INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

THESE EN COTUTELLE INTERNATIONALE

pour obtenir le grade de

DOCTEUR DE L'Institut polytechnique de Grenoble
et
de l'Université Kiel

Spécialité : *«Math et Inform»*

préparée au laboratoire LIG

dans le cadre de **l'Ecole Doctorale** *«MSTII-UFR IMA»*

et au laboratoire LIG

présentée et soutenue publiquement

par

Florian DIEDRICH

Le Janvier 29,2009

Approximation Algorithms for Linear Programs and Geometrically Constrained Packing Problems
Algorithmes d'approximation pour des programmes linéaires et les problèmes de Packing avec des contraintes géometriques

Denis TRYSTRAM
Klaus JANSEN

JURY

M. Franz FAUPEL, Président
M. Evripidis BAMPIS, Rapporteur
M. Klaus JANSEN, Rapporteur
M. Denis TRYSTRAM, Directeur de thèse
M. Martin SKUTELLA, Co-encadrant (le cas échéant)
M. Dieter BLESSENOHL, Examinateur
M. Thomas WILKE, Examinateur

Approximation Algorithms for Linear Programs and Geometrically Constrained Packing Problems[1]

Dissertation

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

Dipl.-Inf. Florian Diedrich

Kiel
2009

[1] algorithmes d'approximation pour des programmes linéaires et les problèmes de Packing avec des contraintes géometriques

1. Gutachter: Prof. Klaus Jansen
2. Gutachter: Prof. Denis Trystram
3. Gutachter: Prof. Evripidis Bampis
4. Gutachter: Prof. Martin Skutella

Tag der mündlichen Prüfung: 29.01.2009
Zum Druck genehmigt: 14.04.2009

Abstract

In this thesis we approach several problems with approximation algorithms; these are feasibility problems as well as optimization problems.

In Chapter 1 we give a brief introduction into the general paradigm of approximation algorithms, motivate the problems, and give an outline of the thesis.

In Chapter 2, we discuss two algorithms to approximately generate a feasible solution of the *mixed packing and covering problem* which is a model from convex optimization. This problem includes a large class of linear programs. The algorithms generate approximately feasible solutions within $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ and $O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$ iterations, respectively, where in each iteration a *block problem* which depends on the specific application has to be solved. Both algorithms, applied to linear programs, can result in column generation algorithms.

In Chapter 3, we implement an algorithm for the so-called *max-min-resource sharing problem*. This is a certain convex optimization problem which, similar to the problem in Chapter 1, includes a large class of linear programs. The implementation, which is included in the appendix, is done in C++. We use the implementation in the context of an AFPTAS for Strip Packing in order to evaluate dynamic optimization of a parameter in the algorithm, namely the *step length* used for interpolation. We compare our choice to the static step length proposed in the analysis of the algorithm and conclude that dynamic optimization of the step length significantly reduces the number of iterations.

In Chapter 4, we study two closely related scheduling problems, namely non-preemptive *scheduling with fixed jobs* and *scheduling with non-availability* for sequential jobs on m identical machines under the makespan objective, where m is constant. For the first problem, which does not admit an FPTAS unless $\mathsf{P} = \mathsf{NP}$, we obtain a new PTAS. For the second problem, we show that a suitable restriction (namely the permanent availability of one machine) is necessary to obtain a bounded approximation ratio. For this restriction, which does not admit an FPTAS unless $\mathsf{P} = \mathsf{NP}$, we present a PTAS; we also discuss the complexity of various special cases. In total, the results are basically best possible.

In Chapter 5, we continue the studies from Chapter 4 where now the number m of machines is part of the input, which makes the problem algorithmically harder. Scheduling with fixed jobs does not admit an approximation ratio better than $3/2$, unless $\mathsf{P} = \mathsf{NP}$; here we obtain an approximation ratio of $3/2 + \epsilon$ for any $\epsilon > 0$. For scheduling with non-availability, we require a constant *percentage* of the machines to be permanently

available. This restriction also does not admit an approximation ratio better than $3/2$ unless $\mathsf{P} = \mathsf{NP}$; we also obtain an approximation ratio of $3/2 + \epsilon$ for any $\epsilon > 0$. With an interesting argument, the approximation ratio for both problems is refined to *exactly* $3/2$. We also point out an interesting relation of scheduling with fixed jobs to Bin Packing. As in Chapter 4, the results are in a certain sense best possible.

Finally, in Chapter 6, we conclude with some remarks and open research problems.

Resumée

Dans cette thèse, nous traitons plusieurs problèmes à l'aide d'algorithme d'approximation. Les problèmes sont aussi bien des problèmes de faisabilité que des problèmes d'optimisation.

Dans le Chapitre 1, nous donnons une brève introduction aux algorithmes d'approximation, motivons les problmes et donnons le plan de la thèse.

Dans le Chapitre 2, nous traitons deux algorithmes qui génèrent des solutions approximativement réalisables du *mixed packing and covering problem* qui est un modèle venant de l'optimisation convexe. Ce problème contient une grande variété de programmes linéaires. Les algorithmes génèrent des solutions approximativement réalisable en respectivement $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ et $O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$ itérations, oú à chaque itération, un *block problem* qui dépend des particularités de l'application doit tre résolu. Les deux algorithmes dans le cadre de la programmation linéaire peuvent tre des algorithmes de génération de colonnes.

Dans le Chapitre 3, nous implémentons un algorithme pour le problème classique *max-min-resource sharing problem*. Il s'agit d'un problème d'optimisation convexe qui, comme le problème du Chapitre 2, inclut de nombreux programmes linéaires. L'implémentation de ce problème, fournit en annexe, est effectuée en C++. Notre implémentation de ce se situe dans le contexte d'un algorithme AFPTAS pour le problème de *Strip Packing* dans le but d'évaluer l'optimisation dynamique d'un paramètre de l'algorithme, le *step length* utilisé pour l'interpolation. Nous comparons notre choix à la valeur statique proposée dans l'analyse de l'algorithme et concluons que l'optimisation dynamique du paramètre réduit significativement le nombre d'itérations.

Dans le Chapitre 4, nous étudions deux problèmes d'ordonnancement non-préemptif proches: le problème *scheduling with fixed jobs* et *scheduling with non-availability* pour des tâches séquentielles sur m machines identiques, optimisant le makespan, où m est constant. Pour le premier problème qui n'admet pas de FPTAS (sauf si $\mathsf{P} = \mathsf{NP}$), nous construisons un algorithme PTAS. Pour le second problème, nous montrons qu'une hypothèse raisonnable (la disponibilité permanente d'une machine) est nécessaire pour obtenir un rapport d'approximation borné. Pour le problème restreint à cette hypothèse, qui n'admet pas de FPTAS sauf si $\mathsf{P} = \mathsf{NP}$, nous présentons un algorithme PTAS. Nous discutons également de la complexité de nombreux cas particuliers. Au final, les résultats obtenus sont les meilleurs possible.

Dans le Chapitre 5, nous continuons d'étudier les problèmes du Chapitre 4, où main-

tenant le nombre de machines m fait parti de l'instance du problème, ce qui rend le problème plus compliqué algorithmiquement. Le problème *scheduling with fixed jobs* n'admet pas d'algorithme d'approximation de rapport meilleur que $3/2$ (sauf si $\mathsf{P} = \mathsf{NP}$), nous obtenons pour ce problème un algorithme de rapport d'approximation $3/2 + \epsilon$ pour n'importe quel $\epsilon > 0$. Pour le problème *scheduling with non-availability*, nous requerrons qu'une fraction constante des machines soit disponible à tout moment. Le problème restreint à cette hypothèse n'admet pas non plus d'algorithme de rapport d'approximation meilleur que $3/2$ (sauf si $\mathsf{P} = \mathsf{NP}$) et nous construisons également des algorithmes d'approximation de rapport $3/2 + \epsilon$ pour tout $\epsilon > 0$. Avec un argument intéressant, nous improuvons le rapport d'approximation á $3/2$ *précisément* pour les deux problémes. Nous exhibons également une relation intéressante entre le problème *scheduling with fixed jobs* et *Bin Packing*. De la même façon que dans le Chapitre 4, ces résultats sont dans un certain sens les meilleurs possibles.

Finalement, dans le Chapitre 6, nous concluons avec quelques remarques et donnons quelques problèmes de recherche ouverts.

Zusammenfassung

In dieser Dissertation studieren wir verschiedene Probleme mit Approximationsalgorithmen; dabei handelt es sich sowohl um Zulässigkeits- als auch um Optimierungsprobleme.

In Kapitel 1 geben wir eine kurze Einführung in das allgemeine Paradigma der Approximationsalgorithmen, motivieren die Probleme und stellen eine Gliederung der Dissertation vor.

In Kapitel 2 diskutieren wir zwei Algorithmen zur approximativen zulässigen Lösung des *gemischten Packungs- und Überdeckungsproblems*, bei dem es sich um ein Modell aus der konvexen Optimierung handelt. Diese Problemformulierung enthält eine große Klasse linearer Programme. Die Algorithmen generieren jeweils eine approximativ zulässige Lösung in $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ und $O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$ Iterationen, wobei in jeder Iteration ein anwendungsabhängiges *Blockproblem* gelöst werden muß. Beide Algorithmen können, falls sie für lineare Programme angewendet werden, gegebenenfalls als Spaltenerzeugungsalgorithmen umgesetzt werden.

In Kapitel 3 implementieren wir einen Algorithmus für das sogenannte *max-min-resource-sharing-Problem*. Bei diesem handelt es sich um ein bestimmtes konvexes Optimierungsproblem, das, ähnlich wie das Problem in Kapitel 1, eine große Klasse linearer Programme enthält. Die Implementierung, die im Anhang beigefügt ist, wurde in C++ ausgeführt. Wir benutzen diese Implementierung im Zusammenhang mit einem AFPTAS für Strip Packing, um die dynamische Optimierung eines Parameters im Algorithmus zu evaluieren, nämlich der zur Interpolation benutzten *Schrittlänge*. Wir vergleichen unsere Auswahl mit der statischen Schrittlänge aus der Analyse des Algorithmus und stellen fest, daß diese signifikant die Anzahl der Iterationen reduziert.

In Kapitel 4 studieren wir zwei eng verwandte Scheduling-Probleme, nämlich nichtpräemptives *Scheduling mit fixierten Jobs* und *Scheduling mit Nichtverfügbarkeit* von sequentiellen Jobs auf m identischen parallelen Maschinen, wobei m konstant ist. Für das erste Problem, das unter der Annahme $\mathsf{P} \neq \mathsf{NP}$ kein FPTAS besitzt, erhalten wir ein neues PTAS. Für das zweite Problem zeigen wir, dass eine geeignete Einschränkung (nämlich die permanente Verfügbarkeit einer bestimmten Maschine) nötig ist, um eine beschränkte Approximationsgüte zu erhalten. Für diese Einschränkung, die unter der Annahme $\mathsf{P} \neq \mathsf{NP}$ kein FPTAS besitzt, stellen wir ein PTAS vor; ferner diskutieren wir die Komplexität diverser Spezialfälle. Insgesamt sind unsere Ergebnisse in einem gewissen Sinne bestmöglich.

In Kapitel 5 setzen wir die Studien aus Kapitel 4 fort, wobei jetzt die Anzahl m der

Maschinen ein Teil der Eingabe ist, was das Problem algorithmisch schwerer macht. Scheduling mit fixierten Jobs erlaubt unter der Annahme $\mathsf{P} \neq \mathsf{NP}$ keine Approximationsgüte besser als 3/2; hier erhalten wir eine Approximationsgüte von $3/2 + \epsilon$ für alle $\epsilon > 0$. Für Scheduling mit Nichtverfügbarkeit setzen wir voraus, dass ein konstanter *Anteil* der Maschinen permanent verfügbar ist. Diese Einschränkung lässt unter der Annahme $\mathsf{P} = \mathsf{NP}$ ebenfalls keine Approximationsgüte besser als 3/2 zu; wir erhalten hier ebenfalls eine Approximationsgüte von $3/2 + \epsilon$ für alle $\epsilon > 0$. Mit einem interessanten Argument verbessern wir die Approximationsgüte fr beide Probleme auf *genau* 3/2. Wir stellen ferner eine interessante Verbindung zwischen Scheduling mit fixierten Jobs und Bin Packing vor. Wie in Kapitel 4 sind die erzielten Ergebnisse in einem gewissen Sinne bestmöglich.

Abschließend halten wir in Kapitel 6 einige Schlussbemerkungen fest und stellen noch offene Forschungsprobleme vor.

Acknowledgements

This thesis is a self-contained presentation of a greater part of my scientific research during my doctoral studies. I am very grateful to all my supervisors, coworkers, coauthors, reporters, examiners and friends for their enduring support. First of all I have to thank my advisor Klaus Jansen who introduced me to the intriguing fields of theoretical computer science, combinatorial optimization and approximation algorithms; his influence can be retraced even to my first undergraduate years. Without his guidance, advice, suggestions, comments, and financial support, this thesis would never have been written. Furthermore I am indebted to my supervisor Denis Trystram, who advised me during my research visits at the Laboratoire d'Informatique de Grenoble. Concerning the overall examination process, I would like to thank my external referees Evripidis Bampis and Martin Skutella; finally I am grateful to Augustin Lux for support in the organization of the co-tutelle process and Thomas Wilke, Dieter Blessenohl, and Franz Faupel for being my examiners.

In particular I would like to thank my parents, my sister, and my entire family, who always supported me in the pursuit of my studies and research. Without the help of my scientific coworkers and coauthors Mihhail Aizatulin, Daniel de Angelis Cordeiro, Olga Gerber, Rolf Harren, Britta Kehden, Gitta Marchand, Frank Neumann, Fanny Pascual, Érik Saule, Ulrich Michael Schwarz, Henning Thomas and Ralf Thöle, the entire body of research presented here would not have been possible. Clearly, science is not a one-man show; results, presentation and the overall process of research is, implicitly or explicitly, a collaborative achievement. I perceived successfully working together with my coauthors as an inspiring and impressive experience.

Furthermore I am very grateful to the Deutsche Forschungsgemeinschaft, who supported my research in the scope of the projects JA 612/10-1 "Entwicklung und Analyse von Approximativen Algorithmen für Gemischte und Verallgemeinerte Packungs- und Überdeckungsprobleme" and the Priority Program 1126 "Algorithmik großer und komplexer Netzwerke". Within the scope of the integrated project "Algorithmic Principles for Building Efficient Overlay Computers" AEOLUS IST-15964 granted by the European Union, I had the valuable possiblity to get in contact with researchers from all over Europe working in different fields of computer science. I am also indebted to the German Academic Exchange Service DAAD, who kindly granted me a "DAAD Doktorandenstipendium" in order to support my research at the Laboratoire d'Informatiqe de Grenoble in Grenoble, Isère, France. I believe that the visits there were very benefi-

cial to me, both scientifically and personally. Here, I am particularly grateful to Denis Naddef, who kindly let me stay at his place when I had severe problems finding an appartment during the first period of my scholarship. Finally I would again like to thank the European Union, the German Academic Exchange Service and the Christian-Albrechts-Universität zu Kiel for the funding of further research visits and presentation of my results at scientific conferences, workshops and symposia.

On the administrative side I am very grateful to Ursula Bazoune, Alba Castiglione, Birte Hänsch, Cornelia Hinrichsen, Ute Iaquinto, Susanne Lüdtke, Claudia Martin, Parvaneh Karimi Massouleh, Dominique Moreira, Isabelle Raffin, Antje Sommmerfeld and Annie-Claude Vial d'Allais, who greatly supported me in various issues.

For the personal part, I would also like to thank Helena Barbas, Christian Buck, Olaf Bonorden, Jihuan Ding, Pierre-François Dutot, Ling Gai, Joachim Gehweiler, Marc Grauel, Haiyang Hou, Sascha Krokowski, Feryal Kamila Moulaï, Grégory Mounié, Lars Prädel, Ulf-Peter Schroeder, Roberto Solis-Oba, Przemysław Sowa, Falk Starke, Haifeng Xu, Deshi Ye, Guochuan Zhang and Hu Zhang. Finally, I would like to thank my dear wife Anne-Marlen for her everlasting support.

Florian Diedrich
Kiel, August 2008 and February 2009

Contents

1. Introduction

Generally speaking, all interesting problems are difficult to solve, since otherwise they were not interesting. This holds especially in algorithmic and computational fields such as theoretical computer science, combinatorial optimization and operations research.

In this thesis we are interested in algorithmic problems. The basic model of computation used here can be formalized via the Turing machine, however we will concentrate more on the general algorithmic concepts without explicitly fixing a specific computational model; this approach is widely used in the field of algorithms. In total, in every case we write of an algorithm, it can be thought of as being implemented by a Turing machine or some other equivalent model of computation. Concerning algorithmic problems, suitable discussions of computational models and associated encoding schemes of problems can be found in the textbooks by Garey & Johnson [47] or Papadimitriou & Steiglitz [122].

In particular, we mainly study optimization problems. These are algorithmic problems which consist of a set of instances of inputs, a general problem definition and an objective function which is to be optimized; depending on the problem, optimization means either minimization or maximization. More precisely, every instance I together with the problem definition specifies a set of feasible solutions. The best objective value attainable for a feasible solution of I is called the optimum of I. We are interested in an algorithm that for every instance I finds a feasible solution with its objective value being equal to the optimum of I.

For most problems, for each instance I the set of feasible solutions is finite or at least it is clear that an optimal solution is contained in a suitable finite subset. In such a case it is possible to search the entire set of feasible solutions, which results in a so-called brute force approach. However, this approach typically results in an exponential runtime bound which is undesireable; we are interested in algorithms with a worst case runtime bound which is asymptotically polynomially bounded in the encoding length of the instance. An algorithm with such a runtime bound is considered to be efficient in theory and mostly also in practice.

However, for many such optimization problems, no efficient algorithm to solve them to optimality has been found. This observation has led to the theory of NP-completeness [47] which provides a solid formal foundation for the understanding of algorithmic problems and the reason why they might be difficult to solve. More precisely, it can be shown that many of these difficult problems have something in common, namely they are NP-hard.

More formally, for decision problems, there are the problem classes P and NP. The class P consists of all decision problems Π which can be solved *deterministically* within a polynomial runtime bound while the class NP consists of all decision problems Π which can be solved *non-deterministically* within a polynomial runtime bound. On the one hand, by definition it is clear that $\mathsf{P} \subseteq \mathsf{NP}$ holds [47, 122]. On the other hand, whether $\mathsf{P} = \mathsf{NP}$ is satisfied has puzzled many researchers since the advent of these formal concepts and remains one of the most fundamental problems in theoretical computer science and applied mathematics.

It is widely believed that $\mathsf{P} \neq \mathsf{NP}$ holds; in particular this assumption implies that no NP-hard problem can be *solved to optimality* within a polynomial runtime bound, just like no NP-complete problem can be *solved* within a polynomial runtime bound. Hence, not only for practical reasons, it is an interesting question whether there is a possibility to circumvent this theoretically founded problem. Fortunately, there is the concept of approximation algorithms which partly provides a remedy for the undesirable facts. More precisely, we can refrain from the goal to solve the problems to optimality but settle for suboptimal (in the case of maximization problems) or superoptimal (in the case of minimization problems) solutions which are feasible and close to an optimal solution. The latter means that we would like the objective value of the generated solution to be suitably bounded in the objective value of an optimal solution.

1.1. Approximation Algorithms

More precisely, for a minimization problem Π, let A be an algorithm to generate feasible solutions for instances of Π within a polynomial runtime bound. For any problem instance $I \in \Pi$, we denote by $\mathrm{OPT}(I)$ the value of an optimal solution of I and by $A(I)$ the value of the solution generated by A. If there is a constant $\alpha \geq 1$ such that

$$A(I) \leq \alpha \mathrm{OPT}(I)$$

holds for any $I \in \Pi$ then we call A an *approximation algorithm* for Π; in this case, α is called the *approximation ratio* or simply *ratio* of A.

In a very similar way, for a maximization problem Π, let A be an algorithm to generate feasible solutions for instances of Π within a polynomial runtime bound. Again, for any problem instance $I \in \Pi$, we denote by $A(I)$ the value of the solution generated by A. If there is a constant $\alpha \leq 1$ such that

$$A(I) \geq \alpha \mathrm{OPT}(I)$$

is satisfied for any $I \in \Pi$ then we also call A an *approximation algorithm* for Π and again α is termed the *approximation ratio* or *ratio* of A.

In either case, we also call A an *α-approximate algorithm* for Π.

Based on this approach of how to solve an NP-hard optimization problem, a plethora of results has been obtained; we refer the reader to the textbooks by Vazirani [142], Wanka [144], and Jansen & Margraf [73] for nice surveys on results and a thorough formal introduction to the subject.

The concept presented above can be slightly generalized; for minimization problems, if we have

$$A(I) \leq \alpha \mathrm{OPT}(I) + \beta$$

for some positive constant β, we say that A is an approximation algorithm with *asymptotic approximation ratio* α. For a maximization problem, if we have

$$A(I) \geq \alpha \mathrm{OPT}(I) - \beta$$

for some positive constant β, we also call A an approximation algorithm with *asymptotic approximation ratio* α.

Furthermore, in the sense of approximability of an NP-hard problem, the best possible result is an *approximation scheme*; for a minimization problem Π, this is a family

$$\{A_\epsilon\}_\epsilon$$

of approximation algorithms parameterized via a desired accuracy $\epsilon \in (0, \infty)$ such that for every ϵ the algorithm A_ϵ is a $(1+\epsilon)$-approximate algorithm for Π.

Likewise, for a maximization problem Π, the best possible result is a family

$$\{A_\epsilon\}_\epsilon$$

of approximation algorithms such that for any ϵ the algorithm A_ϵ is a $(1-\epsilon)$-approximate algorithm for Π.

In either case, the family of algorithms is called a *polynomial-time approximation scheme*, abbreviated as PTAS. Algorithmically speaking, the existence of such a scheme permits a trade-off between running time and the quality of the generated solution. Note that, however, the dependency of the worst-case runtime bound of the scheme is permitted to scale *exponentially* in the inverse of ϵ. If additionally the worst-case runtime bound of the scheme scales *polynomially* in the inverse of ϵ, such a scheme is called a *fully polynomial-time approximation scheme*, abbreviated as FPTAS.

If for a family of algorithms $\{A_\epsilon\}_\epsilon$ for a fixed ϵ the approximation ratio of A_ϵ is asymptotic, say we have

$$A_\epsilon \leq \alpha \mathrm{OPT}(I) + \beta_\epsilon$$

for minimization problems or

$$A_\epsilon \geq \alpha \mathrm{OPT}(I) - \beta_\epsilon$$

for maximization problems, where the constant β_ϵ depends on ϵ, we call $\{A_\epsilon\}$ an *asymptotic approximation scheme*. Depending on the type of runtime bound, the scheme might be an *asymptotic polynomial-time approximation scheme*, abbreviated as APTAS, or an *asymptotic fully polynomial-time approximation scheme*, abbreviated as AFPTAS.

In a certain sense, an FPTAS is more desirable than a PTAS; however, for reasons discussed in detail in [46], optimization problems which are *strongly* NP-hard (and satisfy some additional yet very natural requirement) do not permit an FPTAS unless $\mathsf{P} = \mathsf{NP}$; in such a situation, a PTAS is the best type of algorithm that can be found. In a very similar way, for certain problems it can be proved that there is a suitable bound on the approximability; for minimization problems this means that an approximation with approximation ratio *smaller* than a certain value of α does not exist unless $\mathsf{P} = \mathsf{NP}$ holds; likewise, under the same assumption, for certain maximization problems it might be the case that an approximation algorithm with approximation ratio *larger* than a certain value of α is not possible.

1.2. Outline of This Thesis

The remainder of this thesis is organized as follows. Each chapter is to be understood as a self-contained presentation of the described results and should be accessible to

the interested reader. However, we assume the reader to be familiar with the basic concepts of mathematical notation, algorithms, complexity theory and ideally approximation algorithms; for introductions to these topics, we refer the reader to the textbooks [47, 73, 122, 135, 142, 144].

In *Chapter 2*, we study the approximability of a certain class of mathematical programs. More precisely, we study the so called *mixed packing and covering problem* which can be formulated as

$$\begin{aligned} &\text{compute } x \in B \text{ such that } f(x) \leq a, \quad g(x) \geq b \\ &\text{or correctly decide that } \{x \in B | f(x) \leq a, \quad g(x) \geq b\} = \emptyset; \end{aligned} \tag{MPC}$$

where B is a nonempty convex compact subset of the Euclidean vector space $\mathbb{R}^N$ where $N \in \mathbb{N}$, $f, g : B \to \mathbb{R}^M_+$ are two nonnegative functions where $M \in \mathbb{N}$, $M \geq 2$, f is component-wise convex and g is component-wise concave, and finally, two vectors $a, b \in \mathbb{R}^M_{++}$ which are positive. Note that for the case of *linear* function vectors f and g and B being a polytope this problem can be solved via linear programming, i.e. it is polynomially solvable. Nevertheless we study the problem under the paradigm of approximation algorithms for the following various reasons.

- The exact methods to solve linear programs have polynomial runtime bounds of very large degree; for practical applications it might be desirable to trade-off accuracy for speed.

- In actual practical applications, it might not be necessary to solve the problem instance to optimality which, in computer systems which use floating-point representation of rational numbers, can be achieved only within the limitations of the accuracy of the underlying data structures. This means that here it might be an inappropriate goal to use the implementation of an exact algorithm which, despite its theoretical quality, will not solve the instance to optimality in the actual implementation.

- Complementing the point above, it is debatable whether in an actual application the input data is accurate to more than two or three decimal digits in the first place. Again, the goal to solve the instance to optimality might not be appropriate.

- Many combinatorial optimization problems, as discussed more specifically in Chapter 2 and Chapter 3, permit formulations as linear programs which use a number of variables which is exponential in some more natural and compact formulation

of the problem. For such problems, with the approach used here it is possible to obtain so-called column generation algorithms which permit to obtain an approximate solution without generating an explicit instance of the linear programming formulation of exponential size.

Note that, however, here the motivation to study approximation algorithms is totally different from the one provided by the theory of NP-completeness; the algorithm presented here generates solutions which are not approximately *optimal*, but approximately *feasible*. Based on a suitable subroutines termed as *block solvers*, we obtain algorithms wich for an accuracy parameter ϵ either find an $x \in B$ such that

$$f(x) \leq c(1+\epsilon)a, \quad g(x) \geq (1-\epsilon)b/c$$

or correctly decide that $\{x \in B | f(x) \leq a, g(x) \geq b\} = \emptyset$. The number of iterations (i.e. the number of calls to the respective block solver) of the algorithms are bounded by $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ and $O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$, respectively. In total, our result generalizes the approach pursued in [66]; the results presented here have been published in [32, 33]. As a side issue, we discuss how the pursued approach can be used to approximately solve a multicommodity flow problem. Note that the line of research as well as the techniques used are similar to the one in Chapter 3.

In *Chapter 3*, we present experimental results obtained with an implementation of an approximation algorithm for the max-min resource sharing problem. Although the problem considered here is an optimization problem, it is of similar nature as the one from Chapter 2 and can be formulated as

$$\text{compute } x \in B \text{ such that } f(x) \geq (1-\epsilon)\lambda_{\mathcal{C}}^* e$$

where $f : B \to \mathbb{R}_+^M$ are vectors of M continuous concave nonnegative functions, which are defined on a polytope $B \subseteq \mathbb{R}^N$; finally $e \in \mathbb{R}^M$ denotes the constant unit vector. Furthermore $\lambda_{\mathcal{C}}^* := \max\{\lambda_{\mathcal{C}}(x) | x \in B\}$ is the optimum of the objective function

$$\lambda_{\mathcal{C}} : B \to \mathbb{R}_+^M, \quad x \mapsto \min\{f_m(x) | m \in \{1, \ldots, M\}\}.$$

We implement the algorithm from [54]. Besides artificial general instances, we study the behaviour of the algorithm within the context of an AFPTAS for Strip Packing from [71, 91, 92]. The approach features a relaxation of Strip Packing which results in a linear programming model with a number of variables which may grow exponentially

in the encoding size of the original instance; here the technique from [54] yields an algorithm which uses column generation. We used a generic implementation in C++ which is included in the appendix. Besides minor considerations, we mainly studied the behaviour of the algorithm if the step length used for interpolation to obtain a new iterate $x \in B$ from the theoretical analysis is replaced by a dynamically optimized step length. Via this approach, we obtain a significant speedup of the overall algorithm. The results of Chapter 3 have been published in [1].

The last two chapters deal with in total four closely related very classical scheduling problems; however, we present the results grouped by algorithm design paradigm rather than by model. In Chapter 4, we also present some basic details which are use later for the more sophisticated algorithms in Chapter 5.

In *Chapter 4*, we study the problem of constrained scheduling where the number m of machines is constant. More precisely, we are given m identical parallel machines and n sequential jobs given by their processing times; we consider non-preemptive schedules and would like to minimize the makespan $C_{\max}$, which is the maximum completion time of all jobs. However, our scheduling problem is constrained in one of the following two ways.

- The first k jobs are already fixed in the system, i.e. their position in the schedule is encoded in the instance and we are free to schedule the remaining $n-k$ jobs. This problem is called *scheduling with fixed jobs*. Here, the objective to be minimized is

$$C_{\max} := \max\{C_j | j \in \{1, \ldots, n\}\}$$

  which is the maximum completion time of *all* jobs.

- For each machine there may be some intervals of non-availability (similar as above, these are artificially encoded as the first k jobs) during which no jobs can be processed; the jobs must be scheduled non-preemptively such that there is no overlap between the jobs and the intervals of non-availability. Again, the objective is to minimize the makespan $C_{\max}$. This problem is called *scheduling with non-availability*. Here, the objective to be minimized is

$$C_{\max} := \max\{C_j | j \in \{k+1, \ldots, n\}\}$$

  which is the maximum completion time of the *non-fixed* jobs.

Clearly, both problems are NP-hard since they generalize the well-known scheduling

problem $Pm||C_{\max}$ [127]. They can be motivated, for instance, as different perspectives on the same parallel system. On one hand, scheduling with fixed jobs models the system from the perspective of the *system administrator* who wishes to execute all jobs as soon as possible where some high-priority jobs are already preallocated in the system. On the other hand, scheduling with non-availability models the behaviour of the same system from the perspective of a *user* who wishes only his or her own jobs to be finished as soon as possible under the presence of other jobs, which do not contribute to the subjective makespan however. We approach the problem based on a PTAS for the multiple subset sum problem in combination with dual approximation via binary search over the makespan. For scheduling with fixed jobs, we obtain a PTAS where no FPTAS is possible unless $\mathsf{P} = \mathsf{NP}$ holds; however this had already been discovered by Scharbrodt, Steger & Weisser [130, 131, 132] and is only included for the sake of completeness.

From an algorithmic point of view, scheduling with non-availability behaves quite differently. The problem does not permit a constant approximation ratio in the general case, hence it makes sense to study suitable restrictions. We prove that a restriction where for every time step there is an available machine is not sufficient to obtain a constant approximation ratio; however, if we require to have a machine which is permanently available we obtain a PTAS where no FPTAS is possible unless $\mathsf{P} = \mathsf{NP}$. As a side issue, we discuss the complexity of various special cases of scheduling with fixed jobs as well as scheduling with non-availability. Parts of the results discussed in this chapter have been published as [35].

In *Chapter 5*, we continue the study of constrained scheduling. More precisely, the formal definition of the problem is the same as in Chapter 4, except for the number m of machines is not regarded as being constant but part of the input. Here, the resulting problem formulations behave in a slightly different way as their counterparts where m is constant and the approach used in Chapter 4 cannot be transferred directly. The used techniques include dual approximation via binary search over the makespan, linear grouping and definition of configurations as known from algorithms for classical Bin Packing [30] or Strip Packing [91, 93]. Furthermore, as in Chapter 4, we use a PTAS for the multiple subset sum problem as an algorithmic building block; however the main feature of our technique is a flow-based assignment method of large jobs and configurations which can be analyzed with an elegant cyclic shifting argument. We believe the latter idea to have further nice applications in geometrically or numerically constrained packing problems in combinatorial optimization.

Concerning the specific results, on the one hand, for scheduling with fixed jobs we

obtain an approximation algorithm with ratio $3/2 + \epsilon$ for any positive accuracy ϵ where a ratio better than $3/2$ is impossible unless $\mathsf{P} = \mathsf{NP}$ holds. On the other hand, for scheduling with non-availability, we show that approximation of this problem is at least as hard as approximation of Bin Packing with an additive error, which is an interesting open problem, however. Furthermore, we study a suitable restriction of the problem, namely we require a constant *percentage* of the machines to be permanently available. This restriction, even if we admit only at most one interval of non-availablity per machine, does not admit an approximation ratio better than $3/2$. On the positive side, our approach is proved to yield an approximation ratio of $3/2 + \epsilon$ for any positive accuracy parameter ϵ.

Finally, in *Chapter 6*, we finish the thesis with some concluding remarks as well as open research problems.

2. Approximation of Mixed Packing and Covering Problems

In this chapter we study the so-called *mixed packing and covering problem* which is the feasibility variant of a certain mathematical program. More precisely, an instance of this problem is constituted by a nonempty convex compact subset $\emptyset \neq B \subseteq \mathbb{R}^N$ of the Euclidean vector space $\mathbb{R}^N$ where $N \in \mathbb{N}$, two nonnegative functions $f, g : B \to \mathbb{R}^M_+$ where $M \in \mathbb{N}$, $M \geq 2$, f is component-wise convex and g is component-wise concave, and finally, two vectors $a, b \in \mathbb{R}^M_{++}$ which are positive. The *exact packing and covering problem* then can be formulated as

$$\begin{aligned} &\text{compute } x \in B \text{ such that } f(x) \leq a, \quad g(x) \geq b \\ &\text{or correctly decide that } \{x \in B | f(x) \leq a, \quad g(x) \geq b\} = \emptyset; \end{aligned} \tag{MPC}$$

informally this means that we would like to find a vector $x \in B$ that satisfies the constraints given by $f(x) \leq a$ and $g(x) \geq b$ or to find a proof that no such vector exists.

However, we are interested in a solution that is only approximately feasible. More precisely, we would like to obtain algorithms which, given a suitable constant $c \in \mathbb{R}_+$ and an accuracy parameter $\epsilon \in (0, 1)$, either find an $x \in B$ such that

$$f(x) \leq c(1 + \epsilon)a, \quad g(x) \geq (1 - \epsilon)b/c$$

or correctly decides that $\{x \in B | f(x) \leq a, g(x) \geq b\} = \emptyset$. To this end, we assume that the set B is given only implicitly but can be queried by a so-called *approximate block solver*; this is an algorithm which constructively solves an (easier) feasibility problem on B with a relative error depending on ϵ and c. With the help of this approximate block solver, the mixed packing and covering problem can be solved approximately by generating a suitable sequence of vectors, i.e. by iterating upon a vector $x \in B$.

The special case $c = 1$ has been studied extensively in [66, 70], where two quite different block solvers have been used; we show how both approaches can be generalized

to the case of arbitrary $c \geq 1$. In total, we present the following contributions; parts of the results of this chapter have been published in [32, 33].

1. We generalize the algorithm from [70] by using a more general block solver, resulting in an algorithm which needs only $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ calls to the corresponding block solver.

2. We generalize the algorithm from [66] by using a more general block solver, resulting in an algorithm which needs only $O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$ calls to the corresponding block solver.

The remainder of this chapter is organized as follows. First we formally present the problem and give a brief review of related results. Afterwards we introduce some basic concepts which are necessary for both algorithms. These two algorithms, which are presented afterwards, are based on quite different types of block solvers. In the second to last section we present an application of the first algorithm for a multicommodity flow problem. Finally we conclude with a survey on related results in a broader context and open some open research problems.

2.1. Introduction

As briefly sketched above, we study the approximate general mixed packing and covering problem

$$\begin{array}{c} \text{compute } x \in B \text{ such that } f(x) \leq c(1+\epsilon)a, \quad g(x) \geq (1-\epsilon)b/c \\ \text{or correctly decide that } \{x \in B | f(x) \leq a, \quad g(x) \geq b\} = \emptyset \end{array} \tag{$MPC_{c,\epsilon}$}$$

where $\emptyset \neq B \subseteq \mathbb{R}^N$ is a convex compact set, $f, g : B \to \mathbb{R}^M_+$ are vectors of convex and concave functions, respectively, which are nonnegative on B; $a, b \in \mathbb{R}^M_{++}$ are positive vectors. Note that $a = e = b$ holds without loss of generality where $e \in \mathbb{R}^M_+$ denotes the unit vector; otherwise, similar to [66, 70, 148, 150], we replace f_m and g_m by setting

$$f_m := f_m/a_m \quad \text{and} \quad g_m := g_m/b_m$$

and do a suitable retransformation after solving ($MPC_{c,\epsilon}$).

Our measure of runtime complexity is the number of iterations or *coordination steps*, which is the number of calls to the block solver; the number of coordination steps

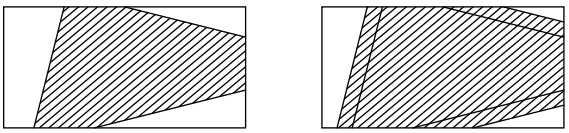


Figure 2.1.: Comparison of (MPC) with the relaxed version ($MPC_{c,\epsilon}$) where $B \subseteq \mathbb{R}^2$ is a rectangle with 3 linear constraints; hatched areas indicate the feasible regions.

in dependence of the instance size is also termed the *coordination complexity*. Price-directive decomposition algorithms have several potential advantages in comparison to classical methods – they can be faster, easier to implement, and often result in column-generation algorithms which can efficiently solve models with an exponential number of variables. Applications for ($MPC_{c,\epsilon}$) are multicommodity flow [148, 150], capacitated network design with fixed total cost [21, 43] and the network access regulation problem [2, 3]. For a conceptual comparision of the exact and the approximate formulation, see Figure 2.1 for some intuition behind our approach.

2.1.1. Related Problems and Previous Results

Related problems are so-called *pure* packing and covering problems. Approximation algorithms for *pure packing problems* with convex functions were studied in [51, 53, 143] where finally a runtime bound of $O(M(\epsilon^{-2} \ln \epsilon^{-1} + \ln M))$ is obtained. Garg & Könemann [49, 50] found a width-independent algorithm with runtime bound $O(M\epsilon^{-2} \ln M)$ for the packing problem with linear constraints. Bienstock & Iyengar [10, 11] described an algorithm for the same problem with runtime bound $O^*(\epsilon^{-1}\sqrt{KN})$ where K is the maximum number of non-zeros per row and in each iteration a quadratic program has to be solved. Similarly, Chudak & Eleutério [25] found an algorithm with coordination complexity $O(\sqrt{N \ln M}(\log\log\sqrt{N} + \epsilon^{-1}))$. Jansen & Zhang [79] presented an algorithm which parallels the result in [54]. They obtain $O(M(\epsilon^{-2} \ln \epsilon^{-1} + \ln M))$ as a runtime bound, although a (weaker) block solver with ratio c is used; for block solvers with arbitrary precision the runtime is improved to $O(M(\epsilon^{-2} + \ln M))$, matching the bound from [54]. Experiments with this algorithm applied to the multicast congestion problem were done by Lu & Zhang in [117].

For so-called *pure covering problems* with concave functions, a runtime bound of $O(M(\epsilon^{-2} + \ln M))$ iterations is obtained in [54]; here a strong block solver with arbitrarily high precision is needed. This was generalized in [74, 75] where a general block

solver is used, resulting in $O(M(\ln M + \epsilon^{-2} \ln c + \epsilon^{-2}))$ coordination steps; however, this bound depends on the ratio c of the block solver. In [67, 69], this shortcoming was removed by presenting an improved algorithm with runtime bound $O(M(\epsilon^{-2} \ln \epsilon^{-1} + \ln M))$ which matches the best known bound for solving the fractional covering problem with a similar approach in [79]. As discussed in Chapter 3, in [1] experiments with the algorithm from [54] were done, where two-dimensional strip-packing from [71] was used as a testbed.

Finally, *mixed packing and covering* problems have been studied. For the mixed packing and covering problem with linear constraints, Plotkin, Shmoys & Tardos [124] proposed algorithms where the coordination complexity depends on the width. Their algorithm uses $O(M^2(\ln^2 \rho)\epsilon^{-2} \ln(\epsilon^{-1} M \ln \rho) \ln \rho)$ calls to an oracle of the form: find a vertex $\hat{x} \in B$ with $f(\hat{x}) \leq va$ and

$$c^T f(\hat{x}) - \sum_{m \in I(v,\hat{x})} d_m g_m(\hat{x}) = \min\{c^T f(x) - \sum_{m \in I(v,x)} d_m g_m(x) | x \text{ is a vertex of } B\},$$

where $I(v, x) = \{m | g_m \leq v b_m\}$, v is constant, and

$$\rho = \max_{x \in B} \max_{m \in [M]} \{f_m / a_m, g_m / b_m\}$$

is the width.

An important result here is [148, 150], where with a different technique a running time of $O(Md\epsilon^{-2} \log M)$ was obtained; here d is the maximum number of constraints any variable appears in. However, here the case with linear f, g and $B = \mathbb{R}^N_+$ is solved; this can be generalized since a polytope B with N vertices can be reduced to $B' = \mathbb{R}^N_+$ by using a variable $x'_i \geq 0$ for each vertex v_i of B that denotes its coefficient in a convex combination of an arbitrary $x \in B$ and studying the problem of finding $x' \in \mathbb{R}^N_+$ such that $\mathcal{P}x' \leq e$, $\mathcal{C}x' \geq e$, $e^T x' \leq 1$ and $e^T x' \geq 1$ where $\mathcal{P}, \mathcal{C}$ are suitable modifications of the packing and covering constraints, respectively [151]. However [148, 150] mentions the case of general B as an open problem which was later solved in [66, 70]. Garg & Khandekar [96] proposed an algorithm for our problem with runtime bound $O(M\epsilon^{-2} \ln M)$; they used the *exponential* potential function and a feasibility oracle that solves the so-called *on-line prediction problem*. We refer the reader to [8] for a survey on the technique and recent theoretical and practical results; however, there the focus is on the *exponential* potential function resulting in different details.

2.1.2. Contributions

Our first contribution is the algorithm which has been published in [32, 33]. Our algorithm uses an approximate block solver; it solves each instance of $(MPC_{c,\epsilon})$ in only $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ iterations. The analysis of the algorithm is simpler than the analysis of the previous algorithms presented in [66, 96]. This first algorithm avoids the *on-line prediction problem* in [96]. Additionally, the algorithm is slightly faster than the others. Furthermore, within some limitations, it might be amenable to the usage of *line search*; this is a heuristic technique discussed in Chapter 3 which might be interesting for practical applications.

Our second contribution is a generalization of the algorithm from [66, 70]; there, the problem is solved with a stronger block solver. We present a generalization with a weaker block solver; our algorithm matches the runtime bound of $O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$ from [66, 70], but permits the usage of more general block solvers.

2.1.3. Main Ideas.

To obtain our results, different techniques from the existing body of literature are put into effect. We use a technical improvement of the class of modified logarithmic potential functions from [54, 66, 67, 69, 70, 74, 75, 79] and the resulting price vectors to govern the direction of optimization for the respective block solvers.

For the first algorithm we use elimination of indices of covering constraints similar to [67, 69], where we later prove suitable bounds for the values of the covering functions. For the second algorithm we use storing a part of the history of iterates and generation of a suitable convex combination after each scaling phase similar to [66, 70]. We use types of stopping rules similar to [66, 67, 69, 70, 79] for early termination of scaling phases. The usage of a more general block solver here is motivated by the results from [67, 69, 79]. Finally, we prove that infeasibility of certain instances of the block problem implies infeasiblity of the original instance, which can be seen as a direct proof in infeasibility of the respective input.

The remainder of this chapter is organized as follows. In Section 2.2 we discuss the basic techniques. In Section 2.3 we describe a first algorithm which is then analyzed Section 2.4. In Section 2.5 we present a second algorithm which is then analyzed in Section 2.6. Finally we conclude the chapter in Section 2.8 with a more detailed survey of related results and open research problems.

2.2. Basic Techniques

In this section we present the used modified logarithmic potential function and its basic properties. Note that these properties do not depend on properties of the used block solvers. As in [54, 79] we use suitable notational abbreviations. For $x, \hat{x}$, and $x' \in B$ we use $f, \hat{f}$ and f' to denote the evaluations of f; $g, \hat{g}$ and g' are defined in a similar way. The algorithms will use sets $A \subseteq \{1, \ldots, M\}$ to define active sets of indices associated with each iterate for the covering functions; these will be denoted as A and A'. In each scaling phase a component g_m of g is eliminated if $g_m \geq T$ where T is a threshold value; for $T \in \mathbb{R}_+$ and $x \in B$ let $A \subseteq \{1, \ldots, M\}$ be the corresponding set of indices of active functions. Furthermore $t \in (0, 1)$ is an additional accuracy parameter similar to [54] which will be changed in each scaling phase.

2.2.1. Modified Logarithmic Potential Function

Here we write $A := A(x, T)$ for short; we use the *potential function*

$$\Phi_t(\theta, x, A) := 2 \ln \theta - \frac{t}{CM}[\sum_{m=1}^{M} \ln(\theta - f_m) + \sum_{m \in A} \ln(g_m - \frac{1}{\theta}) + (M - |A|) \ln T]$$

where C is a constant to be chosen later in the presentation of the algorithms. Note that, intuitively, the last summand means that the evaluation of every function g_m with $m \in M \setminus A$ is replaced by T. For fixed $x \in B$, the potential function is defined for $\theta \in (\lambda_A(x), \infty)$ where

$$\lambda_A(x) := \max\{\max_{m \in \{1,\ldots,M\}} f_m, \max_{m \in A} 1/g_m\}.$$

If there is an $m \in A$ such that $g_m = 0$ we let $\lambda_A(x) := \infty$; furthermore we denote $\lambda(x) := \lambda_{\{1,\ldots,M\}}(x)$. Additionally we define the *reduced potential function* by

$$\phi_t(x, A) := \min\{\Phi_t(\theta, x, A) | \theta \in (\lambda_A(x), \infty)\}$$

and denote $\lambda_A := \lambda_A(x)$ for short where the dependency is clear. Notice that the corresponding minimizer $\theta \in (\lambda_A(x), \infty)$ can be determined from the first-order optimality condition

$$\frac{t}{2CM}[\sum_{m=1}^{M} \frac{\theta}{\theta - f_m} + \frac{1}{\theta} \sum_{m \in A} \frac{1}{g_m - 1/\theta}] = 1 \tag{2.1}$$

which can be seen by calculating the derivation with respect to θ of the right hand side of the definition of the potential function. Furthermore, for any fixed $x \in B$ the continuous function

$$(\lambda_A, \infty) \to \mathbb{R}, \quad \theta \mapsto \frac{t}{2CM}[\sum_{m=1}^{M} \frac{\theta}{\theta - f_m} + \frac{1}{\theta} \sum_{m \in A} \frac{1}{g_m - 1/\theta}]$$

is strictly monotonically decreasing in $\theta \in (\lambda_A, \infty)$. Hence the minimizer $\theta \in (\lambda_A, \infty)$ is uniquely determined; we denote it by $\theta_A(x)$ and remark that it can be approximated arbitrarily close by binary search. Similar as above, we use θ, θ' and λ, λ' to denote the corresponding minimizers and evaluations of λ for $x, x' \in B$ when the dependency is clear. By Lemma 1, $\theta_A(x)$ approximates λ_A for small values of t; the proof is parallel to the one of Lemma 2.1 in [66, 70].

Lemma 1. *We have* $\theta/[1 + t/(2CM)] \geq \lambda_A \geq \theta(1 - t(M + |A|)/(2CM)] \geq \theta(1 - t/C)$ *for any* $t \in \mathbb{R}_+$ *and* $\theta := \theta_A(x)$.

Proof. We consider two cases. *Case 1:* There is an $m \in \{1, \ldots, M\}$ such that $\lambda = f_m$. Then (2.1) yields $t\theta/[2CM(\theta - \lambda)] \leq 1$, from which follows $\lambda \leq \theta[1 - t/(2CM)]$ by rearranging. *Case 2:* There is an $m \in A$ such that $\lambda = 1/g_m$. Then (2.1) yields $(t/\theta)/[2CM(1/\lambda - 1/\theta)] \leq 1$ and we obtain $\lambda \leq \theta/[1 + t/(2CM)]$ by rearranging. Additionally we have $1 - t/(2CM) \leq 1/[1 + t/(2CM)]$; this means $\lambda \leq \theta/[1 + t/(2CM)]$ holds in both cases, which proves the first inequality. For each $m \in \{1, \ldots, M\}$ we have $f_m \leq \lambda$, which yields $\theta/(\theta - f_m) \leq \theta/(\theta - \lambda)$. In a similar way, for each $m \in A$ the inequality $g_m \geq 1/\lambda$ holds. We obtain the inequality $g_m - 1/\theta \geq (\theta - \lambda)/(\lambda\theta)$ by inserting, from which follows

$$(1/\theta)/(g_m - 1/\theta) \leq \lambda/(\theta - \lambda) \leq \theta/(\theta - \lambda)$$

by rearranging and using $\theta < \lambda$ for the last estimation. Using (2.1) we obtain

$$1 = \frac{t}{2CM}[\sum_{m=1}^{M} \frac{\theta}{\theta - f_m} + \sum_{m \in A} \frac{1/\theta}{g_m - 1/\theta}] \leq \frac{t\theta(M + |A|)}{2CM(\theta - \lambda)};$$

solving for λ and $|A| \leq M$ yields $\lambda \geq \theta[1 - t(M + |A|)/(2CM)] \geq \theta(1 - t/C)$. □

Additionally we obtain bounds for the reduced potential $\phi_t(x, A)$ which also basically depend on $\theta_A(x)$; here we have to distinguish between the case $C = 1$ and $C \neq 1$. These bounds are vital for the following analysis. As in [70], the type the of bound and the proof technique are similar to the analysis in [79].

Lemma 2. *Let $C = 1$ and $x \in B$. If $g_m \leq T$ for each $m \in A$, then*

$$\phi_t(x, A) \geq (2 - t) \ln \theta - t \ln T$$

holds. If $T > 1/\lambda_A(x)$ is satisfied, we also have

$$\phi_t(x, A) < 2 \ln \theta + 2t \ln(2M/T) + t \ln[1 + t/(2M)].$$

Proof. Let $\lambda := \lambda_A(x)$. If $g_m \leq T$ for each $m \in A$ holds, we use the definition of the reduced potential function to obtain the calculation

$$2 \ln \theta \leq \phi_t(x, A) + \frac{t}{M}[\sum_{m=1}^{M} \ln \theta + M \ln T] = \phi_t(x, A) + t \ln \theta + t \ln T$$

which we rearrange to yield the first part of the claim. Furthermore we have $f_m \leq \lambda$ for each $m \in \{1, \ldots, M\}$ and $g_m \geq 1/\lambda$ for each $m \in A$. Using $T > 1/\lambda$, we obtain

$$2 \ln \theta > \phi_t(x, A) + \frac{t}{M}[\sum_{m=1}^{M} \ln(\theta - \lambda) + \sum_{m \in A} \ln(1/\lambda - 1/\theta) + (M - |A|) \ln(1/\lambda - 1/\theta)]$$

similar as before. Note that we can also use $\phi_t(x, A) + 2 \ln(\theta - \lambda) + t \ln[1/(\lambda\theta)]$ to write the right hand side above. Lemma 1 yields $\lambda \leq \theta/(1 + t/(2M))$, from which $\theta - \lambda \geq [\theta t/(2M)]/[1 + t/(2M)]$ can be obtained. Using the same bound again, we have $1/(\lambda\theta) \geq [1 + t/(2M)]/\theta^2$; we obtain

$$2 \ln \theta > \phi_t(x, A) + 2t \ln \frac{\theta t/(2M)}{1 + t/(2M)} + t \ln \frac{1 + t/(2M)}{\theta^2}$$

by inserting these bounds into the inequality above. Finally further elementary transformation yields $2 \ln \theta > \phi_t(x, A) + 2t \ln[t/(2M)] - t \ln[1 + t/(2M)]$ which we rearrange for the second bound. □

Lemma 3. *Let $C \neq 1$. If $g_m \leq T$ for each $m \in A$, then $\phi_t(x, A) \geq 2 \ln \theta - t/C \ln(\theta T)$ is satisfied. Furthermore the inequality*

$$\phi_t(x, A) \leq 2 \ln \theta - \frac{t(M - |A|)}{CM} \ln(\theta T) - \frac{t(M + |A|)}{CM} \ln[\frac{t(M + |A|)}{2CM}]$$

is valid.

Proof. If $g_m \leq T$ for each $m \in A$, we use the definition of the reduced potential function

to obtain

$$2\ln\theta \le \phi_t(x,A) + \frac{t}{CM}[\sum_{m=1}^{M} \ln\theta + M\ln T] = \phi_t(x,A) + \frac{t}{C}[\ln\theta + \ln T]$$

which we rearrange to yield the first claim. Application of the function $\ln(\cdot)$ to both sides of (2.1) and using the concavity of the function $\ln(\cdot)$ we obtain

$$0 \ge \ln\frac{t(M+|A|)}{2CM} + \frac{1}{M+|A|}(\sum_{m=1}^{M} \ln\frac{\theta}{\theta - f_m} + \sum_{m\in A} \ln\frac{1}{\theta g_m - 1}).$$

Multiplication of both sides with $t(M+|A|)/(CM)$ yields

$$0 \ge \frac{t(M+|A|)}{CM}\ln\frac{t(M+|A|)}{2CM} + \frac{t(M-|A|)}{CM}\ln\theta - \frac{t}{CM}[\sum_{m=1}^{M}\ln(\theta - f_m) + \sum_{m\in A}\ln(g_m - 1/\theta)].$$

Adding $2\ln\theta - t(M-|A|)/(CM)\ln T$ to both sides yields

$$2\ln\theta - \frac{t(M-|A|)}{CM}\ln T \ge \phi_t(x,A) + \frac{t(M+|A|)}{CM}\ln\frac{t(M+|A|)}{2CM} + \frac{t(M-|A|)}{CM}\ln\theta$$

which we rearrange to obtain the second claim. □

2.2.2. Price Vectors

We define the price vectors in order to ensure that the block solver optimizes in a suitable direction. As in [54, 66, 67, 69, 70, 79] the price vectors are obtained from (2.1) in a natural way; for $x \in B$, $A \subseteq \{1,\dots,M\}$ we use

$$p_m(x,A) := \frac{t\theta}{2CM(\theta - f_m)} \text{ for each } m \in \{1,\dots,M\}$$

and

$$q_m(x,A) := \begin{cases} \frac{t}{2CM(g_m\theta - 1)} & : \text{ for each } m \in A \\ 0 & : \text{ for each } m \in \{1,\dots,M\}\setminus A \end{cases}$$

to define the suitable price vectors. Note that the components of $p(x,A)$ and $q(x,A)$ are the summands in (2.1); hence the entries are nonnegative and we have $e^T p + e^T q = 1$.

If the dependency is clear, we write $\bar{p} := e^T p \leq 1$ and $\bar{q} := e^T q \leq 1$. The proof for the next lemma is similar to the one of Lemma 2.3 in [66, 70].

Lemma 4. *Denoting $p := p(x, A)$ and $q := q(x, A)$, we have*

$$p^T f = \theta[\bar{p} - t/(2C)] \leq \theta[1 - t/(2C)] \text{ and}$$
$$q^T g = [\bar{q} + t|A|/(2CM)]/\theta \leq [\bar{q} + t/(2C)]/\theta \leq [1 + t/(2C)]/\theta.$$

Proof. Using the definition of p, we have

$$\begin{aligned} p^T f &= \frac{t\theta}{2CM} \sum_{m=1}^{M} \frac{f_m}{\theta - f_m} \\ &= \frac{t\theta}{2CM} \sum_{m=1}^{M} (-1 + \frac{\theta}{\theta - f_m}) \\ &= -\frac{t\theta}{2C} + \theta\bar{p} \\ &= \theta[\bar{p} - \frac{t}{2C}] \\ &\leq \theta[1 - \frac{t}{2C}] \end{aligned}$$

which yields the first statement. We use the definition of q and obtain

$$\begin{aligned} q^T g &= \frac{t}{2CM} \sum_{m \in A} \frac{g_m/\theta}{g_m - 1/\theta} \\ &= \frac{t}{2CM\theta} \sum_{m \in A} (1 + \frac{1/\theta}{g_m - 1/\theta}) \\ &= \frac{t|A|}{2CM\theta} + \frac{t}{2CM\theta}\bar{q} \\ &= [\bar{q} + \frac{t|A|}{2CM}]/\theta \\ &\leq [\bar{q} + \frac{t}{2C}]/\theta \\ &\leq [1 + \frac{t}{2C}]/\theta \end{aligned}$$

which in total shows the claim. □

2.3. The First Algorithm

Let $c \geq 1$ be a lower bound for the approximation ratio of the *block solver*, which here is an algorithm that queries B by an approximate feasibility oracle of the form

$$\begin{aligned} &\text{find } \hat{x} \in B \text{ such that } p^T f(\hat{x})/Y(c,t) - q^T g(\hat{x}) Y(c,t) \leq \alpha \\ &\quad \text{or correctly decide that there is no } x \in B \text{ such that} \\ &\qquad p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) \leq \alpha \end{aligned} \qquad (ABS1_c(p,q,\alpha,t))$$

where $t \in (0,1)$, $Y(c,t) := c(1+8/3t)(1+t)$ is a parameter defined for ease of exposition, $p, q \in \mathbb{R}^M_+$ such that $e^T p + e^T q = 1$ and α depends on p and q as defined later. In contrast to [54, 67, 69, 79], $ABS1_c(p,q,\alpha,t)$ solves only a feasibility problem and not an optimization problem; however, this can be done by minimizing a convex function over B. This latter problem was studied by Vaidya [139, 141]. He proposed an algorithm for minimizing a convex function over an arbitrary convex set, given implicitly by a suitable separation oracle. The algorithm performs $O(TnL + n^4 L)$ operations; here T is the total cost of one call to the oracle, i.e. the algorithm performs $O(nL)$ oracle queries. Furthermore L is a suitable bound for the numbers encoded in the instance; however, L is bounded in the instance size. Using a fast matrix multiplication, algorithm from [27], the running time can be improved to $O(TnL + n^{3.38} L)$.

First we discuss what happens if one of the instances of the block problem that we are about to generate renders infeasible. Note that each invocation uses $p, q \in \mathbb{R}^M_+$, $t \in (0, 1/8]$ and $\alpha := 2\bar{p} - 1 - 2t$ and the block solver either finds $\hat{x} \in B$ such that $p^T f(\hat{x})/Y(c,t) - q^T g(\hat{x}) Y(c,t) \leq \alpha$ or correctly decides that no $x \in B$ such that $p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) \leq \alpha$ exists. If the original instance is feasible, there is an $x \in B$ such that $f(x) \leq e$ and $g(x) \geq e$; let x be chosen as such. Then for each $p, q \in \mathbb{R}^M_+$ such that $\bar{p} + \bar{q} = 1$ we have $p^T f(x) \leq \bar{p}$ and $q^T g(x) \geq \bar{q} = 1 - \bar{p}$. Then we have

$$p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) \leq \bar{p}/(1+8/3t) - (1-\bar{p})(1+8/3t) \leq 2\bar{p} - 1 - 2t$$

where for the last step note that $\bar{p} - (1-\bar{p})(1+8/3t)^2 \leq (2\bar{p} - 1 - 2t)(1+8/3t)$ is equivalent to $64/9\bar{p}t^2 \leq 2/3t + 16/9t^2$ and the latter holds since $t \in (0, 1/8]$. This means that the instance of the block problem is feasible. Furthermore, writing $Y := Y(c,t)$,

we have

$$p^T f(x)/Y - q^T g(x)Y \leq \bar{p}/(1+8/3t) - (1-\bar{p})(1+8/3t) \leq 2\bar{p} - 1 - 2t$$

where for the last step note we use the same elementary inequality as above. This means that in each case in which the block solver reports infeasibility, our algorithm terminates and reports that the initial instance is infeasible.

The initial solution $x^{(0)} \in B$ is computed as follows. First we generate M solutions $x^{[1]}, \ldots, x^{[M]}$ by calling $ABS1_c(p, q, \alpha, t)$ with $p := 1/(3M)e \in \mathbb{R}^M_+$, $q_m := 2/3$, $q_i := 0$ for each $i \in \{1, \ldots, M-1\}$ and $\alpha := -1/3 - 2t$ such that

$$\frac{1}{3MY(c,t)} \sum_{\ell=1}^{M} f_\ell(x^{[m]}) \leq \frac{2Y(c,t)}{3} g_m(x^{[m]}) - \frac{1}{3} - 2t$$

for each $m \in \{1, \ldots, M\}$ and a solution $x^{[0]}$ via $ABS1_c(p, q, \alpha, t)$ with $p := 1/Me \in \mathbb{R}^M_+$, $q := 0 \in \mathbb{R}^M_+$ and $\alpha := 1 - 2t$ such that

$$\frac{1}{MY(c,t)} \sum_{\ell=1}^{M} f_\ell(x^{[0]}) \leq 1 - 2t.$$

Then we compute a convex combination

$$x^{(0)} := \sum_{\ell=0}^{M} \mu_\ell x^{[\ell]} \in B;$$

more precisely let

$$I := \{m \in \{1, \ldots, M\} | \sum_{\ell=1}^{M} f_\ell(x^{[m]}) > 2cM\}$$

and set

$$\mu_\ell := \frac{c}{\sum_{m=1}^{M} f_m(x^{[\ell]})} \leq \frac{1}{M+1} \text{ for each } \ell \in I. \text{ Finally we set}$$

$$\mu_\ell := \frac{1 - \sum_{\ell \in I} \mu_\ell}{M + 1 - |I|} \geq \frac{1}{M+1} \text{ for each } \ell \in \{0, \ldots, M\} \setminus I.$$

Lemma 5 asserts a quality bound.

Lemma 5. *If the instance is feasible and $t \leq 1/8$, we have $\lambda(x^{(0)}) \leq 9cM/2$.*

Proof. If the instance is feasible, then each formulated instance of the block problem is also feasible. For each $m \in \{1, \dots, M\}$ use the convexity and nonnegativity of f to obtain

$$\begin{aligned} f_m(x^{(0)}) &\leq \sum_{\ell=0}^{M} \mu_\ell \sum_{m=1}^{M} f_m(x^{[\ell]}) \\ &= \mu_0 \sum_{m=1}^{M} f_m(x^{[0]}) + \sum_{\ell \in \{1,\dots,M\} \setminus I} \mu_\ell \sum_{m=1}^{M} f_m(x^{[\ell]}) + \sum_{\ell \in I} \mu_\ell \sum_{m=1}^{M} f_m(x^{[\ell]}) \\ &\leq \mu_0 M Y(c,t) + 2cM \sum_{\ell \in \{1,\dots,M\} \setminus I} \mu_\ell + \sum_{\ell \in I} c \\ &\leq 3/2cM + 3cM \\ &= 9cM/2 \end{aligned}$$

where we used $t \in (0, 1/8]$ for the last estimation. Using the nonnegativity of f, we have

$$g_m(x^{[m]}) \geq \frac{3(1/3 + 2t)}{2Y(c,t)} \geq \frac{1}{2c[1+t]}$$

for each $m \in \{1, \dots, M\}$. Since g is concave and nonnegative, we obtain

$$g_m(x^{(0)}) = g_m(\sum_{\ell=0}^{M} \mu_\ell x^{[\ell]}) \geq \sum_{\ell=0}^{M} \mu_\ell g_m(x^{[\ell]}) \geq \mu_m g_m(x^{[m]})$$

for each $m \in \{1, \dots, M\}$; we consider two cases. *Case 1:* $m \in \{0, \dots, M\} \setminus I$. Then we have $\mu_m \geq 1/(M+1)$. Here we obtain

$$\mu_m g_m(x^{[m]}) \geq \frac{1}{M+1} \frac{1}{2c[1+t]} \geq \frac{1}{2M} \frac{1}{2c[1+t]} \geq \frac{1}{9cM/2}.$$

Case 2: $m \in I$. Then we have

$$\begin{aligned} \mu_m g_m(x^{[m]}) &> \frac{\sum_{\ell=1}^{M} f_\ell(x^{[m]})}{3MY(c,t)} \frac{3c}{2Y(c,t) \sum_{\ell=1}^{M} f_\ell(x^{[m]})} \\ &= \frac{1}{2cM(1+8/3t)^2[1+t]^2} \\ &\geq \frac{1}{9cM/2}; \end{aligned}$$

furthermore in total we have $\lambda(x^0) \leq 9cM/2$ which finishes the proof. □

Now we present the algorithm itself. The remainder of Section 2.3 concerns itself with showing what happens if one of the three stopping rules is satisfied and how the step length τ is chosen and reduced in Steps 2.3.4 and 2.3.5 of the algorithm in Figure 2.2. Similar to [54, 66, 67, 69, 70, 74, 75, 79] the algorithm in Figure 2.2 uses scaling phases to successively reduce ϵ; similar to [54] an analysis without the scaling phases is possible, but yields a worse runtime bound. The main goal of each scaling phase s is to obtain $x^{(s)} \in B$ such that $\lambda(x) \leq c/(1-\epsilon_s)$ holds. Using this approach we gradually reduce ϵ_s until $\epsilon_s \leq \epsilon/2$. Then we have

$$f_m(x^s) \leq c/(1-\epsilon/2) < c(1+\epsilon)$$

and

$$g_m(x^{(s)}) \geq (1-\epsilon/2)/c > (1-\epsilon)/c;$$

thus our instance will be solved by the output of the final scaling phase. Since in each scaling phase some components of g are eliminated, more precisely we aim at $\lambda_A(x) \leq c(1+\epsilon_s) \leq c/(1-\epsilon_s)$ where $A \subseteq \{1,\ldots,M\}$ is the set of indices of the active functions. Later we show that the values for the eliminated functions are suitably bounded. The algorithm uses the threshold values

$$T_s := \begin{cases} [M^p(1-t_s/C)]/\lambda_{\{1,\ldots,M\}}(x^{(s-1)}) & : \quad s=1 \\ (1-t_s/C)/[\lambda_{\{1,\ldots,M\}}(x^{(s-1)})\epsilon_s^q] & : \quad s \geq 2 \end{cases}$$

where p, q are constants to be defined later.

We use three stopping rules for termination of each scaling phase. For Stopping Rule 1 we simply test whether $\lambda_A(x) \leq c(1+\epsilon_s) \leq c/(1-\epsilon_s)$ holds. For Stopping Rule 2 we define similar to [54, 67, 69, 79] a parameter

$$\nu(x,\hat{x}) := \frac{(p^T f - p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)}{(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} + q^T g)} \tag{2.2}$$

that depends on the current iterate x and the approximate block solution $\hat{x}$. We terminate the current scaling phase as soon as $\nu(x,\hat{x}) \leq t_s$ holds, where $t_s := \epsilon_s/8$ is an auxiliary parameter. In case of termination x meets the phase requirement, as can be seen in Lemma 6.

Lemma 6. *Let $\epsilon \in (0,1)$ and $t = \epsilon/8$. For a given $x \in B$ let p, q as in Subsection 2.2.2 and $\hat{x} \in B$ computed by ABS1$_c(p,q,\alpha,t)$ using $\alpha := 2\bar{p} - 1 - 2t \leq 2\bar{p} - 1 - t - t/C$. If*

1. Set $s := 0$, $\epsilon_0 := 1$, $t_0 := 1/8$. Compute initial solution $x^{(0)}$.
   If $\lambda(x^{(0)}) \leq c(1 + \epsilon/2)$, go to Step 3.
2. Repeat Steps 2.1 – 2.3 {scaling phase s} until $\epsilon_s \leq \epsilon/2$ or $\lambda(x^{(s)}) \leq c(1 + \epsilon/2)$.
   2.1. Set $s := s + 1$, $\epsilon_s := \epsilon_{s-1}/2$, $x := x^{(s-1)}$, and T_s as above.
   2.2. If Stopping Rule 1 is satisfied, go to Step 2.4.
   Set $A := \{m \in \{1, \ldots, M\} | g_m < T_s\}$.
   2.3. Repeat Steps 2.3.1 – 2.3.6 {coordination phase} forever.
   2.3.1. If Stopping Rule 1 or Stopping Rule 3 is satisfied go to Step 2.4.
   2.3.2. Compute θ, p and q as in Subsection 2.2.2, let $t_s := \epsilon_s/8$, $\alpha := 2\bar{p} - 1 - 2t_s$ and call $\hat{x} := ABS(p, q, \alpha, t_s)$.
   2.3.3. If Stopping Rule 2 is satisfied, go to Step 2.4.
   2.3.4. Compute suitable $\tau \in (0, 1)$ and set $x' := (1 - \tau)x + \tau\hat{x} \in B$.
   2.3.5. If $\max\{(1 - \tau)g_m + \tau\hat{g}_m | m \in A\} > T_s$ then reduce τ to τ' and set $x' := (1 - \tau')x + \tau'\hat{x}$.
   2.3.6. Set $A := A \setminus \{m \in \{1, \ldots, M\} | g_m(x') \geq T_s\}$ and $x := x'$.
   2.4. Set $x^{(s)} := x$. {end of scaling phase s}
3. Return the final iterate $x^{(s)} \in B$.

Figure 2.2.: The first approximation algorithm for the mixed problem.

$\nu(x, \hat{x}) < t$, *then* $\lambda_A(x) \leq c(1 + \epsilon) \leq c/(1 - \epsilon)$ *holds.*

Proof. The inequality $\nu(x, \hat{x}) < t$ is equivalent to

$$(p^T f - p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g) < t[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} + q^T g)].$$

Lemma 4 yields $q^T g = [1 - \bar{p} + t|A|/(2CM)]/\theta \leq [1 - \bar{p} + t/(2C)]/\theta$ as well as

$$p^T f = \theta[\bar{p} - t/(2C)].$$

We denote $X := 1 + t + t/C - 2\bar{p}$ and obtain

$$q^T \hat{g}\theta(1 - t) - X < p^T \hat{f}(1 + t)/\theta \tag{2.3}$$

by rearranging and inserting the statements above. Next we show $\theta \leq c(1 + 8t)$; for a

contradiction assume $\theta > c(1+8t)$. Since $\hat{x}$ is computed by $ABS1_c(p,q,\alpha,t)$,

$$p^T\hat{f}/Y(c,t) - q^T\hat{g}Y(c,t) \leq -X. \tag{2.4}$$

Consider three cases. *Case 1:* $X > 0$ holds. Solving (2.4) for $q^T\hat{g}$, inserting into (2.3) and solving the result for $\theta(1-t)$ yields

$$\theta(1-t) \leq [p^T\hat{f}(1+t)/\theta + X]/[p^T\hat{f}/Y(c,t)^2 + X/Y(c,t)];$$

furthermore we have $(1+t)/\theta < (1+t)/[c(1+8t)] \leq Y(c,t)/Y(c,t)^2$ which can be verified by elementary calculation. Inserting into the inequality above and factoring out $Y(c,t)$ yields $\theta < c(1+8/3t)[1+t]/(1-t) \leq c(1+8t)$ where we use $t \in (0, 5/16]$ for the last estimation; this is a contradiction. *Case 2:* $X = 0$ holds. Then (2.3) yields

$$q^T\hat{g}\theta(1-t) < p^T\hat{f}(1+t)/\theta.$$

Solving (2.3) for $q^T\hat{g}$ and inserting into (2.4) above, we obtain

$$p^T\hat{f}\theta(1-t)/Y(c,t)^2 < p^T\hat{f}(1+t)/\theta.$$

This means $p^T\hat{f} \neq 0$, since otherwise $0 < 0$ yields a contradiction. We have

$$\theta^2 < Y(c,t)^2(1+t)/(1-t) \leq c^2(1+8t)^2$$

by rearranging where the last estimation holds since $t \in (0, 1/8]$. Application of the function $\sqrt[2]{\cdot}$ to both sides yields a contradiction. *Case 3:* $X < 0$ holds. Then solving (2.4) for $p^T\hat{f}$, inserting into (2.3) and solving the result for $\theta(1-t)$ yields

$$\theta(1-t) < [(-X + q^T\hat{g}Y(c,t))Y(c,t)(1+t)/\theta]/[-X/[\theta(1-t)] + q^T\hat{g}].$$

We have $Y(c,t)(1+t) \leq c(1+8t)$ and $Y(c,t)^2(1+t)/\theta \leq c(1+8t)(1-t)$ for each $t \in (0, 1/8]$ which can be proved elementarily. Using these estimations, solving for θ and factoring out $c(1+8t)$ yields $\theta \leq c(1+8t)$, a contradiction. Thus, $\lambda_A(x) < \theta \leq c(1+8t) = c(1+\epsilon)$ holds and the claim is proved. □

The motivation behind Stopping Rule 3 is to estimate the quality of x based on the quality of $x^{(s-1)}$, the input of the current scaling phase. For this the quality is known either because of Lemma 5 or since $x^{(s-1)}$ is the output of the previous scaling phase; this

nice intuitive idea is also used in [1, 66, 67, 69, 70, 79]. In order to formulate Stopping Rule 3 we define

$$\omega_s := \begin{cases} 2/[9M(1-\epsilon_1)] & : \quad s = 1 \\ (1-\epsilon_{s-1})/(1-\epsilon_s) & : \quad s \geq 2 \end{cases}$$

which depends on the scaling phase s; for Stopping Rule 3 we terminate the current scaling phase as soon as $\lambda_A(x) \leq \omega_s \lambda(x^{(s-1)})$ is satisfied. The desired result is obtained by Lemma 7 which can be proven in a similar way as Lemma 3.4 in [67, 69].

Lemma 7. *Let $x^{(s-1)}$ be the input for and x an iterate in scaling phase s; furthermore suppose that $\lambda_A(x) \leq \omega_s \lambda(x^{(s-1)})$ holds. If*

$$\lambda(x^{(s-1)}) \leq \begin{cases} 9cM/2 & : \textit{ for } s = 1 \\ c/(1-\epsilon_{s-1}) & : \textit{ for } s \geq 2 \end{cases}$$

we have $\lambda_A(x) \leq c/(1-\epsilon_s)$.

Proof. For $s = 1$ we obtain

$$\lambda_A(x) \leq \frac{9}{2} CM \frac{1}{(9/2)M(1-\epsilon_1)} = \frac{c}{1-\epsilon_1}$$

and for $s \geq 2$ we have

$$\lambda_A(x) \leq \frac{c}{(1-\epsilon_{s-1})} \frac{1-\epsilon_{s-1}}{1-\epsilon_s} = \frac{c}{1-\epsilon_s}$$

which in total shows the claim. □

Next we describe how to choose the step length τ and eventually reduce it in Steps 2.3.4 and 2.3.5. In the following we use A to denote the set of indices *before* execution of Step 2.3.6 and A' to denote the set of indices *after* execution of Step 2.3.6.

First we present some observations which are independent from the step length that is chosen; for any step length $\gamma \in (0,1)$ we use the convexity of f and the definition of p to obtain

$$\begin{aligned} \theta - f'_m &\geq \theta - (1-\gamma) f_m - \gamma \hat{f}_m \\ &= (\theta - f_m)(1 + \gamma \frac{f_m - \hat{f}_m}{\theta - f_m}) \\ &= (\theta - f_m)[1 + \frac{2\gamma CM}{t\theta} p_m (f_m - \hat{f}_m)] \end{aligned}$$

for each $m \in \{1, \ldots, M\}$. In a similar way, since g is concave, we obtain

$$\begin{aligned} g'_m - 1/ \quad \theta &\geq (1-\gamma)g_m + \gamma\hat{g}_m - 1/\theta \\ &= (g_m - 1/\theta)(1 + \gamma\frac{\hat{g}_m - g_m}{g_m - 1/\theta}) \\ &= (g_m - 1/\theta)[1 + \frac{2\gamma CM\theta}{t} q_m(\hat{g}_m - g_m)] \end{aligned}$$

for each $m \in A$. We aim at bounding the absolute values of the last summands in the terms in square brackets by $1/2$; to this end, any step length $\gamma \in (0,1)$ will be called *feasible* if and only if

$$\max\{ \max_{m\in\{1,\ldots,M\}} |\frac{2\gamma CM}{t\theta} p_m(f_m - \hat{f}_m)|, \max_{m\in A} |\frac{2\gamma CM\theta}{t} q_m(\hat{g}_m - g_m)| \} \leq \frac{1}{2} \tag{2.5}$$

holds. By (2.5) for a feasible step length γ, any step length $\gamma' \in (0, \gamma)$ is feasible as well. If $\gamma \in (0,1)$ is a feasible step length, using $\theta - f_m > 0$ we obtain $\theta - f'_m > 0$ for each $m \in \{1, \ldots, M\}$. In a similar way we use $g_m - 1/\theta > 0$ to obtain $g'_m - 1/\theta > 0$ for each $m \in A$. Hence $\phi_t(x', A') \leq \Phi_t(\theta, x', A')$ holds in this case, which is important for the further analysis. We use

$$\tau := \frac{t^2}{4CM[(p^T f + p^T \hat{f})/\theta + (q^T\hat{g} + q^T g)\theta]} \tag{2.6}$$

to obtain the step length τ mentioned in Step 2.3.4 of the Algorithm in Figure 2.2.

Lemma 8. *The step length τ defined by* (2.6) *is feasible.*

Proof. We have $p^T f/\theta + q^T g\theta = 1 - t/(2C) + t|A|/(2CM) > 1 - t/2 \geq 3/4$ by Lemma 4, so the denominator of (2.6) is positive; since furthermore $t > 0$ holds, $\tau > 0$ ist satisfied. Since $p^T f/\theta + q^T g\theta \geq 3/4$ we obtain $\tau \leq t^2/(3CM) < 1$; therefore we have $\tau \in (0,1)$. By inserting the definition (2.6) of τ we obtain

$$\begin{aligned} |\frac{2\tau CM}{t\theta} p_m(f_m - \hat{f}_m)| &\leq \frac{2\tau CM}{t\theta}(p^T f + p^T \hat{f}) \\ &= \frac{2M}{t}\frac{t^2}{4M}\frac{(p^T f + p^T\hat{f})/\theta}{(p^T f + p^T\hat{f})/\theta + (q^T\hat{g} + q^T g)\theta} \\ &\leq \frac{t}{2} \\ &\leq \frac{1}{2} \end{aligned}$$

for each $m \in \{1, \ldots, M\}$. In a very similar way, we obtain

$$\begin{aligned} |\frac{2\tau CM\theta}{t} q_m(\hat{g}_m - g_m)| &\leq \frac{2\tau CM\theta}{t}(q^T\hat{g} + q^T g) \\ &= \frac{2M}{t}\frac{t^2}{4M}\frac{(q^T\hat{g} + q^T g)\theta}{(p^T f - p^T\hat{f})/\theta + (q^T\hat{g} + q^T g)\theta} \\ &\leq \frac{t}{2} \\ &\leq \frac{1}{2} \end{aligned}$$

for each $m \in A$, which proves that τ defined by (2.6) is feasible. □

We focus on the case that the condition in Step 2.3.5 is true. In this case we have $\hat{g}_m > T$ for at least one $m \in A$. Then we compute the uniquely determined $\tau' \in (0, 1)$ such that $\max\{(1 - \tau')g_m + \tau'\hat{g}_m\} = T$ holds, which can be done in $O(M)$ time. By construction $\tau' < \tau$ holds, hence τ' is feasible.

Theorem 9. *In each iteration of the inner loop of the algorithm in Figure 2.2, we have*

$$\phi_t(x, A) - \phi_t(x', A') > \alpha t^3/(CM)$$

where $\alpha = 1/4$ if the condition in Step 2.3.5 of the algorithm in Figure 2.2 is false and $\phi_t(x, A) - \phi_t(x', A') \geq 0$ otherwise. In the latter case, Step 2.3.6 of the algorithm in Figure 2.2 eliminates at least one index from A.

Proof. Stopping Rule 2 cannot be satisfied since otherwise {coordination phase} would have terminated; consequently $\nu(x, \hat{x}) \geq t$ holds. Similar as in the proof of Lemma 6, this can be rewritten to

$$(p^T f - p^T\hat{f})/\theta - \theta(q^T\hat{g} - q^T g) \geq t[(p^T f + p^T\hat{f})/\theta + \theta(q^T\hat{g} + q^T g)] \tag{2.7}$$

which will be used later. Let γ be the step length that is used for the interpolation which is feasible in either case. Furthermore exactly one of the three following cases occurs. *Case 1:* $\gamma = \tau$ and $A' = A$. Then by using the definition of the potential function and

the bounds from the discussion we obtain

$$\begin{aligned}
\Phi_t(\theta, x', A') &= 2\ln\theta - \frac{t}{CM}[\sum_{m=1}^{M}\ln(\theta - f'_m) + \sum_{m\in A}\ln(g'_m - 1/\theta) + (M - |A|)\ln T] \\
&\leq 2\ln\theta - \frac{t}{CM}[\sum_{m=1}^{M}\ln(\theta - f_m) + \sum_{m=1}^{M}\ln(1 + \frac{2\tau CM}{t\theta}p_m(f_m - \hat{f}_m)) \\
&\quad + \sum_{m\in A}\ln(g_m - 1/\theta) + \sum_{m\in A}\ln(1 + \frac{2\tau CM\theta}{t}q_m(\hat{g}_m - g_m)) \\
&\quad + (M - |A|)\ln T].
\end{aligned}$$

Case 2: $\gamma = \tau$ and $A' \subset A$. Since we use τ, the condition in Step 2.3.5 of the algorithm in Figure 2.2 is false, which means that $\max\{(1-\tau)g_m + \tau\hat{g}_m | m \in A\} \leq T$ holds. In particular this means that $\ln T \geq \ln((1-\tau)g_m + \tau\hat{g}_m - 1/\theta)$ holds for each $m \in A$. Similar to Case 1 we use the bound from the discussion and have

$$\begin{aligned}
\Phi_t(\theta, x', A') &= 2\ln\theta - \frac{t}{CM}[\sum_{m=1}^{M}\ln(\theta - f'_m) + \sum_{m\in A'}\ln(g'_m - 1/\theta) + (M - |A'|)\ln T] \\
&\leq 2\ln\theta - \frac{t}{CM}[\sum_{m=1}^{M}\ln(\theta - f'_m) + \sum_{m\in A}\ln((1-\tau)g_m - \tau\hat{g}_m - 1/\theta) \\
&\quad + (M - |A|)\ln T] \\
&\leq 2\ln\theta - \frac{t}{CM}[\sum_{m=1}^{M}\ln(\theta - f_m) + \sum_{m=1}^{M}\ln(1 + \frac{2\tau CM}{t\theta}p_m(f_m - \hat{f}_m)) \\
&\quad + \sum_{m\in A}\ln(g_m - 1/\theta) + \sum_{m\in A}\ln(1 + \frac{2\tau CM\theta}{t}q_m(\hat{g}_m - g_m)) \\
&\quad + (M - |A|)\ln T].
\end{aligned}$$

Note that in Case 1 and Case 2 we have obtained the same bound for $\Phi_t(\theta, x', A')$. Further rearrangement yields

$$\begin{aligned}
\Phi_t(\theta, x', A') \leq \phi_t(x, A) - \frac{t}{CM}[\sum_{m=1}^{M}\ln(1 + \frac{2\tau CM}{t\theta}p_m(f_m - \hat{f}_m)) \\
+ \sum_{m\in A}\ln(1 + \frac{2\tau CM\theta}{t}q_m(\hat{g}_m - g_m))].
\end{aligned}$$

Since τ is feasible, the inequality $\ln(1+z) \geq z - z^2$ for each $z \in [-1/2, \infty)$ yields

$$\ln(1 + \frac{2\tau CM}{t\theta} p_m(f_m - \hat{f}_m)) \geq \frac{2\tau CM}{t\theta} p_m(f_m - \hat{f}_m) - (\frac{2\tau CM}{t\theta} p_m(f_m - \hat{f}_m))^2$$

for each $m \in \{1, \ldots, M\}$ and

$$\ln(1 + \frac{2\tau CM\theta}{t} q_m(\hat{g}_m - g_m)) \geq \frac{2\tau CM\theta}{t} q_m(\hat{g}_m - g_m) - (\frac{2\tau CM\theta}{t} q_m(\hat{g}_m - g_m))^2$$

for each $m \in A$. Using this and suitable rearrangement to continue the calculations above, we obtain

$$\begin{aligned}
\phi_t(x, A) - \phi_t(x', A') &\geq \frac{2\tau}{\theta}(p^T f - p^T \hat{f}) + 2\tau\theta(q^T \hat{g} - q^T g) \\
&\quad - \frac{4CM\tau^2}{t\theta^2}(p^T f + p^T \hat{f})^2 - \frac{4CM\tau^2\theta^2}{t}(q^T \hat{g} - q^T g)^2 \\
&= 2\tau[(p^T f - p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)] \\
&\quad - \frac{4CM\tau^2}{t}[(p^T f + p^T \hat{f})^2/\theta^2 + \theta^2(q^T \hat{g} + q^T g)^2] \\
&\geq 2\tau[(p^T f - p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)] \\
&\quad - \frac{4CM\tau^2}{t}[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)]^2
\end{aligned}$$

in Case 1 and Case 2. Finally we use (2.7) and the definition of τ by (2.6) to continue the chain of inequalities from above to obtain

$$\begin{aligned}
\phi_t(x, A) - \phi_t(x', A') &\geq 2\tau t[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} + q^T g)] \\
&\quad - \frac{4CM\tau^2}{t}[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} + q^T g)]^2 \\
&= 2\tau t[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} + q^T g)] \\
&\quad - \tau t[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} + q^T g)] \\
&= \frac{t^3}{4CM}.
\end{aligned}$$

Case 3: $\gamma = \tau'$ and we have $A' \subset A$. Consequently the condition in Step 2.4.4 of the algorithm in Figure 2.2 is true, which means that $\max\{(1-\tau)g_m + \tau\hat{g}_m | m \in A\} > T$; by construction of τ', we have $(1-\tau')g_m + \tau'\hat{g}_m \leq T$ for each $m \in A$. Furthermore this

implies that

$$\ln T \geq \ln((1-\tau')g_m + \tau\hat{g}_m - 1/\theta)$$

holds for each $m \in A$. Similar to the cases above we have

$$\Phi_t(\theta, x', A') \leq \phi_t(x, A) - \frac{t}{CM}[\sum_{m=1}^{M} \ln(1 + \frac{2\tau' CM}{t\theta}) p_m(f_m - \hat{f}_m) + \sum_{m \in A} \ln(1 + \frac{2\tau' CM\theta}{t} q_m(\hat{g}_m - g_m))].$$

Using the same arguments as in Case 1 and Case 2 and $\tau' < \tau$ we obtain

$$\begin{aligned}
\phi_t(x, A) - \phi_t(x', A') &\geq 2\tau'[(p^T f - p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)] \\
&\quad - \frac{4CM\tau'^2}{t}[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)]^2 \\
&\geq 2\tau'[(p^T f - p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)] \\
&\quad - \frac{4CM\tau'\tau}{t}[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)]^2 \\
&= 2\tau'[[(p^T f - p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)] \\
&\quad - \frac{2CM\tau}{t}[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} - q^T g)]^2].
\end{aligned}$$

Inserting the inequality (2.7) and the definition of τ by (3.2) finally yields

$$\phi_t(x, A) - \phi_t(x', A') \geq \tau' t[(p^T f + p^T \hat{f})/\theta + \theta(q^T \hat{g} + q^T g)] \geq 0$$

which concludes the proof. □

2.4. Analysis of the First Algorithm

First we show that within a scaling phase that does not terminate by Stopping Rule 3, the difference of the reduced potentials of the initial solution and the iterate can not be arbitrarily large but is suitably bounded. In the statement of Lemma 10, the constants are $p = 1031$ and $q = 219$.

Lemma 10. *Let $x \in B$ be the initial iterate of scaling phase s and let $x' \in B$ arbitrary such that the pair x, x' does not satisfy Stopping Rule 3. Denote by A, A' and θ, θ' the corresponding sets of active indices and the minimizers of the potential function,*

respectively. Then

$$D_s := \phi_t(x, A) - \phi_t(x', A') \leq \begin{cases} (6 + p/(8C)\epsilon_s) \ln M + 5/8 & : \quad s = 1 \\ (2 + q)/(8C)\epsilon_s \ln \epsilon_s^{-1} + 6\epsilon_s & : \quad s \geq 2 \end{cases}$$

holds, where p and q are constants.

Proof. We denote $\lambda := \lambda_A(x)$, $\lambda' := \lambda_{A'}(x')$, and $t = t_s$. By using the bounds from Lemma 3 and rearranging we have

$$D_s \leq (2 - t/C) \ln \frac{\theta}{\theta'} + \frac{t|A|}{CM} \ln(\theta T) + \frac{t(M + |A|)}{CM} \ln \frac{2CM}{t(M + |A|)}$$

where $T = T_s$ as defined before; we aim at obtaining bounds for the three summands above. By Lemma 1 we have $\theta \leq \lambda/(1 - t/C)$ and $\theta' > \lambda'$. We obtain

$$\ln \frac{\theta}{\theta'} < \ln \frac{\lambda}{\lambda'(1 - t/C)} \leq \ln \frac{\lambda}{\lambda'} + \ln(1 + 2t/C) < \ln \frac{1}{\omega_s} + \frac{2t}{C}$$

where we use $t \in (0, 1/2]$ and the fact that the pair x, x' does not satisfy Stopping Rule 3. In total, the first summand can be bounded by $2[\ln(1/\omega_s) + (2t)/C]$. For the second summand we use again $\theta \leq \lambda/(1 - t/C)$ and the definition of T_s to obtain

$$\ln(\theta T) \leq \begin{cases} p \ln M & : \quad s = 1 \\ q \ln(1/\epsilon_s) & : \quad s \geq 2. \end{cases}$$

In total we can bound the second summand by

$$\frac{t|A|}{CM} \ln(\theta T) \leq \begin{cases} (tp)/C \ln M & : \quad s = 1 \\ (tq)/C \ln(1/\epsilon_s) & : \quad s \geq 2. \end{cases}$$

The third summand can be rewritten to

$$\frac{t(M + |A|)}{CM} \ln 2 + \frac{t(M + |A|)}{CM} \ln(\frac{CM}{t(M + |A|)});$$

note that the function $(0, 1/\mathrm{e}] \to \mathbb{R}, z \mapsto z \ln(1/z) = -z \ln z$ is monotonically increasing. Furthermore we have $0 < t(M + |A|)/[CM] \leq 2t/C \leq 1/\mathrm{e}$ for each $t \in (0, 1/8]$. This

means that the third summand is bounded by

$$\frac{t(M+|A|)}{CM}\ln\frac{2CM}{t(M+|A|)} \leq \frac{2t}{C}\ln 2 + \frac{2t}{C}\ln\frac{C}{2t} = \frac{2t}{C}\ln\frac{C}{t}.$$

In total, using $\ln[(1-\epsilon_s)/(1-2\epsilon_2)] \leq \ln(1+2\epsilon_2) \leq 2\epsilon_2$ and $t_s = \epsilon_s/8$ for $s \geq 2$ as well as $t \in (0, 1/8]$, we have

$$D_s \leq \begin{cases} (6 + pt_s/C)\ln M + 4t_s/C + 2t_s/C\ln(C/t_s) & : \quad s = 1 \\ [5 + 1/(4C)\ln(8C)]\epsilon_s + (2+q)/(8C)\epsilon_s\ln(1/\epsilon_s) & : \quad s \geq 2. \end{cases}$$

For $s = 1$ we use $t_1 = \epsilon_1/8$ as well as $t_1/C = 1/(16C) \leq 1/16 \leq 1/\mathrm{e}$ and again the behaviour of $(0, 1/\mathrm{e}] \to \mathbb{R}, z \mapsto z\ln(1/z)$ to obtain

$$\begin{aligned} D_s &\leq (6 + \frac{pt_1}{C})\ln M + \frac{4t_1}{C} + \frac{2t_1}{C}\ln\frac{C}{t_1} \\ &= (6 + \frac{p}{8C}\epsilon_s)\ln M + \frac{4t_1}{C} + 2(\frac{t_1}{C}\ln\frac{C}{t_1}) \\ &\leq (6 + \frac{p}{8C}\epsilon_s)\ln M + \frac{1}{4} + 2(\frac{1}{16}\ln 16) \\ &\leq (6 + \frac{p}{8C}\epsilon_s)\ln M + \frac{1}{4} + 3\frac{2}{16} \\ &= (6 + \frac{p}{8C}\epsilon_s)\ln M + \frac{5}{8} \\ &\leq (6 + \frac{p}{8C}\epsilon_s)\ln M + 1. \end{aligned}$$

For $s \geq 2$ we use $C = 8$ and have

$$\begin{aligned} D_s &\leq [5 + \frac{1}{4C}\ln(8C)]\epsilon_s + \frac{2+q}{8C}\epsilon_s\ln(1/\epsilon_s) \\ &= \frac{2+q}{8C}\epsilon_s\ln(1/\epsilon_s) + [5 + \frac{1}{4C}\ln(8C)]\epsilon_s \\ &= \frac{2+q}{8C}\epsilon_s\ln(1/\epsilon_s) + [5 + \frac{1}{32}\ln 64]\epsilon_s \\ &\leq \frac{2+q}{8C}\epsilon_s\ln(1/\epsilon_s) + 6\epsilon_s \end{aligned}$$

which proves the claim. □

Lemma 10 states that a scaling phase that is not terminated by Stopping Rule 3 or Stopping Rule 1 must be terminated by Stopping Rule 2 since ϕ_t decreases by at least $\alpha t^3/(CM)$ in each iteration or at least does not increase; in the latter case Step 2.3.6 of

the algorithm in Figure 2.2 eliminates indices, which is possible at most M times.

Lemma 11. *Let N_s denote the number of iterations in scaling phase s. Then*

$$N_s \leq \begin{cases} [8MC/(\alpha t_s^3) + pM/(\alpha t_s^2)] \ln M & : \quad s = 1 \\ [51MC/(\alpha t_s^2) + qM/(\alpha t_s^2)] \ln \epsilon_s^{-1} & : \quad s \geq 2 \end{cases}$$

holds.

Proof. Consider scaling phase s. Let x, x' denote the initial solution and the solution after $N_s' := N_s - 1$ iterations. For any two consecutive iterations without index elimination the reduced potential is decreased by at least $\alpha t^3/(CM)$ where $t := t_s$; furthermore there are at most M iterations in which indices for covering constraints are eliminated.

In total, we have $D_s \geq \phi_t(x, A) - \phi_t(x', A') \geq \alpha t^3/(CM)(N_s' - M)$; we rearrange the resulting expression to obtain

$$N_s \leq \frac{MC}{\alpha t^3} D_s + M + 1.$$

We consider the following two cases. *Case 1:* We have $s = 1$. Then we obtain

$$\begin{aligned} N_s &\leq \frac{MC}{\alpha t_1^3} D_s + M + 1 \\ &\leq \frac{MC}{\alpha t_1^3} D_s + 2M \\ &\leq \frac{MC}{\alpha t_1^3} [(6 + \frac{pt_1}{C}) \ln M + 1] + 2M \\ &= \frac{MC}{\alpha t_1^3} (6 + \frac{pt_1}{C}) \ln M + \frac{MC}{\alpha t_1^3} + \frac{2M\alpha t_1^3}{\alpha t_1^3} \\ &\leq \frac{MC}{\alpha t_1^3} (6 + \frac{pt_1}{C}) \ln M + \frac{MC}{\alpha t_1^3} + \frac{MC}{\alpha t_1^3} \\ &\leq \frac{MC}{\alpha t_1^3} (8 + \frac{pt_1}{C}) \ln M \\ &= [\frac{8MC}{\alpha t_1^3} + \frac{pM}{\alpha t_1^2}] \ln M \end{aligned}$$

where we used $M \geq 2$, $\alpha = 1/4$ and $t_1 = 1/16$. *Case 2:* $s \geq 2$ holds. Then we have

$$\begin{aligned} N_s &\leq \frac{MC}{\alpha t_s^3} D_s + M + 1 \\ &\leq \frac{MC}{\alpha t_s^3} D_s + 2M \\ &\leq \frac{MC}{\alpha t_s^3} [\frac{(2+q)}{C} t_s \ln \epsilon_s^{-1} + 48 t_s] + 2M \\ &= \frac{2M}{\alpha t_s^2} \ln \epsilon_s^{-1} + \frac{qM}{\alpha t_s^2} \ln \epsilon_s^{-1} + \frac{48MC}{\alpha t_s^2} + 2M \\ &\leq \frac{qM}{\alpha t_s^2} \ln \epsilon_s^{-1} + \frac{2MC}{\alpha t_s^2} \ln \epsilon_s^{-1} + \frac{48MC}{\alpha t_s^2} \ln \epsilon_s^{-1} + 2MC \ln \epsilon_s^{-1} \\ &\leq \frac{qM}{\alpha t_s^2} \ln \epsilon_s^{-1} + \frac{51MC}{\alpha t_s^2} \ln \epsilon_s^{-1} \end{aligned}$$

finishing the proof. □

Before studying the eliminated indices more closely, we present the runtime complexity the algorithm in Figure 2.2.

Theorem 12. *The total number of iterations of the algorithm in Figure 2.2 is bounded by*

$$O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1})).$$

Proof. Clearly we have $N_1 \in O(M \ln M)$ and $N_s \in O(M \epsilon_s^{-2} \ln \epsilon_s^{-1})$ for $s \geq 2$ by Lemma 11. Summing over all scaling phases the total number of iterations is

$$O(M(\ln M + \ln \epsilon^{-1} \sum_{s=1}^{\lceil \log \epsilon^{-1} \rceil} 2^{2s})).$$

Since the summation term above is bounded by $O(\epsilon^{-2})$, the claim follows. □

Now we study the eliminated functions; the goal is to show that the values $g_m(x^{(s)})$ for $m \in \{1, \ldots, M\} \setminus A$ at the end of phase s are sufficiently large. The main idea is similar to [67, 69]; since the covering functions g are concave and nonnegative on B, we have $g_m(x') \geq (1-\tau) g_m(x) + \tau g_m(\hat{x}) \geq (1-\tau) g_m(x)$ for any two consecutive iterates x, x' in a scaling phase. Furthermore we have

$$(p^T f + p^T \hat{f})/\theta + (q^T \hat{g} + q^T g)\theta \geq 1 - t/(2C) + t|A|/(2CM) \geq 1/2$$

by Lemma 4, hence in any case we obtain $\tau' \leq \tau \leq t^2/(2CM) =: \tau_s$. This means that in each interpolation step g_m is scaled down by a certain factor which is at least $1 - t^2/(2CM)$, however.

Lemma 13. *Let $x^{(s)}$ be the output of scaling phase s. If*

$$\lambda(x^{(s-1)}) \leq \begin{cases} 9cM/2 & : \text{for } s = 1 \\ c/(1-\epsilon_{s-1}) & : \text{for } s \geq 2 \end{cases}$$

then $g_m(x^{(s)}) \geq (1-\epsilon_s)/c$ for each $m \in \{1, \ldots, M\}$.

Proof. We consider only the eliminated components $m \in \{1, \ldots, M\} \setminus A$; for the other ones the lemmas above imply the claim. In the worst case an index $m \in \{1, \ldots, M\}$ is eliminated at the beginning of the scaling phase, which means $g_m(x^{(s-1)}) \geq T_s$. In this case we have $g_m(x^{(s)}) \geq (1-\tau_s)^{N_s} T_s$. We aim at proving the stronger inequality $g_m(x^{(s)}) \geq (1-\tau_s)^{N_s} T_s \geq 1/c \geq (1-\epsilon_s)/c$. In the following we use the inequality $(1-z)^\ell \geq (1-\ell z)$ for each $z \in (0,1)$ and $\ell \in \mathbb{N} \cup \{0\}$ which can be proved by induction on ℓ or found in [99], page 45. First we study phase $s = 1$; with the help of Lemma 11 we obtain

$$\begin{aligned} (1-\tau_1)^{N_1} &\geq (1 - \frac{t_1^2}{2CM})^{(\frac{8MC}{\alpha t_1^3} + \frac{pM}{\alpha t_1^2}) \ln M} \\ &\geq (1 - \frac{MC}{t_1^2}\frac{t_1^2}{2CM})^{(\frac{8}{\alpha t_1} + \frac{p}{\alpha C}) \ln M} \\ &= (1/2)^{(\frac{8}{\alpha t_1} + \frac{p}{\alpha C}) \ln M} \\ &\geq (1/M)^{(\frac{8}{\alpha t_1} + \frac{p}{\alpha C})} \end{aligned}$$

where we used $(1/2)^{\ln M} = 1/(2^{\ln M}) \geq 1/(2^{\log M}) = 1/M$ for the last inequality. This means that it is sufficient to show that

$$(1/M)^{(\frac{8}{\alpha t_1} + \frac{p}{\alpha C})} T_1 = (1/M)^{(\frac{8}{\alpha t_1} + \frac{p}{\alpha C})} M^p \frac{1 - t_1/C}{\lambda(x^{(0)})} \geq 1/c$$

holds. We use $\lambda(x^{(0)}) \leq 9cM/2$ which holds by Lemma 5 and $1 - t_1/C \geq 127/128$; inserting these bounds yields that the inequality above is implied by

$$(1/M)^{(\frac{8}{\alpha t_1} + \frac{p}{\alpha C})} M^p \frac{127}{576cM/2} \geq 1/c$$

which can be rearranged to

$$M^{p-1-\frac{8}{\alpha t_1}-\frac{p}{\alpha C}} \geq 576/127.$$

Since $M \geq 2$, this is satisfied if $p - 1 - 8/(\alpha t_1) - p/(\alpha C) \geq 11/5$ holds. Using $\alpha = 1/4$, $C = 8$ and $t_1 = 1/16$, elementary calculation yields that this can be satisfied by choosing $p = 1031$. Now let $s \geq 2$. Similar to the analysis above we have

$$\begin{aligned}
(1-\tau_s)^{N_s} &\geq (1 - \frac{t_s^2}{2CM})^{(\frac{51CM}{\alpha t_s^2}+\frac{qM}{\alpha t_s^2})\ln \epsilon_s^{-1}} \\
&\geq (1 - \frac{CM}{t_s^2}\frac{t_s^2}{2CM})^{(\frac{51}{\alpha}+\frac{q}{\alpha C})\ln \epsilon_s^{-1}} \\
&= (1/2)^{(\frac{51}{\alpha}+\frac{q}{\alpha C})\ln \epsilon_s^{-1}} \\
&\geq \epsilon_s^{(\frac{51}{\alpha}+\frac{q}{\alpha C})}
\end{aligned}$$

where we used $(1/2)^{\ln \epsilon_s^{-1}} = 1/(2^{\ln \epsilon_s^{-1}}) \geq 1/(2^{\log \epsilon_s^{-1}}) = 1/\epsilon_s^{-1} = \epsilon_s$ for the last estimation. Here it is sufficient to show that

$$\epsilon_s^{(\frac{51}{\alpha}+\frac{q}{\alpha C})} T_s = \epsilon_s^{(\frac{51}{\alpha}+\frac{q}{\alpha C})} \epsilon_s^{-q} \frac{1 - t_s/C}{\lambda(x^{(s-1)})} \geq 1/c$$

holds. Parallel to the argumentation before we use $\lambda(x^{(s-1)}) \leq c/(1-\epsilon_{s-1}) \leq 2c$ and $1 - t_s/C \geq 2/3$ to obtain that the inequality above is implied by

$$\epsilon_s^{(\frac{51}{\alpha}+\frac{q}{\alpha C})} \epsilon_s^{-q} \frac{1}{3c} \geq 1/c$$

which can be rearranged to

$$(\epsilon_s^{-1})^{q-\frac{51}{\alpha}-\frac{q}{\alpha C}} \geq 3.$$

We have $\epsilon_s \leq 1/4$, which implies $\epsilon_s^{-1} \geq 4$; hence $q - 51/\alpha - q/(2C) \geq 4/5$ is sufficient. Using $\alpha = 1/4$ and $C = 8$, this is satisfied by $q = 219$. □

2.5. The Second Algorithm

In this section we present an algorithm that is based on the same approach as the algorithm in Section 2.3. The main differences are a different block problem and storing a part of the history of iterates; this second algorithm needs only $O(M\epsilon^{-2}\ln(M\epsilon^{-1}))$ iterations, a bound that is also independent from the approximation ratio of the block

solver and the input data; it is a generalization of the algorithm presented in [70].

More precisely, we assume that there is a *block solver* which can query B by a approximate feasibility oracle of the form

$$\text{find } \hat{x} \in B \text{ such that } p^T f(x) \leq c(1+t) \sum_{m=1}^{M} p_m$$

and

$$q^T g(x) \geq \frac{1}{c(t+1)} \sum_{m=1}^{M} q_m \qquad (ABS2_c(p,q,t))$$

or decide that there is no such $\hat{x} \in B$ such that

$$p^T f(x) \leq \sum_{m=1}^{M} p_m \text{ and } q^T g(x) \geq \sum_{m=1}^{M} q_m$$

where $c \in \mathbb{R}_+$ is a constant and $\epsilon \in (0,1)$ is an additional accuracy parameter. Note that, in contrast to [54, 69, 79], this block solver needs to solve only a feasibility problem and not an optimization problem.

The remainder of this chapter is organized as follows. First we discuss the second algorithm to approximately solve the mixed packing and covering problem; afterwards in Section 2.6 we present a subsequent analysis; note that the entire construction is based on basically the same foundations as in Section 2.2. Finally we conclude with some remarks in Section 2.8.

For the second approximation algorithm we use the same modified logarithmic potential function as defined in Subsection 2.2.1, where we have $C = 1$, which simplifies the analysis in some details. Furthermore, the price vectors are defined exactly as in Subsection 2.2.2.

Similar as before we describe how to obtain a suitable initial solution; furthermore we will present the algorithm itself. Then we will prove that the stopping rules used here are correct in the sense that each of them asserts a certain property we want the iterate x to have before each scaling phase halts.

First we clarify what happens if one of the instances of the block problem that we are about to generate renders infeasible. If there is a solution $x \in B$ to our original instance, $f(x) \leq e$ and $g(x) \geq e$ holds. By elementary calculation it is easily seen that in this case x is also a feasible solution that may be returned by $ABS2_c(p,q,t)$ where c, p, q and t are arbitrary feasible parameters. Consequently, if one such instance is infeasible, we conclude that the original instance is infeasible as well and terminate. For

ease of presentation we suppose that in the following only feasible instances of the block problem are generated.

The initial solution for the algorithm in Figure 2.3 is computed as follows. For each $m \in \{1, \ldots, M\}$ we call the approximate block solver $ABS2_c(p, q, 1/2)$ where $p := 1/Me \in \mathbb{R}^m_+$ and $q := e_m \in \mathbb{R}^m_+$ to obtain a block solution $\hat{x}^{[m]} \in B$ such that

$$p^T f(\hat{x}^{[m]}) \leq c(1+t) = \frac{3c}{2} \quad \text{and} \quad q^T g(\hat{x}^{[m]}) \geq \frac{1}{c(1+t)} = \frac{2}{3c}$$

holds. Then we use the convex combination

$$x^{(0)} := \frac{1}{M} \sum_{m=1}^{M} \hat{x}^{[m]} \in B$$

as our initial solution, for which Lemma 14 establishes a quality bound.

Lemma 14. *The inequality $\lambda(x^{(0)}) \leq 3cM/2$ holds.*

Proof. By construction of $x^{(0)} \in B$ we have

$$\frac{1}{M} \sum_{\ell=1}^{M} f_\ell(\hat{x}^{[m]}) \leq \frac{3c}{2} \quad \text{and} \quad g_m(\hat{x}^{[m]}) \geq \frac{2}{3c} \text{ for each } m \in \{1, \ldots, M\}.$$

Using the convexity and nonnegativity of f_m, we obtain

$$f_m(x^{(0)}) = f_m(\frac{1}{M} \sum_{\ell=1}^{M} \hat{x}^{[\ell]}) \leq \frac{1}{M} \sum_{\ell=1}^{M} f_m(\hat{x}^{[\ell]}) \leq \frac{1}{M} \sum_{m=1}^{M} \sum_{\ell=1}^{M} f_m(\hat{x}^{[\ell]}) \leq \frac{3cM}{2}$$

and furthermore, since g_m is convex and nonnegative, we have

$$g_m(x^{(0)}) = g_m(\frac{1}{M} \sum_{\ell=1}^{M} \hat{x}^{[\ell]}) \geq \frac{1}{M} \sum_{\ell=1}^{M} g_m(\hat{x}^{[\ell]}) \geq \frac{1}{M} g_m(\hat{x}^{[\ell]}) \geq \frac{2}{3cM}$$

for each $m \in \{1, \ldots, M\}$. Suitable rearranging yields $\lambda(x^{(0)}) \leq 3cM/2$ and completes the proof. □

Now we present the algorithm itself. The remainder of this section concerns itself with showing what happens if one of the three stopping rules is satisfied. In the sequel we explain how to calculate the convex combination in Step 2.5 and how to choose the step length $\tau \in (0, 1)$ in Step 2.4.3 and Step 2.4.4 of the algorithm in Figure 2.3. Then,

1. Compute the initial solution $x^{(0)} \in B$, set $s := 0$ and $\epsilon_0 := 1$. If $\lambda(x^{(s)}) \leq c(1+\epsilon)$, go to Step 3.

2. Repeat Steps 2.1 – 2.5 {scaling phase} until $\epsilon_s \leq \epsilon/2$ or $\lambda(x^{(s)}) \leq c(1+\epsilon)$ holds.
   2.1. Set $s := s+1$, $\epsilon_s := \epsilon_{s-1}/2$, $x := x^{(s-1)}$, and $T_s := 2112M^3\epsilon_s^{-2}/\lambda(x)$.
   2.2. Set $A := \{m \in \{1,\ldots,M\} | g_m < T_s\}$ and $k := 0$. If $A \neq \{1,\ldots,M\}$ then set $k := k+1$ and $x_k := x$.
   2.3. If Stopping Rule 1 is satisfied then set $y := x$ and go to Step 2.5.
   2.4. Repeat Steps 2.4.1 – 2.4.7 {coordination phase} forever.
      2.4.0. If Stopping Rule 3 is satisfied, set $y := x$ and go to Step 2.5.
      2.4.1. Compute θ, p and q, let $t_s := \epsilon_s/32$ and call $\hat{x} := ABS2(p, q, t_s)$.
      2.4.2. If Stopping Rule 2 is satisfied, set $y := x$ and go to Step 2.5.
      2.4.3. Compute suitable $\tau \in (0,1)$ and set $x' := (1-\tau)x + \tau\hat{x} \in B$.
      2.4.4. If $\max\{(1-\tau)g_m + \tau\hat{g}_m | m \in A\} > T_s$ then reduce τ to τ' and set $x' := (1-\tau')x + \tau'\hat{x}$.
      2.4.5. Set $A' := A \setminus \{m \in \{1,\ldots,M\} | g_m(x') \geq T_s\}$ and $x := x'$.
      2.4.6. If $A' \subset A$ then set $k := k+1$, $x_k = x'$, and $A := A'$.
   2.5. Compute {mixing phase} $x^{(s)} \in B$ as a suitable convex combination of $x_1,\ldots,x_k$ and y. Set $x := x^{(s)}$.

3. Return the final iterate $x \in B$.

Figure 2.3.: The second approximation algorithm for the mixed problem.

these results will yield the partial correctness of the second algorithm. Afterwards we will show that the algorithm acutally terminates, which means that the loop beginning in Step 2.4 will halt in each scaling phase.

The algorithm in Figure 2.3 uses a scaling phase implementation to successively reduce the error parameter; note that without the scaling phases similar to [54] an analysis is possible, but yields a worse runtime bound. The main goal of each scaling phase s is to obtain an iterate $x^{(s)} \in B$ such that $\lambda(x) \leq c/(1-\epsilon_s)$ holds. Using this approach, we gradually reduce ϵ_s until $\epsilon_s \leq \epsilon/2$; in this case we have

$$f_m(x^{(s)}) \leq \frac{c}{1-\epsilon/2} < c(1+\epsilon) \quad \text{and} \quad g_m(x^{(s)}) \geq \frac{1-\epsilon/2}{c} > \frac{1-\epsilon}{c}$$

because $\epsilon \in (0,1)$ and our instance of $(MPC_{c,\epsilon})$ will be solved by the output of the

final scaling phase. Since in each scaling phase some components of g are eliminated, we will aim at $\lambda_A(x) \leq c(1+\epsilon_s/4)$ where $A \subseteq \{1,\ldots,M\}$ is the set of indices of the noneliminated functions. We will show that a suitable convex combination of some of the iterates in scaling phase s yields an output $x^{(s)}$ for which the inequality

$$\lambda(x^{(s)}) \leq c/(1-\epsilon_s)$$

holds.

We use three stopping rules for termination of each scaling phase before the convex combination mentioned above is computed. For Stopping Rule 1 we simply test whether $\lambda_A(x) \leq c(1+\epsilon_s/4)$ is satisfied. For Stopping Rule 2 similar to [54, 79], we define the parameter $\nu(x,\hat{x})$ exactly as via 2.1 in Section 2.3; we terminate the current scaling phase as soon as $\nu(x,\hat{x}) \leq t_s$ holds, where $t_s := \epsilon_s/32$ is an auxiliary accuracy parameter. In case of termination we have reached our goal, as can be seen in Lemma 15; note the similarity to Lemma 6.

Lemma 15. *Let $\epsilon_s \in (0,1)$ and $t_s := \epsilon_2/32$. Let $x \in B$ and $p := p(x)$ and $q := q(x)$, furthermore let $\hat{x}$ be computed by ABS2$_c(p,q,t)$. If $\nu(x,\hat{x}) \leq t_s$, then $\lambda_A(x) \leq c/(1+\epsilon/4)$ holds.*

Proof. We write $t := t_s$; by Lemma 4 and since $\hat{x}$ is computed by ABS2$_c(p,q,t)$ we have

$$p^T f = \theta(\bar{p} - t/2), \quad q^T g \leq (1-\bar{p}+t/2)/\theta, \quad p^T \hat{f} \leq c(1+t)\bar{p}, \quad \text{and} \quad q^T \hat{g} \geq \frac{1-\bar{p}}{c(1+t)}.$$

Solving the inequality $\nu(x,\hat{x}) \leq t$ for $\theta(1-t)q^T\hat{g}$ and inserting the inequalities above, we obtain

$$\theta\frac{(1-\bar{p})(1-t)}{c(1+t)} \leq (1+t)(1-\bar{p}) + t + \bar{p}(t + c\frac{t(2+t)+1}{\theta} - 1).$$

We show $\theta \leq c(1+8t)$ by aiming at a contradiction and assume that $\theta > c(1+8t)$ holds. Inserting yields

$$\frac{\theta(1-t)(1-\bar{p})}{c(1+t)} < (1+t)(1-\bar{p}) + t + t\bar{p}\frac{9t-5}{1+8t}.$$

For each $\alpha \in (0,1)$ the inequalities $(9\alpha-5)/(1+8\alpha) \leq -1$ and $\alpha < 4/17$ are equivalent; since we have even $t \leq 1/32$, we obtain

$$\frac{\theta(1-t)(1-\bar{p})}{c(1+t)} < (1+t)(1-\bar{p}) + t - t\bar{p} = (1-\bar{p})(1+t) + (1-\bar{p})t.$$

This means $\bar{p} \neq 1$, because otherwise $0 < 0$ yields a contradiction; so we have $\bar{p} \in (0,1)$ and obtain

$$\theta \leq c\frac{(1+2t)(1+t)}{(1-t)} < c(1+8t)$$

by rearranging, where $t \in (0, 1/2)$ yields the last inequality. Thus, we have obtained a contradiction and $\theta \leq c(1+8t) = c(1+\epsilon_s/4)$ holds. Finally $\lambda_A(x) < \theta \leq c(1+\epsilon_s/4)$ concludes the proof. □

The intuition behind Stopping Rule 3 of the algorithm in Figure 2.3 is to estimate the quality of the current iterate based on the quality of the iterate and the quality of $x^{(s-1)}$ which is the input of the current scaling phase. The quality of $x^{(s-1)}$ is known either because of Lemma 14 or by the fact that $x^{(s-1)}$ is the output of the previous scaling phase; this nice idea, as used for the first approximation algorithm, is also used in [69, 70, 79]. More precisely, for Stopping Rule 3, we define a parameter

$$\omega_s := \begin{cases} 2/(3M(1-\epsilon_s/4)) & : \quad s = 1 \\ (1-\epsilon_{s-1})/(1-\epsilon_s/4) & : \quad s \geq 2 \end{cases}$$

for each $s \in \mathbb{N} \setminus \{0\}$ depending on the scaling phase. For Stopping Rule 3 we terminate the current scaling phase as soon as $\lambda_A(x) \leq \omega_s \lambda(x^{(s-1)})$ is satisfied; Lemma 16 yields the desired result.

Lemma 16. *Let $x^{(s-1)} \in B$ be the input for and $x \in B$ an iterate in a scaling phase $s \in \mathbb{N} \setminus \{0\}$; furthermore suppose that $\lambda_A(x) \leq \omega_s \lambda(x^{(s-1)})$ holds. If*

$$\lambda(x^{(s-1)}) \leq \begin{cases} \frac{3Mc}{2} & : \quad s = 1 \\ \frac{c}{1-\epsilon_s} & : \quad s \geq 2 \end{cases}$$

is satisfied, the inequality $\lambda_A(x) \leq c/(1-\epsilon_s/4)$ holds.

Proof. For $s = 1$ we have

$$\lambda_A(x) \leq \omega_1 \lambda(x^{(0)}) \leq \frac{2}{3M(1-\epsilon_1/4)} \frac{3Mc}{2} = \frac{c}{1-\epsilon_s/4}$$

and for $s \geq 2$ we have

$$\lambda_A(x) \leq \omega_s \lambda(x^{(s-1)}) \leq \frac{1-\epsilon_{s-1}}{1-\epsilon_s/4} \frac{c}{(1-\epsilon_{s-1})} = \frac{c}{1-\epsilon_s/4}$$

which shows the claim. □

Stopping Rule 1, Lemma 15 and Lemma 16 imply that $\lambda_A(x) \leq c/(1-\epsilon/4)$ holds for the current iterate x whenever a scaling phase is terminated.

Now we describe and analyze the implementation of the inner loop in more detail. To this end, let $A \subseteq \{1,\ldots,M\}$ denote the set of active indices at the beginning of an iteration of {coordination phase} in Step 2.4.1 and let $A' \subseteq \{1,\ldots,M\}$ denote the set of active indices computed in Step 2.4.6; notice that by construction $A' \subseteq A$ holds, which will be used later.

First we present some observations which are independent from the step length that is chosen. With the same calculations as in Section 2.3 using $C = 1$, for any step length $\gamma \in (0,1)$ we use the convexity of f to obtain

$$\theta - f'_m \geq (\theta - f_m)[1 + \frac{2\gamma M}{t\theta} p_m(f_m - \hat{f}_m)]$$

for each $m \in \{1,\ldots,M\}$. In a similar way, since g is concave, we obtain

$$g'_m - 1/\theta \geq (g_m - 1/\theta)[1 + \frac{2\gamma M\theta}{t} q_m(\hat{g}_m - g_m)]$$

for each $m \in A$. Exactly as in Section 2.3 we aim at bounding the absolute values of the last summands in the terms in square brackets by $1/2$; to this end, any arbitrary step length $\gamma \in (0,1)$ again will be called *feasible* if and only if condition (2.5) is satisfied; note that for the second approximation algorithm we use $C = 1$. We remark that by (2.5) for a feasible step length τ, any step length $\tau' \in (0,\tau)$ is feasible as well. If $\gamma \in (0,1)$ is a feasible step length, using $\theta - f_m > 0$ we obtain $\theta - f'_m > 0$ for each $m \in \{1,\ldots,M\}$. In a similar way we use $g_m - 1/\theta > 0$ to obtain $g'_m - 1/\theta$ for each $m \in A$. Hence $\phi_t(x', A') \leq \Phi_t(\theta, x', A')$ holds, which is important for the further analysis.

The step length τ mentioned in Step 2.4.3 is again obtained by using the definition (2.6), where $C = 1$. Since t and the denominator used above are positive, $\tau > 0$ is satisfied. Using Lemma 4 results in $p^T f/\theta + q^T g\theta = 1 - t/2 + t|A|/(2M) > 1 - t/2 \geq 1/2$, which means that $\tau \leq t^2/(2M) < 1$ holds; therefore we have $\tau \in (0,1)$. Note that using $C = 1$, the step length τ is feasible by Lemma 8. Next we describe Step 2.2.4 in detail. If the condition in Step 2.2.4 is satisfied, there is an $m \in A$ such that $\hat{g}_m > T$ holds. In this case we compute the uniquely determined $\tau' \in (0,1)$ such that $\max\{(1-\tau)g_m + \tau\hat{g}_m | m \in A\} = T$ holds; notice that this can be done in $O(M)$ time. Furthermore in this calculation τ' will be assigned a value smaller than τ, hence

τ' is feasible.

In the following we prove that if in Step 2.2.4 the condition is *not* satisfied, we have a guaranteed minimum increase in the reduced potential of the iterate. Furthermore we show that if in Step 2.2.4 the condition *is* satisfied, the reduced potential of the iterate does not decrease; however, since in this case by construction $A' \subset A$ holds, this can happen at most M times in each scaling phase. Note that the statement and the proof of the following result are parallel to Lemma 9.

Theorem 17. *In each iteration of the inner loop we have* $\phi_t(x, A) - \phi_t(x', A') > t^3/(4M)$ *if the condition in Step 2.2.4 is not satisfied and* $\phi_t(x, A) - \phi_t(x', A') \geq 0$ *if the condition in Step 2.2.4 is satisfied.*

Proof. Stopping Rule 2 cannot be satisfied, since otherwise the inner loop {coordination phase} would have terminated; consequently $\nu(x, \hat{x}) > t$ holds. Similar as in the proof of Lemma 15 and as mentioned in (2.7), this can be rewritten to

$$[(p^T f - p^T \hat{f})/\theta + (q^T \hat{g} - q^T g)\theta] > t[(p^T f + p^T \hat{f})/\theta + (q^T \hat{g} + q^T g)\theta]$$

which will be used later. Let γ be the step length that is used for the interpolation which is feasible in either case.

Furthermore exactly three cases can occur. *Case 1:* We use τ and we have $A' = A$ after the interpolation. Then by using the definition of the reduced potential, inserting the bounds from above and using $C = 1$ and the same calculation as in the proof of Lemma 9, we obtain

$$\begin{aligned}\Phi_t(\theta, x', A') \;\leq\; & 2\ln\theta - \frac{t}{M}[\sum_{m=1}^{M} \ln(\theta - f_m) + \sum_{m=1}^{M} \ln(1 + \frac{2\tau M}{t\theta} p_m(f_m - \hat{f}_m)) \\ & + \sum_{m \in A} \ln(g_m - 1/\theta) \\ & + \sum_{m \in A} \ln(1 + \frac{2\tau M\theta}{t} q_m(\hat{g}_m - g_m)) + (M - |A|)\ln T].\end{aligned}$$

Case 2: We use τ and we have $A' \subset A$ after the interpolation. Consequently, the condition in Step 2.4.4 is not satisfied, which means that

$$\max\{(1-\tau)g_m + \tau\hat{g}_m | m \in A\} \leq T$$

holds. In particular this means that $\ln T \geq \ln((1-\tau)g_m) + \tau\hat{g}_m - 1/\theta)$ holds for each

$m \in A$. Furthermore, since τ is feasible, we have $\theta - f'_m > 0$ for each $m \in \{1, \ldots, M\}$ and $g'_m - 1 > 1/\theta$ for each $m \in A$. Similar to Case 1, we use the concavity of g, the bounds from above, basically the same calculation as in the proof of Lemma 9, and $C = 1$ to obtain

$$\begin{aligned}\Phi_t(\theta, x', A') \;\leq\; & 2\ln\theta - \frac{t}{M}[\sum_{m=1}^{M} \ln(\theta - f_m) + \sum_{m=1}^{M} \ln(1 + \frac{2\tau M}{t\theta} p_m(f_m - \hat{f}_m)) \\ & + \sum_{m\in A} \ln(g_m - 1/\theta) \\ & + \sum_{m\in A} \ln(1 + \frac{2\tau M\theta}{t} q_m(\hat{g}_m - g_m)) + (M - |A|)\ln T].\end{aligned}$$

Note that in Case 1 and Case 2 we have obtained exactly the same bound for $\Phi_t(\theta, x', A')$. In both cases using $\phi_t(x', A') \leq \Phi_t(\theta, x', A')$ and further rearrangement yields

$$\begin{aligned}\Phi_t(\theta, x', A') \leq\; & \phi_t(x, A) \\ & - \frac{t}{M}[\sum_{m=1}^{M}(1 + \frac{2\tau M}{t\theta}) p_m(f_m - \hat{f}_m) + \sum_{m\in A} \ln(1 + \frac{2\tau M\theta}{t} q_m(\hat{g}_m - g_m)))].\end{aligned}$$

Since τ is feasible, the elementary inequality $\ln(1+\alpha) \geq \alpha - \alpha^2$ for each $\alpha \in [-1/2, \infty)$ yields

$$\ln(1 + \frac{2\tau M}{t\phi} p_m(f_m - \hat{f}_m)) \geq \frac{2\tau M}{t\phi} p_m(f_m - \hat{f}_m) - (\frac{2\tau M}{t\phi} p_m(f_m - \hat{f}_m))^2$$

for each $m \in \{1, \ldots, M\}$ and

$$\ln(1 + \frac{2\tau M\phi}{t} q_m(\hat{g}_m - g_m)) \geq \frac{2\tau M\phi}{t} q_m(\hat{g}_m - g_m) - (\frac{2\tau M\phi}{t} q_m(\hat{g}_m - g_m))^2$$

for each $m \in A$. Using this and suitable rearrangement to continue the calculations above as in the proof of Lemma 9, we obtain

$$\begin{aligned}\phi_t(x, A) - \phi_t(x', A') \;\geq\; & 2\tau[(p^T f - p^T \hat{f})/\theta + (q^T \hat{g} - q^T g)\theta] \\ & - \frac{4M\tau^2}{t}[(p^T f + p^T \hat{f})\theta + (q^T \hat{g} + q^T g)\theta]^2\end{aligned}$$

in Case 1 and Case 2. Finally we use (2.7) to continue the chain of inequalities from

above to obtain

$$\phi_t(x, A) - \phi_t(x', A') > t^3/(4M)$$

which proves the first part of the claim. *Case 3:* We use τ and we have $A' \subset A$ after the interpolation. Consequently, the condition in Step 2.4.4 is satisfied, which means $\max\{(1-\tau)g_m + \tau\hat{g}_m | m \in A\} > T$; by construction of τ', we have $(1-\tau')g_m + \tau'\hat{g}_m \leq T$ for each $m \in A$. In particular this means that $\ln T \geq \ln((1-\tau)g_m) + \tau\hat{g}_m - 1/\theta)$ holds. Similar as above, we have

$$\begin{aligned}\Phi_t(\theta, x', A') \leq \phi_t(x, A) \\ - \frac{t}{M}[\sum_{m=1}^{M} \ln(1 + \frac{2\tau' M}{t\theta})p_m(f_m - \hat{f}_m) + \sum_{m \in A} \ln(1 + \frac{2\tau' M\theta}{t} q_m(\hat{g}_m - g_m))].\end{aligned}$$

Using the same arguments as in Case 1 and Case 2 as in the proof of Lemma 9 we obtain

$$\begin{aligned}\phi_t(x, A) - \phi_t(x', A') &\geq 2\tau'[(p^T f - p^T \hat{f})/\theta + (q^T \hat{g} - q^T g)\theta] \\ &\quad - \frac{4M\tau'^2}{t}[(p^T f + p^T \hat{f})/\theta + (q^T \hat{g} + q^T g)\theta]^2 \\ &\geq 2\tau'[(p^T f - p^T \hat{f})/\theta + (q^T \hat{g} - q^T g)\theta] \\ &\quad - \frac{4M\tau'\tau}{t}[(p^T f + p^T \hat{f})/\theta + (q^T \hat{g} + q^T g)\theta]^2 \\ &= 2\tau'[[(p^T f - p^T \hat{f})/\theta + (q^T \hat{g} - q^T g)\theta] \\ &\quad - \frac{2M\tau}{t}[(p^T f + p^T \hat{f})/\theta + (q^T \hat{g} + q^T g)\theta]^2].\end{aligned}$$

Inserting the inequality (2.7) and the definition of τ yields

$$\phi_t(x, A) - \phi_t(x', A') \geq \tau t[(p^T f + p^T \hat{f})/\theta + (q^T \hat{g} + q^T g)\theta]$$

and concludes the proof. □

Theorem 17 will be used in Section 4 to obtain a bound on the number of iterations. However, next we discuss how to implement Step 2.5 {mixing phase} of the algorithm in Figure 2.3. More precisely, first we show that the values of f can not be arbitrary large, which is important for the proof of Lemma 19.

Lemma 18. *In any iteration of* {coordination phase} *of the algorithm in Figure 2.3, the inequality* $\max\{f_m | m \in \{1, \ldots, M\}\} \leq 4cM/t$ *holds.*

Proof. For the initial solution $x^{(0)} \in B$ we have $f_m(x^{(0)}) \leq 3cM72 \leq 3cM/t < 4cM/t$ for

each $m \in \{1, \ldots, M\}$. For each block solution $\hat{x} \in B$ we have $p^T \hat{f} \leq c(1+\tau)\bar{p} \leq 2c\bar{p} \leq 2c$. We aim at a contradiction and assume that $\max\{f_m | m \in \{1, \ldots, M\}\} > cM/t$ holds. Let $m \in M$ such that $f_m = \max\{f_m | m \in \{1, \ldots, M\}\}$ holds. Then we obtain

$$p_m = \frac{t}{2M} \frac{\theta}{\theta - f_m} > \frac{t}{2M};$$

on the other hand we have

$$2c = \frac{t}{2M} \frac{4cM}{t} < p_m \hat{f}_m \leq p^T \hat{f} \leq 2c,$$

a contradiction. This means that $\hat{f}_m \leq 4cM/t$ holds for each $m \in M$. Furthermore, x' is obtained from x and $\hat{x}$ by interpolation. Using the convexity of f we obtain

$$\begin{aligned} f_m(x') &= f_m((1-\tau)x + \tau\hat{x})) \\ &\leq (1-\tau) f_m(x) + \tau f_m(\hat{x}) \\ &\leq (1-\tau) 4cM/t + \tau 4cM/t \\ &= 4cM/t \end{aligned}$$

for each $m \in \{1, \ldots, M\}$, from which inductively the claim follows. □

Finally we prove properties of the threshold value and explain how to compute the convex combination in Step 2.5. More precisely, the threshold value to be used for an iteration of {scaling phase} is defined by

$$T := 2112 M^3 \epsilon_s^{-2} / \lambda(y)$$

where $y \in B$ is the initial solution for the first scaling phase or the final iterate of the previous scaling phase. Notice that $\lambda(y) \geq 1$ holds, since otherwise the algorithm in Figure 2.3 had computed a feasible solution and would have terminated by evaluating the condition in Step 1. The convex combination computed in Step 2.5 {mixing phase} is defined by

$$x^{(s)} := \sum_{i=1}^{k} \frac{\epsilon_s^2}{264M^2} x_i + (1 - \frac{k\epsilon_s^2}{264M^2}) y \in B$$

where $x_1, \ldots, x_k$ are the iterates that are stored in Step 2.2 or Step 2.4.6 and y is the last iterate of the scaling phase that is stored in Step 2.3 or Step 2.4.2.

Next we prove that the final iterate of each scaling phase actually has the properties

we have indicated in before.

Lemma 19. *The final iterate $(x^{(s)})$ of scaling phase s satisfies $\lambda(x^{(s)}) \leq c/(1-\epsilon_s)$.*

Proof. First we use the convexity of f and Lemma 18; for each iterate $x \in B$, in particular for each $x \in \{x_1, \ldots, x_k\}$, we have $\max\{f_m(x) | m \in \{1, \ldots, M\}\} < cM/t$. Hence we obtain

$$\begin{aligned} f_m(x^{(s)}) &\leq \frac{\epsilon_s^2}{264M^2} \sum_{i=1}^{k} f_m(x_i) + (1 - \frac{k\epsilon_s^2}{264M^2}) f_m(y) \\ &= \frac{\epsilon_s^2}{264M^2} \frac{4kM}{t} + (1 - \frac{k\epsilon_s^2}{264M^2}) f_m(y) \end{aligned}$$

for each $m \in \{1, \ldots, M\}$. Since y satisfies one of the three stopping rules, we have $f_m(y) \leq c/(1-\epsilon_s/4)$ for each $m \in \{1, \ldots, M\}$. Furthermore we have $k \leq M$ and $t = \epsilon_s/32$, which yields

$$f_m(x^{(s)}) \leq \frac{32\epsilon_s}{66} + (1 - \frac{k\epsilon_s^2}{264M^2}) \frac{1}{1-\epsilon_s/4}$$

for each $m \in \{1, \ldots, M\}$. The first summand is at most $\epsilon_s/2$ and the second summand is bounded by $1 + \epsilon_s/2$, hence we have $f_m(x^{(s)}) \leq 1 + \epsilon_s$ for each $m \in \{1, \ldots, M\}$. Using the concavity and nonnegativity of g, for each $m \in \{1, \ldots, M\} \setminus A$ we have

$$g_m(x^{(s)}) \geq \frac{\epsilon_s^2}{264M^2} g_m(x_i) \geq \frac{\epsilon_s^2}{264M^2} T = 8M/\lambda(x^{(s-1)})$$

for each $x_i \in \{x_1, \ldots, x_k\}$. For the first scaling phase that means we have

$$g_m(x^{(1)}) \geq 4M/(3cM) > 1/c(1-\epsilon_s)$$

for each $m \in \{1, \ldots, M\} \setminus A$ by Lemma 14; for the other scaling phases we inductively obtain

$$g_m(x^s) \geq 8cM(1-\epsilon_s) \geq 8c(1-\epsilon_s) = 8(1-2\epsilon_s) \geq 1-\epsilon_s$$

for each $m \in \{1, \ldots, M\} \setminus A$, where the last inequality holds since $\epsilon_s \leq 7/15 < 1/2$ is satisfied. Finally, again using the concavity and nonnegativity of g, for each $m \in A$ we have

$$g_m(x^{(s)}) \geq (1 - \frac{k\epsilon_s^2}{264M^2}) g_m(y) \geq (1 - \frac{k\epsilon_s^2}{264M^2})(1-\epsilon_s/4) \geq 1-\epsilon_s$$

where the last inequality holds since $\epsilon_s \leq 1$ is satisfied; in total, the claim follows. □

2.6. Analysis of the Second Algorithm

In this section we show that the algorithm in Figure 2.3 terminates and obtain a bound for the running time. First we show that within a scaling phase that does not terminate by Stopping Rule 2, the difference of the reduced potentials of the initial solution and the iterate can not arbitrary large.

Lemma 20. *Let $x \in B$ be the initial iterate of a scaling phase and let $x' \in B$ arbitrary in such a way that x' does not satisfy Stopping Rule 2. Furthermore denote by A, A' and θ, θ' the corresponding associated sets of active indices and the minimizers of the potential function, respectively. Then*

$$\phi_t(x, A) - \phi_t(x', A') \leq (2 - t) \ln(1/\omega) + O(t \ln[M/t])$$

holds, where ω denotes the parameter for Stopping Rule 2 in the respective scaling phase.

Proof. For notational simplicity we write $\lambda := \lambda(x, A)$ and $\lambda' := \lambda(x', A')$. By Lemma 2 we have the bounds

$$\phi_t(x, A) \leq 2 \ln \theta + 2 \ln(2M/T) + t \ln(1 + t/(2M))$$

and

$$\phi_t(x', A') \geq (2 - t) \ln \theta' - t \ln T;$$

we obtain

$$\phi_t(x, A) - \phi_t(x', A') \leq (2 - t) \ln(\theta/\theta') + t \ln \theta + t \ln T + 2t \ln(2M/t) + t \ln[1 + t/(2M)]$$

by rearranging. Lemma 1 yields $\theta < \lambda/(1 - t)$ and $\theta' > \lambda'[1 + t/(2M)] > \lambda'$, from which we obtain

$$\begin{aligned} \ln(\theta/\theta') &< \ln(\lambda/[\lambda'(1 - t)]) \\ &= \ln(\lambda/\lambda') + \ln[1/(1 - t)] \\ &< \ln(1/\omega) + \ln[1/(1 - t)] \\ &\leq \ln(1/\omega) + \ln(1 + 2t) \end{aligned}$$

since x' does not satisfy the condition of Stopping Rule 2; we used $t \in (0, 1/2]$ for the last inequality. Now we aim at bounding the other summands; more precisely we use

the definition of T and have

$$\begin{aligned} t\ln\theta + t\ln T &\leq t\ln[\lambda/(1-t)] + t\ln[2112M^3/(1024t^2\lambda)] \\ &\leq t\ln(\lambda/t) + t\ln[2112M^3/1024t^2\lambda)] \\ &= t\ln[33M^3/16t^3)] \\ &= 3t\ln[M/(Ct)] \in O(t\ln[M/t]) \end{aligned}$$

where $C = \sqrt[3]{33/16} \approx 1,27$. The remaining summands are also bounded by $O(t\ln M/t)$ which means that we have $\phi_t(x,A) - \phi_t(x',A') \leq (2-t)\ln(1/\omega) + O(t\ln[M/t])$ which finishes the proof. □

Informally, Lemma 20 states that a scaling phase that is not terminated because of Stopping Rule 3 must be terminated by Stopping Rule 2; more precisely, by Theorem 17 the reduced potential of the iterate decreases by at least $t^3/(4M)$ if no function index is eliminated and at least does not decrease if a function index is eliminated. However, the latter happens at most M times. Hence each scaling phase must terminate after a finite number of iterations. Finally, we present a suitable upper bound for the total number of iterations.

Theorem 21. *The algorithm in Figure 2.3 terminates after at most $O(M\epsilon^{-2}\ln(M\epsilon^{-1}))$ iterations.*

Proof. Let $s \in \mathbb{N}\setminus\{0\}$ and consider the s-th scaling phase. Let N_s be the number of iterations in scaling phase s and let x denote the initial iterate and x' the iterate in the last iteration; let $N_s' := N_s - 1$. Then, by Lemma 20 we have that

$$(2-t)\ln(1/\omega_s) + O(t\ln[M/t]) \geq \phi_t(x,A) - \phi_t(x',A') \geq t^3/M(N_s' - M)$$

holds. Furthermore $\ln(1/\omega_1) \in O(\ln M)$ and, since $\epsilon_s \in (0,1/4)$ and

$$\ln(1/\omega_s) \leq \ln(1+4\epsilon_s) \in O(\epsilon_s)$$

is satisfied for each $s \in \mathbb{N}\setminus\{0,1\}$. Now suitable rearrangement yields

$$N_s' \in O(M\epsilon_s^{-2}\ln(M\epsilon_s^{-1}))$$

for each $s \in \mathbb{N}\setminus\{0\}$. Summation over all scaling phases yields that the total number of

iterations is bounded by

$$O(M \ln(M\epsilon^{-1}) \sum_{k=0}^{\lceil \log(2/\epsilon) \rceil} 2^{2k});$$

the sum above is bounded by $O(\epsilon^{-2})$, hence the total number of iterations is

$$O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$$

which finishes the proof. □

2.7. Application for a Multicommodity Flow Problem

In this section we discuss how to apply the first algorithm for a multicommodity flow problem. In Subsection 2.7.1 we motivate and formally describe the problem. In Subsection 2.7.2 we give a brief survey over related work and finally in Subsection 2.7.3 we discuss how the first algorithm presented in this chapter can be applied to approximately solve it. There we start with a simpler version of our problem to expose the basic approach; later we refine the approach for a more general problem.

2.7.1. Problem Definition

In this problem we are given a directed graph $G = (V, E)$; furthermore we are given $k \in \mathbb{N}$ *commodities* $(s_1, t_1), \ldots, (s_k, t_k) \in V \times V$. Formally, the commodities are pairs of *source* and *terminal* nodes indicating origin and destination of the i-th commodity. Intuitively we can imagine that we have to route the i-th commodity from s_i to t_i in the graph by finding a suitable s_i-t_i-flow for each $i \in \{1, \ldots, k\}$. Furthermore, we also denote $n := |V|$ and $M := |E|$.

Furthermore, the assignment of flow is constrained by suitable requirements. On one hand, for each $i \in \{1, \ldots, k\}$ we are given a *demand* $d_i \in \mathbb{R}_+$ indicating that an amount of at least d_i of the i-th commodity is to be routed from s_i to t_i. On the other hand, for each arc $e \in E$ we are given a *capacity* $c_e \in \mathbb{R}_+$ indicating that the total amount of flow routed over e should not exceed c_e. Furthermore, for each $i \in \{1, \ldots, k\}$ we denote by

$$P_i := \{p' | p' \text{ is an } s_i\text{-}t_i \text{ path in } G \text{ without node repetition}\}$$

the set of all s_i-t_i-paths in G; for each $i \in \{1, \ldots, k\}$ and $p' \in P_i$, we are given a *cost* $w_i(p') \in \mathbb{R}_+$ defining the cost of routing one unit of the i-th commodity along p. Here we consider the following two possibilities of modelization.

- In the first case we are given k values $w_1, \ldots, w_k$ which indicate the cost of routing one unit of flow from s_i to t_i. This means that the cost depend on the commodity only. We call this case the *commodity version* of the problem.
- In the second case we are given k nonnegative cost functions $w_i : E \to \mathbb{R}_+$ on the arcs which in a natural way for each commodity induce a cost for each path. This means that the cost of a path depends on the commodity as well as on the edges of the specific path. We call this case the *edge-commodity version* of the problem.

Finally, we are given a *budget* $W \in \mathbb{R}$ which we like to spend in total for the assignment of flow. For any $i \in \{1, \ldots, k\}$ and $p' \in P_i$ we use a variable $x_i(p') \in \mathbb{R}_+$ to denote the amount of the i-th commodity routed along p'.

Now the *commodity version of the fractional multicommodity flow problem with fixed budget* can be described as the following linear feasibility problem.

$$\begin{array}{lcll} \sum_{i=1}^{k} \sum_{p' \in P_i} w_i x_i(p') & = & W & & (1) \\ \sum_{p' \in P_i} x_i(p') & \geq & d_i & \text{for each } i \in \{1, \ldots, k\} & (2) \\ \sum_{i=1}^{k} \sum_{e \in p' \in P_i} x_i(p') & \leq & c_e & \text{for each } e \in E & (3) \\ x_i(p') & \geq & 0 & \text{for each } i \in \{1, \ldots, k\},\ p' \in P_i & (4) \end{array}$$

The constraints model the requirements informally described above. Constraint (1) means that the total cost incurred by the assignment of flow must be exactly W, the total budget. Constraint (2) requires the assigned flow of the i-th commodity to be at least the corresponding demand d_i for each $i \in \{1, \ldots, k\}$. Constraint (3) models the arc constraints by requiring the total flow routed over arc e to be at most c_e for any $e \in E$. Finally, the last constraint (4) requires the path variable $x_i(p')$ to be nonnegative for any $i \in \{1, \ldots, k\}$ and $p' \in P_i$.

Likewise, the *edge-commodity version of the fractional multicommodity flow problem with fixed budget* can be described as the following linear feasibility problem.

$$\begin{array}{lcll} \sum_{i=1}^{k} \sum_{p' \in P_i} w_i(p') x_i(p') & = & W & & (1) \\ \sum_{p' \in P_i} x_i(p') & \geq & d_i & \text{for each } i \in \{1, \ldots, k\} & (2) \\ \sum_{i=1}^{k} \sum_{e \in p' \in P_i} x_i(p') & \leq & c_e & \text{for each } e \in E & (3) \\ x_i(p') & \geq & 0 & \text{for each } i \in \{1, \ldots, k\},\ p' \in P_i & (4) \end{array}$$

Similar as for the *commodity version*, constraint (1) asserts that the total budget is spent, constraint (2) asserts covering of the demands, while constraint (3) asserts that the edge capacities are met.

After giving a brief survey over related work, we discuss how we can find an approximate solution to both problem formulations with the algorithm from Section 2.3. However, we will permit the constraints (2) and (3) to be slightly violated, i.e. we will aim at obtaining a multicommodity flow of total flow value exactly W in which for each commodity the demand d_i is slightly less than required while for each arc $e \in E$ the arc capacity c_e is slightly exceeded.

2.7.2. Related Work

In general, multicommodity flow problems are a popular area of research and there are similar results for different models. Leighton et al. [111, 112] studied the *maximum concurrent flow problem* where no costs are considered. Here the objective is to obtain a multicommodity flow which does not exceed the arc capacities and maximizes the percentage of each demand d_i to be routed, i.e. to find the maximum z such that at least z percent of the demand of each commodity can be feasibly routed. The authors present an ϵ-approximation algorithm with runtime bound $O(k^2 Mn \log k \log^3 n)$. Furthermore, Plotkin, Shmoys & Tardos [124] studied the *minimum-cost multicommodity flow problem*, where the goal is to compute a feasible flow that *exactly* satisfies each demand and has a minimum cost, where the cost on each edge is equal for each commodity. They obtained an algorithm which for any fixed $\epsilon \in (0, 1)$ finds a near optimal flow in running time $O(k^2 M \log n(M + n \log n) \log C)$, where $C := \sum_{e \in E} w(e)c_e$ is an upper bound for the total cost; the actual minimum cost is found via binary search. Bienstock & Iyengar [9] studied a formulation without any cost on the arcs; they studied a different *maximum concurrent flow problem* where the goal is to obtain a flow that *exactly* satisfies the demands and minimizes the maximum arc congestion, i. e. a flow which minimizes

$$\max\{\frac{\sum_{i=1}^{k} f_{i,e}}{c_e} | e \in E\}$$

where for any $e \in E$, $i \in \{1, \ldots, k\}$, we denote by $f_{i,e}$ the amount of commodity i routed through arc e. They obtained an ϵ-approximation algorithm which uses

$$O^*(\epsilon^{-1} K^2 M^{0.5}(L_U + M \log D + M \log \epsilon^{-1}) + \epsilon^{-1} nM)$$

shortest path problems, where L_U denotes the number of bits needed to store the capacities and $D := \sum_{i=1}^{k} d_i$. Likewise, Garg & Könemann [49, 50] studied the *maximum multicommodity flow problem*, where we have neither costs on the arcs nor demands; the objective is to maximize the total flow. They obtained an ϵ-approximation algorithm which performs at most $O(\epsilon^{-2}km \log L)$ shortest path computations, where $L \leq n$ is the maximum number of arcs on any s_i-t_i-path for any commodity $i \in \{1, \ldots, k\}$. Furthermore, they studied the *maximum concurrent flow* problem, where the objective is to maximize the percentage of d_i routed for any commodity $i \in \{1, \ldots, k\}$. For this variation they presented an ϵ-approximation algorithm which uses at most $O(\epsilon^{-2}k \log k \log M)$ calls to a minimum cost s-t-flow solver. They also modified this algorithm to replace the minimum flow computations by shortest path computations; in total, they obtained an ϵ-approximation algorithm which uses at most $O(\epsilon^{-2}(k \log M + M) \log M)$ shortest path computations. Finally, Fleischer [41, 42] also studied various multicommodity flow problems. For the maximum multicommodity flow problem, she obtained an FPTAS with runtime bound $O^*(\epsilon^{-2}kM^2)$; this runtime bound is independent from the number k of commodities and improves the runtime bound of $O^*(\epsilon^{-2}kM^2)$ obtained by Garg & Könemann [49, 50] and Grigoriadis & Khachiyan [53]. For the maximum concurrent flow problem (which in this chapter is termed the edge-commodity version of the problem where the cost function on the edges is the unit vector for each demand and the objective is to maximize the percentage of each demand satisfied) she obtained a running time of $O^*(\epsilon^{-2}M(M+k))$. This result improves the running times of $O^*(\epsilon^{-2}M(M+k)+kT_{MF}(n))$ obtained by Garg & Könemann [49, 50], where $T_{MF}(n)$ is the time needed to compute a solution to the maximum flow problem in a network with n nodes. Her result also improves the runtime bound of $O^*(\epsilon^{-2}kMn)$ obtained by Leighton et al. [111, 112] and Radzik [125, 126]. For the minimum cost concurrent flow problem (which is the problem also studied in this chapter, where the objective is to minimize the budget) she obtained a runtime bound of $O^*(\epsilon^{-2}M(M + k)I)$, where I is the encoding length of the largest integer occuring in the instance; this result improves the running times of $O^*((\epsilon^{-2}kM(M + k) + kMn)I)$ by Garg & Könemann [49, 50] and $O^*(\epsilon^{-2}kMnI)$ by Grigoriadis & Khachiyan [52]. Furthermore, the algrithm for the maximum multicommodity flow problem from [41, 42] has been implemented and experimtally evaluated in [16].

In both multicommodity flow problems, the underlying shortest path problems will turn out to be polynomially solvable. However, we will also prove that approximate algorithms for the corresponding shortest path problems can be used as well. Conse-

quently, we will give a brief survey over approximate approaches to various shortest path problems, both polynomially solvable and NP-complete.

Concerning classical shortest path problems, the well-known algorithm by Bellman & Ford solves the problem in $O(|E|n)$ time. For integral edge weights with absolute value bounded by N, the algorithm of Gabow & Tarjan [45] yields a runtime bound of $O(\sqrt{n}|E|\log(nN))$. However, if only nonnegative edge weights are permitted, the algorithm proposed by Dijkstra [37] solves the problem in $O((n+|E|)\log n)$ time [81]; however the running time can be improved to $O(n\log n + |E|)$ [44]. For the case of undirected graphs with positive integral weights, Thorup [137] presented an algorithm with running time $O(|E|)$. Klein [98] presented a parallel PTAS for the problem formulation with nonnegative weights on an undirected graph. His algorithm uses $\mathrm{e}\log n/\epsilon^{-2}$ processors and has a runtime bound of $O(\sqrt{n}\epsilon^{-2}\log n)$.

For the special case of planar graphs, the problem is algorithmically easier. The first algorithm for planar graphs was proposed by Lipton, Rose & Tarjan [116], yielding a runtime bound of $O(n^{3/2})$. Later Henzinger et al. [57] presented an algorithm with runtime bound $O(n^{4/3}\log^{2/3} D)$, where $D := \sum_{e\in E}\ell(e)$. Fakcharoenphol & Rao [40] found an algorithm with running time $O(n\log^3 n)$ Recently, Klein, Mozes & Weimann [97] presented an algorithm for planar graphs where positive and negative edge weights are permitted. Their algorithm has a runtime bound of $O(n\log^2 n)$.

Warburton [145] studied the restricted shortest path problem also called RSP. Here we are given an acyclic directed graph $G = (V, E)$ and two weight functions ℓ, $\ell' : E \to \mathbb{N}$ on the edges, modelling the length and the transition time of each edge. Furthermore we are given i, $j \in V$. A directed i-j-path p' is called a T-path if $\ell'(p') \leq T$. The restricted shortest path problem is to compute, for a given value of T, a T-path with minimum length with respect to ℓ. For this problem, which is NP-hard, the author presents an FPTAS with running time $O(n^3\epsilon^{-1}\log n\log B)$, where $B = (n-1)\max\{\ell(e)|e\in E\}$ is an upper bound for the optimal value. Later, this running time was improved by Phillips [123] who presented an algorithm with running time $O((|E|n\epsilon + (n^2/\epsilon)\log(n^2\epsilon))\log\log UB/LB)$, where UB and LB are upper and lower bounds for the optimum, respectively; her algorithm also does not require the graph to be acyclic. Furthermore, Hassin [56] proposed an FPTAS with running time $O(|E|n^2\epsilon^{-1}\log(n\epsilon^{-1}))$. Lorenz & Raz improved this running time to $O(|E|n(\log\log n + \epsilon^{-1}))$; the FPTAS presented by the authors also removed the requirement of G being acyclic. Finally, Ergün, Sinha & Zhang [38] presented an improved FPTAS for this problem with running time $O(|E|n\epsilon^{-1})$; however, their algorithm re-

quires the underlying graph to be acyclic.

2.7.3. Application of the First Algorithm

In this subsection we describe how to approximately solve the *commodity version* as well as the *edge-commodity version* of the *fractional multicommodity flow problem with fixed budget*. First we discuss the *commodity version* to expose the basic approach, which we later refine to solve the *edge-commodity version*.

Solving the Commodity Version

To solve the *commodity version*, we pursue the same approach as in [71] by using a suitable simplex as the underlying block for our algorithm where in our case we obtain a distorted standard simplex. More precisely, regarding the total budget as fixed and requiring the path variables to be nonnegative, the polytope defined by the constraints (1) and (4) is easily identified as a standard simplex which is distorted by path costs w_i for any $i \in \{1, \ldots, k\}$ and $p' \in P_i$, however, by which it constitutes a suitable block B for our algorithmic framework. More precisely, let

$$B := \{(x_i(p'))_{p' \in P_i, i \in \{1,\ldots,k\}} | x_i(p') \geq 0, \sum_{i=1}^{k} \sum_{p' \in P_i} w_i x_i(p') = W\}$$

denote the set of all flow assignments on the path variables which are nonnegative and sum up to cost exactly W. Likewise, we can rearrange the constraints (2) and (3) to define the packing and covering constraints by introducing function vectors $f : B \to \mathbb{R}^k$, $g : B :\to \mathbb{R}^{|E|}$ via

$$\begin{array}{rcrcl} g_i(x) & := & \sum_{p' \in P_i} \frac{x_i(p')}{d_i} & \geq & 1 \quad \text{for each } i \in \{1, \ldots, k\}, \\ f_e(x) & := & \sum_{i=1}^{k} \sum_{e \in p' \in P_i} \frac{x_i(p')}{c_e} & \leq & 1 \quad \text{for each } e \in E. \end{array}$$

In an actual implementation, we would use $M := \max\{k, |E|\}$ functions for each type of constraint by simply introducing redundant constraints. Next we discuss the block problem. As mentioned before, the block problem for the first algorithm can be formulated as

$$\text{find } \hat{x} \in B \text{ such that } p^T f(\hat{x})/Y(1,t) - q^T g(\hat{x}) Y(1,t) \leq \alpha$$
$$\text{or correctly decide that there is no } x \in B \text{ such that}$$
$$p^T f(\hat{x})/(1 + 8/3t) - q^T g(\hat{x})(1 + 8/3t) \leq \alpha$$

where $Y(1,t) := (1+8/3t)(1+t)$ and α is a parameter as defined in the algorithm; note that here we assume $c = 1$ since the block problem will turn out to be polynomially solvable. If the block problem is decided in the negative, the algorithm concludes that the given instance of the multicommodity problem is infeasible. The block problem can be solved by simply minimizing the function $p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t)$, which is linear in our case, over the simplex B. More precisely, we obtain

$$\begin{aligned}
& p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) \\
= \; & \frac{1}{(1+8/3t)} \sum_{e\in E} \frac{p_e}{c_e} \sum_{i=1}^{k} \sum_{e\in p'\in P_i} x_i(p') - (1+8/3t) \sum_{i=1}^{k} \frac{q_i}{d_i} \sum_{p'\in P_i} x_i(p') \\
= \; & \frac{1}{(1+8/3t)} \sum_{e\in E} \sum_{i=1}^{k} \sum_{e\in p'\in P_i} \frac{p_e}{c_e} x_i(p') - (1+8/3t) \sum_{i=1}^{k} \sum_{p'\in P_i} \frac{q_i}{d_i} x_i(p') \\
= \; & \sum_{e\in E} \sum_{i=1}^{k} \sum_{e\in p\in P_i} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} x_i(p') - \sum_{i=1}^{k} \sum_{p'\in P_i} (1+8/3t) \frac{q_i}{d_i} x_i(p') \\
= \; & \sum_{i=1}^{k} \sum_{e\in E} \sum_{e\in p'\in P_i} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} x_i(p') - \sum_{i=1}^{k} \sum_{p'\in P_i} (1+8/3t) \frac{q_i}{d_i} x_i(p') \\
= \; & \sum_{i=1}^{k} \sum_{p'\in P_i} \sum_{e\in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} x_i(p') - \sum_{i=1}^{k} \sum_{p'\in P_i} (1+8/3t) \frac{q_i}{d_i} x_i(p') \\
= \; & \sum_{i=1}^{k} \sum_{p'\in P_i} \Big(\sum_{e\in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} - (1+8/3t) \frac{q_i}{d_i} \Big) x_i(p')
\end{aligned}$$

and in a very similar way we obtain

$$\begin{aligned}
& p^T f(\hat{x})/[(1+8/3t)(1+t)] - q^T g(\hat{x})(1+8/3t)(1+t) \\
& \qquad = \sum_{i=1}^{k} \sum_{p'\in P_i} \Big(\sum_{e\in p'} \frac{1}{(1+8/3t)(1+t)} \frac{p_e}{c_e} - (1+8/3t)(1+t) \frac{q_i}{d_i} \Big) x_i(p').
\end{aligned}$$

Now for each $p' \in \cup_{i=1}^{k} P_i$ let $\ell(p')$ denote the length of p' given by the arc weights

$$\frac{1}{(1+8/3t)} \frac{p_e}{c_e}$$

for each $e \in E$, i.e.

$$\ell(p') := \sum_{e\in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_e}.$$

Since the set B defined above is a simplex, its optimum is attained at a vertex, which is a solution in which exactly one variable

$$x_i(p') = \frac{W}{w_i}$$

while all other components are zero. Such a solution can be found by selecting a commodity $i \in \{1, \dots, k\}$ and an s_i-t_i-path $p' \in P_i$ such that the value

$$\frac{(\ell(p') - (1 + 8/3t)\frac{q_i}{d_i})W}{w_i}$$

is minimized. However, for a fixed commodity $i \in \{1, \dots, k\}$, this expression is minimized if and only if the value of $\ell(p')$ is minimized. In total, this means that the block problem can be formulated as

$$\begin{gathered}
\text{find a commodity } i \in \{1, \dots, k\} \text{ and an } s_i\text{-}t_i\text{-path } p' \text{ such that} \\
\frac{\ell(p')W}{(1+t)} - (1 + 8/3t)(1+t)\frac{q_i}{d_i}W \leq \alpha w_i \\
\text{or correctly decide that there is no} \\
\text{commodity } i \in \{1, \dots, k\} \text{ and } s_i\text{-}t_i\text{-path } p' \text{ such that} \\
\ell(p')W - (1 + 8/3t)\frac{q_i}{d_i}W \leq \alpha w_i
\end{gathered}$$

Hence, we obtain the following result.

Lemma 22. *The underlying block problem of the commodity version can be solved with an exact algorithm A for the directed shortest path problem with nonnegative edge weights.*

Proof. Let $p, q \in \mathbb{R}^M_+$ and $\alpha \in \mathbb{R}$ be the parameters used in an iteration of the algorithm from Figure 2.2. Note that the arc weights defined by ℓ are nonnegative. Let A be an exact algorithm for the shortest path problem in directed graphs with nonnegative edge weights.

In order to solve the block problem, for all commodities $i \in \{1, \dots, k\}$, we execute A on G with the arc weight function ℓ to find a shortest s_i-t_i path p'_i. Let $i' \in \{1, \dots, k\}$ such that

$$\ell(p'_{i'})W - (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'}$$

is minimized.

Case 1: We have

$$\ell(p'_{i'})W - (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} \leq 0.$$

In this case

$$\frac{\ell(p'_{i'})W}{(1+t)} - (1 + 8/3t)(1+t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} \leq 0$$

is satisfied as well and we return $x_{i'}(p')$ as the block solution.

Case 2: We have

$$\ell(p'_{i'})W - (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} > 0.$$

In this case there is no commodity $i \in \{1, \ldots, k\}$ such that there is an s_i-t_i-path p'_i such that

$$\ell(p'_i)W - (1 + 8/3t)\frac{q_i}{d_i}W - \alpha w_i \leq 0.$$

holds. In this case we decide the block problem in the negative; we conclude that the instance of the multicommodity flow problem is infeasible. □

In total, this means that a solution of the block problem can be obtained by solving a shortest path problem for each commodity $i \in \{1, \ldots, k\}$ and selecting the commodity for which a shortest path of minimum length is found. Note that the block problem can be solved to optimality within polynomial time, i.e. we even have $c = 1$ since each shortest path computation can be carried out e.g. in time $O(n^2)$ time via Dijkstra's algorithm or, in the case of a planar graph, $O(n \log^3 n)$ time via an algorithm by Klein et al. [97]. However, the shortest path problem can be solved in $O(m)$ time using an algorithm by Thorup [137] if the underlying graph is undirected. In total, we have proved the following theorem.

Theorem 23. *The algorithm in Figure 2.2 can be used to approximately solve the commodity version of the fractional multicommodity flow problem with fixed budget in* $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ *iterations where* $M = \max\{k, |E|\}$ *and each occuring block problem can be solved via k shortest path computations where only nonnegative edge weights occur. In total, this results in a runtime bound of*

$$O(kT_{SP}(n)M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$$

where $T_{SP}(n)$ *is the time needed for a shortest path computation in a graph with n vertices.*

Furthermore, we like to point out that it would be worthwhile to investigate whether

it is possible to use the approach by Radzik [125, 126], where in each iteration, the commodities $i \in \{1, \dots, k\}$ are served in a round-robin manner.

Note that we can use a preprocessing step which scales each flow by a factor of $1/(1+\epsilon)$ to obtain a solution which exactly satisfies the arc capacities, uses slightly less total budget and routes a slightly smaller percentage of the demand of each commodity.

Finally, we would like to point out that the approach described above also can be used to optimize the resource W. To this end, let $w_{\max} := \max\{i \in \{1, \dots, k\} | w_i\}$ be the largest cost associated with a commodity. Note that $C := \sum_{e \in E} w_{\max}(e) c_e$ is an upper bound for the total cost incurred by any feasible multicommodity flow. Now very similar as in the application for Strip Packing in [71], we perform binary search over the resource and obtain a polynomial runtime bound. Clearly, using the minimum value of $W \in (0, C]$ for which the fractional multicommodity flow problem is approximately feasible, we use at most $\log C$ binary search steps where $\log C$ is polynomially bounded in the encoding length of the instance. In total, we obtain the following result.

Theorem 24. *The algorithm in Figure 2.2 can be used to approximately solve the commodity version of the fractional minimum cost multicommodity flow problem in*

$$O((\log C) M (\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$$

iterations where $M = \max\{k, |E|\}$ *and each occuring block problem can be solved in* $O(kT_{SP}(n))$ *time, where* $T_{SP}(n)$ *is the time needed for a shortest path computation in a graph with* n *vertices. In total, this results in a runtime bound of*

$$O(\log C (kT_{SP}(n)M)(\ln M + \epsilon^{-2} \ln \epsilon^{-1})).$$

Similar as before, we can scale the resulting flow with a factor of $1/(1+\epsilon)$ to obtain a flow of smaller cost which exactly satisfies the arc capacities but routes slightly less percentage of each demand. In total, this means that for any fixed value of ϵ, the algorithm in Figure 2.2 yields a runtime bound of

$$O(\log C (kT_{SP}(n)M)(M \ln M)) = O(\log C (kT_{SP}(n)M)(M \ln n)).$$

In comparison, the algorithm from [124] (where the cost of routing flow only depends on the arcs) yields a runtime bound of

$$O(\log C (k^2 T_{SP}(n)M)(M \log n));$$

in total, we improve the running time by a factor of k where in our problem we consider a different cost function for the flow.

Comment. Although in the application discussed here the block problem can be solved in polynomial time, in the following we argue that an approximate algorithm for the underlying shortest path problem can be used here as well.

If we aim at an approximate solution, the resulting block problem can be formulated as follows.

$$\text{find } \hat{x} \in B \text{ such that } p^T f(\hat{x})/[c(1+t)(1+8/3t)] - q^T g(\hat{x}) c(1+t)(1+8/3t) \leq \alpha$$
$$\text{or correctly decide that there is no } x \in B \text{ such that}$$
$$p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) \leq \alpha$$

Similar as before, we obtain

$$p^T f(\hat{x})/[c(1+t)(1+8/3t)] - q^T g(\hat{x}) c(1+t)(1+8/3t)$$
$$= \sum_{i=1}^{k} \sum_{p' \in P_i} \Big(\sum_{e \in p'} \frac{1}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - c(1+t)(1+8/3t) \frac{q_i}{d_i} \Big) x_i(p')$$

from which we conclude which block problem is to be solved. For each $p' \in \cup_{i=1}^{k} P_i$ let $\ell(p')$ denote the length of p' given by the arc weights

$$\frac{1}{(1+8/3t)} \frac{p_e}{c_c}$$

for each $e \in E$, i.e.

$$\ell(p') := \sum_{e \in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_c}.$$

As before, since B is a simplex its optimum is attained at a vertex, which is a solution in which exactly one variable $x_i(p') = W/w_i$ while all other components are zero. Such a solution can be found by selecting a commodity $i \in \{1, \ldots, k\}$ and an s_i-t_i-path p' such that the value

$$\frac{(\ell(p') - (1+8/3t)\frac{q_i}{d_i})}{w_i} W$$

is minimized, which can be done respectively by minimizing the value of $\ell(p')$. In total,

this means that the block problem can be formulated as

$$\text{find a commodity } i \in \{1,\dots,k\} \text{ and an } s_i\text{-}t_i\text{-path } p' \text{ such that}$$
$$\frac{\ell(p')W}{c(1+t)} - c(1+8/3t)(1+t)\frac{q_i}{d_i}W \leq \alpha w_i$$
$$\text{or correctly decide that there is no}$$
$$\text{commodity } i \in \{1,\dots,k\} \text{ and } s_i\text{-}t_i\text{-path } p' \text{ such that}$$
$$\ell(p')W - (1+8/3t)\frac{q_i}{d_i}W \leq \alpha w_i$$

Hence, we obtain the following result.

Lemma 25. *The underlying block problem of the commodity version can be solved with an approximation algorithm A with ratio $c(1+t)$ for the directed shortest path problem with nonnegative edge weights.*

Proof. Let $p, q \in \mathbb{R}^M_+$ and $\alpha \in \mathbb{R}$ be the parameters used in an iteration of the algorithm from Figure 2.2. Note that the arc weights defined by ℓ are nonnegative. Let A be an approximation algorithm with ratio $c(1+t)$ for the shortest path problem in directed graphs with nonnegative edge weights.

In order to solve the block problem, for all commodities $i \in \{1,\dots,k\}$, we execute A on G with the arc weight function ℓ to find an approximately shortest s_i-t_i path p'_i. Let $i' \in \{1,\dots,k\}$ such that

$$\frac{\ell(p'_{i'})W}{c(1+t)} - c(1+8/3t)(1+t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'}$$

is minimized.

Case 1: We have

$$\frac{\ell(p'_{i'})W}{c(1+t)} - c(1+8/3t)(1+t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} \leq 0.$$

In this case the block problem is decided in the positive and we return $x_{i'}(p'_{i'})$ as the block solution.

Case 2: We have

$$\frac{\ell(p'_{i'})W}{c(1+t)} - c(1+8/3t)(1+t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} > 0.$$

We show that in this case there is no commodity $i \in \{1,\dots,k\}$ such that there is an

s_i-t_i-path p'_i such that

$$\ell(p'_i)W - (1 + 8/3t)\frac{q_i}{d_i}W - \alpha w_i \leq 0.$$

holds. Aiming at a contradiction, we assume that there is an $i' \in \{1, \ldots, k\}$ and an $s_{i'}$-$t_{i'}$-path $p'_{i'}$ such that

$$\ell(p'_{i'})W - (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} \leq 0.$$

is satisfied. Consequently, since $c \geq 1$, there is an $i' \in \{1, \ldots, k\}$ such that A finds an $s_{i'}$-$t_{i'}$-path p' such that

$$\frac{\ell(p'_{i'})W}{c(1+t)} - (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} \leq 0.$$

holds. However, in this case

$$\frac{\ell(p'_{i'})W}{c(1+t)} - c(1 + 8/3t)(1+t)\frac{q_{i'}}{d_{i'}}W - \alpha w_{i'} \leq 0.$$

is satisfied as well, which yields a contradiction. In this case we decide the block problem in the negative; we conclude that the instance of the multicommodity flow problem is infeasible. □

Very similar as before, we can now use an algorithm for the single-source shortest path problem with ratio $c(1 + t)$ to either find a commodity $i \in \{1, \ldots, k\}$ and an s_i-t_i-path p such that

$$\frac{\ell(p')}{c(1+t)} \leq \frac{\alpha w_i}{W} + c(1+t)(1 + 8/3t)\frac{q_i}{d_i}$$

holds or correctly decide that there is no commodity $i \in \{1, \ldots, k\}$ such that there is an s_i-t_i-path p' for which the expression

$$\ell(p') \leq \frac{\alpha w_i}{W} + (1 + 8/3t)\frac{q_i}{d_i}$$

is satisfied. As a consequence, we can use an approximate algorithm for the underlying shortest path problem and obtain the following result. Here, the multicommodity flow we obtain may violate both the demand conditions and the edge conditions by a factor of $c(1 + \epsilon)$.

Theorem 26. *The algorithm in Figure 2.2 can be used to approximately solve the*

commodity version of the fractional multicommodity flow problem with fixed budget in $O(M(\ln M+\epsilon^{-2}\ln\epsilon^{-1}))$ *iterations where* $M = \max\{k, |E|\}$ *and each occuring block problem can be solved via* k *approximate shortest path computations where only nonnegative edge weights occur. In total, this results in a runtime bound of*

$$O(kT_{SP}(n)M(\ln M + \epsilon^{-2}\ln\epsilon^{-1}))$$

where $T_{SP}(n)$ *is the time needed for a shortest path computation in a graph with* n *vertices.*

Solving the Edge-Commodity Version

To solve the *edge-commodity version*, we apply the same approach as in the previous subsubsection. Again we use a suitable simplex as the underlying block for our algorithm. Regarding the total budged as fixed and requiring the path variables to be nonnegative, the polytope defined by the constraints (1) and (4) again is a distorted standard simplex which constitutes a suitable block B for our algorithmic framework.

More precisely, let

$$B := \{(x_i(p'))_{p'\in P_i, i\in\{1,\dots,k\}} | x_i(p') \geq 0, \sum_{i=1}^{k}\sum_{p\in P_i} w_i(p')x_i(p') = W\}$$

denote the set of all flow assignments on the path variables which are nonnegative and sum up to cost exactly W.

Similar as for the *commodity version*, we can rearrange the constraints (2) and (3) to define the packing and covering constraints by defining function verctors $f : B \to \mathbb{R}^k$, $g : B :\to \mathbb{R}^{|E|}$ via

$$\begin{array}{rcll} g_i(x) & := & \sum_{p'\in P_i} \frac{x_i(p')}{d_i} & \geq \quad 1 \quad \text{for each } i \in \{1,\dots,k\}, \\ f_e(x) & := & \sum_{i=1}^{k}\sum_{e\in p'\in P_i} \frac{x_i(p')}{c_e} & \leq \quad 1 \quad \text{for each } e \in E. \end{array}$$

In an actual implementation, we would use $M := \max\{k, |E|\}$ functions for each type of constraint by simply introducing redundant constraints. Next we discuss the block problem. As mentioned before, the block problem for the first algorithm can be

formulated as

$$\text{find } \hat{x} \in B \text{ such that } p^T f(\hat{x})/Y(1,t) - q^T g(\hat{x}) Y(1,t) \leq \alpha$$
$$\text{or correctly decide that there is no } x \in B \text{ such that}$$
$$p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) \leq \alpha$$

where $Y(1,t) := (1+8/3t)(1+t)$; note that here we assume $c = 1$ since again the block problem will turn out to be polynomially solvable. As in the previous subsubsection, we obtain

$$p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) = \sum_{i=1}^{k} \sum_{p' \in P_i} \left(\sum_{e \in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} - (1+8/3t) \frac{q_i}{d_i} \right) x_i(p').$$

and

$$p^T f(\hat{x})/[(1+8/3t)(1+t)] - q^T g(\hat{x})(1+t)(1+8/3t)$$
$$= \sum_{i=1}^{k} \sum_{p' \in P_i} \left(\sum_{e \in p'} \frac{1}{(1+8/3t)(1+t)} \frac{p_e}{c_e} - (1+8/3t)(1+t) \frac{q_i}{d_i} \right) x_i(p').$$

Since our set B is a simplex, its optimum for any linear objective function is attained at a vertex, which is a solution in which exactly one variable

$$x_i(p') = \frac{W}{w_i(p')}$$

while all other components are zero. Note that the condition

$$\left(\sum_{e \in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} - (1+8/3t) \frac{q_i}{d_i} \right) \frac{W}{w_i(p')} \leq \alpha$$

can be rearranged to

$$\sum_{e \in p'} \left(\frac{W}{(1+8/3t)} \frac{p_e}{c_e} - \alpha w_i(e) \right) \leq (1+8/3t) \frac{q_i}{d_i} W.$$

Likewise, the condition

$$\left(\sum_{e \in p'} \frac{1}{(1+t)(1+8/3t)} \frac{p_e}{c_e} - (1+t)(1+8/3t) \frac{q_i}{d_i} \right) \frac{W}{w_i(p')} \leq \alpha$$

can be rearranged to

$$\sum_{e\in p'}\left(\frac{W}{(1+t)(1+8/3t)}\frac{p_e}{c_e}-\alpha w_i(e)\right)\leq(1+t)(1+8/3t)\frac{q_i}{d_i}W$$

In total, this means that the block problem can be formulated as

find a commodity $i\in\{1,\dots,k\}$ and an s_i-t_i-path p' such that
$$\sum_{e\in p'}\left(\frac{W}{(1+t)(1+8/3t)}\frac{p_e}{c_e}-\alpha w_i(e)\right)\leq(1+t)(1+8/3t)\frac{q_i}{d_i}W$$
or correctly decide that there is no
commodity $i\in\{1,\dots,k\}$ and s_i-t_i-path p' such that
$$\sum_{e\in p'}\left(\frac{W}{(1+8/3t)}\frac{p_e}{c_e}-\alpha w_i(e)\right)\leq(1+8/3t)\frac{q_i}{d_i}W$$

Similar as before, for each $p'\in\cup_{i=1}^k P_i$ let $\ell(p')$ denote the length of p' given by the arc weights

$$\frac{W}{(1+8/3t)}\frac{p_e}{c_e}-\alpha w_i(e)$$

for each $e\in E$, i.e.

$$\ell(p'):=\sum_{e\in p'}\left(\frac{W}{(1+8/3t)}\frac{p_e}{c_e}-\alpha w_i(e)\right).$$

In particular this means that negative edge weights may occur. Consequently, cycles of negative length may occur in the graph, which makes the notion of a shortest path ill-defined. However, this difficulty is already circumvented by considering only paths without node repetition. In total we obtain the following result.

Lemma 27. *The underlying block problem of the edge-commodity version can be solved with an exact algorithm A for the directed shortest path problem with arbitrary edge weights.*

Proof. Let $p,q\in\mathbb{R}_+^M$ and $\alpha\in\mathbb{R}$ be the parameters used in an iteration of the algorithm from Figure 2.2. Note that the arc weights defined by ℓ may be negative. Let A be an exact algorithm for the shortest path problem in directed graphs with nonnegative edge weights.

In order to solve the block problem, for all commodities $i\in\{1,\dots,k\}$, we execute A on G with the arc weight function ℓ to find a shortest s_i-t_i path p_i'. Let $i'\in\{1,\dots,k\}$

such that

$$\ell(p'_{i'}) - (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W$$

is minimized.

Case 1: We have

$$\ell(p'_{i'}) \leq (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W$$

In this case we also have

$$\sum_{e \in p'_{i'}}\left(\frac{W}{(1+t)(1+8/3t)}\frac{p_e}{c_e} - \alpha w_{i'}(e)\right) \leq \ell(p'_{i'}) \leq (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W \leq (1+t)(1+8/3t)\frac{q_{i'}}{d_{i'}}W$$

and we return $x_{i'}(p')$ as the block solution.

Case 2: We have

$$\ell(p'_{i'}) > (1 + 8/3t)\frac{q_{i'}}{d_{i'}}W.$$

In this case there is no commodity $i \in \{1, \ldots, k\}$ such that there is an s_i-t_i path p'_i such that

$$\sum_{e \in p'}\left(\frac{W}{(1+8/3t)}\frac{p_e}{c_e} - \alpha w_i(e)\right) \leq (1 + 8/3t)\frac{q_i}{d_i}W$$

holds. In this case we decide the block problem in the negative; we conclude that the instance of the multicommodity flow problem is infeasible. □

In total, this means that we can solve the block problem by computing a shortest path in G with respect to the edge weight function ℓ' for each commodity $i \in \{1, \ldots, k\}$. Note that in contrast to the previous subsubsection, negative edge weights may occur. Shortest path problems like these can be solved e.g. in $O(n^3)$ time via the algorithm by Floyd & Warshall or in time $O(n \log^3 n)$ via a recent result by Klein et al. [97]; in total, we obtain the following result.

Theorem 28. *The algorithm in Figure 2.2 can be used to approximately solve the edge-commodity version of the fractional multicommodity flow problem with fixed budget in $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ iterations where $M = \max\{k, |E|\}$ and each occuring block problem can be solved via k shortest path computations where negative edge weights may occur. In total, this results in a runtime bound of*

$$O(kT_{SP}(n)M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$$

where $T_{SP}(n)$ is the time needed for a shortest path computation in a graph with n vertices.

As before we can use a preprocessing step which scales each flow by a factor of $1/(1+\epsilon)$ to obtain a solution which exactly satisfies the arc capacities, uses slightly less total budget and routes a slightly smaller percentage of the demand of each commodity.

Again the approach described above can be used to optimize the budget W. To this end, let $w_{\max} := \max\{i \in \{1,\dots,k\}, e \in E | w_i(e)\}$ be the largest cost associated with a commodity. Note that $C := \sum_{e\in E} w_{\max}(e)c_e$ is an upper bound for the total cost incurred by any feasible multicommodity flow. Now very similar as in the previous subsubsection, we perform binary search over the resource and obtain a polynomial runtime bound. Clearly, using the minimum value of $W \in (0, C]$ for which the fractional multicommodity flow problem is approximately feasible, we use at most $\log C$ binary search steps where $\log C$ is polynomially bounded in the encoding length of the instance. Hence we obtain the following result.

Theorem 29. *The algorithm in Figure 2.2 can be used to approximately solve the edge-commodity version of the fractional minimum cost multicommodity flow problem in*

$$O((\log C)M(\ln M + \epsilon^{-2}\ln\epsilon^{-1}))$$

iterations where $M = \max\{k, |E|\}$ and each occuring block problem can be solved in $O(kT_{SP}(n))$ time, where $T_{SP}(n)$ is the time needed for a shortest path computation in a graph with n vertices where negative edge weights may occur. In total, this results in a runtime bound of

$$O(\log C(kT_{SP}(n)M)(\ln M + \epsilon^{-2}\ln\epsilon^{-1})).$$

Similar as before, we can scale the resulting flow with a factor of $1/(1+\epsilon)$ to obtain a flow of smaller cost which exactly satisfies the arc capacities but routes slightly less percentage of each demand. In total, this means that for any fixed value of ϵ, the algorithm in Figure 2.2 yields a runtime bound of

$$O(\log C(kT_{SP}(n)M)(M\ln M)) = O(\log C(kT_{SP}(n)M)(M\ln n)).$$

In comparison, the algorithm from [124] (where the cost of routing flow only depends on the arc) yields a runtime bound of

$$O(\log C(k^2T_{SP}(n)M)(M\log n));$$

in total, we improve the running time by a factor of k and exchanging $\log n$ for $\ln M$ where in our problem we permit a more general cost function for the flow.

Comment. As in the previous subsubsection, again we remark that an approximate algorithm for the underlying shortest path problem can be used.

Again the block problem can be formulated as

$$\text{find } \hat{x} \in B \text{ such that } p^T f(\hat{x})/[c(1+t)(1+8/3t)] - q^T g(\hat{x}) c(1+t)(1+8/3t) \leq \alpha$$
$$\text{or correctly decide that there is no } x \in B \text{ such that}$$
$$p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t) \leq \alpha$$

and similar as before, we obtain

$$p^T f(\hat{x})/[c(1+t)(1+8/3t)] - q^T g(\hat{x}) c(1+t)(1+8/3t)$$
$$= \sum_{i=1}^{k} \sum_{p' \in P_i} \Big(\sum_{e \in p'} \frac{1}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - c(1+t)(1+8/3t) \frac{q_i}{d_i} \Big) x_i(p')$$

and

$$p^T f(\hat{x})/(1+8/3t) - q^T g(\hat{x})(1+8/3t)$$
$$= \sum_{i=1}^{k} \sum_{p' \in P_i} \Big(\sum_{e \in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} - (1+8/3t) \frac{q_i}{d_i} \Big) x_i(p').$$

Since our set B is a simplex, its optimum for any linear objective function is attained at a vertex, which is a solution in which exactly one variable

$$x_i = \frac{W}{w_i(p')}$$

while all other components are zero. Now, note that the condition

$$\Big(\sum_{e \in p'} \frac{1}{(1+8/3t)} \frac{p_e}{c_e} - (1+8/3t) \frac{q_i}{d_i} \Big) \frac{W}{w_i(p')} \leq \alpha$$

can be rearranged to

$$\sum_{e \in p'} \Big(\frac{W}{(1+8/3t)} \frac{p_e}{c_e} - \alpha w_i(e) \Big) \leq (1+8/3t) \frac{q_i}{d_i} W.$$

Likewise, the condition

$$(\sum_{e\in p'} \frac{1}{c(1+t)(1+8/3t)}\frac{p_e}{c_e} - c(1+t)(1+8/3t)\frac{q_i}{d_i})\frac{W}{w_i(p')} \leq \alpha$$

can be rearranged to

$$\sum_{e\in p'}(\frac{W}{c(1+t)(1+8/3t)}\frac{p_e}{c_e} - \alpha w_i(e)) \leq c(1+t)(1+8/3t)\frac{q_i}{d_i}W.$$

In total, this means that the block problem can be formulated as

find a commodity $i \in \{1,\dots,k\}$ and an s_i-t_i-path p' such that
$$\sum_{e\in p'}(\frac{W}{c(1+t)(1+8/3t)}\frac{p_e}{c_e} - \alpha w_i(e)) \leq c(1+t)(1+8/3t)\frac{q_i}{d_i}W$$
or correctly decide that there is no
commodity $i \in \{1,\dots,k\}$ and s_i-t_i-path p' such that
$$\sum_{e\in p'}(\frac{W}{(1+8/3t)}\frac{p_e}{c_e} - \alpha w_i(e)) \leq (1+8/3t)\frac{q_i}{d_i}W$$

Similar as before, for each $p' \in \cup_{i=1}^k P_i$ let $\ell(p')$ denote the length of p' given by the arc weights

$$\frac{W}{(1+8/3t)}\frac{p_e}{c_e} - \alpha w_i(e)$$

for each $e \in E$, i.e.

$$\ell(p') := \sum_{e\in p'}(\frac{W}{(1+8/3t)}\frac{p_e}{c_e} - \alpha w_i(e)).$$

As mentioned before, permitting edges of negative length may result in the occurence of cycles of negative length; for this reason, we only consider paths without node-repetition. However, since both paths with nonnegative and negative length may occur, the classical notion of an approximation algorithm can not be applied here; we have to be more precise what kind of approximate solutions we are interested in.

As we will see in the proof of the next lemma, an algorithm A with the following property is sufficient. Let $G = (V, E)$, be a directed graph, $s, t \in V$, $\ell : V \to \mathbb{R}$. Let $c \geq 1$, $t \in (0, 1)$. Let OPT denote the length of a shortest s-t-path in G without node repetition. If OPT ≤ 0, then A returns an s-t-path p' without node repetition such that $\ell(p') \leq 0$. IF OPT > 0, then A return an s-t-path p' without node repetition such that

$\ell(p')/[c(1+t)] \leq \mathrm{OPT}$.

In total, we obtain the following result.

Lemma 30. *The underlying block problem of the edge-commodity version can be solved with an approximation algorithm A with ratio $c(1+t)$ for the directed shortest path problem with arbitrary edge weights.*

Proof. Let $p, q \in \mathbb{R}_+^M$ and $\alpha \in \mathbb{R}$ be the parameters used in an iteration of the algorithm from Figure 2.2. Note that the arc weights defined by ℓ may be negative. Let A be an approximation algorithm with ratio $c(1+t)$ for the shortest path problem in directed graphs with arbitrary edge weights.

In order to solve the block problem, for all commodities $i \in \{1, \ldots, k\}$, we execute A on G with the arc weight function ℓ to find an approximately shortest s_i-t_i path p_i'. Let $i' \in \{1, \ldots, k\}$ such that

$$\ell(p_{i'}') - c(1+t)(1+8/3t)\frac{q_{i'}}{d_{i'}}W$$

is minimized.

Case 1: We have

$$\sum_{e \in p_{i'}'} \left(\frac{W}{c(1+t)(1+8/3t)}\frac{p_e}{c_e} - \alpha w_{i'}(e)\right) \leq c(1+t)(1+8/3t)\frac{q_{i'}}{d_{i'}}W.$$

In this case the block problem is decided in the positive and we return $x_{i'}(p_{i'}')$ as the block solution.

Case 2: We have

$$\sum_{e \in p_{i'}'} \left(\frac{W}{c(1+t)(1+8/3t)}\frac{p_e}{c_e} - \alpha w_{i'}(e)\right) > c(1+t)(1+8/3t)\frac{q_{i'}}{d_{i'}}W.$$

We show that in this case there is no commodity $i' \in \{1, \ldots, k\}$ such that there is an $s_{i'}$-$t_{i'}$-path $p_{i'}'$ such that

$$\sum_{e \in p_{i'}'} \left(\frac{W}{(1+8/3t)}\frac{p_e}{c_e} - \alpha w_{i'}(e)\right) \leq (1+8/3t)\frac{q_{i'}}{d_{i'}}W$$

holds. Aiming at a contradiction, we assume that there is an $i' \in \{1, \ldots, k\}$ and an

$s_{i'}$-$t_{i'}$-path $p'_{i'}$ such that

$$\sum_{e \in p'_{i'}} \left(\frac{W}{(1+8/3t)} \frac{p_e}{c_e} - \alpha w_{i'}(e)\right) \leq (1+8/3t) \frac{q_{i'}}{d_{i'}} W$$

is satisfied. Consequently, there is an $i' \in \{1, \ldots, k\}$ such that A finds an $s_{i'}$-$t_{i'}$-path p' such that

$$\sum_{e \in p'_{i'}} \left(\frac{W}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - \frac{\alpha w_{i'}(e)}{c(1+t)}\right) \leq (1+8/3t) \frac{q_{i'}}{d_{i'}} W$$

holds. Here we have to distinguish two cases.

Case 2.1: $\alpha \geq 0$. In this case we obtain

$$\begin{aligned}\sum_{e \in p'_{i'}} \left(\frac{W}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - \alpha w_{i'}\right) &\leq \sum_{e \in p'_{i'}} \left(\frac{W}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - \frac{\alpha w_{i'}(e)}{c(1+t)}\right) \\ &\leq (1+8/3t) \frac{q_{i'}}{d_{i'}} W \leq c(1+t)(1+8/3t) \frac{q_{i'}}{d_{i'}} W,\end{aligned}$$

which is a contradiction. In this case we decide the block problem in the negative; we conclude that the instance of the multicommodity flow problem is infeasible.

Case 2.2: $\alpha < 0$. In this case we obtain

$$\begin{aligned}\sum_{e \in p'_{i'}} \left(\frac{W}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - \alpha w_{i'}(e)\right) &\leq \sum_{e \in p'_{i'}} \left(\frac{W}{(1+8/3t)} \frac{p_e}{c_e} - \alpha w_{i'}(e)\right) \\ &= c(1+t) \sum_{e \in p'_{i'}} \left(\frac{W}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - \frac{\alpha w_{i'}(e)}{c(1+t)}\right) \leq c(1+t)(1+8/3t) \frac{q_{i'}}{d_{i'}} W,\end{aligned}$$

which is a contradiction. In this case we decide the block problem in the negative; we conclude that the instance of the multicommodity flow problem is infeasible. □

Very similar as before, we can now use an algorithm for the single-source shortest path problem with ratio $c(1+t)$ to either find a commodity $i \in \{1, \ldots, k\}$ and an s_i-t_i-path p such that

$$\sum_{e \in p'} \left(\frac{W}{c(1+t)(1+8/3t)} \frac{p_e}{c_e} - \alpha w_i(e)\right) \leq c(1+t)(1+8/3t) \frac{q_i}{d_i} W$$

holds or correctly decide that there is no commodity $i \in \{1, \ldots, k\}$ such that there is an

s_i-t_i-path p' for which the expression

$$\sum_{e \in p'} \left(\frac{W}{(1+8/3t)} \frac{p_e}{c_e} - \alpha w_i(e) \right) \leq (1+8/3t) \frac{q_i}{d_i} W$$

is satisfied.

As a consequence, we can use an approximate algorithm for the underlying shortest path problem and obtain the following result. Here, the multicommodity flow we obtain may violate both the demand conditions and the edge conditions by a factor of $c(1+\epsilon)$.

Theorem 31. *The algorithm in Figure 2.2 can be used to approximately solve the edge-commodity version of the fractional multicommodity flow problem with fixed budget in $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ iterations where $M = \max\{k, |E|\}$ and each occuring block problem can be solved via k approximate shortest path computations where negative edge weights may occur. In total, this results in a runtime bound of*

$$O(kT_{SP}(n)M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$$

where $T_{SP}(n)$ is the time needed for a shortest path computation in a graph with n vertices.

2.8. Conclusion

In this chapter we have presented two approximation algorithms for the mixed packing and covering problem with coordination complexity $O(M(\ln M + \epsilon^{-2} \ln \epsilon^{-1}))$ and $O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$, respectively. Both algorithms are based on the so-called Lagrangian decomposition approach. Furthermore, we have discussed how the first algorithm can be applied to approximately solve a multicommodity flow problem. Historically speaking, both of these algorithms are the consequent development from the algorithms in [54, 67, 79] by Grigoriadis, Khachiyan, Porkolab, Villavicencio, Jansen & Zhang, where so called pure packing and covering problems are solved with a similar approach. Unlike the most basic algorithm from [54], our algorithms use more general block solvers and do not require their precision to be arbitrarily high. We assumed above that the price vectors are computed exactly, which is impractical since we cannot solve (2.1) for θ; however an approximation for which only $O(M \ln(M\epsilon^{-1}))$ arithmetic operations per iteration are necessary is suitable as well, which can be shown with an elementary analysis that is very similar to Subsection 4.2 in [66, 70]. In total, we contributed efficient and

Table 2.1.: Algorithmic Results for Exact Solvers for Linear Programs.

Running Time	Reference	Remark
$O(MN^3L)$	[55]	L is a bound for the encoding length of the input
$O(((M+N)N^2+(M+N)^{1.5}NL)$	[140]	L is a bound for the encoding length of the input
$O(((M+N)N^2+(M+N)^{1.5}NL\sqrt{M+N})$	[83, 84]	L is a bound for the encoding length of the input

simple approximation algorithms with data-independent coordination complexity which solves an important class of feasibility problems, namely so-called mixed problems. In each iteration our algorithm needs to approximately solve a feasibility problem over B. Our results also solve a generalization of an open problem from [148, 150]. Within the limitations of the proof of Lemma 13, τ is amenable to line search – as a further idea for research, we suggest to evaluate whether line search for optimizing the step length τ (a heuristic improvement discussed in Chapter 3) yields a larger difference in the reduced potential similar to [1]. Besides, it is an open problem whether a reduction of $O(\epsilon^{-2})$ to $O(\epsilon^{-1})$ in the runtime bound is possible. Furthermore, no *lower* bound in terms of ϵ^{-1} on the number of iterations is known: In [54], only a lower bound of $\Theta(M)$ is mentioned for the covering problem. Very similar, in [53], the same bound of $\Theta(M)$ is presented for the packing problem. Hence, it is an interesting open question whether there is an instance or class of instances on which a suitable lower bound for the number of iterations can be proved.

Furthermore, we have applied one of our approximation algorithms to solve a multicommodity flow problem. We have solved feasibility formulations for a fixed budget W. This approach can be used to optimize the budget via binary search over W. Here, it remains an interesting open question how the computations of solutions for different choices of the target budget W differ; it might be worthwhile to see if the computations are related and if the running times of the algorithms can be improved.

Finally, we would like to present a brief survey on the development and recent results in approximation algorithms for problems similar to the ones presented in this chapter. The results can be summarized as in Table 2.2. There, in [9], K is the maximum number of zeroes per row in the constraints. In [43, 48] C is the largest entry in the objective function and the upper bounds for the variables. In [148, 150], d is the maximum number of constraints any variable appears in. In [149], λ^* is the optimal value and ρ' is an instance-dependent parameter. N is the number of constraints. In [103, 104], K'

is the number of nonzero entries in the constraint matrix.

Historically speaking, the first results were obtained for *pure packing and covering problems*. These are important classes of problems, which can be solved relatively well by approximation algorithms. More precisely these are the optimization problems

$$\text{compute } x \in B \text{ such that } f(x) \leq (1+\epsilon)\lambda_{\mathcal{P}}^* e \tag{P_ϵ}$$

$$\text{compute } x \in B \text{ such that } g(x) \geq (1-\epsilon)\lambda_{\mathcal{C}}^* e \tag{C_ϵ}$$

where $f, g : B \to \mathbb{R}_+^M$ are vectors of M continuous functions, which are convex and concave, respectively, and are defined on a polytope $B \subseteq \mathbb{R}^N$. Furthermore f and q are required to be nonnegative; $e \in \mathbb{R}^M$ denotes the constant unit vector. Finally

$$\lambda_{\mathcal{P}}^* := \min\{\lambda_{\mathcal{P}}(x) | x \in B\} \quad \text{and} \quad \lambda_{\mathcal{C}}^* := \max\{\lambda_{\mathcal{C}}(x) | x \in B\}$$

are the optima of the objective functions

$$\lambda_{\mathcal{P}} : B \to \mathbb{R}_+^M, \quad x \mapsto \max\{f_m(x) | m \in \{1, \ldots, M\}\}$$

$$\lambda_{\mathcal{C}} : B \to \mathbb{R}_+^M, \quad x \mapsto \min\{g_m(x) | m \in \{1, \ldots, M\}\}$$

and via these optimization problems one can algorithmically solve the corresponding feasibility problems. In these we have to test whether there is a point $x \in B$ such that $\lambda_{\mathcal{P}}$ or $\lambda_{\mathcal{C}}$, respectively, are smaller or larger, respectively, as a given parameter.

In contrast to these, one can study the so-called *mixed problems*; these are feasibility problems which feature *both* types of constraints, which makes the problem harder to solve. More precisely here we are given $f, g : B \to \mathbb{R}_+^M$ as above, furthermore nonnegative vectors $a, b \in \mathbb{R}_+^M$ and can formulate the problem

$$\begin{gathered}\text{compute } x \in B \text{ such that } f(x) \leq (1+\epsilon)a \text{ and } g(x) \geq (1-\epsilon)b \\ \text{or decide that } \{x | x \in B, f(x) \leq a, g(x) \geq b\} \text{ is empty}\end{gathered} \tag{MPC_ϵ}$$

which is a feasibility problem. Note that (MPC_ϵ) is already relatively general since it includes linear programs with nonnegative coefficients; in total, all these types of problems include settings where f and g are linear. These linear cases of the aforementioned problems are particularly important because they occur in the modelization of different practically motivated problems from combinatorial optmization. Particularly interesting

Table 2.2.: Algorithmic results concerning packing, covering, and mixed problems. In [9], K is the maximum number of zeroes per row in the constraints. In [43, 48] C is the largest entry in the objective function and the upper bounds for the variables. In [148, 150], d is the maximum number of constraints any variable appears in. In [149], λ^* is the optimal value and ρ' is an instance-dependent parameter. N is the number of constraints. In [103, 104], K' is the number of nonzero entries in the constraint matrix.

Problem	Coordination complexity	Reference	Remark
(P_ϵ)	$O(M\epsilon^{-2} + M \ln M)$	[79]	
(P_ϵ)	$O^*(\epsilon^{-1}\sqrt{KN \log M})$	[9]	quadratic block problem
(P_ϵ)	$O^*(\epsilon^{-1}\sqrt{KN})$	[10, 11]	quadratic block problem
(P_ϵ)	$O(\sqrt{N \ln M}(\log\log\sqrt{N} + \epsilon^{-1}))$	[25]	non-linear block problem
$(P_{max,\epsilon})$	$O(K' + (M+N)\log(K')\epsilon^{-2})$	[103, 104]	runtime bound met only with high probability
$(P_{c,\epsilon})$	$O(M \ln M + M\epsilon^{-2} \ln \epsilon^{-1})$	[79]	general block solver
$(P_{c,\epsilon})$	$O(M \ln M + M\epsilon^{-3} \ln c + M\epsilon^{-2})$	[74, 75]	general block solver
(P_{max}, ϵ)	$\tilde{O}(M\epsilon^{-2} \ln M)$	[49, 50]	
(P_{max}, ϵ)	$O(M\epsilon^{-2} \ln M)$	[96]	
$(C_{c,\epsilon})$	$O(M \ln M + M\epsilon^{-2} \ln \epsilon^{-1})$	[69, 67]	general block solver
$(C_{c,\epsilon})$	$O(c\rho' \ln M/(\lambda^*\epsilon^2))$	[149]	general block solver
(C_ϵ)	$O(M\epsilon^{-2} + M \ln M)$	[54]	
$(C_{min,\epsilon})$	$O(M\epsilon^{-2} \log(MC))$	[43]	with *box constraints*
$(C_{min,\epsilon})$	$O(M\epsilon^{-2} \log M + \min\{N, \log\log C\})$	[48]	
$(C_{min,\epsilon})$	$O(M\epsilon^{-2} \ln M)$	[96]	with *box constraints*
$(C_{min,\epsilon})$	$O(K' + (M+N)\log(K')\epsilon^{-2})$	[103, 104]	runtime bound met only with high probability
(MPC_ϵ)	$O(Md \log M\epsilon^{-2})$	[148, 150]	special case $B = \mathbb{R}_+^N$
(MPC_ϵ)	$O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$	[66, 70]	
(MPC_ϵ)	$O(M \ln M + M\epsilon^{-2} \ln \epsilon^{-1})$	[32, 33] this chapter	
(MPC_ϵ)	$O(M^2(\log^2 \rho)\epsilon^{-2} \log(\epsilon^{-1}M \log \rho) \log \rho)$	[124]	
(MPC_ϵ)	$O(M\epsilon^{-2} \ln M)$	[96]	block problem is on-line prediction problem
$(MPC_{c,\epsilon})$	$O(M \ln M + M\epsilon^{-2} \ln \epsilon^{-1})$	[32, 33] this chapter	general block solver
$(MPC_{c,\epsilon})$	$O(M\epsilon^{-2} \ln(M\epsilon^{-1}))$	this chapter	general block solver

are the linear programs

$$\text{compute } h^* := \max\{r(x)|g(x) \leq e, x \in \mathbb{R}_+^N\} \tag{P_{max}}$$

$$\text{compute } h^* := \min\{r(x)|f(x) \geq e, x \in \mathbb{R}_+^N\} \tag{C_{max}}$$

where $r : \mathbb{R}_+^N \to \mathbb{R}$ is a linear objective function. If the optimum h^* is positive, approximation algorithms for the problems (P_ϵ) and (C_ϵ) can then be used to solve (P_{max}) and (C_{max}) approximately. These are termed as *linear packing problem* and *linear covering problem* or alternatively as *fractional packing* and *fractional covering*. Likewise, we use $(P_{max,\epsilon})$ and $(C_{max,\epsilon})$ to denote the approximate versions of these problems.

As mentioned before, the type of algorithms obtained in this chapter fit the approaches pursued by Grigoriadis, Khachiyan, Porkolab, & Villavicencio [53, 54]; refining their basic approach, Jansen & Zhang later obtained results for related problems [66, 67, 69, 70, 74, 75]. Later Bienstock & Iyengar [9, 10, 11] studied an approach where in each iteration a quadratic problem has to be solved. In the approach pursued by Chudak & Eleutério [25], the problem to be solved in each iteration is also non-linear. Futhermore, Garg & Könemann [49, 50] studied fractional packing problems motivated from multicommodity flow problems; in their approach is similar to that of Young [148, 150]. Furthermore, Fleischer [43] and Garg & Könemann [48] studied the case of linear programs with nonnegative variables which are suitably bounded from above (so called *box constraints*). Plotkin, Shmoys & Tardos [124] proposed algorithms for the mixed problem where the coordination complexity depends on an instance parameter, namely the width. Finally, Khandekar [96] studied pure packing and covering problems as well as the mixed problem; the algorithms in [96] are based on the so-called *on-line prediction problem*.

However, all of these results are approximate algorithms which typically result in column generation algorithms which greatly depend on the block solver necessary for the respective model. In contrast to these, the same or similar problem formulations for the linear cases where B is a simplex and f and g are linear, could be solved with one of the various exact methods. The most prominent of these is the Simplex algorithm [122], which does not yield a polynomial runtime bound, however, due to the worst-case example by Klee & Minty [122]. For practical application however, note that the running time of the Simplex algorithm on average is polynomially bounded [13, 14]. The Ellipsoid algorithm proposed by Khachiyan [94, 95] was the first algorithm for linear programming yielding a polynomial runtime bound. However, we like to point out that the

model of computation used there requires square roots to be computed exactly, which causes some implementational difficulties. We give a brief summary of running times for these as follows, where N denotes the number of variables and M denotes the number of constraints; furthermore, L is a bound for the encoding length of the instance.

In [55], Grötschel et al obtain a runtime bound of $O(MN^3L)$. Furthermore, in [140], Vaidya obtained a runtime bound of $O(((M+N)N^2+(M+N)^{1.5}NL)$. Finally, Karmarkar [83, 84] obtained a runtime bound of $O(((M+N)N^2+(M+N)^{1.5}NL\sqrt{M+N})$. These results are also summarized in Table 2.1. Concerning these exact methods, we refer the reader to the textbook [128] which deals with linear and nonlinear programming as a main focus or to the textbooks [55, 101, 122] which discuss linear programming in the context of other problems from combinatorial optimization.

3. Implementation of Max-Min Resource Sharing

In this chapter we implement an algorithm for the max-min resource sharing problem using a new line search technique to find a suitable step length. Our approach uses a modified potential function that is less costly to evaluate, thus heuristically simplifying the computation. Observations concerning the quality of the dual solution and oscillating behavior of the algorithm are made and numerical observations are discussed. We study a certain class of linear programs, namely the computational bottleneck of an algorithm for approximately solving *strip packing* with an approach based on LP models. For these, we obtain practical running times. Our implementation is able to solve instances for small accuracy parameters $\epsilon \in (0, 1)$ for which the methods proposed in theory are out of practical interest. More precisely, the algorithm used here improves the known runtime bound of

$$O(M^6 \ln^2(Mn/(at)) + M^5 n/t + \ln(Mn/(at)))$$

to the more favourable runtime bound

$$O(M(\epsilon^{-2} + \ln M) \max\{M + \epsilon^{-3}, M \ln \ln(M\epsilon^{-1})\}),$$

where n denotes the number of items, M the number of distinct item widths, a the width of the narrowest item and t is a desired additive tolerance.

3.1. Introduction

In this chapter we implement an algorithm by Grigoriadis et al. [54] to approximately solve the so-called *max-min resource sharing problem*. We study the behaviour of our implementation on artificial instances as well as in the context of an asymptotic approximation scheme for the strip packing problem. For our approach to the strip packing

problem, the algorithm under consideration is used as a subroutine to approximately solve an underlying configuration LP.

First we introduce thet the max-min resource sharing problem. Here we are given a non-empty convex set $B \subseteq \mathbb{R}^N$ and a set of M non-negative concave functions $f_i(x) : B \to \mathbb{R}_+$, and the goal is to find a vector $x \in B$ that maximizes the value $\min\{f_1(x), \ldots, f_M(x)\}$. Note that in the case in which B is a simplex and the functions f_i are linear, the problem can be solved via linear programming. However, we furthermore assume the existence of a suitable subroutine to solve a (simpler) optimization problem on B, which is termed the *block problem*; this subroutine is called the *block solver*. With the help of the subroutine, we will be able to iteratively obtain a $(1-\epsilon)$-approximate solution based on Lagrangian decomposition.

For a given relative error $\epsilon \in (0, 1]$, a vector $x \in B$ is a $(1-\epsilon)$-approximate solution for the max-min resource sharing problem if $f_i(x) \geq (1-\epsilon)\lambda^*$ holds, where

$$\lambda^* := \max\{\lambda \mid f(x) \geq \lambda e, x \in B\}$$

is the actual optimum of the instance. The algorithm is based on the so-called Lagrangian or *price-directive* decomposition method and computes a sequence of vectors in B to iteratively approximate an optimal solution. One such iteration can be described as follows.

1. Use the current iterate $x \in B$ to compute $p(f(x)) := (p_1(f(x)), \ldots, p_M(f(x)))$, the so-called *price vector*.

2. Compute a solution $\hat{x} \in B$ of the block problem using the price vector $p(f(x))$; here this price vector is used to govern the direction of optimization.

3. Move the current iterate from x to $(1-\tau)x + \tau\hat{x}$, where $\tau \in (0, 1]$ is a suitable step length.

Each such iteration is called a *coordination step*. The primary measure of the algorithm here is the number of coordination steps, i.e. the number of calls to the block solver. This measure for the running time is termed the *coordination complexity*.

More precisely, the associated *block problem* is to compute

$$\Lambda(p) := \max\{p^T f(x) \mid x \in B\}$$

given the price vector $p := p(f(x))$. We suppose that there is an approximate block

solver $ABS(p,t)$ that for any

$$p \in P := \{p \in \mathbb{R}^M_+ \mid e^T p = 1\}$$

and given tolerance $t \in (0,1)$ computes $\hat{x} := \hat{x}(p) \in B$ such that

$$p^T f(\hat{x}) \geq (1-t)\Lambda(p)$$

holds. In total, we study an implementation of the algorithm described in [54] that, provided the existence of $ABS(p,t)$, finds for any $\epsilon \in (0,1)$ a solution to the following problem.

$$\text{compute } x \in B \text{ such that } f(x) \geq (1-\epsilon)\lambda^* e \qquad (R_\epsilon)$$

As an application, we study the *strip packing* problem. Here we are given a list L of n rectangular items. As a target area we are given a strip $[0,1] \times [0,\infty)$ of width 1 and infinite height. We are interested in non-rotational non-overlapping arrangements of the items of L ito the strip; the objective is to minimize the toal height of the packing, i.e. the highest part of the strip covered by an item. The approach from [71, 92] involves an interesting configuration LP which we solve with the proposed algorithm.

The remainder of this chapter is organized as follows. In Section 3.2 we present in more detail the algorithm from [54] that we have implemented and in Section 3.3 we comment on our implementation. We present some observations concerning performance in Section 3.4. In Section 3.5 we present results from [92], their modification from [71] as well as some computational results. Finally, we conclude in Section 3.6.

3.1.1. Previous Results and Related Problems

In [124] the authors considered the linear feasibility variant of the fractional covering problem (with linear functions) which is to find an $x \in B$ such that

$$f(x) = Ax \geq (1-\epsilon)b$$

where $A \in \mathbb{R}^{M \times N}$ and b is an M-dimensional positive vector. The problem is solved there via Lagrangian decomposition using exponential potential reductions. The number of iterations (calls to the corresponding block solver) here is $O(M + \rho \ln^2 M + \epsilon^{-2}\rho \ln(M\epsilon^{-1}))$, where

$$\rho := \max_{1 \leq m \leq M} \max_{x \in B} a_m^T x / b_m$$

is the *width* of B relative to $Ax \geq b$. In [100] the author proposed an algorithm for the fractional covering problem which uses $O(M\bar{\rho}\log_{1+\epsilon}(\epsilon^{-1}))$ iterations where $\bar{\rho}$ is also a data dependent bound. In [149], the fractional covering problem is studied with general approximate block solvers $ABS_c(p,t)$ which compute an $\hat{x} \in B$ such that

$$p^T f(\hat{x}) \geq (1-t)\Lambda(p)/c$$

(with arbitrary ratio $c \geq 1$). Young proposed an algorithm with $O(c\rho' \ln M/(\lambda^*\epsilon^2))$ calls to the block solver for the fractional covering problem, where

$$\rho' := \max_{1\leq m\leq M} \max_{x\in B} a_m^T x/b_m - \min_{1\leq m\leq M} \min_{x\in B} a_m^T x/b_m$$

and λ^* is the optimum value of the fractional covering problem, respectively. The first big step towards the general max-min resource sharing problem was done in [54]. The authors proposed an algorithm for this problem with standard block solvers (with approximation ratio $c = 1$) that uses only $O(M(\epsilon^{-2} + \ln M))$ calls to the block solver, which is a bound that does neither depend on the width ρ nor on the optimal value λ^*. The closely related fractional covering problem was also studied by Khandekar [96]; he proposed an algorithm which needs $O(M\epsilon^{-2}\ln M)$ iterations, each of which requires a call to a suitable optimization procedure. However, his approach is not amenable to line search; it is unclear whether the algorithm can be accelerated by taking larger steps. In [74, 76] the authors studied the general max-min resource sharing (and also fractional covering) problem with general approximate block solvers which permit an arbitrary ratio $c \geq 1$. They proposed an approximation algorithm that uses at most $O(M(\ln M + \epsilon^{-2} + \epsilon^{-3}\ln c))$ iterations, which is a bound that depends on the approximation ratio c. In [69] the author found an approximation algorithm which needs $O(M(\ln M + \epsilon^{-2}\ln(\epsilon^{-1})))$ iterations. Closely related problems are the fractional packing problem [22, 50, 124, 149] and min-max resource sharing problem [51, 53, 80, 143]; these problems have been studied with approaches similar to [54]. Experimental results have been obtained in [8] where the choice of the step length τ is regarded as an essential decision. Similar results can be found in [80, 117]. In [117] the authors implemented the algorithm from [80] for experimentation. More computational results concerning network problems are found in [15]. Furthermore, in [7] the authors have experimented with an implementation for solving maximum multicommodity flow problems. In [138] similar experimental results can be found, where building on work from [117] and modification of the step length also has heuristically improved running time. All of these

are applications of solvers for the min-max resource sharing problem, however. Furthermore, in [10, 11] a different approach is used; here, a fewer number of iterations is obtained by approximating a quadratic program in each iteration. For the fractional packing problem, Chudak & Eleutério [25] found an algorithm with coordination complexity $O(\sqrt{N \ln M}(\log\log\sqrt{N}+\epsilon^{-1}))$; here, in each iteration a non-linear block problem is solved as well. Koufogiannakis & Young [103, 104] studied the fractional packing and covering problems and proposed an algorithm which – with high probability – in time $O(n+(r+c)\log(n)\epsilon^{-2})$ generates a near-optimal solution; here n is the number of non-zeros in the constraint matrix while r and c denote the number of rows and columns, respectively. In addition, the *mixed* problem has been studied; this is a feasibility problem which is amenable to approximation via various techniques — see [33, 114, 148, 150] for various results.

3.1.2. Applications

Applications of the max-min resource sharing problem can be found in [8, 12, 20, 68, 76, 92, 105, 106, 120, 124, 134, 150]; the model can be used for scheduling problems, path coloring problems, and fractional coloring problems in unit disk graphs. These combinatorial optimization problems can be modelled as a max-min resource sharing problem with an exponential number N of variables and a polynomial number M of constraints, which typically result in column generation algorithms. In these applications the block problem is hard to solve or to approximate but can be approximated by a (general) approximate block solver. The running time of these algorithms is dominated by the product of the number of iterations and the running time of the approximate block solver.

3.1.3. New Contributions

We implement the algorithm for the max-min resource sharing problem described in [54]. Our implementation uses a simplified line search technique for determining a suitable step length. For solving special cases of linear programs (namely relaxations of *strip packing*) the runtime bound of

$$O(M^6 \ln^2(Mn/(at)) + M^5 n/t + \ln(Mn/(at)))$$

from the approach in [85, 102] is improved to

$$O(M(\epsilon^{-2} + \ln M) \max\{M + \epsilon^{-3}, M \ln \ln(M\epsilon^{-1})\});$$

here n denotes the number of items, M is the number of distinct item widths, a is the width of the narrowest item and t is the desired additive tolerance. In the context of strip packing, we discuss the behaviour of our algorithm on randomly generated instances as well as instances from the literature as found in [61].

The improved running time is due to an algorithm from [71], which has been fine-tuned to yield practical running times. Observations concerning the quality of the dual solution and oscillating behavior of the algorithm are made and numerical results for random instances are briefly discussed.

3.2. Algorithm Description

Here we briefly review the implemented algorithm; however for a detailed analysis, we refer the reader to [54, 71]. As sketched above, we study the following optimization problem.

$$\text{maximize } \lambda \text{ such that } \lambda \leq \min\{f_i(x) : i \in \{1, \ldots, M\}\}, \quad x \in B \tag{R}$$

As mentioned before, $B \subseteq \mathbb{R}^N$ is a nonempty convex compact set and $f_i : B \to \mathbb{R}_+$ is a nonnegative convex function on B for each $i \in \{1, \ldots, M\}$. Let

$$f(x) := (f_1(x), \ldots, f_M(x))^T$$

for each $x = (x_1, \ldots, x_N) \in B$. Furthermore let

$$\lambda^* := \max\{\lambda : f(x) \geq \lambda e, x \in B\}$$

where $e = (1, \ldots, 1)^T \in \mathbb{R}_+^M$ denotes the constant unit vector. We want to compute a $(1 - \epsilon)$-approximate solution to the problem (R), i.e. for any error parameter $\epsilon \in (0, 1)$ we want to solve the following approximate version.

$$\text{compute } x \in B \text{ such that } f(x) \geq (1 - \epsilon)\lambda^* e \tag{R_ϵ}$$

In order to solve this approximate max-min resource sharing problem we consider the block problem

$$\text{compute } \Lambda(p) := \max\{p^T f(x) : x \in B\}$$

for $p \in P$; here the block problem is the maximization of a linear function over B. We suppose the existence of an *approximate block solver* which solves

$$\text{compute } \hat{x} := \hat{x}(p) \text{ such that } p^T f(\hat{x}) \geq (1-t)\Lambda(p) \qquad (ABS(p,t))$$

where $t \in (0,1)$ is an accuracy parameter. This means that the block solver $ABS(p,t)$ is a family of approximation algorithms for the block problem with approximation ratio $1/(1-t)$. By duality we have

$$\lambda^* = \max_{x \in B} \min_{p \in P} p^T f(x) = \min_{p \in P} \max_{x \in B} p^T f(x).$$

This implies $\lambda^* = \min\{\Lambda(p) : p \in P\}$. Based on this equality we can define the problem of finding a $(1+\epsilon)$-approximate solution to the dual problem as follows.

$$\text{compute } p \in P \text{ such that } \Lambda(p) \leq (1+\epsilon)\Lambda^* \qquad (D_\epsilon)$$

Next we discuss the approximation algorithm for the max-min resource sharing problem from [54]. The goal is to compute a solution with objective value at least $(1-\epsilon)\lambda^*$, provided that there is the aforementioned block solver $ABS(p,t)$. The algorithm solves both (R_ϵ) and (D_ϵ) in $O(M(\ln M + \epsilon^{-2}))$ iterations; each iteration requires a call to $ABS(p, \Theta(\epsilon))$ and a coordination overhead of $O(M \ln\ln(M\epsilon^{-1}))$ operations for numerical computations. The algorithm is based on the *logarithmic potential function*

$$\Theta_t(\theta, f(x)) = \ln\theta + \frac{t}{M}\sum_{m=1}^{M} \ln(f_m(x) - \theta),$$

where $\theta \in \mathbb{R}$,

$$f(x) = (f_1(x), \ldots, f_M(x))^T$$

is the function value for a vector $x \in B$, and t, which depends on ϵ, is a tolerance parameter. For $\theta \in (0, \lambda(f(x))$, where $\lambda(f(x)) = \min\{f_1(x), \ldots, f_M(x)\}$, the function Φ_t is well defined. The maximizer $\theta(f(x))$ of the function $\Theta_t(\phi, f(x))$ is given by the

first order optimality condition

$$\frac{t\theta}{M}\sum_{m=1}^{M}\frac{1}{f_m(x)-\theta}=1$$

which can be seen by calculating the derivation of the right hand side with respect to θ. This equality has a uniquely determined root since the function

$$g(\theta)=\frac{t\theta}{M}\sum_{m=1}^{M}\frac{1}{f_m(x)-\theta}$$

is strictly increasing. The price vector $p=p(f(x))$ for a fixed $f(x)$ is defined by

$$p_m(f(x))=\frac{t}{M}\frac{\theta(f(x))}{f_m(x)-\theta(f(x))}$$

for each $m\in\{1,\ldots,M\}$. It can be shown that

$$p(f(x))=(p_1(f(x)),\ldots,p_M(f(x)))\in P$$

and that $\lambda(f(x))$ approximates $\theta(f(x))$; for details, see [54, 71].The vector $p(f(x))$ is used in the block solver $ABS(p(f(x)),t)$ as the next direction for optimization. On one hand, if f_m is much larger than $\theta(f(x))$, then $p_m(f(x))$ is close to 0; on the other hand, if f_m is close to $\theta(f(x))$, then $p_m(f(x))$ is close to 1.

Let $\phi_t(f(x))=\Phi_t(\theta(f(x)),f(x))$ be the reduced potential value. Now we define a parameter $\nu=\nu(x,\hat{x})$ by

$$\nu(x,\hat{x})=\frac{p^Tf(\hat{x})-p^Tf(x)}{p^Tf(\hat{x})+p^Tf(x)},$$

where $p=p(f(x))\in P$ and $\hat{x}\in B$ is an approximate block solution generated by $ABS(p(f(x)),t)$. Note that $\nu(x,\hat{x})\leq 1$. In [54, 71] it is proved that if $\nu(x,\hat{x})\leq t$ for $t=\epsilon/6$ (the stopping criterion), then x solves the primal problem (R_ϵ) and the price vector $p(f(x))$ solves (D_ϵ). Now the main algorithm can be descibed as follows.

The algorithm is a direct implementation of the Lagrangian decomposition scheme. The algorithm starts with an initial vector

$$x=x^0=\frac{1}{M}\hat{x}^{(m)}$$

where $\hat{x}^{(m)}$ is the solution of $ABS(e_m,1/2)$ and e_m is the unit vector with all zero

1. Compute initial solution $x^{(0)}$, $s := 0$, $\epsilon_0 := 1/4$;
2. Repeat {scaling phase}
   2.1. $s := s + 1$; $\epsilon_s := \epsilon_{s-1}/2$; $t = \epsilon_s/6$; $x := x^{(s-1)}$;
   2.2. While *true* do begin {coordination phase}
      2.2.1. compute $\theta(x)$ and $p(x)$;
      2.2.2. $\hat{x} := ABS(p(x), t)$;
      2.2.3. compute $\nu(x, \hat{x})$;
      2.2.4. If $\nu(x, \hat{x}) \leq t$ then begin $x^{(s)} := x$; break; end;
      2.2.5. compute step length τ and set $x := (1 - \tau)x + \tau\hat{x}$;
      end;
   2.3. until $\epsilon_s \leq \epsilon$;
3. Return $x^{(s)}$.

Figure 3.1.: Approximation algorithm for the min-max resource sharing problem.

coordinates except its m-th component which is set to 1. Then, the algorithm moves the iterate from x to $(1 - \tau)x + \tau\hat{x}$ where $\hat{x}$ is the solution of $ABS(p(f(x)), t)$ until the stopping criterion $\nu(x, \hat{x})$ is satisfied. The outermost loop embeds this approach into scaling phases in order to improve the running time, see [54, 71] for a detailed analysis. There, it is also shown that this algorithm solves both (R_ϵ) and (D_ϵ) in $O(M(\ln M + \epsilon^{-2}))$ iterations. In order to prove this results, one needs to show the following.

1. The reduced potential values are monotonically increasing from one vector x to the next vector $x' := (1 - \tau)x + \tau\hat{x}$, i.e. $\phi_t(f(y)) - \phi_t(f(x))$.
2. There is an upper bound for the difference $\phi_t(f(y)) - \phi_t(f(x))$ for any two vectors $x, y \in B$ with $\lambda(f(x)) > 0$ and $\lambda(f(y)) > 0$.

The step length τ is chosen as

$$\tau := \frac{t\theta(f(x))\nu(x, \hat{x})}{2M(p^T f(\hat{x}) + p^T f(x))}$$

to prove property 1 above for the reduced potential values.

Although the implementation using scaling phases is more an implementational detail, we like to point out that the algorithm without the scaling phases has a larger runtime bound of $O(M\epsilon^{-1}(\ln M + \epsilon^{-1}))$. The basic idea behind the scaling phase implementation

is to reduce the parameter t to the desired accuracy. In the s-th scaling phase we set $\epsilon_s := \epsilon_s/2$ and $t_s := \epsilon_s/6$ and use the current approximate solution x^{s-1} as the initial solution. For phase $s = 0$, we use the initial point $x^0 \in B$. For this point we have

$$p^T f(x^0) \geq \frac{1}{2M}\Lambda(p)$$

for each $p \in P$. We set $\epsilon_0 := (1 - 1/(2M))$. The initial solution satisfies

$$f_m(x^0) \geq \frac{1}{2M}\lambda^* = (1 - 1 + \frac{1}{2M})\lambda^* = (1 - \epsilon_0)\lambda^*$$

for each $m \in \{1, \ldots, M\}$.

Finally, the root $\theta(f(x))$ in general cannot be computed exactly. Therefore, we need a numerical overhead of $O(M \ln\ln(M\epsilon^{-1}))$ arithmetic operations per iteration to approximately compute $\theta(f(x))$. However, for further details, we refer to [54, 71].

Theorem 32. *For any given relative accuracy $\epsilon \in (0, 1)$ the algorithm in Figure 5.1 computes a solution x of (R_ϵ) in $O(M(\ln M + \epsilon^{-2}))$ coordination steps.*

3.3. Implementation

We have implemented the algorithm from Section 3.2 in C++. Our implementation uses abstract classes which need to be implemented for each specific application; the following types of problems have been tested.

1. Linear problems with $n = 1$ where the block vector is a number from an interval and the block solver returns a value near to one of the interval bounds as a block solution $\hat{x}$.

2. A multidimensional linear case implemented with CPLEX [65]. Vectors and functions are implemented using CPLEX data structures and the block problem is solved by the CPLEX optimizer. Input is read in a standard MPS format supported by CPLEX. This permits easy verification of results by running the input through a CPLEX optimizer that solves the max-min problem.

3. The fractional strip packing problem as described in [92, 71]. The block solver uses an FPTAS for the *unbounded knapsack* problem from [107, 90]; the results are presented in Section 3.5 and source code for this application is made available in the appendix.

Type 1 is a special case of type 2, but the simpler block solver allowed tests with more iterations quickly, while running a CPLEX block solver permits about 20 calls per second for small problems.

3.3.1. Choice of the Step Length

We use a line search to find a step length τ that maximizes the reduced potential ϕ_t instead of using (3.2). However, out of the values for τ as defined by (3.2) and the value of τ determined by our line search, we take the value which yields the larger change of the reduced potential. As a consequence, the number of iterations can never be larger than the one theoretically predicted by Theorem 32.

More precisely, we simplify ϕ_t in order to speed up its evaluation as follows. Let $x, \hat{x}, t$ be as in Step (2.2.5) and $\theta = \theta(x)$; for each $\tau \in (0,1)$ and $x' = (1-\tau)x + \tau\hat{x}$ we define $\tilde{\phi}_t$ as follows.

$$\tilde{\phi}_t(\tau) = \begin{cases} t/M \sum_{m=1}^{M} \ln(f_m(x') - \theta) & \text{if } \theta < f(x') \\ -\infty & \text{otherwise} \end{cases}$$

In the case that $\theta < f(x')$ it is thus simply $\tilde{\phi}_t(\tau) = \Phi_t(\theta, x') - \ln(\theta)$. As $\tilde{\phi}_t(\tau)$ is convex for $\theta < f(x')$, we can use binary search to approximate its maximum. We assume that f is linear and compute $f(x') = (1-\tau)f(x) + \tau f(\hat{x})$ which spares us the actual evaluation of $f(x')$ which could be expensive, e.g. if x' had many non-zero components. In particular we study the case where f is linear since this is satisfied for the application considered in Section 3.5. This search technique yields a dramatic improvement of the runtime, as shown in Section 3.4 and Section 3.5. The increase of ϕ_t in each step is usually several orders greater than the one theoretically predicted; see Subsection 3.4.3 and Subsection 3.5.2 for a numerical comparison of both methods. There are several advantages of using $\tilde{\phi}_t$ over calculating with $\phi_t(\tau) = \Phi_t(\theta(x'), x')$. Usage of the same θ for all evaluations of $\tilde{\phi}_t$ allows us to spare the expensive calculation of $\theta(x')$ in every step of the search. It also allows us to leave out the constant summand $\ln(\theta)$ of the potential function which improves the numerical quality; $\ln(\theta)$ often dominates the value of the potential function with the rest being relatively small. Computing with a smaller function makes the algorithm more sensitive to changes in the function. Concerning the quality of the solution, we observed that there is little difference between τ computed using ϕ_t and $\tilde{\phi}_t$.

We neglect the fact that performing search adds to numerical overhead, as the coordination complexity is of main interest; as mentioned before, the number of coordination

steps is the primary measure of complexity studied here since typically a call to the block solver is quite costly. In total, this means that the number of calls to the block solver should be as small as possible.

3.3.2. An Additional Stopping Rule

The stopping rule using $\nu(x, \hat{x})$ is based on a comparison of primal and dual points. We give an additional criterion that is based on the quality of the final iterate of the previous scaling phase (or initial solution in the first scaling phase), similar to [80]. We introduce the parameter

$$\omega_s = \begin{cases} 2M(1-\epsilon_1) & \text{for the first scaling phase} \\ (1-\epsilon_s)/(1-2\epsilon_s) & \text{for all other phases} \end{cases}$$

and terminate the current scaling phase as soon as $\lambda(x) \geq \omega_s \lambda(y)$, where x is the current iterate and y is the final iterate of the previous scaling phase (or initial solution in the first scaling phase). An analysis similar to [80] shows that the inequality implies that the last iterate of each phase has indeed the requested quality. We comment on this in Subsection 3.5.2.

3.4. Performance

Our tests with CPLEX for small values of M were carried out with the setting $B = [-100, 100]^n$, $\epsilon = 1/1000$ and exact block solvers returning vertices of B. We used linear functions f_m with uniformly random coefficients. The domains of the coefficient distributions were selected so that the values of the functions lie in $[0, 200]$. The tests were made with $\epsilon_0 = 10$ instead of $\epsilon_0 = 1/4$ as in [54]. This was done so that $t_1 = 5/6$ and the initial scaling phase would not begin with a too small t, causing slow progress. Observations presented in this subsection refer to both the algorithm from [54] and our modification where τ is determined by line search. Our primary goal was to test the dependence of the number of iterations on n and M under such conditions. We present some observations made during testing.

3.4.1. Improvement of the Dual Solution

We noticed that the primal solution computed by the algorithm from Section 3.2 often has a much higher precision than requested; the algorithm does not terminate as soon as the solution x reaches the desired precision but continues to iterate. Consider the instance

$$n = 1,\ M = 2,\ B = [-100, 100],\ f(x) = (x + 100, -x + 100) \tag{3.1}$$

where the optimum is $x^* = 0$ yielding $\lambda^* = \lambda(x^*) = 100$. Suppose that in some iteration $x = -0.1$ and $\hat{x} = 100$. The stopping rule gets satisfied if $\nu - t \leq 0$. Plotting $\nu - t$ against t like in Figure 3.2 shows that this is not the case for $t \leq 0.02$. However the value $\lambda(x)$ approximates $\lambda^* = 100$ with relative precision of 0.001, which is 20 times smaller. This means that in this case the termination criterion is too strict; although the termination criterion is not satisfied, the desired accuracy of the primal solution is already met. This is clearly a drawback of the algorithm, however it is an open problem whether or not this can be improved.

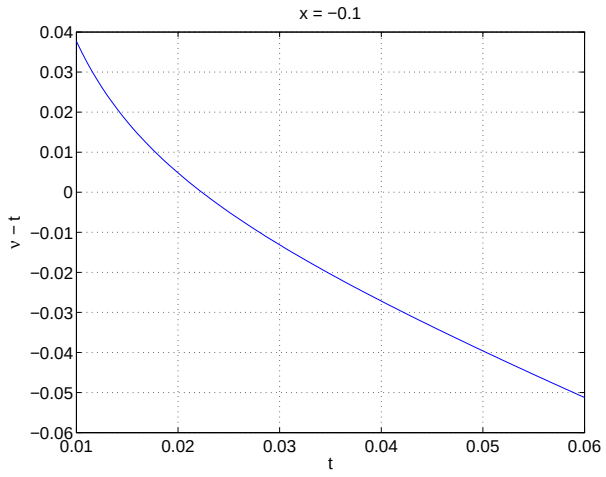


Figure 3.2.: Dependence of $\nu - t$ on t for $x = -1/10$.

Moreover the ratio of the minimal t for which the stopping rule is satisfied (in the following we shall call such t *critical*) to the actual precision of the current solution differs for different values x of the solution. This is illustrated by the plot of critical t in Figure 3.3.

This behavior can be explained by the fact that ν depends on the quality of the dual solution in addition to the quality of the primal solution. Furthermore the dual solution is often improving slower than the primal solution. As a way to measure the

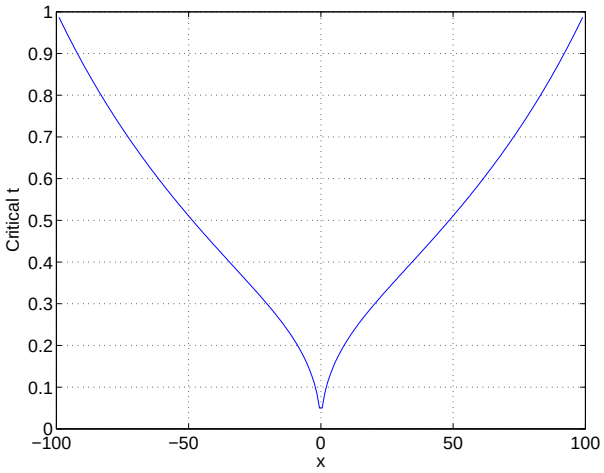


Figure 3.3.: Dependence of critical t on x.

quality of the dual solution let us consider the error $\hat{l}$ with which $p^T f(\hat{x})$ approximates λ^*, that is $\hat{l} = 1 - \lambda^*/p^T f(\hat{x})$. This expression evaluated over B (with $\lambda^* = 100$ and $\hat{x} = -100 \cdot \text{sign}(x)$) is shown in Figure 3.4, whereas critical t was used in calculation of p. Using values of t smaller than the critical t results in even worse quality of the dual solution. The behavior described above leads to the fact that primal solutions get calculated to a precision which is several orders higher than the precision actually requested. This effect has been observed in all instances we tested with our algorithm. This is a drawback if one is interested only in the primal solution, which is often the case.

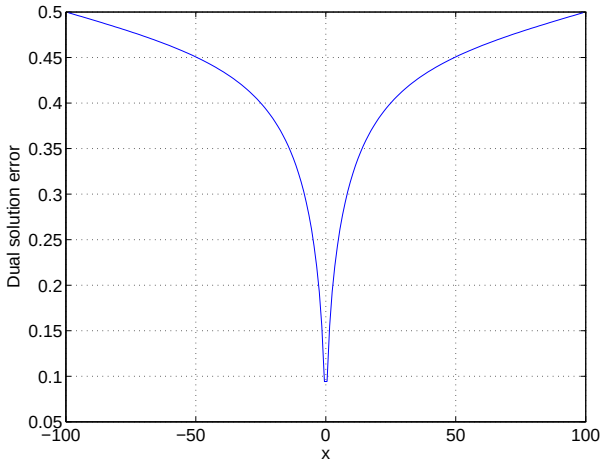


Figure 3.4.: Error of the dual solution.

3.4.2. Oscillations

A typical behavior of the algorithm from Section 3.2 is that the current solution x oscillates around some value that very slowly converges to the optimum. One such example with $n = 2$ and $M = 10$ is shown in Figure 3.5; there, a contour plot of the objective function $\lambda(x)$ for a part of the instance is presented. The zig-zagging line indicates the positions of the iterates x of the algorithm; more precisely, points connected by segments correspond to the sequence of iterations generated by the algorithm. Points marked with a circle are those in which the tolerance t gets decreased, i.e. a new scaling phase begins. Notice how the amplitude of the oscillations decreases together with t. In total, we remark that by no means the algorithm optimizes in the direction of the actual optimum in a straightforward way. Conversely, the iterations oscillate and only the general direction of the iterates is towards an optimal point. The same behaviour is documented in Figures 3.6 and Figure 3.7. In all of these figures, $n = 2$ indicates the dimension and M indicates the number of functions in the objective.

Such oscillating behavior occurs when block solutions produced by the block solver lead the algorithm in a direction different from the one in which the potential ϕ_t mainly grows. Small steps are made so that ϕ_t would not get reduced and thus the algorithm makes a lot of iterations while trying to follow the growth of ϕ_t. The interesting fact is that following the growth of ϕ_t often has very little to do with approaching the actual solution. Consider the following example.

$$n = 2,\ M = 3,\ B = [-100, 100] \times [0, 200]$$
$$f_1(x, y) = x + 100, \quad f_2(x, y) = -x + 100, \quad f_3(x, y) = y$$

The optimal set is $N = \{\, x^* \in B \mid \lambda(x^*) = \lambda^* \} = \{0\} \times [100, 200]$ and the iterations are shown in Figure 3.7. The algorithm crosses N in each iteration, but instead of stopping there the search for maximal ϕ_t takes it further resulting in oscillating behavior. This is caused by the function f_3, which plays a significant role in calculating ϕ_t even for $y > 100$. As $f_3(x, y)$ grows with y, so does ϕ_t thus effectively misleading the algorithm, which then tries to increase y more than necessary in each iteration. It is worth noting that oscillations also occur when θ is computed exactly in step length search (see Section 3.3.1) or when fixed step length as in (3.2) is used. We call functions like f_3 above for which $f(x) > \lambda(x)$ but which still play a significant role in determining ϕ_t *shadow functions*. Oscillations described here actually occur in all nontrivial instances with $n = 2$ that we tested so far. By nontrivial we mean that the solution does not lie in a vertex of

B (otherwise it is usually reached within about 10 steps). The described behavior was also observed in cases with $n \gg M$, there however the oscillating pattern is not so distinct. Still it seems that cases with $n \gg M$ are less subject to the problem described in this section. In our tests 10 random instances with $n = 2000$ and $M = 10$ got solved in under 200 iterations, whereas 9 of 10 instances with $n = 4$ and $M = 10$ caused a timeout (a case in which we stopped the algorithm) with more than 900 iterations. One reason for the oscillations is the fact that here the optima of the block problems are attained at vertices of B; hence, the algorithm also optimizes only in the direction of vertices. In total, the role that is played by the oscillations in large instances needs further investigation.

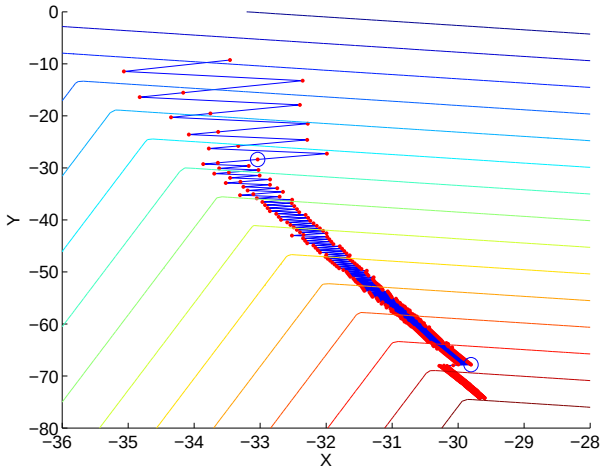


Figure 3.5.: Oscillations, $n = 2$ and $M = 10$.

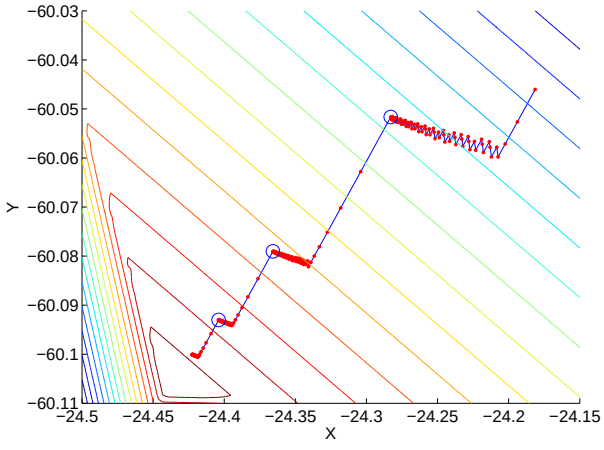


Figure 3.6.: Oscillations, $n = 2$ and $M = 10$.

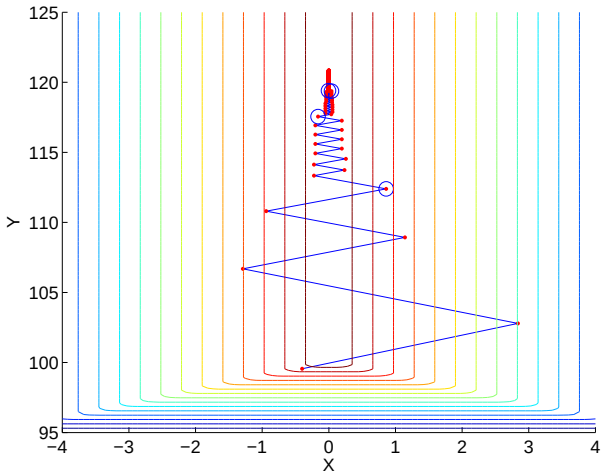


Figure 3.7.: Oscillations, $n = 2$ and $M = 3$.

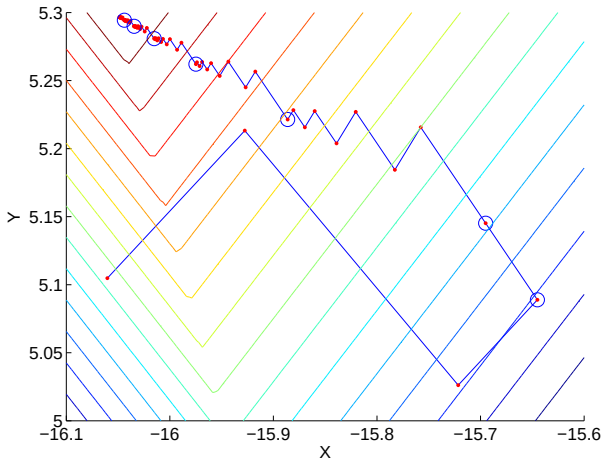


Figure 3.8.: Oscillations, $n = 2$ and $M = 3$.

3.4.3. Numerical Results

In this subsection we discuss the different choices of τ and the effect of M on the running time.

Comparison of Strategies for Step Length Choice

We compared the running time of the algorithm with line search for determining τ to the runtime of the algorithm that uses fixed step length from (3.2). Using our quick block solver for the case $n = 1$ we tested random instances with 4 different values of M, setting $\epsilon = 1/100$. For each M we took the mean of the coordination complexity over 20 instances. The results for line search (rounded to nearest integer) are shown in Table 3.1. For fixed step size most of the instances had a timeout with tens of thousands of iterations. In case of $n > 1$ the version with fixed step length timeouts with more than 1000 iterations on almost all instances, whereas the version with line search often can solve them in less than 10 iterations.

Table 3.1.: Coordination complexity for line search, $n = 1, \epsilon = 1/100$.

M	Mean	Standard deviation
2	9	7
10	16	6
100	34	20
1000	116	127

Dependence of Runtime on M

We performed a series of tests with fixed $n = 500$ and 18 values of M ranging from 2 to 170. For each value of M we took average coordination complexity over 10 tests. The results did not reveal any certain dependence of coordination complexity on M — the standard deviation is too high. This is due to the fact that some tests are solved almost instantly (within under 10 iterations) which usually happens when the solution lies in a vertex of the block. However, some single tests need thousands of iterations. This probably happens when oscillations with small step length occur. For plots of mean coordination complexities and standard deviations see Figure 3.9. We are currently performing bigger tests with more instances to reveal the dependence on M.

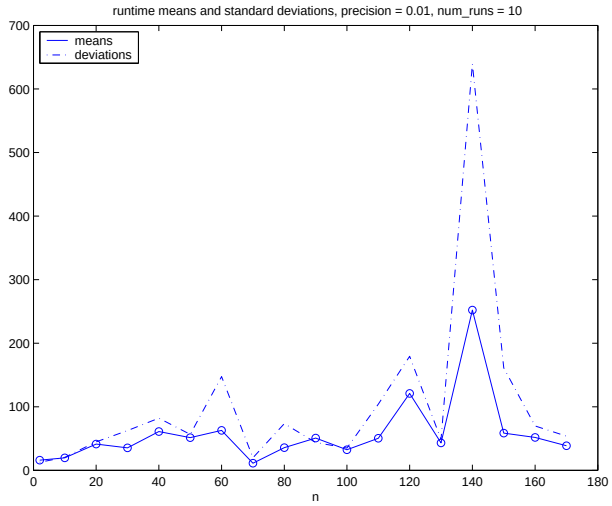


Figure 3.9.: Plot of mean coordination complexities and standard deviations.

3.5. Application for Strip Packing

We used our implementation to solve the computational bottleneck of an approximation algorithm for *strip packing* described below using an approach from [71] and [92]; we proceed with experimental results. Strip packing is a two-dimensional geometrically constrained packing problem which has been approached with various techniques. The problem was first studied in [5, 26] and the currently best known *absolute* approximation ratio 2 was obtained with different combinatorial approaches in [133, 136], where especially the algorithm from [136] is often used as a building block for algorithms for other packing geometrically constrained packing problems like 2D knapsack [78], 3D strip packing and 3D knapsack [31]. Concerning *asymptotic* approximation ratios, in [4] bound of 5/4 was obtained. The currently best known asymptotic performance bound of $T_\infty \approx 1.69$ was obtained in [6], where T_∞ denotes the Harmonic number. Furthermore, in [147] an on line version of the problem is discussed. Finally, the strip packing algorithm presented here is used in [77] as a building block for an approximation algorithm for 3D strip packing.

3.5.1. Solving Strip Packing via Fractional Covering

In *strip packing* we are given a list L of n rectangles with widths w_i and heights $h_i \in (0, 1]$. As a target area we are given a strip $[0, 1] \times [0, \infty)$ of width 1 and infinite height. We study axis-aligned arrangements of L without overlap into the strip; these are arrangements in which the rectangles are packed into the strip, the vertical sides of the rectangles

are parallel to the vertical sides of the strip, and the interiors of the rectangles do not intersect. Furthermore, we are interested in minimizing the *packing height*, i.e. in minimizing the maximum height of the top edge of a packed rectangle.

This problem is NP-hard, but permits an *asymptotic fully polynomial time approximation scheme* (AFPTAS) [92]. See [92] and [71] for a detailed presentation on the LP relaxation used in the approach. We focus on the LP model used in [92] and approximately solve it via the algorithm for the max-min resource sharing problem; this approach is presented in [71].

Suppose only M distinct widths $w'_1, \ldots, w'_M$ of rectangles occur. A *configuration* is a multiset of widths which sum up to less than 1, i.e. corresponding items can occur at the same level. Let q be the number of *all* configurations $C_1, \ldots, C_q$ and let α_{ij} denote the number of occurences of w'_i in configuration C_j. For each $i \in \{1, \ldots, M\}$ let β_i denote the sum of all heights of items in L of width w'_i. The *fractional strip packing* problem is defined as

$$\text{minimize } e^T x \text{ subject to } x \geq 0 \text{ and } Ax \geq b \tag{C}$$

where $x, e \in \mathbb{R}^q$, $A \in \mathbb{R}^{M \times q}$ with $A_{ij} = \alpha_{ij}$ for each i and j and $b \in \mathbb{R}^M$ with $b_i = \beta_i$ for each i. Note that q may grow exponentially in M, causing implementational difficulties – it is impossible to explicitly encode all possible configurations within a polynomial runtime bound. This problem is circumvented by using a column generation approach. More precisely, as proposed in [71], we solve

$$\text{compute } x \in B \text{ that satisfies } Ax \geq b \text{ and } e^T x \leq (1+\epsilon)h^* \tag{C_ϵ}$$

where h^* denotes the minimal packing height and $B = \mathbb{R}^q_+$. In [71], the algorithm from [54] is used to solve

$$\text{compute } x \in P \text{ that satisfies } Ax \geq (1-\epsilon)\lambda^* b$$

where

$$P := \{x \in \mathbb{R}^q : \sum_{i=1}^{q} x_i = h\};$$

here h is a fixed packing height. It can be shown that we can solve the problem for $h = 1$ and scale the resulting solution by $(1-\epsilon)\min\{f_i(x) i \in \{1, \ldots, M\}\}^{-1}$ to obtain a $(1+\epsilon)$-approximation of h^*. This means that we finally get a solution of approximately minimal packing height.

Using the algorithm from [54] the block solver is implemented with an FPTAS for the *unbounded knapsack problem* from [90, 107, 118] which has a runtime bound of $O(n + \epsilon^{-3})$. The algorithm from [54] is implemented by column generation. We need

$$O(M\epsilon^{-2} + M \ln M)$$

coordination steps, thus the runtime complexity is

$$O(M(\epsilon^{-3}(\epsilon^{-2} + \ln M) + M(\epsilon^{-2} + \ln M))),$$

which is more efficient than $O(M^6 \ln^2(Mn/(at)) + M^5 n/t + \ln(Mn/(at)))$ as obtained in [85, 102].

3.5.2. Computational Experiments

We used random instances (where *random* means using a pseudo-random number generator) of 100, 1000 and 10000 items, discretized them as in [92] according to $\epsilon \in (0, 1)$ and solved the resulting instances of (C_ϵ) with the algorithm from Section 3.2 where τ was both statically chosen and determined by line search. We observed that the latter results in a significant reduction of the coordination complexity, see for instance Figure 3.10 and Figure 3.18 where we present the average number of iterations for instances with 1000 and 10000 items.

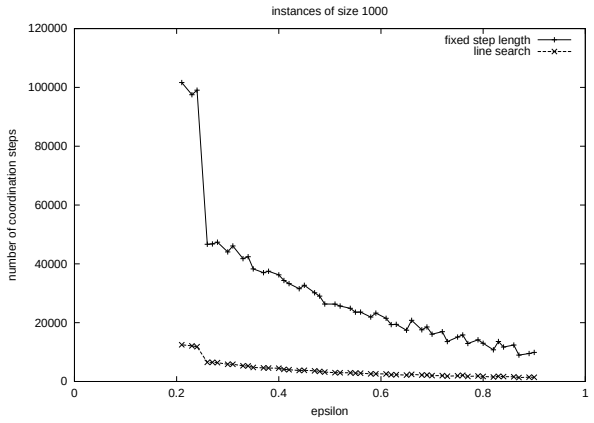


Figure 3.10.: Average number of iterations for instances of size 1000.

The termination criterion described in Subsection 3.3.2 was *never* satisfied — the bound $\omega_s \lambda(y)$ is lower if the quality of y from the previous scaling phase is worse. To put it the other way round, the bound gets worse if the quality of y is better than

required. We have to deal with opposing effects — on one hand, we want to leave each scaling phase with a good solution; on the other hand, the better the solution, the less suitable it is for obtaining a good bound for the next scaling phase. We conclude that the criterion from Subsection 3.3.2 here is of limited heuristic value.

Furthermore we have used instances from the literature to evaluate the behaviour of our algorithm. More precisely, we have executed our implementation on the instances discussed in [61]. There, the authors study seven categories C1–C7 of packing problems, where each category is represented by three problem instances. These instances are also available online via a webpage at

`http://www.simplex.t.u-tokyo.ac.jp/~imahori/packing/instance.html`

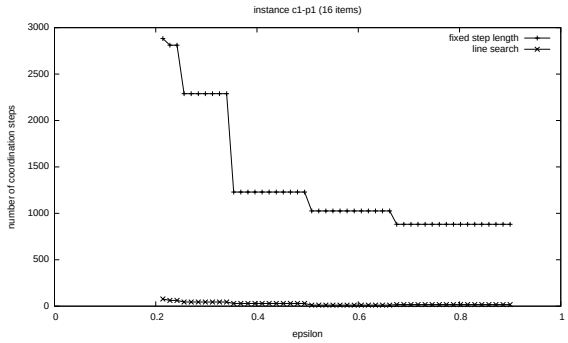
besides the printed form in [61]. The advantage of using line search instead of the fixed choice for determination of the step length τ is also evident on these realistic instances; this can be seen in Figures 3.11–3.17.

In Subsection 3.4.2 the oscillative behavior of the algorithm is discussed. However, for larger values of M, it is difficult to present oscillation graphically. In the application for *fractional strip packing* we have therefore considered the proportion of block solver calls in which a block solution $\hat{x}$ was returned that had been used before in a previous iteration; see Figure 3.19 and Figure 3.20 for a grapical presentation.

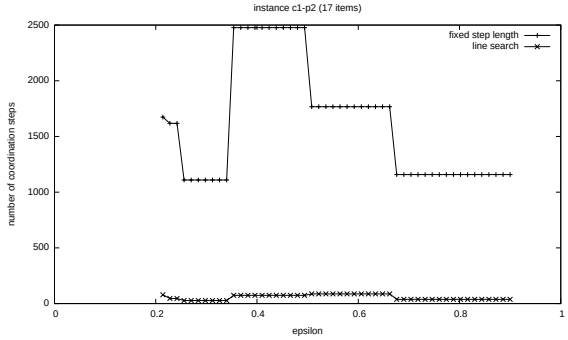
We observed that the selected configurations are by no means evenly distributed; we omit a graphical presentation due to space limitations. This suggests further investigation. Comparing the results for τ statically chosen to τ determined by line search, the latter does not seem to have any significant effect on this behaviour.

3.6. Conclusion

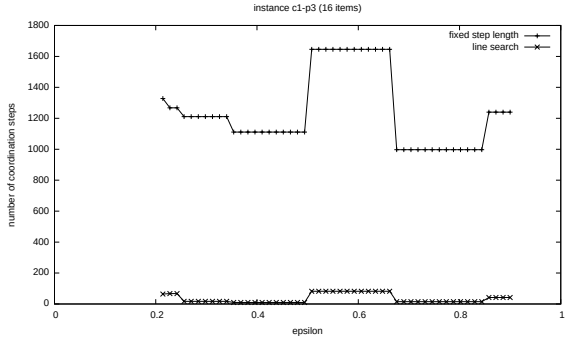
We have implemented an approximation algorithm from [54]; we tested our modified line search for approximating an optimal step length, which turned out to be far superior to using (3.2). We analyse the improvement rate of the dual solution and find that its precision is growing slower than that of the primal solution. We observed that the runtime for a special class of instances is considerably improved. Interestingly, oscillations are shown, which is undesirable and has not been addressed before; our observation might inspire future research. The results on random instances suggest that the runtime is greatly dependent on the instance, sometimes the problem being solved in several steps and sometimes requiring more than thousand of iterations.



(a) Number of iterations for instance C1-P1 with 16 items

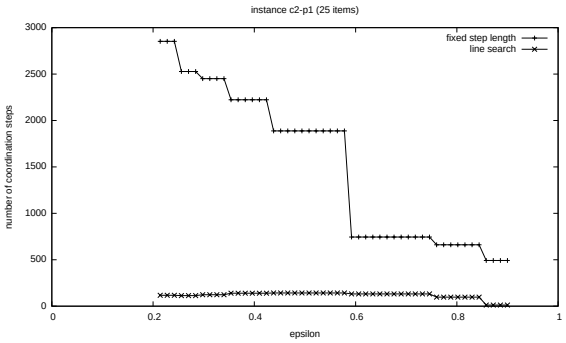


(b) Number of iterations for instance C1-P2 with 17 items

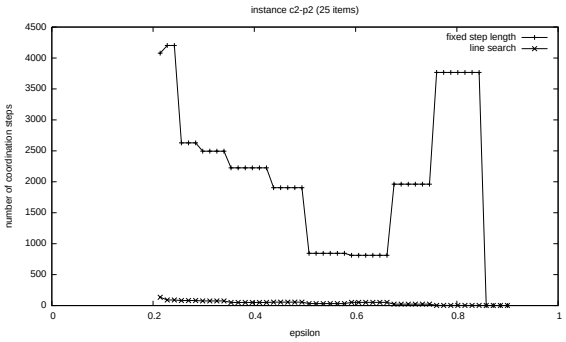


(c) Number of iterations for instance C1-P3 with 16 items

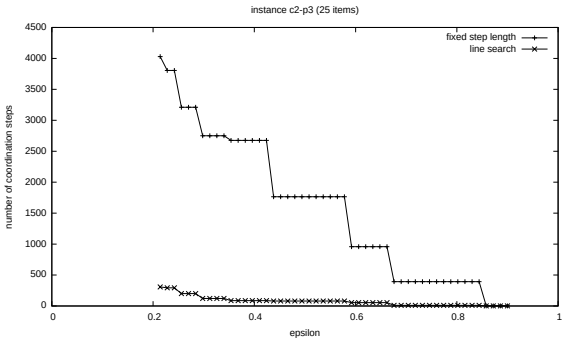
Figure 3.11.: Number of iterations for instances of category C1.



(a) Number of iterations for instance C2-P1 with 25 items

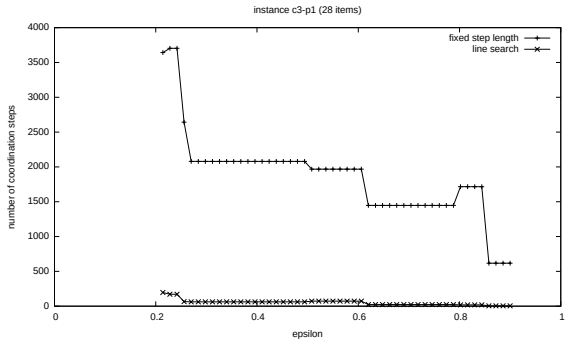


(b) Number of iterations for instance C2-P2 with 25 items

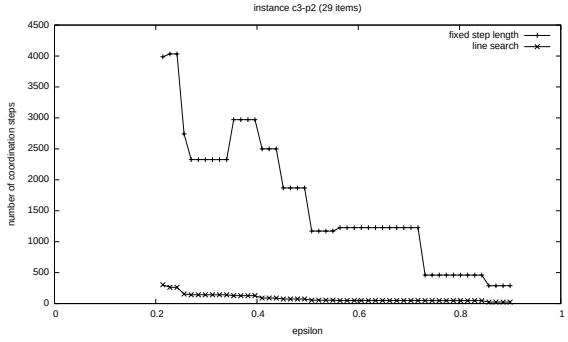


(c) Number of iterations for instance C2-P3 with 25 items

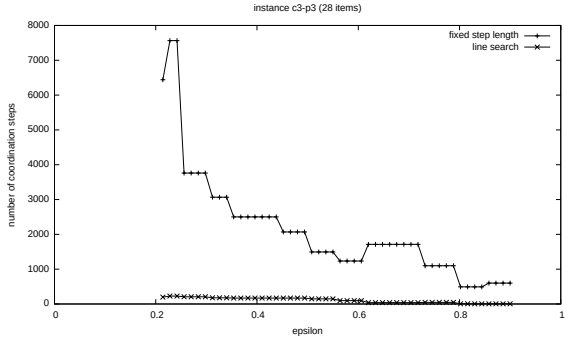
Figure 3.12.: Number of iterations for instances of category C2.



(a) Number of iterations for instance C3-P1 with 28 items

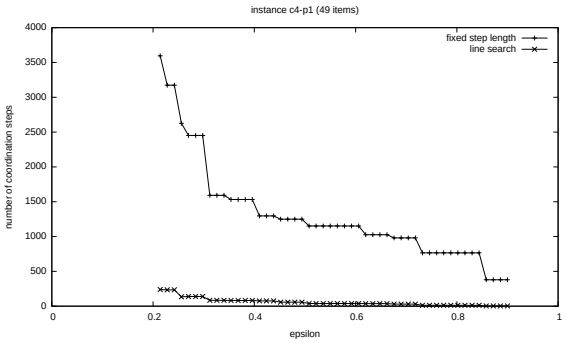


(b) Number of iterations for instance C3-P2 with 29 items

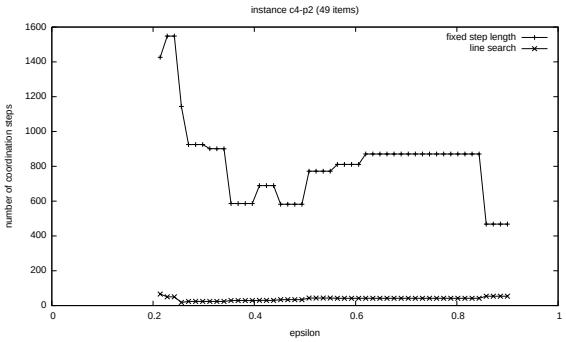


(c) Number of iterations for instance C3-P3 with 28 items

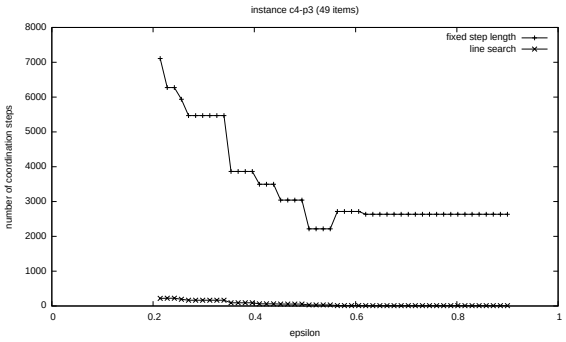
Figure 3.13.: Number of iterations for instances of category C3.



(a) Number of iterations for instance C4-P1 with 49 items

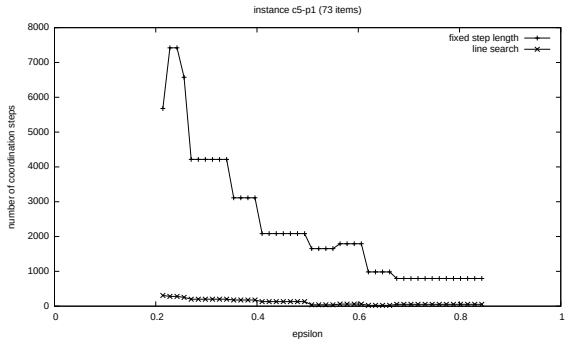


(b) Number of iterations for instance C4-P2 with 49 items

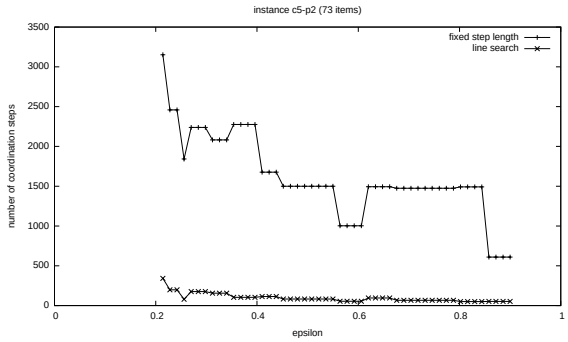


(c) Number of iterations for instance C4-P3 with 49 items

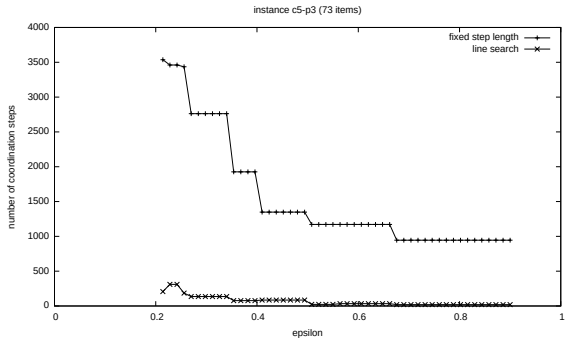
Figure 3.14.: Number of iterations for instances of category C4.



(a) Number of iterations for instance C5-P1 with 73 items

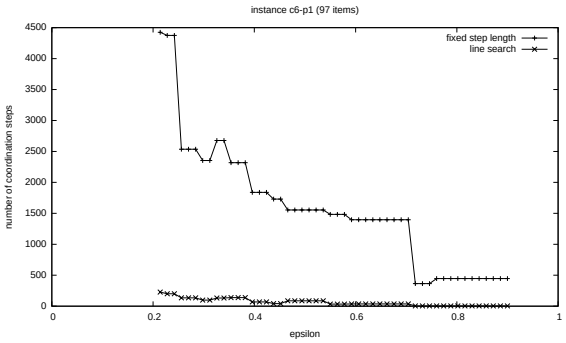


(b) Number of iterations for instance C5-P2 with 73 items

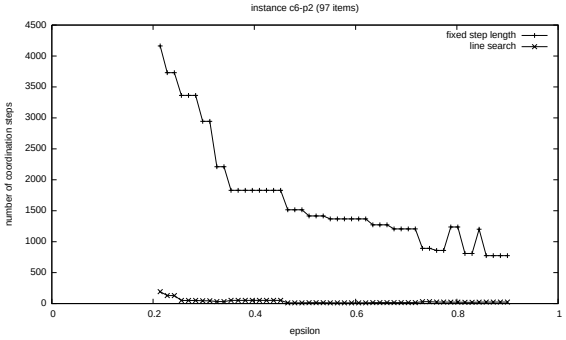


(c) Number of iterations for instance C5-P3 with 73 items

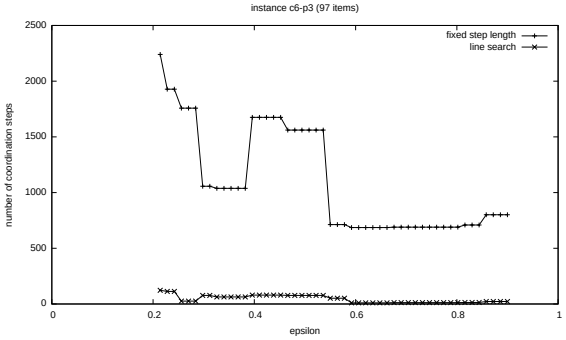
Figure 3.15.: Number of iterations for instances of category C5.



(a) Number of iterations for instance C6-P1 with 97 items

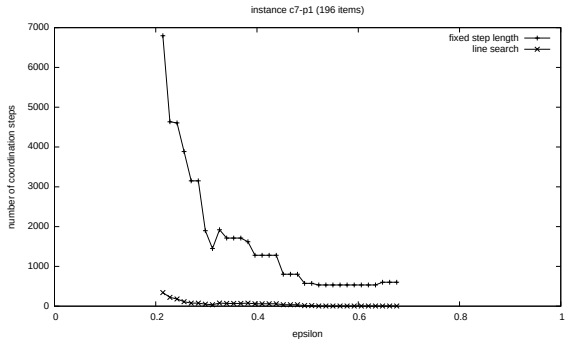


(b) Number of iterations for instance C6-P2 with 97 items

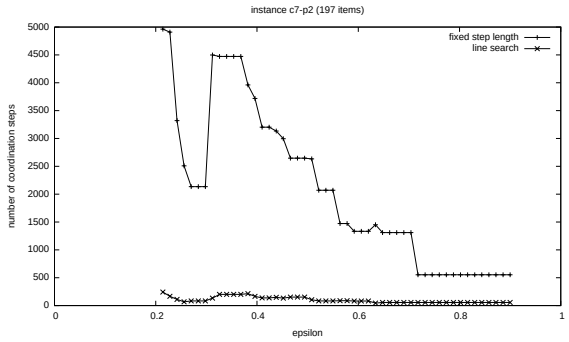


(c) Number of iterations for instance C6-P3 with 97 items

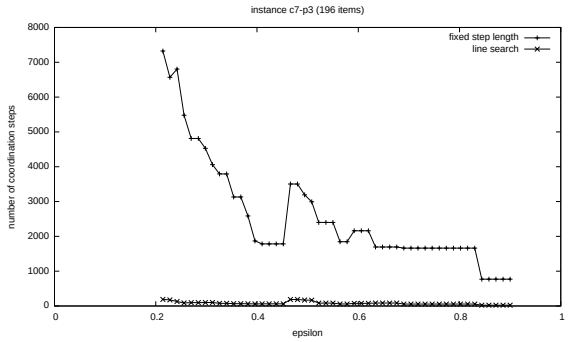
Figure 3.16.: Number of iterations for instances of category C6.



(a) Number of iterations for instance C7-P1 with 196 items



(b) Number of iterations for instance C7-P2 with 197 items



(c) Number of iterations for instance C7-P3 with 196 items

Figure 3.17.: Number of iterations for instances of category C7.

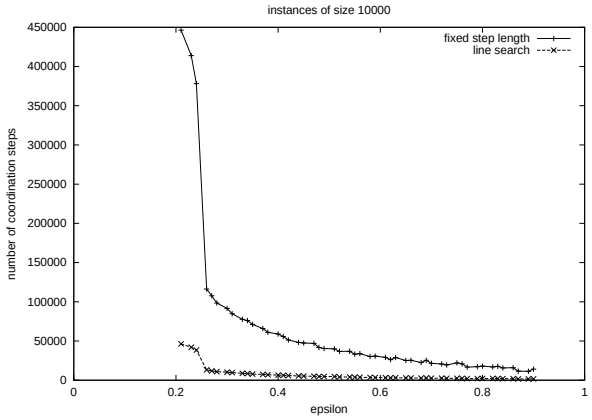


Figure 3.18.: Average number of iterations for instances of size 10000.

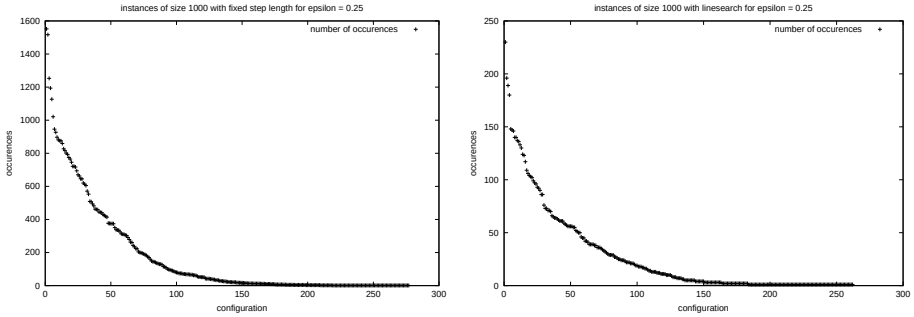


Figure 3.19.: Number of choices of configurations for an instance of size 1000.

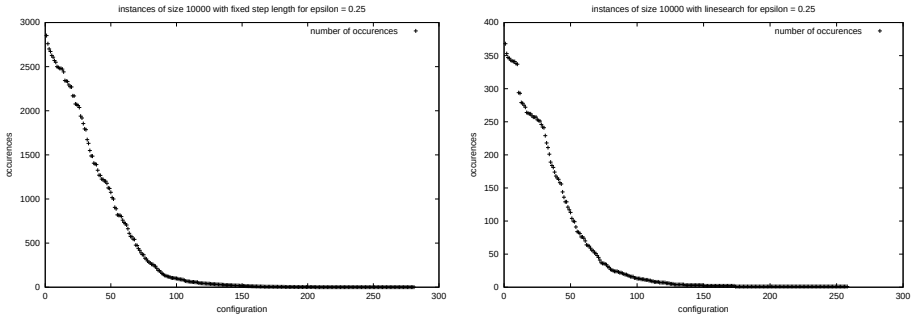


Figure 3.20.: Number of choices of configurations for an instance of size 10000.

4. Constrained Scheduling for m Constant

In this chapter we study the problem of non-preemptively scheduling n independent sequential jobs, given by their processing times, on a system of m identical parallel machines; more precisely, unlike in Chapter 5, m is considered to be constant. The objective is to minimize the makespan $C_{\max}$, which is the maximum completion time of all jobs. However, our scheduling problem will be additionally constrained in one of two ways.

- The first type of constraint we study is a subset of the n jobs being fixed in the system, i.e. the executing machine and starting time for each of the fixed jobs is already prespecified in the instance. This setting is called *scheduling with fixed jobs*.

- The second type of constraint we study is non-availabilty of machines during prespecified time intervals which are given as a part of the instance. This setting is called *scheduling with non-availability*.

Here, the same perspective and motivation as in Chapter 5 apply. Both problems are closely related, in particular they can be formally described via the same encoding scheme of instances. However, the objective function $C_{\max}$ behaves differently for both settings. On one hand, for scheduling with fixed jobs, the fixed jobs contribute to the makespan; for this setting, the objective function models the behaviour of the system from the perspective of the *administrator* who wishes to execute and complete *all* submitted jobs as soon as possible. On the other hand, for scheduling with non-availability, the intervals of non-availability do *not* contribute to the makespan; for this setting, the objective function models the behaviour of the system from the perspective of a *user* who wishes to execute and complete his or her *own* submitted jobs as soon as possible. Needless to discuss, both formulations are in additional ways practically relevant; the

first one occurs in parallel computing platforms since high-priority jobs are present in the system while the second one may occur due to regular maintenance of machines.

For both settings we contribute approximation algorithms with tight ratios. More precisely, we show that scheduling with fixed jobs admits a PTAS and is strongly NP-hard; however, for constant m, this was already known [130, 131, 132] and is included only for the sake of completeness and to illustrate our approach. Furthermore, similar as in Chapter 5, we show that for scheduling with non-availability, a restriction is necessary to obtain a bounded approximation ratio. However, this restriction admits a PTAS and is strongly NP-hard, which means that in a certain sense our result is best possible; surprisingly, although several special cases of this problem have already been studied under the paradigm of approximation algorithms [63, 109, 113, 115], no PTAS had been found until the discovery of our result. As a side issue, we discuss the complexity of various special cases. Finally, note that the problems studied in this chapter are very similar to the ones discussed in Chapter 5; however, as we shall see, the problems behave differently from the point of view of computational complexity and also the techniques applied to obtain the results are quite different. Parts of this chapter have been published in [35] or are currently submitted for publication [36].

4.1. Introduction

In parallel machine scheduling, an important issue is the scenario where either some jobs are already fixed in the system [131, 132] or intervals of non-availability of some machines must be taken into account [35, 63, 109, 113, 115]. The first problem occurs since high-priority jobs are present in the system while the latter problem is due to regular maintenance of machines; both models are also relevant for turnaround scheduling [121] and overlay computing where machines are donated on a volunteer basis. In either case, we obtain deterministic off-line models capturing realistic industrial settings.

Note that, from a naïve point of view, the two problems studied here are the same as the ones discussed in Chapter 5; however, regarding the number m of machines constant makes the problem easier to approximate, as we shall see in the sequel.

4.1.1. Problem Definition

Our two problems can be described by the same encoding of instances and only differ in the objective function. An instance consists of m, the number of machines, which is

considered to be constant, and n jobs given by processing times $p_1, \ldots, p_n$. The first k jobs are fixed via a list $(m_1, s_1), \ldots, (m_k, s_k)$ giving a machine index and starting time for the respective job. We assume that these fixed jobs do not overlap. A schedule is a non-preemptive assignment of the jobs to machines and starting times such that the first k jobs are assigned as encoded in the instance and that the jobs do not intersect.

If the objective is to minimize the makespan for *all* jobs including the fixed ones, we call the problem *scheduling with fixed jobs*. Alternatively we can regard the k fixed jobs as intervals of non-availability which do not contribute to the makespan. Here the objective is to minimize the makespan over the *non-fixed* jobs only; this problem is called *scheduling with non-availability*. For the latter problem, we also permit infinite length of the non-availability intervals, i.e. we permit $p_i = \infty$ for $i \in \{1, \ldots, k\}$. For the sake of generality, we will also write of fixed *objects*; for scheduling whith fixed jobs we mean by this the fixed jobs, while for scheduling with non-availability we mean by this the non-availability intervals.

In the sequel we use $P(I) := \sum_{j=1}^{n} p_j$ to denote the total processing time of an instance I and for each $S \subseteq \{1, \ldots, n\}$ we write $P(S) := \sum_{j \in S} p_j$ for the total processing time of S. Finally let $p_{\max} := \max\{p_j | j \in \{k+1, \ldots, n\}\}$ denote the maximum processing time of non-fixed jobs. More formally, a schedule is a function $\sigma : \{k+1, \ldots, n\} \to \{1, \ldots, m\} \times [0, \infty)$ which maps each job to its executing machine and starting time; furthermore σ is required to be non-preemptive and there may be no intersection between the jobs or the jobs and non-availability intervals. If σ is clear from the context it may be dropped from notation. Furthermore, for a fixed schedule σ, for each $j \in \{1, \ldots, n\}$ let $s_j(\sigma)$ denote the starting time of the non-fixed job j; since k objects are fixed in the schedule, we have $s_j(\sigma) = s_j$ for any schedule σ and $j \in \{1, \ldots, n\}$, i.e. the starting times of the first k objects do not depend on the schedule. Finally, for a fixed schedule σ, for each $j \in \{1, \ldots, n\}$ let $C_j(\sigma) := s_j(\sigma) + p_j$ denote the finishing time of the fixed object j.

As mentioned before, for either problem formulation the objective is to minimize the makespan $C_{\max}$. For scheduling with fixed jobs, for any schedule σ the objective is defined by

$$C_{\max}(\sigma) := \max\{C_j | j \in \{1, \ldots, n\}\}$$

while for scheduling with non-availability, for any schedule σ the objective is defined by

$$C_{\max}(\sigma) := \max\{C_j | j \in \{k+1, \ldots, n\}\}.$$

Furthermore, for both problem formulations we use $C^*_{\max}$ to denote the optimal makespan.

In the literature, scheduling with non-availability is also called *non-resumable scheduling with availability constraints* [35, 109, 113, 115]. The makespan $C_{\max}$ is one of the most well-studied objectives in the field of scheduling and usually regarded as an "easy" objective in the sense that most problem formulations permit good approximation algorithms. However, note that both problems generalize the well-known problem $\mathrm{P}m||C_{\max}$ [127] where neither fixed jobs nor non-availability intervals are present; hence, both scheduling with fixed jobs and scheduling with non-availability are NP-hard.

4.1.2. Results

The problem of scheduling with fixed jobs was already studied by Scharbrodt, Steger & Weisser [130, 131, 132]. For this strongly NP-hard problem they present a PTAS. However, the same positive result can be achieved with our technique; we include a corresponding discussion for the sake of completeness and to illustrate our approach. In total, the following results hold for scheduling with fixed jobs.

Theorem 33. *Scheduling with fixed jobs for m constant admits a PTAS and is strongly* NP*-hard. Hence scheduling with fixed jobs for m constant does not admit an FPTAS unless* $\mathsf{P} = \mathsf{NP}$.

However, note that for scheduling with fixed jobs, it is straightforward to obtain an approximation algorithm with ratio 3 by simply scheduling all non-fixed jobs via list scheduling after the completion of the last fixed jobs, as discussed in [130]. This basic idea can be refined to yield an approximation algorithm with ratio $2 + \epsilon$ by replacing the list scheduling algorithm with an FPTAS for $\mathrm{P}m||C_{\max}$ from [127].

Unlike scheduling with fixed jobs, scheduling with non-availability, no matter if m is constant or part of the input, without any further restriction is inapproximable within a constant ratio unless $\mathsf{P} = \mathsf{NP}$, as shown by Eyraud-Dubois, Mounié & Trystram [39]; this subject is discussed in Subsection 4.3.2. Similar as in Chapter 5, the inapproximability is circumvented by requiring at least one machine to be permanently available. However, researchers so far have only studied the problem where there is at most one interval of non-availability per machine. First, the even more restricted case where the intervals of non-availability start at time zero was studied. Here Lee [108] and Lee et al. [110] proved that LPT yields a ratio of $3/2 - 1/(2m)$ and can be modified to yield a ratio of $4/3$. For the same problem, Kellerer [86] found an algorithm with a tight ratio of $5/4$. Furthermore, Hwang et al. briefly pointed out that this problem admits a PTAS [63].

A more general case is the setting where the at most one interval per machine may have an arbitrary position. For this problem Lee [109] showed that general list scheduling yields a ratio of m and proved a ratio of $1/2 + m/2$ for LPT, with an incomplete proof, however. Hwang et al. studied the ratio of LPT for the same scenario but assumed that at least $m - \lambda$ machines are available simultaneously. They first obtained a ratio of 2 for $\lambda \leq m/2$ [62] which they later refined to a ratio of $1 + \lceil 1/(1 - \lambda/m) \rceil /2$ for λ arbitrary [63]. For $\lambda = m - 1$, this yields $1 + m/2$; we will compare this with the ratio of a new greedy algorithm in Subsection 4.3.1. Concerning further algorithmic results, we refer the reader to [113], Chapter 22, or [129] for surveys and the articles [82, 108] for more results on single-machine problems. For scheduling with non-availability, our technique yields a PTAS which is tight for a necessary restriction.

Theorem 34. *Scheduling with non-availability for m constant where at least one machine is permanently available admits a PTAS and is strongly* NP*-hard. Hence scheduling with non-availability for m constant does not admit an FPTAS unless* P = NP*.*

Historically speaking, the following point is interesting. On one hand, one might argue that our approach is heavily based on a PTAS for the Multiple Subset Sum Problem, which results from a modelization that is not very technical and straightforward in its nature. On the other hand, PTASes for the Multiple Subset Sum Problem are relatively new [18, 24, 72] and the modelization of our scheduling problems via this approach was only very briefly mentioned in one publication by Liao, Shyur & Lin so far [115]; furthermore, the article [115] deals with a very restricted two-machine problem and does not approach the problem with the paradigm of approximation algorithms.

4.1.3. Techniques Used in Our Approach

Besides dual approximation [59] via binary search on the optimal makespan, the approach taken in our work is based on multiple subset sum problems. These are special cases of knapsack problems, which belong to the oldest problems studied in combinatorial optimization and theoretical computer science; hence we benefit from the fact that they are relatively well understood. For the classical problem (KP) with one knapsack, besides the result by Ibarra & Kim [64], Lawler presented a sophisticated FPTAS [107] which was later improved by Kellerer & Pferschy [89]; see also the textbooks by Martello & Toth [118] and Kellerer et al. [90] for surveys. The case where the item profits equal their weights is called the subset sum problem and denoted as SSP. The problem with *multiple* knapsacks (MKP) is a natural generalization of KP; the case with multiple

knapsacks where the item profits equal their weights is called the *multiple* subset sum problem, denoted by MSSP. Various special cases and extensions of these problems have been studied [17, 18, 19, 23, 24, 28, 29, 72, 87, 88], finally yielding PTASes for various problem formulations [18, 23, 24, 72, 88] including the case upon which our approach is based. Furthermore for the two-machine problems we use dynamic programming and scaling of the state space by rounding the instance; however this is a widely used technique which is extensively discussed in [135, 146].

The remainder of Chapter 4 is organized as follows. In Section 4.2, we present results for scheduling with fixed jobs for m constant. More precisely, in Subsection 4.2.1 we discuss the algorithmic results; note that here we present the most basic technique of conversion of the instance to intervals of availability, which is used as a subroutine in almost all of the algorithms presented in Chapter 4 and Chapter 5. Furthermore, in Subsection 4.2.2 we complement these approximation algorithms by suitable inapproximability results. In Section 4.3 we present our new results for scheduling with non-availability for m constant. More precisely, in Subsection 4.3.1 we present approximation algorithms. The positive results from Subsection 4.2.1 are complemented in Subsection 4.3.2 by suitable hardness proofs. Finally we finish Chapter 4 in Section 4.4 with a conclusion.

4.2. Scheduling with Fixed Jobs

In this section we describe the results for scheduling with fixed jobs for m constant. In Subsection 4.2.1 we present approximation algorithms, more precisely a PTAS and a greedy algorithm based on the same idea as well as some special cases of the problem; these are complemented in Subsection 4.2.2 with suitable hardness results.

4.2.1. Approximation Algorithms

First we discuss how to obtain the sets of intervals of availability for each machine $i \in \{1, \ldots, m\}$ from the encoded instance. We describe the algorithm in its full generality, i.e. we write of fixed *objects* which can be either fixed jobs or intervals of non-availability; recall that for scheduling with non-availability, we permit $p_i = \infty$ for $i \in \{1, \ldots, k\}$. We also assume that the fixed part of the instance $(m_1, s_1), \ldots, (m_k, s_k)$ is sorted lexicographically non-decreasingly with respect to both components, which can be algorithmically achieved by using e.g. Quicksort.

1. For each $i \in \{1, \ldots, m\}$ set $M_i := \{\ell \in \{1, \ldots, k\} | m_\ell = i\}$.
2. For each $i \in \{1, \ldots, m\}$ set $\kappa_i := |M_i|$, we denote $M_i = \{\ell_{i_1}, \ldots, \ell_{i_{\kappa_i}}\}$.
3. For each $i \in \{1, \ldots, m\}$ set $A_i := \emptyset$.
4. For each $i \in \{1, \ldots, m\}$ with $\kappa_i = 0$ set $A_i := \{[0, \infty)\}$.
5. For each $i \in \{1, \ldots, m\}$ with $\kappa_i \neq 0$ execute Steps 5.1–5.2.
   5.1. Set $t := 0$.
   5.2. For each $i' \in \{1, \ldots, \kappa_i\}$ execute Steps 5.2.1–5.2.5.
      5.2.1. If $i' = 1$ and $s_{\ell_{i_{i'}}} = 0$ then set $t := s_{\ell_{i_{i'}}} + p_{\ell_{i_{i'}}}$.
      5.2.2. If $i' = 1$ and $s_{\ell_{i_{i'}}} > 0$ then set $A_i := A_i \cup \{[t, s_{\ell_{i_{i'}}})\}$, $t := s_{\ell_{i_{i'}}} + p_{\ell_{i_{i'}}}$.
      5.2.3. If $i' \in \{2, \ldots, \kappa_i - 1\}$ then $A_i := A_i \cup \{[t, s_{\ell_{i_{i'}}})\}$, $t := s_{\ell_{i_{i'}}} + p_{\ell_{i_{i'}}}$.
      5.2.4. If $i' = \kappa_i$ and $p_{\ell_{i_{\kappa_i}}} = \infty$ then $A_i := A_i \cup \{[t, s_{\ell_{i_{i'}}})\}$.
      5.2.5. If $i' = \kappa_i$ and $p_{\ell_{i_{\kappa_i}}} \neq \infty$ then $A_i := A_i \cup \{[t, s_{\ell_{i_{i'}}}), [s_{\ell_{i_{i'}}} + p_{\ell_{i_{i'}}}, \infty)\}$.

Figure 4.1.: Algorithm GenAvail.

First we discuss how to process the instance to obtain the intervals of availability; the approach is sketched in Figure 4.3. The algorithm in Figure 4.1 uses the encoding of the instance to obtain the lists $A_1, \ldots, A_m$ where for each $i \in \{1, \ldots, m\}$ the list $A(i)$ contains the set of intervals during which the machine i is available for an infinite planning horizon $[0, \infty)$. More precisely, Step 1 generates lists $M_1, \ldots, M_m$ where for each $i \in \{1, \ldots, m\}$ the set M_i contains the indices of objects which are fixed on machine i; Step 2 defines the variable κ_i for each $i \in \{1, \ldots, m\}$ to be the number of objects being fixed on machine i for each $i \in \{1, \ldots, m\}$. Afterwards, for each $i \in \{1, \ldots, m\}$ the list A_i is defined to be empty; however, in Step 4, each machine hosting no fixed object is defined to be permanently available. Finally, Step 5 iterates all machines hosting fixed objects. For each machine indexed by i, we use a time pointer t indicating the next time step during which machine i potentially becomes available next; consequently, t is initialized with 0 in Step 5.1. In Step 5.2, we iterate the fixed objects hosted by machine i; we use i' to denote the index of the object. For the first object we distinguish between the cases where its starting time is 0 or greater than 0. In the first case, no interval of availability is generated in Step 5.2.1 and only the time pointer t is updated. In the second case, we generate a corresponding interval of availability and update the time pointer in Step 5.2.1. Step 5.2.3 deals with the case that the iterated object is neither

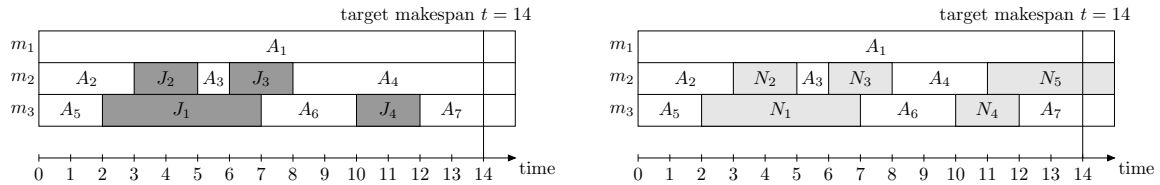
1. For each $i \in \{1, \ldots, m\}$ set $A_i(t) = \emptyset$.
2. For each $i \in \{1, \ldots, m\}$ set $\kappa_i := |A_i|$, we denote $A_i = \{[s_{i_1}, t_{i_1}), \ldots, [s_{i_{\kappa_i}}, t_{i_{\kappa_i}})\}$.
3. For each $i \in \{1, \ldots, m\}$ execute Step 3.1.
   3.1. For each $i' \in \{1, \ldots, \kappa_i\}$ execute Step 3.1.1.
      3.1.1. If $s_{i_{i'}} \leq t$ then set $t_{i_{i'}} := \min\{t_{i_{i'}}, t\}$, $A_i(t) := A_i(t) \cup \{[s_{i_{i'}}, t_{i_{i'}})\}$.

Figure 4.2.: Algorithm GenAvailFinite.

the first nor the last one on the current machine i; consequently we generate an interval of availability and perform a corresponding update of the time pointer t. Finally, the last two steps deal with the case that the iterated object is the last one on the current machine. In the case covered in Step 5.2.4, the last object occupies the current machine forever and we generate a finite last interval for the current machine; note that this case can only occur for scheduling with non-availability. Finally, in the case covered by Step 5.2.5, the last object has finite length. In this case, we generate a second to last interval of availability of finite length and finally a last interval of infinite length for the current machine.

However more important is the algorithm in Figure 4.2. This algorithm takes as an input the lists $A_1, \ldots, A_m$ generated by the algorithm in Figure 4.1 and truncates the intervals of availability to a finite planning horizon $[0, t)$ for a given target makespan t; more precisely we generate lists $A_1(t), \ldots, A_m(t)$ where for each $i \in \{1, \ldots, m\}$ the list A_i contains the intervals of availability for the given planning horizon $[0, t)$. In Step 1, each list $A_i(t)$ is defined to be empty; in Step 2 we define for each $i \in \{1, \ldots, m\}$ the variable κ_i to be the number of intervals of availability on machine i for the infinite planning horizon $[0, \infty)$. In Step 3, we iterate the machines. More precisely, in Step 3.1 for machine i we iterate the intervals of availability for machine i; if the currently considered interval of availability for machine i starts before the end t of the planning horizon $[0, t)$, it is added to the list $A_i(t)$ where, if necessary, the ending time of the interval of availability is truncated to t to fit exactly into the planning horizon. Note that the running time of both algorithms in Figure 4.1 and Figure 4.2 is polynomially bounded in m and k and does not depend on n. For our packing approach, only the sizes of the generated intervals of availability are important.

In the sequel, the intervals of availability are simply called gaps. Let $q(t)$ denote the total number gaps generated for a fixed makespan t; since each of the k fixed objects is



(a) Sketch illustrating the approach of the algorithms in Figure 4.1 and Figure 4.2 for scheduling with fixed jobs $J_1, \ldots, J_4$, which are indicated by dark grey areas. For target makespan 14, we generate gaps corresponding to $A_1, \ldots, A_7$; intervals A_1 and A_7 end at time 14.

(b) Sketch illustrating the approach of the algorithms in Figure 4.1 and Figure 4.2 for scheduling with non-availability intervals $N_1, \ldots, N_5$, which are indicated by light grey areas. For target makespan 14, we generate gaps corresponding to $A_1, \ldots, A_7$; intervals A_1 and A_7 end at time 14.

Figure 4.3.: This sketch illustrates the approach of the algorithms in Figure 4.1 and Figure 4.2.

adjacent to at most 2 gaps and we cannot have less gaps than machines unless a machine is totally occupied, we have

$$q \leq \max\{2k, m\} \leq \max\{2n, m\}$$

which is polynomially bounded in the encoding length of the instance.

Next, as mentioned before, we use a PTAS for MSSP where the knapsack capacities are permitted to be different from [18, 24, 72]. We denote the PTAS for MSSP by MSSPPTAS. More precisely, for a fixed target makespan t, the algorithm MSSPPTAS is used as a relaxed decision procedure for dual approximation as in [59]. The basic approach is to use suitable lower and upper bounds on the optimal makespan which get successively refined. For each candidate makespan t, we use MSSPPTAS to schedule as much load as possible in the planning horizon $[0, t)$. This process is terminated as soon as the bounds coincide; the remaining load which could not be scheduled in the interval $[0, t)$ is scheduled in the interval $[t, \infty)$. In total, this approach can be formulated as in Figure 4.4.

As we will see with the following analysis, the algorithm in Figure 4.4 yields a PTAS for scheduling with fixed jobs.

Theorem 35. *The algorithm in Figure 4.4 is a PTAS for scheduling with fixed jobs for m constant.*

Proof. First of all, each feasible schedule must execute the fixed jobs, hence the optimal makespan can not be smaller than the completion time of any of the fixed jobs. This

1. Use the algorithm in Figure 4.1 to generate A_i for each $i \in \{1, \ldots, m\}$.
2. Set $LB := \max\{s_j + p_j | j \in \{1, \ldots, k\}\}$ and $UB := LB + P(\{k+1, \ldots, n\})$.
3. While $UB - LB > 1$ repeat Steps 3.1–3.3.
   3.1 Set $t := \lfloor (LB + UB)/2 \rfloor$. Use the algorithm in Figure 4.2 to generate $A_i(t)$ for each $i \in \{1, \ldots, m\}$, the lists of gaps for each machine for fixed planning horizon $[0, t)$.
   3.2 Use MSSPPTAS with accuracy ϵ/m to select a set of jobs $S \subseteq \{k+1 \ldots, n\}$ such that

$$P(S) \geq (1 - \epsilon/m) \max\{P(S') | S' \subseteq \{k+1, \ldots, n\},$$
$$S' \text{ permits a feasible packing into the intervals in } A_1(t), \ldots, A_m(t)\}.$$

   3.3 If $P(S) < (1 - \epsilon/m)P(\{k+1, \ldots, n\})$ then set $LB := t$ else store S and set $UB := t$.
4. Schedule the jobs in the last stored set S into the interval $[0, UB)$ as indicated by the solution generated by MSSPPTAS when S was returned; schedule the jobs in $\{k+1, \ldots, n\} \setminus S$ in the interval $[UB, \infty)$ on the first machine without unnecessary idle time.

Figure 4.4.: Algorithm FixedScheduler.

means that in Step 2, LB is initialized with a valid lower bound for $C^*_{\max}$. Furthermore, executing all non-fixed jobs $\{k+1, \ldots, n\}$ after the completion of the latest fixed job yields a feasible schedule with makespan $LB + P(\{k+1, \ldots, n\})$, which means that in Step 2 the variable UB is initialized with a valid upper bound for $C^*_{\max}$. In total, the lower bound LB and the upper bound UB are initialized to have the following properties.

1. $LB \leq C^*_{\max}$.
2. There is a set $S \subseteq \{k+1, \ldots, n\}$ such that the jobs in S permit a feasible schedule into the time horizon $[0, UB)$ and $P(S) \geq (1 - \epsilon/m)P(\{k+1, \ldots, n\})$.

The second property is due to the fact that, since $C^*_{\max} \leq UB$, all jobs can be scheduled in $[0, UB)$ and thus it is impossible that the algorithm MSSPPTAS returns a set $S \subseteq \{k+1 \ldots, n\}$ such that $P(S) < (1 - \epsilon/m)P(\{k+1 \ldots, n\})$ holds; both properties are invariant under the update of LB and UB in Step 3.3. The number of iterations of the

binary search in Step 3 is bounded by

$$\log P(\{k+1,\ldots,n\}) \leq \log(np_{\max}) = \log n + \log p_{\max}$$

which is polynomially bounded in the encoding length of the instance. On termination of the binary search in Step 3, $LB = UB$ holds, hence $UB \leq C^*_{\max}$ since $LB < C^*_{\max}$ is satisfied. This means that the set S selected in Step 4 can be scheduled in $[0, UB)$ and satisfies $P(S) \geq (1-\epsilon/m)P(\{k+1,\ldots,n\})$; hence

$$P(\{k+1,\ldots,n\} \setminus S) \leq \epsilon P(\{k+1,\ldots,n\})/m$$

holds. Furthermore the jobs in $\{k+1,\ldots,n\} \setminus S$ can be scheduled on the first machine in $[UB, \infty)$ since the first machine is available. We have

$$P(\{k+1,\ldots,n\})/m \leq C^*_{\max};$$

in total, the makespan of the schedule generated by the algorithm in Figure 4.4 is bounded by

$$UB + \epsilon P(\{k+1,\ldots,n\})/m \leq C^*_{\max} + \epsilon C^*_{\max} = (1+\epsilon)C^*_{\max}$$

and we obtain the desired approximation ratio. Since the running time of MSSPPTAS is polynomially bounded in q and n, the claim is proven. □

However, since the running time of MSSPPTAS may grow exponentially in $1/\epsilon$, the running time of the algorithm in Figure 4.4 may also grow exponentially in m. MSSP does not admit an FPTAS even for the special case of two knapsacks of equal capacity, unless $\mathsf{P} = \mathsf{NP}$ holds, as discussed in [90], Subsection 10.4. Hence it is impossible for the approach used above to yield an FPTAS for scheduling with fixed jobs for m constant by replacing MSSPPTAS with a better algorithm, which is not surprising in the light of Corollary 43 in Subsection 4.2.2. In total, we have proved the first part of Theorem 33.

Furthermore, it is a natural question whether special cases of scheduling with fixed jobs for m constant are easier to approximate. However, clearly the approximation ratio of the PTAS presented above cannot be improved, but there is an FPTAS for scheduling with fixed jobs on 2 machines with exactly one fixed job. This FPTAS can be obtained with a very different approach, namely we use a dynamic programming formulation and scaling of the state space in order to obtain a polynomial runtime bound.

More precisely, without loss of generality, the fixed job is on machine 2 in the interval $[s,t)$. First note that using list scheduling to place the jobs in $\{k+1\ \ldots,n\}$ in the interval $[t,\infty)$ yields a 3-approximation which we use later. Let C' denote the makespan of a feasible schedule obtained in this way. Hence we have $C^*_{\max} \leq C' \leq 3C^*_{\max}$; furthermore we denote by A the interval $[0,\infty)$ on machine 1, by B the interval $[0,s)$ on machine 2 and by C the interval $[t,\infty)$ on machine 2. For a partial schedule σ, i.e. a schedule that assigns a subset of the jobs, we use $A(\sigma)$ to denote its load in A, $B(\sigma)$ to denote its load in B and $C(\sigma)$ to denote its load in C. The states of the dynamic program can be organized as a table by defining

$$F[j,x,y] := \min\{\infty, \min\{B(\sigma) | \sigma \text{ is a schedule for the jobs in } \{k+1,\ldots,j\} \text{ such that } A(\sigma)=x \text{ and } C(\sigma)=y\}\}$$

for each $j \in \{k+1,\ldots,n\}$ and $x,y \in \{0,\ldots,C'\}$, where ∞ indicates the nonexistence of such a schedule; we also suppose that F evaluates to infinity for negative values of x and y. Next we prove a suitable recurrence relation which permits to solve the problem to optimality via dynamic programming.

Theorem 36. *The recurrence relation*

$$F[j,x,y] = \begin{cases} \min\{F[j-1,x-p_j,y], & \\ \quad F[j-1,x,y-p_j]\} & : F[j-1,x,y]+p_j > s \\ \min\{F[j-1,x-p_j,y], & \\ \quad F[j-1,x,y-p_j], F[j-1,x,y]+p_j\} & : F[j-1,x,y]+p_j \leq s \end{cases}$$

for each $j \in \{k+2,\ldots,n\}, x,y \in \{0,\ldots,C'\}$ is satisfied for the function F defined above.

Proof. Note that the cases in the statement above correspond to the cases in which the job with index $j \in \{k+2,\ldots,n\}$ can or can not be be placed in the area B. Now let $j \in \{k+2,\ldots,n\}, x,y \in \{0,\ldots,C'\}$; then one of the following two cases occurs.

Case 1: $F[j-1,x,y] = \infty$. Then there is no schedule σ for the Jobs in $\{k+1,\ldots,j\}$ such that $A(\sigma)=x$ and $C(\sigma)=y$. We consider the following two subcases.

Case 1.1: $F[j-1,x,y]+p_j \geq s$. If $F[j-1,x-p_j,y] \in \mathbb{N}\setminus\{0\}$, there is a schedule σ' for the jobs in $\{1,\ldots,j-1\}$ such that $A(\sigma')=x-p_j$ and $C(\sigma')=y$. Placement of job j in σ' on machine 1 yields a schedule σ'' for the jobs in $\{k+1,\ldots,j\}$ with $A(\sigma'')=x-p_j+p_j=x$

and $B(\sigma'') = y$, in contradiction to $F[j, x, y] = \infty$. If $F[j-1, x, y-p_k] \in \mathbb{N} \cup \{0\}$, there is a schedule σ' for the jobs in $\{k+1, \ldots, j-1\}$ such that $A(\sigma') = x$ and $C(\sigma') = y - p_j$. Placement of job j in σ' on machine 2 in the time interval $[t, \infty)$ yields a schedule σ'' with $A(\sigma'') = x$ and $C(\sigma'') = y - p_j + p_j = y$, in contradiction to $F[j, x, y] = \infty$. In total we have $F[j-1, x-p_j, y] = \infty = F[j-1, x, y-p_j]$ and we obtain

$$\min\{F[j-1, x-p_j, y], F[j-1, x, y-p_j]\} = \infty = F[j, x, y].$$

Case 1.2: $F[j-1, x, y+p_j] < s$. If $F[j-1, x-p_j, y] \in \mathbb{N} \cup \{0\}$, we obtain the same contradiction as above which yields $F[j-1, x-p_j, y] = \infty$. If $F[j-1, x, y-p_j] \in \mathbb{N} \cup \{0\}$, we obtain the same contradiction as above, which yields $F[j-1, x.y-p_j] = \infty$. If $F[j-1, x, y] \in \mathbb{N} \cup \{0\}$, there is a schedule σ' for the jobs in $\{k+1, \ldots, j\}$ such that $A(\sigma') = x$ and $B(\sigma') = y$. Placement of job j in σ' on machine 2 in the time interval $[0, s)$ yields a schedule σ'' for the jobs in $\{k+1, \ldots, j\}$ such that $A(\sigma'') = x$ and $B(\sigma'') = y$, in contradiction to $F[j, x, y] = \infty$. In total we have

$$F[j-1, x-p_j, y] = F[j-1, x, y-p_j] = F[j-1, x, y] = \infty$$

and we obtain

$$\min\{F[j-1, x-p_j, y], F[j-1, x, y-p_j], F[j-1, x, y] + p_j\} = \infty = F[j, x, y].$$

Case 2: $F[j, x, y] \in \mathbb{N} \cup \{0\}$. Then there is a schedule σ for the jobs in $\{1, \ldots, j\}$ such that $A(\sigma) = x$ and $B(\sigma) = y$. Select σ in such a way that $B(\sigma) = F[j, x, y]$ holds. We consider the following two subcases.

Case 2.1: $F[k-1, x, y] + p_j \geq s$. From the definition of F it follows that in σ job j does not run on machine 2 in the time interval $[0, s)$. If job j runs in σ on machine 1, removal of job j in σ yields a schedule σ' for the jobs in $\{k+1, \ldots, j-1\}$ such that $A(\sigma') = x - p_j$ and $C(\sigma') = y$. From the definition of F it follows that $F[j-1, x-p_j, y] \leq B(\sigma')$. Aiming at a contradiction we assume that $F[j-1, x-p_j, y] < B(\sigma')$ holds. Then there is a schedule σ'' for the jobs in $\{1, \ldots, j-1\}$ such that $A(\sigma'') = x - p_j$, $C(\sigma'') = y$ and $B(\sigma'') < B(\sigma)$. Placement of job j in σ'' on machine 1 yields a schedule σ''' for the jobs in $\{k+1, \ldots, j\}$ with $A(\sigma''') = x - p_j + p_j = x$, $C(\sigma''') = y$ and $B(\sigma''') < B(\sigma') = B(\sigma)$, in contradiction to the choice of σ. In total we obtain

$$F[j-1, x-p_j, y] = B(\sigma') = B(\sigma) = F[j, x, y].$$

If job j runs in σ on machine 2 in the time interval $[t, \infty)$, removal of job j in σ yields a schedule σ'' for the jobs in $\{1, \ldots, j-1\}$ with $A(\sigma') = x$ and $C(\sigma') = y - p_j$. From the definition of F it follows that $F[j-1, x, y-p_j] \leq B(\sigma')$. Aiming at a contradiction we assume that $F[j-1, x, y-p_j] < B(\sigma')$ holds. Then there is a schedule σ'' for the jobs in $[j-1]$ with $A(\sigma'') = x$, $C(\sigma'') = y - p_j$ and $B(\sigma'') < B(\sigma')$. Placement of job j in σ'' on machine 2 in the time interval $[t, \infty)$ yields a schedule σ''' for the jobs in $\{k+1, \ldots, j\}$ with $A(\sigma''') = x$, $C(\sigma''') = y - p_j + p_j = y$ and $B(\sigma''') < B(\sigma') = B(\sigma)$, in contradiction to the choice of σ. In total we have

$$F[j-1, x, y-p_j] = B(\sigma') = B(\sigma) = F[j, x, y].$$

Case 2.2: $F[j-1, x, y] + p_j < s$. If job j in σ runs on machine 1, we obtain

$$F[j-1, x-p_j, y] = F[j, x, y]$$

with the same arguments as above. If job j in σ runs on machine 2 in the interval $[t, \infty)$, we obtain $F[j-1, x, y-p_j] = F[j, x, y]$ with the same arguments as above. If job j runs in σ on machine 2 in the interval $[0, s)$, removal of job j in σ yields a schedule σ' for the jobs in $\{k+1, \ldots, j-1\}$ with $A(\sigma') = x$, $B(\sigma') = B(\sigma) - p_j$ and $C(\sigma') = y$. From the definition of F it follows that $F[j-1, x, y] \leq B(\sigma')$. Aiming at a contradiction we assume that $F[j-1, x, y] < B(\sigma')$ holds. Then there is a schedule σ'' for the jobs in $\{1, \ldots, j-1\}$ with $A(\sigma'') = x$, $C(\sigma'') = y$ and $B(\sigma'') < B(\sigma')$. Placement of job j in σ'' on machine 2 in the interval $[0, s)$ yields a schedule σ''' for the jobs in $\{k+1, \ldots, j\}$ with $A(\sigma''') = x$, $C(\sigma''') = y$ and

$$B(\sigma''') = B(\sigma'') + p_j < B(\sigma') + p_j = B(\sigma) - p_j + p_j = B(\sigma),$$

in contradiction to the choice of σ. Finally we have

$$F[j-1, x, y] = B(\sigma') = B(\sigma) - p_j = F[j, x, y] - p_j,$$

which can be rearranged to $F[j-1, x, y] + p_j = F[j, x, y]$.

In total, the claim is proven. □

Based on the theorem above, either inductively by iterating over $j \in \{k+1, \ldots, n\}$ or recursively using so-called lazy evaluation, we can solve the problem of scheduling with fixed job on two machines with one fixed job to optimality. Inductive evaluation of all

states of the dynamic program can be carried out as implemented in Figure 4.5. For ease of presentation we assume that the evaluation of $F[j, x, y]$ yields ∞ for negative values of x and y.

1. For each $y, x \in \{0, \ldots, C'\}$ set $F[k+1, x, y] = \infty$.
2. Set $F[k+1, p_{k+1}, 0] = 0$ and $F[k+1, 0, p_{k+1}] = 0$.
3. If $p_{k+1} \leq s$ set $F[k+1, 0, 0] = p_{j+1}$.
4. For each $j = k+2$ to n execute Step 4.1.
   - 4.1 For each $x, y \in \{0, \ldots, C'\}$ execute Step 4.1.1.
     - 4.1.1 If $F[j-1, x, y] + p_j > s$ then set

$$F[j, x, y] = \min\{F[j-1, x-p_j, y], F[j-1, x, y-p_j]\}$$

else set

$$F[j, x, y] = \min\{F[j-1, x-p_j, y], F[j-1, x, y-p_j], F[j-1, x, y] + p_j\}.$$

Figure 4.5.: Algorithm DynamicProgramming.

Hence, evaluation of the entire state space can be carried out within the pseudopolynomial runtime bound $O(nC'^2) = O(n^3 p_{\max}^2)$. In total, after evaluation of the state space, we can solve the problem of scheduling with fixed jobs with one fixed job to optimality within the same runtime bound by selecting $x, y \in \{0, \ldots, C'\}$ in order to minimize the value

$$f(x, y) := \begin{cases} \max\{x, t+y\} & : F[n, x, y] \neq \infty \\ \infty & : F[n, x, y] = \infty \end{cases} \tag{4.1}$$

which, in the case $f(x, y) \neq \infty$, is the makespan of a corresponding schedule. A suitable schedule can either be found by backtracking or maintaining suitable auxiliary data structures while evaluating the states.

Now, very similar as in [107], we discretize the state space of the dynamic program by defining a scaling factor $K := \epsilon C'/(3n)$ and introducing scaled job running times $q_j := \lceil p_j/K \rceil$ for each $j \in \{k+1, \ldots, n\}$. The values q_j are used for computation of the indices on the x and y axes while the values p_j are still used to compute the values for the states of the dynamic program, where now $x, y \in \{0, \ldots, \lceil C'/K \rceil\}$. Hence, the discretized makespans of schedules for the jobs in $\{k+1, \ldots, n\}$ now have the load

values Kx and Ky for the intervals A and C, respectively. Furthermore, the performed rounding decreases the runtime bound for evaluation to

$$O(n(C'/K)^2) = O(n(n/\epsilon)^2) = O(n^3\epsilon^{-2}).$$

In total, the values of f defined above are modified by replacing x by Kx and y by Ky in the maximum expression in (4.1); finally, the described algorithm yields the following result.

Theorem 37. *Scheduling with fixed jobs on two machines with one fixed job admits an FPTAS.*

Proof. We obtain $\lceil C'/K \rceil \in O(n/\epsilon)$, hence the runtime bound of the sketched algorithm is bounded by $O(n^3/\epsilon^2)$ which is polynomial in both $1/\epsilon$ and the encoding length of the instance. Furthermore the inequality

$$Kq_j \geq p_j > K(q_j - 1)$$

is valid for each $j \in \{k+1, \ldots, n\}$; with calculations similar to those in [107], for each $S \subseteq \{k+1, \ldots, n\}$ we obtain

$$\begin{aligned}
K\sum_{j\in S} q_j - \sum_{j\in S} p_j &\leq K\sum_{j\in S} q_j - \sum_{j\in S} K(q_j - 1) \\
&= K\sum_{j\in S} q_j - K\sum_{j\in S} + K|S| \\
&\leq Kn \\
&= \frac{\epsilon C'}{3n} n \\
&= \frac{\epsilon C'}{3} \\
&\leq \frac{\epsilon 3 C^*_{\max}}{3} \\
&= \epsilon C^*_{\max}.
\end{aligned}$$

Rearrangement yiels

$$K\sum_{j\in S} q_j \leq \sum_{j\in S} p_j + \epsilon C^*_{\max}$$

for each $S \subseteq \{k+1, \ldots, n\}$. In particular, the latter inequality is satisfied for suitable job sets $S_1, S_2 \subseteq \{k+1, \ldots, n\}$ which constitute the machine loads in A and C in an optimal schedule; in total this yields the desired approximation ratio. □

Note that with the same technique, it is possible to obtain an FPTAS for scheduling with fixed jobs for m constant where only one fixed job is present, for arbitrary m. Here scheduling the jobs indexed by $\{k+1,\dots,n\}$ in the interval $[t,\infty)$ using m machines via list scheduling yields a 3-approximation. Let C' denote the makespan of a feasible schedule obtained in this way. Without loss of generality, the fixed job is on the last machine; then, for a partial schedule σ, for each $i \in \{1,\dots,m-1\}$, we denote by $A_i(\sigma)$ the load on machine A_i in σ. Finally we denote by $A_m(\sigma)$ the load on machine m in the interval $[t,\infty)$ and by $B(\sigma)$ the load on machine m in the interval $[0,t)$. Correspondingly we define the states of our dynamic program; these can be organized as a table by defining

$$F[j,x_1,\dots,x_m] := \min\{\infty, \min\{B(\sigma) | \sigma \text{ is a schedule for the jobs in } \{k+1,\dots,j\} \text{ such that } A_i(\sigma) = x_i \text{ for each } i \in \{1,\dots,m\}\}\}$$

for each $j \in \{k+1,\dots,n\}$ and $x_1,\dots,x_m \in \{0,\dots,C'\}$, where again ∞ indicates the nonexistence of such a schedule; again we also suppose that F evaluates to infinity for negative values of x and y. With the same arguments as before we can obtain a result that generalizes the recurrence relation used for 2 machines.

Theorem 38. *The recurrence relation*

$$F[j,x_1,\dots,x_m] = \begin{cases} \min\{F[j-1, \\ \quad x_1,\dots,x_i-p_j,\dots,x_m] \mid \\ \quad i \in \{1,\dots,m\}\} & : F[j-1,x_1,\dots,x_m]+p_j > s \\ \min\{\min\{ \\ \quad F[j-1, \\ \quad x_1,\dots,x_i-p_j,\dots,x_m] \mid \\ \quad i \in \{1,\dots,m\}\}, \\ \quad F[j-1,x,y]+p_j\} & : F[j-1,x_1,\dots,x_m]+p_j \le s \end{cases}$$

for each $j \in \{k+2,\dots,n\}$,$x_1,\dots,x_n \in \{0,\dots,C'\}$ is satisfied for the function F defined above.

Based on this recurrence relation we can proceed similar as for the case with 2 machines; the details for the evaluation of the dynamic programming are omitted, but evaluation of the entire state space can be carried out within the pseudopolynomial runtime bound $O(nC'^m) \le O(n^{m+1}p_{\max}^m)$. In total, after evaluation of the state space,

we can solve the problem of scheduling with fixed jobs with one fixed job to optimality within the same runtime bound by selecting $x, y \in \{0, \ldots, C'\}$ in order to minimize the value

$$f(x_1, \ldots, x_m) := \begin{cases} \max\{x_1, \ldots, x_{m-1}, t + x_m\} & : F[n, x_1, \ldots, x_m] \neq \infty \\ \infty & : F[n, x_1, \ldots, x_m] = \infty \end{cases} \tag{4.2}$$

which, in the case $f(x_1, \ldots, x_m) \neq \infty$, is the makespan of a corresponding schedule. A suitable schedule can either be found by backtracking or maintaining suitable auxiliary data structures while evaluating the states.

As before, we discretize the state space of the dynamic program by defining

$$K := \epsilon C'/(3n)$$

and using scaled job running times $q_j := \lceil p_j/K \rceil$ for each $j \in \{k+1, \ldots, n\}$. Hence, the discretized makespans of schedules for the jobs in $\{k+1, \ldots, n\}$ now have the load values $Kx_1, \ldots, Kx_m$ in the intervals $A_1, \ldots, A_m$. respectively. Furthermore, the performed rounding decreases the runtime bound for evaluation to

$$O(n(C'/K)^m) = O(n(n/\epsilon)^m) = O(n^{m+1}\epsilon^{-m}).$$

In total, the values of f defined above are modified by replacing x_i by Kx_i for each $i \in \{1, \ldots, m\}$ in (4.2). Finally, the sketched algorithm yields the following result; the proof of the approximation ratio can be carried out very similar as the proof of Theorem 37.

Theorem 39. *Scheduling with fixed jobs on m machines where m constant with one fixed job admits an FPTAS.*

So far, the results obtained in this subsection are more of theoretical interest since the running times of the approximation schemes obtained will be out of practical interest. Hence, as a side issue, we are interested in the question what can be achieved by using simpler algorithms for MSSP. In [28] a greedy 2-approximation algorithm for MSSP with running time $O(n^2)$ is briefly mentioned; the subject is also discussed in [90], Subsection 10.4.1, with a slightly different approach. Here we present the algorithm from [28] in Figure 4.6.

Theorem 40. *The algorithm in Figure 4.6 is a 2-approximation algorithm for MSSP; furthermore this approximation ratio is asymptotically attained.*

1. Sort items by size in non-increasing order yielding $p_1 \geq \cdots \geq p_n$; sort knapsacks by capacity in non-decreasing order yielding $c_1 \leq \cdots \leq c_m$.

2. Iterate items in the order generated in Step 1; at each step, assign the current item to the knapsack with minimum index it can be feasibly packed into, if any. Discard the current item otherwise.

Figure 4.6.: Algorithm GreedyMSSP.

Proof. By the sorting generated in Step 1, w.l.o.g. we have $p_j \leq c_m$ for each $j \in \{1, \ldots, n\}$ since other items can not occur in any feasible solution. Let A be the assignment generated by the algorithm in Figure 4.6; we denote by $A(i)$ the total load that A assigns to knapsack i for each $i \in \{1, \ldots, m\}$. Let $U \subseteq \{1, \ldots, n\}$ be the set of items which are not packed by A. If $U = \emptyset$ all items are packed and A is an optimal assignment; hence suppose $U \neq \emptyset$. For each $j \in \{1, \ldots, n\}$ and $i \in \{1, \ldots, m\}$ we call j *admissible* to i if and only if $p_j \leq c_i$; furthermore a knapsack i is called *half-full* if and only if $A(i) \geq c_i/2$. If there are only half-full knapsacks the claim follows; hence suppose there are knapsacks which are not half-full and let $i' := \max\{i \in \{1, \ldots, m\} | i \text{ is not half-full}\}$. Aiming at a contradiction, assume $i' = m$, hence m is not half-full. A assigns at least one item to m, since otherwise $U \neq \emptyset$ is violated. Consequently because m is not half-full, A assigns only items of size at most $c_m/2$ to m. Since U contains the items which are not packed by A, U contains only items of size at most $c_m/2$, which A would assign to m, a contradiction; hence $i' < m$ holds. Let $c := c_{i'}$ and note that $c < c_{i'+1}$ holds; aiming at a contradiction, assume $c = c_{i'+1}$. Let j be the last item assigned to $i' + 1$ by A, which must exist since $i' + 1$ is half-full. If $i' + 1$ contains at least two items, they cannot be both larger than $c/2$, hence A would assign p_j to i' which is not half-full; this yields a contradiction. If $i' + 1$ contains only the item j, every item that A assigns to i' must be smaller than p_j; consequently, the algorithm in Figure 4.6 tries to pack j *before* every item packed in i'; since $c_{i'} = c_{i+1}$ item j is admissible to i' and packed there by A, a contradiction. In total, $c < c_{i'+1}$ holds.

A knapsack $i \in \{1, \ldots, m\}$ will be called *small* if and only if $i \in \{1, \ldots, i'\}$ and will be called *large* if and only if $i \in \{i' + 1, \ldots, m\}$; in a similar way, an item $j \in \{1, \ldots, n\}$ is called *small* if and only if $p_j \leq c$ and called *large* if and only if $p_i > c$. By this definition, a large item is not admissible to a small knapsack and every large bin is half-full.

We show that A packs every small item of the instance into a small knapsack. Aiming at a contradiction, assume that there is a small item $j \in U$. Item j is admissible to

knapsack i', so A tries to pack it there; i' is not half-full, so every item packed there prior to j is smaller than $c/2$. Consequently $p_j < c/2$, so A assigns j to knapsack i', a contradiction. Hence, every small item in the instance is packed. Next suppose that there is a small item j which A assigns to a large knapsack. However, it is tried to be packed in knapsack i' first. Since i' is not half-full, every item packed there prior to j is smaller than $c/2$. Consequently $p_j < c/2$ and A assigns j to a knapsack with index at most i', a contradiction. In total, every small item of the instance is packed into a small knapsack.

Let OPT be an optimal packing of the instance and let P_{OPT} denote its total profit. Let P^S_{OPT} be the total profit of small items in OPT and P^L_{OPT} be the total profit of large items in OPT; let P_A denote the total profit obtained by A and P^S_A, P^L_A denote the total profit of small and large items in A, respectively. In total we obtain

$$P_A = P^S_A + P^L_A \geq P^S_{\text{OPT}} + P^L_A \geq P^S_{\text{OPT}} + P^L_{\text{OPT}}/2 \geq P_{\text{OPT}}/2$$

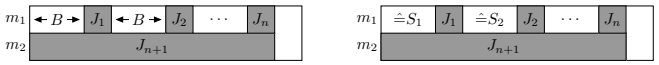
which yields the approximation ratio. The bound is tight even for one knapsack which can be seen by defining an instance with capacity $B \in \mathbb{N}^*$, n even, and 3 items $p_1 := B/2 + 1$ and $p_2 := p_3 := B/2$. Here the choice of items 2 and 3 yields an optimal profit of B, while the algorithm in Figure 4.6 selects item 1; since $\lim_{B\to\infty}(B/2+1)/B = 2$, the ratio is tight. □

By using the algorithm from Figure 4.6 instead of MSSPPTAS and changing the bound $1 - \epsilon/m$ to $1/2$ in Step 3 of the algorithm in Figure 4.4 we obtain an approximation algorithm with ratio $1 + m/2$ for scheduling with fixed jobs for m constant by following the lines of the proof of Theorem 35. In total, we obtain the following result. Note that in total, we obtain the same approximation ratio as in [109] for a more general problem; however, this generalization comes at the cost of a larger runtime bound.

Theorem 41. *Scheduling with fixed jobs for m constant admits a greedy algorithm with approximation ratio $1 + m/2$.*

4.2.2. Hardness Results

In this subsection we discuss the hardness of scheduling with fixed jobs for m constant. More precisely, scheduling with fixed jobs for m constant is strongly NP-hard via a straightforward reduction from 3-Partition, as discussed in [130, 131, 132]. However we



(a) Structure of fixed jobs, indicated by dark grey areas.

(b) Optimal solution of I'.

Figure 4.7.: Sketch illustrating the proof of Theorem 42.

include the proof for the sake of completeness. In total, scheduling with fixed jobs for m constant is substantially harder than $\mathrm{P}m||C_{\max}$ [127] which permits an FPTAS.

Theorem 42. *Scheduling with fixed jobs for m constant is strongly* NP*-hard for every $m \in \mathbb{N}$.*

Proof. We use a reduction from 3-Partition which is strongly NP-complete [47]; see Figure 4.7 for a sketch of the construction. Given an instance I of 3-Partition we define an instance I' of scheduling with fixed jobs for m constant. First we define n fixed jobs of size 1 by setting $p_j := 1$ for each $j \in \{1, \ldots, n\}$ and we define $m-1$ dummy jobs of size $n(B+1)$ by setting $p_j := n(B+1)$ for each $j \in \{n+1, \ldots, n+m\}$. These are the fixed jobs; the first n jobs function as delimiters on the first machine for the classes of the elements of I by fixing them via the list $(1, j(B+1)-1)$ for each $j \in \{1, \ldots, n\}$. The second $m-1$ dummy jobs just occupy the remaining machines by fixing them via $(j-n+1, 0)$ for each $j \in \{n+1, \ldots, n+m\}$. Finally concerning the non-fixed jobs, we set $p_{j-n-m} := a_j$ for each $\in \{n+m+1, \ldots, n+m+3n\}$ to copy the elements of I to I'. Note that by this construction I' can be generated from I in running time polynomial in the encoding length of I. Furthermore, note that I' has an optimal makespan of $C^*_{\max} = n(B+1)$ if and only if I is a yes-instance of 3-Partition by executing the small jobs according to the partition $S_1, \ldots, S_n$ in the intervals $[0, B), \ldots, [(n-1)(B+1), n(B+1)-1)$ on machine 1. Conversely in a schedule with makespan exactly $n(B+1)$ the small jobs must be put on machine 1 which indicates the partition of S into $S_1, \ldots, S_n$ since no more than 3 small jobs can fit into an interval of length B. In total, scheduling with fixed jobs for m constant is strongly NP-hard for any $m \in \mathbb{N}$. □

Since the objective values of feasible schedules for scheduling with fixed jobs for m constant are integral and $C^*_{\max} \leq \max\{s_j + p_j | j \in \{1, \ldots, k\}\} + P(\{k+1, \ldots, n\})$, the next result immediately follows from [46].

Corollary 43. *Scheduling with fixed jobs for m constant does not admit an FPTAS for any $m \in \mathbb{N}$ unless* P = NP*.*

In total, we have proved the second part of Theorem 33. Next we complement the algorithmic results obtained in Theorem 37 and Theorem 39.

Theorem 44. *Scheduling with fixed jobs for m constant does not admit an FPTAS, even if there is at most one fixed job per machine, for any $m \geq 2$ unless* $\mathsf{P} = \mathsf{NP}$.

Proof. We use a reduction from the following problem, Equal Cardinality Partition or ECP for short, which is NP-complete [47]; see Figure 4.8 for a sketch of the construction.

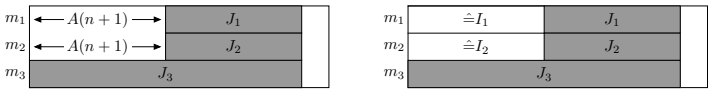
- *Given:* Finite list $I = (a_1, \ldots, a_n)$ of even cardinality with $a_i \in \mathbb{N}^*$ for each $i \in \{1, \ldots, n\}$, $A \in \mathbb{N}^*$ such that $\sum_{i=1}^{n} a_i = 2A$ holds.
- *Question:* Is there a partition of the list I into lists I_1 and I_2 such that $|I_1| = n/2 = |I_2|$ and $\sum_{i \in I_1} a_i = A = \sum_{i \in I_2} a_i$ holds?

Given an instance I of ECP we define an instance I' of scheduling with fixed jobs for $m \geq 2$ with at most one fixed job per machine as follows. First we define fixed jobs $p_1 := p_2 := A(n+1)$ with fixed positions $(1, A(n+1))$ and $(2, A(n+1))$; for each remaining machine with index $j \in \{3, \ldots, m\}$ greater than 2 we define a fixed job $p_j := 2A(n+1)$ with position $(j, 0)$. Finally we encode the items of I by defining a non-fixed small job $p_{m+j} := 2A + a_j$ for each $j \in \{1, \ldots, n\}$. I' is generated from I in running time polynomial in the length of I. Furthermore I' has an optimal makespan of $C^*_{\max} = 2A(n+1)$ if and only if I is a yes-instance by executing the small jobs according to the partition I_1 and I_2 on machines 1 and 2; conversely in a schedule with makespan $2A(n+1)$ the small jobs must run on machines 1 and 2 which indicates the partition of I into I_1 and I_2 since no more than $n/2$ jobs fit into a gap of length $A(n+1)$. Let I be a yes-instance of ECP and consider a suboptimal schedule of I'; the makespan of a suboptimal schedule of I' must be at least $2A(n+1) + A$ since every non-fixed job in I' has a processing time larger than A. Given an FPTAS for scheduling with fixed jobs for m constant, choose $\epsilon \in (0, 1)$ such that

$$1 + \epsilon < \frac{2A(n+1) + A}{2A(n+1)} = \frac{2n+3}{2n+2}$$

holds, which is equivalent to $\epsilon < 1/(2n+2)$; consequently ϵ can be chosen in such a way that $1/\epsilon$ is polynomially bounded in n and hence polynomially bounded in the encoding length of I. Then, the FPTAS generates a schedule with makespan $C_{\max}$ such that

$$C_{\max} \leq (1+\epsilon) C^*_{\max} < \frac{2A(n+1) + A}{2A(n+1)} 2A(n+1) = 2A(n+1) + A$$



(a) Structure of fixed jobs. (b) Optimal solution of I'.

Figure 4.8.: Sketch illustrating the proof of Theorem 44.

holds. Hence I' is solved to optimality in polynomial time and I is identified as a yes-instance of ECP, which is impossible unless $\mathsf{P} = \mathsf{NP}$. □

4.3. Scheduling with Non-Availability

In this section we present the results for scheduling with non-availability for m constant; more precisely, in Subsection 4.3.1, we present approximation algorithms. However, these use the assumption that at least one of the m machines is permanently available. If this is not the case, the problem is inapproximable, as discussed in Subsection 4.3.2, where we complement our approximation algorithms with suitable hardness results.

4.3.1. Approximation Algorithms

Again we use the same approach as in Subsection 4.2.1 to obtain a PTAS for scheduling with non-availability for m constant, for $m \geq 2$. Here we require the first machine to be permanently available; the reason for this assumption is discussed in Subsection 4.3.2. Later we discuss the cases which admit FPTASes for the case where only one interval of non-availability is permitted. As before, we use the algorithms in Figure 4.1 and Figure 4.2 to obtain the gaps for our approach, where q denotes the total number of gaps. Again we use a PTAS for MSSP where the knapsack capacites are permitted to be different from [18, 24, 72]; as before, this PTAS for MSSP is denoted by MSSPPTAS. However, in contrast to the algorithm in Figure 4.4, we use different lower and upper bounds for the binary search to find the target makespan t; more precisely, the lower bound for the binary search is 0 while the upper bound is the total processing times of the jobs to schedule. In total, the approach can be described as in Figure 4.9.

As we will see in the sequel, the algorithm in Figure 4.9 yields a PTAS for our problem.

Theorem 45. *The algorithm in Figure 4.9 is a PTAS for scheduling with non-availabiliy for m constant where at least one machine is permanently available.*

1. Use the algorithm in Figure 4.1 to generate A_i for each $i \in \{1, \ldots, m\}$.
2. Set $LB := 0$ and $UB := P(\{k+1, \ldots, n\})$.
3. While $UB - LB > 1$ repeat Steps 3.1–3.3.
   3.1 Set $t := \lfloor (LB - UB)/2 \rfloor$. Use the algorithm in Figure 4.2 to generate $A_i(t)$ for each $i \in \{1, \ldots, m\}$, the lists of gaps for each machine for fixed planning horizon $[0, t)$.
   3.2 Use MSSPPTAS with accuracy ϵ/m to select a set of jobs $S \subseteq \{k+1 \ldots, n\}$ such that

$$P(S) \geq (1 - \epsilon/m) \max\{P(S') | S' \subseteq \{k+1, \ldots, n\}, \\ S' \text{ permits a feasible packing into the intervals in } A_1(t), \ldots, A_m(t)\}.$$

   3.3 If $P(S) < (1 - \epsilon/m)P(\{k+1, \ldots, n\})$ then set $LB := t$ else store S and set $UB := t$.
4. Schedule the jobs in the last stored set S into the interval $[0, UB)$ as indicated by the solution generated by MSSPPTAS when S was returned; schedule the jobs in $\{k+1, \ldots, n\} \setminus S$ in the interval $[UB, \infty)$ on the first machine without unnecessary idle time.

Figure 4.9.: Algorithm NonAvailabilityScheduler.

Proof. Since the first machine is available at each time step $t \in [0, \infty)$, the sum of processing times $P(\{k+1, \ldots, n\})$ is an upper bound for the optimal makespan $C^*_{\max}$; hence in Step 2, the lower bound LB and the upper bound UB are initialized to have the following properties.

1. $LB \leq C^*_{\max}$.
2. There is a set $S \subseteq \{k+1 \ldots, n\}$ such that the jobs in S permit a feasible schedule into the time horizon $[0, UB)$ and $P(S) \geq (1 - \epsilon/m)P(\{k+1, \ldots, n\})$.

The second property is due to the fact that, since $C^*_{\max} \leq UB$, all jobs can be scheduled in $[0, UB)$ and thus it is impossible for the algorithm MSSPPTAS to return a set $S \subseteq \{k+1, \ldots, n\}$ such that $P(S) < (1 - \epsilon/m)P(\{k+1 \ldots, n\})$ holds; both properties are invariant under the update of LB and UB in Step 3.3. The number of iterations of the binary search in Step 3 is bounded by

$$\log P(\{k+1 \ldots, n\}) \leq \log(np_{\max}) = \log n + \log p_{\max}$$

which is polynomially bounded in the encoding length of I. On termination of the binary search in Step 3, $LB = UB$ holds, hence $UB \leq C^*_{\max}$ since $LB < C^*_{\max}$ is satisfied. This means that the set S selected in Step 4 can be scheduled in $[0, UB)$ and satisfies $P(S) \geq (1-\epsilon/m)P(\{k+1,\ldots,n\})$; hence $P(\{k+1,\ldots,n\} \setminus S) \leq \epsilon P(\{k+1,\ldots,n\})/m$ holds. Furthermore the jobs in $\{k+1,\ldots,n\} \setminus S$ can be scheduled on the first machine in $[UB, \infty)$ since the first machine is always available. We have

$$P(\{k+1,\ldots,n\})/m \leq C^*_{\max};$$

in total, the makespan of the schedule generated by the algorithm in Figure 4.9 is bounded by

$$UB + \epsilon P(\{k+1,\ldots,n\})/m \leq C^*_{\max} + \epsilon C^*_{\max} = (1+\epsilon)C^*_{\max}$$

and we obtain the desired approximation ratio. Since the running time of MSSPPTAS is polynomially bounded in q and n the claim is proved. □

With the analysis above, we have proved the first part of Theorem 34. However, since the running time of MKPPTAS may grow exponentially in $1/\epsilon$, the running time of the algorithm in Figure 4.9 may also grow exponentially in m. MSSP does not admit an FPTAS even for the special case of two knapsacks of equal capacity, unless $\mathsf{P} = \mathsf{NP}$ holds, as discussed in [90], Subsection 10.4. Hence it is impossible for the approach used above to yield an FPTAS for scheduling with non-availabilty for constant m where at least one machine is permanently available by replacing MSSPPTAS with a better algorithm, which is not surprising in the light of Corollary 52 in Subsection 4.3.2.

For $m = 1$ the situation is different. Lee [109] remarked that scheduling with non-availability for one machine is strongly NP-hard via reduction from 3-Partition. The problem is inapproximable in the general case by Theorem 49 and remains inapproximable if the number of non-availability intervals is restricted to two, as can be seen in Lemma 56 in Subsection 4.3.2. However, if there is only one interval of non-availability, an FPTAS can be obtained since SSP, the Subset Sum Problem, admits an FPTAS [88, 90]. This case corresponds to a simple knapsack problem; if all tasks can be scheduled before the interval of non-availability, we get an optimal solution; otherwise we use the FPTAS for SSP to schedule as much load as possible before the reservation. With this simple approach, we obtain the following result.

Theorem 46. *Scheduling with non-availability on one machine with one non-availability*

interval admits an FPTAS and is NP*-hard.*

As in [115] we study the case $m = 2$ with one interval of non-availability. Without loss of generality, the interval of non-availability is on machine 2 in the interval $[s, t)$. We show how to obtain an FPTAS based on dynamic programming and scaling the state space; the algorithmic approach used here is the same as in Subsection 4.2.1. Here $C' := P(\{k+1, \ldots, n\})$ yields a 2-approximation since all jobs can be scheduled on the first machine. Hence we have $C^*_{\max} \leq C' \leq 2C^*_{\max}$; furthermore we denote by A the interval $[0, \infty)$ on machine 1, by B the interval $[0, s)$ on machine 2 and by C the interval $[t, \infty)$ on machine 2. For a (partial) schedule σ we use $A(\sigma)$ to denote its load in A, $B(\sigma)$ to denote its load in B and $C(\sigma)$ to denote its load in C. In the same way as for scheduling with fixed jobs, the states of the dynamic program can be organized as a table by defining

$$F[j, x, y] := \min\{\infty, \min\{B(\sigma) | \sigma \text{ is a schedule for the jobs in } \{k+1, \ldots, j\} \text{ such that } A(\sigma) = x \text{ and } C(\sigma) = y\}\}$$

for each $j \in \{k+1, \ldots, n\}$ and $x, y \in \{0, \ldots, C'\}$, where ∞ indicates the nonexistence of such a schedule. In Theorem 36 we already established a suitable recurrence relation which permits to solve the problem to optimality via dynamic programming.

Hence, either inductively by iterating over $j \in \{k+1, \ldots, n\}$ or recursively using so-called lazy evaluation, we can solve the problem of scheduling with non-availability on two machines with one interval of non-availability to optimality. Inductive evaluation of all states of the dynamic program can be carried out as implemented in Figure 4.5. Again for ease of presentation we assume that the evaluation of $F[j, x, y]$ yields ∞ for negative values of x and y.

In total, evaluation of the entire state space can be carried out within the pseudopolynomial runtime bound $O(nC'^2) = O(n^3 p^2_{\max})$. In total, after evaluation of the state space, we can solve the problem of scheduling with non-availability with one interval of non-availability to optimality within the same runtime bound by selecting $x, y \in \{0, \ldots, C'\}$ in order to minimize the value

$$f(x, y) := \begin{cases} \max\{x, t + y\} & : F[n, x, y] \neq \infty, y > 0 \\ \max\{x, F[n, x, y]\} & : F[n, x, y] \neq \infty, y = 0 \\ \infty & : F[n, x, y] = \infty \end{cases}$$

which, in the case $f(x, y) \neq \infty$, is the makespan of a corresponding schedule. A suitable schedule can either be found by backtracking or maintaining suitable auxiliary data structures while evaluating the states.

Now, as in [107], we discretize the state space of the dynamic program by defining a scaling factor $K := \epsilon C'/(2n)$ and introducing scaled job running times $q_j := \lceil p_j/K \rceil$ for each $j \in \{k+1, \ldots, n\}$. The values q_j are used for computation of the indices on the x and y axes while the values p_j are still used to compute the values for the states of the dynamic program, where now $x, y \in \{0, \ldots, \lceil C'/K \rceil\}$. Hence, the discretized makespans of schedules for the jobs in $\{k+1, \ldots, n\}$ now have the load values Kx and Ky for the intervals A and C, respectively. In total, the values of f defined above are modified by replacing x by Kx and y by Ky in the maximum expressions; finally, the described algorithm yields the following result.

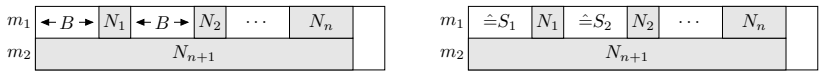
Theorem 47. *Scheduling with non-availability on two machines with one interval of non-availability admits an FPTAS.*

Proof. We obtain $\lceil C'/K \rceil \in O(n/\epsilon)$, hence the runtime bound of the sketched algorithm is bounded by $O(n^3/\epsilon^2)$ which is polynomial in both $1/\epsilon$ and the encoding length of the instance. Furthermore the inequality

$$Kq_j \geq p_j > K(q_j - 1)$$

is valid for each $j \in \{k+1, \ldots, n\}$; with calculations similar to those in [107], we obtain

$$\begin{aligned}
K\sum_{j\in S} q_j - \sum_{j\in S} p_j &\leq K\sum_{j\in S} q_j - \sum_{j\in S} K(q_j - 1) \\
&= K\sum_{j\in S} q_j - K\sum_{j\in S} + K|S| \\
&\leq Kn \\
&= \frac{\epsilon C'}{2n} n \\
&= \frac{\epsilon C'}{2} \\
&\leq \frac{\epsilon 2 C^*_{\max}}{2} \\
&= \epsilon C^*_{\max}.
\end{aligned}$$



(a) Structure of non-availability intervals.

(b) Optimal solution of I'.

Figure 4.10.: This sketch illustrates the proof of Theorem 49.

for each $S \subseteq \{k+1, \ldots, n\}$. Rearangement yiels

$$K \sum_{j \in S} q_j \leq \sum_{j \in S} p_j + \epsilon C^*_{\max}$$

for each $S \subseteq \{k+1, \ldots, n\}$. In particular, the latter inequality is satified for suitable job sets $S_1, S_2 \subseteq \{k+1, \ldots, n\}$ which constitute the machine loads in A and C in an optimal schedule; in total this yields the desired approximation ratio. □

Finally we remark that the construction above can be generalized very similar as in Theorem 39 to obtain the following result.

Theorem 48. *Scheduling with non-availability on m machines where m constant with one interval of non-availability admits an FPTAS.*

4.3.2. Hardness Results

Here first we show that scheduling with non-availability for m constant is inapproximable in the most general formulation. The proof is based on a construction by Lee [109], however there it was only remarked that LPT performs arbitrarily badly for the problem. Alternatively, the basic argument of the proof is similar to the inapproximability result from [39].

Theorem 49. *Scheduling with non-availability for m constant does not admit a polynomial time algorithm with a constant approximation ratio unless* $\mathsf{P} = \mathsf{NP}$.

Proof. Let $c \in \mathbb{R}$, $c \geq 1$; suppose there is an approximation algorithm A for scheduling with non-availability for m constant with ratio c. We use a reduction from the following strongly NP-complete problem 3-Partition [47]; see Figure 4.10 for a sketch of the proof.

- *Given:* Index set $S = \{1, \ldots, 3n\}$, $a_i \in \mathbb{N}^*$ for each $i \in S$ and $B \in \mathbb{N}^*$ such that $B/4 < a_i < B/2$ for each $i \in S$ and $\sum_{i=1}^{3n} a_i = nB$ holds.

- *Question:* Is there a partition of the set S into $S_1, \ldots, S_n$ such that $\sum_{i \in S_j} a_i = B$ holds for each $j \in \{1, \ldots, n\}$?

Given an instance I of 3-Partition we define an instance I' of scheduling with non-availability for m constant; we define intervals of non-availability by setting $p_j := 1$ for each $j \in \{1, \ldots, n-1\}$ and fixing these via $(1, j(B+1) - 1)$; furthermore we define an interval of non-availability by setting $p_n := n(B+1)(\lceil c \rceil - 1) + 1$ and fix this as $(1, n(B+1) - 1)$. Finally we have $m - 1$ intervals of non-availability of length $p_{n+j} := \lceil c \rceil n(B+1)$ for each $j \in \{1, \ldots, m-1\}$ and fix these via $(1 + j, 0)$ for each $j \in \{1, \ldots, m-1\}$. Finally we encode the items of I by setting $p_{n+m-1+j} := a_j$ for each $j \in \{1, \ldots, 3n\}$. I' can be generated from I in time polynomial in the encoding length of I and yields an optimal makespan $C^*_{\max} = n(B+1) - 1$ if and only if I is a yes-instance of 3-Partition; furthermore, any suboptimal schedule of I' for a yes-instance I of 3-Partition has a makespan $C_{\max} > \lceil c \rceil n(B+1)$. For any yes-instance I of 3-Partition, A generates a schedule for I' with makespan $C_{\max}$ such that

$$C_{\max} \leq cC^*_{\max} = c(n(B+1) - 1) < \lceil c \rceil n(B+1)$$

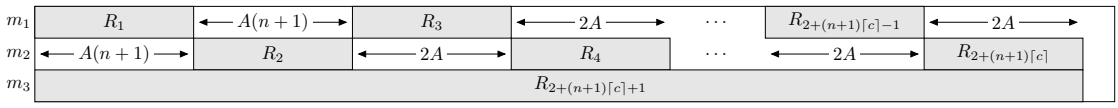
holds. Hence I is identified as a yes-instance of 3-Partition, which is impossible unless $\mathsf{P} = \mathsf{NP}$ holds. □

The inapproximability of the general case is due to the permission of intervals in which no machine is available. Hence it is reasonable to suppose that at each time step there is an available machine. However, as we will see next, this restriction is not sufficient to obtain a bounded approximation ratio.

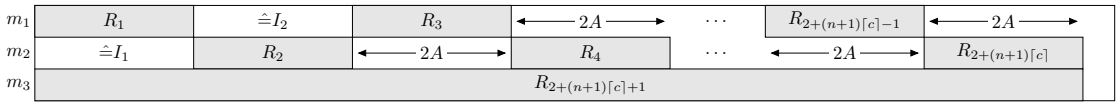
Theorem 50. *Scheduling with non-availability for m constant, even if for each time step there is an available machine, does not admit a polynomial time algorithm with a constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$.*

Proof. Let $c \in \mathbb{R}$, $c \geq 1$. Suppose there is an approximation algorithm B with ratio c for scheduling with non-availability where for each time step there is an available machine. We use a reduction from the following NP-complete problem Equal Cardinality Partition or ECP for short [47]. The construction is sketched in Figure 4.11.

- *Given:* Finite list $I = (a_1, \ldots, a_n)$ of even cardinality with $a_i \in \mathbb{N}^*$ for each $i \in \{1, \ldots, n\}$, $A \in \mathbb{N}^*$ such that $\sum_{i=1}^{n} a_i = 2A$ holds.



(a) In the structure of intervals of non-availability of the generated instance I', for every time step there is an available machine.



(b) For an optimal solution of I' for a yes-instance I we have $C^*_{\max} = 2A(n+1)$ and every suboptimal solution of I' has a makespan of $C_{\max} > 2A(n+1)(\lceil c\rceil + 1)$.

Figure 4.11.: This sketch illustrates the proof of Theorem 50.

- *Question:* Is there a partition of the list I into lists I_1 and I_2 such that $|I_1| = n/2 = |I_2|$ and $\sum_{i\in I_1} a_i = A = \sum_{i\in I_2} a_i$ holds?

Given an instance I of ECP we define an instance I' of scheduling with non-availability for arbitrary $m \geq 2$ where for each time step there is an available machine as follows. First we define two intervals of non-availability by setting

$$p_1 := p_2 := A(n+1)$$

and fixing these via $(1, 0)$ and $(2, A(n+1))$. Furthermore we define additional intervals of non-availability by setting

$$p_{2+\ell} := 2A$$

for each $\ell \in \{1, \ldots, (n+1)\lceil c\rceil\}$ and fix these via

$$(1 + (\ell - 1 \bmod 2), 2A(n+\ell))$$

for each $\ell \in \{1, \ldots, (n+1)\lceil c\rceil\}$. Furthermore we define dummy intervals of non-availability by setting

$$p_{2+(n+1)\lceil c\rceil+\ell} := 2A(n+1)(\lceil c\rceil + 1)$$

for each $\ell \in \{1, \ldots, m-2\}$ and fix these via

$$(2+\ell, 0)$$

for each $\ell \in \{1, \ldots, m-2\}$. Finally we copy the items of I by defining

$$p_{j+2+(n+1)\lceil c\rceil+m-2} := 2A + a_j$$

for each $j \in \{1, \ldots, n\}$. Note that I' can be generated algorithmically from I in a running time which is polynomially bounded in the encoding length of I. Furthermore, no job of I' can be scheduled in the interval $[2A(n+1), 2A(n+1)(\lceil c \rceil + 1))$. Hence I' has an optimal makespan of $C^*_{\max} = 2A(n+1)$ if and only if I is a yes-instance of ECP by executing the jobs according to the partition I_1 and I_2 on machines 1 and 2 in the time intervals $[0, A(n+1))$ and $[A(n+1), 2A(n+1))$, respectively. Conversely in a schedule with makespan $2A(n+1)$ all jobs must be scheduled during the time interval $[0, 2A(n+1))$, indicating the partition of I into I_1 and I_2 since no more than $n/2$ jobs fit into an availability interval of length $A(n+1)$. Let I be a yes-instance of ECP and consider a suboptimal schedule of I'. The makespan of a suboptimal schedule of I' must be at least $2A(n+1)(\lceil c \rceil + 1)$ since every job in I' has processing time larger than $2A$ and there must be a job which is not scheduled in $[0, 2A(n+1))$. Since the approximation ratio of B is c, the algorithm B generates a schedule with makespan $C_{\max}$ such that

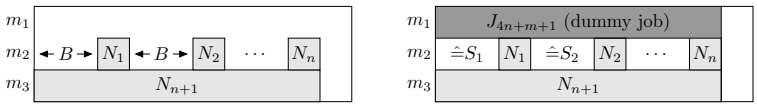
$$C_{\max} \leq cC^*_{\max} = c2A(n+1) < 2A(n+1)(\lceil c \rceil + 1)$$

holds. Hence I' is solved to optimality in polynomial time and I is identified as a yes-instance of ECP, which is impossible unless $\mathsf{P} = \mathsf{NP}$ holds. □

Consequently it is sensible to assume that at least one machine is always available; this assumption is used for all of the algorithms presented in Subsection 4.3.1. Next we present an inapproximability result which shows that the PTAS for scheduling with non-availability for m constant is close to best possible; hence scheduling with non-availability for m constant is substantially harder than $\mathrm{P}m||C_{\max}$ which permits an FPTAS [127].

Theorem 51. *Scheduling with non-availability for fixed m with the first machine permanently available is strongly* NP*-hard for $m \geq 2$.*

Proof. We use reduction from 3-Partition which is strongly NP-complete [47]; see Figure 4.12 for a sketch of the construction. Given an instance I of 3-Partition we define an instance I' of scheduling with non-availability for $m \geq 2$. We define n intervals of non-availability by setting $p_j := 1$ for $j \in \{1, \ldots, n\}$ and fixing these via $(2, j(B-1))$ for each $j \in \{1, \ldots, n\}$. Furthermore we define an interval of non-availability via $p_{n+j} := n(B+1)$ for each $j \in \{3, \ldots, m\}$ at position $(j, 0)$ to occupy all machines except the first two ones. Next we encode the items of I'; more precisely we set $p_{n+m+j} := a_j$ for each $j \in \{1, \ldots, 3n\}$. Finally we have a dummy job of size $p_{4n+m+1} := n(B+1)$. Note that I' can be generated from I in time polynomial in the length of I and has an optimal makespan of $C^*_{\max} = n(B+1)$ if and only if I is a yes-instance of 3-Partition



(a) Structure of non-availability intervals.

(b) Optimal solution of I'.

Figure 4.12.: Sketch illustrating the proof of Theorem 51.

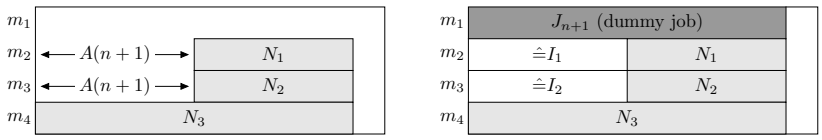
by putting the small jobs according to the existing partition $S_1, \ldots, S_n$ in the intervals $[0, B), \ldots, [(n-1)(B+1), n(B+1)-1)$ on machine 2 and putting the dummy job on machine 1; conversely in a schedule with makespan exactly $n(B+1)$ the dummy job must be put on machine 1 and hence the small jobs run on machine 2 which indicates the partition of S into $S_1, \ldots, S_n$ since no more than 3 small jobs can fit into an interval of length B. In total, scheduling with non-availability for $m \geq 2$ is strongly NP-hard. □

Since the objective values of feasible schedules for scheduling with non-availability are integral and $C^*_{\max} \leq P(I)$, the next result immediately follows from [46].

Corollary 52. *Scheduling with non-availability for m constant with the first machine permanently available does not admit an FPTAS for $m \geq 2$ unless* P = NP.

Again it is a natural question whether the problem becomes easier if the number of non-availability intervals per machine is restricted to one. Surprisingly, this is not the case, which can be shown by adaptation of a construction from [17]. The following result implies that scheduling with non-availability for m constant with at most one interval of non-availability per machine for $m \geq 3$ is strongly NP-hard.

Theorem 53. *Scheduling with non-availability for m constant with the first machine permanently available does not admit an FPTAS, even if there is at most one non-availability interval per machine, for $m \geq 3$ unless* P = NP.



(a) Structure of non-availability intervals.

(b) Optimal solution of I'.

Figure 4.13.: Sketch illustrating the proof of Theorem 53.

Proof. We use a reduction from the following problem, Equal Cardinality Partition or ECP for short, which is NP-complete [47]; see Figure 4.13 for a sketch of the construction.

- *Given:* Finite list $I = (a_1, \ldots, a_n)$ of even cardinality with $a_i \in \mathbb{N}^*$ for each $i \in \{1, \ldots, n\}$, $A \in \mathbb{N}^*$ such that $\sum_{i=1}^{n} a_i = 2A$ holds.
- *Question:* Is there a partition of the list I into lists I_1 and I_2 such that $|I_1| = n/2 = |I_2|$ and $\sum_{i \in I_1} a_i = A = \sum_{i \in I_2} a_i$ holds?

Given an instance I of ECP we define an instance I' of scheduling with non-availability for $m \geq 3$ as follows. First we define intervals of non-availability by $p_1 := p_2 := A(n+1)$ with fixed positions $(2, A(n+1))$ and $(3, A(n+1))$; for each remaining machine with index $j \in \{4, \ldots, m\}$ greater than 3 we define an interval of non-availability $p_{j-1} := 2A(n+1)$ with position $(j, 0)$. Furthermore we define a dummy job $p_m := 2A(n+1)$. Finally we encode the items of I by defining $p_{m+j} := 2A + a_j$ for each $j \in \{1, \ldots, n\}$. In total I' is generated from I in running time polynomial in the length of I. Furthermore I' has an optimal makespan of $C^*_{\max} = 2A(n+1)$ if and only if I is a yes-instance by executing the small jobs according to the partition I_1 and I_2 on machines 2 and 3 and putting the dummy job on machine 1; conversely in a schedule with makespan $2A(n+1)$ the dummy job is put on machine 1 and hence the small jobs run on machines 2 and 3 which indicates the partition of I into I_1 and I_2 since no more than $n/2$ jobs fit into an availability interval of length $A(n+1)$. Let I be a yes-instance of ECP and consider a suboptimal schedule of I'; the makespan of a suboptimal schedule of I' must be at least $2A(n+1) + A$ since every job in I' has a processing time larger than A and is scheduled either on machine $i \in \{1, \ldots, m\}$ or on machine 1 together with the dummy job, unless the dummy job is scheduled on a machine other than the first one. Given an FPTAS for scheduling with non-availability for m constant, choose $\epsilon \in (0, 1)$ such that

$$1 + \epsilon < \frac{2A(n+1) + A}{2A(n+1)} = \frac{2n+3}{2n+2}$$

holds, which is equivalent to $\epsilon < 1/(2n+2)$; consequently ϵ can be chosen in such a way that $1/\epsilon$ is polynomially bounded in n and hence polynomially bounded in the encoding length of I. Then, the FPTAS generates a schedule with makespan $C_{\max}$ such that

$$C_{\max} \leq (1+\epsilon)C^*_{\max} < \frac{2A(n+1) + A}{2A(n+1)} 2A(n+1) = 2A(n+1) + A$$

holds. Hence I' is solved to optimality in polynomial time and I is identified as a

yes-instance of ECP, which is impossible unless $\mathsf{P} = \mathsf{NP}$. □

Note that the construction uses at most one interval of non-availability per machine. Furthermore, by removing the first machine and the dummy job the construction can be modified to yield the following result. The idea shows that restriction of the most general case by permitting only one non-availability interval per machine is also hard to approximate for at least two machines.

Theorem 54. *Scheduling with non-availability for m constant does not admit an FPTAS, even if there is at most one non-availability interval per machine, for $m \geq 2$ unless $\mathsf{P} = \mathsf{NP}$.*

Theorem 53 does not cover the case $m = 2$ for which there is an FPTAS, see Theorem 47; however, for the case $m = 2$, we obtain a similar result if we permit an arbitrary constant number of non-availability intervals as in Theorem 51.

Theorem 55. *Scheduling with non-availability for $m = 2$ where the first machine is permanently available, if more than one interval of non-availability is permitted, does not admit an FPTAS unless $\mathsf{P} = \mathsf{NP}$.*

Proof. We use reduction from Equal Cardinality Partition, also denoted as ECP; given an instance I we define I' using 2 intervals of non-availability by setting

$$p_1 := p_2 := A(n+1)$$

and fixing these via $(2, A(n+1))$ and $(2, 3A(n+1))$. Furthermore we introduce *small* jobs by setting $p_{j+2} := 2A + a_j$ for each $j \in \{1, \ldots, n\}$ and a *dummy* job $p_{n+3} := 4A(n+1)$. Similar as before instance I' yields $C^*_{\max} = 4A(n+1)$ if and only if I is a yes-instance. Every suboptimal schedule of I' for a yes-instance I has a makespan at least $4A(n+1)+A$. Given an FPTAS for scheduling with non-availability for $m = 2$ where the first machine is permanently available with more than one interval of non-availability permitted, choose $\epsilon \in (0, 1)$ to satisfy

$$1 + \epsilon < \frac{4A(n+1) + A}{4A(n+1)} = \frac{4n+5}{4n+4},$$

which is equivalent to $\epsilon < 1/(4n+4)$; again ϵ can be chosen to have $1/\epsilon$ polynomially bounded. The FPTAS generates a schedule with makespan $C_{\max}$ such that

$$C_{\max} \leq (1+\epsilon)C^*_{\max} < 4A(n+1) + A;$$

hence I is identified as a yes-instance of ECP, which is impossible unless $\mathsf{P} = \mathsf{NP}$. □

Next, we discuss the hardness of scheduling with non-availability for $m = 1$ if more than one interval of non-availability is permitted; we obtain the following inapproximability result.

Lemma 56. *Scheduling with non-availability for $m = 1$, if more than one interval of non-availability is permitted, does not admit a constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$.*

Proof. Let $c \in \mathbb{R}$, $c \geq 1$; suppose there is an approximation algorithm A for scheduling with non-availability for $m = 1$ where one interval of non-availability is permitted with ratio c. For an instance I of Partition, which is known to be NP-complete [47], given by $I = \{a_1, \ldots, a_n\}$ such that $\sum_{i \in I} a_i = 2B$, define an instance I' of scheduling with non-availability for $m = 1$ where more than one interval of non-availability is permitted by defining two intervals of non-availability via $p_1 := 1$ and $p_2 := \lceil c \rceil (2B + 2) - (2B + 1)$ and fixing these via $(1, B)$ and $(1, 2B + 1)$. Finally we copy the items of I by setting $p_{2+j} := a_j$ for each $j \in \{1, \ldots, n\}$.

Then I is a yes-instance of Partition if and only if I' has an optimal makespan of $C^*_{\max} = 2B + 1$. However, any suboptimal schedule of I' for a yes-instance I of Partition has a makespan $C_{\max} > \lceil c \rceil (2B + 2)$. For any yes-instance I of Partition, A generates a schedule for I' with makespan $C_{\max}$ such that $C_{\max} \leq cC^*_{\max} = c(2B + 1) < \lceil c \rceil (2B + 2)$ holds. Hence, I is identified as a yes-instance of Partition, which is impossible unless $\mathsf{P} = \mathsf{NP}$ holds. □

4.4. Conclusion

In this chapter we have studied scheduling on a constant number of identical parallel machines with fixed jobs and non-availability. For both problems we obtained approximation algorithms which are tight in the sense that we obtained PTASes where FPTASes do not exist unless $\mathsf{P} = \mathsf{NP}$ holds; as an important tool we have used a PTAS for MSSP. More precisely, for scheduling with fixed jobs, we obtained the same result as in [130, 131, 132]. However, we have shown that the interesting special case where only one fixed job is permitted admits an FPTAS. On the other hand, permitting fixed jobs on all machines but restricting the number of fixed jobs per machine to one does not make the problem easier than the general case. Furthermore, we have presented a faster greedy algorithm based on a greedy algorithm for MSSP. In total, the complexity results for scheduling with fixed jobs can be summarized as in Table 4.1.

Table 4.1.: Complexity results for scheduling with fixed jobs.

Problem	$m = 1$	$m = 2$	$m \geq 3$
arbitrary fixed jobs	strongly NP-hard, PTAS, no FPTAS unless P = NP		
at most one fixed job per machine	NP-hard, FPTAS	strongly NP-hard, PTAS, no FPTAS unless P = NP	
at most one fixed job in total	NP-hard, FPTAS		

For scheduling with non-availability the situation is slightly different. Here we have shown via Theorem 49 and Theorem 50 that it is necessary to have at least one machine permanently available to obtain a bounded approximation ratio. From a complexity point of view, this restriction then behaves in a similar way as scheduling with fixed jobs. The results can be summarized as in Table 4.2.

Table 4.2.: Complexity results for scheduling with non-availability.

Problem	$m = 1$	$m = 2$	$m \geq 3$
arbitrary non-availability intervals	no polynomial time algorithm with constant approximation ratio unless P = NP		
at most one non-availablity interval per machine	NP-hard, FPTAS	no polynomial time algorithm with constant approximation ratio unless P = NP	
arbitrary non-availability intervals, at least one machine permanently available	P ($k = 0$)	strongly NP-hard, PTAS, no FPTAS unless P = NP	
at most one non-availability interval par machine, at least one machine permanently available	P ($k = 0$)	NP-hard, FPTAS	strongly NP-hard PTAS, no FPTAS unless P = NP
at most one non-availability interval in total	NP-hard, FPTAS		

Finally, it is also an interesting question how the problems under consideration can be approximated if the number m of machines is considered as part of the input; this question is addressed in the next chapter.

5. Constrained Scheduling for m Part of the Input

In this chapter we study the problem of non-preemptively scheduling n independent sequential jobs, given by their processing times, on a system of m identical parallel machines; more precisely, unlike in Chapter 4, m is considered to be part of the input. The objective is to minimize the makespan $C_{\max}$, which is the maximum completion time of all jobs. However, our scheduling problem will be additionally constrained in one of two ways.

- The first type of constraint we study is a subset of the n jobs being fixed in the system, i.e. the executing machine and starting time for each of the fixed jobs is already prespecified in the instance. This setting is called *scheduling with fixed jobs*.

- The second type of constraint we study is non-availability of machines during prespecified time intervals which are given as a part of the instance. This setting is called *scheduling with non-availability*.

Here, the same perspective and motivation as in Chapter 4 apply. Both problems are closely related, in particular they can be formally described via the same encoding scheme of instances. However, the objective function $C_{\max}$ behaves differently for both settings. On one hand, for scheduling with fixed jobs, the fixed jobs contribute to the makespan; for this setting, the objective function models the behaviour of the system from the perspective of the *administrator* who wishes to execute and complete *all* submitted jobs as soon as possible. On the other hand, for scheduling with non-availability, the intervals of non-availabity do *not* contribute to the makespan; for this setting, the objective function models the behaviour of the system from the perspective of a *user* who wishes to execute and complete his or her *own* submitted jobs as soon as possible. Needless to discuss, both formulations are in additional ways practically relevant; the first one

occurs in parallel computing platforms since high-priority jobs are present in the system while the second one may occur due to regular maintenance of machines.

For both problem formulations we contribute approximation algorithms with tight ratios. Scheduling with fixed jobs for m part of the input was briefly mentioned in [130, 131, 132] where it was shown not to admit an approximation algorithm with ratio better than 3/2 unless $\mathsf{P} = \mathsf{NP}$. We complement this negative result by presenting an approximation algorithm with ratio $3/2 + \epsilon$; later this ratio is refined to 3/2. For scheduling with non-availability, similar to Chapter 4, we show that a restriction (namely a setting where a constant fraction of the machines is permanently available) is necessary to obtain a bounded approximation ratio. This restriction admits an approximation algorithm with ratio $3/2+\epsilon$, but an approximation algorithm with ratio better than 3/2 is ruled out unless $\mathsf{P} = \mathsf{NP}$ holds. With the same techniques used for scheduling with fixed jobs, the approximation ratio of this algorithm is refined to 3/2. Furthermore, as a side issue, we study a weaker restriction of scheduling with reservations, namely the one where only one machine is permanently available. For this restriction, the existence of an approximation algorithm with constant ratio implies the existence of an approximation algorithm for Bin Packing with additive error. However, whether or not the latter exists is an interesting open problem, as discussed in [58], Chapter 2, page 67. Parts of this chapter have been accepted for publication [34].

5.1. Introduction

In parallel machine scheduling, an important issue is the scenario where either some jobs are already fixed in the system [130, 131, 132] or intervals of non-availability of some machines must be taken into account [35, 63, 109, 113, 115]. The first problem occurs since high-priority jobs are present in the system while the latter problem is due to regular maintenance of machines; both models are relevant for turnaround scheduling [121] and overlay computing where machines are donated on a volunteer basis.

Note that, from a naïve point of view, the two problems studied here are the same as the ones discussed in Chapter 4; however, regarding the number m of machines part of the input makes the problem harder to approximate, as we shall see in the sequel.

5.1.1. Problem Definition

Both problems under consideration can be described by the same encoding of instances and only differ in the objective function. An instance consists of m, the number of machines, which is part of the input, and n jobs given by processing times $p_1, \ldots, p_n \in \mathbb{N}$. The first k jobs are fixed via a list $(m_1, s_1), \ldots, (m_k, s_k)$ giving a machine index and starting time for the respective job. We assume that these fixed jobs do not overlap. A schedule is a non-preemptive assignment of the jobs to machines and starting times such that the first k jobs are assigned as encoded in the instance and that the jobs do not intersect.

If the objective is to minimize the makespan for *all* jobs including the fixed ones, we call the problem *scheduling with fixed jobs*. Alternatively we can regard the k fixed jobs as intervals of non-availability which do not contribute to the makespan. Here the objective is to minimize the makespan over the *non-fixed* jobs only; this problem is called *scheduling with non-availability*. For the latter problem, we denote by $\rho \in (0, 1)$ the *percentage* of machines which are permanently available and also permit infinite length of the non-availability intervals.

In the sequel we use $P(I) := \sum_{j=1}^{n} p_j$ to denote the total processing time of an instance I and for each $S \subseteq \{1, \ldots, n\}$ we write $P(S) := \sum_{j \in S} p_j$ for the total processing time of S. Finally let $p_{\max} := \max\{p_j | j \in \{k+1, \ldots, n\}\}$ denote the maximum processing time of non-fixed jobs. More formally, a schedule is a function $\sigma : \{k+1, \ldots, n\} \to \{1, \ldots, m\} \times [0, \infty)$ which maps each job to its executing machine and starting time; furthermore σ is required to be non-preemptive and there may be no intersection between the jobs or the jobs and non-availability intervals. If σ is clear from the context it may be dropped from notation. Furthermore, for a fixed schedule σ, for each $j \in \{1, \ldots, n\}$ let $s_j(\sigma)$ denote the starting time of the non-fixed job j; since k objects are fixed in the schedule, we have $s_j(\sigma) = s_j$ for any schedule σ and $j \in \{1, \ldots, n\}$, i.e. the starting times of the first k objects do not depend on the schedule. Finally, for a fixed schedule σ, for each $j \in \{1, \ldots, n\}$ let $C_j(\sigma) := s_j(\sigma) + p_j$ denote the finishing time of the fixed object j.

As mentioned before, for either problem formulation the objective is to minimize the makespan $C_{\max}$. For scheduling with fixed jobs, for any schedule σ the objective is defined by

$$C_{\max}(\sigma) := \max\{C_j | j \in \{1, \ldots, n\}\}$$

while for scheduling with non-availability, for any schedule σ the objective is defined by

$$C_{\max}(\sigma) := \max\{C_j | j \in \{k+1, \ldots, n\}\}.$$

Furthermore, for both problem formulations we use $C^*_{\max}$ to denote the optimal makespan.

In the literature, scheduling with non-availability is also called *non-resumable scheduling with availability constraints* [35, 109, 113, 115]. The makespan $C_{\max}$ is one of the most well-studied objectives in the field of scheduling; for this objective, most problem formulations permit good approximation algorithms. However, both problems generalize the well-known problem $P||C_{\max}$ [60] and hence are strongly NP-hard and also hard to approximate.

5.1.2. Results

Scheduling with fixed jobs was studied by Scharbrodt, Steger & Weisser [130, 131, 132]. They mainly studied the problem for m constant; for this strongly NP-hard formulation (which consequently does not admit an FPTAS) they present a PTAS. They also found approximation algorithms for general m with ratios 3 [130] and $2 + \epsilon$ [132]; since the finishing time of the last fixed job is a lower bound for $C^*_{\max}$, we can simply use a PTAS for the well-known problem $P||C_{\max}$ [60] to schedule the remaining $n - k$ jobs after the fixed job which finishes last. Finally, Scharbrodt, Steger & Weisser [132] proved that for scheduling with fixed jobs there is no approximation algorithm with ratio $3/2 - \epsilon$, unless P = NP, for any $\epsilon \in (0, 1/2]$. Complementing this negative result, we obtain a tight ratio with our new approach.

Theorem 57. *Scheduling with fixed jobs for m part of the input admits an approximation algorithm with ratio* $3/2 + \epsilon$ *for any* $\epsilon \in (0, 1/2]$. *Furthermore, scheduling with fixed jobs admits an approximation algorithm with ratio* $3/2$. *Finally, scheduling with fixed jobs for m part of the input does not admit a polynomial time approximation algorithm with ratio better than* $3/2 - \epsilon$, *for any* $\epsilon \in (0, 1/2]$, *unless* P = NP.

Unlike scheduling with fixed jobs, scheduling with non-availability without any further restriction is inapproximable within a constant ratio unless P = NP, as shown by Eyraud-Dubois, Mounié & Trystram [39]. The inapproximability is circumvented by requiring at least one machine to be permanently available. The case with m constant, arbitrary non-availability intervals, and at least one machine permanently available, is strongly NP-hard but can be solved by a PTAS by Diedrich et al. [35] which is presented in

Chapter 4. For general m, researchers so far have only studied the problem where there is at most one interval of non-availability per machine. First, the even more restricted case where the intervals of non-availability start at time zero was studied. Here Lee [108] and Lee et al. [110] proved that LPT yields a ratio of $3/2 - 1/(2m)$ and can be modified to yield a ratio of $4/3$. For the same problem, Kellerer [86] found an algorithm with a tight ratio of $5/4$. Furthermore, Hwang et al. [63] briefly pointed out that this problem admits a PTAS. A more general case is the setting where the at most one interval per machine may have an arbitrary position. For this problem Lee [109] showed that general list scheduling yields a ratio of m and proved a tight ratio of $1/2 + m/2$ for LPT. Hwang et al. studied the ratio of LPT for the same scenario but assumed that at least $m - \lambda$ machines are available simultaneously. They first obtained a ratio of 2 for $\lambda \leq m/2$ [62] which they later refined to a ratio of $1 + \lceil 1/(1 - \lambda/m) \rceil /2$ for λ arbitrary [63]. For $\lambda = m - 1$, this yields $1 + m/2$; if $\rho = (m - \lambda)/m$ denotes the percentage of permanently available machines, this yields $1 + \lceil 1/\rho \rceil /2$ which depends on ρ. Concerning further results, we refer the reader to [113], Chapter 22, or [129] for surveys. For the sake of completeness, some results about single-machine problems can be found in the articles [35, 82, 108]. Finally, for scheduling with non-availability, our new technique yields an improved approximation ratio independent from ρ which is tight, as shown in Subsection 5.3.2.

Theorem 58. *Scheduling with non-availability, where the percentage $\rho \in (0, 1)$ of permanently available machines is constant, admits an approximation algorithm with ratio $3/2 + \epsilon$ for any $\epsilon \in (0, 1/2]$. Furthermore, this problem admits an approximation algorithm with ratio $3/2$. Finally, for this problem there is no approximation algorithm with ratio $3/2 - \epsilon$, unless $\mathsf{P} = \mathsf{NP}$, for any $\epsilon \in (0, 1/2]$.*

In addition, we show that approximation of scheduling with non-availability within a constant ratio is at least as hard as approximation of Bin Packing with an additive error; however, whether this is possible is an interesting open problem, as discussed in [58], Chapter 2, page 67.

5.1.3. Techniques Used in our Approach

In contrast to previous approaches we use a new technique for rounding and assignment of large jobs which is carried out via a class of network flow problems. To bound the error incurred by this way of assignment, we use an interesting cyclic shifting technique and a redistribution argument. We believe that this approach for rounding and assignment

of suitable items will find other applications in related packing or scheduling problems. Furthermore, we use techniques like dual approximation [59], partition of the instance, linear grouping and rounding known from Bin Packing [30] or Strip Packing [91, 93], and definition of configurations. Our modelization also involves the multiple subset sum problem which is also denoted by MSSP. As an algorithmic building block we use a PTAS for MSSP from [18] where the knapsack capacities are permitted to be different. Alternatively, a PTAS for the multiple knapsack problem (MKP) can be used [23, 24, 72]. In particular, if the number of target areas is large, the recent PTAS by Jansen [72] yields a runtime bound which is polynomial in both $1/\epsilon$ and the encoding size of the instance. Knapsack type problems belong to the oldest and most fundamental problems in combinatorial optimization and theoretical computer science; we refer the reader to [90, 118] for in-depth surveys or the papers [18, 24, 64, 72, 107] for literature on these problems.

The remainder of Chapter 5 is organized as follows. In Section 5.2, we present results for scheduling with fixed jobs for m part of the input. More precisely, in Subsection 5.2.1 we discuss the algorithmic results while in Subsection 5.2.2 we complement these by suitable inapproximability results. In Section 5.3 we present our new results for scheduling with non-availability for m part of the input. More precisely, in Subsection 5.3.1 we present an approximation algorithm which is complemented in Subsection 5.3.2 by a suitable hardness result. Finally we finish Chapter 5 in Section 5.4 with a conclusion.

5.2. Scheduling with Fixed Jobs

In this section we describe the results for scheduling with fixed jobs for m part of the input. In Subsection 5.2.1 we present approximation algorithms; these are complemented in Subsection 5.2.2 with suitable hardness results.

5.2.1. Algorithms

In this section we prove the first and second part of Theorem 57. We may assume that $m \leq n$. Otherwise, we have $m > n$, and in this case there are at least $m - k$ machines without fixed jobs. Since we have exactly $n - k$ non-fixed jobs, every job that has to be scheduled can be executed on a free machine of its own, solving the instance to optimality.

Our modelization is based on the multiple subset sum problem (MSSP) which can be

1. Set $\epsilon' := \epsilon/3$. Set $LB := C^{\text{fix}}_{\max}$ and $UB := C^{\text{fix}}_{\max} + np_{\max}$. Let σ_{saved} be the empty schedule.

2. While $UB - LB > 1$ repeat Steps 2.1–2.3.

   2.1 Set $T := \lceil (UB + LB)/2 \rceil$. Generate gap sets $G_L(T)$ and $G_S(T)$. Generate the sets $J_L(T)$, $J_M(T)$ and $J_S(T)$, as described in Subsubsection "Job Classification and Generation of Gaps". Apply linear grouping and rounding to the jobs in $J_M(T)$ and generate all possible configurations $\kappa^{(1)}, \ldots, \kappa^{(c_2)}$. Set *found* := *false*.

   2.2 For each possible choice of the values $q'_i(T,k)$ for each interval index $k \in \{1, \ldots, c_3\}$ and group index $i \in \{1, \ldots, c_4 + 1\}$ execute Step 2.1.

      2.2.1 For each possible choice of values $c(k,i,\ell)$ for $(k,i,\ell) \in \mathcal{I}$ execute Steps 2.2.1.1–2.2.1.2.

         2.2.1.1 Generate the network flow model $N(T)$ for the specific choice of configurations, values $q'_i(T,k)$ and values $c(k,i,\ell)$ as described in Subsubsection "Assignment of Jobs to Large Gaps via Network Flow" and solve it. If the value of the network flow is smaller than $|J_L(T)|$, proceed with the next iteration of the loop in Step 2.2.1. Otherwise assign the jobs in $J_L(T)$ to the gaps in $G_L(T)$ as indicated by the network flow, resulting in a schedule σ. Use a PTAS for MSSP as described in Subsubsection "Packing of Medium and Small Jobs" to add a suitable subset of $J_M(T) \cup J_S(T)$ to the schedule σ. Let $P'(\sigma)$ be the total processing time of jobs *not* scheduled in σ.

         2.2.1.2 If $P'(\sigma) > 3\epsilon' Tm$, proceed with the next iteration of the loop in Step 2.2.1. If $P'(\sigma) \leq 3\epsilon' Tm$, set $\sigma_{\text{saved}} := \sigma$, set *found* := *true*, go to Step 2.3.

   2.3 If *found* := *true* set $UB := T$ else set $LB := T$.

3. Use the list scheduling algorithm from Subsubsection "Packing of Medium and Small Jobs" to add the jobs which are not yet scheduled in σ after the makespan of σ.

Figure 5.1.: The approximation algorithm for scheduling with fixed jobs. We assume that each loop is taken to the next iteration if T, the values $q'_i(T,k)$ of the values $c(i,k,\ell)$ are determined to be infeasible.

formally defined as follows. We are given a set $\{1, \ldots, n\}$ of items, each item i having a positive integer weight w_i, and a set $\{1, \ldots, m\}$ of knapsacks, each knapsack j having a nonnegative integer capacity c_j; the objective is to select a subset of items of maximum total weight that can be packed into the knapsacks.

Our algorithm is described in Figure 5.1. It is based on the dual approximation paradigm [59] by using binary seach on the makespan. First we set $\epsilon' := \epsilon/3$.

For any subset $S \subseteq \{k+1, \ldots, n\}$ let

$$P(S) := \sum_{i \in S} p_i$$

denote the total processing time of S and for any $j \in \{1, \ldots, k\}$ let

$$C_j := s_j + p_j$$

denote the completion time of the fixed job j. Let

$$C_{\max}^{\text{fix}} := \max\{C_j | j \in \{1, \ldots, k\}\}.$$

Note that

$$C_{\max}^{\text{fix}} \leq C_{\max}^{*} \leq C_{\max}^{\text{fix}} + np_{\max}$$

holds, where $p_{\max} := \max\{p_j | j \in \{k+1, \ldots, n\}\}$ denotes the maximum processing time of the non-fixed jobs. In total, the remaining $n-k$ jobs indexed by $\{k+1, \ldots, n\}$ can be scheduled on one machine in the interval $[C_{\max}^{\text{fix}}, C_{\max}^{\text{fix}} + np_{\max})$. If we use binary search as in the outermost loop in the algorithm in Figure 5.1, we obtain a search space of size at most $np_{\max}$ for the target makespan; we will find a suitable target makespan (i.e. one for which we can schedule all large jobs and almost all load) in $O(\log(np_{\max}))$ steps which is polynomially bounded in the encoding size of the instance. If the algorithm in Figure 5.1 reaches Step 3, the upper bound UB is the smallest target makespan for which in Step 2.2.1.2 a suitable schedule can be found. As we will see in the following, $C_{\max}^{*}$ is also a suitable schedule, which means that if we reach Step 3, we have $UB \leq C_{\max}^{*}$.

For any target makespan T, we use the technique described below which involves a PTAS for MSSP [18, 72] to schedule as much load as possible in the interval $[0, T)$. In the sequel we show that for the optimal makespan $T = C_{\max}^{*}$, we can algorithmically find a schedule which executes almost all load in the interval $[0, C_{\max}^{*})$; the remainig load is put in the interval $[C_{\max}^{*}, \infty)$ via list scheduling, causing an error which will

be suitably bounded however. In Subsubsections "Job Classification and Generation of Gaps"–"Packing of Medium and Small Jobs" let $T \in [C_{\max}^{\text{fix}}, C_{\max}^{\text{fix}} + np_{\max})$ denote a candidate for the makespan; we call such a T *feasible* if there is a schedule with makespan at most T and infeasible otherwise. Furthermore, the k fixed jobs are preassigned as indicated by $(m_1, s_1), \ldots, (m_k, s_k)$.

In the sequel, the presentation of our algorithm is presented in subsubsections. We have chosen this more structured way of presentation unlike in the previous chapter since here the ideas are more involved and have to be described in more detail.

Job Classification and Generation of Gaps

For T we generate all intervals of availability of a machine, in the following called *gaps*, within the planning horizon $[0, T)$ from the encoded fixed jobs. This can be easily achieved in time polynomially bounded in the instance size by processing the starting times and execution times of the fixed jobs, as discussed in detail in Subsection 4.2.1.

Let $q(T) \in \mathbb{N}^*$ denote the number of gaps and let $G(T) := \{G_1, \ldots, G_{q(T)}\}$ denote the set of gaps. For each $i \in \{1, \ldots, q(T)\}$ we also use G_i to denote the size of gap G_i. Note that

$$|q(T)| \leq k + m \leq 2n$$

since at most k fixed jobs induces a gap "left" to it and there are at most m gaps whose "right" limit is not created by a fixed job but by the limit of the planning horizon. In total, $|q(T)|$ is polynomially bounded in the instance size. The set of gaps is partitioned into *large* and *small* gaps via

$$\begin{aligned} G_L(T) &:= \{G \in G(T) | G > T/2\}, \\ G_S(T) &:= \{G \in G(T) | G \leq T/2\}. \end{aligned}$$

Let $q_L(T) := |G_L(T)|$, $q_S(T) := |G_S(T)|$ be the number of large and small gaps for target makespan T. Since there is at most one large gap per machine, we have $q_L(T) \leq m$. We define

$$\begin{aligned} J_L(T) &:= \{i \in \{k+1, \ldots, n\} | p_i \in (T/2, T]\}, \\ J_M(T) &:= \{i \in \{k+1, \ldots, n\} | p_i \in (\epsilon' T, T/2]\}, \\ J_S(T) &:= \{i \in \{k+1, \ldots, n\} | p_i \in (0, \epsilon' T]\} \end{aligned}$$

to partition the set of non-fixed jobs into *large*, *medium* and *small* jobs. Note $J_L(T)$

can be only packed into the at most m gaps of $G_L(T)$. Hence, if $|J_L(T)| > q_L(T)$, T is infeasible and can be discarded. In the sequel we assume that there are no unnecessary idle times in the gaps, i.e. a set of jobs placed in a gap is scheduled as a continuous block which starts as early as possible and the idle times are positioned at the end of the gap.

Definition of Configurations for Medium Jobs

We obtain few distinct job sizes in $J_M(T)$ via rounding. This construction removes some jobs from the schedule; these will not be executed in $[0, T)$ anymore. However, the total processing time of the removed jobs can be suitably bounded. Now fix a makespan T and a schedule σ with makespan T; note that the construction will be valid in particular for $T = C^*_{\max}$, an argument which will be used later. Since $p_i > \epsilon' T$ for each $i \in J_M(T)$ and the interval $[0, T)$ provides an amount of total processing time of at most mT,

$$n' := |J_M(T)| \leq \lfloor \frac{mT}{\epsilon' T} \rfloor = \lfloor \frac{m}{\epsilon'} \rfloor \leq \frac{m}{\epsilon'}$$

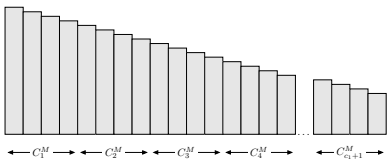
holds. We apply linear grouping and rounding as in [30] to the medium jobs in $J_M(T)$ by setting $c_1 := \lceil 1/\epsilon'^2 \rceil$ and creating $c_1 + 1$ groups. The following construction is sketched in Figure 5.2.

We sort the medium jobs in $J_M(T)$ in non-increasing order of processing time and in this order create groups of cardinality $\lfloor n'/c_1 \rfloor$ where the last group is possibly of smaller cardinality. We denote the resulting groups by $C^M_1, \ldots, C^M_{c_1+1}$. Next for each $i \in \{1, \ldots, c_1 + 1\}$ the processing times of medium jobs in C^M_i are rounded up to the largest processing time occurring in the respective group, i.e. we define

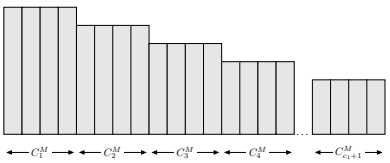
$$q_i := \max\{p_j | j \in C^M_i\}$$

and the resulting rounded groups are denoted by $\tilde{C}^M_1, \ldots, \tilde{C}^M_{c_1+1}$. We remove C^M_1 from σ, resulting in a *partial* schedule with makespan at most T and some free space. Note that by embedding the items of $\tilde{C}^M_i$ into the space for $\tilde{C}^M_{i-1}$ for each $i \in \{2, \ldots, c_1\}$, we can reschedule the medium jobs in $J_M(T) \setminus C^M_1$ based on the assignment in σ. We use σ_1 to denote the resulting partial schedule.

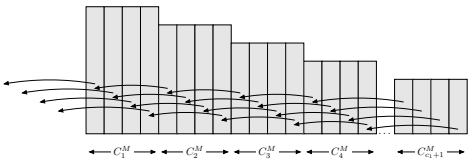
Using this approach we have limited the number of distinct item sizes in $J_M(T)$ to c_1 at the cost of removing C^M_1 from the schedule. In total, we have $|C^M_1| \leq n'/c_1$ and each job in C^M_1 is no larger than $T/2$. Hence, the total processing time of C^M_1 can be



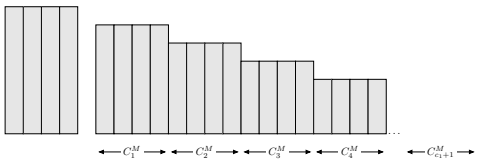
(a) Arrangement of the jobs in $J_M(T)$ in non-increasing order of size. Creation of $c_1 + 1$ groups of equal cardinality where the last group is possibly of smaller cardinality.



(b) In each group C_i^M the job sizes are rounded up to the largest size occurring in the respective group.



(c) Each group is shifted to the next larger group; the first group is removed.



(d) The first group is removed from the schedule, the remaining rounded jobs from $J_M(T)$ are accommodated by the rearrangement.

Figure 5.2.: This sketch illustrates the linear grouping and rounding technique used in Subsubsection "Definition of Configurations for Medium Jobs".

bounded by

$$P(C_1^M) \leq \frac{Tn'}{2c_1} \leq \frac{Tm}{2\epsilon' c_1} = \frac{Tm}{2\epsilon' \lceil 1/\epsilon'^2 \rceil} \leq \frac{\epsilon' Tm}{2}.$$

Note that $G \leq T$ for each gap G and $p_i > \epsilon' T$ for each medium job $i \in J_M(T)$. Hence at most $\lfloor 1/\epsilon' \rfloor$ medium jobs from $J_M(T)$ can occur in each gap in σ_1. Now a *configuration* is a c_1-tuple $(a_1, \ldots, a_{c_1})$ with $\sum_{i=1}^{c_1} a_i \leq 1/\epsilon'$ and $a_i \in \mathbb{N}_0$ for each $i \in \{1, \ldots, c_1\}$. For a configuration the i-th component denotes the number of items of size q_i. Each configuration naturally corresponds to a choice of rounded items from $J_M(T)$ which can occur together in a gap, i.e. $\sum_{i=1}^{c_1} a_i q_i \leq T$. Note that this definition also includes the configuration in which all entries are zero.

Example. Suppose we have $\epsilon' = 1/3$, which means that we have $c_1 = 3$ rounded sizes. Suppose these are

$$q_1 = \frac{7}{14}T, \quad q_2 = \frac{6}{14}T, \quad q_3 = \frac{5}{14}T.$$

Note that for these sizes, it is impossible for a configuration to have $\sum_{i=1}^{3} a_i = 3$, since this would yield

$$\sum_{i=1}^{3} a_i q_i \geq \frac{15}{14}T > T.$$

In total we obtain the following configurations.

$\sum_{i=1}^{3} a_i = 0$	$\sum_{i=1}^{3} a_i = 1$	$\sum_{i=1}^{3} a_i = 2$
$(0,0,0)$	$(0,0,1)$	$(0,1,1)$
	$(0,1,0)$	$(0,0,2)$
	$(1,0,0)$	$(1,0,1)$
		$(1,1,0)$
		$(2,0,0)$

For the general case, let c_2 denote the number of configurations. Furthermore, for any $n \in \mathbb{N}$, $k \in \mathbb{N}$ let

$$D_n^k := |\{(a_1, \ldots, a_n) | a_i \in \mathbb{N}_0, \sum_{i=1}^{n} a_i \leq k\}| = \binom{n+k-1}{k};$$

the last equality follows from modeling the combinatorial situation as combinations with repetition. In total, we obtain

$$c_2 \leq \sum_{j=0}^{\lfloor 1/\epsilon' \rfloor} D_n^j = \sum_{j=0}^{\lfloor 1/\epsilon' \rfloor} \binom{\lfloor 1/\epsilon' \rfloor + j - 1}{j} = \frac{1}{\lfloor 1/\epsilon' \rfloor} \sum_{j=0}^{\lfloor 1/\epsilon' \rfloor} \frac{(\lfloor 1/\epsilon' \rfloor + j - 1)!}{j!}.$$

However, in the sequel we will use the simpler bound

$$c_2 \leq |\{\kappa \in \{0, \ldots, \lfloor 1/\epsilon' \rfloor\}^{c_1} | \sum_{i=1}^{c_1} \kappa_i \leq \lfloor 1/\epsilon' \rfloor\}|$$
$$\leq |\{\kappa \in \{0, \ldots, \lfloor 1/\epsilon' \rfloor\}^{c_1}| = (\lfloor 1/\epsilon' \rfloor + 1)^{c_1}$$

which states that c_2 is independent from the encoding size of the input. We denote all configurations by $\kappa^{(1)}, \ldots, \kappa^{(c_2)}$. For each such a configuration $\kappa^{(\ell)} = (\kappa_1^{(\ell)}, \ldots, \kappa_{c_1}^{(\ell)})$ with upper index $\ell \in \{1, \ldots, c_2\}$ we denote by

$$s_\ell := \sum_{i=1}^{c_1} \kappa_i^{(\ell)} q_i$$

the *total size* of the ℓ-th configuration. So far, by removing C_1^M from σ, we have defined a partial schedule σ_1 with makespan at most T and a simpler structure in which only a small amount of total processing time is not scheduled. Furthermore, each job which is not included in the schedule is a medium job. We have established Lemma 59 and later use enumeration to find a suitable choice of configurations for a target makespan T.

Lemma 59. *For every feasible makespan T there is a partial schedule σ_1 with makespan at most T and the following properties. Every large job from $J_L(T)$ is scheduled in a gap from $G_L(T)$. Every small job from $J_S(T)$ is scheduled. Almost all medium jobs from $J_M(T)$ are scheduled, i.e. there are only medium jobs from $J_M(T)$ which are not scheduled, and the total processing time of these is at most $\epsilon' Tm/2$. In each gap in $G_L(T)$ there are at most three objects, namely a large job from $J_L(T)$, a configuration and a set of small jobs from $J_S(T)$.*

Discretization of Suitable Large Jobs

We discretize the large jobs in $J_L(T)$ which are packed together in a gap with a non-empty configuration in σ_1 from Lemma 59. The large jobs in $J_L(T)$ which are packed into a gap from $G_L(T)$ with the empty configuration in σ_1 are *not* discretized. The construction described here leaves the small jobs from $J_S(T)$ untouched and modifies only the large jobs and configurations in gaps in $G_L(T)$.

We assume that in σ_1 in each large gap from $G_L(T)$ there is a job from $J_L(T)$; otherwise we introduce an artificial "large" job of size 0 for such a gap. Hence we obtain $|J_L(T)| = |G_L(T)| \leq m$, i.e. there are as many large jobs as large gaps. Now let $c_3 := \lceil 1/\epsilon' \rceil - 1$, and for each $k \in \{1, \ldots, c_3\}$ let

$$I_k(T) := (k\epsilon' T, (k+1)\epsilon' T].$$

Finally for each $k \in \{1, \ldots, c_3\}$, $\ell \in \{1, \ldots, c_2\}$ let

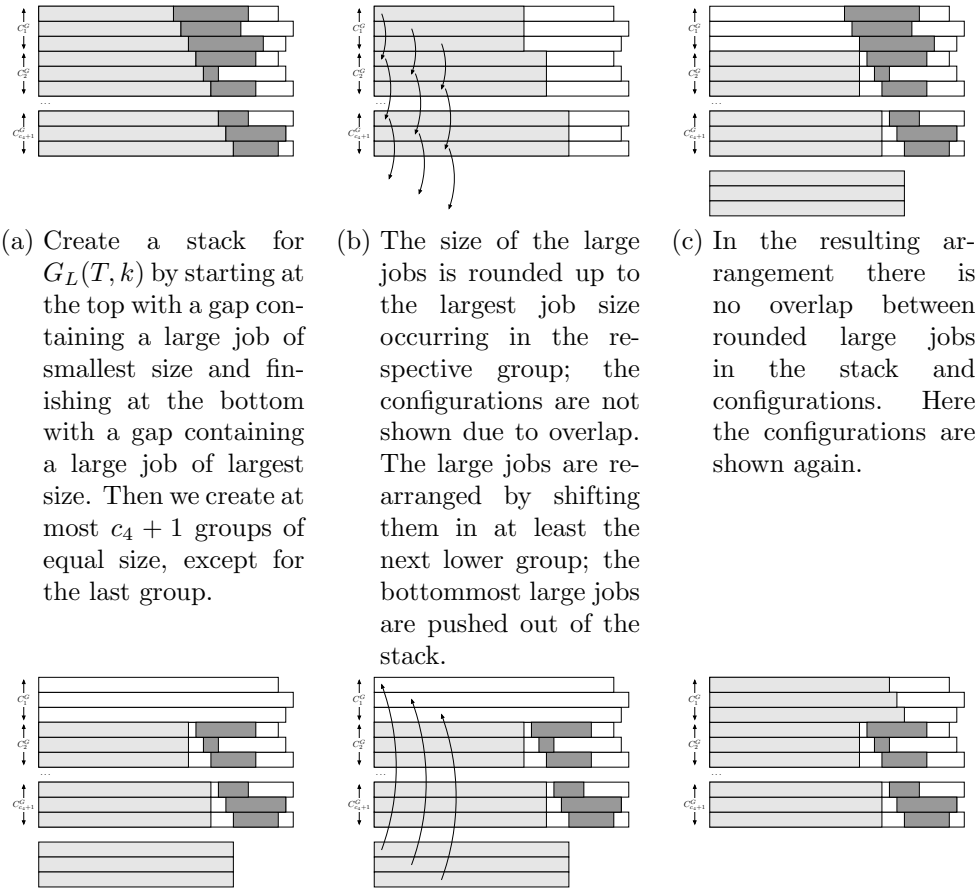
$$J_L(T, k) := \{j \in J_L(T) | p_j \in I_k(T)\}$$

denote the set of large jobs with processing times in the interval $I_k(T)$.

In each gap in $G_L(T)$ there are exactly three objects, namely a large job, a configuration, and a set of small jobs which may be empty however. We define

$$G_L(T, k) := \{G \in G_L(T) | \text{in } \sigma_1 \text{ gap } G \text{ contains a job from } J_L(T, k) \text{ and a non-empty configuration}\}.$$

Now we present a construction to round the large jobs from $J_L(T)$ contained in $G_L(T, k)$ under a small loss of total size of the medium jobs in $J_M(T)$; the approach is illustrated in Figure 5.3. There, light grey areas indicate the large jobs, dark grey areas indicate the configurations, and white areas indicate small jobs or idle time.



(a) Create a stack for $G_L(T,k)$ by starting at the top with a gap containing a large job of smallest size and finishing at the bottom with a gap containing a large job of largest size. Then we create at most $c_4 + 1$ groups of equal size, except for the last group.

(b) The size of the large jobs is rounded up to the largest job size occurring in the respective group; the configurations are not shown due to overlap. The large jobs are rearranged by shifting them in at least the next lower group; the bottommost large jobs are pushed out of the stack.

(c) In the resulting arrangement there is no overlap between rounded large jobs in the stack and configurations. Here the configurations are shown again.

(d) To accomodate the rounded large jobs from below the stack, the configurations in the topmost group C_1^G are removed. The idle time created there is large enough to contain the rounded large jobs from below the stack.

(e) The rounded large jobs from below the stack are shifted in the gaps of the topmost group C_1^G. For ease of exposition, the job sizes there are restored to the non-rounded value.

(f) In the final arrangement all rounded large jobs from $J_L(T,k)$ are packed. The only jobs which are not packed are the configurations from the first group C_1^G.

Figure 5.3.: This sketch illustrates the construction from Subsubsection "Discretization of Suitable Large Jobs"; light grey areas indicate the large jobs from $G_L(T,k)$, dark grey areas indicate the associated configurations and white areas indicate small jobs or idle time.

Conceptually arrange the gaps in $G_L(T,k)$ along with their contents in σ_1 in a vertical stack, starting at the top with a gap containing a job from $J_L(T,k)$ of smallest size to the bottom finishing with a gap containing a job from $J_L(T,k)$ of largest size. Except for the small jobs from $J_S(T)$, each of these gaps contains exactly two objects, namely a job from $J_L(T,k)$ and a non-empty configuration. If two such objects occur together in a large gap we will call them *associated*. Similar as in [30], we apply linear grouping to the stack. More precisely, we set $c_4 := \lceil 1/\epsilon' \rceil$ and aim at creating c_4+1 groups; beginning from the top, we create c_4+1 groups $C_1^G, \ldots, C_{c_4+1}^G$ of size $\lfloor |G_L(T,k)|/c_4 \rfloor$ where the last group is possibly of smaller cardinality. However, if in the stack there are $n' < c_4+1$ gaps, we also create c_4+1 groups. In this case, the first n' groups are of cardinality 1; the remaining c_4+1-n' groups are empty and are not used in the construction. Now in each group C_i^G, each large job is rounded up to the size of the largest large job occurring in C_i^G, namely to

$$q_i'(T,k) := \max\{p_j | j \in J_L(T),\ j \text{ occurs in a gap from } C_i^G\};$$

if there are $n' < c_4+1$ gaps in the stack, the values for the empty groups are set to $-\infty$. Now it remains to show that the rounded large jobs in the stack can be packed together with nearly all of their associated configurations.

To this end, we use an elegant cyclic shifting argument which is sketched in Figure 5.3. Each rounded large job from $J_L(T,k)$ is shifted downwards exactly $\lfloor |G_L(T,k)|/c_4 \rfloor$ gaps in the stack into at least the next larger group, where it can be safely packed together with the configuration packed there. The large jobs in the $\lfloor |G_L(T,k)|/c_4 \rfloor$ gaps at the bottom are pushed out of the stack. Hence, a total number of $\lfloor |G_L(T,k)|/c_4 \rfloor$ large jobs from $J_L(T)$ not packed. We remove the configurations in the group C_1^G and denote the set of non-packed associated configurations by $I(T,k)$. Then, the set $I(T,k)$ is removed from the schedule. Now the uppermost $\lfloor |G_L(T,k)|/c_4 \rfloor$ gaps in the stack are empty. Every non-packed configuration has a total size of at least $\epsilon' T$ and at most $T/2$ since it was packed together with a large job in σ_1. Consequently, we can use

$$P(I(T,k)) \leq \frac{T}{2} \lfloor |G_L(T,k)|/c_4 \rfloor \leq \frac{T|G_L(T,k)|}{2c_4} \tag{1}$$

to bound the total processing time of the non-packed configurations. Concerning the $\lfloor |G_L(T,k)|/c_4 \rfloor$ large jobs from $J_L(T,k)$ which are not packed, we note that the size of each of them is at most $(k+1)\epsilon' T$. The size of each of the empty gaps in the upper parts of the stack is at least $k\epsilon' T + \epsilon' T = (k+1)\epsilon' T$. Consequently, the large jobs which have

been pushed out of the stack can be feasibly put in the uppermost gaps, i.e. the gaps in C_1^G. Just for ease of exposition, the sizes of the large jobs which are now in the gaps in C_1^G are restored to their non-rounded sizes. By dropping the rounding of the large jobs now in C_1^G, a large job is rounded if and only if it is packed together with a non-empty configuration.

Note that, algorithmically, it is not possible to obtain $q'_1(T,k),\ldots,q'_{c_4+1}(T,k)$, the rounded job sizes for a fixed interval index $k \in \{1,\ldots,c_3\}$, directly, since the schedule σ_1 is not available. However, each value $q'_i(T,k)$ is a size of a large job from $J_L(T)$ or $-\infty$, i.e. one of $m+1$ possible values. Hence, the number of possible choices for the values $q'_i(T,k)$ is bounded by $(|J_L(T)|+1)^{c_4+1} \leq (m+1)^{c_4+1}$. This means that the values $q'_i(T,k)$ resulting from the application of the cyclic shifting technique can be found by enumeration within a polynomial runtime bound, since also the number of interval indices $k \in \{1,\ldots,c_3\}$ is bounded by a constant. In total, all values $q'_i(T,k)$ can be enumerated in $(m+1)^{c_3(c_4+1)}$ steps.

Lemma 60. *By applying the rounding and cyclic shifting for all interval indices $k \in \{1,\ldots,c_3\}$, only medium jobs from $J_M(T)$ are lost. The total processing time of these is bounded by $\epsilon' Tm/2$. Furthermore, the number of different sizes for the rounded large jobs is bounded by $c_3(c_4+1)$.*

Proof. By construction only medium jobs from $J_M(T)$ are not packed. Now we use the bound (1), the estimation $\sum_{k=1}^{c_3} |G_L(T,k)| \leq |G_L(T)| \leq m$, and $c_4 = \lceil 1/\epsilon' \rceil$ to obtain the chain of inequalities

$$P(\cup_{k=1}^{c_3} I(T,k) = \sum_{k=1}^{c_3} P(I(T,k)) \leq T/(2c_4) \sum_{k=1}^{c_3} |G_L(T,k)| \leq \frac{T}{2c_4}|G_L(T)| \leq \frac{Tm}{2c_4} \leq \frac{\epsilon' Tm}{2}$$

which yields the desired bound. Since we have c_3 interval indices and for each of these we generate at most c_4+1 rounded sizes for large jobs, we obtain the claim. □

We use σ_2 to denote the resulting partial schedule in which all jobs which are now removed, more precisely the medium jobs in $\cup_{k=1}^{c_3} I(T,k)$, do not occur. Let

$$\mathcal{I} := \{1,\ldots,c_3\} \times \{1,\ldots,c_4+1\} \times \{1,\ldots,c_2\}$$

denote the set of triples for indices for intervals, rounded sizes of large jobs, and config-

urations. For each $(k, i, \ell) \in \mathcal{I}$ let

$$c(k,i,\ell) := |\{G \in G_L(T) | \text{in } \sigma_2 \text{ gap } G \text{ contains a job from } J_L(T,k) \\ \text{of rounded size } q'_i(T,k) \text{ and a non-empty configuration } \kappa^{(\ell)}\}| \leq m$$

denote the number of large jobs from $J_L(T)$ with rounded size $q'_i(T,k)$ which are packed together with the non-empty configuration $\kappa^{(\ell)}$ in σ_2. In total, we obtain the following result.

Lemma 61. *For every feasible makespan T there is a partial schedule σ_2 with makespan at most T and the following properties. Every large job from $J_L(T)$ is scheduled in a gap from $G_L(T)$. Every small job from $J_S(T)$ is scheduled. Almost all medium jobs from $J_M(T)$ are scheduled, i.e. there are only medium jobs from $J_M(T)$ which are not scheduled, and the total processing time of these is at most $\epsilon' Tm/2 + \epsilon' Tm/2 = \epsilon' Tm$. In each large gap from $G_L(T)$ there are exactly three objects, namely a possibly rounded large job from $J_L(T)$, possibly of size 0, a possibly empty configuration and a possibly empty subset of small jobs from $J_S(T)$. The number of sizes for the rounded large jobs is bounded by $c_3(c_4+1)$. For each $(k,i,\ell) \in \mathcal{I}$ there is a nonnegative integer $c(k,i,\ell) \leq m$ which indicates how often a rounded large job of size $q'_i(T,k)$ is packed together with a non-empty configuration $\kappa^{(\ell)}$.*

Clearly, there are at most $m^{c_2 c_3(c_4+1)}$ choices for the values $c(k,i,\ell)$, hence these can be found by enumeration. However, algorithmically we have to deal with the problem that, even if the values $c(k,i,\ell)$ are known, it is difficult to find the assignment of large jobs in $J_L(T)$ and associated configurations exactly as in σ_2.

However, with Lemma 62, we will show that by using a straightforward greedy argument for the small jobs in $J_S(T)$, any feasible assignment of the large jobs in $J_L(T)$ and the associated confgurations to the gaps in $G_L(T)$ can be extended to a partial schedule under a small loss of processing time. This means that obtaining any such assignment is sufficient for our construction. To this end we define a multiset of large jobs and configurations; more precisely let

$$J_{LC}(T) := \{j \in J_L(T) | \text{job } j \text{ is rounded (or not rounded) as in } \sigma_2\} \\ \cup \{\kappa^{(\ell)} | \kappa^{(\ell)} \text{ occurs in a gap in } G_L(T) \text{ in } \sigma_2\}$$

denote the large jobs and configurations as they occur in the large gaps in σ_2. We obtain the following result.

Lemma 62. *Let T be a feasible makespan. Let σ_3 be a partial schedule of makespan at most T which assigns exactly the (possibly rounded) large and medium jobs in $J_{LC}(T)$ to the large gaps in $G_L(T)$ in any feasible way. Then σ_3 can be extended to a partial schedule σ_4 with makespan at most T and the following properties. Every large job from $J_L(T)$ and a subset of almost all medium and small jobs are scheduled in a large gap from $G_L(T)$, i.e. there are only medium and small jobs from $J_M(T) \cup J_S(T)$ which are not scheduled and the total processing time of these is at most $\epsilon' Tm/2 + \epsilon' Tm/2 + \epsilon' Tm = 2\epsilon' Tm$.*

Proof. Let σ_2 denote the schedule from Lemma 61. Let

$$I_S := \{j \in J_S(T) | \text{job } j \text{ is scheduled in a gap from } G_L(T) \text{ in } \sigma_2\}.$$

Remove the small jobs from I_S in σ_2 and do not change the schedule in the gaps which are not large. Let

$$P_{LG} := \sum_{i=1}^{q_L(T)} G_i - P(J_{LC}(T))$$

denote the remaining free processing time in the gaps from $G_L(T)$ after I_S is removed. Clearly we have $P(I_S) \leq P_{LG}$. From the resulting schedule remove all jobs from $J_{LC}(T)$ (i.e. the jobs in the large gaps) and reschedule them again as in σ_3. Clearly, this does not change the total load in $G_L(T)$, i.e. in $G_L(T)$ there is still an amount of P_{LG} of idle time. Since $P(I_S) \leq P_{LG}$, we can distribute the small jobs in I_S to the gaps in $G_L(T)$ in a first fit manner, fractionalizing jobs which cannot be accommodated completely. In this way, at most $|G_L(T)| - 1 \leq m - 1 \leq m$ small jobs are fractionalized. The set of these is called S and is removed from the schedule; the resulting schedule is denoted by σ_4. Since for each $j \in S$ we have $p_j \leq \epsilon' T$ and $|S| \leq m$, we have $P(S) \leq \epsilon' Tm$. Furthermore, except for the gaps in $G_L(T)$, σ_4 is identical to σ_2 and the jobs in $J_{LC}(T)$ are scheduled as in σ_3. □

Assignment of Jobs to Large Gaps via Network Flow

In Lemma 62 we have argued that basically it is not important how the large and medium jobs in $J_{LC}(T)$ are packed into the gaps in $G_L(T)$ once the set $J_{LC}(T)$ is known, even if we do not know the contents of the remaining gaps. In this subsubsection we show how both the selection of suitable configurations and the assignment to the gaps can be done via enumeration of a class of network flow models. Solutions of the network flow model will also decide whether or not a large job is rounded.

Given a makespan T we use a network flow model to find a feasible assignment, if one exists, for $J_L(T)$ and the associated configurations. Suppose that suitable rounded job sizes $q'_i(T,k)$ for $k \in \{1,\dots,c_3\}$ and $i \in \{1,\dots,c_4+1\}$ are given. Furthermore, the associated configurations are given implicitly by the values $c(k,i,\ell) \leq m$ for each $(k,i,\ell) \in \mathcal{I}$.

We define a directed acyclic graph $N(T) = (V(T), E(T))$ where $V(T)$ consists of five layers; the construction is sketched in Figure 5.4. More precisely, the *node set* $V(T)$ of $N(T)$ is defined as follows.

1. In the first layer there is only s, the *source* node.
2. In the second layer there are the at most m large jobs from $J_L(T)$ which we call *job* nodes.
3. In the third layer there is a set of nodes to govern the assignment of large jobs and configurations, namely exactly the set $\mathcal{I}$ which will be termed as *interval-size-configuration* nodes.
4. In the fourth layer there are the at most m large gaps from $G_L(T)$ which we call *gap* nodes.
5. In the fifth layer there is only t, the *terminal* node.

Likewise, the *arc set* $E(T)$ of $N(T)$ is constructed in order to encode the possibilities of arranging large jobs together with configurations in large gaps. More precisely, $E(T)$ is defined as follows.

1. The source s is connected to each large job node, i.e. $(s,j) \in E(T)$ for each $j \in J_L(T)$; these arcs connect the first and the second layer.
2. Each job node is connected to an interval-size-configuration node if it is contained in the interval and can be possibly rounded to the size, i.e.

$$(j,(k,1,\ell)) \in E(T) :\Leftrightarrow p_j \in (0, q'_1(T,k)]$$

and

$$(j,(k,i,\ell)) \in E(T) :\Leftrightarrow p_j \in (q'_{i-1}(T,k), q'_i(T,k)]$$

for $i \in \{2,\dots,c_4+1\}$ for each $j \in J_L(T)$ and $(k,i,\ell) \in \mathcal{I}$; these arcs connect the second to the third layer.

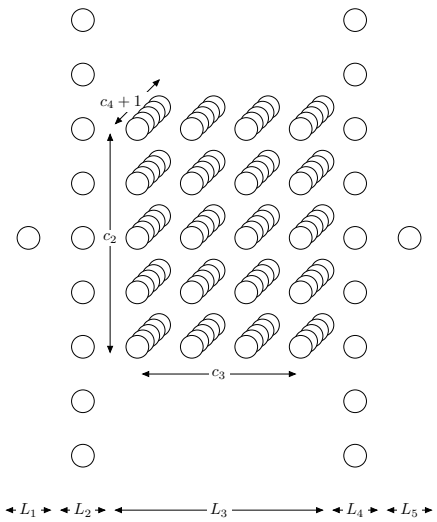


Figure 5.4.: Sketch of the network used for assignment of large jobs and configurations; edges are not shown. Layers 1 (L_1) to 5 (L_5) are arranged from left to right. Arrows labelling layer 3 indicate the number of nodes in the corresponding dimension. In total, there are $|\mathcal{I}|$ nodes in layer 3.

3. Each interval-size-configuration node is connected to a gap node if a job of the rounded size can be packed together with the configuration into the gap, i.e.

$$((k, i, \ell), G) \in E(T) :\Leftrightarrow q'_i(T, k) + s_\ell \leq G$$

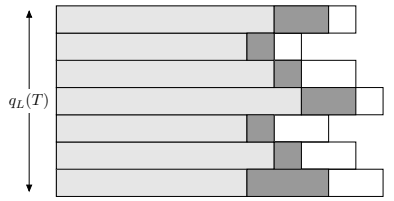
and $q'_i(T, k) \neq -\infty$, for each $(k, i, \ell) \in \mathcal{I}$ and $G \in G_L(T)$; these arcs connect the third to the fourth layer. Routing of flow along such an edge indicates that the corresponding large job is *rounded* and packed together with a non-empty configuration.

4. Each gap node is connected to the terminal node, i.e. $(G, k) \in E(T)$ for each $G \in G_L(T)$; these arcs connect the fourth to the fifth layer.

5. Each job node is connected to each gap node it fits into, more precisely

$$(j, G) \in E(T) :\Leftrightarrow p_j \leq G$$

for each $j \in J_L(T)$, $G \in G_L(T)$; these arcs connect the second to the fourth layer. Routing of flow along such an edge indicates that the corresponding large job is *not rounded* and packed together with the empty configuration.



(a) In the partial schedule σ_3, the large gaps in $G_L(T)$ are filled with large jobs from $J_L(T)$ and configurations as indicated by the choice of values $c(k, \ell)$.

(b) For the assignment of jobs from I_2, we consider the remaining space. The set of smaller "large" gaps is denoted by $\tilde{G}_L(T)$.

Figure 5.5.: This sketch illustrates the large gaps in the schedule σ_3. Light grey areas indicate large jobs, dark grey ares indicate configurations, and white areas indicate idle time.

The arcs in $E(T)$ are endowed with a lower capacity 0 and an upper capacity of 1. Finally for each $(k, i, \ell) \in \mathcal{I}$ we require the flow in the interval-size-configuration node (k, i, ℓ) to be exactly $c(k, i, \ell)$ which can be done by expanding a node to two nodes connected by an artificial edge with suitable flow constraints.

Note that the encoding size of $N(T)$ is polynomially bounded in the encoding size of the instance. Furthermore $N(T)$ has an optimal s-t-flow of value $|J_L(T)|$ if and only if the jobs of $J_L(T)$ (possibly rounded to the values $q'_i(T, k)$) together with the selected configurations, implicitly given by the values $c(k, i, \ell)$, can be packed into the gaps in $G_L(T)$; a corresponding packing is then given in a natural way via the network flow.

However, the values $c(k, i, \ell)$ for each $(k, i, \ell) \in \mathcal{I}$ have to be enumerated. We have $|G_L(T)| \leq m$ and hence at most m rounded large jobs from $J_L(T)$ of a certain size can be packed together with a certain configuration; consequently, as mentioned in Lemma 61, $c(k, i, \ell) \leq m$ holds for each $(k, i, \ell) \in \mathcal{I}$. Furthermore there is only a constant number of at most $|\mathcal{I}| = c_2 c_3 (c_4+1)$ nodes in the third layer. Since each of these nodes gets assigned a capacity value $c(k, i, \ell) \in \{0, \ldots, m\}$, the quantity $(m+1)^{c_2 c_3 (c_4+1)}$ is an upper bound for the number of possible assignments of flow restrictions on configuration-interval nodes.

Packing of Medium and Small Jobs

Let $I := \{k+1, \ldots, n\}$. We enumerate over all possible choices for the rounded job sizes $q'_i(T, k)$ for each $i \in \{1, \ldots, c_4 + 1\}$ and $k \in \{1, \ldots, c_3\}$; we also enumerate over all possible choices of values $c(k, i, \ell)$ for each $(k, i, \ell) \in \mathcal{I}$. Next we describe how to find a suitable schedule, if one exists, given one such choice of values. First we can

use the network flow model from Subsubsection "Assignment of Jobs to Large Gaps via Network Flow" to find a feasible assignment, if one exists, of the large jobs and associated configurations to the gaps in $G_L(T)$.

Next we discuss Step 2.2.1.2 of the algorithm in Figure 5.1. Let I_1 denote the set of jobs assigned in this way and denote by σ_3 the corresponding partial schedule; let $I_2 := I \setminus I_1$. The assignment in σ_3 is done without unnecessary idle time in the large gaps and is fixed in the candidate solution, i.e. we aim at extending σ_3. Consequently, the large gaps in $G_L(T)$ become smaller since σ_3 assigns some jobs there, as sketched in Figure 5.5. More precisely, we denote by S_i the set of jobs scheduled in the large gap G_i for $i \in \{1, \dots, q_L(T)\}$ and introduce new gaps $\tilde{G}_i$ of sizes

$$\tilde{G}_i := G_i - \sum_{j \in S_i} p_j$$

for each $i \in \{1, \dots, q_L(T)\}$. Furthermore we use $\tilde{G}_L(T) := \{\tilde{G}_1, \dots, \tilde{G}_{q_L(T)}\}$ to denote the set of new gaps and let $G'(T) := \tilde{G}_L(T) \cup G_S(T)$. Now G' is to be algorithmically filled with jobs from I_2. If T is feasible, by Lemma 62 there is a subset $I_3 \subseteq I_2$ such that I_3 can be scheduled in $G'(T)$ and

$$P(I_1) + P(I_3) \geq P(I) - 2\epsilon' Tm$$

holds; let I_3 be chosen as such. We use a PTAS for MSSP to select $I_4 \subseteq I_2$ such that $P(I_4) \geq (1 - \epsilon')P(I_3)$. In total we obtain

$$\begin{aligned} P(I_1) + P(I_4) &\geq P(I_1) + (1 - \epsilon')P(I_3) \\ &\geq (1 - \epsilon')(P(I_1) + P(I_3)) \geq (1 - \epsilon')(P(I) - 2\epsilon' Tm) \end{aligned}$$

unless T is infeasible and can safely be discarded. In total, for the optimal makespan $T = C^*_{\max}$ and a suitable choice of values $c(k, i, \ell)$ we can schedule a total load of at least $(1 - \epsilon')(P(I) - 2\epsilon' Tm)$ in $[0, T)$. Hence after T there remains a total processing time of at most

$$P(I) - (1 - \epsilon')(P(I) - 2\epsilon' Tm) \leq 2\epsilon' Tm + \epsilon' P(I) \leq 2\epsilon' Tm + \epsilon' Tm = 3\epsilon' Tm$$

to schedule and the size of all of these jobs is bounded by $T/2$. We can use any list scheduling algorithm to execute this small load in the interval $[T, \infty)$; the following analysis is sketched in Figure 5.6. Let T' denote the last step in $[T, \infty)$ where there is

no idle machine and let T'' denote the last time step in $[T', \infty)$ where there is a busy machine. Now we use the well-known Graham bounds. Here we obtain

$$|[T', T'')| \leq T/2 \leq C^*_{\max}/2$$

for the last part of the schedule and

$$|[T, T')| \leq \frac{3\epsilon' Tm}{m} \leq 3\epsilon' C^*_{\max}$$

for the middle part of the schedule, hence the makespan of our algorithmically generated schedule can be bounded by

$$\begin{aligned} |[0, T)| + |[T, T')| + |[T', T'')| &\leq C^*_{\max} + 3\epsilon' C^*_{\max} + \frac{1}{2} C^*_{\max} \\ &= (3/2 + 3\epsilon') C^*_{\max} = (3/2 + \epsilon) C^*_{\max}. \end{aligned}$$

By carrying out the entire construction from Subsubsections "Job Classification and Generation of Gaps"–"Packing of Medium and Small Jobs", we have established the first part of Theorem 57. Next we show how we can obtain a ratio of 3/2 via a modification of the list scheduling approach discussed above.

To this end, we use the algorithm in Figure 5.1 with $\epsilon := 3/24$, which results in $\epsilon' = 1/24$. Furthermore we modify Step 3 of the Algorithm in Figure 5.1 as follows. When reaching Step 3, let I' denote the set of jobs which are not scheduled; as discussed above, we have $T \leq C^*_{\max}$ and $P(I') \leq 3\epsilon' Tm$. Furthermore, we have $p_j \leq T/2$ for each $j \in I'$. Next we partition I' in two sets of larger and smaller jobs by setting

$$\begin{aligned} I'_L(T) &:= \{j \in I' | p_j > T/4\}, \\ I'_S(T) &:= \{j \in I' | p_j \leq T/4\}. \end{aligned}$$

Let $n'' := |I'_L|$; since $P(I') \leq 3\epsilon' Tm$, we have $n''T/4 \leq 3\epsilon' Tm$; suitable rearrangement and using $\epsilon' = 1/24$ yields

$$n'' \leq 12\epsilon' m = 12m/24 = m/2.$$

Note that, since n'' is integral, we also have $n'' \leq \lfloor m/2 \rfloor$. Now, since in the time interval $[T, \infty)$ all machines are available, we use the first $n'' \leq \lfloor m/2 \rfloor$ machines to schedule the jobs in I'_L, where each job is scheduled on a machine of its own starting at time T.

Similar as before, we use list scheduling to schedule the jobs in I'_S on the last $m - n''$ machines.

Next we distinguish two cases. *Case 1:* Scheduling of the jobs in I'_S does not increase the makespan of the generated schedule. In this case, the makespan of the algorithmically generated schedule is bounded by

$$T + T/2 \leq \frac{3}{2}T \leq \frac{3}{2}C^*_{\max}.$$

Case 2: Scheduling of the jobs in I'_S *does* increase the makespan of the generated schedule. Let T' denote the last step in $[T, \infty)$ where there is no idle machine and let T'' denote the last time step in $[T, \infty)$ where there is a busy machine. Using the Graham bounds, we obtain $[T', T'') \leq T/4$ for the last part of the schedule and

$$|[T, T')| \leq \frac{3\epsilon' Tm}{m - n''} \leq \frac{3\epsilon' Tm}{m - m/2} = \frac{Tm/8}{m/2} = \frac{T}{4}$$

for the middle part of the schedule. In total, using these bounds, the makespan of the generated schedule can be bounded by

$$|[0, T)| + |[T, T')| + |[T', T'')| \leq T + \frac{T}{4} + \frac{T}{4} = \frac{3}{2}T \leq \frac{3}{2}C^*_{\max};$$

this estimation establishes the second part of Theorem 57. In total, we have proven our first main result.

Comment. In the algorithm in Figure 5.1, we can also use a greedy 2-approximation algorithm for MSSP [28] instead of the rather sophisticated PTASes [18, 24, 64, 72, 107]. Consequently, we have modify the algorithm in Figure 5.1 as follows. In Step 1, we set $\epsilon' := \epsilon/2$ instead of $\epsilon/3$ and in Step 2.2.1.2, we compare the amount of non-scheduled processing time to $(1/2+2\epsilon')Tm$ instead of $3\epsilon' Tm$. If we carry out the same construction as before with this modification, for the list scheduling step we obtain

$$|[T, T')| \leq \frac{(1/2 + 2\epsilon')Tm}{m} \leq (\frac{1}{2} + 2\epsilon')C^*_{\max}.$$

In total, the length of the generated schedule then can be bounded by

$$|[0, T)| + |[T, T')| + |[T', T'')| \ \leq \ C^*_{\max} + (\frac{1}{2} + 2\epsilon')C^*_{\max} + \frac{1}{2}C^*_{\max} \ = \ (2 + \epsilon)C^*_{\max},$$

which means that this approach yields the same approximation ratio as using a PTAS

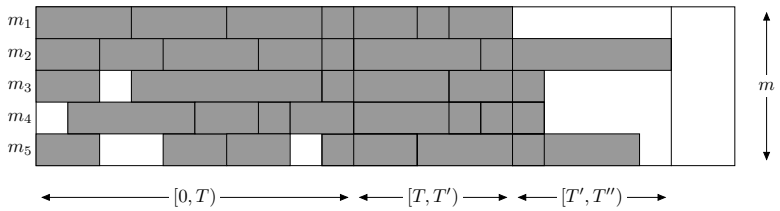


Figure 5.6.: Illustration of the list scheduling from Subsububsection "Packing of Medium and Small Jobs" which finally arranges the hitherto non-scheduled jobs. The jobs are indicated by dark grey areas while light grey areas indicate the periods of non-availability; white areas indicate idle time. The jobs in the interval $[0, T)$ are scheduled via the technique described in Subsubsections "Job Classification and Generation of Gaps"–"Assignment of Jobs to Large Gaps via Network Flow" and a PTAS for MSSP; the jobs in $[T, T'')$ are assigned via list scheduling, hence in $[T, T')$ there is no idle time.

for $\mathrm{P}||C_{\max}$ [60] to schedule all non-fixed jobs after the last fixed job.

5.2.2. Hardness Results

Here we present the hardness result from [130, 131, 132] for the sake of completeness. Lemma 64 constitutes the second part of Theorem 57. The proof is based on the NP-completeness of a certain problem which we define first; the problem is then used in the proofs of Lemma 64 and Lemma 65.

Definition 1. *The following problem is called Modified-3-Partition.*

- Given: *Lists A, B which contain n respectively $2n$ elements of sizes $a_i \in \mathbb{N}$ for each $i \in \{1, \ldots, n\}$, $b_i \in \mathbb{N}$ for each $i \in \{1, \ldots, 2n\}$ and $L \in \mathbb{N}$ such that*

$$\sum_{i=1}^{n} a_i + \sum_{i=1}^{2n} b_i = nL$$

  holds.

- Question: *Is there a $\pi \in S_{2n}$ such that $a_i + b_{\pi(2i-1)} + b_{\pi(2i)} = L$ holds for each $i \in \{1, \ldots, n\}$?*

Next we prove the following result.

Lemma 63. *Modified-3-Partition is strongly* NP*-complete.*

Proof. We prove the claim via reduction from the following problem Numerical Matching with Target Sums, or NMTS for short, which is known to be strongly NP-complete [47].

- *Given:* Disjoint sets X and Y with $|X| = |Y| = m \in \mathbb{N}$, $X = \{x_1, \ldots, x_m\}$, $Y = \{y_1, \ldots, y_m\}$ and a target vector $(B_1, \ldots, B_m) \in \mathbb{N}^m$.
- *Question:* Is there a partition of $X \cup Y$ into m disjoint sets $A_1, \ldots, A_m$ in such a way that for each $i \in \{1, \ldots, m\}$ the set A_i contains exactly one element from X and exactly one element from Y and $\sum_{z \in A_i} z = B_i$ holds?

Let I be an instance of NMTS; we set $d := \max\{B_i | i \in \{1, \ldots, m\}\}$. Furthermore we set $a_i := d+1-B_i$ for each $i \in \{1, \ldots, m\}$ and $A := \{a_1, \ldots, a_m\}$. We define $b_i := d+1-x_i$ for each $i \in \{1, \ldots, m\}$ and call $b_1, \ldots, d_m$ the X-elements; we set $b_{m+i} := y_i$ for each $i \in \{1, \ldots, m\}$ and call $b_{m+1}, \ldots, b_{2m}$ the Y-elements. Finally we set $B := \{b_1, \ldots, b_{2n}\}$ and $L := 2(d+1)$. In total this construction defines an instance I' of Modified-3-Partition. Note that I' can be generated from I in a running time that is polynomially bounded in the encoding length of I. Next we show by mutual implication that I' is a yes-instance of Modified-3-Partition if and only if I' is a yes-instance of NMTS.

If I' is a yes-instance of Modified-3-Partition, there is the desired permutation $\pi \in S_{2m}$. Let π be chosen as such; let $i \in \{1, \ldots, m\}$. Then we have

$$a_i + b_{\pi(2i-1)} + b_{\pi(2i)} = L = 2(d+1).$$

Aiming at a contradiction we suppose that $\pi(2i-1), \pi(2i) \in \{1, \ldots, m\}$ holds, which means that both $b_{\pi(2i-1)}$ and $b_{\pi(2i)}$ are X-elements. Then we obtain

$$b_{\pi(2i-1)} + b_{\pi(2i)} > d + 1 + d + 1 = 2(d+1) = L,$$

which is a contradiction. Hence, for each $i \in \{1, \ldots, m\}$, at most one out of $b_{\pi(2i-1)}$ and $b_{\pi(2i)}$ is an X-element. As a consequence of the pigeonhole principle, for each $i \in \{1, \ldots, m\}$, either $b_{\pi(2i-1)}$ is an X-element and $b_{\pi(2i)}$ is a Y-element or vice versa; without loss of generality, for each $i \in \{1, \ldots, m\}$ the element $b_{\pi(2i-1)}$ is an X-element while $b_{\pi(2i)}$ is a Y-element. For each $i \in \{1, \ldots, m\}$ we set

$$A_i := \{x_{\pi(2i-1)}, y_{\pi(2i)-m}\}.$$

Since π is bijective, we have $X \cup Y = \dot{\bigcup}_{i=1}^{m} A_i$; furthermore for each $i \in \{1, \ldots, m\}$ the set A_i contains exactly one element of X and exactly one element of Y. Furthermore

for each $i \in \{1, \ldots, m\}$ we have

$$\begin{aligned}\sum\nolimits_{z \in A_i} z &= x_{\pi(2i-1)} + y_{\pi(2i)} = b_{\pi(2i-1)} - (d+1) + b_{\pi(2i)} \\ &= 2(d+1) - a_i - (d+1) = (d+1) - a_i = (d+1) - (d+1) + B_i = B_i,\end{aligned}$$

which shows that I is a yes-instance of NMTS.

If I is a yes-instance of NMTS, there is the desired disjoint partition $X \cup Y = \dot{\bigcup}_{i=1}^{m} A_i$. Let the sets $A_1, \ldots, A_m$ be chosen as such; then for each $i \in \{1, \ldots, m\}$, the set A_i contains exactly one element of X and exactly one element from Y and we have $\sum_{z \in A_i} = B_i$. For each $i \in \{1, \ldots, m\}$ let i_X be the index of the element of X in A_i and let i_Y be the index of the element of Y in A_i. Furthermore let $\pi \in S_{2m}$ defined by

$$\pi(2i-1) := i_X \text{ and } \pi(2i) := i_Y + m \text{ for each } i \in \{1, \ldots, m\}.$$

Let $i \in \{1, \ldots, m\}$; then we have

$$\begin{aligned}a_i + b_{\pi(2i-1)} + b_{\pi(2i)} &= a_i + b_{i_X} + b_{i_Y+m} = a_i + d + 1 + x_{i_X} + y_{i_Y} \\ &= a_i + d + 1 + B_i = d + 1 - B_i + d + 1 + B_i = 2(d+1) = L\end{aligned}$$

which shows that I' is a yes-instance of Modified-3-Partition. In total, the claim is proved. □

Lemma 64. *Scheduling with fixed jobs for m part of the input does not admit a polynomial time approximation algorithm with absolute approximation ratio $3/2 - \epsilon$, unless $\mathsf{P} = \mathsf{NP}$, for any $\epsilon \in (0, 1/2]$.*

Proof. We aim at a contradiction and suppose there is a polynomial-time approximation algorithm A for our scheduling problem with approximation ratio $3/2 - \epsilon$. We use a reduction from Modified-3-Partition which is strongly NP-complete.

Given an instance I of Modified-3-Partition we define an instance I' of scheduling with fixed jobs as follows. We choose $K \in \mathbb{N}$ such that $K > \max\{L, (1/2 - \epsilon)L/(2\epsilon)\}$; furthermore we use n machines and define fixed jobs $p_j := a_j$ for each $j \in \{1, \ldots, n\}$ and fix these via $(j, 2K + L - a_j)$ for each $j \in \{1, \ldots, n\}$. Finally we introduce non-fixed jobs by setting $p_{n+j} := K + b_j$ for each $j \in \{1, \ldots, 2n\}$. Note that I' can be generated from I in running time polynomial in the encoding length of I.

Note that for a yes-instance I of Modified-3-Partition, we can execute the non-fixed jobs on machines $1, \ldots, n$ according to the existing permutation π; this yields a makespan

of $C^*_{\max} = 2K + L$. Conversely, in a schedule with makespan $2K + L$ the non-fixed jobs run on machines $1, \ldots, n$. Note that the processing time of each non-fixed job is larger than K; consequently, we have $3K > 2K + L$, hence it is impossible that more than 2 non-fixed jobs run on the same machine in the interval $[0, 2K + L)$. This means that on each machine, exactly 2 non-fixed jobs are executed, which indicates the desired permutation π. In total, I' has an optimal makespan of $C^*_{\max} = 2K + L$ if and only if I is a yes-instance of Modified-3-Partition.

Now let I be a no-instance of Modified-3-Partition. Then, in any schedule for I', there must be a non-fixed job which is scheduled after $2K + L$. In total, we obtain a job with completion time at least $3K + L$; hence the makespan of any schedule for I' must be at least

$$3K + L.$$

Next we show that we can use the algorithm A as an exact algorithm for the above problem as follows. For each instance I of Modified-3-Partition we generate an instance of our scheduling problem as described above and apply the algorithm A to the instance I'. If the makespan of the generated schedule for I' is smaller than $3K + L$, we decide that I is a yes-instance of Modified-3-Partition.

Let I be a yes-instance of Modified-3-Partition. Note that $K > (1/2 - \epsilon)L/(2\epsilon)$ can be rearranged to $K2\epsilon > (1/2 - \epsilon)L$. Using this inequality we obtain

$$(\frac{3}{2} - \epsilon)(2K + L) = 3K + (\frac{3}{2} - \epsilon)L - K2\epsilon < 3K + (\frac{3}{2} - \epsilon)L - (\frac{1}{2} - \epsilon)L = 3K + L.$$

Now we use this inequality to argue that the algorithm A generates for I' a solution with value

$$C_{\max} \leq (3/2 - \epsilon)C^*_{\max} = (3/2 - \epsilon)(2K + L) < 3K + L,$$

where in the last step we used the estimation from above. For a no-instance I of Modified-3-Partition, the algorithm A generates for I' a schedule with makespan at least $3K + L$, which is a lower bound for the optimal makespan of I'.

In total, we can algorithmically decide whether any instance I of Modified-3-Partition is a yes-instance or a no-instance within a polynomial runtime bound, which is impossible unless $\mathsf{P} = \mathsf{NP}$ holds. □

5.3. Scheduling with Non-Availability

Here we describe how our approach can be applied to scheduling with non-availability where a constant percentage of the machines is permanently available; the idea is basically the same as for scheduling with fixed jobs, but results in a construction which is slightly more technical in nature. The main reason for this is that, in terms of complexity, scheduling with fixed jobs and non-availability behave in a slightly different way, as we shall see in the sequel.

5.3.1. Algorithms

Now we present the approximation algorithm for scheduling with non-availability where the ratio of permanently available machines is constant. Similar to [63], we use $\lambda \in \{1, \ldots, m-1\}$ to denote the number of machines which are permitted to be temporarily unavailable. Since the machines are identical, we assume that the *first* $m - \lambda$ machines are permanently available; in total, $\rho = (m - \lambda)/m = 1 - \lambda/m$ is the percentage of permanently available machines. As for scheduling with fixed jobs, we may assume that $m \leq n$ holds. Next we describe the first algorithm mentioned in Theorem 58 by using the ideas from Subsection 5.2.1.

Let $I := \{1, \ldots, n\}$; for scheduling with non-availability, the total processing time of the instance is bounded by $P(I) \leq np_{\max}$. This yields an upper bound for the optimal makespan since all jobs can be scheduled on the permanently available machine in the time interval $[0, P(I))$. Similar as before we perform binary search for the makespan in $[0, P(I))$, which yields a suitable makespan in $O(\log(np_{\max}))$ steps. The gap classification for a target makespan T is done as in Subsubsection "Job Classification and Generation of Gaps", yielding $G_L(T)$ and $G_S(T)$; likewise, the partition into large, medium and small gaps is done as in Subsubsection "Job Classification and Generation of Gaps". We proceed as before by defining configurations for medium jobs as in Subsubsection "Definition of Configurations for Medium Jobs"; the rounding results in a loss of processing time of at most $\epsilon' Tm/2$, but still all large jobs are scheduled. The discretization of large jobs is carried out as in Subsubsection "Discretization of Suitable Large Jobs" which again results in an additional loss of total load $\epsilon' Tm/2$ by using the enumeration of network flow models as in Subsubsection "Assignment of Jobs to Large Gaps via Network Flow"; by guessing the assignment of large jobs to large gaps we lose again an amount of $\epsilon' Tm$ of processing time. In the innermost loop of our algorithm, we pack the remaining small and medium jobs using a PTAS for MSSP from [18, 72]. Similar as in

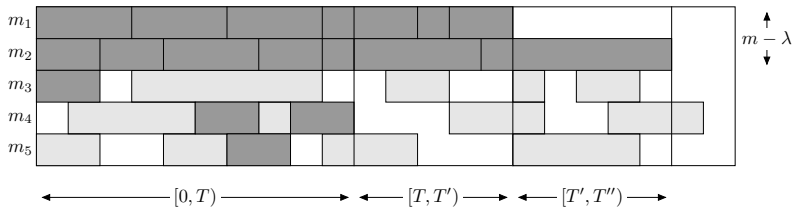


Figure 5.7.: Illustration of the list scheduling from Subsection 5.3.1 for the respective list scheduling which finally arranges the hitherto non-scheduled jobs. The jobs are indicated by dark grey areas while light grey areas indicate the periods of non-availability; white areas indicate idle time. The jobs in the interval $[0, T)$ are scheduled via the technique described in Subsubsections "Job Classification and Generation of Gaps"–"Assignment of Jobs to Large Gaps via Network Flow" and a PTAS for MSSP; the jobs in $[T, T'')$ are assigned via list scheduling, hence in $[T, T')$ there is no idle time on the first $m - \lambda$ machines.

the algorithm for scheduling with fixed jobs, we set $\epsilon' := \epsilon(1 - \lambda/m)/3 = \epsilon\rho/3$.

In total, for the optimal target makespan $T = C^*_{\max}$ a total amount of processing time of at least $(1-\epsilon')(P(I)-2\epsilon' Tm)$ can be scheduled in the interval $[0, T)$; consequently, we have only medium and small jobs with total processing time of at most $2\epsilon' Tm+\epsilon' P(I)$ to schedule. We use list scheduling to pack the remaining jobs in the time interval $[T, \infty)$ on the first $m - \lambda$ machines which are free by assumption; let $M := \{1, \ldots, m - \lambda\}$ denote the set of these. The analysis is illustrated in Figure 5.7.

Similar as in Subsubsection "Packing of Medium and Small Jobs", let T' denote the last step in $[T, \infty)$ where there is no idle machine in M and let T'' denote the last time step in $[T', \infty)$ where there is a busy machine in M. Again we use the Graham bounds and obtain $|[T', T'')| \leq T/2 \leq C^*_{\max}/2$ and

$$|[T, T')| \leq \frac{2\epsilon' Tm + \epsilon' P(I)}{m - \lambda} \leq \frac{2\epsilon' Tm + \epsilon' m C^*_{\max}}{m - \lambda} \leq \frac{2\epsilon' C^*_{\max} + \epsilon' C^*_{\max}}{1 - \lambda/m} = \frac{3\epsilon'}{1 - \lambda/m} C^*_{\max},$$

Hence the makespan of the generated schedule can be bounded by

$$|[0, T)| + |[T, T')| + |[T', T'')| \leq C^*_{\max} + \frac{3}{1 - \lambda/m}\epsilon' C^*_{\max} + \frac{1}{2} C^*_{\max} = (3/2 + \epsilon) C^*_{\max}.$$

In total, we have presented the first algorithm mentioned in Theorem 58. Next we show

how we can obtain a ratio of 3/2 via a the same modification of the list scheduling approach as discussed in Subsubsection "Packing of Medium and Small Jobs". To this end, we use the algorithm discussed above with $\epsilon := 3/24$, which results in $\epsilon' = \rho/24$. When reaching Step 3 of our algorithm, let again denote I' the set of jobs which are not scheduled; as discussed above, we have $T \leq C^*_{\max}$ and $P(I') \leq 3\epsilon' Tm$. Furthermore, we have $p_j \leq T/2$ for each $j \in I'$. As in Subsubsection "Packing of Medium and Small Jobs", we partition I' by defining

$$\begin{aligned} I'_L(T) &:= \{j \in I' | p_j > T/4\}, \\ I'_S(T) &:= \{j \in I' | p_j \leq T/4\}. \end{aligned}$$

Again let $n'' := |I'_L|$; since $P(I') \leq 3\epsilon' Tm$, we have $n''T/4 \leq 3\epsilon' Tm$; suitable rearrangement and using $\epsilon' = \rho/24$ yields

$$n'' \leq 12\epsilon' m = 12\rho m/24 = \rho m/2.$$

Again, since n'' is integral, we have $n'' \leq \lfloor \rho m/2 \rfloor$. Similar as in Subsubsection "Packing of Medium and Small Jobs", in the time interval $[T, \infty)$ the first $m - \lambda = \rho m$ machines are available. We use the first $n'' \leq \lfloor \rho m/2 \rfloor$ machines to schedule the jobs in I'_L, where each job is scheduled on a machine of its own starting at time T. Again we use list scheduling to schedule the jobs in I'_S on the next $m - \lambda - n''$ machines.

Next we distinguish two cases. *Case 1:* Scheduling of the jobs in I'_S does not increase the makespan of the generated schedule. In this case, the makespan of the algorithmically generated schedule is bounded by

$$T + T/2 \leq \frac{3}{2}T \leq \frac{3}{2}C^*_{\max}.$$

Case 2: Scheduling of the jobs in I'_S *does* increase the nmakespan of the generated schedule. Let T' denote the last step in $[T, \infty)$ where there is no idle machine and let T'' denote the last time step in $[T, \infty)$ were there is a busy machine. Using the Graham bounds as before, we obtain $[T', T'') \leq T/4$ for the last part of the schedule and

$$|[T, T')| \leq \frac{3\epsilon' Tm}{m - \lambda - \rho m/2} = \frac{3\epsilon' Tm}{\rho m - \rho m/2} = \frac{3\epsilon' Tm}{\rho m/2} = \frac{6\epsilon' T}{\rho} = \frac{6\epsilon' \rho T}{24\rho} = \frac{T}{4}$$

for the middle part of the schedule. As before, using these bounds, the makespan of the

generated schedule is bounded by

$$|[0,T)| + |[T,T'')| + |[T',T'')| \leq T + \frac{T}{4} + \frac{T}{4} = \frac{3}{2}T \leq \frac{3}{2}C^*_{\max};$$

this estimation establishes the second algorithm mentioned in Theorem 58. In total, we have shown our second main result.

Comment. In our algorithm with ratio $3/2 + \epsilon$, we can also use a greedy 2-approximation algorithm for MSSP [28] instead on a PTAS; we have to modify our algorithm as follows. In Step 1, we set $\epsilon' := \epsilon(1 - \lambda/m)/2 = \epsilon\rho/2$ instead of $\epsilon(1 - \lambda)/3$ and in Step 2.2.1.2, we compare the amount of non-scheduled processing time to $(1/2+2\epsilon')Tm$ instead of $3\epsilon'Tm$. If we carry out the same construction as before with this modification, for the list scheduling step we obtain

$$\begin{aligned}|[T,T')| \leq \frac{(1/2+2\epsilon')Tm}{m-\lambda} &= \frac{(1/2+2\epsilon')Tm}{\rho m} \\ &= \frac{(1/2+2\epsilon')T}{\rho} = \frac{(1/2+\epsilon/\rho)T}{\rho} = (\frac{1}{2\rho}+\epsilon)T \leq (\frac{1}{2\rho}+\epsilon)C^*_{\max}\end{aligned}$$

for the middle part of the schedule. In total, the length of the generated schedule then can be bounded by

$$|[0,T)| + |[T,T')| + |[T',T'')| \leq C^*_{\max} + (\frac{1}{2\rho}+\epsilon)C^*_{\max} + \frac{1}{2}C^*_{\max} = (\frac{3}{2}+\frac{1}{2\rho}+\epsilon)C^*_{\max}$$

which means that this approach yields an approximation ratio which depends on ρ, the percentage of permanently available machines.

5.3.2. Hardness Results

The general problem of scheduling with non-availability without any further restriction does not admit a constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$ holds; this follows from the fact that scheduling with non-availability for m constant is also inapproximable unless $\mathsf{P} = \mathsf{NP}$, as shown in detail in Theorem 49, or, alternatively, from the fact that scheduling parallel jobs on parallel machines with non-availability is inapproximable unless $\mathsf{P} = \mathsf{NP}$ [39].

Theorem 65. *Scheduling with non-availability for m part of the input does not admit a polynomial time algorithm with a constant approximation ratio unless* $\mathsf{P} = \mathsf{NP}$.

Earlier, Lee [109] only pointed out that LPT performs arbitrarily badly. In either

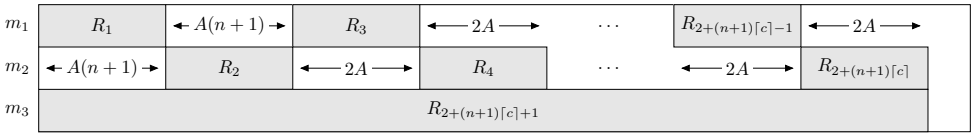
case the inapproximability is due to the permission of time steps where no machine is available. Since the periods of non-availability do not contribute to the makespan, scheduling with non-availability admits a gap-creating reduction which separates the objective values of optimal solutions and suboptimal solutions of yes-instances. However, the restriction to instances where for each time step there is an available machine is not sufficient to obtain a constant approximation ratio, as can be seen via a reduction from Equal Cardinality Partition.

Theorem 66. *Scheduling with non-availability, even if for each time step there is an available machine, does not admit a polynomial time algorithm with a constant approximation ratio unless* $\mathsf{P} = \mathsf{NP}$.

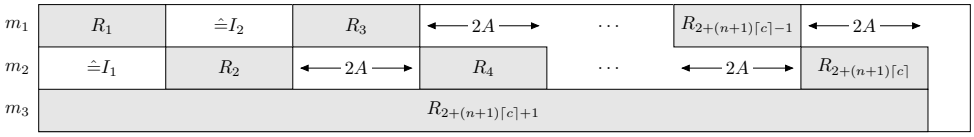
Proof. Let $c \in \mathbb{R}$, $c \geq 1$. We aim at a contradiction and suppose that there is an approximation algorithm B with constant ratio c for scheduling with non-availability where for each time step there is an available machine. We use a reduction from the following NP-complete problem Equal Cardinality Partition (ECP) [47]. The construction is sketched in Figure 5.8.

- *Given:* Finite list $I = (a_1, \ldots, a_n)$ of even cardinality with $a_i \in \mathbb{N}^*$ for each $i \in \{1, \ldots, n\}$, $A \in \mathbb{N}^*$ such that $\sum_{i=1}^{n} a_i = 2A$.
- *Question:* Is there a partition of the list I into lists I_1 and I_2 such that $|I_1| = n/2 = |I_2|$ and $\sum_{i \in I_1} a_i = A = \sum_{i \in I_2} a_i$?

Given an instance I of ECP we define an instance I' of scheduling with non-availability for arbitrary $m \geq 2$ where for each time step there is an available machine as follows.



(a) In the structure of intervals of non-availability of the generated instance I', for every time step there is an available machine.



(b) For a yes-instance I of ECP, for I' we have $C^*_{\max} = 2A(n+1)$; for every no-instance I of ECP, for the instance I' we have $C^*_{\max} > 2A(n+1)(\lceil c \rceil + 1)$.

Figure 5.8.: This sketch illustrates the proof of Theorem 66.

First we define two intervals of non-availability by setting $p_1 := p_2 := A(n+1)$ and fixing these via $(1,0)$ and $(2, A(n+1))$; This means that job 1 is scheduled on machine 1 starting at time 0 and job 2 is scheduled on machine 2 starting at time $A(n+1)$. Furthermore we define additional intervals of non-availability by setting

$$p_{2+\ell} := 2A$$

for each $\ell \in \{1, \ldots, (n+1)\lceil c \rceil\}$ and fix these via list entries

$$(1 + (\ell - 1 \bmod 2), 2A(n+\ell))$$

for each $\ell \in \{1, \ldots, (n+1)\lceil c \rceil\}$. Furthermore we define dummy intervals of non-availability by setting

$$p_{2+(n+1)\lceil c \rceil+\ell} := 2A(n+1)(\lceil c \rceil + 1)$$

for every $\ell \in \{1, \ldots, m-2\}$ and fix these via

$$(2+\ell, 0)$$

for each $\ell \in \{1, \ldots, m-2\}$, which means that all machines except machine 1 and machine 2 are not available in the time interval $[0, 2A(n+1)(\lceil c \rceil + 1))$. Finally we copy the items of I by defining

$$p_{j+2+(n+1)\lceil c \rceil+m-2} := 2A + a_j > 2A$$

for each $j \in \{1, \ldots, n\}$. Note that I' can be generated algorithmically from I in a running time which is polynomially bounded in the encoding length of I. Since

$$p_{2+(n+1)\lceil c \rceil+\ell} > 2A,$$

no job of I' can be scheduled in the interval

$$[2A(n+1), 2A(n+1)(\lceil c \rceil + 1)).$$

Note that for a yes-instance I of ECP, I' has an optimal makespan of value

$$C^*_{\max} = 2A(n+1);$$

if I_1 and I_2 constitute the desired partition, we have

$$\sum_{i \in I_1} 2A + a_i = \sum_{i \in I_2} 2A + a_i = 2A\frac{n}{2} + |A| = An + A = A(n+1),$$

which means that we can execute the job sets indicated by I_1 and I_2 in the intervals $[0, A(n+1))$ and $[A(n+1), 2A(n+1))$ respectively. Conversely, in a schedule with makespan $2A(n+1)$ all jobs must be scheduled during the time interval $[0, 2A(n+1))$ since no more than $n/2$ jobs fit into an availability interval of length $A(n+1)$; such a schedule indicates the partition of I into I_1 and I_2. In total, I' has an optimal makespan of $C^*_{\max} = 2A(n+1)$ if and only if I is a yes-instance of ECP.

Now let I be a no-instance of ECP and consider an optimal schedule of I'. The makespan of the optimal schedule of I' is at least

$$2A(n+1)(\lceil c \rceil + 1)$$

since every job in I' has processing time larger than $2A$ and there must be a job which is not scheduled in $[0, 2A(n+1))$.

Next we show that we can use the algorithm B as an exact algorithm for ECP as follows. For each instance I of ECP we generate an instance I' of our scheduling problem as described above and apply the algorithm B to the instance I'. If the makespan of the generated schedule for I' is smaller than $2A(n+1)(\lceil c \rceil + 1)$, we decide that I is a yes-instance of ECP.

If I is a yes-instance of ECP, the algorithm B generates for I' a schedule with makespan

$$C_{\max} \leq cC^*_{\max} = c2A(n+1) < 2A(n+1)(\lceil c \rceil + 1);$$

for a no-instance I of ECP, the algorithm B generates for I' a schedule with makespan at least $2A(n+1)(\lceil c \rceil + 1)$, which is a lower bound for the optimal makespan of I'.

In total, we can algorithmically decide whether any instance I of ECP is a yes-instance or a no-instance within a polynomial runtime bound, and this is impossible unless $\mathsf{P} = \mathsf{NP}$ holds. □

Consequently we assume that at least one machine is always available. The algorithm we are about to present will use the assumption that the percentage ρ of permanently available machines is constant. Surprisingly, even this restriction is algorithmically hard to approximate. Theorem 67 yields the inapproximability result from Theorem 58.

Theorem 67. *Scheduling with non-availability, even if the ratio $\rho \in (0,1)$ of permanently available machines is constant, does not admit a polynomial time approximation algorithm with an absolute approximation ratio o $3/2 - \epsilon$, unless* $\mathsf{P} = \mathsf{NP}$*, for any $\epsilon \in (0, 1/2]$.*

Proof. We aim at a contradiction and suppose there is a polynomial-time approximation algorithm A for our scheduling problem with approximation ratio $3/2 - \epsilon$. We use a reduction from Modified-3-Partition which is strongly NP-complete.

Given an instance I of Modified-3-Partition we define an instance I' of scheduling with non-availability where a percentage of at least $\rho \in (0,1)$ machines is permanently available as follows; the construction is sketched in Figure 5.9. We choose $K \in \mathbb{N}$ such that $K > \max\{L, (1/2 - \epsilon)L/(2\epsilon)\}$; furthermore we use

$$m := \lceil \frac{n}{1-\rho} \rceil$$

machines and define n suitable intervals of non-availability by setting $p_i := a_i$ for each $i \in \{1, \ldots, n\}$ which are fixed via $(i + m - n, 2K + L - a_i)$. As sketched in Figure 5.9, these jobs are fixed to finish at time $2K + L$. Note that

$$\frac{m-n}{m} = 1 - \frac{n}{m} = 1 - \frac{n}{\lceil \frac{n}{1-\rho} \rceil} \geq 1 - \frac{n}{n/(1-\rho)} = 1 - (1-\rho) = \rho$$

holds. In the further presentation of the proof, we assume that the first $m - n$ machines are permanently available. Furthermore we introduce small jobs by defining

$$p_{n+i} := b_i + K$$

for each $i \in \{1, \ldots, 2n\}$. Finally we define $m - n$ dummy jobs

$$p_{3n+i} := 2K + L$$

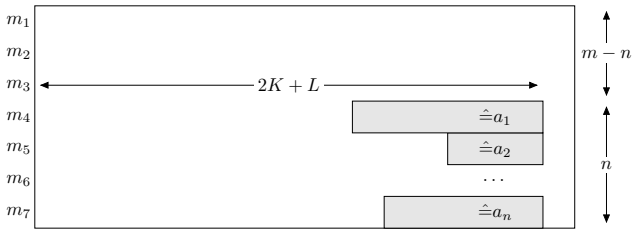


Figure 5.9.: This sketch illustrates the proof of Theorem 67.

for each $i \in \{1, \ldots, m-n\}$. Note that I' can be generated from I in running time polynomial in the encoding length of I. Note that for a yes-instance I of Modified-3-Partition, we can execute the dummy jobs of I' on the machines $1, \ldots, m-n$. Finally we use the existing permutation π; since

$$a_i + b_{\pi(2i-1)} + b_{\pi(2i)} = L$$

for each $i \in \{1, \ldots, n\}$, we have

$$b_{\pi(2i-1)}K + b_{\pi(2i)}K = 2K + L - a_i.$$

This means that the small jobs corresponding to $b_{\pi(2i-1)}$ and $b_{\pi(2i)}$ can be executed in the interval $[0, 2K + L - a_i)$ on machine $m - n + i$ for each $i \in \{1, \ldots, n\}$. Consequently, I' has an optimal makespan of $C^*_{\max} = 2K + L$. Conversely, in a schedule with makespan $2K + L$ the dummy jobs must be executed on machines $1, \ldots, m-n$, hence the small jobs must run on machines $m-n+1, \ldots, m$. Note that the processing time of each small job is larger than K; consequently, we have $3K > 2K + L$, hence it is impossible that more than 2 small jobs run on the same machine in the interval $[0, 2K + L)$. This means that on each machine $i \in \{m-n, m\}$, exactly 2 small jobs are executed, which indicates the desired permutation π. In total, I' has an optimal makespan of $C^*_{\max} = 2K + L$ if and only if I is a yes-instance of Modified-3-Partition.

Now let I be a no-instance of Modified-3-Partition. Then in any schedule for I' two cases can occur.

Case 1: The dummy jobs run on the machines in $\{1, \ldots, m-n\}$. Then there is a small job which is either scheduled together with a dummy job or on one machines $\{m-n+1, \ldots, m\}$ after the interval of non-availability. In total, we obtain a job with completion time at least $3K + L$. *Case 2:* There is a dummy job which runs on one of the machines in $\{m-n+1, \ldots, m\}$. Since its processing time is $2K + L$, it must run after the interval of non-availability; here we also obtain a completion time at least $3K + L$.

In total, the makespan of any schedule of I' must be at least

$$3K + L.$$

Next we show that we can use the algorithm A as an exact algorithm for Modified-3-Partition as follows. For each instance I of Modified-3-Partition we generate an instance

of our scheduling problem as described above and apply the algorithm A to the instance I'. If the makespan of the generated schedule for I' is smaller than $3K + L$, we decide that I is a yes-instance of Modified-3-Partition.

Let I be a yes-instance of Modified-3-Partition. Note that the inequality

$$K > (1/2 - \epsilon)L/(2\epsilon)$$

can be rearranged to $K2\epsilon > (1/2 - \epsilon)L$. Using this inequality we obtain

$$(\frac{3}{2} - \epsilon)(2K + L) = 3K + (\frac{3}{2} - \epsilon)L - K2\epsilon < 3K + (\frac{3}{2} - \epsilon)L - (\frac{1}{2} - \epsilon)L = 3K + L.$$

Now we use this inequality to argue that the algorithm A generates for I' a solution with value

$$C_{\max} \leq (3/2 - \epsilon)C^*_{\max} = (3/2 - \epsilon)(2K + L) < 3K + L,$$

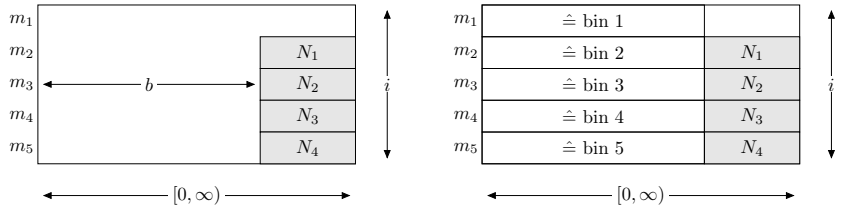
where in the last step we used the estimation from above. For a no-instance I of Modified-3-Partition, the algorithm A generates for I' a schedule with makespan at least $3K + L$, which is a lower bound for the optimal makespan of I'.

In total, we can algorithmically decide whether any instance I of Modified-3-Partition is a yes-instance or a no-instance within a polynomial runtime bound, which is impossible unless $\mathsf{P} = \mathsf{NP}$ holds. □

Comment. Note that in the construction from the proof, we can also use $\epsilon := 1/n$, which means that there is also no approximation algorithm for the problem under discussion with approximation ratio $3/2 - 1/n$. Furthermore the construction from the proof uses at most one interval of non-availability per machine; hence, the result is also valid if the number of non-availability intervals per machine is restricted to one.

Without the restriction of a constant percentage of machines being permanently available, scheduling with non-availability yields an interesting connection to the well-known problem Bin Packing; the existence of an approximation algorithm for scheduling with non-availability with constant ratio implies the existence of an approximation algorithm for Bin Packing with additive error. However, this is an open problem, as discussed in [58], Chapter 2, page 67. Theorem 68 can be seen as an informal reason for scheduling with non-availability being hard to approximate.

Theorem 68. *Suppose there is a polynomial time algorithm for scheduling with non-availability where at least one machine is permanently available with absolute approxi-*



(a) The structure of intervals of non-availability of the generated instance I'_i roughly resembles i bins.

(b) For an optimal solution of I'_i for $i \geq \mathrm{OPT}(I)$ has a makespan of $C^*_{\max} \leq b$.

Figure 5.10.: This sketch illustrates the proof of Theorem 68.

mation ratio $c \in \mathbb{N} \setminus \{1\}$. Then there is a polynomial time algorithm for Bin Packing with additive error $2(c-1)$.

Proof. Let A be an algorithm for scheduling with non-availability with approximation ratio $c \in \mathbb{N} \setminus \{1\}$; the following construction is illustrated in Figure 5.10.

For each instance I of Bin Packing with n items and bin size b we define n instances I'_i for $i \in \{1, \ldots, n\}$ of scheduling with non-availability by setting $m = i$ and defining intervals of non-availability $(j+1, b)$ of size ∞ for each $j \in \{1, \ldots, i-1\}$. Note each I'_i can be generated from I within a polynomial runtime bound. For each instance I of Bin Packing, n is an upper bound for $\mathrm{OPT}(I)$, the minimum number of bins in which the items of I can be packed.

Let I be an instance of Bin Packing. Let

$$n' := \min\{i \in \{1, \ldots, n\} \setminus \{1\} | A(I'_i) \leq cb\}$$

which can be found in polynomial time by enumeration since n is a lower bound for the encoding length of I. Hence $A(I'_{n'-1}) > cb$, from which follows $C^*_{\max}(I'_{n'-1}) > b$. This means that it is impossible to pack the items of I in less than n' bins of size b, hence $\mathrm{OPT}(I) \geq n'$ holds. Consider the schedule for $I'_{n'}$ generated by A. The schedule for the machines $2, \ldots, n'$ yields $n'-1$ bins. Furthermore, the jobs scheduled on machine 1 can be packed in $1 + 2(c-1)$ bins by packing all jobs from intervals of the form $[\ell b, (\ell+1)b)$ into one bin and packing each job crossing the boundaries of such adjacent intervals into a separate bin. In total, the number of bins needed for this packing can be bounded by

$$n' - 1 + 1 + 2(c-1) \leq \mathrm{OPT}(I) + 2(c-1),$$

hence the approach yields an algorithm for Bin Packing which uses at most $2(c-1)$ additional bins. □

5.4. Conclusion

In this chapter we have studied scheduling with fixed jobs and non-availability where the number m of machines is part of the input.

For scheduling with fixed jobs, we have presented an approximation algorithm with ratio $3/2+\epsilon$ for any $\epsilon \in (0, 1/2]$; furthermore, we have shown how to improve this ratio to $3/2$. Our result improves the previously best known approximation ratio of $2+\epsilon$ by Scharbrodt, Steger & Weisser [130]. Finally, this bound is best possible since for the problem under consideration, a ratio better than $3/2$ is impossible unless $\mathsf{P} = \mathsf{NP}$, as already shown in [130, 131, 132]. In total, we have obtained a tight approximation result.

For scheduling with non-availability where the number m of machines is part of the input, we have studied the restricted scenario where a constant fraction of the machines is permanently available. For this scenario we have presented an approximation algorithm with ratio $3/2+\epsilon$ for any $\epsilon \in (0, 1/2]$. Using the same techniques as for scheduling with fixed jobs, this ratio can be improved to $3/2$. Furthermore we have proved that here a ratio better than $3/2$ is impossible unless $\mathsf{P} = \mathsf{NP}$.

As a side issue, we have pointed out an interesting relation of scheduling with non-availability to Bin Packing. More precisely, if as in Chapter 4, we require only one machine to be permanently available, the problem is algorithmically at least as hard as approximation of Bin Packing within an additive error. However, whether the latter is possible or not is currently an open question. Finally, it would also be desirable to improve the running times of the obtained algorithms.

6. Concluding Remarks

In this thesis we have approached several problems with approximation algorithms. We have addressed the mixed packing and covering problem, which is a fairly general algorithmic modelization tool; furthermore we have studied an implementation of the max-min resource sharing problem and finally obtained tight or basically tight approximation algorithms for scheduling with fixed jobs and scheduling with non-availability.

Concerning the results obtained in Chapter 2, we like to point out that approximately solving fractional mathematical problems, especially linear programs, is an intriguing field which is rich and powerful in its techniques; here the textbook by Bienstock [8] discusses the history, theoretical foundation and practical applications of the general approach. However, Bienstock describes the *exponential* potential function, whereas our algorithms are based on the *logarithmic* potential function and technical approach pursued in [54], which was later successfully extended in [66, 67, 69, 70, 79] by using more general block solvers and mixed packing and covering problems instead of pure packing or covering problems. For most applications these algorithms can be implemented as column-generation algorithms, like the application for the multicommodity flow problem in Chapter 2 and Strip Packing in Chapter 2. They are also of practical interest since, despite the usually lengthy elementary analysis, the implementations tend to be rather compact and typically will not use sophisticated data structures. In the context with our algorithms, it is an interesting open question whether a *lower* bound on the number of iterations in dependency of M and ϵ can be found.

Concerning the results in Chapter 3, we have implemented the algorithm from [54] in C++; we used an LP relaxation arising in the context of an AFPTAS for Strip Packing by Kenyon & Rémila [91, 92], which was later refined by Jansen [71]; the approach is also discussed in detail in [73]. We evaluated a dynamic optimization of the step length used for interpolation in comparison with the static step length used in the analysis in order to maximize the change in the reduced potential; we achieved a remarkable speedup. Furthermore, our implementation is fully parameterized in the sense that it is completely decoupled from the specific application. This means that it can be relatively easily used

for implementations for other applications. Furthermore we would like to point out the similiarity between the algorithms in [54] and [79]. We propose to implement the algorithm in [79] in the same generality for experimentation; here we also expect that dynamic optimization of the step length results in a significant speedup compared to the step length from the analysis. Similar to the algorithms in Chapter 2 and the algorithm from [79], no *lower* bound on the number of iterations in dependency of M and ϵ is known.

Concerning the scheduling problems studied in Chapters 4 and 5, we like to point out that despite the fact that the problem formulations are very natural, the problems apparently have not been fully studied under the paradigm of approximation algorithms; here the results by Scharbrodt, Steger & Weisser [130, 131, 132], Hwang et al. [63], the survey paper by Lee [109], and the paper by Kellerer [86] constitute positive exceptions. On one hand, however, these achievements have not led to the discovery of an PTAS for scheduling with non-availability for m constant or the approximation algorithms with ratios $(3/2+\epsilon)$ or $3/2$ for scheduling with non-availability for m part of the input. On the other hand, however, for all of the algorithms presented in Chapters 4 and 5, we build on a PTAS for the multiple subset sum problem from which we greatly benefit. Although this problem is relatively old and its formulation is also very natural and straightforward, only quite recently a PTAS for the most general case has been obtained by Caprara, Kellerer & Pferschy [18] where later Jansen [72] obtained a parameterized approximation scheme, which is in a certain sense more efficient. This recent development clearly shows that classical combinatorial problems themselves are still very interesting and useful for the modelization of other algorithmic problems.

Furthermore, the problem of the approximability of scheduling with fixed jobs for m part of the input was addressed briefly in [130, 131, 132] where a lower bound of $3/2$ was obtained. In Chapter 5, we have basically complemented this result by presenting an approximation ratio with ratio $3/2+\epsilon$ for any accuracy parameter ϵ; however, it remains an interesting open question whether there is an approximation algorithm for this problem with ratio *exactly* $3/2$. Furthermore, since our algorithm is more of theoretical interest, we like to point out that it would be worthwhile to either reduce its elimination steps or to perform an analysis of some greedy strategy like LPT for practical applications.

List of Figures

List of Tables

Bibliography

[1] M. Aizatulin, F. Diedrich, and K. Jansen. Implementation of approximation algorithms for the max-min resource sharing problem. In C. Àlvarez and M. J. Serna, editors, *WEA*, volume 4007 of *Lecture Notes in Computer Science*, pages 207–218. Springer, 2006.

[2] F. Baille, E. Bampis, and C. Laforest. Bicriteria scheduling of parallel degradable tasks for network access under pricing constraints. In *Proceedings of the International Network Optimizazion Conference, INOC*, pages 37–42.

[3] F. Baille, E. Bampis, and C. Laforest. Maximization of the size and the weight of schedules of degradable intervals. In K.-Y. Chwa and J. I. Munro, editors, *COCOON*, volume 3106 of *Lecture Notes in Computer Science*, pages 219–228. Springer, 2004.

[4] B. S. Baker, D. J. Brown, and H. P. Katseff. A 5/4 algorithm for two-dimensional packing. *J. Algorithms*, 2(4):348–368, 1981.

[5] B. S. Baker, E. G. C. Jr., and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9(4):846–855, 1980.

[6] N. Bansal, X. Han, K. Iwama, M. Sviridenko, and G. Zhang. Harmonic algorithm for 3-dimensional strip packing problem. In N. Bansal, K. Pruhs, and C. Stein, editors, *SODA*, pages 1197–1206. SIAM, 2007.

[7] G. Batra, N. Garg, and G. Gupta. Heuristic improvements for computing maximum multicommodity flow and minimum multicut. In Brodal and Leonardi [16], pages 35–46.

[8] D. Bienstock. *Potential function methods for approximately solving linear programming problems: theory and practice*. Kluwer, 2002.

[9] D. Bienstock and G. Iyengar. Faster approximation algorithms for packing and covering problems. Technical report, Department of IEOR, Columbia University, 2004.

[10] D. Bienstock and G. Iyengar. Solving fractional packing problems in $O^*(1/\epsilon)$ iterations. In L. Babai, editor, *STOC*, pages 146–155. ACM, 2004.

[11] D. Bienstock and G. Iyengar. Approximating fractional packings and coverings in $O(1/\epsilon)$ iterations. *SIAM J. Comput.*, 35(4):825–854, 2006.

[12] J. Błażewicz, W. Cellary, R. Slowinski, and J. Węglarz. Scheduling under resource constraints – deterministic models. *Annals of Operations Research*, 7:359, 1986.

[13] K.-H. Borgwardt. The average number of pivot steps required by the simplex method in polynomial. *Zeitschrift für Operations Research*, 26:157–177, 1982.

[14] K.-H. Borgwardt. Some distribution-independent results about the asymptotic order of the average number of pivot steps of the simplex method. *Mathematics of Opeations Research*, 7:441–462, 1982.

[15] M. Bouklit, D. Coudert, J.-F. Lalande, C. Paul, and H. Rivano. Approximate multicommodity flow for WDM networks design. In J. F. Sibeyn, editor, *SIROCCO*, volume 17 of *Proceedings in Informatics*, pages 43–56. Carleton Scientific, 2003.

[16] G. S. Brodal and S. Leonardi, editors. *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3669 of *Lecture Notes in Computer Science*. Springer, 2005.

[17] A. Caprara, H. Kellerer, and U. Pferschy. The multiple subset sum problem. Technical report, Technische Universität Graz, 1998.

[18] A. Caprara, H. Kellerer, and U. Pferschy. A PTAS for the multiple subset sum problem with different knapsack capacities. *Inf. Process. Lett.*, 73(3-4):111–118, 2000.

[19] A. Caprara, H. Kellerer, and U. Pferschy. A 3/4-approximation algorithm for multiple subset sum. *J. Heuristics*, 9(2):99–111, 2003.

[20] I. Caragiannis, A. Ferreira, C. Kaklamanis, S. Perennes, and H. Rivano. Fractional path coloring with applications to WDM networks. In F. Orejas, P. G. Spirakis,

and J. van Leeuwen, editors, *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 732–743. Springer, 2001.

[21] R. D. Carr, L. Fleischer, V. J. Leung, and C. A. Phillips. Strengthening integrality gaps for capacitated network design and covering problems. In *SODA*, pages 106–115, 2000.

[22] M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. A. Plotkin. Approximating a finite metric by a small number of tree metrics. In *FOCS*, pages 379–388, 1998.

[23] C. Chekuri and S. Khanna. A PTAS for the multiple knapsack problem. In *SODA*, pages 213–222, 2000.

[24] C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM J. Comput.*, 35(3):713–728, 2005.

[25] F. A. Chudak and V. Eleutério. Improved approximation schemes for linear programming relaxations of combinatorial optimization problems. In M. Jünger and V. Kaibel, editors, *IPCO*, volume 3509 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2005.

[26] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Comput.*, 9(4):808–826, 1980.

[27] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[28] M. Dawande, J. Kalagnanam, P. Keskinocak, F. S. Salman, and R. Ravi. Approximation algorithms for the multiple knapsack problem with assignment restrictions. Technical report, IBM Research Division, 1998.

[29] M. Dawande, J. Kalagnanam, P. Keskinocak, F. S. Salman, and R. Ravi. Approximation algorithms for the multiple knapsack problem with assignment restrictions. *J. Comb. Optim.*, 4(2):171–186, 2000.

[30] W. F. de la Vega and G. S. Lueker. Bin packing can be solved within $1+\epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[31] F. Diedrich, R. Harren, K. Jansen, R. Thöle, and H. Thomas. Approximation algorithms for 3d orthogonal knapsack. In J. Yi Cai, S. B. Cooper, and H. Zhu, editors, *TAMC*, volume 4484 of *Lecture Notes in Computer Science*, pages 34–45. Springer, 2007.

[32] F. Diedrich and K. Jansen. An approximation algorithm for the general mixed packing and covering problem. In B. Chen, M. Paterson, and G. Zhang, editors, *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 128–139. Springer, 2007.

[33] F. Diedrich and K. Jansen. Faster and simpler approximation algorithms for mixed packing and covering problems. *Theor. Comput. Sci.*, 377(1-3):181–204, 2007.

[34] F. Diedrich and K. Jansen. Improved approximation algorithms for scheduling with fixed jobs. In Mathieu [119], pages 675–684.

[35] F. Diedrich, K. Jansen, F. Pascual, and D. Trystram. Approximation algorithms for scheduling with reservations. In S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, editors, *HiPC*, volume 4873 of *LNCS*, pages 297–307. Springer, 2007.

[36] F. Diedrich, K. Jansen, F. Pascual, and D. Trystram. Approximation algorithms for scheduling with reservations. Accepted for publication in Algorithmica, 2008.

[37] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[38] F. Ergün, R. K. Sinha, and L. Zhang. An improved fptas for restricted shortest path. *Inf. Process. Lett.*, 83(5):287–291, 2002.

[39] L. Eyraud-Dubois, G. Mounié, and D. Trystram. Analysis of scheduling algorithms with reservations. In *IPDPS*, pages 1–8. IEEE, 2007.

[40] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.

[41] L. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. In *FOCS*, pages 24–31, 1999.

[42] L. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13(4):505–520, 2000.

[43] L. Fleischer. A fast approximation scheme for fractional covering problems with variable upper bounds. In J. I. Munro, editor, *SODA*, pages 1001–1010. SIAM, 2004.

[44] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[45] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

[46] M. R. Garey and D. S. Johnson. "strong" NP-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, 1978.

[47] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[48] N. Garg and R. Khandekar. Fractional covering with upper bounds on the variables: Solving LPs with negative entries. In S. Albers and T. Radzik, editors, *ESA*, volume 3221 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 2004.

[49] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *FOCS*, pages 300–309, 1998.

[50] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM J. Comput.*, 37(2):630–652, 2007.

[51] M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Optimization*, 4:86–107, 1994.

[52] M. D. Grigoriadis and L. G. Khachiyan. Approximate minimum-cost multicommodity flows in $\tilde{O}(\epsilon^{-2}KNM)$ time. *Math. Program.*, 75:477–482, 1996.

[53] M. D. Grigoriadis and L. G. Khachiyan. Coordination complexity of parallel price-directive decomposition. *Mathematics of Operations Research*, 21:321–340, 1996.

[54] M. D. Grigoriadis, L. G. Khachiyan, L. Porkolab, and J. Villavicencio. Approximate max-min resource sharing for structured concave optimization. *SIAM J. on Optimization*, 11(4):1081–1091, 2001.

[55] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, 1988.

[56] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17(1):36–42, 1992.

[57] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.

[58] D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1996.

[59] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

[60] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM J. Comput.*, 17(3):539–551, 1988.

[61] E. Hopper and C. C. H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128(1):34–57, 2000.

[62] H.-C. Hwang and S. Y. Chang. Parallel machines scheduling with machine shutdowns. *Computers and Mathematics with Applications*, 36(3):21–31, 1998.

[63] H.-C. Hwang, K. Lee, and S. Y. Chang. The effect of machine availability on the worst-case performance of LPT. *Disc. App. Math.*, 148(1):49–61, 2005.

[64] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

[65] ILOG CPLEX Division, 889 Alder Avenue, Suite 200, Incline Village, NV 89451, USA. *CPLEX 7.500*.

[66] K. Jansen. Approximation algorithms for mixed fractional packing and covering problems. In Lévy et al. [114], pages 223–236.

[67] K. Jansen. Approximation algorithms for the general max-min resource sharing problem: Faster and simpler. In T. Hagerup and J. Katajainen, editors, *SWAT*, volume 3111 of *Lecture Notes in Computer Science*, pages 311–322. Springer, 2004.

[68] K. Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39(1):59–81, 2004.

[69] K. Jansen. An approximation algorithm for the general max-min resource sharing problem. *Math. Program.*, 106(3):547–566, 2006.

[70] K. Jansen. Approximation algorithm for the mixed fractional packing and covering problem. *SIAM Journal on Optimization*, 17(2):331–352, 2006.

[71] K. Jansen. Approximation algorithms for min-max and max-min resource sharing problems and applications. In E. Bampis, K. Jansen, and C. Kenyon, editors, *Efficient Approximation and Online Algorithms*, volume 3484 of *Lecture Notes in Computer Science*, pages 156–202. Springer, 2006.

[72] K. Jansen. Parameterized approximation scheme for the multiple knapsack problem. In Mathieu [119], pages 665–674.

[73] K. Jansen and M. Margraf. *Approximative Algorithmen und Nichtapproximierbarkeit*. Gruyter, 2008.

[74] K. Jansen and L. Porkolab. On preemptive resource constrained scheduling: Polynomial-time approximation schemes. In W. Cook and A. S. Schulz, editors, *IPCO*, volume 2337 of *Lecture Notes in Computer Science*, pages 329–349. Springer, 2002.

[75] K. Jansen and L. Porkolab. On preemptive resource constrained scheduling: polynomial time approximation schemes. *SIAM J. Discrete Math.*, 20:545–563, 2006.

[76] K. Jansen and L. Porkolab. On preemptive resource constrained scheduling: Polynomial-time approximation schemes. *SIAM J. Discrete Math.*, 20(3):545–563, 2006.

[77] K. Jansen and R. Solis-Oba. An asymptotic approximation algorithm for 3d-strip packing. In *SODA*, pages 143–152. ACM Press, 2006.

[78] K. Jansen and G. Zhang. Maximizing the total profit of rectangles packed into a rectangle. *Algorithmica*, 47(3):323–342, 2007.

[79] K. Jansen and H. Zhang. Approximation algorithms for general packing problems with modified logarithmic potential function. In R. A. Baeza-Yates, U. Montanari,

and N. Santoro, editors, *IFIP TCS*, volume 223 of *IFIP Conference Proceedings*, pages 255–266. Kluwer, 2002.

[80] K. Jansen and H. Zhang. Approximation algorithms for general packing problems and their application to the multicast congestion problem. 2007. To appear in Mathematical Programming.

[81] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[82] I. Kacem. Approximation algorithms for the makespan minimization with positive tails on a single machine with a fixed non-availability interval. *Journal of Combinatorial Optimization*, 2007.

[83] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, 1984, Washington, D.C., USA*, pages 302–311. ACM, 1984.

[84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[85] N. Karmarkar and R. M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *FOCS*, pages 312–320. IEEE, 1982.

[86] H. Kellerer. Algorithms for multiprocessor scheduling with machine release times. *IIE Transactions*, 30(11), 1998.

[87] H. Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In D. S. Hochbaum, K. Jansen, J. D. P. Rolim, and A. Sinclair, editors, *RANDOM-APPROX*, volume 1671 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 1999.

[88] H. Kellerer, R. Mansini, U. Pferschy, and M. G. Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *J. Comput. Syst. Sci.*, 66(2):349–370, 2003.

[89] H. Kellerer and U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *J. Comb. Optim.*, 3(1):59–71, 1999.

[90] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[91] C. Kenyon and E. Rémila. Approximate strip packing. In *FOCS*, pages 31–36, 1996.

[92] C. Kenyon and E. Rémila. A near-optimal solution to a two-dimensional cutting stock problem. *Math. Oper. Res.*, 25(4):645–656, 2000.

[93] C. Kenyon and E. Rémila. A near-optimal solution to a two dimensional cutting stock problem. *Mathematics of Operations Research*, 25:645–656, 2000.

[94] L. G. Khachiyan. Polynomial algorithms in linear programming. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 20:1093–1096, 1980.

[95] L. G. Khachiyan. Polynomial algorithms in linear programming. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 20:53–72, 1980.

[96] R. Khandekar. *Lagrangian relaxation based algorithms for convex programming problems.* PhD thesis, Indian Institute of Technology, New Delhi, 2004.

[97] P. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: a linear-space ($\log^2$)-time algorithm. In Mathieu [119], pages 236–245.

[98] P. N. Klein and S. Sairam. A parallel randomized approximation scheme for shortest paths. In *STOC*, pages 750–758. ACM, 1992.

[99] D. E. Knuth. *The art of computer programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1968.

[100] J. Könemann. Fast combinatorial algorithms for packing and covering problems. Master's thesis, Max-Planck-Institute for Computer Science Saarbrücken, 1998.

[101] B. Korte and J. Vygen. *Combinatorial Optimization – Theory and Algorithms.* Springer, 2000.

[102] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms, Algorithms and Combinatorics 2.* Springer, 2000.

[103] C. Koufogiannakis and N. E. Young. Beating simplex for fractional packing and covering linear programs. In *FOCS*, pages 494–504. IEEE Computer Society, 2007.

[104] C. Koufogiannakis and N. E. Young. Beating simplex for fractional packing and covering linear programs. *CoRR*, abs/0801.1987, 2008.

[105] K. L. Krause, V. Y. Shen, and H. D. Schwetman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *J. ACM*, 22(4):522–550, 1975.

[106] K. L. Krause, V. Y. Shen, and H. D. Schwetman. Errata: "analysis of several task-scheduling algorithms for a model of multiprogramming computer systems". *J. ACM*, 24(3):527, 1977.

[107] E. L. Lawler. Fast approximation algorithms for knapsack problems. *Math. Oper. Res.*, 4(4):339–356, 1979.

[108] C.-Y. Lee. Parallel machines scheduling with non-simultaneous machine available time. *Disc. App. Math.*, 30:53–61, 1991.

[109] C.-Y. Lee. Machine scheduling with an availability constraint. *J. Global Optimization, Special Issue on Optimization of Scheduling Applications*, 9:363–384, 1996.

[110] C.-Y. Lee, Y. He, and G. Tang. A note on "parallel machine scheduling with non-simultaneous machine available time". *Disc. App. Math.*, 100(1-2):133–135, 2000.

[111] F. T. Leighton, F. Makedon, S. A. Plotkin, C. Stein, É. Stein, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *J. Comput. Syst. Sci.*, 50(2):228–243, 1995.

[112] F. T. Leighton, F. Makedon, S. A. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. In *STOC*, pages 101–111. ACM, 1991.

[113] J. Y.-T. Leung, editor. *Handbook of Scheduling*. Chapman & Hall, 2004.

[114] J.-J. Lévy, E. W. Mayr, and J. C. Mitchell, editors. *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*. Kluwer, 2004.

[115] C.-J. Liao, D.-L. Shyur, and C.-H. Lin. Makespan minimization for two parallel machines with an availability constraint. *European J. of Operational Research*, 160:445–456, 2003.

[116] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[117] Q. Lu and H. Zhang. Implementation of approximation algorithms for the multicast congestion problem. In S. E. Nikoletseas, editor, *WEA*, volume 3503 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2005.

[118] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990.

[119] C. Mathieu, editor. *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*. SIAM, 2009.

[120] T. Matsui. Approximation algorithms for maximum independent set problems and fractional coloring problems on unit disk graphs. In J. Akiyama, M. Kano, and M. Urabe, editors, *JCDCG*, volume 1763 of *Lecture Notes in Computer Science*, pages 194–200. Springer, 1998.

[121] N. Megow, R. H. Möhring, and J. Schulz. Turnaround scheduling in chemical manufacturing. In *In Proceedings of the 8th Workshop on Models and Algorithms for Planning and Scheduling Problems, MAPSP 2007, Istanbul, Turkey*, 2007.

[122] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 1998.

[123] C. A. Phillips. The network inhibition problem. In *STOC*, pages 776–785, 1993.

[124] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20:257–301, 1995.

[125] T. Radzik. Fast deterministic approximation for the multicommodity flow problem. In *SODA*, pages 486–492, 1995.

[126] T. Radzik. Fast deterministic approximation for the multicommodity flow problem. *Math. Program.*, 77:43–58, 1997.

[127] S. Sahni. Algorithms for scheduling independent tasks. *J. ACM*, 23(1):116–127, 1976.

[128] R. Saigal. *Linear Programming – A Modern Integrated Analysis*. Kluwer, 1995.

[129] E. Sanlaville and G. Schmidt. Machine scheduling with availability constraints. *Acta Inf.*, 35(9):795–811, 1998.

[130] M. Scharbrodt. *Produktionsplanung in der Prozessindustrie: Modelle, effiziente Algorithmen und Umsetzung.* PhD thesis, Fakultät für Informatik, Technische Universität München, 2000.

[131] M. Scharbrodt, A. Steger, and H. Weisser. Approximability of scheduling with fixed jobs. In *SODA*, pages 961–962, 1999.

[132] M. Scharbrodt, A. Steger, and H. Weisser. Approximability of scheduling with fixed jobs. *J. Scheduling*, 2:267–284, 1999.

[133] I. Schiermeyer. Reverse-Fit: A 2-optimal algorithm for packing rectangles. In J. van Leeuwen, editor, *ESA*, volume 855 of *Lecture Notes in Computer Science*, pages 290–299. Springer, 1994.

[134] E. R. Schreinerman and D. H. Ullman. *Fractional Graph Theory: A Rational Approach to the Theory of Graphs.* Wiley Interscience Series in Discrete Mathematics. Wiley, 1997.

[135] P. Schuurman and G. J. Woeginger. Approximation schemes - a tutorial. To appear in the book "Lectures on Scheduling", 2008.

[136] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM J. Comput.*, 26(2):401–409, 1997.

[137] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[138] N. Vaideeswaran. Approximation algorithms for multicast networks' congestion problem. Master's thesis, Technical Faculty of Engineering, Universität zu Kiel, 2005.

[139] P. M. Vaidya. A new algorithm for minimizing convex functions over convex sets (extended abstract). In *FOCS*, pages 338–343. IEEE, 1989.

[140] P. M. Vaidya. An algorithm for linear programming which requires $O(((m+n)n^2+(m+n)^{1.5}n)L$ arithmetic operations. *Math. Program.*, 47:175–201, 1990.

[141] P. M. Vaidya. A new algorithm for minimizing convex functions over convex sets. *Math. Program.*, 73:291–341, 1996.

[142] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[143] J. Villavicencio and M. D. Grigoriadis. Approximate lagrangian decomposition with a modified karmarkar logarithmic potential. volume 450 of *Lecture Notes in Economics and Mathematical Systems*, pages 741–485. Springer, 1997.

[144] R. Wanka. *Approximationsalgorithmen: Eine Einführung*. Vieweg+Teubner, 2006.

[145] A. Warburton. Approximation of pareto optima in multiple-objective, shortest-path problems. *Operations Research*, 35(1):70–79, 1987.

[146] G. J. Woeginger. When does a dynamic programming formulation guarantee the existence of an FPTAS? In *SODA*, pages 820–829, 1999.

[147] D. Ye and G. Zhang. On-line scheduling mesh jobs with dependencies. *Theor. Comput. Sci.*, 372(1):94–102, 2007.

[148] N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *FOCS*, pages 538–546, 2001.

[149] N. E. Young. Randomized rounding without solving the linear program. *CoRR*, cs.DS/0205036, 2002.

[150] N. E. Young. Sequential and parallel algorithms for mixed packing and covering. *CoRR*, cs.DS/0205039, 2002.

[151] N. E. Young. personal communication, 2004.

A. Implementation to Chapter 3

Contents of file `efficient\basics\basics.h`

```
// ------------------------------------------------------------------------
// basics.h: function templates. fdi - 24oct05
// ------------------------------------------------------------------------
#ifndef BASICS_INCLUDED
#define BASICS_INCLUDED
#pragma warning ( disable : 4786 )                          // ms specific
#include <vector>
#include <iterator>
template <class T> T maximum( T a, T b ){ return ( a > b ) ? a : b; }
template <class T> T minimum( T a, T b ){ return ( a < b ) ? a : b; }
template <class T> void
delete_element( std::vector<T>& v, unsigned int j ){
  std::vector<T>::iterator i = v.begin(); std::advance( i, j ); v.erase( i ); }
#endif // ndef BASICS_INCLUDED
```

Contents of file `efficient\basics\mat.h`

```
// ------------------------------------------------------------------------
// mat.h: class template. fdi - 24oct05
// ------------------------------------------------------------------------
#ifndef MAT_INCLUDED
#define MAT_INCLUDED
#pragma warning ( disable : 4786 )                          // ms specific
#include "vec.h"
#include "basics.h"
#include <vector>
#include <iterator>
#include <iostream>
#include <iomanip>
template <class T> class mat{ public:
  // ------------------------------------------------------------------------
  // mat constructor and destructor
  // ------------------------------------------------------------------------
  mat(){ r = 0; c = 0; } mat( mat<T>& rhs ){ *this = rhs; }
```

```
virtual ~mat(){ for ( unsigned int i = 0; i < r; delete_row( i ), i++ );
  pRows.clear(); }
// ------------------------------------------------------------------------
// mat setter and getter methods, operators
// ------------------------------------------------------------------------
void resize( unsigned int newr, unsigned int newc ){
  unsigned int i, keep = minimum( newr, r ); std::vector<T>* pNewVec;
  for ( i = 0; i < keep; i++ )                          // resize rows
    (*pRows[i]).resize( newc, (T)0 );                   // that are kept
  if ( keep < r ){                                      // remove rows
    for( i = keep; i < r; delete pRows[i], i++ );
    pRows.resize( keep );                               // remove pointers to
  }                                                     // deleted rows
  else                                                  // introduce new rows
    for( i = keep; i < newr; i++ ){
      pNewVec = new std::vector< T >;
      (*pNewVec).resize( newc, (T)0 );
      pRows.push_back( pNewVec );
    }
  r = newr; c = newc;                                   // adjust members
}
unsigned int num_rows() const { return r; }
unsigned int num_cols() const { return c; }
T& operator()( unsigned int i, unsigned int j ) { return (*pRows[i])[j]; }
mat<T>& operator=( mat<T>& rhs ){
  std::vector<std::vector<T>*>::iterator i;
  if ( this == &rhs ) return *this;
  for ( i = pRows.begin(); i != pRows.end();(**i).clear(), delete *i, i++ );
  pRows.clear();
  for ( i = rhs.pRows.begin(); i != rhs.pRows.end(); i++ ){
    std::vector<T>* pNewVec = new std::vector<T>;
    if ( *i != NULL )
      (*pNewVec).assign( (**i).begin(), (**i).end() );
    pRows.push_back( pNewVec );
  }
  r = rhs.r; c = rhs.c; return *this;
}
T min(){ T out = (*this)( 0, 0 );
  for ( unsigned int i = 0; i < r; i++ )
    for ( unsigned int j = 0; j < c; j++ )
      out = minimim( out, (*this)( i, j ) ); return out; }
T max(){ T out = (*this)( 0, 0 );
```

```
  for ( unsigned int i = 0; i < r; i++ )
    for ( unsigned int j = 0; j < c; j++ )
      out = maximum( out, (*this)( i, j ) ); return out; }
mat<T> operator-(){ mat<T> r = *this;
  for ( unsigned int i = 0; i < r.num_rows(); i++ )
    for ( unsigned int j = 0; j < r.num_rows(); j++ )
      r( i, j ) = -r( i, j ); return r; }
mat<T> operator+( mat<T>& rhs ){ mat<T> r = *this;
  for ( unsigned int i = 0; i < r.num_rows(); i++ )
    for ( unsigned int j = 0; j < r.num_rows(); j++ )
      r( i, j ) += rhs( i, j ); return r; }
mat<T> operator-( mat<T>& rhs ){ mat<T> r = *this;
  for ( unsigned int i = 0; i < r.num_rows(); i++ )
    for ( unsigned int j = 0; j < r.num_rows(); j++ )
      r( i, j ) -= rhs( i, j ); return r; }
mat<T> operator*( mat<T>& rhs ){ mat<T> r; r.resize( (*this).r, rhs.c );
  for ( unsigned int i = 0; i < r.num_rows(); i++ )
    for ( unsigned int j = 0; j < r.num_cols(); j++ )
      for ( unsigned int k = 0; k < rhs.num_rows(); k++ )
        r( i, j ) += (*this)( i, k ) * rhs( k, j ); return r; }
vec<T> operator*( vec<T>& rhs ){ vec< T > r; r.resize( (*this).r );
  for ( unsigned int i = 0; i < (*this).r; i++ )
    for ( unsigned int j = 0; j < (*this).c; j++ )
      r( i ) += (*this)( i, j ) * rhs( j ); return r; }
const mat<T>& operator+=( mat<T>& rhs ){
  for ( unsigned int i = 0; i < r; i++ )
    for ( unsigned int j = 0; j < c; j++ )
      (*this)( i, j ) += rhs( i, j ); return *this; }
const mat<T>& operator-=( mat<T>& rhs ){
  for ( unsigned int i = 0; i < r; i++ )
    for ( unsigned int j = 0; j < c; j++ )
      (*this)( i, j ) -= rhs( i, j ); return *this; }
const mat<T>& operator*=( T l ){
  for ( unsigned int i = 0; i < r; i++ )
    for ( unsigned int j = 0; j < c; j++ )
      (*this)( i, j ) *= l; return *this; }
const mat<T>& operator/=( T l ){
  for ( unsigned int i = 0; i < r; i++ )
    for ( unsigned int j = 0; j < c; j++ )
      (*this)( i, j ) /= l; return *this; }
bool operator==( const mat<T>& rhs ) const {
  for ( unsigned int i = 0; i < r; i++ )
```

```
      for( unsigned int j = 0; j < c; j++ )
        if ( (*this)( i, j ) != rhs( i, j ) ) return false; return true; }
  bool operator!=( const mat<T>& rhs ) const { return !( *this == rhs ); }
  // -------------------------------------------------------------------------
  // mat various manipulation routines
  // -------------------------------------------------------------------------
  void append_identity(){ unsigned int oldCols = c; resize( r, c + r );
    for ( unsigned int i = 0; i < r; (*this)( i, oldCols + i ) = (T)1, i++ ); }
  void swap_rows( unsigned int i1, unsigned int i2 ){
    std::swap( pRows[i1], pRows[i2] ); }
  void delete_row( unsigned int Row ){
    std::vector< std::vector< T >* >::iterator i = pRows.begin();
    std::advance( i, Row ); if ( *i != NULL ){ (**i).clear(); delete (*i); }
    pRows.erase( i ); r--; }
  void scale_row( unsigned int i, T l ){
    for ( unsigned int j = 0; j < c; (*this)( i, j ) *= l, j++ ); }
private:
  unsigned int r, c; std::vector<std::vector<T>*> pRows;
};
template <class T>
std::ostream& operator<<( std::ostream& str, mat<T> rhs ){ unsigned int i,j;
  if ( rhs.num_cols() == 0 ) return str;
  for ( i = 0; i < rhs.num_rows(); str << std::endl, i++ ){
    for ( j = 0; j < rhs.num_cols() - 1; str << " ", j++ )
      str << std::setw( 0 ) << rhs( i, j );
    str << std::setw( 0 ) << rhs( i, j );
  }
  return str;
}
#endif // ndef MAT_INCLUDED
```

Contents of file `efficient\basics\vec.h`

```
// -------------------------------------------------------------------------
// vec.h: class template. fdi - 24oct05
// -------------------------------------------------------------------------
#ifndef VEC_INCLUDED
#define VEC_INCLUDED
#pragma warning ( disable : 4786 )                        // ms specific
#include "basics.h"
#include <vector>
#include <iostream>
#include <iomanip>
template <class T> class vec{ public:
```

```
  void resize( unsigned int n ){ v.resize( n, (T)0 ); }
  unsigned int size() const { return v.size(); }
  void delete_component( unsigned int i ){ delete_element( v, i ); }
  void kill(){ for ( unsigned int i = 0; i < v.size(); v[i] = (T)0, i++ ); }
  T min(){ T r = v[0];
    for ( unsigned int i = 0; i < v.size(); r = minimum( r, v[i] ), i++ );
    return r; }
  T max(){ T r = v[0];
    for ( unsigned int i = 0; i < v.size(); r = maximum( r, v[i] ), i++ );
    return r; }
  T& operator()( unsigned int i ){ return v[i]; }
  vec<T> operator-() const { vec<T> r = *this; unsigned int i;
    for ( i = 0; i < r.size(); r.v[i] = -r.v[i], i++ ); return r; }
  vec<T> operator+( const vec<T>& rhs ) const { vec<T> r = *this;
    for ( unsigned int i = 0; i < r.size(); r.v[i] += rhs.v[i], i++ );
    return r; }
  vec<T> operator-( const vec<T>& rhs ) const { vec<T> r = *this;
    for ( unsigned int i = 0; i < r.size(); rhs.v[i] -= rhs.v[i], i++ );
    return r; }
  T operator*( const vec<T>& rhs ) const { T r = 0;       // euclidean product
    for ( unsigned int i = 0 ; i < v.size(); r += v[i] * rhs.v[i], i++ );
    return r; }
  const vec<T>& operator+=( const vec<T>& rhs ){
    for ( unsigned int i = 0; i < v.size(); v[i] += rhs.v[i], i++ );
    return *this; }
  const vec<T>& operator-=( const vec<T>& rhs ){
    for ( unsigned int i = 0; i < v.size(); v[i] -= rhs.v[i], i++ );
    return *this; }
  const vec<T>& operator*=( T l ){                        // scalar mult
    for ( unsigned int i = 0; i < v.size(); v[i] *= l, i++ ); return *this; }
  bool operator==( const vec<T>& rhs ) const {
    for ( unsigned int i = 0; i < v.size(); i++ )
      if ( v[i] != rhs.v[i] ) return false; return true; }
  bool operator!=( const vec<T>& rhs ) const { return !( *this == rhs ); }
private: std::vector<T> v;
};
template <class T> vec<T> operator*( T l, const vec<T>& rhs ){ vec<T> r = rhs;
  for ( unsigned int i = 0; i < r.size(); r(i) *= l, i++ ); return r; }
template <class T> std::ostream& operator<<( std::ostream& str, vec<T> rhs ){
  unsigned int i;
  for ( i = 0; i < rhs.size() - 1; str << std::setw( 0 )
                                       << rhs( i ) << " ", i++ );
```

```
  str << std::setw( 0 ) << rhs( i ) << std::endl; return str; }
#endif // ndef VEC_INCLUDED
```

Contents of file efficient\knapsack\knap_item.h

```
// ------------------------------------------------------------------------
// knap_item.h: class template. fdi - 22jul05
// ------------------------------------------------------------------------
#ifndef KNAP_ITEM_INCLUDED
#define KNAP_ITEM_INCLUDED
#pragma warning ( disable : 4786 )                          // ms specific
template <class T1, class T2> class knap_item{ public:
  knap_item(){ p = (T1)0; a = (T2)0; };
  knap_item( T1 p, T2 a ){ this->p = p; this->a = a; };
  virtual ~knap_item(){}; T1 p; T2 a;
};
#endif // ndef KNAP_ITEM_INCLUDED
```

Contents of file efficient\knapsack\knapsack.h

```
// ------------------------------------------------------------------------
// knapsack.h: function templates. fdi - 21jul05
// ------------------------------------------------------------------------
#ifndef KNAPSACK_INCLUDED
#define KNAPSACK_INCLUDED
#pragma warning ( disable : 4786 )                          // ms specific
#include "../basics/basics.h"
#include "../knapsack/knap_item.h"
#include <math.h>
#include <vector>
// ------------------------------------------------------------------------
// class modelling items in which more information is represented
// ------------------------------------------------------------------------
template <class T1, class T2, class T3> class knap_int{ public:
  knap_int(){ p = (T1)0; a = (T2)0; q = 0; r = (T3)0; i = 0; m = 0; };
  knap_int( T1 p, T2 a, unsigned int q, T3 r, unsigned int i, unsigned int m ){
    this->p = p; this->a = a; this->q = q; this->r = r; this->i = i;
    this->m = m; };
  virtual ~knap_int(){}; T1 p; T2 a; unsigned int q; T3 r; unsigned int i, m;
};
// ------------------------------------------------------------------------
// class modelling entries of the data structure for solver
// ------------------------------------------------------------------------
template <class T1, class T2, class T3> class entry{
```

```
public:
  entry(){ a = T2(0); parent_a = T2(0); q = 0; parent_q = 0; i = 0; m = 0; };
  virtual ~entry(){}; T2 a, parent_a; unsigned int q, parent_q, i, m;
};
template <class T1, class T2, class T3>
bool dom( entry<T1, T2, T3>& e1, entry<T1, T2, T3>& e2 ){ // evaluate dominance
  return ( e1.q >= e2.q ) && ( e1.a <= e2.a ); }          // relation
template <class T1, class T2, class T3>
bool big( entry<T1, T2, T3>& e1, entry<T1, T2, T3>& e2 ){ // evaluate order
  return ( e1.q > e2.q ) && ( e1.a > e2.a ); }            // relation
template <class T1, class T2, class T3> std::vector< knap_int<T1, T2, T3> >
knap_preprocessor ( std::vector< knap_item<T1, T2> > in, T2 b ){
  std::vector< knap_int<T1, T2, T3> > res;
  for ( unsigned int i = 0; i < in.size(); i++ )          // iterate items
    if ( ( in[i].p > (T1)0 ) && ( in[i].a <= b ) )        // item nontrivial
      res.push_back( knap_int<T1, T2, T3>                 // store item
        ( in[i].p, in[i].a, 0, (T3)in[i].p / (T3)in[i].a, i, 1 ) );
  return res;
}
template <class T1, class T2, class T3>                  // evaluate phi
T1 calc_phi( knap_int<T1, T2, T3>& item, T2 b ){
  if ( item.m != 0 ) return item.p * (T1)floor( (T3)b / (T3)item.a );
  else return (T1)0; }
template <class T1, class T2, class T3>
T1 calc_p_zero( std::vector< knap_int<T1, T2, T3> > in, T2 b ){
  T1 p_max = in[0].p; knap_int<T1, T2, T3> max_item = in[0];
  for ( unsigned int i = 0; i < in.size(); i++ ){         // update maximum
    p_max = maximum( p_max, in[i].p );                    // profit and most
    if ( in[i].r > max_item.r ) max_item = in[i];         // efficient item
  }
  return maximum( p_max, calc_phi( max_item, b ) );
}
template <class T1, class T2, class T3>
void separate( std::vector< knap_int<T1, T2, T3> >& in,
               std::vector< knap_int<T1, T2, T3> >& large,
               knap_int<T1, T2, T3>& best_small_item, T3 t ){
  large.clear(); best_small_item = knap_int<T1, T2, T3>   // init best small
  ( (T1)0, (T2)0, T3(0), (T3)0, 0, 0 );                   // item
  for ( unsigned int i = 0; i < in. size(); i++ )         // iterate items
    if ( (T3)in[i].p > t ) large.push_back( in[i] );      // store large item
    else if ( in[i].r > best_small_item.r )               // store small item
           best_small_item = in[i];                       // if it is better
```

```
}
template <class T1, class T2, class T3> unsigned int get_k( T1 p, T3 t ){
  return (unsigned int)ceil( log( (T3)p / t ) / log( 2.0f ) ) - 1; }
template <class T1, class T2, class T3>
void scale_profits( std::vector< knap_int< T1, T2, T3> >& in, T3 t, T3 k ){
  for ( unsigned int i = 0; i < in.size(); i++ ){         // see page 348
    unsigned int int_k = get_k<T1, T2, T3>( in[i].p, t ); // section 8
    unsigned int power = (unsigned int)pow( (double)2.0f, (int)int_k );
    in[i].q = (unsigned int)floor( (T3)in[i].p / ( (T3)power * k ) ) * power;
  }
}
template <class T1, class T2, class T3>
void insert( knap_int<T1, T2, T3> item,
             std::vector< knap_int<T1, T2, T3> >& in, T2 b ){
  do {
    if ( in[item.q].m == 0 ) in[item.q] = item;           // slot is free
    else if ( item.a < in[item.q].a ) in[item.q] = item;  // item is better
    item.p += item.p; item.a += item.a;                   // provide enough
    item.q += item.q; item.m += item.m;                   // multiples
  } while ( item.a <= b );
}
template<class T1, class T2, class T3> std::vector< entry<T1, T2, T3> >
modified_merge( std::vector< entry<T1, T2, T3> > a,       // modified mergesort
                std::vector< entry<T1, T2, T3> > b ){     // discarding
  std::vector< entry<T1, T2, T3> > r;                     // dominated entries
  std::vector< entry<T1, T2, T3> >::iterator a_it = a.begin();
  std::vector< entry<T1, T2, T3> >::iterator b_it = b.begin();
  while ( ( a_it != a.end() ) && ( b_it != b.end() ) ){
    if ( big( *a_it, *b_it ) ){ r.push_back( *a_it ); a_it++; continue; }
    if ( big( *b_it, *a_it ) ){ r.push_back( *b_it ); b_it++; continue; }
    if ( dom( *a_it, *b_it ) ){ b_it++; continue; }
    if ( dom( *b_it, *a_it ) ){ a_it++; continue; }
  }
  while ( a_it != a.end() ){ r.push_back( *a_it ); a_it++; }
  while ( b_it != b.end() ){ r.push_back( *b_it ); b_it++; }
  return r;
}
template<class T1, class T2, class T3>                    // perform one
void iterate( std::vector< entry<T1, T2, T3> >& ent_lst,  // iteration
              knap_int<T1, T2, T3> item, T2 b ){
  std::vector< entry<T1, T2, T3> > new_lst;
  std::vector< entry<T1, T2, T3> >::iterator i;
```

```
  for ( i = ent_lst.begin(); i != ent_lst.end(); i++ ){
    entry<T1, T2, T3> e = *i;                              // build new entry
    e.parent_q = e.q; e.q += item.q; e.i = item.i;         // from original
    e.parent_a = e.a; e.a += item.a; e.m = item.m;         // entry
    if ( e.a <= b ) new_lst.push_back( e );                // check validity
  }                                                        // of new entry
  ent_lst = modified_merge( ent_lst, new_lst );            // merge lists
}
template<class T1, class T2, class T3>
void optimize( std::vector< entry< T1, T2, T3> >& in,
               std::vector< knap_int<T1, T2, T3> >& large, T2 b ){
  std::vector< knap_int<T1, T2, T3> >::iterator it;
  for ( it = large.begin(); it != large.end(); it++ ) iterate( in, *it, b );
}
template<class T1, class T2, class T3>
unsigned int calc_best_index( std::vector< entry< T1, T2, T3 > >& in,
                              knap_int<T1, T2, T3>& item, T2 b, T3 k ){
  unsigned int r = 0;
  T3 p = (T3)in[0].q * k + (T3)calc_phi( item, b - in[0].a );
  for ( unsigned int i = 0; i < in.size(); i++ )
    if ( (T3)in[i].q * k + (T3)calc_phi( item, b - in[i].a ) > p ){
      p = (T3)in[i].q  * k + (T3)calc_phi( item, b - in[i].a ); r = i; }
  return r;
}
template<class T1, class T2, class T3>
void back_track( std::vector< entry< T1, T2, T3 > >& in, unsigned int i,
                 std::vector< unsigned int >&s ){
  for ( entry<T1, T2, T3> e = in[i]; i != in.size(); i++ )
    if ( dom( in[i], e ) ){                                // take for solution
      e = in[i]; s[e.i] += e.m;                            // copy and provide
      e.q = e.parent_q; e.a = e.parent_a;                  // decode parent
    }
}
template <class T1, class T2, class T3> std::vector<unsigned int>
knapsack( std::vector< knap_item<T1, T2> > in, T2 b, T3 eps ){
  std::vector< knap_int<T1, T2, T3> >                      // generate internal
  clean_list = knap_preprocessor<T1, T2, T3>( in, b );     // list
  T1 p_zero = calc_p_zero<T1, T2, T3>( clean_list , b );   // two approximation
  T3 t = (T3)0.5f * eps * (T3)p_zero;                      // threshold value
  T3 k = (T3)0.5f * eps * t;                               // scaling factor
  std::vector< knap_int<T1, T2, T3> > large;               // large items
  knap_int<T1, T2, T3> best_small_item;
```

```
  separate<T1, T2, T3> ( clean_list, large, best_small_item, t );
  scale_profits( large, t, k );
  unsigned int space_size = (unsigned int)floor( 8.0f / ( eps * eps ) );
  std::vector< knap_int<T1, T2, T3> > large_space;
  large_space.resize( space_size );
  for ( unsigned int i = 0; i < large.size(); i++ )       // generate
    insert( large[i], large_space, b );                   // needed multiples
  std::vector< knap_int<T1, T2, T3> > large_needed;
  std::vector< knap_int<T1, T2, T3> >::reverse_iterator it;
  for ( it = large_space.rbegin(); it != large_space.rend(); it++ )
    if ( it->m != 0 ) large_needed.push_back( *it );
  std::vector< entry<T1, T2, T3> > entry_list;            // prepare list
  entry_list.push_back( entry<T1, T2, T3>() );
  optimize( entry_list, large_needed, b );
  unsigned int best_i = calc_best_index( entry_list, best_small_item, b, k );
  std::vector< unsigned int > solution;
  solution.resize( in.size(), 0 );
  back_track( entry_list, best_i, solution );             // backtrack
  if ( best_small_item.m != 0 ){                          // check for small
    T2 rest_b = b - entry_list[best_i].a;                 // item
    unsigned int i = (unsigned int)                       // calc multiplicity
                     floor( rest_b / best_small_item.a ); // of small item
    solution[best_small_item.i] += i * best_small_item.m; // add small items
  }
  return solution;
}
#endif // ndef KNAPSACK_INCLUDED
```

Contents of file `efficient\max_min\max_min.h`

```
// ------------------------------------------------------------------------------
// max_min.h: function templates. fdi - 14mar06
// ------------------------------------------------------------------------------
#ifndef MAX_MIN_INCLUDED
#define MAX_MIN_INCLUDED
#pragma warning ( disable : 4786 )                        // ms specific
#include "../basics/basics.h"
#include "../basics/vec.h"
#include <math.h>
template <class T>
T phi( vec<T>& fx, T t, T theta ){                        // evaluate the
  T r = (T)0;                                             // potential
  for ( unsigned int m = 0; m < fx.size(); m++ ){         // function
    r += log( fx( m ) - theta );
```

```
  }
  r *= t / (T)fx.size();
  r += log( theta );
  return r;
}
template <class T>
T firstorder( vec<T>& fx, T t, T theta ){                // evaluate the
  T r = (T)0;                                            // first order
  for ( unsigned int m = 0; m < fx.size(); m++ ){        // optimality
    r += (T)1 / ( fx( m ) - theta );                     // condition
  }
  return ( ( t * theta ) / (T)fx.size() ) * r;
}
template <class T>
T thetabin( vec<T>& fx, T t, T tol ){                    // initialize lower
  T lambda = fx.min();                                   // and upper bound
  T lb = lambda / ( (T)1 + t );                          // from lemma
  T ub = lambda / ( (T)1 + t / (T)fx.size() );
  do {                                                   // iterate binary
    T pos = ( lb + ub ) / (T)2;                          // search until
    T eval_pos = firstorder( fx, t, pos );               // bounds within
    if ( eval_pos < (T)1 ) {                             // tolerance
      lb = pos;
    } else {
      ub = pos;
    }
  } while ( ub - lb > (T)2 * tol );
  return ( lb + ub ) / (T)2;
}
template <class T>
T phired( vec<T>& fx, T t, T tol ){                      // approximate theta
  return phi( fx, t, thetabin( fx, t, tol ) );
}
template <class T>
vec<T> pricevec( vec<T>& fx, T t, T theta ){
  vec<T> p;
  p.resize( fx.size() );                                 // init result
  for ( unsigned int m = 0; m < fx.size(); m++ ){        // calculate result
    p( m ) = theta / ( fx( m ) - theta );                // entries
  }
  p *= ( t / (T)fx.size() );
  return p;
```

```
}
template <class T>
T nufixed( vec<T>& fx, vec<T>& fxhat, vec<T>& p ){
  T ptfx = p * fx;                                        // generate scalar
  T ptfxhat = p * fxhat;                                  // products
  return ( ptfxhat - ptfx ) / ( ptfxhat + ptfx );
}
template <class T>
T taufixed( vec<T>& fx, vec<T>& fxhat, vec<T>& p, T nu, T t, T theta ){
  T d = (T)2 * (T)fx.size() * ( ( p * fxhat ) + ( p * fx ) );
  return ( t * theta * nu ) / d;
}
template <class T>
T simplepot( vec<T>& fx, vec<T>& fxhat, T theta, T tau ){
  vec<T> fxprime = ( ( (T)1 - tau ) * fx ) + ( tau * fxhat );
  unsigned int m;
  for ( m = 0; m < fxprime.size(); m++ ){
    if ( theta >= fxprime( m ) ) return -HUGE_VAL;
  }
  T r = (T)0;
  for ( m = 0; m < fxprime.size(); m++ ){
    r += log( fxprime( m ) - theta );
  }
  return r;
}
template <class T>
T tauline( vec<T>& fx, vec<T>& fxhat, T theta, T tol ){
  T lb = (T)0;                                            // intialize lower
  T ub = (T)1;                                            // and upper bound
  do {                                                    // loop finds the
    T step = ( ub - lb ) / (T)3;                          // maximum of the
    T posl = lb + step;                                   // unimodal function
    T posr = ub - step;
    T evalposl = simplepot( fx, fxhat, theta, posl );
    T evalposr = simplepot( fx, fxhat, theta, posr );
    if ( evalposl < evalposr ) {                          // reduce interval
      lb = posl;                                          // length by one
    } else {                                              // third
      ub = posr;
    }
  } while ( ub - lb > (T)2 * tol );                       // iterate close
  return ( lb + ub ) / (T)2;                              // enough
```

```
}
template <class T>
vec<T> initsol( vec<T> ( *f )( vec<T> x ),
                vec<T> ( *blocksolver )( vec<T> p, T eps ),
                T eps, unsigned int M, unsigned int N ){
  vec<T> x;
  x.resize( N );
  for ( unsigned int i = 0 ; i < M ;i++ ){              // generate price
    vec<T> p;                                           // vector
    p.resize( M );
    p( i ) = (T)1;
    vec<T> xhat = blocksolver( p, (T)1 / (T)2 );        // get block sol
    x.resize( maximum( x.size(), xhat.size() ) );       // col generation
    x += xhat;
  }
  x *= (T)1 / (T)M; return x;                           // normalize
}
template <class T>
unsigned int imp( vec<T>& x, vec<T> ( *f )( vec<T> x ),
                  vec<T> ( *blocksolver )( vec<T> p, T eps ),
                  T eps, T tol, T omega ){
  unsigned int i = 0;
  vec<T> fx = f( x );                                   // calc lambda
  T lambda = fx.min();                                  // of last iterate
  T t = eps / (T)6;                                     // step 1
  while ( true ) { i++;                                 // step 2
    fx = f( x );                                        // step 2.1
    T theta = thetabin( fx, t, tol );
    vec<T> p = pricevec( fx, t, theta );
    vec<T> xhat = blocksolver( p, t );                  // step 2.2
    x.resize( xhat.size() );                            // if col generation
    vec<T> fxhat = f( xhat );
    T nu = nufixed( fx, fxhat, p );                     // step 2.3
    if ( nu <= t ) {                                    // step 2.4
      break;
    }
    T tau1 = taufixed( fx, fxhat, p, nu, t, theta );    // get step lengths
    T tau2 = tauline( fx, fxhat, theta, tol );
    vec<T> xprime1  = ( ( (T)1 - tau1 ) * x ) + ( tau1 * xhat );
    vec<T> fxprime1 = f( xprime1 );
    T pot1          = phired( fxprime1, t, tol );
    vec<T> xprime2  = ( ( (T)1 - tau2 ) * x ) + ( tau2 * xhat );
```

```
    vec<T> fxprime2 = f( xprime2 );
    T pot2          = phired( fxprime2, t, tol );
    if ( pot1 > pot2 ) {
      x = xprime1;
    } else {
      x = xprime2;
    }
    if ( omega != (T)0 ) {                                  // check for
      if ( f( x ).min() >=  omega * lambda ) {              // and apply second
        break;                                              // stopping rule
      }
    }
  }
  return i;                                                 // return number of
}                                                           // iterations
template <class T>
unsigned int impscal( vec<T>& x, vec<T> ( *f )( vec<T> x ),
                      vec<T> ( *blocksolver )( vec<T> p, T eps ),
                      T eps, T tol, unsigned int M ){
  unsigned int i = 0;
  T epss = (T)1 / (T)2;
  T omegas = ( (T)1 - epss / (T)2 ) * (T)2 * (T)M;          // for stopping rule
  do {                                                      // scaling phase
    epss /= (T)2;                                           // coordination
    i += imp( x, f, blocksolver, epss, tol, omegas );       // phase
    omegas = ( (T)1 - epss / (T)2 ) / ( (T)1 - epss );      // setup next omegas
  } while ( epss > eps );
  return i;                                                 // return number of
}                                                           // iterations
#endif // ndef MAX_MIN_INCLUDED
```

Contents of file `efficient\strip\rect_item.h`

```
// ------------------------------------------------------------------------
// rect_item.h: class template. fdi - 21jul05
// ------------------------------------------------------------------------
#ifndef RECT_ITEM_INCLUDED
#define RECT_ITEM_INCLUDED
#pragma warning ( disable : 4786 )                          // ms specific
template <class T> class rect_item{
public:
  rect_item( T w, T h ){ this->w = w; this->h = h; };
  rect_item(){ w = (T)0; h = (T)0; }; virtual ~rect_item(){}; T w, h;
};
```

```
#endif // ndef RECT_ITEM_INCLUDED
```

Contents of file `efficient\strip\strippack.h`

```
// ----------------------------------------------------------------------------
// strippack.h: function templates. fdi - 05nov06
// ----------------------------------------------------------------------------
#ifndef STRIPPACK_INCLUDED
#define STRIPPACK_INCLUDED
#pragma warning ( disable : 4786 )                          // ms specific
#include "../basics/basics.h"
#include "../basics/vec.h"
#include "../basics/mat.h"
#include "../knapsack/knap_item.h"
#include "../knapsack/knapsack.h"
#include "../max_min/max_min.h"
#include "rect_item.h"
#include <math.h>
#include <vector>
#include <algorithm>
// ----------------------------------------------------------------------------
// class modelling items in which more information is represented
// ----------------------------------------------------------------------------
template <class T> class rect_int{ public: T w, h; unsigned int i;
  rect_int(){ w = (T)0; h = (T)0; i = 0;}; virtual ~rect_int(){};
  rect_int( T w, T h, int i ){ this->w = w; this->h = h; this->i = i; };
};
// ----------------------------------------------------------------------------
// predicates for rect_int
// ----------------------------------------------------------------------------
template <class T>
bool higher( const rect_int<T>& lhs, const rect_int<T>& rhs ){
  return lhs.h > rhs.h; }
template <class T>
bool wider( const rect_int<T>& lhs, const rect_int<T>& rhs ){
  return lhs.w > rhs.w; }
// ----------------------------------------------------------------------------
// class wrapping the dependencies for function vector and block solver
// ----------------------------------------------------------------------------
template <class T> class wrap{ public: wrap(){}; virtual ~wrap(){};
  static vec<T> beta; static vec<T> w_prime; static mat<T> a;
  static std::vector< std::vector< rect_int<T> > > classes;
  static std::vector< std::vector<unsigned int> > configs;
  static unsigned int num_old_configs;
```

```
  static unsigned int num_new_configs;
  static std::vector<unsigned int> config_count;
  static vec<T> f( vec<T> x ){ vec<T> fx = a * x;
    for ( unsigned int i = 0; i < fx.size(); fx( i ) /= beta( i ), i++ );
    return fx;
  }
  static vec<T> blocksolver( vec<T> p, T eps ){ unsigned int j;
    std::vector< knap_item<T, T> > in; vec<T> xhat;       // create knap inst
    for ( j = 0; j < beta.size(); j++ )
      in.push_back( knap_item<T, T>( p( j ), w_prime( j ) ) );
    std::vector<unsigned int>
    cfg = knapsack<T, T, T>( in, (T)1, eps );             // create config
    for ( j = 0; j < configs.size(); j++ )                // iterate configs
      if ( configs[j] == cfg ){                           // configuration has
        xhat.resize( configs.size() ); xhat( j ) = (T)1;  // been taken before
        config_count[ j ]++;
        num_old_configs++;
        return xhat;                                      // return vertex
      }
    num_new_configs++;
    config_count.push_back( 1 );
    configs.push_back( cfg ); j = configs.size() - 1;     // store new config
    a.resize( a.num_rows(), configs.size() );             // adapt matrix
    for ( unsigned int i = 0; i < cfg.size(); i++ )       // generate new
      a( i, j ) = (T)cfg[i];                              // column
    xhat.resize( configs.size() ); xhat( j ) =(T)1;       // create and
    return xhat;                                          // return vertex
  }
};
unsigned int wrap<float>::num_old_configs;
unsigned int wrap<float>::num_new_configs;
std::vector<unsigned int> wrap<float>::config_count;
vec<float> wrap<float>::beta;
vec<float> wrap<float>::w_prime;
mat<float> wrap<float>::a;
std::vector< std::vector< rect_int<float> > > wrap<float>::classes;
std::vector< std::vector<unsigned int> > wrap<float>::configs;

unsigned int wrap<double>::num_old_configs;
unsigned int wrap<double>::num_new_configs;
std::vector<unsigned int> wrap<double>::config_count;
vec<double> wrap<double>::beta;
```

```
vec<double> wrap<double>::w_prime;
mat<double> wrap<double>::a;
std::vector< std::vector< rect_int<double> > > wrap<double>::classes;
std::vector< std::vector<unsigned int> > wrap<double>::configs;

unsigned int wrap<long double>::num_old_configs;
unsigned int wrap<long double>::num_new_configs;
std::vector<unsigned int> wrap<long double>::config_count;
vec<long double> wrap<long double>::beta;
vec<long double> wrap<long double>::w_prime;
mat<long double> wrap<long double>::a;
std::vector< std::vector< rect_int<long double> > > wrap<long double>::classes;
std::vector< std::vector<unsigned int> > wrap<long double>::configs;
template <class T>
void strip_separate( std::vector< rect_int<T> >& in,
                     std::vector< rect_int<T> >& wide,
                     std::vector< rect_int<T> >& narrow, T eps ){
  T eps_prime = eps / ( (T)2 + eps ); wide.clear(); narrow.clear();
  for ( unsigned int i = 0; i < in.size(); i++ ){
    if ( in[i].w < eps_prime )
      narrow.push_back( rect_int<T>( in[i].w, in[i].h, i ) );
    else
      wide.push_back( rect_int<T>( in[i].w, in[i].h, i ) );
  }
}
template <class T> T calc_total_height( std::vector< rect_int<T> >& in ){
  T res = (T)0; for ( unsigned int i = 0; i < in.size(); res += in[i].h, i++ );
  return res;
}
template <class T> std::vector< std::vector< rect_int<T> > >
classify( std::vector< rect_int<T> >& in, T eps ){
  std::vector< std::vector< rect_int< T > > > res;
  T eps_prime = eps / ( (T)2 + eps ); T total_height = (T)0;
  T increment = calc_total_height( in ) / ( (T)1 / ( eps_prime * eps_prime ) );
  std::sort( in.begin(), in.end(), wider<T> );
  for ( unsigned int i = 0; i < in.size(); i++ ){         // iterate items
    total_height += in[i].h;                              // increment height
    if ( total_height > (T)( res.size() ) * increment )   // check for class
      res.push_back( std::vector< rect_int<T> >() );      // overflow
    res.back().push_back( in[i] );                        // add rectangle to
  }                                                       // last class
  return res;
```

```
}
template <class T> std::vector< rect_int<T> >
strip_preprocessor( std::vector< rect_item<T> > in ){
  std::vector< rect_int<T> > res;
  for ( unsigned int i = 0; i < in.size(); i++ )          // iterate and store
    if ( ( in[i].w > (T)0 ) && ( in[i].h > (T)0 ) )       // nontrivial items
      res.push_back( rect_int<T>( in[i].w, in[i].h, i ) );
  return res;
}
template <class T>
vec<T> calc_beta( std::vector< std::vector< rect_int<T> > >& in ){
  vec<T> res; res.resize( in.size() );
  for ( unsigned int i = 0; i < in.size(); i++ )
    res( i ) = calc_total_height( in[i] );
  return res;
}
template <class T>
vec<T> calc_w_prime( std::vector< std::vector< rect_int<T> > >& in ){
  vec<T> res; res.resize( in.size() );
  for ( unsigned int i = 0; i < in.size(); i++ )
    res( i ) = in[i][0].w;
  return res;
}
template <class T>
vec<T> strippack( std::vector< rect_item<T> > in, T eps, T tol,
                  unsigned int& iter,
                  unsigned int& old_sol, unsigned int& new_sol ){
  // T factor = (T)242 / (T)100; eps /= factor;           // to discuss
  wrap<T>::num_old_configs = 0;
  wrap<T>::num_new_configs = 0;
  std::vector< rect_int<T> >                              // generate internal
  clean_list = strip_preprocessor<T>( in );               // list
  std::vector< rect_int<T> > wide;                        // generate lists for
  std::vector< rect_int<T> > narrow;                      // partition
  strip_separate( clean_list, wide, narrow, eps );
  wider<T>( rect_int<T>(), rect_int<T>() );               // force template
  higher<T>( rect_int<T>(), rect_int<T>() );              // instantiation
  wrap<T>::classes.clear();                               // clear wrapped
  wrap<T>::configs.clear();                               // objects
  wrap<T>::config_count.clear();
  wrap<T>::beta.resize( 0 );
  wrap<T>::w_prime.resize( 0 );
```

```
  wrap<T>::a.resize( 0, 0 );
  wrap<T>::classes = classify( wide, eps );               // gen classes
  wrap<T>::beta = calc_beta( wrap<T>::classes );          // gen lp rhs
  wrap<T>::w_prime = calc_w_prime( wrap<T>::classes );    // gen rounded widths
  wrap<T>::a.resize( wrap<T>::classes.size(), 0 );        // gen matrix
  vec<T> res  = initsol<T>
  ( wrap<T>::f, wrap<T>::blocksolver, eps, wrap<T>::classes.size(), 0 );
  iter = impscal<T>
  ( res, wrap<T>::f, wrap<T>::blocksolver, eps, tol, wrap<T>::classes.size() );
  old_sol = wrap<T>::num_old_configs;
  new_sol = wrap<T>::num_new_configs;

  std::sort( wrap<T>::config_count.rbegin(), wrap<T>::config_count.rend() );

  for ( unsigned int i = 0; i < wrap<T>::config_count.size(); i++ )
    std::cout << i+1 << " " << wrap<T>::config_count[i] << std::endl;
  std::cout << std::endl;
  // std::cout << wrap<T>::config_count.size() << std::endl;
  wrap<T>::config_count.clear();
  return res;
}
#endif // ndef STRIPPACK_INCLUDED
```

Contents of file `strippacker\strippacker.cpp`

```
// strippacker.cpp : Definiert den Einsprungpunkt fr die Konsolenanwendung.

#include <iostream>
#include <iomanip>
#include <vector>
#include <fstream>

#include <stdio.h>
#include <float.h>
#include <time.h>

#include "../efficient/strip/rect_item.h"
#include "../efficient/strip/strippack.h"

template <class T>
std::vector< rect_item<T> > generate_instance( unsigned int n ){
  std::vector< rect_item<T > > r;
  for ( unsigned int i = 0; i < n; i++ )
    r.push_back( rect_item<T>( (T)rand() / (T)(RAND_MAX) ,
```

```
                              (T)rand() / (T)(RAND_MAX) ) );
  return r;
}
int main( int argc, char* argv[] ){
  srand( 1 );
  std::cout << std::setiosflags( std::ios::fixed )
            << std::setiosflags( std::ios::internal )
            << std::setiosflags( std::ios::showpos )
            << std::setfill( '0' )
            << std::setprecision( 10 );
  long double tol = DBL_EPSILON;                    // tolerance for epsilon
  unsigned int iter;                                // number of iterations
  unsigned int old_sol;                             // number of old solutions
  unsigned int new_sol;                             // number of new solutions

  unsigned int instance_size = 1000;        // number of items in an instance
  unsigned int num_instances = 10;          // number of instances to average
  unsigned int num_steps = 50;              // number of steps from ub to lb
  long double lb = 0.2f;                    // lower bound for epsilon
  long double ub = 0.9f;                    // upper bound for epsilon
  long double step_width =                  // step width for epsilon
    ( ub - lb ) / (long double)num_steps;

  std::vector< std::vector< rect_item<long double> > > instances;
  instances.resize( num_instances );

  unsigned int i, j, k;

  // generate and save random instances

  for ( j = 0; j < num_instances; j++ ){
    instances[j] = generate_instance<long double>( instance_size );
    char filename[255];
    sprintf( filename,
             "instances/inst_%05i_%02i_ins.txt\0", instance_size, j );
    std::ofstream outfile;
    outfile.open( filename, std::ios::out );
    outfile << std::setiosflags( std::ios::fixed )
            << std::setiosflags( std::ios::internal )
            << std::setiosflags( std::ios::showpos )
            << std::setfill( '0' )
            << std::setprecision( 2 );
```

```
  for ( k = 0; k < instance_size; k++ ){
    outfile << instances[j][k].w  << " " << instances[j][k].h << std::endl;
  }
  outfile.close();
}

// solve and average random instances

std::vector< std::ofstream* > separate_solutions;
separate_solutions.resize( num_instances );
for ( j = 0; j < num_instances; j++ ){
  char filename[255];
  sprintf( filename,
           "instances/inst_%05i_%02i_sol.txt\0", instance_size, j );
  separate_solutions[j] = new std::ofstream();
  separate_solutions[j]->open( filename, std::ios::out );
  ( *separate_solutions[j] ) << std::setiosflags( std::ios::fixed )
                             << std::setiosflags( std::ios::internal )
                             << std::setiosflags( std::ios::showpos )
                             << std::setfill( '0' )
                             << std::setprecision( 2 );
}

for ( i = 0; i < num_steps; i++ ){
  unsigned int iter_avg = 0;
  double time_avg = 0.0f;
  double old_percentage_avg = 0.0f;
  long double eps = ub - (long double)i * step_width;
  for ( j = 0; j < num_instances; j++ ){
    time_t time_one, time_two, time_diff;
    unsigned int iter;
    unsigned int old_sol;
    unsigned int new_sol;
    double old_percentage = 0.0f;
    time( &time_one );
    vec<long double> solution =
    strippack( instances[j], eps, tol, iter, old_sol, new_sol );
    time( &time_two );
    time_diff = (time_t)difftime( time_two, time_one );
    (*separate_solutions[j]) << iter << " "
                             << eps << " " << time_diff << std::endl;
    old_percentage = (double)old_sol / ( (double)(old_sol + new_sol) );
```

```
      old_percentage_avg += old_percentage;
      iter_avg += iter;
      time_avg += time_diff;
    }
    iter_avg /= num_instances;
    time_avg /= (double)num_instances;
    old_percentage_avg /= (double)num_instances;
    std::cout << "epsilon: " << eps << std::endl;
    std::cout << "average number of iterations: " << iter_avg << std::endl;
    std::cout << "average time to solve one instance: "
              << time_avg << " seconds" <<std::endl;
    std::cout << "average percentage of old configurations: "
              << old_percentage_avg << std::endl;
  }

  for ( j = 0; j < num_instances; j++ ){
    separate_solutions[j]->close();
    delete separate_solutions[j];
  }

return 0;
}
```