FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Mehr Parallelismus in Single-Source Shortest Path Algorithmen

Simulation und Implementierung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Michael Kainer, BSc
Matrikelnummer e01325106

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Univ.Prof. Dr. Jesper Larsson Träff, MSc PhD

Wien, 1. Dezember 2018

Michael Kainer	Jesper Larsson Träff

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

More Parallelism in Single-Source Shortest Path Algorithms

Simulation and Implementation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Michael Kainer, BSc
Registration Number e01325106

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Jesper Larsson Träff, MSc PhD

Vienna, 1st December, 2018

Michael Kainer	Jesper Larsson Träff

Erklärung zur Verfassung der Arbeit

Michael Kainer, BSc
Dr. Heinrich Bachstraße 39
2442 Unterwaltersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Dezember 2018

Michael Kainer

Acknowledgements

I want to start by thanking my family, who was always supportive and provided the support that made it possible for me to pursue my studies in a hassle-free environment. Without their support, this chapter in my life would only have been half as fun as it turned out to be.

Prof. Träff not only made this thesis possible, but was also very supportive through the whole process of creating this thesis. We had interesting discussions, whiteboard sessions, and a fruitful exchange of knowledge. He also provided me with valuable feedback to improve this thesis and bring it into the form it is today. Of course, this thesis would not have been possible if Prof. Träff and his team did not organize their amazing courses on parallel computing that started my fascination with this topic. Many thanks to Prof. Träff and his team for all of that.

Furthermore, I want to thank all my friends and colleagues at TU Vienna. They form the basis of a nice environment to study in, which in my opinion is the most important factor for a successful and fulfilling academic adventure.

Kurzfassung

Das Single-Source Shortest Path Problem (kürzester Pfad von einem Startpunkt aus) ist eines der am häufigsten untersuchten Probleme in der Informatik. Allerdings ist das Problem dafür bekannt enorm schwierig parallelisierbar zu sein. Dies liegt vor allem an dem sogenannten Transitive Closure Bottleneck.

1998 wurde ein Algorithmus namens Δ-Stepping vorgeschlagen um das Single-Source Shortest Path Problem zu parallelisieren. Dieser Algorithmus hat als Basis für eine Reihe von zusätzlicher Forschung im Bereich solcher Algorithmen gedient und zählt heute als der Standardalgorithmus zum Parallelisieren des Single-Source Shortest Path Problems.

Eine weitere Idee das Single-Source Shortest Path Problem zu parallelisieren wurde von Crauser et al., ebenfalls 1998, publiziert. Diese Idee hat allerdings nie viel Aufmerksamkeit erlangt und ist aus heutiger Sicht nicht mehr sehr bekannt.

Das Ziel dieser Arbeit ist es zu evaluieren ob diese Idee eine mögliche Alternative für Δ-Stepping darstellt. Des Weiteren werden verschiedene Möglichkeiten betrachtet diesen Lösungsansatz zu erweitern.

Um diese Ziele zu erreichen werden zuerst die Algorithmen eingeführt, danach werden sie als korrekt bewiesen und letztendlich implementiert. Jeder Algorithmus wird zweimal implementiert: Eine Implementierung dient zum Analysieren der Anzahl an Phasen die benötigt werden um das Single-Source Shortest Path Problem zu lösen. Diese Anzahl an Phasen stellt eine wichtige Kennzahl dar um die Qualität solcher Algorithmen zu beurteilen. Die zweite Implementierung dient zur Performancemessung auf den Systemen der Forschungsgruppe.

Die Analyse der Phasenanzahl hat ergeben, dass die vorgeschlagenen Erweiterungen der original Formulierung des Algorithmus die Anzahl an Phasen um einiges reduziert und daher zu einer theoretischen Verbesserung des Algorithmus führt. Leider konnte keine effiziente Implementierung dieser Verbesserungen gefunden werden und daher konnte diese theoretische Verbesserung nicht in die Praxis umgesetzt werden. Nichtsdestotrotz hat sich herausgestellt, dass der original Algorithmus von Crauser et al. eine ausgezeichnete Performance im Vergleich zu Δ-Stepping aufweist und daher als Alternative dazu gesehen werden kann.

Abstract

The single-source shortest path problem is one of the most studied problems in computer science. Nevertheless, it is infamous for being notoriously hard to parallelize due to the so-called transitive closure bottleneck.

In 1998 an algorithm called Δ-stepping was proposed to parallelize the single-source shortest path problem. This algorithm formed the basis for further research in even more optimized algorithms, and eventually became to be the de-facto standard in parallelizing the single-source shortest path problem.

A second idea by Crauser et al. to parallelize the single-source shortest path problem was published in 1998. This idea did not get a lot of attention, and seems to have been forgotten.

The goal of this thesis is to evaluate the viability and performance of Crauser et al.'s approach in comparison to Δ-stepping. Furthermore, several ideas to improve or change this approach are evaluated.

In order to do this, these algorithms are first introduced, then proven correct, and finally implemented. There are two implementations of each algorithm. One implementation is to analyze the number of phases each algorithm needs to solve the single-source shortest path problem, a key metric in assessing the quality of such algorithms. The second implementation is optimized to measure the performance on the systems provided by the research group.

The phase analysis shows that the improvements to Crauser et al.'s original formulation of their algorithm reduce the number of phases by a large margin, therefore leading to a theoretical gain. Unfortunately, no efficient implementation was found for these improvements, and therefore they cannot enhance real-world performance. Nevertheless, Crauser et al.'s original algorithm performs very well and is competitive to Δ-stepping.

Contents

CHAPTER 1

Introduction

The single-source shortest path problem aims to answer the following question about weighted graphs: Given a distinguished starting node, what is the path with the least cost to all other nodes in the graph? Intuitively, the user wants to move from a starting location to a destination location in the shortest amount of time. Single-source shortest path algorithms calculate the path the user should take.

The single-source shortest path problem is one of the most famous and most studied problems in computer science. Not only in theory, but also in practice, this problem is of high relevance. Obvious applications include routing problems, which are not only limited to route planning on road maps, but are also relevant in circuits, e.g., to find critical signal paths [8], or in video games, e.g., to control the movement of game pieces. Furthermore, the single-source shortest path problem is often required in order to solve more complex problems, and can therefore be considered a fundamental building block in algorithm design. Examples include the all-pairs shortest path problem, approximation algorithms, or scheduling problems. Ahuja et al. [3, Chapter 4.4] offer an exhaustive description of many more applications of the single-source shortest path problem. Sometimes, the single-source shortest path problem crops up in completely unexpected places. A nice example of such an occurrence is the C++ pretty-printer clang-format, which uses Dijkstra's algorithm [26, Problem 2] to find the visually most pleasing way to break overlong lines [40].

In a sequential setting the single-source shortest path problem is well understood, and there are multiple algorithms to solve this problem efficiently. Some of these algorithms are going to be discussed in detail in the next chapter, and some of these algorithms serve as basis for parallel implementations. For a more detailed overview of these basic algorithms one can reference to Introduction to Algorithms [18, Chapter 24] or the study of Cherkassky et al. [15]. Zwick's survey [70] additionally contains an overview of problems closely related to the single-source shortest path problem, and algorithms to solve them.

Parallelization in the context of this thesis is restricted to parallelization on CPUs in a shared-memory system. The concrete systems used for benchmarking will be described in Section 5.7. Parallelization utilizing distributed memory systems or parallelization with the help of accelerators, especially GPUs, is not considered in this thesis. Since the benchmark is implemented using C++, the C++ memory-model has to be used. Chapter 5 gives a brief overview of this memory-model and how it has to be used to ensure correct semantics of the implementation, and how it can be used to enhance the efficiency of the implementation. The theoretical sections of this thesis assume the usual comparison based model of computation.

Unfortunately, the single-source shortest path problem is very hard to parallelize, in the sense that there is no both time-efficient and work-efficient parallel algorithm to solve the problem for general graphs. The reason for this is the so-called *transitive closure bottleneck* [41]. Intuitively, one can explain the difficulties with parallelizing this problem by considering a graph that contains a long sequential chain. While processing this chain only one processor can perform meaningful work, namely the processor that is looking at the single node at the frontier. This problem can be reduced to a problem called list-ranking. Although list-ranking by itself is perfectly parallelizable [17], such techniques have not been applied to the single-source shortest path problem because in order to apply list-ranking one would have to know in advance that there is such a list in the graph, and it does not seem to be possible to detect this efficiently.

The basis of this thesis is Crauser et al.'s paper „A Parallelization of Dijkstra's Shortest Path Algorithm“ [20], which proposes a simple idea to parallelize Dijkstra's algorithm. The goal of this thesis is threefold: First, the ideas of Crauser et at. are being refined and explored by trying some simple and new ideas. Second, an empirical analysis of Crauser et al.'s original algorithms and of the new ideas is performed. Third, a benchmark of a shared-memory implementation to evaluate the performance of these algorithms compared to a state-of-the-art algorithm is performed. Note that Crauser et al.'s approach is limited by the transitive closure bottleneck as well, and can therefore not lead to a time-efficient and work-efficient parallel algorithm. The same holds true for the refinements made in this thesis. Nevertheless, this restriction only becomes relevant in a worst-case scenario. If one considers the average case, Crauser et al.'s approach leads to a high performance and is simple to implement, as is shown in Chapter 5. Although the algorithm does not improve the situation in a worst-case analysis, it is still valuable in a practical setting.

The remainder of this thesis is structured as follows:

- The remaining sections of this chapter introduce the fundamental concepts required thorough the remainder of this thesis. It contains the mathematical preliminaries required for the thesis and formally introduces the single-source shortest path problem. Furthermore, related problems are introduced as well.
- Chapter 2 gives an overview of the research that has been done in this field. It

contains an overview of the state-of-the art of the parallel single-source shortest path problem, and an overview of the previous research in this field.

- Chapter 3 explains the ideas and motivation behind the algorithms. It formally introduces the algorithm by Crauser et al. and the new variations thereof. Furthermore, this chapter proves them correct.

- After having defined the algorithms, they will be analyzed empirically to find the potential for parallelism under the assumption that there is an infinite amount of processors available. Chapter 4 discusses the merits and weaknesses of this empirical analysis, the methodology, and finally the results thereof.

- Lastly, a benchmark is performed to check if the theoretical performance can be achieved in real implementations as well. Chapter 5 describes how the algorithms presented can be implemented efficiently, and concludes with the methodology and results of the benchmark.

1.1 Mathematical Preliminaries

This section aims to give an exact definition of the single-source shortest path problem and all other mathematical tools required in this thesis. To define the single-source shortest path problem, graphs, as used in this thesis, need to be defined first.

Definition 1.1 (Directed Graph)**.** *A directed graph $G = \langle V, E \rangle$ consists of a finite set of* nodes *V that are unidirectionally connected via some* edges *$E \subseteq V \times V$. Each edge $\langle v, v' \rangle \in E$ is defined by an originating node $v \in V$, called the* source*, and a target node $v' \in V$, called the* destination*.*

Definition 1.2 (Weighted Directed Graph)**.** *A weighted directed graph additionally defines a function* $\text{cost}(v, v') : V \times V \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ *that maps each edge to a non-negative real number, which is called the* cost*, or* weight*, of the edge. If the edge $\langle v, v' \rangle$ does not exist, the function assumes the value ∞.*

Definition 1.3 (Outgoing Edges, Incoming Edges)**.** *For each node $v \in V$* $\text{outgoing}(v) \subseteq E$ *is the set of all edges such that $\langle v, v' \rangle \in E$, i.e., all edges that originate at v. Conversely, let* $\text{incoming}(v) \subseteq E$ *be the set of all edges such that $\langle v', v \rangle \in E$, i.e., all edges whose destination is v.*

This definition of weighted graphs limits the costs to be non-negative on purpose. Without this limitation, the single-source shortest path problem gets more difficult, as will be briefly discussed in Section 1.2. Due to this increased difficulty most algorithms to solve the problem with non-negative edge weights are not applicable anymore, and therefore graphs with negative edge weights will not be covered in this thesis.

Definition 1.4 (Path, Cost, Shortest Path, Distance)**.** *A path between two nodes v and v' is a sequence of edges $[e_1, ..., e_n]$ in a graph such that the source of e_1 is v, the source of e_i is the destination of e_{i-1} for $i > 1$, and the destination of e_n is v'.*

The cost of a path is the sum of the costs of all edges in the path.

A shortest path between v and v' is a path between them with the smallest cost.

The cost of a shortest path is called distance between the two nodes, and is designated as $\mathrm{dist}(v, v')$*. If there is no path between v and v', the distance is ∞.*

Note that the shortest path between two nodes is not necessarily unique, although for the purposes of this thesis this fact does not matter. Whenever „the shortest path“ is mentioned, any of the possible shortest paths can be chosen.

Definition 1.5 (Single-Source Shortest Path Problem)**.** *Given a designated source node $s \in V$ in a weighted, directed graph, what is the distance* $\mathrm{dist}(s, v)$ *for each $v \in V$?*

For simplicity, in the context of the single-source shortest path problem let $\mathrm{dist}(v) = \mathrm{dist}(s, v)$, i.e., $\mathrm{dist}(v)$ designates the distance from the starting node s to v. This definition of the single-source shortest path problem is only concerned with the distances to all nodes. The following section gives an overview of related problems.

Definition 1.6 (Shortest Path Tree)**.** *The shortest path tree is a tree rooted at the designated source node s that contains all reachable nodes. Furthermore, each path from s to any other node in the tree corresponds to a shortest path from s to the other node in the original graph.*

Since shortest paths are not necessarily unique, the shortest path three is not unique as well. All algorithms analyzed in this thesis can be easily extended to generate a shortest path tree without additional work. Nevertheless, there are algorithms where this is not possible without additional work, as will be seen in Section 2.3.

The term ∞ was already used multiple times without explicitly defining its semantics. For purposes of this theses, semantics similar to IEEE 754 floating point numbers are assumed, i.e., let $-\infty < r < \infty$ for all real numbers r and use the „intuitive“ meaning for mathematical operations: $r + \infty = \infty$, $r - \infty = -\infty$, $r \cdot \infty = \infty$ and $r/\infty = 0$. Furthermore, let $\min_{\emptyset}(\cdot) = \infty$ and $\max_{\emptyset}(\cdot) = -\infty$. This simplifies a lot of the definitions later on because these definitions remove almost all corner cases regarding to nonexistent edges or empty sets.

Furthermore, this thesis utilizes the probability distributions as summarized in Table 1.1. $\mathbb{E}(x)$ denotes the expected value of the random variable x.

For performance analysis the usual Big-O notation $\mathcal{O}(\cdot)$ is used:

$$\mathrm{f}(x) = \mathcal{O}(\mathrm{g}(x)) \iff \exists c > 0 \; \exists n_0 > 0 \; \forall n > n_0 : \mathrm{f}(n) \leq c \cdot \mathrm{g}(n)$$

$\mathcal{A}(p)$	Bernoulli distribution with a success chance of p
$\mathcal{B}(n, p)$	Binomial distribution with n experiments with a success chance of p
$\mathcal{P}(\lambda)$	Poisson distribution with the rate λ
$\mathcal{PB}(A)$	Poisson binomial distribution with the vector of success chances A
$\mathcal{U}(a, b)$	Continuous uniform distribution for the domain $[a, b]$

Table 1.1: Summary of the probability distributions used in this thesis.

While this definition is only for functions in a single variable, it is usually „intuitively“ extended to functions in multiple variables. It turns out that doing so mathematically rigorously is non-trivial, but luckily works out fine for the use-case of algorithm analysis. Details about this topic can be found in Howell's technical report about asymptotic notation with multiple variables [37].

1.2 Related Problems

There are several problems that are closely related to the single-source shortest path problem. Even though they will not be covered in this thesis, this overview gives an impression of how large this field of research is, and might be helpful in branching out to find inspirations on how to solve the single-source shortest path problem more efficiently.

An immediate generalization of the single-source shortest path problem as defined in this thesis is to allow negative edge costs. This small variation makes the problem NP-complete [33, ND29: Comment] and is not covered by most algorithms, with the notable exception of the Bellman-Ford algorithm: This algorithm continues to work normally as long as there is no negative-weight cycle in the graph. If there is such a cycle in the graph, the algorithm can at least detect this situation and finish gracefully [18, Chapter 24].

Definition 1.7 (All-Pairs Shortest Path Problem)**.** *Given a graph, what is the distance* $\text{dist}(v, v')$ *for all node pairs* v *and* v'*?*

The all-pairs shortest path problem is an immediate generalization of the single-source shortest path problem since the restriction of a single starting node is removed. Interestingly, this problem is easier to parallelize than the single-source shortest path problem. Intuitively, this is because there is just more work to do. This problem can be solved sequentially using Floyd's algorithm [29] or using multiple runs of any algorithm to solve the single-source shortest path problem.

Definition 1.8 (Single-Source Single-Destination Shortest Path Problem)**.** *Given a graph, what is the distance* $\text{dist}(v, v')$ *between a designated source node* v *and a designated destination node* v'*?*

The single-source single-destination shortest path problem is an immediate specialization of the single-source shortest path problem. Instead of considering all destination nodes,

i.e., calculating the shortest path tree, one is only concerned about the shortest path to a single destination node. This problem is in general not easier than the single-source shortest path problem but can be solved in practice more efficiently by utilizing guided search strategies, which utilize additional information based on the structure of the input graph. For example, two-dimensional maps obey the triangle-inequality, which can be leveraged for such guided searches. An example of such an algorithm is the so-called A* (A-star) algorithm [36].

Definition 1.9 (Longest Path Problem)**.** *Given a graph and two designated nodes in this graph v and v', what is the longest path from v to v' without visiting a node twice?*

This seemingly innocuous variation of the single-source single-destination shortest path problem is actually NP-complete [33, ND29] and even in the class of NP-complete problems it is considered to be very difficult in the sense that is it provably very hard to find approximate solutions, even for unweighted graphs [42]. Naturally, there are no efficient algorithms to solve this problem in general.

CHAPTER 2

Related Work

The single-source shortest path problem has been studied for at least 70 years, and there is a lot of published literature and research, both of theoretical and of practical nature. This chapter aims to give an overview of the state-of-the art parallel algorithms to solve the single-source shortest path problem and the sequential algorithms they are based on.

There are two major approaches for solving the single-source shortest path problem: label setting and label correcting. Label setting algorithms visit all nodes in the input graph in such an order that the correct distances to all nodes can be set in one pass. For this to be correct a lot of thought has to be put into the order in which the nodes are visited. Algorithms of this nature are usually similar to Dijkstra's algorithm. Label correcting algorithms visit the nodes in the input graph in a much more relaxed order, and therefore may assign wrong distances to some nodes. Label correcting algorithms recognize these mistakes and subsequently correct them. This may lead to redundant work because at the time of correction the mistake might already have propagated itself. Due to this, other nodes may have to be revisited as well, aggravating this effect. Algorithms of this nature are usually similar to the Bellman-Ford algorithm. Other approaches to solve the single-source shortest path problem exist as well, but they either need preprocessing, are restricted to special classes of graphs or are probabilistic.

2.1 Label Setting Algorithms

Dijkstra's algorithm [26, Problem 2] is the most famous and, even today, the most performant and most used sequential single-source shortest path algorithm. Dijkstra's algorithm maintains a tentative distance for each node that is initialized to 0 for the source node and to ∞ for all other nodes. The algorithm finds a correct label setting ordering of the nodes by placing them into a priority queue ordered by the tentative distance, and repeatedly settling the node with the smallest tentative distance. During settling, each neighbor of a node is considered. If a neighbor's tentative distance is greater

than the tentative distance of the settled node plus the edge cost to the neighboring node, the tentative distance of the neighboring node is updated to the new, smaller, value. Newly reached nodes are added to the priority queue.

The performance of this algorithm is highly dependent on the performance of the data structure used for the priority queue. By using a Fibonacci heap [32] to implement the priority queue it is possible to reach the theoretical optimal runtime for Dijkstra's algorithm, namely $\mathcal{O}(|V|\log|V| + |E|)$, for arbitrary graphs with real-valued non-negative edge costs in a comparison-based computation model. This is due to the fact that it is trivially possible to sort using priority queues, and therefore Dijkstra's algorithm. Furthermore, the converse direction holds as well, i.e., if the time-bound of sorting is $\mathcal{O}(f(n))$ it is always possible to obtain a priority queue with the time-bound $\mathcal{O}(f(n))$ for delete and insert, and $\mathcal{O}(1)$ for find-minimum [68]. Even though the Fibonacci heap gives the best theoretical time-bounds, research in the area of priority queues continued. Consequently, there are many more possible heap implementations, e.g., pairing heaps [31], relaxed heaps [27], or Brodal heaps [11].

Once one considers more powerful machine models, i.e., models that can perform more than just comparisons on numbers, one can obtain better time-bounds for Dijkstra's algorithm due to the fact that one has better time-bounds for sorting and subsequently for priority queues. Thorup presents one such priority queue for the RAM model of computation [67] leading to a runtime of $\mathcal{O}(|E|\log\log|E|)$ for Dijkstra's algorithm.

Crauser et al. [20] noticed that the strict visiting order employed by Dijkstra's algorithm can be relaxed with some additional bookkeeping. Due to this relaxation it is possible to freely choose between multiple nodes to visit next. In a parallel implementation this allows for settling all those nodes in parallel. While the authors did include a theoretical analysis of their approach for random graphs, they unfortunately did not include benchmarks of their approach.

Dijkstra's algorithm and Crauser et al.'s work form the basis of this master thesis and therefore both algorithms will be explained in much more detail at a later point in this work.

It is possible to solve the single-source shortest path problem in linear time for undirected graphs with integer weights [65]. This algorithm can also be generalized to work with IEEE floating-point numbers [66]. Even though, the algorithm is extremely complicated and hard to implement, it was implemented during a Bachelor's thesis [59]. Unfortunately, the algorithm was found to be much slower than Dijkstra's algorithm. Therefore, one can conclude that Thorup's results are more of a theoretical nature than of a practical. Nevertheless, Crobak et al. [21] parallelized a simplified version of Thorup's algorithm.

Brodal et al. [12] developed a parallel priority queue with support for a decrease-key operation that can be used to implement Dijkstra's algorithm in linear time in the number of nodes, given enough processors.

Another approach with a shared data-structure, in this case a shared weighted tree, is

presented by Shi and Spencer [63]. They modify Dijkstra's algorithm in such a way that it does not only settle the chosen node itself but also its k nearest neighbors.

2.2 Label Correcting Algorithms

The first nowadays well-known algorithm to solve the single-source shortest path problem was the Bellman-Ford algorithm. It was independently described by Ford in a paper about the maximum network flow problem [30, Chapter 7] in 1956, by Moore [54] in 1957, and by Bellman [5] in 1958. This algorithm maintains a set of fringe nodes and a tentative distance for each node. At the beginning only the source node is in the set of fringe nodes. Furthermore, the tentative distance is initialized to ∞ for all nodes, except for the source node where it is initialized to 0. In each iteration all nodes in the set of fringe nodes are settled and removed from the set. Each node whose tentative distance was changed during the settling phase is then (re-)added to the set of fringe nodes. The algorithm terminates when the set of fringe nodes is empty. This takes at most as many iterations as there are nodes in the graph. Since each iteration is embarrassingly parallel, this algorithm is often used as basis for a parallelized single-source shortest path algorithm. Since a lot of redundant work is performed during execution, this algorithm is considered to be inefficient.

Δ-stepping (delta-stepping) [53][1] is a refinement of the Bellman-Ford algorithm that replaces the set of fringe nodes with multiple buckets of fringe nodes. This algorithm is primarily controlled by a parameter Δ which is a positive real number. The first bucket contains all nodes with the tentative distance 0 to Δ, the second bucket contains all nodes with the tentative distance Δ to 2Δ, and so on. In each iteration the algorithm settles all nodes in the first non-empty bucket and subsequently removes them from the bucket. All nodes whose tentative distance was changed in this process are then (re-)added into their appropriate bucket. Note that this means that a node might be re-added into the bucket it was in before the iteration. This can be optimized by splitting each iteration into two phases: In the first phase one considers only edges whose cost are less than Δ, so-called light edges. Once there have been no re-insertions into the current bucket, one considers the other edges, the so-called heavy edges. This optimization uses the fact, that it is unnecessary to propagate information via heavy edges as long as there might still be the possibility that the tentative distance in the bucket changes due to light edges.

The performance of this algorithm is highly dependent on the chosen Δ. Which value is good is dependent on the structure of the input graph, but in general there is a tradeoff between large and small Δ's: On one hand, a large Δ increases parallelism. This is because all nodes in a single iteration can be settled in parallel. Therefore, if Δ is large, there are usually more nodes in each bucket, leading to more parallelism. On the other hand, a small Δ decreases the amount of redundant work. This is because the buckets are smaller, and therefore lead to less nodes that are settled too early and have subsequently

[1] This idea was already published in 1998 [51]. The referenced article from 2003 additionally contains ideas and refinements that were introduced afterwards.

to be re-inserted into a bucket and later settled again. Meyer and Sanders have proven that at least for random graphs there exists a Δ that leads to a good tradeoff between those two extremes.

Furthermore, they refined this algorithm to be able to deal with graphs where a good Δ cannot be chosen a priori and optimized the algorithm even more by introducing shortcut edges for each bucket. This way buckets can be cleared in a constant number of iterations [52].

Δ-stepping will be explained in much more detail later in this work, because this is currently the state-of-the art algorithm for solving the single-source shortest path problem in parallel.

Madduri et al. [49] did an experimental study of simple Δ-stepping on a quite unusual Cray MTA-2 machine. They were able to achieve a good relative speedup of more than 30 when using 40 processors for various classes of graphs.

Chakaravarthy et al. [13] implemented Δ-stepping on a distributed memory cluster and developed multiple improvements: First, they refined the notion of light and heavy edges to further optimize the phases of each iteration. Second, they introduced a heuristic to greatly reduce the number of edges that need to be relaxed after clearing a bucket. Third, they noticed that starting with Δ-stepping and switching to a simple Bellman-Ford algorithm at a later point during execution improves the performance. To improve the scaling behavior across their cluster, they also implemented a load balancing scheme. The authors claim a „four orders of magnitude improvement over the best published results“ at the time of writing, which implies that Δ-stepping can indeed be used for large-scale graph processing in a real-world setting.

Blelloch et al. [7] extended Δ-stepping to radius-stepping by making the parameter Δ, and therefore the bucket sizes, variable. Each node in the graph is assigned a radius. All radii are valid but the performance of this algorithm is directly dependent on a good choice of these radii. For each iteration radius-stepping chooses the minimum of tentative distance plus the node's radius as new threshold. All nodes whose tentative distance is less than or equal to this threshold are then settled as in Δ-stepping. The authors have proven that for so-called (k, ρ)-graphs this algorithm is indeed efficient. A (k, ρ)-graph is a graph where each node v fulfills the property that each of the ρ nodes with the smallest distances from v can be reached from v with a path with a maximum length of k. Furthermore, the authors provided an algorithm that allows any graph to be transformed into this class by adding shortcut edges.

Dijkstra Strip Mined Relaxation, short DSMR [50], is a different label-correcting approach to solve the single-source shortest path problem in parallel. The idea is to distribute the nodes in the graph and split the algorithm into phases. In each phase each processor runs Dijkstra's algorithm until a certain number of edges has been relaxed. Relaxations that would have to reach out for remote nodes are buffered. Once this number of edges has been reached, the buffered requests are exchanged in an all-to-all communication and executed on the local sub-graph. This is repeatedly done until the algorithm has nothing

more to do. The authors furthermore provided two preprocessing algorithms to speed up their algorithm. DSMR works particularly well on scale-free graphs [4]. Intuitively, these are graphs that have few nodes with a very high degree and a lot of nodes with a low degree. For example, social networks fulfill this property.

Nikas et al. [56] parallelized Dijkstra's algorithm by using helper threads to speculatively relax edges that are deeper in the priority queue with the hopes that these relaxations save some work later on. To avoid lock contention and other locking overheads they utilized transactional memory to access the shared data structures. On their testing system they achieved a speedup of 1.84 for 14 threads, therefore proving that transactional memory and speculative relaxations are a viable strategy to speed up the otherwise completely sequential algorithm by utilizing multiple helper threads.

2.3 Other Algorithms

A different idea to solve the single-source shortest path problem is based on so called contraction hierarchies [34]. Each node is assigned a unique priority. All priorities are correct but a good assignment of priorities is essential for good performance of this algorithm. While finding the optimal priorities is non-trivial, simple heuristics lead to good performance in a real-world setting. Now, the input graph is preprocessed with the help of an auxiliary graph, called the overlay graph: The overlay graph starts as a copy of the input graph. All nodes are then traversed in priority order and contracted. During the contraction of a node v each path $[u, v, w]$ is considered. For each such path a shortcut edge $\langle u, w\rangle$ with $\text{cost}(u, w) = \text{cost}(u, v) + \text{cost}(v, w)$ is inserted into both graphs if $\text{dist}(u, w) > \text{cost}(u, v) + \text{cost}(v, w)$ in the overlay graph, while v is ignored for calculating $\text{dist}(u, w)$. Intuitively, a shortcut edge is inserted if it is required to preserve the shortest path distances between predecessors and successors of v in spite of the removal of v from the overlay graph. Finally, the contracted node v is removed from the overlay graph. At the end of this process, the overlay graph is empty.

To execute a single-source single-destination shortest path query on such a preprocessed graph two shortest path searches using Dijkstra's algorithm are run. One starts from the starting node and only considers edges such that the priority of nodes is increasing and the other starts at the destination node and only considers edges such that the priority of nodes is decreasing. It can be shown that the two searches will meet at a common node v. Once this has occurred, a shortest path has been found: It is the concatenation of the shortest path from the starting node to v and the shortest path from v to the destination node. If necessary, one has to map the path, which uses shortcut edges, back to a path without shortcut edges.

Parallel Hardware-Accelerated Shortest Path Trees, in short PHAST, developed by Delling et al. [23], extend the idea of contraction hierarchies to calculate the whole shortest path tree and therefore solve the single-source shortest path problem. Furthermore,

Delling et al. parallelize their algorithm using SSE instructions[2] to handle more than one starting node in parallel and finally port their algorithm to CUDA [57] to make it run on GPUs. They also devise a simple parallelization scheme for single instances. Nevertheless, PHAST needs preprocessing on the graph it is being used on, and therefore it is only worthwhile to use when multiple queries are performed on a single graph.

Contraction hierarchies work well on graphs with a low highway dimension [1, Section 3, Definition 1]. Given a graph $G = \langle V, E \rangle$, the highway dimension is defined as the smallest integer h that fulfills the following condition: For all $r \in \mathbb{R}_{\geq 0}$ and for all $v \in V$, one constructs a subgraph G' consisting only of nodes w such that $\text{dist}(v, w) \leq 4r$ and all edges between these nodes. For each such subgraph G' a set $S \subseteq E \wedge |S| \leq h$ is constructed, such that for each shortest path π in G' with cost greater than r and originating at v $\pi \cap S$ is non-empty. Intuitively, graphs with edges that are used often in shortest paths have small sets S, and therefore a low highway dimension. For example road networks have a low highway dimension because for most shortest paths one—literally—has to use highways.

There are several theoretical results with respect to graphs using separator decompositions. A separator is a set of nodes whose removal splits a graph into two subgraphs that are not connected with each other anymore. In other words: All paths from one subgraph to the other subgraph have to cross the separator. Furthermore, one requires that these subgraphs are roughly of the same size, in the sense that they only differ by a given fraction. Planar graphs are proven to have small separator decompositions fulfilling these properties [48]. Examples of such approaches include Cohen's work [16] for directed graphs where the separator decomposition is given and Träff and Zaroliagis' work for planar graphs [69].

[2]Streaming SIMD Extensions: An instruction set specialized in providing instruction level parallelism by working on multiple data items at once. Consult a CPU manufacturer's documentation for more details, for example AMD's Architecture Programmer's Manual [2].

CHAPTER 3

Algorithms

This chapter describes the label setting algorithms used and developed during this thesis. All algorithms solve the single-source shortest path problem on directed graphs exactly and deterministically.

The rest of this chapter is structured as follows: First, Dijkstra's algorithm will be introduced and proven correct. Second, an algorithm scheme will be derived from Dijkstra's algorithm to ease the definition of all other algorithms. Third, this scheme will be proven correct as long as the used sub-algorithms fulfill certain properties. Lastly, all analyzed algorithms will be defined using the aforementioned algorithm scheme.

3.1 Dijkstra's Algorithm

Dijkstra's algorithm [26, Problem 2] takes a weighted graph and a source node as input, and returns the distance from the source node for each node, and optionally the shortest-path tree. To achieve this, the algorithm maintains a tentative distance for each node. The basic idea of Dijkstra's algorithm is to initialize the tentative distance for each node to ∞, except for the source node where it gets initialized to zero. After that the non-settled node with the smallest, non-infinite, tentative distance is chosen to be settled. During settling each outgoing edge of this node is considered and if following this edge leads to a shorter path than the tentative distance of the destination node, the tentative distance of the destination node is updated. This step is repeated until there are no such nodes left. After finishing this algorithm all tentative distances are equal to the shortest distances from the source node. A formal description of this algorithm can be seen in Algorithm 3.1.

The set of all nodes V is partitioned into three pairwise disjoint sets, unsettled U, fringe F, and settled S, during each point in time in the algorithm. All nodes, except the starting node, start in U. Once a node is first reached during settling of another node, it moves

Require: a directed graph $G = \langle V, E \rangle$ and a starting node s
Ensure: $\forall v \in V : \mathit{tent}[v] = \mathrm{dist}(v)$
  for all $v \in V$ **do**
    $\mathit{tent}[v] \coloneqq \infty$
  end for
  $\mathit{fringe} \coloneqq \{s\}$
  $\mathit{tent}[s] \coloneqq 0$
  while *fringe* is not empty **do**
    $v \coloneqq \mathrm{argmin}_{v \in \mathit{fringe}}\, \mathit{tent}[v]$
    remove v from *fringe*
    for all $\langle v, v' \rangle \in \mathrm{outgoing}(v)$ **do**
      if $\mathit{tent}[v] + \mathrm{cost}(v, v') < \mathit{tent}[v']$ **then**
        insert v' into *fringe* if $\mathit{tent}[v'] = \infty$
        $\mathit{tent}[v'] \coloneqq \mathit{tent}[v] + \mathrm{cost}(v, v')$
      end if
    end for
  end while

Algorithm 3.1: Dijkstra's Algorithm

from U to F. Each node that has been settled ends up in S. A formal definition of these sets, independent of Dijkstra's algorithm, follows.

Definition 3.1 (Unexplored, Fringe, Settled)**.** *A node is in the set U and is called* unexplored *iff its tentative distance is ∞. The set S contains some nodes called* settled *for which* $\mathrm{tent}(v) = \mathrm{dist}(v)$ *holds. Note that there might be such nodes which are not in S yet. Finally, all nodes that are reachable from S by using a single edge and are not in S themselves are in the set F and are called* fringe*.*

Nodes which are unreachable will forever remain unexplored, while at the end of Dijkstra's algorithm all reachable nodes will be settled. There are no fringe nodes at the end of the algorithm.

Definition 3.2 (To Relax, To Settle)**.** *To relax an edge $\langle v, v' \rangle$ means to evaluate* $\mathrm{tent}(v) + \mathrm{cost}(v, v') < \mathrm{tent}(v')$*. If this inequality holds,* $\mathrm{tent}(v')$ *is set to* $\mathrm{tent}(v) + \mathrm{cost}(v, v')$*. This corresponds to the innermost if of Algorithm 3.1.*

To settle a node means to relax all its edges and move the node from F to S.

Definition 3.1 and 3.2 are not specific to Dijkstra's algorithm. Most single-source shortest path algorithms, and certainly all considered in this thesis, use the same notion of the three sets U, F and S and utilize an analogous relaxation and settling operation.

To show the correctness of Dijkstra's algorithm with respect to the single-source shortest path problem the following two notions are introduced:

Definition 3.3 (Soundness)**.** *An algorithm is called sound iff for each node* $v \in S$ $\text{tent}(v) = \text{dist}(v)$*, i.e., an algorithm is called sound if it fulfills the definition of* S*.*

Soundness is not enough for an algorithm to be correct as it does not state anything about nodes that are not in S. Nevertheless, soundness is strong enough to guarantee that each node that is considered to be done has the correct distance assigned to it, i.e., soundness allows to define the notion of a partial solution to the single-source shortest path problem.

Definition 3.4 (Single-Source Shortest Path Partial Solution)**.** *A partial solution for the single-source shortest path problem consists of a set of nodes where for each node* $\text{tent}(v) = \text{dist}(v)$ *holds.*

The empty solution without any nodes is trivially a partial solution for the single-source shortest path problem. Just the source node, i.e., $\{s\}$ with $\text{tent}(s) = \text{dist}(s) = 0$, is a trivial partial solution as well.

Definition 3.5 (Completeness)**.** *An algorithm is called complete iff after termination* F *is empty and for all nodes* $v \in U$ $\text{tent}(v) = \text{dist}(v) = \infty$*, i.e., the remaining nodes in* U *are unreachable. All other nodes have to be in* S *by definition, i.e., they have to be settled.*

Completeness intuitively states that the algorithm settles all reachable nodes. Soundness and completeness together imply a correct algorithm to solve the single-source shortest path problem.

3.1.1 Proof of Correctness

This proof is more elaborate than later proofs in order to be able to reference it in the proofs for the other algorithms. At first, the soundness of Dijkstra's algorithm is shown by induction over the set of settled nodes and then an argument is given for the completeness of the algorithm. Let a phase be an iteration of the outermost loop in Dijkstra's algorithm.

Lemma 3.1. *The starting node* s *can be settled immediately.*

Proof. It is safe to settle this node, since by definition $\text{dist}(s) = 0$ and by the initialization $\text{tent}(s) = 0$. Therefore after the initial step $S = \{s\}$ is a valid set of settled nodes. □

This lemma is going to be the induction base later on.

The next lemma will be the induction step and is motivated by the way Dijkstra's algorithm selects a node to be settled in each phase. This lemma is not applicable for the starting node because the starting node does not have a predecessor. In the final proof this will not matter because the starting node will be handled by Lemma 3.1.

Lemma 3.2. *Assume S to be a partial solution to the single-source shortest path problem, then for the chosen node $v = \operatorname{argmin}_{v \in F} \operatorname{tent}(v)$, with the tentative predecessor p, there does not exist another predecessor p' such that settling p' reduces the tentative distance of v.*

This non-existence of such a predecessor p' implies that $\operatorname{tent}(v) = \operatorname{dist}(v)$, which is the soundness condition required for Dijkstra's algorithm to move a node from F to S.

Proof. In other words, there must not exist a predecessor p' such that $p' \neq p$ and $\operatorname{dist}(p') + \operatorname{cost}(p', v) < \operatorname{dist}(p) + \operatorname{cost}(p, v) = \operatorname{tent}(v)$. This will be shown using a proof by cases over all possible origins of p'.

- Assume p' to be in S. Furthermore, v is in F by definition. This means that the potentially better predecessor p' has already been settled, and therefore the edge $\langle p', v \rangle$ has already been relaxed. Therefore, if p' was a better predecessor for v it would already have been discovered. Since this is not the case, because $p' \neq p$, p' cannot be a better predecessor for v (see the inner if of Dijkstra's algorithm).
- Assume p' to be in F or U. Since p' is not in S each path from s to p' has to leave S somehow since s is in S and p' is not. The lower bound of the length of such a path is given by $\operatorname{dist}(b)$, with b being some node in S, plus an outgoing edge from b leaving S:

  $$\min_{b \in S, f \in F} (\operatorname{dist}(b) + \operatorname{cost}(b, f)) \leq \operatorname{dist}(p')$$

  This situation is depicted in Figure 3.1. Since v was chosen such that $\operatorname{tent}(v) = \operatorname{dist}(p) + \operatorname{cost}(p, v)$ is minimal, i.e., $b = p$ and $f = v$ in the equation above. Therefore, $\operatorname{tent}(v) \leq \operatorname{dist}(p')$ which means that $\operatorname{dist}(p') + \operatorname{cost}(p', v) < \operatorname{tent}(v)$ cannot hold. Therefore, p' cannot be in F or U.

This concludes the proof that no better predecessor p' for v can exist. □

Theorem 3.3. *Dijkstra's algorithm is sound.*

Proof. Induction over the number of elements in S. The induction base is provided by Lemma 3.1. The induction hypothesis is that S is a partial solution for the single-source shortest path problem. The induction step is then provided by adding the chosen node using Lemma 3.2 to S. □

Theorem 3.4. *Dijkstra's algorithm is complete.*

Proof. Each node reachable from s will eventually be put into the set F since Dijkstra's algorithm always inserts all unexplored nodes reachable via outgoing edges of settled nodes (innermost if-block) into F. Furthermore, since the algorithm will move one node from F to S in each phase, and only terminates once F is empty, completeness is given. □

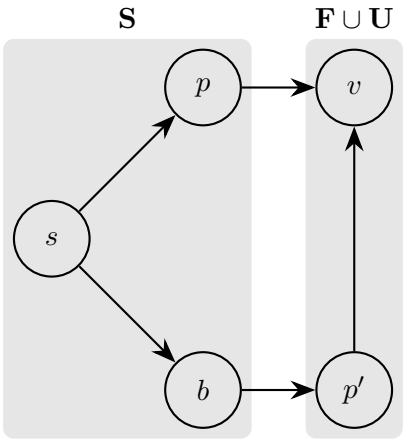

Figure 3.1: p' can only be reached via a „border“ node $b \in S$ plus at least one outgoing edge.

3.2 Algorithm Scheme

All other algorithms and criteria discussed in this chapter follow the same structure as Dijkstra's algorithm: Newly reached nodes are moved from U to F, and in each phase some nodes fulfilling a predicate φ are settled, and subsequently moved from F to S. In case of Dijkstra's algorithm the predicate is „the tentative distance is minimal.“ This intuition leads to the definition of the algorithm scheme as seen in Algorithm 3.2.

Definition 3.6 (Phase)**.** *A phase is an iteration of the outermost loop of the algorithm scheme. In other words: A phase is concerned with all nodes proposed by a criteria φ in a single iteration.*

To reason about the correctness of this algorithm scheme the definitions of soundness and completeness have to be adapted to arbitrary criteria.

Definition 3.7 (Soundness)**.** *A criteria φ is called sound if for each node v $\varphi(v) \rightarrow \text{tent}(v) = \text{dist}(v)$.*

Definition 3.8 (Completeness)**.** *A criteria φ is called complete if as long as F is not empty there exists a node $v \in F$ such that $\varphi(v)$ is true.*

Theorem 3.5. *The algorithm scheme is sound if the criteria φ is sound.*

Proof. Induction over the set of settled nodes. The induction base is given by Lemma 3.1. The induction hypothesis is that S is a partial solution to the single-source shortest path problem. The induction step follows immediately from the soundness of φ: For each node v that is settled $\varphi(v)$ is true, which immediately implies that $\text{tent}(v) = \text{dist}(v)$. □

Theorem 3.6. *The algorithm scheme is complete if the criteria φ is complete.*

Require: a directed graph $G = \langle V, E \rangle$, a starting node s, and a criteria φ
Ensure: $\forall v \in V : \mathit{tent}[v] = \mathrm{dist}(v)$ if φ is sound and complete
  for all $v \in V$ **do**
    $\mathit{tent}[v] \coloneqq \infty$
  end for
  $\mathit{fringe} \coloneqq \{s\}$
  $\mathit{tent}[s] \coloneqq 0$
  while *fringe* is not empty **do**
    for all $v \in \mathit{fringe} \wedge \varphi(v)$ **do**
      remove v from *fringe*
      for all $\langle v, v' \rangle \in \mathrm{outgoing}(v)$ **do**
        if $\mathit{tent}[v] + \mathrm{cost}(v, v') < \mathit{tent}[v']$ **then**
          insert v' into *fringe* if $\mathit{tent}[v'] = \infty$
          $\mathit{tent}[v'] \coloneqq \mathit{tent}[v] + \mathrm{cost}(v, v')$
        end if
      end for
    end for
  end while

Algorithm 3.2: Algorithm Scheme

Proof. A complete criteria always proposes at least one node to be settled in F by definition. Therefore, eventually all nodes in F will be settled. During the settling of a node, all neighboring and so far unreached nodes will be moved to F. Therefore, transitively all reachable nodes will end up in F, and will eventually, by the completeness of φ, be settled. □

3.3 Motivation & Discussion

Dijkstra's algorithm always chooses the node with the smallest tentative distance, relaxes its edges, and then chooses the next node with the, now, smallest tentative distance, and so on. This approach makes Dijkstra's algorithm sequential, i.e., there is a chain of calculations that depend on each other: The choice of the next node with the smallest tentative distance is potentially dependent on the relaxations performed by the current choice.

As long as we consider algorithms that are Dijkstra-like, this is actually the best result possible in the worst-case: If there is a graph that looks like a linked list, i.e., each node has a single edge to a successor, there is no way to work around this dependency in Dijkstra's algorithm.

Once there are multiple edges per node, a possible parallelization scheme is to parallelize the relaxations of outgoing edges of the node that is currently being settled. This parallelization scheme can only work with very dense graphs and is primarily of theoretical

interest. The reason behind this is that this scheme requires to maintain a shared priority queue. Such shared priority queues cannot be implemented efficiently.

The alternative approach considered in this thesis is based on the key observation by Crauser et al. that the settling-order of Dijkstra's algorithm can actually be relaxed by relaxing the condition „node with the smallest tentative distance." It turns out that there are other criteria, which will be explained in this chapter, that do not yield a single node but a whole set of nodes that are safe to be settled. This is the primary motivation for the algorithm scheme and the definition of phase therein: Each phase is concerned with settling a safe set of nodes.

All nodes which can be settled in one phase do not interfere with each other afterwards because a sound criteria has to guarantee that there is no way for any unsettled node to be part of a better path to the nodes being settled in the current phase. Therefore, these nodes can settled in any order, and even more so, they can be settled in parallel. The details of how this can be done in practice will be presented in Chapter 5. It suffices to say that the degree of parallelism is directly proportional to the number of phases *and* the number of nodes in each phase.

This approach is still not completely conflict free. If two nodes v and w can be settled in the same phase, and both have an edge $\langle v, x\rangle$, resp. $\langle w, x\rangle$, there might be a conflict while updating the tentative distance of x. How this conflict is avoided will be explained in-depth in Chapter 5.

Since the degree of parallelism is directly proportional to the number of phases, i.e., a lower number of phases leads to more potential parallelism, the number of phases is an important metric to measure the quality of criteria. This is because after each phase a communication and synchronization phase has to happen, which means that processors have to wait for each other, potentially leading to idle processors. This is due to the calculations of the next phase which depend on the calculations performed in the current phase. As with Dijkstra's algorithm the phases form a chain of computations that cannot be reordered, but unlike Dijkstra's algorithm, in each phase more than one node is settled.

In practice this is implemented in such a way that each processor is responsible for a certain subset of all nodes. This means that in the worst case, even through there are ample of nodes in each phase, only one processor could happen to perform all the work. There is no way to notice such load imbalances using solely the number of phases because they depend on the way the graphs are generated and distributed among the processors. The simulation as presented in Chapter 4 is oblivious to these factors as well. To alleviate this problem it is possible to implement load balancing.

To summarize: The number of phases is an important metric to judge the quality of a criteria. If a criteria can propose many nodes for settling per phase, the number of phases becomes smaller, thus leading to more parallelism.

3.4 Implied Criteria

Some algorithms introduced in this chapter will have two variations. Usually there is one variation, called the *static* variant, that only considers data that is immediately available in the input graph, and another variation, called the *dynamic* variant, that tries to consider information which only emerges during the execution of the algorithm scheme. In most cases, the latter variation will settle a superset of the nodes of the static variation in each phase, i.e., the predicate holds true for more nodes. This motivates the study of implied criteria.

Theorem 3.7. *Let φ and ψ be two criteria such that $\varphi(v) \to \psi(v)$ and ψ is sound. Then φ is sound.*

Proof. This is a direct consequence of the transitivity of logical implication.

$$\overbrace{(\varphi(v) \to \psi(v))}^{\text{precondition}} \wedge \overbrace{(\psi(v) \to \text{tent}(v) = \text{dist}(v))}^{\text{soundness of } \psi} \implies \overbrace{(\varphi(v) \to \text{tent}(v) = \text{dist}(v))}^{\text{soundness of } \varphi} \qquad \square$$

Theorem 3.8. *Let φ and ψ be two criteria such that $\varphi(v) \to \psi(v)$ and φ is complete. Then ψ is complete as well.*

Proof. Consider the set of nodes defined by the criteria. φ chooses all nodes $\Phi = \{v : v \in F \wedge \varphi(v)\}$, while ψ chooses all nodes $\Psi = \{v : v \in F \wedge \psi(v)\}$. $\varphi(v) \to \psi(v)$ means that whenever $\varphi(v)$ is true that $\psi(v)$ has to be true as well. This implies that $\Phi \subseteq \Psi$. Since Φ is already large enough in each phase to be complete, Ψ has to be as well, since it is potentially larger. $\square$

Notice how the roles of φ and ψ are swapped with respect to soundness and completeness.

3.5 Combining Criteria

It is also possible to combine criteria to form new criteria. This leads to the following observations.

Theorem 3.9. *Given the criteria ψ and ρ such that both criteria are sound, then these criteria can be combined into a sound criteria φ disjunctively: $\varphi(v) \equiv \psi(v) \vee \rho(v)$.*

Proof. This follows from the following logical implication.

$$\begin{aligned} &\overbrace{(\psi(v) \to \text{tent}(v) = \text{dist}(v))}^{\text{soundness of } \psi} \wedge \overbrace{(\rho(v) \to \text{tent}(v) = \text{dist}(v))}^{\text{soundness of } \rho} \\ &\qquad \implies \underbrace{((\psi(v) \vee \rho(v)) \to \text{tent}(v) = \text{dist}(v))}_{\text{soundness of } \varphi} \end{aligned} \qquad \square$$

Corollary 3.10. *$\varphi(v) \equiv \bigvee_{i=1}^{k} \psi_i(v)$ is a sound criteria if all ψ_i are sound.*

Theorem 3.11. *A combined criteria $\varphi(v) \equiv \psi(v) \vee \rho(v)$ is complete if at least one of ψ or ρ is complete.*

Proof. This follows from the fact that the two criteria are joined disjunctively, i.e., if one criteria always chooses at least one node in each phase, an additional disjunction cannot change this fact. In other words: A union-operation on a set cannot make a non-empty set empty. □

Corollary 3.12. *$\varphi(v) \equiv \bigvee_{i=1}^{k} \psi_i(v)$ is a complete criteria if at least one ψ_i is complete.*

3.6 Oracle

As explained in Chapter 1 Crauser et al.'s algorithm is limited by the infamous transitive closure bottleneck: In a graph that only consists of a single chain, no parallelism can be achieved by using their algorithm. The same restriction applies to the here introduced algorithm scheme, which is a generalization of Crauser et al.'s algorithm.

Nevertheless, it would be interesting to know the lower bound of the number of phases that can be expected from this algorithm scheme. Answering this question obviously depends on the chosen criteria and type of graph, and will be answered empirically in Chapter 4. This leads to the question if there is an optimal criteria, especially if there actually is a lower bound in the number of phases. If there is such an optimal criteria, it would have to reach this lower bound, and can therefore be used to empirically measure the quality of all other criteria in comparison to the optimal lower bound.

By the definition of soundness of a criteria, a node may be chosen by a criteria if $\text{tent}(v) = \text{dist}(v)$. This leads to the definition of an omniscient criteria φ, called the *oracle*, that is literally defined as $\varphi(v) \equiv (\text{tent}(v) = \text{dist}(v))$. The oracle always chooses all nodes that are by definition safe to settle.

In contrast to the other criteria defined later, the oracle needs in its definition already the solution to the single-source shortest path problem. This is also the reason why it is considered to be omniscient.

Theorem 3.13. *The oracle is the criteria with the smallest amount of phases in the given algorithm scheme.*

Proof. For the algorithm scheme to be sound the criteria has to be sound, and by the definition of soundness for criteria it follows that there cannot be a more exhaustive criteria than this oracle. Each more exhaustive criteria would have to allow the settlement of at least one node for which $\text{tent}(v) \neq \text{dist}(v)$ holds and therefore cannot be sound. □

Theorem 3.14. *The oracle is sound.*

Proof. By definition the oracle settles all nodes in F for which $\text{tent}(v) = \text{dist}(v)$ holds. This is literally the soundness condition. □

Theorem 3.15. *The oracle is complete.*

Proof. As argued in the proof of Dijkstra's algorithm, at least for the node with the minimal tentative distance in F $\text{tent}(v) = \text{dist}(v)$ holds. Therefore, at least this one node from F will be settled in each phase. Eventually all nodes in F will be settled and the criteria is therefore complete. □

3.7 Crauser's Criteria

3.7.1 Crauser's In Criteria

Intuitively, this criteria [20] exploits the following observation: If the smallest tentative distance in F is M, then all shortest paths that can be found from now on can only be of cost M or higher. Furthermore, the shortest path to some node v for that the shortest path has not yet been found has to pass through one of the incoming edges of v. This means that M plus the cost of an incoming edge of v is the lower bound for any new shortest path that could potentially be found for v. Therefore, if the tentative distance of v is already below this lower bound, it is not possible to find a cheaper path for v, and therefore v is save to be settled. An example can be seen in Figure 3.2.

Formally, this criteria settles a node v if $i(v) \leq M$ where

$$M = \min_{v \in F} \text{tent}(v)$$

is the smallest tentative distance in F and

$$i(v) = \text{tent}(v) - \min_{p \in V} \text{cost}(p, v)$$

is the tentative distance minus the smallest incoming edge cost. This characterizes the maximum allowed value allowed for any potential predecessor of v to still obtain an improvement of $\text{tent}(v)$. This formulation of this criteria will be called the *static* variant because it only uses information that does not depend on the state of the sets U, F and S, i.e., all information required for deciding this criteria is immediately available in the input graph.

The *dynamic* formulation of this criteria is defined as above, except

$$i(v) = \text{tent}(v) - \min_{p \in F \cup U} \text{cost}(p, v)$$

i.e., incoming edges from already settled nodes are not considered. Intuitively, this is possible because no new tentative distances can be discovered from these predecessors, since they are already settled and their outgoing edges are already relaxed.

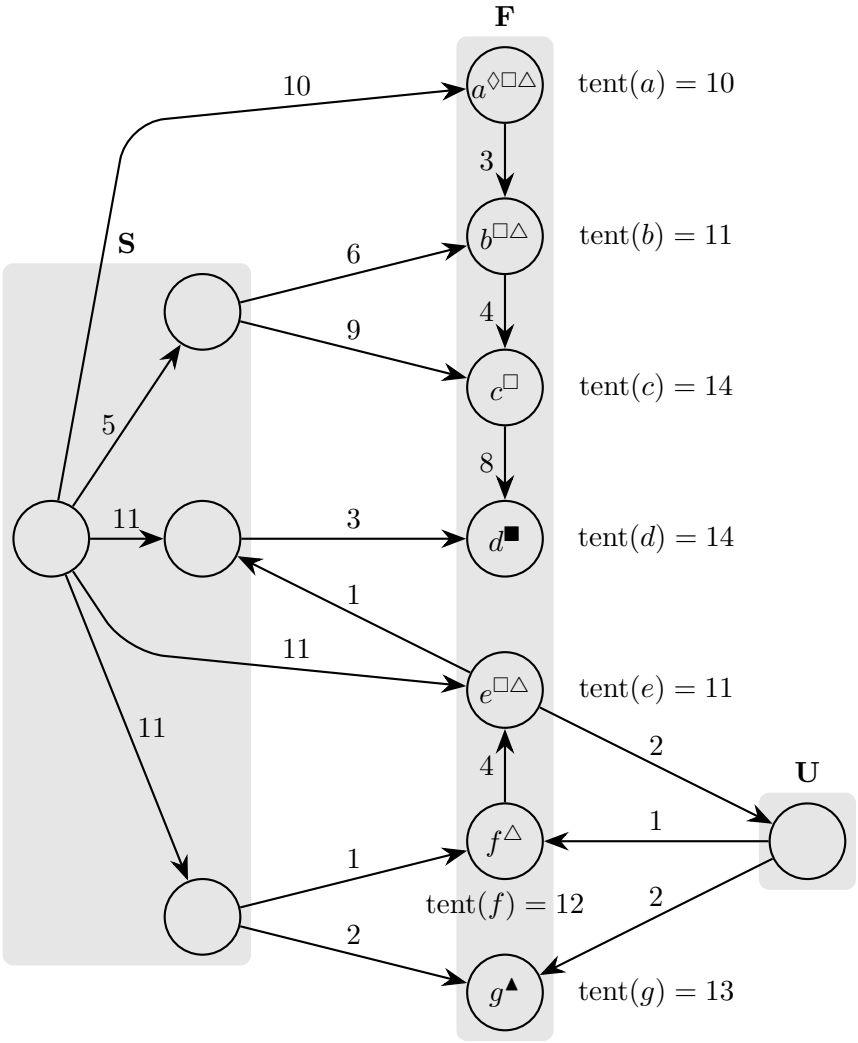

Figure 3.2: A demonstration of Crauser's criteria. **Dijkstra's algorithm** ($\Diamond$) would just settle a. **Crauser's In criteria** ($\Box$) uses $M = 10$ (the minimum tentative distance) as threshold, and settles a (tent$(a) - 10 = 0$), b (tent$(b) - 3 = 8$), c (tent$(c) - 4 = 10$), and e (tent$(e) - 4 = 7$). The **dynamic** ($\blacksquare$) variant additionally finds d (tent$(d) - 8 = 6$, not minus 3). **Crauser's Out criteria** ($\triangle$) uses $L = 12$ (see node e: tent$(e) + 1 = 12$) as threshold, and therefore settles a, b, e, and f because their tentative distances are less than or equal to L. For the **dynamic** ($\blacktriangle$) variant $L = 13$ because the outgoing edge back into S from e is ignored, therefore tent$(e) + 2 = 13$ (not plus 1). This additionally settles the node g. One can see that all criteria settle the node Dijkstra's algorithm would have chosen. Additionally, there are nodes that are settled by the In criteria but not by the Out criteria and the other way around.

Theorem 3.16. *The dynamic variant of Crauser's In criteria is sound.*

Proof. Consider any chosen node v and its tentative predecessor p. To prove soundness the possibility of a better predecessor p', i.e., one that fulfills $\text{dist}(p') + \text{cost}(p', v) < \text{dist}(p) + \text{cost}(p, v)$, is ruled out.

By the same argument as in the proof of Dijkstra's algorithm, p' cannot possibly be in S. It remains to show that p' cannot exist in F or U as well (proof of Lemma 3.2, first half).

The minimal cost to reach $F \cup U$ from s is M (proof of Lemma 3.2, second half). Therefore $M \leq \text{dist}(p')$. The additional edge from p' to v has a cost of $\text{cost}(p', v)$. Therefore, the path $\pi' = [..., p', v]$ has a cost of at least $M + cost(p', v)$. Furthermore, $\min_{p \in F \cup U} \text{cost}(p, v) \leq \text{cost}(p', v)$ by definition. This leads to a lower bound for the cost of π' of $M + \min_{p \in F \cup U} \text{cost}(p, v)$.

By the criteria $\text{tent}(v) - \min_{p \in F \cup U} \text{cost}(p, v) \leq M$ holds. This is equivalent to $\text{tent}(v) \leq M + \min_{p \in F \cup U} \text{cost}(p, v)$. The second half of this inequality is equal to the previously found lower bound for the cost of π'. This finally leads to:

$$\overbrace{\text{tent}(v)}^{\text{tentative distance}} \leq \overbrace{M + \min_{p \in F \cup U} \text{cost}(p, v)}^{\text{bound given by the criteria}} \leq \overbrace{\text{dist}(p') + \text{cost}(p', v)}^{\text{cost of the alternative path}}$$

Therefore $\text{dist}(p') + \text{cost}(p', v) < \text{dist}(p) + \text{cost}(p, v) = \text{tent}(v)$ cannot hold and no better alternative path via p' exists. Therefore, $p' \notin F \cup U$. □

Corollary 3.17. *By Theorem 3.7 the static variant is sound as well.*

Theorem 3.18. *The static variant of Crauser's In criteria is complete.*

Proof. In each phase at least the node that induced the value of M, i.e., the one with the minimal tentative distance, is settled. This is because for this node $i(v) \leq M$ has to hold even if the minimal-cost incoming edge has cost zero. □

Corollary 3.19. *By Theorem 3.8 the dynamic variant is complete.*

3.7.2 Crauser's Out Criteria

While Crauser's In criteria focuses on incoming edges, Crauser's Out criteria [20] focuses on outgoing edges. Here, the observation is that each new path that can be found has to have at least the cost of reaching some node v in F plus an outgoing edge of v. Therefore, all nodes whose tentative distance is already smaller than the smallest of these values can be settled since there is no way that any new path could reduce the tentative distance of these nodes. An example can be seen in Figure 3.2.

In its *static* variant this criteria settles a node v if it fulfills $\text{tent}(v) \leq L$ where

$$L = \min_{f \in F, f' \in V} (\text{tent}(f) + \text{cost}(f, f'))$$

characterizes the minimum cost of any new path that could possibly be found as argued above. Therefore, all nodes with a tentative distance lower than this bound can be relaxed.

In the *dynamic* variant L is defined as

$$L = \min_{f \in F, f' \in F \cup U} (\text{tent}(f) + \text{cost}(f, f'))$$

with the same intuition as for Crauser's In criteria, i.e., if a node is already settled it cannot have any influence on shortest paths of unsettled nodes.

Lemma 3.20. *If $v \in F \cup U$ and $\text{tent}(v) = \text{dist}(v)$, then $L \leq \text{dist}(v) + \text{cost}(v, v')$ with v' being any successor of v in F or U.*

Proof. If v is unreachable, $\text{tent}(v) = \text{dist}(v) = \infty$ and the lemma holds trivially.

If v is reachable, it has to be in F. This leads to:

$$L = \min_{f \in F, f' \in F \cup U} (\text{tent}(f) + \text{cost}(f, f')) \leq \text{tent}(v) + \text{cost}(v, v') = \text{dist}(v) + \text{cost}(v, v')$$

The last equality in this equation holds because by assumption $\text{tent}(v) = \text{dist}(v)$. □

Lemma 3.21. *If $v \in F \cup U$ and v is reachable, then the shortest path to v contains some node $q \in F \cup U$ that fulfills $\text{tent}(q) = \text{dist}(q)$.*

Proof. The shortest path for any node in F can be split in two parts. The path starts at the starting node followed by any amount of nodes in S. Then the path has any amount of nodes in $F \cup U$ followed by v itself, which is in $F \cup U$ by the precondition of this lemma.

The proof is by induction over the number of nodes in the second part, the ones in $F \cup U$, of the shortest path to v.

The base case is that the number of nodes in the second part is 1. In this case $q = v$ because $\text{tent}(v) = \text{dist}(v)$ has to hold. This is because the correct predecessor of v is already in S, and therefore has already to be the tentative predecessor of v. This means that the tentative distance of v already has to be correct.

The induction hypothesis is that the lemma holds for all nodes of the second half of the path.

In the induction step the second half of the path is extended by one node. Since the path now has at least two nodes in $F \cup U$, and by the induction hypothesis for all nodes, except the new one, of the second half the lemma holds, the lemma holds for the new node as well. This is because the new node is guaranteed to have a predecessor for whom the lemma holds, and the lemma itself propagates to successors in a path. □

Theorem 3.22. *The dynamic variant of Crauser's Out criteria is sound.*

Proof. Consider any node v and its tentative predecessor p for which the criteria holds true. Any better predecessor $p' \neq p$ is ruled out:

As already argued in the first half of Lemma 3.2 $p' \notin S$. It is therefore enough to show that no better predecessor $p' \in F \cup U$ exists.

By Lemma 3.21 there exists a node q in the path $[s, ..., p']$ such that $\text{tent}(q) = \text{dist}(q)$. Applying Lemma 3.20 to this node q means that the cost to leave q, and therefore the cost to leave p', is at least L. Since $\text{tent}(v) \leq L$, this implies that p' cannot possibly be a better predecessor, i.e., $\text{dist}(p') + \text{cost}(p', v) < \text{tent}(v)$ cannot be true. □

Corollary 3.23. *By Theorem 3.7 the static variant is sound.*

Theorem 3.24. *The static variant of Crauser's Out criteria is complete.*

Proof. In each phase at least the node that defines L can be settled, therefore all nodes in F will eventually be settled. □

Corollary 3.25. *By Theorem 3.8 the dynamic variant is complete.*

3.8 Heuristic

The idea behind this criteria is the following: If one could estimate the distance to reach all predecessors of a node a priori, it would be possible to decide which predecessors are irrelevant and will never lead to a better tentative distance. Once there are no relevant predecessors left, it is possible to settle the node. This is achieved by introducing a heuristic function to estimate the distance to each node.

A heuristic is considered *admissible* if $h(v) \leq \text{dist}(v)$, i.e., the heuristic always has to underestimate the real distance. Given an admissible heuristic $h(v)$, this criteria settles a node if

$$\text{tent}(v) \leq \min_{p \in F \cup U} (h(p) + \text{cost}(p, v))$$

Theorem 3.26. *Given an admissible heuristic $h(v)$, the heuristic criteria is sound.*

Proof. Again, the only interesting case to prove is if for any chosen node v with the tentative predecessor p there exists a better predecessor p' in $F \cup U$.

By construction of h it follows that $h(p') \leq \text{dist}(p')$. Therefore, the path $[..., p', v]$ has to have cost of at least $h(p') + \text{cost}(p', v)$. Furthermore, v was chosen because $\text{tent}(v) \leq \min_{p \in F \cup U}(h(p) + \text{cost}(p, v))$ and $\min_{p \in F \cup U}(h(p) + \text{cost}(p, v)) \leq h(p') + \text{cost}(p', v)$ holds by definition. This leads to:

$$\overbrace{\text{tent}(v) \leq \min_{p \in F \cup U} (h(p) + \text{cost}(p, v))}^{\text{holds by the criteria}} \leq \overbrace{h(p') + \text{cost}(p', v) \leq \text{dist}(p') + \text{cost}(p', v)}^{\text{holds by the admissibility of } h}$$

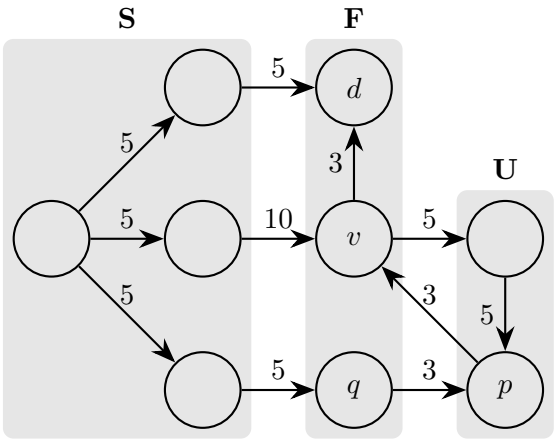

Figure 3.3: An example for a node v that might be a candidate for Träff's Bridge criteria. $\text{tent}(d) = 10$ is the minimum tentative distance, while $\text{tent}(v) = 15$ and $\text{tent}(q) = 10$ as well. Since v does not have any predecessor in F it might be eligible for Träff's Bridge criteria. It remains to find the length of the cheapest two-step backwards path from v through U. In this case the path is $[q, p, v]$ with a cost of 6. Since $\text{tent}(v) - 6 = 9$ is less than the threshold of 10, v is indeed settled by Träff's Bridge criteria.

Therefore, $\text{dist}(p') + \text{cost}(p', v) < \text{dist}(p) + \text{cost}(p, v)$ cannot hold. Therefore, $p' \notin F \cup U$. □

The heuristic strategy is not complete, just consider $h(v) = 0$ for all nodes: In this case $\text{tent}(v) \leq \min_{p \in F \cup U}(0 + \text{cost}(p, v))$ would have to hold, which is impossible once the tentative distance of a node is larger than the cost of any incoming edge. Therefore, once such a node exists in the graph this node will never be settled by this criteria, making the Heuristic criteria incomplete.

3.9 Träff's Bridge Criteria

If one considers a node that does not have predecessors in F, it is only possible to obtain a new tentative distance to this node via U. Therefore, the tentative distance of this node can only be improved if the cost of leaving S, plus entering U, plus leaving U again to reach the node is small enough to improve the tentative distance of the node in question. Träff's Bridge criteria defines a lower bound for this cost and can therefore identify some nodes that fulfill this observation. In some sense, the node forms a „bridge“ between S and U without being touched by nodes in F. The general idea is shown in Figure 3.3.

This criteria settles nodes that have no predecessors in F and fulfill the following inequality:

$$\text{tent}(v) - \min_{p \in U}(\text{cost}(p, v) + \min_{q \in V} \text{cost}(q, p)) \leq \min_{n \in F} \text{tent}(n)$$

Theorem 3.27. *Träff's Bridge criteria is sound.*

Proof. As usual consider a chosen node v with the tentative predecessor p and assume that a better predecessor p' exists. Since this criteria prohibits predecessors in F it is enough to show that p' cannot exist in U.

If p' is in U, the path to v has to look like $[..., f, ..., p', v]$ with $f \in F$. Furthermore, $f \neq p'$ because $f \in F$ and $p' \in U$, additionally $f \neq v$ because shortest paths never contain cycles, i.e., f, p' and v are disjoint nodes. As argued in Lemma 3.2 the minimal cost to reach f has to be $\min_{n \in F} \text{tent}(n)$.

Therefore, a lower bound of the cost to reach v has to be the cost to reach f, plus the distance from f to p', which can be bounded by the minimal incoming edge cost of p', plus $\text{cost}(p', v)$. This leads to a lower bound A of the cost of this path:

$$A = \overbrace{\min_{n \in F} \text{tent}(n)}^{\text{to reach } f} + \overbrace{\min_{p'' \in V} \text{cost}(p'', p')}^{\text{to reach } p' \text{ from } f} + \overbrace{\text{cost}(p', v)}^{\text{from } p' \text{ to } v} \leq \overbrace{\text{dist}(p') + \text{cost}(p', v)}^{\text{the potentially better path}}$$

By the criteria the following is true as well:

$$\text{tent}(v) \leq \min_{n \in F} \text{tent}(n) + \min_{p \in U}(\text{cost}(p, v) + \min_{q \in V} \text{cost}(q, p)) = B$$

It remains to show that $B \leq A$ since then $\text{tent}(v) \leq B \leq A \leq \text{dist}(p') + \text{cost}(p', v)$ which rules out that p' can be a better predecessor. The term $\min_{n \in F} \text{tent}(n)$ immediately cancels out, left is:

$$\min_{p \in U}(\text{cost}(p, v) + \min_{q \in V} \text{cost}(q, p)) \leq \min_{p'' \in V} \text{cost}(p'', p') + \text{cost}(p', v)$$

Assume that $\hat{p}$ and $\hat{q}$ are the p and q that induce the value of the left term. Then the left term represents the path $\pi_B = [\hat{q}, \hat{p}, v]$. This path represents the minimum cost of a path of the form $[x, y, v]$ with $x \in V$ and $y \in U$. Assume that $\hat{p}''$ is the node that induces the value of the right term. Then the right term represents a path $\pi_A = [\hat{p}'', p', v]$. Since $p' \in U$ and $\hat{p}'' \in V$, π_A is in the same class of paths for which π_B represents the minimum-cost path. Therefore, the cost of π_B has to be less than or equal to the cost of π_A, and therefore the inequality holds. This concludes the proof that $p' \notin U$. □

This criteria is not complete. The reason for the incompleteness is that it is required for nodes to not have any predecessor in F. Unfortunately, it can occur that no such node exists, i.e., that each node in F has a predecessor in F. One such example can be seen in the graph depicted in Figure 3.4. Complete graphs observe this problem as well, simply because as long as $|F| > 1$ there must be a predecessor in F for each node.

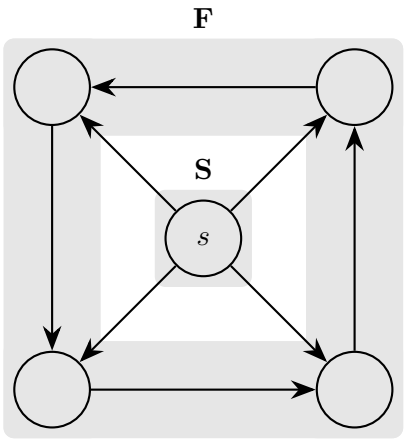

Figure 3.4: An example of a graph where Träff's Bridge criteria cannot continue after settling the starting node s because each node in F has a predecessor in F.

CHAPTER 4

Simulation

Already Crauser et al. [20, Section 6] included a simulation in their work of their algorithm to find empirical evidence of the quantitative quality of their algorithm. They claim that Crauser's Out criteria needs $2.5\sqrt{|V|}$ phases and that Crauser's Inout criteria needs $6\sqrt[3]{|V|}$ phases on average for randomly uniform graphs.

Similarly to their paper, this work contains an analysis of the quality of all criteria, in terms of number of phases, in this chapter. This chapter is divided into three sections, first the idea behind this analysis, the information obtained from it, and the drawbacks will be discussed. Second, the methodology of the analysis will be presented. Finally, the results will be depicted and discussed.

The basic idea of this analysis is to implement Algorithm 3.2 with various criteria and count the number of phases on a multitude of graph families. Furthermore, each node gets assigned a phase-number, which is the number of the phase the node was settled in. By convention, the starting node is the only node settled in the special phase 0. The basic idea of this analysis is captured in Algorithm 4.1.

The simulation is completely performance oblivious. This means that the additional overhead of various criteria is completely ignored. Unfortunately, the complexity of the criteria analyzed is non-trivial and leads in some cases to serious performance degradations, as will be seen in Chapter 5. This is even more pronounced due to the fact that Dijkstra's

for enough repititions **do**
  graph := generate a graph
  run Algorithm 3.2 on *graph*
  output the number of phases
end for

Algorithm 4.1: Simulation

algorithm performs the bare minimum of work required to solve the single-source shortest path problem, not only in an asymptotic sense, but also in absolute numbers: Each edge is only looked at once. For example, already Crauser's In criteria, one of the cheapest criteria, has to look twice at each edge: Since the criteria requires to know the smallest incoming edge for each node, it has to scan through all edges once to find the minimum for each node, leading to two accesses to each edge in the graph, i.e., double the amount than Dijkstra's algorithm.

Finally, the data obtained by this simulation represents only an average case for the chosen, theoretical classes of graphs: On one hand, the results depend heavily on the graph generation algorithm, which is a hard problem in itself and will be discussed in the next section, and on the other hand, the results are only an aggregation of multiple runs, i.e., the results only talk about the average number of phases of the given sample of graph instances. This average case analysis can be enough for practical purposes, but if worst-case guarantees are needed, e.g., in safety critical systems, or in systems with untrusted user input, this analysis might not be enough.

Since the analysis includes an algorithm that is known to be optimal in the number of phases, namely the oracle of Section 3.6, it is possible to quantitatively measure the difference of each criteria to the optimum. This might serve as inspiration for further research into promising criteria or as empirical base for proving certain probabilistic bounds on criteria. Furthermore, the analysis can be used to explore differences between various types of input graphs.

4.1 Methodology

The simulation consists of three main ingredients: the generated graph, the criteria executed on the graph, and the processing of the output data. The simulation is performed by a *simulation tool* that is written in C++ and is available for the general public.[1]

4.1.1 Input Graphs

Input graph generation is a difficult problem for itself, since it is hard to characterize the structure of real-world graphs like social networks or road networks. Furthermore, there are multiple vastly different classes of real-world graphs and it depends solely on the use-case which graph class is applicable. So, although there are public datasets of big graphs, e.g., Stanford's Large Network Dataset Collection [46] or the graphs of the Ninth DIMACS Implementation Challenge [24], they cannot be used for a statistical analysis, since each such graph is just a single instance of an unspecified graph class. Nevertheless, such single instances can be analyzed with respect to the number of phases, and the tool supports reading graphs from input files.

[1] https://github.com/kaini/sssp-simulation

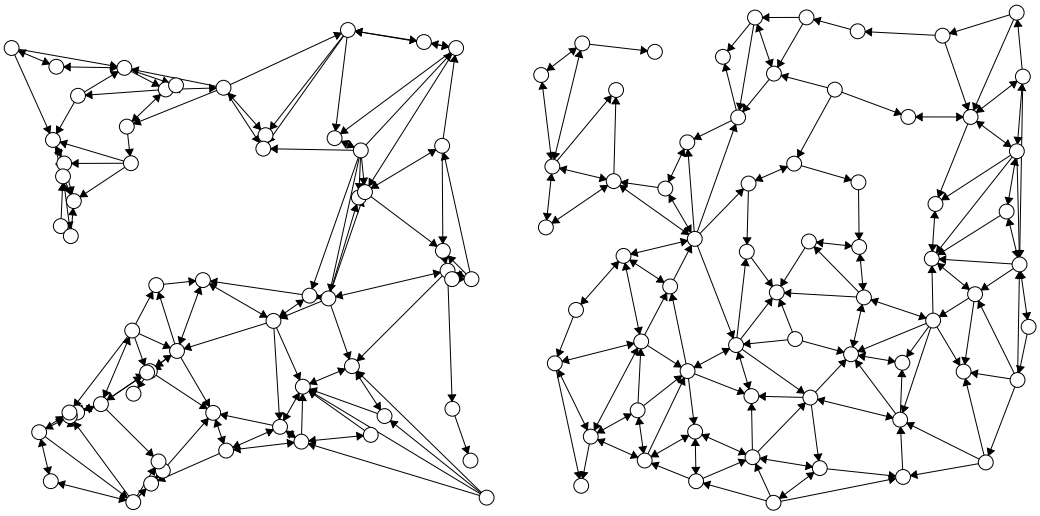

Figure 4.1: The difference between uniformly sampled positions (left) and Poisson disk sampled positions (right). One can clearly see clusters and holes in the uniformly sampled positions, while the Poisson disk sampled positions are much more regular.

The generation of graphs is randomized but seeded, i.e., on a single system it is possible to generate the same graphs if one supplies the same seed and graph generation parameters. Graph generation is split into three phases: First, the nodes are generated and positions are assigned to them. Node positions are irrelevant for most types of graphs, they are only needed for the visualization and for Euclidean distances. Second, the edges are generated. Third, the cost function is generated.

Nodes and Positions

The simulation tool provides two ways to initialize the position of nodes. The simplest method of doing so is to uniformly distribute the node positions in a $\langle 0, 0\rangle$ to $\langle 1, 1\rangle$ square.

Definition 4.1 (Uniformly Distributed Positions)**.** *The positions of the nodes in a graph are distributed uniformly if each position* $\langle x, y\rangle \sim \langle \mathcal{U}(0, 1), \mathcal{U}(0, 1)\rangle$.

While uniformly distributed positions are well understood mathematically, they are not visually pleasing since they tend to generate holes and clusters in the distribution of the nodes. It is well known that humans do not consider uniformly distributed positions as „pretty.“ To solve this issue it is possible to generate node positions using so-called Poisson disk sampling. The difference between those two can be seen in in Figure 4.1.

Poisson disk sampling is similar to uniformly random positions with the single exception that each position must have a minimum distance to all other positions. A naïve algorithm to achieve this, is to randomly generate a candidate point and check it against all already

chosen points. If there is one point such that the minimal distance is violated, reject the current candidate and try again. This approach is very inefficient. Bridson [10] describes a simple and efficient Poisson disk sampling procedure in a beautiful one-page paper, which was implemented in the simulation tool. Davies [22] published a nice, interactive visualization of Bridson's algorithm on his website.

Definition 4.2 (Poisson Disk Distributed Positions)**.** *The positions of the nodes in a graph are Poisson disk distributed if their positions were obtained by using a sampling procedure as described above.*

Poisson disk sampled positions are not of interest for the purposes of this analysis and will only serve as basis for the visualizations generated by the tool.

Edges

The simplest way of generating edges is to consider each possible edge with a given probability. This leads to the first method of generating edges:

Definition 4.3 (Uniformly Distributed Edges)**.** *The edges of a graph are uniformly distributed if the existence of each possible edge, excluding self loops and multiple edges between nodes, is determined by $\mathcal{A}(p)$.*

A graph with v vertices and uniformly distributed edges with the probability p is also called an *uniformly random graph*. Uniformly random graphs are well understood mathematical objects and enjoy a lot of mathematical properties that can come in handy for proofs. For details one can for example reference the work of Karp [43]. One notable property is that connected components of such graphs are either very large or very small [43, Section 2.1].

Since there are $|V|(|V|-1)$ possible edges, the number of edges is $|E| \sim \mathcal{B}(|V|(|V|-1), p)$. The number of outgoing (or incoming) edges for each node is $|\text{incoming}(v)| \sim |\text{outgoing}(v)| \sim \mathcal{B}(v-1, p)$.

A naïve algorithm to generate such graphs is to draw a random number from 0 to 1 for each possible edge and check that it is below p. This approach takes $\mathcal{O}(|V|^2)$ time. Using the distributions described in the previous paragraph it is possible to devise a much faster, $\mathcal{O}(|V| + |V|^2 p) = \mathcal{O}(|V| + \mathbb{E}(|E|))$ in the average case, algorithm. Instead of considering each possible edge, one considers each node and draws the number of outgoing edges. Then one takes this amount of random destinations to create the edges. This algorithm can be seen in Algorithm 4.2.

Uniformly random graphs are interesting from a mathematical perspective, but they are not appropriate for visualization purposes since the edges are all over the place and it is impossible for a human observer to discern the structure of the graph. For this reason the tool is able to generate planar graphs.

Definition 4.4 (Planar Graph)**.** *A graph is planar if it is possible to arrange the nodes in two-dimensional space in such a way that no two edges cross each other.*

Require: a set of nodes V and an edge probability p
Ensure: created uniformly distributed edges for V
  $nodes \coloneqq [0, 1, ..., |V| - 1]$
  for all $v \in V$ **do**
    $edges \coloneqq$ draw from $\mathcal{B}(|V| - 1, p)$
    shuffle $edges + 1$ elements of $nodes$ {e.g. with a partial Fisher-Yates shuffle [28]}
    for all $0 \leq i < edges$ **do**
      if $nodes[i] = v$ {to avoid self-loops} **then**
        add edge $\langle v, nodes[edges] \rangle$ to graph
      else
        add edge $\langle v, nodes[i] \rangle$ to graph
      end if
    end for
  end for

Algorithm 4.2: Fast Sampling of Uniformly Distributed Edges

The tool does not try to generate uniformly random planar graphs or some designated class of planar graphs. The algorithm simply takes the positions of the nodes and tries to add edges that do not intersect with already added edges and furthermore avoids too long edges. This leads to a highly biased selection from the set of all planar graphs, is mathematically intractable, and therefore is not suited for any analysis. Nevertheless, this algorithm is simple to implement and leads to graphs that are pleasing to look at.

The last method to generate edges is to generate layered graphs.

Definition 4.5 (Layered Graph)**.** *Assume that all nodes are classified into n classes. Then nodes in the ith class may only connect to nodes in the $(i+1)$th and $(i-1)$th class.*

Layered graphs were added to the tool out of curiosity with the hope of interesting results. 2-layered graphs, also called bipartite graphs, i.e., a graph where only the classes 0 and 1 exist and all paths have to alternate between these two classes at each step, are a well-known sub-class of this class.

An example for all three types of graphs can be seen in Figure 4.2.

Edge Costs

The only piece missing for finished graphs is the edge-cost function. The tool implements the following three cost functions:

Definition 4.6 (Uniform Edge Costs)**.** *The cost for each edge is drawn independently from $\mathcal{U}(0, 1)$.*

This is the usual definition of edge costs for uniformly random graphs, as used in the original analysis of Crauser's criteria and Meyer's Δ-stepping. Note, that the actual

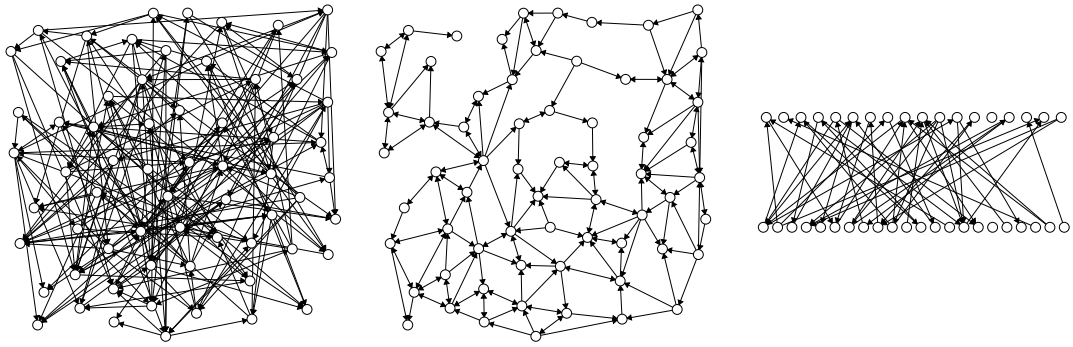

Figure 4.2: Example of an uniformly random graph (left), a planar graph (center), and a 2-layered graph (right). Notice how in the 2-layered graph nodes connect only from the lower half to the upper half and vice-versa. The nodes in the layered graph are arranged this way for clarity only, in the simulation these nodes are still located at random positions in their respective half of the plane, but of course, their overall structure remains layered.

range of the uniform cost is irrelevant as long as it is non-negative, since all ranges can be mapped into a zero-to-one range.

Definition 4.7 (Constant One Edge Costs)**.** *All edges have cost 1.*

This cost function degenerates the single-source shortest path problem into a breadth-first search problem.

Definition 4.8 (Euclidean Distance Edge Costs)**.** *Given an edge between the nodes v and v' with v being at the position $\langle x_v, y_v \rangle$ and v' being at the position $\langle x_{v'}, y_{v'} \rangle$, then the cost of this edge is $\sqrt{(x_v - x_{v'})^2 + (y_v - y_{v'})^2}$.*

This definition of the cost function simply takes the length of a straight line between the two nodes connected by each edge.

Kronecker Graphs

The graphs generated so far are very synthetic and highly unrealistic in the sense that they do not look like and do not enjoy the properties of real-world networks, like social networks, citation networks, or messaging patterns in communities [45, Section 2.1 & 2.2]. These types of graphs have some properties that are not captured by the methods of generating graphs defined so far.

One of these properties is that there are a few nodes with a lot of edges, e.g., celebrities in social networks, and a lot of nodes with comparatively few edges. Furthermore, such networks have some form of a self similarity, i.e., the structure on the macro level is similar to the structure on the micro level. Lastly, natural networks have a very low effective diameter. The Kronecker graph paper [45, Section 2.1] gives an overview of

these properties. This is also the reason for the well-known paradox that everyone knows everyone else via a very low amount of intermediate persons. For example, on Facebook everyone knows everyone else on average with three and a half hops [6].

Kronecker graphs [45] aim to realistically model networks with such properties. The authors prove that Kronecker graphs fulfill several properties which are typical for natural networks and furthermore provide empirical evidence that their method of generating synthetic networks leads to networks which are very close to natural networks.

To define Kronecker graphs the Kronecker product has to be defined.

Definition 4.9 (Kronecker Product, Kronecker Power)**.** *The Kronecker product of two matrices A and B with the dimensions $m_A \times n_A$ and $m_B \times n_B$ is defined as*

$$A \otimes B = \begin{pmatrix} a_{1,1}B & a_{1,2}B & \dots & a_{1,n_A}B \\ a_{2,1}B & a_{2,2}B & \dots & a_{2,n_A}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m_A,1}B & a_{m_A,2}B & \dots & a_{m_A,n_A}B \end{pmatrix}$$

where the resulting matrix is a $m_A m_B \times n_A n_B$ matrix.

The Kronecker power A^k is defined as

$$A^k = \underbrace{A \otimes \cdots \otimes A}_{k\text{-times}}$$

with the result being a $m_A^k \times n_A^k$ matrix.

A Kronecker graph is now defined as the Kronecker power of the adjacency matrix of some initiator graph. Usually the initiator graph is very small, i.e., the adjacency matrix of the initiator graph has the size 2×2 or 3×3. Since the Kronecker power observes an exponential growth in the matrix size, the generated graphs become large very fast. Since adjacency matrices are 0/1-matrices, all Kronecker powers lead to 0/1-matrices as well, and therefore induce exactly one graph for each power.

In order to use this technique to define whole families of graphs one replaces the adjacency matrix with a probability matrix A such that A^k has the size $|V| \times |V|$. The number in cell i, j represents the probability that the edge $\langle i, j \rangle$ exists in the graph. It is important to allow self-edges in these matrices, otherwise whole blocks in higher powers become zero, therefore ruling out many edges that should be possible to be generated. Later, a method will be introduced to get rid of self-edges in the final graph. The whole matrix represents a set of independent Bernoulli distributions and therefore represents in whole a Poisson binomial distribution. A visualization of Kronecker powers on probability matrices can be seen in Figure 4.3.

To explain the implemented algorithm to generate such graphs, the following properties are required.

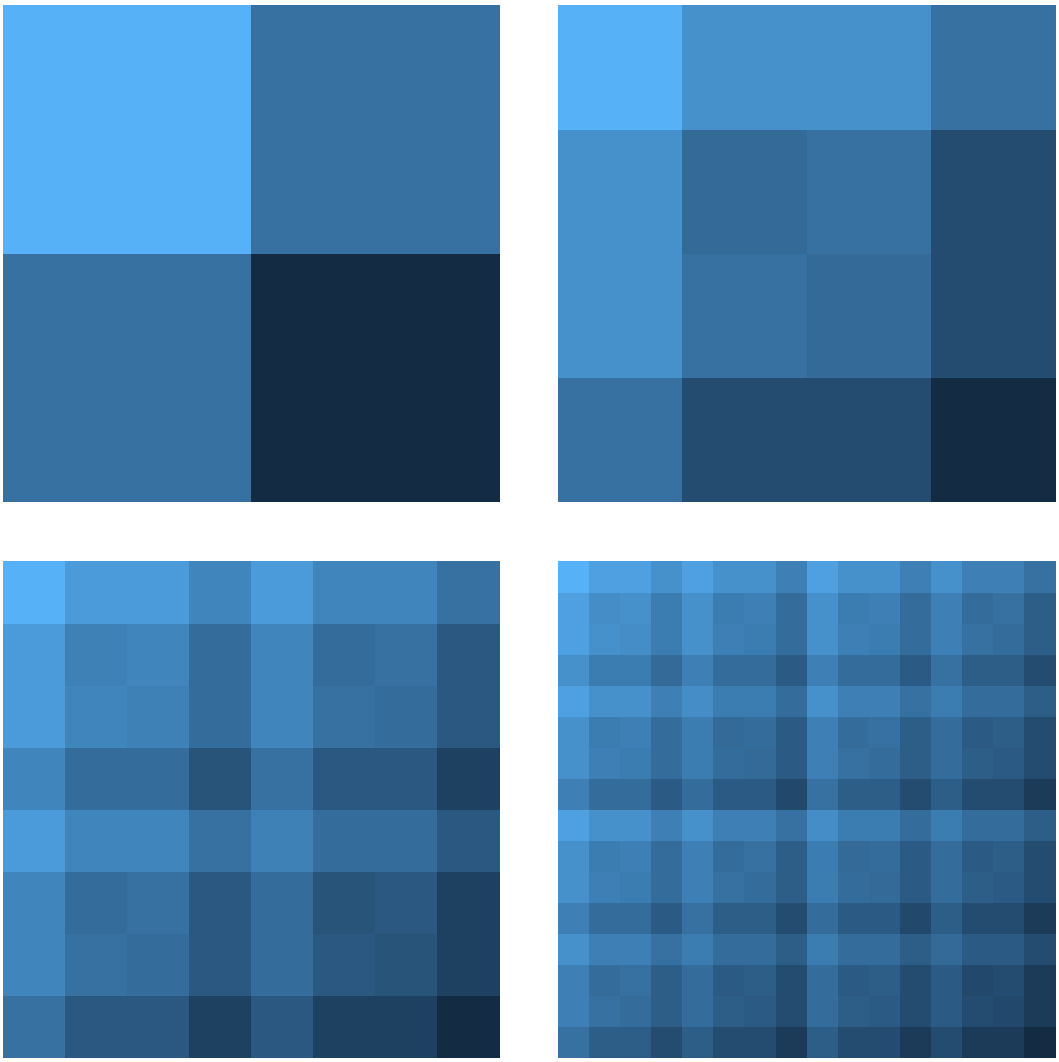

Figure 4.3: A visualization of the Kronecker process for $A = (0.57, 0.19; 0.19, 0.05)$: top left A^1, top right A^2, bottom left A^3, bottom right A^4. The colors are renormalized in each step such that the highest probability is bright and the lowest probability is dark. Furthermore, the color scale is logarithmic to make the small probabilities in higher powers more visible. The matrix A is the same as used by the Graph 500 benchmark [35].

Lemma 4.1. $\sum(A \otimes B) = (\sum A)(\sum B)$, *where* $\sum X$ *means the sum of all elements in* X.

Proof. That this holds can be seen if the sum over the Kronecker product is written explicitly. The definition of the Kronecker product is

$$A \otimes B = \begin{pmatrix} a_{1,1}B & a_{1,2}B & \dots & a_{1,n_A}B \\ a_{2,1}B & a_{2,2}B & \dots & a_{2,n_A}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m_A,1}B & a_{m_A,2}B & \dots & a_{m_A,n_A}B \end{pmatrix}$$

which leads to

$$\begin{aligned} \sum(A \otimes B) &= (\sum a_{1,1}B) + (\sum a_{1,2}B) + \cdots + (\sum a_{m_A,n_A}B) \\ &= a_{1,1}(\sum B) + a_{1,2}(\sum B) + \cdots + a_{m_A,n_A}(\sum B) \\ &= (a_{1,1} + a_{1,2} + \cdots + a_{m_A,n_A})(\sum B) \\ &= (\sum A)(\sum B) \end{aligned}$$

□

Theorem 4.2. *The expected number of edges in a Kronecker graph* A^k *is* $\mathbb{E}(|E|) = (\sum A)^k$.

Proof. By definition $|E| \sim \mathcal{PB}(A^k)$, with the expected value $\mathbb{E}(|E|) = \sum(A^k)$. That this is equal to $(\sum A)^k$ is an immediate consequence of Lemma 4.1. □

An exact implementation of a sampling algorithm for Kronecker graphs would have to explicitly calculate A^k and then sample the Bernoulli distribution for each cell. The last step alone takes $\mathcal{O}(|V|^2)$ steps and space, and is therefore way too inefficient to generate large graphs.

For this reason the authors propose a more efficient sampling algorithm in their paper [45, Section 3.6]. This algorithm can be seen in Algorithm 4.3.

The basic idea is that one first samples the number of edges using a Poisson distribution, which can be done without calculating A^k due to Theorem 4.2, i.e., just sample from $\mathcal{P}((\sum A)^k)$.

With the number of edges set, it remains to decide between which nodes these edges are going to be placed. This will be decided independently for each edge. The location of each edge is determined by a recursive descent through the initiator matrix.

For simplicity assume that the initiator matrix A is a 2×2 matrix. In the first step a random cell of A is chosen, of course proportionally to the probabilities of each cell. This first choice determines the quadrant the final edge is going to end up in the adjacency matrix.

Require: an initiator matrix A with dimensions $n \times n$ and the parameter k
Ensure: created a Kronecker graph without double-edges and self-edges
  $A_\Sigma \coloneqq$ inclusive prefix sum of A
  edges $\coloneqq$ sample from $\mathcal{P}((\sum A)^k)$
  for all $0 \leq e < edges$ **do**
    cell $\coloneqq 0$
    granularity $\coloneqq 1$
    for all $0 \leq i < k$ **do**
      value $\coloneqq$ sample from $\mathcal{U}(0, \max A_\Sigma)$
      index $\coloneqq$ find index of first element not less than *value* in A_Σ {binary search}
      cell $\coloneqq$ *cell* + *index* $*$ *granularity*
      granularity $\coloneqq$ *granularity* $* n * n$
    end for
    $\langle u, v \rangle = \langle cell \text{ div } n^k, cell \text{ mod } n^k \rangle$
    emit edge $\langle u, v \rangle$ if it does not exist yet and $u \neq v$; redo full iteration otherwise
  end for

Algorithm 4.3: Fast Sampling of Kronecker Graphs [45, Section 3.6]

Due the recursive nature of Kronecker graphs, the chosen quadrant itself is split again into four quadrants with the probabilities of A. Therefore, again a random cell of A is chosen to determine the quadrant in the quadrant where the edge is going to end up.

This process repeats k times. After these k times, the final cell in the adjacency matrix for the edge has been determined by repeatedly choosing a quadrant. The edge is going to be the one corresponding to the final cell. If at this point it is detected that this would lead to a self-edge or a double-edge the process is repeated.

This process can be compared with zooming into a fractal, with the exception that the process stops after k steps. The assumption of a 2×2 initiator matrix was made to be able to use the word quadrant. If the initiator matrix is a 3×3 matrix one would have to talk about ninths, 4×4 would correspond to sixteenths, and so on.

While this method does not exactly yield Kronecker graphs, the authors claim that graphs generated using this method are essentially indistinguishable from real Kronecker graphs. Additionally, this algorithm is a direct generalization of the R-MAT algorithm [14] to generate random graphs.

The number of edges in the resulting graph is directly encoded in the initiator matrix A, as can be seen in the assignment to *edges* in the sampling algorithm. This especially means that if $\sum A = 1$ the number of edges for any power of A stays 1. Therefore, it is recommended to define the initiator matrix such that $\sum A = 1$ and then multiply it by some constant to control the expected number of edges.

Kronecker graphs, especially R-MAT generated graphs, have been used by various researchers to analyze their algorithms [13, 21, 49, 50] and are used by the Graph 500

benchmark [35].

4.1.2 Implemented Criteria

The simulation tool implements all criteria defined in Chapter 3 in such a way that they can be mixed and matched arbitrarily at runtime. This allows an analysis of all interesting combinations of criteria. While this feature incurs a runtime overhead, this does not matter for the purposes of a simulation because all measures taken from the simulation are run-time oblivious.

This is achieved by implementing an algorithm scheme similar to the one presented in Algorithm 3.2. The only change made to this scheme is that it is extended with callbacks that have to be called to feed the implementations of the criteria with information to allow them to perform their calculations. Each criteria is solely defined by the following interface:[2]

initialize(graph, starting node) Called once to initialize the criteria to be prepared to solve the single-source shortest path problem for the given graph and starting node.

changed predecessor(node, predecessor, distance) Called when a node gets a new predecessor, and therefore a new distance, assigned.

relaxed node(node) Called when a node has been moved from the fringe state into the settled state.

relaxable nodes() Called at the start of each phase to get a list of the nodes that can be settled in the current phase. Of course it is no error if multiple criteria propose the same node.

This usage of this interface can be seen in in Algorithm 4.4. Basically, this algorithm is an implementation of Dijkstra's algorithm with the exception that Dijkstra's criteria is *not* used to decide the nodes to be settled. Instead, the callbacks of all passed criteria are called to build a set of nodes to be settled in each phase. Whenever the tentative distance of a node changes, or whenever a node is being settled, this is communicated to all criteria via the interface as well.

4.1.3 Statistical Methods

Each measurement was repeated 100 times with a different seed, i.e., a different randomly generated graph. The simulation tool outputs a single line for each phase. Each line consists of information about the graph generation and results of the simulation. Fields that are irrelevant, for example „uniform node count“ if „position algorithm“ is set to „poisson,“ are set to the special null-value `NA`. A list of all fields follows:

[2] See `criteria.cpp` for details.

Require: a directed graph $G = \langle V, E \rangle$, a starting node s, and a set of criteria Φ
Ensure: $\forall v \in V : \mathit{tent}[v] = \mathrm{dist}(v)$ if Φ is sound and complete
  for all $\varphi \in \Phi$ **do**
    φ.initialize(G, s)
    φ.changed predecessor(s, null, 0)
  end for
  for all $v \in V$ **do**
    $\mathit{tent}[v] \coloneqq \infty$
  end for
  $\mathit{tent}[s] \coloneqq 0$
  loop
    $\mathit{todo} \coloneqq \bigcup_{\varphi \in \Phi} \varphi$.relaxable nodes()
    if *todo* is empty **then**
      break
    end if
    for all $v \in \mathit{todo}$ **do**
      for all $\langle v, v' \rangle \in \mathrm{outgoing}(v)$ **do**
        if $\mathit{tent}[v] + \mathrm{cost}(v, v') < \mathit{tent}[v']$ **then**
          $\mathit{tent}[v'] \coloneqq \mathit{tent}[v] + \mathrm{cost}(v, v')$
          for all $\varphi \in \Phi$ **do**
            φ.changed predecessor(v', v, $\mathit{tent}[v']$)
          end for
        end if
      end for
      for all $\varphi \in \Phi$ **do**
        φ.relaxed node(v)
      end for
    end for
  end loop

Algorithm 4.4: „Dijkstra's algorithm“ utilizing a generalized interface

Position algorithm $\in \{\text{poisson}, \text{uniform}\}$: The algorithm that was used to sample the positions of all nodes.

- **Poisson minimum distance** $\in \mathbb{R}_{\geq 0}$: The minimum distance between two sampled nodes.
- **Poisson maximum reject** $\in \mathbb{N}$: The maximum number of retries for the fast Poisson sampling algorithm.
- **Uniform node count** $\in \mathbb{N}$: The number of uniformly sampled nodes.

Edge algorithm $\in \{\text{planar}, \text{uniform}, \text{layered}, \text{kronecker}\}$: The algorithm that was used to sample the edges between the nodes.

- **Planar edge probability** $\in [0, 1]$: The probability of the existence of each edge in planar graphs.
- **Uniform edge probability** $\in [0, 1]$: The probability of the existence of each edge in uniformly random graphs.
- **Layered edge probability** $\in [0, 1]$: The probability of the existence of each edge in layered graphs.
- **Layer count** $\in \mathbb{N}_{\geq 1}$: The number of layers in a layered graph.
- **Kronecker initiator matrix** $\in \mathbb{R}_{\geq 0}^{x,x}; x \in \mathbb{N}_{\geq 1}$: The initiator matrix A for Kronecker graphs.
- **Kronecker power** $\in \mathbb{N}_{\geq 1}$: The Kronecker power k.

Cost algorithm $\in \{\text{uniform}, \text{one}, \text{euclidean}\}$: The algorithm that was used to sample edge costs.

Graph file (path on the file system): If the graph was loaded from an existing file, this field contains the path of the file.

Algorithm (list of criteria): The combination of criteria that was simulated.

Seed $\in \mathbb{Z}$: The seed used for the random number generator for graph generation.

Node count $\in \mathbb{N}$: The number of nodes in the generated graph.

Phase $\in \mathbb{N}$: The phase this output line represents.

Relaxed $\in \mathbb{N}$: The number of nodes that moved from fringe to settled in this phase.

Due to the size of the graphs, the high repetition count, and the number of criteria these files reach sizes which are too large to be processed by simple R-scripts [60]. To solve this problem the files are first pre-aggregated using a small tool written in Rust [61][3]:

[3] Rust is a programming language primarily developed by Mozilla Corp. with the aim to offer a safe alternative for „systems programming," i.e., the niche that C and C++ are currently dominating. Rust was chosen to implement this aggregation tool just out of curiosity.

As described above the file contains one line for each phase for each thread. These leads to multiple gigabytes of data. The Rust tool aggregates these lines along the phase-dimension, i.e., all phases that belong to a single run are aggregated into a single line. This reduces the amount of data to multiple megabytes. These aggregated files are then fed into an R-script which groups by criteria and calculates the mean of the repeated runs.

This data is then used to apply a curve-fitting algorithm to find a matching growth functions and to plot the data as seen in the next section. Each result was fitted either using the function $a + b \cdot |V|^c$ with the three parameters a, b and c, or the function $a + b \cdot \log_2 |V|$ with the two parameters a and b. In the end, it was always very easy to identify the correct of the two functions as the p-values of the parameters of the worse fit were multiple orders of magnitudes lower than the p-values of the better fit (small p-values are evidence that the fit is bad). For example for Crauser's In criteria the logarithmic fit lead to p-values of $2 \cdot 10^{-7}$ and $3 \cdot 10^{-14}$ for a and b, i.e., an extremely small probability that the fit is good. While the polynomial fit lead to p-values of 0.92, 0.58, and 0.002 for a, b, and c.

4.2 Results

All algorithms as defined in the previous chapter are being simulated with the addition of some combinations, i.e., 1. Crauser's Out in the static variation, 2. Crauser's In in the static variation, 3. Crauser's Inout in the static variation, 4. Crauser's Out in the dynamic variation, 5. Crauser's In in the dynamic variation, 6. Crauser's Inout in the dynamic variation, 7. the heuristic (straight-line distance) with Crauser's Inout in the dynamic variation, 8. Träff's Bridge Criteria with Crauser's Inout in the dynamic variation, and 9. the oracle. The heuristic was only simulated on graphs with Euclidean edge costs because that is the only class of graphs considered in this thesis where a sensible admissible heuristic exists.

Each algorithm, except the heuristic, was simulated on the following eight graph types:

- Uniformly distributed edge costs on 1. bipartite, 2. Kronecker, and 3. uniform graphs.
- Euclidean distance edge costs on 1. bipartite and 2. uniform graphs. Since nodes in Kronecker graphs are used to model things like social networks, citation networks, or world wide web links, they do not have a natural spatial dimension. For this reason there is no notion of an Euclidean distance. Of course it would be possible to just assign random positions in the plane to the nodes of a Kronecker graph but the immediate question would be what this is supposed to mean semantically, i.e., this would be no different from assigning random edge costs, except that a strange distribution of the edge costs would emerge.
- Constant 1 edge costs on 1. bipartite, 2. Kronecker, and 3. uniform graphs.

For Kronecker graphs the initiator matrix $(0.57, 0.19; 0.19, 0.05)$ was used, which is the same that is used by the Graph 500 benchmark.

The results on graphs with uniformly distributed edge costs can be seen in Figure 4.4. There is no difference between bipartite and uniform graphs, while Kronecker graphs have fewer phases on average. Furthermore, one can observe that Crauser's static Inout leads to less phases than Crauser's dynamic In as well as Crauser's dynamic Out criteria. Only Crauser's dynamic Inout beats the static one, with the exception of Kronecker graphs where Crauser's dynamic In is already slightly better than Crauser's static Inout. Träff's Bridge criteria leads to a minuscule improvement over Crauser's dynamic Inout criteria. Notable is that all criteria observe a polynomial growth of the number of phases (with the exponent being less than 1, i.e., sublinear), while the Oracle's growth is logarithmic.

Figure 4.5 depicts the results for graphs with Euclidean distances between nodes. For the aforementioned reason, Kronecker graphs are missing in these results. The observations are similar to the ones with uniformly distributed edges, except that there is a notable difference in the number of phases for bipartite and uniform graphs. The heuristic is not able to improve Crauser's dynamic Inout criteria by any significant margin. Again, all Criteria observe polynomial growth in the number of phases, while the Oracle is logarithmic in the number of phases.

Using graphs with constant edge costs degenerates the single-source shortest path problem to a breadth-first search problem. This can be clearly observed in the simulation because all criteria are already optimal with respect to the oracle. This can be seen in Figure 4.6. Note, that all other criteria have been removed from the graph because they would perfectly overlap with the oracle. This leads to the conclusion that already the weakest of all criteria, namely Crauser's static Out criteria, is enough to solve the single-source shortest path problem on these graphs optimally with respect to the oracle. Also, the number of phases is exceptionally low, e.g., there are only 8 phases for uniform graphs with around 80000 nodes. For Kronecker graphs of this size there are only 5 phases.

The results of the curve-fitting are summarized in Table 4.1. All graphs use the fitted functions as lines, while they display the real data as points. There are almost no proven bounds on the number of phases. Crauser et al. prove the bounds for the static Out and Inout criteria on uniformly random graphs to be $\mathcal{O}(|V|^{1/2})$, and respectively $\mathcal{O}(|V|^{1/3})$, which matches the empirical data. A notable observation is that the oracle only needs $\mathcal{O}(\log|V|)$ phases in all cases. This implies that there is still a considerable gap between the theoretical optimum and the criteria utilized so far.

Furthermore, the Full USA graph from the Ninth DIMACS Implementation Challenge [24] was analyzed. This graph is a distance graph of the street network[4] of the USA. The graph consists of 58 million edges connecting 24 million nodes. Crauser's static Inout criteria needed 308 thousand phases, while the dynamic Inout criteria finished in 231 thousand phases. Notice how these numbers of phases are about a factor of 1500 higher than the

[4]The graph has many errors and cannot be used for real-world route planning but it is close enough to be realistic.

Edges	Criteria	Number of Phases for Edge Costs		
		Uniform	**Euclidean**	**One**
Bipartite	Crauser's Out (Static)	$2.51 \cdot \lvert V\rvert^{0.5}$	$3.69 \cdot \lvert V\rvert^{0.24}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's In (Static)	$2.27 \cdot \lvert V\rvert^{0.5}$	$3.25 \cdot \lvert V\rvert^{0.24}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's Out (Dynamic)	$1.7 \cdot \lvert V\rvert^{0.49}$	$3.17 \cdot \lvert V\rvert^{0.23}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's In (Dynamic)	$1.52 \cdot \lvert V\rvert^{0.45}$	$4.16 \cdot \lvert V\rvert^{0.2}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's Inout (Static)	$3.97 \cdot \lvert V\rvert^{0.34}$	$5.58 \cdot \lvert V\rvert^{0.17}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's Inout (Dynamic)	$3.91 \cdot \lvert V\rvert^{0.29}$	$6.82 \cdot \lvert V\rvert^{0.14}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Heuristic†	N/A	$7.02 \cdot \lvert V\rvert^{0.14}$	N/A
	Träff's Bridge†	$4.27 \cdot \lvert V\rvert^{0.28}$	$8.11 \cdot \lvert V\rvert^{0.13}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Oracle	$1.71 \cdot \log_2(\lvert V\rvert)$	$0.98 \cdot \log_2(\lvert V\rvert)$	$0.4 \cdot \log_2(\lvert V\rvert)$
Kronecker	Crauser's Out (Static)	$1.79 \cdot \lvert V\rvert^{0.51}$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
	Crauser's In (Static)	$2.17 \cdot \lvert V\rvert^{0.43}$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
	Crauser's Out (Dynamic)	$1.68 \cdot \lvert V\rvert^{0.42}$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
	Crauser's In (Dynamic)	$3.01 \cdot \lvert V\rvert^{0.32}$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
	Crauser's Inout (Static)	$3.49 \cdot \lvert V\rvert^{0.31}$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
	Crauser's Inout (Dynamic)	$4.03 \cdot \lvert V\rvert^{0.24}$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
	Heuristic†	N/A	N/A	N/A
	Träff's Bridge†	$4.17 \cdot \lvert V\rvert^{0.23}$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
	Oracle	$1.17 \cdot \log_2(\lvert V\rvert)$	N/A	$0.11 \cdot \log_2(\lvert V\rvert)$
Uniform	Crauser's Out (Static)	$2.48 \cdot \lvert V\rvert^{0.5}$	$3.16 \cdot \lvert V\rvert^{0.34}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's In (Static)	$2.28 \cdot \lvert V\rvert^{0.5}$	$2.78 \cdot \lvert V\rvert^{0.34}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's Out (Dynamic)	$1.66 \cdot \lvert V\rvert^{0.5}$	$2.41 \cdot \lvert V\rvert^{0.33}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's In (Dynamic)	$1.43 \cdot \lvert V\rvert^{0.46}$	$2.26 \cdot \lvert V\rvert^{0.3}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's Inout (Static)	$3.97 \cdot \lvert V\rvert^{0.34}$	$5.19 \cdot \lvert V\rvert^{0.21}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Crauser's Inout (Dynamic)	$3.75 \cdot \lvert V\rvert^{0.29}$	$6.09 \cdot \lvert V\rvert^{0.17}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Heuristic†	N/A	$6.96 \cdot \lvert V\rvert^{0.16}$	N/A
	Träff's Bridge†	$4.19 \cdot \lvert V\rvert^{0.28}$	$7.19 \cdot \lvert V\rvert^{0.15}$	$0.4 \cdot \log_2(\lvert V\rvert)$
	Oracle	$1.69 \cdot \log_2(\lvert V\rvert)$	$1.06 \cdot \log_2(\lvert V\rvert)$	$0.4 \cdot \log_2(\lvert V\rvert)$

Table 4.1: Empirical number of phases for all simulated types of graphs. Each dataset was fitted using the function $a + b \cdot |V|^c$ or $a + b \cdot \log_2(|V|)$. The term a was left for clarity. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

expected number of phases for uniform graphs with Euclidean distances. Although it is surprising that this disparity is so high, some disparity is to be expected since a street network has a vastly different structure from a uniformly random graph.

Concluding, one can observe that Crauser's original Inout criteria is already a very strong criteria. Remarkably, in most cases the Inout criteria is already stronger than the dynamic In as well as the dynamic Out criteria. Only the dynamic Inout criteria improves upon the static variant. Since, as will be seen in the next chapter, the dynamic criteria incur an expensive overhead they are not able to beat the static criteria in practice even though the number of phases is smaller. If one works with graphs without edge weights, there is absolutely no need to implement any powerful criteria. Plain Crauser's Out criteria is already enough to solve the single-source shortest path problem on such graphs in the optimal number of phases.

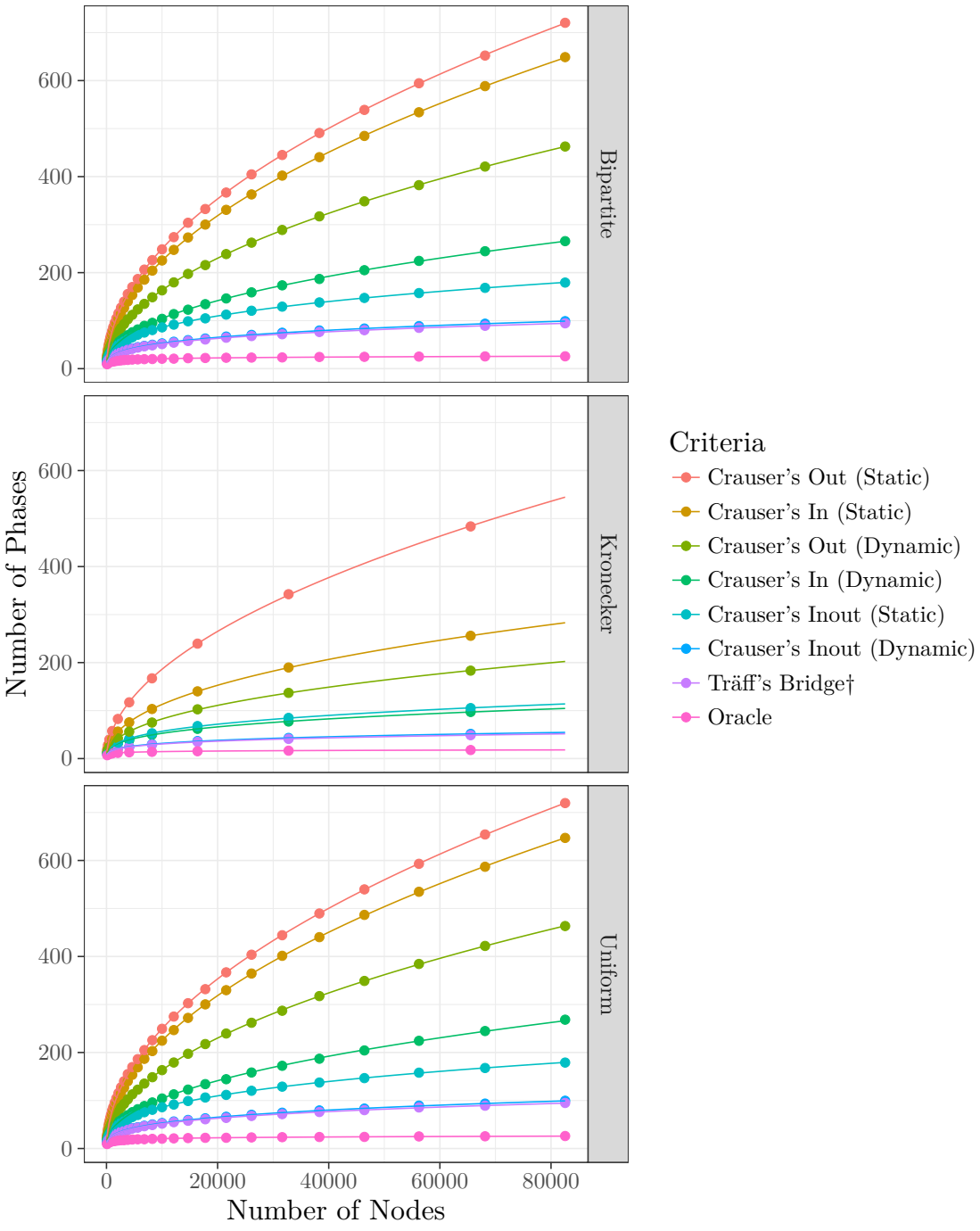

Figure 4.4: Average number of phases of various criteria on various types of graphs with uniformly distributed edge costs. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

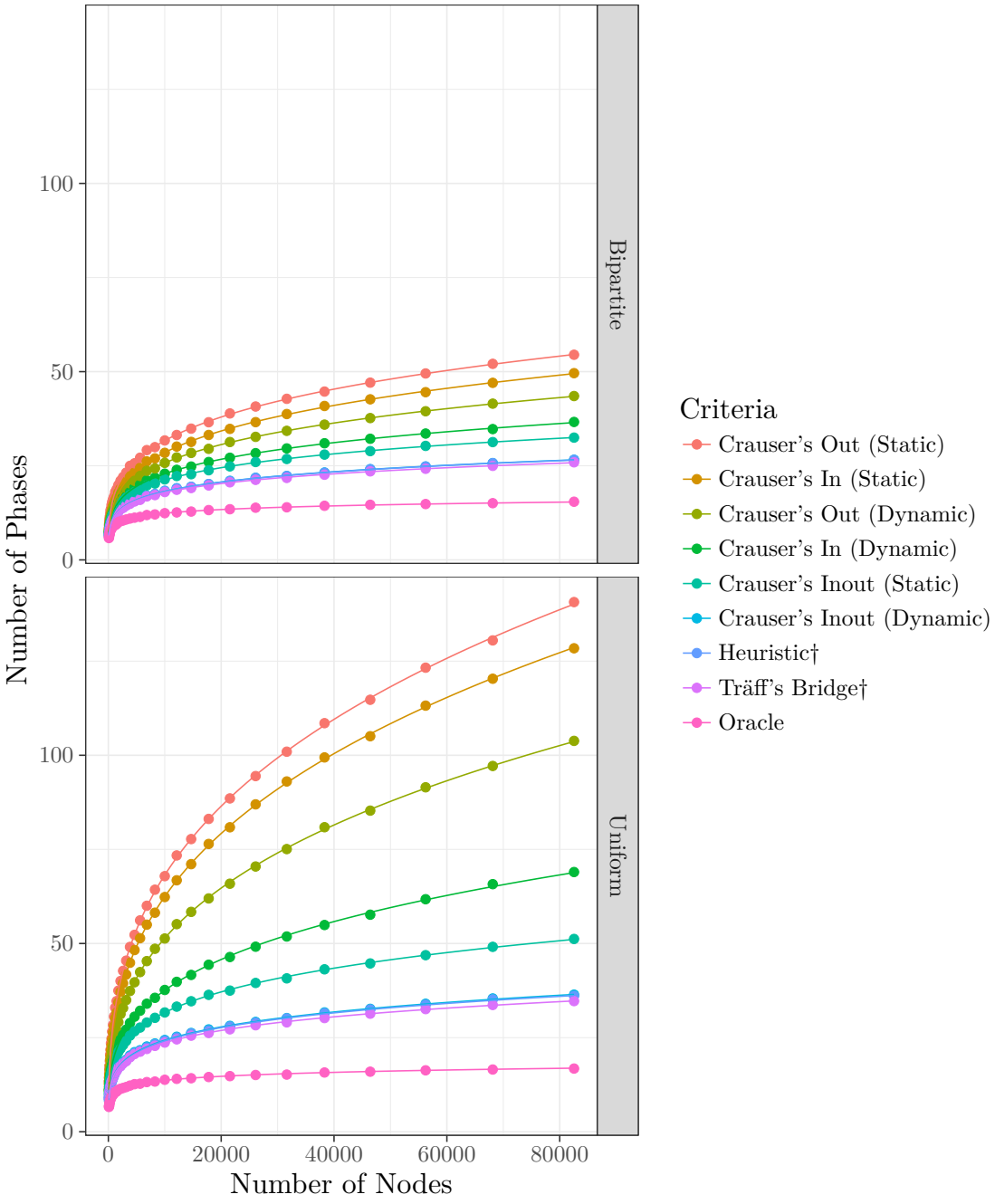

Figure 4.5: Average number of phases of various criteria on various types of graphs with the Euclidean distance as edge costs. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

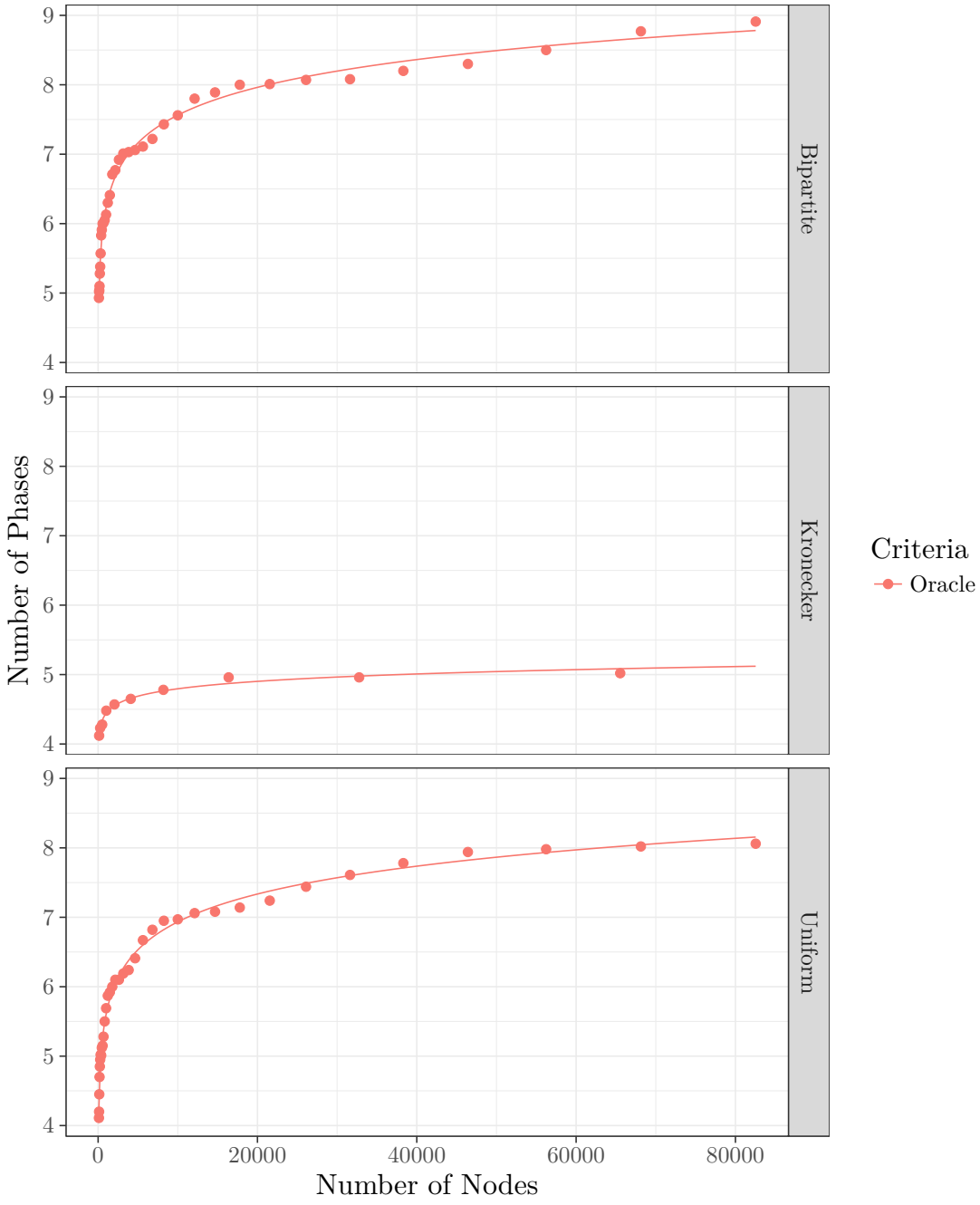

Figure 4.6: Average number of phases of the oracle on various types of graphs with the constant edge cost of 1. Note that all other criteria have been omitted since their results are the same as for the oracle.

CHAPTER 5

Implementation

This chapter gives an overview of how to decide the criteria introduced in Chapter 3 efficiently and furthermore provides some optimizations that can be implemented. This is followed by a presentation of the results of benchmark runs on two shared memory systems.

The criteria presented in this thesis have been implemented efficiently in a *benchmarking tool*. The implementation of this benchmarking tool is available for the general public.[1] The purpose of this implementation is to obtain real performance numbers for the systems provided by the research group. The benchmarking tool performs multiple steps:

1. The command-line arguments are read and parsed: The command-line arguments control the graph generation algorithm, the number of threads to be used, and optionally if the results of the algorithm should be validated against a simple implementation of Dijkstra's algorithm. The criteria to be chosen is not controlled by a command-line argument but by choosing the right executable, i.e., the build process generates an executable for each combination of criteria that has been measured. This is done this way due to performance reasons.

2. Spawn threads: The correct amount of threads is spawned. The threads are then pinned to processing units to avoid any performance impact caused by the operating system by sporadically rescheduling threads across CPUs.

3. Graph generation: Based on the command-line arguments supplied in the first step, the graph is now generated in parallel, or read from an input file. This step often takes a considerable amount of time, usually more than the single-source shortest path algorithm itself.

[1] https://github.com/kaini/sssp-shm

4. The time measurement starts once all threads reach this point.
5. Algorithm execution: The chosen algorithm with the chosen criteria is now executed.
6. The time measurement ends.
7. Validation: If requested by the user, the results of the previously run algorithm are validated against a plain implementation of Dijkstra's algorithm.

5.1 Preliminaries

The tool is written in C++ and has been tested on Microsoft Windows, Linux, and SunOS/Solaris systems. This section aims to give a brief overview of the memory model as defined by C++ followed by a description of some basic data structures that are used by the implementation of the algorithms later on.

5.1.1 Memory Model

This section gives an informal summary of the C++ memory model. For a detailed definition of the C++ memory model one has to reference the current C++ standard [39]. This section is not concerned about how the C++ memory model maps to hardware but is only concerned about describing the abstract model as defined by the standard.

On the highest level C++ is defined, unlike most modern programming languages, by utilizing an addressable RAM. Each object[2] has a non-zero size and an unique address in this memory. It suffices to say that the language standard then defines the semantics of aliasing and address arithmetic as one would expect. These definitions are enough to reason about memory in sequential programs.

Unfortunately, programs with multiple threads make things much more complicated because there needs to be a notion of synchronizing memory accesses between these threads to allow communication. C++ opted to choose a highly fine-grained model based on so-called *atomics*. Atomics are nothing more than objects with special memory access operations. It is forbidden to access non-atomics by multiple threads at once.

All atomic objects have a load, store, exchange, and compare-and-exchange operation. Load and store are self-explanatory. Exchange atomically returns the current value of the object and replaces it with a new value. Compare-and-exchange atomically performs an exchange if the comparison with a given reference object succeeds. Numerical atomic objects additionally have atomic fetch-and-add, fetch-and-sub, fetch-and-and, fetch-and-or, fetch-and-xor, increment, and decrement operations. It is allowed to perform all these operations at the same time from any amount of threads.

Having synchronized operations on single atomic variables is not enough to efficiently write multi-threaded programs. Therefore, there are also rules that govern how memory

[2] „Object“ in this context is not to be confused with the notion of object from object-oriented programming. Here it just means a „thing“ in memory.

writes, including to non-atomic objects, become visible for other threads relatively seen to atomic memory accesses. This act of „memory writes becoming visible“ is called *happened-before relationship*, i.e., if a thread can see something that another thread has done, the actions of the second thread happened-before the lookup of the first thread. The rules of how happened-before relations are formed form the core of the C++ memory model. The way to control these relations is that each of the aforementioned atomic operations takes a *memory ordering* as argument that governs which memory accesses are going to be visible, i.e., defines the happened-before relations. The for this thesis relevant memory orderings are:

relaxed No memory accesses are made visible.

acquire All writes by other threads before accessing the same atomic variable using a release ordering are made visible.

release All writes before the release ordered access are going to be visible to other threads that access the same atomic variable using an acquire ordering.

acq rel Combined acquire and release.

There are more memory orderings, e.g., a sequential consistent one, but they are not going to be used in this thesis. A nice overview of all memory orderings can be found in the cppreference wiki [19].

The most important concept in this context is the so-called *acquire-release ordering*: Whenever a thread accesses an atomic variable using an acquire memory ordering, all memory writes by other threads that happened before an access to the same atomic variable using the release ordering become visible to the thread that issued the access with the acquire memory ordering.

Additionally, there is the concept of *fences*. A fence is an operation that takes a memory ordering just like atomics, and therefore has the same synchronization effects as atomics. The only difference to atomics is that there is no underlying object or memory access. Fences exist solely to establish order.

Note that using volatile objects for cross-thread synchronization is almost always wrong and should be avoided at all cost. Unfortunately, using them on x86/AMD64 systems often seems to work but this behavior is not guaranteed by the C++ memory model by any means.

5.1.2 Inbox/Relaxed Vector

An *inbox* is a data structure with the following operations:

push back $\mathcal{O}(1)$: Inserts an item at the end of the inbox.

for each $\mathcal{O}(n)$: Iterates over all items in the inbox.

clear $\mathcal{O}(1)$: Removes all items from the inbox.

size $\mathcal{O}(1)$: Returns the number of items in the inbox.

Furthermore, the following constraints have to be considered when using an inbox:

- Any amount of threads is allowed to execute push back in parallel but while doing so no other operation of the inbox may be executed.
- Only a designated thread, the so-called owner of the inbox, may access for each, clear, and size during a period of quiescence with respect to push back.
- This data structure does not guarantee any memory synchronization at all, i.e., before the owner can read the items in the inbox, an appropriate synchronizing operation has to be executed manually.

The inbox is implemented by using a self-implemented relaxed vector.[3] The vector consists of an array of memory blocks *data*, i.e., an array of atomic pointers, and an atomic index *at* which designates the next free slot in the memory blocks. The chunks themselves are not atomic, only the pointers to them.

At initialization $at \coloneqq 0$ and $data \coloneqq [\text{null}, ..., \text{null}]$. The maximum number of blocks, and therefore the maximum number of elements in the relaxed vector, is fixed at initialization time.

The push back operation can be seen in Algorithm 5.1. The idea is that *at* is atomically incremented by one to obtain the slot for the element to be inserted. Then the function checks if the block for the given slot is already allocated. If so, the element is just written there. If not, the thread allocates the whole memory block and tries to compare-and-swap it into the array of block-pointers. If this compare-and-swap does not succeed, another thread allocated the block in question in the meantime. This means that the current thread has to free the just allocated memory and just uses the in-the-meantime allocated block for insertion. Notice how there are only two relaxed atomic memory accesses in the non-allocating case.

The for each function just iterates over all chunks and the elements in them up to *at*. Since for each may only be called by the owner during a period where no other thread accesses the relaxed vector, there is not much to take care of: A simple iteration over the storage suffices. Similarly simple, size just returns *at*, and clear sets $at \coloneqq 0$.

A simple alternative to this data structure would be a standard vector protected by a mutex. There are two major issues with this approach though. First, a mutex slows

[3] See `relaxed_vector.cpp` for the implementation.

Require: an element to insert e and the relaxed vector V
Ensure: e is inserted into V
  $index \coloneqq V.at$.fetch-and-add(1, relaxed)
  $chunk \coloneqq index$ div $chunk\_size$
  $position \coloneqq index$ mod $chunk\_size$
  $ptr \coloneqq V.data[chunk]$.load(relaxed)
  if $ptr =$ null **then**
    $new\_chunk \coloneqq$ allocate memory for a chunk
    if compare-and-exchange(relaxed) ptr in $V.data[chunk]$ with new_chunk succeeds **then**
      $ptr \coloneqq new\_chunk$
    else
      free new_chunk
      {the compare-and-exchange populated ptr with the correct pointer}
    end if
  end if
  $ptr[position] \coloneqq e$

Algorithm 5.1: The push back Operation of a Relaxed Vector

down the push back operation considerably if multiple threads have to push to the same vector. Since this data structure is used for storing remote relaxations, a high throughput is important. Second, a mutex incurs a memory fence, which additionally increases the cost although the memory fence is not needed.

In the given implementation the only point of contention is $V.at$.fetch-and-add(1, relaxed), since the fetch-and-add operation could be issued by multiple threads at the same time. An approach to remedy this would be to fetch-and-add with a higher number than 1, e.g., 16. This would mean that a thread reserves 16 elements at once and only has to execute a new fetch-and-add once the 16 elements are used. But this approach incurs additional implementation complexity: Since a thread might not use all 16 elements there are holes in the array. This means that the for each operation has to know which elements are used and which are not used, for example by utilizing an additional flag per element. Furthermore, each thread needs a local counter to remember its index in the inbox. Such thread-local counters are not needed in the given implementation.

5.1.3 Collective Operations

The C++ threads are manually grouped together in thread groups. Each thread group has access to some shared data, as described in this section, which is used to implement *collective operations*.

Collective operations are functions that can be called in a thread group with the restriction that all threads participating in a thread group have to issue the same sequence of collective

Require: shared variables in the thread group: *generation*, *waiting*
Ensure: all threads continue execution once they all entered the barrier
  fence(release)
  current_generation := *generation*.load(relaxed)
  current_waiting := *waiting*.fetch-and-add(1, relaxed) + 1
  if *current_waiting* = *thread_count* **then**
    generation.store(!*current_generation*, relaxed)
    waiting.fetch-and-sub(*thread_count*, relaxed)
  else
    while *generation* = *current_generation* **do**
      yield {busy wait}
    end while
  end if
  fence(acquire)

Algorithm 5.2: Barrier

operations. There is a hand full of collective operations implemented.[4]

Barrier

A barrier stops execution of all threads that invoked the barrier, until all threads in the thread group reach the barrier. At this point in time all threads in the thread group are allowed to continue execution. Optionally, the barrier can also emit memory fences to make memory writes from before the barrier visible to all threads after the barrier.

The implementation used is inspired by Boost's [9] implementation of barriers. Unfortunately, Boost's implementation uses expensive primitives, and can therefore not be used for the purposes of this thesis. The custom implementation for this thesis uses only relaxed accesses to atomic variables.

The implementation can be seen in Algorithm 5.2. It assumes that the thread group has access to two shared variables: *generation*, an atomic boolean value, and *waiting*, an atomic integer. The basic idea of this barrier is that *waiting* is counted up until the thread count is reached. Once this is the case, a single thread, namely the last thread that incremented *waiting*, flips the value of *generation*. This is the signal for all other threads to continue execution.

The barrier uses C++'s `std::this_thread::yield` to inform the operating system about the busy-waiting loop, which might allow the operating system scheduler to deschedule the current thread to allow other threads to make progress. Depending on the operating system, the number of threads, and the CPU topology this might or might not have an effect on the performance.

[4] See `collective_functions.hpp` for the implementation.

Require: an operation $\otimes$, a starting value s, and a contribution for the current thread c;
shared variables in the thread group: *into*
Ensure: *into* contains the result of the reduce operation
into.store(s, relaxed)
barrier
current_value := *into*.load(relaxed)
wanted_value := $c \otimes$ *current_value*
while *wanted_value* $\neq$ *current_value* $\wedge$
into.compare-and-exchange(*current_value*, *wanted_value*, relaxed)
does not succeed **do**
{the compare-and-exchange refreshed the value in *current_value*}
wanted_value := $c \otimes$ *current_value*
end while
barrier

Algorithm 5.3: Reduce

It would have been very interesting to implement a barrier by using the special x86 instructions `MONITOR` and `MWAIT`. Unfortunately, these instructions are only allowed to be executed in privileged mode, i.e., in kernel space, and are therefore of no use for the benchmarking tool.

Reduce

Given t participants, a starting value s, a value for each thread v_i, and a commutative and associative operation $\otimes$, then a reduce operation calculates

$$v = s \otimes \bigotimes_{i=1}^{t} v_i$$

such that this result is available to each thread participating in the reduce operation. A common example would be that $\otimes = \min$ to find the minimum of a value across all threads.

Algorithm 5.3 shows the used implementation of this algorithm. The algorithm assumes access to a shared atomic variable *into* that will be used to store the result. The basic idea of this implementation is that before the first barrier it is ensured that the starting value is in *into*. After that each thread tries to add its own contribution using a compare-and-exchange operation until it succeeds. Once all threads have added their contribution, they may continue execution.

Performance measurements of the whole algorithms have shown that the performance impact of the reduce operation is completely negligible compared to the whole runtime. Therefore, although more efficient reduce algorithms exist, this simple and straightforward implementation was chosen.

Require: a function *fn* to be executed once; shared variables in the thread group: *do*
Ensure: once a thread returns from this function, *fn* has been executed by some thread
  do.store(1, relaxed)
  barrier
  if *do*.exchange(0, relaxed) **then**
    call *fn*
  end if
  barrier

Algorithm 5.4: Single

Single

The single operation is a collective operation that ensures that the given function is only executed by exactly one thread, but does not allow other threads to continue until the given function has been executed. This operation comes in handy for initialization work.

The implementation, seen in Algorithm 5.4, is quite simple: A shared atomic boolean *do* is used to decide if the current thread is supposed to execute the function or not. Each thread exchanges the value with false. The thread that manages to exchange the initially stored true value is the one that has to execute the passed function.

5.1.4 Non-Collective Operations

Non-collective operations can be called by any thread in the thread group at any point in time, without participation of the other threads. Of course, it is allowed that multiple threads call these operations at the same time.

Memory Allocations

There are three allocators used in the benchmarking tool. First and foremost the system allocator, i.e., the default behavior of C++'s new-operation. This allocator is avoided and only used in performance oblivious initialization code, and to implement the more refined allocators. Nevertheless, to optimize this allocator Intel Threading Building Blocks' [38] allocator was used instead of the default system allocator on platforms that are supported by it.

For memory that has to be shared between multiple threads, hwloc's [58] memory allocator is used. This is a special NUMA-aware allocator that can be used to tune allocations in such a way that they have the most efficient layout with respect to the memory latencies by the various threads. All memory allocations in performance sensitive parts of the applications were carefully crafted in such a way that they use all advantages that hwloc can offer.

Lastly, there are a lot of data structures that do not need to be shared between threads. For the allocations for these data structures a special thread-local memory allocator

Require: a shared atomic number a and an argument b
Ensure: $a = \min(a, b)$
  $a' = a$.load(relaxed)
  while $b < a' \wedge$
        a.compare-and-exchange(a', b, relaxed) does not succeed **do**
    {the compare-and-exchange operation refreshed the value in a'}
  end while

Algorithm 5.5: Atomic Min

was implemented. This memory allocator uses the system memory allocator to, rarely, allocate large chunks of memory. These chunks will then be managed using buddy memory allocation [44, Chapter 2.5]. Since this allocator avoids the system allocator most of the times, it does not need any system wide lock and has a very low overhead. For allocation heavy data structures, for example for Fibonacci heaps, this allocator even improves the performance in single-threaded applications.

Atomic Min

The atomic min operation is a simple operation to atomically store the minimum of an atomic variable a and a non-atomic argument b in a. This can be achieved by a simple compare-and-exchange operation, as can be seen in Algorithm 5.5. As with the reduce operation, the performance of this operation is not relevant for the overall performance of the algorithms, and therefore this simple implementation was chosen.

5.2 Decision Procedures

This section aims to give an overview of how each of the criteria can be decided. This will be done by providing an algorithm with placeholders that will be filled in with the decision procedure by each criteria. Algorithm 5.6 shows this algorithm scheme. The placeholders are marked with circled numbers.

In all implementations of the decision procedures Fibonacci heaps [32] are used as priority queue whenever one is needed. Pairing heaps [31] were tried as well, unfortunately it turned out that they perform worse than Fibonacci heaps. Both heap implementations are provided by the Boost library [9]. Fibonacci heaps are an allocation intensive data structure but since the tool implements a custom thread-local allocator, as described in Section 5.1.4, this does not matter from a performance perspective. In principle, there are many other priority queue implementations available as well but most of them lack support for the decrease-key operation, an operation that allows to update the priority of an item already in the queue, which is required for efficient implementations of these decision procedures. This is also an intuitive explanation why many heap implementations do not reach the performance of Fibonacci heaps for this use-case: Since the decrease-key operation will be invoked many times, about $\mathcal{O}(|E|)$ times, it is important that this

Require: a directed graph $G = \langle V, E \rangle$ and a starting node s
Ensure: all reachable nodes are correctly settled
  ① {initialize the priority queues}
  add s to the priority queues
  while priority queues are not empty {iterations of this loop are called phases} **do**
    $todo := \emptyset$
    while ② {condition to find nodes to be settled} **do**
      insert node to be settled into *todo*
      remove the node from all priority queues
    end while
    for $n \in todo$ **do**
      settle n
      update priority queue entries for the destinations of outgoing(n) if required
    end for
    ③ {additional bookkeeping if required}
  end while

Algorithm 5.6: Algorithm Scheme for Deciding the Criteria

particular operation is very cheap. The performance of the other operations, which are invoked about $\mathcal{O}(|V|)$ times, does not matter too much. Furthermore, the size of the priority queue is bounded by $\mathcal{O}(|V|)$.

All these decision procedures can be implemented by not using any priority queues at all. It is always possible to just store all nodes in F in a simple array and iterate over all of them, naïvely checking the criteria as they are defined in Chapter 3. This has been implemented in the benchmark as well, and as will be seen in Section 5.7, is faster than using priority queues.

A description of the decision procedure for each criteria follows. Note that the circled numbers, ①, ②, and ③, reference to the placeholders in Algorithm 5.6.

5.2.1 Dijkstra's Algorithm

This summarizes the usual implementation of Dijkstra's algorithm.

① To implement Dijkstra's algorithm, a single priority queue ordered by the tentative distance of each node suffices. The smallest tentative distance has to be at the tip of the priority queue.

② The only node to be settled in each phase is the smallest node in the priority queue.

③ No additional bookkeeping required.

5.2.2 Crauser's Out Criteria

To decide Crauser's Out criteria, it is assumed that each node knows its cheapest outgoing edge. The performance impact of this requirement will be discussed at a later point in time.

① To decide Crauser's Out criteria, two priority queues are required. The first one is the same as for Dijkstra's algorithm, i.e., ordered by the tentative distance of each node, and will be called T. The second one is a priority queue O ordered by the tentative distance plus the cheapest outgoing edge cost for each node.

② To find the nodes to be settled, one evaluates $T.\text{head} \leq O.\text{head}$. As long as this evaluates to true, the head of T can be added to the set of nodes to be settled and removed from all priority queues.

③ No additional bookkeeping is required for the static variation. The dynamic variant needs to update the cheapest outgoing edge potentially for each node. To achieve this, each non-settled node is considered. If the destination of the currently cheapest edge is settled, a new cheapest edge is looked for. This can be achieved in three different ways:

One, a simple linear scan through the outgoing edges of each node can be performed to find the new cheapest outgoing edge. This adds a per-phase overhead of $\mathcal{O}(|\text{outgoing}(n)|)$ for each such node. In the worst case this sums up to $\mathcal{O}(|E|)$. In practice this method turns out to be the most efficient and was implemented in the benchmark.

Two, in a preprocessing phase the array of outgoing edges for each node is heapified. This amounts to an preprocessing overhead of $\mathcal{O}(|E|)$. Now, instead of a linear scan, one can use find-min to find the new cheapest edge. Note however, that multiple calls to find-min may be required if multiple neighbors of the node in question were settled. A single find-min adds a per-phase per-node overhead of $\mathcal{O}(\log|\text{outgoing}(n)|)$. In the worst case find-min has to be called outgoing(n) times. In total this sums up to about $\mathcal{O}(|V|\log|E|)$, not assuming the worst-case.

Three, in a preprocessing phase the array of outgoing edges for each node is sorted. This amounts to an preprocessing overhead of $\mathcal{O}(|E|\log|E|)$. Whenever the destination of the head of this sorted list is settled, it is removed. This is possible in $\mathcal{O}(1)$ time. Note however, that similarly to the second approach this step might have to be repeated. This way, the head of the sorted list is always the cheapest outgoing edge. In total this sums up, again not assuming the worst case, to $\mathcal{O}(|V|)$ time overhead per phase.

The described method of deciding the static variation of this criteria was already proposed as it is here by Crauser et al.

5.2.3 Crauser's In Criteria

This criteria assumes that each node knows its cheapest incoming edge. The dynamic variation even assumes that each node knows all its incoming edges.

① To decide this criteria, similarly to the Out criteria, two priority queues are needed. T is a priority queue ordered by the tentative distance of each node, exactly as it is used in Dijkstra's algorithm. I is a priority queue ordered by the tentative distance minus the cheapest incoming edge.

② To find nodes to be settled one evaluates $I.\text{head} \leq T.\text{head}$. As long as this evaluates to true, the head of I can be added to the set of nodes to be settled and removed from all priority queues.

③ Completely analogous to Crauser's Out criteria, except that one has to work with the incoming edges instead of the outgoing edges, i.e., the initialization cost and per-phase overhead is $\mathcal{O}(1)$ and $\mathcal{O}(|E|)$ for not pre-sorting the incoming edges, $\mathcal{O}(|E|)$ and $\mathcal{O}(|V|\log|E|)$ for pre-heapifing the incoming edges, and $\mathcal{O}(|E|\log|E|)$ and $\mathcal{O}(|V|)$ for pre-sorting the incoming edges.

Again, for the static variant, Crauser et al. did already propose this method of deciding the criteria.

5.2.4 Heuristic

This criteria assumes that each node additionally knows its incoming edges.

Each node v has to maintain a list of its predecessors sorted by $h(p)+\text{cost}(p,v)$. Whenever a node is settled, each successor s has to be considered: While the minimum of the sorted list of predecessors of s settled, it has to be removed from this list. Therefore, after this step, the list of predecessors is either empty, or is lead by a predecessor that is not settled yet. In this step it is already possible to check if s is eligible for settling in the next phase, by comparing its new tentative distance with the lead of the aforementioned list. Note, that this approach does not fit into the algorithm scheme like the other criteria do.

This criteria is only implemented in the simulation, not in the benchmark. This is because the benchmark does not generate graphs for which a sensible heuristic exists. Furthermore, according to the results of the simulation, the heuristic is a very weak criteria, but it is very expensive to execute, rendering its minuscule improvements in the number of phases completely useless.

5.2.5 Träff's Bridge Criteria

This criteria assumes that each node knows its incoming edges and the minimum cost incoming edge. Furthermore, each node needs to know the minimum sum of two „backward“ steps. In other words, each node needs to know the minimum of the cheapest incoming edge of a predecessor plus the edge cost to reach this predecessor.

① Again, two priority queues are needed. The first priority queue, T, is ordered by the tentative distance of each node. The second priority queue, II, is ordered by the tentative distance minus the minimum two-steps-backwards cost for each node.

Additionally, a counter has to be maintained for each node. This counter is going to count the number of predecessors in F for each node. Initially, it is set to zero for all nodes.

② To find candidates to settle $II.\text{head} \leq T.\text{head}$ is evaluated. As long as this evaluates to true, the head of II can be removed from all queues and added to the set of candidates to be settled.

In a second step, all candidates have to be checked if they have an incoming edge originating from any node in F. If the aforementioned counter for any given node is zero, the given node does not have such an edge and can be settled. If the counter is not zero, the node has to be re-added into the priority queues and cannot be settled.

③ Whenever a node moves from U to F, the counter for all reachable nodes by an outgoing edge of this node has to be increased by one. Consequently, whenever a node moves from F to S, the counter for all these nodes has to be decreased by one. Both steps incur an overhead of $\mathcal{O}(|\text{outgoing}(n)|)$ for each node during the whole execution of the algorithm.

To implement Träff's Bridge criteria dynamically, one would have to update the aforementioned two-step-backwards minimum after execution of each phase, similarly how the dynamic Crauser's criteria have to update the minimum incoming/outgoing edge.

Unfortunately, doing so would be prohibitively expensive for this criteria. A possible two-step implementation is: First, use the same method as for dynamic Crauser's In criteria to find the cheapest incoming edge for each non-settled node. Second, use the method for dynamic Crauser's In criteria again, except that this time instead of looking at the cheapest incoming edge, it is now possible to look at the cheapest two-step-backwards path since, due to the first step, the cheapest incoming edge for each node is already known. In total this amounts to double the overhead compared to dynamic Crauser's In criteria, in addition to the overhead needed for deciding Träff's Bridge criteria in the static variation.

Unfortunately, since the simulation suggests that this criteria only improves the number of phases by a small margin, and since the implementation is quite complicated, it is not to be expected that this criteria improves the performance in the benchmarks.

5.3 Graph Representation

A (sub-)graph is represented by two arrays. The first array is an array of all edges, arranged in such a way that all edges that originate at the same node are next to each

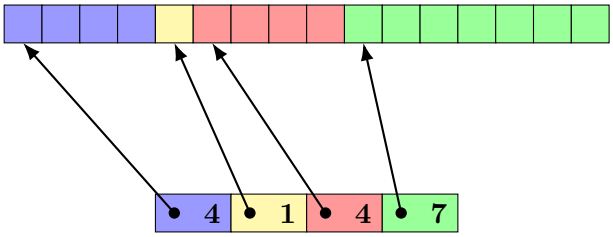

Figure 5.1: The layout of graphs in memory in the benchmarking tool. The lower array represents nodes, four in this example, while the upper array contains all edges grouped by node.

other. Each edge consists of the three fields: source, destination and cost. The second array represents the nodes. Each node is nothing more than a pointer and a size. The size is the number of edges that belong to the given node, and the pointer points to the first edge of the node in question in the first array. Since the edges are ordered by their origin, this suffices to find all edges of any given node. Nodes are identified by their index in the second array. This can be seen in Figure 5.1.

Notice how iterating over the edges of a single node is cache friendly. Furthermore, iterating over edges of subsequent, or even all, nodes is cache friendly as well.

This graph representation can easily be distributed among threads by splitting both arrays into multiple chunks and moving these chunks to their respective threads. This is also the way this is implemented in the benchmarking tool.

5.3.1 Graph Generation

Generating uniformly random graphs in parallel is simple. Each thread is assigned a certain set of graph nodes it is responsible for. For each of these nodes, the thread draws a random number from $\mathcal{B}(|V|, p)$, with p being the edge chance. This number represents the number of outgoing edges for the node in question. Lastly, the edges are generated with a uniformly random cost and a random destination. Note, that this process allows self-edges and multiple-edges, although their chance of appearing is quite low.

The generation of Kronecker graphs is more expensive, nevertheless not complicated. A single thread samples the total number of edges from $\mathcal{P}((\sum A)^k)$, with A being the initiator matrix and k being the Kronecker power. Each thread now generates $1/t$-th of the sampled number of edges using the recursive descend as described in Section 4.1.1. Since each thread is responsible for a certain set of nodes, the generated edges have to be exchanged in such a way that all edges are moved to the thread that is responsible for the edge's origin. This is implemented by placing the edges into an inbox for each thread. Again, this process allows for self-edges and multiple-edges. The first phase of this algorithm is embarrassingly parallel, this approach is only made expensive by the edge exchange required at the end.

If one wants to generate very large graphs, e.g., graphs with more than 10^9 nodes, one would have to use more efficient approaches. For example, Sanders and Schulz [62] present an embarrassingly parallel, highly efficient algorithm to generate graphs with properties similar to Kronecker graphs, albeit not Kronecker graphs.

5.3.2 Additional Node Information

As mentioned during describing the decision procedures of each criteria, some criteria expect to have access to some additional information. For example, Crauser's static In criteria expects to know the cheapest incoming edge for each node. This subsection aims to give an overview of the approaches to and the cost of obtaining such information.

Cheapest Outgoing Edge Cost

Obtaining this information is trivial. A simple linear scan through the list of outgoing edges for each node suffices. The overhead therefore is $\mathcal{O}(|E|)$, respective $\mathcal{O}(|E|/t)$ under the assumption that each of the t threads is responsible for $|E|/t$ edges.

Cheapest Incoming Edge Cost

Obtaining the cheapest incoming edge cost is a little bit more involved since the incoming edges are not known to the single nodes. Algorithm 5.7 shows the chosen approach. The basic idea is to have a shared array of atomic floats called *cheapest*. Each entry corresponds to a single node. After initializing each entry of this array to ∞, each thread iterates over all edges. For each edge an atomic minimum at the edge's destination node's entry in the array with the cost of the edge is executed. The time complexity of this approach is $\mathcal{O}(|E|/t)$, assuming that the atomic-min operation, as described in Section 5.1.4, can be completed in $\mathcal{O}(1)$ time.

This assumption means that the compare-and-exchange operation in atomic-min fails only $\mathcal{O}(1)$ times. This assumption is reasonable for this algorithm because iterating over all edges and accessing the array entry corresponding to their *destination* leads to a completely random access pattern into the array. Therefore, it is highly unlikely that two threads access the same element of the array at the same time, i.e., it is highly unlikely that the compare-and-exchange in atomic-min fails.

Incoming Edges

If a thread needs to actually know all incoming edges of the nodes it is responsible for, the information exchange becomes more expensive. This is achieved in two phases. In the first phase, each thread counts the number of edges it has to send to all other threads. These counts are then atomically summed for each thread. In the next step, each thread therefore knows how many edges it will receive, and can therefore allocate an array of appropriate size. Finally, all threads place a copy of the edges in the respective per-thread array. This concludes the information transfer. As post processing step, each

Require: a directed graph $G = \langle V, E \rangle$
Ensure: each thread knows the cheapest incoming edge cost for its nodes
  for all nodes v this thread is responsible for **do**
    cheapest$[v]$.store$(\infty, \text{relaxed})$
  end for
  barrier
  for all nodes v this thread is responsible for **do**
    for $\langle v, v' \rangle \in \text{outgoing}(v)$ **do**
      atomic-min(*cheapest*$[v']$, cost(v, v'))
    end for
  end for
  barrier
  for all nodes v this thread is responsible for **do**
    store *cheapest*$[v]$.load(relaxed) locally
  end for

Algorithm 5.7: Finding the Cheapest Incoming Edge Cost

thread has to sort the incoming edges, so that they can be accessed efficiently for each node. The time cost of this algorithm is $\mathcal{O}(|E|/t \cdot \log(|E|/t))$ when using a traditional sorting algorithm. By using bucket-sort [18] it is possible to reduce this time to $\mathcal{O}(|E|/t)$, although at the cost that the memory consumption is doubled since bucket-sort is an out-of-place sorting algorithm.

Cheapest Two-Steps-Backwards Edge Cost

To find the cheapest size-two incoming path for each node, first all incoming edges are collected, as described in the previous section, and the minimum incoming edge cost for all nodes is calculated, again as described previously.

Using this knowledge, each thread can iterate over the nodes it is responsible for, and therefore iterate over their incoming edges, and add the cheapest incoming edge of each edge's source by accessing the shared array *cheapest*. This way, the minima of all two-steps-backwards paths is found.

5.4 Parallelization

There were three approaches considered of how to distribute and parallelize the algorithm scheme.

- Each thread is responsible for a certain subset of all nodes, and therefore the edges belonging to these nodes. This is the chosen approach. This approach does not need any shared data-structures, besides an inbox for each thread to buffer remote relaxations. As described in Section 5.1.2 such an inbox can be

implemented efficiently. Since each thread only knows $\mathcal{O}(|V|/t)$ nodes, the size of all data-structures is bounded by $\mathcal{O}(|V|/t)$, leading to an improvement of the run-times of all data-structure operations.

- Each thread is responsible for a certain subgraph and the algorithm scheme is implemented by using a shared data-structures. This approach is considered infeasible because such shared data-structures, e.g., a parallel priority queue with support for decrease-key, are extremely hard to implement in practice, and even harder to implement efficiently. Most of these data-structures are of theoretical interest only.

- Each thread is responsible for a certain subset of edges, no matter their origins or destinations, and the algorithm scheme is implemented by *not* using shared data-structures. This approach has the major problem that all threads need to keep track of the priority queues for the criteria themselves. Since each thread needs to be informed about all nodes each of these priority queues is of size $\mathcal{O}(|V|)$, which means that the performance of all priority queue operations is as-if there was only a single thread. In other words: The work of all priority queue operations is multiplied by a factor $\mathcal{O}(t)$.

The memory containing the subgraph data structure is allocated in such a way that it is most efficient for the thread, as explained in Section 5.1.4. Algorithm 5.8 shows the parallel algorithm scheme to decide each criteria. Notice, how the parallel algorithm scheme utilizes the same placeholders as the sequential algorithm scheme. The placeholders are filled with the same sub-algorithms as described in Section 5.2. Furthermore, this algorithm assumes that there is a shared array *inboxes* which contains an inbox for each thread. There are no additional shared data structures, i.e., each thread maintains its own set of priority queues. The basic idea of the parallelization is that remote relaxations are stored in the inbox of the receiving thread and executed after the local relaxations, while the bounds required to decide the criteria are obtained by reduce operations.

Träff's criteria needs an additional barrier because ② has to access a shared array that counts the number of predecessors for each node, see Section 5.2.5 for details. Since during local relaxations, this array can already be changed, there has to be an additional barrier separating the fill-phase of *todo* and the settling-phase of *todo*. The other criteria do not have such a dependency.

Additionally, depending on the chosen criteria, ① has to be extended by the approaches shown in the previous section, so that each thread has all the data required to efficiently decide ②.

There is one optimization left that was not explained so far. Step ③ for the dynamic criteria requires to know if the source/destination of a given edge has already been settled. This check would have to access the thread-local data of the thread that is responsible for each given source/destination. This would entail to access the thread-local memory

Require: a directed subgraph graph $G = \langle V, E \rangle$ and a starting node s
Ensure: all reachable nodes are correctly settled
① {initialize the local priority queues & exchange required data}
add s to the priority queues if this thread is responsible for s
loop
use a reduce operation to find the thresholds of the criteria
{this works by reducing over the tips of the priority queues}
break if the thresholds are ∞
$todo := \emptyset$
while ② {condition to find nodes to be settled using the reduction's result} **do**
insert node to be settled into *todo*
remove the node from all priority queues
end while
barrier if Träff's criteria is used
for $n \in todo$ **do**
for $\langle n, n' \rangle \in \text{outgoing}(n)$ **do**
if this thread is responsible for n' **then**
relax $\langle n, n' \rangle$
else
store $\langle n, n', \text{tent}(n) + \text{cost}(n, n') \rangle$ in the $inboxes[t]$ of the responsible thread t
end if
end for
end for
barrier
execute all relaxations in the thread's inbox
empty the inbox
barrier
③ {additional bookkeeping for the dynamic criteria}
barrier
end loop

Algorithm 5.8: Parallel Algorithm Scheme for Deciding the Criteria

of another thread. Additionally such accesses to foreign memory might require additional synchronization measures or atomic primitives.

To avoid this overhead an additional array of atomic doubles, called *seen_distances*, is used. Each node has a corresponding entry in this array. Initially, all entries are set to ∞. Whenever a thread settles are node, it atomically sets the corresponding entry to $-\infty$. Whenever a thread *places* a relaxation into an inbox, or performs a local relaxation, it updates the value using an atomic store to the potential new value. Notice how this approach avoids any synchronization at the cost that a higher value might override a lower value due to race-conditions.

Now, to quickly decide in ③ if a given node is settled, it suffices to check that *seen_distances* is set to $-\infty$. Due to the aforementioned race-conditions this check is not complete, i.e., it might flag a node as non-settled even though it is settled, but always sound, i.e., it will never flag a non-settled node as settled.

Furthermore, this array can be used to reduce the memory traffic to the inboxes. Instead of placing a relaxation into the inbox, it is first checked if the new tentative distance implied by the relaxation is less than the current distance in *seen_distances* for the destination node. If this is not the case, the relaxation is just thrown away. Again notice, that due to the race-conditions this might allow relaxations that do not improve the tentative distance to be placed into an inbox, but a relaxation that can improve the tentative distance will never be rejected.

5.5 Δ-Stepping

Δ-stepping [53] is currently considered the state-of-the art algorithm to parallelize the single-source shortest path problem on shared as well as on distributed memory systems. The algorithm can be seen in Algorithm 5.9.

The idea of the algorithm is the following. Similar to the Bellman-Ford algorithm, the algorithm maintains a tentative distance for each node. Furthermore, the algorithm maintains, conceptually an infinite amount, of buckets. A range is associated with each bucket. The size of this range is controlled by a parameter Δ, hence the name of this algorithm. The range for the first bucket is 0 to Δ. The range of the second bucket is Δ to 2Δ, and so on. In practice one only needs $\max_{v,v' \in V} \text{cost}(v, v')/\Delta$ buckets.

At the start of the algorithm the tentative distances are initialized to ∞, except for the starting node where the tentative distance is being initialized to 0. Additionally, the starting node is placed into the first bucket. Whenever the tentative distance of any node is updated, it is removed from the bucket it currently belongs to and inserted into the bucket corresponding to its new tentative distance.

The algorithm now performs the following steps repeatedly until all buckets are empty: All nodes in the first non-empty bucket are removed from the bucket and are relaxed. Relaxed means that the node's tentative distance plus outgoing edge cost are compared

Require: the parameter Δ, a graph $G = \langle V, E \rangle$, and a starting node s
Ensure: $tent[v]$ contains the distance for each v
  for all $v \in V$ **do**
    $tent[v] \coloneqq \infty$
  end for
  $buckets[0] \coloneqq \{s\}$
  $tent[v] \coloneqq 0$
  while $buckets \neq \emptyset$ **do**
    $b \coloneqq$ first non-empty $buckets$ index
    $heavytodo \coloneqq \emptyset$
    while $buckets[b] \neq \emptyset$ **do**
      $todo \coloneqq buckets[b]$
      $heavytodo \coloneqq heavytodo \cup todo$
      $buckets[b] \coloneqq \emptyset$
      relax-all($todo$, $<$)
    end while
    relax-all($heavytodo$, $\geq$)
  end while

  function relax-all($nodes$, $\circ$)
    for all $v \in nodes$ **do**
      for all $\langle v, v' \rangle \in \text{outgoing}(v) \wedge \text{cost}(v, v') \circ \Delta$ **do**
        if $tent[v] + \text{cost}(v, v') < tent[v']$ **then**
          remove v' from it's bucket
          $tent[v'] \coloneqq tent[v] + \text{cost}(v, v')$
          add v' into it bucket $tent[v']/\Delta$
        end if
      end for
    end for
  end function

Algorithm 5.9: Δ-Stepping [53]

with the tentative distance of the neighbor. If this comparison leads to a better distance to reach the neighbor, the neighbor's tentative distance is updated. This also means that the bucket the neighboring node is in might change. This also means that a node that was just removed from a bucket is being re-inserted. This process repeats until all buckets are empty.

To reduce the number of unnecessary relaxations there exists a simple optimization. One splits the set of edges into two subsets: Heavy edges are those with cost greater or equal to Δ, and light edges are those with cost less than Δ. Now, one empties the first non-empty bucket B but only relaxes light edges. If B is non-empty after this step, this is repeated until B is eventually empty. Only after that, the heavy edges, which were ignored until now, are relaxed. To further optimize this, one can pre-partition the outgoing edges of each node into heavy and light edges. If done, one saves time by only iterating over the relevant set of edges.

In a parallel setting, as with the criteria, it is assumed that each thread is responsible for a subset of all nodes. Furthermore, it is assumed that each thread has an inbox and maintains local buckets. The parallel implementation of Δ-stepping then only needs minor extensions with respect to the here presented sequential implementation. A summary of the parallel Δ-stepping implementation can be seen in Algorithm 5.10.

First, an edge can only be relaxed locally if the edge's destination and edge's source are handled by the same thread. If this is not the case, the relaxation is buffered in the inbox of the respective thread. After iterating over all edges in a bucket, the inboxes are used to apply the relaxations from foreign threads.

Second, to avoid race conditions two pairs of barriers have to be introduced. Each pair, as seen in the pseudocode, protects the processing of the thread's inbox. These barriers are required because during relaxations, remote relaxations are placed into the inbox of the respective destination thread. Therefore, all threads have to finish placing relaxations, before they may access their inboxes. The barrier after processing the inboxes exists for the same reason, i.e., no thread may start placing relaxations into the inboxes until all threads are finished processing their inbox.

Third, to decide if the algorithm is finished it suffices to know that all buckets are empty. This can be achieved if each thread checks if its local buckets are empty, resulting in a boolean. These booleans are then reduced using the operator $\wedge$. If the result is true, there are not any nodes in the buckets globally, and therefore the algorithm is finished. This is the condition of the outer while-loop in the pseudocode.

Fourth, to decide if a bucket is emptied, the same method as just described is used, with the change that only the number of nodes in the current bucket is checked, instead of the number of nodes in all buckets. This is the condition of the inner while-loop in the pseudocode.

Require: the parameter Δ, a subgraph $G = \langle V, E \rangle$, and a starting node s
Ensure: $tent[v]$ contains the distance for each v the thread is responsible for
  for all $v \in V$ **do**
    $tent[v] \coloneqq \infty$
  end for
  $buckets[0] \coloneqq \{s\}$ if the thread is responsible for s
  $tent[v] \coloneqq 0$
  $b \coloneqq 0$
  while globally $buckets \neq \emptyset$ {reduce local bucket sizes to decide this} **do**
    $b \coloneqq b + 1$
    $heavytodo \coloneqq \emptyset$
    while globally $buckets[b] \neq \emptyset$ {reduce size of $buckets[b]$ to decide this} **do**
      $todo \coloneqq buckets[b]$
      $heavytodo \coloneqq heavytodo \cup todo$
      $buckets[b] \coloneqq \emptyset$
      relax-all($todo$, $<$)
      barrier
      execute relaxations in this thread's *inbox*
      empty the inbox
      barrier
    end while
    relax-all($heavytodo$, $\geq$)
    barrier
    execute relaxations in this thread's *inbox*
    empty the inbox
    barrier
  end while

Algorithm 5.10: Parallel Δ-Stepping

5.6 Complexity

This section aims to give an overview of the total complexity of the algorithms when combining all the approaches described and analyzed so far. The summary can be seen in Table 5.1.

All algorithms have an initialization overhead of at least $\mathcal{O}(|V|)$ to allocate and initialize memory for the tentative distances, and for an array of pointers into the various Fibonacci heaps to allow for $\mathcal{O}(1)$ decrease-key and $\mathcal{O}(\log(n))$ removal.

Crauser's static In, or Out, or the combination of both, additionally has an initialization overhead of $\mathcal{O}(|E|)$ because for each edge the minimum incoming/outgoing edge has to be found. This is, as described in Section 5.3.2, implemented as a linear scan over all edges. The same holds true for Träff's Bridge criteria. These criteria also do not incur

Algorithm	**Sequential Time** **Initialization**	**Σ All Phases**
Dijkstra's Algorithm	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|V\| \log\|V\| + \|E\|)$
Crauser's In/Out Heap		
static	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| \log\|V\| + \|E\|)$
dynamic plain	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| \log\|V\| + \|E\| + \|V\|\|E\|)$
dynamic pre-heapifing	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| \log\|V\| + \|E\| + \|V\| \log\|E\|)$
dynamic pre-sorting	$\mathcal{O}(\|V\| + \|E\| \log\|E\|)$	$\mathcal{O}(\|V\| \log\|V\| + \|E\|)$
Crauser's In/Out Array		
static	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2 + \|E\|)$
dynamic plain	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2 + \|E\| + \|V\|\|E\|)$
dynamic pre-heapifing	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2 + \|E\| + \|V\| \log\|E\|)$
dynamic pre-sorting	$\mathcal{O}(\|V\| + \|E\| \log\|E\|)$	$\mathcal{O}(\|V\|^2 + \|E\|)$
Träff's Bridge Heap	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| \log\|V\| + \|E\|)$
Träff's Bridge Array	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2 + \|E\|)$

Table 5.1: Overview of the sequential time complexities of the presented algorithms.

any additional overhead, in an asymptotic sense, during execution of the phases. After the initialization, the only additional overhead to Dijkstra's algorithm is that multiple priority queues have to be managed. This fact is oblivious to an asymptotic analysis.

As described earlier, there are three possible implementations for the dynamic criteria:

- The plain approach does not do any further preprocessing and uses a linear scan after each phase to find the new minima. Since in general $|F| = \mathcal{O}(|V|)$, and the minima have to be validated for all nodes in F after each phase, this means that potentially each edge has to be looked at at the end of a phase. Since in the worst case there are $|V|$ phases, the total overhead is $\mathcal{O}(|V||E|)$. If it is assumed[5] that the set of edges that have to be accessed after each phase is divided by n after each phase, $\sum_{k=0}^{\infty}(|E|/n^k) = \mathcal{O}(|E|)$ holds, and therefore the overhead would only be $\mathcal{O}(|E|)$.
- The pre-heapifing approach has to heapify the edges in the initialization phase. Since this is possible in linear time, no asymptotic overhead occurs compared to the static criteria. After each phase, instead of linear scans through the incoming/outgoing edges, find-mins suffice. Since find-min is possible in $\log n$ time, and there are at most $|V|$ phases, this leads to a total overhead[6] of $\mathcal{O}(|V| \log|E|)$.
- The pre-sorting approach has to sort the incoming/outgoing edges during the initialization phase, therefore leading to an overhead of $\mathcal{O}(|E| \log|E|)$. After each

[5]This assumption is just a thought experiment, no experiments were conducted, and therefore no evidence was collected, that would validate this assumption.

[6]In fact the per-phase overhead would be $\sum_{e \in \hat{E}} \log|e|$, with $\hat{E}$ being a partition of E. This sum is approximated using $\mathcal{O}(\log|E|)$. Similar approximations are done thorough the whole analysis.

phase it now suffices to look at the top of the sorted incoming/outgoing edges and remove it until the top is an edge with a non-settled source/destination. In total this means that each edge is only looked at $\mathcal{O}(1)$ additional times. Therefore, the total overhead is $\mathcal{O}(|E|)$, which is asymptotically irrelevant compared to Dijkstra's algorithm.

The complexity of the parallel algorithms, assuming t threads, depends on many factors. For initialization time one can, assuming that each thread is responsible for roughly the same amount of nodes and edges, replace $|V|$ and $|E|$ with $|V|/t$ and $|E|/t$.

For the time complexity of the phases this is not so simple. In the worst case, e.g., a graph that is just a long linked list, the complexity stays the same, because there is no parallelism in this case.

Only under the following two very strong assumptions it holds that $|V|$ and $|E|$ can be replaced with $|V|/t$ and $|E|/t$ in the sequential runtimes to obtain the parallel runtimes. First, each thread has to be responsible for roughly the same amount of nodes and edges. Second, in each phase the criteria have to find a set of nodes such that the number of nodes to be settled and number of edges to be relaxed in each phase are roughly equal amongst all threads.

5.7 Benchmark

The benchmark was performed on two systems. „mars," a shared memory machine with eight Intel Xeon E7-8850 CPUs, which have a base clock speed of 2 GHz and 10 cores/20 threads each. In total the system has about 1 TiB main memory and 80 cores/160 threads. And „nebula," a shared memory machine with with two AMD EPYC 7551 CPUs. Each of these CPUs has a base clock speed of 2 GHz and 32 cores/64 threads each. Furthermore, the system has about 256 GiB of memory.

Each measurement was repeated 30 times, each time with a different seed, and the median of the results is presented here. The high repetition count, the two graph types, the various criteria, the heap vs. array based implementation, different Δ's for Δ-stepping, and the different thread counts add up to a total benchmark runtime of a few days. For the analysis, the median of the maximum of all thread-times (per run) is used.

Additionally, the data is presented in two columns: The first column „with initialization" includes ①, as seen in Algorithm 5.6 and 5.8, in the time measurement, while the second column „without initialization" does not include this time.

The benchmarks were performed on two types of graphs. Firstly, uniformly distributed graphs as defined in Definition 4.3 and Definition 4.6 with the exception that self-edges and multiple-edges are allowed to simplify the parallel implementation of the graph generation. Secondly, Kronecker graphs as defined in Section 4.1.1 with the initiator matrix $2.5 \cdot (0.57, 0.19; 0.19, 0.05)$ and the Kronecker power 20. Again, with the

simplification that self-edges and multiple-edges are allowed. How these graphs were obtained is described in Section 5.3.1.

The comparison includes Crauser's In, Crauser's Out, and Crauser's Inout criteria in their static variant, Crauser's Inout criteria in its dynamic variant, Träff's Bridge criteria, and Δ-stepping with various values for Δ. The reference speed for the absolute speedup is an efficient sequential implementation of Dijkstra's algorithm.

Each algorithm was executed using 1, 2, 4, 10, 20, 30, 40, 60, 80 and 160 threads on mars. These numbers were chosen with care: 1, 2, and 4 were used because these thread counts are common on home computers. 10, 20, 30, 40, 60, and 80 use exactly one, two, three, four, six, and eight sockets of the given system. The jump from 80 to 160 makes use of the, so far unused, hyper-threads. On nebula the algorithms were executed using 1, 2, 4, 8, 16, 32, 64, 96 and 128 threads because of a similar reasoning.

Figure 5.2, 5.3, and 5.4 depict the absolute time, absolute speedup, and relative speedup of the benchmarked algorithms on uniformly distributed graphs on mars. The static Crauser's Inout criteria performs exceptionally well compared to the other criteria and Δ-stepping. This is followed, albeit with a large gap, by the static Crauser's Out and the static Crauser's In criteria. After that Δ-stepping follows. The dynamic criteria are hopelessly slow. Furthermore, it can be observed that using arrays instead of priority queues consistently leads to better performance. Nevertheless, the static Crauser's Inout criteria only achieves an absolute speedup of 15 with 40 threads and drops in performance for higher thread counts. With 40 threads the absolute efficiency is only 38 %, while with 160 threads the absolute efficiency is at merely 5 %. All other criteria, but not Δ-stepping, exhibit a similar drop in performance with more than 40 threads.

Figure 5.5, 5.6, and 5.7 depict the results for Kronecker graphs on mars. The results draw a similar picture to uniformly distributed graphs with the exception that static Crauser's In, Out and Inout criteria are much closer together, and that all algorithms perform much worse than in uniformly distributed graphs. The best achieved absolute speedup is 4.5 for 80 threads. This is equivalent to an absolute efficiency of about 5 %. For some reason Δ-stepping performs exceptionally bad on this class of graphs.

Figure 5.8, 5.9, and 5.10 show the time, absolute speedup and relative speedup of uniform graphs on nebula. Overall, the picture is not much different to mars with two notable differences: First, the absolute times are much smaller than mars, i.e., nebula seems to be much more computationally capable for this kind of problem. Second, after reaching their peak, the efficiency does not drop as much as on mars. This can be clearly seen when looking at the absolute speedup. Δ-stepping even continues to scale for all number of threads, while it stops to scale for higher number of threads on mars. This might imply either that nebula's hyper-threads are more capable than mars', or that nebula's memory system is stronger than mars'.

There do not seem to be any benchmarks of Crauser's approach in the literature, and even benchmarks on shared memory machines for Δ-stepping are scarce. Meyer et al. [53, Section 9] themselves claim an absolute speedup compared to „an optimized

implementation of Dijkstra's algorithm“ of 3.1 for random d-regular graphs, i.e., graphs where most nodes have the same number of edges, on a *distributed* memory system. This leads to an absolute efficiency of 19 %. For dense graphs the performance is worse, unfortunately no absolute speedup is given by Meyer et al.

Lesnikov and Chernoskutov [47] compared Dijkstra's algorithm, the Bellman-Ford algorithm, and Δ-stepping on a shared-memory system. They claim an absolute speedup of about 3.6 compared to Dijkstra's algorithm with 12 threads when benchmarking using R-MAT graphs, i.e., a special case of Kronecker graphs. These performance numbers are much better than the ones achieved by the implementation of this thesis. Unfortunately, there is no information in this paper about their implementation of Dijkstra's algorithm. Since the implementation of Dijkstra's algorithm for this thesis is highly optimized by using Fibonacci heaps with a custom allocator the absolute speedup might be much lower than compared with a unoptimized implementation of Dijkstra's algorithm. Interestingly, they claim a relative speedup of 2.7 for 12 threads while the implementation for this thesis reaches about 2.4 for 10 threads. These numbers seem to match up.

A fairly recent result by Dhulipala et al. [25] achieves high speedups for a multitude of graph algorithms, including the single-source shortest path problem. They achieve, depending on the graph instance, a *relative* speedup of 28 to 67 for 72 threads for a parallel implementation of the Bellman-Ford algorithm. Interesting to note is that they use the graph compression scheme from Ligra+ [64] to achieve high performance on very large graphs. Nevertheless, according to the previously discussed results of Lesnikov and Chernoskutov [47] Dijkstra's algorithm is about 2.5 times faster compared to the Bellman-Ford algorithm. If one assumes this ratio of performance, this means that Dhulipala et al. only reach an estimated absolute speedup of about 11 to 27 compared to Dijkstra's algorithm (with 72 threads). Depending on the level of optimization performed on Dijkstra's algorithm this could be even worse.

Apart from Ligra+ there are other graph processing frameworks as well. One recent example is Galois [55]. By using this framework the authors achieve a *relative* speedup of 12 with 20 threads, i.e., a relative efficiency of 60 %, and a relative speedup of 9 with 40 threads, i.e., a relative efficiency of 40 %, when benchmarking using a road graph.

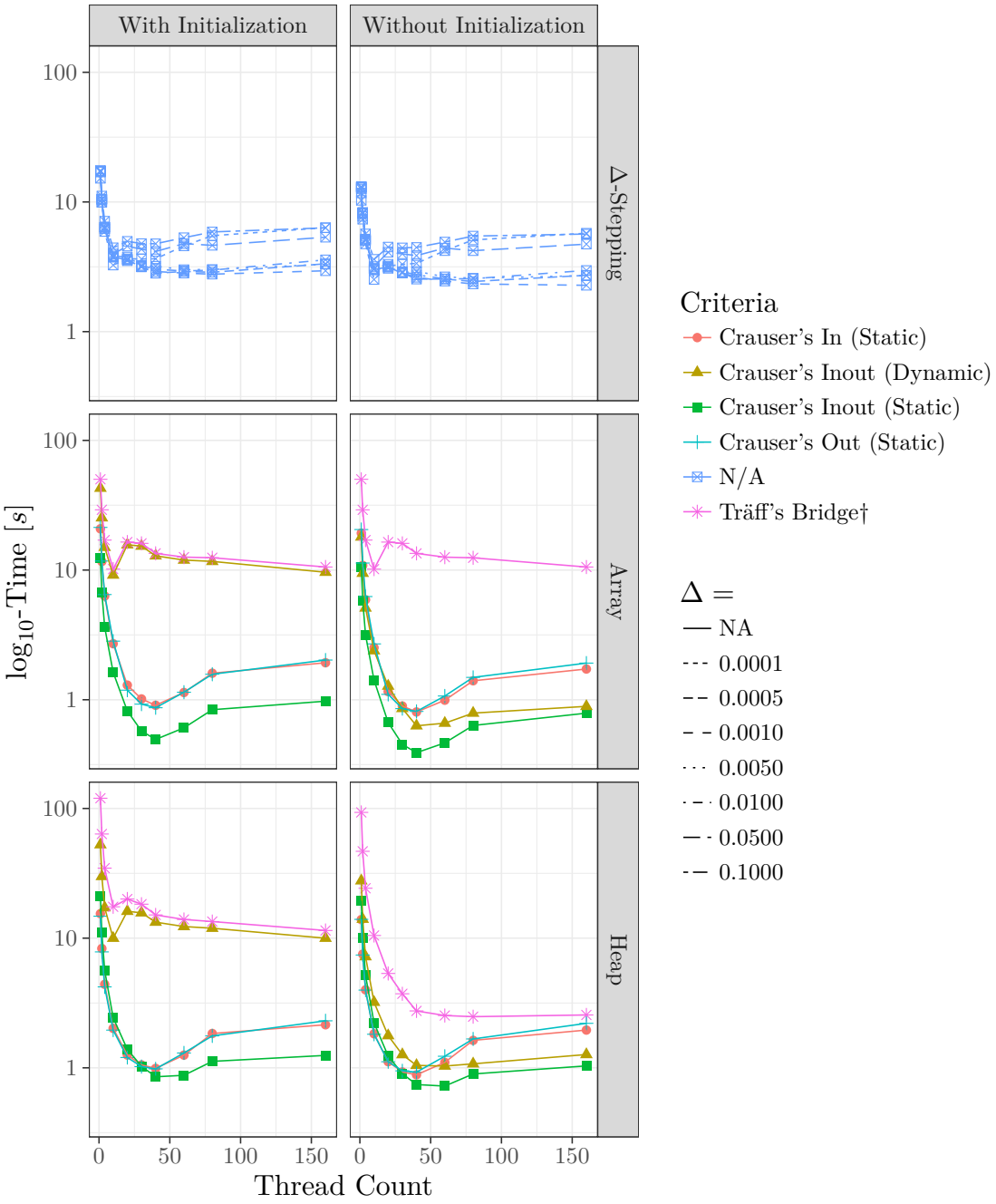

Figure 5.2: Absolute time needed (log-scale) on an uniform graph with 1000000 nodes and an edge chance of 0.0001, i.e., about 100 edges per node and therefore about 100 million edges in total on mars. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

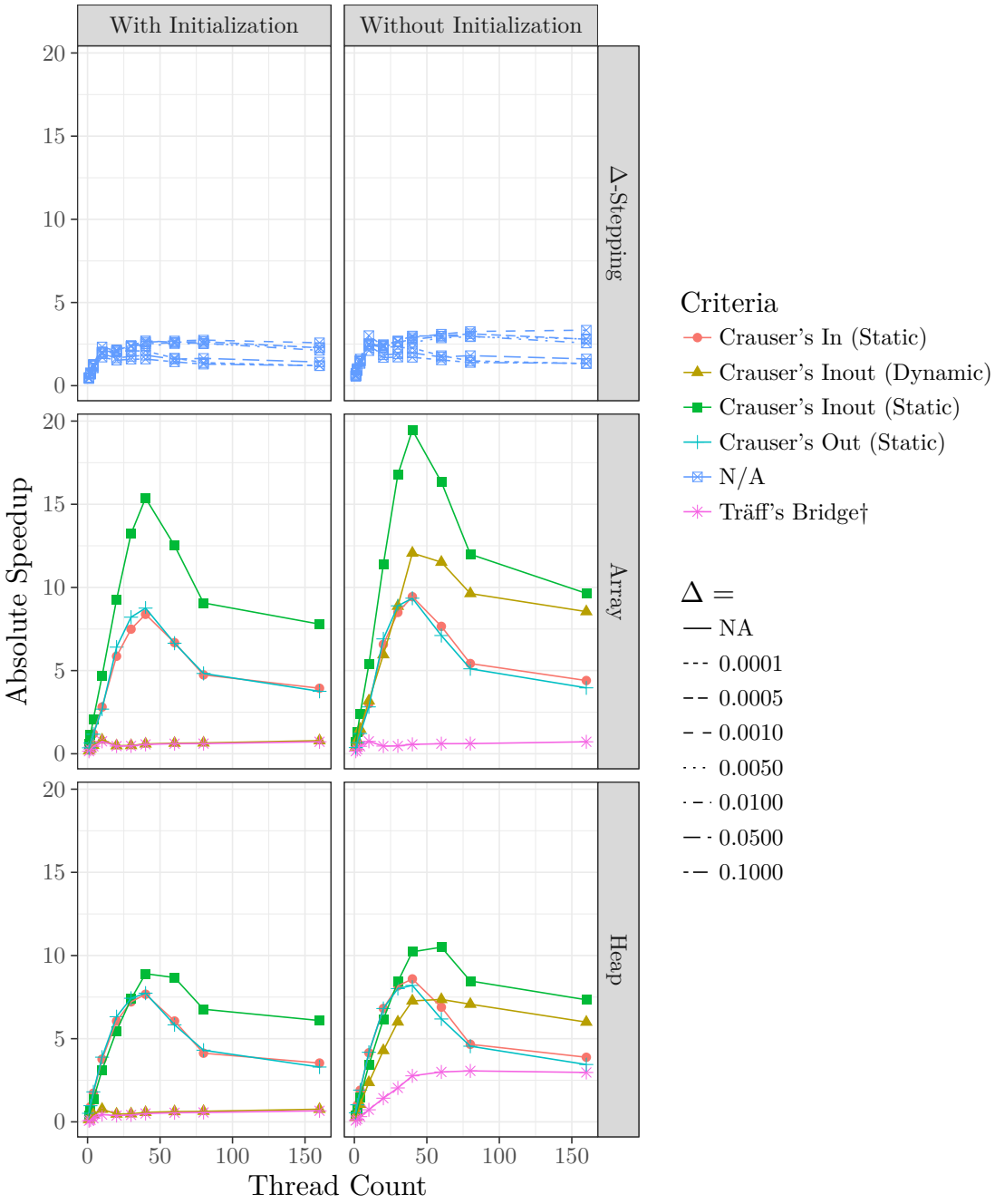

Figure 5.3: Absolute speedup compared to single-threaded Dijkstra's algorithm on an uniform graph with 1000000 nodes and an edge chance of 0.0001, i.e., about 100 edges per node and therefore about 100 million edges in total on mars. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

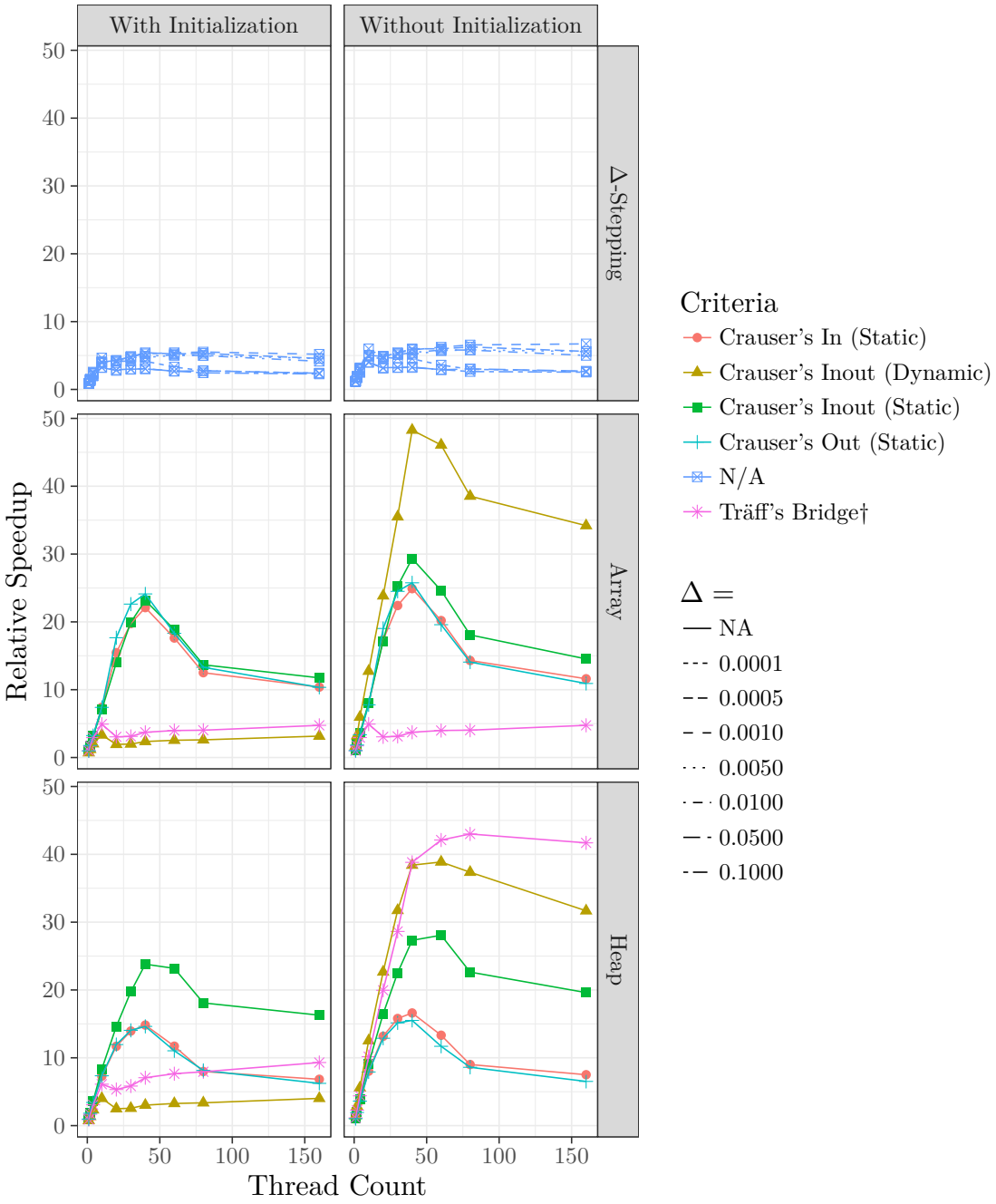

Figure 5.4: Relative speedup compared to the performance with a single thread on an uniform graph with 1000000 nodes and an edge chance of 0.0001, i.e., about 100 edges per node and therefore about 100 million edges in total on mars. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

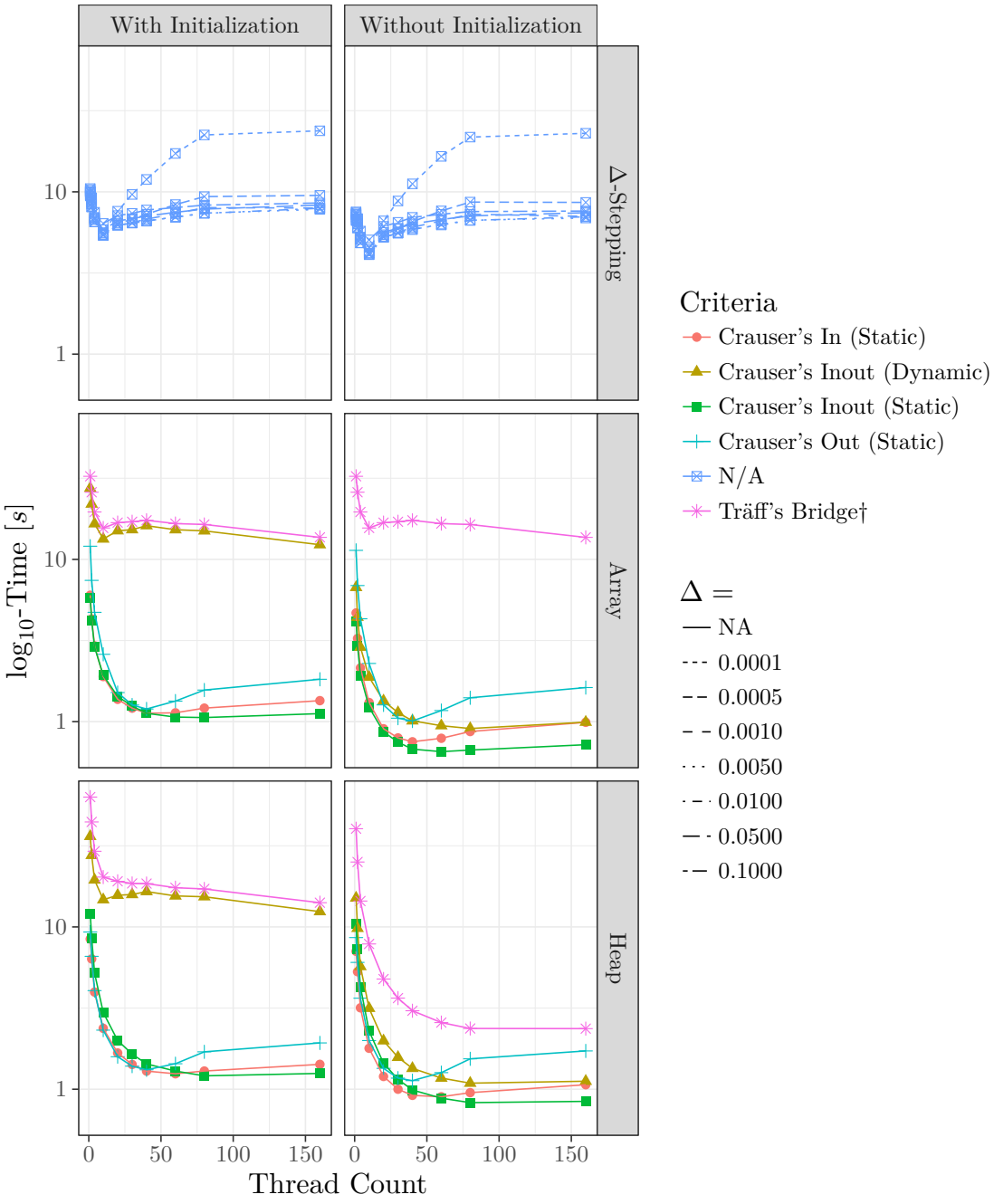

Figure 5.5: Absolute time needed (log-scale) on a Kronecker graph induced by the matrix $(2.5 \cdot (0.57, 0.19; 0.19, 0.05))^{20}$ on mars. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

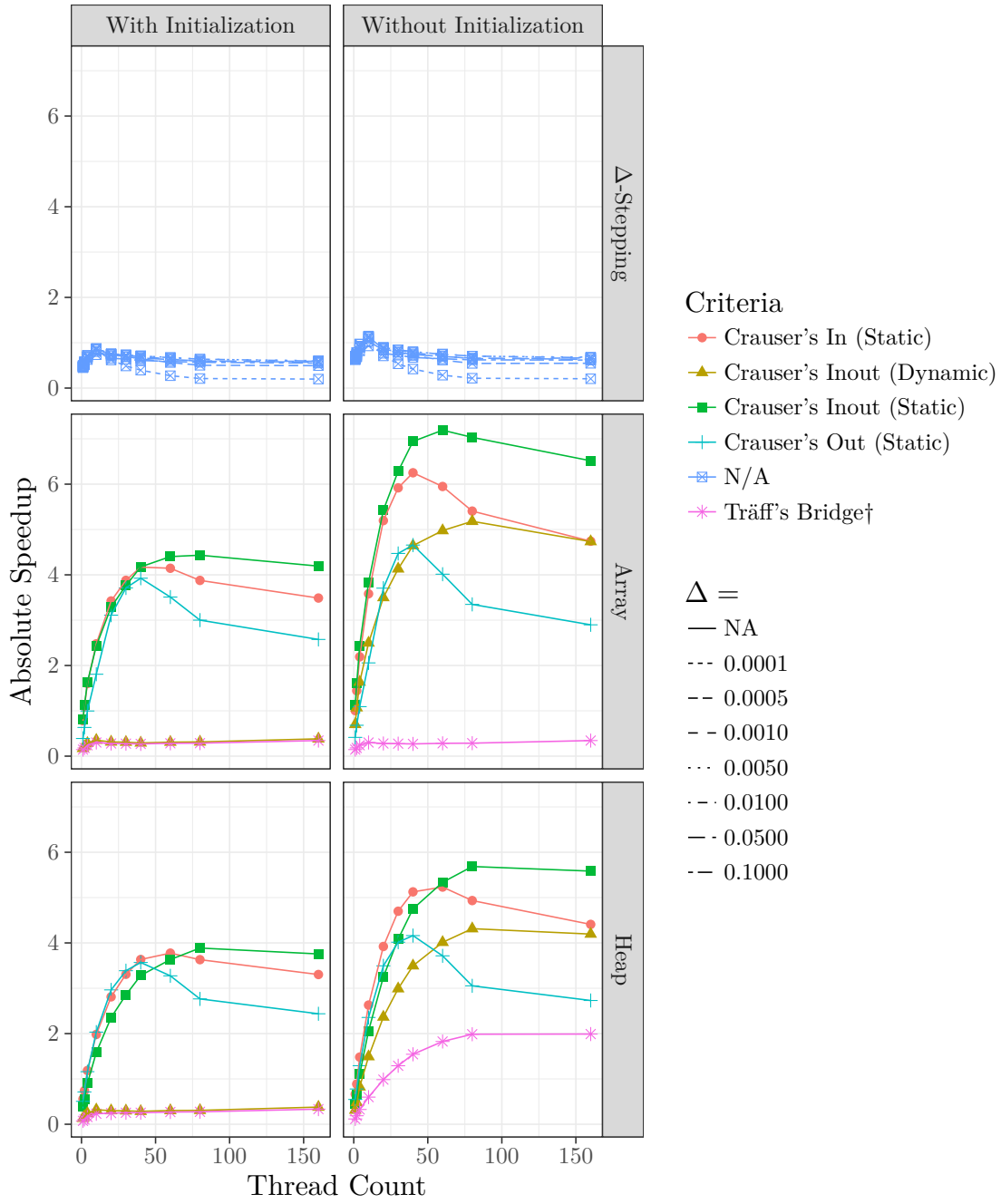

Figure 5.6: Absolute speedup compared to single-threaded Dijkstra's algorithm on a Kronecker graph induced by the matrix $(2.5 \cdot (0.57, 0.19; 0.19, 0.05))^{20}$ on mars. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

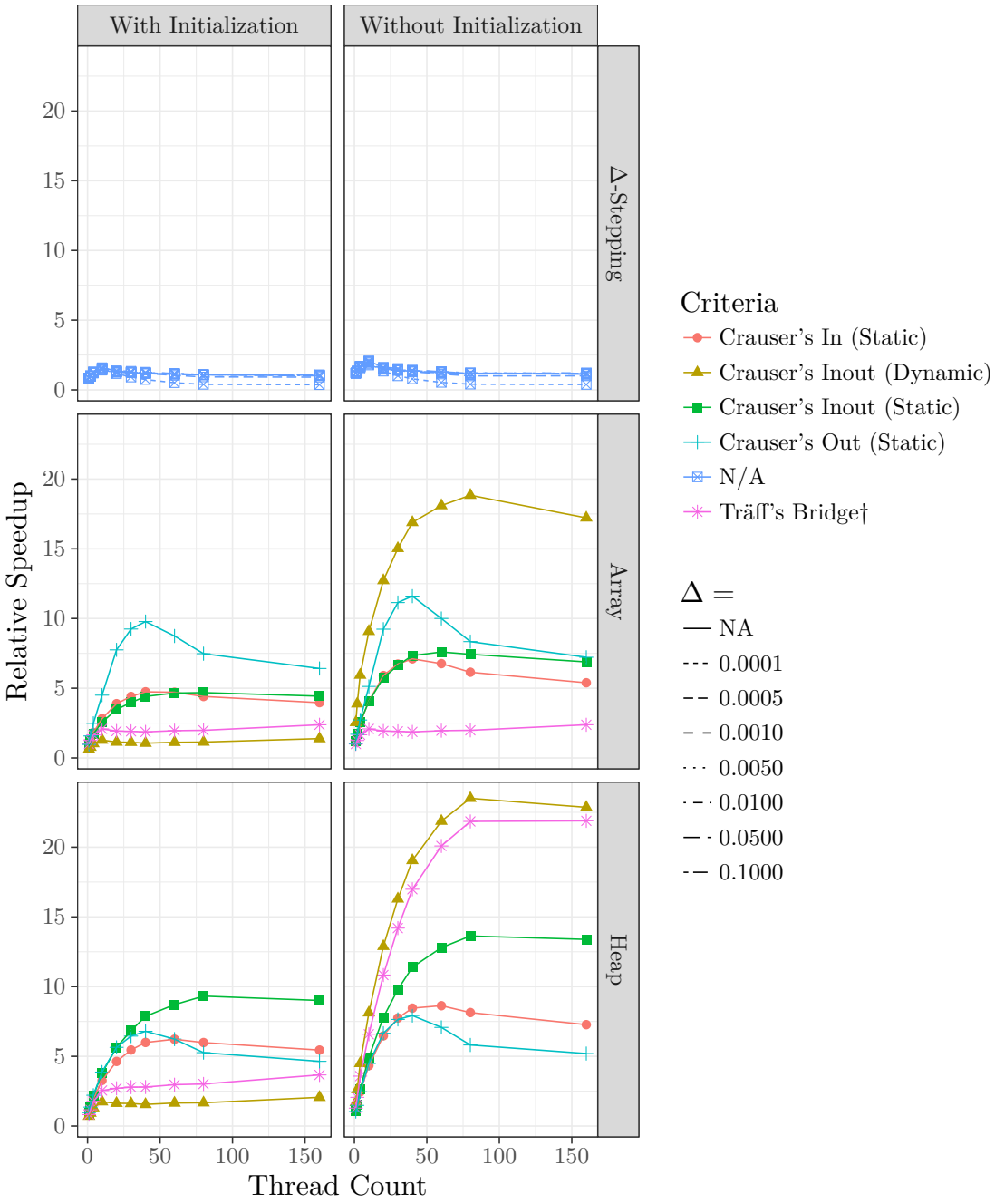

Figure 5.7: Relative speedup compared to the performance with a single thread on a Kronecker graph induced by the matrix $(2.5 \cdot (0.57, 0.19; 0.19, 0.05))^{20}$ on mars. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

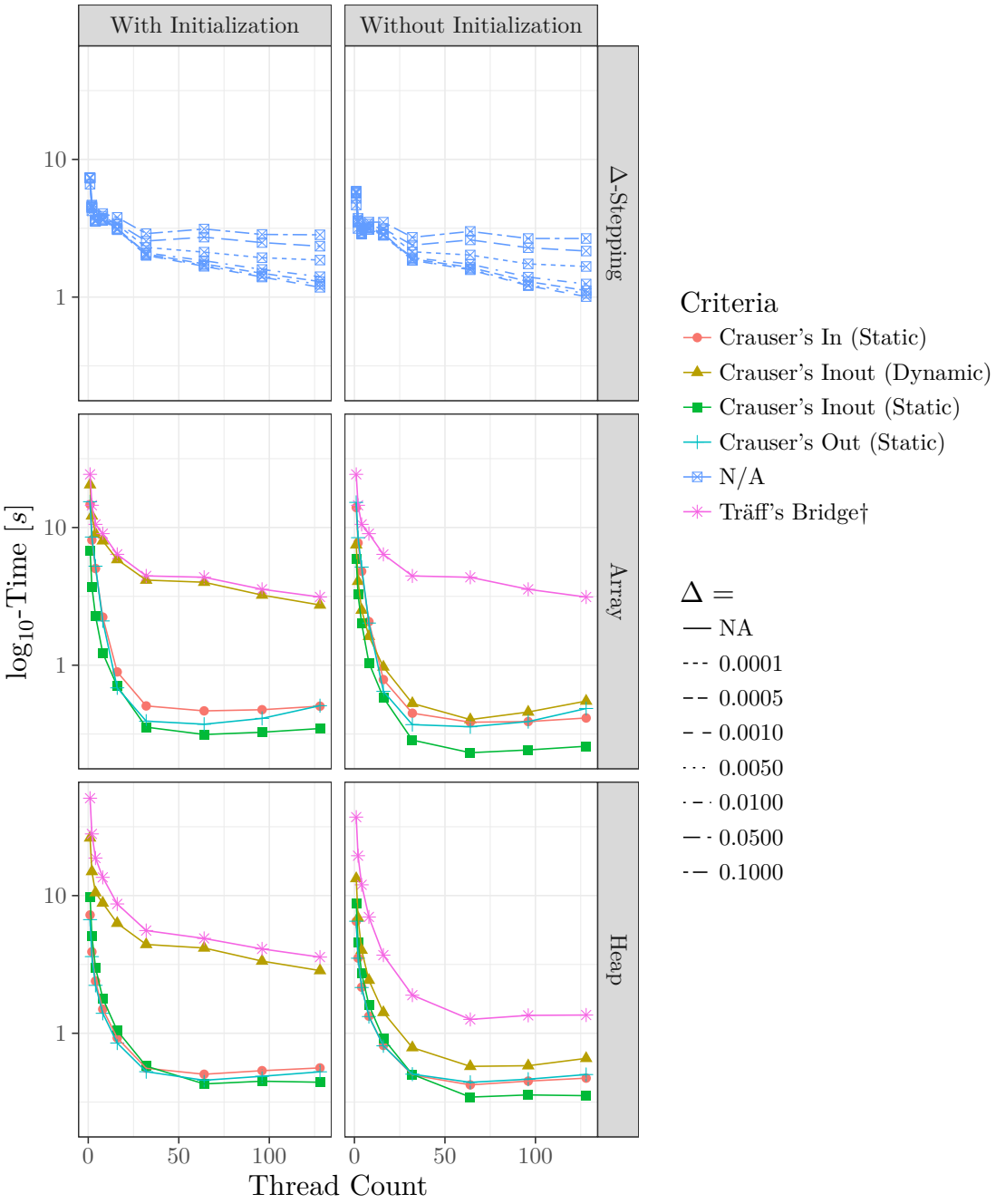

Figure 5.8: Absolute time needed (log-scale) on an uniform graph with 1000000 nodes and an edge chance of 0.0001, i.e., about 100 edges per node and therefore about 100 million edges in total on nebula. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

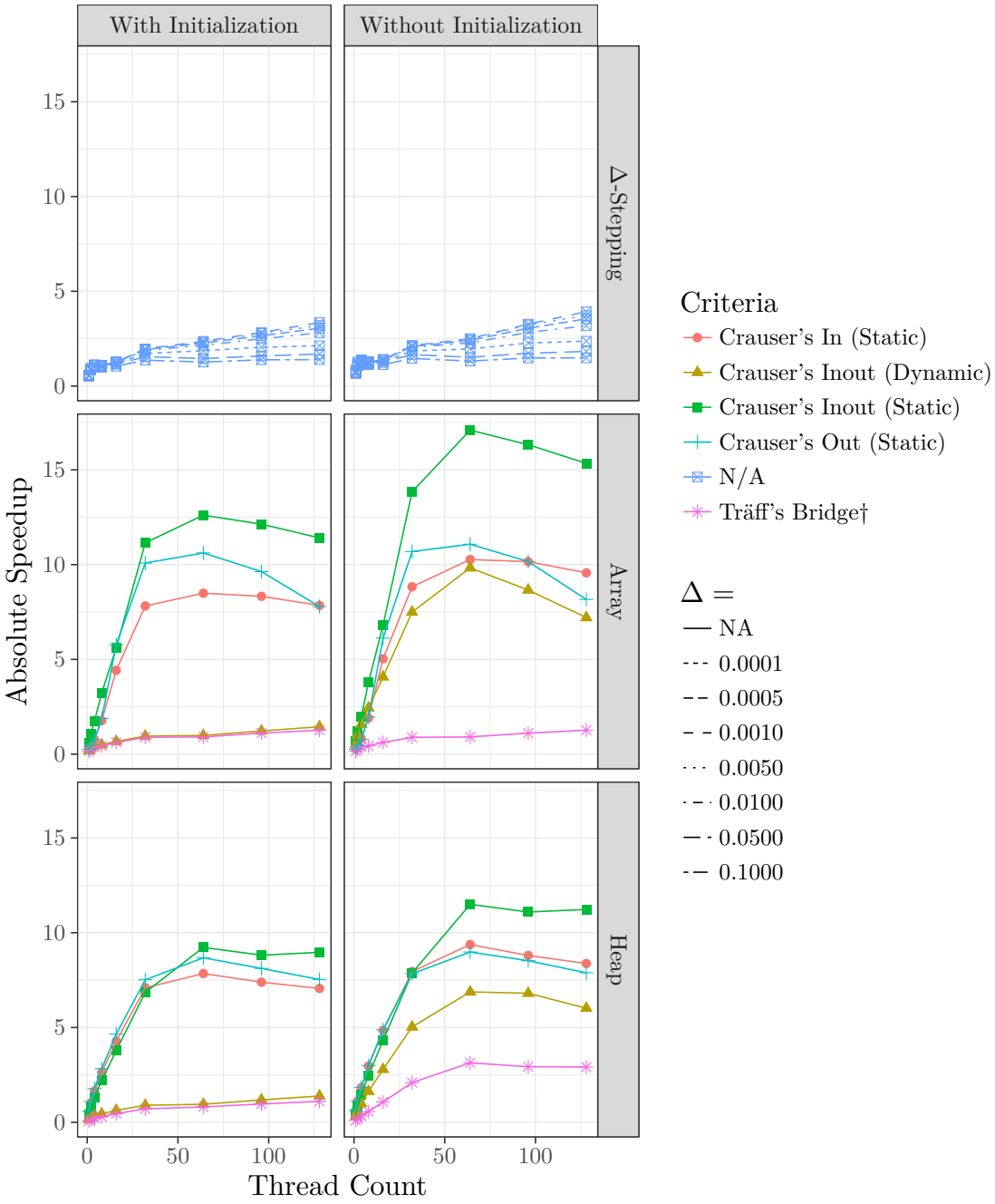

Figure 5.9: Absolute speedup compared to single-threaded Dijkstra's algorithm on an uniform graph with 1000000 nodes and an edge chance of 0.0001, i.e., about 100 edges per node and therefore about 100 million edges in total on nebula. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

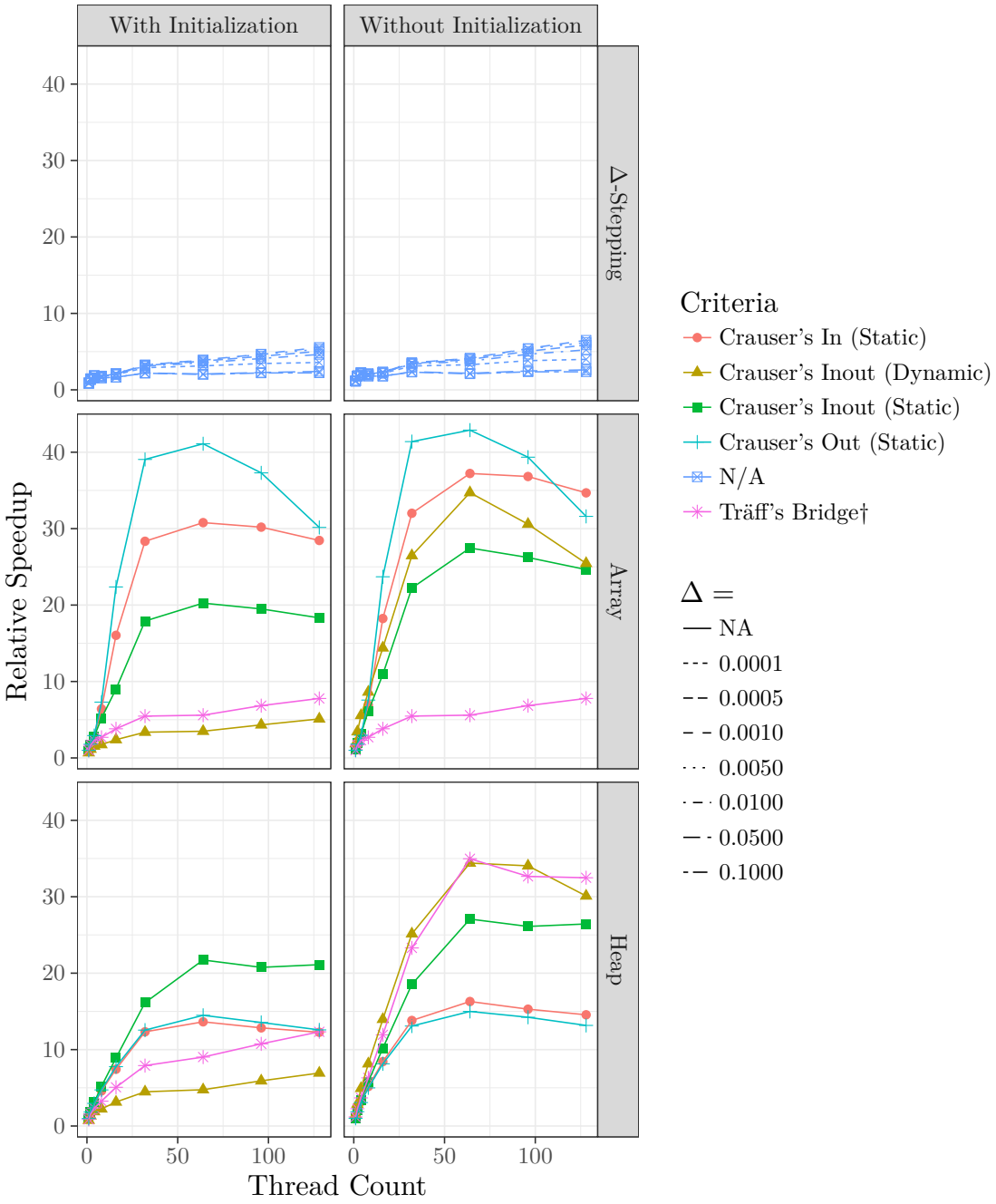

Figure 5.10: Relative speedup compared to the performance with a single thread on an uniform graph with 1000000 nodes and an edge chance of 0.0001, i.e., about 100 edges per node and therefore about 100 million edges in total on nebula. Criteria marked with a dagger (†) are combined with Crauser's Inout (Dynamic) criteria.

CHAPTER 6

Conclusion and Further Research

In Chapter 3 Dijkstra's algorithm, Crauser's original In and Out criteria, the dynamic variation thereof, an heuristic approach, Träff's Bridge criteria, and the oracle has been introduced and proven correct. Furthermore, it has been shown that it is possible to combine criteria and end up with new valid criteria.

This was followed by a simulation (Chapter 4) of these criteria. The results of this simulation validated Crauser et al.'s claims about their static criteria in terms of number of phases. Furthermore, the dynamic variations thereof turned out to be able to improve the number of phases considerably compared to the original versions. This indicates a theoretical potential for more parallelism when using the dynamic criteria instead of the static criteria. The heuristic and Träff's Bridge criteria, unfortunately, did not achieve a notable improvement compared to the static criteria.

Open Question. Does a class of graphs with a heuristic exist, such that the heuristic criteria becomes good? Or is it the case that the heuristic approach cannot work in context of the single-source shortest path problem?

Open Question. Proofs of the average number of phases empirically found in the simulation would be desireable, i.e., proofs for the contents of Table 4.1.

Interestingly, it turned out that the lower bound in number of phases is logarithmic, while all other criteria are still polynomial in the number of phases, although with an exponent less than one.

Open Question. Is it possible to reach the lower bound with a criteria that does not need the solution to the single-source shortest path problem a priori?

The thesis concluded with an implementation and benchmark of the criteria (Chapter 5). The benchmark compared the defined criteria with Δ-stepping and a highly optimized Dijkstra's algorithm on the shared-memory systems provided by the research group.

The original formulation of Crauser's criteria, i.e., the static variants, turned out to be competitive with Δ-stepping and should be considered to be an alternative to Δ-stepping. Unfortunately, no efficient implementation was found for the dynamic criteria, and therefore they remain too slow to be of use in a practical setting. This leads to multiple further questions.

Open Question. Is it possible to implement the dynamic criteria in an efficient manner?

Open Question. How does Crauser et al.'s algorithm scale on distributed memory systems?

Open Question. How does Crauser et al.'s algorithm perform very large graphs?

Open Question. How does Crauser et al.'s algorithm compare with Δ-stepping if for both algorithms a state-of-the-art load-balancing scheme would be implemented?

Bibliography

[1] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 782–793, 2010.

[2] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions*. Advanced Micro Devices Inc., 2018.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Pearson Education, 1993.

[4] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[5] R. E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] S. Bhagat, M. Burke, C. Diuk, I. O. Filiz, and S. Edunov. Three and a half degrees of separation. https://research.fb.com/three-and-a-half-degrees-of-separation/, 2016.

[7] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan. Parallel shortest paths using radius stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 443–454, 2016.

[8] P. Bolzhauser, A. Sulistio, G. Angst, and C. Reich. Parallelized critical path search in electrical circuit designs. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 10–17, 2009.

[9] Boost. Boost C++ libraries. https://www.boost.org/.

[10] R. Bridson. Fast poisson disk sampling in arbitrary dimensions. In *Proceedings of the 34th International Conference on Computer Graphics and Interactive Techniques Sketches (ACM SIGGRAPH Sketches)*, page 22:1, 2007.

[11] G. S. Brodal. Worst-case efficient priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 52–58, 1996.

[12] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.

[13] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 889–901, 2014.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: a recursive model for graph mining. In *Proceedings of the 4th SIAM International Conference on Data Mining (SDM)*, pages 442–446, 2004.

[15] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[16] E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.

[17] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, 1989.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[19] cppreference.com Community. std::memory_order. https://en.cppreference.com/w/cpp/atomic/memory_order.

[20] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 1450 of *Lecture Notes in Computer Science (LNCS)*, pages 722–731, 1998.

[21] J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[22] J. Davies. Poisson-disc sampling. https://www.jasondavies.com/poisson-disc/.

[23] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: hardware-accelerated shortest path trees. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 921–931, 2011.

[24] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS implementation challenge: shortest paths. http://www.dis.uniroma1.it/challenge9/download.shtml, 2006.

[25] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[26] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[27] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM (CACM)*, 31(11):1343–1354, 1988.

[28] R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd, 1938.

[29] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM (CACM)*, 5(6):345, 1962.

[30] L. R. Ford. *Network Flow Theory*. RAND Corporation, 1956.

[31] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1–4):111–129, 1986.

[32] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[33] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[34] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Workshop on Experimental and Efficient Algorithms (WEA)*, volume 5038 of *Lecture Notes in Computer Science (LNCS)*, pages 319–333, 2008.

[35] Graph 500 List. Graph 500. https://graph500.org/.

[36] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[37] R. R. Howell. On asymptotic notation with multiple variables. Technical Report 2007-4, Department of Computing and Information Sciences, Kansas State University, 2008.

[38] Intel Corporation. Intel threading building blocks. https://www.threadingbuildingblocks.org/.

[39] ISO/IEC 14882:2017. Programming languages — C++. Standard, 2017.

[40] D. Jasper. clang-format: automatic formatting for C++. https://llvm.org/devmtg/2013-04/jasper-slides.pdf, 2013.

[41] M.-Y. Kao and P. N. Klein. Towards overcoming the transitive-closure bottleneck: efficient parallel algorithms for planar digraphs. *Journal of Computer and System Sciences*, 47(3):459–500, 1993.

[42] D. Karger, R. Motwani, and G. D. S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

[43] R. M. Karp. The transitive closure of a random digraph. *Random Structures & Algorithms*, 1(1):73–93, 1990.

[44] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley Professional, 3rd edition, 1997.

[45] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research (JMLR)*, 11:985–1042, 2010.

[46] J. Leskovec and A. Krevl. SNAP datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[47] D. Lesnikov and M. Chernoskutov. Performance analysis of Δ-stepping algorithm on CPU and GPU. In *Proceedings of the 47th International Youth School-Conference „Modern Problems in Mathematics and its Applications“ (MPMA)*, pages 164–169, 2016.

[48] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics (SIAP)*, 36(2):177–189, 1979.

[49] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the 9th Meeting on Algorithm Engineering and Expermiments (ALENEX)*, pages 23–35, 2007.

[50] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali. DSMR: a parallel algorithm for single-source shortest path problem. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*, pages 32:1–32:14, 2016.

[51] U. Meyer and P. Sanders. Δ-stepping: a parallel single source shortest path algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA)*, volume 1461 of *Lecture Notes in Computer Science (LNCS)*, pages 393–404, 1998.

[52] U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *Proceedings of the 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science (LNCS)*, pages 461–470, 2000.

[53] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[54] E. F. Moore. The shortest path through a maze. In *International Symposium on the Theory of Switching, Part II*, volume 30 of *Annals of the Computation Laboratory of Harvard University*, pages 285–292, 1957.

[55] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

[56] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris. Employing transactional memory and helper threads to speedup Dijkstra's algorithm. In *Proceedings of the 38th International Conference on Parallel Processing (ICPP)*, pages 388–395, 2009.

[57] NVIDIA Corporation. CUDA zone. https://developer.nvidia.com/cuda-zone.

[58] Open MPI Project. Portable hardware locality (hwloc). https://www.open-mpi.org/projects/hwloc/.

[59] N. Prühs. Implementation of Thorup's linear time algorithm for undirected single-source shortest paths with positive integer weights. Bachelor's Thesis, Department of Computer Science, Christian-Albrechts-Universität zu Kiel, 2009.

[60] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2018.

[61] Rust Project Developers. The Rust programming language. https://doc.rust-lang.org/book/.

[62] P. Sanders and C. Schulz. Scalable generation of scale-free graphs. *Information Processing Letters*, 116(7):489–491, 2016.

[63] H. Shi and T. H. Spencer. Time–work tradeoffs of the single-source shortest paths problem. *Journal of Algorithms*, 30(1):19–32, 1999.

[64] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: parallel processing of compressed graphs with Ligra+. In *Proceedings of the 2015 Data Compression Conference (DCC)*, pages 403–412, 2015.

[65] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[66] M. Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35(2):189–201, 2000.

[67] M. Thorup. On RAM priority queues. *SIAM Journal on Computing (SICOMP)*, 30(1):86–109, 2000.

[68] M. Thorup. Equivalence between priority queues and sorting. *Journal of the ACM (JACM)*, 54(6):28:1–28:27, 2007.

[69] J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *Journal of Parallel and Distributed Computing*, 60(9):1103–1124, 2000.

[70] U. Zwick. Exact and approximate distances in graphs — a survey. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA)*, volume 2161 of *Lecture Notes in Computer Science (LNCS)*, pages 33–48, 2001.