The first part is simple: since f is a maximum flow, there is

no path from s to t in (V, E_f) by Theorem 7.1.4 and thus $t \notin \{s\} \cdot (E_f)^\star$, while $s \in S$ is trivially true, because $\mathsf{I} \sqsubseteq (E_f)^\star$. The second part is more technical and we refer to Ford Jr. and Fulkerson [FF62] for more details.

It is interesting to note that while the idea for the set S above is remarkably clever, computing this set is now quite simple with our means. In fact, we only need to compute a maximum flow, its residual network and in that network find the set of all vertices that are reachable from the source. Fortunately, we already have functions that solve each of these problems and thus obtain a solution as follows.

```
minimumCut :: (Ord ν, Num ν) ⇒ Network ν → Vec ()
minimumCut net = reachableWith (⊗) s [residual] where
  s        = toVec [source net]
  residual = snd (maximumFlow′ net)

(⊗) :: Vec α → Mat β → Vec α
(⊗) = vecMatMult leftmostUnion (sMultWith (λ_ x _ → x))
```

The auxiliary vector-matrix multiplication (⊗) is very similar to (⊗) from Section 4.3.2, but ignores the values in the matrix instead of the values in the vector. Applying this function to the example network from Figure 7.1(a) yields the following result.

```
ghci> minimumCut network
(0 | ()) (1 | ()) (2 | ()) (3 | ()) (5 | ()) (6 | ())
```

The above result is exactly the cut that we have seen after Definition 7.3.1.

7.4 Related Work and Discussion

In our approach we have restricted ourselves to networks, thus disallowing an edge from w to v if there is already an edge from v to w. This is a proper restriction, because there are graphs that do not satisfy this property. Still, it is possible to transform the general case to the special one of networks. To achieve that, for all vertices $v, w \in V$ such that $(v, w), (w, v) \in E$ one can add a new vertex called $x_{v,w}$, remove the edge (v, w), add two new edges $(v, x_{v,w})$ and $(x_{v,w}, w)$ and set the capacity of these new edges to $c(v, w)$.

As an alternative to this approach, one can modify the definition of a flow, by removing the non-negativity condition, but adding the condition that in case of $v, w \in V$ such that $(v, w), (w, v) \in E$ we have

$$f(v, w) = -f(w, v) .$$

In this approach the computation of the updated flow as well as the updated residual capacity is not as simple as we have seen in Section 7.1, because several of the computational steps in our derivation depend on the fact that $E \sqcap E^\top = \mathsf{O}$ holds.

As we have mentioned at the beginning of this chapter, the maximum flow problem has been solved in Haskell by Erwig and Miljenovic [EM14] in the library *fgl*.

However, there are some differences to our approach. Most importantly, Erwig and Miljenovic allow general graphs and take the second approach to flows described above. While our implementation is based on an algebraic approach, the *fgl* implementation is less algebraic, but closer to the usual, graph-theoretic solution. Additionally, our actual maximum flow function is a variation of the auxiliary function *maximumFlow′*, which at the same time computes a maximum flow and its residual capacity. This modular approach makes an implementation of the minimum cut function particularly simple, because it requires the residual capacity of a maximum flow and thus can be implemented without any recomputations.

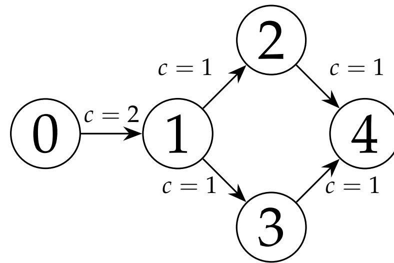


Figure 7.4. A network with non-disjoint augmenting paths.

Finally, we can replace the computation of a single flow augmenting path by a maximal set of shortest paths, which are all pairwise disjoint except for their start and end vertices. Such a set can be obtained in a similar fashion as we have discussed before for the function *disjointPaths*. The resulting function then bears some resemblance with the maximum flow algorithm due to Dinitz [Din06]. However, the idea of the Dinitz algorithm is to improve a flow with augmenting paths of the shortest length such that there is no more augmenting path of this length left in the residual graph. This condition is *not* guaranteed by simply taking a set as described above. Consider for example the graph from Figure 7.4 with $s = 0$ and $t = 4$. Then $(0, 1, 2, 4)$ is a shortest augmenting path (of the zero flow). But so is $(0, 1, 3, 4)$ and both paths have the same length and can be used after another to improve the zero flow, although they are not disjoint up to s and t. Still, the number of iterations in which one searches for augmenting paths is less or equal than in the original algorithm, which is likely to improve the complexity by a constant factor.

Chapter 8

Algebraic Problems

In this chapter we consider some exemplary problems, all of which can be modelled algebraically. The first part, Section 8.1, of this chapter deals with the *control in approval voting*, where one wishes to change the outcome of a special type of election. In the second part, Section 8.2, we consider the problem whether a graph is *balanced* or *clustered*. This type of graph property is of interest in the analysis of social networks [Bat94]. The third part, Section 8.3 then considers the *centrality problem*, which describes a certain type of measure of graph vertices, which again originates in social network theory [Bat94].

To the best of our knowledge, the results of Section 8.1 have not been presented in Haskell before. The main concept of our implementation has been presented before at the RAMiCS 2014 conference [BDS14], but without providing the actual code. As for the contents of Section 8.2 and Section 8.3 we simply present an implementation of the results of Batagelj [Bat94] with our means. To improve the legibility, we repeat several results from the above source. While we mention this on occasion, we wish to point out that our contribution from Section 8.2 and Section 8.3 is merely the implementation – all theory has been developed elsewhere. Also, there is a Haskell library *igraph* [GS13] that provides some of the functionality we describe in Section 8.2 and Section 8.3. However, the corresponding functions are obtained from bindings to a C library and not implemented in Haskell directly. The *fgl* library [EM14] does not provide functions for these purposes.

8.1 Approval Voting

Approval voting is a voting method, which is often chosen as an election for real-world problems like finding a common time for a meeting or deciding where to go on the next field trip. An *approval election* is a triple (V, A, c), where V is a finite set of *voters*, A is a finite set of *alternatives*, and $c : V \times A \to \{ 0, 1 \}$ is a function, such that for each voter $v \in V$ the alternatives the voter approves of are exactly the elements of the set $\{ a \in A \mid c(v, a) = 1 \}$. Note that in our notation c can be considered to be a matrix. For simplicity, we assume that there are $n, m \in \mathbb{N}$ such that $V = \mathbb{N}_{<n}$ and $A = \mathbb{N}_{<m}$. For every $S \subseteq V$ we define

$$\mathsf{votes}_S : A \to \mathbb{N}\,, \quad a \mapsto \left(\left(\sum_{v \in S} \mathsf{e}^{|V|}(v) \right) \cdot c \right)_a \,,$$

where $\mathsf{e}^{|V|}(v)$ denotes the v-th standard unit vector, cf. Definition 2.4.4. Before we continue, we list some simple properties of votes.

8.1.1 Proposition (Properties of votes)**.**
Let (V, A, c) be an approval election.

(1) For all $S \subseteq V$ and all $a \in A$ we have

$$\mathsf{votes}_S(a) = \sum_{v \in S} c_{v,a} \ .$$

(2) For all $S, T \subseteq B$ such that $S \cap T = \emptyset$ and all $a \in A$ the following holds

$$\mathsf{votes}_{S \cup T}(a) = \mathsf{votes}_S(a) + \mathsf{votes}_T(a) \ .$$

(3) For all $S, T \subseteq V$ such that $S \subseteq T$ holds and all $a \in A$ we get

$$\mathsf{votes}_S(a) \leqslant \mathsf{votes}_T(a) \ .$$

(4) For all $S \subseteq V$ and all $a \in A$ we have $\mathsf{votes}_S(a) = |\{ p \in S \mid c_{p,a} = 1 \}|$.

(5) For all $S \subseteq V$ and all $a \in A$ we have the inequality $\mathsf{votes}_S(a) \leqslant |S|$.

Proof. Statement (1). Let $S \subseteq V$ and $a \in A$. Then we get

$$\begin{aligned} & \mathsf{votes}_S(a) \\ = \ & \wr \text{ definition of votes } \wr \\ & \left(\left(\sum_{v \in S} \mathsf{e}^{|V|}(v) \right) \cdot c \right)_a \\ = \ & \wr \text{ definition of vector-matrix multiplication } \wr \\ & \left(\sum_{v \in S} c_v \right)_a \\ = \ & \wr \text{ additivity of projections and } (c_v)_a = c_{v,a} \wr \\ & \sum_{v \in S} c_{v,a} \ . \end{aligned}$$

Statement (2). Let $S, T \subseteq V$ such that $S \cap T = \emptyset$ and $a \in A$. Then the following holds:

$$\begin{aligned} & \mathsf{votes}_{S \cup T}(a) \\ = \ & \wr \text{ by Proposition 8.1.1.(1) } \wr \\ & \sum_{v \in S \cup T} c_{v,a} \\ = \ & \wr \text{ partitioning the sum } \wr \\ & \left(\sum_{v \in S} c_{v,a} \right) + \left(\sum_{v \in T} c_{v,a} \right) \\ = \ & \wr \text{ by Proposition 8.1.1.(1) } \wr \end{aligned}$$

$$\mathsf{votes}_S(a) + \mathsf{votes}_T(a) \ .$$

Statement (3). Let $S, T \subseteq V$ such that $S \subseteq T$ holds and $a \in A$. Then we get

$$\begin{aligned}
& \mathsf{votes}_S(a) \\
= \quad & \wr \text{ by Proposition 8.1.1.(1) } \wr \\
& \sum_{v \in S} c_{v,a} \\
\leqslant \quad & \wr\ c_{v,a} \geqslant 0 \text{ for all } v \in T \setminus S\ \wr \\
& \left(\sum_{v \in S} c_{v,a}\right) + \left(\sum_{v \in T \setminus S} c_{v,a}\right) \\
= \quad & \wr \text{ by Proposition 8.1.1.(1) } \wr \\
& \mathsf{votes}_S(a) + \mathsf{votes}_{T \setminus S}(a) \\
= \quad & \wr \text{ by Proposition 8.1.1.(2), since } S \cap (T \setminus S) = \varnothing\ \wr \\
& \mathsf{votes}_{S \cup (T \setminus S)}(a) \\
= \quad & \wr \text{ since } S \subseteq T \text{ we have } T = (T \cap S) \cup (T \setminus S) = S \cup (T \setminus S)\ \wr \\
& \mathsf{votes}_T(a) \ .
\end{aligned}$$

Statement (4). Let $S \subseteq V$. Then we find

$$\begin{aligned}
& \mathsf{votes}_S(a) \\
= \quad & \wr \text{ by Proposition 8.1.1.(1) } \wr \\
& \sum_{v \in S} c_{v,a} \\
= \quad & \wr\ c_{v,a} \in \{ 0, 1 \}\ \wr \\
& \sum_{v \in S,\ c_{v,a} = 1} c_{v,a} \\
= \quad & \wr \text{ inlining } c_{v,a}\ \wr \\
& \sum_{v \in S,\ c_{v,a} = 1} 1 \\
= \quad & \wr \text{ counting elements of a set } \wr \\
& |\{ v \in S \mid c_{v,a} = 1 \}| \ .
\end{aligned}$$

Statement (5). Let $S \subseteq V$. Then we get the following inequality:

$$\begin{aligned}
& \mathsf{votes}_S(a) \\
= \quad & \wr \text{ by Proposition 8.1.1.(4). } \wr \\
& |\{ v \in S \mid c_{v,a} = 1 \}| \\
\leqslant \quad & \wr\ \{ v \in S \mid c_{v,a} = 1 \} \subseteq S \text{ and cardinality is monotonic } \wr \\
& |S| \ .
\end{aligned}$$

//

By Proposition 8.1.1.(4) the value $\mathsf{votes}_S(a)$ denotes the number of voters in S that

are in favour of the alternative a. Thus the definition coincides with the intuition. However, the definition in terms of a vector-matrix multiplication is particularly convenient for working with this value.

The natural question in this context is, which alternative is the most preferred one. To that end one can simply compute for each alternative the number of voters that approve of this alternative and define an alternative to be winning if and only if it has the maximal number of votes. Clearly, there is not necessarily a unique winner, but a winner always exists.

The relational model of approval voting has been discussed before [BDS14]. We use this model for an implementation as follows.

```
type Voter = Int
type Alternative = Int
type AppVoting = Mat ()
votesWithVoters :: AppVoting → Vec α → Vec (Number Int)
votesWithVoters = flip (⊙1,+)
(⊙1,+) :: Num ν ⇒ Vec α → Mat β → Vec ν
(⊙1,+) = vecMatMult (bigUnionWith (+)) (sMultWith (λ_ _ _ → 1))
results :: AppVoting → Vec (Number Int)
results v = votesWithVoters v (voters v)
winners :: AppVoting → Vec α → Vec (Number Int)
winners v ps = fst (restrictToMax (votesWithVoters v ps))
voters :: AppVoting → Vec ()
voters = verticesWith ()
restrictToMax :: (Num ν, Ord ν) ⇒ Vec (Number ν) → (Vec (Number ν), Number ν)
restrictToMax v = (filterVec (≡ m) v, m) where
  m = foldr max 0 (values v)
maxIndices :: (Num ν, Ord ν) ⇒ Vec (Number ν) → [Int]
maxIndices = fromVec ∘ fst ∘ restrictToMax
maxVec :: (Num ν, Ord ν) ⇒ Vec (Number ν) → Number ν
maxVec = snd ∘ restrictToMax
values :: Vec α → [α]
values = map snd ∘ unVec
```

The auxiliary function *restrictToMax* computes a pair, where the first component is the list of indices that have a maximal value in the vector and the second component is the maximal value in the vector. The idea behind this implementation is rather simple. We use a non-square matrix to represent the election, where the rows are indexed with the voters and the columns are indexed with the alternatives, such that every row denotes the choices a certain voter approves of. Given such an election and a vector of voters, we can compute the number of approving voters for each

alternative by counting how often this alternative occurs among the rows associated with the participating voters. To achieve this, we use a very simple vector-matrix multiplication that completely ignores both the values in the matrix and the values in the vector and merely counts the number of times, a certain "successor" is reached. Consider for example the following approval voting matrix

$$c := \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

and suppose that it is represented in Haskell by the constant *votes* :: *AppVoting*. Then we get the following example calls.

```
ghci> votesWithVoters votes (verticesWith () votes)
(0 | 3) (1 | 3) (2 | 3) (3 | 1) (4 | 3) (5 | 3) (6 | 4) (7 | 3) (8 | 2) (9 | 3)
ghci> winners votes (verticesWith () votes)
(6 | 4)
ghci> votesWithVoters votes (toVec [0,1,4,5])
(0 | 2) (1 | 3) (2 | 2) (4 | 3) (5 | 2) (6 | 3) (7 | 2) (8 | 1) (9 | 1)
ghci> winners votes (toVec [0,1,4,5])
(1 | 3) (4 | 3) (6 | 3)
```

Note that in the first example, where every voter participates in the vote, we have exactly one winning alternative, namely alternative 6 with four votes. If the set of voters is reduced to $\{0,1,4,5\}$, the we have three winning alternatives, namely the alternatives $1,4$ and 6 with three votes each.

We now consider the control of approval voting. In this setting one assumes that an election chair knows the votes of all voters. This chair then tries to control the election by removing as few voters as possible to make a certain alternative a winning one[1]. This is known as the *constructive control problem*. More formally, given an $a \in A$ we are looking for an $S \subseteq V$ with minimal cardinality such that a is a winning alternative in $(V \setminus S, A, c|_{(V \setminus S)\times A})$. This control problem is known to be NP-complete [HHR07]. In the context of social choice such an election is called *resistant to control* — it is not immune to control, but finding a possible control set is computationally difficult.

While NP-complete, we can still try to solve the control problem as fast as possible. To that end we use a simple generate-and-test approach. Before we describe this approach, we first observe how many voters need to be removed for a (non-winning)

[1]There are different types of control aside from the one described above. For example, the *destructive control problem* asks for a set of voters with maximal cardinality, such that a given alternative does *not* win.

alternative to win. Let us assume that $a \in A$ is an alternative and that $S \subseteq V$ is a subset of voters such that a is a winning alternative in $(V \setminus S, A, c|_{(V \setminus S) \times A})$. Let $w \in A$ be a winning alternative in (V, A, c). We claim that the following holds:

$$|S| \geqslant \mathsf{votes}_V(w) - \mathsf{votes}_V(a)\ . \tag{8.1.1.i}$$

This inequality is obtained from the following estimate:

$$
\begin{aligned}
& \mathsf{votes}_V(w) \\
= \quad & \wr \text{ by Proposition 8.1.1.(2) } \wr \\
& \mathsf{votes}_{V \setminus S}(w) + \mathsf{votes}_S(w) \\
\leqslant \quad & \wr\ a \text{ is winning in } (V \setminus S, A, c|_{(V \setminus S) \times A})\ \wr \\
& \mathsf{votes}_{V \setminus S}(a) + \mathsf{votes}_S(w) \\
\leqslant \quad & \wr \text{ by Proposition 8.1.1.(3) since } V \setminus S \subseteq V\ \wr \\
& \mathsf{votes}_V(a) + \mathsf{votes}_S(w) \\
\leqslant \quad & \wr \text{ by Proposition 8.1.1.(5) } \wr \\
& \mathsf{votes}_V(a) + |S|\ .
\end{aligned}
$$

Thus to make a win the election, we need to remove at least as many voters as the difference between the number of votes of any winner in the election and the number of votes for a in the election.

Now let us assume that we have a function

```
powerlistFrom :: Int → Int → [[Int]]
```

at hand, such that for non-negative k, n :: *Integer* the value *powerlistFrom n k* is the representation of

$$\{\, S \subseteq \mathbb{N}_{<n} \mid |S| \geqslant k \,\}$$

as a list of lists in increasing order of their length[2]. For example we get the following results.

```
ghci> powerlistFrom 3 0
[[], [0], [1], [2], [0,1], [0,2], [1,2], [0,1,2]]
ghci> powerlistFrom 5 4
[[0,1,2,3], [0,1,2,4], [0,1,3,4], [0,2,3,4], [1,2,3,4], [0,1,2,3,4]]
ghci> powerlistFrom 4 2
[[0,1], [0,2], [1,2], [0,3], [1,3], [2,3], [0,1,2], [0,1,3], [0,2,3], [1,2,3], [0,1,2,3]]
```

We can use this function as the generator of sets we need to test. The actual test for a set S is then, whether a is among the winning alternatives. To test that, we compute *votesWithVoters v (voters v)* and subtract the vector[3] *votesWithVoters v (toVec s)*, where

[2] The Haskell module *Data.List* contains the function *subsequences* :: $[\alpha] \to [[\alpha]]$ that computes all subsequences (the "powerlist") of a list. However, the elements of the result list are not increasingly sorted with respect to their length.

[3] We could compute *votesWithVoters v (voters v \ toVec s)*, but set difference is likely slower than a numerical subtraction.

s is the list representation of S. In this result vector we then compute those indices that have a maximal value. The desired alternative wins, if one of the following two conditions holds: the alternative is contained in the above index list or the index list is empty. The latter part is important, because this means that we have removed so many voters already that there are no more votes for anyone, thus any remaining alternative is automatically a winner. If either possibility applies, we have found a set that turns the desired alternative into a winning one and we stop the search. Otherwise we continue with the next set. Note that for every alternative there exists a set of voters we can remove, such that the alternative wins with the remaining voters, because we can simply remove all voters. Our implementation then looks as follows, where the *Integer* in the result denotes the number of sets that have been searched by the exhaustive search.

```
generateAndTestApproval :: AppVoting → Alternative → (Integer, [Voter])
generateAndTestApproval v alt =
  head (dropWhile (losing ∘ snd) (zip [1 ..] (findSets v alt))) where
  res     = results v
  losing s = not (emptyOrElem alt (maxIndices (res −ᵥ votesWithVoters v (toVec s))))

emptyOrElem :: Eq α ⇒ α → [α] → Bool
emptyOrElem x xs = null xs ∨ x ∈ xs

(−ᵥ) :: (Eq ν, Num ν) ⇒ Vec (Number ν) → Vec (Number ν) → Vec (Number ν)
a −ᵥ b = removeZeroes (a +ᵥ fmap negate b)
```

The first component of the result of *generateAndTestApproval* is the number of sets that have been analysed before the set in the second component has been found. Now all we need is the function *findSets* which returns those sets we need to search to find a possible match. We implement this function as an application of the *powerlistFrom* function.

```
findSets :: AppVoting → Alternative → [[Voter]]
findSets v alt = powerlistFrom (size (voters v)) (atLeast v alt)

atLeast :: AppVoting → Alternative → Int
atLeast v alt = number (maxVec res − (res ! alt)) where
  res = results v

size :: Vec α → Int
size = length ∘ unVec
```

The function *findSets* simply calls *powerlistFrom* with the number of voters and the minimal number of voters we need to remove. The latter number can be computed as the difference between the maximal value in the overall result and the number of votes for the desired alternative in the overall result, as we have discussed above in Equation (8.1.1.i).

Recall that in our example from before, we had the winning alternative 6 with 4

votes. We can now try to change the outcome of the election.

```
ghci> generateAndTestApproval votes 2
(1, [0])
ghci> generateAndTestApproval votes 6
(0, [ ])
ghci> generateAndTestApproval votes 3
(22, [0, 1, 2, 4])
```

Non-existent alternatives behave as alternatives with no votes.

```
ghci> generateAndTestApproval votes 1000
(22, [0, 1, 2, 3, 4, 5])
```

The missing function *powerlistFrom* can be implemented using a special technique that is often employed in the context of memoisation in Haskell. The key idea is to define a constant that contains all necessary values. This constant then behaves as a computed table, because constants are shared throughout the program. The concrete implementation is based upon the implementation of binomial coefficients with this very technique by Fischer [Fis].

```
powerlistFrom :: Int → Int → [[Int]]
powerlistFrom n k = map toList (concatMap (subsetsOfSize n) [k .. n])

subsetsOfSize :: Int → Int → [Seq Int]
subsetsOfSize n k = allSubsets !! n !! k

allSubsets :: [[[Seq Int]]]
allSubsets = [[subsets n k | k ← [0 ..]] | n ← [0 ..]]

subsets :: Int → Int → [Seq Int]
subsets n k
  | n < 0 ∨ n < k ∨ k < 0 = [ ]
  | k ≡ 0                 = [⟨⟩]
  | otherwise             =                     subsetsOfSize (n − 1) k
                            ++ map (⊳(n − 1)) (subsetsOfSize (n − 1) (k − 1))
```

In this implementation we restrict ourselves to the *Int* data type only for the sake of simplicity. Haskell provides the functions *genericIndex* and *genericLength*, which are generalisations of (!!) and *length* and are applicable for more general types than just *Int*. One particularly interesting alternative is *Integer*, because it is unbounded. Since we explicitly use *Int* for the vertices (indices of a matrix), we need to transform these indices to *Integer*s to be able to perform arithmetic operations involving indices and *Integer*s, because all arithmetic operations in Haskell are homogeneous and there is no implicit overloading. Obtaining a more general implementation using the above approach is a mere technicality and we omit it only for the sake of presentation.

Despite the fact that the function *generateAndTestApproval* has a worst-case exponential complexity in the number of voters, the time consumptions for elections with

a small number of voters are still moderate. Using a compiled version of the above function, we found that the generate-and-test approach can test about 50000 sets per second. In case of elections with less than 20 voters this means that a minimal number of voters that need to be removed to make any particular alternative win the election can be computed in at most 21 seconds. With 22 voters this number increases to 90 seconds, which is due to the exponential growth of the power set. In summary, the generate-and-test approach is suited for elections with a small number of voters, while computing the winning alternatives with a given vector of voters is applicable for every election, because this computation is linear in the size of the election matrix.

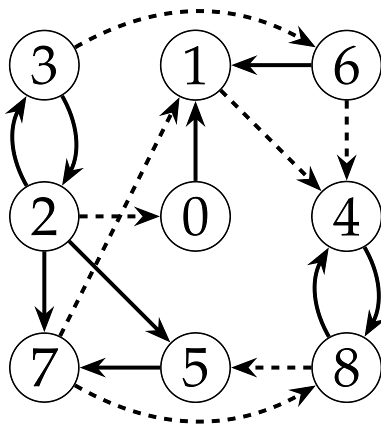
8.2 Balanced Graphs and Clustered Graphs

In this section we deal with two related graph problems, namely testing whether a graph is balanced and whether it is clustered. Let $G = (V, E, f)$ be a $\{-1, 1\}$-labelled graph. The graph G is called *clustered* if and only if there exists a partition $\mathcal{P} \subseteq 2^V$ of V such that the following conditions hold[4]:

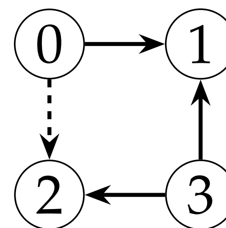
(1) $f^{-1}(\{1\}) \subseteq \bigcup \{ P \times P \mid P \in \mathcal{P} \}$,

(2) $f^{-1}(\{-1\}) \subseteq \bigcup \{ P \times Q \mid P \in \mathcal{P} \wedge Q \in \mathcal{P} \wedge P \neq Q \}$.

A clustered graph, whose partition contains exactly two elements is called *balanced*. Intuitively, a clustered graph is a graph that can be partitioned, such that all edges with the label 1 connect vertices from the same set in the partition, while all edges with the label -1 connect vertices from two different sets in the partition. In Figure 8.1



(a) A clustered graph.



(b) A non-clustered graph.

Figure 8.1. Clustered and non-clustered graphs

[4]This is essentially the point-free version of the definition given by Batagelj [Bat94].

we provide two example graphs, where the bold edges denote edges with label 1, while the dashed edges are edges with label -1. The graph in Figure 8.1(a) is a clustered graph, where the partition is $\{\{0,1,6\},\{2,3,5,7\},\{4,8\}\}$. On the other hand, the graph in Figure 8.1(b) is not clustered, because the edges $(0,1),(3,1),(3,2)$ yield that all vertices must belong to the same partition set, while the edge $(0,2)$ yields that the vertices $0,2$ need to be in two different partition sets.

It turns out that checking whether a graph is clustered (or even balanced) is simple using a special notion of paths. Let

$$G_{\text{sym}} := \left(V, E \cup E^{\top}\right)$$

and

$$f' : E \cup \left(E^{\top} \setminus E\right) \to \{-1,1\} \ , \quad (v,w) \mapsto \begin{cases} f(v,w) & : (v,w) \in E \\ f(w,v) & : \text{otherwise} \ . \end{cases}$$

We call a walk w in G_{sym}

$$\text{positive} \ :\Longleftrightarrow \prod_{i=0}^{|w|-2} f'(w_i, w_{i+1}) = 1 \ ,$$
$$\text{negative} \ :\Longleftrightarrow \prod_{i=0}^{|w|-2} f'(w_i, w_{i+1}) = -1 \ ,$$

where use the common convention that empty products are 1, so that walks of length zero and walks of length one are automatically positive. Clearly, being positive is equivalent to the fact that the number of edges with label -1 is even.

8.2.1 Theorem (Characterisation of clustered graphs)**.**
Then the following hold:

(1) G is clustered if and only if there is no cycle in G_{sym} that contains exactly one edge with label -1.

(2) G is balanced if and only if every cycle in G_{sym} is positive. //

This theorem is stated in this way by Batagelj [Bat94] and a variant of the proof of the first statement for symmetric graphs is provided by Davis [Dav67]. However, a careful observation of the arguments in the proof make them applicable for directed graphs as well. The proof of the second statement (with the same restrictions) is given by Riley [Ril69].

The statement of Theorem 8.2.1.(1) can be rephrased relationally. Let

$$E_1 := f^{-1}(\{1\}) \ ,$$
$$E_{-1} := f^{-1}(\{-1\}) \ .$$

Then by Theorem 8.2.1.(1) the graph G is clustered if and only if there is no cycle in G_{sym} with exactly one edge from E_{-1}. The latter condition can be expressed

algebraically as[5]

$$\left(E_1 \cup E_1^{\top}\right)^{\star} \cap E_{-1} = \mathsf{O} \; .$$

The equivalence of the above expressions is simple to verify using logical transformations and is already mentioned by Batagelj [Bat94].

We can use this formula to implement a test for being clustered. First, we define a data type for the edge labels.

```
data Sign = Pos | Neg
  deriving (Show, Eq)
```

Next, we can implement the test using previously defined functions as follows.

```
isClustered :: Mat Sign → Bool
isClustered m = isEmptyMat (starSymClosure ePos ⊓| eNeg) where
  ePos = toBool (positive m)
  eNeg = toBool (negative m)

filterMatrix :: (α → Bool) → Mat α → Mat α
filterMatrix p = Mat ∘ fmap (filterVec p) ∘ matrix

positive :: Mat Sign → Mat Sign
positive = filterMatrix (Pos ≡)

negative :: Mat Sign → Mat Sign
negative = filterMatrix (Neg ≡)

starSymClosure :: (Eq κ, KleeneAlgebra κ) ⇒ Mat κ → Mat κ
starSymClosure = starClosure ∘ symClosure

symClosure :: Semiring σ ⇒ Mat σ → Mat σ
symClosure m = m ⊞ transposeSquare m
```

Suppose that the graphs from Figure 8.1(a) and Figure 8.1(b) are implemented in Haskell as the constants *clustered*, *nonClustered* :: *Mat Sign* respectively. Then we get the following example applications.

```
ghci> isClustered clustered
True
ghci> isClustered notClustered
False
```

Instead of the relational approach above, in which relational components like the reflexive-transitive closure are mixed with matrix operations like filtering, one can take another, more algebraic approach. To that end one can define a Kleene algebra and reduce the test for being clustered to another star closure operation. The Kleene algebra in question is the set $K_C := \{ \bot, n, p, a, q \}$, where the elements of this set are considered to be distinct letters and the index C is used only for disambiguation.

[5] It may seem that one also needs to consider $(E_{-1})^{\top}$, but since every path in $E_1 \cup E_1^{\top}$ can be reversed, this is not the case.

Batagelj [Bat94] then defines the operations on these five elements with the Cayley tables in Table 8.1. The intuition behind these values and the operations on these

Table 8.1. Operations in the clustered Kleene algebra.

$+$	$\bot$	n	p	a	q
$\bot$	$\bot$	n	p	a	q
n	n	n	a	a	n
p	p	a	p	a	p
a	a	a	a	a	a
q	q	n	p	a	q

$\cdot$	$\bot$	n	p	a	q
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
n	$\bot$	q	n	n	q
p	$\bot$	n	p	a	q
a	$\bot$	n	a	a	q
q	$\bot$	q	q	q	q

$\star$	$\bot$	n	p	a	q
	p	a	p	a	p

values are different types of walks and their alternatives (addition) or concatenation (multiplication) [Bat94]. The most important value for the test is the letter p, which denotes the fact that there is at least one walk (between two vertices) with only 1-labelled edges and no walk with exactly one -1-labelled edge. The value p is neutral with respect to the multiplication. The structure $(K_C, +, \cdot, ^\star, \bot, p)$, where the operations are the ones from Table 8.1, is called the *clustered Kleene algebra*[6]. The clustered Kleene algebra is in fact a Kleene algebra[7]. We can now consider the adjacency matrix of a graph G as a matrix over K_C by identifying 1 with p and -1 with n. Thus instead of a $\{-1, 1\}$-labelled graph, we consider a $\{n, p\}$-labelled graph and obtain its adjacency matrix as follows

$$A_{G,K_C} : V \times V \to K_C \ , \quad e \mapsto \begin{cases} p & : e \in E \wedge f(e) = 1 \\ n & : e \in E \wedge f(e) = -1 \\ \bot & : \text{otherwise} \ , \end{cases}$$

where we use Definition 2.3.2 and Convention 2.3.9 for this representation. Using this special adjacency matrix, Batagelj [Bat94] shows that the following two statements are equivalent:

(1) G is clustered

(2) The matrix $\left(A_{G,K_C} + A_{G,K_C}^\top\right)^*$ contains only the p value along its main diagonal.

We can use this characterisation for an implementation as well. First, we define the necessary Kleene algebra.

```
data Clustered = Bot | N | P | A | Q
  deriving (Show, Eq)
```

[6]Batagelj [Bat94] uses semirings that are idempotent and *complete*, where a complete semiring is a semiring in which the sum function is extended to (countable) sequences and satisfies the associativity, commutativity and distributivity laws. In a complete semiring $(S, +, \cdot, 0, 1)$ a closure function is defined as $^\star : S \to S,\ s \mapsto \sum_{k \in \mathbb{N}} s^k$. It is easily verified that a complete, idempotent semiring with the closure function as above is a Kleene algebra.

[7]Since K_C is finite, one can implement the operation tables in Haskell and check the Kleene algebra laws by exhaustively testing all possible combinations of values.

We omit the implementation of the operation tables from Table 8.1 for simplicity, because one can copy most of the cases using pattern matching, while combining some cases. Next, we need an explicit conversion function from (abstract) signs into the clustered Kleene algebra.

$signToClustered :: Sign \to Clustered$
$signToClustered\ Pos\ = P$
$signToClustered\ Neg = N$

All we need to do now is to compute the star closure of the given matrix in the clustered Kleene algebra and to check the main diagonal of the result. Note that the identity matrix over the clustered Kleene algebra is a matrix that has the value $p = 1_{K_C}$ along its main diagonal and $\bot = 0_{K_C}$ everywhere else. Thus we can intersect the star closure of the symmetric closure with the identity matrix and check whether the resulting matrix is the identity matrix.

$isDiagonalOne :: KleeneAlgebra\ \kappa \Rightarrow (\alpha \to \kappa) \to Mat\ \alpha \to Bool$
$isDiagonalOne\ embed\ m = starSymClosure\ m' \sqcap_{|}\ i \equiv i$ **where**
  $m' = fmap\ embed\ m$
  $i\ \ = identityOf\ m$
$isClustered2 :: Mat\ Sign \to Bool$
$isClustered2 = isDiagonalOne\ signToClustered$

Note that the type of *identityOf m* is automatically inferred to be $Mat\ \kappa$ in the function *isDiagonalOne*.

The test for balanced graphs works in a similar fashion, but with another, simpler Kleene algebra. Let $K_B := \{ \bot_B, \pi, \nu, \alpha \}$, where the elements are again considered to be distinct letters. Again, the different values describe a special classification of walks between two vertices: the value $\bot_B$ denotes no walks, π stands for only positive walks, ν denotes only negative walks, and α stands for at least one positive and at least one negative walk. With this intuition, the Cayley tables in Table 8.2 are natural consequences of considering addition to be the alternative and multiplication to be the composition of walks. The structure $(K_B, +, \cdot, ^\star, \bot_B, \pi)$, where $+, \cdot$, and * are the

Table 8.2. Operations in the balanced Kleene algebra.

$+$	$\bot_B$	ν	π	α
$\bot_B$	$\bot_B$	ν	π	α
ν	ν	ν	α	α
π	π	α	π	α
α	α	α	α	α

$\cdot$	$\bot_B$	ν	π	α
$\bot_B$	$\bot_B$	$\bot_B$	$\bot_B$	$\bot_B$
ν	$\bot_B$	π	ν	α
π	$\bot_B$	ν	π	α
α	$\bot_B$	α	α	α

*	$\bot_B$	ν	π	α
	π	α	π	α

operations from Table 8.2, is called the *balanced Kleene algebra*. Note that the addition coincides with the addition in the clustered Kleene algebra, where the value q is omitted and we write π for p, ν for n and α for a. Again, one can verify that the balanced Kleene algebra is indeed a Kleene algebra.

Batagelj provides a characterisation of balanced graphs in terms of the balanced Kleene algebra, too. Consider the adjacency matrix of G as a matrix over K_B

$$A_{G,K_B} : V \times V \to K_B \ , \quad e \mapsto \begin{cases} \pi & : e \in E \wedge f(e) = 1 \\ \nu & : e \in E \wedge f(e) = -1 \\ \bot_B & : \text{otherwise} \ . \end{cases}$$

Then the following statements are equivalent [Bat94]:

(1) G is balanced.

(2) The matrix $\left(A_{G,K_B} + A_{G,K_B}^{\top}\right)^*$ contains only the value π along its main diagonal.

This result allows an implementation of a test for being balanced in essentially the same fashion as before. One simply needs to define the balanced Kleene algebra

```
data Balanced = Bot_B | PB | NB | AB
  deriving (Eq, Show)
```

and to provide an instance definition of *KleeneAlgebra* for this data type, which we omit again for simplicity. Then we can simply reuse our function *isDiagonalOne*, by supplying a function that embeds abstract signs into the balanced Kleene algebra.

```
isBalanced :: Mat Sign → Bool
isBalanced = isDiagonalOne signToBalanced

signToBalanced :: Sign → Balanced
signToBalanced Pos  = PB
signToBalanced Neg = NB
```

The tests *isClustered2* and *isBalanced* are indicators that our approach to graph problems is particularly convenient: we use higher-order functions to abstract the computation scheme, a purely algebraic closure operation and an intersection function, which is reminiscent of a relational intersection.

8.3 The Centrality Problem

As a final application we consider the problem of computing the *betweenness centrality of a vertex*, which also has been considered by Batagelj [Bat94]. Let $G = (V, E)$ be a graph. We define a pseudo-quasimetric[8] on the graph by setting

$$\begin{aligned} d : V \times V &\to \mathbb{R}_{\geqslant 0} \cup \{ \infty \} \ , \\ (v, w) &\mapsto \min \{ |p| - 1 \mid p \in V^* \wedge p \text{ is path from } v \text{ to } w \text{ in } G \} \ , \end{aligned}$$

[8] The function d satisfies the properties of a quasimetric (a non-negative function that satisfies the coincidence axiom and the triangle inequality), but is not quasimetric in the analytical sense, because we allow the value ∞.

where we assume that $\min \emptyset = \infty$. For all $v, w \in V$ we define

$$\gamma_{v,w} := |\{ p \in V^* \mid p \text{ is shortest path from } v \text{ to } w \}| .$$

The value $\gamma_{v,w}$ is called[9] *the geodesic number of u and v*. For all $v, w \in V$ we define

$$\sigma_{v,w} : V \to \mathbb{N} , \quad x \mapsto \begin{cases} \gamma_{v,x} \cdot \gamma_{x,w} & : d(v,w) = d(v,x) + d(x,w) \\ 0 & : \text{otherwise} . \end{cases}$$

The value $\sigma_{v,w}(t)$ denotes the number of shortest paths from v to w that pass through t. We proceed to define

$$W : V \to 2^{V \times V} , \quad t \mapsto \{ (v,w) \in V \times V \mid v \neq w \wedge v \neq t \wedge w \neq t \wedge \gamma_{v,w} \neq 0 \} .$$

For every $t \in V$ the pairs in $W(t)$ denote those pairs of vertices (v,w) that are connected via a non-trivial shortest walk from v to w that starts and ends in a different vertex than t. Clearly, we have for all $t \in V$ that

$$W(t) \leqslant (|V| - 1) \cdot (|V| - 2) ,$$

where the right-hand side of the inequality is simply due to the omission of the condition $\gamma_{v,w} \neq 0$ in the definition of $W(t)$. With this value at hand, one can define the *betweenness centrality* as the following function

$$C_B : V \to \mathbb{Q}_{\geqslant 0} , \quad t \mapsto \sum_{(v,w) \in W(t)} \frac{\sigma_{v,w}(t)}{\gamma_{v,w}} .$$

The idea behind this function is to measure how central a vertex is. For vertices that are intersection points of many shortest paths, the betweenness centrality value is high, otherwise it is low. More intuitively, one can consider the scaled variant

$$C'_B := \frac{1}{(|V| - 1) \cdot (|V| - 2)} \cdot C_B .$$

Then a C'_B value that is close to 1 denotes high centrality, while a value that is close to 0 denotes low centrality.

Regardless of the scaling, the main components for the computation of C_B and C'_B are the values $\gamma_{v,w}$ and $\sigma_{v,w}(t)$ for all $v, w, t \in V$. To compute these values, one can use another Kleene algebra [Bat94]. Let $K_{BC} := (\mathbb{N} \cup \{ \infty \})^2$. Then one defines the two operations

$$+_{BC} : K_{BC} \times K_{BC} \to K_{BC} , \quad ((m,i),(n,j)) \mapsto \left(\min \{ m, n \} , \begin{cases} i & : m < n \\ i + j & : m = n \\ j & : \text{otherwise} \end{cases} \right)$$

and

$$\cdot_{BC} : K_{BC} \times K_{BC} \to K_{BC} , \quad ((m,i),(n,j)) \mapsto (m + n, i \cdot j) .$$

The intuition behind these definitions is that the second component of a K_{BC} value

[9] The name "geodesic" comes from (differential) geometry and means "shortest path".

denotes the number of shortest paths between two vertices and the fist component denotes the length of these paths. Then the addition can be thought of as an alternative of two path sets, where one needs to take the shorter alternative in case there is one and to combine the alternatives otherwise. The multiplication is then the computation of prolonged shortest paths: the length of the concatenation is the sum of the individual length and the number of possibilities to create a shortest path is the product of the individual possibilities. It is simple to see that $0_{BC} := (\infty, 0)$ is neutral with respect to $+_G$ and the value $1_{BC} := (0, 1)$ is neutral with respect to $\cdot_{BC}$. Batagelj shows that

$$(K_{BC}, +_{BC}, \cdot_{BC}, (\infty, 0), (0, 1))$$

is indeed a semiring, called the *geodesic semiring*. Note that while this semiring looks like the product of the tropical semiring with some other semiring over the natural numbers, this is not the case, because the operations of the products are independent of each other, while the addition in K_{BC} uses both the first and the second component to compute the second component alone.

To make the above structure a Kleene algebra, we also need a star closure operation. This operation can be defined as follows:

$$^\star : K_{BC} \to K_{BC}\ , \quad (m, i) \mapsto \begin{cases} (0, \infty) & : m = 0 \wedge i \neq 0 \\ (0, 1) & : \text{otherwise}\ . \end{cases}$$

The structure

$$(K_{BC}, +_{BC}, \cdot_{BC}, ^\star, (\infty, 0), (0, 1))$$

is called the *geodesic Kleene algebra* and Batagelj [Bat94] shows that it is in fact a Kleene algebra. What is more, he shows that computing the Kleene closure of the following variant of the adjacency matrix

$$A_{G,K_{BC}} : V \times V \to K_{BC}\ , \quad e \mapsto \begin{cases} (1, 1) & : e \in E \\ 0_{BC} & : e \notin E \end{cases}$$

results in the fact that for every $v, w \in V$ such that $v \neq w$ we get

$$\left(A_{G,K_{BC}}\right)^+ (v, w) = (d(v, w), \gamma_{v,w})\ .$$

We can use these results for an implementation. First, we define the geodesic Kleene algebra, for which we reuse the tropical Kleene algebra.

```
data Geodesic λ ω =
  Geodesic {geodesicLength :: Tropical λ, geodesicWeight :: Tropical ω}
    deriving (Show, Eq)
```

The first component actually uses the semiring instance for *Tropical* ω, while the second component simply uses the new special value *Max* for ∞. To that end we assume that we have the following two functions at hand

```
tropAdd, tropMult :: Num ν ⇒ Tropical ν → Tropical ν → Tropical ν
```

which are the extensions of the numerical addition and multiplication to the tropical

semiring in the sense that *Min* behaves as a zero value, while *Max* behaves as ∞. With these two functions we can then define the Kleene algebra instance for *Geodesic* as follows.

```
instance (Ord λ, Monoid λ, Num ν) ⇒ Semiring (Geodesic λ ν) where
  Geodesic m i ⊕ Geodesic n j = Geodesic (m ⊕ n) k where
    k | m < n     = i
      | m ≡ n     = i `tropAdd` j
      | otherwise = j
  zero = Geodesic Max Min
  Geodesic m i ⊛ Geodesic n j = Geodesic (m ⊛ n) (i `tropMult` j)
  one = Geodesic Min (Weight 1)
instance (Ord λ, Monoid λ, Num ν) ⇒ KleeneAlgebra (Geodesic λ ν) where
  star (Geodesic l w) = Geodesic MinWeight (f l w) where
    f MinWeight MinWeight = Weight 1
    f MinWeight _         = MaxWeight
    f _         _         = Weight 1
```

We can now use the result about the values in the Kleene closure of a matrix over the geodesic Kleene algebra to compute the centrality of vertices. To avoid recomputations, we compute the betweenness centrality for all vertices as a vector.

```
betweenness :: Mat α → Vec Rational
betweenness m = mkVec (map (λv → (v, between v) vs) where
  between t = sum [σ v w t % γ v w | (v, w) ← pairs, v ≠ t, t ≠ w]
  pairs = [(v, w) | v ← vs, w ← vs, γ v w ≠ 0]
  vs    = vertices m
  σ v w t | dvt `tropAdd` dtw ≡ d v w = gvt · gtw
          | otherwise                 = 0
          where (dvt, gvt) = both v t
                (dtw, gtw) = both t w
  d v w = fst (both v w)
  γ v w = snd (both v w)
  both v w = (fmap getSum (geodesicLength q), tropToInteger (geodesicWeight q))
    where q = matrixAt kcm v w
  kcm = kleeneClosure (fmap (const geodesicLabel) m)
geodesicLabel :: Geodesic (Sum Integer) Integer
geodesicLabel = Geodesic (Weight (Sum 1)) (Weight 1)
matrixAt :: Semiring σ ⇒ Mat σ → Int → Int → σ
matrixAt m row col = m !!! row ! col
```

The main part of the implementation is straightforward: we compute for each vertex

t its betweenness centrality as in the definition of $C_B(t)$. To aid legibility we perform two list comprehensions. The first one generates those pairs of vertices that are connected by a shortest path in the local variable *pairs*. The second one in *between* simply removes from this list those pairs, in which t occurs at some position. The type *Rational* is Haskell's version of arbitrary precision rational numbers and the function $(\%) :: Integer \to Integer \to Rational$ is a smart constructor that creates a rational number from two integers. The function σ is defined exactly as before, up to currying. The interesting parts are the functions γ and d. The basic call for both functions is the query operation to the Kleene closure of the specially labelled matrix m. We label every edge with the value *geodesicLabel*. The geodesic weight of *geodesicLabel* is simply the consideration of 1 in the set $\mathbb{N} \cup \{ \infty \}$. The geodesic length, however, needs to have a *Monoid* instance due to the definition of the *Semiring* instance for *Geodesic*, which requires this instance. This is because the monoid operation is used as the multiplication in the non-extreme case in the tropical semiring. In our case, the operation we are interested in is the addition of integers. Instead of defining a monoid instance on integers, we use the wrapper data type *Sum* from the Haskell module *Data.Monoid*, which provides an additive monoid instance for *Sum* ν, where ν is a numerical type, and the accessor function $getSum :: Sum\ \alpha \to \alpha$ that unwraps the value. Now for γ we need the actual numerical value denoted by the tropical one. To that end we use the following function

```
tropToInteger :: Tropical Integer → Integer
tropToInteger MinWeight  = 0
tropToInteger (Weight w) = w
tropToInteger _          = error "Not a number."
```

that maps *MinWeight* to 0, a *Weight* w to w and yields an error for infinity. A similar approach is required for the function d as well, because we need only the tropical value and no longer the *Sum* context. To that end we define a *Functor* instance for *Tropical*

```
instance Functor Tropical where
  fmap f (Weight w) = Weight (f w)
  fmap _ MinWeight  = MinWeight
  fmap _ MaxWeight  = MaxWeight
```

that simply maps a function over a *Weight* w value and returns the constants (with new types) otherwise. With this definition we can transform *Weight* $(Sum\ \alpha)$ to *Weight* α by using *fmap getSum*, which is exactly what we use in the above implementation.

To see an example application consider the graph from Figure 8.2, which is the same one as provided as an example by Batagelj [Bat94]. Suppose that this graph is represented in Haskell by the constant $exampleCentrality :: Mat\ ()$. We then get the Kleene closure of the modified graph as follows, where we omit all constructors and write values of the type *Geodesic* as pairs for the sake of presentation,

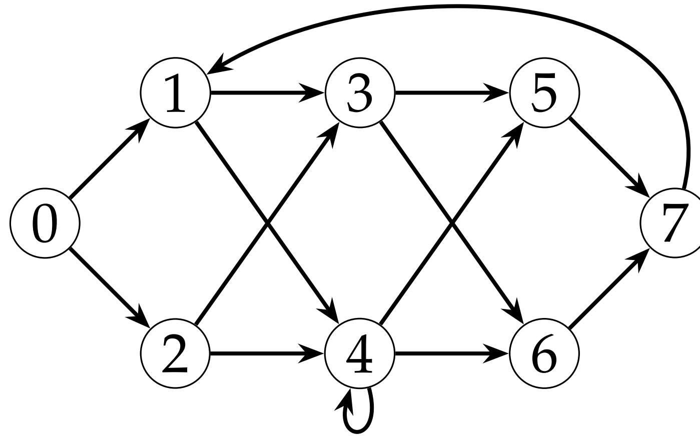


Figure 8.2. An example graph for centrality.

because the values in the result matrix have very long textual representations like *Geodesic* {*geodesicLength* = *Weight* {*weight* = *Sum* {*getSum* = 3}}, *geodesicWeight* = *Weight* {*weight* = 5}}, which are very useful for single values, but are difficult to read in case of many values.

ghci> *kleeneClosure* (*fmap geodesicLabel exampleCentrality*)
0 : (1 | (1,1)) (2 | (1,1)) (3 | (2,2)) (4 | (2,2)) (5 | (3,4)) (6 | (3,4)) (7 | (4,8))
1 : (1 | (4,4)) (3 | (1,1)) (4 | (1,1)) (5 | (2,2)) (6 | (2,2)) (7 | (3,4))
2 : (1 | (4,4)) (3 | (1,1)) (4 | (1,1)) (5 | (2,2)) (6 | (2,2)) (7 | (3,4))
3 : (1 | (3,2)) (3 | (4,2)) (4 | (4,2)) (5 | (1,1)) (6 | (1,1)) (7 | (2,2))
4 : (1 | (3,2)) (3 | (4,2)) (4 | (1,1)) (5 | (1,1)) (6 | (1,1)) (7 | (2,2))
5 : (1 | (2,1)) (3 | (3,1)) (4 | (3,1)) (5 | (4,2)) (6 | (4,2)) (7 | (1,1))
6 : (1 | (2,1)) (3 | (3,1)) (4 | (3,1)) (5 | (4,2)) (6 | (4,2)) (7 | (1,1))
7 : (1 | (1,1)) (3 | (2,1)) (4 | (2,1)) (5 | (3,2)) (6 | (3,2)) (7 | (4,4))
ghci> *betweenness exampleCentrality*
(0 | 0 % 1)
(1 | 37 % 2)
(2 | 5 % 2)
(3 | 9 % 1)
(4 | 9 % 1)
(5 | 13 % 2)
(6 | 13 % 2)
(7 | 17 % 1)

Indeed, the vertex 2 is located on precisely the following shortest walks:

$$(0,2,3),(0,2,4),(0,2,3,5),(0,2,4,5),(0,2,3,6),(0,2,4,6),$$
$$(0,2,3,5,7),(0,2,3,6,7),(0,2,4,5,7),(0,2,4,6,7)\ .$$

There are two shortest walks from 0 to 3 and from 0 to 4, four shortest walks from 0 to 5 and from 0 to 6, as well as eight shortest walks from 0 to 7, which yields that

$$C_B(2) = \frac{1}{2} + \frac{1}{2} + \frac{2}{4} + \frac{2}{4} + \frac{4}{8} = \frac{5}{2},$$

which is exactly the value in the result vector.

Despite some technicality of the definition of the betweenness function, it is still comparatively simple in its structure. The simple implementation is particularly due to the theoretical foundations, which provide the necessary properties of the Kleene closure over the geodesic Kleene algebra [Bat94]. Aside from some type transformations and efficiency improvements[10], the function is basically following the mathematical definition. Also, we benefit from the fact that most of the functions and instances that are required for the above definition have either been defined before or are available through standard Haskell libraries.

8.4 Related Work and Discussion

We have presented a simple technique for computing the number of votes for each alternative. In our approach, we have chosen a purely relational version of the voting matrix, because only the value () occurs in the matrix. The vector-matrix multiplication we used is applicable to arbitrary matrix values, however, because it ignores both the vector and matrix values and merely counts the number of non-zero entries in every column of the matrix. Instead of ignoring these values, we could have used a numerical approach by using the type synonym

type $AppVotingNum = Mat\ Integer$

to represent an approval election. This approach is more flexible in the sense that if all present values in the matrix are non-negative, we can interpret these values as the number of votes a voter assigns to an alternative. If all votes are bounded by a fixed, previously chosen number, this kind of election is known as *range voting*. If all values are equal to 1, we then have the original approval election. In this numerical approach, we can use the vector-matrix multiplication

$(\odot_+) :: Num\ \nu \Rightarrow Vec\ \alpha \rightarrow Mat\ \nu \rightarrow Vec\ \nu$
$(\odot_+) = vecMatMult\ (bigUnionWith\ (+))\ (\lambda_\ _\ row \rightarrow row)$

which ignores the values in the vector, considering all these values to be 1 and then performs a usual, numerical multiplication similar to $(\odot)$. One additional modification is to actually require the vector that denotes the voters to contain numbers as well and to use $(\odot)$ directly. Instead of using our framework of vector-matrix multiplications for such an implementation, one can take actual matrix libraries

[10] The most important improvement is that the Kleene closure is computed exactly once and then used for all vertices.

(possibly with bindings to other languages). For instance, the Haskell libraries *matrix* [Día15] and *hmatrix* [Rui15] provide data types for dense matrices and vectors that are based upon the data type *Vector* from the *vector* library [Les15] which in turn is based upon arrays. Despite the fact that these structures are dense, they are known to be fast for numerical operations.

The actual control that is based upon testing powerset elements in a specific order is clearly inherently inefficient. However, since the underlying problem is NP-complete [HHR07], an exact solution (a minimal set of voters that need to be removed) is difficult to find. In fact, the rationale behind using elections that are computationally difficult to control, for instance where the control is an NP-complete problem, is that it is unlikely that an efficient algorithm for controlling an election exists and that a generate-and-test approach is typically impractical. Two alternative generate-and-test implementations for control are discussed by Berghammer, Danilenko, and Schnoor [BDS14]. The first one relies on relation algebra (with additional, non-standard relations) and an implementation thereof using BDDs [BN05] and the second one uses an implementation in C, which avoids any data structures by interpreting integers as sets. The C implementation is the fastest one, our Haskell implementation is slower than C and the relational implementation is the slowest one. These measurements are, however, strongly biased: the C implementation is a custom-tailored algorithm, the relational implementation is provided in a framework that is general enough to perform arbitrary relational computations, and the Haskell implementation is also based upon a more expressive framework and not optimised for efficiency, but for modularity.

Considering the problems of balanced and clustered graphs, as well as the centrality problem, it is possible to use the work of Dolan [Dol13] to achieve similar implementations, because the solutions to all three problems are based upon the computation of a star or Kleene closure over a certain semiring, which is possible with Dolan's approach with some additional auxiliary functions for the comparison of matrices. In Section 3.7 we have seen that this particular implementation has a worse complexity in terms of time and space consumption than ours, because the Dolan approach uses matrices where every position is filled with a semiring value, while we omit all zero values.

Finally, the Haskell library *igraph* [GS13] provides predicates that check whether a graph is balanced and whether it is clustered, as well as functions that compute the actual cluster partitions and the betweenness centrality of vertices. However, all of these features are achieved using bindings to a C library that provides these very functions and not implemented in Haskell directly.

Chapter 9

Conclusion

In this final chapter we summarise our previous results, discuss related work, including our work in different areas and give some pointers to future research. One major area of interest is the field of functional logic programming, which combines the paradigms of functional programming with logical features. In our examples we restrict ourselves to the language Curry [Han12], which is syntactically similar to Haskell, but provides explicit non-determinism and logical variables.

9.1 Summary

In our approach to graph algorithms in Haskell we have first considered a classical application of computing the Kleene closure of a matrix over a Kleene algebra (with tests). Our results from Chapter 3 have indicated that the well-known row-based approach to adjacency matrices is a good choice for graph representations in Haskell. This observation is particularly true in case of explicitly sparse matrices, in which we can omit zero values and thus save a significant amount of space. Not only did our approach result in a short and algebraically flavoured function, but it was also more efficient than a simple implementation with arrays and also more efficient than the elegant implementation of Dolan [Dol13] who used the blockwise representation of matrices for the computation of the star closure.

Despite the fact that Chapter 3 focussed on a single algorithm, the modularity that usually comes with functional programming yielded the main data types *Mat* and *Vec*, as well as several auxiliary functions that we have used in later chapters. These auxiliary functions included purely algebraic ones like $(+_{\mathsf{v}})$ for the addition of vectors over an idempotent semiring, but also functions that are typical for functional programming, when it comes to dealing with containers like *filterVec*, which keeps only those key-value pairs in a vector that satisfy a certain predicate.

In Chapter 4 we took a detailed look at a scheme that is ubiquitous in graph algorithms, namely the computation of the successors of a set of vertices combined with the collection of some information along the way. While it is well-known that in many cases this can be abstracted to a vector-matrix multiplication over a semiring, we have taken a more liberal approach by abstracting the actual computation scheme and relaxing the condition of an underlying structure. We have noted that the scheme is an instance of the well-established generate-then-prune paradigm in functional

programming. This computational structure is based upon non-strictness and usually works by defining an over-approximation of the problem solution that creates a large structure with many unnecessary branches and then trimming these branches until only the desired ones remain. Since all values are computed only once they are demanded, there is only little overhead associated with this approach, but the resulting code is often simpler than the removal of the redundant parts on the fly.

Aside from some classical examples of the convenience of this abstracted vector-matrix multiplication, we have also provided a non-trivial application that can be used to transpose matrices in Section 4.3.5. We have discussed possible correctness proofs for properties of vector-matrix multiplications in Section 4.7 and proved the correctness of the transposition function. Additionally, we have shown that starting with a purely relational foundation one can define a reachability scheme, which we have discussed in Section 4.4. From this scheme we have derived two useful applications, namely the reachable vertices of a vertex set and the computation of those vertices in a target set that are reachable along a shortest path from a start set. Both functions are interesting in their own right as functions for a graph library. However, we used both functions in several non-trivial applications: the reachability function was used for minimum vertex covers in Section 6.3 and for minimum cuts in Section 7.3, while the shortest path function was particularly useful for the computation of maximum matchings in Section 5.3 and the computation of maximum flows in Section 7.2. It is important to note that in the last example the shortest path function is responsible for the fact that the maximum flow function is polynomial in the size of the graph. We have discussed this in Section 7.1.

The final main result of Chapter 4 was a function that computes a maximal set of pairwise disjoint shortest paths from one set to another which we have implemented in Section 4.5. We used this function for an implementation of the improved maximum matching function due to Hopcroft and Karp [HK73]. The resulting function *disjointPaths* is an interesting construction from the computational point of view. The basic idea due to Dinitz [Din06] is to perform a combination of a breadth-first search and a depth-first search on the graph. In our implementation we use a reachability strategy that is very similar to a breadth-first search with a special vector-matrix multiplication that collects too much information. We then use a pruning technique, which is very similar to the one presented by King and Launchbury [KL95] for their depth-first search implementation. Thus the *disjointPaths* function is in fact a combination of the two searches mentioned above, but in a functional setting.

We continued with applications of our abstracted reachability and vector-matrix multiplication scheme. In Chapter 5 we have considered the maximum matching problem in bipartite graphs and implemented two different versions of a function that finds such a matching: a canonic one and an efficient one based upon the computation of disjoint paths from Section 4.5. To obtain a rather short implementation we have provided a purely relational characterisation of the existence of augmenting paths and proved this result. Also, since being bipartite is difficult to express in the graph

type, we have dealt with predicates that check whether a graph is bipartite or not. To that end we have developed two functions. The first one has a cubic complexity in the number of vertices and merely answers the question, whether the graph is bipartite or not. The second function has a quadratic complexity in the number of vertices. Additionally, this more efficient function also computes a concrete bipartition in case the graph is bipartite. These predicates allow a safe implementation of a maximum matching function, which actually checks if the argument is bipartite. The function *findBipartition* is used explicitly later for the computation of a minimum vertex cover in Section 6.3, because Kőnig's construction of a vertex cover in a bipartite graph requires an actual bipartition, while the computation of a maximum matching in a bipartite graph does not.

In Chapter 6 we have considered minimum vertex covers in bipartite graphs. To that end we used the informal specification of such a vertex cover given by Diestel [Die00], which is based upon Kőnig's theorem, and transformed it into a formal definition in the terminology of relation algebra. The resulting relational definition of a minimum vertex cover is concise and comparatively simple, because it can be expressed with a small number of relational operations. We have provided a purely relational proof of Kőnig's theorem in Section 6.2 and an implementation of the function that computes minimum vertex covers in bipartite graphs in Section 6.3.

While most of the previous results used a relational context only, the results of Chapter 7 dealt with a function that computes a maximum flow in a network and another one that computes a minimum cut in a network. The latter is obtainable from an intermediate result of the former with purely relational operations. To compute a maximum flow in a network, however, one needs to consider the edge labels, which are the capacities of the edges. In our approach, both the vector-matrix multiplication generators and the function *shortestWith* are parametric in the matrix values. With this abstraction we were able to use the edge labels of network graphs and to implement a maximum flow function in a very similar fashion as the function that computes a maximum matching in a bipartite graph.

Finally, in Chapter 8 we have considered two types of problems with an algebraic basis. The first one is the manipulation of approval voting. We have shown that it is very simple to express the required computations in terms of (linear) algebraic functions. This algebraic foundation is in turn rather simple to implement in Haskell, particularly in our vector-matrix multiplication framework. We have then used a simple generate-and-test approach to find a solution for the NP-complete problem of approval voting manipulation through the removal of voters. The second algebraic application considered certain graph properties that are of interest in social networks, namely clustered and balanced graphs, as well as the betweenness centrality of vertices. We have used the results of Batagelj [Bat94] to solve these three problems using special Kleene algebras and the Kleene closure from Chapter 3.

In all our implementations we have used a concrete model for vectors and matrices, but only to provide (mostly) complete implementations. We have discussed possible

abstractions in Section 4.6. On several occasions we have used the function *unVec* that transforms a vector into an association list. In our particular implementation this function is trivial, because vectors are just wrapped association lists. However, using another implementation for vectors, one can simply substitute another implementation of this function and all of our code remains valid.

Overall, we have shown how one can combine (relation-)algebraic modelling with functional programming to obtain solutions for a variety of problems. Clearly, most of our functions are prototypical in the sense that we rarely defined general purpose functions beforehand to use them later, but usually provided the required functions along the way. In our *gwaf* library [Dan15] we remedy some of these shortcomings of the presentation and provide a more parametric implementation of most functions discussed in this dissertation. Still, in most cases the resulting functions are either simple themselves in terms of length and construction or compositions of simple functions. Even more technical problems like the flow problem or the computation of disjoint paths can be solved modularly in our framework.

9.2 Future Work

We now wish to outline some areas for future research in the field of functional graph algorithms. This section is divided in two parts, the first one deals with purely functional programming, while the second one considers applications in functional logic programming. The second part also contains references to results that have been published already [Dan14a].

9.2.1 Functional Programming

One important part of functional libraries are optimisations. These concern both, the actual implementation as well as compiler flags that deal with inlining complex functions or rewriting function definitions altogether to fit into a certain pattern, which is simple to compile to efficient code. We have not dealt with such optimisations so far. While some of the compiler rules are usually rather straightforward, some require additional testing and comparison. We believe that there is quite some room for improvement in terms of efficiency (both time and space) because many of our functions are implicitly based on lists and making these foundations explicit allows well-established compiler optimisations like the *foldr* / *build* rule [GLJ93] to fuse away intermediate structures.

Additionally, we wish to study the type class abstractions that we have discussed in Section 4.6 in more detail. In particular, the interaction of the various heterogeneous set operations may lead to further insights concerning possible choices in data structures. We have hinted at the semantics of the type class functions before, but a proper definition requires the derivation of sensible algebraic rules, particularly when

it comes to the interaction of different classes. For instance, one might require certain absorption laws of the type classes *Unionable* and *Intersectable*. Also, type classes usually become more efficient by an explicit specification of how to handle particular instances. These specifications constitute another area in which compiler annotations can lead to optimised code, while still offering users a very general approach to set operations.

As for actual graph problems, there are interesting related problems for those we have considered in Chapter 5 to Chapter 8, which we now discuss individually.

9.2.1.1 Non-Bipartite Matching

While we have dealt with the case of bipartite graphs only, the maximum matching problem is of great interest for arbitrary graphs as well. For instance, it can be applied to solve the "lab partner problem", where one wishes to assign as many people as possible to pairs, subject to, say, time restrictions, without having the possibility to divide them into two groups such that each pair consist of exactly one person from each group.

In the non-bipartite case, the theorem of Berge, Lemma 5.1.4, remains true, since it is stated for general graphs. However, the search for an augmenting path is not as simple as in the case of a bipartite graph. We have hinted at this issue before while we discussed shortcuts in augmenting candidates in Lemma 5.2.3. Recall that we have shown in Figure 5.3 that in the non-bipartite case there may be augmenting candidates that cannot be transformed into an augmenting path. In the example of Figure 5.3 our function *augmentingPath* from Section 5.2 will first call *shortestWith*, which will (correctly) find that both elements of the set $\{0, 5\} = \mathsf{uncovered}(M)$ are reachable along an alternating *walk* that starts in $\{0, 5\}$. Unfortunately, since none of these walks is a path, the result provided by *augmentingPath* is not an augmenting path and the resulting "improvement" is no longer a matching. This is due to the fact that the search is performed in the product graph of $E \setminus M$ and M. In case of a bipartite graph, a path in the product graph can be unfolded to a path in the original graph. However, in case of a non-bipartite graph, this construction is no longer possible, because odd cycles can occur.

It may seem that this is a problem that is related to our implementation only, but in fact it is not and will affect every kind of classical path search algorithm. Without going into detail, the heart of this problem is the fact that the order, in which vertices are visited and marked as such is of crucial importance. In our case, we do not mark the intermediate vertex of the product $(E \setminus M) \cdot M$, which is correct in the bipartite case, but may lead to results that are not a matching in the general case. Adding this intermediate vertex prematurely can, however, discard a path too early and thus lead to results that are matchings, but not maximum matchings.

This general problem is solved by Edmonds [Edm65] using a special technique for dealing with odd cycles along augmenting candidates. The resulting algorithm

is more technical in nature than the one obtained from Berge's lemma, but yields a maximum matching in arbitrary graphs in polynomial time. To the best of our knowledge there is no relational version of this algorithm at the time of this writing. We are confident that Edmonds's solution can be expressed in a more algebraically flavoured fashion, which then can be implemented in our framework. The necessary relational expression is an interesting starting point for future research.

9.2.1.2 Weighted Matching

Aside from the discrete matching problem, where the edges carry no information, there is the so-called *weighted matching problem*. For this problem one additionally assumes a weight function $u : E \to \mathbb{Q}_{\geqslant 0}$ such that $u(x,y) = u(y,x)$ for all $(x,y) \in E$ and defines a matching $M \subseteq E$ to be u-maximum if and only if

$$\sum_{e \in M} u(e) = \max \left\{ \sum_{e \in D} u(e) \;\middle|\; D \subseteq E \wedge D \text{ matching} \right\}$$

holds. This problem generalises the original matching problem, because one can choose $u = e \mapsto 1$, which results in

$$\sum_{e \in M} u(e) = |M| \,.$$

This problem can be subdivided into two cases again, the first one being the bipartite version and the second one the general one. The bipartite version has been solved by Kuhn [Kuh55] and relies on a special type of path search, similar to the concept of an augmenting path. The resulting algorithm uses a so-called *weighted vertex cover*, which we outline briefly in Section 9.2.1.3. The interesting observation about this is that it is dual to the one we give in Section 6.1: the algorithm requires a vertex cover to compute a matching, while the construction in Section 6.1 requires a matching to compute a vertex cover. The general case is solvable by adapting the non-bipartite solution [Edm65] to the case of weighted matchings. The resulting solution is due to Gabow [Gab85].

The weighted matching problem is known to have algebraic roots. In fact, Kuhn [Kuh55] derives his solution from an approach with matrices. However, the solution relies heavily on the components of matrices and most arguments are provided pointwise, rather than as statements about graph matrices. Still, it is of great interest to derive a purely algebraic version of both algorithms, because such a version is likely to be close to an executable implementation. Finding these algebraic descriptions is another topic for future research.

9.2.1.3 Weighted Vertex Cover

Another variation of the vertex cover problem discussed so far is the *weighted vertex cover problem*. Given a symmetric graph $G = (V, E)$ and a function $u : E \to \mathbb{Q}_{\geqslant 0}$ with

the property that for all $(x,y) \in E$ we have $u(x,y) = u(y,x)$, a function $c : V \to \mathbb{Q}$ is called a u-vertex-cover if and only if

$$\forall\, x,y \in V : (x,y) \in E \Rightarrow c(x) + c(y) \geqslant u(x,y) \ .$$

A u-vertex-cover c is called a minimum u-vertex-cover if and only if the following holds:

$$\sum_{v \in V} c(v) = \min \left\{ \sum_{v \in V} d(v) \;\middle|\; d : V \to \mathbb{Q}_{\geqslant 0} \wedge d \text{ vertex cover} \right\} \ .$$

This abstraction is similar to the one of weighted matchings that we have discussed in Section 9.2.1.2. It is well-known that choosing $u = e \mapsto 1$ we get that for every vertex cover $C \subseteq V$ its characteristic function

$$\chi_C : V \to \mathbb{Q}_{\geqslant 0} \ , \quad v \mapsto \begin{cases} 1 & : v \in C \\ 0 & : \text{otherwise} \end{cases}$$

is a u-vertex-cover.

In case of a graph that is additionally bipartite, the problem is solvable in polynomial time using the algorithm of Kuhn [Kuh55], which we have already mentioned in Section 9.2.1.2, because it computes both, a minimal u-vertex-cover and a maximal u-matching at the same time. In the general case the problem is NP-complete, because the special case of regular vertex covers in arbitrary graphs is already NP-complete [Kar72].

It is of particular interest to solve the weighted vertex cover in an algebraic fashion, because this is likely to allow an algebraic solution for the weighted matching problem. Also, in the non-bipartite case one can consider an approximation algorithm similar to the one we discussed in Section 6.4 and implement it in our framework as well.

9.2.1.4 Minimum Cost Flows

The maximum flow problem in a network has a related problem, in which one considers not only the actual flow value that denotes the amount of goods transported through the network, but also the costs that are associated with this transport. Let $N = (V, E, s, t, c)$ be a network and $p : E \to \mathbb{Q}$, where p denotes the *price* of each edge. One then defines the cost of an edge function as

$$\mathsf{cost} : \mathbb{Q}^E \to \mathbb{Q} \ , \quad f \mapsto \sum_{e \in E} f(e) \cdot p(e) \ .$$

For every $f : E \to \mathbb{Q}_{\geqslant 0}$ and every $r \in \mathbb{Q}_{\geqslant 0}$ one defines

$$f \text{ is an } r\text{-flow} \ :\Longleftrightarrow \ f \text{ is a flow} \ \wedge \ |f| = r \ ,$$

where $|f|$ is the flow value as defined in Definition 7.1.2. Given an $r \in \mathbb{Q}_{\geqslant 0}$ one then wishes to find an r-flow f that additionally satisfies

$$\mathsf{cost}(f) = \min \left\{ \mathsf{cost}(g) \;\middle|\; g \in (\mathbb{Q}_{\geqslant 0})^E \wedge g \text{ is an } r\text{-flow} \right\} \ ,$$

if there exists such a flow. In the positive case a flow with the above property is called *minimum cost flow (with respect to p)*.

The solution to this problem has first been presented by Klein [Kle67]. The basic idea is similar to the one of the Ford-Fulkerson theorem and characterises minimum cost flows using the notion of augmenting cycles, which are a special type of cycles in the residual graph (with respect to the given flow). Just as in the case of the Ford-Fulkerson theorem, the direct implementation of this characterisation results in a non-polynomial algorithm. However, this has been remedied in case of the minimum cost flow problem by Goldberg and Tarjan [GT88], who provided a polynomial algorithm, which in each improvement iteration finds an augmenting cycle that has minimum average costs. This concept bears some resemblance to the improvement of Edmonds and Karp [EK72] that we have discussed in Section 7.1. However, this is a quite sophisticated solution, because the search for such a cycle is not as simple as the search for a shortest path between two vertices.

It is known that the minimum cost flow problem can be solved using linear programming, because both the side conditions, as well as the function one wishes to minimise are linear. For instance, we have $\mathsf{cost}(f) = \langle f|p\rangle$, where $\langle x|y\rangle$ denotes the scalar product of x and y. This knowledge already suggests that the problem is solvable using linear algebra. We suspect that the canonical (non-polynomial) algorithm for finding minimum cost flows can be incorporated in our framework quite well. For improvement in the sense of Goldberg and Tarjan [GT88] one needs to find an augmenting cycle with minimal average costs. Whether or not this is possible with algebraic means, particularly ones suited for an implementation in our approach, is an interesting question for future research.

9.2.1.5 Election Winners and Control

We have dealt with only one example of elections, namely approval voting and a control thereof by removing voters. As we have already mentioned in Section 8.1 there are other types of control, like precluding alternatives [HHR07] or removing alternatives. Aside from the different types of controls, the actual election protocol can be variated as well. For example, in *plurality voting* each voter assigns a strict linear order to the alternatives, which is to say that every voter ranks all alternatives with distinct ranks. A winner is then any alternative with the highest number of rank one assignments.

Essentially, for every type of election there are several types of controls: there is constructive (a chosen alternative wins) and destructive (a chosen alternative does not win) control, which is each possible by adding or removing voters or alternatives. The complexities of these kinds of controls are known in many cases [HHR07]. While some problems are considered to be simple in the sense that the control requires only a polynomial effort in the size of the election, others are known to be computationally hard. Several of these problems have been already dealt with

in terms of relation algebra [BDS14; BS14; Ber14]. We have already seen that many results that are expressible in terms of relation algebra are rather simple to implement in our framework. It is interesting to see how such implementations compare to purely relational ones, for instance in terms of BDDs [BN05].

As we have mentioned before, one motivation behind different election systems is to obtain one that is resistant[1] to control: computing a control possibility is computationally difficult, which is usually expressed in terms of being NP-complete. Despite the fact that NP-complete problems are difficult to solve exactly and efficiently, in many cases a generate-and-test approach is surprisingly fast, albeit still exponential in the problem size. For example, constructive control of approval voting by removal of voters is NP-complete, but in elections with a small number of voters an exhaustive approach is sufficiently fast to be practically applicable. The particular times for different NP-complete controls would provide insights into the actual practicality of resistant elections.

9.2.1.6 Social Measures

We have considered three measures that are of relevance in social networks in Section 8.2 and Section 8.3. Aside from these three measures, there exist others like *degree centrality* or *closeness centrality* [RBSG11]. The Haskell library *igraph* [GS13] provides functions for closeness centrality as well as betweenness centrality and it would be of interest to compare their respective complexities. While we avoided the comparison of different languages so far, in this particular case one would actually compare Haskell functions among each other and not necessarily custom-tailored algorithms in C (which are typical for C) with a compositional and likely more parametric approach in Haskell (which is typical for Haskell).

One further application in this field is the actual computation of clusters in a clustered graph. These clusters can be computed by partitioning the graph with respect to a certain equivalence relation, which is related to the reachability relation for symmetric graphs [Ril69]. In our approach, one could use a similar technique as we used for the computation of a bipartition of a non-connected graph in Section 5.5.3 and compute maximal sets of pairwise reachable vertices in a certain subgraph of the symmetric closure of the input graph. The *igraph* library provides this functionality, which is why an implementation in our framework would be of interest as well.

Some of the measures mentioned above are dealt with in an algebraic fashion by Rusinowska, Berghammer, Swart, and Grabisch [RBSG11], but rely on non-standard relations, like a relation for the comparison of set cardinalities or the powerset relation. It is of interest to see whether these requirements can be relaxed using different algebraic means like semirings or Kleene algebras and then implemented

[1]There is also the notion of *control immunity*. However, the Gibbard-Satterthwaite theorem [Gib73; Sat75] states that such an election with three or more alternatives is always dictatorial, which is to say that a single voter decides the outcome of the election.

with our current means. Additionally, the results of Batagelj [Bat94] which we used for all of our implementations in Section 8.2 and Section 8.3 contain an additional Kleene algebra that is an extension of the geodesic Kleene algebra, which is called *geosetic Kleene algebra*. This Kleene algebra can be used to compute for all pairs of vertices the length of a shortest path between these vertices, as well as all vertices that are located on these shortest paths. The definition of this Kleene algebra in Haskell is too technical to be presented here as an example application. Still, the underlying problem is obviously of interest and its solution in Haskell could be implemented in the future.

9.2.2 Functional Logic Programming

On several occasions, particularly in Chapter 5 to Chapter 7, the functions we have implemented are based upon the computation of an intermediate result with a special property. In case of matchings and flows, we require a certain type of path, while for vertex covers we require a maximum matching. Note that in all cases the objects of interest are described with the indefinite article, because the actual choice of such an object does not matter, as long as the desired property is satisfied. Also, in all cases the result, say a maximum matching, is usually not unique. An interesting question is now, whether the choices made along the way somehow reflect upon the result.

We have considered this problem before [Dan14a] in the functional logic programming language Curry [Han12]. In this work we considered the independence of choices as an instance of non-determinism and prototypically studied the effect of these choices on the result. Consider for example the task of finding a path that starts in a given vertex, and then traverses edges of a list of graphs *gs* in cyclic sequence, which is to say that the i-th edge of the path (starting with $i = 0$) belongs to the graph *gs* !! (*i* ‘*mod*‘ *length gs*) and ends in a given set of vertices. Such a function can be used to find an augmenting path in a very declarative fashion. In Curry, we can implement this function as follows[2].

```
type Path = [Vertex]
pathThrough :: [Graph] → Vertex → [Vertex] → Path
pathThrough gs from tos = find [] from gs where
  find vis s (g : gs)
    | s ∈ tos                          = [s]
    | isEdge g s i ∧ not (i ∈ vis) = s : find (s : vis) i (gs ++ [g])
    where i free
```

Note that there are two crucial differences to a similar Haskell implementation. Most

[2]We take a simple approach with a list of visited vertices rather than a more efficient structure only for the sake of presentation. For the same reason we use the inefficient construction $gs +\!\!+ [g]$, which can be easily replaced by an efficient one using functional lists [Hug86] that have a constant time concatenation function.

importantly, the intermediate vertex for the next step is a free variable, which indicates that Curry will search through all possible vertices to find a fitting one. Also, the guard conditions are incomplete and there is no *otherwise* case. In Haskell this would result in an explicit error, while in Curry functions may return no result and in this case no result is returned if there is no path from the start vertex to the target set. For instance, calling *pathThrough* with an empty list of graphs returns no path.

The explicit non-determinism can be observed in the interactive mode of the KiCS2 compiler [BHPR11]. If a function has more than one possible result, KiCS2 asks the user whether more results are desired. In the particular case of maximum matchings, we have implemented a path search function that is explicitly non-deterministic and then used it in our maximum matching function, which makes the maximum matching function non-deterministic as well. This implementation comes with two main benefits.

First, all possible maximum matchings in a given graph are listed in sequence. With a little variation one can also return a list of paths chosen in each iteration to see, in which direction the recursive function descends. Consider again the graph from Figure 5.1 and suppose that it is represented in Curry by the value *graph*. Then KiCS2 yields the following results, where *showEdges* is an auxiliary function that lists all edges in a graph as pairs.

```
kics2> showEdges (maximumMatching graph2)
[(0,1),(1,0),(2,5),(3,6),(4,7),(5,2),(6,3),(7,4)]
More values? [Y(es)/n(o)/a(ll)] Y
« four more times the same result »
More values? [Y(es)/n(o)/a(ll)] Y
[(0,1),(1,0),(2,5),(3,6),(5,2),(6,3),(7,8),(8,7)]
More values? [Y(es)/n(o)/a(ll)] n
```

The first result is the maximum matching from from Figure 5.1(b) and the second one has the edges $(4,7),(7,4)$ replaced with $(7,8),(8,7)$, which obviously is a maximum matching as well.

The second benefit of the declarative approach is that it does not depend on any particular vertex order. As we have discussed in Section 9.2.1.1, this order is responsible for the inability to find augmenting paths with the usual means in a non-bipartite graph. In Curry, the vertex order is delegated to the search for intermediate vertices and thus works for non-bipartite graphs as well. However, since all vertex orders are considered implicitly, the resulting function is no longer polynomial in the size of the graph, which yields long computation times for rather small graphs already.

It is very convenient to have the possibility to observe the different branches created by implicit non-determinism, particularly when it comes to presentation as, for instance, in a lecture about graph theory. It also allows to actually find all maximum matchings without any additional overhead, but merely by querying the

compiler for all possible solutions. In case of unique maximum matchings, one can observe the confluence of the algorithm and count the number of times the unique result can be obtained.

Aside from the maximum matching problem, we have also considered path finding functions and a maximum flow function before [Dan14a]. Using a non-deterministically computed maximum matching we can also compute minimum vertex covers as we did in Section 6.3, because, again, any maximum matching can be used to compute a minimum vertex cover and it can be of interest to find all minimum vertex covers in a graph. As for more sophisticated applications, note that the matching improvement that we have implemented in Section 5.4 can be considered non-deterministic as well, because it is based upon the computation of a set of shortest, pairwise disjoint augmenting paths, but any such set is suitable for a matching improvement.

In Section 3.2 we have mentioned that the particular decomposition of matrices for a computation of the Kleene closure in the style of Dolan [Dol13] does not matter. Similarly, in our particular implementation of the Kleene closure we only need a vertex order and we can abstract it as we have discussed in Section 3.5. Another abstraction is to explicitly exploit the non-determinism and to avoid an actual choice of an order altogether. Curry provides functions that guide the search through the search space according to different strategies, most notably the breadth-first strategy and the depth-first strategy. An interesting application in this area is to compare these different search strategies in terms of actual complexities and possibly to implement additional strategies that are particularly suited for providing good vertex orders for the computation of the Kleene closure.

To summarise, research in the area of functional logic programming including graphs can focus on the non-determinism that is ubiquitous in graph theory and make it explicit to observe different or confluent results. Since Curry comes with functional features as well, many parts of our framework[3] can be realised in Curry as well. On several occasions concrete choices, like the function *maybeSome*, should be replaced with proper non-deterministic ones. It is reasonable to assume that one can come up with a generic approach to counting, enumerating and returning all different branches of a non-deterministic graph function as well as checking whether its result is unique.

[3]The abstractions from Section 4.6 are an exception, because at the time of this writing Curry does not support higher-kind type classes like *Functor*.

Bibliography

[AAM03] Eric Allender, Vikraman Arvind, and Meena Mahajan. "Arithmetic Complexity, Kleene Closure, and Formal Power Series". In: *Theory of Computing Systems* 36.4 (2003), pp. 303–328.

[Alu09] Paolo Aluffi. *Algebra: Chapter 0*. American Mathematical Society, 2009.

[Bat94] Vladimir Batagelj. "Semirings for Social Networks Analysis". In: *The Journal of Mathematical Sociology* 19.1 (1994), pp. 53–68.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[BDHS16] Rudolf Berghammer, Nikita Danilenko, Peter Höfner, and Insa Stucke. *Cardinality of Relations with Applications*. Submitted for publication. 2016.

[BDS14] Rudolf Berghammer, Nikita Danilenko, and Henning Schnoor. "Relation Algebra and RelView Applied to Approval Voting". In: *Proceedings of the 14th International Conference on Relational and Algebraic Methods in Computer Science, RAMiCS 2014, Marienstatt, Germany, 2014.* Ed. by Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller. Vol. 8428. Lecture Notes in Computer Science. Springer, 2014, pp. 309–326.

[Ber08] Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Vieweg-Teubner, 2008.

[Ber11] Rudolf Berghammer. "A Functional, Successor List Based Version of Warshall's Algorithm with Applications". In: *Proceedings of the 12th International Conference on Relational and Algebraic Methods in Computer Science, RAMiCS 2011, Rotterdam, The Netherlands, 2011.* Ed. by Harrie C. M. de Swart. Vol. 6663. Lecture Notes in Computer Science. Springer, 2011, pp. 109–124.

[Ber14] Rudolf Berghammer. "Relation Algebra, RelView, and Plurality Voting". In: *Proceedings of the 16th International Workshop on Computer Algebra in Scientific Computing, CASC 2014, Warsaw, Poland, 2014*. Ed. by Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov. Vol. 8660. Lecture Notes in Computer Science. Springer, 2014, pp. 13–27.

[Ber57] Claude Berge. "Two Theorems in Graph Theory". In: *PNAS*. Vol. 43 (9). National Academy of Sciences, 1957, pp. 842–844.

[BG03] Andreas Blass and Yuri Gurevich. "Algorithms: A Quest for Absolute Definitions". In: *Bulletin of the EATCS* 81 (2003), pp. 195–225.

[BHPR11] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. "KiCS2: A New Compiler from Curry to Haskell". In: *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Prog. (WFLP 2011)*. Vol. 6816. Lecture Notes in Computer Science. Springer, 2011, pp. 1–18.

[BHS16] Rudolf Berghammer, Peter Höfner, and Insa Stucke. "Cardinality of Relations and Relational Approximation Algorithms". In: *Journal of Logical and Algebraic Methods in Programming* 85.2 (2016), pp. 269–286.

[BN05] Rudolf Berghammer and Frank Neumann. "RelView – An OBDD-Based Computer Algebra System for Relations". In: *Proceedings of the 8th International Workshop on Computer Algebra in Scientific Computing, CASC 2005, Kalamata, Greece, 2005*. Ed. by Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov. Vol. 3718. Lecture Notes in Computer Science. Springer, 2005, pp. 40–51.

[BS14] Rudolf Berghammer and Henning Schnoor. "Control of Condorcet Coting: Complexity and a Relation-Algebraic Approach". In: *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, 2014*. Ed. by Ana L. C. Bazzan, Michael N. Huhns, Alessio Lomuscio, and Paul Scerri. IFAAMAS/ACM, 2014, pp. 1365–1366.

[Chr12] Jan Christiansen. "Investigating Minimally Strict Functions in Functional Programming". Dissertation. Christian-Albrechts-Universität zu Kiel, 2012.

[CLRS01] Thomas H. Conway, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[Con71] John H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London. Republished in 2012 by Dover Publications, New York, 1971.

[CP13] Koen Claessen and Michal H. Palka. "Splittable Pseudorandom Number Generators Using Cryptographic Hashing". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, 2013*. Ed. by Chung-chieh Shan. ACM, 2013, pp. 47–58.

[Dan12] Nikita Danilenko. "Using Relations to Develop a Haskell Program for Computing Maximum Bipartite Matchings". In: *Proceedings of the 13th International Conference on Relational and Algebraic Methods in Computer Science, RAMiCS 2012, Cambridge, UK, 2012.* Ed. by Wolfram Kahl and Timothy G. Griffin. Vol. 7560. Lecture Notes in Computer Science. Springer, 2012, pp. 130–145.

[Dan14a] Nikita Danilenko. “Exploring Non-Determinism in Graph Algorithms”. In: *Proceedings of the 28th Workshop on (Constraint) Logic Programming (WLP 2014), Proceedings of the 23rd International Workshop on Functional and (Constraint) Logic Programming, Wittenberg, Germany, 2014*. Ed. by Stefan Brass and Johannes Waldmann. Vol. 1335. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 125–139. URL: `http://ceur-ws.org/Vol-1335/wflp2014_paper4.pdf`.

[Dan14b] Nikita Danilenko. “Functional Kleene Closures”. In: *24th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2014, Canterbury, UK, 2014. Revised Selected Papers*. Ed. by Maurizio Proietti and Hirohisa Seki. Vol. 8981. Lecture Notes in Computer Science. Springer, 2014, pp. 241–258.

[Dan14c] Nikita Danilenko. “Graph Problems and Vector-Matrix Multiplications in Haskell”. In: *15th International Symposium on Trends in Functional Programming, TFP 2014, Soesterberg, The Netherlands, 2014*. Ed. by Jurriaan Hage and Jay McCarthy. Vol. 8843. Lecture Notes in Computer Science. Springer, 2014, pp. 51–67.

[Dan15] Nikita Danilenko. *`gwaf` – Graphs With an Algebraic Flavour*. 2015. URL: `https://github.com/nikitaDanilenko/gwaf`.

[Dav67] James A. Davis. “Clustering and Structural Balance in Graphs”. In: *Human Relations* 20.2 (1967), pp. 181–187.

[Día15] Daniel Díaz. *The `matrix` Package*. Version 0.3.4.3. Mar. 2015. URL: `http://hackage.haskell.org/package/matrix`.

[Die00] Reinhard Diestel. *Graph Theory, 2nd Edition*. Vol. 173. Graduate texts in mathematics. Springer, 2000.

[Din06] Yefim Dinitz. “Dinitz’ Algorithm: The Original Version and Even’s Version”. In: *Essays in Memory of Shimon Even*. Ed. by Oded Goldreich, Arnold Rosenberg, and Alan Selman. Vol. 3895. Lecture Notes in Computer Science. Springer, 2006, pp. 218–240.

[Dol13] Stephen Dolan. “Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra”. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, USA, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 101–110.

[DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2nd edition)*. Cambridge University Press, 2002.

[Edm65] Jack Edmonds. “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.

[EK72] Jack Edmonds and Richard Manning Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *Journal of the ACM* 19.2 (1972), pp. 248–264.

[Ell12] Conal Elliott. *Reimagining Matrices*. 2012. URL: `www.conal.net/blog/posts/reimagining-matrices`.

[EM14] Martin Erwig and Ivan Lazar Miljenovic. *The `fgl` Package*. Version 5.5.0.1. Apr. 2014. URL: `http://hackage.haskell.org/package/fgl`.

[Erw01] Martin Erwig. "Inductive Graphs and Functional Graph Algorithms". In: *Journal of Functional Programming* 11.05 (2001), pp. 467–492.

[FF62] Lester Randolph Ford Jr. and Delbert Ray Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[Fis] Sebastian Fischer. *Tabulated Binomial Coefficients*. URL: `www-ps.informatik.uni-kiel.de/~sebf/%20haskell/tabulated-binomial-coefficients.lhs.html`.

[FK98] Hitoshi Furusawa and Wolfram Kahl. *A Study on Symmetric Quotients*. Tech. rep. 1998-06. Fakultät für Informatik, Bundeswehruniversität München, 1998.

[Gab85] Harold N. Gabow. "A Scaling Algorithm for Weighted Matching on General Graphs". In: *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA,1985*. IEEE Computer Society, 1985, pp. 90–100.

[Gal11] Jean Gallier. *Discrete Mathematics*. Springer, 2011.

[Gib73] Allan Gibbard. "Manipulation of Voting Schemes: A General Result". In: *Econometrica* 41.4 (1973), pp. 587–601.

[GLJ93] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. "A Short Cut to Deforestation". In: *FPCA*. 1993, pp. 223–232.

[GS13] George Giorgidze and Nils Schweinsberg. *The `igraph` Package*. Version 0.1.1. Jan. 2013. URL: `http://hackage.haskell.org/package/igraph`.

[GT88] Andrew V. Goldberg and Robert Endre Tarjan. "Finding Minimum-Cost Circulations by Canceling Negative Cycles". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, 1988, Chicago, Illinois, USA*. Ed. by Janos Simon. ACM, 1988, pp. 388–397.

[Han12] Michael Hanus. *Curry: An Integrated Functional Logic Language (Vers. 0.8.3)*. 2012. URL: `http://www.curry-language.org`.

[HHR07] Edith Hemaspaandra, Lane A. Hemaspaandra, and Jörg Rothe. "Anyone but Him: The Complexity of Precluding an Alternative". In: *Artificial Intelligence* 171.5-6 (2007), pp. 255–285.

[HK73] John E. Hopcroft and Richard M. Karp. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM Journal on Computing* 2.4 (1973), pp. 225–231.

[Hoa83] Charles Anthony Richard Hoare. "An Axiomatic Basis for Computer Programming (Re-print)". In: *Communications of the ACM* 26.1 (1983), pp. 53–56.

[HOW06] Bernd Heidergott, Geert Jan Olsder, and Jacob van der Woude. *Max Plus at Work*. Princeton University Press, 2006.

[HP06] Ralf Hinze and Ross Paterson. "Finger Trees: A Simple General-Purpose Data Structure". In: *Journal of Functional Programming* 16.2 (2006), pp. 197–217.

[Hug86] R. John M. Hughes. "A Novel Representation of Lists and Its Application to the Function 'Reverse'". In: *Information Processing Letters* 22.3 (1986), pp. 141–144.

[Hut99] Graham Hutton. "A Tutorial on the Universality and Expressiveness of Fold". In: *Journal of Functional Programming* 9.4 (1999), pp. 355–372.

[Joh98] Thomas Johnsson. "Efficient Graph Algorithms Using Lazy Monolithic Arrays". In: *Journal of Functional Programming* 8.4 (1998), pp. 323–333.

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a Symposium on the Complexity of Computer Computations, 1972, New York.* Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103.

[Kaw06] Yasuo Kawahara. "On the Cardinality of Relations". In: *Proceedings of the 9th International Conference on Relational and Algebraic Methods in Computer Science, RAMiCS 2006, Manchester, UK, 2006.* Ed. by Renate A. Schmidt. Vol. 4136. Lecture Notes in Computer Science. Springer, 2006, pp. 251–265.

[Kis12] Oley Kiselyov. *The `random-shuffle` Package*. Version 0.0.4. Detailed information available at `http://okmij.org/ftp/Haskell/perfect-shuffle.txt`. June 2012. URL: `http://hackage.haskell.org/package/random-shuffle`.

[KL95] David J. King and John Launchbury. "Structuring Depth-First Search Algorithms in Haskell". In: *POPL*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 344–354.

[Kle67] Morton Klein. "A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems". In: *Management Science* 14.3 (1967), pp. 205–220.

[Kme] Edward Kmett. *The `graphs` package*. Version 0.6.0.1. URL: `http://hackage.haskell.org/package/graphs`.

[Koz90] Dexter Kozen. "On Kleene Algebras and Closed Semirings". In: *Proceedings of Mathematical Foundations of Computer Science 1990, MFCS'90*. Ed. by Branislav Rovan. Vol. 452. Lecture Notes in Computer Science. Springer, 1990, pp. 26–47.

[Koz94] Dexter Kozen. "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". In: *Information and Computation* 110.2 (1994), pp. 366–390.

[KS96] Dexter Kozen and Frederick Smith. "Kleene Algebra with Tests: Completeness and Decidability". In: *10th International Workshop on Computer Science Logic, CSL '96, Utrecht, The Netherlands, 1996, Selected Papers*. Ed. by Dirk van Dalen and Marc Bezem. Vol. 1258. Lecture Notes in Computer Science. Springer, 1996, pp. 244–259.

[KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson New International Edition, 2006.

[Kuh55] Harold W. Kuhn. "The Hungarian Method for the Assignment Problem". In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97.

[KW91] Yugo Kashiwagi and David S. Wise. *Graph Algorithms in a Lazy Functional Programming Language*. Technical report (Indiana University, Bloomington. Computer Science Department). Computer Science Department, Indiana University, 1991.

[LEc88] Pierre L'Ecuyer. "Efficient and Portable Combined Random Number Generators". In: *Communications of the ACM* 31.6 (1988), pp. 742–749.

[Leh77] Daniel J. Lehmann. "Algebraic Structures for Transitive Closure". In: *Theoretical Computer Science* 4.1 (1977), pp. 59–76.

[Les15] Roman Leshchinsky. *The `vector` Package*. Version 0.10.12.3. Mar. 2015. URL: `http://hackage.haskell.org/package/vector`.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters". In: *POPL*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 333–343.

[Lip11] Miran Lipovača. *Learn You A Haskell for Great Good!* Online version available at `http://learnyouahaskell.com`. No Starch Press, Inc., 2011.

[Mar09] Simon Marlow. *Haskell 2010 Language Report*. 2009. URL: `https://www.haskell.org/onlinereport/haskell2010`.

[McB02] Conor McBride. "Faking It: Simulating Dependent Types in Haskell". In: *Journal of Functional Programming* 12.4&5 (2002), pp. 375–392.

[Mit13] Neil Mitchell. *The `hoogle` Package*. Version 4.2.26. The search engine is available at `https://www.haskell.org/hoogle`. Dec. 2013.

[MP08] Conor McBride and Ross Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (2008), pp. 1–13.

[OCo11] Russell O'Connor. 2011. URL: `http://r6.ca/blog/20110808T035622Z.html`.

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell: Code You Can Believe In*. Available online at `http://book.realworldhaskell.org`. O'Reilly, 2008.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer (2. ed.)* Cambridge University Press, 1996.

[Pet13] Alberto Pettorossi. *Techniques for Searching, Parsing, and Matching*. Aracne, 2013.

[RBSG11] Agnieszka Rusinowska, Rudolf Berghammer, Harrie C. M. de Swart, and Michel Grabisch. "Social Networks: Prestige, Centrality, and Influence - (Invited Paper)". In: *Proceedings of the 12th International Conference on Relational and Algebraic Methods in Computer Science, RAMiCS 2011, Rotterdam, The Netherlands, 2011.* Ed. by Harrie C. M. de Swart. Vol. 6663. Lecture Notes in Computer Science. Springer, 2011, pp. 22–39.

[Ril69] James E. Riley. "An Application of Graph Theory to Social Psychology". In: *The Many Faces of Graph Theory*. Vol. 110. Lecture Notes in Mathematics. 1969, pp. 275–280.

[RL99] Fethi Rabhi and Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999.

[Rui15] Alberto Ruiz. *The `hmatrix` Package*. Version 0.16.1.5. Mar. 2015. URL: `http://hackage.haskell.org/package/hmatrix`.

[San95] David Sands. "A Naïve Time Analysis and its Theory of Cost Equivalence". In: *Journal of Logic and Computation* 5.4 (1995), pp. 495–541.

[Sat75] Mark Allen Satterthwaite. "Strategy-proofness and Arrow's Conditions: Existence and Correspondence Theorems for Voting Procedures and Social Welfare Functions". In: *Journal of Economic Theory* 10.2 (1975), pp. 187–217.

[SS93] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs*. Springer, 1993.

[Ste15] Dominic Steinitz. *The seeds generated by split are not independent*. 2015. URL: `https://github.com/haskell/random/issues/25`.

[Swa11] Harrie C. M. de Swart, ed. *Relational and algebraic methods in computer science - 12th international conference, RAMICS 2011, rotterdam, the netherlands, may 30 - june 3, 2011. proceedings*. Vol. 6663. Lecture Notes in Computer Science. Springer, 2011.

[Wad89] Philip Wadler. "Theorems for Free!" In: *FPCA*. 1989, pp. 347–359.

[WTM98] Philip Wadler, Walid Taha, and David MacQueen. *How to Add Laziness to a Strict Language Without Being Odd*. Tech. rep. 1998.

Appendix A

Proofs

A.1 Proofs from Chapter 3

In this section we provide a construction of Boolean algebras within a given idempotent semiring. For a uniform approach we require some additional results from the field of lattice theory. We refer to the textbook by Davey and Priestley [DP02] for an introduction into this topic.

One important tool in this section are intervals, which are defined in the same way as they are typically defined on the real numbers.

A.1.1 Definition (Interval)**.**
Let $(O, \leqslant)$ be an ordered set. For all $a, b \in O$ we define

$$[a, b] := \{ \, x \in O \mid a \leqslant x \leqslant b \, \} \ .$$

Whenever there is a risk of ambiguity of the set or the order, we add an index for clarity. For instance, for any $P \subseteq O$ such that $a, b \in P$ we have

$$[a, b]_P = \{ \, x \in P \mid a \leqslant x \leqslant b \, \} \ .$$

Note that if $a \leqslant b$ holds, then we have $[a, b] \neq \emptyset$. ⫽

The following result deals with interval subsets of distributive complementary lattices. Its statements are well-known and we include them only for the sake of completeness (in fact, this is an exercise in the textbook of Davey and Priestley [DP02]).

A.1.2 Proposition (Complementary intervals)**.**
Let $\mathcal{L} = (L, \sqcup, \sqcap, \bot, \top)$ be a Boolean algebra. Let $a, b \in L$ such that $a \sqsubseteq b$ and set $I := [a, b]$. Then we get the following results:

(1) The function

$$f : L \to L \ , \quad x \mapsto (x \sqcup a) \sqcap b$$

is a lattice homomorphism and we have that $f(L) = I$. In particular, the structure $(I, \sqcup|_{I \times I}, \sqcap|_{I \times I})$ is a distributive lattice.

(2) Let $\bar{\ } : L \to L$ be the complement function (cf. Section 2.2 for details) in $\mathcal{L}$. Then

$$\mathcal{I} := (I, \sqcup|_{I \times I}, \sqcap|_{I \times I}, a, b)$$

is complementary and its complement function is $f \circ \bar{\ }$. In particular, if $a \sqsubset b$, then $\mathcal{I}$ is a Boolean algebra.

Proof. Statement (1). Let $x, y \in L$. Then we find

$$\begin{aligned} & f(x \sqcup y) \\ = \; & \wr \text{ Definition of } f \;\S \\ & ((x \sqcup y) \sqcup a) \sqcap b \\ = \; & \wr \text{ associativity, commutativity and idempotence of } \sqcup \;\S \\ & ((x \sqcup a) \sqcup (y \sqcup a)) \sqcap b \\ = \; & \wr \text{ distributivity of } L \;\S \\ & ((x \sqcup a) \sqcap b) \sqcup ((y \sqcup a) \sqcap b) \\ = \; & \wr \text{ Definition of } f \;\S \\ & f(x) \sqcup f(y) \;. \end{aligned}$$

In a similar fashion we get

$$f(x \sqcap y) = f(x) \sqcap f(y) \;.$$

Thus f is a lattice homomorphism. In addition we have

$$\begin{aligned} & a \\ \sqsubseteq \; & \wr \text{ since } a \sqsubseteq b \text{ we have } a = a \sqcap b \sqsubseteq (x \sqcup a) \sqcap b \;\S \\ & (x \sqcup a) \sqcap b \\ = \; & \wr \text{ Definition of } f \;\S \\ & f(x) \\ = \; & \wr \text{ Definition of } f \;\S \\ & (x \sqcup a) \sqcap b \\ \sqsubseteq \; & \wr \text{ monotonicity of } \sqcap \text{ in both components } \S \\ & b \;. \end{aligned}$$

Thus $f(x) \in I$, which shows $f(L) \subseteq I$. Now let $i \in I$. Then we have $a \sqsubseteq i \sqsubseteq b$ and thus

$$\begin{aligned} & f(i) \\ = \; & \wr \text{ definition of } f \;\S \\ & (i \sqcup a) \sqcap b \\ = \; & \wr\; a \sqsubseteq i, \text{ thus } i \sqcup a = i \;\S \\ & i \sqcap b \\ = \; & \wr\; i \sqsubseteq b, \text{ thus } i \sqcap b = i \;\S \\ & i \;, \end{aligned}$$

which shows $I \subseteq f(L)$. Since homomorphic images of distributive lattices are again distributive lattices, we conclude the proof. //

In the following lemma we deal with Boolean algebras within a given semiring. The key idea behind this lemma is that we can take a semiring $(S, +, \cdot, 0, 1)$ and add

some operations which are reminiscent of those of relation algebra. In fact, in case of a relation algebra, we have $+ = \cup$, $\cdot$ is the relational composition, $0 = \mathsf{O}$ and $1 = \mathsf{I}$. However, there are also operations $\cap$ and $^{-}$, as well as an additional constant, namely L, which yield a Boolean algebra within a semiring. In a relation algebra it is simple to see that the set of partial identities is a Boolean algebra. We wish to use a similar argument in a more general case. To this end we assume the existence of suitable operations and constants, such that we can duplicate the relation algebraic proof in a more general setting.

A.1.3 Lemma (Partial identities of Boolean algebras).
Let $(S,+,\cdot,0,1)$ be an idempotent semiring. Let $B \subseteq S$, $\top \in B$, $^{-} : B \to B$ and $\sqcap : B \times B \to B$ such that the following conditions are satisfied:

(a) $(B,+,\sqcap,0,\top)$ *is a Boolean algebra and* $^{-}$ *is its complement function.*

(b) $1 \in B$.

Then the following hold:

(1) $\mathsf{PI}_S(B) = [0,1]_B$ *and*

$$(\mathsf{PI}_S(B), +|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}, \sqcap|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}, 0, 1)$$

is a Boolean algebra. Additionally, we have that

$$^{-\mathsf{PI}_S(B)} = (- \sqcap 1) \circ {}^{-} .$$

(2) The function $\cdot|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}$ *is idempotent if and only if* $\cdot|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)} = \sqcap|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}$.

Proof. Statement (1). By prerequisite (a) we have that $0 \in B$ and by prerequisite (b) we get that $1 \in B$. Since 0 is the least element of S, $0 \leqslant_S b$ holds for all $b \in B$. Thus

$$[0,1]_B = \{\, b \in B \mid 0 \leqslant_S b \leqslant_S 1 \,\} = \{\, b \in B \mid b \leqslant_S 1 \,\} = \mathsf{PI}_S(B) .$$

Since $0 \leqslant_S 1$, Proposition A.1.2.(2) yields that

$$\mathcal{P} := (\mathsf{PI}_S(B), +|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}, \sqcap|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}, 0, 1)$$

is a Boolean algebra. Now let $f : \mathsf{PI}_S(B) \to \mathsf{PI}_S(B)$, $b \mapsto b \sqcap 1$. Then we have for all $b \in \mathsf{PI}_S(B)$

$$f(b) = b \sqcap 1 = (b + 0) \sqcap 1 .$$

Again by Proposition A.1.2.(2) we have that the complement function of $\mathcal{P}$ is the function $f \circ {}^{-}$. Since $f = (- \sqcap 1)$, this concludes the proof of Statement (1).
Statement (2). Suppose that $\cdot|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)} = \sqcap|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}$ holds. Since $\sqcap$ is an operation in a Boolean algebra (or a lattice, more generally) it is idempotent, which immediately yields the idempotence of $\cdot|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}$.
Now suppose that $\cdot|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}$ is idempotent. We show that for all $a, b \in \mathsf{PI}_S(B)$ the element $a \cdot b$ is the greatest lower bound of $\{\, a, b \,\}$ with respect to $\leqslant_S$. Let $a, b \in \mathsf{PI}_S(B)$. Then we have $a \cdot b \leqslant_S a \cdot 1 = a$ and similarly $a \cdot b \leqslant_S 1 \cdot b = b$. Thus $a \cdot b$ is a lower

bound of $\{ a, b \}$. Now let $l \in B$ be another lower bound of $\{ a, b \}$. Then we have $l \leqslant_S a$ and $l \leqslant_S b$. In particular we have $l \leqslant_S a \leqslant_S 1$ and thus $l \in \mathsf{PI}_S(B)$. We calculate

$$\begin{aligned} & l \\ = \quad & \wr\ \cdot|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)} \text{ is idempotent } \int \\ & l \cdot l \\ \leqslant_S \quad & \wr \text{ monotonicity of } \cdot \text{ in both components } \int \\ & a \cdot b\ . \end{aligned}$$

Thus $a \cdot b$ is the greatest lower bound of $\{ a, b \}$. Boolean algebra rules also yield that the greatest lower bound of $\{ a, b \}$ is $a \sqcap b$. This yields $a \cdot b = a \sqcap b$ and thus $\cdot|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)} = \sqcap|_{\mathsf{PI}_S(B)\times\mathsf{PI}_S(B)}$. ⫽

Now that we are equipped with the above lemma, we can provide proofs for all of the statements in Example 3.3.2.

A.1.4 Proof (Proof of Example 3.3.2)**.**

Proof. Statement 3.3.2.(1). We omit the proof of this statement, because it is a simple, but lengthy exercise in renaming 0 to F and 1 to T.
Statement 3.3.2.(2). Set $(S, +, \cdot, 0, 1) := (\mathcal{R}(T, T), \cup, \cdot, \mathsf{O}, \mathsf{I})$, $B := \mathcal{R}(T, T)$, $\sqcap := \cap$ and $\top = \mathsf{L}$. Then $B \subseteq S$ and $(B, \cup, \sqcap, \mathsf{O}, \top)$ is a Boolean algebra that satisfies $\mathsf{I} \in B$. Clearly, $\leqslant_S = \subseteq$ which yields $P = \mathsf{PI}_S(B)$ and thus by Lemma A.1.3.(1) we find that

$$(P, +|_{P\times P}, \sqcap|_{P\times P}, 0, 1)$$

is a Boolean algebra. In [Ber08] it is shown that $r \cdot r = r$ holds for all $r \in P$. Thus $\cdot|_{P\times P}$ is idempotent and Lemma A.1.3.(2) yields $\cdot|_{P\times P} = \sqcap|_{P\times P}$, which with the above result shows that

$$(P, +|_{P\times P}, \cdot|_{P\times P}, 0, 1)$$

is a Boolean algebra.
Statement 3.3.2.(3). Let $B' := B^{n\times n}$, $\sqcap$ be the componentwise multiplication of matrices and

$$\top := \sum_{i,j\in\mathbb{N}_{<n}} \mathsf{e}(i, j)\ .$$

It is a known fact that $(B', \boxplus|_{B'\times B'}, \sqcap|_{B'\times B'}, \mathbb{0}_n, \top)$ is a Boolean algebra and since $0, 1 \in B$, we have $\mathbb{1}_n \in B'$. By definition we get that $P = \mathsf{PI}_{K^{n\times n}}(B')$. By Lemma A.1.3.(1) we find that

$$(P, +|_{P\times P}, \sqcap|_{P\times P}, \mathbb{0}_n, \mathbb{1}_n)$$

is a Boolean algebra. We now show that $\sqcap|_{P\times P} = \boxtimes|_{P\times P}$. Let $p \in P$. Then p has non-zero entries at most along its main diagonal. Additionally, we have

$$p^2 \leqslant_n \mathbb{1}_n \boxtimes p = p \leqslant_n \mathbb{1}_n$$

and thus p^2 also has non-zero entries at most along its main diagonal. Now let

$i \in \mathbb{N}_{<n}$. Then we find

$$\begin{aligned}
& (p \boxtimes p)_{i,i} \\
= \; & \wr \text{ definition of matrix multiplication } \int \\
& \sum_{j=1}^{n} p_{i,j} \cdot p_{j,i} \\
= \; & \wr \text{ non-zero entries at most along main diagonal } \int \\
& p_{i,i} \cdot p_{i,i} \\
= \; & \wr \; (B, +|_{B \times B}, \cdot|_{B \times B}, 0, 1) \text{ is a Boolean algebra, thus } \cdot|_{B \times B} \text{ is idempotent } \int \\
& p_{i,i} \; .
\end{aligned}$$

Hence we have $p \boxtimes p = p$ and thus $\cdot|_{P \times P}$ is idempotent. By Lemma A.1.3.(2) we have that $\boxtimes|_{P \times P} = \sqcap|_{P \times P}$ and thus

$$(P, \boxplus|_{P \times P}, \boxtimes|_{P \times P}, \mathbb{0}_n, \mathbb{1}_n)$$

is a Boolean algebra.
Statement 3.3.2.(4). By Statement 3.3.2.(1) we know that with $B := \{ 0, 1 \}$ the structure

$$(B, +|_{B \times B}, \cdot|_{B \times B}, 0, 1)$$

is a Boolean algebra. We now show that $P = \mathsf{PI}_{K^{n \times n}}(B^{n \times n})$.
"$\subseteq$": Let $a \in P$. Then there is an $S \in \mathcal{P}(\mathbb{N}_{<n})$ such that $a = \sum_{i \in S} \mathsf{e}(i,i)$. For all $i \in S$ we have that $\mathsf{e}(i,i) \in B^{n \times n}$ and since $B^{n \times n}$ is closed under addition, we have $a \in B^{n \times n}$. Additionally, for every $i \in S$ we have $\mathsf{e}(i,i) \leqslant_n \mathbb{1}_n$. Thus $\mathbb{1}_n$ is an upper bound of $\{ \mathsf{e}(i,i) \mid i \in S \}$. The least upper bound of this set is $\sum_{i \in S} \mathsf{e}(i,i)$ and thus we get

$$a = \sum_{i \in S} \mathsf{e}(i,i) \leqslant \mathbb{1}_n \; ,$$

which together with $a \in B^{n \times n}$ yields $a \in \mathsf{PI}_{K^{n \times n}}(B^{n \times n})$.
"$\supseteq$": Let $a \in \mathsf{PI}_{K^{n \times n}}(B^{n \times n})$ and set $S_a := \{ i \in \mathbb{N}_{<n} \mid a_{i,i} = 1 \}$. Then for all $r, s \in \mathbb{N}_{<n}$ we have

$$\begin{aligned}
& \left(\sum_{i \in S_a} \mathsf{e}(i,i) \right)_{r,s} \\
= \; & \wr \text{ additivity of the index function } \int \\
& \sum_{i \in S_a} \mathsf{e}(i,i)_{r,s} \\
= \; & \wr \text{ property of } \mathsf{e}(i,i) \text{ and restriction to } S_a \int \\
& \begin{cases} 1 & : r = s \wedge r \in S_a \\ 0 & : \text{otherwise} \end{cases} \\
= \; & \wr \text{ definition of } S_a \text{ and } a \leqslant_n \mathbb{1}_n \int \\
& a_{r,s} \; .
\end{aligned}$$

Thus $a = \sum_{i \in S_a} \mathsf{e}(i,i)$ and since $B^{n\times n} \subseteq K^{n\times n}$, we obtain $a \in P$. Thus $P = \mathsf{PI}_{K^{n\times n}}(B^{n\times n})$ and Statement 3.3.2.(3) yields that $\mathsf{PI}_{K^{n\times n}}(B^{n\times n})$ is a Boolean algebra with the restricted operations of the Kleene algebra $K^{n\times n}$. ⫽

A.2 Proofs from Chapter 5

In this section we provide proofs that we omitted in Section 5.2 and a proof for the missing direction of Equation (5.5.3.ii).

A.2.1 Proof (Proof of Proposition 5.2.1)**.**

Proof. Statement 5.2.1.(1). Let $i, j \in \mathbb{N}_{<|p|}$ such that $i \leqslant j$ and let

$$q := (p_k)_{k=i}^{j} .$$

Then we get

$$|q| = j - i + 1 \leqslant |p| - 1 - i + 1 \leqslant |p| .$$

Let $a \in \mathbb{N}_{<|q|-1}$. We reason as follows:

$$\begin{aligned}
& \textbf{true} \\
\iff \quad & \wr \text{ length of subwalks and } p \text{ is } A\text{-}B\text{-alternating } \wr \\
& (q_a, q_{a+1}) = (p_{a+i}, p_{a+i+1}) \in \begin{cases} A & : a + i \text{ even} \\ B & : \text{otherwise} , \end{cases} \\
\iff \quad & \wr \text{ number theory } \wr \\
& (q_a, q_{a+1}) \in \begin{cases} A & : (i \text{ even} \wedge a \text{ even}) \vee (i \text{ odd} \wedge a \text{ odd}) \\ B & : (i \text{ even} \wedge a \text{ odd}) \vee (i \text{ odd} \wedge a \text{ even}) \end{cases} \\
\iff \quad & \wr \text{ unfolding the cases } \wr \\
& (q_a, q_{a+1}) \in \begin{cases} A & : i \text{ even} \wedge a \text{ even} \\ B & : i \text{ even} \wedge a \text{ odd} \\ B & : i \text{ odd} \wedge a \text{ even} \\ A & : i \text{ odd} \wedge a \text{ odd} . \end{cases}
\end{aligned}$$

Thus if i is even, then q is A-B-alternating and B-A-alternating otherwise.

Statement 5.2.1.(2). Assume the prerequisite:

$$p = () \vee q = () \vee \left(p \neq () \neq q \wedge |p| \text{ even} \wedge \left(p_{|p|-1}, q_0\right) \in B\right) .$$

Case 1: $p = ()$ or $q = ()$.
Then $p +\!\!+ q \in \{ p, q \}$ and since both p and q are A-B-alternating walks, so is $p +\!\!+ q$.

Case 2: $p \neq ()$, $q \neq ()$, $|p|$ even and $(p_{|p|-1}, q_0) \in B$.

Let $i \in \mathbb{N}_{<|p \mathbin{+\!\!+} q|-1}$. Then we have

$$\Big((p \mathbin{+\!\!+} q)_i, (p \mathbin{+\!\!+} q)_{i+1}\Big) = \begin{cases} (p_i, p_{i+1}) & : i < |p| - 1 \\ \Big(p_{|p|-1}, q_0\Big) & : i = |p| - 1 \\ \Big(q_{i-|p|}, q_{i-|p|+1}\Big) & : i > |p| - 1\ . \end{cases} \tag{A.2.1.a}$$

Thus we obtain the following chain of equivalences:

$$\begin{aligned}
& \textbf{true} \\
\iff\ & \wr\ p, q \text{ are } A\text{-}B\text{-alternating, Equation (A.2.1.a) and } \Big(p_{|p|-1}, q_0\Big) \in B\ \wr \\
& \Big((p \mathbin{+\!\!+} q)_i, (p \mathbin{+\!\!+} q)_{i+1}\Big) \in \begin{cases} A & : i < |p| - 1 \wedge i \text{ even} \\ B & : i < |p| - 1 \wedge i \text{ odd} \\ B & : i = |p| - 1 \\ A & : i > |p| - 1 \wedge i - |p| \text{ even} \\ B & : i > |p| - 1 \wedge i - |p| \text{ odd} \end{cases} \\
\iff\ & \wr\ i - |p| \text{ even } \iff i \text{ even, since } |p| \text{ is even}\ \wr \\
& \Big((p \mathbin{+\!\!+} q)_i, (p \mathbin{+\!\!+} q)_{i+1}\Big) \in \begin{cases} A & : i < |p| - 1 \wedge i \text{ even} \\ B & : i < |p| - 1 \wedge i \text{ odd} \\ B & : i = |p| - 1 \\ A & : i > |p| - 1 \wedge i \text{ even} \\ B & : i > |p| - 1 \wedge i \text{ odd} \end{cases} \\
\iff\ & \wr\ \text{combining cases, } |p| - 1 \text{ is odd}\ \wr \\
& \Big((p \mathbin{+\!\!+} q)_i, (p \mathbin{+\!\!+} q)_{i+1}\Big) \in \begin{cases} A & : i \text{ even} \\ B & : i \text{ odd}\ . \end{cases}
\end{aligned}$$

Thus $(p \mathbin{+\!\!+} q)$ is in fact A-B-alternating. //

A.2.2 Proof (Proof of Lemma 5.2.2)**.**

Proof. Statement 5.2.2.(1). By Definition 5.1.3 we know that $|p| \geqslant 2$. Assume that $|p|$ is odd. Then $|p| - 2$ is odd and thus $(p_{|p|-2}, p_{|p|-1}) \in M$ by Definition 5.1.3. But then $p_{|p|-1} \notin \mathsf{uncovered}(M)$ and by Definition 5.1.3 we have $p_{|p|-1} \in \mathsf{uncovered}(M)$, which is a contradiction. Thus $|p|$ is even.
Statement 5.2.2.(2). Let $i \in \mathbb{N}_{<|p|}$.
"$\Longleftarrow$": By Definition 5.1.3 we already know that $p_0, p_{|p|-1} \in \mathsf{uncovered}(M)$.
"$\Longrightarrow$": We show this by contraposition. Suppose that $i \notin \{\,0, |\,p| - 1\,\}$. If i is odd, then $(p_i, p_{i+1}) \in M$ by Definition 5.1.3 and thus by the symmetry of M also $(p_{i+1}, p_i) \in M$. Hence $p_i \notin \mathsf{uncovered}(M)$. If i is even, then $i - 1 \geqslant 0$ and $i - 1$ is odd. Thus $(p_{i-1}, p_i) \in M$ and again this yields that $p_i \notin \mathsf{uncovered}(M)$. //

A.2.3 Proof (Proof of Lemma 5.2.3)**.**

Proof. Since p is not a path, there are $s, t \in \mathbb{N}_{<|p|}$ such that $s \neq t$ and $p_s \neq p_t$. We may assume without loss of generality that $s < t$. The sequence $c := (p_k)_{k=s}^{t}$ is a cycle in G. By the theorem of Kőnig there are no odd cycles in G, because G is bipartite and thus $|c| = t - s + 1$ is odd, which yields

$$s, t \text{ even} \vee s, t \text{ odd}. \tag{A.2.3.a}$$

We claim that the following holds:

$$s > 0 \wedge t < |p| - 1 . \tag{A.2.3.b}$$

Proof. We show this by contradiction. Assume that $s = 0$ or $t = |p| - 1$. By Lemma 5.2.2.(1) we know that $|p|$ is even and thus $|p| - 1$ is odd.
Case 1: $s = 0$.
Then $t \neq |p| - 1$, because t is even by Equation (A.2.3.a). Thus we have

$$p_s = p_0 \in \mathsf{uncovered}(M) .$$

Since $t \notin \{0, |p| - 1\}$, we get that $p_t \notin \mathsf{uncovered}(M)$ by Lemma 5.2.2.(2), which is a contradiction to the fact that $p_s = p_t$.
Case 2: $t = |p| - 1$.
Then t is odd and thus $s > 0$ by Equation (A.2.3.a). Since $s < t$, we have that $s \notin \{0, |p| - 1\}$ and thus $p_t \in \mathsf{uncovered}(M)$, but $p_s \notin \mathsf{uncovered}(M)$ by Lemma 5.2.2.(2), which, again, contradicts the fact that $p_s = p_t$. ⫽

We now show that the abbreviated sequence $(p_k)_{k=0}^{s} +\!\!+ (p_k)_{k=t+1}^{|p|-1}$ is an $(E \setminus M)$-M-alternating path. We distinguish two cases, namely whether t is odd or even.
Case 1: t is odd.
Then $q := (p_k)_{k=0}^{s}$ and $r := (p_k)_{k=t+1}^{|p|-1}$ are both $(E \setminus M)$-M-alternating by Proposition 5.2.1.(1), because $t + 1$ is even. The paths q, r are both non-empty by Equation (A.2.3.b). We have that $q_{|q|-1} = p_s = p_t$ and $r_0 = p_{t+1}$. Since t is odd and p is $(E \setminus M)$-M-alternating, we have that $(q_{|q|-1}, r_0) = (p_t, p_{t+1}) \in M$. Finally, we have $|q| = s + 1$, which is even since s, t are odd by Equation (A.2.3.a). Thus Proposition 5.2.1.(2) yields that $q +\!\!+ r$ is $(E \setminus M)$-M-alternating.
Case 2: t is even.
Then $q := (p_k)_{k=0}^{s-1}$ and $r := (p_k)_{k=t}^{|p|-1}$ are both $(E \setminus M)$-M-alternating by Proposition 5.2.1.(1). By Equation (A.2.3.a) we know that s is even and thus $(p_{s-1}, p_s) \in M$, since p is $(E \setminus M)$-M-alternating. We also find that $q_{|q|-1} = p_{s-1}$ and $r_0 = p_t = p_s$ and thus $(q_{|q|-1}, r_0) = (p_{s-1}, p_s) \in M$. We have that $|q| = s$, which is even, since both s, t are even by Equation (A.2.3.a). Proposition 5.2.1.(2) yields that $q +\!\!+ r$ is $(E \setminus M)$-M-alternating and since

$$q +\!\!+ r = (p_k)_{k=0}^{s-1} +\!\!+ (p_k)_{k=t}^{|p|-1} = (p_k)_{k=0}^{s} +\!\!+ (p_k)_{k=t+1}^{|p|-1} ,$$

the sequence $(p_k)_{k=0}^{s} +\!\!+ (p_k)_{k=t+1}^{|p|-1}$ is an $(E \setminus M)$-M-alternating walk as well. In both

cases, the abbreviated path is $(E \setminus M)$-M-alternating and its first and last vertices are different and both are not contained in $\mathsf{uncovered}(M)$. Thus it is an M-augmenting candidate. Note that the resulting walk is strictly shorter than the original one. Since any augmenting walk contains only a finite number of cycles, we can remove all these cycles and obtain a cycle-free augmenting walk, which is a path. ⫽

A.2.4 Proof (Proof of Equation (5.5.3.ii), direction "$\Longrightarrow$".)**.**

Proof. Suppose that G is bipartite. A straightforward induction shows that for every $i \in I$ we have

$$s_i \subseteq p_v \cdot E \ ,$$

since $(s_i)_{i \in I}$ is the sequence of reachability steps from p_v. Set $I_{\text{even}} := \{\, i \in I \mid i \text{ even} \,\}$. Then we get

$$\begin{aligned}
& a \\
= \;& \wr \text{ definition of } a \ \wr \\
& \bigcup_{\text{even}} s_i \\
\subseteq \;& \wr\ s_i \subseteq p_v \cdot E \text{ and monotonicity of the union } \wr \\
& \bigcup_{\text{even}} p_v \cdot E^i \\
= \;& \wr \text{ composition distributes over unions } \wr \\
& p_v \cdot \bigcup_{\text{even}} E^i \\
\subseteq \;& \wr \text{ adding additional powers and power rule } \wr \\
& p_v \cdot \bigcup_{k \in \mathbb{N}} (E \cdot E)^k \\
= \;& \wr \text{ for relations we have } r^\star = \bigcup \{\, r^n \mid n \in \mathbb{N} \,\} \text{ [SS93] } \wr \\
& p_v \cdot (E \cdot E)^\star \ .
\end{aligned}$$

Similarly, we get $b \subseteq p_v \cdot (E \cdot E)^\star \cdot E$. Observe that we have the following equivalence:

$$\begin{aligned}
& \textbf{true} \\
\Longleftrightarrow \;& \wr\ G \text{ is bipartite and Equation (5.5.2.ii) } \wr \\
& (E \cdot E)^\star \cap E = \mathsf{O} \\
\Longleftrightarrow \;& \wr \text{ Boolean algebra } \wr \\
& (E \cdot E)^\star \subseteq \overline{E} \\
\Longleftrightarrow \;& \wr\ a^\star \cdot a^\star = a^\star \text{ holds in every Kleene algebra } \wr \\
& (E \cdot E)^\star \cdot (E \cdot E)^\star \subseteq \overline{E} \\
\Longleftrightarrow \;& \wr\ {}^\top \circ {}^\star = {}^\star \circ {}^\top \text{ [SS93, Prop. 3.2.6] and } E \cdot E \text{ is symmetric } \wr \\
& ((E \cdot E)^\star)^\top \cdot (E \cdot E)^\star \subseteq \overline{E}
\end{aligned}$$

$$\begin{aligned}
&\iff \quad \wr \text{ Schröder equivalence } \wr \\
&\quad (E \cdot E)^\star \cdot E \subseteq \overline{(E \cdot E)^\star} \\
&\iff \quad \wr \text{ Boolean algebra } \wr \\
&\quad (E \cdot E)^\star \cdot E \cap (E \cdot E)^\star = \mathsf{O} \,.
\end{aligned} \tag{A.2.4.a}$$

This yields the desired result:

$$\begin{aligned}
&(a \cdot E \cap a) \cup (b \cdot E \cap b) \\
\subseteq \;& \wr \text{ see above } \wr \\
&(p_v \cdot (E \cdot E)^\star \cdot E \cap p_v \cdot (E \cdot E)^\star) \cup (p_v \cdot (E \cdot E)^\star \cdot E \cdot E \cap p_v \cdot (E \cdot E)^\star \cdot E) \\
\subseteq \;& \wr\ a^\star \cdot a \leqslant 1 + a^\star \cdot a = a^\star \text{ in every Kleene algebra } \wr \\
&(p_v \cdot (E \cdot E)^\star \cdot E \cap p_v \cdot (E \cdot E)^\star) \cup (p_v \cdot (E \cdot E)^\star \cap p_v \cdot (E \cdot E)^\star \cdot E) \\
= \;& \wr \text{ idempotence of the union and commutativity of the intersection } \wr \\
&p_v \cdot (E \cdot E)^\star \cdot E \cap p_v \cdot (E \cdot E)^\star \\
= \;& \wr\ p_v \text{ is univalent, thus } p_v \cdot (r \cap s) = p_v \cdot r \cap p_v \cdot s \text{ [SS93, Prop. 4.2.2.ii)] } \wr \\
&p_v \cdot ((E \cdot E)^\star \cdot E \cap (E \cdot E)^\star) \\
= \;& \wr \text{ second factor is } \mathsf{O} \text{ by Equation (A.2.4.a) and } \mathsf{O} \text{ is annihilating } \wr \\
&\mathsf{O} \,.
\end{aligned}$$

//

A.3 Proofs from Chapter 7

A.3.1 Proof (Proof of Equation (7.2.iii) and Equation (7.2.iv)).

Proof. Equation (7.2.iii). Let $v, w \in V$
"$\Longrightarrow$": Suppose that $c(v, w) \neq 0$ and $(v, w) \in A(p)$. By extension we have $c(e) = 0$ for all $e \in (V \times V) \setminus E$, which yields $(v, w) \in E$ and thus $(v, w) \in A(p) \cap E$.
"$\Longleftarrow$": We reason as follows:

$$\begin{aligned}
&(v, w) \in A(p) \cap E \\
\iff \;& \wr \text{ definition of the intersection } \wr \\
&(v, w) \in A(p) \land (v, w) \in E \\
\iff \;& \wr \text{ since } \varphi \land \varphi \iff \varphi \wr \\
&(v, w) \in A(p) \land (v, w) \in A(p) \land (v, w) \in E \\
\Longrightarrow \;& \wr\ (v, w) \in A(p) \subseteq E_f \text{ and definition of } E_f \text{, Definition 7.1.3 } \wr \\
&(v, w) \in A(p) \land c_f(v, w) > 0 \land (v, w) \in E \\
\Longrightarrow \;& \wr \text{ definition of } c_f \text{ on } E \text{, Definition 7.1.3 } \wr \\
&(v, w) \in A(p) \land c(v, w) - f(v, w) > 0 \\
\Longrightarrow \;& \wr\ f \geqslant 0 \text{, since } f \text{ is a flow, thus } 0 < c(v, w) - f(v, w) \leqslant c(v, w) \wr \\
&(v, w) \in A(p) \land c(v, w) \neq 0 \,.
\end{aligned}$$

Equation (7.2.iii). Let $v, w \in V$
"$\Longrightarrow$": Suppose that $c(v,w) \neq 0$ and $(v,w) \in A(p)^\top$. We get $(v,w) \in E$ just as above and thus $(v,w) \in A(p)^\top \cap E$.
"$\Longleftarrow$": We argue as follows:

$$
\begin{aligned}
& (v,w) \in A(p)^\top \cap E \\
\Longleftrightarrow\ & \wr \text{ definition of the intersection } \wr \\
& (v,w) \in A(p)^\top \wedge (v,w) \in E \\
\Longleftrightarrow\ & \wr \text{ since } \varphi \wedge \varphi \iff \varphi \wr \\
& (v,w) \in A(p)^\top \wedge (v,w) \in A(p)^\top \wedge (v,w) \in E \\
\Longleftrightarrow\ & \wr \text{ definition of the transposition } \wr \\
& (v,w) \in A(p)^\top \wedge (w,v) \in A(p) \wedge (w,v) \in E^\top \\
\Longrightarrow\ & \wr \text{ since } A(p) \subseteq E_f \text{ and definition of } E_f\text{, Definition 7.1.3 } \wr \\
& (v,w) \in A(p)^\top \wedge c_f(w,v) > 0 \wedge (w,v) \in E^\top \\
\Longrightarrow\ & \wr \text{ definition of } c_f \text{ on } E^\top\text{, Definition 7.1.3 } \wr \\
& (v,w) \in A(p)^\top \wedge f(v,w) > 0 \\
\Longrightarrow\ & \wr\ f \leqslant c\text{, since } f \text{ is a flow } \wr \\
& (v,w) \in A(p)^\top \wedge c(v,w) > 0 \\
\Longrightarrow\ & \\
& (v,w) \in A(p)^\top \wedge c(v,w) \neq 0\ .
\end{aligned}
$$

Nomenclature

Haskell Functions

$(!!!)$	Query of matrix rows, page 49
$(!)$	Index query of semiring vectors, page 48
$(\cap_{\mathsf{l}})$	Left-biased intersection of vectors, page 84
$(\cup_{\mathsf{l}})$	Left-biased binary union, page 83
(Δ)	Symmetric difference function for matrices, page 121
$(+_{\mathsf{v}})$	Addition of idempotent semiring vectors, page 50
$(-_{\mathsf{v}})$	Subtraction of two vectors, page 179
$(\boxminus)$	Subtraction of matrices, page 168
$(\boxplus)$	The addition of two semiring matrices, page 127
$(\bullet'_{\mathsf{v}})$	Canonic scalar multiplication of semiring vectors, page 49
$(\bullet_{\mathsf{v}})$	Optimised scalar multiplication of semiring vectors, page 49
$(\circledast)$	Semiring multiplication, page 44
$(\oplus)$	Semiring addition, page 44
$(\odot)$	Vector-matrix multiplication in semirings (naïve version), page 76
$(\odot')$	Vector-matrix multiplication in semirings, page 76
$(\odot_{\mathbb{U}})$	A vector-matrix multiplication that extends the reachability forest, page 91
$(\odot_{+})$	A numerical vector-matrix multiplication that considers all vector values to be 1, page 192
$(\odot_{1,+})$	A vector-matrix multiplication that counts how many times a successor has been reached, page 176
$(\odot_{?})$	Vector-matrix multiplication that checks whether the vector has successors, page 78

$(\odot_{\approx})$ Vector-matrix multiplication that extends all walks, page 80

$(\odot_{\leftarrow})$ Vector-matrix multiplication that collects the immediate predecessors, page 155

$(\odot_{\sim,\Sigma})$ Vector-matrix multiplication that prolongs a path and updates sum of all values along that path, page 167

$(\odot_{\sim})$ Vector-matrix multiplication that extends a single (deterministically chosen) walk, page 79

$(\ogreaterthan)$ Vector-matrix multiplication that ignores the vector values, page 77

$(\ogreaterthan^{S})$ A variant of $(\ogreaterthan_{\rightarrow})$ for semiring matrices, page 77

$(\ogreaterthan_{\rightarrow})$ Discrete successors multiplication, page 77

$(\olessthan)$ Vector-matrix multiplication that ignores the matrix values, page 171

$(\sqcap_{\mathsf{r}})$ Left-forgetful intersection of matrices, page 168

$(\sqcap_{\mathsf{l}})$ Right-forgetful intersection of two matrices, page 127

$(\backslash)$ Relative difference on vectors, page 87

addEdge Inserts an edge into an *AuxGraph*, page 120

addEdgeSym Inserts an edge and its flipped counterpart into an *AuxGraph*, page 120

allSubsets An infinite list containing all subsets of all sets with a certain cardinality, page 180

allUnion A union of vectors that collects all information, page 80

Alternative Data type for alternatives, page 176

AppVoting Data type for approval voting, page 176

AppVotingNum A numerical alternative to *AppVoting*, page 192

Arc Type synonym for arcs, page 47

at Type class operation for abstract queries, page 98

atLeast Computes the least number of voters that need to be removed for an alternative to win, page 179

augmentFlow	Augments a flow according to Theorem 7.1.4, page 168
augmentingPath	Finds augmenting paths for matchings, page 118
augmentingPaths	A function that computes a list of pairwise disjoint augmenting paths, page 124
augmentMatching	A function that augments a matching according to Berge's lemma, page 122
augmentMatchingHK	A function that augments a given matching using the strategy of Hopcroft and Karp, page 124
AuxGraph	A graph that can be modified efficiently, page 119
Balanced	The balanced Kleene algebra, page 186
betweenness	Returns the betweenness centrality of all vertices, page 189
bigCupWith	A folded version of *cupWith*, page 96
bigUnionWith′	A variant of *bigUnionWith* that uses *IntMap*s, page 70
bigUnionWith	Folded *unionWith*, page 69
cap	Type class operation for right-forgetful intersections, page 97
capacity	*Network* accessor function, returns the capacity, page 166
capWithKey	Type class operation for heterogeneous intersections that may depend on keys, page 97
chop	A pruning operation that finds disjoint paths, page 93
Clustered	The clustered Kleene algebra, page 185
cmpFst	A comparison function for pairs with respect to their first components, page 95
componentwise	A function that applies a given operation to every connected component of a symmetric graph, page 132
contains	Test for being contained in the monadic set, page 92
cup	Type class operation for left-forgetful unions, page 96
cupWith	Type class operation for (heterogeneous) unions, page 96
disjointPaths	A function that finds a set of pairwise disjoint shortest paths, which is maximal with respect to inclusion, page 93

emptyAuxGraph	Returns an empty *AuxGraph* of a special size, page 120
emptyMat	Returns an empty graph of the same size as its argument, page 122
emptyOrElem	Checks whether the list is empty or the value is contained in the list, page 179
emptyVec	The empty vector, page 48
evens	A function that computes the even positions in a list (no implementation), page 130
fAugmentingPath	Finds an f-augmenting path if such a path exists, page 168
filterMatrix	Filters the value of a matrix, page 183
filterVec	A filter function for vectors, page 49
findBipartition	Returns a bipartition of a graph and *Nothing* if the graph is non bipartite, page 132
findBipartitionConnected	Finds a bipartition in a connected graph, page 130
findSets	Finds the sets that need to be tested for an alternative, page 179
fmapMaybeWithKey	A version of *fmap* that depends on keys and may yield no result, page 98
fmapWithKey	A variant of *fmap* for vectors that can use the keys, page 127
Forest	A forest is a list of trees, page 90
forestMult	A scalar multiplication that extends the given forest and uses it as a new label, page 91
fromVec	Returns the vertices of a vector, page 75
generateAndTestApproval	Generates all variants of voter removal and returns the first one that makes the given alternative a winning one, page 179
genericCapWithKey	An intersection of *KeyMaybeFunctor*s with *Lookup*s, page 98
Geodesic	The geodesic semiring, page 188
geodesicLabel	Label for embedding matrices in the geodesic semiring, page 189

geodesicLength	Accessor function for *Geodesic*, returns the length component, page 188
geodesicWeight	Accessor function for *Geodesic*, returns the weight component, page 188
hasSuccs	Checks whether a list of vertices has successors, page 78
hasSuccsMult	Test for being non-empty as a scalar multiplication, page 78
IdempotentSemiring	Type class for idempotent semirings, page 44
identityOf	Returns the identity matrix of the same size as its argument, page 127
include	Insertion operation into the monadic set, page 92
includeAll	A folded version of *include*, page 132
insertWith	Inserts a value at a key, defined in *Data.IntMap*
Intersectable	Type class for heterogeneous intersections, page 97
intersectionWithKey	Intersection of two vectors as a vector, page 72
intersectionWithKeyL	An implementation of the vector intersection, page 95
intersectMatWithKey	An intersetion operation for matrices, page 127
intMapToVec	Conversion function from *IntMap* to *Vec*, page 120
isBipartite′	A variant of *isBipartite* with less operations, page 128
isBipartite	Checks whether a graph is bipartite, page 127
isBipartiteFast	Checks whether a graph is bipartite efficiently, page 133
isClustered2	Kleene algebra version of *isClustered*, page 185
isClustered	Checks whether a graph is clustered, page 183
isDiagonalOne	Checks whether the diagonal of a matrix is one everywhere, page 185
isEmptyMat	Checks whether a matrix is empty, page 127
isEmptyVec	Predicate that checks whether a vector is empty, page 48
isOne	Checks whether a semiring element is *one*, page 44

isSimple	Checks whether the KAT closure can be computed with *simpleKleene*, page 45
isSymmetric	Checks whether a matrix is symmetric, page 126
isWeakBipartition	Checks whether all edges of a graph connect only vertices from the two given sets, page 128
isZero	Checks whether a semiring element is *zero*, page 44
KAT	Type class for Kleene algebras with tests, page 45
katPlus	Kleene closure in Kleene algebras with tests, page 46
katStar	Star closure in Kleene algebras with tests, page 46
KeyMaybeFunctor	Type class for keyed functors with a *Maybe*-valued *fmap*, page 98
KleeneAlgebra	Type class for Kleene algebras, page 44
kleeneClosureMatrix	The Kleene closure of a matrix, page 50
kleeneClosureMatrixWith	A version of *kleeneClosureMatrix* with an explicit order parameter, page 52
leftmostUnion	Left-biased union of a list of vectors, page 77
Lookup	Type class for abstract queries, page 98
Mat	Data type for matrices, page 47
matrix	Accessor function for matrices, returns the underlying vector of vectors, page 47
matrixAt	Returns the value at the given position in a matrix over a semiring, page 189
maximumFlow′	Computes the maximum flow and the residual capacity of that flow in a network, page 169
maximumFlow	Computes the maximum flow in a network, page 169
maximumMatching′	A function that computes a maximum matching and its relative complement in a bipartite graph, page 122
maximumMatching	A function that computes a maximum matching in a bipartite graph, page 122

maximumMatchingHK′	A function that computes a maximum matching and its complement in a bipartite graph using the technique of Hopcroft and Karp, page 125
maximumMatchingHK	A function that computes a maximum matching in a bipartite graph using the technique of Hopcroft and Karp, page 125
maximumMatchingSafe	A variant of *maximumMatching* that checks the graph for being symmetric and bipartite, page 125
maxIndices	Returns the indices that have maximum values in the vector, page 176
maxVec	Returns the maximum value in a vector, which is set to 0 if the vector is empty, page 176
maybeSome	Returns a key-value pair from a vector or *Nothing* if the vector is empty, page 119
mergeWith	A fully parametric merge function, page 94
minimumCut	A function that computes a minimum cut in a network, page 171
mkVec	Conversion of association lists to vectors, page 75
Neg	Constructor for the negative sign, page 183
negative	Returns a submatrix that contains only negative values, page 183
Network	Data type for networks, page 166
newMat	Computes step $k+1$ from step k in the Kleene iteration, page 51
Number	Data type for wrapped numbers, page 76
odds	A function that computes the odd positions in a list (no implementation), page 130
one	Neutral element with respect to multiplication in a semiring, page 44
opCons	A function that adds a new element to a list, where the element is the result of a binary function, page 95

orderedLookup	A lookup function on ordered lists, page 48
outMult	A scalar multiplication that extends the list of outgoing values at every position, page 82
Path	A data type for (shortest) paths; same as *Walk*, page 92
pathsToUndirectedGraph	A folded version of *pathToUndirectedGraph*, page 124
pathToGraphWith	Inserts a path into a matrix, page 169
pathToUndirectedGraph	Inserts all (symmetric) edges of a path into a given graph, page 120
PlainVec	A data type with efficient index access containing only () values, page 119
Pos	Constructor for the positive sign, page 183
positive	Returns a submatrix that contains only positive values, page 183
powerlistFrom	Returns the power set of a set starting with a given cardinality, page 180
preTranspose	A transposition variant with possibly missing empty rows, page 83
reachableWith	A function that computes the reachable vertices, page 88
reachForest	A function that computes the reachability forest between two vectors, page 91
removeMatZeroes	Removes zero values from a semiring matrix, page 127
removeZeroes	removes zeroes, page 49
restrictToMax	Returns a pair consisting of the vector restricted to those indices with a maximal value and said value, page 176
results	A function that computes the outcome of an election with all voters, page 176
rowIndices	The indices of the rows of a matrix, page 50
runNew	Creates a new set and computes its effect, page 92
Semiring	Type class for semirings, page 44

SetM	A monadic set interface, page 92
shortestWith	A function that computes the intersection of the source vector with the first non-empty reachability step if such a step exists and the empty vector otherwise, page 88
Sign	Data type for positive and negative signs, page 183
simpleKleene	KAT function that computes the Kleene closure, page 45
sink	*Network* accessor function, returns the sink, page 166
size	Returns the number of filled positions in a vector, page 179
sMultWith	Scalar multiplication generator, page 71
source	*Network* accessor function, returns the source, page 166
sparseTests	List of tests with sum 1, page 45
splitEvenOdd	Splits a list into its even and odd positions, page 130
starClosure	The star closure of matrices, page 127
starSymClosure	The composition *starClosure* ∘ *symClosure*, page 183
stepsWith	A function that computes the reachability steps, page 87
subsets	Returns the list of subsets of a certain size, page 180
subsetsOfSize	Returns all subsets with size k of a set with size n, page 180
symClosure	Symmetric closure of a matrix, page 183
symDifference	Symmetric difference function for vectors, page 121
τ	Recursive version of the mathematical function τ, page 50
thirdArg	A ternary function that ignores its first two arguments, page 95
times	Matrix multiplication over semirings, page 126
toBoolMat	Transforms a matrix into a Boolean matrix, page 128
toGraph	Conversion function from *AuxGraph* to *Mat* (), page 120
toVec	Conversion of vertex lists to unit-labelled vectors, page 75
toVecFrom	Conversion from vertex lists to vectors, page 75

toVecWith	Conversion of vertex lists to vectors with the same value, page 75
transposeNonSquare	Transposition of non-square matrices, page 84
transposeSquare	Transposition of square matrices, page 83
Tree	A data type for rose trees, page 90
tropAdd	Numerical addition of tropical values, page 188
Tropical	The tropical semiring, page 166
tropMult	Numerical multiplication of tropical values, page 188
tropToInteger	Maps tropical integers to integers, page 190
uncovered	Returns the vertices without successors of a graph, page 119
Unionable	Type class for heterogeneous unions, page 96
unionMatWith	A union operation for matrices, page 127
unionWith	An implementation of the vector union, page 95
unNumber	Accessor function for *Number*s, returns the wrapped number, page 76
unVec	Accessor function for vectors, returns the underlying association list, page 47
unVecArray	Selector function for *VecArray*, page 97
values	Returns the values of a vector as a list, page 176
Vec	Data type for vectors, page 47
VecArray	An array version of vectors, page 97
vecMatMult	Vector-matrix multiplication generator, page 72
Vertex	Vertex type in Haskell, page 75
vertexCoverApprox	Approximation function for vertex covers, page 155
vertices	Returns the vertices of a matrix as a list (same as *rowIndices*), page 132
verticesWith	Returns the vertices of a matrix as a vector, page 83

Voter	Data type for voters, page 176
voters	The vector of all voters, page 176
voteswithPlayers	A function that computes the outcome of an election with the given voters, page 176
Walk	Data type for walks, page 79
walkMult	A scalar multiplication that extends the given walk and uses the new walk as a label, page 79
walksMult	A scalar multiplication that extends all walks and uses the new list of walks as a label, page 80
weight	Accessor function for *Tropical*, returns the weight of a *Weight* element, page 166
winners	Computes the winners of an approval voting with a given set of voters, page 176
withAt	A lookup function with a default value, page 48
zero	Neutral element with respect to addition in a semiring, page 44

Mathematical Functions and Symbols

$(x_k)_{k=a}^{b}$	Restriction of a sequence, Section 2.2, page 9
1_n	Multiplicative unit in $S^{n \times n}$ (same as $\mathbb{1}_n$), Convention 2.3.8, page 15
$[a, b]$	Interval in an order, page 209
$+\!\!+$	Mathematical list concatenation, page 10
$\bullet$	Mathematical scalar multiplication of a matrix, page 21
Δ	Symmetric difference operation, page 112
$\mathbb{1}$	A singleton set $\mathbb{1} = \{ \square \}$, Section 2.2, page 9
$\square$	A fixed object, Section 2.2, page 9
$\leqslant_S$	The order of an idempotent semiring S, Definition 2.3.5, page 14
$\lVert - \rVert_c$	Sum of all capacities in an edge set, Definition 7.3.1, page 170

$\odot_{p,\mu}$	Mathematical version of the abstracted vector-matrix multiplication, page 102
$\sqcap_{\mathsf{r}}$	Left-forgetful intersection of two functions, page 163
$\sum_{p,e}$	A folded version of p with neutral element e, Definition 4.7.3, page 101
$\|f\|$	The flow value, Definition 7.1.2, page 158
$\|-\|$	Set cardinality, length of a sequence, Section 2.2, page 10
Ξ	Edges between a set and its complement, Definition 7.3.1, page 170
c_f	The residual capacity, Definition 7.1.3, page 159
E_f	The edges of the residual network, Definition 7.1.3, page 159
S^*	The set of all finite sequences over S, Section 2.2, page 10
$+_p$	Lifting of a binary operation to a binary vector operation, Definition 4.7.2.(1), page 101
$A(p)$	Directed edges along a path, Theorem 7.1.4, page 159
A_G	Adjacency matrix of G, page 12
asSet	Conversion from semiring vectors to sets, Definition 4.1.1, page 64
asVector	Conversion from sets to semiring vectors, Definition 4.1.1, page 63
A_{Z}	Shorthand for $A \cup \{ \mathsf{Z} \}$, page 100
bal	The balance of a vertex function, Definition 7.1.2, page 158
$\boxplus$, $\boxtimes$	Addition and multiplication of matrices in Kleene algebra notation, Theorem 3.2.5, page 15
$\bullet$	Scalar multiplication of a matrix, Lemma 3.4.1, page 39
$\|\cdot\|_{X,Y}$	Relational cardinality, Definition 6.2.2, page 138
cost	The cost of an edge function, page 201
cover(G)	The cardinality of a minimum vertex cover in a graph, Definition 6.1.1, page 136

$\mathsf{cut}(N)$	The capacity of a minimum cut in a network, Definition 7.3.1, page 170
$\mathsf{e}^n(i)$	Standard unit vectors, Definition 2.4.4, page 20
$\mathsf{e}^{n,m}(i,j)$	Standard unit matrices, Definition 2.4.4, page 20
$E(p)$	The symmetric closure of the set of edges along a path , page 112
$\mathsf{exA}(p, \mathsf{Z})$	Extension of p such that Z is a neutral element, Definition 4.7.1.(1), page 100
$\mathsf{exM}(\mu, \mathsf{Z}_A, \mathsf{Z}_B, \mathsf{Z}_C)$	Extension of μ, such that Z_A, Z_B and Z_C behave as zeroes, Definition 4.7.1.(2), page 100
$\mathsf{flow}(N)$	The value of a maximum flow in a network, Definition 7.1.2, page 158
$\mathsf{matching}(G)$	The cardinality of a maximum matching in a graph, Definition 5.1.1, page 109
$\mathsf{PI}_S(T)$	Partial identities, Definition 2.4.7, page 22
pr_j	Projection in the j-th component, Section 2.2, page 9
$\mathsf{sm}(\mu)$	Lifting of an indexed multiplication to an indexed scalar multiplication, Definition 4.7.2.(2), page 101
$^\star, ^+$	Star closure and Kleene closure, Definition 3.2.1, page 29
τ	A function used to compute the Kleene closure, page 35
$\mathrm{transpose}(g)$	Transposition of a binary function, page 163
$\mathsf{uncovered}$	Uncovered vertices of a relation, Definition 5.1.3, page 111
votes_S	The number of votes of an alternative, page 174
Z_t	A constant Z-function with domain $\mathbb{N}_{<t}$, page 100
$\mathbb{0}_n$, $\mathbb{1}_n$	Zero and identity matrix of (square) dimension n, Theorem 3.2.5, page 15

Index

adjacency matrix, 12
approval election, 171
Approval voting, 171

balance, 156
bipartition, 108

capacity, 156
    residual, 157
cut, 168
    capacity, 168
    minimum, 168
cycle, 16
    even, 16
    odd, 16

flow, 156
    r-flow, 199
    minimum cost, 200
    value, 156

graph, 10
    L-labelled, 10
    balanced, 179
    clustered, 179
    edge label, 10

interval, 213

Kleene algebra, 29
    balanced, 183
    clustered, 182
    geodesic, 186
    with tests, 34

lattice, 9
    Boolean algebra, 9
    complementary, 9
    complete, 9
    distributive, 9

length, 10

matching, 107
    u-maximum, 198
    augmenting candidate, 110
    augmenting path, 110
    maximum, 107
    number, 107

network, 156
    residual, 157

partial identity, 22
path, 16
    A-B-alternating, 109
    M-augmenting, 110
    f-augmenting, 157
projection, 9

relation
    abstract, 17
    algebra, 17
    cardinality, 136
    injective, 18
    point, 19
    surjective, 18
    total, 18
    type, 17
    univalent, 18
    vector, 19

semiring, 12
    idempotent, 13
    order, 14
    tropical, 13
sink, 156
source, 156
standard unit
    matrix, 20

vector, 20

type, 9

uncovered, 109

vertex cover, 134
  u-vertex-cover, 199
  minimum, 134
  minimum u-vertex-cover, 199
  number, 134

walk, 16
  from v to w, 16
  negative, 180
  positive, 180

Colophon

This document was typeset using the LaTeX typesetting system, the `book` document class, and the Palatino font. The main layout options were obtained using the `kcss` style file from the Kiel Computer Science Series from the Department of Computer Science of the Kiel University. The index was built by `makeindex`. Most references were obtained from the DBLP search engine at `http://dblp.uni-trier.de/`.

The code layout has been generated by lhs2TeX. All images were created using Ti*k*Z. The entire LaTeX code has been edited in the Kile editor. All Haskell code was written using Kate and tested with GHC 7.6.3.

Both, the text files and the Haskell files were maintained with the `git` revision control system. The text files were managed using the GitLab manager provided by the Department of Computer Science of the Kiel University, while the Haskell files were maintained on `https://github.com/`.