On Euler Tours in Streaming Models and some Games on Graphs

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Christian-Albrechts-Universität zu Kiel

vorgelegt von
Jan Schiemann
Kiel, 2021

Erster Gutachter:	Prof. Dr. Anand Srivastav
Zweiter Gutachter:	Prof. Dr. Ulrich Meyer
Tag der mündlichen Prüfung:	14.09.2021

Contents

1 Introduction **4**
1.1 Euler Tours in the Graph Streaming Model . . . 4
1.2 Euler Tours in the StrSort Model . . . 4
1.3 The Bridge-Burning Cops and Robbers Game . . . 5
1.4 Swap Equilibrium Graphs in the Extreme Vertex Destruction Model . . . 6

2 Euler Tours in the Graph Streaming Model **7**
2.1 Introduction . . . 7
2.1.1 Euler tours in the graph-streaming model . . . 7
2.1.2 Previous work . . . 7
2.1.3 Our contribution . . . 8
2.1.4 Organization of the article . . . 9
2.2 Preliminaries . . . 9
2.3 Idea of the algorithm . . . 9
2.3.1 Subtour merging in unrestricted RAM . . . 9
2.3.2 Subtour merging in limited RAM . . . 10
2.3.3 High level description . . . 12
2.4 The Algorithm and Main Results . . . 13
2.4.1 The Algorithm in Pseudo Code . . . 13
2.4.2 Main Results and Proof Idea . . . 16
2.5 Detailed Analysis . . . 17
2.5.1 Subtour representation by equivalence classes . . . 17
2.5.2 Memory requirement . . . 19
2.5.3 Correctness . . . 20
2.6 Conclusion . . . 27
2.7 Appendix . . . 28

3 Euler Tours in the StrSort Model **30**
3.1 Introduction . . . 30
3.1.1 StrSort and W-stream models . . . 30
3.1.2 Previous results . . . 31
3.1.3 Our contribution . . . 31
3.2 Preliminaries . . . 32

3.3 General idea of EulerStr . . . 32
3.3.1 Tour merging in the RAM model . . . 32
3.3.2 Tour merging within limited memory . . . 33
3.4 The W-stream step . . . 35
3.5 The PL-StrSort algorithm . . . 38
3.5.1 Rotating Tours . . . 39
3.5.2 Information edges and depth . . . 39
3.5.3 The merging step . . . 40
3.6 Conclusion . . . 45
3.7 Appendix . . . 46

4 The bridge-burning Cops and Robbers Game 50
4.1 Introduction . . . 50
4.1.1 Cops and Robbers . . . 50
4.1.2 Burning Bridges . . . 51
4.1.3 Our Contribution . . . 52
4.2 Graphs with high capture time . . . 52
4.3 Suggestions for future work . . . 65

5 Swap Equilibrium Graphs in the Extreme Vertex Destruction Model 67
5.1 Introduction . . . 67
5.1.1 Our Contribution . . . 68
5.2 Preliminaries . . . 68
5.3 A lower bound for the social cost of SE . . . 71

Deutsche Zusammenfassung

Die Arbeit befasst sich mit verschiedenen graphentheoretischen Fragestellungen aus den Gebieten des Graphstreaming Modells und der Spieltheorie.

In Kapitel 2 wird eine Methode gezeigt, in einem Eulerschen Graphen eine Eulertour zu finden, wenn nur $\mathcal{O}(n \text{ polylog}(n))$ RAM zur Verfügung steht, wobei n die Anzahl der Knoten des Graphen ist. Im Graphstreaming Modell sind die Kanten des Graphen auf einem Eingabeband gespeichert. Der Stream wird Kante für Kante in den internen Speicher eingelesen und bearbeitet. Wie in der klassischen Methode zum Finden von Eulertouren finden auch wir Kreise, die wir graduell zu immer größeren Touren verschmelzen. Das Suchen von Kreisen benötigt nur $\mathcal{O}(n \log(n))$ RAM. Dazu speichern wir pro Knoten nur eine konstante Menge an Variablen, die den letzten Kreis kennzeichnen, der den Knoten beinhaltet, und angaben zu möglichen Vorgänger- und Nachfolgerknoten. Da Die Lösung selbst nicht in den Speicher passt, schreiben wir die Eulertour sukzessive auf ein Ausgabeband. Das Ergebnis ist ein Algorithmus, der das Eingabeband nur einmal durchläuft und mit $\mathcal{O}(n \log(n))$ RAM eine Eulertour in Form einer Nachfolgerfunktion ausgibt, was in diesem Rahmen optimal ist (siehe [30]).

Im Graphstreaming Modell hat das Speichern der Lösung als Nachfolgerfunktion einen entscheidenden Nachteil. Die Kanten der Eulertour stehen auf dem Ausgabeband nicht in der Reihenfolge der Tour. Möchte man die Eulertour mit einem anderen Streaming Algorithmus weiterverarbeiten, kann dies zu Komplikationen führen. Kapitel 3 beschäftigt sich daher mit der Aufgabenstellung, eine Eulertour zu finden und die Kanten in der richtigen Reihenfolge auszugeben. Dazu benötigen wir allerdings ein Streaming Modell, in dem auch Sortierungen möglich sind. Im sogenannten *StrSort Modell* von Aggarwal et al. werden alternierend Streaming- und Sortierungsdurchläufe durchgeführt. In einem Streamingdurchlauf wird schrittweise von einem Eingabeband abgelesen und ein Ausgabeband beschrieben. Dieses Ausgabeband ist das Eingabeband des nächsten Durchlaufs. In der Sortierungsphase werden die Elemente des Ausgabebandes anhand einer globalen Halbordnung sortiert. Da dieses Modell um einiges potenter ist, wird der RAM üblicherweise auf eine logarithmische Größe begrenzt.

Wir präsentieren einen Algorithmus, der einen Hybriden aus Graphstreaming und StrSort Modell verwendet. Er besteht aus zwei Schritten.

Schritt 1 ist ein einziger Durchlauf des Streams mit $\mathcal{O}(n \log(n))$ RAM. Wie im vorigen Kapitel werden Kreise des Graphen gefunden. Implizit wird ein Baum konstruiert, wobei jeder Knoten des Baumes für einen Kreis des Graphen steht, und zwei benachbarte Knoten im Baum stets ein gemeinsamer Knoten der beiden repräsentierten Kreise bedeutet. In diesem Vorbereitungslauf erhält jede Kante ein Label, dass die Position der Kante in ihrem Kreis, und die Position des Kreises im repräsentierenden Baum anzeigt.

In Schritt 2 werden in $\mathcal{O}(\log(n))$ alternierenden Streaming- und Sortierungsdurchläufen nach und nach die Kreise zu immer größeren Touren verschmolzen und die Labels dementsprechend angepasst, bis schlussendlich die Kanten in der Reihenfolge einer Eulertour stehen.

Kapitel 4 befasst sich mit dem Graphenspiel *Cops and Robbers*. Auf einem gegebenen

Graphen bewegen sich ein Räuber und einer oder mehrere Polizisten. Natürlicherweise versuchen die Polizisten, den Räuber zu fangen, der das verhindern will. Eine der neusten Varianten des Spiels wurde 2018 von Kinnersley und Peterson eingeführt, das sogenannte *Bridge-Burning Cops and Robbers*. Abwechselnd ziehen Räuber und Polizisten. Während eines Zuges kann man auf einem Knoten bleiben oder entlang einer Kante zu einem benachbarten Knoten gehen. Die Besonderheit des Spiels: Wenn sich der Räuber entlang einer Kante bewegt, wird diese danach 'verbrannt', also vom Graphen entfernt. Dadurch hat der Räuber die Möglichkeit zu gewinnen, indem er eine Graphenkomponente kreiert, auf der er sich befindet, und die frei von Polizisten ist. Bei einem Graphen, bei dem die Polizisten das Spiel bei optimaler Spielweise gewinnen, kann man sich die *Capture Time* anschauen, die angibt, wie lange die Polizisten beim Fangen des Räubers höchstens brauchen.

Wir beweisen eine Vermutung von Kinnersley und Peterson bezüglich der Capture Time bei nur einem Polizisten. Wir zeigen, dass es Graphen auf n Knoten gibt, bei dem ein einzelner Polizist bei optimaler Spielweise gewinnt, und dabei eine Capture Time von $\Omega(n^3)$ besitzt.

In Kapitel 5 werfen wir mit dem *Extreme Vertex Destruction Modell* einen Blick auf die Stabilität von Netzwerken, die von eigennützigen Spielern erstellt wurden. Diese Graphen- und Spieltheoretische Modellierung von Kliemann et al. betrachtet den Schaden, den das Entfernen eines Knotens aus einem Netzwerk-Graphen anrichten kann. Dieser Schaden ist definiert als die Menge der Knotenpaare, die sich nun nicht mehr in der gleichen Komponente befinden. Die Knoten, bei denen der Schaden maximal ist, wird *Max-Sep Knoten* genannt. Im Extreme Vertex Destruction Modell wird einer dieser Max-Sep Knoten gleichverteilt per Zufall ausgewählt und zerstört.

Kliemann et al. zeigten, dass es Graphen auf n Knoten gibt, die sich im *Swap-Gleichgewicht* befinden und einen durchschnittlichen Schaden von $\Omega(n^{3/2})$ haben. Diesen Wert verbessern wir in diesem Kapitel auf die scharfe Schranke von $\Omega(n^2)$.

Summary

In this thesis, we take a look at several graph theoretical problems. We present two streaming algorithms for finding an Euler tour in a graph, prove a tight bound on the capture time in the Bridge-Burning Cops and Robbers Game and solve an open problem for the Extreme Vertex Destruction Model.

In Chapter 2, we consider the classical Euler tour problem. An Euler tour of a graph is a closed trail such that each edge is visited exactly once. Several algorithms are known for finding them. We take a modern look at this problem in the context of the graph streaming model. Here, RAM is of size $\mathcal{O}(n \text{ polylog}(n))$, where n is the number of nodes, and the graph is given as a stream of its edges. Since large graphs cannot be stored in this restricted memory space, the usual algorithms are not transferable to this model. We bypass this hurdle and give a one-pass algorithm for finding Euler tours in the graph streaming model.

In Chapter 3, we regard a lesser-known streaming model, the so-called StrSort model, to tackle a downside of our algorithm mentioned above. The algorithm stores an Euler tour on an output tape in form of a successor function. The order of the edges is given, but the edges are not actually sorted in the order of the Euler tour. Therefore, further processing the tour with another streaming algorithm might become difficult. In the StrSort model, less RAM is given, but the stream is regularly sorted in a global partial order. We give an algorithm for sorting the edges of a graph according to a found Euler tour, that has a preparation step in the graph streaming model and a processing step in the StrSort model.

The traditional game of cops and robbers is the topic of Chapter 4. On a given graph, cops and robbers move along the edges from node to node. Naturally, the cops try to capture the robber, who wants to prevent that. We consider one of the newest twists of this classical game. In the so-called Bridge-Burning Cops and Robbers Game, every time the robber traverses an edge, this edge is deleted afterwards. The analysis of this game is a challenge because of the ever-changing graph. We study winning strategies of a single cop and make statements on the maximum number of turns of such strategies.

In Chapter 5, we study networks formed by selfish agents. The agents (or players) are represented by the nodes of a graph, where an edge between two nodes implies a connection between the represented players in the network. When a node is ‘destroyed’, i.e. the adjacent edges are deleted, the network is damaged and some players lose the connection to each other. In the Extreme Vertex Destruction Model, we observe the impact of such a deletion on swap equilibrium graphs and consider their robustness. We pick and destroy a node in a certain way and make statements on the maximum amount of damage that can cause on a swap equilibrium graph.

Chapter 1

Introduction

In this chapter, we give an outline of the different topics considered in this thesis. We give brief explanations of the problems and previous work, and state our results with short mentions of our respective approaches.

1.1 Euler Tours in the Graph Streaming Model

In graph theoretical problems, where the size of an instance is of a larger order than the available RAM size, the *graph streaming model*, introduced by Feigenbaum et al. [13] is a popular model to tackle the problem of insufficient memory space. Let $G = (V, E)$ be a graph on n nodes and m edges. The RAM size is determined as $\mathcal{O}(n \text{ polylog}(n))$, while the graph itself is given as a stream of its edges. In Chapter 2, we study the problem of finding an Euler tour in G, i.e. a closed trail containing every edge of E. Our main result is the first one-pass streaming algorithm computing an Euler tour of G (or stating that an Euler Tour does not exist) in the form of an edge successor function with only $\mathcal{O}(n \log(n))$ RAM, which is optimal for this setting (e.g. Sun and Woodruff ([30],2015)). Since the output size of $\mathcal{O}(m)$ can be much larger than the RAM size, we use a write-only tape to gradually output the solution. Our approach is to partition the edges into edge-disjoint cycles and to merge the cycles until a single Euler tour is achieved. Since the limited RAM allows the processing of only a constant number of cycles at once, cycles have to be merged that partially are no longer present in RAM. We solve this problem with an edge swapping technique, for which storing two certain edges per node is sufficient to merge tours without having all tour edges in RAM. For the analysis we consider a successor function on the set of edges and equivalence classes corresponding to cycles, and give conditions under which the swapping of edge successors leads to a merging of equivalence classes.

1.2 Euler Tours in the StrSort Model

The algorithm given in Chapter 2 outputs an Euler Tour in form of a successor function. When the Euler Tour has to be further processed in the Graph Streaming Model, this

might be impractical. It is only natural to request a graph stream on which the edges are sorted according to a given Euler Tour, which would significantly improve the value of the result. In Chapter 3 we consider the problem of computing Euler tours in undirected graphs and sorting the edges accordingly in the *StrSort model*, which was introduced by Aggarwal et al. [1]. Again, the graph is given as a stream of its edges and can only be read sequentially, but while conducting a pass over the stream we are allowed to write another stream which will be the input for the next pass. Additionally, items in the stream are sorted between passes in a global partial order. The model is appropriate for solving problems in the streaming context, when the size of the output is in the order of m and the semi-streaming model with $\mathcal{O}(n \log(n))$ RAM is too restricted. We give the first algorithm in the StrSort model and resolve the problem within the following complexity bounds. In a processing step in the W-stream model, we compute a new stream of the edges with some additional information, using $\mathcal{O}(n \log(n))$ RAM and only a single pass. In the main step, we show that $\mathcal{O}(\log(n))$ passes are required, using only $\mathcal{O}(\log(n))$ RAM, to sort the edges and output the Euler tour in form of consecutive edges. Let $G = (V, E)$ be an undirected graph containing an Euler Tour. Like in Chapter 2, the underlying idea is the merging of disjoint cycles. This time, we consider an out-tree $\vec{T} = (W, \vec{F})$ corresponding to G, where each node of W corresponds to a cycle, and two nodes in $\vec{T}$ are connected if the corresponding cycles share a common node. This tree also might be too large for the internal memory. In each sorting step, we want to merge each node in $\vec{T}$ of odd degree with its predecessor, resulting in a tree of at most half the size. The according cycles are also merged. In a preprocessing step, we give each edge of E in the stream the information in which cycle it is, which position it has in the cycle, and which position it would have after a cycle merge. Storing this additional information at the start is sufficient to sort the edges according to the tours given by $\vec{T}$ at each sorting step.

1.3 The Bridge-Burning Cops and Robbers Game

The game *cops and robbers* as a graph-theoretical problem was introduced in the early 1980s. It is a two player game played on a graph, where one player controls a set of cops and the other player controls a robber. Cops and robber move along the edges from node to node. The cops' objective is to capture the robber, while the robber wants to avoid that.

Over the years, many variations of this game were studied. An interesting twist to the concept was given by Kinnersley and Peterson [21] in 2018 with the so-called *bridge-burning cops and robbers*. In this variant, every time the robber traverses an edge, this edge is deleted afterwards. Here, the robber has another method of winning than prolonging the game indefinitely. He can actually escape by getting on a connected component without cops. Kinnersley and Peterson looked at the *capture time* of graphs, i.e. the maximum number of turns the cops need to win, assuming they can always capture the robber with the right strategy. For one cop, they showed that for every $n \in \mathbb{N}$ there exists a graph on n nodes with a capture time of $\Omega(n^2)$.

In Chapter 4, we construct graphs with a capture time of $\Omega(n^3)$, where n is the number of nodes. As shown in [21], this bound is tight. Our graphs have a very low number of nodes with odd degree. With the fact that in every connected component of a graph, the number of nodes with odd degree is even, we can show that the robber has very limited options to escape to a different connected component than the cop. With that, we can proof that a single cop can always win on these graphs. Further, we show that if the cop steps on certain nodes, the robber wins. Since the cop has to prevent these nodes, his only choice is to use a path of length linear in n. In fact, we show that the cop has to walk along the whole path every time he wants the robber to even move from one node to another. Since the robber is able to move $\Omega(n^2)$ times, we get a capture time of $\Omega(n^3)$.

1.4 Swap Equilibrium Graphs in the Extreme Vertex Destruction Model

In Chapter 5, we study the *extreme vertex destruction model*, a network creation game introduced by Kliemann et al. [22] to determine the robustness of networks formed by selfish agents. For the sake of terminology, instead of 'node' we use the synonym 'vertex' in this chapter. The damage a network can experience is simulated by the removal of a vertex v, i.e. the deletion of all the adjacent edges. It is measured by the *separation* of v, the number of vertex pairs that are no longer connected after v is destroyed. The vertices with the highest separation are called *max-sep vertices*. Deleting one of them causes the highest amount of damage. In the extreme vertex destruction model, a max-sep vertex is picked uniformly at random and destroyed. The cost of a player is the expected number of vertices, the player is no longer connected to after the destruction. To check the graph for stability, players can make a so-called *swap*. They can delete an adjacent edge and can instead create another adjacent edge. A graph has a *swap equilibrium* (SE), if no single player can reduce his (expected) cost by performing a swap.

Kliemann et al. (2017) showed that there is a family of SE graphs with total cost of $\Omega(n^{3/2})$, where n is the number of vertices. We prove that SE graphs can be even more vulnerable by presenting a family of SE graphs with total cost of $\Omega(n^2)$. Those graphs consist of four big stars and two rather small stars that are connected to a K_4 in the center. Two vertices of the K_4 are the only max-sep vertices. First, we show that those graphs are indeed SE graphs. Would a vertex perform a swap, it would either increase the separation of the 'worse' max-sep vertex, making it the only max-sep vertex of the new graph, or it would become a max-sep vertex itself. Either way, a swap would not lower the cost. Finally, we show that the graphs have an expected separation of $\Omega(n^2)$.

Chapter 2

Euler Tours in the Graph Streaming Model

2.1 Introduction

2.1.1 Euler tours in the graph-streaming model

In the Euler tour problem, we are looking for a closed trail in an undirected graph $G = (V, E)$, where $n = |V|$ and $m = |E|$, such that each edge is visited exactly once. It is a well studied graph-theoretic problem with applications in the field of big data. For example, solving the traveling salesman problem with the well-known Christofides algorithm requires an Euler tour [8]. Further, for de novo genome assembly, large de Bruijn graphs can be examined with Eulerian paths [27]. For the processing of large graphs, the *graph streaming* or *semi streaming* model introduced by Feigenbaum et al. [13] has been studied extensively over the last decade. In this model, a graph with n nodes and m edges is given as a stream of its edges. Random-access memory (RAM, also called internal memory) is restricted to $\mathcal{O}(n \text{ polylog}(n))$ edges at a time, see, e.g., the survey [23] for a detailed introduction. In consequence, the model cannot be applied to problems where the size of the solution exceeds this amount of memory. Since the size of an Euler tour is m, which might even be $\Theta(n^2)$, we need a relaxation of the model that allows us to store the output separate from the RAM. An obvious solution for this problem is the addition of a write-only output tape with the sole purpose of storing the Euler tour. In this setting it is common to output the Euler tour in the form of an edge-successor function (e.g. [4], [9]).

2.1.2 Previous work

Graph streaming with the usage of an additional output tape resembles the model used by Grohe et al. [17]. They consider Turing machines with a read/write input tape, multiple read/write tapes of significantly smaller size, called ‘internal memory tapes’, and an additional write-only output tape. They count the number of times the head of the input tape changes it’s direction. This is closely related to the streaming model by

Feigenbaum et al. since a streaming pass can be remodeled as a run on the input tape with two direction changes of the head. In a more recent work by François et al. [15], a read-only input tape and a write-only output tape are considered for the problems of stream reverting and sorting.

Another related model is the *W-streaming model* introduced by Demetrescu et al. [10], which is a relaxation of the classical streaming model. It originated as a more restrictive alternative to the *StrSort model* introduced by Aggarwal et al. [29, 1]. At each pass, an output stream is written, which becomes the input stream of the next pass. Finding an Euler tour in trees in W-streaming has been studied in multiple papers (e.g., [9]), but to the best of our knowledge the general Euler tour problem has hardly been considered in a streaming model so far. However, there are some general results for transferring PRAM algorithms to the W-streaming model. Atallah and Vishkin [4] presented a PRAM algorithm for finding Euler tours, using $\mathcal{O}(\log(n))$ time and $n+m$ processors. Transferred to the W-streaming model with the methods from [9], this algorithm computes an Euler tour in the form of a bijective successor function within $p = \mathcal{O}(m \text{ polylog}(n)/s)$ passes, where s is the RAM-capacity.

Sun and Woodruff [30] showed that even a one-pass streaming algorithm for verifying whether a graph is Eulerian needs $\Omega(n \log(n))$ RAM. This implies that the minimum RAM-requirement of a one pass streaming algorithm with additional output tape for finding an Euler tour also is $\Omega(n \log(n))$.

2.1.3 Our contribution

We present the streaming algorithm Euler-Tour for finding an Euler tour in a graph in form of a bijective successor function or stating that the graph is not Eulerian, using only one pass, $\mathcal{O}(n \log(n))$ bits of RAM and an additional write-only output tape. This is not only a significant improvement over previous results, but is in the view of the lower bound of Sun and Woodruff [30] the first optimal algorithm in this setting. Atallah and Vishkin [4] find edge disjoint tours (in our case cycles) and connect them by pairwise swapping the successor edges of suitable edges. This idea is easy to implement without memory restrictions but the implementation gets distinctly more complicated with limited memory space: We cannot store all cycles in RAM. Therefore, we have to output edges and their successors before finding resp. processing all cycles. Our idea is to keep specific edges of some cycles in RAM along with additional information so that we are able to merge following cycles regardless of their appearance with already processed tours which likely are no longer present in RAM.

We develop a new mathematical foundation by partitioning the edges into equivalence classes induced by a given bijective successor function and prove structural properties that allow to iteratively change this function on a designated set of edges so that the modified function is still bijective. Translated to graphs this is a tour merging process. This mathematical approach is quite general and might be useful in other routing scenarios in streaming models.

2.1.4 Organization of the article

In Section 2.2 we give some basic definitions. Our algorithm consists of several subroutines and technical data structures. Therefore, for the reader's convenience Section 2.3 is dedicated to an informal and example-based description. Section 2.4 contains the pseudo code of the algorithm. In Section 2.5, we show the connection of the concepts of Euler tours and successor functions and then show that the required RAM of the algorithm does not exceed $\mathcal{O}(n\log(n))$ and that the output actually depicts an Euler tour (Theorem 2.2).

2.2 Preliminaries

Let $\mathbb{N} := \{1, 2, \ldots\}$ denote the set of natural numbers. For $n \in \mathbb{N}$ let $[n] := \{1, \ldots, n\}$. In the following, we consider a simple graph $G = (V, E)$ without loops where V denotes the set of nodes and E the set of (undirected) edges. A *walk* in G is a finite sequence $T = (v_1, \ldots, v_\ell)$ of nodes of G with $\{v_i, v_{i+1}\} \in E$ for all $i \in \{1, \ldots, \ell-1\}$. If additionally every edge is visited at most once, we call it *trail.* The *length* of T is $\ell-1$. The (directed) edge set of T is $E(T) := \{(v_i, v_{i+1}) | i \in [\ell-1]\}$. We also write $e \in T$ instead of $e \in E(T)$. For a directed edge e we denote by $e_{(1)}$ its first and by $e_{(2)}$ its second component. A trail $T = (v_1, \ldots, v_\ell)$ with $v_1 = v_\ell$ is called a *tour.* In tours, we usually do not care about starting point and end point, so we slightly abuse the notation and write $v_{i+\ell}$ or $v_{i-\ell}$ for a node v_i, identifying $v_0 := v_\ell$ and $v_{\ell+1} := v_2$ and so on. If additionally $v_i \neq v_j$ holds for all $i, j \in [\ell-1], i \neq j$ (and $\ell \geq 3$), we call T a *cycle.* An *Euler tour* of G is a tour T with $E(T) = E$. Since in the streaming model the graph is represented as a set of edges, we often use the edges for the depiction of tours. With $e_i := \{v_i, v_{i+1}\}$ for all $i \in [l-1]$, T can be written as $T = (e_1, \ldots, e_l)$. Here, we also use the slightly abusive index notation. Note that for the tour T the edges are distinct. For $i \in [l]$, we call e_{i+1} the *successor edge of* e_i in tour T. Our algorithm outputs an Euler tour $T = (v_1, \ldots, v_{|E|}, v_1)$ in form of a *successor function*, i.e., for every $i \in [|E|]$, we output the triple (v_i, v_{i+1}, v_{i+2}), where $\{v_{i+1}, v_{i+2}\}$ is the successor edge of $\{v_i, v_{i+1}\}$ in T.

2.3 Idea of the algorithm

In this section we explain the new algorithmic idea in a more informal way. First we describe how merging of subtours can be accomplished without RAM limitation clarifying why this does not work in the streaming environment. Thereafter we explain our merging technique, its locality and RAM efficiency.

2.3.1 Subtour merging in unrestricted RAM

Recall that the Euler tour will be presented by a successor function, so for every edge we will compute the corresponding successor edge in the tour. Let $G = (V, E)$ be an Eulerian graph and T, T' be edge-disjoint tours in G. The tour induces an orientation

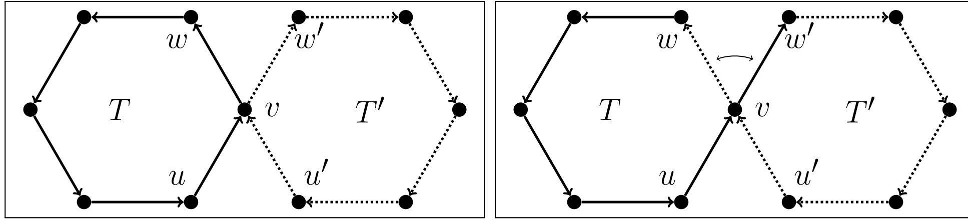


Figure 2.1: Merging tours by swapping two edges

of the edges in a canonical way. If T and T' have a common node v, it is easy to merge them to a single tour: T has at least one in-going edge (u, v) with a successor edge (v, w), and T' has at least one in-going edge (u', v) with a successor edge (v, w'). By changing the successor edge of (u, v) from (v, w) to (v, w') and the successor edge of (u', v) to (v, w), we get a tour containing all edges of $T \cup T'$ (see Figure 2.1). The same principle can be applied when merging more than two tours at once. When we have a tour T and tours $T_1, \dots, T_k$, $k \in \mathbb{N}$, such that $T, T_1, \dots, T_k$ are pairwise edge-disjoint and for every $j \in [k]$ there is a common node v_j of T and T_j, switching the successor edges of two in-going edges per node v_j as described above creates a tour containing the edges of $T \cup T_1 \cup \dots \cup T_k$.

We can use this method as a simple algorithm for finding an Euler tour:
a) Find a partition of E into edge disjoint cycles.
b) Iteratively pick a cycle C and merge it with all tours encountered so far which have at least one common node with C.

Such a merging process certainly converges to a tour covering all nodes, if a subtour obtained by merging some subtours does not decompose into some subtours again. If we use a local swapping technique to merge tours, this can very well happen, if swapping is applied again to some other node of the merged tour (see Figure 2.2). In the RAM model this problem does not appear, since we can keep all tours in RAM and avoid such fatal nodes. But in the streaming model with $O(n \log n)$ RAM it is far from being obvious how to implement an efficient tour merging for the following reasons.

1. We cannot keep every intermediate tour in RAM, so we have to regularly remove some edges together with their successors from RAM, even if we do not know the edges yet to come. On the other hand, we have to keep edges in RAM which are essential in later merging steps.

2. Sometimes we have to merge cycles with tours that had already left RAM. Therefore, we must keep track of common nodes and the related edges.

2.3.2 Subtour merging in limited RAM

We start with an example of four cycles $C_1, \dots, C_4$, all sharing one node v (see Figure 2.3). Let's assume that the cycles are in the same order as found by the algorithm. Let $(u_1, v), \dots, (u_4, v)$ be the respective in-going edges and $(v, w_1), \dots, (v, w_4)$ be the

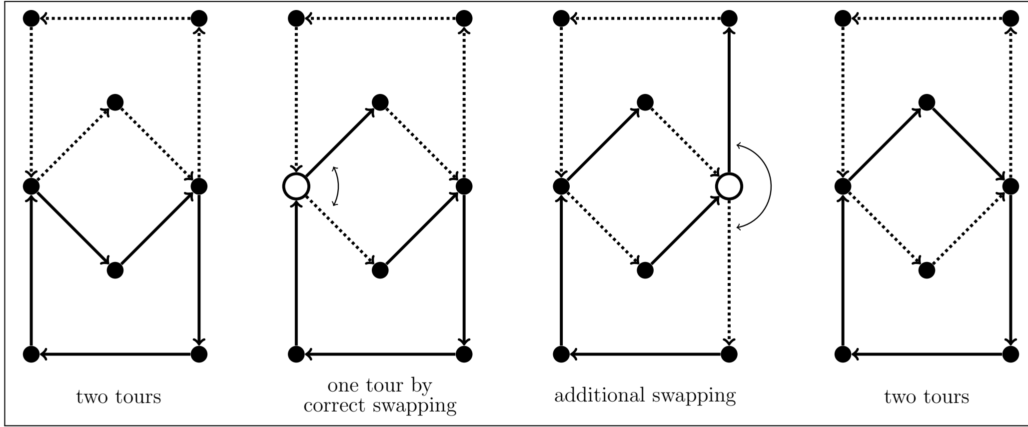


Figure 2.2: Multiple edge-swapping can destroy the merging effect

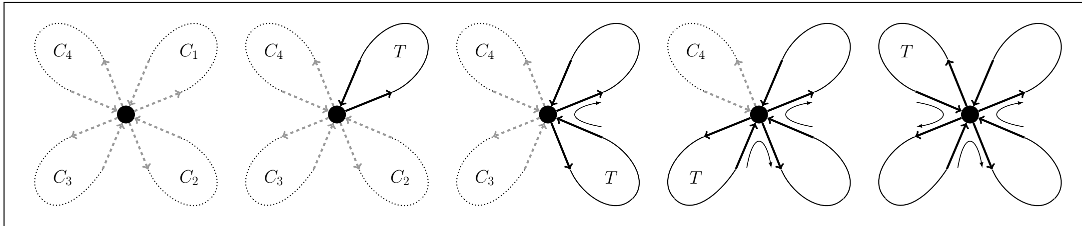


Figure 2.3: Successive merging of cycles

respective out-going edges. By swapping the successor edges of (u_1, v) and (u_2, v) as explained before, we construct a tour T containing all edges from C_1 and C_2. We then merge this tour with C_3 swapping the successor edges of (u_1, v) and (u_3, v). Finally, C_4 is merged with T by swapping the successors of (u_1, v) and (u_4, v). The successor edges are now as follows:

$$(u_1, v) \longrightarrow (v, w_4) \qquad (u_2, v) \longrightarrow (v, w_1)$$
$$(u_3, v) \longrightarrow (v, w_2) \qquad (u_4, v) \longrightarrow (v, w_3)$$

For $i > 1$ and cycle C_i, the successor of the edge (u_i, v) is edge (v, w_{i-1}), the out-going edge of C_{i-1}. The edge (u_1, v) of the cycle C_1 has the out-going edge of the last cycle as its successor edge. The edge (u_1, v) is the first in-going edge of v and called the *first-in edge of* v. Let us briefly show how this merging can be implemented in the streaming model with an additional output tape. When C_1 is kept in RAM, we store the edge (u_1, v), since we don't know its final successor edge yet. We also keep the edge (v, w_1) in RAM, because it will be the successor edge of C_2. We call such an edge the *potential successor edge of* v.

The crux is that, no matter how many cycles are merged at the node v, we get along with only two additional edges in RAM at a time: the first-in edge, which will never change after it is initialized and the recent potential successor, which will be replaced

every time we merge a new cycle at v. In our example, when merging a cycle C_i for $i > 1$, we assign the edge (v, w_{i-1}) as successor edge of (u_i, v) and replace (v, w_{i-1}) by (v, w_i) in RAM as potential successor edge of v. All other edges are written on the output tape with their corresponding successors and don't have to be stored in RAM. Finally, when no more cycles with node v occur, we can write (u_1, v) together with the recent successor edge (in our case this is (v, w_4)) on the output tape.

Now, let us consider the more complicated case, where we wish to merge a cycle C with multiple tours at several nodes. Consider a cycle C and tours $T_1, \dots, T_j$. Let $v_1, \dots, v_j$, be nodes such that v_i belongs to T_i and C for all i. We distinguish between merging at three types of nodes:

1. For the nodes $v_1, \dots, v_j$ we swap the successor edges.

2. Nodes in C and in $T_1 \cup \dots \cup T_j \backslash \{v_1, \dots, v_j\}$: as only one successor edge swapping per tour is needed, these additional common nodes are not used, hence for every $v \in T_1 \cup \dots \cup T_j \backslash \{v_1, \dots, v_j\}$ the in-going edge (u, v) of C keeps its successor edge, so nothing happens here.

3. Nodes in $C \backslash (T_1 \cup \dots \cup T_j)$. These nodes are visited by the algorithm for the first time. Since we might want to merge C with future cycles at these nodes, we store for every $v \in C \backslash (T_1 \cup \dots \cup T_j)$ the in-going edge (u, v) of C as first-in edge and the out-going edge (v, w) of C as potential successor edge.

At any point of time we store at most one cycle C and two edges per node (a first-in edge and a potential successor) in RAM, so $\mathcal{O}(n \log n)$ RAM suffices. Note that the very first cycle found by the algorithm consists only of type 3 nodes, so every edge will become a first-in edge.

2.3.3 High level description

For the readers convenience we give a high level description of our algorithm. A detailed description in pseudo code together with an outline of the analysis and the proof of the main theorem will follow in the next sections. We denote the set of first-in edges by F.

1. Iteratively:

   1.1. Read edges from the input stream until the edges in RAM contain a cycle C.

   1.2. If a node v of C is visited for the first time,

      a) store the in-going edge (u, v) of C in F (we will process these $\leq n$ edges in step 2.),

      b) remember the out-going edge (v, w) as potential successor edge of v.

   1.3. Every node v that has already been visited, has thereby been assigned to a unique tour T with $v \in C \cap T$. For each tour that shares a node with C, choose exactly one common node.

1.4. For each node v chosen in step 1.3. 'swap the successors'. That means, we write the in-going edge e on the output tape and take the recent potential successor edge of v as successor edge for e. Then, save the out-going edge as new potential successor edge of v.

1.5. For each edge that has not been stored in F (step 1.2.) or written on the output tape (step 1.4.) so far, write this edge on the output tape and take as successor the following edge in C.

1.6. Assign all tours with common nodes together with all newly visited nodes to a single tour.

2. After the end of the input stream is reached, all edges have either been written on the output tape or stored in F. For every edge $(u, v) \in F$, write it on the output tape and take as its successor the potential successor edge of v.

An example of how the algorithm works can be found in the appendix of th chapter.

2.4 The Algorithm and Main Results

2.4.1 The Algorithm in Pseudo Code

To enable a clear and structured analysis, in this section we present the pseudo-code for our algorithm. For a better understanding it is split up into several procedures that correspond to the steps from our high level description in Section 2.3.3. Note that these procedures are not independent algorithms, since they access variables from the main algorithm. The output is an Euler tour on G, given in the form of a successor function δ^*. To be more precise, the output is a sequence of triples (v_1, v_2, s) written on the output tape with $v_1, v_2, s \in V$ and $\{v_1, v_2\} \in E$. Each of these triples represents the information $\delta^*((v_1, v_2)) = (v_2, s)$. If a triple (v_1, v_2, s) is written on the output tape, we say that the edge (v_2, s) is *marked as successor* of the edge (v_1, v_2). For every node $v \in V$ we store the two values $s(v)$ and $t(v)$. If v is considered for the first time, $s(v) = t(v) = 0$. Otherwise, $s(v) \in V$ indicates that $(v, s(v))$ is the potential successor edge of v and $t(v) \in [n]$ represents the tour that v is assigned to at the moment.

Algorithm 1: EULER-TOUR

input : Undirected graph $G = (V, E)$, edge by edge on a stream S
output: Euler tour on G, i.e. a successor function δ^*, if there is one

1 $c := 0$; $F := \varnothing$; $E_{\text{int}} := \varnothing$; for every $v \in V$: $s(v) := 0, t(v) := 0$;
2 **for** *every edge e on S* **do**
3 $\quad E_{\text{int}} := E_{\text{int}} \cup \{e\}$;
4 $\quad$ **if** $G_{\text{int}} = (V, E_{\text{int}})$ *contains a cycle C* **then**
5 $\quad\quad$ MERGE-CYCLE (C);
6 **if** $E_{\text{int}} = \varnothing$ **then**
7 $\quad$ ERROR: At least one node with odd degree exists;
8 **if** *there exist* u, v *with* $t(u) \neq t(v) \neq 0$ **then**
9 $\quad$ ERROR: Graph is not connected;
10 WRITE-F;

The algorithm searches the stream for cycles (Step 1.1. in our high level description in Section 2.3.3) and whenever a cycle is found, we will run the procedure MERGE-CYCLE on this cycle. Next we state the procedure MERGE-CYCLE, which implements the steps 1.2. to 1.6. in Section 2.3.3.

Procedure Merge-Cycle

input : Ordered directed cycle $C = (v_1, \ldots, v_k)$ of length k

1 NEW-NODES ;
2 CONSTRUCT-J-M;
3 MERGE;
4 WRITE;
5 UPDATE;
6 **for** *every edge* $e \in C$ **do**
7 $\quad$ delete e from E_{int}

Let us now explain all the procedures that are used in MERGE-CYCLE. The procedure NEW-NODES implements step 1.2. from Section 2.3.3. If a node $v \in C$ is processed the very first time by the algorithm, this is indicated by $t(v) = 0$. If this is the case, we store the C-edge in-going to v in the set F and define $s(v)$ as the next node on C. So the edge (v, v_{i+1}) becomes the potential successor of v.

Procedure New-Nodes

1 **for** $i = 1, \ldots, k$ **do**
2 $\quad$ **if** $t(v_i) = 0$ **then**
3 $\quad\quad s(v_i) = v_{i+1}$;
4 $\quad\quad F = F \cup \{(v_{i-1}, v_i)\}$;

The procedure CONSTRUCT-J-M is a realization of step 1.3. in Section 2.3.3. Recall that the t-values of the nodes represent the tours that we have constructed so far. For every possible t-value $j \in [n]$, we pick exactly one node v with $t(v) = j$ if there is one. These nodes are stored in J, their t-values are stored in M. The nodes in J are the nodes we want to use for merging tours. If two nodes have the same t-value, this means they are already part of the same tour (see Lemma 2.13), so we have to avoid using both of them for merging.

Procedure Construct-J-M

```
1 M = ∅;  J = ∅;
2 for j = 1, ..., n do
3   if exists i ∈ [k] with t(v_i) = j then
4     add exactly one v_i with t(v_i) = j to the set J;
5     M = M ∪ {j};
```

In the procedure MERGE, we use the nodes from J to merge all tours that share a node with the cycle C by edge-swapping (step 1.4. in Section 2.3.3).

Procedure Merge

```
1 for each v_i ∈ J do
2   write (v_{i-1}, v_i, s(v_i)) on the output tape;
3   s(v_i) := v_{i+1};
```

In the procedure WRITE, we take care of all the edges that have not been stored in F and have not been written on the output tape in the procedure MERGE (Step 1.5. in Section 2.3.3).

Procedure Write

```
1 for each edge (v_i, v_{i+1}) ∈ C that has not been written on the output tape or
   added to F do
2   write (v_i, v_{i+1}, v_{i+2}) on the output tape;
```

In the procedure UPDATE we update the t-values to implement step 1.6.. This way we ensure that the t-values of the nodes still represent the tours they belong to. If $M = \varnothing$, the cycle C has no intersecting nodes with already constructed tours, so it is declared as a new tour. Otherwise, all nodes from the cycle C and all intersecting tours now belong to one single tour, represented by the t-value a.

Procedure Update

```
1  a := 0;
2  if M = ∅ then
3  |  c := c + 1;
4  |  a := c;
5  else
6  |  a := min(M);
7  for each v ∈ V do
8  |  if t(v) ∈ M then
9  |  |  t(v) := a;
10 for i = 1, ..., k do
11 |  t(v_i) = a;
```

Finally, in the procedure Write-F (step 2. in Section 2.3.3), the first-in edges that have been stored in F during the algorithm are written on the output tape with proper successors. Note that for each $(u, v) \in F$ it holds $s(v) \neq 0$.

Procedure Write-F

```
1  for each edge (u, v) ∈ F do
2  |  write (u, v, s(v)) on the output tape;
```

2.4.2 Main Results and Proof Idea

For the readers convenience we first sketch the proof idea and new mathematical techniques in the analysis. The goal is to show that the algorithm Euler-Tour works as claimed, that is, the memory requirement does not exceed the $\mathcal{O}(n \log n)$ bound and the output successor function δ^* determines an Euler tour for the input graph G.

We state the two main results of this paper:

Lemma 2.1. *Algorithm* Euler-Tour *needs at most $\mathcal{O}(n \log n)$ bits of RAM.*

Theorem 2.2. *If G is Eulerian, δ^* determines an Euler tour on G.*

The memory estimation is done in Subsection 2.5.2 and is a careful analysis of the Algorithm Euler-Tour and its subroutines. The main arguments are that at any point of time we store at most two edges per node (a first-in edge and a potential successor), the edges of the cycle that is recently processed and a label for every node (its t-value). The correctness proof turns out to be much more complicated. Every bijective successor function δ partitions the edges of G into edge-disjoint tours. In this case 'being in the same tour' forms an equivalence relation on the set of edges, which is denoted by $\equiv_\delta$. We prove this fact in Theorem 2.6. This allows us to use equivalence classes, when analyzing the dynamic change of subtours induced by successor functions in an algebraic

framework. This analysis takes place in Subsection 2.5.3 and is concluded by Theorem 2.2, which states that the output successor function δ^* indeed induces an Euler Tour on G. We start with a successor function δ^c. In Lemma 2.9 we show that δ^c is bijective and that the equivalence classes of the corresponding equivalence relation $\equiv_{\delta^c}$ are simply the directed cycles found during the algorithm EULER-TOUR (line 4). Next we construct a recursive sequence of bijective successor functions $\delta_0^*, \ldots, \delta_N^*$ and their corresponding equivalence classes with $\delta_0^* = \delta^c$ and $\delta_N^* = \delta^*$, where N denotes the total number of cycles found by the algorithm. The differences from δ_{k+1}^* to δ_k^* basically correspond to the successor swapping process in the Procedure MERGE-CYCLE running on the $(k+1)$-th cycle found by the algorithm. At this point we need the lemmata 2.12 and 2.13, the crux of the analysis. Lemma 2.12 ensures that the successor swapping leads to a union of all involved equivalence classes and finally Lemma 2.13 tells us that all successor functions from the sequence $\delta_0^*, \ldots, \delta_N^*$ are bijective, that the procedure UPDATE works correctly and that after an edge (u, v) has been processed by EULER-TOUR, all processed edges incident in u or v belong to the same equivalence class. Using that Eulerian Graphs are connected, the proof of Theorem 2.2 then follows with a few simple arguments, because by applying Theorem 2.6 it suffices to show that all edges of G are in the same equivalence class of the relation $\equiv_{\delta^*}$.

2.5 Detailed Analysis

2.5.1 Subtour representation by equivalence classes

In this subsection we present some basic definitions and results that allow us to transfer the problem of tour merging in a graph to the notion of equivalence relations on E. This will facilitate an elegant and clear analysis of our algorithm.

Definition 2.3. *(i) Let $G = (V, E)$ be an undirected graph. An* orientation *of the edges of G is a function $R : E \to V^2$ such that for every edge $\{u, v\} \in E$ either $R(\{u, v\}) = (u, v)$ or $R(\{u, v\}) = (v, u)$. So $R(G) := (V, R(E))$ is a directed graph. For an oriented edge $e = (u, v)$ we write $e_{(1)} := u$ and $e_{(2)} := v$.*

(ii) Let $\vec{G} = (V, \vec{E})$ be a directed graph. A successor function *on $\vec{G}$ is a function $\delta : \vec{E} \to \vec{E}$ with $\delta(e)_{(1)} = e_{(2)}$ for all $e \in \vec{E}$.*

(iii) Let $\vec{G} = (V, \vec{E})$ be a directed graph with successor function δ. We define the relation $\equiv_\delta$ on $\vec{E}$ as follows: For $e, e' \in E$, $e \equiv_\delta e' :\Leftrightarrow \exists k \in \mathbb{N} : \delta^k(e) = e'$, where δ^k denotes the k-wise composition of δ.

So $e \equiv_\delta e'$ means that e' can be reached from e by iteratively applying δ.

Lemma 2.4. *Let δ be a bijective successor function on a directed graph $\vec{G} = (V, \vec{E})$. Then $\equiv_\delta$ is an equivalence relation on $\vec{E}$.*

Proof. Reflexivity: Let $e \in \vec{E}$. Since $\vec{E}$ is finite, there exists $k \in \mathbb{N}$ with the following property: There exists $k' \in \mathbb{N}$ with $k' < k$ and $\delta^k(e) = \delta^{k'}(e)$. Otherwise, all elements

of the sequence $(\delta^\ell(e))_{\ell\in\mathbb{N}}$ would be pairwise distinct, in contradiction to the fact that there exist only $|E|$ edges. Let k be minimal with this property. Since δ is injective, it follows that $\delta^{k-1}(e) = \delta^{k'-1}(e)$ and the minimality of k enforces that $k'-1 \notin \mathbb{N}$. So $k' = 1$, therefore $\delta^k(e) = \delta(e)$ and by the injectivity of δ we have $\delta^{k-1}(e) = e$.

Symmetry: Let $e, e' \in \vec{E}$ with $e \neq e'$ and $e \equiv_\delta e'$. Then there exists a minimal $k \in \mathbb{N}$ with $\delta^k(e) = e'$. As shown above, there also exists a $k' \in \mathbb{N}$ with $\delta^{k'}(e) = e$. Then $k < k'$, because otherwise $\delta^{k-k'}(e) = \delta^{k-k'}(\delta^{k'}(e)) = \delta^k(e) = e'$, in contradiction to the minimality of k. It follows that $\delta^{k'-k}(e') = \delta^{k'-k}(\delta^k(e)) = \delta^{k'}(e) = e$.

Transitivity: Let $e, e', e'' \in \vec{E}$ with $e \equiv_\delta e'$ and $e' \equiv_\delta e''$. Then there exist $k_1, k_2 \in \mathbb{N}$ with $\delta^{k_1}(e) = e'$ and $\delta^{k_2}(e') = e''$. So we have $\delta^{k_1+k_2}(e) = e''$. □

In the following we denote the equivalence class of an edge $e \in \vec{E}$ w.r.t. $\equiv_\delta$ by $[e]_\delta$.

The following technical lemma is necessary to show in Theorem 2.6 that the equivalence classes of δ always form tours on $\vec{G}$.

Lemma 2.5. *Let $\vec{G} = (V, \vec{E})$ be a directed graph with bijective successor function δ and the corresponding equivalence relation $\equiv_\delta$. Then we have:*

(i) Let $e \in \vec{E}$ and $k_1, k_2 \in \mathbb{N}_0$ with $k_1 \neq k_2$ and $\delta^{k_1}(e) = \delta^{k_2}(e)$. Then $|k_1 - k_2| \geq |[e]_\delta|$.

(ii) For all $e \in \vec{E}$ we have $\delta^{|[e]_\delta|}(e) = e$.

Proof. (i): Assume for a moment that there exist $e \in \vec{E}$ and $k_1, k_2 \in \mathbb{N}$ with $\delta^{k_1}(e) = \delta^{k_2}(e)$ and $0 < |k_1 - k_2| < |[e]_\delta|$. Without loss of generality let $k_1 > k_2$. We have $\delta^{k_1-k_2}(\delta^{k_2}(e)) = \delta^{k_1}(e) = \delta^{k_2}(e)$ and via induction for every $s \in \mathbb{N}$, we get $\delta^{s(k_1-k_2)}(\delta^{k_2}(e)) = \delta^{k_2}(e)$. For the set $M := \{\delta^k(e) | k_2 \leq k < k_1\}$, we have $|M| \leq k_1 - k_2 < |[e]_\delta|$. But on the other hand, we also have $[e]_\delta \subseteq M$: let $e' \in [e]_\delta = [\delta^{k_2}(e)]_\delta$ and let $\ell \in \mathbb{N}$ with $e' = \delta^\ell(\delta^{k_2}(e))$. Then there exist unique $s, r \in \mathbb{N}_0$ with $0 \leq r < k_1 - k_2$ and $\ell = s(k_1 - k_2) + r$. So

$$e' = \delta^n(\delta^{k_2}(e)) = \delta^r(\delta^{s(k_1-k_2)}(\delta^{k_2}(e))) = \delta^r(\delta^{k_2}(e)) = \delta^{k_2+r}(e) \in M.$$

Now, $|M| \leq k_1 - k_2 < |[e]_\delta| \leq |M|$, a contradiction.

(ii): Assume that there exists $e \in \vec{E}$ with $\delta^{|[e]_\delta|}(e) = e' \neq e$. Define $M := \{\delta^k(e) | 1 \leq k \leq |[e]_\delta|\}$. Clearly $M \subseteq [e]_\delta$.

Case 1: $e \in M$. Then $\delta^0(e) = e = \delta^k(e)$ for some k with $1 \leq k < |[e]_\delta|$. By (i) we get $k = |k - 0| \geq |[e]_\delta|$, a contradiction.

Case 2: $e \notin M$. Then $|M| < |[e]_\delta|$, By the pigeonhole principle, there exist $1 \leq k_1, k_2 \leq |[e]_\delta|$ with $\delta^{k_1}(e) = \delta^{k_2}(e)$ in contradiction to (i). □

Theorem 2.6 (Structure Theorem)**.** *Let $\vec{G} = (V, \vec{E})$ be a directed graph with bijective successor function δ such that $e \equiv_\delta e'$ for all $e, e' \in \vec{E}$. Then δ determines an Euler tour on $\vec{G}$ in the following sense: For every $e \in \vec{E}$ the sequence $(e_{(1)}, \delta(e)_{(1)}, \ldots, \delta^{|\vec{E}|}(e)_{(1)})$ is an Euler tour on $\vec{G}$.*

Proof. Let $e \in \vec{E}$. Note that $[e]_\delta = \vec{E}$. The sequence $(e_{(1)}, \delta(e)_{(1)}, \dots, \delta^{|\vec{E}|}(e)_{(1)})$ consists of $|\vec{E}|$ edges, namely $e, \delta(e), \dots, \delta^{|\vec{E}|-1}(e)$. These edges are pairwise distinct, because otherwise we would have $\delta^{k_1}(e) = \delta^{k_2}(e)$ for some $k_1, k_2 \in \{0, \dots, |\vec{E}| - 1\}$. Hence $|k_1 - k_2| < |\vec{E}| = |[e]_\delta|$ in contradiction to Lemma 2.5 (i). So the sequence is a trail. By applying Lemma 2.5 (ii), we get $e = \delta^{|[e]_\delta|}(e) = \delta^{|\vec{E}|}(e)$, thus the trail is a tour on $\vec{G}$ and since it has length $|\vec{E}|$, it is an Euler tour on $\vec{G}$. □

Before we start with a detailed memory- and correctness analysis, we show that at the end of the algorithm, every edge $\{u, v\} \in E$ has been written on the output tape exactly once, either in the form (u, v) or in the form (v, u).

Lemma 2.7. *(i) After each processing of an edge in the algorithm* EULER-TOUR *(lines 2 to 5), the graph $G_{\text{int}} = (V, E_{\text{int}})$ is cycle-free, so $|E_{\text{int}}| \leq n$. If all nodes have even degree in G, after completion of* EULER-TOUR*, $E_{\text{int}} = \varnothing$.*

(ii) If all nodes have even degree in G, after completion of EULER-TOUR *every edge $\{u, v\} \in E$ has been written on the output tape either in the form (u, v, s) or in the form (v, u, s) for some $s \in V$.*

Proof. We start by proving the first part of (i) via induction over the number of already processed edges. If there are no edges processed so far, then $E_{\text{int}} = \varnothing$, so G_{int} is cycle-free. Now let $k \in [|E|] \cup \{0\}$, let G_k, G_{k+1} denote G_{int} after k resp. $k + 1$ edges have been processed and let G_k be cycle-free. Let e denote the $(k + 1)$-th processed edge. When e is added to G_{int}, it may produce a cycle C. If e does not produce a cycle, then $G_{k+1} = G_k \cup \{e\}$ is cycle-free and we are done. If e produces a cycle C, then according to lines 6, 7 in MERGE-CYCLE all C-edges are deleted from E_{int}. Because $e \in C$, we get $G_{k+1} = (G_k \cup \{e\}) \setminus C \subseteq G_k$ and we are done by the induction hypothesis.

Now assume for a moment that $E_{\text{int}} \neq \varnothing$ at the end of EULER-TOUR. We know that G_{int} is cycle-free at this time, so G_{int} contains a node with odd degree in G_{int}. Because we always delete whole cycles, the degree of this node in G has to be odd as well, but then G is not an Eulerian graph. In this case we might output a message that G does not contain an Euler tour.

(ii). During the execution of EULER-TOUR every edge from E is added to E_{int} at some point of time and there is only one way for an edge to be deleted from E_{int} again, namely in line 7 of the procedure MERGE-CYCLE. At that time the edge has either been written on the output tape by the procedure MERGE or WRITE (in which case we are done), or it has been added to F in the procedure NEW-NODES. In that case it is written on the output tape in WRITE-F. Because, according to (i), $E_{\text{int}} = \varnothing$ at the end of EULER-TOUR,by then every edge must have been written on the output tape in exactly one of these ways. □

2.5.2 Memory requirement

Proof of Lemma 2.1. We consider the different variables and sets.

Variable c: c is initialized with 0 and changed in the procedure UPDATE if and only if $M = \varnothing$ at that time. This only happens if in the procedure CONSTRUCT-J-M for every node v of the considered cycle, we have $t(v) = 0$, which means that none of the cycle nodes was considered before. This case can occur at most $n/3$ times during the algorithm, because there are at most $n/3$ node disjoint cycles in G. So $c \leq n/3$ and $\log n$ bits suffice to store c.

Variable $s(v)$: With this variable we store the label of a node, so for fixed $v \in V$, $\log n$ bits suffice and altogether $n \log n$ bits suffice.

Variable $t(v)$: We prove that for any $v \in V$ $t(v) \leq n$ at any time: Assume for a moment that this is not the case. Let T be the first point of time at which $t(v)$ is set to a value $> n$ for some $v \in V$. $t(v)$ is only changed in the procedure UPDATE, line 9 or 11. In both cases $t(v)$ is set to a which is either c (line 4) or $\min(M)$ (line 6). We already showed $c \leq n/3 < n$. Hence, by our assumption, $\min(M) > n$ at that time. But this implies that at the time of construction of M, there already existed a node $u \in V$ with $t(u) > n$ (procedure CONSTRUCT-J-M, line 3) in contradiction to the choice of T.

Sets E_{int}, F, J, M: Because a single element of each of these sets can be stored in $\log n$ bits, it suffices to show that the cardinalities of these sets do not exceed n. For E_{int}, this is shown in Lemma 2.7. For J and M, this follows directly from the construction (procedure CONSTRUCT-J-M). In the set F, for every node only the first edge entering this node is stored (procedure NEW-NODES, lines 2 and 4), so clearly $|F| \leq n$. □

2.5.3 Correctness

In this subsection, we prove by a series of lemmas that the output successor function δ^* determines an Euler tour on G, provided that G is Eulerian (Theorem 2.2). This is done with the help of our structure theorem (Theorem 2.6), where bijectivity of δ^* and the condition that δ^* induces only one equivalence class is required. In the following, we show that these assumptions are true for δ^* by generating a sequence of bijective successor functions $\delta^*_0, \ldots, \delta^*_N$ such that $\delta^*_N = \delta^*$ and δ^*_{i+1} emerges from δ^*_i by swapping of edge successors.

Lemma 2.7 (ii) induces an orientation on E, which we call R^*: For all $\{u, v\} \in E$, we define

$$R^*(\{u, v\}) := \begin{cases} (u, v) & \text{if } (u, v) \text{ has been written on the output tape} \\ (v, u) & \text{if } (v, u) \text{ has been written on the output tape.} \end{cases}$$

Let $C_1, \ldots, C_N$ denote the cycles found by the algorithm EULER-TOUR (lines 2-5) in chronological order. We use this ordering only for the sake of analysis. Note that the cycles $C_1, \ldots, C_N$ form a partition of $\vec{E}$. For $k \in [N]$ and a variable $x \in \{s(v), t(v), \ldots | v \in V\}$, we denote by x_k the value of x after the k-th call of MERGE-CYCLE. With x_0 we denote the initial value of x.

Definition 2.8. *For each $i \in [N]$, let $C_i = (v^{(i)}_1, \ldots, v^{(i)}_{\ell_i})$ the cycle supplied to* MERGE-CYCLE. *Define $\delta^c_i : E(C_i) \to E(C_i)$ by $\delta^c_i(v^{(i)}_j, v^{(i)}_{j+1}) := (v^{(i)}_{j+1}, v^{(i)}_{j+2})$ for every $j \in [\ell_i]$ and define the successor function $\delta^c : R^*(E) \to R^*(E)$ by $\delta^c|_{E(C_i)} := \delta^c_i$ for all $i \in [N]$.*

So δ^c is the canonical successor function induced by the cycles $C_1, \dots, C_N$.

Lemma 2.9. *(i) The successor function δ^c is bijective.*

(ii) For any two edges e, e' we have $e \equiv_{\delta^c} e' \Leftrightarrow \exists i \in [N] : e, e' \in C_i$.

Proof. (i). We first show that δ^c is surjective: Let $e \in R^*(E)$. Then e belongs to some cycle $C_k = (v_1^{(k)}, \dots, v_{\ell_k}^{(k)})$ for some $k \in [N]$. Hence $e = (v_i^{(k)}, v_{i+1}^{(k)})$ for some $i \in [\ell_k]$. Then $\delta(v_{i-1}^{(k)}, v_i^{(k)}) = (v_i^{(k)}, v_{i+1}^{(k)}) = e$. Because $R^*(E)$ is finite, δ^c is bijective.

(ii). Let $e, e' \in R^*(E)$ with $e \equiv_{\delta^c} e'$ and $k \in [N]$ such that $e \in C_k$. Since $\delta^c(E(C_k)) = E(C_k)$, it follows that $e' \in C_k$. Hence, there exist $i, j \in [\ell_k]$ with $e = (v_i^{(k)}, v_{i+1}^{(k)})$ and $e' = (v_j^{(k)}, v_{j+1}^{(k)})$. W.l.o.g. let $i < j$ and set $r := j - i$. Then $(\delta^c)^r(e) = e'$, so $e \equiv_{\delta^c} e'$. □

Let $k \in \{0, \dots, N\}$. We consider the time right after the k-th iteration of Merge-Cycle. For $k = 0$ this means the very beginning of the algorithm. We call edges from $\bigcup_{i=1}^{k} E(C_i)$ *processed edges*, since those edges have already been loaded into E_{int} and then have been deleted from there. All processed edges can be divided into two types:

- Type A: The edge has been written on the output tape with a dedicated successor.
- Type B: The edge has been added to F.

These are the only possible cases for processed edges, because an edge which is deleted from E_{int} is either written on the output tape or added to F (procedure Write). This leads to the following definition.

Definition 2.10. *For every $k \in \{0, \dots, N\}$ define the function $\delta_k : \bigcup_{i=1}^{k} E(C_i) \to \bigcup_{i=1}^{k} E(C_i)$ by*

$$\delta_k((u,v)) := \begin{cases} e' & \text{if } (u,v) \text{ is of type A with successor } e' \\ (v, s(v)) & \text{if } (u,v) \text{ is of type B} \end{cases}$$

$$\text{and define } \delta_k^* := \begin{cases} \delta_k \text{ on } \bigcup_{i=1}^{k} E(C_i) \\ \delta^c \text{ on } \bigcup_{i=k+1}^{N} E(C_i). \end{cases}$$

Note that $\delta_0^* = \delta^c$ and $\delta_N^* = \delta^*$.

Lemma 2.11. *Let $k, \ell \in \{0, \dots, N\}$ with $k < \ell$. Then for any $v, v' \in V$, $e \in R^*(E)$, we have*

(i) If $t_k(v) = t_k(v') \neq 0$, then $t_\ell(v) = t_\ell(v')$.

(ii) If $e \in C_\ell$, then $[e]_{\delta_k^*} = [e]_{\delta^c}$.

Proof. (i). Let $v, v' \in V$ with $t_k(v) = t_k(v') \neq 0$. Assume for a moment that $t_\ell(v) \neq t_\ell(v')$. Then there exists $k \leq k' < \ell$ such that $t_{k'}(v) = t_{k'}(v')$ and $t_{k'+1}(v) \neq t_{k'+1}(v')$. Because $t_k(v) \neq 0$ and the t-value of v is never set to 0 after its initialization, $t_{k'}(v) \neq 0$. We take a closer look at the $(k'+1)$-th call of MERGE-CYCLE. If for a node its t-value is changed in this call, it is set to $a_{k'+1}$ (line 9 or 11 in UPDATE), so we may assume that $t_{k'+1}(v) = a_{k'+1} \neq t_{k'+1}(v')$. But this implies that $t_{k'}(v) \in M$ or $v \in C_{k'+1}$. In the latter case, since $t_{k'}(v) \neq 0$, we also have $t_{k'}(v) \in M$ (procedure CONSTRUCT-J-M). But then, $t_{k'}(v') = t_{k'}(v) \in M$ and therefore $t_{k'+1}(v') = a_{k'+1} = t_{k'+1}(v)$, in contradiction to our assumption.

(ii). Let $e \in C_\ell$. With Lemma 2.9 *(ii)*, we get $[e]_{\delta^c} = E(C_\ell)$. Since $\ell > k$, we have $\delta_k^*(e') = \delta^c(e')$ for any $e' \in C_\ell$. Hence, $\delta_k^*(e) = \delta^c(e)$ and by induction we get $(\delta_k^*)^j(e) = (\delta^c)^j(e) \in C_\ell$ for any $j \geq 1$, which proves the claim. □

The next lemma describes the cycle merging in terms of equivalence classes.

Lemma 2.12. *Let $\vec{G} = (V, \vec{E})$ be a directed graph with bijective successor function δ and the related equivalence relation $\equiv_\delta$. Let $r \in \mathbb{N}$ and $e_1, \dots, e_r \in \vec{E}$ with $e_i \equiv_\delta e_j$ for every $i, j \in [r]$. Let $e'_1, \dots, e'_r \in \vec{E}$ with $e'_i \not\equiv_\delta e'_j$ and $e_i \not\equiv_\delta e'_i$ for every $i, j \in [r]$. Let δ' be a successor function on $\vec{G}$ with $\delta'(e) = \delta(e)$ for every $e \in \vec{E} \setminus \{e_1, \dots, e_r, e'_1, \dots, e'_r\}$ and $\delta'(e_i) = \delta(e'_i)$ and $\delta'(e'_i) = \delta(e_i)$ for any $i \in [r]$. Then, δ' is bijective and*

$$[e_1]_{\delta'} = \bigcup_{i=1}^{r} [e'_i]_\delta \cup [e_1]_\delta \tag{P1}$$

$$[e]_{\delta'} = [e]_\delta \text{ for any } e \in \vec{E} \setminus [e_1]_{\delta'}. \tag{P2}$$

Let us briefly explain the meaning of the quite technical notation in this lemma. The reader may think of $e_1, \dots, e_r$ as edges of the same cycle C. The edges $e'_1, \dots, e'_r$ are edges that do not belong to C, but to several tours that share at least one node with C, such that $(e_i)_{(2)} = (e'_i)_{(2)}$ for all $i \in [r]$. The condition $e'_i \not\equiv_\delta e'_j$ reflects the fact that we have to choose exactly one common node per tour for merging, as already explained in Section 2.3, see Figure 2.2. We obtain the new successor function δ' by performing the successor swapping as described in step 1.4. in Section 2.3.3. $(P1)$ tells us that via this swapping all associated equivalence classes are merged which means that all affected tours become one big tour. $(P2)$ makes sure that tours that don't share nodes with C are not changed at all.

Proof. First we note that δ' is bijective, because δ is bijective. We prove the lemma via induction over r. Let $r = 1$. To shorten notation, we write e and e' instead of e_1 and e'_1. We start with proving

$$[e]_{\delta'} \subseteq [e]_\delta \cup [e']_\delta. \tag{2.1}$$

We show that for any $e'' \in [e]_\delta \cup [e']_\delta$, we have $\delta'(e'') \in [e]_\delta \cup [e']_\delta$: Let $e'' \in [e]_\delta \cup [e']_\delta$. Then there exists $k \in \mathbb{N}$ such that $e'' = \delta^k(e)$ or $e'' = \delta^k(e')$. If $e'' \in \{e, e'\}$, then $\delta'(e'') \in$

$\{\delta(e), \delta(e')\}$. Otherwise $\delta'(e'') = \delta(e'') = \delta^{k+1}(e)$ or $\delta'(e'') = \delta^{k+1}(e')$, respectively. In each case we have $\delta'(e'') \in [e]_\delta \cup [e']_\delta$. Since $e \in [e]_\delta \cup [e']_\delta$, it follows by induction on n that $(\delta')^n(e) \in [e]_\delta \cup [e']_\delta$ for any $n \in \mathbb{N}$, so $[e]_{\delta'} \subseteq [e]_\delta \cup [e']_\delta$.

Next, we show

$$[e]_\delta \cup [e']_\delta \subseteq [e']_{\delta'}. \tag{2.2}$$

Let $e'' \in [e']_\delta$. Then there exists $k \in \{1, \dots, |[e']_\delta|\}$ with $e'' = \delta^k(e')$. Since by assumption $e \notin [e']_\delta$ and by Lemma 2.5 (i) $\delta^\ell(e') \neq e'$ for all $\ell \in \{1, \dots, k-1\}$, we have

$$\delta^k(e') = \delta(\delta^{k-1}(e')) = \delta'(\delta^{k-1}(e')) = \delta'(\delta'(\delta^{k-2}(e'))) = \cdots = (\delta')^{k-1}(\delta(e')).$$

Hence, $e'' = \delta^k(e') = (\delta')^{k-1}(\delta(e')) = (\delta')^{k-1}(\delta'(e)) = (\delta')^k(e) \in [e]_{\delta'}$. So we have

$$[e']_\delta \subseteq [e]_{\delta'} \tag{2.3}$$

and analogously we get

$$[e]_\delta \subseteq [e']_{\delta'}, \tag{2.4}$$

By (2.3) $\delta(e') \in [e']_\delta \subseteq [e]_{\delta'}$, so $[\delta(e')]_{\delta'} = [e]_{\delta'}$ and thus using the assumption $\delta(e') = \delta'(e)$,

$$[e]_{\delta'} = [\delta(e')]_{\delta'} = [\delta'(e)]_{\delta'} = [e']_{\delta'}. \tag{2.5}$$

Combining (2.3), (2.4), and (2.5), we proved (2.2). With (2.1), (2.2), and (2.5), we have

$$[e]_{\delta'} \subseteq [e]_\delta \cup [e']_\delta \subseteq [e']_{\delta'} = [e]_{\delta'},$$

so property (P1) is proven. For the proof of (P2), let $e'' \in \vec{E}$ with $e'' \not\equiv_\delta e, e'' \not\equiv_\delta e'$. Then, $\delta^k(e'') \notin \{e, e''\}$ for all $k \in \mathbb{N}$, so we get

$$\delta^k(e'') = \delta(\delta^{k-1}(e'')) = \delta'(\delta^{k-1}(e'')) = \cdots = (\delta')^k(e'').$$

But this implies $[e'']_\delta = [e'']_{\delta'}$.

Induction step: Now let $r \in \mathbb{N}$ and let the claim be true for all $k \leq r \in \mathbb{N}$. Let $e_1, \dots, e_{r+1} \in \vec{E}$ with $e_i \equiv_\delta e_j$ for every $i, j \in [r+1]$. Let $e'_1, \dots, e'_{r+1} \in \vec{E}$ with $e'_i \not\equiv_\delta e'_j$ and $e'_i \not\equiv_\delta e_i$ for every $i \neq j \in [r+1]$. Let δ' be a successor function on $\vec{G}$ with $\delta'(e) = \delta(e)$ for every $e \in \vec{E} \setminus \{e_1, \dots, e_{r+1}, e'_1, \dots e'_{r+1}\}$ and $\delta'(e_i) = \delta(e'_i)$ and $\delta'(e'_i) = \delta(e_i)$ for every $i \in [r+1]$. We define a successor function δ_r for $\vec{G}$ by

$$\delta_r := \begin{cases} \delta' \text{ on } \vec{E} \setminus \{e_{r+1}, e'_{r+1}\} \\ \delta \text{ on } \{e_{r+1}, e'_{r+1}\}. \end{cases}$$

With the induction hypothesis applied to δ and δ_r, we get by (P1)

$$[e_1]_{\delta_r} = \bigcup_{i=1}^{r} [e'_i]_\delta \cup [e_1]_\delta \tag{2.6}$$

and by (P2)

$$[e'_{r+1}]_{\delta_r} = [e'_{r+1}]_\delta. \tag{2.7}$$

Now we apply the induction hypothesis to δ_r and δ' as follows: We take δ_r instead of δ, δ' remains, $r = 1$ and e_1 resp. e'_1 are replaced by e_{r+1} resp. e'_{r+1}. Then P1 gives

$$[e_{r+1}]_{\delta'} = [e'_{r+1}]_{\delta_r} \cup [e_{r+1}]_{\delta_r}. \tag{2.8}$$

Since $e_1 \equiv_\delta e_{r+1}$, we get with (2.6)

$$e_{r+1} \in [e_{r+1}]_\delta = [e_1]_\delta \subseteq [e_1]_{\delta_r},$$

which implies

$$[e_{r+1}]_{\delta_r} = [e_1]_{\delta_r}. \tag{2.9}$$

Summarizing, we have

$$\begin{aligned} [e_{r+1}]_{\delta'} &\overset{(2.8)}{=} [e'_{r+1}]_{\delta_r} \cup [e_{r+1}]_{\delta_r} \\ &\overset{(2.9)}{=} [e'_{r+1}]_{\delta_r} \cup [e_1]_{\delta_r} \qquad (2.10) \\ &\overset{(2.6)}{=} [e'_{r+1}]_{\delta_r} \cup \Big(\bigcup_{i=1}^{r} [e'_i]_\delta \cup [e_1]_\delta \Big) \\ &\overset{(2.7)}{=} [e'_{r+1}]_{\delta} \cup \Big(\bigcup_{i=1}^{r} [e'_i]_\delta \cup [e_1]_\delta \Big) \\ &= \bigcup_{i=1}^{r+1} [e'_i]_\delta \cup [e_1]_\delta. \qquad (2.11) \end{aligned}$$

So (P1) is proved, if $[e_{r+1}]_{\delta'} = [e_1]_{\delta'}$. By (2.11) $[e_1]_\delta \subseteq [e_{r+1}]_{\delta'}$, so $e_1 \in [e_{r+1}]_{\delta'}$ and hence

$$[e_{r+1}]_{\delta'} = [e_1]_{\delta'}. \tag{2.12}$$

For the proof of (P2), let $e \in \vec{E} \backslash [e_1]_{\delta'}$. Since $e \notin [e_1]_{\delta'}$, by (2.10) and (2.12) $e \notin [e_1]_{\delta_r}$. Applying the induction hypothesis to δ and δ_r, (P2) gives us $[e]_{\delta_r} = [e]_\delta$. We know that $[e_{r+1}]_{\delta'} = [e_1]_{\delta'}$, so $e \notin [e_{r+1}]_{\delta'}$. As above, we apply the induction hypothesis to δ_r and δ' and get $[e]_{\delta'} = [e]_{\delta_r}$. Altogether, $[e]_{\delta'} = [e]_{\delta_r} = [e_\delta]$. □

Lemma 2.13. *Let $k \in \{0, \dots, N\}$. Then, δ^*_k is bijective and for any $(u, v), (u', v') \in R^*(E)$, we have*

(i) If $(u, v), (u', v')$ are processed edges, then $(u, v) \equiv_{\delta^*_k} (u', v') \Leftrightarrow t_k(u) = t_k(u')$.

(ii) If (u, v) is a processed edge, then $t_k(u) = t_k(v)$.

(iii) If $t_k(u) = 0$, then $(u, v) \equiv_{\delta^*_k} (u', v') \Leftrightarrow (u, v) \equiv_{\delta^c} (u', v')$.

Claim (i) says that the procedure UPDATE works correctly, i.e., that the t-value of a node (if it isn't 0) always represents the tour it currently is associated to. Claim (ii) says that after an edge has been processed, both of its nodes are associated to the same tour. So, after the algorithm has terminated, every node of G is in the same tour as its neighbor.

Proof. We prove all claims via one induction over k. For $k = 0$ we have $\delta_0^* = \delta^c$ which is bijective (Lemma 2.9). Moreover, no edge has been processed so far, so (i) and (ii) are trivially fulfilled and (iii) follows directly from $\delta_0^* = \delta^c$.

Now let all of the claims be true for a fixed $k \in \{0, \ldots, N-1\}$. We start with proving the bijectivity and (i) for $k+1$.

To do so we take a closer look at the $(k+1)$-th call of MERGE-CYCLE. Suppose that $\delta_k^* \neq \delta_{k+1}^*$. This change must happen in one of the procedures NEW-NODES, MERGE or WRITE, since these are the only procedures in which edges are written on the output tape or added to F. First, note that for every edge e written on the output tape during WRITE or added to F in NEW-NODES it holds $\delta_k^*(e) = \delta_{k+1}^*(e)$:

If $e = (v_i^{(k+1)}, v_{i+1}^{(k+1)})$ is written on the output tape during WRITE, it is written in the form $(v_i^{(k+1)}, v_{i+1}^{(k+1)}, v_{i+2}^{(k+1)})$, so $\delta_{k+1}^*(e) = (v_{i+1}^{(k+1)}, v_{i+2}^{(k+1)}) = \delta^c(e) = \delta_k^*(e)$.

If $e = (v_{i-1}^{(k+1)}, v_i^{(k+1)})$ is added to F during NEW-NODES, it becomes a type-B-edge at this point, so $\delta_{k+1}^*(e) = (v_i^{(k+1)}, s_k(v_i^{(k+1)}))$. Furthermore, $s_{k+1}(v_i^{(k+1)})$ is set to $v_{i+1}^{(k+1)}$ in line 3, so $\delta_{k+1}^*(e) = (v_i^{(k+1)}, v_{i+1}^{(k+1)}) = \delta^c(e) = \delta_k^*(e)$.

So it suffices to consider the procedure MERGE: Here we process every node from the set J_{k+1}. Let $r := |J_{k+1}|$, for instance $J = \{w_1, \ldots, w_r\}$. Each of these nodes w_i has been processed before, hence, there is a unique edge in C_{k+1} that points to w_i and which we denote by e_i. Moreover, there is a unique edge in F_k pointing to w_i and which we denote by e_i'. Now let $i \in [r]$. We process w_i in two steps:

Step 1: $(w_i, s(w_i))$ is marked as successor of e_i. So directly after this step, e_i and e_i' share the same successor, while the out-going edge of w_i in C_{k+1} has lost its predecessor.

Step 2: $s(w_i)$ is set to the next node on C_{k+1}, so that the out-going edge of w_i becomes the successor of e_i' in C_{k+1} concerning δ_{k+1}^*.

In these two steps we swapped the successors of e_i and e_i' and did not change anything else, so we get

$$\delta_{k+1}^*(e) = \delta_k^*(e) \text{ for any } e \in \vec{E} \setminus \{e_1, \ldots, e_r, e_1', \ldots, e_r'\}$$

and for any $i \in [r]$

$$\delta_{k+1}^*(e_i) = \delta_k^*(e_i') \text{ and } \delta_{k+1}^*(e_i') = \delta_k^*(e_i).$$

Let $i, j \in [r]$ with $i \neq j$. We have $e_i \equiv_{\delta_k^*} e_j$, since $e_j \in C_{k+1} = [e_i]_{\delta^c} = [e_i]_{\delta_k^*}$. We also have $e_i' \not\equiv_{\delta_k^*} e_j'$, which follows from $t_k(w_i) \neq t_k(w_j)$ (CONSTRUCT-J-M, line 4) together with the induction hypothesis. Finally we have $e_i \not\equiv_{\delta_k^*} e_i'$, because $e_i' \notin E(C_{k+1}) = [e_i]_{\delta^c} = [e_i]_{\delta_k^*}$.

So we can apply Lemma 2.12 with $\delta = \delta_k^*$ and $\delta' = \delta_{k+1}^*$. This ensures the bijectivity of δ_{k+1}^* and for every processed edge e by property P2 we get

$$\begin{aligned} e \in [e_1]_{\delta_{k+1}^*} &\Leftrightarrow e \in \bigcup_{i=1}^{r} [e_i']_{\delta_k^*} \cup [e_1]_{\delta_k^*} \Leftrightarrow t_k(e_{(1)}) \in M_k \vee e \in C_{k+1} \\ &\Leftrightarrow t_{k+1}(e_{(1)}) = a_{k+1}, \end{aligned}$$

where the second equivalence follows with the induction hypothesis: We have

$$
\begin{aligned}
e \in \bigcup_{i=1}^{r} [e'_i]_{\delta_k^*} &\Leftrightarrow \exists i \in [r] : e \equiv_{\delta_k^*} e'_i \\
&\Leftrightarrow \exists i \in [r] : t_k(e_{(1)}) = t_k((e'_i)_{(1)}) \\
&\Leftrightarrow \exists i \in [r] : t_k(e_{(1)}) = t_k((e'_i)_{(2)}) \in M_k
\end{aligned}
$$

Analogously we get

$$
\begin{aligned}
e \notin [e_1]_{\delta_{k+1}^*} &\Leftrightarrow e \notin \bigcup_{i=1}^{r} [e'_i]_{\delta_k^*} \cup [e_1]_{\delta_k^*} \Leftrightarrow t_k(e_{(1)}) \notin M_k \wedge e \notin C_{k+1} \\
&\Leftrightarrow t_{k+1}(e_{(1)}) = t_k(e_{(1)}) \neq a_{k+1}.
\end{aligned}
$$

Now we are able to complete the proof of (i): Let $(u, v), (u', v')$ be processed edges.

Case 1: $(u, v), (u', v') \in [e_1]_{\delta_{k+1}^*}$. Then $(u, v) \equiv_{\delta_{k+1}^*} (u', v')$ and $t(u) = a_{k+1} = t(u')$.

Case 2: $(u, v) \in [e_1]_{\delta_{k+1}^*}, (u', v') \notin [e_1]_{\delta_{k+1}^*}$. Then $(u, v) \not\equiv_{\delta_{k+1}^*} (u', v')$ and $t(u) = a_{k+1} \neq t(u')$.

Case 3: $(u, v) \notin [e_1]_{\delta_{k+1}^*}, (u', v') \in [e_1]_{\delta_{k+1}^*}$. Analog to case 2.

Case 4: $(u, v), (u', v') \notin [e_1]_{\delta_{k+1}^*}$. Then $t_{k+1}(u) = t_k(u), t_{k+1}(u') = t_k(u')$ and with P2 of Lemma 2.12 we get $[(u, v)]_{\delta_{k+1}^*} = [(u, v)]_{\delta_k^*}$ and $[(u', v')]_{\delta_{k+1}^*} = [(u', v')]_{\delta_k^*}$. So

$$
(u, v) \equiv_{\delta_{k+1}^*} (u', v') \Leftrightarrow (u, v) \equiv_{\delta_k^*} (u', v')
$$

$$
\Leftrightarrow t_k(u) = t_k(u') \Leftrightarrow t_{k+1}(u) = t_{k+1}(u').
$$

(ii). Let (u, v) be a processed edge. If $(u, v) \in C_{k+1}$, then at the end of MERGE-CYCLE both $t(u)$ and $t(v)$ are set to the same value a. If $(u, v) \notin C_{k+1}$, then (u, v) already was a processed edge before. By induction hypothesis (ii) we have $t_k(u) = t_k(v)$ and applying Lemma 2.11(i) we get $t_{k+1}(u) = t_{k+1}(v)$.

(iii). Let $u \in V$ with $t_{k+1}(u) = 0$. That means that u is not processed in the first $k+1$ calls of MERGE-CYCLE. Hence, $(u, v) \in E(C_\ell)$ for some $\ell > k + 1$. Lemma 2.11(ii) gives $[(u, v)]_{\delta_{k+1}^*} = [(u, v)]_{\delta^c}$, therefore we get $(u', v') \in [(u, v)]_{\delta_{k+1}^*} \Leftrightarrow (u', v') \in [(u, v)]_{\delta^c}$. □

Finally, we prove Theorem 2.2.

Proof of Theorem 2.2. According to Theorem 2.6, it suffices to show that δ^* is bijective and that $e \equiv_{\delta^*} e'$ for any $e, e' \in R^*(E)$. Remember that $\delta^* = \delta_N^*$, so by Lemma 2.13 δ^* is bijective. For the second property, let $e, e' \in R^*(E)$ with $e = (u, v)$ and $e' = (u', v')$. If G is Eulerian, it is connected, so there exists a u-u'-path P in G. For every edge on P, either the edge itself or the corresponding reversed edge has been processed during the algorithm EULER-TOUR. By Lemma 2.13 (ii), $t_N(x) = t_N(y)$ for all nodes x, y of P, hence, $t_N(u) = t_N(u')$ and by Lemma 2.13 (i), we get $e \equiv_{\delta_N^*} e'$. Since $\delta_N^* = \delta^*$, we are done. □

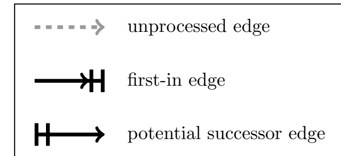
2.6 Conclusion

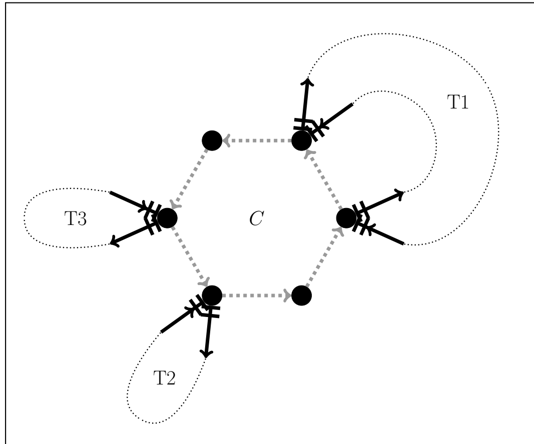
We have presented a one-pass algorithm with $\mathcal{O}(n \log(n))$ RAM for finding Euler tours in undirected graphs in the graph streaming model with an additional write-only output tape. This gives two possible directions for future work.

- Are there other well suited graph-theoretical problems, where the size of a solution exceeds $\mathcal{O}(n \operatorname{polylog}(n))$ RAM but which can be solved in the graph streaming model with an additional output tape?
- Can our technique of linking tours to equivalence classes be useful for other routing problems?

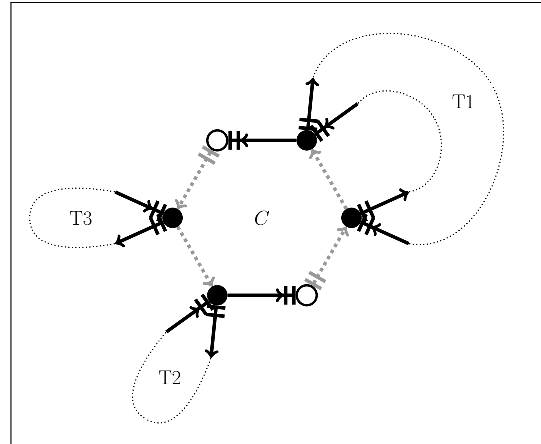
2.7 Appendix

On the following two pages we present a working example for the method Merge-Cycle that corresponds to the steps 1.2. to 1.6. in our high level description. Note that every node has at most one in-going first-in edge and one out-going potential successor edge at a time.

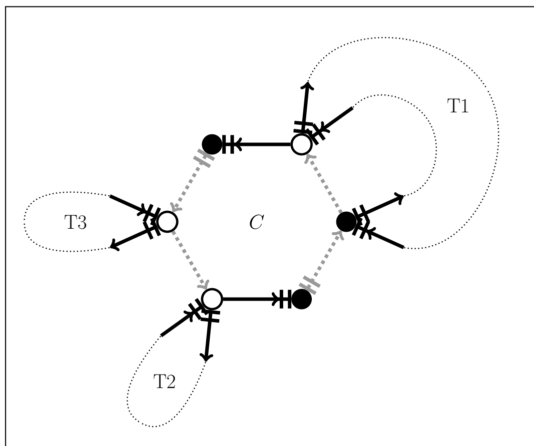




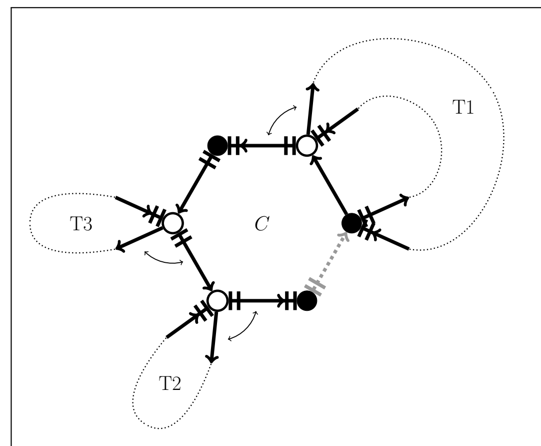
A cycle C has been found.



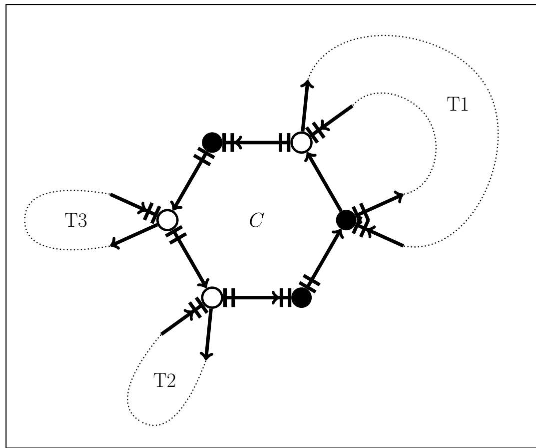
Step 1.2. For every new node the in-going edge becomes a first-in edge and the outgoing edge becomes a potential successor.



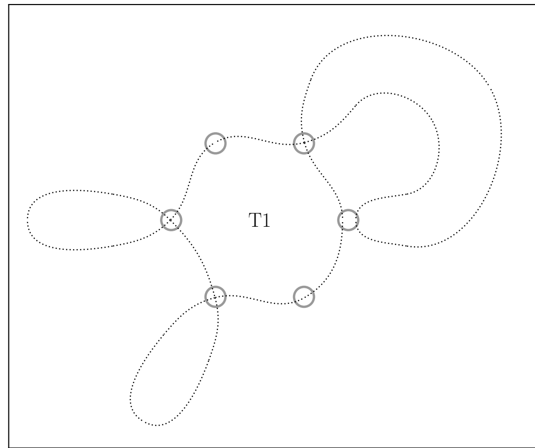
Step 1.3. For each intersecting Tour T_1, T_2, T_3 we choose one common node.



Step 1.4. The successors of the chosen nodes are swapped with potential successor edges.



Step 1.5. For the rest of the edges the successor stays the same.



Step 1.6. The tours and the cycle have been merged to one tour.

Chapter 3

Euler Tours in the StrSort Model

3.1 Introduction

In the last chapter we gave an overview of the definition and some applications of Euler tours. We showed how to handle the Euler tour problem in the graph streaming model. Our algorithm found an Euler tour and stored it in form of a successor function on an output tape. This can be disadvantageous if the solution has to be further processed in another streaming algorithm, since the edges are not actually in the right order on the output tape. In applications we want the edges to be sorted on the stream, which cannot be done in semi-streaming. This problem calls for a relaxation of the graph streaming model.

3.1.1 StrSort and W-stream models

Aggarwal et al. [29, 1] presented a less restrictive streaming model, called *StrSort-model*. It consists of alternating streaming and sorting passes. A streaming pass consists of a Turing machine with local memory of size M and two tapes. On one tape, the Turing machine reads a sequence $S = x_1, \ldots, x_k$ of $k \in \mathbb{N}$ items. On the other tape, an output stream is written. On both tapes, the Turing machine can move only left-to-right. In a sorting pass, a Turing machine with a global partial order sorts items on a tape according to this order and returns the sorted items as output.

Definition 3.1. *StrSort*(p_{Str}, p_{Sort}, M) *is the class of functions computable by the composition of up to* p_{Str} *streaming passes and* p_{Sort} *sorting passes, each with memory* M*, where:*

- *the local memory is maintained between streaming passes*
- *streams produced at intermediate stages are of length* $\mathcal{O}(a)$*, where* a *is the length of the input stream.*

Using only $\mathcal{O}(\text{polylog}(n))$ memory space and $\mathcal{O}(\text{polylog}(n))$ passes is sufficient for solving some graph problems in this streaming model, such as minimum spanning tree,

maximal independent set and mincut [29]. A more recent paper concerns the construction of spanners in weighted graphs [11]. This motivates the following definition of Aggarwal et al. of the class *PL-StrSort*:

Definition 3.2. *PL-StrSort* $:= \cup_k$ *StrSort* $(O(\log^k n), O(\log^k n), O(\log^k n))$

Demetrescu et al. [10] showed for a few graph problems (e.g. single source shortest paths in directed graphs) that the sorting steps are not necessary. In the so-called W-stream model, which uses only the streaming steps (i.e. $\text{StrSort}(p_{\text{Str}}, 0, M)$), they show a tradeoff between internal memory and streaming passes for undirected connectivity and single-source shortest paths in directed graphs.

3.1.2 Previous results

In the RAM model, finding Euler tours in polynomial time is relatively easy, and various algorithms are known. But the problem gets significantly more complicated considering big data graphs in streaming or external memory models. Sun and Woodruff [30] showed that a one-pass algorithm for computing an Euler tour would need $\Omega(n \log(n))$ RAM. Atallah and Vishkin [4] gave an algorithm for computing the *successor function* of an Euler tour in PRAM with running time $\mathcal{O}(\log(n))$ and $n + m$ processors. Since PRAM algorithms can be transferred to W-streaming [9], this result implicitly gives an algorithm solving the Euler tour problem in $\mathcal{O}(m \text{ polylog}(n)/s)$ number of passes with s bits of RAM. Further, Demetrescu et al. [9] showed an upper bound of $\mathcal{O}(n \log(n)/s)$ passes for finding Euler tours in the very special case of a tree (by doubling the tree edges). But to the best of our knowledge no algorithm for computing sorted Euler tours in general graphs in a streaming model has been presented so far.

3.1.3 Our contribution

We give a 2-step StrSort-algorithm *EulerStr* for finding an Euler tour in a graph $G = (V, E)$ with $n := |V|$ and $m := |E|$. The first and preprocessing step is a single pass W-stream algorithm with memory space $\mathcal{O}(n \log(n))$, a common bound in the semi-streaming context. In this step we compute an input stream of edges supplying additional information, thus call them *information edges*. The second and main step is a PL-StrSort algorithm with $O(\log(n))$ alternating streaming and sorting passes and $O(\log(n))$ memory space. The stream length will be $\mathcal{O}(m \log(n))$ the whole time. We output the edges in the order given by the computed Euler tour.

A standard technique to compute Euler tours is to start with (sub-)tours and to merge them. The merging step becomes efficient when we represent the tours through vertices of a suitable tree. Unfortunately, its size can be $\Omega(m)$, thus it exceeds the memory space in streaming models. Our technical innovation is to construct the tree keeping only $\mathcal{O}(n \log(n))$ data in RAM, steadily outsourcing processed data on the output stream, and finally to use the tree structure for an efficient sorting of the edges. Our result might be a basis for solving other problems in combinatorial optimization under the big data issue where Euler tours are needed in subprocedures. The open question arising from our

result is whether or not the problem can be solved solely within polylogarithmic RAM and passes (PL-StrSort algorithm).

3.2 Preliminaries

For $k \in \mathbb{N}$ let $[k] := \{1, \ldots, k\}$. Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E. A *walk* of length k is a sequence $v_1, \ldots, v_{k+1}$ of vertices, where $e_i := \{v_i, v_{i+1}\} \in E$ for all $i \in [k]$. For simplification, we omit the vertices and denote the walk by the sequence of edges $e_1, \ldots, e_k$. A *trail* is a walk without repeating edges, i.e. for all $i, j \in [k]$: $i \neq j \Leftrightarrow e_i \neq e_j$. A *tour* is a trail with the property $v_1 = v_{k+1}$, i.e. a closed trail. An *Euler tour* is a tour that uses each edge in E exactly once. A graph that contains an Euler tour is called Eulerian. A path is a walk without repeating vertices or edges. A cycle is a tour with $v_i \neq v_j$ for all $i, j \in [k]$.

A rooted tree is a tree in which one vertex r is designated as a root. In a rooted tree, the depth of a vertex v is the length of the unique path to its root. The vertex u adjacent to v which is on the v-r-path is called predecessor of v. If for a vertex w, the vertex v is the predecessor of w, then w is called a successor of v. For a directed edge $\vec{e} = (u, v)$, u is called the tail, and v the head of $\vec{e}$.

Definition 3.3 (Out-tree)**.** *An out-tree is a rooted, directed tree, where all edges point to the respective successor.*

The input stream consists of the m edges of G, given in arbitrary order.

3.3 General idea of EulerStr

3.3.1 Tour merging in the RAM model

There are two ways to represent an Euler tour. In the graph-theoretic representation the edges are given in the order of the tour, while a weaker form is to give only a successor function, which for an edge returns its successor edge w.r.t. the Euler tour (e.g. [4]). Our goal is to compute the Euler tour in the graph-theoretic sense. Let us first explain the basic idea of creating and merging cycles in the RAM model. In Section 3.3.2, we will show why this direct approach fails in a streaming model and present our algorithmic innovation. Let $G = (V, E)$ be an undirected Eulerian graph. First, we partition E into edge-disjoint tours $C_1, \ldots, C_q$ ($q \in \mathbb{N}$) with lengths $l_1, \ldots, l_q$. For each tour, we choose a *direction* in which the tour is traversed. According to these directions, all edges become directed. Let $C_i = e_i^1, \ldots, e_i^{l_i}$ and $C_j = e_j^1, \ldots, e_j^{l_j}$ be tours with a common vertex v. It is easy to merge these tours and get the edges in the right order: let e_i^s be the edge of C_i with head v. If the tail of e_j^1 is v, the edges of C_j can be inserted in C_i right after the edge e_i^s, so the resulting tour is $e_i^1, \ldots, e_i^s, e_j^1, \ldots, e_j^{l_j}, e_i^{s+1}, \ldots, e_i^{l_i}$. This can be done iteratively, and since an Eulerian graph is connected, the result finally is a simple tour. During the merging we must ensure that in each step both tours share a vertex v and one of the tours starts with v.

To get an order in which we merge the tours by inserting edges, we construct an out-tree $\vec{T} = (W, \vec{F})$, where $W = \{w_1, \ldots, w_q\}$ and for all $i, j \in [q] : (w_i, w_j) \in \vec{F} \Rightarrow C_i$ and C_j share a common vertex in G (see Figure 3.1). Again, such a tree exists, because G is connected. If w_j is the predecessor of w_i in $\vec{T}$, we call C_j the *predecessor tour* of C_i and C_i a *successor tour* of C_j.

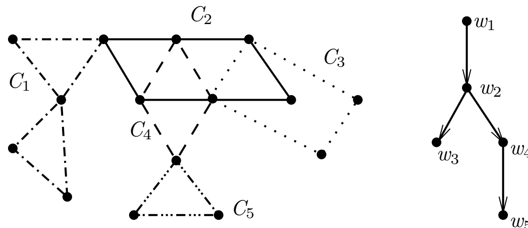


Figure 3.1: Partition G into tours $C_1, \ldots, C_5$ with a corresponding out-tree $\vec{T}$.

We merge tours along $\vec{T}$. Let w_r be the root of $\vec{T}$. For every $i \in [q] \backslash \{r\}$ we sort the edges of C_i such that the tail of the first edge is a common vertex of the predecessor tour. Once this is done, we can iteratively merge the tours along the edges of $\vec{T}$ as described above (see Figure 3.2). After each step, $\vec{T}$ can be updated accordingly. During the merging process, the tail of the first edge of every intermediate tour does not change, and every intermediate tour still has a common vertex with the predecessor tour, so the required conditions are fulfilled all the time.

High level description of tree-merging:

1. Partition the graph G into edge disjoint tours $C_1, \ldots, C_q$.
2. Create a rooted tree $\vec{T}_G$, where each vertex w_i represents a cycle C_i of G. Having an edge $\{w_i, w_j\}$ in $\vec{T}_G$ implies that the tour C_i and C_j share a common vertex in G.
3. Iteratively merge adjacent vertices in $\vec{T}_G$, while merging the represented tours in G.

3.3.2 Tour merging within limited memory

Our goal is to design an algorithm which uses one W-streaming pass with $\mathcal{O}(n \ \log(n))$ RAM followed by additional StrSort passes with only $\mathcal{O}(\log(n))$ RAM. Thus, we have to use our first pass as efficient as possible. Finding edge disjoint tours with $\mathcal{O}(n \ \log(n))$ space is fairly easy. When n edges are in local memory, the subgraph formed by those edges is guaranteed to contain at least one cycle/tour. So iteratively, up to n edges can be stored in local memory and a tour can be delivered. There are fundamental problems

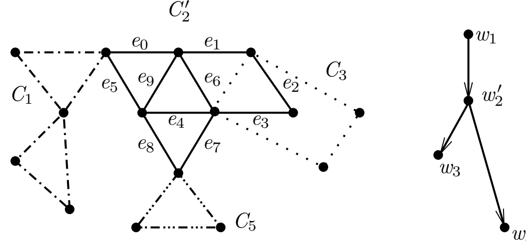


Figure 3.2: Merging of $C_2 = e_0, e_1, e_2, e_3, e_4, e_5$ and $C_4 = e_6, e_7, e_8, e_9$ to get a new tour $C_2' = e_0, \underline{e_6, e_7, e_8, e_9}, e_1, e_2, e_3, e_4, e_5$

of transferring the tree merging algorithm to the streaming environment, in particular steps 2 and 3.

- **(Tour Number Problem)** The number of tours found can be $\Theta(n^2)$, therefore the edges of $\vec{T}$ would occupy $\Theta(n^2 \ \log(n))$ memory space. Hence, parts of $\vec{T}$ have to be output during the first streaming step and before the out-tree can be constructed.
- **(Efficient Sorting Problem)** For inserting a tour C_i in the predecessor tour C_j, sorting steps with a global partial order will be used. The edges of C_i have to receive labels such that during a sorting step they are placed right behind the correct edge of C_j with a common vertex as head.

We will resolve the first problem by only keeping $\mathcal{O}(n)$ edges of $\vec{T}$ in RAM and output the other edges immediately. For the second problem, we will assign to each edge a label using additional memory space on the output stream during the W-stream step. The hard work and core of this paper is to design efficient algorithms for both problems and to prove their correctness.

Before we can state the algorithm, we introduce appropriate data structures, the graph labeling mentioned above and the notion of so called information edges.

Definition 3.4 (Graph labeling). *On the output stream, we identify an edge e_i^k of a tour C_i with length l_i with a 6-tuple*

$$e_i^k := (v_i^k, v_i^{k+1}, i, k, \bar{i}, \bar{k}) \text{ for } k \in \{1, \dots, l_i\}, \text{ where} \tag{3.1}$$

- $\{v_i^k, v_i^{k+1}\} \in E$ *and* $\bar{i}, \bar{k} \in \{0\} \cup [m]$.
- *When walking along C_i, e_i^k is passed from v_i^k to v_i^{k+1}.*
- *If $\bar{i} = 0 = \bar{k}$: k is the placement of e_i^k in C_i, i.e. e_i^k is the k-th edge of tour C_i.*

Let us briefly explain the effect of graph labeling for tour merging. Let C_j be the predecessor tour of C_i and $e_j^{k'}$ be the edge of C_j behind which the tour C_i has to be

inserted. Let the fifth and sixth entry of all edges of C_i and C_j have the value 0. Relabel the edges e_i^k of C_i to $(v_i^k, v_i^{k+1}, j, k', i, k)$. After sorting the edges lexicographically by the last four entries, C_i is directly behind $e_j^{k'}$ on the stream, so C_i and C_j are merged correctly. Now, since all edges are in one common tour, they have to be relabeled again.

Note that this labeling technique only works when the labels of C_j are not changed at the same time. Hence, during a merging step we only change labels of tours whose representing vertices in the out-tree have odd depth. This way, every predecessor tour in a tour merging has unchanged labels. With an out-tree of height $h \leq m$ we need $\mathcal{O}(\log(h))$ merging steps.

Now, we explain how the 6-tuples for graph labeling are updated by transferring values from one item to a target item. This holds also for the information edges introduced shortly. For the relabeling of the edges with only $\mathcal{O}(\log(n))$ RAM, we use a basic technique in the StrSort environment. Let $x_a = (l_a, v_a)$ and $x_b = (l_a, v_a)$ be items on the stream with labels l_a, l_b and values v_a, v_b. To change the value of x_b to v_a, we can create an additional item $y = (l_a, l_b, 0)$. After sorting by label and using the first label of y, x_a and y are consecutive on the stream, so even with $\mathcal{O}(\log(n))$ RAM we can read both items, store the value v_a in the third entry of y and then output $x_a = (l_a, v_a)$ and $y = (l_a, l_b, v_a)$. After another sorting step, where the second label of y is used, x_b and y are consecutive on the stream. Now, we can read both items and change x_b to (l_b, v_a).

As additional items we use the edges of our out-tree $\vec{T}$ in a modified version, called *'information edges'* in contrast to the *graph edges* of G.

Definition 3.5 (Information edge)**.** *For a tour C_j and its predecessor tour C_i, the information edge f_i^j is defined as follows:*

$$f_i^j := (i, j, d_i, v, p_i, o) \tag{3.2}$$

- *$d_i \in \{*\} \cup \{0, \ldots, m\}$, $v \in V$, $p_i \in \{*\} \cup [m]$ and $o \in \{0, 1\}$.*
- *f_i^j represents the edge $\{w_i, w_j\} \in \vec{T}$ and if $o = 0$, then w_i is the predecessor of w_j in $\vec{T}$. If $o = 1$, then w_i is the successor of w_j in $\vec{T}$.*
- *If $d_i \neq *$, then d_i is the depth of w_i in $\vec{T}$.*
- *v is a common vertex of C_i and C_j in G.*
- *If $p_i \neq *$, then p_i is the placement of the edge in C_i which has v as its head.*

f_i^j has the labels i, j and v for transmitting values between edges of C_i and C_j, and can transmit the needed value p_i to C_j if d_i is an even number, i.e. if the merging is actually taking place. If a value is not known, the related entry is marked with a '$*$'. The W-stream step and generation of information edges is presented in the following section.

3.4 The W-stream step

In this section, we describe the one pass W-stream step within $\mathcal{O}(n \log(n))$ memory. A detailed example is given in the appendix of the chapter. In this pass, we want to

partition the Eulerian graph G into edge-disjoint cycles and output the edges according to Definition 3.4. Additionally, we create the out-tree $\vec{T}$ and output the related information edges according to Definition 3.5. However, there will be two points that need a constant number of additional StrSort passes with $\mathcal{O}(\log(n))$ RAM. First, we cannot convert the tree in RAM into an out-tree until the tree is completed. But the tree might not fit into RAM as only $\mathcal{O}(\log(n))$ memory is available. We solve this problem by outputting information edges of $\vec{T}$ that are guaranteed to be *leaf* edges. These edges will miss the information about the depth of the predecessor. Second, when creating an edge of T, we do not know yet which vertex will be the predecessor or successor in $\vec{T}$. Since we have to constantly output tours before the out-tree is constructed, these tours will not be sorted and it may happen that the first edge does not begin with a common vertex of the predecessor tour.

Let $C_1, \dots, C_q$ be the tours found by our algorithm in this order. For $i \in [q]$ let G_i denote the subgraph of G consisting of the union of the tours $C_1, \dots, C_i$. Let $\{v_1, \dots, v_n\}$ be the vertices of G. For each $j \in [n]$, we store the following values in RAM:

- $\mathrm{fir}_j \in [m]$ is the label of the first tour that contains v_j, i.e. $v_j \notin G_{\mathrm{fir}_j - 1}$ and $v_j \in G_{\mathrm{fir}_j}$.
- $\mathrm{comp}_j \in [n]$ is the label of the connected component of v_j in the current subgraph G_i. It will be updated in each iteration step.

We start with an empty tree T. We give a high-level description of the algorithm (for the pseudo-code see the appendix).

W-stream algorithm

1. Initialization: $T := \varnothing$, $G_0 := \varnothing$, $i := 1$
2. Iteratively do, until the input stream is empty:
   - 2.1. Read the input stream until n edges are stored in RAM.
   - 2.2. Find a tour and call it C_i.
   - 2.3. Check if C_i fulfills one of the following conditions ($G_i := G_{i-1} \cup C_i$):
     - C_i contains a vertex that is not in G_{i-1}.
     - G_i has fewer connected components than G_{i-1}.
   - 2.4. If one of those conditions is fulfilled:
     - 2.4.1. Add the vertex w_i to T.
     - 2.4.2. For each connected component in G_{i-1} sharing vertices with C_i: choose such a vertex v_j, create $\{w_i, w_{\mathrm{fir}_j}\}$ in T and store v_j as common vertex of C_i and C_{fir_j}.
   - 2.5. If those conditions are not fulfilled:
     - 2.5.1. Pick one vertex v_j of C_i and output the information edge $(\mathrm{fir}_j, i, *, v_j, *, 0)$
   - 2.6. Output the edges of C_i according to Definition 3.4 and set $i := i + 1$.

3. Repeat steps 2.2. to 2.6. for the remaining edges until no more cycles can be found.

4. Let q be the number of cycles found in step 2.2. If at this point, edges of G remain in RAM, or if G_q is not connected, state that G is not Eulerian.

5. Convert T into an out-tree $\vec{T}$ and output the edges of $\vec{T}$ as information edges (6-tuples).

Now, we prove that the W-stream algorithm gives the required output within $\mathcal{O}(n \log(n))$ RAM (Theorem 3.8). First, we need two technical lemmata.

Lemma 3.6. *If G is an Eulerian graph, the W-stream algorithm outputs a partition of G into edge-disjoint tours. If G is not Eulerian, the algorithm will state this fact.*

Proof. It is well known that G is Eulerian iff the graph is connected and every vertex has even degree. Since in a tour every vertex has even degree, G is Eulerian iff G_q is connected and every vertex in $G \backslash G_q$ has even degree. The algorithm notices by the values $\text{comp}_1, \ldots, \text{comp}_n$ if G_q is connected. At the end of the algorithm, either $G \backslash G_q = \varnothing$ and G is Eulerian, or $G \backslash G_q \neq \varnothing$ and there is still an edge left in RAM. In the former case, every edge of G was written on the stream exactly once, so $C_1, \ldots, C_q$ is indeed a partition of G. In the latter case, no more tours can be found in $G \backslash G_q$, hence a vertex in $G \backslash G_q$ has a degree of 1. But then, not every vertex in $G \backslash G_q$ has even degree and G is not Eulerian. □

Lemma 3.7. *For every $i \in [q]$ and every $k, l \in [i]$ with $k \neq l$, the tours C_k and C_l are connected in G_i iff after the i-th iteration w_k and w_l are connected in T.*

Proof. We prove this statement by induction over i. It holds for $i = 1$. So, let the statement be true for arbitrary but fixed $i \in [q-1]$. Let $k, l \in [i+1]$ with $k \neq l$. Let us assume that C_k and C_l are connected in G_{i+1}. We have two cases. If C_k and C_l were already connected in G_i, then by the induction hypothesis w_k and w_l were connected in T after iteration i, so they are still connected after iteration $i+1$ since no edges are deleted in T. If C_k and C_l were not connected in G_i, w_k and w_l were in different connected components in iteration i. Since the connected components of C_k and C_l are connected by C_{i+1} in G_{i+1}, in step 2.4.2. of iteration $i+1$, edges are created in T between w_{i+1} and the connected components of w_k and w_l, so w_k and w_l are connected in T after iteration $i+1$.

Now, let us assume that C_k and C_l are not connected in G_{i+1}. Then, they were not connected in G_i and C_{i+1} is vertex disjoint with the connected component of C_k or C_l. Hence, in step 2.4.2. of iteration $i+1$, an edge from w_{i+1} to the connected component of w_k or w_l will not be created in T, and since w_k and w_l are not connected after iteration i (induction hypothesis), they are not connected after iteration $i+1$. □

Theorem 3.8. *Let G be an Eulerian graph.*

a) The set of information edges returned by the W-stream algorithm form an out-tree $\vec{T}^*$ on vertices $w_1, \ldots, w_q$ such that for every edge in $\vec{T}^*$ the represented tours have a common vertex.

b) The W-stream algorithm never uses more than $\mathcal{O}(n\log(n))$ RAM.

Proof. a) Since G is Eulerian, $G = G_q$ by Lemma 3.6 and because G as an Eulerian graph is connected, G_q is connected as well. Hence, by Lemma 3.7 the graph T is connected after iteration q. Furthermore, edges are created in T only in step 2.4.2., and only one edge per connected component to a new vertex. Hence, no cycle is created in this step. So, T is a tree after iteration q and $\vec{T}$ is an out-tree at the end of the algorithm. The only edges missing in $\vec{T}$ are the information edges outputted in step 2.5.1. Every tour not represented in $\vec{T}$ has exactly one information edge with a neighbor in $\vec{T}$, so only leafs are missing in $\vec{T}$. We define the expanded graph $\vec{T}^*$ consisting of the vertices $w_1, \ldots, w_q$ and all information edges. $\vec{T}^*$ is also a tree, because only leafs are added to $\vec{T}$. Since an information edge f_i^j with $i, j \in [q]$ was only created in step 2.5.1. or in step 2.4.2., when the represented tours C_i and C_j have a common vertex, we are done.

b) A vertex is only created in T if one of the conditions in step 2.3. is fulfilled. This cannot happen more than $2n$ times, so T only needs $\mathcal{O}(n\log(n))$ RAM. Additionally, we store the two values fir_i and comp_i per vertex v_i and up to n edges for finding a tour. Furthermore, we need a constant number of variables (for more details see the pseudo code in the appendix). Altogether, $\mathcal{O}(n\log(n))$ RAM is sufficient for the algorithm. □

3.5 The PL-StrSort algorithm

In this section, we explain in detail how to use the information edges for merging the tours. Note that at this point we are entering the StrSort model and are restricted to $\mathcal{O}(\log(m)) = \mathcal{O}(\log(n))$ RAM. As mentioned in the previous section, we have to address a few problems:

1. The tree $\vec{T}^*$ was created at the end of the W-stream algorithm, so most tours were output before their predecessor tours were determined. The orders of their graph edges have to be changed, so that the tail of the first edge is a common vertex of the predecessor tour.

2. The information edges with no vertex contained in T were output before the rooted tree $\vec{T}$ was created, so they miss the information about the depth of the predecessor in $\vec{T}^*$.

3. All information edges lack the last information: the position of the graph edge of the predecessor tour, behind which the successor tour will be inserted.

Problem 3 can be tackled as follows. The algorithm will iteratively merge tours and update information edges according to a new rooted tree $\vec{T}^{**}$ with height about half the height of the original tree $\vec{T}^*$. At that point, the information edges will miss the information about graph edge positions again.

Further, we will present StrSort algorithms using $\mathcal{O}(1)$ passes and $\mathcal{O}(\log(n))$ RAM for each of the problems 1 and 2 in the following subsections. As explained in Section

3.3.2, the sorting steps are used to place edges needing information next to edges having that information, and both are held in RAM for the information transfer during the next streaming step.

3.5.1 Rotating Tours

Let C_j be a tour with $w_j \in \vec{T}^*$. Let d_j be the depth of w_j in $\vec{T}^*$. If $d_j > 0$, then w_j has a predecessor w_i in $\vec{T}^*$. The information edge f_i^j contains a common vertex v of C_i and C_j, but the order of C_j stored in the graph edges was not changed according to v during the W-stream algorithm. The order of C_j will be changed as follows:

Procedure RotCirc

1. Sort the graph edges by tour label and placement, and the information edges by successor tour s.t. in the stream a tour C_j is stored directly behind the information edge f_i^j with second entry j.

2. While streaming a tour C_j: store v from f_i^j, count the number l_j of edges in the tour, and find the placement p of the edge with v as its tail.

3. Create a 3-tuple $g_j := (j, p, l_j)$ as additional item on the stream. With the next tour go to step 2.

4. Sort in the same way as in step 1. Place each 3-tuple in front of the related tour.

5. In the next streaming step, after reaching g_j, store l_j and p.

6. For $k \in \{1, \dots, l_j\}$: read edge $e_j^k := (v_j^k, v_j^{k+1}, j, k, 0, 0)$ and output $(v_j^k, v_j^{k+1}, j, ((k - p) \bmod l_j) + 1, 0, 0)$.

7. Delete p, l_j and g_j. Choose the next tour and go to step 5.

3.5.2 Information edges and depth

Let C_j be a tour with $w_j \notin T$. Then, there is exactly one information edge with second entry j. Let C_i be the stored predecessor tour and f_i^j be the associated information edge. Then, $w_i \in T$ and the third entry of f_i^j is '$*$'. We need the depth d_i of w_i in $\vec{T}^*$. If w_i is the root of $\vec{T}^*$, then $d_i = 0$. Otherwise, w_i has a predecessor w_k in $\vec{T}^*$. Then, the information edge concerning $\{w_i, w_k\}$ contains the depth d_k of w_k and $d_i = d_k + 1$. With two streaming steps, one sorting step and $f_i^j = (i, j, *, v, *, 0)$ for some $v \in V$, we will get the needed information d_k from f_k^i if it is available:

Procedure InfoDepth

1. Change $f_i^j = (i, j, *, v, *, 0)$ to $(j, i, *, v, *, 1)$, i.e. change predecessor and successor of all information edges with $*$ as third entry.

2. Sort the information edges lexicographically according to the successor and the orientation (i.e. second and sixth entry).

3. If before $(j, i, *, v, *, 1)$ there is no edge with second entry i and without $*$ as third entry, output a depth of 0, i.e. $(i, j, 0, v, *, 0)$.

4. If there is such an edge, e.g. $(k, i, d_k, v, *, 0)$, then for all edges $(j, i, *, v, *, 1)$ with i as second entry, output $(i, j, d_k + 1, v, *, 0)$.

3.5.3 The merging step

We proceed to the merging step and its analysis.

Lemma 3.9. *According to the W-stream algorithm and the two preparation steps* RotCirc *and* InfoDepth*, the graph edges and information edges have the following properties:*

a) The set of edges of G is partitioned into $q \in \mathbb{N}$ tours. For each $i \in [q]$ the tour C_i of length l_i is represented by the l_i graph edges $e_i^j = (v_i^j, v_i^{j+1}, i, j, 0, 0)$ for $j \in [l_i - 1]$ and $e_i^{l_i} = (v_i^{l_i}, v_i^1, i, l_i, 0, 0)$.

b) $\vec{T}^* = (W^*, \vec{F}^*)$ with $W^* := \{w_1, \ldots, w_q\}$ is an out-tree and it holds $((w_i, w_j) \in \vec{F}^* \Leftrightarrow$ there exists an information edge with first entry i and second entry j).

c) For $i, j \in [q]$ let f_i^j be an information edge. Then it has the form $f_i^j = (i, j, d_i, v, *, 0)$, where w_i is the predecessor of w_j in $\vec{T}^*$, d_i is the depth of w_i and v is a common vertex of C_i and C_j. Furthermore, $v_j^1 = v$.

Proof. a) By Lemma 3.6, the graph is partitioned into edge-disjoint tours. The specific form of the graph edges is achieved by steps 2.2. and 2.6. of the W-stream algorithm.

b) By Theorem 3.8 $\vec{T}^*$ is an out-tree. The first and second entries of the information edges are determined by the direction of these edges in steps 2.5.1. and 5. and are correct according to Definition 3.4.

c) After the W-stream algorithm, the first and second entry of f_i^j are correct by part b). That may have changed during InfoDepth, but these changes were undone at the end of this procedure. There are two cases to consider:

- f_i^j was output in step 5. of the W-stream algorithm. Here, the out-tree $\vec{T}$ was created beforehand and the depth of the predecessor vertex was stored in the information edge. The common vertex was stored in step 2.4.2. and output with f_i^j in step 5.

- f_i^j was output in step 2.5.1. of the W-stream algorithm. There, the common vertex was also output. Furthermore, f_i^j received d_i during InfoDepth. The information edge was used in RotCirc to sort the tour such that the tail of the first edge is the vertex v. Hence, $v_j^1 = v$. □

Let h be the height of $\vec{T}^*$. Let us informally describe the StrSort algorithm. It will modify graph edges and information edges s.t. the properties stated in Lemma 3.9 are fulfilled and the out-tree represented by the information edges has height $\lfloor h/2 \rfloor$. The number of graph edges will stay the same, still representing the edges of G. After $\mathcal{O}(\log(h)) = \mathcal{O}(\log(n))$ iterations of the algorithm, the underlying out-tree has a height of 0, so the graph edges form a single tour, i.e. an Euler tour of G.

We will consider an iteration step in our StrSort algorithm. We say that a tour C_i has even (odd) depth if the representative vertex w_i in $\vec{T}^*$ has even (odd) depth. Likewise, we say that an information edge has even (odd) depth when the predecessor vertex has even (odd) depth. For information edges f and f', f' will be called the predecessor edge of f in $\vec{T}^*$ if the successor vertex of f' is the predecessor vertex in f. During one iteration, every tour C_i with odd depth will be inserted in its predecessor tour. Therefore, the information edges of odd depth will not be used for the merging process, since its successor tour has even depth. Instead, they will be updated for the next iteration. The high-level description of the algorithm is:

Algorithm Merge-Tour:

1. Updating the information edges of odd depth.
2. Transmitting the new coordinates to the edges of the tours with odd depth.
3. Relabeling the newly formed tours.

We state the subroutines 1 to 3:

Updating the information edges of odd depth For the update of an information edge f_i^j of odd depth, two values have to be changed:

- The depth changes from d_i to $(d_i - 1)/2$. This is done at the end of the iteration in order to prevent conflicts.
- The predecessor vertex of f_i^j changes, since the represented tour has odd depth and will be merged with his predecessor tour. Therefore, the predecessor vertex of the predecessor vertex in $\vec{T}^*$ will be the new predecessor vertex of f_i^j.

Let $f_i^j = (i, j, d_i, v, *, 0)$ be the successor edge of information edge $f_k^i = (k, i, d_k, v', *, 0)$. f_i^j needs k as its new first entry. First, we switch the values i and j of f_i^j and change the last entry to 1. Then, we sort lexicographically according to the second and sixth entry of the information edges. In this way, every information edge with predecessor i is directly behind f_k^i, and the value k can be transmitted during the following streaming step. Finally, the modified edges regain their original first and second entry. The procedure is as follows.

Procedure InfoUpdate

1. For all information edges $f_i^j = (i, j, d_i, v, *, 0)$ with odd d_i: change edge to $(j, i, d_i, v, *, 1)$.
2. Sort in the following way:
   - Information edges are placed in front of graph edges.
   - Information edges: $(i_1, j_1, d_{i_1}, v_1, *, o_1)$ is in front of $(i_2, j_2, d_{i_2}, v_2, *, o_2)$ iff $\big((j_1 < j_2)$ or $(j_1 = j_2$ and $o_1 < o_2)$ or $(j_1 = j_2$ and $o_1 = o_2$ and $i_1 < i_2)\big)$.
   - The order of the graph edges does not matter.
3. Stream: For every information edge $(i, j, d_i, v, *, 0)$ (with 0 as last entry):

   3.1. Store i in RAM and output $(i, j, d_i, v, *, 0)$.

   3.2. As long as information edges of form $(i', j, d_j, v', *, 1)$ are read, output the information edge $(i, i', d_j, v', *, 0)$ instead.

Transmitting the new coordinates An information edge $f_i^j = (i, j, d_i, v, *, 0)$ with even d_i needs to find a graph edge of tour C_i with the head v. Therefore, the sorting step makes use of the values i and v to place f_i^j directly behind a suitable graph edge e_i^k in order to transmit the value k. In another sorting step, f_i^j is placed in front of the tour C_j, so in the following streaming step the edges of C_j can receive the values i and k and be modified as explained in Section 3.3.2. The following is a more detailed description.

Procedure Transmit

1. Sort in the following way:
   - Place information edges with odd depth in the front and in arbitrary order.
   - Information edges with even depth: $(i_1, j_1, d_{i_1}, v, *, 0)$ is in front of $(i_2, j_2, d_{i_2}, v', *, 0)$ iff $\big((i_1 < i_2)$ or $(i_1 = i_2$ and $v < v')\big)$.
   - Graph edge and information edge with even depth: $(v_i^j, v_i^{j+1}, i, j, 0, 0)$ is in front of $(i', j', d_{i'}, v', *, 0)$ iff $\big((i < i')$ or $(i = i'$ and $v_i^{(j+1)} \leq v')\big)$.
   - Graph edges: $(v_i^j, v_i^{(j+1)}, i, j, 0, 0)$ is in front of $(v_{i'}^{j'}, v_{i'}^{j'+1}, i', j', 0, 0)$ iff $\big((i < i')$ or $(i = i'$ and $v_i^{j+1} < v_{i'}^{j'+1})$ or $(i = i'$ and $v_i^{j+1} = v_{i'}^{j'+1}$ and $j < j')\big)$.
2. Stream: For every graph edge $(v_i^j, v_i^{j+1}, i, j, 0, 0)$:

   2.1. Read all information edges of even depth until the next graph edges follows.

   2.2. For each such information edge $(i', j', d_{i'}, v', *, 0)$, output $(i', j', d_{i'}, v', j, 0)$ instead.
3. Sort the following way:

- Graph edges: $(v_i^j, v_i^{j+1}, i, j, 0, 0)$ is in front of $(v_{i'}^{j'}, v_{i'}^{j'+1}, i', j', 0, 0)$ iff $\big((i < i')$ or $(i = i'$ and $j < j')\big)$.
- Information edges: $(i_1, j_1, d_{i_1}, v, p_1, 0)$ is in front of $(i_2, j_2, d_{i_2}, v', p_2, 0)$ iff $(j_1 < j_2)$.
- Information edge, graph edge: $(i', j', d_{i'}, v', p, 0)$ is in front of $(v_i^j, v_i^{j+1}, i, j, 0, 0)$ iff $(j' \leq j)$.

4. Stream: For every information edge $(i', j', d_{i'}, v', p, 0)$ with even $d_{i'}$:

 4.1. Store i' and p in internal memory, delete the information edge *without* output.

 4.2. As long as graph edges $(v_i^j, v_i^{j+1}, i, j, 0, 0)$ are read, output $(v_i^j, v_i^{j+1}, i', p, i, j)$ instead.

Relabeling the newly formed tours After a lexicographical sorting of the graph edges by the last four entries, the tour merging is done and the edges of the newly formed tours are consecutively placed on the stream. The first edge of each tour kept its tour label, since successor tours are always inserted *behind* an edge. Therefore, we can use this label and a counter c to relabel the following edges. The procedure is as follows:

Procedure Relabel

1. For each graph edge of the form $(v, v', i, 1, 0, 0)$ (with fourth entry 1):

 1.1. Set counter $c := 2$.

 1.2. Store i and output the edge.

 1.3. Do the following until a graph edge is read that does not have i as third entry:

  1.3.1. Read the next graph edge (it has the form $(\bar{v}, \bar{v}', i, x, j, y)$) and output $(\bar{v}, \bar{v}', i, c, 0, 0)$ instead. Then set $c := c + 1$

Lemma 3.10. *After an iteration step of algorithm Merge-Tour, the properties stated in Lemma 3.9 are fulfilled for an out-tree $\vec{T}^{**}$ with height $\lfloor h/2 \rfloor$, where h is the height of the original out-tree $\vec{T}^{*}$.*

Proof. Property a): The only change of the graph edges in algorithm Merge-Tour happens during the tour merging. Since the merging of tours results in tours, G is still partitioned into edge-disjoint tours. The graph edges have the required form due to step 1.3.1. in Relabel.

Property b): When merging connected vertices in a tree, the result is still a tree. The out-tree $\vec{T}^{*}$ changed during the algorithm in the following way: Each vertex with odd depth is merged with its predecessor. Therefore, for each vertex with even depth in $\vec{T}^{*}$, the new predecessor in $\vec{T}^{**}$ is the predecessor of the predecessor in $\vec{T}^{*}$.

We show that $\vec{T}^{**}$ is represented correctly by the remaining information edges. We consider the following two cases.

- Let w_i be a vertex in $\vec{T}^*$ with even depth $d_i \geq 2$. After the W-stream algorithm, there was exactly one information edge with i as its second entry. This information edge was not deleted in step 4.1. of TRANSMIT. The first entry changed in step 3.2. of INFOUPDATE such that the correct predecessor of w_i in $\vec{T}^{**}$ is displayed. Since second entries in information edges do not change in the StrSort algorithm of Section 3.5.3, there is still only this information edge with i as its second entry.
- Let w_j be a vertex in $\vec{T}^*$ with odd depth. This vertex is merged with its predecessor and is not a vertex in $\vec{T}^{**}$. The unique information edge with second entry j is deleted in step 4.1. of TRANSMIT.

Therefore, the only information edges left after algorithm Merge-Tour are the ones that correctly represent the edges of the out-tree $\vec{T}^{**}$.

Property c): Let f_i^j be an information edge associated with the tours C_i and C_j after algorithm Merge-Tour. Then, C_i contains the tour that was the predecessor tour of C_j after the W-stream step, so C_i also contains their common vertex v. Furthermore, the information of the depth was updated in step 3.2. of INFOUPDATE. Therefore, the form of f_i^j is correct. Since the first edge of C_j did not change, C_j is still in an order such that the tail of the first edge is v.

Let $h \geq 1$ be the height of $\vec{T}^*$. For the vertices with depth h, the corresponding information edges have a depth of $h-1$. If h is even, then $h-1$ is odd and at the end of the iteration step the depth of these information edges changes to $((h-1)-1)/2$, so the height of $\vec{T}^{**}$ is $(h-2)/2+1 = h/2 = \lfloor h/2 \rfloor$. If h is odd, then these vertices are merged with their predecessors and do not exist in $\vec{T}^{**}$. In this case, the depth of the predecessors changes to $((h-2)-1)/2$, so the height of $\vec{T}^{**}$ is $(h-3)/2+1 = \lfloor h/2 \rfloor$.

Altogether, all properties of Lemma 3.9 are fulfilled for the out-tree $\vec{T}^{**}$ with height $\lfloor h/2 \rfloor$. □

Theorem 3.11. *a) After $\mathcal{O}(\log(n))$ iterations of algorithm Merge-Tour, the order of the graph edges on the output stream represents an Euler tour on the graph G.*

b) The StrSort algorithms use $\mathcal{O}(\log(n))$ RAM.

c) The stream never exceeds a length of $\mathcal{O}(m \log(n))$.

Proof. a) Lemma 3.10 states that after every iteration of algorithm Merge-Tour, the properties of Lemma 3.9 are fulfilled with an out-tree of about half of the original height. After $\mathcal{O}(\log(h)) = \mathcal{O}(\log(m)) = \mathcal{O}(\log(n))$ iterations, the out-tree consists of only one vertex. Hence, the represented tour contains every graph edge and is therefore an Euler tour.

b) Every streaming step only stores a constant number of graph or information edges simultaneously in RAM, each of size $\mathcal{O}(\log(n))$. Hence, $\mathcal{O}(\log(n))$ RAM is used.

c) After the W-stream algorithm, there are m graph edges and at most m information edges with $\mathcal{O}(\log(n))$ memory space each. The only time additional items are created during the StrSort algorithms is in ROTCIRC, step 3. There are at most m additional items of size $\mathcal{O}(\log(n))$. Therefore, the stream never exceeds a length of $\mathcal{O}(m \log(n))$. □

3.6 Conclusion

We have presented an algorithm for finding Euler tours in undirected graphs in the StrSort model. It uses a single pass preparation step with $\mathcal{O}(n \log(n))$ memory space, followed by a PL-StrSort algorithm. Considering this result, various open questions arise:

- Can the preparation step be replaced by an StrSort algorithm using $\mathcal{O}(\log(n))$ passes and memory space? In this case, the Euler tour problem could be solved entirely by a PL-StrSort algorithm. However, as indicated by Ruhl [29], finding cycles with a StrSort algorithm might be difficult.

- Are there more problems where a single pass with larger RAM enables it to be solved by a PL-StrSort algorithm? Such a step might be a useful addition to the StrSort model.

3.7 Appendix

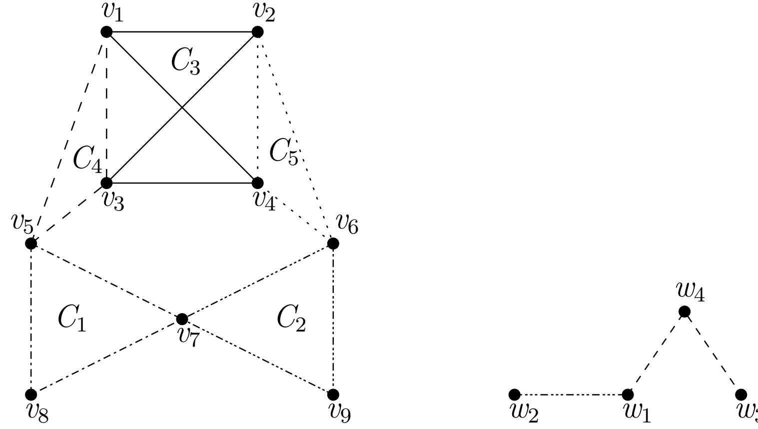


Figure 3.3: Partition into tours and created tree.

Example of the W-stream step Figure 3.3 gives an example on a graph with vertices $v_1, \ldots, v_9$. Assume that the tours found are $C_1, \ldots, C_5$ in that order. C_1 is the first tour containing v_5, so a vertex w_1 in T is created. We set $\text{fir}_i := 1$ and $\text{comp}_i := 1$ for $i \in \{5, 7, 8\}$ and output the expanded edges $(v_5, v_7, 1, 1, 0, 0)$, $(v_7, v_8, 1, 2, 0, 0)$ and $(v_8, v_5, 1, 3, 0, 0)$. C_2 is the first tour containing v_6 and shares the vertex v_7 with C_1 (this information is stored in fir_7), so $\text{comp}_6 := 1$, $\text{comp}_9 := 1$ and w_2 is created in T with edge $\{w_1, w_2\}$. Furthermore $\text{fir}_6 := 2$ and $\text{fir}_9 := 2$, because v_6 and v_9 are used for the first time. Additionally, the expanded edges $(v_6, v_7, 2, 1, 0, 0)$, $(v_7, v_9, 2, 2, 0, 0)$ and $(v_9, v_6, 2, 3, 0, 0)$ are output. With C_3, we set $\text{fir}_i := 3$ for $i \in [4]$ and have a new connected component in G_3 with $\text{comp}_i = 3$ for $i \in [4]$. We place a vertex w_3 in T without additional edges and output the expanded edges. C_4 connects the components '1' and '3'. For each component a common vertex of C_4 is chosen. Vertices v_1 and v_5 are selected with $\text{comp}_5 = 1$ and $\text{comp}_1 = 3$. We create a vertex w_4 and since $\text{fir}_5 = 1$ and $\text{fir}_1 = 3$, we connect the vertex with edges $\{w_4, w_1\}$ and $\{w_4, w_3\}$ in T. Again, we output the expanded edges. The tour C_5 contains only vertices previously used and does not connect components in G_4, so there is no additional vertex in T. However, we select the vertex v_2, and since $\text{fir}_2 = 3$, we output the information edge $(3, 5, *, v_2, *, 0)$. Finally, we output the edges $(v_2, v_4, 5, 1, 0, 0)$, $(v_4, v_6, 5, 2, 0, 0)$ and $(v_6, v_2, 5, 3, 0, 0)$. Now, T can be rooted and directed, e.g. with root w_4. In this case, the following information edges are outputted at the end of the algorithm: $(4, 1, 0, v_5, *, 0)$, $(4, 3, 0, v_1, *, 0)$ and $(1, 2, 1, v_7, *, 0)$.

The W-stream algorithm in pseudo code

Algorithm 2: Algorithm TOUR-FIND

input : Undirected graph $G = (\{v_1, \ldots, v_n\}, E)$ with edges in arbitrary order, $m := |E|$

output: m graph edges and q information edges for $q \leq m$

```
1  comp_i := * for all i ∈ [n];
2  fir_i := * for all i ∈ [n];
3  cir := 0;
4  s := false, s_cr := false ;         // indicates if vertex in T will be or is
     created
5  s_edge := * ;                              // indicated potential edge in T
6  s_vert := * ;                               // indicated common vertex in G
7  T := (W, F), W := ∅, F := ∅;
8  S_comp := {0} ;   // keeps track of conn. comp. concerning current tour
9  comp* := *;
10 repeat
11    read stream until (n edges are in internal memory) or (end of stream);
12    find tour C = v_{i_1} − e_i^1 − ··· − v_{i_{l_i}} − e_i^{l_i} − v_{i_1} with vertices v_{i'_1}, ..., v_{i'_{l'}}
        (l_i, l' ∈ ℕ) in internal memory;
13    if there is no such tour, return 'graph is not Eulerian';
14    cir := cir + 1;
15    NEW-TEST(C) ;                   // does C contain a new vertex? -> s
16    COMP-TEST(C) ;                    // does C connect components? -> s
17    if s = false then
18       output information edge (s_edge, cir, *, v_{s_vert}, *, 0);
19       sort C, s.t. C = v_{i_1} − e_1 − ··· − v_{i_{l_i}} − e_{l_i} − v_{i_1} with v_{i_1} = v_{s_vert};
20    for j:=1 to l_i-1 do
21       output graph edge (v_{i_j}, v_{i_{j+1}}, cir, j, 0, 0);
22    output graph edge (v_{i_{l_i}}, v_{i_1}, cir, l_i, 0, 0);
23    delete C from internal memory;
24    s := false, s_cr := false, s_edge := *, s_vert := *, S_comp := {0}, comp* := *;
25 until (end of stream) and (no edges in internal memory);
26 for i:=1 to n-1 do
27    if comp_i ≠ comp_{i+1} then
28       return 'graph is not Eulerian'
29 write T as rooted tree;
30 for every w_i ∈ W, let d_i be the depth of w_i in T;
31 for every information edge (i, j, *, v, *, 0) in internal memory output information
     edge (i, j, d_i, v, *, 0);
```

Algorithm 3: Algorithm NEW-TEST

```
1  for j:=1 to l' do
2      if fir_{i'_j} = * then
3          s := true;
4          fir_{i'_j} := cir;
5      else
6          if s_edge = * then
7              s_edge := fir_{i'_j};
8              s_vert := i'_j;
9              S_comp := S_comp ∪ {comp_{i'_j}};
10             comp* := comp_{i'_j}
11 if s = true then
12     create vertex w_cir, W := W ∪ {w_cir};
13     if s_edge ≠ * then
14         create edge {w_cir, w_{s_edge}}, F := F ∪ {{w_cir, w_{s_edge}}};
15         create information edge (s_edge, cir, *, v_{s_vert}, *, 0);
16     else
17         for j:=1 to l' do
18             comp_{i'_j} := cir
```

When the algorithm TOUR-FIND finds a tour C_i in line 12, it is tested, if C_i uses a vertex of G for the first time (NEW-TEST) or connects connected components in G_{i-1} (COMP-TEST). In NEW-TEST, the lines 2-4 test if a vertex is used for the first time. If this is the case, s indicates that a new vertex w_i is created in the tree T. Lines 6-10 test if the tour uses a vertex used by a tour C_j before. If w_i is created, an edge $\{w_i, w_j\}$ is stored and an information edge is output (lines 11-15). S_{comp} keeps track of the connected components in G_{i-1} touched by C_i. If C_i only uses new vertices, there will be a new connected component in G_i. This is noted in lines 17-18. Algorithm COMP-TEST starts if C_i uses a vertex used before. Let A_k be the connected component of that vertex in G_{i-1}. In COMP-TEST it is tested if C_i uses vertices which are not in A_k and not used for the first time. If this happens for the first time, and there is not already a vertex w_i in T, such a vertex is created in line 4-8 with the necessary graph and information edge. Otherwise, just the graph and information edge is output. In lines 13-15 the variables comp_k are updated. If after NEW-TEST and COMP-TEST there is still no vertex w_i in T, in line 18 of TOUR-FIND an information edge is output. The third entry is '$*$', indicating that C_i has no representative in T. In lines 20-21, the tour is output such that the tail of the first edge is a common vertex of the tour noted in the information edge. The connectivity of G is tested in lines 26-28. Finally the rooted tree is built, and the stored

Algorithm 4: Algorithm COMP-TEST

1 **if** $comp^* \neq *$ **then**
2 **for** *j:=1 to l'* **do**
3 **if** $comp_{i'_j} \neq comp^*$ **then**
4 **if** $s =$ false **then**
5 $s :=$ true;
6 create vertex w_{cir}, $W := W \cup \{w_{\text{cir}}\}$;
7 create edge $\{w_{\text{cir}}, w_{s_{\text{edge}}}\}$, $F := F \cup \{\{w_{\text{cir}}, w_{s_{\text{edge}}}\}\}$;
8 create information edge $(s_{\text{edge}}, \text{cir}, *, v_{s_{\text{vert}}}, *, 0)$;
9 **if** $comp_{i'j} \notin S_{comp}$ **then**
10 create edge $\{w_{\text{cir}}, w_{\text{fir}_{i'_j}}\}$, $F := F \cup \{\{w_{\text{cir}}, w_{\text{fir}_{i'_j}}\}\}$;
11 create information edge $(\text{fir}_{i'_j}, \text{cir}, *, v_{i'_j}, *, 0)$;
12 $S_{\text{comp}} := S_{\text{comp}} \cup \text{comp}_{i'_j}$;
13 **for** *k:=1 to n* **do**
14 **if** $comp_k \in S_{comp} \backslash \{comp^*\}$ **then**
15 $\text{comp}_k := \text{comp}^*$;

information edges are updated and output.

Chapter 4

The bridge-burning Cops and Robbers Game

4.1 Introduction

4.1.1 Cops and Robbers

Cops and robbers is a two player full information game played on a graph G. One player controls a set of cops, while the other player controls a single robber. At the start of the game, player 1 places all cops on different nodes of G. After that, player 2 places the robber on a node not occupied by a cop. Then, the game proceeds with alternate turns for the cops and the robber. At a cops' turn, each cop can either move from his position along an edge to a neighboring node or stay in position. At a robber's turn, the robber also can move to a neighboring node or stay in position. If at someone's turn a cop is on the same node as the robber, the robber is *captured*. If the cops can capture the robber in a finite number of turns, they win the game. Otherwise the robber wins. A graph which optimally played is won by the cops is called *cop-win graph*.

Cops and robbers is a well-studied graph-theoretical problem. The game was introduced in the early 1980s. Some of the first results ([25],[28]) characterized graphs on which one cop is able to capture a robber. In 1984, Aigner and Fromme ([2]) considered the minimum number of cops on a graph to capture a robber, the so-called *cop number* of a graph. Another interesting aspect of the game was introduced by Bonato et al. ([6]). The *capture time* of a cop-win graph states the minimum number of turns the cops need to win the game, independent of the strategy chosen by the robber. I.e. it is the number of turns the cops need to capture the robber, if he tries not to be captured for as many turns as possible. We refer to [18] for a survey of the traditional cops and robbers game.

During the decades, countless different variations of the game were created. In the context of mobility for example, the cops can be restricted to one moving cop per turn([26], [5]), the robber can move faster than the cops ([14]) or the robber is able to move anywhere in one turn as long as there is a path with no cops on it([24]). We refer to [7] for a wider view on the subject. One of the newest variations is the bridge-

burning cops and robbers game.

4.1.2 Burning Bridges

Kinnersley and Peterson ([21], 2018) introduced an interesting variant of cops and robbers, called *bridge-burning cops and robbers*. Here, whenever the robber traverses an edge, this edge is deleted from the graph afterwards. In this variant, there are two ways for the robber to win: either he creates and enters a connected components without cops or a situation occurs where the 'best move' for the cops and the robber is to stay in position and thereby the cops do not catch the robber.

Kinnersley and Peterson determined the cop number c_b of trees, grids, tori and hypercubes for the bridge-burning cops and robbers game. They also looked at the capture time $\text{capt}_b(G)$ of graphs G with a cop number of 1. First, they gave a simple argument for the fact that every graph G on n nodes with $c_b(G) = 1$ has a capture time of $\text{capt}_b(G) = \mathcal{O}(n^3)$. Then, they showed that for every $n \in \mathbb{N}$ there exists a graph G on n nodes with $c_b(G) = 1$ and $\text{capt}_b(G) = \Omega(n^2)$. For $l \in \mathbb{N}$, we define the notation $[n] := \{1, \ldots, l\}$. Let $n \in \mathbb{N}$ and $k, m \in \mathbb{N}$ such that $n = k(m+2)$ and $m(k-1)$ is even. Consider the graph $G = (V, E)$ defined as follows. The set of nodes V consists of the sets $U := \{u_i : i \in [k]\}$, $W := \{w_i : i \in [k]\}$, and $S_j := \{s_i^j : i \in [m]\}$ for $j \in [k]$. $G|_U$ is a complete subgraph. $G|_{S_1 \cup \cdots \cup S_k}$ is a complete k-partite subgraph. Additionally, for every $i \in [k]$ and every $s \in S_i$ we have $(u_i, s) \in E$ and $(u_i, w_i) \in E$.

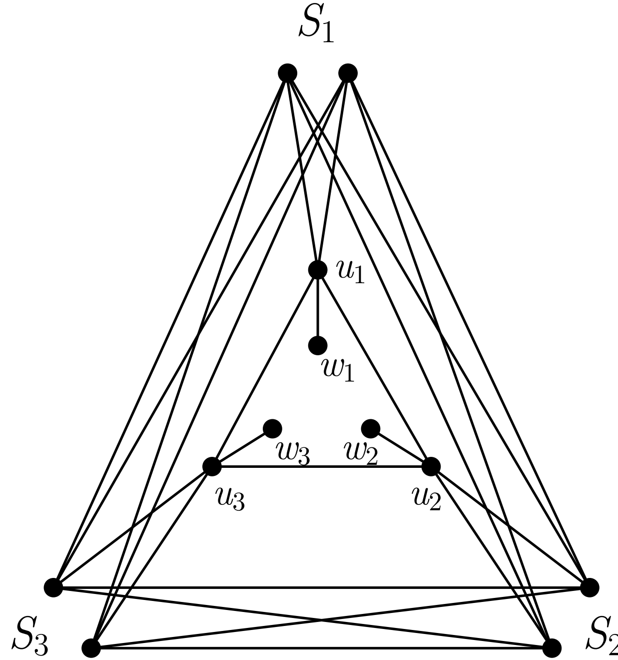


Figure 4.1: Graph with capture time of $\mathcal{O}(n^2)$ ($k = 3$, $m = 2$)

.

Basically, the cop's strategy is to move along the nodes in U to force the robber to move along an Euler tour in $S_1 \cup \cdots \cup S_k$ until all those edges are deleted and the robber can no longer move.

Finally, Kinnersley and Peterson conjectured that the upper bound of n^3 for the capture time of graphs with a cop number of 1 is tight, i.e. there are graphs on n nodes with a capture time of $\Omega(n^3)$ (Conjecture 5.3 in [21]). In such a graph, the robber would not move every turn, but would wait a linear number of turns each time before moving. Therefore, we are looking for a graph such that on the one hand the robber can stay in position for a long period of time without being captured and on the other hand the cop can 'leave the robber alone' for a couple of turns without the robber isolating himself in the meantime.

In 2020, Herrman et al. [19] proved tight bounds for the game with multiple cops. They showed that there is a constant C such that for every $k \geq 3$ and n sufficiently large, there is a graph G_n on n nodes such that with k cops G_n is a cop-win graph and has a capture time of

$$capt_b(G_n) \geq C\frac{n^{k+2}}{k^{k+2}}.$$

They leave the conjecture of Kinnersley and Peterson as an open problem.

4.1.3 Our Contribution

We prove the conjecture of Kinnersley and Peterson, i.e. we find a function $f : \mathbb{N} \to \mathbb{N}$ with $f \in \Omega(n^3)$ and for every $n \in \mathbb{N}$ construct a graph G_n on n nodes with $c_b(G_n) = 1$ and $\text{capt}_b(G_n) \geq f(n)$. We limit the options for the robber's starting position by creating an immediate threat he has to take care of. Furthermore, most of the nodes will have even degree, so that the robber cannot easily isolate himself by creating a connected component. Similar to [21], we also give a possible escape route for the robber that the cop has to defend. Thereby we limit the cop's options as well. Altogether, we have a subgraph on which the robber cannot isolate himself and a path of size $\Theta(n)$ which the cop has to traverse every time he wants the robber to move.

4.2 Graphs with high capture time

Again, for $l \in \mathbb{N}$, we use the notation $[l] := \{1, \ldots, l\}$. For a graph $G = (V, E)$, let $v \in V$ and $S \subseteq V$. The distance between v and S is defined as

$$\text{dist}(v, S) := \min\{\text{dist}(v, w) | w \in S\}. \tag{4.1}$$

Bonato et al. ([6]) define the capture time $capt_b(G)$ of a game on a graph G with cop number b with b cops and one robber as the minimum number of turns over all possible games, assuming that the robber is trying to avoid being captured for as long as possible.

We define our graph $G^* = (V^*, E^*)$ with the desired properties. If not stated otherwise, $n := |V^*|$ will be the number of nodes and $m := |E^*|$ will be the number of edges.

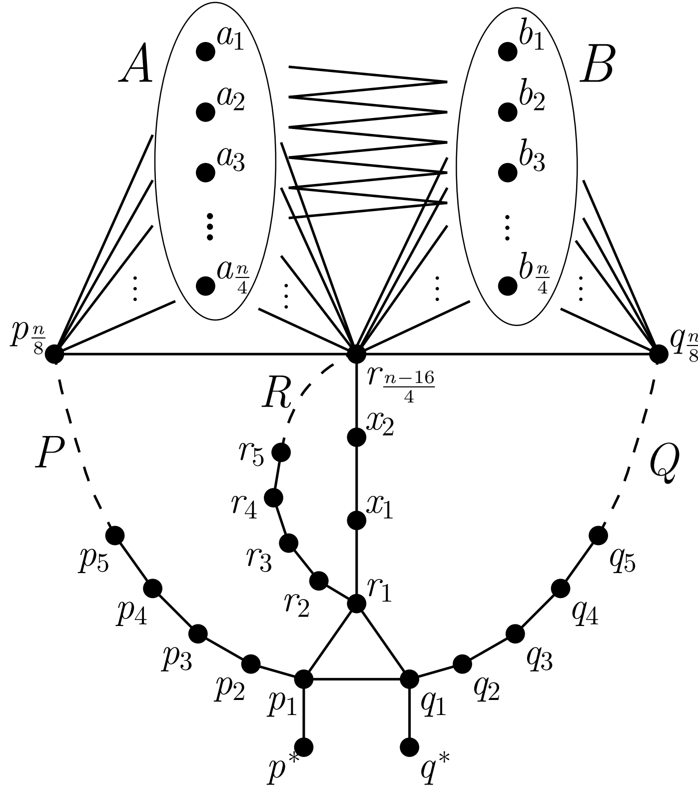


Figure 4.2: Graph G^*.

Definition 4.1 (Graph G^* for 8|n). *Let $n \geq 72$ such that $n = 8k$ for some $k \in \mathbb{N}$. We define the graph $G^* = (V^*, E^*)$ on $|V^*| = n$ nodes as follows. The set of nodes V^* consists of*

- *the sets $A := \{a_i : i \in [\frac{n}{4}]\}$ and $B := \{b_i : i \in [\frac{n}{4}]\}$,*
- *the sets $\{p_i : i \in [\frac{n}{8}]\}$, $\{q_i : i \in [\frac{n}{8}]\}$ and $\{r_i : i \in [\frac{n-16}{4}]\}$,*
- *the nodes x_1, x_2, p^* and q^*.*

E^* is the set of the following edges.

- *For all $a_i \in A$, $b_j \in B$ with $i, j \in [\frac{n}{4}]$: $(a_i, b_j) \in E^*$, the induced subgraph $G^*|_{A \cup B}$ is a complete bipartite graph.*
- *For all $i \in [\frac{n}{4}]$: $(a_i, p_{\frac{n}{8}}) \in E^*$ and $(b_i, q_{\frac{n}{8}}) \in E^*$.*
- *For all $v \in A \cup B \cup \{p_{\frac{n}{8}}, q_{\frac{n}{8}}\}$: $(r_{\frac{n-16}{4}}, v) \in E^*$.*
- *For all $i \in [\frac{n}{8} - 1]$: $(p_i, p_{i+1}) \in E^*$ and $(p_i, p_{i+1}) \in E^*$, i.e. we have paths $P := p_1$-p_2-$\cdots$-$p_{\frac{n}{8}}$ and $Q := q_1$-q_2-$\cdots$-$q_{\frac{n}{8}}$.*

- *For all* $i \in [\frac{n-16}{4} - 1]$*:* $(r_i, r_{i+1}) \in E^*$*, i.e. we have a path* $R := r_1\text{-}r_2\text{-}\cdots\text{-}r_{\frac{n-16}{4}}$*.*
- $\{(r_1, x_1), (x_1, x_2), (x_2, r_{\frac{n-16}{4}})\} \subset E^*$*.*
- $\{(p_1, q_1), (p_1, r_1), (q_1, r_1), (p^*, p_1), (q^*, q_1)\} \subset E^*$*.*

Definition 4.2 (Graph G^* for $8\nmid n$)**.** *Let* $n > 72$ *such that* $n = 8k + l$ *for some* $k \in \mathbb{N}$ *and* $l \in [7]$*. The graph* $G^* = (V^*, E^*)$ *on* $|V^*| = n$ *nodes is defined similar to the graph* G^*_{8k}*, with additional nodes* $p^*_1, \ldots, p^*_7$ *and additional edges* $(p_1, p^*_1), \ldots, (p_1, p^*_l)$*.*

Proposition 4.3. *Let* $k \in \mathbb{N}_{\geq 8}$ *and* $l \in [7]$*. If the cop has a strategy to win the game on* G^*_{8k}*, he has a strategy to win the game on* G^*_{8k+l} *with the same capture time.*

Proof. Assume that the cop has a strategy to win the game on G^*_{8k}. In this strategy, he will start on p_1 or a neighbor of p_1, because otherwise the robber could start on p_1 and move to p^* on his first move, removing the edge (p_1, p^*) and winning, because he is on a different connected component than the cop. Thus, if the robber starts on p^*, he will get captured after one or two moves of the cop. This happens analogously if the robber starts on a node p^*_i for $i \in [l]$. If the robber does not start on one of the nodes in $p^*, p^*_1, \ldots, p^*_l$, the cop can use the same strategy as on G^*_{8k}, ignoring the nodes $p^*_1, \ldots, p^*_l$ and winning in the same number of turns as in G^*_{8k}, because the robber will never visit one of the nodes in $p^*_1, \ldots, p^*_l$ during the game. Otherwise, he could have visited the node p^* instead. Thus, he would have won the game on G^*_{8k}, a contradiction. □

Observation 4.4. *The only nodes in* G^* *with odd degree are* p^* *and* q^**.*

Proof. p^* and q^* have odd degree. For $v \in A \cup B$, $\deg(v) = \frac{n}{4} + 2$. We have $\deg(p_{\frac{n}{8}}) = n/4 + 2 = \deg(q_{\frac{n}{8}})$, $\deg(r_{\frac{n-16}{4}}) = 2 \cdot \frac{n}{4} + 4$ and $\deg(u) = 4$ for $u \in \{p_1, q_1, r_1\}$. The only nodes left have a degree of 2. Therefore, all nodes in $V^* \setminus \{p^*, q^*\}$ have even degree. □

First, we state a basic result frequently used in graph theory, for example in the proof of Tutte's theorem.

Lemma 4.5. *Let* G *be a graph and* H *be a connected component of* G*. The number of nodes in* H *with odd degree is even.*

The next lemma shows the importance of the robber's starting position. During the game edges are deleted, so the graph G^* may split into connected components.

Lemma 4.6. *In the bridge-burning cops and robbers game on* G^**, let the starting position of the robber be a node* $v \in V^* \setminus \{p_1, q_1, p^*, q^*\}$*. As long as he does not visit a node in* $\{p_1, q_1\}$*, the robber's current position is on the same connected component as* v*.*

Proof. Since the robber does not start on a node in $\{p, q, p^*, q^*\}$, as long as he does not visit p_1 or q_1, the edges (p^*, p_1), (q^*, q_1) and (p_1, q_1) will never be deleted. Therefore, the connected components during the game played on G^* are the same as on the graph $\bar{G}_n := (V^*, E^* \cup (p^*, q^*))$, where we add the edge (p^*, q^*) to G^*. By Observation 4.4, there are no nodes in $\bar{G}_n$ with odd degree. Now, consider the following three situations during the game.

a) When every node has even degree, the robber's position is v and he moves to a node u, the edge (u, v) is deleted and v and u have odd degree.

b) If v and the robber's current position $u \neq v$ are the only nodes with odd degree and the robber moves to a node $u' \neq v$, the edge (u, u') is deleted, u has even degree and u' has odd degree. So again, v and the robber's current position are the only nodes with odd degree.

c) If v and the robber's current position $u \neq v$ are the only nodes with odd degree and the robber moves to v, the edge (u, v) is deleted, every node has even degree and v is the robber's current position.

- The game starts with a)
- a) leads to b) or c)
- b) leads to b) or c)
- c) leads to a)

So, aside from not moving, these are the only kinds of possible moves for the robber. Hence, as long as the robber does not visit p_1 or q_1, only two cases occur during the game:

- When the robber is on v, he is in the same connected component as v.
- When the robber is on a node $u \neq v$, these two nodes are the only ones with odd degree. With Lemma 4.5, we get that u and v are in the same connected component.

□

As mentioned earlier, there are two possibilities for the robber to win. He can either escape by getting on a different connected component than the cop, or he wins because both he and the cop stops moving for the rest of the game, a so called *stalemate*. During the game, the cop has to make sure that the robber cannot isolate himself. He will have to 'guard' certain nodes, i.e. prevent the robber from accessing certain nodes. For example, in the beginning of the game, the cop has to guard p_1 and q_1, so that the robber cannot escape to p^* or q^*. On the other hand, the cop cannot just stay in position to guard these nodes, because that would end up in a stalemate.

The following lemma shows, that it is possible for the cop to guard a clique and still prevent a stalemate.

Lemma 4.7. *Let G be a graph and K be a clique in G. Let the cop's position be a node $u \in K$ and robber's position be a node $v \notin K$. Assume that the robber can isolate himself, i.e. get to a different connected component than the cop, only if he first reaches one of the nodes in K. Then, the cop wins.*

Proof. Let $G \in (V, E)$. The strategy of the cop is to guard the nodes in K while avoiding a stalemate. He will move in a way such that he is always closer to K than the robber, so that the robber cannot get to a node in K without getting captured. Also his moves will force the robber to move, avoiding a stalemate.

To be more precise, the strategy for the cop is to force the robber to move to a node $v' \notin K$, so that the general assumption of the lemma still holds. Since every time the robber moves an edge is deleted, the robber can only move a finite number of times. But since by the action of the cop the robber was not able to move to a node in K, he is not on a different connected component than the cop, so the robber did not win. Hence, at some turn the robber is unable to make such a move and the cop wins simply by walking to the robber's position on a shortest path from K.

If it is robber's turn, he cannot move to a node in K, because he would be caught in the cop's next turn, as K is a clique.

So, let us consider a cop's turn. Let $\mathcal{P}$ be the set of paths from a node of K to v. Let $P_k \in \mathcal{P}$ be a path of minimal length $l(P_k)$ with $k \in K$ being the unique node of K in $\mathcal{P}$. We devise the following strategy for the cop. First, the cop moves to k. Then, at every cop's turn

- while the robber stays at v, the cop moves towards the robber along P_k,
- if the robber moves away from v, the cop moves along P_k towards k.

Obviously, the robber has to move at some turn, otherwise the cop would simply walk to the robber and catch him. So we can assume that the robber will move in some turn. After the robber moved, his distance from K is at least $l(P_k) - 1$, while the cop has a distance from K of at most $l(P_k) - 1$, as he is currently not at v. Since it is cop's turn now, he can reach K before the robber. The robber is not able to reach a node in K without getting captured afterwards, because by that time the cop is already on k, and k is connected to every node in K. Therefore, when the cop returns to k, the robber is not in K and the general assumption of the lemma is still satisfied. Since we are in the bridge-burning game, and after every move of the robber an edge is deleted, after at most $|E|$ turns the cop catches the robber and wins. □

We can now prove the main result stating a winning strategy for the cop.

Theorem 4.8. *There is a winning strategy for the cop for the bridge-burning cops and robbers game played on G^*.*

Proof. We let the cop start on the node r_1 and consider the different starting positions of the robber.

Case 1: If the robber starts at the neighboring nodes p_1, q_1, r_2 or x_1, the cop moves there in his first turn and wins.

Case 2: Let the robber start at a node p_i with $2 \leq i \leq \frac{n}{8} - 1$. The case q_i is analogous. Define the paths $P_p := p_1\text{-}p_2\text{-}\cdots\text{-}p_i$ and $P_x := p_1\text{-}r_1\text{-}x_1\text{-}x_2\text{-}r_{\frac{n-16}{4}}\text{-}p_{\frac{n}{8}}\text{-}p_{\frac{n}{8}-1}\text{-}\cdots\text{-}p_i$.

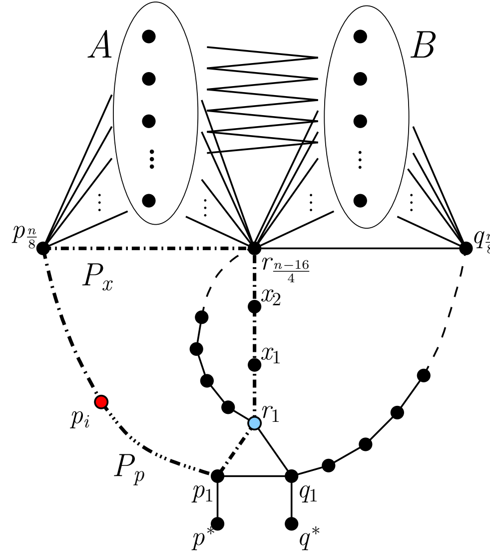


Figure 4.3: Case 2, Paths P_p and P_x.

a) If $l(P_p) \geq l(P_x)$, the cop moves towards p_i along P_x until the robber moves for the first time. We can assume that the robber moves before the cop reaches p_i, otherwise the cop wins. So, the cop has not reached p_i.

   (i) If the robber moves to p_{i-1}, the cop moves to p_1 along P_x. Since $l(P_p) \geq l(P_x)$ and the cop is not at p_i, he can reach p_1 before the robber. Also, the edge (p_i, p_{i-1}) is deleted, so the robber can only move along P_p towards p_1. Hence, after reaching p_1, the cop can simply walk along P_p to capture the robber.

   (ii) If the robber moves to p_{i+1}, the cop moves to p_1 along P_x. He can reach p_1 before the robber can reach one of the nodes p_1 or q_1, because the edge (p_i, p_{i+1}) is deleted and all shortest paths from p_{i+1} to any of these nodes use the path P_x. After the robber's move, exactly the nodes p_i, p_{i+1}, p^* and q^* have odd degree and the robber's position is p_{i+1}. Lemma 4.6 states that as long as the robber does not visit p_1 or q_1, he will always be in the same connected component as p_i. Since the edge (p_i, p_{i+1}) is deleted, p_1, q_1 and p_i will be in the same component as long as the robber does not delete any of the edges (q^*, q_1), (q_1, p_1), (p^*, p_1) or in P_p. As long as the cop is in the same connected component as p_1 and q_1, he is in the same connected component as p_i, so by Lemma 4.6 he is in the same connected component as the robber. So, as long as the cop is in the same connected component as p_1 and q_1, the only chance for the robber to get to a different connected component than the cop is to reach p_1 or q_1, and thereafter isolate himself by walking to p^*, q^* or p_2. Now, with the cop on p_1 and the robber not on a node in the clique $K := \{p_1, q_1\}$,

the assumptions of Lemma 4.7 are fulfilled. By Lemma 4.7 the cop wins the game.

b) If $l(P_p) \leq l(P_x)$, the cop moves towards p_i along P_p until the robber moves for the first time.

(i) If the robber moves to p_{i-1}, the edge (p_i, p_{i-1}) is deleted and the cop can continue to walk towards the robber and captures him.

(ii) If the robber moves to p_{i+1}, we are in a case similar to case a(ii). Again, the robber can only get to a different connected component than the cop if in some turn he gets to a node in $K := \{p_1, q_1\}$. Since $l(P_p) \leq l(P_x)$, the cop is closer to K and can move to p_1. When he arrives at p_1, the robber does not move to a node in K, otherwise he would be captures in the next turn. So, the assumptions of Lemma 4.7 are fulfilled and the cop wins.

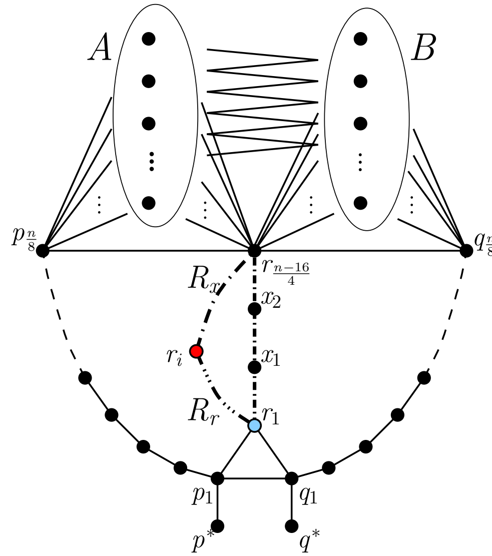


Figure 4.4: Case 3, Paths R_r and R_x

Case 3: Let the robber start at a node r_i with $2 \leq i \leq \frac{n-16}{4} - 1$. This case is similar to the case above with r_1 instead of p_1. Here, we have the paths $R_r := r_1\text{-}r_2\text{-}\cdots\text{-}r_i$ and $R_x := r_1\text{-}x_1\text{-}x_2\text{-}r_{\frac{n-16}{4}}\text{-}r_{\frac{n-16}{4}-1}\text{-}\cdots\text{-}r_i$. Here, if the robber moves to r_{i+1}, the only nodes with odd degree are p^*, q^*, r_i and r_{i+1}. After that, he can only reach one of the nodes p^*, q^* or r_i, if he can reach p_1, q_1 or r_1. So, this case is similar to Case 2 with the clique $K = \{p_1, q_1, r_1\}$. Again, the cop can establish the assumptions of Lemma 4.7 and win the game.

Case 4: When the robber starts at x_2, the cop moves to x_1 and forces the robber to move to $r_{\frac{n-16}{4}}$ and delete the edge $(x_2, r_{\frac{n-16}{4}})$. Now, the cop returns to r_1 and the case is

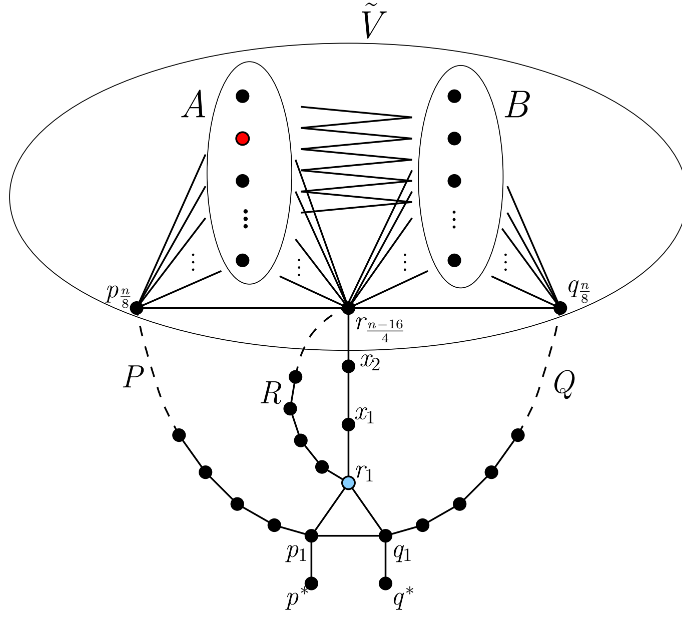


Figure 4.5: Case 5

again similar to Case 3 with $K := \{p_1, q_1, r_1\}$.

Case 5: Let the robber start at a position in $v \in \tilde{V} := A \cup B \cup \{p_{\frac{n}{8}}, q_{\frac{n}{8}}, r_{\frac{n-16}{4}}\}$. In the first three cop turns, the cop moves to $r_{\frac{n-16}{4}}$. The robber can prevent that if he starts at $r_{\frac{n-16}{4}}$, moves to x_2 and deletes the edge $(x_2, r_{\frac{n-16}{4}})$. But then, he would be captured in x_2 in the following turn. So, we may assume that the cop reached $r_{\frac{n-16}{4}}$. In these three turns, at most two edges incident to $r_{\frac{n-16}{4}}$ were deleted by the robber, say $(u, r_{\frac{n-16}{4}})$, $(w, r_{\frac{n-16}{4}})$ with $u, w \in \tilde{V}$.

Since u and w have a degree of at least 18 and the robber deleted at most two edges in $\tilde{V}$, there are $u', w' \in \tilde{V}$ such that (u, u'), (w, w'), $(u', r_{\frac{n-16}{4}})$ and $(w', r_{\frac{n-16}{4}})$ were not deleted. The situation is as seen in Figure 4.6.

Every node in $\tilde{V}$ is a neighbor of $r_{\frac{n-16}{4}}$, u' or w', with $r_{\frac{n-16}{4}}$ being the neighbor of every node in $\tilde{V} \backslash \{u, w\}$. Now, the strategy for the cop is basically to walk back and forth the three nodes $r_{\frac{n-16}{4}}$, u' and w', thereby forcing the robber to constantly move, while simultaneously preventing the robber from walking to $r_{\frac{n-16}{4}}$. Since that way the robber can't reach $r_{\frac{n-16}{4}}$ without getting captured, he cannot use the edge $(r_{\frac{n-16}{4}}, x_2)$. When the robber flees to the path P, using the edge $(p_{\frac{n}{8}}, p_{\frac{n}{8}-1})$, the cop can intercept the robber by moving to p_1 via $(r_{\frac{n-16}{4}}, x_2)$, (x_2, x_1), (x_1, r_1) and (r_1, p_1), and then start walking on path P until he reaches the robber. When the robber flees along the path Q or R, the cop can intercept the robber in a similar

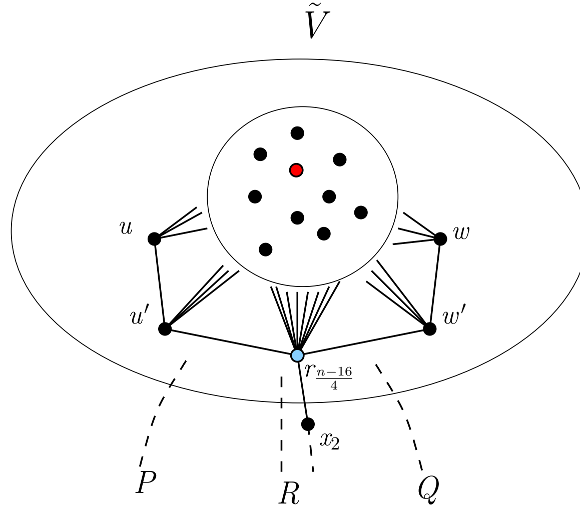


Figure 4.6: Case 5, the cop moves back and forth the nodes u', $r_{\frac{n-16}{4}}$ and w'

way.

At every turn, the strategy of the cop is as follows.

- If the robber is at a neighboring node, move to that node and capture him.
- If the robber is at u, move to u' along the shortest path.
- If the robber is at w, move to w' along the shortest path.
- If the robber is at a node in $\tilde{V} \backslash \{u, w\}$, move to $r_{\frac{n-16}{4}}$ along the shortest path.
- If the robber uses one of the paths P, Q or R, walk to r_1 along the shortest path and then intercept the robber.

Note that except for the last point, a shortest path of the strategy is of length at most 2. As described above, when the robber uses one of the paths P, Q or R, and the cop walks that path in the opposite direction, the cop captures the robber. Also, the cop is always at $r_{\frac{n-16}{4}}$ or a neighbor of $r_{\frac{n-16}{4}}$, so the robber can never move to $r_{\frac{n-16}{4}}$ without getting captured. Hence, altogether the robber cannot reach p_1 or q_1.

By Lemma 4.6, the robber will always be in the same connected component as v, his starting node. We will show that the cop will always be in the same component as v. Because the cop's strategy forces the robber to move at least every other turn ($dist(u', w') \leq 2$), and both stay in the connected component C of v, after at most $2 \cdot |C|$ turns the cop captures the robber or forces him to use a path P, Q or R, which also results in a cop's win.

If $(v, r_{\frac{n-16}{4}})$ still exists, the robber will not be able to delete this edge, because he cannot move to $r_{\frac{n-16}{4}}$ without getting captured, since the cop is always on $r_{\frac{n-16}{4}}$,

u' or w'. Hence, if $(v, r_{\frac{n-16}{4}})$ still exists, the cop is always in the same connected component as v. So, let us assume that $(v, r_{\frac{n-16}{4}})$ was deleted in the first two robber's turns. I.e. $v \in \{u, w\}$, say $v = u$. The robber cannot delete $(u', r_{\frac{n-16}{4}})$ without getting captured, because again the cop is too close to the node $r_{\frac{n-16}{4}}$. For the robber to move from u to u' and delete (u, u'), the cop has to have a distance from u' of at least 2. That can only happen, if the cop is on w'. But the cop only moves to w', if the robber is on w. As soon as the robber leaves w, according to the cop's strategy the cop will leave w' and walk to either u' or $r_{\frac{n-16}{4}}$. Therefore, the robber cannot delete the edge (u, u') without getting captured in the following turn. Also, the robber cannot delete the edge $(u', r_{\frac{n-16}{4}})$, because he cannot reach $r_{\frac{n-16}{4}}$ without getting captured. Thus, $v = u$ is always in the same connected component as $r_{\frac{n-16}{4}}$ and as the cop.

□

Finally, we prove a high capture time of G^* by devising a robber's strategy.

Theorem 4.9. *G^* has a capture time of at least $capt(G^*) \geq (\frac{n}{4} - 1) \cdot ((\frac{n}{4} - 2)^2 - 1)$.*

Proof. If the cop's starting position has a distance to p_1 of at least 2, the robber could start at p_1 and isolate himself in the next turn by moving to p^*. Analogously, cop's distance to q_1 has to be at most 1. So, cop's starting position must be a node in $\{p_1, q_1, r_1\}$.

Let us define the strategy for the robber. First, we let the robber start in x_2. In his first turn, he moves to $r_{\frac{n-16}{4}}$ and deletes the edge $(x_2, r_{\frac{n-16}{4}})$. Now, the distance of the cop to the induced subgraph $G^*|_{A \cup B}$ is at least $\min\{l(P), l(Q), l(R)\} + 1 \geq \frac{n}{8} - 1 + 1 \geq 9$. So, after 7 additional moves by the robber, the cop's position will still not be a node in $A \cup B$ or a neighbor of $A \cup B$. Therefore, the robber can make the following moves. He is aiming for the deletion of the edges $(r_{\frac{n-16}{4}}, p_{\frac{n}{8}})$ and $(r_{\frac{n-16}{4}}, q_{\frac{n}{8}})$:

1. Move to $p_{\frac{n}{8}}$ and delete the edge $(r_{\frac{n-16}{4}}, p_{\frac{n}{8}})$.
2. Move to $a_{\frac{n}{4}}$ and delete the edge $(p_{\frac{n}{8}}, a_{\frac{n}{4}})$.
3. Move to $r_{\frac{n-16}{4}}$ and delete the edge $(a_{\frac{n}{4}}, r_{\frac{n-16}{4}})$.
4. Move to $q_{\frac{n}{8}}$ and delete the edge $(r_{\frac{n-16}{4}}, q_{\frac{n}{8}})$.
5. Move to $b_{\frac{n}{4}}$ and delete the edge $(q_{\frac{n}{8}}, b_{\frac{n}{4}})$.
6. Move to $r_{\frac{n-16}{4}}$ and delete the edge $(b_{\frac{n}{4}}, r_{\frac{n-16}{4}})$.
7. Move to a_1 and delete the edge $(r_{\frac{n-16}{4}}, a_1)$.

After this sequence, the cop is not on a node in $A \cup B$ or a neighbor of $A \cup B$, hence he is somewhere in $P \cup Q \cup R \cup \{x_1, x_2, p^*, q^*\}$.

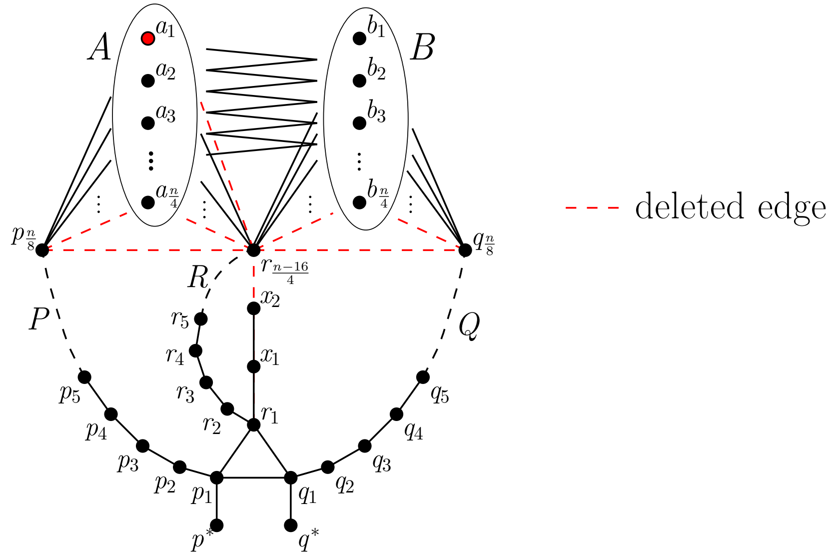


Figure 4.7: Graph G^* after robber's first 8 moves

Before stating the final part of robber's strategy, we elucidate the general idea behind the strategy. The path R is almost twice as long as the paths P or Q. If the cop would move to the node $r_{\frac{n-16}{4}-1}$, while the robber is in $A \cup B$, the distance of the robber to p^* is lower than the distance of the cop to p^*, therefore the robber could walk to p^* and isolate himself. So, using the path R is not beneficial for the cop and we can ignore this path. Also, we can ignore the nodes x_1 and x_2 for the cop, since the edge $(x_2, r_{\frac{n-16}{4}})$ was deleted by robber's first move. See Figure 4.8 for the essential parts of the graph.

The idea for the robber is not to move except the cop is on a neighboring node, or when the robber can win. When the robber is on a_1, for the cop to catch the robber, he has to move to a neighboring node, i.e. a node in $B \cup \{r_{\frac{n-16}{4}}, p_{\frac{n}{8}}\}$.

If at some point the cop moves to a node in $B \cup \{r_{\frac{n-16}{4}}\}$, the robber would move to $p_{\frac{n}{8}}$. At this point, because the edge $(r_{\frac{n-16}{4}}, p_{\frac{n}{8}})$ was deleted, the robber is not on a neighboring node of the cop. Also, the distance from the cop to p^* is by 2 higher than the distance from the robber to p^*. Hence, the robber could now just walk to p^* on P and isolate himself without the cop being able to interfere. Basically, when the cop comes too far around from one side, the robber can escape on the other side.

Therefore, when the robber is on a_1, the only way for the cop to reach a_1 without losing is by moving to $p_{\frac{n}{8}}$. At this point, the robber can move to e.g. b_1 (see figure 4.9). This situation is similar to the one above. If the cop would move from $p_{\frac{n}{8}}$ to a node in A, the robber can move to $q_{\frac{n}{8}}$, and then walk to q^* on the path Q, without being hindered the cop. Reaching q^*, he is isolated and thereby wins the game. So, for the cop to get to b_1, he has to walk to $q_{\frac{n}{8}}$ all the way on P and Q. As soon as the cop reaches $q_{\frac{n}{8}}$, the robber simply moves to e.g. a_2 and the same situation occurs.

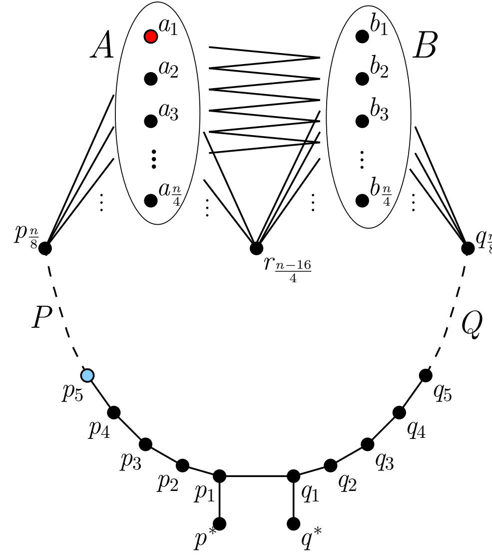


Figure 4.8: Essential parts of the graph.

With that strategy, the cop will keep walking back and forth between $p_{\frac{n}{8}}$ and $q_{\frac{n}{8}}$ along the paths P and Q. Each time, it will take the cop $l(P) + 1 + l(Q) = \frac{n}{4} - 1$ turns. The robber only has to move when the cop reaches $p_{\frac{n}{8}}$ resp. $q_{\frac{n}{8}}$, so the robber only has to move only once after $\frac{n}{4} - 1$ turns. When he can make $\Theta(n^2)$ such moves on $A \cup B$, it will take the cop $\Theta(n^3)$ turns to catch the robber.

Let $C := A \cup B \backslash \{a_{\frac{n}{4}}, a_{\frac{n}{4}-1}, b_{\frac{n}{4}}, b_{\frac{n}{4}-1}\}$. Since $G^*|_C$ is a complete bipartite graph, every node in $G^*|_C$ has $\frac{n}{4} - 2$ neighbors. $\frac{n}{8} \in \mathbb{N}$, so $\frac{n}{4} - 2$ is even. Hence, every node in $G^*|_C$ has even degree. Thus, there is an Euler tour $T := a_{i1}\text{-}b_{j1}\text{-}a_{i2}\text{-}b_{j2}\text{-}\cdots\text{-}a_{ik}\text{-}b_{jk}\text{-}a_{i1}$ in $G^*|_C$ with $k := \frac{n^2}{32} - \frac{n}{2} + 2$, since $G^*|_C$ has $(\frac{n}{4} - 2)^2 = \frac{n^2}{16} - n + 4$ edges. The robber will move along the Euler tour T every time he is forced to move by the cop.

Now, we will continue with the details of robber's strategy. As described above, 8 turns have passed. Robber's moves 1. to 8. were possible regardless of cop's moves, because cop's distance to the robber initially was at least $\frac{n}{8} \geq 9$. As mentioned above, after robber's 8th move, the cop's position will not be a node in $A \cup B$ or a neighbor of $A \cup B$. Hence, it will be in $P \cup Q \cup R \cup \{x_1, x_2, p^*, q^*\} \backslash \{p_{\frac{n}{8}}, q_{\frac{n}{8}}, r_{\frac{n-16}{4}}\}$. The remaining robber's strategy is as follows. Below, we will explain, why this strategy is possible.

At every robber's turn:

a) If the robber is in A and the cop is in $P \cup Q \cup R \cup \{x_1, x_2, p^*, q^*\} \backslash \{p_{\frac{n}{8}}, r_{\frac{n-16}{4}}\}$, he does not move.

b) If the robber is in B and the cop is in $P \cup Q \cup R \cup \{x_1, x_2, p^*, q^*\} \backslash \{q_{\frac{n}{8}}, r_{\frac{n-16}{4}}\}$, he does not move.

c) If the robber is in A and the cop is at $p_{\frac{n}{8}}$, he moves to the next node in the Euler

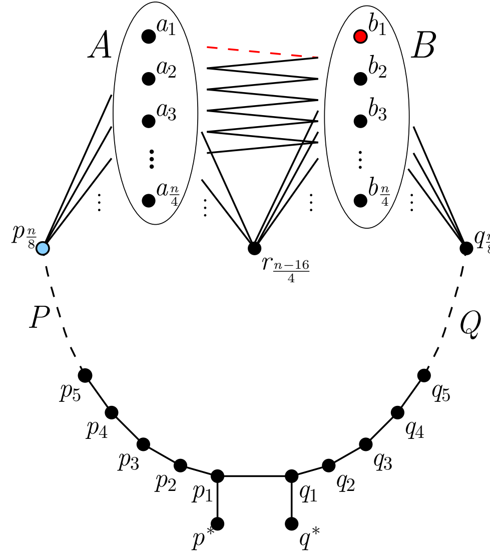


Figure 4.9: Cop on $p_{\frac{n}{8}}$, robber moved to b_1.

tour T.

d) If the robber is in B and the cop is at $q_{\frac{n}{8}}$, he moves to the next node in the Euler tour T.

e) If the robber is in A and the cop is in $B \cup \{r_{\frac{n-16}{4}}\}$, for the rest of the game he walks on the path P to the node p^*, then never moves again.

f) If the robber is in B and the cop is in $A \cup \{r_{\frac{n-16}{4}}\}$, for the rest of the game he walks on the path Q to the node q^*, then never moves again.

g) If c) or d) cannot be fulfilled because the Euler tour is finished, he does not move for the rest of the game.

The only cases, where the robber is not in $A \cup B$ are e) or f), where he left A on path P resp. B on path Q. There, the rest of robber's strategy is stated, too. Further, it cannot happen that the cop moves to A (resp. B), when the robber is still there, because the robber would have moved the turn before, according to c) or e) (resp. d) or f)). Therefore, all possible cases are covered.

Let the robber be in $A \cup B$, and w.l.o.g. be in A. We consider the cases a), c), e) and g).

Case a) When the cop is in $P \cup Q \cup R \cup \{x_1, x_2, p^*, q^*\} \setminus \{p_{\frac{n}{8}}, r_{\frac{n-16}{4}}\}$, he is not on a neighboring node of the robber. So, when the robber does not move, he will not get captured in the following turn.

Case c) When the cop is at $p_{\frac{n}{8}}$ and the robber moves to a node in B, the cop is no longer on a neighboring node of the robber, so the robber will not get captured in the following turn.

Case e) When the cop is in $B \cup \{r_{\frac{n-16}{4}}\}$, we have the situation illustrated in Figure 4.8. The robber will move to $p_{\frac{n}{8}}$. Because the edge $(r_{\frac{n-16}{4}}, p_{\frac{n}{8}})$ was deleted in the first 8 moves of the robber, cop and robber have a distance of 2. Since $(r_{\frac{n-16}{4}}, x_2)$ was deleted by the first move of the robber, the cop has a distance of at least $l(Q)+1+2 = \frac{n}{8}+2$ to p^*, while the robber has a distance of $l(P)+1 = \frac{n}{8}$ to p^*. The cop is too far away from the robber and the node p^*. Thus, the robber can follow the strategy stated in case e) and walks to p^* without being intercepted by the cop.

Case g) When c) is not an option anymore, i.e. when the robber finished walking on T, he stops to move entirely. In this case, the robber will get captured by the cop in at most $n-1$ additional turns.

Cases b), d) and f) work analogously.

If the robber is in A and the cop is in $P \cup Q \cup R \cup \{x_1, x_2, p^*, q^*\} \setminus \{p_{\frac{n}{8}}, r_{\frac{n-16}{4}}\}$, the cop wants to walk to the robber without causing case e), where then he would lose as shown above. So he must walk to the robber via the node $p_{\frac{n}{8}}$ and without entering nodes in $B \cup \{r_{\frac{n-16}{4}}\}$. That is only possible by walking along the path P. But then, we are in case c). Similarly, when the robber is in B and the cop is in $P \cup Q \cup R \cup \{x_1, x_2, p^*, q^*\} \setminus \{q_{\frac{n}{8}}, r_{\frac{n-16}{4}}\}$, the cop wants to walk to the robber preventing case f). The only way is to walk to the robber via the node $q_{\frac{n}{8}}$ and without entering nodes in $A \cup \{r_{\frac{n-16}{4}}\}$. That is only possible by walking along the path Q. But this is case d).

As shown in Figure 4.8, the cop will avoid the cases e) and f) by moving from $p_{\frac{n}{8}}$ to $q_{\frac{n}{8}}$, back and forth along the paths P and Q. Every time the cop reaches $p_{\frac{n}{8}}$ or $q_{\frac{n}{8}}$, the robber moves once along the Euler tour T. Otherwise, the robber does not move. Each time, the cop needs $l(P)+1+l(Q) = \frac{n}{4}-1$ turns to get from $p_{\frac{n}{8}}$ to $q_{\frac{n}{8}}$ or vice versa. With this strategy, the robber moves along T once every $\frac{n}{4}-1$ turns. T has $(\frac{n}{4}-2)^2$ edges.

So, altogether, the game takes at least $(\frac{n}{4}-1)\cdot((\frac{n}{4}-2)^2-1)$ turns, hence $\text{capt}(G^*) = \Omega(n^3)$. □

Since the function $f : \mathbb{N} \to \mathbb{N}, n \mapsto (\frac{n}{4}-1)\cdot((\frac{n}{4}-2)^2-1)$ is in $\Omega(n^3)$, Theorem 4.8 and 4.9 give the desired result.

Corollary 4.10. *There is a function $f : \mathbb{N} \to \mathbb{N}$ with $f \in \Omega(n^3)$ such that for every $n \in \mathbb{N}$ there is a graph G on n nodes with $c_b(G) = 1$ and $\text{capt}_b(G) \geq f(n)$.*

4.3 Suggestions for future work

We refer to [21] for open problems regarding the bridge-burning cops and robbers game, such as finding necessary and sufficient characteristics for graphs with a cop number of 1 (similar to [25] and [28] for the traditional game) and determining the cop number of specific classes of graphs.

We think that the consideration of multiple cooperating robbers might be of interest and suggest the following additions to the game.

- Player 2 controls a set of robbers.
- If it is the robbers' turn, each robber can either move from his position along an edge to a neighboring node or stay in position. After the turn every edge used by at least one robber is deleted.
- If the cops can capture at least one robber in a finite number of turns, they win the game.

With this setup, long paths are no longer possible for graphs with a low cop number, since two or more robbers can easily isolate themselves. This might result in graphs with interesting characteristics.

Chapter 5

Swap Equilibrium Graphs in the Extreme Vertex Destruction Model

5.1 Introduction

In this chapter, we take a look at the *extreme vertex destruction model*, a special kind of *network creation game*. Network creation games are a game- and graph-theoretic approach of analyzing the properties of networks created by a set of selfish performing agents in distinction from a centralized formed network. The players of this game are represented by the vertices of a graph. The game has only one turn, in which each player chooses a subset of the other players, the players he wants to connect to. After the turn, the edges of the graph are formed according to the rules of the specific game, for example an edge between two vertices is created if at least one of the represented players chose the other player in his turn. Finally, each player obtains a certain cost (or utility) depending on the rules of the game and the resulting graph. This cost could for example be the sum of distances to the other players in the graph.

The concept of a network forming by agents choosing to create and delete edges was introduced by Jackson and Wolinsky [20] in 1996 to study social and economic networks in terms of stability and efficiency. The network creation game itself was introduced by Fabrikant et al. [12] in 2003. In their setting, players could buy edges, and their costs of the players were determined by the number of bought edges and a function over the distances to the other players. They studied *Nash equilibria*, graphs in which no single player would be strictly better off by changing the strategy. An alternative for this concept was given by Alon et al. [3] in 2010 in the form of a so-called *swap equilibrium* (SE). A graph is called a swap equilibrium if no single player can strictly improve the cost by removing one of the incident edges and creating a new incident edge to some other player. The advantage over Nash equilibria lies in the reduced size of the strategy space, that does not grow exponentially with the number of players. Therefore, finding such equilibria is more manageable.

In 2017, Kliemann et al. [22] studied with the *vertex destruction model* a cost function representing the robustness of SE graphs. With a certain probability distribution, one

vertex is chosen and 'destroyed', i.e. all its incident edges are deleted. The number of separated vertex pairs after a vertex v is destroyed is called the *separation* of v. The cost of a player is the expected number of vertices he is separated from after the vertex destruction. They considered two types of probability distributions. The *uniform vertex destroyer* picks a vertex uniformly at random. The *extreme vertex destroyer* considers the set of so-called *max-sep vertices* and picks one of them uniformly at random. A vertex is a max-sep vertex if it has maximum separation over all vertices in the graph. The destruction of one of these vertices would cause the maximum amount of damage to a network. They showed that that an SE that is a tree and only has one max-sep vertex, has less than 8 vertices. They also presented a family of SE graphs with total cost of $\Omega(n^{3/2})$, where n is the number of vertices, showing the possible inefficiency of such graphs.

Kliemann et al. stated the following conjectures for the extreme vertex destruction model:

- There is only a finite number of SE graphs with exactly one max-sep vertex.
- There is only a finite number of SE trees.
- There is a family of SE graphs with total cost of $\Omega(n^2)$, where n is the number of vertices. This bound is tight.

The first two conjectures were tackled by Glazik [16] in 2019. He showed that an SE graph with exactly one max-sep vertex can only be a path of length 2 or 4 and that there are no SE trees on 6 or more vertices.

5.1.1 Our Contribution

We prove the third conjecture of Kliemann et al. and show that there is a family of SE graphs with a social cost of $\Omega(n^2)$, which is asymptotically optimal.

5.2 Preliminaries

Since proving the conjectures of Kliemann et al. was a joint work with Glazik, the definitions in this section are similar to those used in [16].

For $n \in \mathbb{N}$, let $[n] := \{1, \ldots, n\}$. Let $G = (V, E)$ be a graph and $H \subseteq V$ be a set of vertices. We write H for the induces subgraph $\left(H, E \cap \binom{H}{2}\right)$.

Let $n \in \mathbb{N}_{\geq 3}$ and let $\mathcal{G}_n$ be the set of all *connected* graphs on n vertices.

Definition 5.1 (Swap). *Let $G = (V, E) \in \mathcal{G}_n$ be a connected graph and $s = (a, b, c)$ with $a, b, c \in V$ a triple of vertices of G such that $\{a, b\} \in E$ and $\{a, c\} \notin E$. Denote by G^s the graph that is obtained from G by removing $\{a, b\}$ and inserting $\{a, c\}$. If G^s is a connected graph, we call s a* swap.

Next, we give the definition of a swap equilibrium. Therefor we consider the cost function of a general network creation game. The cost function will later be defined as the expected number of vertices a player is no longer connected to after the destruction of a vertex.

Definition 5.2 (Cost function and swap equilibrium). *A* cost function $\mathcal{C} : \mathcal{G}_n \times V \longrightarrow \mathbb{R}, (G, v) \mapsto \mathcal{C}(G, v)$ *assigns to each pair of a graph* $G \in \mathcal{G}_n$ *and vertex* $v \in V$ *a cost* $\mathcal{C}(G, v) \in \mathbb{R}$.

A graph $G \in \mathcal{G}_n$ *is called a* Swap equilibrium *(SE) if* $\mathcal{C}(G, a) \leq \mathcal{C}(G^s, a)$ *for all swaps* $s = (a, b, c)$*, i.e. if no single player can strictly reduce his cost by performing a swap.*

For defining the cost function used in the extreme vertex destruction model, we first need to clarify the *vertex destroyer*, which is used to assign each vertex in a graph a certain probability that it is picked and destroyed.

Definition 5.3 (Vertex destroyer). *A* vertex destroyer $\mathcal{D}$ *maps each graph* $G \in \mathcal{G}_n$ *to a probability measure* $\mathcal{D}(G, \cdot)$ *on the vertices* V *of* G.

In the extreme vertex destruction model, only vertices will be considered whose destruction causes maximum damage to the network. The amount of total damage is defined as *separation*. The damage caused to a single player is called *relevance*, and the cost for a player is defined as the expected damage caused to him. These terms will be clarified in the following definitions.

Definition 5.4 (Relevance and separation). *For* $u, v \in V$*, define*

$$\mathcal{R}_u(v) := \{w \in V | u \text{ lies on every } v\text{-}w\text{-path in G}\}.$$

The relevance *of* $u \in V$ *for* $v \in V$ *is defined as*

$$\mathrm{rel}_{G,u}(v) := \mathrm{rel}_u(v) := |\mathcal{R}_u(v)|.$$

Given a vertex destroyer $\mathcal{D}$*, the* cost *for a player* v *in* G *is defined as*

$$\mathcal{C}(G, v) := \sum_{u \in V} \mathrm{rel}_u(v) \mathcal{D}(G, u).$$

The separation *of* $v \in V$ *is defined as*

$$\mathrm{sep}_G(v) := \left| \bigcup_{u \in V} \{(u, w) | w \in \mathcal{R}_v(u)\} \right| = \sum_{u \in V} \mathrm{rel}_v(u).$$

As mentioned earlier, the separation $\mathrm{sep}_G(v)$ of a vertex $v \in V$ in a graph $G \in \mathcal{G}_n$ is the number of (ordered) vertex pairs that after the destruction of v, i.e. the deletion of all adjacent edges, are no longer in the same connected component. The relevance $\mathrm{rel}_u(v)$ of u for v is the number of vertices that are in a different connected component than v

after the destruction of u. Therefore, the cost $\mathcal{C}(G, v)$ of a player v in a graph G is the expected number of players he is no longer connected to after a vertex u is randomly chosen with probability $\mathcal{D}(G, u)$ and destroyed.

Since we are mostly talking about a specific graph, we will write $\text{sep}^s(u)$ instead of $\text{sep}_{G^s}(u)$ and $\text{rel}^s_u(v)$ instead of $\text{rel}_{G^s,u}(v)$.

The expected total cost is called *social cost*.

Definition 5.5 (Social cost)**.** *The* social cost *of a graph G is defined as*

$$\text{SC}(G) = \sum_{v \in V} \mathcal{C}(G, v).$$

Now, we can define the *extreme vertex destroyer*. The extreme vertex destroyer chooses and destroys one of the vertices with maximum separation uniformly at random.

Definition 5.6. *A vertex v is called a* max-sep vertex *if* $\text{sep}(v) \geq \text{sep}(u)$ *for all $u \in V$. Let* $\text{MS}(G)$ *be the set of all max-sep vertices in G. The* extreme vertex destroyer $\mathcal{D}_{\text{ev}}$ *is defined by*

$$\mathcal{D}_{\text{ev}}(G, v) = \begin{cases} 1/|\text{MS}(G)| & \text{if } v \in \text{MS}(G) \\ 0 & \text{else.} \end{cases}$$

To determine the separation of a vertex by counting separated vertex pairs can often get complicated. As alternative, Glazik [16] uses the notion of so-called *flaps*. The v-flaps of a vertex v are the connected components resulting from a destruction of v. The set of these connected components is defined as $\mathcal{F}(v)$. With the flaps of v, the separation can be computed the following way.

Proposition 5.7. *Let $G = (V, E)$, $v \in V$ and $\mathcal{F}(v) = \{A_1, \ldots, A_k\}$ for some $k \in \mathbb{N}$. Then,*

(i) For every $i \in [k]$ and every $u \in A_i$ it holds $\text{rel}_v(u) = n - |A_i|$.

(ii) $\text{sep}_G(v) = n^2 - 1 - \sum_{i=1}^{k} |A_i|^2$.

Proof. By definition of the flaps, for every $i \in [k]$ and $u \in A_i$ we have $\mathcal{R}_v(u) = V \setminus A_i$, so we get $\text{rel}_v(u) = n - |A_i|$ and (i) is proved. With $\text{rel}_v(v) = n - 1$, we get

$$\begin{aligned} \text{sep}_G(v) &= \sum_{u \in V} \text{rel}_v(u) \\ &= \text{rel}_v(v) + \sum_{i=1}^{k} \sum_{u \in A_i} \text{rel}_v(u) \\ &= n - 1 + \sum_{i=1}^{k} \sum_{u \in A_i} (n - |A_i|) \end{aligned}$$

$$= n - 1 + \sum_{i=1}^{k} |A_i|(n - |A_i|)$$

$$= n - 1 + (n-1)n - \sum_{i=1}^{k} |A_i|^2$$

$$= n^2 - 1 - \sum_{i=1}^{k} |A_i|^2.$$

□

5.3 A lower bound for the social cost of SE

Let $N \in \mathbb{N}$ arbitrary. In this section we give an example for an SE graph G on $n \geq N$ vertices and a social cost $\mathrm{SC}(G) = \Omega(n^2)$. Note that this is asymptotically maximal, since the social cost of each graph on n vertices is bounded by $\sum_{u \in V}(n-1) < n^2$. An illustration of such a graph G is given in Figure 5.1. We start with an auxiliary lemma.

Lemma 5.8. *For $k \in \mathbb{N}$ with $k \geq 10$ define*

$$\lambda(k) := \tfrac{1}{2}\left(\sqrt{11k^2 + 4k + 5} - 3k - 3\right)$$

$$\mu(k) := \tfrac{1}{2}\left(\sqrt{11k^2 + 4k - 3} - 3k - 1\right)$$

and let $I_k \subseteq \mathbb{R}$ be the interval $I_k := (\lambda(k), \mu(k))$. Then for $k_0 \in \mathbb{N}_{\geq 10}$ I_{k_0} or I_{k_0+1} contains an integer ℓ.

Proof. First note that for the derivative λ', we have

$$\lambda'(k) = \frac{1}{2}\left(\frac{11k + 2}{\sqrt{11k^2 + 4k + 5}} - 3\right) \to \frac{\sqrt{11} - 3}{2} \text{ for } k \to \infty.$$

So for $k \geq 7$, $\lambda'(k) \in (0.15, 0.16)$. On the other hand, $\mu(k) - \lambda(k) \overset{k\to\infty}{\longrightarrow} 1$, so for $k \geq 6$ we have $\mu(k) - \lambda(k) \in (0.9, 1)$. Now let $k_0 \geq 10$. Then, $\lambda(k_0) > 0$. Because $\lambda'(k) \in (0.15, 0.16)$ for all $k \in [k_0, k_0 + 1]$, by the mean value theorem we get that $\lambda(k_0 + 1) \in (\lambda(k_0) + 0.15, \lambda(k_0) + 0.16) \subset (\lambda(k_0), \mu(k_0)) = I_{k_0}$. Since

$$\mu(k_0 + 1) > \lambda(k_0 + 1) + 0.9 > \lambda(k_0) + 0.15 + 0.9 = \lambda(k_0) + 1.05,$$

the set $I_{k_0} \cup I_{k_0+1}$ forms an interval $(\lambda(k_0), \mu(k_0 + 1))$. Its length is at least 1.05 and therefore contains an integer ℓ. □

Remark 5.9. *For $k \geq 10$ we have $\lambda'(k) \in (0.15, 0.16)$, so $\lambda(k) \in (0.15k, 0.16k)$. Similarly, it holds $\mu'(k) \in (0.15, 0.16)$, so $\mu(k) \in (0.15k, 0.16k)$.*

Definition 5.10. *For $k, \ell \in \mathbb{N}$ define the graph $G_{k,\ell} = (V, E)$ depicted in Figure 5.1 as follows. The base is a K_4 with vertices named u, v, w_0 and w_0'. The vertices w_0 and w_0' are connected to $k-1$ leaves each, named $w_1, \ldots, w_{k-1}$ and $w_1', \ldots, w_{k-1}'$, respectively. The vertex u is connected to two stars; one star with center x_0 and leaves $x_1, \ldots, x_{k-1}$ and the other star with center y_0 and leaves $y_1, \ldots, y_{\ell-1}$. The vertex v is analogously connected to two stars with vertices named $x_0', x_1', \ldots, x_{k-1}'$ and $y_0', y_1', \ldots, y_{\ell-1}'$. In total, $G_{k,\ell}$ contains $n := 4k + 2\ell + 2$ vertices.*

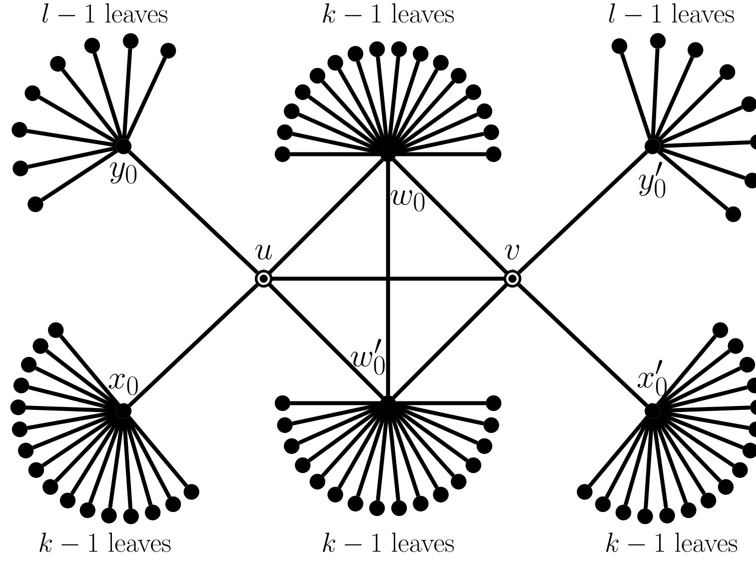


Figure 5.1: SE graph $G_{k,\ell}$ with high social cost.

Theorem 5.11. *For $k \in \mathbb{N}$ with $k \geq 353$ and $\ell \in I_k \cap \mathbb{N}$ the graph $G_{k,\ell}$ is an SE on $n = 4k + 2\ell + 2$ vertices and* $\mathrm{SC}(G_{k,\ell}) > 0.41n^2 = \Omega(n^2)$.

Proof. First we prove that v and u are the only max-sep vertices. We compute the separation of the different vertices using Proposition 5.7 (ii) and the fact that $n = 4k + 2\ell + 2$:

$$\begin{aligned}
\mathrm{sep}(u) &= n^2 - 1 - \left(k^2 + \ell^2 + (n - k - \ell - 1)^2\right) \\
&= n^2 - 1 - k^2 - \ell^2 - n^2 - k^2 - \ell^2 - 1 + 2nk + 2k\ell + 2n - 2k\ell - 2k - 2\ell \\
&= 2n\,(k + \ell + 1) - 2k^2 - 2\ell^2 - 2k\ell - 2k - 2\ell - 2 \\
&= 2\,(4k + 2\ell + 2)\,(k + \ell + 1) - 2k^2 - 2\ell^2 - 2k\ell - 2k - 2\ell - 2 \\
&= 8k^2 + 8k\ell + 8k + 4k\ell + 4\ell^2 + 4\ell + 4k + 4\ell + 4 \\
&\quad - 2k^2 - 2\ell^2 - 2k\ell - 2k - 2\ell - 2
\end{aligned}$$

$$= 6k^2 + 2\ell^2 + 10k\ell + 10k + 6\ell + 2 \tag{5.1}$$

$$\begin{aligned}
\mathrm{sep}(w_0) &= n^2 - 1 - \left(k - 1 + (n-k)^2\right) \\
&= n^2 - 1 - k + 1 - n^2 + 2nk - k^2 \\
&= 2nk - k^2 - k \\
&= 2\,(4k + 2\ell + 2)\,k - k^2 - k \\
&= 8k^2 + 4k\ell + 4k - k^2 - k \\
&= 7k^2 + 4k\ell + 3k
\end{aligned} \tag{5.2}$$

$$\begin{aligned}
\mathrm{sep}(y_0) &= n^2 - 1 - \left(\ell - 1 + (n-\ell)^2\right) \\
&= n^2 - 1 - \ell + 1 - n^2 + 2n\ell - \ell^2 \\
&= 2n\ell - \ell^2 - \ell \\
&= 2\,(4k + 2\ell + 2)\,\ell - \ell^2 - \ell \\
&= 8k\ell + 4\ell^2 + 4\ell - \ell^2 - \ell \\
&= 3\ell^2 + 8k\ell + 3\ell
\end{aligned} \tag{5.3}$$

For symmetry reasons we have $\mathrm{sep}(v) = \mathrm{sep}(u)$, $\mathrm{sep}(y_0') = \mathrm{sep}(y_0)$ and $\mathrm{sep}(x_0) = \mathrm{sep}(x_0') = \mathrm{sep}(w_0) = \mathrm{sep}(w_0')$. By assumption $\ell \in I \cap \mathbb{N}$, so $\ell > \lambda(k)$. A straightforward calculation gives

$$\begin{aligned}
\mathrm{sep}(u) - \mathrm{sep}(w_0) &= 2\ell^2 + 6k\ell + 7k + 6\ell + 2 - k^2 \\
&> 2(\lambda(k))^2 + 6k\lambda(k) + 7k + 6\lambda(k) + 2 - k^2 \\
&= 2\left(\frac{1}{2}\left(\sqrt{11k^2+4k+5} - 3k - 3\right)\right)^2 \\
&\quad + 6k\left(\frac{1}{2}\left(\sqrt{11k^2+4k+5} - 3k - 3\right)\right) + 7k \\
&\quad + 6k\left(\frac{1}{2}\left(\sqrt{11k^2+4k+5} - 3k - 3\right)\right) + 2 - k^2 \\
&= 2 \cdot \frac{1}{4}\left(\left(\sqrt{11k^2+4k+5}\right)^2 + (-3k-3)^2\right) \\
&\quad + 2 \cdot \frac{1}{4} \cdot \left(2\sqrt{11k^2+4k+5}\,(-3k-3)\right) \\
&\quad + 3k\sqrt{11k^2+4k+5} + 3k\,(-3k-3) + 7k \\
&\quad + 3\sqrt{11k^2+4k+5} + 3\,(-3k-3) + 2 - k^2 \\
&= \frac{1}{2}\left(11k^2+4k+5\right) + \frac{1}{2}(-3k-3)^2 + \sqrt{11k^2+4k+5}\,(-3k-3) \\
&\quad + 3k\sqrt{11k^2+4k+5} - 9k^2 - 9k + 7k
\end{aligned}$$

$$
\begin{aligned}
&+3\sqrt{11k^2+4k+5}-9k-9+2-k^2\\
&=\sqrt{11k^2+4k+5}\,(-3k-3+3k+3)\\
&\quad+\frac{1}{2}\left(11k^2+4k+5\right)+\frac{1}{2}\left(-3k-3\right)^2\\
&\quad-9k^2-9k+7k-9k-9+2-k^2\\
&=\frac{11}{2}k^2+2k+\frac{5}{2}+\frac{1}{2}\left(9k^2+9+2\cdot 3k\cdot 3\right)\\
&\quad-9k^2-9k+7k-9k-9+2-k^2\\
&=\frac{11}{2}k^2+2k+\frac{5}{2}+\frac{9}{2}k^2+\frac{9}{2}+9k\\
&\quad-9k^2-9k+7k-9k-9+2-k^2\\
&=\left(\frac{11}{2}+\frac{9}{2}-9-1\right)k^2+(2+9-9+7-9)\,k+\left(\frac{5}{2}+\frac{9}{2}-9+2\right)\\
&=0
\end{aligned}
$$

With Remark 5.9 we have $l \in I_k \subset (0.15k, 0.16k)$. Thus it holds $\ell < k$ and

$$\mathrm{sep}(w_0) = 7k^2 + 4k\ell + 3k > 4k\ell + 3\ell^2 + 4k\ell + 3\ell = \mathrm{sep}(y_0).$$

In total we have $\mathrm{sep}(u) > \mathrm{sep}(w_0) > \mathrm{sep}(y_0)$. Because all leaves have a smaller separation, u and v are the only max-sep vertices in $G_{k,\ell}$.

We turn to the SE property and show that none of the possible swaps is profitable. Denote

$$
\begin{aligned}
A &:= \{u\} \cup \{x_i | i \in \{0,\dots,k-1\}\} \cup \{y_i | i \in \{0,\dots,\ell-1\}\\
B &:= \{v\} \cup \{x_i' | i \in \{0,\dots,k-1\}\} \cup \{y_i' | i \in \{0,\dots,\ell-1\}\\
W &:= \{w_i | i \in \{0,\dots,k-1\}\} \cup \{w_i' | i \in \{0,\dots,k-1\}\}.
\end{aligned}
$$

For a compact argumentation we need the following lemma based on the notations developed in the proof so far. After the lemma and its proof we will continue the proof of the theorem. We consider certain swaps that result in a unique max-sep vertex.

Lemma 5.12. *Let $z, \bar{z}$ with $z \notin \{u, v, x_0, x_0', y_0, y_0', w_0, w_0'\}$ be vertices in $G_{k,\ell}$.*

i) If $s = (z, \bar{z}, a)$ is a swap such that $a \in A$ (resp. $a \in B$) and u (resp. v) becomes the only max-sep vertex of $G^s_{k,\ell}$, then the swap is not profitable for z.

ii) If $s = (z, \bar{z}, a)$ is a swap such that $a \in \{w_i | i \in \{0,\dots,k-1\}\}$ (resp. $a \in \{w_i' | i \in \{0,\dots,k-1\}\}$) and w_0 (resp. w_0') becomes the only max-sep vertex of $G^s_{k,\ell}$, then the swap is not profitable for z.

Proof. Since u and v are the only max-sep vertices of $G_{k,\ell}$, we have $\mathcal{C}(G_{k,\ell}, z) = \mathrm{rel}_{G_{k,\ell},u}(z)/2 + \mathrm{rel}_{G_{k,\ell},v}(z)/2$. Note that $\mathrm{rel}_{G_{k,\ell},u}(z) < n$ and $\mathrm{rel}_{G_{k,\ell},v}(z) < n$. If $z \in A$, then $\mathrm{rel}_{G_{k,\ell},v}(z) = |B| = k + \ell + 1$, hence $\mathcal{C}(G_{k,\ell}, z) < n/2 + (k + \ell + 1)/2$. If $z \in B \cup W$, then $\mathrm{rel}_{G_{k,\ell},u}(z) = |A| = k + \ell + 1$, hence $\mathcal{C}(G_{k,\ell}, z) < (k + \ell + 1)/2 + n/2$

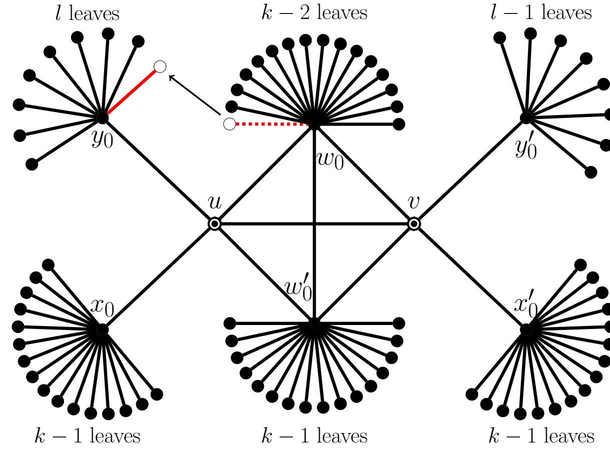


Figure 5.2: Lemma 5.12: example of swap $s = (z, \bar{z}, a)$

With $l < 0.16$ and $k \geq 353$ we have $\mathcal{C}(G_{k,\ell}, z) < (n+k+l+1)/2 = (5k+3l+3)/2 \leq \frac{5}{2}k + \frac{3}{2}(0.16k+1) < 3k$.

i) Let $a \in A$ and let u be the only max-sep vertex of $G^s_{k,\ell}$. Then $\mathcal{C}(G^s_{k,\ell}, z) = \text{rel}_{G^s_{k,\ell},u}(z)$. We have $a \in A$, so in $G^s_{k,\ell}$, u is on the unique path between z and each vertex in $B \cup W \backslash \{z\}$ (see Figure 5.2). $|B \cup W \backslash \{z\}| > 3k$, therefore $\mathcal{C}(G^s_{k,\ell}, z) > 3k$. Hence, the swap would not be profitable for z.

ii) Let $a \in \{w_i | i \in \{0, \dots, k-1\}\}$ and let w_0 be the only max-sep vertex of $G^s_{k,\ell}$. Then $\mathcal{C}(G^s_{k,\ell}, z) = \text{rel}_{G^s_{k,\ell},w_0}(z)$. We have $a \in \{w_i | i \in \{0, \dots, k-1\}\}$, so in $G^s_{k,\ell}$, w_0 is on the unique path between z and each vertex in $A \cup B \cup \{w'_i | i \in \{0, \dots, k-1\}\} \backslash \{z\}$. $|A \cup B \cup \{w'_i | i \in \{0, \dots, k-1\}\} \backslash \{z\}| > 3k$, therefore $\mathcal{C}(G^s_{k,\ell}, z) > 3k$. Hence, the swap would not be profitable for z. ■ □

Now, consider all possible swaps, grouped by the vertex who performs the swap.

- $x_i, i \in \{1, \dots, k-1\}$: Let $i \in \{1, \dots, k-1\}$. All swaps performed by x_i are of the form $s = (x_i, x_0, a)$ for some vertex a.
  - For $a = x_j$ for some $j \in [k-1] \setminus \{i\}$, only the paths from x_i to the other vertices change. Now, these paths contain x_j. Therefore, the separation of x_j increases to $n^2 - 1 - (n-2)^2 - 1 = 4n - 6$ (Proposition 5.7 (ii)), while the separations of the other vertices stay the same. Hence, u and v remain the only max-sep vertices and all costs stay the same.
  - For $a \in \{u\} \cup \{y_j | j \in \{0, \dots \ell - 1\}\}$, again only the paths from x_i to the other vertices change. So, for $a \in \{y_j | j \in \{0, \dots \ell - 1\}$ the increase in separation is at most linear in n and thereby insignificant compared to the separations of u and v. In $G_{k,\ell}$, u is on the path from x_i to the vertices in $X := W \cup B \cup \{u\} \cup \{y_j | j \in$

$\{0, \ldots \ell - 1\}$ with $|X| = 3k + 2l + 2$, In $G^s_{k,\ell}$, u is on the path from x_i to at least the vertices in $Y := W \cup B \cup \{u\} \cup \{x_j | j \in \{0, \ldots \ell - 1\}$ with $|Y| = 4k + l + 2 > |X|$, since $l < 0.16k$. Hence, the relevance of u for x_i increases, so the separation of u increases and u becomes the only max-sep vertex in $G^s_{k,\ell}$. By Lemma 5.12 (i), the swap is not profitable for x_i.

– For $a \in B$, the relevance of v for x_i increases, since in $G^s_{k,\ell}$, v lies on the paths from x_i to the vertices in $A \cup W$, while in $G_{k,\ell}$ v only lies on the paths from x_i to B. The relevance of u for x_i decreases, since in $G^s_{k,\ell}$, u only lies on the paths from x_i to the vertices in A, while in $G_{k,\ell}$ u lies on the paths from x_i to $W \cup B \cup \{u\}$. So, the separation of u for decreases, while the separation of v increases. Hence, u becomes the only max-sep vertex in $G^s_{k,\ell}$. By Lemma 5.12 (i), the swap is not profitable for x_i.

– Let $a \in W$, w.l.o.g. $a = w_j$ for some $j \in \{0, \ldots, k-1\}$. We show that w_0 is the only max-sep vertex of $G^s_{k,\ell}$, so the swap is not profitable by Lemma 5.12 (ii). For $j \neq 0$ we have with Proposition 5.7 (ii) as $\{x_i, a\}$ is now a component,

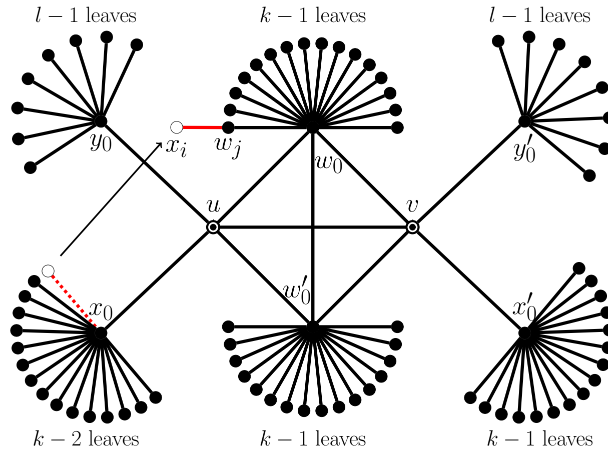


Figure 5.3: Swap $s = (x_i, x_0, a)$ with $a \in W$

$$\begin{aligned} \text{sep}^s(w_0) &= n^2 - 1 - ((k-2) + 2^2 + (n-k-1)^2) \\ &= 2n(k+1) - k^2 - 3k - 4 \\ &= 7k^2 + 4k\ell + 9k + 4\ell \end{aligned} \tag{5.4}$$

For visualization of the following arguments, see Figure 5.3. Only the paths from x_i to the other vertices change with the swap s. For the vertices, that are leaves in $G_{k,\ell}$ and $G^s_{k,\ell}$, i.e. $z \in Z := \{x_h | h \in \{1, \ldots, i-1, i+1 \ldots, k-1\}\} \cup \{y_h | h \in \{1, \ldots, \ell-1\}\} \cup \{x'_h | h \in \{1, \ldots, k-1\}\} \cup \{y'_h | h \in \{1, \ldots, \ell-1\}\} \cup \{w'_h | h \in \{1, \ldots, k-1\}\} \cup \{w_h | h \in \{1, \ldots, k-1\} \backslash \{j\}\}$, z lies in $G_{k,\ell}$

and $G^s_{k,\ell}$ only on the x_i-path to z, so the relevance of z for x_i, and thereby the separation of z does not change. The relevance of u for x_i decreases, since in $G^s_{k,\ell}$, u only lies on the paths from x_i to the vertices in A, while in $G_{k,\ell}$ u lies on the paths from x_i to the vertices in $W \cup B \cup \{u\}$. The relevance of x_0 for x_i decreases, since in $G^s_{k,\ell}$, x_0 only lies on the paths from x_i to the vertices in $\{x_h | h \in \{0, \dots, i-1, i+1 \dots, k-1\}\}$, while in $G_{k,\ell}$ x_0 lies on the paths from x_i to more than the vertices in $W \cup B \cup \{u\}$. So, the separations of u and x_0 decrease. The relevance of $z \in \{v, x'_0, y_0, y'_0, w'_0\}$ for x_i stays the same, because z lies on the x_i-paths to the exact same vertices in $G_{k,\ell}$ and $G^s_{k,\ell}$. So, the separation of z stays the same.
Therefore, for all vertices $b \in V \setminus \{w_0, w_j\}$ the separation does not increase by swap s, since the relevance of b for x_i either decreases or stays the same. So, since u is a max-sep vertex in $G_{k,\ell}$, we have $\text{sep}^s(b) \leq \text{sep}(b) \leq \text{sep}(u)$ and because $\ell < \mu(k)$, we get with 5.1 and 5.4, as $\ell < \mu(k)$,

$$\begin{aligned}
\text{sep}^s(w_0) - \text{sep}(u) &= 7k^2 + 4kl + 9k + 4l \\
&\quad - \left(6k^2 + 2l^2 + 10kl + 10k + 6l + 2\right) \\
&= k^2 - 2\ell^2 - 6k\ell - k - 2\ell - 2 \\
&> k^2 - 2\mu(k)^2 - 6k\mu(k) - k - 2\mu(k) - 2 \\
&= k^2 - 2 \cdot \left(\frac{1}{2}\left(\sqrt{11k^2 + 4k - 3} - 3k - 1\right)\right)^2 \\
&\quad - 6k \cdot \frac{1}{2}\left(\sqrt{11k^2 + 4k - 3} - 3k - 1\right) - k \\
&\quad - 2 \cdot \frac{1}{2}\left(\sqrt{11k^2 + 4k - 3} - 3k - 1\right) - 2 \\
&= k^2 - \frac{1}{2}\left(11k^2 + 4k - 3\right) \\
&\quad - (-3k - 1)\sqrt{11k^2 + 4k - 3} \\
&\quad - \frac{1}{2}(-3k - 1)^2 - 3k\sqrt{11k^2 + 4k - 3} \\
&\quad - 3k(-3k - 1) - k - \sqrt{11k^2 + 4k - 3} + 3k + 1 - 2 \\
&= k^2 - \frac{11}{2}k^2 - 2k + \frac{3}{2} + (3k + 1)\sqrt{11k^2 + 4k - 3} \\
&\quad - \frac{9}{2}k^2 - 3k - \frac{1}{2} - 3k\sqrt{11k^2 + 4k - 3} \\
&\quad + 9k^2 + 3k - k - \sqrt{11k^2 + 4k - 3} + 3k + 1 - 2 \\
&= (3k + 1 - 3k - 1)\sqrt{11k^2 + 4k - 3} \\
&\quad + \left(1 - \frac{11}{2} - \frac{9}{2} + 9\right)k^2 + (-2 - 3 + 3 - 1 + 3)k \\
&\quad + \left(\frac{3}{2} - \frac{1}{2} + 1 - 2\right)
\end{aligned}$$

$$= 0.$$

Hence

$$\mathrm{sep}^s(w_0) > \mathrm{sep}(u) > \mathrm{sep}(b) > \mathrm{sep}^s(b) \text{ for all } b \in V \backslash \{w_0\}.$$

Thus, w_0 is the only max-sep vertex of G^s and the swap is not profitable by Lemma 5.12 (ii). Finally note that

$$\mathrm{sep}^{(x_i,x_0,w_0)}(w_0) = \mathrm{sep}^{(x_i,x_0,w_j)}(w_0) + 2$$

for every $j \in [k-1]$, because here the deletion of w_0 would also separate x_i and w_j, so the same argument applies for $a = w_0$.

- $y_i, i \in [\ell-1]$: Let $i \in [\ell-1]$. All swaps performed by y_i are of the form $s = (y_i, y_0, a)$ for some vertex a.

  For $a = x_j$ and $j \in \{0, \dots, k-1\}$ or $a \in W$, the swap is not profitable with the same arguments as in the case of the swap $(x_i, x_0, a), a \in W$. The cases $a = u$ or $a \in B$ are analog to the cases (x_i, x_0, u) and $(x_i, x_0, a), a \in B$ respectively.

- $w_i, i \in [k-1]$: Let $i \in [k-1]$. All swaps performed by w_i are of the form $s = (w_i, w_0, a)$ for some vertex a.

  If $a = w_j$ for $j \in [k-1]$, the only change is that w_j is added to every w_i-path (except the w_i-w_j-path). So, the separation of w_j increases by $2(n-2)$, while the separations of the other vertices stay the same. So, u and v are still the only max-sep vertices after the swap. Also, because of the argument above, $\mathrm{rel}_{G_{k,\ell},w_i}(u) = \mathrm{rel}_{G^s_{k,\ell},w_i}(u)$ and $\mathrm{rel}_{G_{k,\ell},w_i}(v) = \mathrm{rel}_{G^s_{k,\ell},w_i}(v)$, thus the swap is not profitable for w_i. The case $a = w'_j, j \in \{0, \dots, k-1\}$ is analog to the case $(x_i, x_0, a), a \in W$ and the cases $a \in A$ and $a \in B$ are both analog to the case $(x_i, x_0, a), a \in B$.

- x_0: All swaps performed by x_0 are of the form $s = (x_0, u, a)$ for some vertex a.

  - The arguments for the cases $a \in W$ and $a = x'_j, j \in \{0, \dots \ell-1\}$ are similar to the case $(x_i, x_0, a), a \in W$ and the case $(x_i, x_0, a), a = x'_j, j \in \{0, \dots \ell-1\}$. The separation of w_0 (or w'_0, respectively) increases even stronger than in that case.
  - For $a = y'_j, j \in \{0, \dots, \ell-1\}$, see Figure 5.4. The separation of u decreases, since u is no longer on the paths from the vertices in $\{x_h, h \in \{0, \dots, k-1\}\}$ to the vertices in $W \cup B$. The swap increases the separation of v, since before the swap v was on the paths from the vertices in $\{x_h, h \in \{0, \dots, k-1\}\}$ to the vertices in B, while after the swap v is on the paths from the vertices in $\{x_h, h \in \{0, \dots, k-1\}\}$ to at least the vertices in W ($|B| < |W|$). Similarly, the swap increases the separation of y'_0. For the other vertices the separation stays the same because they separate the exact same set of vertex pairs before and after the swap. So, after the swap v and y'_0 are the only possible max-sep

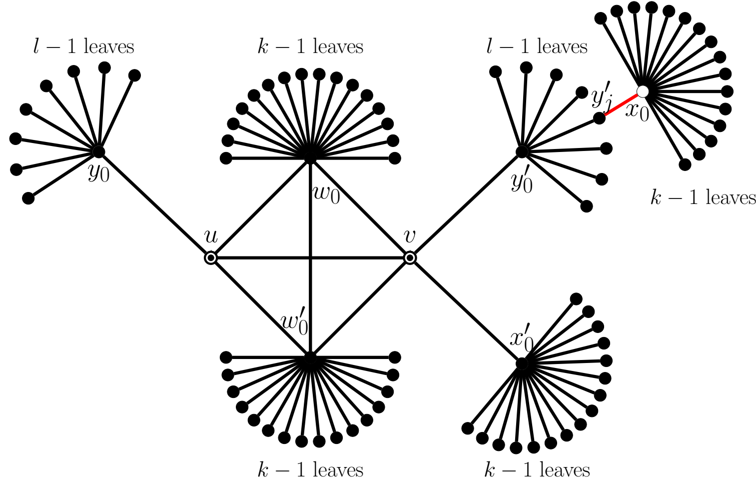


Figure 5.4: Swap $s = (x_0, u, y'_j)$ with $j \in \{0, \ldots, \ell - 1\}$

vertices. Both vertices separate x_0 and the set $X := W \cup \{u\} \cup \{x'_h, h \in \{0, \ldots, k-1\}\}$ with $|X| > 3k$. So, we have $\mathcal{C}(G^s_{k,\ell}, x_0) > 3k$, while $\mathcal{C}(G_{k,\ell}, x_0) < 3k$ by the calculation in the proof of Lemma 5.12. Hence, the swap is not profitable for x_0.

– For $a = y_j$ for some $j \in \{0, \ldots, \ell - 1\}$, see Figure 5.5. We show that y_0 is the only max-sep vertex of $G^s_{k,\ell}$, so the swap is not profitable. Let $j > 0$. If y_j is destroyed, we have $\ell - 2$ single vertices y_h, $h \neq j$, a component $\{x_h, h \in \{0, \ldots, k-1\}\} \cup \{y_j\}$ and a component with the rest of the vertices. We get by Proposition 5.7 (ii)

$$\begin{aligned}
\operatorname{sep}^s(y_0) &= n^2 - 1 - (\ell - 2) - (k+1)^2 - (n - 1 - (\ell - 2) - (k+1))^2 \\
&= n^2 - 1 - \ell + 2 - (k+1)^2 - (n - l - k)^2 \\
&= n^2 - 1 - \ell + 2 - k^2 - 2k - 1 \\
&\quad - n^2 - \ell^2 - k^2 + 2n\ell + 2nk - 2\ell k) \\
&= n^2(1-1) + 2n\ell + 2nk - 2k\ell - k^2(1+1) \\
&\quad - \ell^2 - \ell - 2k + (-1 + 2 - 1) \\
&= 2n(k+\ell) - 2k\ell - 2k^2 - 2k - \ell^2 - \ell \\
&= 2(4k + 2\ell + 2)(k+\ell) - 2k\ell - 2k^2 - 2k - \ell^2 - \ell \\
&= 8k^2 + 4k\ell + 4k + 8k\ell + 4\ell^2 + 4\ell - 2k\ell - 2k^2 - 2k - \ell^2 - \ell \\
&= k^2(8-2) + \ell^2(4-1) + k\ell(4+8-2) + k(4-2) + \ell(4-1) \\
&= 6k^2 + 3\ell^2 + 10k\ell + 2k + 3\ell
\end{aligned} \tag{5.5}$$

The separation of u decreases with the swap, because now u is no longer on

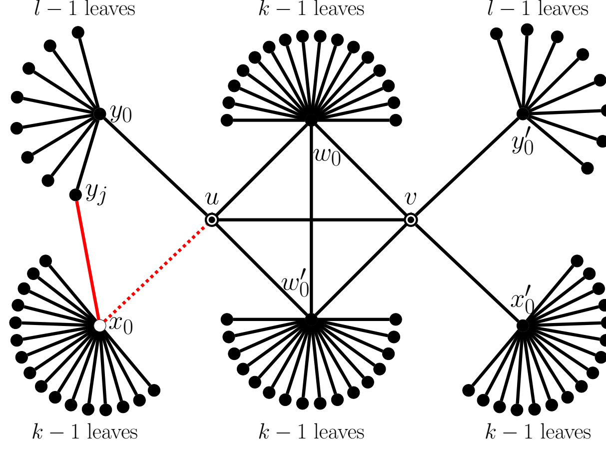


Figure 5.5: Swap $s = (x_0, u, y_j)$ with $j \in \{0, \ldots, \ell-1\}$

the paths between the vertices of $\{x_h, h \in \{0, \ldots, k-1\}\}$ and the vertices of $\{y_h, h \in \{0, \ldots, \ell-1\}\}$ For all vertices $b \in V \setminus \{y_0, y_j, u\}$ the separation stays the same, because b separates the exact set of vertex pairs before and after the swap (see Figure 5.5). We already showed that u is a max-sep vertex, so $\text{sep}^s(b) \leq \text{sep}(b) \leq \text{sep}(u)$ We have with 5.1, 5.5 and with $\ell \in I_k \cap \mathbb{N}$, so $\ell > \lambda(k)$,

$$\begin{aligned}\text{sep}^s(y_0) - \text{sep}(u) &= \ell^2 - 8k - 3\ell - 2 \\ &= \ell(\ell-3) - 8k - 2 \\ &> \lambda(k)(\lambda(k)-3) - 8k - 2\end{aligned}$$

The latter term is non-negative if $\lambda(k) \geq \sqrt{\frac{17}{4} + 8k} + \frac{3}{2}$. Since $\lambda(k) \overset{k\to\infty}{\longrightarrow} \frac{\sqrt{11}-3}{2} \cdot k$, this inequality is fulfilled if k is sufficiently large. In fact, for all $k \geq 353$ the inequality holds, so y_0 is the only max-sep vertex of $G^s_{k,\ell}$. Note that

$$\text{sep}^{(x_0,u,y_0)}(y_0) = \text{sep}^{(x_0,u,y_j)}(y_0) + 2k$$

for every $j \in [k-1]$, since in this case y_0 also separates y_j and the k vertices in $\{x_h, h \in \{0, \ldots, k-1\}\}$. Thus the same argument applies for $a = y_0$.

- y_0: All swaps performed by y_0 are of the form $s = (y_0, u, a)$ for some vertex a.

  The cases $a = x_i$ or $a \in W$ are not profitable for the same reasons as in the case $(x_i, x_0, a), a \in W$. The case $a \in B$ is analog to the case $(x_i, x_0, a), a \in B$.

- w_0: For a swap, w_0 has to delete one of the edges $\{w_0, u\}, \{w_0, v\}$, or $\{w_0, w_0'\}$. As already mentioned, in all cases the subgraph remains two-connected, so it does not

matter which edge is deleted and we may assume that all swaps performed by w_0 are of the form $s = (w_0, u, a)$ for some vertex a.

The deletion of $\{w_0, u\}$ does not change any separation, since there are still two vertex-disjoint w_0-u-paths in the resulting graph. It follows that $\text{sep}^s(b) \leq \text{sep}(b)$ for all $b \in V$.

If $a \in A$ (resp. $a \in B$), the separation of v (resp. u) will stay the same. If $a \in W$, the separation of u and v will stay the same. So, either u or v is the only max-sep vertex of $G^s_{k,\ell}$, or $\text{MS}(G^s_{k,\ell}) = \{u, v\}$. Because of the symmetry of $G_{k,\ell}$, in all cases the cost for w_0 remains the same.

- u: This case is quite similar to the above case. Analogously, we may assume that all swaps performed by u are of the form $s = (u, v, a)$ for some vertex a and we know that $\text{sep}^s(b) \leq \text{sep}(b)$ for all $b \in V$.

  Because $\text{sep}^s(u) = \text{sep}(u)$, there are only two different cases to consider: If $a \in B$, then $\text{sep}^s(v) < \text{sep}(v)$ and u is the only max-sep vertex of $G^s_{k,\ell}$. Otherwise, u and v stay the only max-sep vertices and the cost of u does not change.

All possible swaps of $v, w'_i, x'_i$and y'_i are symmetric to the swaps of u, w_i, x_i and y_i that have already been considered.

We showed that $G_{k,\ell}$ is a swap equilibrium. For the social cost, with Remark 5.9 we have $\lambda(k), \mu(k) \in (0.15k, 0.16k)$, so we get

$$n = 2 + 4k + 2\ell < 4k + 2\mu(k) + 2 < 4.33k$$

and

$$\begin{aligned}
\text{SC}(G_{k,\ell}) &= \text{sep}(u) \\
&= 6k^2 + 2\ell^2 + 10k\ell + 10k + 6\ell + 2 \\
&> 6k^2 + 2\lambda(k)^2 + 10k\lambda(k) \\
&> 6k^2 + 0.045k^2 + 1.5k^2 \\
&> 0.4n^2
\end{aligned}$$

□

Bibliography

[1] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '04, pages 540–549, Washington, DC, USA, 2004. IEEE Computer Society.

[2] M. Aigner and M. Fromme. A game of cops and robbers. *Discrete Applied Mathematics*, 8(1):1 – 12, 1984.

[3] N. Alon, E. Demaine, M. Hajiaghayi, and T. Leighton. Basic network creation games. In *SPAA*, pages 106–113. ACM, 2010.

[4] Mikhail Atallah and Uzi Vishkin. Finding euler tours in parallel. *J. Comput. Syst. Sci.*, 29(3):330–337, December 1984.

[5] D. Bal, A. Bonato, W. B. Kinnersley, and P. Prałat. Lazy cops and robbers on hypercubes. *Combinatorics, Probability and Computing*, 24(6):829–837, 2015.

[6] A. Bonato, P. Golovach, G. Hahn, and J. Kratochvíl. The capture time of a graph. *Discrete Mathematics*, 309(18):5588 – 5595, 2009. Combinatorics 2006, A Meeting in Celebration of Pavol Hell's 60th Birthday (May 1-5, 2006).

[7] A. Bonato and R. Nowakowski. *The Game of Cops and Robbers on Graphs*, volume 61 of *Student Mathematical Library*. Sep. 2011.

[8] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[9] Camil Demetrescu, Bruno Escoffier, Gabriel Moruz, and Andrea Ribichini. Adapting parallel algorithms to the W-stream model, with applications to graph problems. *Theor. Comput. Sci.*, 411(44-46):3994–4004, October 2010.

[10] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. *ACM Trans. Algorithms*, 6(1):6:1–6:17, December 2009.

[11] Michael Elkin and Shay Solomon. Fast constructions of light-weight spanners for general graphs. 12, 07 2012.

[12] A. Fabrikant, A. Luthra, E. Maneva, C. Papadimitriou, and S. Shenker. On a network creation game. In *PODC*, pages 347–351. ACM, 2003.

[13] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, December 2005.

[14] F. V. Fomin, P. A. Golovach, J. Kratochvíl, N. Nisse, and K. Suchan. Pursuing a fast robber on a graph. *Theoretical Computer Science*, 411(7):1167 – 1181, 2010.

[15] Nathanaël François, Rahul Jain, and Frédéric Magniez. Unidirectional input/output streaming complexity of reversal and sorting. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, pages 654–668, 2014.

[16] Christian Glazik. *Positional and Detection Games.* PhD thesis, 2019. `https://macau.uni-kiel.de/receive/macau_mods_00000127`.

[17] Martin Grohe, Christoph Koch, and Nicole Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, pages 1076–1088, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[18] G. Hahn. Cops, robbers and graphs. *Tatra Mountains Mathematical Publications*, 36:163–176, Jan. 2007.

[19] Rebekah Herrman, Peter van Hintum, and Stephen G. Z. Smith. Capture times in the bridge-burning cops and robbers game. *CoRR*, abs/2011.00317, 2020.

[20] M.O. Jackson and A. Wolinsky. A strategic model of social and economic networks. *Journal of Economic Theory*, 71(1):44–74, 1996.

[21] W. B. Kinnersley and E. Peterson. Cops, robbers, and burning bridges. *arXiv e-prints*, page arXiv:1812.09955, Dec. 2018.

[22] L. Kliemann, E. Sheykhdarabadi, and A. Srivastav. Swap equilibria under link and vertex destruction. *Games*, 8(1):14, 2017.

[23] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

[24] A. Mehrabian. Cops and robber game with a fast robber on expander graphs and random graphs. *Annals of Combinatorics*, 16(4):829–846, Oct. 2012.

[25] R. Nowakowski and P. Winkler. Vertex-to-vertex pursuit in a graph. *Discrete Math.*, 43(2-3):235–239, Jan. 1983.

[26] D. Offner and K. Ojakian. Variations of cops and robber on the hypercube. *Australas. J Comb.*, 59:229–250, 2014.

[27] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[28] A. Quilliot. Jeux et pointes fixes sur les graphes. *Thèse de 3ème cycle, Université de Paris VI*, pages 131–145, 1978.

[29] Jan Matthias Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2003. AAI0805714.

[30] Xiaoming Sun and David P. Woodruff. Tight Bounds for Graph Problems in Insertion Streams. In Naveen Garg, Klaus Jansen, Anup Rao, and José D. P. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2015)*, volume 40 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 435–448, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Erklärung

Hiermit erkläre ich, dass

- die Abhandlung - abgesehen von der Beratung durch Herrn Srivastav - nach Inhalt und Form die eigene Arbeit ist,
- die Arbeit weder ganz noch zum Teil schon einer anderen Stelle im Rahmen eines Prüfungsverfahrens vorgelegen hat, veröffentlicht worden ist oder zur Veröffentlichung eingereicht wurde,
- die Arbeit unter Einhaltung der Regeln guter wissenschaftlicher Praxis der Deutschen Forschungsgesellschaft entstanden ist und
- mir kein akademischer Grad entzogen wurde.

Jan Schiemann
Kiel, 2021