Weak Contraction Hierarchies Work!

Bachelor Thesis of

Tim Zeitz

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dipl.-Inform. Julian Dibbelt
	Dr. Ignaz Rutter
	Dipl.-Inform. Ben Strasser

Time Period: June 1st 2013 – September 30th 2013

Statement of Authorship

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

Karlsruhe, 13th September 2013

Abstract

Modern shortest-path-algorithms are able to answer queries on road networks up to six orders of magnitude faster than the algorithm of Dijkstra. The key ingredient to this success is a preprocessing stage where additional data are generated to accelerate the queries. A popular and successful approach of these techniques are Contraction Hierarchies. In this work, we extend the approach of Contraction Hierarchies to be able to handle arbitrary metrics. We use a recently found connection between Contraction Hierarchies and the well studied elimination game to split the preprocessing into a metric independent and a metric dependent stage. We present a set of simple algorithms based on the idea of Weak Contraction Hierarchies [Col12] in a proof-of-concept implementation. An extensive evaluation shows that these algorithms deliver results comparable to other state-of-the-art methods and are able to incorporate arbitrary metrics on continental sized networks in less then a minute.

Deutsche Zusammenfassung

Moderne Algorithmen können kürzeste-Wege-Anfragen auf Straßengraphen um bis zu sechs Größenordnungen schneller beantworten als der Algorithmus von Dijkstra. Der Schlüssel dazu ist eine Vorberechnungsphase, in der zusätzliche Daten zur Beschleunigung der späteren Anfragen berechnet werden. Ein bekannter und erfolgreicher Ansatz sind Contraction Hierarchies. Diesen Ansatz entwickeln wir in dieser Arbeit dahingehend weiter, dass beliebige Metriken verarbeitet werden können. Dazu nutzen wir aktuelle theoretische Resultate, die eine Verbindung zwischen Contraction Hierarchies und Elimination Game aufgedeckt haben. Damit können wir die Vorberechnungsphase in einen metrikunabhängigen und einen metrikabhängigen Teil aufspalten. Basierend auf der Idee der Weak Contraction Hierarchies [Col12] entwickeln wir eine Proof-of-Concept Implementierung einiger simpler Algorithmen. Eine ausführliche Auswertung der Algorithmen zeigt, dass diese Resultate liefern deren Peformance durachaus vergleichbar ist mit der anderer aktueller Methoden. Unsere Algorithmen können auf einem Straßennetzwerk der Größe eines Kontinents beliebige Metriken in weniger als einer Minute einführen.

Contents

1. Introduction

One of the most popular problems of theoretical computer science is the point-to-point shortest path problem. Most commonly known from routing in road networks, its applications go a lot further. Although the Dijkstra-Algorithm discovered more than 50 years ago runs on road networks in almost linear time, the past decades have still seen a lot of research on this topic – see [BFM09, GH05, SS06, GKW06, Lau04, GSSD08] for example. Considering real world applications, the requirements often go somewhat further than only to find the shortest path between two points. Travel times may be the most common metric, but they are by far not the only one. Often enough multiple metrics for the same road network need to be considered. Travel times by bike will differ a lot from travel times by car. And even if one limits an application to one specific metric, this metric might change over time.

This is a problem since most common approaches tackle the problem in two steps. In a first step, the network will be preprocessed and some additional data will be collected. These additional data are used to accelerate the second step, the actual query. Based on this idea many speedup techniques were developed in recent years. The ninth DIMACS implementation challenge [DGJ09] brought up and expedited many of the new techniques. Some of them outperform a traditional Dijkstra by several orders of magnitude (up to six in special cases). These methods work well as long as the information gathered by preprocessing stays valid. But if this preprocessing is metric dependant, an everyday traffic jam may invalidate the whole result of the costly preprocessing. For this reason, this work considers metric independent preprocessing.

The algorithms presented in this work are based on Contraction Hierarchies which were introduced by Geisberger et al. [GSSD08]. The approach of Contraction Hierarchies has two phases. First, the nodes of the graph get successively contracted, ordered by a certain node order, and some additional edges – shortcuts – will be generated. Second, actual queries (accelerated by the additional edges) are run on the graph. To be able to use Contraction Hierarchies without any given metric, one has to deal with two more problems. To contract the graph, we need a node order based only on the graph's topological features. And as queries concern a certain metric, we need to be able to introduce arbitrary metrics to our preprocessed data.

1.1 Related Work

Since the shortest path problem is a very common and popular problem in computer science, there has been a lot of research on it. The oldest shortest-path-algorithms were invented in the 1950's, Dijkstra's algorithm was proposed 1959 [Dij59], the Bellman-Ford-Moore-algorithm between 1956 and 1959 [Bel58, For56, Moo59]. A lot of the traditional approaches can be found in [AMO93]. During the past decade, especially boosted by the ninth DIMACS implementation challenge [DGJ09], a lot of speedup techniques were developed. A recent overview and evaluation of the different approaches can be found in [Som12].

The fastest (considering query times) currently known approach is named *hublabeling* [CHKZ03, ADGW11, ADGW12]. On a continental sized road network, hublabeling yields distance query times in the average case only five times slower than a distance table lookup. Among other problems, this speed is paid both in time and space with a very expensive preprocessing.

Contraction Hierarchies were proposed in 2008 by Geisberger et al. [GSSD08]. Since then, they have seen a lot of theoretical as well as practical development [GSSV12]. To gain a better understanding of why techniques such as Contraction Hierarchies work so well, the *highway dimension* was introduced by Abraham et al. [AFGW10, ADF$^+$11]. In [BCRW13, Col12], a theoretical framework to study Contraction Hierarchies was developed. It enables proving upper bounds on the search space size for certain classes of graphs. This framework delivers the theoretical basis for this thesis.

An important topic in this work is the ability of the algorithms to adept arbitrary metrics. The currently most advanced approach to this topic is Customizable Route Planning [DGPW13]. CRP utilizes multilevel overlay graphs and is thus a successor to [HSW08, SWW00]. The preprocessing of SHARK [BD09] can also be modified to adapt multiple metrics, but they must be known in advance. Considering Contraction Hierarchies, some possibilities were explored in [GSSV12], to incorporate small changes to the metric without having to repeat the whole contraction.

1.2 Outline

Throughout this work we design a simple three-step-approach similar to the one in [DGPW11] which is able to adapt arbitrary metrics and changes to existing ones. For that we create a proof-of-concept implementation of Weak Contraction Hierarchies and evaluate its potential for real world application. The implementation of these algorithms is very close to the theoretical idea behind them and still allows us to study their theoretical properties on several real world examples.

As many of the algorithms used throughout this work are not new, we will not split this work into a "related work" part and a part with our own results. Instead, we go through the different stages of the method and explain important theoretical results, adapted known algorithms and new algorithms in turn. We start by establishing some basic concepts and algorithms in Chapter 2. Chapter 3, 4 and 5 each cover the details for one of the three phases. For each of the phases we will recap already existing algorithms and methods. In fact, quite a lot of the existing approaches can be reused for Weak Contraction Hierarchies.

Besides the algorithmic details of the contraction process, Chapter 3 contains a summary of the theoretical research leading to the idea of Weak Contraction Hierarchies. Additional, we will study the behavior of a contraction based on a nested dissection order. In Chapter 4 we will cover the details on how to apply arbitrary metrics or introduce changes to existing ones. Chapter 5 considers query algorithms. Chapter 6, which is the core of this work, contains an extensive experimental analysis of the algorithms. There we evaluate the idea of Weak Contraction Hierarchies concerning its potential for real world applications.

2. Preliminaries

In this chapter, we are going to establish notation and terms used in this work, introduce basic theoretical concepts and recap important algorithms.

This work deals mostly with strictly positive weighted directed graphs. We denote such a *graph* as $G = (V, A, len_G)$, where V is the set of nodes, A the set of arcs and len_G an *arc metric*. The metric is a function $len_G : A \mapsto \mathbb{R}_{>0}$ and assigns a strictly positive number to each arc. For non existing arcs len_G will be ∞. *Arcs* are ordered pairs (u, v) with $u, v \in V$, where the *source node* is u and the *target node* v. Multiarcs are not allowed. We apply set operations componentwise to graphs.

Other types of graphs are *unweighted graphs* $G = (V, A)$ or *undirected graphs* $G = (V, E)$. Unless stated otherwise, when talking of graphs we refer to weighted directed graphs. *Edges* are represented as unordered pairs $\{u, v\}$. We denote with $*G = (V, *A)$ the undirected version of a directed graph. The set $*A$ is defined as $*A = \{\{u, v\} \mid (u, v) \in A\}$. We distinguish between arcs and edges: arcs are directed, edges undirected. Most of the definitions in this chapter use only directed graphs, but unless stated otherwise, they also apply to undirected graphs. We will refer to $|V|$ with n and to $|A|$ with m.

Paths in a graph will be represented as a list of nodes: $p = (v_1, v_2, \ldots, v_n)$ with $v_{1..n} \in V$ and $(v_i, v_{i+1}) \in A$. The number of arcs in a path p is $|p| - 1$. For technical reasons, we allow paths with only one node. The length of a path p is the sum over the weights of all arcs and is denoted by $len_G(p)$. A shortest s-t-path is a path from s to t with minimal length among all s-t-paths; the minimal length will be denoted by the distance $dist(s, t)$.

Another important concept in this work is the one of node separators.

Definition 2.1. *Let $G = (V, A)$ be a graph. A subset of nodes $S \subseteq V$ is called a* node separator*, if it splits the graph into two distinct components $X, Y \subseteq V$ so that X, Y and S form a partition of the graph and that no two nodes of X and Y are adjacent.*

Separators are widely used in divide-and-conquer algorithms. By removing the separator from the graph, the algorithm can be applied recursively to the two distinct subgraphs. We are going to use them to establish a metric independent order. To achieve good results, one has to use "good" separators. That usually means small separators and similarily sized components. Finding a separator of minimal size, where the subgraphs X, Y include each less nodes than $\alpha * n$ with α a constant ranging between 1/2 and 1 (commonly referred to as the minimum balanced separator problem), is known to be a $\mathcal{NP}$-hard on arbitrary graphs [BJ92]. With some restrictions to the graph the problem gets somewhat easier. The planar separator theorem [LT79] states that on planar graphs a 2/3-balanced separator with $|S| \in \mathcal{O}(\sqrt{n})$ can be found in linear time. Experiments in Chapter 6 and [DGRW11] show that road networks admit even smaller separators.

In the *point-to-point shortest path problem* we are given a graph G and two of its nodes s and t and we look for $dist(s,t)$. A popular solution to this problem was given more than 50 years ago by Edgar Dijkstra in [Dij59]. He proposed algorithm 2.1, commonly known as the Dijkstra algorithm. Despite its age, this algorithm is still an important component of many approaches to the shortest path problem. Current implementations of that algorithm need a few seconds to answer an average query on a continental sized road network. To find the shortest s-t-path, the algorithm visits each node v of the graph ordered ascendingly by $dist(s,v)$. For this purpose, each node v has a *tentative distance* $d(v)$ initialized to ∞. The tentative distance of s is set to 0. Starting with s, the algorithm updates the tentative distances of each neighbour node of the current node. This process is referred to as *arc relaxing*. We call a node *discovered*, if its tentative distance was updated at least once. This process is then repeated on the next node which is one of the unsettled nodes with the lowest tentative distance. This *settles* the node and its distance from s. Keeping track of the tentative distances requires some sort of priority queue. The algorithm's performance strongly depends on that queue. The theoretically best known data structure considering arbitrary graphs is a *Fibonacci heap* as proposed in [FT87]. Such a heap yields an asymptotic running time in $\mathcal{O}(n \log n + m)$. Practical implementations mostly use k-heaps because of better constants. On sparse graphs, they deliver a asymptotic running time in $\mathcal{O}(n \log n)$ as do Fibonacci heaps. For road networks, experimental suggest a quasi linear running time.

Algorithm 2.1: DIJKSTRA'S ALGORITHM

```
    Input: Graph $G = (V, A, len_G)$, source node $s$
    Data: Priority queue Q
    Output: $dist(s,v)$ for all $v \in V$

    // Initialization
 1  for each $v \in V$ do
 2      d($v$) ← $\infty$
 3  Q.insert($s$, 0)
 4  d($s$) ← 0

    // Main loop
 5  while Q is not empty do
 6      $u$ ← Q.deleteMin()
 7      for each ($u$, $v$) $\in A$ do
 8          if d($u$) + $len_G(u,v)$ < d($v$) then
 9              d($v$) ← d($u$) + $len_G(u,v)$
10              if Q.contains($v$) then
11                  Q.decreaseKey($v$, d($v$))
12              else
13                  Q.insert($v$, d($v$))
```

This algorithm not only computes the distance from s to t, but also the distance from s to all nodes. Since this information is not required in most cases, there are some approaches to reduce this unnecessary overhead. A first step is to introduce a stopping criterion. We can safely stop the algorithm, if the current node u is equal to the target node t. Once we visit a node, the distance is settled and will not change anymore. Unfortunately, that does not change anything concerning the asymptotic running time.

Note: the algorithm can be modified to obtain not only the distance but also the shortest path from s to t. For this, the algorithm stores a predecessor for each node. When an arc gets relaxed and yields an distance improvement, the current node will be assigned as the predecessor of the arc's target node. The path can then be obtained by following the chain of predecessors backward starting at t.

A common improved version of the algorithm - which is in particular important to Contraction Hierarchies - is called *bidirectional Dijkstra's algorithm.* In this algorithm, we run two instances of Dijkstra's algorithm: one starting from s and another one starting from t with all arcs reversed. This can reduce the total number of settled nodes significantly compared to the unidirectional version of the algorithm. But it is not possible to stop as soon as both searches meet at some common node. The bidirectional variant of Dijkstra's algorithm requires a more sophisticated stopping criterion. The idea for the criterion as shown in algorithm 2.2, is to stop as soon as all paths containing nodes not yet settled, are longer than the shortest already discovered path.

One can run the two searches in parallel with two threads. When running both searches sequentially, one has to decide in which search the next step should be performed. Our implementation alternates the instances which is a common approach.

Algorithm 2.2: BIDIRECTIONAL DIJKSTRA'S ALGORITHM

```
  Input: Graph $G = (V, A, len_G)$, source node $s$, target node $t$
  Data: Priority queues $\mathsf{Q}_s, \mathsf{Q}_t$
  Output: $dist(s, t)$
1 Run two Dijkstra instances, one from $s$ and one on the reversed graph from $t$.
  // Main loop
2 while $\mathsf{Q}_s \cup \mathsf{Q}_t \neq \emptyset$ and $\min_{v \in V} \mathsf{d}_s(v) + \mathsf{d}_t(v) > \min_{u \in \mathsf{Q}_s} \mathsf{d}_s(u) + \min_{w \in \mathsf{Q}_t} \mathsf{d}_t(w)$ do
3     Perform a single step of one instance with a non empty queue
4 return $\min_{v \in V} \mathsf{d}_s(v) + \mathsf{d}_t(v)$
```

3. Contraction Hierarchies

Contraction Hierarchies are a speedup technique for the point-to-point shortest path problem and were introduced by Geisberger et al. in [GSSD08]. Broadly speaking, the main idea behind the technique is to exploit the fact that some roads (like highways) are more important than others. A node order is established which models the "importance" of each node. With that order it is possible to decrease the number of nodes (and arcs) the search algorithm has to consider. One can remove "unimportant" nodes in a way that preserves distances. This process is called contraction. In this chapter we are going to examine the contraction process. The result of this process is called a contraction hierarchy. With that hierarchy, an improved query can be run which will be explained in detail in Chapter 5. In this chapter we start with an explanation of the traditional contraction process. Then we discuss the necessary changes to run this process without a given metric. Later on we will recap some theoretical properties of Contraction Hierarchies based on the formal model of weak contraction hierarchies as first discussed in [Col12]. This leads us to a good metric independent node order which we examine closely in the last section.

3.1 Algorithmic Approach

As mentioned above, the main idea for Contraction Hierarchies is to remove unimportant nodes in a way that preserves the distances among all other nodes in the graph. Let us suppose we want to remove the node v. The distance between two other nodes s and t ($s, t \neq v$) might change through the removal of v if and only if v is part of all shortest paths between s and t. Any such path p must contain arcs (u, v) and (v, w). Since p is a shortest s-t-path, the path (u, v, w) must be a shortest u-w-path or we could shorten p by flipping (u, v, w) for the supposed better path which is a contradiction. To preserve the shortest paths, we need to insert a new arc (u, w) with weight $len_G(u, v) + len_G(v, w)$. If such an arc already exists, we need to update its weight. Such an arc is called *shortcut*. To be able to safely remove a node v, we need to check its neighbourhood for shortest paths containing v, and insert the necessary shortcuts to preserve the distances. This process is called *contraction*. Algorithm 3.1 shows a pseudocode version of the routine.

The most difficult part of this routine is to check if (u, v, w) is a shortest path. To gain that information, one can run a local version of Dijkstras's algorithm (called witness search) starting from each node u where an arc (u, v) exists. The local search can stop as soon as all nodes w with an arc (v, w) are settled or if the current distance $\min_{x \in Q} d(x)$ is greater than $len(u, v) + len(v, w)$. If the witness search is run completely for every node, one

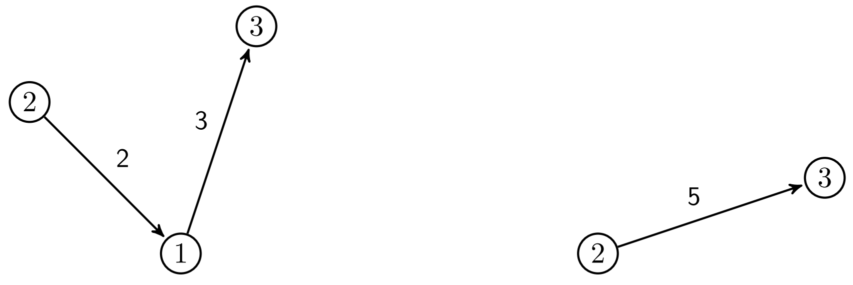

Figure 3.1: Contraction of vertex 1

Algorithm 3.1: Node Contraction

```
    Input: Graph $G = (V, A, len_G)$, node $v$
    Output: Graph $G$ without node $v$
1 for each $u \in V$ with $(u, v) \in A$ do
2     for each $w \in V$ with $(v, w) \in A$ do
3         if $(u, v, w)$ is a unique shortest path then
4             $A \leftarrow A \cup \{(u, w)\}$
5             $len_G(u, w) \leftarrow len_G(u, v) + len_G(v, w)$
6 return $(V \setminus \{v\}, \{(a, b) \mid a, b \neq v\}, len_G)$
```

obtains a minimal contraction hierarchy (minimal for the used node order). But executing the whole witness search is often too slow. Thus, the search is sometimes stopped before it is finished. It may occur that a path shorter than (u, v, w) was not discovered when the local search gets aborted. But as this path still remains in the graph, we only insert an unnecessary shortcut. As the shortcut is longer (or equal) to the actual shortest path it will still preserve the graph's distances. We obtain a valid yet suboptimal contraction hierarchy. But one has to be careful not to insert too many shortcuts, because otherwise the query slows down significantly.

This routine is executed on every vertex until the graph is empty. The result strongly depends on the order in which the nodes are contracted. We are going to refer to this order as $\alpha : V \mapsto \mathbb{N}$. The result of this process is called a contraction hierarchy. A contraction hierarchy $\bar{G}_\alpha = (\bar{G}_\alpha^\wedge, \bar{G}_\alpha^\vee)$ consists of two acyclic graphs, one containing arcs directed from nodes with lower ranks to ones with higher and the other vice versa. Algorithm 3.2 shows the whole preprocessing.

Algorithm 3.2: Contraction Hierarchy Preprocessing

```
    Input: Graph $G = (V, A, len_G)$, order $\alpha$
    Output: Contraction hierarchy $\bar{G}_\alpha = (\bar{G}_\alpha^\wedge, \bar{G}_\alpha^\vee)$
1 $G' = (V', A', len'_G) \leftarrow G$
2 for each $v \in V$ ordered ascending by $\alpha(v)$ do
3     $G' \leftarrow$ contract($G'$, $v$)
4     $A \leftarrow A \cup A'$
5 return $((V, \{(u, v) \in A \mid \alpha(u) < \alpha(v)\}), (V, \{(u, v) \in A \mid \alpha(u) > \alpha(v)\}))$
```

There is still an open question and that is how to obtain α. Since many shortcuts might slow down the query, one key goal of traditional Contraction Hierarchy implementations is to produce as few shortcuts as possible. But finding an order which minimizes the amount of added shortcuts is known to be $\mathcal{NP}$-hard [BCK+10]. Another important target is the minimization of the number of the nodes in the later search space. One has to use heuristical approaches to obtain a good order. Instead of using a precomputed order, the typical implementations usually decide on the fly which node to contract next. In most cases, the key criterion for this decision is the number of shortcuts the contraction of a node might cause, but several other values might be used as well [GSSV12].

3.2 Metric Independent Contraction

From the implementation point of view, contraction on an unweighted graph is actually easier than on a weighted graph. Because it is not possible to decide whether some three nodes make up a unique shortest path, one simply has to add every possible shortcut. Not witness search is required. This creates some problems because as more nodes get contracted, the graph quickly becomes very dense until it finally becomes a clique. Additionally, the query has to deal with many, many arcs later on. Because less information is used, it is not unexpected that the result is a little worse. The simplified algorithm can be found in 3.3. This algorithm was already mentioned in [Col12]. There it has the purpose to construct the maximum weak contraction hierarchy. The next section will explain that term and how it is related to this algorithm.

Algorithm 3.3: WITNESSLESS NODE CONTRACTION

```
Input: Unweighted graph $G = (V, A)$, node $v$
Output: Graph $G$ without node $v$
1 for each $u \in V$ with $(u, v) \in A$ do
2     for each $w \in V$ with $(v, w) \in A$ do
3         $A \leftarrow A \cup \{(u, w)\}$
4 return $(V \setminus \{v\}, \{(a, b) \mid a, b \neq v\})$
```

We would like to emphasize that the result of the whole contraction process strongly depends on the order in which the nodes are contracted. The traditional heuristics to choose the next node will not work as good as before because less structural information is available. As every possible shortcut is inserted, the amount of new shortcuts will not give as much information on the graph's structure as before. So we have to obtain an order through a different way. From the study of the Contraction Hierarchies' theoretical properties, a suggestion on such an order arose. We are going to have a look at these in the next section.

3.3 Weak Contraction Hierarchies

This section gives a brief overview on the theoretical properties of Contraction Hierarchies as discovered in [Col12, BCRW13]. We start by giving a formal definition of an algorithmic contraction hierarchy as developed in the preceding section. Recall that the result of Contraction Hierarchy preprocessing is a pair of graphs, one containing all upward arcs and the other one all downward arcs. They contain arcs from the original graphs and shortcut arcs inserted during the contraction of nodes. For a given graph $G = (V, A, len_G)$ and an order α we define $P_\alpha(s,t) = \{v \in V \mid \alpha(v) > \alpha(s) \vee \alpha(v) > \alpha(t) \wedge dist(s,v) + dist(v,t) = dist(s,t) < \infty\}$ the set of nodes on a shortest path between s and t which have an higher rank than the minimum rank of s an t. In conclusion algorithmic contraction hierarchies can be characterized as following:

Definition 3.1. *Let $G = (V, A, len_G)$ be a graph and let α be an order of its nodes. The arcs of an* algorithmic contraction hierarchy *$\bar{G}_\alpha = (\bar{G}^\wedge_\alpha, \bar{G}^\vee_\alpha) = ((V, \bar{A}^\wedge_\alpha), (V, \bar{A}^\vee_\alpha))$ are then*

$$\bar{A}^\wedge_\alpha = \{(u,v) \in A \mid \alpha(u) < \alpha(v)\} \cup \{(u,v) \mid \alpha(u) < \alpha(v) \wedge P_\alpha(u,v) = \{u,v\}\}$$

$$\bar{A}_\alpha^\vee = \{(u,v) \in A \mid \alpha(u) > \alpha(v)\} \cup \{(u,v) \mid \alpha(u) > \alpha(v) \wedge P_\alpha(u,v) = \{u,v\}\}$$

Furthermore, shortcut arcs have the length of the distance between their nodes: $len_\alpha(u,v) = dist(u,v)$.

A detailed proof of the equivalence of this definition to the result of the algorithms can be found in [BCRW13]. A closer look reveals that we can omit the first subset of the arc sets without changing distances. If an arc (u,v) is no unique shortest path, removing it will preserve the distances. If it is, then $P_\alpha(u,v) = \{u,v\}$ and the arc is contained in the second subset. This leads us to the definition of a *formal contraction hierarchy.*

Definition 3.2. *Let* $G = (V, A, len_G)$ *be a graph and let* α *be an order of its nodes. The arcs of a* formal contraction hierarchy $G_\alpha = (G_\alpha^\wedge, G_\alpha^\vee) = ((V, A_\alpha^\wedge), (V, A_\alpha^\vee))$ *are then*

$$A_\alpha^\wedge = \{(u,v) \mid \alpha(u) < \alpha(v) \wedge P_\alpha(u,v) = \{u,v\}\}$$
$$A_\alpha^\vee = \{(u,v) \mid \alpha(u) > \alpha(v) \wedge P_\alpha(u,v) = \{u,v\}\}$$

Shortcut lengths are the same as in definition 3.1.

Although algorithmic and formal contraction hierarchies are equivalent concerning distances, the arc sets may differ. Furthermore, actual implementations will produce arc sets different from the ones defined as algorithmic contraction hierarchies. Since only heuristics are used to decide whether a shortcut is necessary, more shortcuts than needed may be inserted. The concept of weak contraction hierarchies aims to cover all those slightly differing contraction hierarchies.

Definition 3.3. *A* weak contraction hierarchy H_α *of a graph* $G = (V, A)$ *and an order* α *is a pair of graphs* $(H_\alpha^\wedge, H_\alpha^\vee) = ((V, B_\alpha^\wedge), (V, B_\alpha^\vee))$ *fulfilling the following three conditions.*

1. $G_\alpha \subseteq H_\alpha$
2. $\alpha(u) < \alpha(v)$ *for all* $(u,v) \in B_\alpha^\wedge$ *and all* $(v,u) \in B_\alpha^\vee$
3. *If* (u,w) *is an arc in* H_α *but not contained in* A*, then there exists one (or more) pair of arcs* $(u,v) \in B_\alpha^\vee$ *and* $(v,w) \in B_\alpha^\wedge$

This definition is not concerned with the metric but only the structure of contraction hierarchies. We are going to take the arc lengths into consideration in Chapter 4. Property 1 of definition 3.3 makes clear that a formal contraction hierarchy is the smallest possible weak contraction hierarchy. On the other, hand one could ask how big a weak contraction hierarchy can become. For that we are going to define the *maximal weak contraction hierarchy*

Definition 3.4. *A weak contraction hierarchy* H_α *is maximal if, and only if it satisfies the following conditions:*

1. *Each arc of* A *is contained in* H_α.
2. *For any two arcs* $(u,v) \in B_\alpha^\vee$ *and* $(v,w) \in B_\alpha^\wedge$, H_α *contains also* (u,w).

This definition exactly matches the result of the metric independent (witnessless) contraction algorithm. Algorithm 3.3 was proposed in [Col12] to compute the maximal weak contraction hierarchy. Moreover, it is very similar to a different (and well studied) problem in theoretical computer science – the so called *elimination game.* It works almost like witnessless contraction but on undirected graphs. That is, nodes are removed by a given order. For each removed node, edges are inserted - a clique between all its neighbours. We denote the set of all additional inserted edges with F_α. In [Col12], the similarities between the elimination game and the maximum weak contraction hierarchy were discovered and stated in the following theorem.

Theorem 3.5. *Let $G = (V, A)$ be a graph, $(H_\alpha^\wedge, H_\alpha^\vee) = ((V, B_\alpha^\wedge), (V, B_\alpha^\vee))$ the maximum weak contraction hierarchy of it, based on the node order α. The contraction hierarchy (and with that any weak contraction hierarchy) is a subgraph of the result of the elimination game.*

$$*(H_\alpha^\wedge \cup H_\alpha^\vee) \subseteq *G_\alpha = (V, *A \cup F_\alpha)$$

With the elimination game, another concept is introduced: the elimination tree.

Definition 3.6. *Given an undirected graph $G = (V, E)$, a node order α and a set of edges inserted during the elimination game F_α, the arcs of the* elimination tree *$T(G, \alpha) = (V, A_\alpha^T)$ are defined as follows:*

$$A_\alpha^T = \{(u, v) \mid \{u, v\} \in E \cup F_\alpha \mid \alpha(u) < \alpha(v) \leq \alpha(w) : \forall\{u, w\} \in E \cup F_\alpha\}$$

The root of the tree is the node with maximum rank.

The elimination tree is constructed from the result of the elimination game. For each node it contains only the arc to the lowest higher node. The node with the highest rank is the tree's root. In [Col12], it was discovered that the elimination tree and the search space in Contraction Hierarchies are closely related to each other. Theorem 3.8 states this result. We are going to give slightly different proof than the one given in [Col12] for that result in this work. But first we need a formal definition of search space.

Definition 3.7. *The* search space *$\mathcal{S}$ of a node v in a given graph $G = (V, A)$ is the set of all nodes w, where a path from v to w exists. The* reverse search space *$\mathcal{R}$ of a node v is the set of all nodes w where a path from w to v exists.*

$$\mathcal{S}(v, (V, A)) = \{w \in V \mid \exists p = (v, \ldots, w)\})$$

$$\mathcal{R}(v, (V, A)) = \{w \in V \mid \exists p = (w, \ldots, v)\})$$

The reverse search space is equivalent to the normal search space when all arcs are reversed. Now we can prove the following theorem.

Theorem 3.8. *Let $G = (V, A)$ be a graph, $(H_\alpha^\wedge, H_\alpha^\vee) = ((V, B_\alpha^\wedge), (V, B_\alpha^\vee))$ some weak contraction hierarchy of it based on the node order α. For any node $v \in V$*

$$\mathcal{S}(v, H_\alpha^\wedge) \subseteq \mathcal{S}(v, T(*G, \alpha))$$

$$\mathcal{R}(v, H_\alpha^\vee) \subseteq \mathcal{S}(v, T(*G, \alpha))$$

Proof. First note that the second case is equivalent to the first with reversed arcs. So we are going to prove only the first. We use induction over the nodes of V starting with the highest ranked node.

Basis: Let v be the node with maximum $\alpha(v)$. By definition $B_\alpha^\wedge$ will not contain any outgoing arcs from v. So $\mathcal{S}(v, H_\alpha^\wedge)$ must be empty. For that, it is trivially a subset of whatever $\mathcal{S}(v, T(*G, \alpha))$ might be. One may note that $\mathcal{S}(v, T(*G, \alpha))$ is empty as well, since the elimination tree also contains only upward arcs.

Inductive step: We assume that the theorem holds for all nodes with rank greater than n. We will prove that that concludes that it also holds for the node with rank n. Let v be that node. Due to Theorem 3.5, all arcs from $H_\alpha^\wedge$ are also present in $*G_\alpha$. If there is only one outgoing arc from v in $H_\alpha^\wedge$ to the target node w, then $\mathcal{S}(v, H_\alpha^\wedge) = \{v\} \cup \mathcal{S}(w, H_\alpha^\wedge)$. This arc exists in $*G_\alpha$ (as an edge), too. If it is the only arc, w is the lowest higher node and the arc is included in the elimination tree, too. If there are additional edges in $\{v, x\} \in *G_\alpha$ with $\alpha(x) < \alpha(w)$, then the edge $\{x, w\}$ was inserted during the removal of v. There still could be another node y above x and below w, but then again a shortcut must exist. Since the ranks are only integers, the number of nodes between v and w is finite and there exists a path from v to w in the elimination tree. We conclude that $\{v\} \cup \mathcal{S}(w, T(*G, \alpha)) \subseteq \mathcal{S}(v, T(*G, \alpha))$ and since $\mathcal{S}(w, H_\alpha^\wedge) \subseteq \mathcal{S}(w, T(*G, \alpha))$, the theorem holds true due to the inductive condition.

Let us suppose that there is more than one outgoing arc v in $H_\alpha^\wedge$. We denote the targets of each of those arcs with w_1 to w_i. Let $\alpha(w_n) < \alpha(w_{n+1})$. As w_1 to w_i are all included in $\mathcal{S}(v, H_\alpha^\wedge)$, we must prove that they are in $\mathcal{S}(v, T(*G, \alpha))$, too. Since w_1 is the lowest higher node, it is in $\mathcal{S}(v, T(*G, \alpha))$, due to the same reasons used in the case of only one arc. As both edges $\{v, w_1\}$ and $\{v, w_2\}$ exist in $*G_\alpha$, the edge $\{w_1, w_2\}$ was inserted during removal of v, if it did not exist earlier on. If w_2 is the lowest higher connected node to w_1, it is reachable in the elimination tree and thus also included in the search space. If there is another connected node u with $\alpha(w_1) < \alpha(u) < \alpha(w_2)$, then the removal of w_1 causes the insertion of the edge $\{u, w_2\}$ (again if it did not exist before). If there is still a node between u and w_2 the argument applies again until finally w_2 is reached in the elimination tree. So we conclude $w_2 \in \mathcal{S}(w_1, T(*G, \alpha))$. If we apply this argument inductively, we get $w_{j+1} \in \mathcal{S}(w_j, T(*G, \alpha))$. We finally conclude that for all $j \in \mathbb{N}$ with j ranging between 1 and i, w_j is in $\mathcal{S}(v, T(*G, \alpha))$. Due to the inductive condition, that proves the theorem's statement. □

As the search space in any contraction hierarchy is a subset of the search space in the elimination tree, the height of the tree limits the size of any contraction hierarchy search space. No search space can become larger than the longest path from any leaf to the root in the elimination tree. Earlier studies of the elimination game came up with heuristics to obtain node orders yielding elimination trees with small height. One of them is nested dissection [BGHK92]. Since a separator based order is metric independent, it is probably a good order for the metric independent contraction. The next section will cover it in detail.

3.4 Nested Dissection Order

As mentioned before, the node orders of traditional Contraction Hierarchies are obtained by fast heuristics. Thus it is very hard to define them in a formal way – not to mention proving any non-trivial statements about their properties. As for nested dissection orders, things work out slightly better. In this section we are going to define nested dissection orders, and then study their behaviour on an extensive example.

We get a nested dissection of the graph if we, using a node separator, cut the graph in two halves, and then repeat the process recursively on each of the halves. The process is stopped when the node sets get smaller than a certain value n_0. For the following considerations we will assume n_0 is 1, although in practice on would use a bigger constant.

Definition 3.9. *A* nested dissection *for a given graph $G = (V, A)$ is a tupel $N = (R, L, S)$. The halves R and L are either nested dissections again or sets of nodes of G; S is always a set of nodes. If either R or L is a node set, then the number of nodes in the set must be smaller than n_0.*

We define:

$$nodes(N) := \begin{cases} nodes(R) \cup nodes(L) \cup S & \text{if } N = (R, L, S) \text{ a nested dissection} \\ N & \text{if } N \subseteq V \end{cases}$$

No arcs between $nodes(R)$ and $nodes(L)$ are allowed.

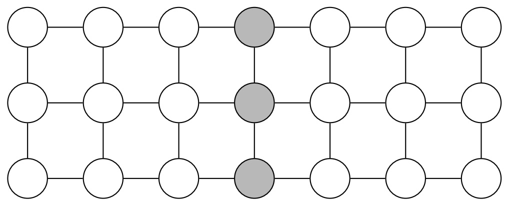
Figure 3.2: The grid graph $Grid(3, 2)$ and its separator.

For a node order α constructed from a nested dissection $N = (R, L, S)$, we demand that the nodes in the separator S have higher ranks than the nodes in the halves R and L. More formal:

$$\forall s \in S, x \in nodes(R) \cup nodes(L) : \alpha(s) > \alpha(x) \tag{3.1}$$

Nodes within the separator or inside the final node sets may be ordered arbitrarily. One may note that there are still multiple possibilities to rank the nodes even without considering the separators and final sets. For example, if we have the nested dissection $((RR, RL, RS), (LR, LL, LS), S)$, both ($RR$, RL, RS, LR, LL, LS, S) and (RR, RL, LR, LL, RS, LS, S) would fulfill the condition of (3.1). But as the halves are independent of each other, the resulting contraction hierarchy will be the same. As the tool we used to obtain our orders gave us the first type of orders - where each separator comes immediately after the sets in which it splits the graph - we are going to consider these orders in the examples. For any properties of the orders we will only rely on condition (3.1).

3.5 Behavior of Contraction Ordered by Nested Dissection

To get a better impression of the behavior of a nested dissection order, we closely examine the contraction of simple idealized road graphs - which are still not far away from the reality of certain American cities. We investigate the contraction of a grid graph.

Definition 3.10. *In a (undirected)* grid graph *$G = (V, E)$, each node $v \in V$ can be identified with a pair of integer coordinates: $v = (x, y)$. With that edges are defined as follows:*

$$E := \{\{(x_1, y_1), (x_2, y_2)\} \mid (|x_1 - x_2| = 1 \wedge y_1 = y_2) \vee (|y_1 - y_2| = 1 \wedge x_1 = x_2)\}$$

To get minimal and perfectly balanced separators, we only consider grid graphs of a certain size. We demand the existence of a small perfectly balanced separator and recursively for each subgraph. For example, a chain of three nodes is such a graph since we could take the middle node as a minimal perfectly balanced separator. A chain of seven nodes could be separated by the middle node into two chains of three. And so on. The valid grid sizes form a sequence of numbers fulfilling the following condition: $gridsize(n) = 2 * gridsize(n-1) + 1$. With $gridsize(1) := 1$ this is (as can be shown by induction) equivalent to the following closed function:

$$gridsize(n) := 2^n - 1$$

And, of course, we can extend the graphs not only into the first dimension but also into the second. Thus, we can introduce the following definition:

$$Grid(width, height) := \left(\left\{(x, y) \mid x \in \mathbb{N}_{\geq 0}, x < 2^{width} - 1, y \in \mathbb{N}_{\geq 0}, y < 2^{height} - 1\right\}, E\right)$$

where E are the induced grid graph edges. Let us assume that *width* is greater or equal to *height*; if not, we just flip the graph by 90 degrees.

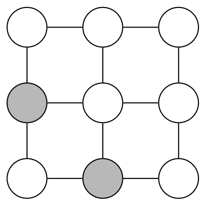
Figure 3.3: A smaller but unbalanced separator.

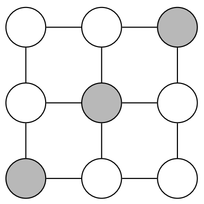
Figure 3.4: Another perfectly balanced separator.

In a grid graph $Grid(width, height)$, the set $S = \{(gridsize(width - 1), y) \mid y \in \mathbb{N}_{\geq 0}, y < gridsize(height))\}$ (less formal: the nodes which cut the graph into two halves along the longer side) is a small perfectly balanced separator. We want to point out that S is neither a minimal separator nor the only perfectly balanced. Figure 3.3 shows a smaller separator and Figure 3.4 a different perfectly balanced separator.

Nevertheless, S is a good separator and its size is within $\mathcal{O}(\sqrt{n})$. Thus it is absolutely sufficient for our purposes. So we will base the node order on such separators. An example of a contraction using such an order can be seen in Figures 3.5 to 3.15. They show the contraction of the grid graph $Grid(3, 2)$. The darker the color of a node, the higher its rank.

We can observe some interesting facts from this example: When separators are contracted, they are fully connected and form a clique. Contracting a clique with n nodes yields a running time in $\mathcal{O}(n^3)$ assuming constant edge lookup and constant node removal. We conclude that the size of the final separator, which is expected to be the largest, will have significant impact on the contraction's performance.

The final clique is not only composed of the topmost separator, but up to three topmost separators, but no more. The leftmost node in Figure 3.11 is a separator for the three leftmost nodes. If this separator consisted of more than one node and would be positioned between the other two separators, all three separators would form a clique (after the contraction of the first node of the lowest separator). But it is not possible to add a fourth separator to this clique, only to parts of it. It would always be separated from some or many of the nodes of the original clique.

In this example, the original graph has 32 edges and during contraction 33 edges are inserted. A traditional Contraction Hierarchy usually creates less or about as many shortcuts as there were original edges. Although the amount of shortcuts for this concrete example seems to fit to that behaviour, the final clique suggests that this might change for larger graphs. Figure 3.16 shows the number of added shortcuts for all grid graphs $Grid(x, y)$ with x between 1 and 7 and y between 1 and 6. The number of edges in the graph is roughly limited by $2n$ (one edge up and on to the right). Different to traditional Contraction Hierarchies, the number of shortcut edges grows faster then the number of original edges. This is no big surprise since we are using much less information. In our experiments, we also tried to use a traditional Contraction Hierarchy order in a witnessless contraction. This yields even more shortcuts – see Section 6.2.2. Tobias Columbus proved that the total number of edges in a weak contraction hierarchy on a graph of a minor-closed class based on a nested dissection order with good separators is in $\mathcal{O}(n \log n)$ [Col12]. Our results fit into that theorem pretty well.

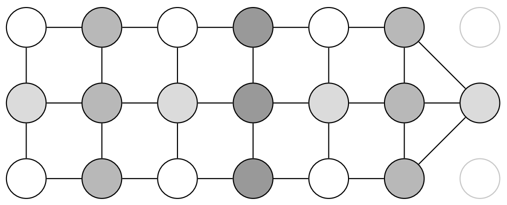
Figure 3.5: First two nodes contracted

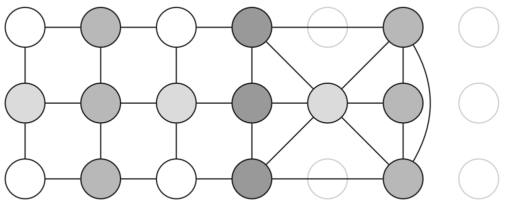
Figure 3.6: First separator and two more nodes contracted

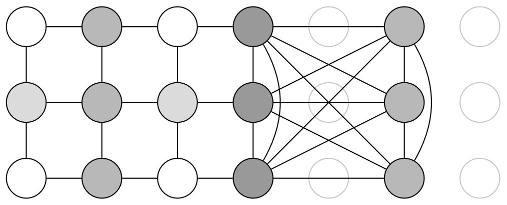
Figure 3.7: Two separators form a clique

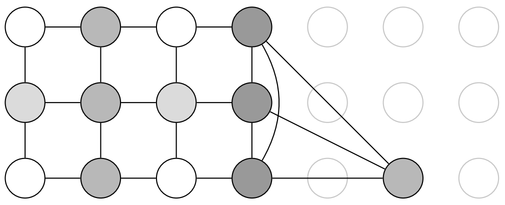
Figure 3.8: The next two nodes contracted - no new shortcuts

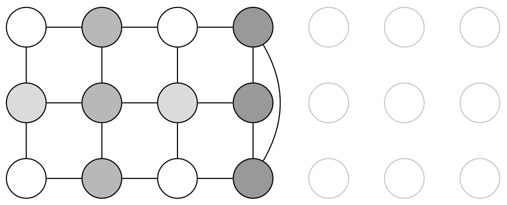
Figure 3.9: Right side completely contracted

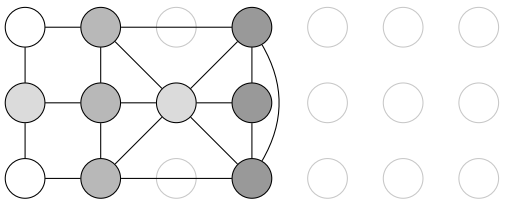
Figure 3.10: First two nodes on the left contracted

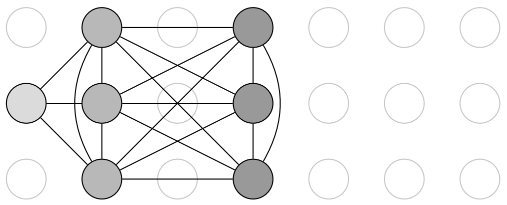
Figure 3.11: Both pairs of separators form a clique

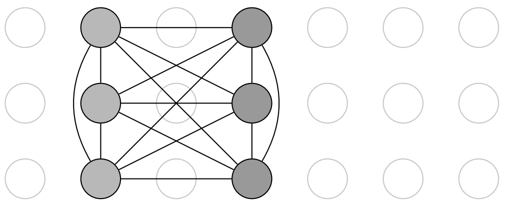
Figure 3.12: The final clique

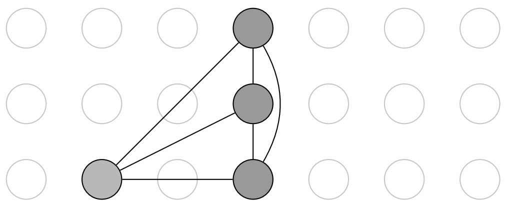
Figure 3.13: No new shortcuts were added anymore

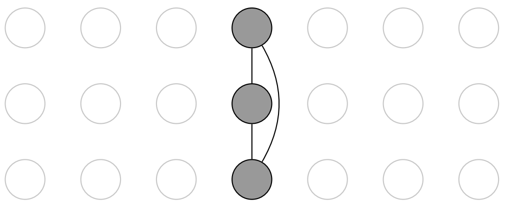
Figure 3.14: Toplevel separator - a clique

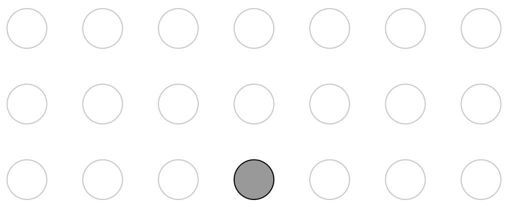
Figure 3.15: The last remaining node

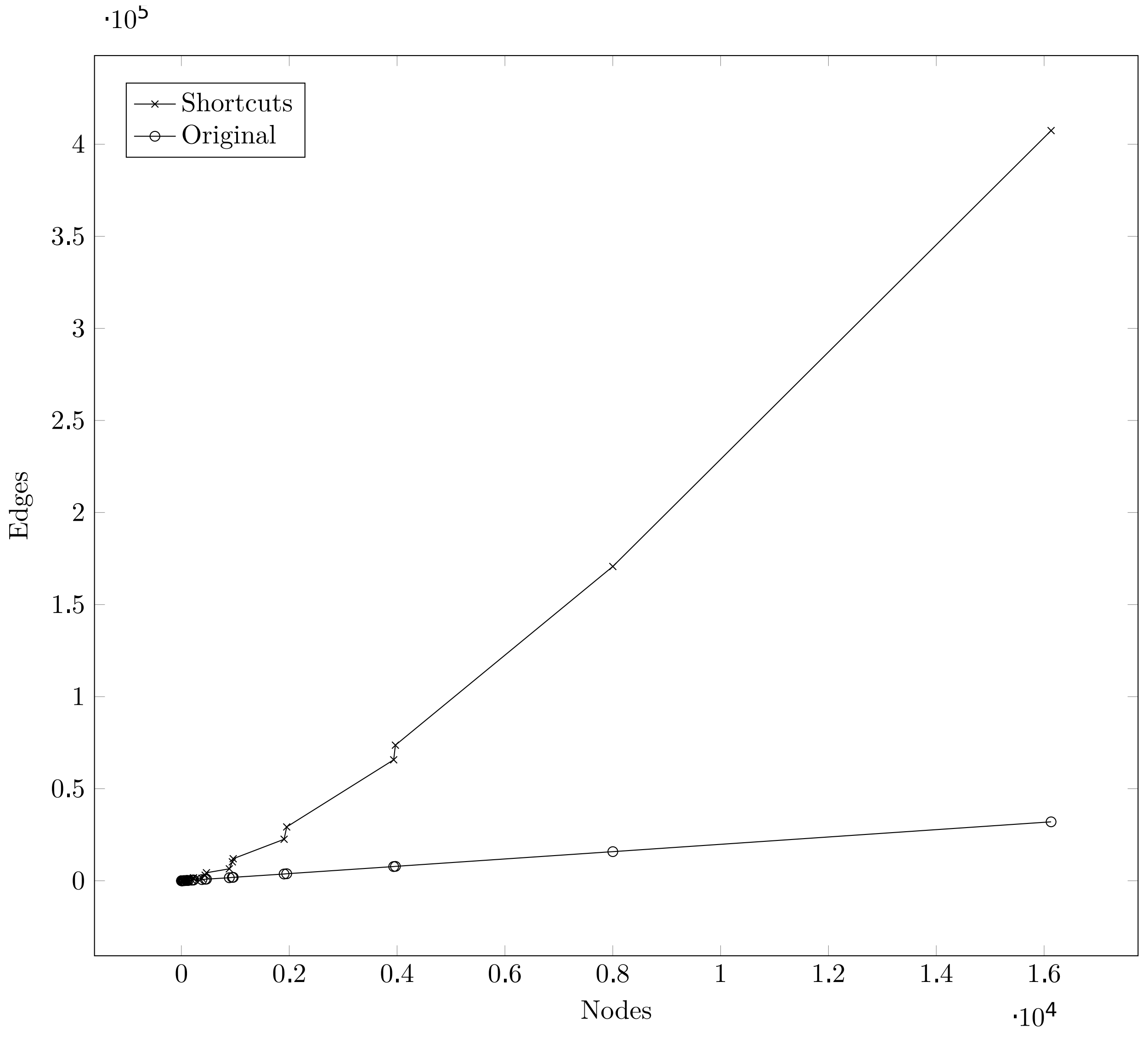

Figure 3.16: Amount of shortcut edges by nodes in grid graphs

4. Customization

In this chapter, we discuss the process of applying a metric to our preprocessed data. This process is denoted with *customization*. The idea to split preprocessing into a metric independent and a metric dependant phase was originally developed in [DGPW11]. Also, the term customization was introduced there. But the idea is not entirely new. [GSSV12] mentions an approach to incorporate small changes to an existing metric without rerunning the whole contraction. The node order is kept and a subset of possibly affected nodes is identified. For these the contraction is run again. We will explore, how this idea is applicable to Weak Contraction Hierarchies. The first section covers the introduction of an arbitrary metric, the second the incorporation of small changes. The performance of these algorithms is crucial to the customizable route planning problem. Contrary to the contraction process, they might be run quite often, probably even multiple times an hour to incorporate real time traffic changes. Their performance is one of our key priorities.

4.1 Basic Approach

In Chapter 3 we discuss the idea of Contraction Hierarchies and a metric independent version of the contraction routine, the witnessless contraction. The result of customization should be equivalent to the contraction hierarchy we would get if we applied the traditional contraction process (but without witness search) to the graph with its metric. So whatever we remove from contraction to be metric independent, we have to make up for in customization.

We start with the results of the witnessless contraction: A graph $G = (V, A)$, an order α, a weak contraction hierarchy G_α. We have to introduce the metric len_G which defines lengths for all arcs in A. Customization includes three things:

First of all, we must calculate the lengths of the shortcuts inserted by contraction. Second, we also have to shorten already existing arcs in A if necessary. Third, we need the information, which node caused the insertion (or update) of an arc length. We will refer to this node as the *supporting node* $sup(u, w)$. (If an arc is no shortcut and was not updated at all, sup will be some null value.) We need this information for some optimizations, and to unpack shortcuts and to turn a shortest path in the extended graph into a shortest path in the original graph.

Algorithm 4.1 shows a pseudocode variant of the emerging algorithm. Since it carries out the parts we removed in metric independent contraction, it looks quite similar to the contraction algorithm.

Algorithm 4.1: Customization

```
Input: Graph $G = (V, A)$, order $\alpha$, metric $len_G$, weak contraction
       hierarchy $G_\alpha = (G_\alpha^\wedge, G_\alpha^\vee)$
Output: Metric $len_\alpha$
1 $len_\alpha \leftarrow len_G$
2 for each $v \in V$ ordered ascending by $\alpha(v)$ do
3     for each $u \in V$ with $(u, v) \in A_\alpha^\vee$ do
4         for each $w \in V$ with $(v, w) \in A_\alpha^\wedge$ do
5             if $len(u, w)_\alpha > len(u, v)_\alpha + len(v, w)_\alpha$ then
6                 $len(u, w)_\alpha \leftarrow len(u, v)_\alpha + len(v, w)_\alpha$
7                 $sup(u, w) \leftarrow v$
8 return $len_\alpha$
```

4.1.1 Microinstructions and Macroinstructions

One may note that the algorithm's operations on the given data are rather simple. Looking up three arc weights, calculating a sum of two of those and possibly assigning an arc weight and a supporting node. This leads to the idea of implementing customization on a simpler data structure than a graph. This idea was proposed in [DW13] under the name of *microinstructions*.

Rather than having a graph data structure, one can store all arc lengths in an array. During contraction, a list of triples is generated. Each triple represents one step of the customization and consists of three arc indices. To execute the step, all three arc lengths must be looked up. Then the sum of the first two is calculated and assigned to the third arc's length if it yields an improvement. This allows to avoid all unnecessary overhead of a complicated graph structure.

Although the approach yields significant speedups for customization, it raises two issues for us. First, for large graphs it produces an enormous amount of data (the number of triples is within $\mathcal{O}(n^3)$). If the triple data exceeds the capabilities of the main memory, the whole process slows down extremely due to the I/O. The second problem is that we have no information about the node of which contraction caused the insertion (or the update) of a shortcut. Of course, we could simply store this information in between the microinstructions but that would only increase the already gigantic amount of microcode further. Due to the immense amount of generated microcode we discarded this approach – the experiments in Chapter 6 use only macrocode.

To reduce the size of the instructions, we can use *macroinstructions*. Instead of listing all triples, one can represent the contraction of a node by listing the indices of all incoming arcs, all outgoing arcs and for each pair of those the index of the potential shortcut arc. Iterating over these pairs of arcs takes a little longer but is still reasonably fast. And because we are simulating the contraction of a certain node, the supporting node of all updated arcs can be set to the current node. We will have a closer look at the implementation in Chapter 6.

4.2 Incremental Metric Updates

In this section we are going to explain the algorithms used to incorporate incremental changes to a metric. A typical use case for this would be the occurrence of a traffic jam, which increases the travel time for a certain road. Of course, one could rerun the whole customization, but especially considering metrics like travel times, such small changes may occur quite often. Therefore, they should be as fast as possible. Even though customization is only a matter of some seconds, it may be worth the effort to accelerate small changes.

We will only consider changes to the metric, but not the deletion or introduction of new arcs, since that would force us to rerun parts of the contraction. In [GSSV12], Geisberger et al. explored the possibilities of introducing small structural changes to a graph without rerunning the whole contraction. It is probably possible to combine the approaches and extend these algorithms to handle new or deleted arcs as well, but that is beyond the scope of this work.

The general idea is to limit customization to the parts of the graph which might actually change. An important thing we should note is that a change to an arc's weight can only affect arcs "above" (with respect to the node order) the changed arc. Another important observation is that the customization of a node v (the two inner loops in algorithm 4.1) does not consider arcs coming from or going to nodes with a lower rank than $\alpha(v)$. This leads us to the conclusion that if we change the length of an arc (u, v), the set of directly affected arcs is a subset of the set of arcs which is affected if we contract the lower ranked node of the both endpoints of (u, v).

When an arc length changes, we have to distinguish between two cases: Either the new length is larger or smaller than the current length. The current length may differ from the one defined in the metric due to shortcuts. If the new length is shorter than the current, we can apply the new weight at once, remove the supporting node if present and propagate the change to other arcs. For that we can reuse the customization process (and the macroinstructions) and apply it to the lower of the two nodes of the changed arc. Since the arc was shortened, shortcuts can only become shorter, too. So if another arc changes, we can repeat the process on that arc. Since probably multiple arcs may be affected, we maintain a priority queue with the lower ranked endpoint of the changed arcs and recustomize the nodes in ascending order of their ranks. Algorithm 4.2 shows the resulting algorithm.

One may note that we do not need to repeat the whole customization of the node. Since only the arc (a, b) is changed, we merely need to reconsider shortcuts containing that arc. As a result, we can drop either the inner or the outer for-each-loop, depending on whether (a, b) is in $A_\alpha^\vee$ or in $A_\alpha^\wedge$. But this would prevent us from reusing the macroinstructions. Thus we would need a graph data structure rather than an array of weights. So this optimization would most likely have a negative impact to running time, and because of that we execute the whole customization of each node.

If the arc length gets longer, things are a little more complicated. Although length changes only affect higher arcs, still lower arcs might affect the changed one. For example, if the changed arc is a shortcut (including original arcs with updated length), we can safely ignore the change since the shortcut length remains valid. If not, still shortcut paths which are longer than the old length but shorter than the new length might exist. So we must reconsider all possible shortcut paths for the changed arc. If the arc length changed after those checks, we must propagate these changes upwards. For that, we can also use the customization algorithm (and the macroinstructions) but we have to modify it slightly. If we encounter any shortcut with a supporting node different from the one we are customizing right now, we can safely leave the arc as it is. The shortcut path over the other node was shorter before and the shortcut path via the current node only got longer. When encountering a shortcut arc caused by the current node, we mark it for a possible length increase and go on. The marked arcs are stored into a priority queue and extracted in ascending order of their lower node. When we extract an arc from the priority queue, we repeat the whole process as before. That is resetting the length to the original one given by

Algorithm 4.2: Shorten Arc

```
Input: Graph $G = (V, A, len_\alpha)$, order $\alpha$, base metric $len_G$, weak contraction
       hierarchy $G_\alpha = (G_\alpha^\wedge, G_\alpha^\vee)$, changing (original) arc $(a, b) \in A$, new length $l$
       such that $l < len_\alpha(a, b)$
Data: Priority queue Q
Output: Updated metric $len_\alpha$

1  $len_G(a, b) \leftarrow l$
2  $len_\alpha(a, b) \leftarrow l$
3  $sup(a, b) \leftarrow$ null
4  if $\alpha(a) < \alpha(b)$ then
5      Q.insert($a$)
6  else
7      Q.insert($b$)
8  while Q is not empty do
9      $v \leftarrow$ Q.extractMin
10     for each $u \in V$ with $(u, v) \in A_\alpha^\vee$ do
11         for each $w \in V$ with $(v, w) \in A_\alpha^\wedge$ do
12             if $len_\alpha(u, w) > len_\alpha(u, v) + len_\alpha(v, w)$ then
13                 $len_\alpha(u, w) \leftarrow len_\alpha(u, v) + len_\alpha(v, w)$
14                 $sup(u, w) \leftarrow v$
15                 Q.insert($\arg\min_{x \in \{u,w\}} \alpha(x)$)
16 return $len_\alpha$
```

the metric, reconsidering all potential shortcut paths, picking the shortest (or the original length if it is shorter than any shortcut) and propagate the change.

When considering practical implementations, some further modifications are necessary. The algorithm as stated above, stores arcs in the priority queue and extracts them ordered by their lower node. But since one node has many arcs, that might lead to the problem that the process is repeated several times on the same node. To avoid this problem, we actually maintain two queues. One only stores the lower nodes of each arc with raised length. For these nodes, we invalidate all shortcuts caused by this node. The second queue contains all nodes which lie on potential shortcut paths for the changed arcs. After the first queue is empty, we apply the customization process to each of the nodes in the second queue in ascending order of their ranks. Algorithm 4.4 shows a pseudocode outline. Actually lines five and six are in a typical contraction hierarchy implementation non trivial to implement. Normally one only stores the arcs going to higher ranked nodes or coming from higher ranked nodes. But in line five we are iterating over arcs going to a lower ranked node. Luckily, we can obtain this list of arcs to lower ranked nodes in advance when we read in the graph (by iterating over all arcs in $A_\alpha^\vee$ and storing them additionally at the starting node).

Algorithm 4.3: Slow Lengthen Arc

```
Input: Graph $G = (V, A, len_\alpha)$, order $\alpha$, base metric $len_G$, weak contraction
       hierarchy $G_\alpha = (G_\alpha^\wedge, G_\alpha^\vee)$, changing (original) arc $(a, b) \in A$, new length $l$
       such that $l > len_\alpha(a, b)$
Data: Priority queue Q
Output: Updated metric $len_\alpha$
1  $len_G(a, b) \leftarrow l$
2  $len_\alpha(a, b) \leftarrow l$
3  $sup(a, b) \leftarrow$ null
4  Q.insert($(a, b)$)
5  while Q is not empty do
6      $(x, y) \leftarrow$ Q.extractMin
7      for each $v \in V$ with $(x, v) \in G_\alpha^\vee$ and $(v, y) \in G_\alpha^\wedge$ do
8          execute-macro-code ($v$)
9      $v \leftarrow \arg\min_{x \in \{a,b\}} \alpha(x)$
10     for each $u \in V$ with $(u, v) \in A_\alpha^\vee$ do
11         for each $w \in V$ with $(v, w) \in A_\alpha^\wedge$ do
12             if $sup(u, w) = v$ then
13                 $len_\alpha(u, w) \leftarrow len_G(u, w)$
14                 $sup(u, w) \leftarrow$ null
15                 Q.insert($(u, w)$)
16 return $len_\alpha$
```

Algorithm 4.4: Efficient Lengthen Arc

Input: Graph $G = (V, A, len_\alpha)$, order α, base metric len_G, weak contraction hierarchy $G_\alpha = (G_\alpha^\wedge, G_\alpha^\vee)$, changing (original) arc $(a, b) \in A$, new length l such that $l > len_\alpha(a, b)$

Data: Priority queues $\mathsf{Q}_{toCustomize}$ and $\mathsf{Q}_{toInvalidate}$

Output: Updated metric len_α

```
1  $len_G(a,b) \leftarrow l$
2  $len_\alpha(a,b) \leftarrow l$
3  $sup(a,b) \leftarrow$ null
4  $\mathsf{Q}_{toInvalidate}$.insert($\arg\min_{x \in \{a,b\}} \alpha(x)$)
5  for each $(a,v) \in A_\alpha^\vee$ do
6      $\mathsf{Q}_{toCustomize}$.insert($v$)
7  while $\mathsf{Q}_{toInvalidate}$ is not empty do
8      $v \leftarrow \mathsf{Q}_{toInvalidate}$.extractMin
9      for each $(u,w) \in A_\alpha^\wedge \cup A_\alpha^\vee$ with $sup(u,w) = v$ do
10         $len_\alpha(u,w) \leftarrow len_G(u,w)$
11         $sup(u,w) \leftarrow$ null
12         $\mathsf{Q}_{toInvalidate}$.insert($\arg\min_{x \in \{u,w\}} \alpha(x)$)
13         for each $(u,x) \in A_\alpha^\vee$ do
14             $\mathsf{Q}_{toCustomize}$.insert($x$)
15 while $\mathsf{Q}_{toCustomize}$ is not empty do
16     $v \leftarrow \mathsf{Q}_{toCustomize}$.extractMin
17     customize ($v$)
18 return $len_\alpha$
```

5. Query

In this chapter, we discuss query algorithms. The algorithms are based on Dijkstra's algorithm but use additional data gathered from contraction and customization to accelerate the search. First, we are going to take a look at the basic original Contraction Hierarchy query algorithm as introduced by Geisberger et al. in [GSSD08]. We are not going to consider any acceleration techniques. In particular, stall-on-demand is not covered (and not implemented) in this work. Second, we are going to introduce a different algorithm special for weak contraction hierarchies which can not be applied to traditional contraction hierarchies. The idea behind this algorithm was originally developed by Tobias Columbus but never published. This chapter delivers the theoretical ideas behind the algorithms and a few theoretical results on their performance. An extensive experimental evaluation is given in in Chapter 6.

5.1 Basic Traditional Contraction Hierarchy Query

In Chapter 3, the motivation for constructing Contraction Hierarchies is to be able to remove unimportant nodes from the search space while preserving distances. We can use this for an efficient query algorithm by removing all nodes with lower ranks than s or t from the search space. Since they were contracted during preprocessing, shortest paths containing them were replaced with shortcuts. To limit the query to nodes higher than s or t, we can use the bidirectional variant of Dijkstra's algorithm on the graphs of the contraction hierarchy. Algorithm 5.2 shows the procedure (as it was proposed in [GSSV12]) in pseudocode.

The algorithm differs in two points from the bidirectional variant of Dijkstras's algorithm. The first is that the two searches do not run on the whole graph but only on parts of it. The forward search contains only arcs to higher ranked nodes, the backward search space only arcs from higher ranked nodes. If we reverse the backward search space, all arcs point upwards, too. The paths the algorithm finds,[1] can be split into two parts of which the first consists only of upwards and the second only of downwards arcs. The second difference is the stopping criterion. Instead of comparing the length of the shortest found path to the sum of the minimum keys, the length is compared to the minimum key of both queues. To prove its correctness, we would have to prove two things: First, there is a shortest path in the search space which can be split up into an up and a down part, and second, the

[1]As before, we can obtain paths by storing a predecessor for each node.

Algorithm 5.1: Basic Contraction Hierarchy Query

```
   Input: Contraction hierarchy $\bar{G}_\alpha = (\bar{G}_\alpha^\wedge, \bar{G}_\alpha^\vee)$, metric $len_\alpha$, order $\alpha$
   Data: Priority queues $\mathsf{Q}_s, \mathsf{Q}_t$
   Output: $dist(s,t)$
 1 Run two Dijkstra instances, one from $s$ on $\bar{G}_\alpha^\wedge$ and one from $t$ on $\bar{G}_\alpha^\vee$ (with reversed
   arcs).
   // Main loop
 2 while $\mathsf{Q}_s \cup \mathsf{Q}_t \neq \emptyset$ and $\min_{v \in V} \mathsf{d}_s(v) + \mathsf{d}_t(v) > \min\{\min_{u \in \mathsf{Q}_s} \mathsf{d}_s(u), \min_{w \in \mathsf{Q}_t} \mathsf{d}_t(w)\}$
   do
 3     Perform a single step of one instance with a non empty queue
 4 return $\min_{v \in V} \mathsf{d}_s(v) + \mathsf{d}_t(v)$
```

algorithm finds that (or one of those) path(s). For both, very extensive proofs can be found in [Col12].

When trying to bound the algorithm's running time, one notices that it depends mainly on the number of nodes and arcs inside the search space. Sadly, it is difficult to prove any bound on these numbers when using a traditional Contraction Hierarchy order. In [AFGW10], certain guarantees on the performance for graphs with low highway dimensions were given. Although it seems reasonable that road networks have a low highway dimension and experimental results support this idea, it is very difficult to compute the actual highway dimension of a concrete graph. On the other hand, node orders such as nested dissection allow proofs on the search space size for graphs with bounded separator sizes. The discussion of these results is part of the next section.

5.2 Weak Contraction Hierarchy Query

Applying the basic Contraction Hierarchy query to a weak contraction hierarchy, is always a valid approach. Any weak contraction hierarchy contains at least all the arcs of the algorithmic contraction hierarchy. The search space gets only additional arcs, the distances stay valid. The problem is that these additional arcs may slow down the query significantly. And in the weak contraction hierarchy we get from the witnessless contraction (the maximum weak contraction hierarchy), there are quite a lot of these additional arcs. The weak query algorithm aims to reduce the overhead of those arcs.

The algorithm uses the same search space as the traditional query but obtains it before the actual distance computation using the elimination tree. Since we need to run a witnessless contraction to obtain the elimination tree, the algorithm is not applicable to contraction hierarchies based on traditional node orders (see Section 6.2.2). Theorem 3.8 guarantees that the traditional search space of a node is included in the path from the node to the root of the elimination tree. By ascending in the tree, starting from s and t, we can compute the search space. Afterwards we can limit the graph to the nodes in the search space and run the bidirectional variant of Dijkstra's algorithm.

Tobias Columbus developed (but never published) an idea to reduce the overhead of the additional shortcuts by utilizing the availability of the search space during the actual search. It aims to temporarily deactivate shortcuts which are not necessary to preserve distances. We recall from Chapter 3 that shortcuts are inserted to preserve distances from shortest paths via nodes removed from the search space. That means, if a node is included in the search space, we can deactivate all shortcuts caused by this node. During customization we store this information. And contrary to the basic query algorithm, we obtain the complete

Algorithm 5.2: WEAK CONTRACTION HIERARCHY QUERY

```
  Input: Graph with shortcuts $G^\alpha = (V, A^\alpha, len_\alpha)$, order $\alpha$, elimination
         tree $T = (V, A_T)$
  Output: $dist(s, t)$
1 $S \leftarrow \{s, t\}$
2 for each $v \in \{s, t\}$ do
3     while $\exists$parent$_T(v)$ do
4         $S \leftarrow P \cup \{$parent$_T(v)\}$
5         $v \leftarrow$ parent$_T(v)$
6 return BiDijkstra($(V \cap S, A^\alpha, len_\alpha)$)
```

search space before the actual query. With this information, we can deactivate unnecessary arcs.

The bigger problem is to technically deactivate the arcs per query in a way that they can be skipped during arc iteration in sublinear time. For this, we order each node's arcs by their supporting node and generate an additional array containing the count of arcs with the same supporting node. Utilizing this additional information, we can jump one block of arcs with common supporting node in constant time.

5.2.1 Partial Contraction and Customization

One approach to accelerate contraction and customization of Contraction Hierarchies is to apply the process only to a part of the graph. In the case of a nested dissection order, one could try to omit the top level separator due to its influence on performance. This approach was already tried out on traditional Contraction Hierarchies. The query algorithm has to switch to the original graph as soon as it reaches the uncontracted nodes. Research has shown that this has significant negative impact on the performance of the query [BDS+08]. But for Weak Contraction Hierarchies, the approach might be worth reconsidering. Since we are already performing a bidirectional Dijkstra's algorithm on the graph, we only need to include the uncontracted nodes in the search space of every query. And there are nodes which are in the search space of almost every query, anyway. Take for example the nodes of the top level separator. Thus contracting (and customizing) them, is probably an unnecessary overhead. We evaluate in Chapter 6 whether this approach yields any significant improvements.

5.3 Performance

As proved in Section 3.8 the height of the elimination tree and the search space size are closely related to each other. In [Col12], these results are used to apply upper bounds found in studies of the elimination game to search space size. The following theorem was stated and proven in [Col12] in a slightly different variant. As it is stated here, it leads us more directly to the conclusions we want to draw.

Theorem 5.1. *Let $G = (V, A)$ be a graph of a class of graphs which admits small balanced node separators such that the separator size is in $\mathcal{O}(\sqrt{n})$ and each of the node subsets has less nodes than $b * n$ with $0.5 < b < 1$. Furthermore, let α be a node order based on a nested dissection utilizing such separators, $G_\alpha = (G_\alpha^\wedge, G_\alpha^\vee)$ a contraction hierarchy and $T = T(*G, \alpha)$ an elimination tree. The maximum number of nodes inside a search space $\max_{v \in V} |\mathcal{S}(v, T)|$ (and for that, both parts of the contraction hierarchy, too) is in $\mathcal{O}(\sqrt{n})$.*

Proof. First let us recall that in a separator based node order, nodes from the separator always have a higher rank than nodes from both halves of that separator (3.1). That means that in a graph with only upward arcs it is impossible to find a path from one side of a separator to the other. Since no nodes of both sides are adjacent to each other, all potential paths would lead trough the separator but the nodes from the separator have higher ranks than the nodes in the halves.

Recall that the elimination tree, the upward part of the contraction hierarchy and the reversed downward part contain only upward arcs. So a node's search space is quite limited. It may contain all nodes in the node's final cell (we limited this number to an arbitrary constant n_0 in definition 3.9), their separator, the next higher separator and so on up to the root separator. The root separator's maximum size is $\mathcal{O}(\sqrt{n})$. The next lower separator has a maximum size of $\mathcal{O}(\sqrt{b * n})$, the following one $\mathcal{O}(\sqrt{b^2 * n})$ and so on. With h as the maximum depth of the nested dissection, the search size thus is limited by the term

$$n_0 + \sum_{h=i}^{h} \sqrt{b^i * n} = n_0 + \sqrt{n} * \sum_{h=i}^{h} \sqrt{b}^i$$

If we let h go to infinity, we can use the geometric sum formula

$$n_0 + \sqrt{n} * \sum_{h=i}^{\infty} \sqrt{b}^i = n_0 + \frac{\sqrt{n}}{1 - \sqrt{b}} \in \mathcal{O}(\sqrt{n})$$

□

We can use this result to give performance bounds for the query algorithms when run with a nested dissection order. Due to the inserted shortcuts, we must assume that the graph is quite dense. So the running time of Dijkstra's algorithm is in $\mathcal{O}(n_{\mathcal{S}} \log n_{\mathcal{S}} + m_{\mathcal{S}})$ (with $n_{\mathcal{S}}$ denoting the number of nodes and $m_{\mathcal{S}}$ the number of arcs inside the search space). We consider $\mathcal{O}(\sqrt{n})$ nodes at most. In the basic query algorithm, all arcs we encounter during a query lead to nodes which are in the search space, too. Thus $m_{\mathcal{S}}$ is in $\mathcal{O}(\sqrt{n_{\mathcal{S}}}^2) = \mathcal{O}(n_{\mathcal{S}})$. So the overall running time of the basic query algorithm is in $\mathcal{O}(\sqrt{n} \log n + n) = \mathcal{O}(n)$.

In contrast to that, the weak query will encounter (but not relax) arcs to nodes outside the search space. Each node may have at most n arcs, so $m_{\mathcal{S}}$ is limited to $\mathcal{O}(n\sqrt{n})$ in this case. The total running time is then within $\mathcal{O}(\sqrt{n} \log n + n\sqrt{n}) = \mathcal{O}(n\sqrt{n})$ which is in theory actually slower than Dijkstra's algorithm on sparse graphs. Nevertheless, the experiments in Chapter 6 show that it still yields significant improvements over Dijkstra's algorithm.

6. Results

In this chapter, we are going to present an extensive experimental evaluation of the algorithms and methods discussed in the chapters before. The code is written in C++11 and was compiled with GCC 4.7.1 with flag -O3. We use the traditional Contraction Hierarchy preprocessing implementation and priority queues - we use 4-heaps - from a graph framework of the department this thesis was written at. All time critical experiments were run on a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. Some of the experiments which gather only statistics were run on the author's private computer. We ran all experiments single threaded. We split our experiments into several programs, but the running time never includes any IO - the times were taken as if the data structures were passed through the main memory. For queries, we ran 10 000 point-to-point queries (except for the plain algorithm of Dijkstra, there we ran only 1 000 due to time limitations) with start and end node each picked uniformly at random.

The graphs used throughout these experiments were made available for the 9th DIMACS Implementation Challenge [DGJ09, dim09]. We used three graphs from the USA; New York (NY), Florida (FLA) and the Western USA (W). Our main testing instance is the DIMACS Europe graph (EU) which was not available online anymore at the time of this writing. It was originally made available by PTV AG. For each of the graphs, we use both travel times and distance metrics. Since there was no distance metric available for the Europe graph at the department, we calculated one based on the node's coordinates. Table 6.1 summarizes their properties. The USA graphs are undirected in the sense that for each forward arc (u, v) there also exists an arc (v, u) with the same length.

As mentioned before, the Weak Contraction Hierarchy and the traditional Contraction Hierarchy approach have quite a lot of similarities. In fact, it is possible to mix the approaches. For example, one can run the basic query algorithm on any weak contraction hierarchy. In this "methodspace", we basically have three parameters: the order, the contraction algorithm and the query algorithm. We explored the different combinations as far as possible but it turns out that some of them do not work well together.

The experiments are structured as the work before. We start with a short section on nested dissection orders. Afterwords we explore contraction, customization and queries in turn.

Table 6.1: Test graphs

				separators	
instance	directed	nodes	arcs	top level	maximum
New York (NY)	no	264 346	730 100	5	47
Florida (FLA)	no	1 070 376	2 687 902	46	46
Western USA (W)	no	6 262 104	15 119 284	87	87
Europe (EU)	yes	18 010 173	42 188 664	428	428

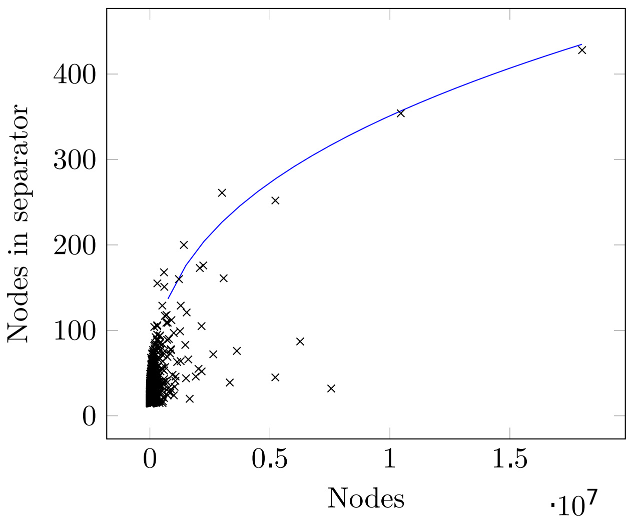

Figure 6.1: Separator size found by METIS against nodes in the graph. The input contains the separators from all our test graphs and also recursively includes their subgraph's separators. Separators smaller than fifteen nodes were omitted. The line is the function $n^{4/11}$.

6.1 Nested Dissection Order

We use the general purpose partitioner METIS [met13] in version 5.1.0 to obtain the nested dissection orders. METIS has the program *ndmetis* exactly for that purpose. Although there are partitioners available (for example PUNCH [DGRW11] or KaHip [SS13]) which are known to deliver smaller separators on road networks by using natural cuts, we decided to limit our research to the orders generated by METIS. Incorporating these other tools was unfortunately not possible due to the time limits for this thesis. The investigation of the impact of other orders with smaller separators is probably a topic worth more research and a promising approach to further optimizing Weak Contraction Hierarchies. However, METIS delivers reasonable small separators. Figure 6.1 shows the development of the separator sizes. They grow slower than $\sqrt{n}$ (the plotted function is $n^{4/11}$). The very small top level separator of the New York graph most likely cuts Manhattan out of the rest of New York. Note that the largest separators of New York and Florida are of about the same size. In the following experiments, we can often see that the performance of the algorithms on these two graphs is very similar.

METIS finds these orders quite quickly. The longest run on the Europe graph takes less than three minutes. Since the metric independent preprocessing time is no key priority to us, optimizing the quality of the order for the cost of more preprocessing time might be an option. METIS has some parameters which would allow us to do so, but as mentioned before, this is beyond the scope of this thesis.

Table 6.2: Contraction by nested dissection order. The weak contraction hierarchy is computed on a directed graph representation, the elimination tree on an undirected one. The undirected representation is more space efficient as each edge needs only to be stored once (versus twice for the directed representation). That causes some speedup even on the actual undirected USA graphs. One could theoretically join both algorithms for undirected graphs, but real world road networks are most unlikely undirected. We include the original number of (undirected) edges for comparison.

	weak ch		elimination tree		
	time [s]	shortcut arcs	time [s]	shortcut edges	original edges
NY	1.26	1 940 762	0.90	970 381	365 050
FLA	3.03	4 388 056	2.23	2 194 028	1 343 951
W	21.48	26 268 682	15.36	13 134 341	7 559 642
EU	176.69	88 621 755	157.87	47 680 211	21 094 332

6.2 Contraction

In this section, we cover the result of the contraction process both in the metric dependent and the metric independent variant.

6.2.1 Metric Independent Contraction

During the contraction, we have to obtain three things: the weak contraction hierarchy, the elimination tree and the macroinstructions for the customization. Constructing the weak contraction hierarchy and the elimination tree works very similar manner. In both cases, we have to contract the graph, except for that we have to use the undirected graph for the elimination tree. For contraction, we need a dynamic graph data structure which allows us to create and delete arcs (or edges). Furthermore, we need both, efficient iteration over the arcs of a node and efficient random access to the arcs. For that reason, and as the graph is unweighted in this stage, we use a different graph data structure than for query and customization. Instead of one large adjacency array for all nodes as used later for query and customization, each node has a set storing the adjacent nodes (two sets in the directed case, one for incoming arcs, one for outgoing). We used arrays for these sets in the beginning, but as more nodes get contracted and the graph grows denser, the arrays soon grow impractical. To overcome this problem, we use hash sets.[1] Table 6.2 shows our results. The contraction on an undirected graph is somewhat faster as we only need to store each edge once. So we could omit the directed contraction if the input graph is undirected but as it is very unlikely to encounter a undirected graph in real world data, we run it anyway. This is also the reason why the USA graphs have exactly half as many shortcut edges as they have shortcut arcs in the directed contraction. For a directed graph, there are slightly more shortcut edges (one edge - two arcs) than there are shortcut arcs as is shown in the Europe graph.

In Section 3.5, we observe that separators form a clique during contraction and that the final clique may have significant influence on the performance of contraction. This behaviour can also be observed on our test graphs. The following example is from contraction of the Europe graph. Figure 6.2 shows that the last few nodes have an extremely high degree. In

[1] We also tried switching over to a matrix based representation when the graph is sufficiently dense, but hash sets solved the problem much better.

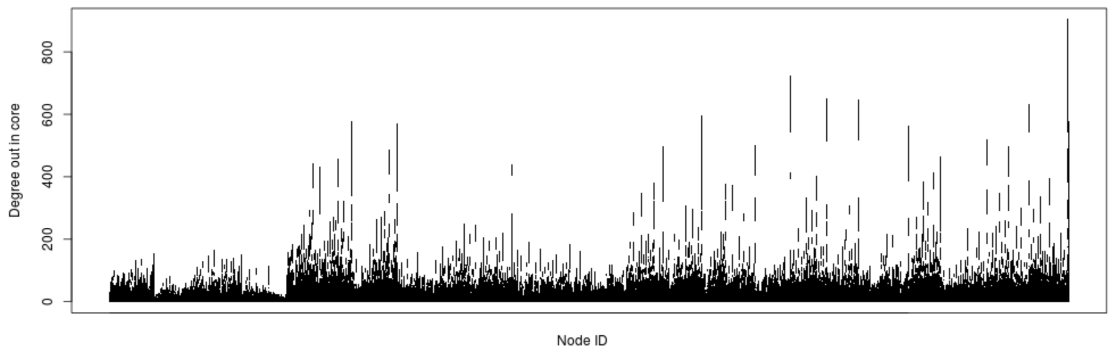

Figure 6.2: The development of the current node's number of outgoing arcs considering only arcs to nodes not yet contracted. As the current node is the lowest uncontracted, these arcs will always go to higher ranked nodes and thus be part of the resulting contraction hierarchy. The instance is the Europe graph. IDs increase from left to right. The sheer amount of nodes (18M) (and the printing resolution limits), often lets appear multiple nodes as on one line, but in fact, each node has a different point.

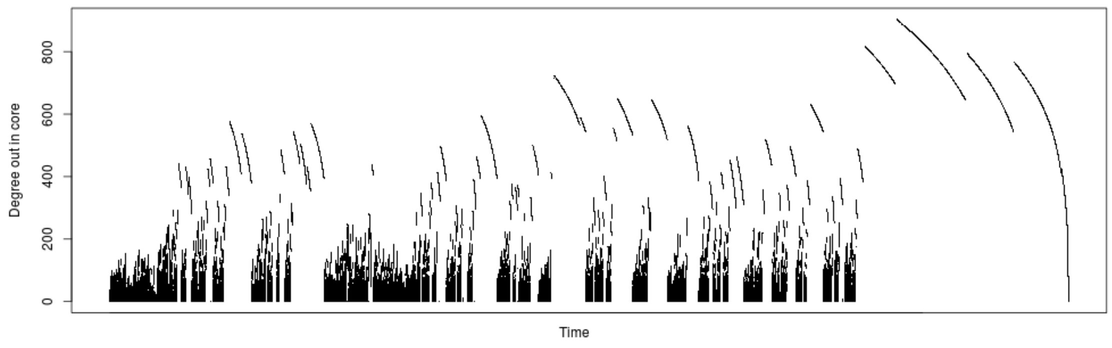

Figure 6.3: The same experiment as in Figure 6.2 but plotted against time (taken after each node was contracted) instead of node IDs. Demonstrates the influence on performance of the last few nodes. The last bow shows the final clique including the top level separator.

a traditional Contraction Hierarchy, the degrees stay about the same during contraction except for the last very few nodes. But the density increase is never as strong as for Weak Contraction Hierarchies. If we take Figure 6.3 into consideration, we can observe that the contraction of these last about 1 000 to 2 000 nodes takes about one fifth of the whole contraction time.

After the actual contraction, we still need to obtain the macroinstructions. For that, we need to build up the graph data structure so we can find out the arc indices for the instructions. The graph data structure consists of two static adjacency arrays, one for outgoing and one for incoming arcs. After contraction, we know all arcs which will exist in the graph. So the graph has no need for any dynamic operations beyond arc length adjustments. The costly part of the generation of the instructions is to find out the actual indices for each arc. We have to iterate through all the arcs of one node, and especially for high ranked nodes the graph is quite dense (the final clique on Europe has more than 500 nodes. Recall Section 3.5 – the final clique may be composed of up to three of the topmost separators).

We also have to consider that the graph is different, depending on which query we plan to use later. If we want to use the weak query, each node has to know all incident arcs including those to lower ranked nodes. If we are using a basic query, we only need arcs

Table 6.3: Metric independent preprocessing results. Split by the targeted query algorithm. Shows the time and space necessary for the macrocode. The total columns show time for the whole preprocessing including computation of the order and buildup of necessary data structures. Both queries require the weak contraction hierarchy, but only the weak query needs the elimination tree.

	weak query			basic query		
	macro generation		total	macro generation		total
	time [s]	space [MB]	prepro. [s]	time [s]	space [MB]	prepro. [s]
NY	7.3	163	9.63	2.38	88	3.75
FLA	7.44	237	13.21	4.54	136	7.9
W	69.17	2 042	109.30	37.01	1 124	60.43
EU	3 221.66	20 139	3 717.16	430.48	10 388	613.14

to higher ranked nodes. This reduces the number of arcs we need to store by 2 and thus also the amount of generated macrocode. One could reduce the amount of necessary macrocode for the weak query, too, by executing the customization only on forward arcs and copying over the weights to the backward arcs after the customization is finished. This could be an improvement for future implementations. Another improvement could be to compress the macrocode. Especially for Europe, a huge amount of macrocode is generated. Simply zipping the macrocode reduces its size (for both querys) to about 60%. Table 6.3 shows an overview over the different results. It also sums up the total metric independent preprocessing times (including times not mentioned explicitly before e.g. the graph data structure building time and the time for the generation of the node order). The macro generation for the weak queries takes on Europe by far more than twice the time than the generation for the basic query. Most likely, the reason is that for the weak query for each step both forward and backward arc indices must be obtained. Thus the execution is very cache inefficient.

6.2.2 Metric Dependent Contraction

For the metric dependent contraction we use the Contraction Hierarchy implementation of the department this thesis was written at. This implementation performs a witness search and obtains its node order on the fly. For that it requires a given metric. Still, it would be interesting to combine this metric dependent contraction with the nested dissection order. Sadly, the on-the-fly computed order is a core feature of this implementation. We had no opportunity to replace it with a nested dissection order and examine how they would perform together. Investigating this would be an interesting topic for future research on Weak Contraction Hierarchies.

But the other way around is interesting, too. We can obtain the node order of a traditional contraction by contracting the graph and extracting the order afterwards and trying to use it for the weak contraction. But this attempt fails miserably. Even on our smallest test graph (NY), the witnessless algorithm is not quite done after one full hour. The traditional order works quite fine for the first 90% of the nodes, but afterwards the graph gets very dense and the contraction slows down extremely. As this is quite unsatisfactory, we tried again on an even smaller graph - a road network of Luxembourg with 30 087 nodes and 69 433 arcs. Even on this graph, the witnessless contraction with traditional order takes two orders of magnitude longer (about 2.5s compared to 0.05s) than the witnessless contraction with the nested dissection order. With the nested dissection order, 112 694 shortcuts are

Table 6.4: Traditional contraction results. A metric is required and the node order is computed on the fly. The implementation was taken from a graph framework of the department this thesis was written at.

	travel times		distances	
	time [s]	shortcut arcs	time [s]	shortcut arcs
NY	6.03	1 035 593	7.65	1 092 825
FLA	9.78	2 559 495	12.33	2 732 097
W	79.99	15 248 138	122.27	16 005 214
EU	457.53	40 743 209	7 355.26	52 102 939

inserted, with the traditional order 437 508. So using traditional orders for witnessless contraction on larger graphs, will lead to some serious performance problems.

We still can combine a traditional order with a traditional contraction (the metric dependent contraction process with witness search) for comparison with our methods. Table 6.4 shows the results. Admittedly, the framework's implementation is based on a general purpose graph and may be not as purposefully optimized as our own. Additionally, the traditional contraction process does not only contract the graph, but computes the order, too. As discussed in [GSSV12], obtaining the order is actually the more complicated part. We definitely observe that the traditional Contraction Hierarchy slows down significantly when applied to distance metrics.

6.3 Customization

In this section, we are going to examine the performance of the customization algorithms. Table 6.5 contains all the data. The results are split up into a part for the weak query and a part for the basic query. Besides from operating on the full graph instead of only the upward arcs, the customization for the weak query algorithm must also generate additional information and reorder the arcs of the graph. The arcs must be ordered by their supporting node and the meta information which counts the arcs with the same supporting node must be gathered. This information is necessary for the efficient arc deactivation (see Chapter 5). The time for these additional computations is noted in the apply columns.

Furthermore, we measured times for each algorithm for both the full customization and incremental updates. For the incremental updates we (successively) picked 1 000 arcs a at random and assigned a new random weight out of the range 1 to $2len(a)$. The most important thing we can observe on this data is that the process is mostly metric independent (contrary to the traditional Contraction Hierarchy). With maximal up to 3 seconds for incremental updates, it is definitely fast enough to incorporate real time traffic updates.

Surprisingly, for all graphs except Europe, the customization time for the weak query is faster than the one for the basic query, although the customization for the basic query only has to execute about half the macrocode. The reason is that in the basic query case we additionally must decide whether the potential shortcut arc is in the forward arc array or in the backward arc array.

6.4 Query

6.4.1 Search Space Properties

In Section 5.3, we discussed the bounds for the search space size found in [Col12]. However, the impact of these bounds for the asymptotic running time is limited as we have to take

Table 6.5: Customization results. Split up by the targeted query algorithm.

		weak query				basic query	
		full customization		incremental update			
	metric	custom. [s]	apply [s]	custom. [ms]	apply [ms]	full custom. [s]	incremental update [ms]
NY	time	0.12	0.18	6.102	0.775	0.16	5.153
	distance	0.12	0.18	5.194	0.755	0.16	5.228
FLA	time	0.21	0.47	4.210	0.405	0.26	4.798
	distance	0.21	0.48	3.949	0.456	0.26	5.066
W	time	1.59	2.79	70.134	5.517	1.95	58.876
	distance	1.57	2.81	68.762	5.210	1.94	57.828
EU	time	19.64	35.46	2 434.890	283.402	16.23	1 681.712
	distance	19.27	26.61	2 210.980	264.578	16.43	1 485.355

the density of the graph after contraction into account. In our first query experiment, we investigated the relationship between the search space size and the number of arcs inside the search space. Figure 6.4 shows the development of the number of arcs with both endpoints inside the search space. These are the arcs the basic query might potentially encounter. But most likely, the basic query will encounter even less arcs. The chart gives the numbers for the joined search space but it is unlikely that the search space of s and t each consists of that complete search space. In most cases there will be some parts of the search space which belong either to the search space of s or t but not in both. In those parts, the query will encounter only either upward or downward arcs.

Furthermore, we observe that the search space seems to be quite dense. About 2 million arcs for 2 000 nodes is as half as many as a clique would have. Nevertheless, the growth does not seem to be quadratical. The reason for this is probably that the higher ranked and therefore denser parts of the graphs are included in most search spaces, the top level separator in almost every one. So the search space can only grow to lower ranked nodes with smaller degrees. Figure 6.5, which additionally includes the arcs which leave the search space, shows almost a linear correlation between the number of arcs and nodes. The weak query algorithm might encounter all these arcs.

It is interesting to note that there are about as many arcs to nodes outside the search space as there are to nodes within. Actually, the weak query algorithm has the intention to reduce the number of arcs which are considered during the search. For that it deactivates unnecessary shortcuts. Figure 6.6 shows the number of arcs we can omit from the search (in red) since their supporting node is part of the search space. Unfortunately, those are even less than the additional arcs going out of the search space. Especially for small search spaces, the number of unnecessary arcs is very small. The reason for that may be again that a small search space consists mostly of very high ranked nodes. The shortcuts between those are most likely caused by nodes with lower ranks. This already gives us a strong hint that the weak query might not perform as well as the basic query. Nevertheless, we are going to examine some details of the algorithm in the next section.

If we draw the conclusion from these charts that the number of arcs encountered by the query algorithm grows linear with the number of nodes in the search space, we can conclude much better asymptotic running times for both query algorithms. If we assume $m_{\mathcal{S}} \in \mathcal{O}(n_{\mathcal{S}})$, and apply these values to the running time of Dijkstra's algorithm, both algorithms have an asymptotic running time in $\mathcal{O}(\sqrt{n}\log n)$.

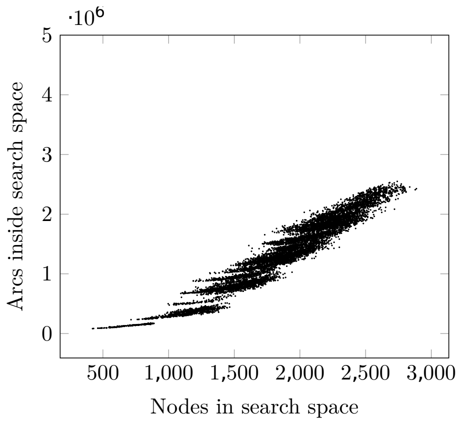

Figure 6.4: Number of arcs with both endpoints in the search space by nodes inside the search space for 10 000 random query search spaces on Europe

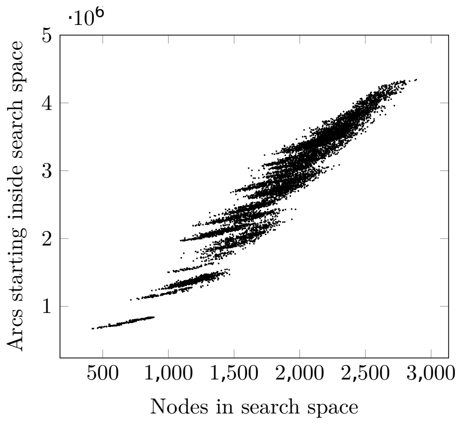

Figure 6.5: Number of arcs with at least one endpoint in the search space by nodes inside the search space for 10 000 random query search spaces on Europe

Table 6.6: Weak query details with different distance data structures (hashmap, distance array, distance array with timestamp) and with or without arc deactivation.

arc deactivation	metric	hash time [ms]	array time [ms]	timestamp time [ms]	settled nodes	relaxed arcs
no	time	16.751	4.732	4.467	1 074.6	603 712.7
	distance	18.902	4.915	4.567	1 164.8	695 710.1
yes	time	9.787	4.910	4.410	1 074.6	212 335.3
	distance	11.593	5.433	5.158	1 164.8	268 334.6

6.4.2 Weak Contraction Hierarchy Query Details

During the development of the weak query algorithm we evaluated several internal details. Table 6.6 shows the results. One particularly interesting option is the choice of the data structure to store the distances. For implementations of Dijkstra's algorithm, typically an array containing the distances is maintained. To avoid reinitializing the whole array for each query, one stores an additional timestamp for each distance. When setting a distance, the timestamp is set to the current time. When accessing the distance and the timestamp is not equal to the current time, an infinity value is returned. To clear the array, the timestamp is incremented. The column timestamp contains the results for this approach. Since the Contraction Hierarchy query visits only very few nodes, it might be more efficient to avoid the timestamp comparisons and reset the visited nodes distances after each query by hand (column array). Finally, one can store the distances in a hash data structure and clear that hash after each query (column hash). The other important decision is whether to deactivate unnecessary arcs or not. Doing so, adds some overhead to the iteration of a node's arcs since one has to take the additional array with the block information into account.

It turns out that the arc deactivation significantly reduces the number of relaxed arcs. But the same nodes are settled (which confirms that the shortcuts are truly unnecessary). Anyway, the impact to performance is limited except in the case of distances in a hashmap. The cause for this is that the distance lookups on the hashmap are very expensive compared to just looking up an value inside an array. The less arcs are relaxed, the less distance

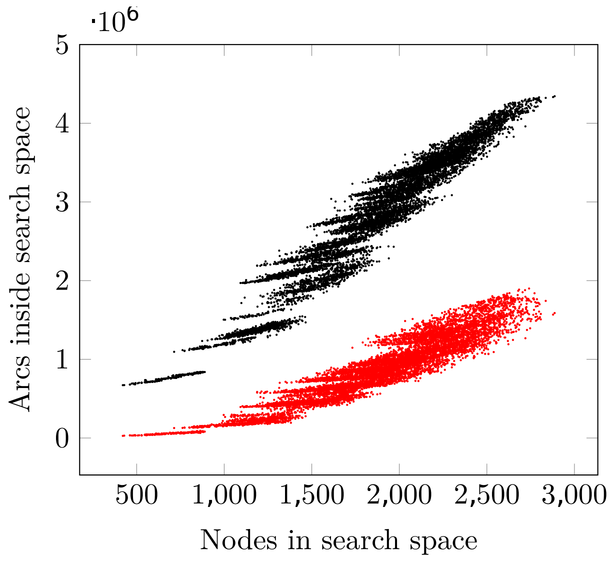

Figure 6.6: Total amount of arcs vs. arcs which can be deactivated (red) by the number of nodes in the query search space on Europe

lookups are necessary. But due to the expensiveness of the lookups, the hashmap is no competitor for the other two. The arc deactivation actually has a small negative impact on most of the other cases. A closer analysis of the arc deactivation data reveals that in the average case there are only 2.3 arcs in a block with the same supporting vertex. That makes the approach somewhat ineffective. In a future implementation, one could try to gather the arc deactivation information only for sufficiently large blocks. The timestamp variant is only slightly superior to the plain array. In all cases, the algorithms perform somewhat slightly worse on the distance metric. For the following experiments, we always use the timestamp variant with arc deactivation enabled.

In Section 5.2.1, we discuss the possibility to contract and customize only parts of the graph. This option was explored before for traditional Contraction Hierarchies in [BDS$^+$08]. Contracting only 95% of the graph shortened the preprocessing by about factor 2 but slowed down the queries by about two orders of magnitude. We explore partial preprocessing on a somewhat smaller scale. Table 6.7 shows the results. We start by omitting only the top level separator and continue by stepwise doubling the size of the uncontracted core. It turns out that the impact of such small uncontracted areas is very limited for both preprocessing and queries. Still, for queries it is larger than what we hoped for. Preprocessing has too many parts (like generating the macroinstructions and generating the arc deactivation information) which are not influenced as strongly as contraction by the final dense parts of the graph. Nevertheless, we can accelerate preprocessing down to two thirds of the original time by omitting only about 7 000 nodes without slowing down queries by more than 2ms. The customization time for 428 uncontracted nodes seems to be a heavy outlier. A closer look reveals that the actual customization took as long as expected but the generation of the arc deactivation information took about 20s less than expected. A reason may be that by chance the graph's arcs were in a very good constellation for the generation of the arc deactivation information. If we repeat the experiment, e.g. with 429 uncontracted notes, the applying time is still too fast but only about 10s. We note that the query slowdown is very small between the three largest uncontracted cores. Weak Contraction Hierarchies seem to be more robust to uncontracted areas in the graph than traditional Contraction Hierarchies. We conclude that it would be probably worth the effort to investigate the behaviour of the Weak Contraction Hierarchy query with even larger uncontracted cores. Still all times (contraction, customization and query) are respectively slower than the ones

Table 6.7: Partial contraction for different numbers of uncontracted nodes on the Europe graph with time metrics. Contraction times including times for computing the weak contraction hierarchy, the elimination tree and the macrocode. We use the weak query so customization times include the actual metric introduction and the generation of the information for arc deactivation. The query uses timestamps and arc deactivation.

	contraction		customization	weak query		
uncontracted nodes	time [s]	macros [MB]	time [s]	time [ms]	settled nodes	relaxed arcs
0	3 565.17	20 139	55.10	4.410	1 074.6	212 335.3
428	3 214.26	19 943	25.81	4.667	1 072.8	233 862.8
856	2 840.56	18 786	50.04	5.097	1 109.4	309 078.1
1 712	1 911.11	16 146	44.21	5.932	1 310.0	419 165.0
3 424	1 915.02	16 145	43.00	5.947	1 342.4	413 661.1
6 848	1 932.48	16 145	39.67	6.109	1 380.2	414 443.3

achieved, when using the basic query algorithm. As the graph for the traditional query stores only upward arcs, macrocode generation time and customization time are much faster than what we can achieve with partial contraction.

6.4.3 Comparison to Traditional Contraction Hierarchies

In this section, we summarize the results on the performance of the query algorithms and how they relate to traditional Contraction Hierarchies. As we mentioned in the beginning of this chapter, we can combine several parts of weak and traditional Contraction Hierarchies. We had to omit several combinations because either they did not work out together or we had no possibility to implement them. The combinations we can actually run are a traditional order with a metric dependent contraction and the basic query and a nested dissection order with a witnessless contraction and both queries. All the numbers can be found in Table 6.8. One may note that the query times for the traditional Contraction Hierarchy are quite a bit slower than the ones presented by Geisberger et al. in [GSSV12]. The implementation of Geisberger et al. has several optimizations, not implemented in the basic query algorithm. The most important one of them is stall-on-demand. Furthermore, the node order used for the best results by Geisberger et al. was obtained using extreme parameter tuning which we did not do at all.

The first thing to note on our own algorithms is that the traditional query outperforms the weak query by more than factor 2 on the larges graph and even larger factors on the smaller graphs. Thus, it is the better choice for the graphs we considered in terms of both query and customization time. This confirms our presumptions made from the observation of the amount of arcs in the search space (see Section 6.4.1). Still, it would be interesting to explore whether the weak query algorithm might become faster on even larger graphs. As observed before, the traditional Contraction Hierarchies perform poorly on distance metrics. The Weak Contraction Hierarchies perform slightly worse on the distance metric, too. But the drawbacks are by far not as strong as for the traditional approach. It is really surprising to see that the basic query on the weak contraction hierarchy was actually faster than the basic query on the traditional contraction hierarchy even on the time metric, too. A closer look reveals that the basic query on the traditional order settles more nodes than the basic query on the nested dissection order but relaxes less arcs. This behaviour, especially with optimizations like stall-on-demand, should be examined closer in future research.

Table 6.8: Comparison of different query algorithms on different graphs and metrics. Weak Contraction Hierarchy uses a nested dissection order and witnessless contraction, the traditional Contraction Hierarchy a traditional order and a metric dependent contraction. Dijkstra's algorithm is run on the original graph.

		Weak CH			
	metric	weak query time [ms]	basic query time [ms]	trad. CH time [ms]	Dijkstra time [ms]
NY	time	0.238	0.055	0.130	13.570
	distance	0.274	0.054	0.148	11.355
FLA	time	0.238	0.050	0.101	75.017
	distance	0.257	0.051	0.151	58.310
W	time	0.943	0.180	0.467	517.808
	distance	1.065	0.195	0.809	401.412
EU	time	4.410	1.922	2.638	1 783.155
	distance	5.158	2.050	15.820	1 275.642

In the real world, most queries are local. To investigate the performance of our algorithms for queries of different ranges, we compute Dijkstra ranks. Figure 6.7 and Figure 6.8 show the results. To construct the ranks, we run Dijkstra's algorithm on 10 000 randomly picked source nodes and store the first, the second, the fourth and so on (all powers of two) node the algorithm settles. We then run queries on these pairs of nodes and sort the results by the rank of the target node.

The plots reveal some interesting facts. All queries with ranks lower 2^{19} have average query times of less then 1ms. Most of the time, the algorithms perform very well on local queries. But on the other hand, both algorithms have heavy outliers with very slow queries for almost every rank. As both charts use the same scale, the weak chart shows these outliers better, but they are present in the basic query chart, too. Because of the nested dissection order, nodes close to each other might lie on different sides of an high ranked separator. In that case, the search space is very large despite the locality of the query.

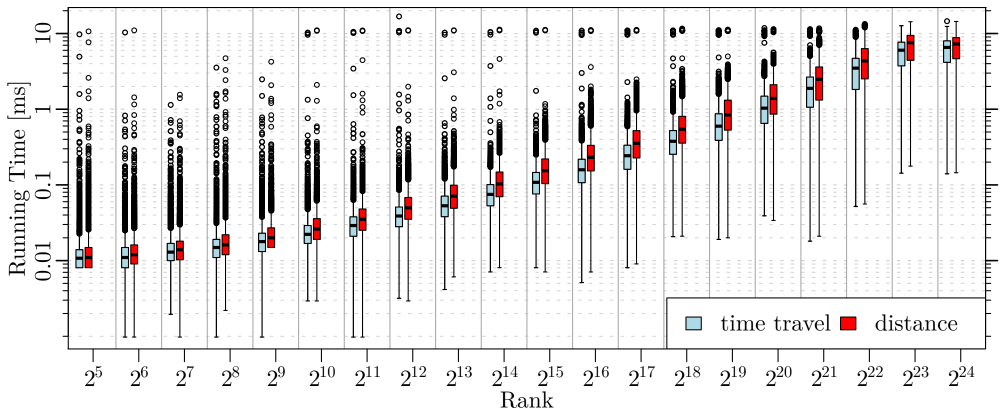

Figure 6.7: Dijkstra ranks for the weak query algorithm (with timestamps and arc deactivation) on a nested dissection order on Europe with travel time metric

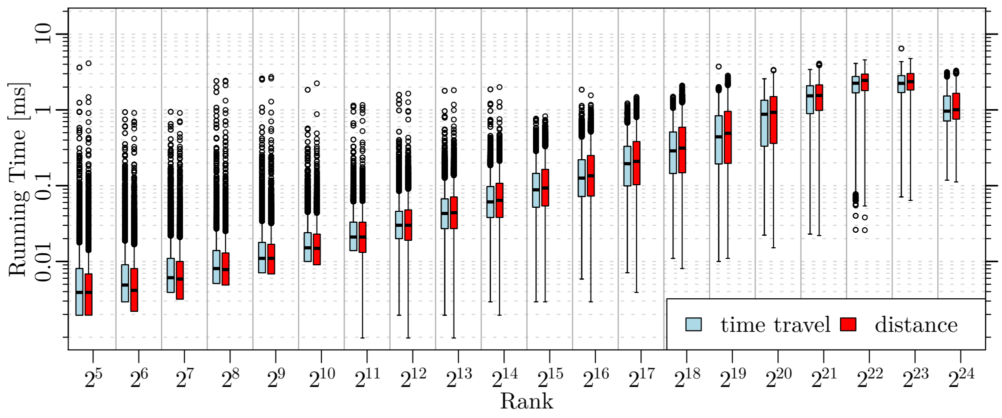

Figure 6.8: Dijkstra ranks for the basic query algorithm on a nested dissection order on Europe with travel time metric

7. Summary

Our Contribution

In this work, we apply the separation of metric independent and metric dependent preprocessing proposed in [DGPW13] to Contraction Hierarchies. For this purpose, we use the theoretical results of [Col12] in which a connection between Contraction Hierarchies and the well studied elimination game was observed. In Section 3.3 we recapitulate these theoretic foundations. Regarding that link, we apply nested dissection orders to Contraction Hierarchies. We analyze the behaviour of contraction based on nested dissection orders on grid graphs in Section 3.5. In Chapter 3, 4 and 5, we discuss the Contraction Hierarchy algorithms and their adaptions to Weak Contraction Hierarchies. Additionally, we picked up a previously unpublished idea by Tobias Columbus to prune the arcs in the search space. We documented and refined the approach in Section 5.2. All these algorithms were implemented and evaluated in Chapter 6.

The algorithms which emerged from the studies can incorporate arbitrary metrics on continental sized road networks in less than one minute. Applying changes to these metrics is a matter of seconds. The query algorithms run in very few milliseconds and are thus applicable for real time scenarios. Despite their power, conceptually they are still very simple algorithms.

Future Work

Still, there is space for improvements. Recent research brought up specialized partitioners for road networks which could be used for the nested dissection orders. One could engineer the algorithms further. A lot of the optimizations for traditional Contraction Hierarchies could be carried over to Weak Contraction Hierarchies. And additionally, as nested dissection splits the graph into several unconnected subgraphs, especially parallelism could deliver great improvements.

Another interesting future research might be the application of local modifications of the node order as proposed in [Col12]. We observe that the search space of Weak Contraction Hierarchies is significantly less dense than feared. That suggests that it might be possible to prove tighter worst case bounds for the query running time. And as nested dissection orders worked quite well in this thesis, it might also be interesting to combine them with other approaches such as Hublabeling.

Conclusion

Weak Contraction Hierarchies work – that is the title of this thesis and its most important result. Utilizing a nested dissection order, we can contract continental sized road networks without witness search and achieve reasonable results which even outperform existing node orders on less well-behaved metrics such as distance metrics. All in all, this work incorporates recent theoretical results on Contraction Hierarchies to deliver a set of simple algorithms to make Contraction Hierarchies customizable and thus better applicable to real world scenarios.

Bibliography

[ADF$^+$11] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. VC-Dimension and Shortest Path Algorithms. In *Proceedings of the 38th International Colloquium on Automata, Languages, and Programming (ICALP'11)*, volume 6755 of *Lecture Notes in Computer Science*, pages 690–699. Springer, 2011.

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In Pardalos and Rebennack [PR11], pages 230–241.

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. Technical Report MSR-TR-2012-46, Microsoft Research, 2012.

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

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

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

[BCRW13] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-Space Size in Contraction Hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13)*, volume 7965 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2013.

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

[BDS$^+$08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In McGeoch [McG08], pages 303–318.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. In Demetrescu et al. [DGJ09], pages 175–192.

[BGHK92] Hans L Bodlaender, John R Gilbert, Hjálmtỳr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. In *Graph-Theoretic Concepts in Computer Science*, pages 1–12. Springer, 1992.

[BJ92] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-Hop Labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[Col12] Tobias Columbus. Search Space Size in Contraction Hierarchies. Master's thesis, Karlsruhe Institute of Technology, October 2012.

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

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In Pardalos and Rebennack [PR11], pages 376–387.

[DGPW13] Daniel Delling, Andrew Vladislav Goldberg, Thomas Pajor, and Renato Fonseca Werneck. Customizable route planning in road networks. In *Sixth Annual Symposium on Combinatorial Search*, 2013.

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011.

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

[dim09] 9th DIMACS Implementation Challenge - Shortest Paths. `http://www.dis.uniroma1.it/challenge9/download.shtml`, 2009. [Online; accessed 4-Septmber-2013].

[DW13] Daniel Delling and Renato F. Werneck. Faster Customization of Road Networks. In SEA'13 [SEA13], pages 30–42.

[For56] Lester R. Ford, Jr. Network Flow Theory. Technical Report P-923, Rand Corporation, Santa Monica, California, 1956.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

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

[GKW06] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 129–143. SIAM, 2006.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch [McG08], pages 319–333.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

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

[LT79] Richard J. Lipton and Robert E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, April 1979.

[McG08] Catherine C. McGeoch, editor. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*. Springer, June 2008.

[met13] METIS. `http://glaros.dtc.umn.edu/gkhome/views/metis`, 2013. [Online; accessed 4-Septmber-2013].

[Moo59] Edward F Moore. *The shortest path through a maze*. Bell Telephone System., 1959.

[PR11] Panos M. Pardalos and Steffen Rebennack, editors. *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*. Springer, 2011.

[SEA13] *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*. Springer, 2013.

[Som12] Christian Sommer. Shortest-Path Queries in Static Networks, 2012. Submitted. Preprint available at `http://www.sommer.jp/spq-survey.htm`.

[SS06] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.

[SS13] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In SEA'13 [SEA13], pages 164–175.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.