Karlsruhe Institute of Technology

On Preprocessing the Arc-Flags Algorithm

Master Thesis of

Moritz Baum

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dr. Reinhard Bauer
	Dr. Ignaz Rutter

Time Period: April 2011 – September 2011

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 30th September 2011

Abstract

Precomputation of auxiliary data in an additional off-line step is a common approach towards improving the query performance of shortest-path queries in large-scale networks. Amongst others, one well-established technique based on this pattern is provided by the arc-flags algorithm. Similar to most approaches aiming at an improvement of query times for single-pair shortest path queries, it leaves a degree of freedom in the implementation of the preprocessing step. Namely, a partition of the underlying graph needs to be specified, which has a large impact on the performance of on-line queries. To evaluate the quality of a certain partition, one often considers the expected number of settled nodes in a random query. Filling the mentioned degree of freedom optimally with respect to this measure is $\mathcal{NP}$-hard on graphs in general. In this thesis, we provide a theoretical analysis concerning the computation of high-quality partitions for arc-flags. We examine the problem of finding optimal partitions on several restricted graph classes and provide first steps towards establishing a border of tractability. As a central result in this investigation, we prove that computing an optimal solution remains $\mathcal{NP}$-hard even on undirected trees. Apart from that, we develop integer linear programs for finding optimal partitions on arbitrary graphs. Different ILP approaches and possibilities for their tuning are discussed. Finally, we introduce two novel greedy approaches that seek to provide high-quality partitions of graphs. Besides an analysis of their complexities, we show that neither of them provides a constant approximation-ratio. A brief case study offers insights into the capability of these approaches on realistic graph instances.

Deutsche Zusammenfassung

Ein bewährter Ansatz zur Verbesserung der Laufzeit von Kürzeste-Wege-Anfragen auf großen Netzwerken ist die Nutzung einer zusätzlichen Vorberechnungsphase. Eine bekannte Umsetzung dieses Vorgangs besteht in der Einführung so genannter Arc-Flags. Wie bei vielen anderen, vergleichbaren Techniken gewährt das Verfahren Freiheitsgrade in der konkreten Umsetzung der Vorberechnung, deren Ausfüllung großen Einfluss auf die Performanz späterer Kürzeste-Wege-Anfragen haben kann. Im Falle von Arc-Flags besteht dieser Freiheitsgrad in der Wahl einer gültigen Partition des Eingabegraphen. Ein gängiges Maß zur Bewertung der Qualität einer solchen Partition ist die Betrachtung der erwarteten Suchraumgröße einer zufälligen Anfrage. Die Berechnung einer Partition, die den erwarteten Suchraum optimiert, ist im Allgemeinen $\mathcal{NP}$-schwer. Ausgehend von diesem Resultat liegt der Schwerpunkt der vorliegenden Arbeit auf einer ausführlichen theoretischen Untersuchung des Problems, zu einem gegebenen Graphen eine im Sinne des erwarteten Suchraums späterer Anfragen qualitativ gute Partition zu finden. Zunächst wird dazu der Einfluss verschiedener eingeschränkter Graphklassen auf die Schwierigkeit der Berechnung einer optimalen Lösung untersucht. Insbesondere wird dabei gezeigt, dass das Problem selbst auf Bäumen $\mathcal{NP}$-schwer bleibt. Zudem werden ganzzahlige lineare Programme zur Berechnung optimaler Partitionen entwickelt und diskutiert. Anschließend werden zwei Polynomialzeit-Algorithem basierend auf dem Greedy-Ansatz vorgestellt. Neben deren Analyse wird bewiesen, dass im Allgemeinen keiner der Ansätze die optimale Lösung mit einem konstanten Faktor approximiert. In einer Fallstudie wird das Potential der beiden Algorithmen auf kleinen, realistischen Eingabeinstanzen beleuchtet.

Contents

1. Introduction

In recent years, route planning has become a widely known application of algorithm engineering. Algorithms for route planning are an integral part of everyday life ever since navigation systems and on-line route planners have become popular. In addition to that, the algorithmic field of routing in street networks is of rising importance in economic areas, such as public transportation and logistics. During the last decade, improvements of algorithms that compute distances and shortest paths on large-scale networks have been studied extensively. As a result, research in this field lead to vast speed-ups of shortest-path queries compared to known approaches, in particular Dijkstra's algorithm. In contrast to common fast heuristics, these speed-up techniques also guarantee to maintain correctness of any computed shortest path.

The typical data structure for the representation of road networks in computation are weighted graphs. Although Dijkstra's algorithm is of polynomial-time complexity on arbitrary graphs, its performance on large realistic graphs is not acceptable for practical applications on current hardware. Speed-up techniques that yield improved query times are based on the assumption that for typical large-scale networks, the network itself mostly remains unchanged and rarely needs to be updated. This enables the opportunity to split the work into two parts. First, in the off-line phase a precomputation step is executed on the input graph to gain additional information about the underlying network. The retrieved data is then used during the on-line phase to improve the performance of shortest-path queries.

There are several approaches that implement the basic idea described above. One of these techniques is the arc-flags algorithm. As a simple metaphor, one may think of arc-flags as additionally provided sign posts in a graph. Just as sign posts would guide a human when reaching an intersection, arc-flags enable Dijkstra's algorithm to exclude single edges from being part of a shortest path. To this end, the corresponding graph is partitioned into a fixed number of regions, each of which is represented by a binary flag. Every edge of the graph is then added a vector of flags that provides all necessary information about whether or not the corresponding edge possibly leads to the target region. If this is not the case, the edge does not have to be traversed by the query-algorithm.

Although the outstanding performance of shortest-path queries enriched by arc-flags has been substantiated in many experimental studies, little is known about the theoretical backgrounds concerning the arc-flags algorithm. In this thesis, we focus on the particular aspect of preprocessing the arc-flags algorithm from a theoretical point of view. It is easy to see that the choice of the partition of a graph has a large impact on query times obtained in the on-line phase. Techniques used in practice to find partitions for given input graphs are of purely heuristic nature. By contrast, we examine the problem of finding partitions of graphs for which one can provide some sort of guarantee on its applicability for the arc-flags algorithm.

1.1 Related Work

We provide a short review of publications related to the domain of this thesis. In the not too distant past, many approaches have been introduced that attack the problem of efficiently handling queries on road networks. For a survey of recent techniques including arc-flags we refer to Delling et al. [DSSW09]. The methods presented there realize different trade-offs that take into account the following aspects.

- Duration and space consumption of the precomputation.
- Query times induced by the preprocessing.
- Amount of space necessary to store the additional information.

In general, *goal-directed* and *hierarchical* approaches can be distinguished. Goal-directed techniques aim at guiding the query into the right direction, for example by pruning edges for which one can assure that they are not part of the sought shortest path. In hierarchical approaches one tries to exploit the diverse relevance of road segments in a street network. For instance, following a highway rather than a rural road would appear to be a good choice on a long-distance query.

Arc-Flags

The idea of *arc-flags* was first introduced by Lauther [Lau97, Lau04]. Substantial experiments concerning shortest-path queries based on arc-flags are given by Köhler et al. [KMS05]. Möhring et al. discuss different ways to partition a given graph in the preprocessing step of the arc-flags algorithm, including two-level partitions in which one recursively divides the cells of a graph [MSS$^+$05, MSS$^+$06, Sch06]. Experimental results are presented to evaluate the diverse proposed approaches. Hilger et al. present ways to efficiently obtain correct arc-flags for a given partition [HKMS06, HKMS09]. In particular, so-called centralized shortest path trees are proposed as an efficient way to compute flags by handling several nodes in one step. Furthermore, experiments are conducted regarding heuristically obtained partitions. It turns out that *multi-way arc separators* [KK98] appear to be most promising for generating high-quality partitions. Such tools are provided, for example, by MeTiS [Kar07] and PARTY [MS04]. Another recently published approach that allows fast computation of valid arc-flags provides a technique to efficiently compute shortest paths between all pairs of nodes of a large-scale graph [DGNW11]. Finally, further experimental evaluation of arc-flags is conducted by Lauther [Lau09].

Combinations With Other Techniques and Dynamic Graphs

There are several attempts to combine the plain arc-flags algorithm with other known speed-up techniques. Three such combinations are presented by Bauer et al. [BDS$^+$10]. In particular, arc-flags are joined with *contraction hierarchies* [GSSD08]. This approach is based on contracting nodes of a given graph in a specified order while inserting shortcut edges to maintain correct distances. For yet another technique, arc-flags are combined with a speed-up technique based on *reach* [Gut04]. The latter involves computing a centrality measure for each node that yields information about its relevance for long-distance queries. Finally, an approach to use arc-flags within *transit-node routing* is presented [BFM$^+$07, BFSS07]. Transit-node routing involves the identification of nodes that are likely to be settled during long-distance queries. Distances to these so-called transit nodes are then computed in advance. Consequently, the query itself can be reduced to simple table look-ups, which allows for very low query times. Yet another technique, called *SHARC*, was first introduced by Bauer and Delling [BD09]. Again, arc-flags are combined with contraction-based routing. The preprocessing phase consists of several steps that are repeated iteratively. In the first step, nodes of the graph are contracted and corresponding shortcuts are added. A following step involves the computation of arc-flags for the modified graph. Brunel et al. propose a space-efficient variant of SHARC [BDGW10].

Routing in time-dependent networks is another related domain. In this scenario, edge weights are functions rather than constant numbers. An edge weight represents the duration it takes to pass that edge at different points in time. Methods to adapt arc-flags to time-dependent graphs are introduced by Delling [Del09]. He also provides applications of SHARC for such graphs [Del08]. Delling et al. propose further adaptations of the arc-flags approach to time-dependent networks [DPW09]. Experiments considering time-dependent scenarios are conducted by Delling and Wagner [DW09]. It turns out that in time-dependent graphs, the performance of SHARC is among the best of all speed-up techniques originally designed for the static case, as pointed out by Bauer et al. [BDW11]. Berretini et al. provide another adaptation of arc-flags to a dynamic scenario, which allows for modifications of edge weights during the on-line phase [BDD09].

Theoretical Results on Speed-Up Techniques

One important aspect of route-planning algorithms is the theoretical examination of their performance. Most speed-up techniques succeeding in practice are based on intuitions about the structure of realistic networks rather than theoretical analysis. A first approach to achieve theoretical results in order to explain the great performance of route-planning algorithms on large street networks is given by Abraham et al. [AFGW10]. In their work, the notion of *highway dimension* is introduced to measure the applicability of a graph for hierarchical speed-up techniques. Interestingly enough, a technique that outperforms most known approaches on street networks was inferred starting from these observations [ADGW11].

Another study investigates the preprocessing steps of common speed-up techniques from a theoretical point of view. Bauer et al. [BCK$^+$10, Bau10] prove $\mathcal{NP}$-hardness of optimally filling the degrees of freedom left in the preprocessing steps of many techniques. Furthermore, there are detailed theoretical studies concerning the speed-up techniques contraction hierarchies [Col09] and *ALT* [Fuc10], the latter of which is a goal-directed approach based on the A^*-algorithm [GH05, GW05].

1.2 Contributions and Outline

The precomputation phase of the arc-flags algorithm consists of two steps. At first, a partition of the graph has to be determined. Although besides validity there are no necessary conditions that must be met, the performance of the query is influenced by the suitability of this partition. Afterwards, correct arc-flags are computed based on the retrieved partition. In this thesis, we present a theoretical study of the first step, in order to gain further insights into its difficulties and to provide new approaches.

On the basis of known general hardness results, we examine several restricted classes of graphs [BCK$^+$10, Bau10]. Note that the graph obtained in the reduction used for the proof by Bauer et al. has the following undesirable properties, which are not shared by most realistic instances.

- The reduced graph contains a large cycle of nodes that is an inherent part of the proof.
- Edge weights occurring in the graph substantially differ.
- The graph is not strongly connected, that is, there are certain unreachable nodes.
- The number of edges in the graph is quadratic in the number of nodes, rendering the graph rather dense.

Clearly, this result leaves much room for a study of restricted classes of graphs that exclude some or all of the these properties. First, we prove that an optimal partition can be computed efficiently on directed and undirected paths. Afterwards, as a central result we establish the hardness of this problem on undirected trees with uniform edge weights. Hence, by providing this result we are able to eliminate all undesirable properties of the known reduction. Moreover, we conjecture that $\mathcal{NP}$-hardness holds for rooted directed trees and directed acyclic graphs as well. In addition to that, we show first steps to extend the proof of hardness to trees with a maximum degree of 3. This suggests that in general, optimal partitions are efficiently computable only for graph classes that in some sense carry a symmetric structure. As for cycles, we propose an algorithm that, given a reasonable assumption about the structure of an optimal partition, computes an optimal solution in polynomial time.

Integer linear programs for certain variants of the problem of finding an optimal partition are introduced. This renders computation of optimal partitions feasible for small input instances. Furthermore, we show how to significantly improve their performance by reducing the number of constraints and the solution space. A dual ILP is inferred that may serve as a starting point for further studies.

Additionally, we present two novel greedy algorithms that compute partitions suitable for arc-flags. These algorithms substantially differ from known techniques to generate partitions and hence provide completely new ideas to attack this problem. However, we also prove that a constant bound on their approximation ratio exists for neither of them. Their performances are compared on real-world instances in a brief case study.

Finally, by relaxing the original idea of arc-flags we deliver basic ideas for encoding the shortest paths between all pairs of nodes of a given graph. In addition to that, we infer ways to efficiently perform this task on restricted classes of graphs after a short precomputation step, which to the best of our knowledge has not been examined before.

Below, the outline of the remainder of this work is summarized. We give a brief description of the subject of each chapter.

Chapter 2. We provide foundations and terminology that is used throughout this thesis. In particular, we introduce notation from graph theory and present Dijkstra's algorithm as well as the concept of linear programming. Furthermore, we give a definition of search-space size and prove basic lemmas that later on shall turn out to be useful.

Chapter 3. In this chapter, we introduce the functionality of arc-flags and specify the work of the preprocessing algorithm. We formally describe the problem of finding optimal partitions for the arc-flags algorithm and study concepts that are used in practice to heuristically cope with this problem.

Chapter 4. Knowing that the problem of finding an optimal partition on arbitrary graphs is $\mathcal{NP}$-hard, we examine certain restricted graph classes. In particular, undirected paths and trees are studied. It turns out that an optimal partition can be computed efficiently for paths, whereas this problem remains hard even for trees. Based on these results, we present first steps towards the design of a polynomial-time algorithm for cycles and the establishment of a border of tractability.

Chapter 5. In Chapter 5 we design linear programs that represent optimal partitions for arc-flags. We first develop an integer linear program for a relaxed version of this problem and subsequently extend this program to cope with the more challenging task of the more complex original variant of the problem. Furthermore, we derive a dual ILP that may be useful for establishing approximation guarantees for future algorithms.

Chapter 6. We present two greedy approaches to receive partitions of high quality for the arc-flags algorithm. We analyze their complexities and prove that there is no approximation ratio guaranteed by any of these approaches. Finally, both algorithms are compared and analyzed on small, realistic instances in a case study.

Chapter 7. Starting from an alternative definition of arc-flags, we develop a way to encode shortest paths between all pairs of nodes of a given graph. Inspired by this approach, we investigate methods to efficiently provide a shortest path between any pair of nodes in some restricted classes of graphs using linear space overhead.

Chapter 8. We close this thesis with final remarks. A summary of the achieved results is given as well as an outlook of future work based on our gained insights and posed questions.

2. Preliminaries

This chapter provides the foundations of this thesis. We introduce basic terminology that is used throughout the following chapters. The definitions presented here may thus serve as a repetition of necessary backgrounds and to circumvent possible ambiguities regarding certain notations. Furthermore, there are slight differences in part of the terminology established here in comparison to its usage in common literature. Apart from the preliminaries introduced in this chapter, we assume the reader to be familiar with basic foundations from complexity theory, such as $\mathcal{O}$-notation and the concept of $\mathcal{NP}$-hardness. For more backgrounds in this area, we refer to standard works on these topics [GJ79, CLRS01].

In all what follows, we shall denote mathematical operations and sets as given in the subsequent listing.

$\mathbb{N}$	$=$	$\{0, 1, 2, \dots\}$	Set of natural numbers including zero
$\mathbb{N}^+$	$=$	$\mathbb{N} \setminus \{0\}$	Set of strictly positive integers
$\mathbb{R}$			Set of real numbers
$\mathbb{R}^+$	$=$	$\{x \in \mathbb{R} \mid x > 0\}$	Set of strictly positive real numbers
$\mathbb{R}^{\geq 0}$	$=$	$\mathbb{R}^+ \cup \{0\}$	Set of non-negative real numbers
$\log$			Logarithm with respect to base two

The remainder of this chapter is arranged as follows. First, basic definitions from graph theory are given in Section 2.1. Note that some of the terminology presented here slightly differs from similar definitions in common literature. Problems in the topic of finding shortest paths and Dijkstra's algorithm are presented in Section 2.2. We also formally introduce the notion of search-space size and summarize basic lemmas that we shall need in subsequent chapters of this thesis. In Section 2.3, Integer Linear Programming as a well-established technique for representing and solving problem instances is introduced.

2.1 Graph Theory

Directed and Undirected Graphs. A *(weighted, directed) graph* is a triple $G = (V, E, \omega)$ of two finite sets V and $E \subseteq V \times V$ and a mapping $\omega \colon E \to \mathbb{R}^+$. Elements of V are called *nodes* or *vertices*, elements of E are called *edges* or *arcs*. Given an edge $(u, v) \in E$, we call v the *head* and u the *tail* of (u, v). Furthermore, we say that u and v are *incident* to (u, v). Nodes u and v connected via an edge (u, v) are also called *neighbors*. The function ω assigns a positive real number $\omega(e)$ to each edge $e \in E$. The value $\omega(e)$ is called the *weight* of an edge e. Here, we assume that all edge weights are strictly positive. In what follows, the weight $\omega((u, v))$ of an edge (u, v) is abbreviated as $\omega(u, v)$. We denote the cardinalities of V and E by $|V| = n$ and $|E| = m$, respectively. If the weight function ω of a graph is not the matter of concern, we omit it from the notation and write $G = (V, E)$ instead.

An *undirected graph* is defined as a directed graph $G = (V, E, \omega)$ that satisfies the property that for all $(u, v) \in E$ there exists an $(v, u) \in E$ such that $\omega(u, v) = \omega(v, u)$. Note that this convention differs from the notation of undirected graphs that is commonly used in graph theory. Contrary to the general representation of an undirected edge by a two-element set $\{u, v\}$, we shall consider edges of an undirected graph to be composed of two opposing directed edges (u, v) and (v, u). This differentiation is necessary for an accurate analysis of the arc-flags algorithm presented in Chapter 3, for its behavior may depend on the direction in which an edge is passed.

Degrees. Let $G = (V, E, \omega)$ be a graph. The *in-degree* of a node $v \in V$ is defined as $\text{in}(v) = |\{(u, v) \in E\}|$. Accordingly, we say that the *out-degree* of v is $\text{out}(v) = |\{(v, u) \colon E\}|$. If G is undirected, the in-degree of each node $v \in V$ equals its out-degree and we simply speak of the *degree* of v.

Subgraphs. Let $G = (V, E, \omega)$ and $G' = (V', E', \omega')$ be graphs. Then, G' is a subgraph of G if and only if $V' \subseteq V$, $E' \subseteq E$ and for all $e' \in E'$ it is $\omega'(e') = \omega(e')$.

Backward Graphs. The backward graph $\overline{G}$ with respect to a given graph $G = (V, E, \omega)$ is defined as $\overline{G} = (V, \overline{E}, \overline{\omega})$ with the reverse edges $\overline{E} = \{(v, u) \mid (u, v) \in E\}$ and the weight function $\overline{\omega} \colon \overline{E} \to \mathbb{R}^+$, where $\overline{\omega}(v, u) = \omega(u, v)$ for all $(v, u) \in \overline{E}$.

Paths, Cycles, Distances and Shortest Paths. Given a graph $G = (V, E, \omega)$, a *path* $P = \langle v_1, \dots, v_k \rangle$ *of size* k is defined as a finite sequence of nodes in V such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. By $|P| \leq k$ we denote the number of distinct nodes that are included in the path. A path P is called simple if no node occurs repeatedly in P, i.e., $|P| = k$. We say that an s-t-path is *unique* if it is simple and there exists only one simple s-t-path in G. We write $(v_i, v_{i+1}) \in P$ for any edge (v_i, v_{i+1}) corresponding to a consecutive pair of nodes in P. The *weight* or *length* of a path P is defined to be $\omega(P) = \sum_{i=0}^{k-1} \omega(v_i, v_{i+1})$. If for a given path $Z = \langle v_1, \dots, v_k \rangle$ the condition $v_1 = v_k$ holds, we say that Z is a *cycle*.

The *distance* mapping $d_G(s, t) \colon V \times V \to \mathbb{R}^{\geq 0}$ for pairs of nodes s and t of G is defined as

$$d_G(s, t) = \begin{cases} 0 & \text{if } s = t, \\ \infty & \text{if there is no path } \langle s, v_2, \dots, v_{k-1}, t \rangle \text{ in } G, \\ \min_{P = \langle s, v_2, \dots, v_{k-1}, t \rangle} \omega(P) & \text{otherwise.} \end{cases}$$

Thus, the distance between two distinct nodes s and t is the minimum length of a path P such that $P = \langle s, v_2, \dots, v_{k-1}, t \rangle$, provided that any path from s to t exists. Otherwise, the distance between s and t is infinite. Note that for directed graphs, in general $d_G(s, t) \neq d_G(t, s)$ holds. A path P is called a *shortest path* from s to t if $\omega(P) = d_G(s, t)$. If $d_G(s, t) < \infty$, we say that t is *reachable* from s. As long as the associated graph is clear from the context, we omit the index G and write $d(s, t)$ instead.

Connectivity. We say that a graph $G = (V, E, \omega)$ is *strongly connected* if for all $u, v \in V$ it is $d(u, v) < \infty$. In other words, a graph is strongly connected if any target node is reachable from any source node. A graph G is *connected* if the graph $G' = (V, E \cup \overline{E})$ is strongly connected, i.e., if for every pair of nodes there exists a path connecting them if one ignores edge directions.

Partitions. A *partition* of a set V into a set $\mathcal{C} \subseteq 2^V$ of cells denotes the division of V into disjoint subsets $C \in \mathcal{C}$ such that $\bigcup_{C\in\mathcal{C}} = V$. We shall refer to the sets C as *cells* of G if $G = (V, E, \omega)$ is a graph and $\mathcal{C}$ is a partition of its nodes. We say that a cell C is *(strongly) connected* if the graph $(C, E \cap (C \times C))$ is (strongly) connected.

Trees, Shortest-Path Trees, Directed Acyclic Graphs. We say that a graph $T = (V, E, \omega)$ is an *(undirected) tree* if T is an undirected graph that is strongly connected and the number of edges is $m = 2(n - 1)$. Note that due to our definition of undirected graphs stated above, we are forced to allow for two single directed edges per undirected connection. Leaving aside these formal details, the definition of undirected trees given here is similar to the usual one.

A graph $G = (V, E, \omega)$ is called a *directed acyclic graph* if G contains no cycles. We define as *directed tree* rooted at node s any graph $T = (V, E, \omega)$ such that $s \in V$, $d(s, v) < \infty$ for all $v \in V$ and the number of edges is $m = n - 1$. If T is a subgraph of an arbitrary graph $G = (V', E', \omega')$ and we have $V = \{v \in V' \colon d_G(s, v) < \infty\}$ plus $d_T(s, v) = d_G(s, v)$ for all $v \in V$, T is called a *shortest-path tree* of s in G.

Given a directed or undirected tree $T = (V, E, \omega)$, a node $v \in V$ is called a *leaf* of V if $\text{in}(v) = 1$. The *height* of a node $u \in V$ is $h(u) = \max_{P=\langle u,\ldots,v\rangle,\text{in}(v)=1} |P| - 1$. The height of a rooted tree is the height of its root node.

2.2 The Shortest-Path Problem

Now, we consider the task of finding shortest paths between given nodes of a directed, weighted graph. There are three particular problems that typically occur in this matter, which are specified below.

1. The *single-pair shortest path (SPSP) problem* is to find for a given graph $G = (V, E, \omega)$ and nodes s and t the distance $d(s, t)$.
2. The *single-source shortest path (SSSP) problem* is to find for a given graph $G = (V, E, \omega)$ and a node s the distances $d(s, v)$ for all $v \in V$.
3. The *all-pairs shortest path (APSP) problem* is to find for a given graph $G = (V, E, \omega)$ the distances $d(u, v)$ for all pairs of nodes $u, v \in V$.

In the context of this work, we are interested in efficiently solving the SPSP problem. In what follows we present Dijkstra's Algorithm as the basis for later extensions. In addition to that, we introduce the search-space size as a simple theoretical measure to estimate running times of shortest-path queries. Finally, basic lemmas for later studies are established, concerning the search-space size induced by Dijkstra's algorithm on strongly connected graphs and the sum of the sizes of subpaths of a given path.

2.2.1 Dijkstra's Algorithm

Probably the most well-known algorithm for computing shortest paths is Dijkstra's algorithm [Dij59]. It solves the single-source shortest path problem on directed graphs with non-negative edge weights. Algorithm 2.1 depicts pseudo code for a variant of Dijkstra's algorithm that not only computes the demanded distances, but additionally outputs a shortest-path tree that covers shortest paths from the source node to any reachable node in the graph.

The algorithm's interface works as follows. It requires as input a weighted directed graph G and a source node s. The output consists of two arrays $\mathsf{d}(\cdot)$ and $\mathsf{pred}(\cdot)$, both of which are of size n. After the execution of Algorithm 2.1, the value of $\mathsf{d}(v)$ for each node v equals

Algorithm 2.1: Dijkstra

```
Input: Graph $G = (V, E, \omega)$, source node $s$
Data: Priority queue Q
Output: Distances $\mathsf{d}(v)$ for all $v \in V$, shortest-path tree of $s$ given by $\mathsf{pred}(\cdot)$

// Initialization
forall $v \in V$ do
    $\mathsf{d}(v) \leftarrow \infty$
    $\mathsf{pred}(v) \leftarrow$ null
Q.insert$(s, 0)$
$\mathsf{d}(s) \leftarrow 0$

// Main loop
while Q is not empty do
    $u \leftarrow$ Q.deleteMin()
    forall $(u, v) \in E$ do
        if $\mathsf{d}(u) + \omega(u, v) < \mathsf{d}(v)$ then
            $\mathsf{d}(v) \leftarrow \mathsf{d}(u) + \omega(u, v)$
            $\mathsf{pred}(v) \leftarrow u$
            if Q.contains$(v)$ then
                Q.decreaseKey$(v, \mathsf{d}(v))$
            else
                Q.insert$(v, \mathsf{d}(v))$
```

the distance from s to v. For any reachable target node t, the algorithm also computes and outputs an actual shortest path that starts at node s. More precisely, the output given by vector $\mathsf{pred}(\cdot)$ encodes a shortest-path tree with root s. After execution of the algorithm, $\mathsf{pred}(v)$ contains for each node $v \neq s$ its parent node in a shortest-path tree if v is reachable from s, otherwise it contains a distinct value `null`. Thus, the graph $T = (V', E')$, with $V' = \{v \in V \mid d(s, v) < \infty\}$ being the set of all nodes reachable from s and $E' = \{(\mathsf{pred}(v), v) \mid s \neq v \in V'\}$ being the edges determined by the vector $\mathsf{pred}(\cdot)$ returned by Dijkstra's algorithm, is the desired shortest-path.

Description of the Algorithm

The algorithm makes use of a priority queue that manages pairs (o, i) of objects and integer keys. Although we do not specify the implementation of the priority queue, it is expected to provide the following operations.

- insert(v, i) inserts the object v with key i into the priority queue.
- deleteMin() returns the object with the minimum key value and removes it from the queue.
- contains(v) returns a boolean value, which is `true` if and only if object v is held in the queue.
- decreaseKey(v, i) sets the key of object v to i. The input parameters are valid only if v is contained in the queue and i is not greater than its current key.

In our context, the priority queue maintains nodes with tentative distance labels as keys. With a suitable priority queue implementation in place, Dijkstra's algorithm works as described below. At first, the necessary data structures are initialized. All tentative distance labels $\mathsf{d}(\cdot)$ except for $\mathsf{d}(s)$ are initialized at infinity. The source distance label is set to $\mathsf{d}(s) = 0$. Then, the source node s is inserted into the priority queue. After this initialization phase, the main phase of the algorithm starts. As long as there are unprocessed nodes, i.e., the priority queue contains at least one more node, a node u with minimum tentative distance label $\mathsf{d}(u)$ is extracted from the queue. We say that a node is *settled* once that it is removed from the queue. In the next step, any edge (u, v) outgoing from u is checked for whether it provides a path to v that is shorter than the shortest distance $\mathsf{d}(v)$ found so far. If this is the case, the according edge is *relaxed*, i.e., the tentative distance of v is set to $\mathsf{d}(u) + \omega(u, v)$ and u is made the parent node of v in the shortest-path tree. The queue entry of v is decreased if it exists, or v is enqueued with key $d(v)$ if it is not contained in the queue yet. This main loop is repeated until no nodes are left in the queue.

Asymptotic Complexity of Dijkstra's Algorithm

The correctness of Dijkstra's algorithm is based on the provable fact that $d(s, v) = \mathsf{d}(v)$ as soon as the node v is settled. The running time of Dijkstra's algorithm depends on the implementation of the priority queue. A naive implementation with DELETE operations that have a time complexity linear in the number of elements in the queue yields a running time of $\mathcal{O}(n^2)$. Using more sophisticated Fibonacci heaps [TF87], the time complexity of Dijkstra's algorithm can be reduced to $\mathcal{O}(m+n\cdot\log n)$. For a detailed analysis of Dijkstra's algorithm and its complexity see the chapter on the single-source shortest path problem in the book by Cormen et al. [CLRS01].

Breaking Ties in the Priority Queue

A matter that has not been dealt with so far is the behavior of the priority queue when the minimum key of the object to be deleted cannot be uniquely determined, i.e., there exist two or more nodes in the queue with the minimum key. Dijkstra's algorithm remains correct regardless of the order in which nodes are returned in this case. However, in all what follows we shall assume that there is a fixed total order $\prec$ on any regarded set of nodes V, such that a node u in the priority queue is extracted on calling DELETEMIN() if and only if for all v in Q it is either $\mathsf{d}(u) < \mathsf{d}(v)$ or $\mathsf{d}(u) = \mathsf{d}(v)$ and additionally $u \prec v$. Furthermore, we denote by $u \preceq v$ that $u \prec v$ or $u = v$. We simply assume that the order $\prec$ is induced by increasing node indices encoded in a given input graph $G = (V, E, \omega)$.

The Stopping Criterion for Dijkstra's Algorithm

As mentioned above, Dijkstra's algorithm solves the SSSP problem on a given graph. Though, in the subsequent chapters of this work we are interested in a query algorithm that solves the SPSP problem. Clearly, the algorithm described above can be used to handle such queries for a given source node s and a target node t. However, since we are only interested in one particular shortest path, Dijkstra's algorithm may possibly perform more work than necessary. A simple modification of Algorithm 2.1 in order to gain a first speed-up works as follows. The so-called *stopping criterion* permits to abort the computation as soon as the target node t is extracted from the priority queue. Since we know that $\mathsf{d}(t) = d(s, t)$ holds once that t is settled, this can safely be done without violating correctness. The stopping criterion can be applied independent of the arc-flags approach presented below and may thus be used in any SPSP query based on Dijkstra's algorithm. Although the stopping criterion does not improve the asymptotic running time of the algorithm, it is commonly used in practice, because it can greatly reduce query times.

2.2.2 Search-Space Size

Despite the fact that Dijsktra's algorithm has a polynomial asymptotic running time, its performance on large-scale networks is still impractical on modern hardware. Although there are no established superior bounds for most speed-up techniques used in practice, they all outperform from Dijkstra's algorithm in terms of query times. Furthermore, for practical concerns, constant factors can be of vast significance. Hence, it appears useful to introduce more accurate measures for the time complexity of speed-up techniques. Considering Dijkstra's algorithm, one finds that its running time is dominated by the operations performed in its main loop. Regardless of the concrete hardware of a machine that would execute the algorithm, query times should be proportional to the number of settled nodes and inspected edges. Let the search space of a distinct query be the set $U \subseteq V$ of nodes that are settled by Dijkstra's algorithm. Then the size $|U|$ of this set can be interpreted as an abstract measure of the hypothetical running time of this query. A formal definition to embody this idea is given below.

Definition 2.1 (Dijkstra Search-Space Size)**.** *Let $G = (V, E, \omega)$ be a graph and $s, t \in V$ two nodes in G. The search-space size $\mathrm{S}_{\mathrm{Dij}}(G, s, t)$ is defined as the number of nodes settled by Dijkstra's algorithm in a query with source node s and target node t when the stopping criterion is applied. Accordingly, $\mathrm{S}^{+}_{\mathrm{Dij}}(G, s, t)$ denotes the number of settled nodes if the stopping criterion is not in use. Furthermore, we define* $\mathrm{S}_{\mathrm{Dij}}(G) = \sum_{s,t \in V} \mathrm{S}_{\mathrm{Dij}}(G, s, t)$ *and* $\mathrm{S}^{+}_{\mathrm{Dij}}(G) = \sum_{s,t \in V} \mathrm{S}^{+}_{\mathrm{Dij}}(G, s, t)$*.*

If the underlying graph G is clear from the context, we shall omit it from the notation and instead simply write $\mathrm{S}_{\mathrm{Dij}}(s, t)$ and $\mathrm{S}^{+}_{\mathrm{Dij}}(s, t)$, respectively. As a simple example, imagine the search-space size of a query between an arbitrary pair of nodes s and t of a given graph $G = (V, E, \omega)$. Assuming that the stopping criterion is in place and recalling that there is an order $\prec$ that determines the sequence in which nodes with the same distance label are extracted, we get a search-space size

$$\mathrm{S}_{\mathrm{Dij}}(G, s, t) = \left|\{v \in V \mid d(s, v) < d(s, t) \vee (d(s, v) = d(s, t) \wedge v \preceq t)\}\right| .$$

One could argue that the number of edges that must be inspected in a query may in fact dominate the running time. In the following, we provide reasons to prefer the investigation of the search-space size induced by nodes rather than edges, as defined above.

- Established hardness results on preprocessing of speed-up techniques use the search-space size stated in Definition 2.1. Hence, we can use these known facts as the starting point of our work, whereas an alternative notion would force us to start from scratch.
- Many graphs that occur in practice are sparse, i.e., for the number of edges of such graphs it is $m \in \mathcal{O}(n)$. Especially, this assumption can safely be made in road networks, which are the main application of the speed-up techniques considered here and the arc-flags algorithm in particular.
- The search space is a rather complex structure that involves the detailed behavior of Dijkstra's algorithm. Taking account of edges rather than nodes would possibly even increase complexity and render a formal analysis less clear and understandable.

Altogether, it appears reasonable to consider the search-space size as introduced in Definition 2.1. Guided by this argumentation, we stick to this node-based definition in all what follows.

2.2.3 Basic Lemmas

The following two lemmas provide basic statements that are going to be useful throughout the subsequent chapters. At first, we investigate the search-space size of Dijkstra's Algorithm on an arbitrary strongly connected graph. Somewhat surprising at first glance, the search-space size only depends on the size of the considered graph in this case.

Lemma 2.2. *Let $G = (V, E, \omega)$ be a strongly connected graph. Then the search-space sizes of Dijkstra's Algorithm with respect to G are $\mathrm{S}^+_{\mathrm{Dij}}(G) = n^3$ and $\mathrm{S}_{\mathrm{Dij}}(G) = n^2(n + 1)/2$.*

Proof. To begin with, we examine the case where the stopping criterion of Dijkstra's algorithm is omitted. Consider an arbitrary s-t-query on G. Because the graph is strongly connected, every node of V eventually gets extracted from the priority queue. Thus, the number of settled nodes of an arbitrary s-t-query is $\mathrm{S}^+_{\mathrm{Dij}}(s, t) = n$. Since there are exactly n^2 distinct pairs of nodes s and t in V, we immediately obtain

$$\mathrm{S}^+_{\mathrm{Dij}}(G) = \sum_{s,t \in V} \mathrm{S}^+_{\mathrm{Dij}}(G, s, t) = n^3. \tag{2.1}$$

Now consider a fixed node s in V and assume that the stopping criterion is applied. Then the query is aborted once that the target node t is reached in an s-t-query. Since the order in which the nodes get extracted from the queue in any query from s is independent of t, we can assign a rank i ranging from 1 to n to each node of the graph that represents its position in this order. Then there are n possible target nodes for a query starting at s and each of them has a distinct rank in $\{1 \dots n\}$. This yields the search-space size regarding all queries with the source node s given below.

$$\sum_{t \in V} \mathrm{S}_{\mathrm{Dij}}(G, s, t) = \sum_{i=1}^{n} i = \frac{n(n + 1)}{2}$$

Since this holds for any of the n possible source nodes, we get the overall search-space size of G for the algorithm Dijkstra.

$$\mathrm{S}_{\mathrm{Dij}}(G) = \sum_{s,t \in V} \mathrm{S}_{\mathrm{Dij}}(G, s, t) = n \cdot \frac{n(n + 1)}{2} \tag{2.2}$$

Given Equations 2.1 and 2.2, the proof is complete. □

The next lemma summarizes the sizes of all distinct paths contained in a graph that consists of a single chain of nodes. This lemma is used in Chapters 4 and 6.

Lemma 2.3. *Let $P = (V, E, \omega)$ be a graph where the sets of nodes and edges are $V = \{v_1, \dots, v_n\}$ and $E = \{(v_i, v_{i+1}), (v_{i+1}, v_i) \mid 0 < i < n\}$, respectively. For arbitrary $u, v \in V$, let $P_{u,v}$ denote the unique path from u to v. The sum of all distinct path sizes in P is $\sum_{u,v \in P} |P_{u,v}| = n^3/3 + n^2 - n/3$.*

Proof. For a given graph P that consists of a single undirected path, we enumerate the sizes of all distinct paths as follows. There are exactly two paths of size n, namely the paths $\langle v_1, \dots, v_n \rangle$ and $\langle v_n, \dots, v_1 \rangle$. Analogously, we have exactly the four paths $\langle v_1, \dots, v_{n-1} \rangle$, $\langle v_2, \dots, v_n \rangle$, $\langle v_{n-1}, \dots, v_1 \rangle$, $\langle v_n, \dots, v_2 \rangle$ of size $(n - 1)$ and so forth. Finally, we have to account for $2(n - 1)$ paths of size 2. In addition to that, there are n paths of size 1, i.e, paths where source and target node are identical. Note that the latter case forms an exception, because we do not have to distinguish two directions. The sum of all these path sizes is summarized below.

$$\begin{aligned}\sum_{u,v\in C}|P_{u,v}| &= n + 2\sum_{i=1}^{n-1}(n-i)(i+1)\\ &= n + 2\left(n\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i^2 - \sum_{i=1}^{n-1} i\right)\\ &= n + n^3 - n^2 + 2n^2 - 2n - \frac{1}{3}\left(2n^3 - n^2 - 2n^2 + n\right) - n^2 + n\\ &= \frac{1}{3}n^3 + n^2 - \frac{1}{3}n\end{aligned}$$

This is what we claimed in the lemma. □

2.3 Linear Programming

A well-established approach for solving hard problems is the translation of given problems into *linear programs*, hereafter abbreviated as LP. The two main reasons for widespread use of this approach are that on one hand many problems can be naturally expressed as linear programs and on the other hand much effort has been spent to develop and engineer solvers for linear programs. In many cases, these solvers provide practically tractable running times for exact solutions or high quality approximations for input instances of moderate size. This section provides a brief problem definition of linear programming. For a more detailed analysis of linear programming and its complexity see, for example, the books of Nemhauser and Wolsey [NW88] or Schrijver [Sch86].

Specification of a Linear Program

Linear programming is an optimization problem in which one has to minimize an objective function under certain constraints. The goal is described by a linear objective function of the following form, where $c = (c_1, \ldots, c_n)^\top \in \mathbb{R}^n$ is a vector of constant coefficients and $x = (x_1, \ldots, x_n)^\top$ is a vector of variables.

$$\text{minimize} \;\; c^\top \cdot x$$

Minimizing the given objective has to be done subject to a number of constraints expressed by inequalities of the form given below, with a matrix $A \in \mathbb{R}^{m\times n}$ of constants and a vector $b = (b_1, \ldots, b_n)^\top \in \mathbb{R}^n$ of constants.

$$\begin{aligned} A \cdot x &\geq b\\ x &\geq 0\end{aligned}$$

Constraints of the form $a \cdot x \leq b$ can be easily constructed if necessary by multiplying a whole inequality by -1. Furthermore, a constraint $a \cdot x = b$ can be expressed by two constraints $a \cdot x \geq b$ and $a \cdot x \leq b$. The range of values that feasible solutions of the vector x may take determines the hardness of a linear program. If x is allowed to be any vector in $\mathbb{R}^n$, the linear program is solvable in polynomial time. However, if some or all of the coefficients of x are required to be integers, optimizing the LP is $\mathcal{NP}$-hard in general. If all variables in x are integers, the LP is called an integer linear program (ILP). If some coefficients of x are restricted to be in $\mathbb{Z}$, while others are allowed to be in $\mathbb{R}$, the LP is called a mixed integer linear program (MILP).

Duality of Linear Programs

Given a linear program in the form shown above with the vector b of constants, its *dual linear program* is defined by the objective function given below, where $y = (y_1, \dots, y_m)^\top \in \mathbb{R}^m$ is a new vector of variables.

$$\text{maximize} \;\; b^\top \cdot y$$

The objective function is maximized subject to the following constraints, where A is the constant matrix and c the coefficient vector from the primal LP defined above.

$$\begin{aligned} A^\top \cdot y &\le c \\ y &\ge 0 \end{aligned}$$

The dual program of an arbitrary LP has the property that all its feasible solutions are smaller or equal than all feasible solutions to the primal one. Furthermore, if there exists an optimal solution it is equal for both of them. These facts can be useful in the analysis of a problem and the design of approximation algorithms for hard problems [Vaz03].

3. Problem Statement

One successful method for improving the performance of shortest-path queries is the addition of a vector of so called arc flags to every edge of the graph. When searching for a certain target node, arc-flags provide auxiliary information about whether a given edge might be part of a shortest path to that target.

The layout of this chapter is organized as follows. The following Section 3.1 provides a formal introduction of the arc-flags algorithm. At first, a brief description of the general ideas and conditions of speed-up techniques in general is given. Thereafter, we formally introduce the arc-flag based approach as an enhancement of Dijkstra's algorithm. Since arc-flags constitute additional information that is not part of the original input graph, they must be computed in advance. A simple way to determine valid arc-flags in a preprocessing step is explained. We finally state in Section 3.2 the problem of filling one crucial degree of freedom during this preprocessing, in order to achieve best possible average search-space sizes during the query phase.

3.1 The Arc-Flags Algorithm

We consider the following realistic scenario. We are given a large, static input graph and assume that the SPSP problem has to be solved many times for yet unknown different pairs of source and destination nodes of this graph. In the following, we describe a general concept for attacking this problem.

Scenario

Although Dijkstra's algorithm can solve the SPSP problem and has sub-quadratic asymptotic time complexity on arbitrary graphs, its running time on large realistic graphs is prohibitive for practical applications. We thus split the routing algorithm into two parts. We allow for a possibly expensive precomputation phase that enables us to enrich the graph by some additional information before pairs of source and target nodes are known. In the subsequent query phase, this data can be used in any query to improve the performance of the underlying algorithm. Methods that implement this approach are called *speed-up techniques*. Note that this scenario in fact facilitates breaking the barrier of the asymptotic run-time of Dijkstra's algorithm considering single queries. For example, given a certain graph we could easily achieve query times in $\mathcal{O}(1)$ if we simply precomputed all distances $d(s, t)$ and one shortest path for each pair of nodes in advance. However, this would require an amount of storage that is at least quadratic in the number of nodes and thus impractical on large graphs containing several millions of nodes and edges. Instead, one wants to find a compromise that delivers a fair trade-off between the following generic parameters.

- The precomputation time must be practicable.
- The memory consumption necessary to store the auxiliary information should be linear in the number of nodes and edges of the graph.
- The achieved query times should be low.

One speed-up technique that yields such a trade-off is realized by the addition of so-called arc-flags to every edge of the graph. The functionality of this approach is presented in the following.

3.1.1 Arc-Flag Based Queries

Arc-Flags provide a speed-up technique based on Dijkstra's algorithm that is used for SPSP queries on undirected and directed graphs as specified in Chapter 2.1. It was first introduced by Lauther [Lau97, Lau04] and has ever since been studied and revised several times, as summarized in Chapter 1.1. The original approach designed for large, static graphs is described below.

Introduction of Binary Flag Vectors

The graph is initially partitioned into a fixed number of k cells. An example of a graph partition into four cells is shown in Figure 3.1. In an s-t-query, we shall refer to the cell that the target node belongs to as the *target cell*. The basic idea behind this approach is to provide information for every edge of the graph that enables the query algorithm to decide whether this edge may possibly be part of a shortest path to any node of the target cell. If this is not the case, the corresponding edge is ignored by the algorithm.

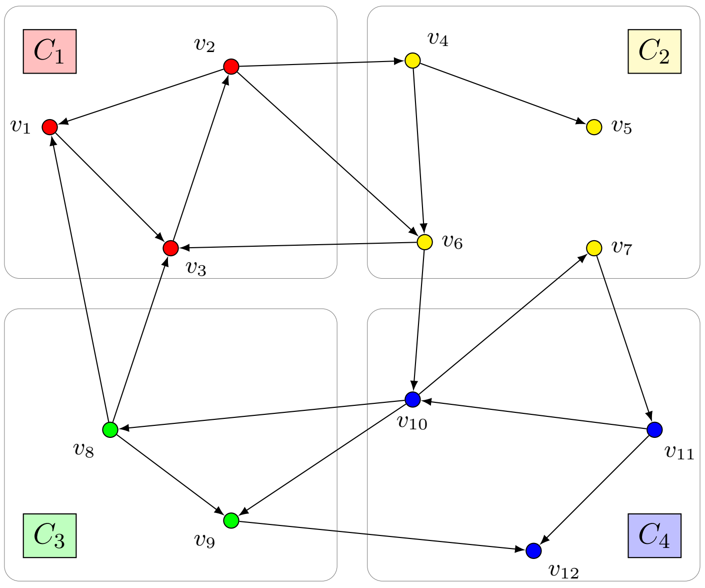

Figure 3.1: A simple example of a graph partition. The graph is divided into four cells $C_1 = \{v_1, v_2, v_3\}$, $C_2 = \{v_4.v_5, v_6, v_7\}$, $C_3 = \{v_8, v_9\}$ and $C_4 = \{v_{10}, v_{11}, v_{12}\}$.

Given a graph $G = (V, E, \omega)$ and a partition $\mathcal{C} = \{C_1, \ldots, C_k\}$ of G, we assign a distinct number in $\{1, \ldots, k\}$ to each cell and define a mapping $c \colon V \to \{1, \ldots, k\}$ such that $c(v)$

is the number of the cell that v belongs to. Storing this information clearly requires space linear in n. Furthermore, we enrich each edge e of the graph by a vector $\mathcal{F}_e(\cdot)$ of k binary flags construed as boolean values. Thus, the additional memory consumption necessary for these arc-flags amounts to $k \cdot m$ bits and therefore is linear in m. We interpret that $\mathcal{F}_{(u,v)}(c(t)) = 1$ means that the edge (u, v) might be important in any query with the target node t. Let $\mathcal{SP}_{s,t}$ denote the set of all shortest s-t-paths in a given graph. Every flag is then supposed to fulfill the following property to retain correctness of the query algorithm.

$$\forall s, t \in V : \mathcal{SP}_{s,t} \neq \emptyset \Rightarrow \exists P \in \mathcal{SP}_{s,t} : \forall (u, v) \in P : \mathcal{F}_{(u,v)}(c(t)) = 1 \tag{3.1}$$

By satisfying this expression we ensure that for any pair of source and target nodes, there exists at least one shortest path for which the flags corresponding to the target cell are set to 1 on all of its edges. We say that flags fulfilling this property are *correct*. Figure 3.2 shows an example of correct flags for the cell assignment of the graph depicted in Figure 3.1. There is a vector of four binary flags added to each edge of the graph with the values according to the cells C_1 to C_4 from left to right. A flag value of 1 for a cell C_i at an edge e is equivalent to the boolean value `true`, a value of 0 represents the value `false`.

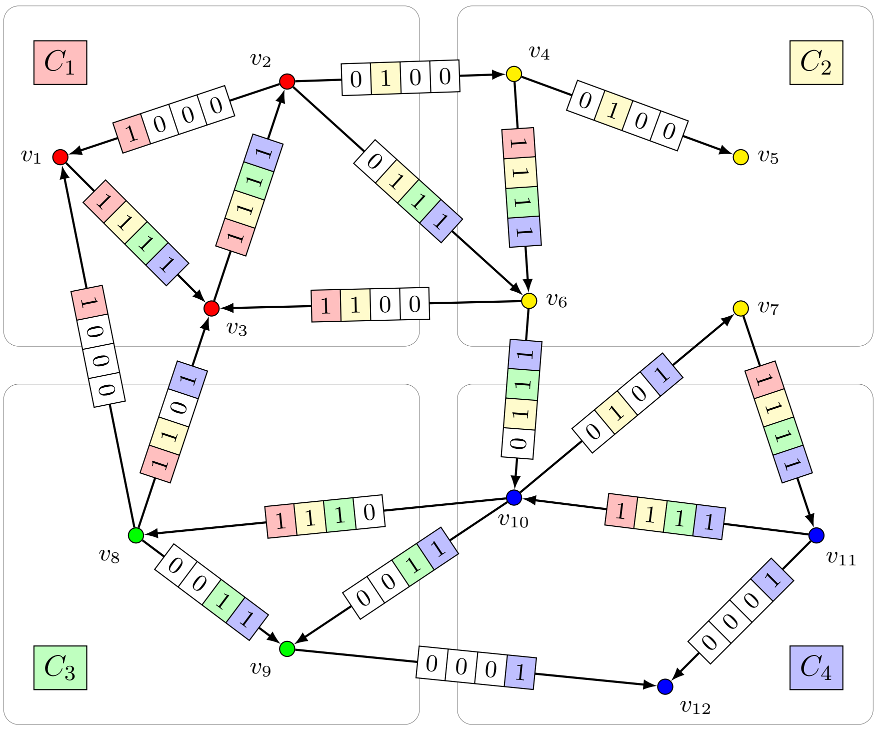

Figure 3.2: The example graph from Figure 3.1 with correct arc-flags, provided that edge weights are uniform.

The Query Algorithm

Provided that correct arc-flags are given for an input graph, the arc-flag based query algorithm is a simple modification of Dijkstra's algorithm. Algorithm 3.1 depicts pseudo code of a version of the arc-flags algorithm that additionally makes use of the stopping criterion introduced in Chapter 2.2 to further reduce the execution time. The algorithm works analogous to the plain version DIJKSTRA, with only one important modification. Before an edge is relaxed, we check if its flag corresponding to the target cell is set. If it is not, we may safely omit that edge and continue with the next step of the algorithm. Equation 3.1 ensures that a shortest path is still found. So, Algorithm 3.1 always returns the correct distance from s to t. Note that distance labels other than $\mathsf{d}(t)$ may hold incorrect distances after execution of the algorithm. This is due to the fact that certain edges are possibly skipped as well as the termination of the algorithm once that t is settled.

Algorithm 3.1: ArcFlags

```
Input: Graph $G = (V, E, \omega)$, cell assignment $c(\cdot)$, source node $s$, target node $t$
Data: Priority queue Q
Output: Distance $d(s,t)$ given by $\mathsf{d}(t)$, shortest path from $s$ to $t$ given by $\mathsf{pred}(\cdot)$

// Precondition:  Correct flags are known for all edges.
// Initialization
1  forall $v \in V$ do
2      $\mathsf{d}(v) \leftarrow \infty$
3      $\mathsf{pred}(v) \leftarrow$ null
4  Q.insert$(s, 0)$
5  $d(s) \leftarrow 0$
// Main loop
6  while Q is not empty do
7      $u \leftarrow$ Q.deleteMin()
8      if $u = t$ then
9          return
10     forall $(u, v) \in E$ do
11         if $\mathcal{F}_{(u,v)}(c(t)) = 1 \land \mathsf{d}(u) + \omega(u,v) < \mathsf{d}(v)$ then
12             $\mathsf{d}(v) \leftarrow \mathsf{d}(u) + \omega(u,v)$
13             $\mathsf{pred}(v) \leftarrow u$
14             if Q.contains$(v)$ then
15                 Q.decreaseKey$(v, \mathsf{d}(v))$
16             else
17                 Q.insert$(v, \mathsf{d}(v))$
```

To construct the shortest s-t-path, starting at t one simply has to follow all parents given by $\mathsf{pred}(\cdot)$ until s is reached. Note that Equation 3.1 leaves a degree of freedom in the concrete choice of flags, so that in case of multiple shortest paths, the path found by Algorithm 3.1 may differ from that returned by plain Dijkstra.

3.1.2 The Preprocessing Step

The preprocessing algorithm must ensure that, given a partition $\mathcal{C}$, Equation 3.1 is satisfied. The easiest way to do so would be to simply set all flags to `true`. Obviously, this would not yield any speed-up. Instead, one would like to set as many flags as possible to `false`, so that Algorithm 3.1 would skip many edges while Equation 3.1 is still preserved. As mentioned above, whenever there exists more than one shortest path between any pair of nodes, Equation 3.1 leaves a degree of freedom in the choice of the path for which flags should be set to maintain correctness. The problem of filling this degree of freedom optimally, i.e., assigning correct flags to edges for a given partition of the graph such that the expected search-space size of Algorithm 3.1 is minimized, is $\mathcal{NP}$-hard [BCK$^+$10, Bau10]. However, this problem is not a matter of concern in our context, so we are rather interested in a simple preprocessing algorithm that yields appropriate arc-flags.

There are several approaches for precomputing correct arc-flags. The basic idea is to compute all shortest paths to each distinct nodes of a cell in order to determine the flags

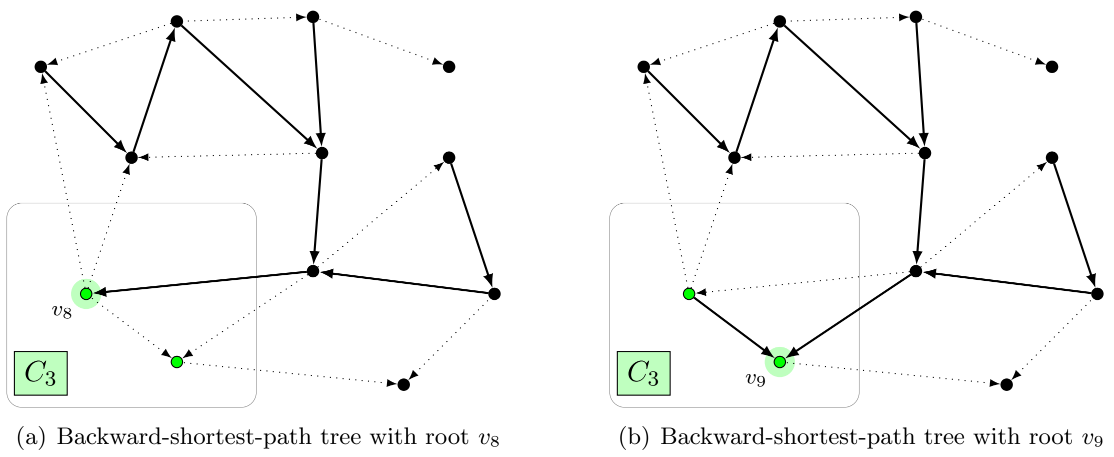
v_8
C_3
(a) Backward-shortest-path tree with root v_8
C_3
v_9
(b) Backward-shortest-path tree with root v_9

Figure 3.3: The backward-shortest-path trees that determine the flags for the cell C_3 of Figure 3.1 if all edge weights are assumed to be uniform. Edges drawn in bold are part of the corresponding tree.

that must be set to maintain correctness. For an overview of different approaches see a work of Hilger et al. [HKMS09]. Another recently published approach named *PHAST* yields a method for practically tractable computation of all-pairs shortest paths [DGNW11]. In general, preprocessing algorithms used in practice are sophisticated procedures which aim at a reduction of the preprocessing time.

Since we do not focus on preprocessing time in this thesis, we present a very simple procedure. Throughout this work, we shall assume that arc-flag vectors of all encountered graphs were computed this way. The basic idea of our preprocessing algorithm is as follows. Given an input graph $G = (V, E, \omega)$, we run Dijkstra's algorithm on the backward graph $\overline{G}$ once for each node $t \in V$. After each run, for every edge (v, u) of the backward-shortest-path tree computed by DIJKSTRA we set $\mathcal{F}_{(u,v)}(c(t)) = 1$ for the corresponding edge (u, v) of the original graph. This yields correct flags which guarantee to cover at least one shortest path for each pair of nodes. For an example see Figure 3.3. The drawings show the backward-shortest-path trees rooted at nodes v_8 and v_9, respectively. Any edge that is part of at least one of these trees has the flag for the cell C_3 set to `true` in the query algorithm, see also Figure 3.2.

3.2 The Problem ArcFlagsPartition

In the preprocessing procedure presented in Section 3.1.2, it was assumed that a certain partition is given. However, the choice of the partition has a major influence on the expected running times of the queries performed later. For example, assigning every node of the graph to the same cell yields a valid partition. Obviously, using this partition would create approximately no gain from using arc-flags at all, since only those edges that never belong to a shortest path would be skipped in the query phase. Assigning each node to its own cell would lead to a contrary result. In this case, the preprocessing algorithm explained above ensures that the only edges visited by the query algorithm would be those actually belonging to one shortest path from s to t. Using n cells this way can be viewed as encoding a solution of the APSP problem. However, the existence of n cells would imply a memory consumption of $m \cdot n$ bits, which is prohibitive for large graphs. We therefore assumed that the number of cells is limited by a fixed number k in Section 3.1.1.

3.2.1 Optimizing Partitions for Arc-Flags

The task is to find a good partition of a given graph when the number of cells is limited by a positive integer k. Finding a good partition means to find a cell assignment that yields low search-space sizes. Note that so far we have only defined the search-space size with respect to Dijkstra's algorithm. In the following, we formally specify the search-space size of ArcFlags. Along the lines of Definition 2.1, we introduce two different notions for both Algorithm 3.1 and a variant of it that does not use the stopping criterion.

Definition 3.1 (Arc-Flags Search-Space Size). *Let $G = (V, E, \omega)$ be a graph, $\mathcal{C}$ a partition of G and $s, t \in V$ two nodes in G. The search-space size $\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$ is defined as the number of nodes settled by the arc-flags algorithm in a query with source node s and target node t when the stopping criterion is applied. Accordingly, $\mathrm{S}^{+}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$ denotes the number of settled nodes if the stopping criterion is not in use. Furthermore, we define $\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}) = \sum_{s,t \in V} \mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$ and $\mathrm{S}^{+}_{\mathrm{AF}}(G, \mathcal{C}) = \sum_{s,t \in V} \mathrm{S}^{+}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$.*

Again, in the case of the underlying graph G or the partition $\mathcal{C}$ being clear from the context, we may omit both from the notation and instead write $\mathrm{S}_{\mathrm{AF}}(s, t)$ and $\mathrm{S}^{+}_{\mathrm{AF}}(s, t)$, respectively. Compared to the search-space size according to Dijkstra's algorithm, things become be little more complicated with regard to arc-flags. Since edges are skipped if their target cell flag is set to 0, we can interpret the query as a plain Dijkstra that runs on a modified, cell-dependent graph G_C. Given the input graph $G = (V, E, \omega)$, all original edges of E without the set target flag are removed in G_C. Hence, we may define the graph $G_{c(t)} = (V, E_{c(t)}, \omega)$ induced by the target cell $c(t)$ using the following set $E_{c(t)}$ of edges.

$$E_{c(t)} = \{e \in E \mid \mathcal{F}_e(c(t)) = 1)\}$$

Note that we abuse notation here, because $c(t)$ is a number, whereas C denotes a set of nodes. Formally, we define G_C as the graph $G_{c(t)}$ for an arbitrary node $t \in C$. This enables us to describe the total search-space size of the arc-flags algorithm when a certain cell assignment is given. The search-space size of an s-t-query then is

$$\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t) = \mathrm{S}_{\mathrm{Dij}}(G_{c(t)}, s, t).$$

Retrieving an Optimal Partition

In what follows, we concentrate on the problem of minimizing $\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$ by choosing a partition $\mathcal{C}$ for a given graph G. As we discussed before, the number of nodes the arc-flags algorithm extracts from the queue until the target is reached has a large influence on the running time of the query. Therefore, we interpret the search-space size as a simple measure of the algorithm's actual running time. Thus, when aiming for a partition that yields good performance, we are looking for a partition such that $\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$ is as small as possible for any pair s, t. Here, we shall focus on minimizing the average search-space size. Assuming that source and target nodes s and t are picked uniformly at random, the expected search-space size apparently is

$$E(\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t)) = \frac{\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C})}{n^2}.$$

Consequently, minimizing the expected search-space size of a random query is equal to minimizing $\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$. Similar observations hold for all different kinds of search-space sizes introduced in Definitions 2.1 and 3.1. For an example, recall the arc-flags depicted in Figure 3.2 based on a given underlying partition $\mathcal{C}$. The total search-space size when using this partition sums up to $\mathrm{S}_{\mathrm{AF}}(\mathcal{C}) = 532$ if ties are broken according to the node indices. When aiming at minimal search-space size, a better partition is given by

$\mathcal{C}_{\text{opt}} = \{\{v_1, v_3, v_6, v_8\}, \{v_2, v_4, v_5\}, \{v_7, v_{10}, v_{11}\}, \{v_9, v_{12}\}\}$. This yields a search-space size of $\mathrm{S}_{\mathrm{AF}}(\mathcal{C}_{\text{opt}}) = 469$. As long as the number of cells is restricted to 4, this is the best we can get. Below, we formalize the problem of minimizing the average search-space size.

Problem ArcFlagsPartition**.** *Given a graph $G = (V, E, \omega)$ and a positive integer k, find a partition $\mathcal{C}$ of G such that $|\mathcal{C}| \leq k$ and $\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C})$ is minimized.*

We call such a partition that minimizes the search-space size of Algorithm 3.1 optimal or *search-space optimal*. The remainder of this work is essentially devoted to the problem ArcFlagsPartition and its complexity.

3.2.2 Investigating Approaches From Practice

The problem ArcFlagsPartition is known to be $\mathcal{NP}$-hard [BCK+10, Bau10]. Preprocessing algorithms used in practice therefore solve this problem heuristically. Möhring et al. examine different strategies to obtain partitions that yield proper query times, some of which require or generate a two-dimensional layout of the graph [MSS+05]. Besides the demand of an efficient procedure to compute a useful partition, these heuristics follow some intuitive ideas of what such a partition should look like. These ideas are summarized below. For a more detailed description see Hilger et al. [HKMS09].

- All cells should contain approximately the same number of nodes.
- Cells should be connected subgraphs.
- Connections between cells are supposed to be sparse. This allows an efficient computation of correct arc-flags.
- The number of flags set to 1 should be as small as possible.

Although following these guidelines yields feasible cell assignments for common instances appearing in practice, none of these intuitions can guarantee to imply an optimal solution. In fact, the optimal solution may differ substantially in any of the criteria listed above. Figure 3.4 yields four examples of such graphs. Their optimal solutions each inevitably violate one of the properties listed above. Below, we examine these examples.

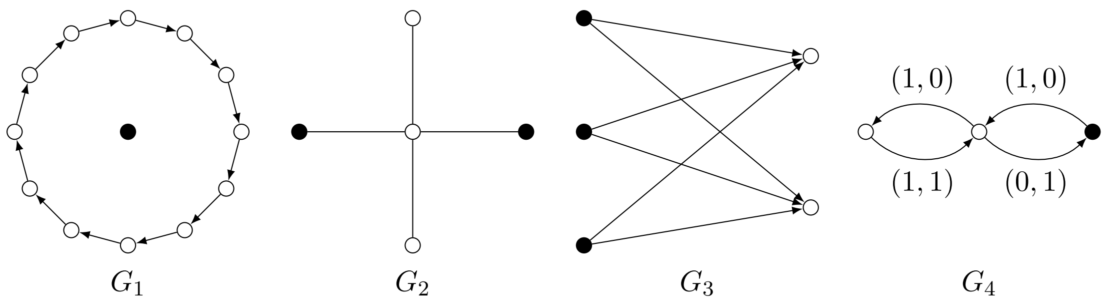

Figure 3.4: Four graphs with their search-space optimal partitions when the number of cells is restricted to 2.

First of all, we see a large difference in the cell sizes of the optimal partition of G_1. Note that even if the size of the cycle is increased arbitrarily, one of the cells in the optimal partition always consists of the central node only. This is due to the fact that the search-space size within the directed cycle cannot be reduced by the addition of arc-flags, see also Chapter 4. The graph G_2 shows an optimal partition that contains a cell that is not strongly connected. In graph G_3, the set of edges connecting nodes of different cells in the optimal partition even contains all edges of the graph. The optimal cells of G_4 induce a total number of five flags that are 1, whereas assigning all nodes of the graph to the same cell would lead to a worse search-space size with only four flags set to 1. As already mentioned for the graph G_1, one can construct similar examples in all cases including graphs of arbitrary size. Hence, optimizing any of the criteria listed above alone is not fruitful in general. Furthermore, without giving examples, we note that one can construct graphs in which an optimal partition must even violate several of the criteria listed above.

4. Search-Space Optimal Cells on Restricted Graph Classes

The problem of finding search-space optimal cells on arbitrary graphs is known to be $\mathcal{NP}$-hard [BCK$^+$10, Bau10]. A logical next step thus is to investigate the problem of computing optimal cells if we restrict the set of admissible input instances. In particular, after a few preliminary steps provided in the following Section 4.1, we show how to optimally assign cells on graphs that consist of a single path of arbitrary size in Section 4.2. Afterwards, we turn to the problem of finding optimal cells on undirected trees in Section 4.3. The main result is the $\mathcal{NP}$-hardness of the problem ArcFlagsPartition even if input graphs are restricted to undirected trees. The chapter is then completed by a short review of other restricted graph classes and concluding remarks in Section 4.4.

4.1 Fundamentals for this Chapter

Before turning to the problem of finding search-space optimal cells on graphs with specified properties, we establish some lemmas that turn out to be useful throughout this chapter. Afterwards, we introduce specific terminology that is going to be helpful in our investigations.

Basic Lemmas on Convex Functions

In the upcoming sections, the problem of distributing a given number of nodes to cells will occur repeatedly. We know from Lemma 2.2 that the search-space size is at least quadratic in the number of nodes of the graph. We are going to face many other situations in which the search-space sizes can be expressed by a polynomial function of the cell sizes. Therefore, we substantiate some claims that are going to be useful in these situations. First of all, recall that a convex function is defined as follows.

Definition 4.1. *A function $f\colon I \to \mathbb{R}$ defined on an interval $I \subseteq \mathbb{R}$ is convex, if for all $x_1, x_2 \in I$ and for $\lambda \in [0,1]$, it is $f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$.*

Moreover, we say that a given function $f\colon I \to \mathbb{R}$ is *increasing* on $I \subseteq \mathbb{R}$ if for all $x_1 \geq x_2 \in I$ it is $f(x_1) \geq f(x_2)$. Lemma 4.2 states a basic fact for convex functions that in turn is helpful for the proof of the subsequent Lemma 4.3.

Lemma 4.2. *Let $f\colon \mathbb{R}^{\geq 0} \to \mathbb{R}$ be a function that is convex on $\mathbb{R}^{\geq 0}$. Then the difference quotient $(f(x_0+h) - f(x_0))/h$ of f is non-decreasing in x_0 for any fixed h.*

Proof. If a function $f\colon \mathbb{R}^{\geq 0} \to \mathbb{R}$ is convex, the following inequation holds for all $x_1 < x_2 < x_3 \in \mathbb{R}^{\geq 0}$.

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1} \leq \frac{f(x_3) - f(x_2)}{x_3 - x_2}$$

A formal proof of this fact can be found, for example, in a book by Browder [Bro96]. Setting $x_2 = x_1 + h$ and $x_3 = x_2 + h$, we immediately obtain

$$\frac{f(x_1 + h) - f(x_1)}{h} \leq \frac{f(x_2 + h) - f(x_2)}{h}$$

for any $x_1 < x_2 \in \mathbb{R}^{\geq 0}$. This proves our claim. □

The following lemma provides a crucial statement about the sum of several functional values of a convex function. Later on, this function shall represent search-space sizes induced by a set of cells. Furthermore, Lemma 4.3 immediately implies the statement of Corollary 4.4, which follows next.

Lemma 4.3. *Let $f\colon \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ be a cost function that is convex and increasing on $\mathbb{R}^{\geq 0}$. Let x and n be two fixed positive integers. Furthermore, let $X = \{x_1, \ldots, x_n\}$ be a set of positive integers subject to $\sum_{i=1}^{n} x_i = x$. Then the total cost $\Gamma = \sum_{i=1}^{n} f(x_i)$ is non-decreasing if the values x_i are modified subject to one of the following rules while maintaining the constraints $\sum_{i=1}^{n} x_i = x$ and $x_i \geq 0$ for all $x_i \in X$.*

1. *Two arbitrary values x_i and x_j are swapped.*
2. *Given two integers x_i, x_j with $x_i \geq x_j$ and a number $d \in \mathbb{N}^+$, the value x_i is increased by d while x_j is decreased by d.*

Proof. Clearly, swapping two elements of X has no influence on the cost Γ. Thus, we can concentrate on the latter case.

For the second case, assume we are given two values x_i, x_j such that $x_i \geq x_j$ holds. Obviously, the resulting cost after increasing x_i and decreasing x_j by the same value $d \in \mathbb{N}^+$ is equal to

$$\Gamma' = \Gamma + f(x_i + d) - f(x_i) + f(x_j - d) - f(x_j).$$

Since f is increasing in $\mathbb{R}^{\geq 0}$, we know that we have $f(x_i + d) - f(x_i) \geq 0$ and similarly $f(x_j - d) - f(x_j) \leq 0$. Consequently, all we need to show is that

$$|f(x_i + d) - f(x_i)| \geq |f(x_j) - f(x_j - d)|$$

holds for any $x_i \geq x_j$. This, however, is clear because we demanded that f is convex and thus the difference quotient

$$g(x) = \frac{f(x + h) - f(x)}{h}$$

is non-decreasing in x for fixed h. We set $h = d$ and $x = x_i$ or $x = x_j - d$, respectively. Since $x_i \geq x_j$ implies $g(x_i) \geq g(x_j - d)$ and due to the constraints of the lemma $g(x_j - d) \geq 0$ holds, we obtain the following desired result.

$$|f(x_i + d) - f(x_i)| = d \cdot g(x_i) \geq d \cdot g(x_j - d) = |f(x_j) - f(x_j - d)|$$

This completes the proof. □

Assume we are given a set $X = \{x_1, \dots, x_n\}$ of natural numbers with $x_i \in \{\lfloor x/n \rfloor, \lceil x/n \rceil\}$ for all $x_i \in X$ such that their sum equals x and a cost function Γ as in Lemma 4.3. Using steps 1 and 2 from the lemma, we can create any set of values x_i that fulfills the constraint $\sum_{i=1}^{n} x_i = x$. In each of these steps, the overall cost is non-decreasing. Hence, we minimize a given convex, increasing cost function if all values x_i are as close to $\lfloor x/n \rfloor$ as possible. Corollary 4.4 follows directly from this observation. It turns out to be useful in Sections 4.2 and 4.3.

Corollary 4.4. *Let $f\colon \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ be an increasing, convex function and x and n two positive integers. For any set X containing n positive integers x_i subject to the constraint $\sum_{i=1}^{n} x_i = x$, the cost $\sum_{i=1}^{n} f(x_i)$ is minimized if $x_i = \lceil x/n \rceil$ for $i \leq x \bmod n$ and $x_i = \lfloor x/n \rfloor$ for $i > x \bmod n$.*

Additional Terminology

In the proofs of the sections below we are going to make use of an alternative notion of the search-space size. The following definition therefore introduces the penalty of a query.

Definition 4.5 (Penalty)**.** *Let $G = (V, E, \omega)$ be a graph such that for all $s, t \in V$ there is exactly one simple path from s to t and let $P_{s,t}$ denote this path. Furthermore, let $\mathcal{C}$ be a partition of G. We say that the penalty of a tuple $(s,t) \in V \times V$ is $\text{pen}_{\mathcal{C}}(s,t) = \mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t) - |(P_{s,t})|$.*

The path $P_{s,t}$ in Definition 4.5 denotes the unique shortest path from s to t. Consider an s-t-query of the algorithm ArcFlags. We know that the algorithm must at least settle all nodes on the path from s to t, and consequently the value of $|P_{s,t}|$ yields a tight lower bound on $\mathrm{S}_{\mathrm{AF}}(\mathcal{C}, s, t)$. Thus, we know that $\text{pen}_{\mathcal{C}}(s,t) \geq 0$ holds and the value $\text{pen}_{\mathcal{C}}(s,t)$ yields an upper bound on the number of settled nodes that could possibly be saved in the corresponding query by changing its underlying partition. Furthermore, since the sum $\sum_{s,t \in P} |P_{s,t}|$ of all path sizes of a graph is a constant, minimizing $\mathrm{S}_{\mathrm{AF}}(\mathcal{C})$ is equal to minimizing $\sum_{s,t \in P} \text{pen}_{\mathcal{C}}(s,t)$. Also note that we are only dealing with undirected paths and trees in the subsequent Sections 4.2 and 4.3 and hence penalties are well-defined for all pairs of nodes we encounter.

In many situations to follow, we distinguish *inter-cell queries* and *intra-cell queries*. Given a graph $G = (V, E)$ and a partition $\mathcal{C}$ of V into cells, we call an s-t-query an inter-cell query if s and t belong to different cells, i.e. $c(s) \neq c(t)$. We say that the *inter-cell search-space size* of the graph G is given by

$$\sum_{\substack{s,t \in V \\ c(s) \neq c(t)}} \mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t).$$

If conversely $c(s) = c(t)$, we say that the s-t-query is an intra-cell query. Note that in an intra-cell query, nodes from other cells may still get settled. Moreover, we define the *intra-cell search-space size* to be

$$\sum_{\substack{s,t \in V \\ c(s) = c(t)}} \mathrm{S}_{\mathrm{AF}}(G, \mathcal{C}, s, t).$$

4.2 Search-Space Optimal Partitions for Paths

We examine the effect of partitions for the arc-flags algorithm on graphs that are of the form $P = (V, E, \omega)$ with $V = \{v_1, \dots, v_n\}$ and $E = \{(v_i, v_{i+1}), (v_{i+1}, v_i) \mid 1 \leq i < n\}$ for an arbitrary $n \in \mathbb{N}^+$. We shall refer to such a graph as a path. Note that we can interpret the index i of a node v_i as its position on the path. The order induced by these indices is not to be confused with the total order $\prec$ which we introduced in Chapter 2 to break ties in the priority queue of Dijkstra's algorithm.

4.2.1 Connectivity of Cells and Independence of Weights

The goal of this section is to determine search-space optimal partitions for paths. We first present two preparatory lemmas that will make the proof of the final theorem in Section 4.2.2 easier. Assume we are given a graph $G = (V, E)$ and a partition $\mathcal{C} = \{C_1, \dots, C_k\}$ of G. We show that if P is a path, there is an optimal partition of P such that all cells of the partition are strongly connected. Recall that a cell C_i is called strongly connected if the graph $(C_i, E \cap (C_i \times C_i))$ is strongly connected.

Lemma 4.6. *Let $P = (V, E, \omega)$ be a path and $\mathcal{C} = \{C_1, \dots, C_k\}$ a partition of P. Let there be at least one cell $C_i \in \mathcal{C}$ that is not strongly connected. Then there exists a partition $\mathcal{C}' = \{C'_1, \dots, C'_k\}$ of P such that all cells in $\mathcal{C}'$ are strongly connected and $\mathrm{S_{AF}}(G, \mathcal{C}') \leq \mathrm{S_{AF}}(G, \mathcal{C})$.*

Proof. Assume we are given a path $P = (V, E, \omega)$ and a partition $\mathcal{C} = \{C_1, \dots, C_k\}$ of P such that at least one of these cells is not strongly connected. Based on $\mathcal{C}$, we construct a partition $\mathcal{C}'$ that only contains strongly connected cells. We then show that the total search-space size induced by $\mathcal{C}'$ is not greater than the total search-space size for $\mathcal{C}$.

Given partition $\mathcal{C} = \{C_1, \dots, C_k\}$, the construction of $\mathcal{C}' = \{C'_1, \dots, C'_k\}$ works as follows. Starting at node v_1, we assign subsequent nodes of the path to ascending cell indices while retaining the cell sizes of $\mathcal{C}$. More formally, we set $C'_1 = \{v_1, \dots, v_{|C_1|}\}$, $C'_2 = \{v_{|C_1|+1}, \dots, v_{|C_1|+|C_2|}\}$ and so forth. For an illustration of the cell construction see Figure 4.1. The upper path shows the original cell assignment with three different cells. The lower path depicts the constructed partition. Clearly, this method yields a valid partition of the underlying graph P.

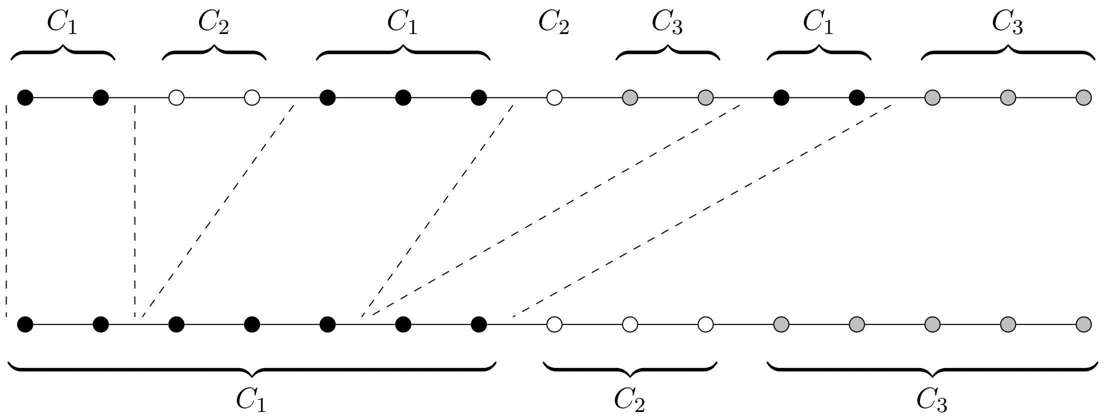

Figure 4.1: Functionality of the procedure for creating strongly connected cells from an arbitrary partition while preserving the cell sizes.

We now show that $\sum_{s,t \in P} \mathrm{pen}_{\mathcal{C}'}(s,t) \leq \sum_{s,t \in P} \mathrm{pen}_{\mathcal{C}}(s,t)$ holds, which implies the lemma. To this end, we distinguish the penalties of inter-cell queries and intra-cell queries on P.

1. *Inter-cell queries.* Let s and t be nodes of different cells, i.e. $c(s) \neq c(t)$. Since all cells in $\mathcal{C}'$ are strongly connected subgraphs, only edges that actually point at the target cell have the corresponding flag set. This situation is depicted in Figure 4.2. Therefore, it is easy to see that the query algorithm starts at s and then settles only nodes in $P_{s,t}$ until the target cell is reached. This is due to the fact that edges pointing into the opposite direction do not have the target flag set to 1. Moreover, once the target cell is entered we know that due to the stopping criterion, the query is aborted as soon as the target node t is reached. Hence, all nodes that are settled during the

query belong to the unique shortest s-t-path $P_{s,t}$. We therefore get $\text{pen}_{\mathcal{C}'}(s,t) = 0$ for all inter-cell queries, which clearly is the best we can get.

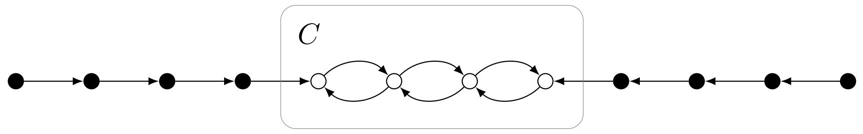

Figure 4.2: Schematic flags of a cell that contains the white nodes. Only edges that have the flag for cell C set to 1 are depicted.

2. *Intra-cell queries.* First of all, observe that the partitions minimizing the intra-cell penalty and the intra-cell search-space size, respectively, may in fact differ. This is due to the fact that when restricting ourselves to intra-cell queries, the sum of all path sizes corresponding to these queries is no longer a constant but depends on the chosen partition. Consequently, since we considered penalties in the first case, we must stick to penalties for intra-cell queries as well.

   We know that the size of each cell in $\mathcal{C}$ is preserved in the corresponding cell in $\mathcal{C}'$, i.e., $|C_i| = |C'_i|$ for all $i \in \{1, \ldots, k\}$. Hence, the total number of intra-cell queries is identical for both partitions. We consider an arbitrary isolated pair of corresponding cells C_i and C'_i of both partitions and show that the sum of all intra-cell penalties cannot increase for the cell C'_i.

   Assume we are given certain cells C_i and C'_i for an $i \in \{1, \ldots, k\}$. Furthermore, let $C_i = \{v_{r_1}, \ldots, v_{r_c}\}$ and $C'_i = \{v_{s_1}, \ldots, v_{s_c}\}$. We compare the sum of all penalties of intra-cell queries starting at a node v_{r_j} to those starting at v_{s_j}. By showing that $\sum_{t \in C'_i} \text{pen}_{\mathcal{C}'}(v_{s_j}, t) \leq \sum_{t \in C_i} \text{pen}_{\mathcal{C}}(v_{r_j}, t)$ for all $j \in \{1, \ldots, c\}$ and for all $i \in \{1, \ldots, k\}$ of the graph, we prove that the overall intra-cell penalty induced by $\mathcal{C}'$ is not greater than the one induced by $\mathcal{C}$.

   Since each cell in $\mathcal{C}'$ is strongly connected and the corresponding flags are set as outlined in Figure 4.2, we can interpret any intra-cell query as a plain DIJKSTRA that uses the stopping criterion and runs on a cell-induced subgraph $(C'_i, E \cap C'_i \times C'_i)$ of P. For any source node v_{s_j} in C'_i, we then get the following search-space size.

$$\sum_{t \in C'_i} \mathrm{S}_{\mathrm{AF}}(P, \mathcal{C}'_i, v_{s_j}, t) = \sum_{z=1}^{|C'_i|} z = \frac{|C'_i| \cdot (|C'_i| + 1)}{2}$$

   To obtain the corresponding penalties, we consider the sizes of all paths $P_{v_{s_j},t}$ from v_{s_j} to any $t \in C'_i$. Since the cell is strongly connected, we know that the cell-induced subgraph contains exactly $j-1$ nodes with an index lower than s_j and $c-j$ nodes with an index greater than s_j. If we divide all of the intra-cell targets into the sets $\{v_{s_x} \mid x < j\}$ and $\{v_{s_x} \mid x \geq j\}$, we immediately obtain the following term that summarizes the path sizes of paths from v_{s_j} to all nodes within the same cell.

$$\sum_{t \in C'} |P_{v_{s_j},t}| = \sum_{z=2}^{j} z + \sum_{z=1}^{|C'_i|-j+1} z$$

If we sum up the penalties of all intra-cell queries corresponding a fixed source node v_{s_j} of the cell C_i', we obtain the following total penalty.

$$\sum_{t \in C_i'} pen_{\mathcal{C}'}(v_{s_j}, t) = \frac{|C_i'| \cdot (|C_i'| + 1)}{2} - \left(\sum_{z=2}^{j} z + \sum_{z=1}^{|C_i'|-j+1} z \right)$$

Conversely, when analyzing all queries that start at the corresponding source v_{s_j} of the original partition $\mathcal{C}$, we have to take into account that nodes in other cells may get settled. Let X_z denote the set of nodes outside cell C_i that get additionally settled in the unique query that settles exactly z nodes in C_i. Note that this specification is unambiguous because the number of nodes settled in C_i ranges from 1 to $|C_i|$ and each quantity is taken precisely once. Hence, we obtain $|C_i|$ sets $X_1, \dots, X_c$. Let further X_z^L for $1 \le z \le j-1$ denote the set of nodes not in cell C_i that are part of the path from v_{r_j} to $v_{r_{j-z}}$. Similarly, we define X_z^R, $1 \le z \le c-j$, for nodes outside C_i on the path from v_{r_j} to $v_{r_{j+z}}$. In total, we get $c-1$ sets X_z^R and X_z^L.

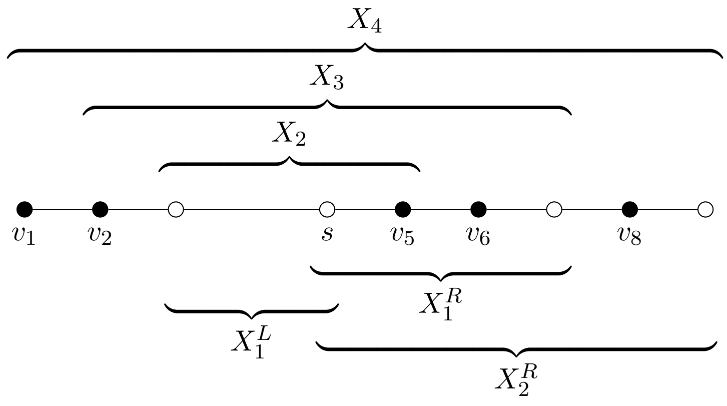

Figure 4.3: Additional penalty for intra-cell queries from a single node when cells are not strongly connected. Edge weights are assumed to be proportional to their length in the illustration and the leftmost node is settled first in case of a tie.

Figure 4.3 shows an exemplary situation. Consider queries from s to all nodes of the same cell drawn white. The sets X_z, X_z^L and X_z^R contain all black nodes below or above their according curly braces. Hence, we have $X_1 = \emptyset, X_2 = \{v_5\}, X_3 = \{v_2, v_5, v_6\}, X_4 = \{v_1, v_2, v_5, v_6, v_8\}, X_1^L = \emptyset, X_1^R = \{v_5, v_6\}$ and $X_2^R = \{v_5, v_6, v_8\}$. It is $X_1^L \subseteq X_2$, $X_1^R \subseteq X_3$ and $X_2^R \subseteq X_4$. Generally speaking, the according penalty turns out to be as follows if we further set $X_0^R = \emptyset$.

$$\begin{aligned}
\sum_{t \in C_i} \mathrm{pen}_{\mathcal{C}}(v_{r_j}, t) &= \sum_{z=1}^{|C_i|} (z + |X_z|) - \left(\sum_{z=2}^{j} \left(z + \left|X_{z-1}^L\right|\right) + \sum_{z=1}^{|C_i|-j+1} \left(z + \left|X_{z-1}^R\right|\right) \right) \\
&= \sum_{z=1}^{|C_i'|} z - \left(\sum_{z=2}^{j} z + \sum_{z=1}^{|C_i'|-j+1} z \right) + \sum_{z=1}^{|C_i|} |X_z| - \left(\sum_{z=1}^{j-1} \left|X_z^L\right| + \sum_{z=1}^{|C_i|-j} \left|X_z^R\right| \right) \\
&= \sum_{t \in C_i'} \mathrm{pen}_{\mathcal{C}'}(v_{s_j}, t) + \underbrace{\sum_{z=1}^{|C_i|} |X_z| - \left(\sum_{z=1}^{j-1} \left|X_z^L\right| + \sum_{z=1}^{|C_i|-j} \left|X_z^R\right| \right)}_{\alpha} \qquad (4.1)
\end{aligned}$$

First of all, note that X_1 is always empty. The remaining sets X_z can be divided into two subsets such that each subset corresponds to queries from v_{r_j} to nodes with lower or greater index, respectively. Since a query must settle any node that belongs to the actual path from source to target, it is clear that for each set X_z one can find a distinct set $X_{z'}^L$ or $X_{z'}^R$ that is a subset of X_z, such that both sets correspond to the query with the same target. See also Figure 4.3 for an illustration. Hence the term α in Equation 4.1 must be non-negative.

We have shown that for both the total inter-cell penalty and the total intra-cell penalty the partition $\mathcal{C}'$ yields a solution that is at least as good as $\mathcal{C}$. Hence, the proof is completed. □

Given Lemma 4.6, we know that when looking for optimal partitions of paths, it is sufficient to restrict the search to partitions that exclusively contain strongly connected cells. Lemma 4.7 shows that provided with this information, we can also ignore edge weights for our purposes.

Lemma 4.7. *Let $P = (V, E, \omega)$ be a path and $\mathcal{C} = \{C_1, \ldots, C_k\}$ a partition of P such that all cells C_i of $\mathcal{C}$ are strongly connected. Then, the total search-space size $\mathrm{S}_{\mathrm{AF}}(\mathcal{C}, G)$ is independent of the weight function ω.*

Proof. Again, we distinguish intra-cell queries and inter-cell queries. Since all cells of $\mathcal{C}$ are strongly connected, we know from the proof of Lemma 4.6 that the inter-cell search spaces cover exactly the shortest paths from the source node to the target node. Intra-cell search spaces, however, were shown above to be equal to the search spaces of a plain Dijkstra that runs on the corresponding cell. From Theorem 2.2 we know that the search-space size of Dijkstra's algorithm only depends on the cell size. So neither the inter-cell search-space size nor the intra-cell search-space size depend on the edge weights. □

4.2.2 Optimal Cell Assignments on Paths

With Lemmas 4.6 and 4.7 in place, we can finally establish optimal partitions on paths. We show that cells of uniform size such that each cell is strongly connected yield an optimal solution.

Theorem 4.8. *Let $P = (V, E, \omega)$ be a path as specified above and k a positive integer. Assume that starting at v_1, nodes are assigned to cells C_i, $1 \le i \le k$, in ascending order such that $|C_i| = \lceil n/k \rceil$ for $1 \le i \le n \bmod k$ and $|C_i| = \lfloor n/k \rfloor$ for $n \bmod k < i \le k$. The partition $\mathcal{C} = \{C_1, \ldots, C_k\}$ yields a search-space optimal partition if the number of cells is limited by k.*

Proof. We are given a path $P = (V, E, \omega)$ and a positive integer k that indicates the maximum number of allowed cells. Our objective is to find a partition $\mathcal{C}$ such that the total penalty $\sum_{s,t \in P} \mathrm{pen}_{\mathcal{C}}(s,t)$ is minimized. From Lemma 4.6 we know that we can safely assume that there is an optimal partition such that all cells are strongly connected. We have seen in the proof of Lemma 4.6 that, provided all cells are strongly connected, the overall inter-cell penalty is 0. Thus, we only have to minimize the intra-cell penalty of all cells. The sum of all path sizes of a cell C is exactly $|C|^3/3 + |C|^2 - |C|/3$ due to Lemma 2.3. Hence, we obtain the following total penalty.

$$\begin{aligned}\sum_{s,t} \mathrm{pen}_{\mathcal{C}}(s,t) &= \sum_{C \in \mathcal{C}} \left(\sum_{u,v \in C} \mathrm{S}_{\mathrm{AF}}(u,v) - \sum_{u,v \in C} |P_{u,v}| \right) \\ &= \sum_{C \in \mathcal{C}} \left(\frac{|C|^2 \cdot (|C|+1)}{2} - \frac{1}{3}|C|^3 - |C|^2 + \frac{1}{3}|C| \right) \\ &= \sum_{C \in \mathcal{C}} \left(\frac{1}{6}|C|^3 - \frac{1}{2}|C|^2 + \frac{1}{3}|C| \right) \qquad (4.2)\end{aligned}$$

This implies we can assign a penalty $p(x) = x^3/6 - x^2/2 + x/3$ to a cell of cardinality x. If we interpret the polynomial p as a continuous function with the cell size as a parameter, we attain a cost function that is non-negative, increasing and convex on $\mathbb{R}^{\geq 0}$. From Corollary 4.4 we know that the total penalty of P then is minimized if we have $n \bmod k$ cells of size $\lceil n/k \rceil$ and $n - (n \bmod k)$ cells of size $\lfloor n/k \rfloor$. Together with the demand for strongly connected cells, the partition $\mathcal{C}$ stated in the lemma fulfills this requirement and hence yields a minimum penalty for P. $\square$

4.3 Hardness on Undirected Trees

A natural next step after finding a way to attain optimal cells for paths is to concentrate on trees. Unfortunately, we are going to find that provided $\mathcal{P} \neq \mathcal{NP}$, there is no efficient algorithm that can guarantee to find optimal cell assignments on undirected trees. The focus of this section hence is on proving that solving ArcFlagsPartition on undirected trees is $\mathcal{NP}$-hard.

4.3.1 Introductory Notes

Before we turn to the hardness result itself, we shortly examine the different influences on the total penalty induced by a partition if the underlying graph is an undirected tree.

Penalties on Undirected Trees

Our goal in this passage is to give a rough idea of what causes the query algorithm to settle additional nodes when searching for a specific target node. To this end, imagine an arbitrary cell C on an undirected tree $T = (V, E, \omega)$. We know that the cell C does not have to be strongly connected. In contrast to the situation we faced when we studied paths, it may in fact be necessary for a search-space optimal partition to include a cell that is not strongly connected. As a simple example of such a situation imagine a star such as the one given in Figure 3.4 at the end of Chapter 3.

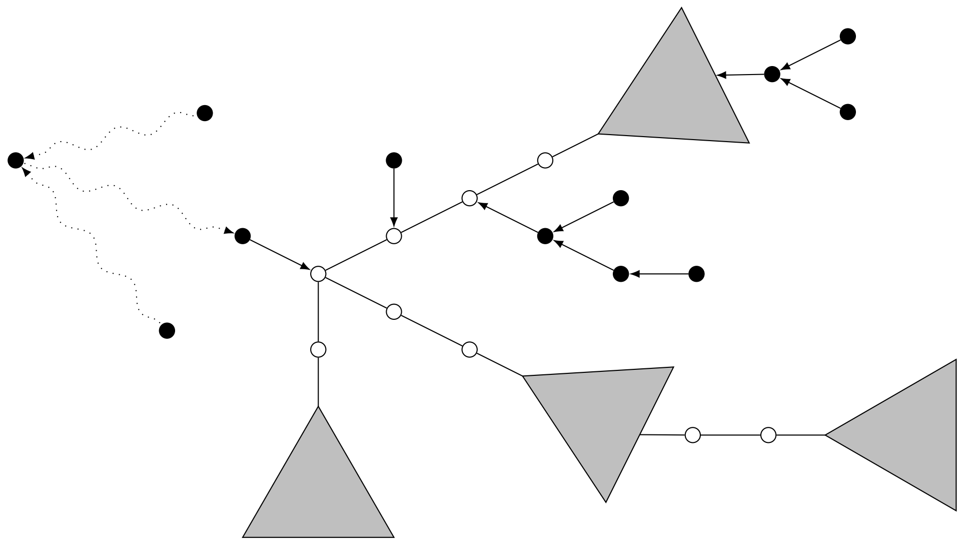

Figure 4.4: A sketch of a minimal set of nodes inducing a strongly connected subtree. Triangles drawn in gray represent all nodes that belong to a certain cell C. Edge directions represent set flags for the cell C.

So assume we are given a cell assignment where at least one cell C is not strongly connected. We know that since we are working on trees, there must be a unique minimal set of nodes

U such that $C \cup U$ induces a strongly connected subgraph of T. See Figure 4.4 for an example. Here, all nodes drawn white belong to the set U. Imagine an inter-cell query with a target node in C. The query algorithm only settles nodes that actually belong to the shortest path from s to t until the first node in $C \cup U$ is settled. Once this has happened, there may be a penalty that depends on the number and the depth of branches that occur on the remaining part of the query to t.

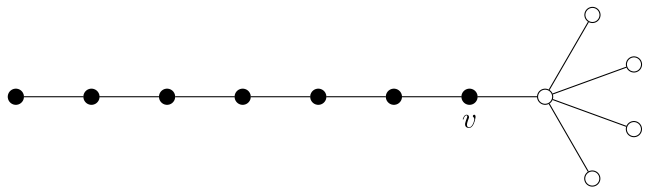

Figure 4.5: An undirected tree and its search-space optimal partition for two cells.

For intra-cell queries, the sizes of C and U mainly determine the search-space size. If U is empty, the intra-cell search-space size of C equals the search-space size of Dijkstra's algorithm on strongly connected graph with $|C|$ nodes. To obtain the penalty of the intra-cell queries, we have to subtract the sizes of all paths with s and t in C. Thus, the intra-cell penalty is minimized if the cell consists of a long path and in turn becomes greater the more branches the cell contains. To see that this in fact may have an influence on the optimal partition, view the tree depicted in Figure 4.5. Although reassigning node v could balance the cell sizes, it would cause a larger search-space size. This is due to the fact that the smaller cell contains more branches. Summarily, to minimize the search-space size one has to consider the following influences.

- The size of a cell mainly determines the intra-cell search-space size. Since the intra-cell search-space size grows at least quadratically in the cell size, the cell sizes of the partition should be in balance.
- The number of nodes from other cells that connect components of a cell should be kept low, because these nodes may additionally get settled in both intra-cell and inter-cell queries without belonging to the shortest path.
- The number of branches within a single cell should be small.

Unfortunately, this information cannot help us in designing an efficient algorithm that generates search-space optimal partitions on trees. However, it might be useful for developing approximative or heuristic approaches that are dedicated to trees.

Hardness Result in the Strong Sense

The remainder of this section is devoted to the proof of $\mathcal{NP}$-hardness of ArcFlagsPartition on undirected trees. To establish this claim, we prove the existence of a polynomial-time reduction from an $\mathcal{NP}$-complete decision problem to the decision variant of the optimization problem ArcFlagsPartition. The decision variant of a minimization problem is to decide whether for a given instance there exists a feasible solution that falls below a given threshold value.

The reduction is made from the problem 3-Partition, which was originally shown to be $\mathcal{NP}$-hard by Garey and Johnson [GJ75]. The task is to separate a given set of weighted elements into triples such that the total weight of each subset equals a given positive integer. This problem is strongly $\mathcal{NP}$-complete and thus remains $\mathcal{NP}$-complete even if all input numbers are polynomial in the number of elements of the input instance. Since the reduction used in our proof only creates numbers of polynomial size relative to the numbers occurring in 3-Partition, this immediately implies strong $\mathcal{NP}$-hardness of ArcFlagsPartition.

Problem 3-Partition**.** *Given a set* $S = \{s_1 \dots s_{3m}\}$ *of* $3m$ *elements, a positive integer* B *and a weight function* $\omega\colon S \to \{0 \dots B\}$ *with* $\sum_{i=1}^{3m} \omega(s_i) = mB$*, decide whether there is a partition of* S *into* m *subsets* S_i*,* $i \in \{1 \dots m\}$*, such that for all* i *it is* $|S_i| = 3$ *and the total weight of each subset is* B*, i.e.,* $\sum_{s \in S_i} \omega(s) = B$*.*

The problem remains $\mathcal{NP}$-complete if one restricts all weights $\omega(s_i)$ to $B/4 < \omega(s_i) < B/2$. We ignore this restriction here for the sake of simplicity. The basic idea of the proof of hardness of ArcFlagsPartition is to transform an input instance of 3-Partition into an undirected tree that requires optimal cells to always cover exactly the graph components that correspond to three elements of S each. To each component of the graph that represents an element s_i of S we attach a number of nodes that equals the weight $\omega(s_i)$ of the according element. Since the cell size has a large impact on the resulting search-space size, the total search-space size then is minimized if the cell sizes are as balanced as possible, that is, there exists a partition into triples of equal weights.

4.3.2 A Preliminary Tool for the Proof of Hardness

Before we begin with the formal proof of $\mathcal{NP}$-hardness, we present and examine a helpful tool. We formally define (m, B, x)*-Trees* that shall later serve for the transformation of instances of 3-Partition into undirected trees.

Definition 4.9 ((m, B, x)-Tree)**.** *Let* m, B, x *be positive integers. An undirected tree* $T = (V, E, \omega)$ *is called* (m, B, x)*-tree if it fulfills the following properties.*

1. *There are* $3m$ *distinct chains of nodes* $v_{i_1}, \dots, v_{i_x}$ *in* V *with the connecting edges* $\{(v_{i_j}, v_{i_{j+1}}), (v_{i_{j+1}}, v_{i_j})\} \subseteq E$ *for all* $1 \leq i \leq 3m$ *and* $1 \leq j < x$*.*
2. *There is a root node* $r \in V$ *with* $\{(r, v_{i_1}), (v_{i_1}, r)\} \subseteq E$ *for all* $1 \leq i \leq 3m$*.*
3. *The last node* v_{i_x} *of each of the* $3m$ *chains has* $B_i \leq B$ *leaves* $v_{i_x,k}$ *attached to it, i.e.,* $v_{i_x,k} \in V$ *and* $\{(v_{i_x}, v_{i_x,k}), (v_{i_x,k}, v_{i_x})\} \subseteq E$ *for* $1 \leq k \leq B_i$ *for all* $1 \leq i \leq 3m$*.*
4. *The total number of attached leaves is* $|\{v_{i_x,k} \in V\colon 1 \leq i \leq 3m, 1 \leq k \leq B_i\}| = mB$*.*
5. *It is* $\omega(e) = 1$ *for all* $e \in E$*.*
6. T *has no additional nodes or edges.*

Figure 4.6 depicts a simple example of an (m, B, x)-tree. The defined (m, B, x)-trees are later used to create undirected trees out of arbitrary instances of the problem 3-Partition. We create one chain of length x for each element in S. A chain that corresponds to a given $s \in S$ then has $\omega(s)$ nodes attached its end. The length x of the chains is specified later. We shall refer to the chains including all additionally attached leaves as *limbs* of the (m, B, x)-tree. Furthermore, when speaking of leaves of an (m, B, x)-tree, we refer to the $m \cdot B$ additionally attached nodes only and exclude any node v_{i_x} at the end of a limb with $B_i = 0$. Hence, the node v in Figure 4.6 would not be counted as a leaf, for instance. In all what follows, we assume that the root r has the least node index according to the order $\prec$ that is used for tie breaking in the query algorithm.

Towards an Optimal Partition of (m,B,x)-Trees

Suppose we are given an (m, B, x)-tree $T = (V, E)$ and our goal is to find a search-space optimal partition with m cells. We shall now prove the following claim. If we set the length x of each limb to $4mB^2$, there always exists a search-space optimal partition of the corresponding (m, B, x)-tree such that each cell covers exactly three limbs of the tree. We call an according partition *limb-balanced.*

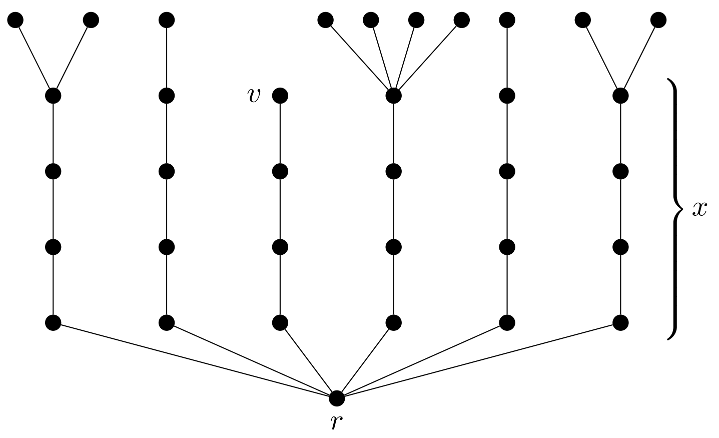

Figure 4.6: An (m, B, x)-tree with $m = 2$, $B = 5$, $x = 4$.

Lemma 4.10. *Let $T = (V, E)$ be an $(m, B, 4mB^2)$-tree with $m \geq 33$. There exists a search-space optimal partition $\mathcal{C} = \{C_1, \ldots, C_m\}$ of T such that each partition $C_i \in \mathcal{C}$ contains the nodes of exactly three entire limbs of T.*

Proof. In what follows, we say that a limb is *monochromatic* if all its limbs are assigned to the same cell. The lemma follows from four basic claims. We show that the search-space size is dominated by long distance queries that pass the root node r. Since the major penalty of such queries stems from the $3m$ branches originating at r, one can observe that it is optimal for these queries to divide the limbs equally among the cells. We then claim that following from this observation there must be a limb-balanced partition such that its inter-cell queries cannot be improved. The basic idea is that an inter-cell query in a limb-balanced partition settles only nodes on the shortest s-t-path until r is reached. The remaining query than works analogous to a query starting at r. Provided with this information, we can show that the search-space size of intra-cell queries of a balanced partition cannot be reduced by splitting cell assignments within a single limb without causing a larger change for the worse concerning the former inter-cell queries. Finally, we show that once that all limbs are monochromatic, a limb-balanced partition that minimizes the search-space size must exist.

Claim 1. There is a limb-balanced partition such that the search-space size $\sum_{t \in V} \mathrm{S}_{\mathrm{AF}}(r, t)$ is minimized. In what follows, we study the search-space size of all queries to a fixed target cell C starting at the root node r. To this end, we examine the search-space size of all queries from r to nodes at a fixed distance d separately and denote this value by

$$S_C^d(r) = \sum_{\substack{v \in C \\ d(r,v)=d}} \mathrm{S}_{\mathrm{AF}}(r, v).$$

We denote the total search-space size of all queries from r to targets in C by $S_C(r)$. Obviously, $S_C(r)$ equals the sum of the search-space sizes $S_C^d(r)$ for all distances $1 \leq d \leq x + 1$, where $x = 4mB^2$. Let furthermore $\ell_{C,d}$ be the number of limbs for which the node at distance d from r is assigned to the target cell C. Similarly, we denote by $\ell_{C,\geq d}$ the number of limbs that possess a node in cell C at a distance from r that is greater or equal d. For instance, in Figure 4.7 we have $\ell_{C,3} = 3$ and $\ell_{C,\geq 3} = 4$ if C represents the cell that contains all nodes filled white.

Next, we first examine queries from r to inner nodes of a limb, before we turn to queries where the target nodes are leaves of a limb. So, consider a query from r to a node v at a fixed distance $d = d(r, v) \leq x$. For a demonstration of the following explanations see

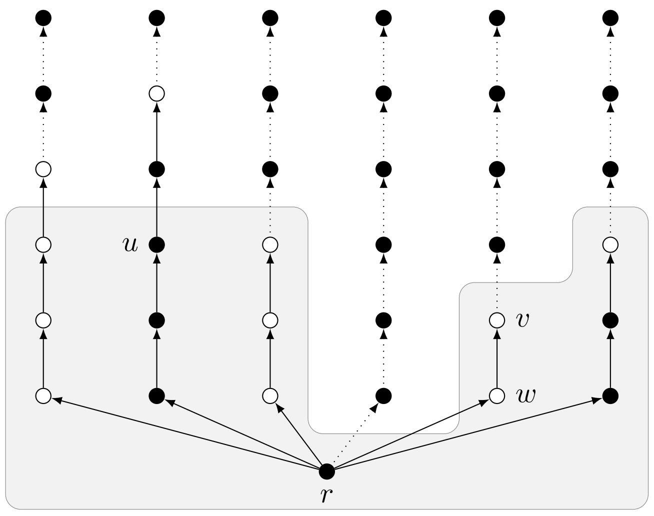

Figure 4.7: A schematic example of the search space induced by the root node. All nodes drawn white are supposed to belong to the considered target cell.

again Figure 4.7. There, all solid edges have the corresponding flag set to 1. The gray box contains a superset of all settled nodes in a query from r to any node at distance $d = 3$ from the target cell. The actual number of settled nodes at distance 3 depends on the node indices that induce the total order $\prec$ for tie breaking. Generally speaking, the search space to a node at distance d includes all nodes at distances smaller than d where the containing limb holds a node at distance d or greater that is assigned to C. We know that there are exactly $\ell_{C,\geq d}$ such limbs. We further have $\ell_{C,d}$ distinct target nodes $t \in C$ at distance d. Hence, the search space of all queries from r to nodes in C at distance d includes at least $(d-1) \cdot l_{C,\geq d} \cdot l_{C,d}$ nodes. In addition to that, there may be certain nodes that belong to a limb which itself holds a node in cell C only at a distance smaller than d. Let A be the set of nodes added to the search space for this reason. In Figure 4.7 we then get $A = \{v, w\}$. We have now summarized all settled nodes t with $d(r,t) < d$.

Finally, the number of settled nodes at a distance of precisely d in a query depends on the index of the target node t. Summing up the queries to all nodes in C at distance d, at least $\sum_{i=1}^{l_{c,d}} i$ nodes are added to the search space. In addition to that, more nodes may be included in the search space that belong to a limb having a node at a distance greater than d assigned to C, whereas the node at distance d is not. Let B_t denote the set of nodes added to the search space of an r-t-query for this reason. Regarding Figure 4.7, we see that $B_t \subseteq \{u\}$ in our example. We can now bound the search-space size S_C of queries from r to all nodes in C at distance d as follows.

$$\begin{aligned} S_C^d(r) &= (d-1) \cdot \ell_{C,\geq d} \cdot \ell_{C,d} + \ell_{C,d} \cdot |A| + \frac{\ell_{C,d} \cdot (\ell_{C,d}+1)}{2} + \sum_{\substack{t \in C \\ d(r,t)=d}} |B_t| \\ &\geq (d-1) \cdot \ell_{C,d}^2 + \frac{\ell_{C,d} \cdot (\ell_{C,d}+1)}{2} \end{aligned} \tag{4.3}$$

The relation in Equation 4.3 becomes an equality if and only if for every limb all nodes belong to the same cell. Therefore, let us assume now we are given a partition in which every limb is monochromatic. Then the optimal number of limbs per cell can be obtained as follows. We can interpret the search-space size $S_C^d(r)$ as a convex polynomial function in the variable $\ell_{C,d}$. From Corollary 4.4 we know that the sum of all m distinct search-space sizes $S_C(r)$ then is minimized if the values of $\ell_{C,d}^2$ are distributed equally for all $1 \leq d \leq x$. Hence, assigning three entire limbs per cell yields an optimal solution for $3m$ limbs and m cells.

So far we have not considered queries from r to nodes at distances $d = x + 1$. Let $n_{C,x+1}$ denote the number of nodes at distance $x + 1$ from r assigned to C, that is, the number of leaves in C. Recall that on the other hand, $\ell_{C,x+1}$ is the number of limbs that have at least one leaf assigned to cell C. Along the lines of the previous discussion, we obtain the following search-space size for nodes at distance $x + 1$, where A has the same meaning as in Equation 4.3.

$$S_C^{x+1}(r) = \underbrace{x \cdot n_{C,x+1} \cdot \ell_{C,x+1}}_{\alpha} + \ell_{C,d} \cdot |A| + \underbrace{\frac{n_{C,x+1} \cdot (n_{C,x+1} + 1)}{2}}_{\beta} \tag{4.4}$$

At first, we examine the term α from Equation 4.4. To minimize α, the number of limbs $\ell_{C,x+1}$ that have to be settled entirely in every query to a leaf of the cell C could possibly be decreased if one rearranged cell assignments cleverly. However, it is clear that $\ell_{C,x+1}$ is at least 1 if C has any leaf assigned, i.e., if $n_{C,x+1} > 0$. Moreover, we have $\ell_{C,x+1} = 3$ in the balanced partition and the total number of leaves is exactly mB, so the maximum gain from rearranging cells is bounded by

$$\begin{aligned}\Delta^-_{S(r)} &\leq \sum_{C \in \mathcal{C}} 3x \cdot n_{C,x+1} - \sum_{C \in \mathcal{C}} x \cdot n_{C,x+1} \\ &= 2xmB. \end{aligned} \tag{4.5}$$

Reassigning nodes to different cells would also have an effect on the search-space size caused by queries from r to inner nodes of the limbs. This search-space size was covered in Equation 4.3. Assume we are given a limb-balanced partition that minimizes the search-space size in Equation 4.3 and we would like to modify this partition in order to decrease α. It is easy to see that splitting cell assignments of nodes belonging to the same limb is not helpful in this context, for this can only increase the values $|A|$ and $\ell_{C,x+1}$. Hence, one would always reassign the nodes of a whole limb at once to minimize α. Assume we adjusted the cells in this manner, obtaining a partition that is not limb-balanced anymore. To obtain the new value of α, we have to update the resulting values of all $\ell_{C,x+1}$. If this was done in iterative steps, one would simultaneously increment an $\ell_{C',d}$ and decrement another $\ell_{C'',d}$ each time. From Lemma 4.3 we know that the search-space size given in Equation 4.3 is non-decreasing regarding all cells for each of these steps. Furthermore, there must be at least one step in which a limb is reassigned starting from a balanced partition with three limbs per cell. This results in the increase of one $\ell_{C',d}$ to 4 and the decrease of another $\ell_{C'',d}$ to 2 for all $1 \leq d \leq x$. Consequently, we get at least the following additional penalty in the search-space size that was summarized by Equation 4.3.

$$\begin{aligned}\Delta^+_{S(r)} &\geq \sum_{d=1}^{x} (d-1) \cdot 4^2 + \sum_{d=1}^{x} (d-1) \cdot 2^2 - 2 \sum_{d=1}^{x} (d-1) \cdot 3^2 \\ &= 2 \frac{x(x-1)}{2} \end{aligned} \tag{4.6}$$

Taking into account that we have $x = 4mB^2$, the change in the search-space size we receive from Equations 4.5 and 4.6 is always non-negative, as shown below.

$$\begin{aligned}\Delta_{S(r)} &= \Delta^+_{S(r)} - \Delta^-_{S(r)} \\ &\geq 4mB^2(4mB^2 - 1) - 8m^2B^3 \\ &\geq 0 \end{aligned} \tag{4.7}$$

Now, consider the term summarized by β in Equation 4.4. Here, the represented search-space size would be minimized if we could keep each limb of the (m, B, x)-tree monochromatic while leveling the cell sizes. However, the number of leaves that are attached at the end of a limb can have any value between 0 and B. Therefore, we cannot guarantee that the number of nodes at distance $x+1$ can be perfectly balanced over all cells as long as we stick to the limb-balanced partition. We know however, that the total number of nodes at distance $x+1$ is exactly mB. To obtain an upper bound on the possible saving, we maximize the terms β over all cells in $\mathcal{C}$ following Lemma 4.3. Starting at an arbitrary distribution of the numbers $n_{C,x+1}$ over the cells of the tree, we iteratively apply one of the two steps from Lemma 4.3, until we reach $n_{C,x+1} = mB$ for a $C \in \mathcal{C}$ and $n_{C',x+1} = 0$ for all $C' \neq C$. We can transform an aribtrary configuration this way and the sum over all values β is always non-decreasing. Summing up the values of the terms β for all cells thus cannot yield a number larger than $mB(mB+1)/2$. The maximum amount of penalty we can save for the whole search-space size $S(r) = \sum_{C\in\mathcal{C}} S_C(r)$ caused by queries from r is thus bounded by

$$\Delta^-_{S(r)} \leq \frac{mB(mB+1)}{2}. \tag{4.8}$$

The only way to reduce the values of the terms β in queries from r to the leaves of an (m,B,x)-tree in comparison to a limb-balanced partition is to divide the leaves of at least one limb into more than one cell. However, splitting the cell assignments at the end of any limb into two or more cells must result in at least one cell C' for which we produced an additional limb that has the flag of C' set to 1 on all x edges leading to the leaves of the according chain. Thus, the corresponding value $\ell_{C',d}$ must be incremented by 1 in Equation 4.3 for all $1 \leq d \leq x$. Hence, we create a growth of the search-space size that is at least

$$\Delta^+_{S(r)} \geq \sum_{d=1}^{x}(d-1) = \frac{x(x-1)}{2}. \tag{4.9}$$

Setting $x = 4mB^2$ and taking into account that m and B are natural numbers, the total change in the search-space size $S(r)$ regarding Equations 4.8 and 4.9 is as follows.

$$\begin{aligned}\Delta_{S(r)} &= \Delta^+_{S(r)} - \Delta^-_{S(r)} \\ &\geq 2mB^2(4mB^2-1) - \frac{mB(mB+1)}{2} \\ &= 8m^2B^4 - 2mB^2 - \frac{1}{2}m^2B^2 - \frac{1}{2}mB \\ &\geq 4m^2B^4 \end{aligned} \tag{4.10}$$

As a result, Equations 4.7 and 4.10 show us that the search-space size $S(r)$ cannot be decreased by choosing cell assignments other than the limb-balanced one that we proposed.

Claim 2. There exists a limb-balanced partition for which the search-space size of all inter-cell queries cannot be reduced. We apply the above reasoning to any s-t-path that passes the root node r. If r is part of the unique path $P_{s,t}$ from s to t, the search space corresponding to an s-t-query must include all nodes from $S_C(r)$ and we clearly have $\mathrm{S}_{\mathrm{AF}}(s,t) \geq \mathrm{S}_{\mathrm{AF}}(r,t)$. Hence, there is a limb-balanced partition that yields an optimal partition as soon as the query algorithm has settled r. However, we have to take account of the search-space size caused by the query from s to r. But if s and t belong to different cells, i.e., $c(s) \neq c(t)$, the penalty of this part of the query is zero in the limb-balanced partition, because flags are only set for edges that head towards the root node r. Consequently, only nodes that belong to the actual path $P_{s,t}$ are settled until r is reached and we cannot improve the search-space size by changing the underlying partition.

We have seen above that there must be a limb-balanced partition such that we cannot improve the search-space size induced by its inter-cell queries. However, the search space of intra-cell queries according to the balanced partition might be improved if we adjust the partition. In what follows, we show that any improvement obtained by changing the underlying partition is outweighed by the penalty it must entail for queries that pass the root node.

Claim 3. There is an optimal partition such that all its limbs are monochromatic. We analyze the limbs of the tree separately. For each limb, we consider all queries for which the source or target node lies in the given limb. Let $\mathcal{C}$ be a best possible limb-balanced partition that serves as our benchmark. We then have to show that splitting the cells of the considered limb does not improve the corresponding search-space size compared to the balanced partition. To achieve this goal, we first derive lower bounds on the increase of the search-space size compared to a limb-balanced partition. Afterwards, we obtain upper bounds on the savings that can be created relative to this partition. By comparing these bounds, we prove the claim.

Assume we are given an arbitrary limb of the tree. Let the nodes of its chain be $v_1, \dots, v_x$, where the distance of r to v_i is exactly i for any $1 \leq i \leq x$. Furthermore, assume the leaves of the limb are given by $v_{x+1}, \dots, v_{x+\beta}$ with $\beta \leq B$. We know that compared to an optimal limb-balanced partition $\mathcal{C}$, we can only hope for improving those queries that are intra-cell queries in $\mathcal{C}$. In order to achieve such improvements, one could divide the nodes of the limb into two or more cells. Then, there must be a cell C for which there is a node v_i that has the largest distance $d(r, v_i) = \min\{i, x+1\}$ occurring for any node in C that belongs to this limb and there is at least one node v_j with $j > i$ left in the limb. Given this value i, let $x_1 = |\{v_j \mid i < j \leq x\}|$ and $x_2 = |\{v_j \mid v_j \in C \vee i \geq j\}|$. In other words, x_2 is the number of nodes in the considered limb that are reachable from r in the cell-dependent graph T_C. Figure 4.8 depicts an example. Below, we distinguish the following cases. If C contains only nodes v_j with $j < x$, we have $x_1 + x_2 = x$ and $x_1, x_2 \geq 1$. Alternatively, the cell C may also contain the node v_x or an arbitrary number of leaves of the limb. In this case we obtain $x_1 = 0$ and $x \leq x_2 \leq x + B$.

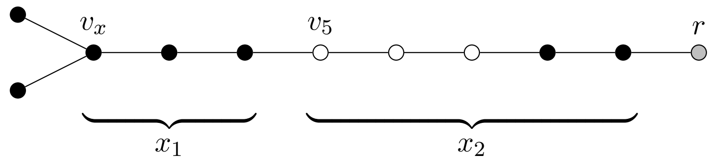

Figure 4.8: Notation for the parts of a limb when v_5 is the farthest node from r in this limb of a certain cell that contains all nodes filled white.

First, we derive lower bounds on the additional penalty caused by splitting a limb into two or more cells. Note that we do not make any assumptions about the cell assignments of other limbs in the tree. Instead, we seek to establish a bound independent of these assignments that provides us with the costs of splitting a limb into several cells. Observe that a monochromatic limb has only one flag set to 1 on all its edges that head towards the leaves. Conversely, a cell C that contains only a subset of the nodes $\{v_1, \dots, v_i\}$ of the given chain causes the i innermost edges of the corresponding limb to have at least two flags set to 1. Below, we minimize the growth in the search-space size $S_C(r)$ given by Equations 4.3 and 4.4 with respect to these changes. We derive lower bounds on the change in this search-space size subject to the only additional condition of setting the flag for C on at most x_2 inner edges of the given limb. This yields a lower bound on the costs of splitting a limb at the node v_{x_2}, regardless of the actual cell assignments of the remaining limbs. Depending on the value of x_2, we distinguish three different cases.

1. $x_1 \geq x_2 \geq 1$. Recall that in Equation 4.3, A denotes the set of nodes settled due to a certain cell containing nodes only at a distance smaller than the considered distance

d. For queries to nodes t in C with $d(r,t) > x_2$, we then have $|A| \geq x_2$. Hence, for $d > x_2$ the bound given by Equation 4.3 becomes

$$S_C^d(r) \geq (d-1) \cdot \ell_{C,d}^2 + \ell_{C,d} \cdot x_2 + \frac{\ell_{C,d} \cdot (\ell_{C,d}+1)}{2}.$$

The search-space size of queries from r to all nodes at a distance greater than x_2 is then minimized if all limbs are assumed to be balanced on all but their x_2 inner nodes. This is due to the fact that an imbalanced partition would inevitably increase the sum $\sum_{C' \in \mathcal{C}} \ell_{C',d}^2$ by at least one, as we observed before. This leads to an increase of $d-1 \geq x_1$ in the corresponding bound of some cell. However, using a limb-balanced partition implies that we have at least $3x_1$ distinct target nodes $t \in C$ such that an r-t-query encounters an additional penalty of x_2. We can charge this penalty for the inter-cell penalty of $(3m-1)x$ source nodes s, regarding all but the modified limb. Hence, we obtain the following bound on the change in the search-space size. Recall that we have requested $m \geq 33$ in the lemma and thus we can safely assume that $(3m-1) \geq 98$.

$$\begin{aligned}\Delta^+ &\geq (3m-1)x \cdot 3x_1x_2 \\ &\geq 26xx_1x_2\end{aligned}$$

2. $x_2 > x_1 \geq 1$. In this case, we consider queries to nodes in C at a distance $d \leq x_2$ from r. Since in the examined limb we have the flags for an additional cell set on its x_2 inner nodes, the value $\ell_{C,\geq d}$ must increase for C in the search-space size stated in Equation 4.3. Again, the best case occurs if we assume that all limbs are balanced on their x_2 inner limbs, which leaves us with $\ell_{C,d} \leq 4$ and $A = B = \emptyset$ for all $d \leq x_2$. However, there must be at least one $\ell_{C',d} \geq 4$ for some cell C'. Hence, every query to a node in C' at a distance $d \leq x_2$ from r pays additional costs of at least $d-1$ and we get a penalty that is at least

$$\begin{aligned}\Delta^+ &\geq (3m-1)x \cdot 3\sum_{d=1}^{x_2}(d-1) \\ &\geq (3m-1)x \cdot 3\frac{x_2(x_2-1)}{2} \\ &\geq 27xx_2(x_2-1) \\ &\geq 26xx_2^2.\end{aligned}$$

3. $x_2 \geq x$. The cell C now includes some or all leaves of the limb or the last node v_x of the chain. Furthermore, we know that at least one of these nodes is assigned to a different cell. The overall penalty of queries starting at r in comparison to a limb-balanced partition when a limb has two set flags on all edges of its chain is bounded by $4m^2B^4 = x^2/4$ in Equation 4.10. But the same penalty can be assigned to any source node that belongs to one of the $(3m-1)$ remaining limbs. Since each of them contains at least x nodes and we have $m \geq 33$, we obtain the following penalty.

$$\begin{aligned}\Delta^+ &\geq (3m-1)x \cdot \frac{x^2}{4} \\ &\geq 24x^3\end{aligned}$$

Note that we can charge these penalties independently for all added cells of the limb, because each of them is responsible for a different flag that must additionally be set to 1 on its inner edges. Moreover, the overall penalty of several limbs that are multicolored must be at least as large as the sum of their penalties obtained above. This is due to the fact that each of them sets x_2 new flags that additionally get settled in queries starting at r, and hence we can assign the lower bounds on the cost corresponding one of the three cases described above to each of them.

Next, we bound the gain that is the result of a partition dividing a single limb into several cells. Namely, we investigate all queries either starting in the modified limb or searching for any node inside it. Assume we are given a fixed cell C such that C does not cover all nodes of the limb. Regardless of the cell assignment of r we know that $S(r)$ cannot be improved, so it is ignored here. We also know that queries cannot be improved once that they passed the root node r. Hence, we only consider those parts of queries that take place within a single limb. Let X_1 and X_2 denote the sets of all up to $x_1 + B$ outer and x_2 inner nodes of the limb, respectively, according to the cell C as in Figure 4.8. Note that X_1 also contains the leaves of the corresponding limb. First of all, we study queries that aim at one of the x_2 inner nodes, some of which belong to the cell C. We know that we cannot improve the search-space size for queries that were inter-cell queries in the balanced partition. However, improvements are possible for former intra-cell queries.

Considering source nodes in other limbs, the assignment of nodes to C may prevent former intra-cell queries from settling nodes that belong to the limb of the source node but not to the unique s-t-path. We know that there are two limbs containing up to $2(x+B)$ nodes in a balanced partition for which this may occur. Hence, for up to x_2 targets at most $(x+B)$ settled nodes are saved. We distinguish penalties according to the three cases studied above. Note that $x_1 \geq x_2$ implies $x_1 \geq x/2 \geq B^2$ and $x_2 > x_1$ implies $x_2 \geq x/2 \geq B^2$.

$$\begin{aligned}\Delta^-_{V\setminus(X_1\cup X_2)\to X_2} &\leq 2(x+B)\cdot(x+B)\cdot x_2\\ &= 2x^2x_2 + 4xx_2B + 2x_2B^2\\ &\leq \begin{cases} 4xx_1x_2 + 4xx_1x_2 + 2xx_2, & \text{if } x_1 \geq x_2\\ 4xx_2^2 + 4xx_2^2 + 2xx_2, & \text{if } x > x_2 > x_1\\ 8x^3, & \text{if } x_1 = 0, x_2 \leq x + B\end{cases} \qquad (4.11)\end{aligned}$$

On the other hand, queries starting in X_2 that head for former intra-cell targets that belong to a different limb may possibly be prevented from settling nodes in X_1. Below, we account for the savings due to this change. Terms bounding the gain are obtained by transformations similar to Equation 4.11. We omit listing them explicitly here for brevity.

$$\begin{aligned}\Delta^-_{X_2\to V\setminus(X_1\cup X_2)} &\leq x_2\cdot 2(x+B)\cdot x_1\\ &= 2xx_1x_2 + 2xx_2B \qquad (4.12)\end{aligned}$$

A query from a source node $s \in X_1$ to a target $t \in X_2$ may not need to settle nodes in X_1 that do not belong to the s-t-path. Hence, there are at most $x_1 + B$ distinct source nodes saving up to $x_1 + B$ nodes on a query to each of x_2 possible target nodes.

$$\begin{aligned}\Delta^-_{X_1\to X_2} &\leq x_2\cdot(x_1+B)^2\\ &= x_1^2x_2 + 2x_1x_2B + x_2B^2 \qquad (4.13)\end{aligned}$$

Compared to a limb-balanced partition, queries from a source $s \in X_2$ to a target $t \in X_2$ could save nodes in X_1, so we get the following bound on the gain.

$$\begin{aligned}\Delta^-_{X_2\to X_2} &\leq (x_1+B)\cdot x_2^2\\ &= x_1x_2^2 + x_2^2B \qquad (4.14)\end{aligned}$$

Now, we consider queries where the target is any of the nodes in X_1. Note that we did not explicitly specify the number of cells occurring in X_1. Depending on their cell assignments, compared to intra-cell queries in the balanced partition incoming from other limbs one may save settled nodes just as in the case of $\Delta_{V' \to X_2}$. However, note that this gain is independent of the presence of the cell C. Hence, we obtain the same gain if we assign the nodes in C to any other cell that is present in X_1. Since we investigate the gain induced by C, we can omit the gains described above. Instead, we examine how C may actually improve the search-space size of queries into X_1. First of all, queries with both their source and target node in X_1 could possibly save settled nodes in X_2.

$$\begin{aligned} \Delta^-_{X_1 \to X_1} &\leq (x_1 + B)^2 \cdot x_2 \\ &= x_1^2 x_2 + 2x_1 x_2 B + x_2 B^2 \end{aligned} \tag{4.15}$$

In addition to that, arbitrary queries from X_2 to X_1 need to be credited for. There are at most x_2 distinct source nodes in X_2 and at most $(x_1 + B)$ distinct targets in X_1. In a balanced partition, these queries are forced to possibly check all nodes in X_2, so at most x_2 nodes can be saved.

$$\begin{aligned} \Delta^-_{X_2 \to X_1} &\leq x_2^2 \cdot (x_1 + B) \\ &= x_1 x_2^2 + x_2^2 B \end{aligned} \tag{4.16}$$

The total gain achieved by assigning some of the nodes of the certain limb to the cell C can be bounded by the sum of all terms given in Equations 4.11 to 4.16. However, the overall penalties found above are greater or equal to their sum in any case. Since we can argue this way for any cell introduced into the limb, there must be a partition in which this limb is monochromatic that induces a lower or equal search-space size.

Claim 4. For any partition $\mathcal{C}$ containing only monochromatic limbs, there is a limb-balanced partition $\mathcal{C}'$ such that $\mathrm{S}_{\mathrm{AF}}(T, \mathcal{C}') \leq \mathrm{S}_{\mathrm{AF}}(T, \mathcal{C})$. To complete the proof, we have to show that given all limbs are required to be monochromatic, a limb-balanced partition exists that minimizes the search-space size. We already know that this is true for inter-cell queries. For intra-cell queries, it is clear that cells are almost strongly connected. The only node that possibly gets settled in an intra-cell query even though it does not belong to that cell is the root node r. Hence, we can bound the intra-cell search-space size of a cell C consisting of i limbs as follows.

$$(ix)^2 \frac{(ix+1)}{2} \leq \sum_{s,t \in C} \mathrm{S}_{\mathrm{AF}}(s,t) \leq (i(x+B)+1)^2 \frac{i(x+B)+2}{2} \tag{4.17}$$

We can interpret both bounds as convex functions in i. Hence, starting from a balanced partition with three limbs per cell, we can adjust these numbers to match any non-balanced partition using the rules stated in Lemma 4.3. The bounds given in Equation 4.17 are non-decreasing in each of these steps and there is one step in which we have two cells containing four and two entire limbs, respectively. We thus know that the change in the search-space size is at least

$$\begin{aligned} \Delta^+ &= 16x^2 \frac{(4x+1)}{2} + 4x^2 \frac{(2x+1)}{2} - 2 \cdot (3(x+B)+1)^2 \frac{3(x+B)+2)}{2} \\ &= 9x^3 - \left(26x^2 + 81x^2 B + 81xB^2 + 72xB + 15x + 27B^3 + 36B^2 + 15B + 2\right) \\ &\geq 9x^3 - 355x^2 B. \end{aligned}$$

Setting $x = 4mB^2$, we obtain a bound that is increasing for $m, B > 0$. Furthermore, setting $m \geq 33$ yields a positive penalty. Hence, we minimize the search-space size if the partition is balanced. Together with Claims 1, 2 and 3, this finishes the proof. □

Note that the requirement of $m \geq 33$ is rather a convenient tool to simplify the proof than a necessary condition. We conjecture that Lemma 4.10 holds for any $m \in \mathbb{N}^+$. However, a more sophisticated analysis would be necessary to prove this and the constant bound provided above suffices for our purposes.

4.3.3 Examining the Search-Space Size of an (m,B,x)-Tree

Before we turn to the proof of hardness on undirected trees in general, we inspect the search-space size of a partition of an (m, B, x)-tree with m cells. We know from Lemma 4.10 that for any sufficiently large (m, B, x)-tree there is a search-space optimal partition where each cell holds three entire limbs. Since we are interested in search-space optimal partitions only, we concentrate on a formal description of the search-space size of such partitions.

Our aim is to find both an upper bound and a lower bound on the optimal search-space size $\mathrm{S}_{\mathrm{AF}}(T, \mathcal{C})$ of an (m, B, x)-tree $T = (V, E)$ if we are given the parameters m and B and a limb-balanced partition $\mathcal{C} = \{C_1, \dots, C_m\}$. Furthermore, we assume that the number of leaves that each cell possesses is known and denote by B_i the number of leaves of the cell C_i. We shall divide the search space of T into three independent components for convenience. First of all, we consider the search-space size of all inter-cell queries, where neither the source nor the target node is the root node r. The second component consists of the intra-cell queries, again ignoring queries where r is the source or the target of the query. Finally, we complete the examination by investigating the search-space size of all queries that either start or end at r.

Inter-Cell Search-Space Size of (m,B,x)-Trees

At first, we study the search-space size of inter-cell queries in T. In particular, we take into account any s-t-query where $c(s) \neq c(t)$ with $s \neq r$, $t \neq r$. Basically, we can divide such a query into two parts. The first part begins when the query algorithm settles s and ends when the root r is settled, excluding r from the search space. We shall refer to this part as the front-query. The second part then covers the search after r has been settled until the target node t is reached, including r. We shall call this part the rear-query. We know that s and t belong to different cells and consequently are not part of the same limb. Thus, the query starts at s and settles exactly the nodes on the path from s to r, because flags for the target cell are only set for edges that lead into this direction. We summarize the front part of the search space for a fixed target node t and count all possible source nodes of an arbitrary cell C_i that is not the target cell. In each of these front-queries we get $d(s, r)$ settled nodes for a source node that is part of the chain, resulting in $\sum_{z=1}^{x} z = x(x+1)/2$ settled nodes per limb, which in total yields $3x(x+1)/2$ for all three limbs of C_i. In addition to that, there are $x + 1$ settled nodes for any of the B_i leaves of cell C_i. Depending on the number of leaves B_i, we get the following inter-cell search-space size for the cell C_i when regarding only the front-part.

$$s_1[B_i] = 3\frac{x(x+1)}{2} + B_i(x+1) \tag{4.18}$$

The number of distinct inter-cell target nodes t that correspond to this search-space size equals the number of all nodes that do not belong to the considered cell C_i holding the source nodes s. Since we ignore the root node r as a possible target for now, we must count all nodes in other limbs including the remaining leaves. Since the number of limbs in T is exactly $3m$ and the total number of leaves is mB, the number of inter-cell targets sums up to the following term.

$$n_1[B_i] = (3m - 3)x + mB - B_i \tag{4.19}$$

Now, we allow for the second part of the search space that consists of all nodes that are settled once the root node is reached. We know that a fixed target cell C_j with $j \neq i$ has exactly $3x + B_j$ nodes. The nodes of the target cell C_j are settled in a deterministic order. Furthermore, the root node is settled in any of these queries, accounting for an increase of

1 per query. For a fixed source node s, the search-space size of all rear-queries considering one target cell therefore becomes $\sum_{t \in C_j} \mathrm{S}_{\mathrm{AF}}(s,t) = \sum_{z=2}^{(3x+B+1)} z$. The total search-space size of queries from one source node to all inter-cell targets is obtained if one sums up the search-space sizes of all distinct target cells.

$$s_2[B_i] = \sum_{j \neq i} \left(\frac{(3x + B_j + 1)(3x + B_j + 2)}{2} - 1 \right) \tag{4.20}$$

It is also known that there are $3x + B_i$ possible source nodes s in the cell C_i, each of which causes the rear search-space size given in Equation 4.20.

$$n_2[B_i] = 3x + B_i \tag{4.21}$$

Moreover, we know that there are m cells in total, and each cell C_i generates the search-space size in Equations 4.18 and 4.20 for a number of target nodes given in Equation 4.19 and a number of source nodes given in Equation 4.21, respectively. The total inter-cell search-space size for all pairs of nodes $s, t \neq r$ with $c(s) \neq c(t)$ then sums up as stated below.

$$\sum_{\substack{s,t \neq r \\ c(s) \neq c(t)}} S(s,t) = \sum_{i=1}^{m} \left(n_1[B_i] \cdot s_1[B_i] + n_2[B_i] \cdot s_2[B_i] \right) \tag{4.22}$$

Intra-Cell Search-Space Size of (m,B,x)-Trees

We can now turn to the intra-cell search space of the given (m, B, x)-tree. Again, we ignore all queries where the root node r is the source or target node of the query. If we furthermore ignore for now that the root node may be settled within an intra-cell query with source $s \neq r$ and target $t \neq r$, we can consider the intra-cell search-space size to be equal to the search-space size of Dijkstra's algorithm stated in Lemma 2.2. Then, we get the following intra-cell search-space size for each cell C_i.

$$s_3[B_i] = \left(\frac{(3x + B_i)^2 (3x + B_i + 1)}{2} \right) \tag{4.23}$$

To preserve the correct value of the intra-cell search-space size of a cell C_i, we certainly have to take account of the root node as well. Therefore, we now cover all s-t-intra-cell queries with $s \neq r$ and $t \neq r$ in which the root node is settled. Clearly, for every such query we have to increase the search-space size given in Equation 4.23 by 1. For this reason, we count all intra-cell queries in which r is settled and add the corresponding number to the intra-cell search-space size given in Equation 4.23. Remember that we required the root r to have the least index according to the order $\prec$. Thus, the node r is settled first whenever it is enqueued and has the lowest index within the queue.

First of all, the root node is certainly settled in any intra-cell query where s and t belong to different limbs of the (m, B, x)-tree, because r is then part of the unique s-t-path. Let B_{i_j} for $j \in \{0, 1, 2\}$ denote the number of leaves in the $(j + 1)$-th limb of the cell C_i. To account for the mentioned queries, we multiply the number of source nodes of each limb by the number of intra-cell targets in the remaining limbs of the corresponding cell.

$$\begin{aligned} s_4[B_i] &= \sum_{j=0}^{2} \left(x + B_{i_j} \right) \cdot \left(2x + B_{i_{(j+1) \bmod 3}} + B_{i_{(j+2) \bmod 3}} \right) \\ &= 6x^2 + 4x \left(B_{i_0} + B_{i_1} + B_{i_2} \right) + 2B_{i_0}B_{i_1} + 2B_{i_0}B_{i_2} + 2B_{i_1}B_{i_2} \\ &= 6x^2 + 4xB_i + \underbrace{2 \left(B_{i_0}B_{i_1} + B_{i_0}B_{i_2} + B_{i_1}B_{i_2} \right)}_{\gamma} \end{aligned} \tag{4.24}$$

Since we are interested in a formulation of the search-space size that is independent of the values B_{i_z}, we would like to find bounds on the term γ rather than the exact value in Equation 4.24. Obviously, we minimize γ if we set $B_{i_0} = B_i$ and $B_{i_1} = B_{i_2} = 0$, which yields a lower bound of

$$\gamma^L = 0.$$

Conversely, we maximize γ as a next step. For the sake of simplicity we assume that $B_i = 3a$ for some $a \in \mathbb{N}$. We show that γ then is maximized if $B_{i_j} = B_i/3$ for $j \in \{0,1,2\}$. To see that this indeed yields a maximum, consider arbitrary values B'_{i_j} with $\sum_{j=0}^{2} B'_{i_j} = B_i$. We can iteratively construct B'_{i_j} from the corresponding B_{i_j} if in every step we simultaneously increment a $B_{i_p} < B'_{i_p}$ while decrementing another $B_{i_q} > B'_{i_q}$ by 1 each, until $B_{i_j} = B'_{i_j}$ holds for all $j \in \{0,1,2\}$. Consider one of these steps and without loss of generality assume that B_{i_0} is incremented and B_{i_1} is decremented. This step then corresponds to the following change in the search-space size

$$\begin{aligned}\Delta_{s_4}[B_i] &= 2\left(\left(B_{i_0}+1\right)\left(B_{i_1}-1\right)+\left(B_{i_1}-1\right)B_{i_2}+\left(B_{i_0}+1\right)B_{i_2}\right)-\gamma\\ &= 2\left(B_{i_0}B_{i_1}-B_{i_0}+B_{i_1}-1+B_{i_1}B_{i_2}-B_{i_2}+B_{i_0}B_{i_2}+B_{i_2}\right)-\gamma\\ &= 2B_{i_1}-2B_{i_0}-2\end{aligned}$$

Moreover, we know that $B_{i_0} \geq B/3$ and $B_{i_1} \leq B/3$. This implies that Δ_{s_4} is negative and consequently setting $B_{i_j} = B_i/3$ for $j \in \{0,1,2\}$ maximizes γ for the case of B_i being divisible by 3 without remainder. But then it is clear that $B_{i_j} = \lceil B_i/3 \rceil$ for $j \in \{0,1,2\}$ yields a feasible upper bound on γ for arbitrary values of B_i. Hence, we obtain an upper bound on $s_4[B_i]$ if we use the following value γ^U in Equation 4.24.

$$\begin{aligned}\gamma^U &= 2\left(B_{i_0}\left\lceil\frac{B_i}{3}\right\rceil + B_{i_2}\left\lceil\frac{B_i}{3}\right\rceil + B_{i_1}\left\lceil\frac{B_i}{3}\right\rceil\right)\\ &= 2B_i\left\lceil\frac{B_i}{3}\right\rceil\end{aligned}$$

The obtained values γ^L and γ^U induce a lower bound $s_4^L[B_i]$ and an upper bound $s_4^U[B_i]$ on $s_4[B_i]$, respectively.

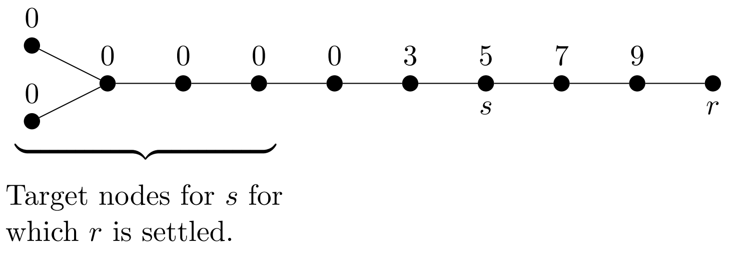

Figure 4.9: Enumeration of queries within limbs for which the root node gets settled. Labels denote the number of queries with the corresponding node as source node in which r is settled.

Finally, intra-cell queries that settle the root node when s and t belong to the same limb must be accounted for. We consider a fixed node s and inspect all queries to target nodes t inside the same limb. Clearly, the root node is only settled if s is located between t and r, because otherwise the stopping criterion ensures that r is never settled. Instead, assume that $d(s,r) < d(t,r)$ holds. The root node r is then settled in queries where $d(s,r) \leq d(s,t)$ holds. Remember that r has the least index of all nodes and that x is an even number. If s is a leaf of the limb or belongs to the $x/2$ outer nodes of the limb, r is never settled in a query to a target inside the same limb. If s belongs to the inner $x/2$ nodes, the root node is settled for exactly $x - 2d(s,r) - 1$ targets inside the chain of the same limb. Furthermore, the root then gets settled on all queries to a leaf of the tree. See also Figure 4.9 for an illustration. Since we have three limbs and B_i leaves in total, we obtain the following search-space size.

$$\begin{aligned} s_5[B_i] &= 3\sum_{z=1}^{x/2}(2z - 1 + B_i) \\ &= 3 \cdot 2\frac{\frac{x}{2}\left(\frac{x}{2}+1\right)}{2} - 3\frac{x}{2} + \frac{x}{2}B_i \\ &= \frac{3}{4}x^2 + \frac{x}{2}B_i \end{aligned} \tag{4.25}$$

Still omitting queries with r as source or target, we obtain the total intra-cell search-space size if we add up the results stated in Equations 4.23 4.24 and 4.25. This yields the following lower bound.

$$\begin{aligned} \sum_{\substack{s,t \neq r \\ c(s)=c(t)}} S(s,t)^L &= \sum_{i=1}^{m} \left(s_3[B_i] + s_4[B_i]^L + s_5[B_i]\right) \\ &= \sum_{i=1}^{m} \left(\frac{(3x+B_i)^2(3x+B_i+1)}{2} + \frac{27}{4}x^2 + \frac{9}{2}xB_i\right) \end{aligned} \tag{4.26}$$

Analogously, the corresponding upper bound on the intra-cell search-space size apart from the search space caused by queries starting at r or searching for r is as follows.

$$\begin{aligned} \sum_{\substack{s,t \neq r \\ c(s)=c(t)}} S(s,t)^U &= \sum_{i=1}^{m} \left(s_3[B_i] + s_4[B_i]^U + s_5[B_i]\right) \\ &= \sum_{i=1}^{m} \left(\frac{(3x+B_i)^2(3x+B_i+1)}{2} + \frac{27}{4}x^2 + \frac{9}{2}xB_i + 2B_i\left\lceil\frac{B_i}{3}\right\rceil\right) \end{aligned} \tag{4.27}$$

Search-Space Size of (m,B,x)-Trees Caused by the Root Node

As a last step, we now make up for all possible queries in T in which r is either the start or the target node. The search-space size caused by the queries from r to the nodes of a fixed cell C_i is $\sum_{z=2}^{(3x+B_i+1)} z$ according to the cell-dependent graph G_{C_i}, adding 1 in each query to account for r. For C_i we then get the following search-space size.

$$s_6[B_i] = \left(\frac{(3x+B_i+1)(3x+B_i+2)}{2} - 1\right) \tag{4.28}$$

We also have to consider all queries in which the target node is r. The search space of any query to the root node at least covers the path from the source node s to the target r. For each cell C_i we have three limbs with x paths of a length ranging from 2 to $x+1$ each and B_i paths of length $x+2$. So the corresponding search-space size for the whole graph T is as follows.

$$s_7 = m\left(3\left(\frac{(3x+1)(3x+2)}{2} - 1\right)\right) + mB(x+2) \tag{4.29}$$

For inter-cell queries with target node r, only the nodes on the path from the source node s to r are visited. Consequently, these queries are covered by Equation 4.29. However,

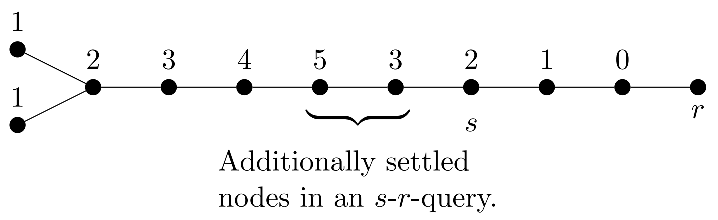

Figure 4.10: Additionally settled nodes on intra-cell queries to the root node. Labels denote the penalty of the corresponding node as source node for the target r.

there is exactly one cell for which the queries from its interior nodes to r are intra-cell queries and these queries produce a search space that additionally may hold nodes that do not belong to the direct path from source to destination. In what follows, we take these nodes into account. Below, let C_{i_r} be the corresponding cell that contains r.

If the source node s belongs to the chain of the limb, the number of additionally settled nodes depends on its position. If s lies on the inner half of the limb, i.e., $d(s,r) \leq x/2$, the number of settled nodes that are not on the s-r-path is exactly $d(s,r) - 1$. Figure 4.10 demonstrates this situation. Recall that r was assigned the least index and is thus always taken first from the queue in case of a tie. Summarized for all nodes with $d(s,r) \leq x/2$ of a single limb, this yields a penalty of $\sum_{z=1}^{x/2}(z-1)$. Conversely, if the source node s is on the outer half and consequently $d(s,r) > x/2$, every node of the limb must be settled in an s-r-query. Since the number of remaining nodes not on the s-r-path is exactly $x - d(s,r) - 1$ plus the number of leaves $B_{i_{rj}}$ of the limb with index $j \in \{0,1,2\}$ of cell C_{i_r}, the penalty charged for this part of the limb becomes $\sum_{z=1}^{x/2}(z-1) + (x/2)B_{i_{rj}}$.

Finally, we have to consider the case where the source node is a leaf of the limb. In this case, the shortest path from s to r covers all nodes of the chain. The query algorithm additionally settles the remaining leaves of the current limb. This leads to $B_{i_{rj}}(B_{i_{rj}} - 1)$ settled nodes for the $j+1$-th limb of the cell C_{i_r}. Lemma 4.3 implies that this is maximized for the whole cell by $B_{i_r}(B_{i_r} - 1)$, because any other distribution of leaves can be transformed into this one in steps of non-decreasing overall numbers of settled nodes. As a feasible lower bound we take a number of 0 settled leaves. The two bounds obtained here are not independent of those given in Equations 4.26 and 4.27. Hence, the total bound on the search-space size we obtain is actually not tight, but it suffices for our purposes. Adding 1 to account for the search-space size $\mathrm{S}_{\mathrm{AF}}(r,r)$, a lower bound on the total penalty that corresponds to intra-cell queries targeting at r is as follows.

$$s_8^L = 6\sum_{z=1}^{\frac{x}{2}}(z-1) + \frac{x}{2} \cdot B_{i_r} + 1 \tag{4.30}$$

The corresponding upper bound on the penalty is obtained by simply inserting the corresponding maximum number of settled leaves, as discussed above.

$$s_8^U = 6\sum_{z=1}^{\frac{x}{2}}(z-1) + \frac{x}{2} \cdot B_{i_r} + B_{i_r} \cdot (B_{i_r} - 1) + 1 \tag{4.31}$$

With $X \in \{U, L\}$ we get the following bounds on the root-induced search-space size by summing up the results from Equations 4.28 and 4.29 and the bounds from Equations 4.30 and 4.31.

$$\sum_{\substack{s=r,\\ t=r}} S(s,t)^X = \sum_{i=1}^{m} s_6[B_i] + s_7 + s_8^X \tag{4.32}$$

We have now reached our goal to find both a valid upper and lower bound on the search-space size of an arbitrary (m, B, x)-tree with a given limb-balanced partition. Taking Equations 4.22, 4.26, 4.27 and 4.32, we get the following bounds S^L and S^U on the search-space size with $X \in \{U, L\}$.

$$S^X = \sum_{\substack{s,t \neq r \\ c(s) \neq c(t)}} S(s,t) + \sum_{\substack{s,t \neq r \\ c(s) = c(t)}} S(s,t)^X + \sum_{\substack{s=r, \\ t=r}} S(s,t)^X \tag{4.33}$$

4.3.4 Proof of Hardness

With the bounds presented above in place, we can now prove hardness of preprocessing the arc-flag algorithm on trees. We work out a polynomial-time reduction from 3-Partition to the problem of deciding whether an instance of ArcFlagsPartition goes below a certain bound by reducing a given instance of 3-Partition to an (m, B, x)-tree. Using the observations made above, we develop bounds on the search-space sizes that correspond to the (m, B, x)-trees of satisfiable and unsatisfiable instances of 3-Partition, respectively. To complete the proof, we then have to show that the corresponding upper bound related to satisfiable instances is always smaller than the lower bound on the search-space size of an unsatisfiable instance.

Theorem 4.11. *The Problem* ArcFlagsPartition *is $\mathcal{NP}$-hard on undirected trees.*

Proof. Given an input instance (S, B, ω) of 3-Partition with $|S| = 3m$, we create an instance (T, k) of ArcFlagsPartition where T is an undirected tree and k denotes the number of allowed cells. First of all, if $m < 33$, we either solve the problem via brute force or fill the input instance with triples $\{0, 0, B\}$ that maintain satisfiability until $m = 33$. Given an instance with $m \geq 36$, we set the number of cells to be $k = m$. The graph T we define as an $(m, B, 4mB^2)$-tree, where $\omega(s_i)$ leaves are attached to the i-th limb of the tree. Since the total number of nodes of such a tree is $1 + 3m \cdot 4mB^2 + mB \in \mathcal{O}\left(m^2B^2\right)$, this reduction can be realized in polynomial time.

We know from Lemma 4.10 that we can limit ourselves to an inspection of partitions with three limbs per cell. We use this knowledge now to obtain an upper bound on the optimal search-space size if the given input instance of 3-Partition is satisfiable. Afterwards, we show that any instance of 3-Partition that is not satisfiable is reduced to a tree such that its optimal search-space size cannot fall below this bound.

For now, assume we are given a satisfiable instance (S, B, ω) of 3-Partition. Then, we know that there exists a partition of $S = \{s_1, \ldots, s_{3m}\}$ into m sets $\{s_{i_1}, s_{i_2}, s_{i_3}\}$ such that $\sum_{j=1}^{3} \omega(s_{i_j}) = B$ for all $i \in \{1, \ldots, m\}$. This implies that we can find a limb-balanced partition of the corresponding graph T of ArcFlagsPartition such that each cell contains exactly three entire limbs and the number of leaves per cell is exactly B. Now, we work out an upper bound S^U_{yes} on the search-space size of such a partition.

Assume we are given a partition of the given tree such that each cell consists of three limbs of a total size of exactly $3x + B$. Then we can derive an upper bound on the optimal search-space size of a tree that corresponds to a satisfiable instance of 3-Partition as follows. First, the inter-cell search-space size given in Equation 4.22 can be bounded.

$$S^U_1 = m\left(n_1[B] \cdot s_1[B] + n_2[B] \cdot s_2[B]\right) \tag{4.34}$$

Analogously, we consider the intra-cell search-space size stated in Equation 4.35, which becomes the following term.

$$S^U_2 = m\left(s_3[B] + s_4[B]^U + s_5[B]\right) \tag{4.35}$$

Finally, the upper bound on the root-induced search-space size in Equation 4.36 yields the term stated below.

$$S_3^U = m \cdot s_6[B] + s_7 + s_8^U \tag{4.36}$$

Using Equations 4.34, 4.35 and 4.36 in Equation 4.33, we obtain an upper bound on the search-space size of an (m, B, x)-tree that corresponds to a satisfiable instance of 3-Partition. We use the bound S_{yes}^U as the input for the decision variant of ArcFlagsPartition.

$$S_{\text{yes}}^U = S_1^U + S_2^U + S_3^U \tag{4.37}$$

What remains to show is that S_{yes}^U is a feasible bound. Obviously, the optimal search-space size of a tree goes below this bound if the corresponding instance of 3-Partition is satisfiable. Next, we cover the trees of unsatisfiable instances. We show that the search-space size obtained in Equation 4.33 yields a convex function in a real-valued parameter B_i. This implies that the search-space size is minimized if the values B_i are balanced. We then derive a lower bound S_{no}^L that corresponds to unsatisfiable instances of 3-Partition and compare it to S_{yes}^U.

To prove the point, we first try to derive a convex and increasing function that assigns costs to each cell C_i of the partition depending on the number B_i of leaves belonging to that cell. Then we can apply Lemma 4.3 to derive lower bounds on the search-space size of an (m, B, x)-tree that corresponds to an unsatisfiable instance of 3-Partition. So, we would like to reformulate the search-space size in order to derive a convex function f in B_i such that the total search-space size equals the sum over all $f(B_i)$ for $1 \le i \le m$. Again, we study the different components of the search space separately. At first, we look at the inter-cell search-space size that we stated in Equation 4.22. Since all we need to show is that the created cost function is convex, we summarize all constant terms, i.e., terms which are independent of B_i. Taking into account that we have $n_2[B_i] = 3x + B_i$ in Equation 4.21, we distinguish the following components of the inter-cell search space size.

$$\begin{aligned}\sum_{\substack{s,t\neq r\\ c(s)\neq c(t)}} S(s,t) &= \sum_{i=1}^{m} \left(n_1[B_i]\cdot s_1[B_i] + n_2[B_i]\cdot s_2[B_i]\right)\\ &= \underbrace{\sum_{i=1}^{m} n_1[B_i]\cdot s_1[B_i]}_{\sigma_1} + \underbrace{\sum_{i=1}^{m} 3x\cdot s_2[B_i]}_{\sigma_2} + \underbrace{\sum_{i=1}^{m} B_i\cdot s_2[B_i])}_{\sigma_3}\end{aligned} \tag{4.38}$$

First, we consider the search-space size σ_1 according to Equation 4.38, which is examined below.

$$\begin{aligned}\sigma_1 &= \sum_{i=1}^{m} n_1[B_i]\cdot s_1[B_i]\\ &= \sum_{i=1}^{m}\left(((3m-3)x + mB - B_i)\left(3\frac{x(x+1)}{2} + B_i(x+1)\right)\right)\\ &= \sum_{i=1}^{m}\left(-(x+1)B_i^2 + \left((3m-3)x(x+1) + mB(x+1) - 3\frac{x(x+1)}{2}\right)B_i + \text{const}\right)\end{aligned}$$

The next step is to inspect the search-space size σ_2, which again can be expressed as a sum of polynomials in B_i in a straightforward manner.

$$\begin{aligned}
\sigma_2 &= \sum_{i=1}^{m} 3x \cdot \sum_{j\neq i} \left(\frac{(3x+B_j+1)(3x+B_j+2)}{2} - 1\right)\\
&= \sum_{i=1}^{m} 3x \cdot \sum_{j\neq i} \left(\frac{1}{2}B_j^2 + \left(3x+\frac{3}{2}\right)B_j + \underbrace{\frac{9}{2}x^2+\frac{9}{2}x}_{\text{const}}\right)\\
&= (m-1)\sum_{i=1}^{m}\left(\frac{3}{2}xB_i^2 + \left(9x^2+\frac{9}{2}x\right)B_i + \text{const}\right)
\end{aligned}$$

Our final aim is to achieve an analogous formulation for the term σ_3 according to Equation 4.38. This task is approached in the following.

$$\begin{aligned}
\sigma_3 &= \sum_{i=1}^{m} B_i \cdot \sum_{j\neq i}\left(\frac{(3x+B_j+1)(3x+B_j+2)}{2} - 1\right)\\
&= \sum_{i=1}^{m} B_i \cdot \sum_{j\neq i}\left(\frac{1}{2}B_j^2 + \left(3x+\frac{3}{2}\right)B_j + \frac{9}{2}x^2 + \frac{9}{2}x\right)\\
&= \frac{1}{2}\sum_{i=1}^{m} B_i \sum_{j\neq i} B_j^2 + \left(3x+\frac{3}{2}\right)\sum_{i=1}^{m} B_i \sum_{j\neq i} B_j + \sum_{i=1}^{m} B_i \sum_{j\neq i}\left(\frac{9}{2}x^2+\frac{9}{2}x\right)\\
&= \frac{1}{2}\sum_{i=1}^{m} B_i^2 \sum_{j\neq i} B_j + \left(3x+\frac{3}{2}\right)\sum_{i=1}^{m} B_i \sum_{j\neq i} B_j + (m-1)\sum_{i=1}^{m} B_i\left(\frac{9}{2}x^2+\frac{9}{2}x\right)\\
&= \frac{1}{2}\sum_{i=1}^{m} B_i^2(mB-B_i) + \left(3x+\frac{3}{2}\right)\sum_{i=1}^{m} B_i(mB-B_i) + \sum_{i=1}^{m} B_i(m-1)\left(\frac{9}{2}x^2+\frac{9}{2}x\right)\\
&= \sum_{i=1}^{m}\left(-\frac{B_i^3}{2} + \left(\frac{mB}{2} - 3x - \frac{3}{2}\right)B_i^2 + \left(\left(3x+\frac{3}{2}\right)mB + (m-1)\left(\frac{9}{2}x^2+\frac{9}{2}x\right)\right)B_i\right)
\end{aligned}$$

Next, we turn to the intra-cell search space and proceed with an analogous approach. Remember that we want to derive a lower bound and therefore consider the search-space size in Equation 4.26. Again, we split the work for convenience and examine the following two components.

$$\sum_{\substack{s,t\neq r\\ c(s)=c(t)}} S(s,t)^L = \underbrace{\sum_{i=1}^{m}\frac{(3x+B_i)^2(3x+B_i+1)}{2}}_{\sigma_4} + \underbrace{\sum_{i=1}^{m}\left(\frac{27}{4}x^2+\frac{9}{2}xB_i\right)}_{\sigma_5} \tag{4.39}$$

We start with the search-space size σ_4 in Equation 4.39, which can simply be transformed in the following way. Again, we obtain a sum of polynomials in the values B_i.

$$\begin{aligned}
\sigma_4 &= \sum_{i=1}^{m}\frac{(3x+B_i)^2\,(3x+B_i+1)}{2}\\
&= \sum_{i=1}^{m}\left(\frac{1}{2}B_i^3 + \left(\frac{9}{2}x+\frac{1}{2}\right)B_i^2 + \left(\frac{27}{2}x^2+3x\right)B_i + \underbrace{\frac{27}{2}x^3+\frac{9}{2}x^2}_{\text{const}}\right)
\end{aligned}$$

Clearly, the term σ_5 already has the desired form and thus requires no further transformations. Finally, we have to examine the search-space size that is caused by queries that include the root node r as source or target node. We investigate each of the following components of the lower bound presented in Equation 4.32 separately.

$$\sum_{\substack{s=r,\\t=r}} S(s,t)^L = \underbrace{\sum_{i=1}^{m} s_6[B_i]}_{\sigma_6} + \underbrace{s_7 + 6\sum_{z=1}^{\frac{x}{2}}(z-1) + 1}_{\sigma_7} + \frac{xB_{i_r}}{2} \tag{4.40}$$

We reformulate σ_6 given in Equation 4.40 and obtain its desired form after a few straightforward transformations.

$$\begin{aligned}\sigma_6 &= \sum_{i=1}^{m}\left(\frac{(3x+B_i+1)(3x+B_i+2)}{2} - 1\right)\\ &= \sum_{i=1}^{m}\left(\frac{1}{2}B_i^2 + \left(3x+\frac{3}{2}\right)B_i + \underbrace{\frac{9}{2}\left(x^2+x\right)}_{\text{const}}\right)\end{aligned}$$

The addend σ_7 contains the parts of s_7 and s_8 that do not contain the value B_i, as can be seen in Equations 4.29 and 4.30. We can thus treat σ_7 as a constant. Furthermore, let us ignore the remaining term $x \cdot B_{i_r}/2$ of Equation 4.40 for now.

We have seen that each of the sums σ_1 to σ_6 presented in Equations 4.38, 4.39 and 4.40 can be reformulated as a sum that iterates over the m values B_i. Thus, we can derive a cost function $f(B_i)$ for each B_i by taking the corresponding addend from σ_z for $z \in \{1\dots6\}$. To account for the term σ_7 we finally add the constant σ_7/m to f. In this way we equally divide the corresponding search-space size over all B_i. We now show that the cost function f is increasing and convex. Afterwards, we apply Lemma 4.3 which tells us that the total search-space size increases in the differences of all B_i from the constant value B.

To see that f is monotonically increasing and convex, we assume for now that the parameters of f can have any real in $\mathbb{R}^{\geq 0}$. We show that f is a polynomial of the form

$$f(B_i) = aB_i^2 + bB_i + c$$

with $a > 0$ and $b > 0$, which implies the desired properties of f. First of all, note that there are exactly two terms in all σ_z given in Equations 4.38, 4.39 and 4.40 that contain a factor B_i^k with $k \geq 3$. Those are the terms $-B_i^3/2$ in σ_3 and $B_i^3/2$ in σ_4, which eliminate each other. Thus f contains only variables B_i to the power at most two. Next, we prove that $a > 0$ by summing up all terms that have a factor B_i^2 in any of the equations stated above. This yields the following value for a in the cost function f.

$$\begin{aligned}a &= \frac{3}{2}(m-1)x + \frac{9}{2}x + \frac{1}{2} + \frac{1}{2} + \frac{1}{2}mB - (x+1) - 3x - \frac{3}{2}\\ &= \frac{1}{2}mB + \frac{3}{2}mx - x - \frac{3}{2}\end{aligned}$$

We can safely assume that m and B are strictly positive, which implies that we always have $a > 0$. If we sum up the terms for b in the same way, we see that there is only one negative term $-3x(x+1)/2 = -3x^2/2 - 3x/2$ that occurs in σ_1. We simply add $27x^2/2 + 3x$ given in σ_4 to this term and obtain a positive result. Having in mind that $m, B \geq 1$, it is clear that all other terms that occur in b must be non-negative and we get $b > 0$.

Given $a > 0$ and $b > 0$, we immediately conclude that the cost function f is convex and increasing on $\mathbb{R}^{\geq 0}$. Furthermore, we have seen that f covers most of the search-space size of a certain cell of the (m, B, x)-tree. However, we left out the search-space size of the term $x \cdot B_{i_r}/2$ stated in Equation 4.40. The value $x \cdot B_{i_r}/2$ depends only on B_{i_r}, and hence we cannot distribute it over all cells. However, Lemma 4.3 tells us that as long as we only take f into account, the search-space size is non-decreasing if we increment $|B_i - B|$ for any $1 \le i \le m$. Since we look for a lower bound S^L_{no} of the search-space size of trees that correspond to unsatisfiable instances of 3-Partition, we are interested in the case when there exists no partition such that each cell has exactly B leaves. Obviously, the best case according to f then occurs if there exists a partition into cells such that that there are cells C_j, C_k with $B_j = B - 1$, $B_k = B + 1$ and $B_i = B$ for all other cells $C_i \neq C_j, C_k$.

With this information in mind, we account for the term $x \cdot B_{i_r}/2$ that occurs in Equation 4.40 and has not yet been dealt with. Since we want to minimize the search-space size, this last component is optimized if the cell that contains the root node has as few leaves as possible. Thus, we set $B_{i_r} = B - 1$ which clearly is the best solution under the constraint of minimizing the distances $|B - B_i|$. Proving that further reducing the size of the cell that contains r is not helpful is postponed for now. Instead, we first provide a lower bound on the search-space size of a partition with $B_{i_r} = B - 1$, $B_j = B + 1$ and $B_i = B$ for all $i \neq i_r, j$. To complete the proof, we show thereafter that this partition indeed yields the best case of an unsatisfiable input instance. In other words, further reducing the size of the cell that contains the root node cannot break the lower bound we are going to infer next.

In Equation 4.37, we derived an upper bound S^U_{yes} on the optimal search-space size if the corresponding instance of 3-Partition is satisfiable. Now, we examine the lower bound S^L_{no} on a partition of an instance if there are cells C_j, C_k with $B_j = B - 1$, $B_k = B + 1$ and $B_i = B$ for all $C_i \neq C_j, C_k$. We also show that $S^L_{\text{no}} - S^U_{\text{yes}} > 0$. By proving that a modification of the term $x \cdot B_{i_r}/2$ in Equation 4.40 cannot further reduce the search-space size, we eventually confirm that S^L_{no} is indeed a feasible lower bound.

Again, we distinguish the different components of the search space of a limb-balanced partition of an (m, B, x)-tree. Our goal is to obtain components S^L_1, S^L_2, S^L_3 with $S^L_{\text{no}} = S^L_1 + S^L_2 + S^L_3$ along the lines of the analysis of S^U_{yes} given by Equation 4.37. Without loss of generality, let $B_1 = B - 1$, $B_2 = B + 1$ and $B_i = B$ for all $3 \le i \le m$. First, we inspect the inter-cell search-space size of the corresponding partition. In the subsequent equations, we shall make use of the abbreviation

$$\phi[B_i] = \frac{(3x + B_i + 1)(3x + B_i + 2)}{2} - 1.$$

Note that $\phi[B_i]$ corresponds to the addends of $s_2[B_i]$ in Equation 4.20. In our partition, we have $m - 2$ cells with B leaves. These account for the following search-space size.

$$S^L_{11} = (m-2)\left(n_1[B] \cdot s_1[B] + n_2[B] \cdot ((m-3) \cdot \phi[B] + \phi[B+1] + \phi[B-1])\right)$$

Moreover, the partition contains one cell that has exactly $B + 1$ leaves. The inter-cell search-space size caused by this cell is as follows.

$$S^L_{12} = (\underbrace{n_1[B+1] \cdot s_1[B+1]}_{\lambda^L_{12}} + \underbrace{n_2[B+1]\left((m-2) \cdot \phi[B] + \phi[B-1]\right)}_{\mu^L_{12}}$$

On the other hand, we have to account for the search-space size induced by the last remaining cell that holds $B-1$ leaves. This search-space size is constituted in the subsequent equation.

$$S^L_{13} = \underbrace{n_1[B-1] \cdot s_1[B-1]}_{\lambda^L_{13}} + \underbrace{n_2[B-1]\left((m-2) \cdot \phi[B] + \phi[B+1]\right)}_{\mu^L_{13}}$$

Clearly, S_{11}^L summarizes the inter-cell search-space size that is caused by $m-2$ cells, while S_{12}^L and S_{13}^L are responsible for one cell each. Altogether, S_{11}^L, S_{12}^L and S_{13}^L summarize the inter-cell search-space size of the given partition, which we denote by $S_1^L = S_{11}^L + S_{12}^L + S_{13}^L$.

Analogously, we split S_1^U given in Equation 4.34 into a component

$$S_{11}^U = (m-2)\left(n_1[B] \cdot s_1[B] + n_2[B] \cdot s_2[B]\right)$$

that accounts for the inter-cell search-space size produced by $m-2$ cells and the following two components that cover one of the remaining cells each.

$$S_{12}^U = S_{13}^U = \underbrace{n_1[B] \cdot s_1[B]}_{\lambda^U} + \underbrace{n_2[B] \cdot s_2[B]}_{\mu^U}$$

Clearly, we have $S_1^U = S_{11}^U + S_{12}^U + S_{13}^U$. To examine the difference of the inter-cell search-space sizes S_1^L and S_1^U, we separately compare the three addends we divided them into. At first, consider the search-space sizes that correspond to $m-2$ cells holding B leaves each. The following difference is obtained after a few elementary steps.

$$\begin{aligned} S_{11}^L - S_{11}^U &= (m-2)(3x+B)\underbrace{(\phi[B+1] + \phi[B-1] - 2\phi[B])}_{=1} \\ &= (m-2)(3x+B) \end{aligned} \tag{4.41}$$

The next step consists of the comparison of the component S_{12}^L that represents the cell with $B+1$ leaves to S_{12}^U. For the sake of transparency and legibility, we first restrict ourselves to the terms λ_{12}^L and λ^U as denoted above. If we consider the definitions of n_1 and s_1 in Equations 4.18 and 4.19, it is clear that incrementing B to $B+1$ adds $(3m-3)x(x+1)$ and subtracts $3x(x+1)/2$ from the corresponding search-space size. Furthermore, we have to add a difference that is the result of the multiplication of $B_i(x+1)$ by $(mB - B_i)$ for $B_i = B-1$ and $B_i = B$, respectively. The obtained result is

$$((m-1)B-1)(B+1)(x+1) - (m-1)B^2(x+1) = mBx + mB - 2Bx - 2B - x - 1.$$

Summarizing these observations, the difference between λ_{12}^L and λ^U is derived in the following term.

$$\lambda_{12}^L - \lambda^U = (3m-3)x(x+1) - \frac{3x(x+1)}{2} + mBx + mB - 2Bx - 2B - x - 1 \tag{4.42}$$

The difference between the terms μ_{12}^L and μ^U turns out to be as below after a few basic transformations.

$$\begin{aligned} \mu_{12}^L - \mu^U &= (m-2) \cdot \phi[B] + (3x+B+1) \cdot \phi[B-1] - (3x+B) \cdot \phi[B] \\ &= -\frac{9}{2}x^2 - 3xB - \frac{3}{2}x - \frac{1}{2}B^2 - \frac{1}{2}B - 1 \end{aligned} \tag{4.43}$$

This completes the change in the search-space size that is expressed by $S_{12}^L - S_{12}^U$. Finally, we have to examine the search-space size of the last remaining cell, which contains $B-1$ leaves. Just as we did before, we compare λ_{13}^L and λ^U first. Along the lines of the approach that lead to Equation 4.42, we obtain the following result.

$$\lambda_{13}^L - \lambda^U = -(3m-3)x(x+1) + \frac{3x(x+1)}{2} - mBx - mB + 2Bx + 2B - x - 1$$

Inspecting the difference between μ_{13}^L and μ^U, we get the result shown below. This completes the examination of the difference $S_{13}^L - S_{13}^U$.

$$\begin{aligned} \mu_{13}^L - \mu^U &= (m-2) \cdot \phi[B] + (3x+B-1) \cdot \phi[B+1] - (3x+B) \cdot \phi[B] \\ &= \frac{9}{2}x^2 + 3xB - \frac{3}{2}x + \frac{1}{2}B^2 - \frac{1}{2}B - 2 \end{aligned} \tag{4.44}$$

The overall difference encountered so far accounts for the bounds on the search-space size of inter-cell queries. We summarize the results provided by Equations 4.41 to 4.44 and obtain the following change in the inter-cell search-space size.

$$\begin{aligned}\Delta_1 &= S_1^L - S_1^U \\ &= S_{11}^L - S_{11}^U + \lambda_{12}^L - \lambda^U + \mu_{12}^L - \mu^U + \lambda_{13}^L - \lambda^U + \mu_{13}^L - \mu^U \\ &= 3mx + mB - 11x - 3B - 5 \qquad (4.45)\end{aligned}$$

For the intra-cell search space, we again study the cells of different sizes consecutively. The intra-cell search spaces of $m-2$ cells with B attached leaves are unaffected by the cell sizes of other cells, but we must compare the lower bound to the upper bound on the corresponding search-space size. Hence, we obtain the following difference according to the intra-cell search-space size of the $m-2$ cells with B leaves.

$$S_{21}^L - S_{21}^U = -2(m-2)B\left\lceil\frac{B}{3}\right\rceil \qquad (4.46)$$

Along the lines of the above discussion, let S_{22}^L denote the lower bound on the intra-cell search-space size induced by the cell with $B+1$ leaves and S_{22}^U the upper bound on the intra-cell search-space size of a cell with B leaves. Clearly, we obtain the following difference due to Equations 4.26 and 4.27.

$$\begin{aligned}S_{22}^L - s_{22}^U &= s_3[B+1] + s_4^L[B+1] + s_5[B+1] - s_3[B] - s_4^U[B] - s_5[B] \\ &= \frac{(3x+B+1)^2(3x+B+2)}{2} - \frac{(3x+B)^2(3x+B+1)}{2} + \frac{9}{2}x - 2B\left\lceil\frac{B}{3}\right\rceil \\ &= \frac{27}{2}x^2 + 9xB + 12x + \frac{3}{2}B^2 + \frac{5}{2}B + 1 - 2B\left\lceil\frac{B}{3}\right\rceil \qquad (4.47)\end{aligned}$$

Next, we turn to the intra-cell search-space size of the remaining cell with $B-1$ leaves. Denoting by S_{23}^L the corresponding search-space size and by S_{23}^U the upper bound for a single cell with B leaves, we get the difference shown below.

$$\begin{aligned}S_{23}^L - S_{23}^U &= s_3[B-1] + s_4^L[B-1] + s_5[B-1] - s_3[B] - s_4^U[B] - s_5[B] \\ &= \frac{(3x+B-1)^2(3x+B)}{2} - \frac{(3x+B)^2(3x+B+1)}{2} - \frac{9}{2}x - 2B\left\lceil\frac{B}{3}\right\rceil \\ &= -\frac{27}{2}x^2 - 9xB - 3x - \frac{3}{2}B^2 + \frac{1}{2}B - 2B\left\lceil\frac{B}{3}\right\rceil \qquad (4.48)\end{aligned}$$

We have provided a lower bound on the intra-cell search-space size $S_2^L = S_{21}^L + S_{22}^L + S_{23}$ that represents the best case corresponding to a reduction from an unsatisfiable instance of 3-Partition. Furthermore, we have the bound $S_2^U = S_{21}^U + S_{22}^U + S_{23}^U$ for the case of B leaves per cell. According to Equations 4.46, 4.47 and 4.48, we get a lower bound on the total change in the intra-cell search-space size that sums up to the following term.

$$\begin{aligned}\Delta_2 &= S_2^L - S_2^U \\ &= S_{21}^L - S_{21}^U + S_{22}^L - S_{22}^U + S_{23}^L - S_{23}^U \\ &= 9x + 3B + 1 - 2mB\left\lceil\frac{B}{3}\right\rceil \qquad (4.49)\end{aligned}$$

Ultimately, we turn to the search-space size that is induced by the root node r. Remember that we assigned r to the cell with the fewest leaves to minimize the overall search-space size. Furthermore, we have $s_6[B_i] = \phi[B_i]$ according to Equation 4.28. We would like to compare the bound S_3^U as given in Equation 4.36 to a lower bound S_3^L for the case of an unsatisfiable instance of 3-Partition. Let S^L_{31} and S^U_{31} denote the search-space sizes according to s_6 given in Equation 4.28. We obtain the difference stated below.

$$\begin{aligned} S^L_{31} - S^U_{31} &= \phi[B+1] + \phi[B-1] - 2\phi[B] \\ &= 1 \end{aligned} \tag{4.50}$$

Moreover, we have to take account of the fact that the cell containing the root node holds $B-1$ and B leaves, respectively. According to Equations 4.30 and 4.31, we get the following difference.

$$\begin{aligned} S^L_{32} - S^U_{32} &= \frac{x}{2}(B-1) - \frac{x}{2}B - B(B-1) \\ &= -\frac{x}{2} - B^2 + B \end{aligned} \tag{4.51}$$

Note that the term s_7 in Equation 4.32 does not depend on the values B_i, hence we can omit it from our considerations. The results stated in Equations 4.50 and 4.51 then yield the difference in the bounds according to queries that start or end at the root node r.

$$\begin{aligned} \Delta_3 &= S^L_3 - S^U_3 \\ &= S^L_{31} - S^U_{31} + S^L_{32} - S^U_{32} \\ &= 1 - \frac{x}{2} - B^2 + B \end{aligned} \tag{4.52}$$

We have now gathered enough information to study the difference between the considered bounds S^U_{yes} and S^L_{no}. Taking Equations 4.45, 4.49 and 4.52, we show that the obtained difference is always non-negative. Recall that we have $x = 4mB^2$ and that we can safely assume $m \geq 2$ and $B \geq 1$.

$$\begin{aligned} \Delta^L_U &= \Delta_1 + \Delta_2 + \Delta_3 \\ &= \underbrace{\left(3m - \frac{5}{2}\right) x}_{\geq 3x} + \underbrace{(m+1)B}_{\geq 3} - 3 - \underbrace{2mB \left\lceil \frac{B}{3} \right\rceil}_{\leq x} - B^2 \\ &\geq 2x - B^2 \\ &\geq x \end{aligned}$$

This proves that the difference between the provided lower bound S^L_{no} for instances of (m, B, x)-trees corresponding to unsatisfiable instances of 3-Partition and the upper bound S^U_{yes} for satisfiable instances is always positive.

The last matter we have to take care of is the question that was postponed when considering the term $x \cdot B_{i_r}/2$ in Equation 4.40. We must check whether reducing the size of the cell that contains r can improve the lower bound. To see that this is not the case, recall that the cost function f we stated above is convex. Lowering the number of leaves of the corresponding cell to $B-2$ implies that on other hand we have to increase another cell size by one. But this implies an additional increase of the search-space size that is at least Δ because f is convex. Furthermore, the term in question is $B_{i_r} \cdot x/2$, so reducing the cell size by 1 reduces our bound by $x/2$. Since we have $\Delta \geq x > x/2$, the upper bound S^U_{yes} clearly cannot be broken this way.

We have seen that every input instance of 3-Partition corresponds to an (m, B, x)-tree T such that its optimal search-space size $\mathrm{S}_{\mathrm{AF}}(T, \mathcal{C}_{\text{opt}})$ is below or equal S^U_{yes} if and only if the input instance is satisfiable. This completes the proof. □

4.4 Other Graph Classes

We close this chapter with short reviews of further restricted classes of graphs. Instead of formal studies, we provide rough ideas of how to approach the investigation of these graphs. The proofs given in Sections 4.2 and 4.3 supply tools to formally complete most tasks, which we omit here.

4.4.1 Directed Paths and Trees

In the previous sections we only studied the undirected versions of paths and trees. We now provide arguments for obtaining similar results on their directed counterparts. We briefly show how to adapt the proof given in Section 4.2 to cover directed paths, before we study the problem of finding optimal cells on rooted directed trees.

Directed Paths

First of all, we examine directed paths, i.e., graphs that are of the form $P = (V, E, \omega)$ with $V = \{v_1, \dots, v_n\}$ and $E = \{(v_i, v_{i+1}) \mid 1 \leq i < n\}$. It is easy to see that the study of undirected paths presented in Section 4.2 directly applies to directed paths. However, since we only defined penalties for strongly connected graphs, we need to extend Definition 4.5 to cover our case. Therefore, we say that the penalty of a query where the target is unreachable equals the search-space size of that query decreased by 1. This way we account for the source node, which must be settled in all cases. Interestingly enough, the obtained penalties remain exactly the same in the directed case when using this definition. In what follows, we give a sketch of the formal proof.

Theorem 4.12. *Let $P = (V, E, \omega)$ be a directed path as specified above and k a positive integer. Assume that starting at v_1, nodes are assigned to cells C_i, $1 \leq i \leq k$, in ascending order such that $|C_i| = \lceil n/k \rceil$ for $1 \leq i \leq n \bmod k$ and $|C_i| = \lfloor n/k \rfloor$ for $n \bmod k < i \leq k$. The partition $\mathcal{C} = \{C_1, \dots, C_k\}$ yields a search-space optimal partition if the number of cells is limited by k.*

Proofsketch: The only change we have to cope with in comparison to undirected paths is the difference in queries from a node v_i to a node v_j where $i > j$. Since the target node is unreachable in this case, the query algorithm starts at v_i and settles all successive nodes until it encounters an edge that does not have the target flag set. To prove the theorem, we proceed along the lines of Section 4.2.

First, it is easy to see that we can use the same procedure as in the proof of Lemma 4.6 to convert an arbitrary partition into one that only has connected cells, without increasing the total search-space size. Inspecting the resulting partition, we find that inter-cell queries again cause a total penalty of 0, because either the exact path is settled or no outgoing edge has the target flag set. Intra-cell query penalties cannot increase in comparison to the original partition, for either the target gets settled or the query settles all nodes up to the rightmost node of the cell. The total penalty accounting for the latter case clearly is minimized if cells are connected.

As the next step, we consider the weight function ω. Since the next node to be taken from the queue is always unique, edge weights can be ignored. This covers the result obtained in Lemma 4.7.

Finally, we have to minimize the overall penalty, just as we did for undirected paths in the proof of Theorem 4.8. The only penalties found are those of intra-cell queries where the target nodes are unreachable. Imagine a node v at a relative position i in an arbitrary cell C, where $1 \leq i \leq |C|$. There are $i - 1$ distinct intra-cell nodes that are unreachable from v. A query from v to any of these nodes then causes all reachable nodes in C to be settled, which induces a penalty of $|C| - i$. This yields a total penalty as shown below.

$$\begin{aligned}
\sum_{s,t} \operatorname{pen}_{\mathcal{C}}(s,t) &= \sum_{C \in \mathcal{C}} \sum_{i=1}^{|C|-1} (i-1) \cdot (|C| - i) \\
&= \sum_{C \in \mathcal{C}} \left(|C| \cdot \sum_{i=1}^{|C|-1} i - \sum_{i=1}^{|C|-1} i^2 - \sum_{i=1}^{|C|-1} |C| + \sum_{i=1}^{|C|-1} i \right) \\
&= \sum_{C \in \mathcal{C}} \left((|C|+1) \frac{(|C|-1)\,|C|}{2} - \frac{1}{6} (|C|-1)\,|C|\,(2\,|C|-1) - |C|\,(|C|-1) \right) \\
&= \sum_{C \in \mathcal{C}} \left(\frac{1}{6} |C|^3 - \frac{1}{2} |C|^2 + \frac{1}{3} |C| \right)
\end{aligned}$$

This is the same result as we obtained in Equation 4.2 of Theorem 4.8. Hence, we get similar optimal partitions for directed paths. □

Directed Trees

We conjecture that one could modify the proof of hardness for undirected trees in order to cover directed trees. Many arguments used throughout the proof presented in Section 4.3 would directly carry over to the directed case. As a difference to the original analysis, queries to unreachable nodes now imply settling all succeeding nodes within a limb as long as the corresponding flags are set to `true`. A reasonable assumption would be that the best partition again puts together triples of limbs. Then it appears obvious that the search-space size again falls below a certain threshold if and only if the corresponding instance of 3-PARTITION is satisfiable. However, a detailed analysis would be necessary to formally adapt the proof to the directed case. Since this would exceed the scope of this work, we only propose the following conjecture.

Conjecture 4.13. *The Problem* ARCFLAGSPARTITION *is $\mathcal{NP}$-hard on rooted directed trees.*

Moreover, this result would immediately imply the hardness of ARCFLAGSPARTITION on directed acyclic graphs. Since directed acyclic graphs occur in the form of time-expanded graphs in time-dependent scenarios [PSWZ07], a proof of Conjecture 4.13 would be desirable.

4.4.2 Cycles

Another interesting restricted class of graphs are cycles. One may think of a directed cycle to be generated from a directed path P by the addition of a single edge (v_n, v_1). In the undirected case, the edge (v_1, v_n) is added as well. Somewhat surprising at first glance, this simple modification has a large impact on the structure of optimal cells for the underlying graphs.

Undirected Cycles

In what follows, we first explain why optimal cells on undirected cycles may substantially differ from those on undirected paths. Afterwards, we propose basic ideas for an approach that computes optimal partitions of arbitrary cycles in polynomial time.

Theorem 4.8 proposes a simple way to obtain optimal partitions for undirected paths by using strongly connected cells of balanced size. For several reasons, the proof of this theorem does not carry over to undirected cycles. Formally, an undirected cycle is a graph $Z = (V, E, \omega)$ with a set of nodes $V = \{v_1, \ldots, v_n\}$ and a set of edges $E = \{(v_i, v_{i+1}) \mid 0 < i < n\} \cup \{(v_n, v_1), (v_1, v_n)\}$. To see why we cannot proceed along the lines of Section 4.2 to obtain optimal cells for cycles, we point out the vital differences when concerning undirected cycles.

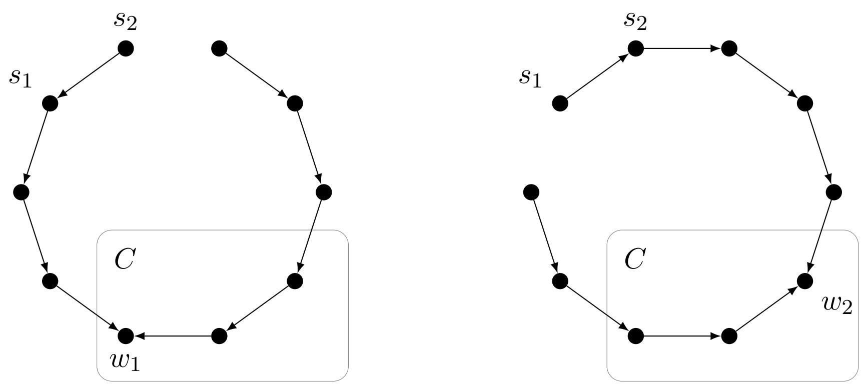

Figure 4.11: Backward-shortest-path trees of two nodes u and v in an undirected cycle.

Imagine a backward-shortest-path tree of an arbitrary node $w \in V$. Observe that every such tree must contain either the edge (u, v) or (v, u) for all but one undirected edge in E. In general, the edge that is left out differs for the trees of most nodes. Figure 4.11 depicts exemplary backward-shortest-path trees of two nodes w_1 and w_2. Imagine w_1 and w_2 belong to the same cell C. In this case, in queries starting at either s_1 or s_2, there are two distinct paths leading into the target cell C with set flags, despite the fact that C is strongly connected. Recall that this was not the case for paths, which is why penalties of inter-cell queries were always 0 in case of a strongly connected target cell. The same effect can even force intra-cell queries to leave a strongly connected cell and settle nodes outside of it. Additionally, since in contrast to the situation for paths, backward-shortest-path trees now depend on the edge weights, we can no longer ignore these weights as we did in case of paths. As a result of these observations, there may even be unique optimal partitions with unbalanced cell sizes. Figure 4.12 shows an example. Note that this partition is purely induced by the edge weights. It prevents queries to nodes that lie beyond one of the heavy edges from settling the whole cycle.

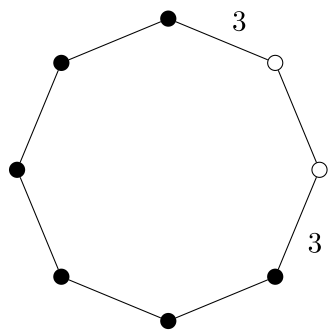

Figure 4.12: An optimal partition of a cycle with two cells. Edge weights are assumed to be 1 unless otherwise labeled.

Presuming that for every undirected cycle, there exists a partition such that all cells are strongly connected, we can provide an algorithm that finds optimal partitions for arbitrary undirected cycles. Although it seems reasonable to assume that search-space optimal cells on undirected cycles must be strongly connected, a prove of this conjecture is yet to be found.

Below, we describe the work of an algorithm based on dynamic programming that assigns optimal strongly connected cells to arbitrary undirected cycles. For cells are strongly connected, we can define a valid partition by naming k cell borders. Cells of the partition are than assigned by merging nodes between a distinct pair of borders into one cell. In what follows, we design an algorithm that outputs k edges of the input cycle, each of which represents a border between two cells.

Assume that the set of edges of the input graph $Z = (V, E, \omega)$ is $E = \{e_1, \dots, e_n\}$, where e_i is an undirected edge. Note that we abuse notation here, because due to Chapter 2, we defined undirected graphs to hold directed edges as well. Formally speaking, e_i represents a set of two edges with alternate source and target node. Our objective is to find a set $E_{\text{opt}} = \{e_{i_1}, \dots, e_{i_k}\} \subseteq E$ of k edges such that these edges represent borders of an optimal partition. For now, imagine the first of k edges to be e_1 and that it actually belongs to E_{opt}. To determine the $k-1$ remaining edges, we proceed as follows. We maintain a table of $k-1$ rows and n columns. For $1 \leq i \leq n$ and $2 \leq j \leq k$, the i-th column of the j-th row is supposed to store information that encodes the best possible partition considering only the edges $e_1, \dots, e_i$ and separating the cycle into j cells. Each table entry consists of an integer that holds the optimal search-space size corresponding to this entry and a pointer to the previous border of the determined partition.

Starting at e_1, we fill the first row of the table as follows. We iteratively process the remaining edges $e_2, \dots, e_n$ of the graph. In each step, e_1 together with the involved edge e_i uniquely determines a tentative cell C. We set the cost of this cell to

$$S_C = \sum_{s \in V, t \in C} \mathrm{S}_{\mathrm{AF}}(s, t). \tag{4.53}$$

To determine S_C, we must compute correct arc-flags for C and execute the arc-flags algorithm for all necessary pairs of nodes, both of which can be done in polynomial time. Moreover, S_C is exactly the search-space size induced by picking border edges e_1 and e_i. The retrieved value of S_C is stored in the i-th column of the first row of T.

The remaining rows of the table are then computed using the following scheme. Given values of edges $e_1, \dots, e_{i-1}$ of the j-th row, the next edge e_i is processed as follows. As we did for the first row, edge e_i is temporarily added to the partition as a border. To obtain the optimal search-space size, one iteratively checks the search-space sizes stored in columns $1 \leq x \leq i-1$ of the $j-1$-th row. To the value of the x-th columns we add the search-space size induced by the cell with borders e_x and e_j, similar to Equation 4.53. This yields the optimal search-space size when setting the borders of the j-th partition to be e_x and e_j. The best of all $i-1$ computed values is finally stored in column i. It determines the best possible partition that covers all nodes between the edges e_1 and e_i when using j cells. Doing this for all table entries eventually gets us the optimal solution in the last column of the last row.

To obtain the actual partition, in each table entry one simply stores the index of the previous border. Starting at the last row of the last column, one retrieves the next border edge by reading the corresponding index i. The next edge can then be found in the i-th column of the $n-1$-th row, and so on.

Recall that we did not specify how to determine the initial edge e_1. However, we can simply perform n runs of the procedure, each time picking a distinct edge to start with. Returning the best of all search-space sizes then clearly gets us the optimal value.

The description given above is only supposed to give a rough idea of an algorithm to generate search-space optimal partitions for undirected cycles. A formal specification and a proof of correctness are left as future work.

Directed Cycles

A directed cycle is a graph $Z = (V, E, \omega)$, where we have $V = \{v_1, \ldots, v_n\}$ and $E = \{(v_i, v_{i+1}) \mid 1 \leq i < n\} \cup \{(v_n, v_1)\}$. The situation for directed cycles is quite simple. Imagine a query of Dijkstra's algorithm on a directed cycle that implements the stopping criterion but makes no use of arc-flags at all. On an arbitrary query, the algorithm settles exactly the unique path between the given nodes s and t. Hence, the total penalty on a directed cycle is always 0 and every valid partition is optimal.

Corollary 4.14. *Let $Z = (V, E, \omega)$ be a directed cycle and $\mathcal{C} = \{C_1, \ldots, C_k\}$ a partition of Z. $\mathcal{C}$ is search-space optimal for Z.*

4.4.3 Towards the Border of Tractability

Before, we realized that optimal cells are efficiently computable for several restricted graph classes such as paths and directed cycles. On the other hand, we showed that the same problem is $\mathcal{NP}$-hard for undirected trees, which in case of $\mathcal{P} \neq \mathcal{NP}$ proves the non-existence of a polynomial-time algorithm for such graphs in general. A natural question that arises is whether one can further restrict the class of undirected trees to obtain a class for which there exists an efficient procedure that assigns optimal cells to an arbitrary input instance.

Restricted Classes of Trees

As an example, we consider undirected trees in which each node has a degree that is bounded by a constant. If this constant is 2, we have exactly the class of undirected paths. Unfortunately, we conjecture that the problem ArcFlagsPartition becomes $\mathcal{NP}$-hard already if we increment this bound to 3. Although the reduction presented in Section 4.3 creates trees with an unbounded maximum degree, this does not seem to be inherently necessary to prove Theorem 4.11. There are two locations in an (m, B, x)-tree where nodes of a degrees larger than 3 may occur. Namely, we have the root node with a degree of $3m$ and the outermost node of the chain of each limb with a maximum degree of B. However, in both cases we can eliminate this problem by inserting more complex subgraphs into the (m, B, x)-tree such that all nodes have a degree bounded by 3. Figure 4.13 depicts a proposal for such a modified (m, B, x)-tree. The root node r is replaced by a chain of nodes, each of which has one limb attached to it. Each leaf of a tree is replaced by two nodes. If an original limb has two or more leaves, the corresponding pairs of nodes are chained. The tree depicted here shows the corresponding modification of the (m, B, x)-tree depicted in Figure 4.6.

Recall that the proof of Theorem 4.11 is based on the idea that optimal partitions consist of three entire limbs each and that the search-space size is minimized if those partitions are balanced. It appears likely that these conditions still hold for the modified reduction after minor changes, such as the increase of x by some polynomial factor. An adaptation of the proof itself, however, would possibly be a lot more complex than the one presented in Section 4.3 due to the less transparent structure of the modified (m, B, x)-tree. Therefore, we only propose the following conjecture.

Conjecture 4.15. *The Problem* ArcFlagsPartition *is $\mathcal{NP}$-hard on undirected trees if their maximum degree is at most* 3.

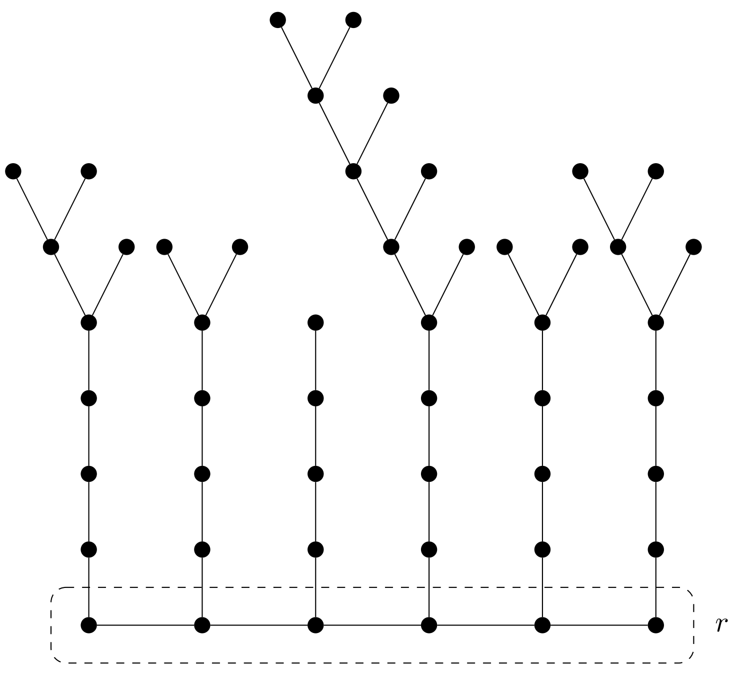

Figure 4.13: A modified (m, B, x)-tree with a maximum degree of 3.

One might consider other restricted classes of trees in order to find a class other than paths for which one can efficiently compute optimal cells. We close this section by considering trees with a limited height. Stars form a special case. A star $S = (V, E, \omega)$ consists of a node r and $n - 1$ leaves that are connected to r via an edge. Hence, stars are exactly the class of trees with a maximum height of 1. Imagine an arbitrary query in an undirected star. If the start and destination node are not the same and $s \neq r$, the second node settled is always r. Starting at r, we obviously minimize the search-space size by balancing the number of leaves per cell. It is easy to see that the same holds for directed stars. Whether or not there are polynomial-time algorithms that compute optimal cells for trees with their height limited to a constant that is greater or equal to 2 is left as an open question. A reduction of 3-Partition to $(m, B, 1)$-trees could provide a negative answer to this question.

Concluding Remarks

Throughout this chapter, we inspected several classes of graphs aiming at methods for computing optimal cells for them. Unfortunately, we could establish efficient procedures only for a few severely restricted classes of graphs, such as paths, directed cycles and stars. An efficient procedure for undirected cycles seems likely to exist. Conversely, we conjecture that the problem is $\mathcal{NP}$-hard for trees even if their degree is bounded by 3. It appears that efficient procedures for finding search-space optimal partitions exist only for very restricted classes of graphs that carry some sense of inherent symmetry, such as the classes mentioned above or special balanced trees. As a final remark, we suggest that the choice of other reasonable measures concerning the quality of a partition, for example the worst-case search-space size, would presumably lead to similar results in terms of polynomial-time solvability.

5. Linear Programs for Search-Space Optimal Partitions

In this chapter, we derive a linear program for solving the problem ArcFlagsPartition. To this end, in Section 5.1 we first introduce an ILP for a relaxed variant of this problem. Afterwards, this ILP is extended to cover the original problem as specified in Chapter 3. Ultimately, in Section 5.2 we discuss a way to obtain a dual ILP for the relaxed variant of the problem.

5.1 Primal Linear Programs

Assume we are given a graph $G = (V, E, \omega)$ and a positive integer k to determine the number of allowed cells. At first, in Section 5.1.1 an ILP for finding an optimal partition for a relaxed version of the problem ArcFlagsPartition is developed, where the stopping criterion of the arc-flags algorithm is omitted. In Section 5.1.2, this approach is refined to deduce a linear program that finds an optimal solution of the original problem by reintroducing the stopping criterion. In both sections, we discuss alternative ideas instead of proposing only one single program. In Section 5.1.3 we then show how to significantly reduce the number of necessary constraints in order to improve the performance of the ILP.

Representation of Cell Assignments

First of all, when looking for an ILP for optimal partitions, a representation of distinct cells and the maximum number of allowed cells k need to be specified. To achieve this goal, we use constraints of an existing ILP approach for graph clustering [Gör10]. Since the cells of the given graph, just like clusters, form a partition of the underlying graph, this approach can be directly applied to our scenario. Clusters of a graph are distinguished using binary cluster variables c_{vi}, such that $c_{vi} = 1$ if and only if node v belongs to cluster i. Furthermore, binary variables x_{uv} are introduced with the interpretation that $x_{uv} = 1$ if and only if nodes u and v are in the same cluster. To obtain consistent values for the latter variables, the following Constraints 5.1 to 5.4 realize an equivalence relation on the variables x_{uv} depending on the cell assignments specified by the corresponding variables c_{ui} and c_{vi} for $i \in \{1, \ldots, k\}$. Constraint 5.4 ensures that each node is assigned to exactly one cell.

$$c_{ui} + c_{vi} - x_{uv} \leq 1 \qquad \forall u < v \in V, 1 \leq i \leq k \tag{5.1}$$

$$x_{uv} + c_{ui} - c_{vi} \leq 1 \qquad \forall u < v \in V, 1 \leq i \leq k \tag{5.2}$$

$$x_{uv} + c_{vi} - c_{ui} \leq 1 \qquad \forall u < v \in V, 1 \leq i \leq k \tag{5.3}$$

$$\sum_{i=1}^{k} c_{vi} = 1, \qquad \forall v \in V \tag{5.4}$$

With these constraints in place, we are able to express cell assignments. This yields a starting point for different approaches to compute the search-space size of a given partition. All linear programs described below use this basis.

5.1.1 Integer Linear Programs for a Relaxed Problem Instance

In what follows, we consider a relaxed version of the problem ArcFlagsPartition, which we specified in Chapter 3. Therefore, we introduce a modified definition of this problem in which one asks for minimizing the search-space size of a modified version of the arc-flags algorithm that does not implement the stopping criterion.

Problem RelaxedArcFlagsPartition. *Given a graph $G = (V, E, \omega)$ and a positive integer k, find a partition $\mathcal{C}$ of G such that $|\mathcal{C}| \leq k$ and $\mathrm{S}^{+}_{\mathrm{AF}}(G, \mathcal{C})$ is minimized.*

As it turns out, the relinquishment of the stopping criterion makes the specification of a linear program easier. Below, we provide two different but closely related approaches for given input parameters $G = (V, E, \omega)$ and k.

A First Approach Using Binary Variables

A straight-forward formulation of the minimal search-space size could work as follows. First, we add variables that represent the arc-flags for a certain assignment of nodes to cells. To set the flags correctly, the backward-shortest-path trees for all nodes of the given graph G need to be computed in advance. Therefore, we assume that as a result of a preprocessing step there are $n \cdot m$ binary constants $b_w(u, v)$ such that $b_w(u, v) = 1$ if and only if the edge (u, v) is part of the backward-shortest-path tree rooted at node w. To decide whether an edge has a certain flag set to 1, we use $n \cdot m$ binary variables $f_t(u, v)$ such that $f_t(u, v)$ represents the value of the flag of the edge (u, v) corresponding to the cell that the target node t belongs to. According to the preprocessing of the arc-flags algorithm presented in Chapter 3, a flag for a certain target cell is set to 1 if and only if the corresponding edge is part of the backward-shortest-path tree rooted at any node of the respective cell. The following Constraint 5.5 assures that the flags are set properly. If two given nodes w and t belong to the same cell, the backward-shortest-path tree of w determines a subset of the set flags in a query to t.

$$f_t(u, v) \geq b_w(u, v) \cdot x_{wt} \qquad \forall t, w \in V, (u, v) \in E \tag{5.5}$$

We further introduce n^3 binary search-space variables $s_{st}(v)$ with the interpretation that $s_{st}(v) = 1$ if and only if the node v is included in the search space of a query from s to t. Following the relaxed problem definition given above, our goal is to minimize the total search-space size $\mathrm{S}^{+}_{\mathrm{AF}}(G, \mathcal{C}) = \sum_{s,t \in V} \mathrm{S}^{+}_{\mathrm{AF}}(G, \mathcal{C}, s, t)$. This directly leads to the following objective function.

$$\text{minimize} \sum_{s,t,v \in V} s_{st}(v) \tag{5.6}$$

Since the query algorithm does not abort its work at its destination t, we can interpret the query as a plain DIJKSTRA that runs on the cell-dependent graph $G_{c(t)} = (V, E_{c(t)})$ with $E_{c(t)} = \{e \in E \mid \mathcal{F}_{(e)}(c(t)) = 1\}$, as introduced in Chapter 3. Thus, we tentatively discard all edges from the graph that do not have the target flag set to 1. A node is then included in the search space of a given source node s if and only if it is reachable in $G_{c(t)}$. This is expressed in the following Constraints 5.7 and 5.8.

$$s_{st}(s) = 1 \qquad \forall s, t \in V \tag{5.7}$$

$$s_{st}(u) + f_t(u,v) - s_{st}(v) \leq 1 \qquad \forall s, t \in V, (u,v) \in E \tag{5.8}$$

Altogether, we obtain an integer linear program that finds an optimal solution for an arbitrary instance of the problem RELAXEDARCFLAGSPARTITION by minimizing the objective function given in Equation 5.6 subject to the constraints specified in Equations 5.1 to 5.5, 5.7 and 5.8.

A Second Approach Using Integer Variables

It is easy to see that the $n^2 + m \cdot n^2$ constraints depicted in Equations 5.7 and 5.8 are highly redundant. This is due to the fact that we omit the stopping criterion: When starting from a certain node s, the search-space size only depends on the target cell that t belongs to. Thus, for a given cell assignment and a distinct node s, $\sum_{v \in V} s_{st}(v)$ is identical for all nodes t that belong to the same cell. Therefore, we can summarize the search-space variables that share the same target cell. To this end, we use $k \cdot n^2$ integer variables $s_{si}(v)$ to represent the former binary search-space variables $s_{st}(v)$ of all targets t in the i-th cell. Analogously to Equation 5.6, our new objective function given by Equation 5.9 aims at minimizing the total search-space size.

$$\text{minimize} \sum_{s,v \in V} \sum_{1 \leq i \leq k} s_{si}(v) \tag{5.9}$$

Instead of the flag variables $f_t(u,v)$ introduced above we now use complementary variables $h_i(u,v)$ that represent the absence of a corresponding edge (u,v) in the graph induced by the i-th cell, i.e., it is $h_i(u,v) = 1 - f_t(u,v)$ if $c(t) = i$. The corresponding Constraint 5.10 given below replaces the constraint in Equation 5.5.

$$h_i(u,v) \leq 1 - b_w(u,v) \cdot c_{wi} \qquad \forall w \in V, (u,v) \in E, 1 \leq i \leq k \tag{5.10}$$

Using Constraints 5.11 and 5.12, we can now get along with only $n \cdot k + n \cdot m \cdot k$ constraints. In addition to that, note that we do not need the variables x_{uv} in this case, which allows us to omit Equations 5.1 to 5.3 in this variant. An integer variable $s_{si}(v)$ now sums up the binary search-space sizes $s_{st}(v)$ of all distinct target nodes t of a given cell represented by the index i. Assume that below, M denotes a large constant such that $M \geq s_{si}(v)$ can be assured for all $s, v \in V$ and $i \in \{1, \ldots, k\}$. Setting $M = n$ satisfies this condition.

$$s_{si}(s) = \sum_{v \in V} c_{vi} \qquad \forall s \in V, 1 \leq i \leq k \tag{5.11}$$

$$s_{si}(u) - s_{si}(v) - M \cdot h_i(u,v) \leq 0 \qquad \forall s \in V, (u,v) \in E, 1 \leq i \leq k \tag{5.12}$$

A second variant of an ILP solving RELAXEDARCFLAGSPARTITION is then given by the Objective 5.9 with respect to the Constraints 5.1 and 5.10 to 5.12. Because usually $k \ll n$ holds, the difference in the number of constraints compared to the first version of the ILP may accordingly be very large. However, the newly introduced search-space variables $s_{si}(v)$ are now integer variables instead of binaries, which extends the solution space. The choice of the right ILP variant should therefore depend on the number of cells. For small values of k the second approach might work faster, whereas for larger values of k the original ILP presented above is recommended.

5.1.2 Applying the Stopping Criterion

Now, we construct a linear program for the original problem ArcFlagsPartition, so the underyling query algorithm is assumed to make use of the stopping criterion. Note that this may indeed have an influence on the search-space optimal partition of the graph. In what follows, we first see why a simple adaptation of the ILP presented in Section 5.1.1 fails, before we derive a correct MILP for the modified task.

An Attempt to Adapt the Established ILP

Recall that in the new scenario, the query algorithm is aborted once the target node has been settled. Assuming there is a fixed order $\prec$ in which nodes are extracted from the priority queue, a node v is now settled by Dijkstra's algorithm in an s-t-query if and only if $d(s,v) < d(s,t)$ or $d(s,v) = d(s,t)$ and additionally $u \preceq t$. This information can be computed on beforehand for all triples s, t, v in polynomial time. Thus, a first idea might be to simply adjust the ILP from Section 5.1.1 by adding binary constants $d_{st}(v)$ with $d_{st}(v) = 1$ if and only if $d(s,v) < d(s,t)$ or $d(s,v) = d(s,t)$ and $u \preceq t$. Adding a single constant to each of the constraints in Equation 5.8, one receives the following adapted constraints instead.

$$s_{st}(u) + f_t(u,v) + d_{st}(v) - s_{st}(v) \leq 2 \qquad \forall s,t \in V, (u,v) \in E \tag{5.13}$$

Unfortunately, the ILP induced by this approach is not correct. Since the query algorithm in an arc-flag enhanced query skips certain edges, computed distances to settled nodes that do not belong to the target cell may be incorrect. For a simple example, see Figure 5.1. Imagine an arc-flags based query from node s to node t, where the edges (s,t) and (t,v) have their flags for cell $c(t)$ set to 1, while (s,v) does not. We further assume that all edge weights are uniform and $v \prec t$. Clearly, the query algorithm immediately settles t and thus we have $\mathrm{S_{AF}}(s,t) = 2$. However, v would belong to the Dijkstra search space of an s-t-query because starting from s, it would be settled first. This leaves us with $s_{st}(t) = 1$, $f_t(t,v) = 1$ and $d_{st}(v) = 1$, which falsely implies that due to Constraint 5.13, $s_{st}(v)$ must be 1 in the obtained ILP.

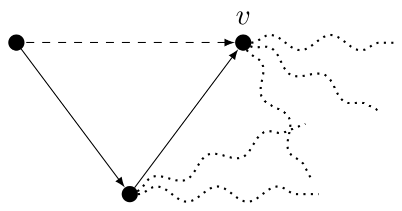

Figure 5.1: A simple example to show the incorrectness of the ILP induced by Constraint 5.13.

However, because the removal of edges cannot reduce the distance between two nodes, the ILP using Constraint 5.13 would yield an upper bound on the actual search-space size. Although in general this ILP would not compute the correct optimum, we can use this observation for later tuning the correct linear program that is developed below. See Section 5.1.3 for further details on tuning of the linear program.

A Correct MILP for Strongly Connected Graphs

For now, let us assume that the underlying graph is strongly connected. Similar to the first ILP examined in Section 5.1.1, we introduce n^3 search-space variables $s_{st}(v)$ and define the objective function as follows.

$$\text{minimize} \sum_{s,t,v \in V} s_{st}(v) \tag{5.14}$$

We have seen that using the arc-flags algorithm, computed distances from the source node to any node that does not belong to the target cell may become larger than the correct distance due to ignored edges. In other words, the computed distances, and consequently the order in which nodes are removed from the queue, depend on the cell assignment. Thus, the distance from a node s to any node that is not in the target cell cannot be determined in advance. Instead, we introduce $n^2 \cdot m$ continuous variables $\delta_{st}(v)$ that represent the distances $d_{G_{c(t)}}(v, s)$ from s to any node v in the target-cell-dependent graph $G_{c(t)} = (V, E_{c(t)}, \omega)$ introduced in Chapter 3. We know that the arc-flags algorithm computes correct distances for all nodes in the target cell. Consequently, $\delta_{st}(t) = d_G(s, t)$ holds for all pairs of nodes s and t. So, for fixed s and t, a node v is included in $\mathrm{S}_{\mathrm{AF}}(s, t)$ if and only if $\delta_{st}(s, v) < d_G(s, t)$ or $\delta_{st}(s, v) = d_G(s, t)$ and $u \preceq t$. The Constraints 5.15 and 5.16 express these relations, where all $d(s, t)$ are supposed to be precomputed constants that hold the corresponding distances of the input graph G. Again, M is supposed to be a sufficiently large number which ensures that the inequalities remain satisfiable.

$$\delta_{st}(v) - d(s,t) + M \cdot s_{st}(v) \geq 0 \qquad \forall s, t \prec v \in V \tag{5.15}$$
$$\delta_{st}(v) - (d(s,t) + \varepsilon) + M \cdot s_{st}(v) \geq 0 \qquad \forall s, t \succeq v \in V \tag{5.16}$$

Observe that Constraint 5.15 ensures that all nodes with $\delta_{st}(s, v) < d_G(s, t)$ are included in the search space. To handle nodes that have a distance to s in $G_{c(t)}$ that is equal to $d_G(s, t)$, we have to check whether $\delta_{st}(s, v) \leq d_G(s, t)$. This is equivalent to $\delta_{st}(s, v) < d_G(s, t) + \varepsilon$ if the constant ε is sufficiently small, as assumed for Equation 5.16. Since we expect edge weights to be positive real numbers, one has to make sure that ε is not greater than any difference of any two path lengths in $G_{c(t)}$. If all weights in G are positive integers, $\varepsilon = 1$ fulfills this requirement. For the constant number M one has to make sure that M is greater or equal than the maximum distance in G, that is, $M \geq \max_{s,t \in V} d_G(s, t)$.

The Constraints 5.15 and 5.16 ensure that all variables $s_{st}(v)$ are set correctly, provided that the objective is to minimize the search-space size and that all variables $\delta_{st}(v)$ have values that preserve the appropriate relation to $d_G(s, t)$. Note that $\delta_{st}(v)$ does not have to hold the exact value of the tentative distance of v from s in $G_c(t)$ as long as this condition is met. In other words, one only has to make sure that $d_{G_{c(t)}}(s, v) \leq d_G(s, t)$ implies $\delta_{st}(v) \leq d_G(s, t)$, because in this case v must be included in the search space of an s-t-query. On the other hand, in an optimal solution, $\delta_{st}(v) > d_G(s, t)$ automatically holds for all other distance variables, for this allows setting $s_{st}(v) = 0$ and the objective is to minimize the search-space size. Consequently, we do not have to handle this case explicitly. The following Constraints 5.18 and 5.19 ensure that the values of all variables $\delta_{st}(v)$ fulfill our requirements. Again, we use variables $h_t(u, v)$ to hold the information whether an edge (u, v) is available in a certain cell-dependent graph $G_{c(t)}$. Along similar lines to the linear programs studied above, Constraint 5.17 ensures that these variables are set properly. An edge (u, v) that is not present in a cell-dependent graph implies that the distance from u to v is not bounded by $\omega(u, v)$. Hence, we set the variables $\delta_{st}(v)$ accordingly in Equations 5.18 and 5.19.

$$h_t(u,v) \leq 1 - b_w(u,v) \cdot x_{wt} \qquad \forall t, w \in V, (u,v) \in E \tag{5.17}$$
$$\delta_{st}(s) = 0 \qquad \forall s, t \in V \tag{5.18}$$
$$\delta_{st}(v) \leq \delta_{st}(u) + \omega(u,v) + M \cdot h_t(u,v) \qquad \forall s, t \in V, (u,v) \in E \tag{5.19}$$

Setting the constant M to a value greater than $\max_{s,t \in V} d_G(s, t)$ suffices for the MILP to work correctly. Summarily, we obtain a linear program to solve the problem ArcFlagsPartition by stating the objective function given in Equation 5.14 subject to the Constraints 5.1 to 5.4 and 5.15 to 5.19.

Extension to Graphs that are not Strongly Connected

There are two simple ways to enable the MILP stated above to cover general graphs, in which certain target nodes may be unreachable. The first approach would be to simply set the constant $d(s,t) = M$ or $d(s,t) = M - \varepsilon$ in Equations 5.15 and 5.16, respectively, if the precomputed distance from s to t is infinite. Then, any reachable node in the cell-dependent graph must be included in the search space if $d(s,t) = \infty$ and the linear program can be adapted without any further changes.

Another approach is to define the constraints subject to the precomputed values for $d(s,t)$. For any pair of nodes s and t, the constraints given above remain unchanged if $d(s,t) < \infty$. Otherwise, if $d(s,t) = \infty$, we remove Constraints 5.15 to 5.19 and replace them by Constraints 5.5, 5.7 and 5.8 presented in Section 5.1.1. Since in case of an unreachable target, all reachable nodes of the cell-dependent graph must be settled, this yields a correct adaptation of the linear program.

5.1.3 Tuning of the Obtained Linear Programs

In what follows, two simple modifications of the linear programs for search-space optimal partitions are presented in order to reduce the number of constraints and variables. These easily implementable changes may improve the performance of an underlying solver. The main idea is to use efficiently precomputable information to reduce the solution space. As before, we assume that the values $d_G(s,t)$ have been determined for all pairs of nodes s, t of the input graph G during an all-pairs shortest path computation that has been run beforehand.

1. Assume there are nodes s, t, v such that $d_G(s,t) < d_G(s,v)$. We know that the distance $d_G(s,t)$ from the source s to the target t is preserved in a query due to the correctness of the arc-flags algorithm. Furthermore, discarding single edges from the graph in a single run of Dijkstra's algorithm cannot reduce the distance between any two nodes. Thus we know that if $d_G(s,t) < d_G(s,v)$ holds, $d_G(s,t) < d_{G_{c(t)}}(s,v)$ and therefore $s_{st}(v) = 0$ follows immediately. Hence, we can drop all variables $s_{st}(v)$ where $d_G(s,t) < d_G(s,v)$ and all constraints including these variables from the linear program.

2. To reduce the solution space, it is helpful to find bounds on any occurrences of the large constant M, which is used in Equations 5.12, 5.15 5.16 and 5.19. In what follows, we look for individual proper bounds on each of these values such that the resulting linear program remains correct. First of all, consider the value of M in Equation 5.12. The search-space variables $s_{si}(v)$ count the number of target nodes for which v is settled when starting at s. Clearly, this number cannot exceed n. By precomputing the exact number of reachable nodes for every node of the graph one could even obtain better bounds. For the equation to be satisfiable in any case, it then suffices to set M to the corresponding number. For Equations 5.15 and 5.16 it is clear that setting $M = d(s,t)$ and $M = d(s,t) + \varepsilon$, respectively, suffices for the MILP to maintain correctness. Finally, we examine the occurrence of M in Equation 5.19. Remember that this value must allow tentative distances $\delta_{st}(v)$ to become larger than $d(s,t)$, for this in turn allows corresponding search-space variables to drop to 0. Obviously, setting M to a value that is marginally larger than $d(s,t)$ suffices for this condition. Hence we may set $M = d(s,t) + \varepsilon$, for instance.

In addition to these improvements, generic approaches such as the use of callbacks may leave more room for tuning. However, an analysis of such techniques is beyond the scope of this thesis.

5.2 Towards a Dual Integer Linear Program

In this section, we derive the dual ILP for a simplified version of the first primal linear program that we obtained in Section 5.1.1. The dual ILP given below may serve as the starting point of a more detailed analysis. For example, such an analysis could aim at a deeper understanding of the problem structure or at the establishment of lower bounds on the optimal solution, which in turn can be useful for finding approximation bounds of polynomial-time algorithms.

It appears reasonable to choose a comparatively simple primal ILP for this analysis. Hence, we consider the first ILP that was presented in Section 5.1.1. First of all, we omit the Constraints 5.1 to 5.3 and the variables x_{uv}. This seems impractical since it saves $3kn^2$ constraints and n^2 variables while it adds $(k-1) \cdot n^2 \cdot m$ new constraints given in Equation 5.21 to compensate for this simplification. However, this modification has no influence on the optimal solution and thus it serves our purpose here. We further simplify the linear program by adding unnecessary constraints. The variables $f_t(u,v)$ for all $(u,v) \in E$ were used to indicate a set flag. To simplify the matter, we introduce additional variables $f_t(u,v)$ for all remaining tuples $(u,v) \notin E$. As long as we have $b_w(u,v) = 0$ for all constants corresponding to pairs (u,v) that do not represent edges of the input graph, the additional variables have no influence on the obtained optimal solution. Moreover, we only request that all variables are integers greater or equal zero instead of binaries for simplicity. This leaves us with the following primal program.

$$\text{minimize} \sum_{s,t,v \in V} s_{st}(v)$$

subject to

$$\sum_{i=1}^{k} c_{vi} = 1 \qquad \forall v \in V \tag{5.20}$$

$$b_w(u,v) \cdot c_{wi} + c_{ti} - f_t(u,v) \leq 1 \qquad \forall t,u,v,w \in V, i \in \{1,\ldots,k\} \tag{5.21}$$

$$s_{st}(s) = 1 \qquad \forall s,t \in V \tag{5.22}$$

$$s_{st}(u) + f_t(u,v) - s_{st}(v) \leq 1 \qquad \forall s,t,u,v \in V \tag{5.23}$$

$$c_{vi}, s_{st}(v), f_t(u,v) \geq 0 \qquad \forall s,t,u,v \in V, i \in \{1,\ldots,k\}$$

Below, we state the corresponding dual program. We introduce n variables C_v for Constraints 5.20, $k \cdot n^4$ variables F_{tuvwi} for the flag-setting Constraints 5.21, n^2 variables S_{st} for the initial search-space sizes defined in Constraints 5.22 and finally n^4 variables S_{stuv} corresponding to Constraints 5.23.

$$\text{maximize} \sum_{v \in V} C_v + \sum_{\substack{t,u,v,w \in V \\ 1 \leq i \leq k}} F_{tuvwi} + \sum_{s,t \in V} S_{st} + \sum_{s,t,u,v \in V} S_{stuv}$$

subject to

$$C_v + \sum_{v,x,y \in V} (b_w(x,y) \cdot F_{vxywi} + b_w(x,y) \cdot F_{wxyvi}) \leq 0 \qquad \forall w \in V, i \in \{1,\ldots,k\} \tag{5.24}$$

$$\sum_{s \in V} S_{stuv} - \sum_{w \in V, 1 \leq i \leq k} F_{tuvwi} \leq 0 \qquad \forall t,u,v \in V \tag{5.25}$$

$$S_{st} + \sum_{v \in V} (S_{stsv} - S_{stvs}) \leq 1 \qquad \forall s,t \in V \tag{5.26}$$

$$\sum_{v \in V} (S_{stuv} - S_{stvu}) \leq 1 \qquad \forall s \neq u, t \in V \tag{5.27}$$

$$F_{tuvwi}, S_{stuv} \leq 0 \qquad \forall s,t,u,v,w \in V, i \in \{1,\ldots,k\}$$

In the dual ILP, constraints given in Equation 5.27 represent the variables c_{vi} of the primal. The variables $f_t(u, v)$ are represented by Equation 5.25. Equation 5.26 holds constraints according to the variables $s_{st}(s)$ and Equation 5.27 covers all remaining $s_{st}(v)$ where $v \neq s$. Unfortunately, it appears that there is no obvious interpretation of the dual program, even for the simplified version given above. However, a more detailed analysis of the dual ILP is beyond the scope of this thesis. This may instead be part of future work on this topic.

6. Greedy Approaches

In this chapter, we examine two greedy approaches finding partitions of arbitrary graphs to serve as cells for the arc-flags algorithm. The first algorithm we present is based on the idea that nodes should be in the same partition if their corresponding backward-shortest-path trees are similar. The second approach assigns nodes to cells one after another and greedily optimizes the search-space size in every step.

From the results in Chapter 4 we know that the problem ArcFlagsPartition is $\mathcal{NP}$-hard even if input graphs are restricted to trees. The question that arises naturally is whether or not one can find algorithms for which there exist provable guarantees concerning the quality of their provided solutions. As we have already mentioned in Chapter 3, algorithms for preprocessing a given graph used in practice are heuristics. Their strategies for finding a partition of a given graph are based on intuitions about the characteristics of a good partition. However, as we also pointed out, one can construct input instances for which these intuitions fail. Hence, it appears to be unlikely that one could establish such guarantees for the corresponding heuristics. In this chapter, we focus on the question whether we can find any guarantees for the two combinatorial approaches presented below. Unfortunately, it turns out that no constant approximation ratio can be established for both of them, even if we restrict input instances to strongly connected graphs.

In the following Sections 6.1 and 6.2 we present the two greedy approaches mentioned above. In each section, we give a brief description of how the algorithm works. An analysis of the time and space complexity of the algorithm follows. Finally, we construct input instances for which the partition generated by the particular approach is of very poor quality. In Section 6.3 we then present a short case study to indicate the qualities of both algorithms on realistic instances.

A Trivial Bound on the Approximation Ratio

First of all, we observe that any algorithm that returns a valid partition of a graph fulfills a trivial approximation ratio concerning the problem ArcFlagsPartition. For an arbitrary graph G, there are n^2 distinct s-t-queries. Each query settles at least the source node s, resulting in a search-space size that is at least 1. Consequently, n^2 is a lower bound on the search-space size $\mathrm{S}_{\mathrm{AF}}(G, \mathcal{C})$, regardless of the partition $\mathcal{C}$. On the other hand, at most n nodes can get settled in any query. We obtain an upper bound on the search-space size of n^3, so the corollary given below follows immediately.

Corollary 6.1. *Let $\mathcal{A}$ be an algorithm that takes as input a graph G and a positive integer k and outputs a partition of G into at most k cells. Then $\mathcal{A}$ is an n-approximation algorithm for* ArcFlagsPartition.

In fact, using Lemma 2.2, we can even bound the approximation ratio by $(n+1)/2$ with similar arguments.

6.1 Merging Cells with Similar Flags

The basic idea of the first greedy approach is as follows. On a query to a given target node t, one may encounter edges that have the flag for the target cell $c(t)$ set to 1, even though they are not part of the backward-shortest-path tree rooted at t. To keep the search-space size of an s-t-query small, one needs to assure that as few edges as possible have this property. Clearly, if exactly those edges belonging to the backward-shortest-path tree of t had the corresponding flag set, the query would only settle nodes that actually lie on the shortest path. This clearly is the best case that may occur in an arc-flag based query. Obviously however, in general one cannot assure that only these edges have the target cell flag set. Instead, the following approach aims at putting nodes into the same partition if the edges of their backward-shortest-path trees cover similar edges to the greatest possible extent.

6.1.1 Algorithm Description and Analysis

Algorithm 6.1 shows our approach, which works as follows. The input of the algorithm consists of a directed graph $G = (V, E, \omega)$ and an integer k denoting the maximum number of cells of the desired partition. In a first step, the backward-shortest-path tree of every node of the graph is computed and stored in an array. This can be done by distinct runs of Dijkstra's algorithm on the reverse graph $\overline{G}$ starting once at every node. Here, we assume that the routine Dijkstra() called in Line 2 returns a set of edges representing the shortest-path tree of the respective source node. In the second step, the differences between these trees are computed and stored as well. In our context, we define the *difference* between two shortest-path trees $T_u = (V_u, E_u, \omega_u)$ and $T_v = (V_v, E_v, \omega_v)$ of two nodes $u, v \in V$ as the size of the set $(E_u \cup E_v) \setminus (E_u \cap E_v)$. In the last step of Algorithm 6.1, the cells of the partition are obtained as follows. Initially, each node is interpreted as a single cell. Its backward-shortest-path tree then holds exactly the edges that have the flag of that cell set to 1. Cells are then iteratively merged in a greedy fashion until k cells are left. Every merging phase consists of the following steps. First, the minimal difference between any pair of cells is determined. Then, the corresponding cells and the sets of flagged edges are merged. Afterwards, the differences between the newly created cell and all remaining cells are updated.

Time and Space Complexity

We proceed with a brief analysis of the running time and the space consumption of Algorithm 6.1. Step one consists of n runs of Dijkstra's algorithm, which runs in $\mathcal{O}(m+n\log n)$ for each node. Hence, we get a total running time that is in $\mathcal{O}(nm+n^2\log n)$. The amount of storage necessary to keep n shortest-path trees is in $\mathcal{O}(n^2)$. The second step has a time complexity of $\mathcal{O}(n^2m)$. Storing n^2 differences again requires quadratic space complexity. The third step is dominated by its two inner loops beginning in Lines 13 and 20, respectively. The first one searches for the minimum of $\mathcal{O}(n^2)$ differences, whereas the second updates the cell differences in $\mathcal{O}(nm)$ steps. Since these loops are executed at most n times each, the time complexity of the last step is in $\mathcal{O}(n^2m+n^3)$. Merging two cells into one requires at most the space necessary to store the two cells separately, so the space

Algorithm 6.1: GreedyMerge

```
    Input: Graph $G = (V, E, \omega)$, positive integer $k$
    Data: Array flags of sets of edges, matrix diff of integers
    Output: A partition $\mathcal{C}$ of $G$ with $|\mathcal{C}| = k$

    // Step 1: Compute and store all backward-shortest-path trees
 1  forall $v \in V$ do
 2      $\mathsf{flags}(\{v\}) \leftarrow$ Dijkstra$(\overline{G}, v)$

    // Step 2: Initialize cell differences and partition
 3  forall $u \prec v \in V$ do
 4      $\mathsf{diff}(\{u\},\{v\}) \leftarrow 0$
 5      forall $e \in E$ do
 6          if $(e \in \mathsf{flags}(\{u\}) \wedge e \notin \mathsf{flags}(\{v\})) \vee (e \notin \mathsf{flags}(\{u\}) \wedge e \in \mathsf{flags}(\{v\}))$ then
 7              $\mathsf{diff}(\{u\},\{v\}) \leftarrow \mathsf{diff}(\{u\},\{v\})+1$

 8  $\mathcal{C} \leftarrow \emptyset$
 9  forall $v \in V$ do
10      $\mathcal{C} \leftarrow \mathcal{C} \cup \{\{v\}\}$

    // Step 3: Greedy minimization of cell differences
11  while $|\mathcal{C}| > k$ do
        // Find the pair of cells with minimum difference
12      $min \leftarrow \infty$
13      forall $C_i, C_j \in \mathcal{C}$ do
14          if $\mathsf{diff}(C_i, C_j) < min$ then
15              $min \leftarrow \mathsf{diff}(C_i, C_j)$
16              $i_1 \leftarrow i$, $i_2 \leftarrow j$

        // Merge the cells with minimum difference
17      $C' \leftarrow C_{i_1} \cup C_{i_2}$
18      $\mathsf{flags}\ (C') \leftarrow \mathsf{flags}\ (C_{i_1}) \cup \mathsf{flags}\ (C_{i_2})$
19      $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{C_{i_1}, C_{i_2}\}) \cup \{C'\}$
        // Update differences
20      forall $C \neq C' \in \mathcal{C}$ do
21          $\mathsf{diff}(C, C') \leftarrow 0$
22          forall $e \in E$ do
23              if $(e \in \mathsf{flags}(C) \wedge e \notin \mathsf{flags}(C')) \vee (e \notin \mathsf{flags}(C) \wedge e \in \mathsf{flags}(C'))$ then
24                  $\mathsf{diff}(C, C') \leftarrow \mathsf{diff}(C, C')+1$
```

consumption does not increase after each step of merging. Overall, the algorithm uses quadratic amount of storage and runs in polynomial time with a complexity of $\mathcal{O}(n^4)$ in general and $\mathcal{O}(n^3)$ if the degree of every node is bounded by a constant.

6.1.2 Bounding the Approximation Ratio

Although Algorithm 6.1 may provide high quality solutions for many input instances, the relation between differences in the number of flagged edges and the resulting search-space size is unclear. It turns out that there are instances for which the partition provided by Algorithm 6.1 is very poor in terms of applicability for the arc-flags algorithm. In what follows, we show that Algorithm 6.1 yields no $\mathcal{O}(\sqrt{n})$-approximation for the problem ArcFlagsPartition, even if we restrict the set of allowed input instances to strongly connected graphs. Since we know from Corollary 6.1 that n is a trivial upper bound on the approximation ratio, this is a rather dissatisfactory result.

We construct a graph for which Algorithm 6.1 provides a rather poor solution. Let k be the number of allowed cells. The computed partition will hold $k-1$ cells of size 1 and one cell of size $n-k+1$. Let $G' = (V', E', \omega')$ be an undirected star, i.e., there is a distinct node r such that $\{(v,r),(r,v)\} \subseteq E'$ for all $v \in V' \setminus \{r\}$ and G' contains no further edges. The number of nodes in G' is specified later. We obtain the desired graph G by adding a set W of $k-1$ nodes $w_1, \ldots, w_{k-1}$ to V' and edges (v, w_i) for all $v \in V'$, $i \in \{1, \ldots, k-1\}$, to E'. In other words, we add edges from every single node of the star to all of the $k-1$ newly added nodes. To make the graph G strongly connected, we finally add $k-1$ edges (w_i, r) for all $i \in \{1, \ldots, k-1\}$. Furthermore, we set all edge weights to one. Figure 6.1 shows a schematic example of the conceived graph G.

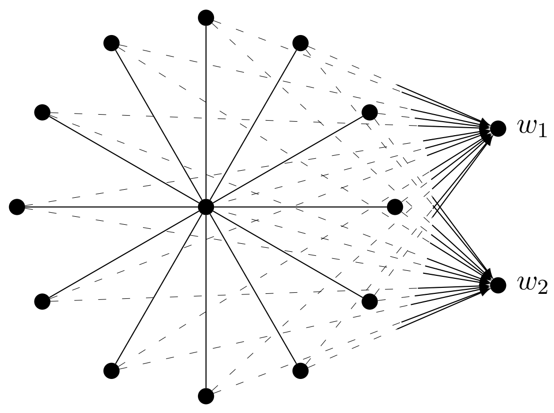

Figure 6.1: A graph that induces bad performance of Algorithm 6.1.

The backward-shortest-path trees of the nodes in G look as depicted in Figure 6.2. Any of these trees contains the $k-1$ edges (w_i, r) for $i \in \{1, \ldots, k-1\}$. Apart from that, backward-shortest-path trees corresponding to nodes in V' only cover nodes of the original star G'. By contrast, the backward-shortest-path trees of the $k-1$ nodes in W cover exactly the edges incoming directly from all nodes in V'. If we look at the initially inferred tree differences, it is then clear that differences between any nodes $u, v \in V'$ are at most 4. Differences between any two nodes $v \in V'$ and $w \in W$ each sum up to $2(n-k)-1$ and the difference between any pair $w_1, w_2 \in W$ is exactly $2(n-k)$.

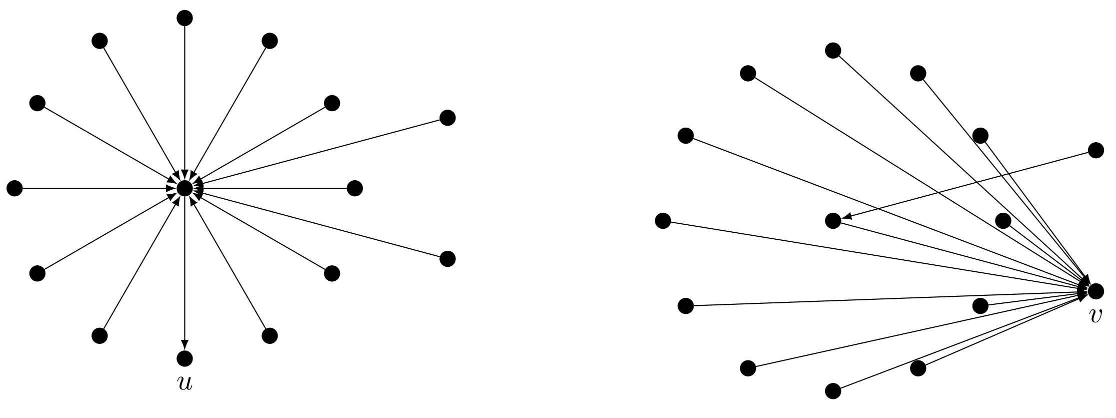

Figure 6.2: Backward-shortest-path trees of two labeled nodes u and v of the adverse graph instance.

In what follows, we show that the partition returned by Algorithm 6.1 given the input parameters G and k is $\mathcal{C} = \{V', \{w_1\}, \dots, \{w_{k-1}\}\}$. From the explanations given above it is clear that the algorithm starts by merging pairs of nodes in V' into one cell. Imagine a set of edges obtained after exclusively merging edges of arbitrary backward-shortest-path trees corresponding to nodes in V'. First of all, every such set contains at least $n-k-1$ of altogether $n-k$ edges (v, r) of E' with the root r of G' being the head of the edge. If we only account for these edges, the maximum difference between any two cells holding only nodes from V' is at most 2. In addition to that, the $n-k$ edges (r, v) with the root being the tail of the edge are possibly part of this set. Hence, the maximum difference between any cells obtained after merging nodes in V' is bounded by $n-k+2$ in total. Moreover, the differences between obtained cells after merging arbitrary nodes in V' and the cells containing single nodes of W remain unaffected. Since we may safely assume that $n \geq k+4$, differences between cells that are subsets of V' are always smaller than the differences of $2(n-k)-1$ between one of these cells and the nodes in W. Consequently, the preprocessing algorithm always merges cells that exclusively contain nodes from V' until the number of cells is reduced to k. On termination, the output of the algorithm is thus $\mathcal{C} = \{V', \{w_1\}, \dots, \{w_{k-1}\}\}$, as claimed above.

The Approximation Ratio of the Adverse Input Instance

To obtain a feasible lower bound on the corresponding search-space size, consider the intra-cell search-space size of the cell V'. Since this cell induces a strongly connected subgraph of G, we get an intra-cell search-space size that we can bound as shown below due to Lemma 2.2. This yields a lower bound on the search-space size S_{greedy} induced by the partition computed by Algorithm 6.1.

$$\begin{aligned} S_{\text{greedy}} &\geq (n-k+1)^2 \frac{(n-k+2)}{2} \\ &\geq \frac{(n-k)^3}{2} \\ &= \frac{1}{2} \cdot (n^3 - 3n^2k + 3nk^2 - k^3) \end{aligned} \tag{6.1}$$

Next, we construct a partition $\mathcal{C}'$ of G in which all nodes $v \neq r \in V'$ are distributed equally over the k cells. The remaining nodes $r, w_1, \dots, w_{k-1}$ are as well assigned to a distinct cell, each. We may safely assume that $n/k \in \mathbb{N}$ and hence the obtained partition consists of k cells of equal size. Note that the given partition is not even claimed to be search-space optimal in general. Consider an arbitrary s-t-query that originates at a node $s \in V$. The only nodes that possibly get settled despite not being in the target cell are s and r. The number of nodes settled in the target cell only depends on the order $\prec$ and takes a distinct value in the range from 1 to n/k for each of the n/k possible target nodes of that cell. Thus, the total number of settled nodes in all queries to a certain target cell can be bounded by $\sum_{i=3}^{n/k+2} i = (n/k+2)(n/k+3) - 3$. Since there are n nodes and k cells in total, an upper bound on the optimal search-space size for G is obtained as follows.

$$\begin{aligned} S_{\text{opt}} &\leq n \cdot k \cdot \left(\frac{\left(\frac{n}{k}+2\right)\left(\frac{n}{k}+3\right)}{2} - 3 \right) \\ &\leq n \cdot k \cdot \frac{\left(\frac{n}{k}+3\right)^2}{2} \\ &= \frac{1}{2k} \cdot \left(n^3 + 6n^2k + 9nk^2\right) \end{aligned} \tag{6.2}$$

Given an upper bound on the optimal search-space size and a lower bound on the search-space size induced by the partition computed by Algorithm 6.1, we finally turn to the approximation ratio corresponding to the input graph G. Setting $k = \sqrt{n}$ we achieve the following bound from Equations 6.1 and 6.2.

$$
\begin{aligned}
\frac{S_{\text{greedy}}}{S_{\text{opt}}} &\geq k \cdot \frac{n^3 - 3n^2k + 3nk^2 - k^3}{n^3 + 6n^2k + 9nk^2} \\
&= \frac{(\sqrt{n} - 3)\, n^3 + 3n^{\frac{5}{2}} - n^2}{n^3 + 6n^{\frac{5}{2}} + 9n^2} \\
&= \left(\sqrt{n} - 3\right) \frac{n + 3\sqrt{n} - 1}{n + 6\sqrt{n} + 9} \qquad (6.3)
\end{aligned}
$$

For large n Equation 6.3 gets close to $\sqrt{n} + b$ for a constant $b \in \mathbb{R}$. Furthermore, we obtain an analogous result if we set $k = a \cdot \sqrt{n}$ for an arbitrary constant $a \in \mathbb{R}^+$. Hence, Algorithm 6.1 does not provide an $f(n)$-approximation for any $f \in \mathcal{O}(\sqrt{n})$ nor a δ-approximation for any constant δ.

6.2 Assigning Nodes Sequentially

We present a second approach for solving the problem ArcFlagsPartition on arbitrary graphs. The idea of this algorithm is to assign single nodes of the given graph to cells one after another. Each time a new node is picked, its cell is chosen optimally with respect to the nodes that have already been processed.

6.2.1 Algorithm Description and Analysis

Algorithm 6.2 shows the sequential approach. It maintains two sets T and U with the invariant that $T \uplus U = V$. In each step of the main loop, a node of the set U is moved into the set T. The routine SelectNode called in Line 7 picks a node from a given set for this purpose. Its concrete implementation is specified later. The node returned by this method is then assigned to a cell such that the current search-space size is minimized. To find the best cell for a certain node u, the node is temporarily assigned once to every cell. Each time, the overall search-space size of queries from any node in G to all nodes that have already been handled, including u, is computed. The cell assignment that causes the lowest search-space size is chosen for u. The function CountingDijkstra called in Line 16 of Algorithm 6.2 is a slightly modified version of Dijkstra's algorithm used to obtain the correct search-space sizes. It requires three parameters, the first of which is a cell-induced subgraph of G, the second is a set of target nodes and the last is the source node. The function is used to compute the change in the search-space size in the following way. To receive the new search-space size of the candidate cell of u, one simply counts number of nodes settled so far during a run of Dijkstra's Algorithm starting from a certain node in G. Every time a node in the target cell is settled, the current value of the counter is added to the search-space size. Doing this for all distinct source nodes and accounting for unreachable nodes at the end yields the search-space size $\sum_{s \in V, t \in C_i} \mathrm{S}_{\mathrm{AF}}(s,t)$ induced by cell C_i of the so far constructed partition $\{C_1, \ldots, C_k\}$. Pseudo code of this procedure is given in Algorithm 6.3. Recall that at the end, the considered node is added to the cell that minimizes the search-space size $\sum_{s \in V, t \in T} \mathrm{S}_{\mathrm{AF}}(s,t) = \sum_{i=1}^{k} \sum_{s \in V, t \in C_i} \mathrm{S}_{\mathrm{AF}}(s,t)$. Since exactly one cell is modified, only the corresponding term changes in the sum given above. Hence, we simply assign u to the cell for which the difference in this term is minimized. To this end, we keep track of the minimum difference in the variable $s_{\min}$. The main loop is then continued until all nodes have been processed.

Time and Space Complexity

In what follows, we give an analysis of the time and space complexity of Algorithm 6.2. First of all, it is easy to see that the main loop dominates the running time of the algorithm.

Algorithm 6.2: GreedySequential

```
Input: Graph $G = (V, E, \omega)$, positive integer $k$
Data: Sets $T$, $U$, $C_1, \ldots, C_k$ of nodes, sets $E_1, \ldots, E_k$ of edges, array $\mathsf{s}(\cdot)$
Output: A partition $\mathcal{C} = \{C_1, \ldots, C_k\}$ of $G$

// Initialization
1  $T \leftarrow \emptyset$
2  $U \leftarrow V$
3  for $i = 1, \ldots, k$ do
4      $E_i \leftarrow \emptyset$
5      $C_i \leftarrow \emptyset$

// Main loop
6  while $U \neq \emptyset$ do
7      $u \leftarrow$ SelectNode($G$, $U$)
8      $T \leftarrow T \cup \{u\}$
9      $U \leftarrow U \setminus \{u\}$
       // Find the best cell for $u$
10     $s_{\min} \leftarrow \infty$
11     $i_{\min} \leftarrow$ null
12     $E' \leftarrow$ Dijkstra($\overline{G}, u$)
13     for $i = 1, \ldots, k$ do
14         $s \leftarrow 0$
15         forall $v \in V$ do
16             $s \leftarrow s +$ CountingDijkstra($(V, E_i \cup E'), C_i, v$)
17         if $s - \mathsf{s}(i) < s_{\min}$ then
18             $s_{\min} \leftarrow s - \mathsf{s}(i)$
19             $i_{\min} \leftarrow i$
20     $\mathsf{s}(i_{\min}) \leftarrow \mathsf{s}(i_{\min}) + s_{\min}$
21     $E_{i_{\min}} \leftarrow E_{i_{\min}} \cup E'$
22     $C_{i_{\min}} \leftarrow C_{i_{\min}} \cup \{u\}$
```

Since U contains n nodes at the beginning and one of them is extracted in every iteration, the loop is executed exactly n times. In all of these steps, the search-space size caused by assigning the node to any of the k cells is computed. To this end, the routine CountingDijkstra is executed n times in each of the k steps of the corresponding inner loop. Since the modified version of Dijkstra's algorithm has the same asymptotic time complexity as the original one presented in Chapter 2, each execution has costs in $\mathcal{O}(m + n \log n)$. In total, we obtain a time complexity that is in $\mathcal{O}(k \cdot (n^2 m + n^3 \log n)) \in \mathcal{O}(kn^4)$. The space consumption is dominated by the space required for storing the flagged edges and the sets of nodes. Observe that these edges are stored to construct the cell-induced subgraphs that are required for the function CountingDijkstra. Since at all times we have $|T| \leq n$, $|U| \leq n$ and $\sum_{i=1}^{k} |C_i| \leq n$, the overall space requirement is in $\mathcal{O}(km + n)$.

Selecting Nodes

So far we have not specified how the function selectNode used in Algorithm 6.2 chooses the next node to be processed. A naive deterministic procedure that works in $\mathcal{O}(1)$ time would be to pick nodes in ascending order according to their index. However, it appears that the order in which nodes are chosen has a large impact on the quality of the provided output of the algorithm GreedySequential. If nodes are simply selected with respect

Algorithm 6.3: COUNTINGDIJKSTRA

```
Input: Graph $G = (V, E, \omega)$, target node set $T$, source node $s$
Data: Priority queue Q
Output: Distances $\mathsf{d}(v)$ for all $v \in V$, search-space size $r$ of queries from $s$ to $T$
forall $v \in V$ do
    $\mathsf{d}(v) \leftarrow \infty$
    $\mathsf{pred}(v) \leftarrow$ null
Q.INSERT$(s, 0)$
$\mathsf{d}(s) \leftarrow 0$
$c \leftarrow 0$   // Counter for settled nodes
$r \leftarrow 0$   // Seach-space size
$t \leftarrow 0$   // Counter for settled nodes in $T$
while Q is not empty do
    $u \leftarrow$ Q.DELETEMIN()
    $c \leftarrow c + 1$
    if $u \in T$ then
        $r \leftarrow r + c$
        $t \leftarrow t + 1$
    forall $(u, v) \in E$ do
        if $\mathsf{d}(u) + \omega(u, v) < \mathsf{d}(v)$ then
            $\mathsf{d}(v) \leftarrow \mathsf{d}(u) + \omega(u, v)$
            $\mathsf{pred}(v) \leftarrow u$
            if Q.CONTAINS$(v)$ then
                Q.DECREASEKEY$(v, \mathsf{d}(v))$
            else
                Q.INSERT$(v, \mathsf{d}(v))$
$r \leftarrow r + (|T| - t) \cdot c$   // Search-space size for unreachable nodes
```

to their index, it seems easy to construct graphs for which the solution computed by Algorithm 6.2 is of poor quality. Instead, we propose a more sophisticated approach for picking the node that is to be processed next. It is based on the following intuition. Since especially the very first nodes that are assigned to cells have a large impact on the solution quality, one would want them to be spread evenly over the graph. A deterministic way of choosing nodes that have a preferably large distance from each other is given below. Initially, the node of maximum distance from any other node in the input graph G is picked. Then, following nodes are chosen to have the minimum distance from any processed node maximized. The proposed Algorithm 6.4 implements this strategy. It requires a unique precomputation of all distances within the graph, which induces additional computational costs in $\mathcal{O}(n^3)$. The algorithm itself clearly runs in $\mathcal{O}(n^2)$ time and requires a linear amount of storage. Thus, the asymptotic time and space complexities of Algorithm 6.2 do not increase when this approach is used to pick nodes of a graph.

6.2.2 Bounding the Approximation Ratio

The remainder of this section is devoted to showing that we cannot provide any constant quality assurance on the outputs produced by Algorithm 6.2, even if the nodes are processed in the rather sophisticated way described above.

Algorithm 6.4: SelectNode

```
Input: Graph $G = (V, E, \omega)$, set of nodes $U$
Data: Array minDist of integers
Output: A node $u \in U$
// Precondition: Distances between pairs of nodes in $V$ are known.
$max \leftarrow -1$
$r \leftarrow$ null
if $U \neq \emptyset$ then
    forall $t \in U$ do
        minDist($t$) $\leftarrow \infty$
        forall $s \in V \setminus U$ do
            minDist($t$) $\leftarrow \min\{$minDist($t$), $d(s,t)\}$
    forall $t \in U$ do
        if minDist($t$) $> max$ then
            $max \leftarrow$ minDist($t$)
            $r \leftarrow t$
else
    forall $s, t \in V$ do
        if $d(s,t) > max$ then
            $max \leftarrow d(s,t)$
            $r \leftarrow t$
return $r$
```

Again, we construct a graph for which the solution of the presented algorithm GreedySequential is of poor quality. For an adverse input instance, consider the undirected tree $G = (V, E, \omega)$ depicted in Figure 6.3. It contains a chain of x nodes with several leaves attached to both ends. First, there are y leaves attached to the leftmost node of the chain. All edges between this node and the leaves have a weight of 1. Edge weights on the chain are set to a small number ε. In particular, $(x-1)\cdot\varepsilon < 1$ shall hold. At the right end of the chain we have $k+1$ additional leaves. According edge weights connecting these leaves to the last node of the chain are supposed to be numbers W with $W > 3$. We divide the set V of nodes into two sets X and Y, where X contains all nodes of the chain plus the $k+1$ leaves that are incident to the rightmost node of the chain. Y holds the remaining nodes, i.e., the y leaves incident to the first node of the chain. In what follows, we examine the search-space sizes of different partitions on this tree.

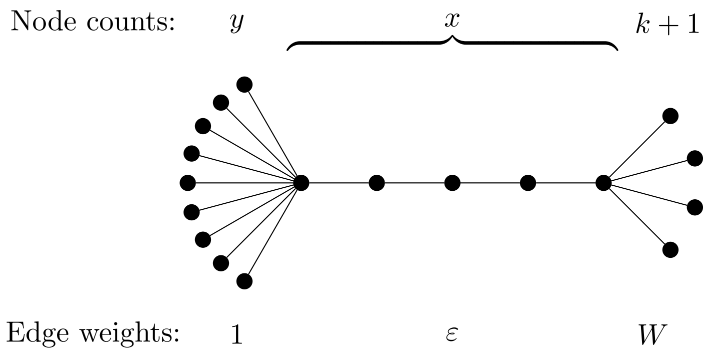

Figure 6.3: An adverse example for the presented greedy approach introduced above.

To begin with the analysis of the approximation ratio concerning the given adverse input instance, we present a technical lemma that is going to be useful during the rest of this section in order to bound occurring search-space sizes.

Lemma 6.2. *Let x, c and n be positive integers. There is a positive integer d such that $(x+c)^n \le 2x^n$ holds for all $x \ge d$.*

Proof. The Lemma follows immediately after a few basic transformations shown below.

$$\begin{aligned}(x+c)^n &= \sum_{k=0}^{n} \binom{n}{k} \cdot x^{n-k} \cdot c^k \\ &= x^n + \sum_{k=1}^{n} \binom{n}{k} \cdot x^{n-k} \cdot c^k \\ &\le x^n + d \cdot x^{n-1} && \text{for a } d \in \mathbb{N} \\ &\le 2x^n && \text{for all } x \ge d\end{aligned}$$

This proves the claim. □

Along the lines of Section 6.1, we proceed by deriving bounds on the optimal search-space size and on the partition computed by algorithm GreedySequential for a given maximum number of cells.

An Upper Bound on the Optimal Search-Space Size

First, we provide an upper bound on the optimal search-space size induced by a tree with the properties given above. Assume that the number of allowed cells is $k+1$ and that we choose the following partition $\mathcal{C}$. All nodes of X are put into one cell. The nodes in Y are divided equally over the k remaining cells. Without loss of generality, let $y = c \cdot k$ for a constant $c \in \mathbb{N}$, so that each of the k cells consists of exactly y/k nodes. We examine the search-space size induced by this partition and derive an upper bound depending on the graph size. In all what follows, assume that x, y and k are large enough for Lemma 6.2 to be applied. More precisely, we demand that $(z+3)^3 \le 2z^2$ holds for $z \in \{x+k, y/k\}$.

We separate the search space into four components and consider these components one by one. At first, we study the queries between nodes in X. Since the corresponding subgraph consists of a single cell that induces a strongly connected subgraph of T, we obtain a search-space size for queries within X that equals the search-space size of Dijkstra's algorithm given in Lemma 2.2.

$$\begin{aligned}S^{\mathcal{C}}_{X\to X} &= \sum_{s\in X, t\in X} \mathrm{S}_{\mathrm{AF}}(G,\mathcal{C},s,t) \\ &= (x+k+1)^2 \frac{x+k+2}{2} \\ &\le \frac{1}{2}(x+k+2)^3 \\ &\underset{\text{Lemma 6.2}}{\le} (x+k)^3 \end{aligned} \tag{6.4}$$

Next, we investigate the search-space size of queries within the set Y. Starting from an arbitrary node, the number of nodes one has to visit ranges from 3 to $(y/k)+2$ nodes for each target cell, depending on the target node index. Queries where the source and target node belong to the same cell form an exception with a search-space size between 1 and

$(y/k)+1$. Taking into account that there are y distinct source nodes and k possible target cells in Y, we get the following search-space size.

$$\begin{aligned} S^{\mathcal{C}}_{Y\to Y} &= \sum_{s\in Y, t\in Y} \mathrm{S}_{\mathrm{AF}}(G,\mathcal{C},s,t) \\ &= y\cdot k\cdot\left(\frac{\left(\frac{y}{k}+2\right)\left(\frac{y}{k}+3\right)}{2}-3\right)-\frac{y}{k}-1 \\ &\leq y\cdot k\cdot\frac{\left(\frac{y}{k}+3\right)^2}{2} \\ &\underset{\text{Lemma 6.2}}{\leq} y\cdot k\left(\frac{y}{k}\right)^2 \\ &= \frac{y^3}{k} \end{aligned} \tag{6.5}$$

For the search-space size caused by queries from nodes in Y to nodes in X, one obtains almost exactly the search-space size of Dijkstra's algorithm running on a strongly connected graph of size $|X|$ increased by 1 to account for the source node in Y. The only difference is that the number of distinct source nodes is y instead of $x+k+1$.

$$\begin{aligned} S^{\mathcal{C}}_{Y\to X} &= \sum_{s\in X, t\in X} \mathrm{S}_{\mathrm{AF}}(G,\mathcal{C},s,t) \\ &= y\left((x+k+2)\frac{(x+k+3)}{2}-1\right) \\ &\underset{\text{Lemma 6.2}}{\leq} y(x+k)^2 \end{aligned} \tag{6.6}$$

Finally, we turn to queries from nodes in X to nodes in Y. Considering the nodes from X that are settled in such queries, we see that for each of the y target nodes exactly the path from the source node s to the leftmost node of the chain is visited. On the other hand, we have $x+k+1$ distinct source nodes and each of them is responsible for settling 1 to y/k nodes for each of the k possible target cells. Hence, we get the search-space size shown below.

$$\begin{aligned} S^{\mathcal{C}}_{X\to Y} &= \sum_{s\in X, t\in X} \mathrm{S}_{\mathrm{AF}}(G,\mathcal{C},s,t) \\ &= y\left(\sum_{i=1}^{x} i+(k+1)(x+1)\right)+(x+k+1)\cdot k\cdot\sum_{i=1}^{y/k} i \\ &\leq y\cdot\frac{(x+k+1)(x+k+2)}{2}+(x+k+1)\cdot k\cdot\frac{\left(\frac{y}{k}\left(\frac{y}{k}+1\right)\right)}{2} \\ &\leq \frac{1}{2}\cdot y\cdot(x+k+2)^2+\frac{1}{2}\cdot(x+k+1)\cdot k\cdot\left(\frac{y}{k}+1\right)^2 \\ &\underset{\text{Lemma 6.2}}{\leq} y(x+k)^2+2(x+k)\frac{y^2}{k} \end{aligned} \tag{6.7}$$

Summing up Equations 6.4 to 6.7 yields the total search-space size. We further set $k=x$ and obtain the following upper bound on the optimal search-space size.

$$\begin{aligned} S_{\mathrm{opt}} &\leq S^{\mathcal{C}}_{X\to X}+S^{\mathcal{C}}_{Y\to Y}+S^{\mathcal{C}}_{Y\to X}+S^{\mathcal{C}}_{X\to Y} \\ &\leq (x+k)^3+\frac{y^3}{k}+y(x+k)^2+y(x+k)^2+2(x+k)\frac{y^2}{k} \\ &= 8x^3+8yx^2+4y^2+\frac{y^3}{x} \end{aligned} \tag{6.8}$$

A Lower Bound on the Search-Space Size Induced by the Greedy Approach

Now, assume that Algorithm 6.2 is used to compute a partition of G and that nodes are picked in an order determined by Algorithm 6.4. The first node that is selected is chosen to be one that has the largest possible distance to another node of the graph. Clearly, any of the $k+1$ leaves in X fulfills this requirement. So assume that one of them is assigned to an arbitrary cell first. The next node is selected to be the one with maximum distance from the first node. But the nodes at maximum distance from the first one are exactly the k remaining leaves in X with a distance of $2W$. Consequently, the $k+1$ leaves are assigned first and each of them is put into a distinct cell. We bound the resulting search-space size induced by such a partition $\mathcal{C}'$ in what follows. In contrast to the partition considered before, all edges that connect nodes on the chain pointing rightward now have all flags set to 1.

Again, we distinguish the same four components of the search-space size as before. For queries within X, we can safely assume that at least the whole s-t-path is settled in a query from a node $s \in X$ to a node $t \in X$. Using Lemma 2.3 yields the following search-space size, ignoring all queries involving any of the $k+1$ leaves.

$$\begin{aligned} S^{\mathcal{C}'}_{X\to X} &\geq \frac{1}{3}x^3 + x^2 - \frac{1}{3}x \\ &\geq \frac{1}{3}x^3 \end{aligned} \tag{6.9}$$

Next, we take account of queries where both nodes belong to Y. We assume that the nodes in Y are again balanced over k cells, which clearly yields the best solution for this case. Then, we know that for a distinct source node, the number of settled nodes in Y ranges from 1 to y/k for targets in the same cell. For targets not in the cell of the source node, this number ranges from 2 to $y/k+1$. In addition to that, all x nodes of the chain must get settled in every single query since the corresponding edges have small weights and all their flags are set to 1. In total, we can bound the search-space size as follows.

$$\begin{aligned} S^{\mathcal{C}'}_{Y\to Y} &\geq y \cdot k \cdot \left(\frac{\left(\frac{y}{k}\right)\left(\frac{y}{k}+1\right)}{2} \right) + y(y-1)x \\ &\geq y \cdot k \cdot \frac{1}{2}\left(\frac{y}{k}\right)^2 + \frac{1}{2}y^2x \\ &= \frac{1}{2}\frac{y^3}{k} + \frac{1}{2}y^2x \end{aligned} \tag{6.10}$$

We can bound the overall search-space size of queries from Y to X with the same argumentation as for Equation 6.6 above, except for the fact that at most one of the leaves in X needs to be settled, as all leaves are assigned to distinct cells.

$$\begin{aligned} S^{\mathcal{C}'}_{Y\to X} &\geq y\left(\sum_{i=2}^{x+1} i + k(x+2)\right) \\ &= y \cdot \frac{1}{2}(x+1)(x+2) - 1 + yk(x+2) \\ &\geq \frac{1}{2}yx^2 + yxk \end{aligned} \tag{6.11}$$

Ultimately, for queries from nodes in X to nodes in Y we know that at least the whole chain must get settled every time. This is again due to the fact that the corresponding flags are set and edge weights were chosen to be small.

$$S^{\mathcal{C}'}_{X\to Y} \geq yx^2 \tag{6.12}$$

From Equations 6.9 to 6.12 directly follows the lower bound on the obtained search-space size given below. Again we set the number of cells to be $k = x$.

$$\begin{aligned} S_{\text{greedy}} &\geq S^{\mathcal{C}'}_{X\to X} + S^{\mathcal{C}'}_{Y\to Y} + S^{\mathcal{C}'}_{Y\to X} + S^{\mathcal{C}'}_{X\to Y} \\ &\geq \frac{1}{3}x^3 + \frac{1}{2}\frac{y^3}{k} + \frac{1}{2}y^2x + \frac{1}{2}yx^2 + yxk + yx^2 \\ &= \frac{1}{3}x^3 + \frac{1}{2}\frac{y^3}{x} + \frac{1}{2}y^2x + \frac{5}{2}yx^2 \end{aligned} \tag{6.13}$$

The Resulting Approximation Ratio

To obtain a bound on the approximation ratio, we use the bounds on S_{opt} and S_{greedy} stated in Equations 6.8 and 6.13. Furthermore, we demanded that $y = c \cdot k = c \cdot x$ for a constant $c \in \mathbb{N}$. Altogether, we find the following approximation ratio r for Algorithm 6.2.

$$\begin{aligned} r &\geq \frac{S_{\text{greedy}}}{S_{\text{opt}}} \\ &\geq \frac{\frac{1}{3}x^3 + \frac{1}{2}\frac{y^3}{x} + \frac{1}{2}y^2x + \frac{5}{2}yx^2}{8x^3 + \frac{y^3}{x} + 8yx^2 + 4y^2} \\ &= \frac{\frac{1}{3}x^3 + \frac{1}{2}c^3x^2 + \frac{1}{2}c^2x^3 + \frac{5}{2}cx^3}{8x^3 + c^3x^2 + 8cx^3 + 4c^2x^2} \\ &= \frac{\left(\frac{1}{2}c^2 + \frac{5}{2}c + \frac{1}{3}\right)x^3 + \frac{1}{2}c^3x^2}{(8c+8)\,x^3 + (c^3 + 2c^2)x^2} \end{aligned} \tag{6.14}$$

We know that the number of nodes is $n = y + x + k + 1 = (c+2)x + 1$, where c is a constant natural number. As n tends to infinity, so does x and hence the bound on the approximation ratio given in Equation 6.14 converges to

$$\delta = \frac{\frac{1}{2}c^2 + \frac{5}{2}c + \frac{1}{3}}{8c+8}.$$

We can choose c arbitrarily in $\mathbb{N}^+$ and clearly δ is divergent for large c. Thus, we can achieve any constant upper bound $\delta \in \mathbb{R}^+$. This proves that Algorithm 6.2 does not yield any constant approximation ratio.

Other Techniques to Determine the Order of Processed Nodes

One might consider to slightly modify the node-selection routine given by Algorithm 6.4 in order to provide a deterministic method that copes well with the input instance we inspected above. For example, distances in Algorithm 6.4 could be measured in the number of hops, i.e., edge weights are ignored during node selection. This results in a more favorable order of nodes for the adverse tree we constructed above. However, a simple modification of this graph infers the same order of selected nodes as above for the adjusted procedure. All one has to do is remove the outgoing edges of the $k+1$ leaves in X at the rightmost node of the chain. Note that edges pointing in the opposite direction remain unchanged. Moreover, we add an edge from each leaf in X to an arbitrary leaf in Y with weight 1. This implies that starting at one of the $k+1$ leaves in X, the node with maximum hop distance is again one of the remaining leaves of X. Hence, the greedy algorithm creates a partition that is similar to the one we examined above, which results in an equally bad approximation ratio.

It appears likely that any deterministic approach that chooses the next processed node and is efficiently computable can be countered by an input instance that induces a bad partition. Another idea thus might be to pick nodes at random. Since it seems quite unlikely that the resulting order induces a bad partition, this might produce good solutions with a high probability. Whether or not randomization can be used to establish a probabilistic bound on the approximation ratio of Algorithm 6.2 remains an open question.

6.3 Case Study

We close this chapter with a brief experimental setup. Note that the case study presented here is supposed to give a rough idea of the performance of the approaches introduced in Sections 6.1 and 6.2 rather than an exhaustive evaluation. Both greedy algorithms presented above were implemented naively without any additional engineering or parallelization, although there appear to be many possible improvements with positive effects on running times. For example, the algorithms include loops of independent computations that one could perform in different threads to save time, such as the tentative assignment of a node to each cell in Algorithm 6.2. All implementations were compiled with Java 1.6. To compute optimal search-space sizes, we implemented the stopping-criterion aware MILP introduced in Chapter 5, including all tuning possibilities mentioned there. We used Gurobi 4.0 as a black-box solver [GRB10].

We examine three different types of graphs for this case study. Namely, partitions were generated for excerpts of a real street network. Moreover, we performed experiments on unit-disk graphs, which were generated by placing a fixed number of nodes into a squared plane, with their positions picked uniformly at random. Nodes were connected by an edge if and only if their Euclidean distance fell below a specified value. This value was chosen such that the average degree of each node was about 10. Finally, grids of a fixed size were used, where a node was placed on each discrete x- and y-coordinate for all $x, y \leq c$ given a constant c and connected to its up to four neighbors via an edge. Edge weights for both the unit-disk graphs and the grids were integers chosen uniformly at random from the set $\{1, \ldots, 1000\}$. Instances of different sizes were generated for each type of graph and each time, different parameters k indicating the number of allowed cells were tested.

Experiments concerning Algorithms 6.1 and 6.2 were performed on a machine equipped with a dual-core AMD Opteron processor 2218 clocked at 2.6 GHz and 32 GB RAM running SUSE Linux 11.3. The linear programs were executed on up to 48 cores of AMD Opteron processors 6172 clocked at 2.1 GHz, provided with 256 GB RAM on a machine also running SUSE Linux 11.3. To compare the quality of both greedy algorithms to the optimal solution, all three approaches were performed on small graphs containing at most 40 nodes. Furthermore, the algorithms GreedyMerge and GreedySequential were tested on larger graphs for more detailed results and to get a first impression of their scalability.

The results of our experiments are listed in Tables 6.1, 6.2 and 6.3. For each run, we list the running time of the corresponding algorithm in seconds as well as the search-space size $\mathrm{S}_{\mathrm{AF}}(\mathcal{C})$ induced by the constructed partition $\mathcal{C}$. Input graphs that represent a part of the street network of the city of Karlsruhe are denoted by G_{Ka}. By G_{unit} and G_{grid}, we denote the generated unit-disk and grid graphs, respectively. In particular, we consider the following input instances.

- Small instances: G^S_{Ka} ($n = 40$, $m = 61$), G^S_{unit} ($n = 32$, $m = 312$) and G^S_{grid} ($n = 25$, $m = 80$).
- Medium instances: G^M_{Ka} ($n = 795$, $m = 1\,652$), G^M_{unit} ($n = 500$, $m = 5\,598$) and G^M_{grid} ($n = 484$, $m = 1\,848$).
- Large instances: G^L_{Ka} ($n = 2\,206$, $m = 4\,693$), G^L_{unit} ($n = 2\,000$, $m = 20\,144$) and G^L_{grid} ($n = 1\,936$, $m = 7\,568$).

We see that both greedy approaches generate partitions of similar quality. The algorithm providing the better output depends on both the input graph and the number of cells, so neither approach outperforms the other in terms of solution quality.

Table 6.1: Performance on small instances

		Merge		Sequential		Linear Program	
Graph	k	Time	S_{AF}	Time	S_{AF}	Time	S_{AF}
G^S_{Ka}	2	0.049 s	25 293	0.307 s	26 525	56 263 s	22 301
	3	0.049 s	18 628	0.320 s	17 693	453 265 s	17 460
	4	0.050 s	17 932	0.343 s	16 833	159 703 s	15 027
G^S_{unit}	2	0.094 s	11 502	0.227 s	11 771	2 321 s	11 454
	3	0.095 s	9 399	0.245 s	9 690	50 690 s	9 232
	4	0.094 s	8 765	0.264 s	8 453	67 671 s	7 996
G^S_{grid}	2	0.031 s	5 861	0.162 s	5 697	283 s	5 627
	3	0.031 s	4 538	0.173 s	4 538	1 833 s	4 479
	4	0.031 s	4 252	0.188 s	3 977	8 775 s	3 977

Table 6.2: Performance on medium instances

		GreedyMerge		GreedySequential	
Graph	k	Time	S_{AF}	Time	S_{AF}
G^M_{Ka}	2	38.853 s	153 270 108	559.436 s	152 372 435
	4	38.989 s	99 664 387	853.127 s	103 071 795
	8	38.658 s	63 414 459	1 671.261 s	62 208 282
G^M_{unit}	2	63.550 s	45 707 575	146.449 s	43 360 751
	4	66.091 s	28 125 277	188.373 s	27 798 149
	8	62.295 s	18 105 421	326.675 s	18 619 313
G^M_{grid}	2	15.977 s	42 095 431	113.355 s	35 930 872
	4	16.127 s	25 494 693	140.917 s	24 076 881
	8	16.041 s	16 610 479	211.196 s	16 218 291

Table 6.3: Performance on large instances

		GreedyMerge		GreedySequential	
Graph	k	Time	S_{AF}	Time	S_{AF}
G^L_{Ka}	2	548 s	3 109 218 136	19 035 s	3 208 415 108
	4	541 s	2 231 385 054	22 125 s	1 959 477 358
	8	536 s	1 287 719 872	29 482 s	1 317 165 206
G^L_{unit}	2	2 878 s	2 516 844 936	22 217 s	2 630 583 944
	4	2 709 s	1 569 466 774	31 263 s	1 636 507 220
	8	2 985 s	1 032 627 816	34 575 s	991 322 491
G^L_{grid}	2	649 s	2 439 891 488	18 535 s	2 498 003 414
	4	611 s	1 527 244 192	19 919 s	1 419 332 997
	8	635 s	987 332 678	26 110 s	911 282 505

Furthermore, the results shown in Table 6.1 suggest that the partitions computed by the greedy approaches are not too far from the optimum on common graphs. However, our implementation of the algorithm GreedyMerge was running considerably faster than GreedySequential on all instances.

Note that in each case, we only recorded the running time of a single execution of the respective algorithm, even though we experienced variability between several runs of the same algorithm given the same input. Hence, running times listed in Tables 6.1, 6.2 and 6.3 are only supposed to give an impression of the performances of the presented approaches rather than providing the basis for a detailed analysis.

Besides the comparatively slow running times of both approaches, we observe that their space consumption is extensive as well. In particular, on the largest instances up to 1 and 5.8 GB of RAM were acquired by the algorithms GreedyMerge and GreedySequential, respectively. This renders their basic versions prohibitive for realistic instances containing millions of nodes. However, elaborate engineering of the naive implementations used in our experiments or combinations with ingredients from established approaches may provide more sophisticated but practical techniques.

7. Node-Sensitive Arc-Flags for Encoding All-Pairs Shortest Paths

We study a new approach for speeding up Dijkstra's algorithm that is based on arc-flags. Imagine that instead of one single partition, we are allowed to choose a distinct partition for each node. We shall refer to this modification of the original approach as node-sensitive arc-flags. On the downside, such node-sensitive flags require additional amount of space because up to n different partitions need to be stored. However, this approach surely yields further speed-up in comparison to the original arc-flags algorithm. In fact, it turns out that node-sensitive arc-flags can be used to encode shortest paths between all pairs of nodes in a given graph. More precisely, given node-sensitive arc-flags, the query algorithm settles exactly the nodes on a shortest s-t-path for arbitrary nodes s and t. We say that a speed-up technique encodes all-pairs shortest paths if this holds.

In the following Section 7.1 we examine node-sensitive arc-flags and their applicability for storing the information for all-pairs shortest paths with relatively low memory consumption. Rather than giving formal studies and guarantees, this section is supposed to provide the general ideas of our approach. Evaluation of the practical usefulness of node-sensitive arc-flags would be part of future work. In Section 7.2, we further abstract from the idea of node-sensitive arc-flags in order to deduce ways to efficiently store information necessary to encode all-pairs shortest paths on restricted graph classes, especially trees.

Throughout this chapter, for any graph $G = (V, E, \omega)$ we assume that numbers in $\mathcal{O}(n)$ can be stored at constant costs, that is, numbers that require at most $\mathcal{O}(\log n)$ bits to represent account for constant space consumption. This is a common precondition in the unit-cost RAM model [AHU74].

7.1 The Concept of Node-Sensitive Arc-Flags

Below, we modify the arc-flags algorithm presented in Chapter 3. In the original approach, for a given graph $G = (V, E, \omega)$, a partition $\mathcal{C} = \{C_1, \ldots, C_k\}$ of the set of nodes V is computed in advance to obtain feasible arc-flags. In case of node-sensitive flags, however, each node $v \in V$ gets its own partition $\mathcal{C}_v = \{C_{v_1}, \ldots, C_{v_k}\}$. This partition and the resulting flags are precomputed and stored for every node in V. When the query algorithm settles a node, its corresponding flags are used to prune the search at any edge that does not have the target flag set to 1. Clearly, this approach may demand orders of magnitudes of additional memory compared to original arc-flags approach. On the other hand, one would expect query times to considerably improve. In what follows, we examine the performance and the memory consumption of node-sensitive arc-flags.

Optimal Partitions for Node-Sensitive Arc-Flags

For now, let us assume that the number of cells k is greater or equal to the maximum out-degree of any node in a given graph $G = (V, E, \omega)$. We obtain optimal cell assignments for an arbitrary node u as follows. Using Dijkstra's algorithm, we compute the shortest-path tree T_u that is rooted at u. Then, for every edge (u, v) incident to u that belongs to the shortest-path tree, we assign v and all nodes reachable from v in T_u to a distinct cell C_{u_v}. See Figure 7.1 for an illustration. This implies that for every outgoing edge, exactly one flag is set and this flag represents a cell containing nodes reachable via this edge on a shortest path. Furthermore, if a shortest path is not unique, only one edge leading to the target node gets the corresponding flag set to 1. Hence, a query algorithm based on node-sensitive flags settles exactly the nodes that lie on a single shortest path to the target node. The only modification necessary in the query algorithm we presented in Chapter 3 to cope with node-sensitive flags is a routine that updates the node-dependent cell membership of the target node every time a new node is settled.

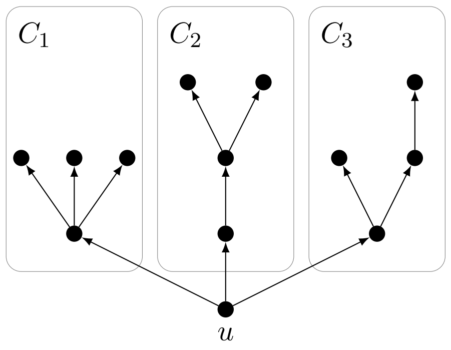

Figure 7.1: Shortest-path tree of a node u and corresponding node-sensitive cells.

As for the space requirement of this approach, observe that we do not need to save the binary flags explicitly. Instead, storing the specified cell for each node suffices for the query algorithm to find the corresponding edge. Note that this node-dependent partition must be stored anyway. Furthermore, there no longer is any need to bound the maximum number of allowed cells per node. Assigning n nodes to cells requires space in $\mathcal{O}(n)$ for each of the n nodes, so the overall space complexity is in $\mathcal{O}(n^2)$. Note that explicitly storing all n^2 shortest paths would require space in $\mathcal{O}(p \cdot n^2)$, where p is the maximum size of a shortest path in G.

In total, we see that we obtain a shortest path $P_{s,t}$ between two given nodes s and t in time $\mathcal{O}(|P_{s,t}|)$. By forcing Dijkstra's algorithm to prefer hop-minimal shortest paths in the preprocessing phase, we can even assure that the size of the returned path $P_{s,t}$ is minimal. However, it is clear that the quadratic space requirements for storing the preprocessed information are prohibitive for large-scale graphs. Therefore, we propose different ideas for reducing the memory consumption of node-sensitive arc-flags. Since we are facing a trade-off between fast query times and low memory consumption, a practical evaluation of these approaches would be necessary to determine the best one.

Reducing Space Requirements

Imagine a realistic large graph containing millions of nodes. If we consider the node-dependent partitions of different nodes, it seems likely that nodes within a small region of the graph possess similar partitions. This leaves room for engineering in order to reduce the space requirements, although it seems difficult to provide asymptotic guarantees in general. Here, we propose the following approach. We define a small set $V_B \subseteq V$ of basic nodes and store the node-sensitive partitions of all nodes in V_B explicitly. Given a node $v_B \in V_B$, for nearby nodes in $v \in V \setminus V_B$ we only store the differences between the partitions $\mathcal{C}_{v_B}$ and $\mathcal{C}_v$, i.e., all information that is sufficient to generate the node-dependent partition of v given the partition $\mathcal{C}_{v_B}$. One would expect that this is far less costly than storing an explicit partition for v as well. Clearly, implementing this approach increases the work of the query algorithm, for it must restore the cell information of all nodes other than the basic nodes on-the-fly. However, one might save a large amount of memory this way. Moreover, the size of the set V_B may serve as a tuning parameter that realizes a trade-off between space consumption and query times.

The original arc-flags algorithm and the node-sensitive approach can be interpreted as two extreme points of a continuous scale that trades time for space. While original arc-flags only need additional space that is linear in the graph size, node-sensitive flags require a quadratic amount of space but guarantee an optimal search-space size if one assumes that the whole path must get settled by the query algorithm. Hence, one could create intermediate levels between these two approaches by allowing distinct partitions for sets of nodes. The sizes of these sets then balance the time and space requirements of the algorithm. Observe that in case of original arc-flags we have one set of size n, whereas node-sensitive flags permit n sets of size 1. Obviously, small sets induce extensive memory consumption and low query times, whereas larger sets of nodes that share a partition reduce space requirements at the cost of worse expected search-space sizes. Before, we have seen that it is easy to find optimal partitions for node-sensitive arc-flags if each node gets its own partition. Consequently, the question arises whether there is a way to efficiently obtain optimal cells if several nodes share the same partition. This problem appears much more difficult than the single-node case described above, even if we restrict the number of nodes per set to two. The following problems arise in this scenario.

- One needs to determine which pairs of nodes should be taken into the same sets in order to achieve globally optimal solutions.
- An optimal partition with respect to several nodes depends on the number of s-t-queries that settle each node of the considered set. In particular, one might want to concentrate on finding a partition that rather serves nodes that are settled more frequently.
- Even if we restrict ourselves to an optimization of the partition for two given nodes and only optimize the search-space sizes of queries starting at these nodes, there is no obvious way of how to assign cells. This is due to the fact that in general, the shortest-path trees of both nodes have overlapping sets of reachable nodes corresponding to their outgoing edges.

In summary, practical experiments seem promising in order to prove the usefulness of our approach rather than a theoretical study. Further analysis of node-sensitive arc-flags including experimental evaluation are left as future work.

7.2 Encoding All-Pairs Shortest Paths on Restricted Graph Classes

The technique of encoding all-pairs shortest paths using node-sensitive arc-flags introduced above can be seen as an abstract view of arc-flags. Instead of flag vectors, partitions of V are used to store all necessary information for obtaining shortest paths. The question arises what information is needed to ensure that only nodes of the actual shortest path are settled by a query algorithm, especially if we consider restricted classes of graphs. In what follows, we derive simple approaches for the restricted classes that were examined in Chapter 4, namely paths, cycles and trees.

Paths

First of all, we take into account graphs $P = (V, E, \omega)$ that consist of a single undirected path. Precomputing a linear amount of information to encode all-pairs shortest paths is simple in this case. Starting at one of the two distinct nodes in V with an out-degree of 1, we simply traverse all nodes of the graph and assign ascending numbers to each of them. Then, the query algorithm only has to compare the assigned numbers of the given source and target node. This unambigously determines the direction that leads to the target node and only those nodes on the unique path from the source to the target get settled in a query. An analogous approach encodes all-pairs shortest paths on directed paths.

Cycles

Next, think of a graph $Z = (V, E, \omega)$ that is an undirected cycle. As explained in the corresponding section of Chapter 4, a shortest-path tree of an arbitrary node s in a cycle consists of directed edges corresponding to all but one undirected edge in E. The unique undirected edge that is not present in this tree provides all necessary information for a query from s to only settle nodes of a shortest path to a given target node.

Based on this idea, we describe how to establish a query algorithm encoding all-pairs shortest paths on cycles. Again, we assign ascending indices i_v to all nodes $v \in V$ by setting $i_w = 1$ for an arbitrary node w and iteratively assigning an index to a yet unvisited neighbor such that nodes with indices j and $j + 1$ are connected via an edge until w is reached again. Then, we obtain a shortest path as follows. For each node $s \in V$, given the unique pair of edges $\{(u, v), (v, u)\}$ that is not part of the precomputed shortest-path tree rooted at s, we store the according indices of u and v. Together with the index of s, we are given all information necessary for the query algorithm to choose the right direction. To avoid case distinctions, assume that $i_u < i_v$ and $i_s < i_u$. In a query to a target node t, the unique neighbor of s with the index $i_s + 1$ is settled if and only if $i_s < i_t < i_u$. If $i_t > i_u$ or $i_t < i_s$, the other neighbor is settled. The initiated path is then followed until the target node is reached. Other cases work analogously.

Unique Paths on Trees and Least Common Ancestors

Finally, we propose an approach for directed and undirected trees to encode all-pairs shortest paths after a linear-time preprocessing step and with a linear amount of auxiliary data needed to be stored. Related approaches have been published in the past to obtain least common ancestors in a rooted tree, i.e., given two nodes u and v, one asks for the unique node w farthest from the root r such that w is in both the path from r to u and from r to v. An algorithm that computes the least common ancestor of arbitrary nodes in a tree in $\mathcal{O}(1)$ after linear-time preprocessing was first presented by Harel and Tarjan [HT84]. Further information on this problem and simplifications of the original approach are provided by Schieber and Vishkin [SV88] and presented in a book by Gusfield [Gus07].

However, to the best of our knowledge, the problem of encoding all-pairs shortest paths on trees has not yet been studied explicitly. Clearly, if one can compute the least common ancestor of two nodes in a roooted tree in constant time, this can be used to obtain the unique path between two nodes by simultaneously starting at both the source and the destination node and following the unique edges that lead towards the root of the tree until the least common ancestor is reached. Though, the mentioned algorithm to compute the least common ancestor is rather complex. Instead, we now present an alternative approach for our purposes. It is based on a simpler precomputation step and provides a straightforward technique to obtain arbitrary shortest paths.

A First Approach for Encoding All-Pairs Shortest Paths on Trees

Given an undirected or directed tree, the precomputation phase works as follows. If the tree is undirected, an arbitrary node is defined to be its root node. Starting at the root, one assigns indices in depth-first-search order to all nodes of the tree, that is, recursively assign ascending numbers to the children of the node and finally to the node itself. In addition to that, the number of nodes belonging to the subtree rooted at each encountered node is computed and stored. Both tasks can be fulfilled in a single run of a modified depth-first search. Since a depth-first search has a time complexity in $\mathcal{O}(n+m) = \mathcal{O}(n)$, this preprocessing step can be performed in linear time [CLRS01].

An s-t-query in a preprocessed undirected tree then works as follows. When a node u is settled, we compare the target-node index to the indices of all neighbors of u. This is done by checking outgoing edges in ascending order of their head indices. Since node indices were determined in a depth-first search, there is at most one edge pointing at a node with an index smaller than the index of u. First, this node is skipped and the query algorithm checks each of the remaining nodes. Let i_t be the index of the target node, i_v the index of the head of an edge (u, v) and n_v the size of the corresponding subtree rooted at v. The edge (u, v) is part of the unique s-t-path if and only if $i_v \leq i_t \leq i_v + n_v$. Hence, the head of this edge is settled if and only if this condition holds. If none of the inspected edges fulfill this requirement and $u \neq t$, the unique edge that points towards the root must lie on the path to the target node and hence is to be settled next or, if the tree is directed, the target is unreachable. This way we can ensure that the query algorithm settles only nodes that belong to the path from s to t. Clearly, this only requires a memory overhead that is linear in n. Let $P_{s,t}$ be the unique s-t-path and $\delta = \max_{v \in V} \operatorname{out}(v)$ the maximum out-degree of T. Since only the nodes of the s-t-path are settled and for each node we have to check at most all its incident edges, the overall complexity of an s-t-query is in $\mathcal{O}(\delta \cdot |P_{s,t}|)$.

Further Improvements of Query Times

In what follows, we slightly modify the preprocessing phase to improve the query complexity obtained above. Assume we are given a rooted tree $T = (V, E, \omega)$ with root node r. Let $\ell(v) = h(r) - |P_{r,v}|$ denote the *level* of v, defined as the height of the root r minus the hop-distance between r and v. As a first step of the preprocessing phase, we compute the values $\ell(v)$ of every node $v \in V$ and the maximum out-degree δ that occurs in the rooted tree. This can be done by performing two additional depth-first search at the beginning, the first of which computes $h(r)$ before the mentioned values $\ell(v)$ and δ are obtained in the second run. Next, we are going to make the tree balanced and δ-ary by temporarily adding dummy nodes and edges to it. We do this during a third depth-first search, in which we also assign the depth-first search indices to all encountered nodes. For any node v with $\ell(v) > 0$ and $\operatorname{out}(v) < \delta$, we attach dummy subtrees $T_{i_v}, 1 \leq i \leq \delta - \operatorname{out}(v)$ to v by adding an edge from v to the root of each subtree. All added subtrees are δ-ary balanced trees of height $\ell(v) - 1$, and they are directed trees if and only if T is directed. Figure 7.2 shows an example of a tentatively created new tree T' that is obtained this way, where all white nodes belong to dummy subtrees. Depth-first search indices are now assigned according to the graph T'. For example, for the graph depicted in Figure 7.2, we obtain the indices $2, 3, 6, 7, 8, 9$ for the nodes v_1 to v_6, respectively. Since we can safely bound the number of added nodes to $\delta \cdot n$ per node, the created tree T' contains no more than $\delta \cdot n^2 \leq n^3$ nodes.

Consequently, the preprocessing can still be performed in polynomial time and the computed numbers require at most $\log n^3 \in \mathcal{O}(\log n)$ bits. Finally, all tentatively added nodes are removed and the indices as well as the values $\ell(v)$ of all remaining nodes are stored. Note that in fact, the dummy nodes do not have to be added to the input tree explicitly, since they are only needed to obtain the corresponding depth-first search indices.

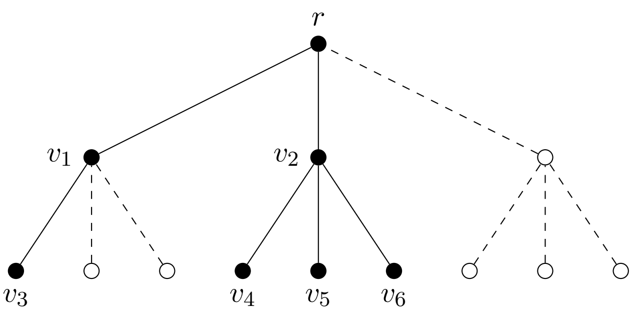

Figure 7.2: The modified tree corresponding to an undirected rooted tree containing the nodes $\{r, v_1, \ldots, v_6\}$.

Consider an arbitrary node u with $\ell(u) = \lambda$. Since the modified tree on which the indices were computed was balanced, we know that the size of the subtree rooted at u in T' is exactly $\sum_{i=0}^{\lambda}(\delta - 1)^i$ if T' is undirected and $\sum_{i=0}^{\lambda} \delta^i$ otherwise. In what follows, we only consider the first case for the sake of simplicity. The query algorithm then works as follows. If a node u with index i_u is settled, we know that the next node is in the subtree rooted at u if and only if $i_u \leq i_t \leq i_u + \sum_{i=0}^{\lambda}(\delta - 1)^i$. Otherwise, the target node is either unreachable if the tree is directed or the next node is the unique predecessor of u on the path to r. If the target node $t \neq u$ lies within the subtree rooted at u, it is $i_t > i_u$ and the next node must have the following index i_v.

$$i_v = i_u + 1 + \left\lfloor \frac{i_t - i_u - 1}{\sum_{i=0}^{\lambda-1}(\delta - 1)^i} \right\rfloor \cdot \sum_{i=0}^{\lambda-1}(\delta - 1)^i$$

The predecessor of a given node can either be determined in another calculation or stored explicitly for each node. Now, assume that for each node v instead of $\ell(v) = \lambda$ we store the value of $\sum_{i=0}^{\lambda-1}(\delta - 1)^i$. Then, for each settled node we find the next node on the unique s-t-path after a case distinction and a single calculation. Altogether, this enables us to find any path $|P_{s,t}|$ of a tree and its distance in $\mathcal{O}(|P_{s,t}|)$ with a very small linear factor after linear preprocessing and with linear memory overhead. Clearly, the directed case works analogously.

An interesting open question is whether this result can help in obtaining a comparable result for directed acyclic graphs. As for finding the least common ancestor, a way to find it in constant time even for directed acyclic graphs was proposed by Bender et al. [BFCP+05]. A similar result for encoding all-pairs shortest paths on directed acyclic graphs would be crucial, since they are of high relevance for routing in practice [PSWZ07].

8. Conclusion

We close our observations with concluding remarks. At first, we summarize the results obtained in this work, before we turn to future work based on the outcomes and open questions we proposed throughout this thesis.

8.1 Summary

In this thesis, we considered several theoretical aspects of filling the degree of freedom during preprocessing of the arc-flags algorithm, namely the specification of a partition for a given graph. The results of our studies are summarized below.

The basis of our work was provided by the hardness result of the problem ArcFlagsPartition on graphs in general. We refined this result and provided further steps on the way of finding a border of tractability by considering restricted graph classes. We showed that optimal cells can be obtained in polynomial time for paths, stars and supposedly for cycles. Additionally, the $\mathcal{NP}$-hardness of computing a search-space optimal partition on undirected trees with uniform edge weights was proven. This result provides a significant refinement of the known proof of hardness regarding general graphs. Furthermore, we conjecture that the problem is hard on directed acyclic graphs and even on trees with a maximum out-degree that is at most 3. Hence, provided that $\mathcal{P} \neq \mathcal{NP}$, only severely restricted graph classes seem to allow an efficient computation of optimal cells in general.

Since polynomial-time algorithms that generate optimal cells for arbitrary graphs are unlikely to exist, we introduced several approaches aiming at the computation of optimal or high-quality solutions. First of all, linear programs yielding optimal partitions were established. Depending on the problem specification, different possible approaches were derived. Another important step in dealing with hard problems is the design and analysis of approximation algorithms, i.e., algorithms that guarantee worst-case bounds on their relative solution quality. One general approach is to analyze a dual linear program in order to gain further insights into the structure of the primal program or even obtain bounds for an approximation algorithm. Unfortunately, this approach does not appear to be promising in our case. Another idea is to consider combinatorial algorithms with regard to the establishment of approximation bounds for them. To this end, we introduced two novel greedy approaches. Their capability was reviewed in a brief case study. However, we showed that neither of the greedy algorithms provides a constant approximation ratio in terms of minimal average search-space size.

Finally, we relaxed the original idea of arc-flags and introduced a node-sensitive variant in which each node possesses its own partition. It turns out that this interpretation yields a way to encode the shortest paths between all pairs of nodes at impractical quadratic memory costs. However, this approach might serve as a starting point for new algorithms that deal with the encoding of all-pairs shortest paths at comparatively low costs. Furthermore, we discussed several methods to encode all-pairs shortest paths in certain restricted graph classes, such that the path itself can be obtained efficiently after a linear-time preprocessing phase.

8.2 Outlook

Although many results were presented for all aspects considered throughout this work, a wide-ranging basis for future work in the domain of preprocessing arc-flags and even route planning in general was provided as well.

Despite the correctness of the result of $\mathcal{NP}$-hardness concerning undirected trees, one may seek to find ways to simplify the yet very extensive proof. For example, one might attempt to construct a proof that reduces given instances of 3-Partition to simpler instances of undirected trees, such as $(m, B, 1)$-trees. Considering the problem ArcFlagsPartition on further restricted classes of trees and directed acyclic graphs, we proposed conjectures that these problems are $\mathcal{NP}$-hard as well. Proofs of these conjectures, however, are yet to be completed formally. We presented a brief sketch of an approach to optimally assign cells in undirected cycles, but again a formal analysis is to be conducted and the proof of a necessary condition needs to be found. Furthermore, it would be interesting to know if there exist restricted classes of graphs other than those we proposed for which optimal cells are efficiently computable.

The dual ILP presented in Chapter 5 remains subject to a deeper analysis, although it appears unlikely that further insights may be gained from such an examination. The primal linear program, on the other hand, could be analyzed with regard to further tuning. Moreover, the greedy algorithms proposed in Chapter 6 could be engineered or parallelized to render them applicable for large-scale graphs. Yet another approach might be to introduce ingredients from established preprocessing techniques for further tuning. However, one would have to find ways to reduce the memory consumption, which is prohibitive for the plain greedy approaches. Furthermore, a more sophisticated experimental setting would be necessary for a deeper analysis of the capability of both algorithms. Especially, it would be interesting to compare the resulting search-space sizes of the greedy approaches to those of heuristics that are used in practice. Finally, since the inability of both greedy algorithms to guarantee a constant bound on the solution quality was shown, another interesting question would be if one can prove inapproximability of the problem ArcFlagsPartition with respect to certain bounds.

Finally, the provided ideas for space-efficient implementations of node-sensitive arc-flags given in Chapter 7 yield only a rough idea of their capability. An experimental evaluation of this approach would provide further insights into this topic. Furthermore, we posed the question whether there is any efficient way to encode all-pairs shortest paths for directed acyclic graphs.

Bibliography

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In Moses Charikar, editor, *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 782–793. SIAM, 2010.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Boston, MA, USA, 1974.

[AMZ09] Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors. *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*. Springer, 2009.

[Bau10] Reinhard Bauer. *Theory and Engineering for Shortest Paths and Delay Management*. PhD thesis, Department of Informatics, Karlsruhe Institute of Technology (KIT), December 2010.

[BCK$^+$10] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing Speed-Up Techniques is Hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[BD09] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.

[BDD09] Emanuele Berretini, Gianlorenzo D'Angelo, and Daniel Delling. Arc-Flags in Dynamic Graphs. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, OpenAccess Series in Informatics (OASIcs), 2009.

[BDGW10] Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner. Space-Efficient SHARC-Routing. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 47–58. Springer, May 2010.

[BDS$^+$10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[BDW11] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 57(1):38–52, January 2011.

[BFCP$^+$05] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest Common Ancestors in Trees and Directed Acyclic Graphs. *J. Algorithms*, 57:2005, 2005.

[BFM$^+$07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[Bro96] Andrew Browder. *Mathematical Analysis: An Introduction.* Undergraduate Texts in Mathematics. Springer, New York, NY, USA, 1996.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 2nd edition, 2001.

[Col09] Tobias Columbus. On the Complexity of Contraction Hierarchies, 2009. Student's thesis, Karlsruhe Institute of Technology.

[Del08] Daniel Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, September 2008. Best Student Paper Award - ESA Track B.

[Del09] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms.* PhD thesis, Fakultät für Informatik, Universität Karlsruhe (TH), 2009.

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book.* American Mathematical Society, 2009.

[DGNW11] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 921–931. IEEE Computer Society, 2011.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In Ahuja et al. [AMZ09], pages 182–206.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[DW09] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ahuja et al. [AMZ09], pages 207–230.

[Fuc10] Fabian Fuchs. On Preprocessing the ALT-Algorithm, 2010. Student's thesis, Karlsruhe Institute of Technology.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

[GJ75] Garey and Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SICOMP: SIAM Journal on Computing*, 4, 1975.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979.

[Gör10] Robert Görke. *An Algorithmic Walk from Static to Dynamic Graph Clustering.* PhD thesis, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2010.

[GRB10] Zonghao Gu, Edward Rothberg, and Robert Bixby. Gurobi 4.0.2. software, December 2010.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[Gus07] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge Univ. Press, New York, NY, USA, 2007.

[Gut04] Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[HKMS06] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [DGJ09], pages 41–72.

[HT84] Dov Harel and Robert Endre Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13:338–355, May 1984.

[Kar07] George Karypis. METIS - Family of Multilevel Partitioning Algorithms, 2007.

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[KMS05] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2005.

[Lau97] Ulrich Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[Lau09] Ulrich Lauther. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags. In Demetrescu et al. [DGJ09], pages 19–40.

[MS04] Burkhard Monien and Stefan Schamberger. Graph Partitioning with the Party Library: Helpful-Sets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004.

[MSS+05] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*, pages 189–202. Springer, 2005.

[MSS+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11(2.8):1–29, 2006.

[NW88] Georg L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, NY, USA, 1988.

[PSWZ07] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2007.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Sch06] Heiko Schilling. *Route Assignment Problems in Large Networks.* PhD thesis, Technische Universität Berlin, 2006.

[SV88] Baruch Schieber and Uzi Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM J. Comput.*, 17:1253–1262, December 1988.

[TF87] Robert E. Tarjan and Michael L. Fredman. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[Vaz03] Vijay V. Vazirani. *Approximation Algorithms.* Springer, New York, NY, USA, 2003.