Institut für Formale Methoden der Informatik

Abteilung Algorithmik

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Bachelorarbeit Nr. 265

Improved approximation schemes for the restricted shortest path problem

David Holzmüller

Studiengang:	Informatik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Prof. Dr. Stefan Funke
begonnen am:	09.11.2015
beendet am:	04.02.2016
CR-Klassifikation:	F.2.2

Improved approximation schemes for the restricted shortest path problem

David Holzmüller

January 30, 2016

Contents

Abstract

In this thesis we present several variants of an approximation scheme for the restricted shortest path problem. This problem concerns finding a shortest path with respect to one criterion while not exceeding a length bound with respect to another criterion. After formally introducing the problem, we describe the approximation scheme by Lorenz and Raz [3]. Our algorithm then introduces additional steps to reduce the runtime complexity from $\mathcal{O}(|E|n(1/\varepsilon + \log\log n))$ to $\mathcal{O}(|E|n(1/\varepsilon + \log\log\log n))$. A variant of this algorithm yields a complexity of $\mathcal{O}(|E|n(1/\varepsilon + (n\log n)/m))$, which is a further improvement for sufficiently dense graphs. Furthermore, a slight modification leads to an $\mathcal{O}(|E|n(1/\varepsilon + 1))$-algorithm for directed acyclic graphs, providing an arguably easier algorithm than the algorithm proposed by Ergun et al. [1].

1 Introduction

The Restricted Shortest Path (RSP) problem concerns bicriteria path optimization. It only optimizes one criterion, but the set of valid paths is restricted by a second criterion. RSP is NP-complete, but it has a fully polynomial time approximation scheme (FPTAS) [3], i. e. there is an algorithm finding a $(1+\varepsilon)$-approximation to the optimum whose time complexity is polynomial in both the size of the input and $1/\varepsilon$. The current best result known to us is the algorithm proposed by Lorenz and Raz [3] with a time complexity of $\mathcal{O}(nm(1/\varepsilon + \log\log n))$, where n is the number of vertices and m the number of edges of the given graph. For directed acyclic graphs, Ergun et al. [1] published an algorithm with time complexity $\mathcal{O}(nm(1/\varepsilon + 1))$. In this thesis, we want to outline the algorithm by Lorenz and Raz and introduce several techniques to reduce its time complexity, resulting in several variants. As a side effect, we equalize the result by Ergun et al. with a slightly different algorithm. Sections 2, 4 and 5 are mainly based on the paper by Lorenz and Raz [3], despite differing in notation and some details. They outline the problem, show an exact solution for integer-weighted graphs and an approximation scheme for RSP. However, the extended exact solver in section 4.2 and the graph G_S^- are not relevant for the algorithm by Lorenz and Raz — they are essential to our improved versions. Section 6 finally presents our algorithm with a detailed analysis. Section 3 gives a brief summary of the algorithms before they are explained in detail.

2 Problem Definition

The Restricted Shortest Path (RSP) problem can be defined as follows:

- Input: A directed graph $G = (V, E, c, r)$ with two weight functions $c, r : E \to \mathbb{R}_0^+$. In the following, $c(e)$ and $r(e)$ are also referred to as the cost and the resource consumption of an edge. Furthermore, start and end vertices $s, t \in V$ and a resource bound $R \in \mathbb{R}_0^+$. In the following, we set $n \coloneqq |V|$ and $m \coloneqq |E|$.

- In order to sum over all edges of a path easily, we will represent a path by the set of its edges. This representation does not allow edges to occur multiple times — however, since we want to find shortest paths, relevant paths are not affected. Furthermore, we want to introduce the set of all (acyclic) paths from s to an arbitrary vertex $v \in V$:

  $$P_v \coloneqq \{\{(v_1, v_2), \ldots, (v_{k-1}, v_k)\} \mid (v_1, \ldots, v_k) \text{ is a path from } s \text{ to } v\} .$$

  Then we can define the cost of a path $p \in P_v$ in G as

  $$C_G(p) \coloneqq \sum_{e \in p} c(e) \tag{2.1}$$

  and the resource consuption of p as

  $$R_G(p) \coloneqq \sum_{e \in p} r(e) . \tag{2.2}$$

- A solution to the RSP problem is a path $p \in P_t$ which does not exceed the resource bound R, i. e. $R_G(p) \leq R$. The goal is to find a solution p that minimizes $C_G(p)$. We refer to such a (potentially non-unique) optimal solution as $P_{\text{opt}}(G)$ and to the unique optimal cost $C_G(P_{\text{opt}}(G))$ as $C_{\text{opt}}(G)$.

3 Algorithm overview

We will now give a short overview over the algorithms and ideas presented in this thesis. In section 4, we will show that if all edges have positive integer cost, we can solve the RSP problem exactly using a simple dynamic programming scheme. Unfortunately, the runtime of the dynamic programming scheme depends linearly on $C_{\text{opt}}(G)$, which might in turn be exponential in the size of the input. To find an approximation for the optimal solution in reasonable time, we can (down-)scale (i. e. divide by a scaling factor S) and then round

up the cost values of G. We can then run the dynamic programming scheme on these modified cost values. This would lead to a $(1+\varepsilon)$-approximation in time $\mathcal{O}(nm(1/\varepsilon+1))$ if we knew an appropriate scaling factor. Unfortunately, we first need a constant approximation for $C_{\text{opt}}(G)$ to get such a scaling factor. By finding the minimal value c' such that a path p from s to t exists which satisfies $R_G(p) \leq R$ and $c(e) \leq c'$ for all $e \in p$, we obtain $L \coloneqq c'$ and $U \coloneqq nc'$ as lower and upper bounds for $C_{\text{opt}}(G)$. To tighten these bounds, we examine $\mathcal{O}(\log n)$ potential scaling factors S_i where $S_i/S_{i+1} = 2$. By running the dynamic programming scheme with a graph scaled by one of these S_i, we can examine whether S_i is too big, acceptable or too small (in the last case we have to stop the dynamic programming scheme after enough steps to obtain a suitable runtime bound). Using binary search on the scaling factors, we can then find an acceptable scaling factor in $\mathcal{O}(nm \cdot \log\log n)$. The algorithm by Lorenz and Raz uses this scaling factor to scale the graph and then run the dynamic programming scheme to get a suitable path.

Our main algorithm basically adds four ideas:

- Do a linear search on i to find an acceptable S_i instead of a binary search. At first, this sounds worse since we potentially have to examine more values for i. However, because we then only approach the target value from one direction, we can test the big scaling factors first. This means that we have small cost values and thus the computational effort needed to compute the exact solution is very low for most of the values of i examined. Ideally, doubling the scaling factor should mean that at most half of the effort is needed. This means we could analyze the algorithm backwards from the lowest investigated scaling factor, doubling the scaling factor in each round. The complexity calculation would then contain a geometric series meaning roughly that the runtime of the last iteration is an upper bound for the cumulative runtime of all other iterations.

- Unfortunately, doubling the scaling factor does not mean that the cost of the scaled graph halves. The reason is that we are essentially rounding up the scaled values to get integer values for the dynamic programming scheme. Instead, we again deviate from the algorithm by Lorenz and Raz by rounding down the scaled values. This modification ensures that the requirement above holds.

- Rounding down the scaled values solves a problem, but brings up another one. The dynamic programming scheme as used in the algorithm by Lorenz and Raz cannot cope with edge cost values that are zero. As Ergun et al. described, this is not a problem if the graph is acyclic.

To solve the problem for general graphs, we incorporate Dijkstra's algorithm into the dynamic programming scheme. This means that the m in the original complexity estimation is replaced by $m + n \log n$, but for graphs with many edges, this is already sufficient for an improved runtime.

- To compensate the $n \log n$ from Dijkstra's algorithm in sparse graphs, we may leave the $\mathcal{O}(\log \log n)$ most expensive values of i to be searched by the binary search algorithm already used by Lorenz and Raz. This is the reason for the $\log \log \log n$-term in the complexity of this variant of our algorithm.

4 Optimal Solution for the Non-Negative Integer Version

For graphs G where the cost function c only takes positive integer values, we can solve the RSP problem in $\mathcal{O}(m \cdot C_{\text{opt}}(G))$ time by a dynamic programming algorithm [2]. We will now outline this algorithm and show how it can be extended to handle edges with cost zero. We can assume that $m \in \Omega(n)$, since we can at first find the weakly connected component of s by using breadth-first search (this works in $\mathcal{O}(m)$ time) and only operate on the explored subgraph (since the equation $|E| \geq |V| - 1$ holds for weakly connected graphs).

4.1 Solving With Positive Integers

We define values $r_{k,v}, k \in \mathbb{Z}, v \in V$ by

$$r_{k,v} \coloneqq \min\{R_G(p) \mid p \in P_v, C_G(p) \leq k\} \ , \tag{4.1}$$

i. e. $r_{k,v}$ denotes the minimal resource consumption amongst all paths from s to v with cost at most k. We already know that no path with negative cost can exist. Thus, we can conclude $r_{k,v} = \infty$ if $k < 0$. Since a path with cost 0 and resource consumption 0 exists from the start node to itself, we also know that $r_{k,s} = 0$ for all $k \geq 0$. If we knew all values for $r_{k,v}$, we could compute $C_{\text{opt}}(G)$ using the formula

$$C_{\text{opt}}(G) = \min\{k \in \mathbb{N}_0 \mid r_{k,t} \leq R\} \ . \tag{4.2}$$

Additionally, we can find an optimal path $P_{\text{opt}}(G)$ in $\mathcal{O}(m)$ time by tracing back the cause for the value of $r_{C_{\text{opt}}(G),t}$.

How can we compute the remaining unknown values $r_{k,v}$, i. e. for $k \geq 0$ and $v \in V \setminus \{s\}$? Because a path from s to $v \neq s$ with cost $\leq k$ and minimal resource consumption must be composed of such an optimal path from s to a node v and an edge from v to v, the equation

$$r_{k,v} = \min_{e=(w,v)\in E} \Big(r_{k-c(e),w} + r(e) \Big) \tag{4.3}$$

holds for all $k \neq s$. If $c(e) \geq 1$ for all $e \in E$, this equation yields a simple dynamic programming scheme — we can simply compute all $r_{k,v}$ for $k = 0, 1, \ldots$ until $r_{k,t} \leq R$ (using a standard growing-array implementation to store these values). The runtime of this algorithm is $\mathcal{O}(m)$ for each row and thus $\mathcal{O}(m \cdot C_{\text{opt}}(G))$ in total. The cause for a value $r_{k,v}$ is an edge e which minimizes the equation above. These edges can be traversed from t to s to find out the optimal paths.

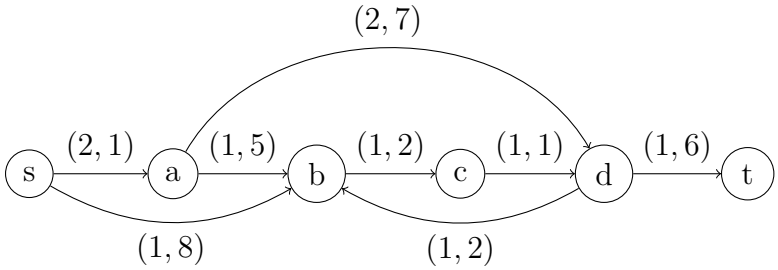

Figure 1: Example graph G_1 without zero-cost edges. Each edge e is labelled with $(c(e), r(e))$.

For illustration, Figure 1 shows a Graph G_1. Computing the values $r_{k,v}$ for G_1 corresponds to finding the cost of the shortest path to $r_{k,v}$ from $r_{k',s}$ for some $k' \geq 0$ in the infinite graph shown in Figure 2. The edge costs in the original graph determine how many levels one has to go down in the infinite graph when taking an edge. The weights of the edges are the corresponding resource consumptions.

4.2 Generalizing to Non-Negative Integers

If we allow $c(e) = 0$ for some edges e, a single row cannot be computed that easily. Here, we will use a variation of the dynamic programming approach explained above to generate a graph $G^{(k)}$. Using $G^{(k)}$, we can then formulate a single source shortest path (SSSP) problem whose solution entries correspond to the row entries we want to find.

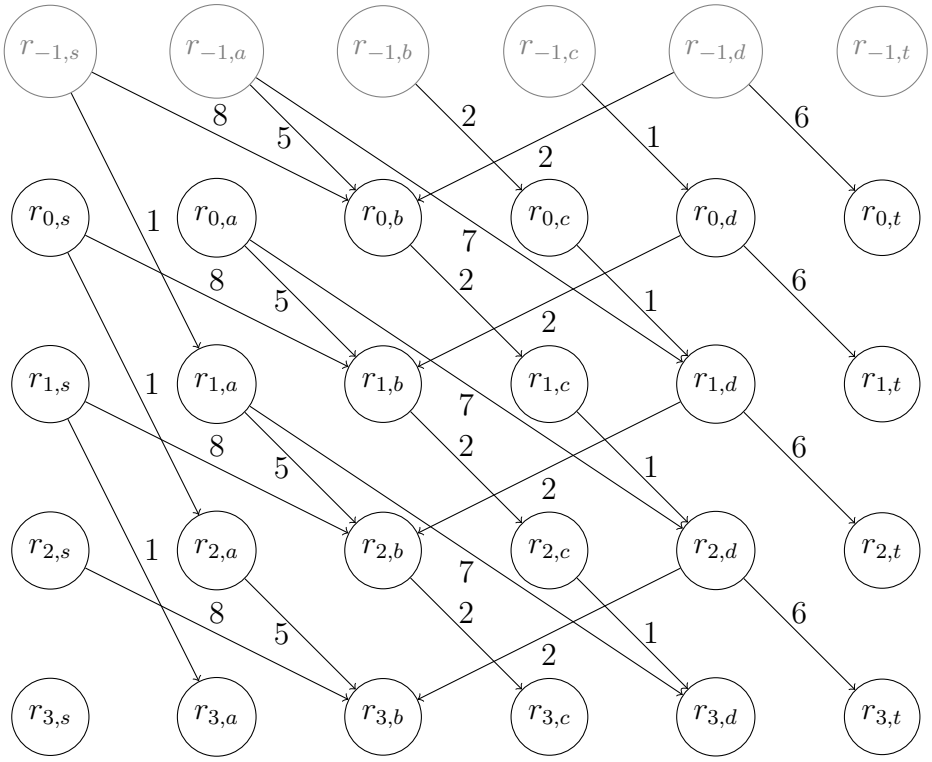

Figure 2: Section of the infinite graph corresponding to the dynamic programming scheme when applied to G_1.

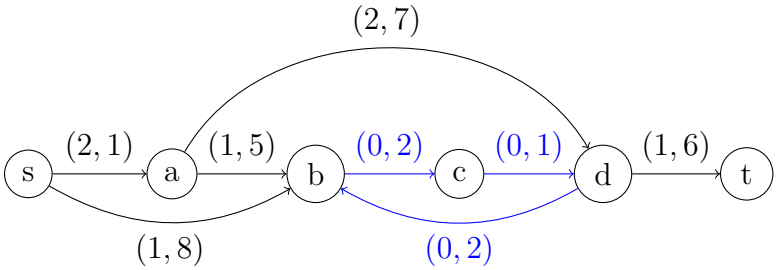

Figure 3: Example graph G_2 including zero-cost edges. Each edge e is labelled with $(c(e), r(e))$.

Figure 3 shows an example for a graph with zero-cost edges. This graph G_2 can again be transformed like G_1 to obtain an illustration for the structure of the dynamic programming algorithm. A section of the corresponding infinite graph is shown in Figure 4.

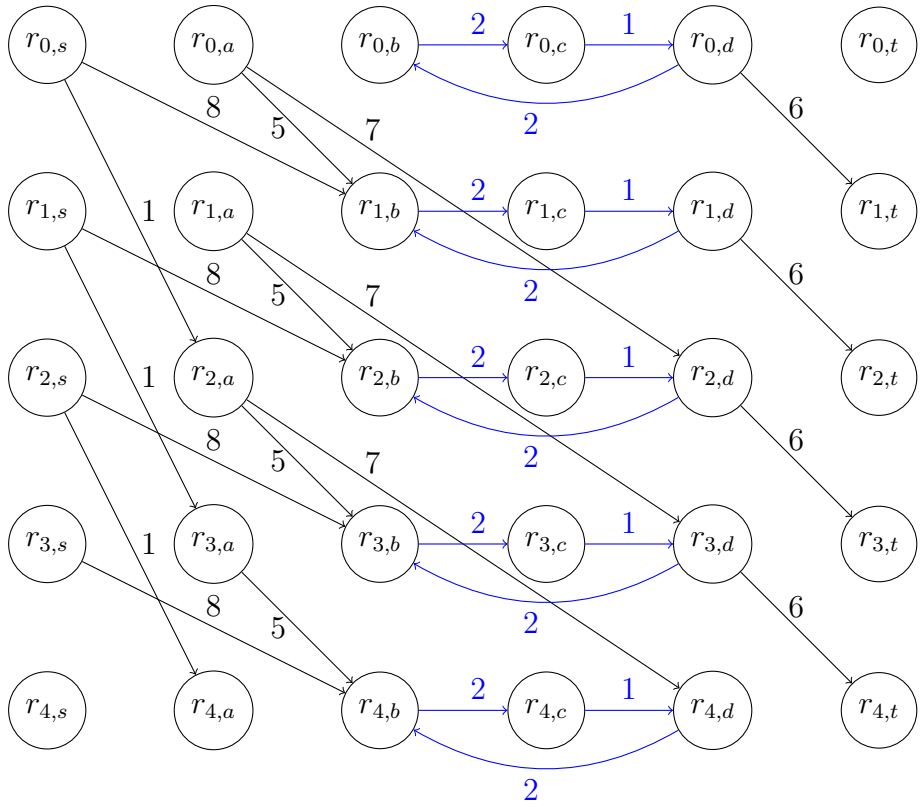

Figure 4: Section of the graph representing the dynamic programming scheme for the graph G_2 from Figure 3.

Since the values $r_{k,v}$ for fixed k might (in equation (4.3)) be dependent on each other, we cannot use a simple traversal of the values inside a row. To solve this problem, we introduce the weighted graphs $G^{(k)} = (V', E', \gamma)$, where $V' := V \cup \{v_s\}$ (v_s is a new vertex), $E' := \{e \in E \mid c(e) = 0\} \cup (\{v_s\} \times V)$ and

$$\gamma((a,b)) := \begin{cases} r((a,b)) & , (a,b) \in E \\ 0 & , (a,b) = (v_s, s) \\ \min\{r_{k-c(e),v} + r(e) \mid e = (v,b) \in E, c(e) > 0\} & , \text{otherwise} . \end{cases}$$

In our example, let us take a closer look at the situation where $k = 4$,

i. e. values for $k < 4$ have already somehow been computed. The already computed values for $r_{k,v}$ are shown inside the nodes in Figure 5. From this graph, we can easily derive the graph $G_2^{(4)}$ shown in Figure 6.

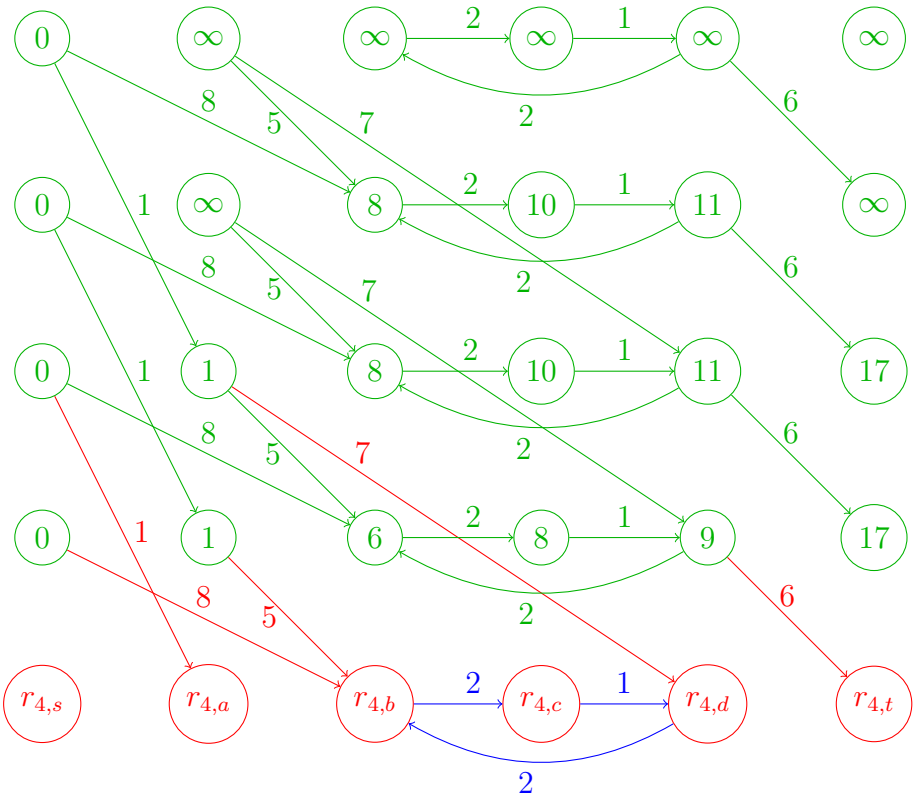

Figure 5: Partially computed values in the infinite graph corresponding to G_2. The edges which are relevant for building $G_2^{(4)}$ and thus for computing the next row entries are highlighted in red.

The weight of an edge (v_s, w) in $G^{(k)}$ is chosen to be the minimum resource consumption of all paths $p \in P_w$ whose cost is at most k *and whose last edge has non-zero cost*. Thus, for every path from v_s to a vertex u in $G^{(k)}$ with weight α, there is a corresponding path from s to u in G with resource consumption α and cost $\leq k$. This shows that $d_{G^{(k)}}(v_s, u) \geq r_{k,u}$, where $d_{G^{(k)}}(v_s, u)$ is defined as the minimum weight of any path from v_s to u in $G^{(k)}$.

Conversely, a path p from s to u in G with cost $\leq k$ and minimal resource consumption can be split into a first part ending at a vertex u' whose last edge is an edge with non-zero cost and a second part which only consists of zero-cost edges. The second part of the path also exists in $G^{(k)}$. By the

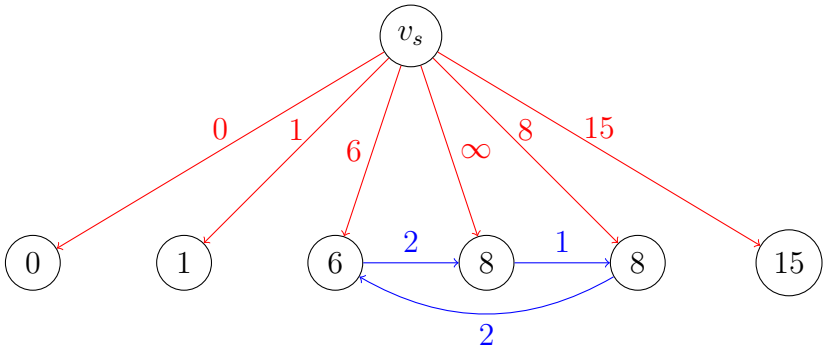

Figure 6: The graph $G_2^{(4)}$ and its shortest path weight entries corresponding to the values $r_{4,v}$.

optimality principle of shortest paths, the first part must itself have minimal resource consumption among all paths from s to u' with cost k. Therefore, the weight of the edge (v_s, u') in $G^{(k)}$ must be the resource consumption of the first part of p. But this shows $d_{G^{(k)}}(v_s, u) \leq r_{k,u}$ and thus $d_{G^{(k)}}(v_s, u) = r_{k,u}$.

Because of this, we can use Dijkstra's algorithm with a fibonacci heap to compute all $r_{k,v}$-values for a fixed k, i. e. a row of the dynamic programming table, in time $\mathcal{O}(m + n \log n)$. Consequently, the running time for the overall algorithm is now $\mathcal{O}((m + n \log n) \cdot (1 + C_{\text{opt}}(G)))$ (the 1 is necessary because $C_{\text{opt}}(G)$ might be zero). Note that we do not have to build the whole graph $G^{(k)}$ to run the algorithm although we would be able to do so without increasing the computational complexity of the overall algorithm.

If G is a directed acyclic graph, we can even solve the SSSP-problem in $\mathcal{O}(m)$ time. If G is an undirected graph with $\text{im}(r) \subseteq \mathbb{N}^+$, we can use an algorithm of Thorup (which assumes constant-time multiplication) and obtain the same time complexity [4]. If we only assume G to fulfill $\text{im}(r) \subseteq \mathbb{N}_0$ (G does not have to be undirected), we can use the priority queue data structure proposed by van Emde Boas et al. and obtain a runtime in $\mathcal{O}(m + n \log \log R)$ [5]. To simplify dealing with these special cases, we propose the following definition:

Definition 1. Let G be a graph as defined in section 2. Then we define $f_G(n, m) := m$ if G is acyclic or undirected with positive integer resource values. This means that we can replace the cost function of G by an arbitrary modified cost function c' with $\text{im}(c') \subseteq \mathbb{N}_0$ and still compute one row of our dynamic programming scheme for G in $\mathcal{O}(m)$. Otherwise, if $\text{im}(r) \subseteq \mathbb{N}_0$, we define $f_G(n, m) := m + n \cdot \min\{\log n, \log \log R\}$. In all other cases, we set $f_G(n, m) := m + n \log n$. Thus, we can compute the exact solution for such a

modified G in time $\mathcal{O}\left(f_G(n,m)(1+C_{\text{opt}}(G))\right)$.

Algorithm 1 shows the rough structure of the algorithm. Additionally, an iteration limit l is included, providing the possibility to test whether $C_{\text{opt}}(G) \leq l$. Using the examinations from above, we obtain the following lemma.

Lemma 2. *Let G be defined as above. Given any modified graph $G' = (V, E, c', r)$ with c' satisfying $\text{im}(c') \subseteq \mathbb{N}_0$ and an iteration limit $l \in \mathbb{N}_0 \cup \{\infty\}$, Algorithm 1 finds out*

- *an optimal path and the optimal cost $C_{opt}(G')$ if $C_{opt}(G') \leq l$*
- *the fact that $C_{opt}(G') > l$ otherwise*

in time $\mathcal{O}\left(f_G(n,m) \cdot (1 + \min\{l, C_{opt}(G')\})\right)$. If $0 \notin \text{im}(c')$, the time complexity bound $\mathcal{O}\left(m \cdot (1 + \min\{l, C_{opt}(G')\})\right)$ can be achieved.

Algorithm 1 Algorithm for solving the RSP Problem exactly, including an optional iteration limit l

```
function ExactRSP(G = (V, E, c, r), s ∈ V, t ∈ V, R ∈ ℝ₀⁺, l ∈ ℕ₀∪{∞})
    for k from 0 to l do
        Compute r_{k,v} for all v
        if r_{k,t} ≤ R then
            Trace the optimal path p back from r_{k,t}
            return (p, k)
        end if
    end for
    return (⊥, ∞)
end function
```

5 FPTAS for RSP

Since all parts of the algorithm by Lorenz and Raz [3] are used in our algorithm and we want to use a slightly different formulation, we will explain it here. Their algorithm finds a $(1+\varepsilon)$-approximation to the RSP problem in time $\mathcal{O}\left(nm\left(1/\varepsilon + \log\log n\right)\right)$. This means that the algorithm will find a solution p_ε to the RSP problem with $C_G(p_\varepsilon) \leq (1+\varepsilon)C_{\text{opt}}(G)$. We may assume that $C_{\text{opt}}(G) > 0$ since we can easily check this condition at the beginning by running Dijkstra's algorithm on all edges with cost zero, finding the smallest

possible resource consumption on this subgraph. Running the algorithm on a graph with an optimal cost of zero would cause a division by zero — this special case can also easily be detected after running the linear approximation described in section 5.2.

5.1 Scaling the Original Problem

Given the graph G and a scaling factor $S \in \mathbb{R}^+$, we can build the graphs G_S^- and G_S^+ with integer costs on their edges by replacing the cost function c of G by c_S^- and c_S^+, respectively. These two functions are defined as follows:

$$c_S^-(e) := \left\lfloor \frac{c(e)}{S} \right\rfloor$$
$$c_S^+(e) := \left\lfloor \frac{c(e)}{S} \right\rfloor + 1 \ .$$

Notice that while the former graph has lower edge cost, the latter graph is guaranteed to have positive integer edge cost and thus enables us to use the algorithm from section 4.1.

Now let us analyze the behavior of the optimal solutions of these scaled graphs in the original graph. Concerning the approximation quality of an optimal scaled graph solution, we can derive[1]

$$\begin{aligned}
C_G(P_{\text{opt}}(G_S^+)) &= \sum_{e \in P_{\text{opt}}(G_S^+)} c\,(e) \\
&= S \cdot \sum_{e \in P_{\text{opt}}(G_S^+)} \frac{c(e)}{S} \\
&\leq S \cdot \sum_{e \in P_{\text{opt}}(G_S^+)} \left(\left\lfloor \frac{c(e)}{S} \right\rfloor + 1 \right) \\
&\leq S \cdot \sum_{e \in P_{\text{opt}}(G)} \left(\left\lfloor \frac{c(e)}{S} \right\rfloor + 1 \right) \\
&\leq S \cdot \sum_{e \in P_{\text{opt}}(G)} \left(\frac{c(e)}{S} + 1 \right) \\
&\leq C_{\text{opt}}(G) + nS \\
&= \left(1 + \frac{nS}{C_{\text{opt}}(G)} \right) C_{\text{opt}}(G) \ . \qquad (5.1)
\end{aligned}$$

[1] A similar bound can be derived for G_S^-, but it will not be used for the algorithms described here.

In addition, we want to analyze the estimation quality in the scaled graphs. It should be clear that

$$C_{\text{opt}}(G_S^+) \geq \frac{C_{\text{opt}}(G)}{S} \tag{5.2}$$

and

$$C_{\text{opt}}(G_S^-) \leq \frac{C_{\text{opt}}(G)}{S} \ . \tag{5.3}$$

Since the cost of each edge in G_S^+ differs exactly by one from the cost of the same edge in G_S^- and the optimal path consists of at most $n-1$ edges, we also know that

$$C_{\text{opt}}(G_S^+) < C_{\text{opt}}(G_S^-) + n \overset{(5.3)}{\leq} \frac{C_{\text{opt}}(G)}{S} + n \tag{5.4}$$

and similarly

$$C_{\text{opt}}(G_S^-) > C_{\text{opt}}(G_S^+) - n \overset{(5.2)}{\geq} \frac{C_{\text{opt}}(G)}{S} - n \ . \tag{5.5}$$

Using these equations, we see that the approach using G_S^- takes

$$\mathcal{O}\left(f_G(n,m) \cdot (1 + C_{\text{opt}}(G_S^-))\right) \subseteq \mathcal{O}\left(f_G(n,m) \cdot \left(1 + \frac{C_{\text{opt}}(G)}{S}\right)\right) \tag{5.6}$$

time, while using G_S^+ leads to a runtime in

$$\mathcal{O}\left(m \cdot C_{\text{opt}}(G_S^+)\right) \subseteq \mathcal{O}\left(m \cdot \left(n + \frac{C_{\text{opt}}(G)}{S}\right)\right) \ . \tag{5.7}$$

You might notice that for big scaling factors S, more specifically for those satisfying $C_{\text{opt}}(G_S^-) \in o\left(n/\log n\right)$, we should use G_S^- because it leads to a faster runtime. Conversely, for smaller scaling factors, the use of G_S^+ might be advantageous. The combination of these two variants is a key to the improved runtime of our algorithm.

By equation (5.1), we know that if we choose

$$S \leq \frac{\varepsilon C_{\text{opt}}(G)}{n} \ , \tag{5.8}$$

the exact solution of G_S^+ yields a $(1+\varepsilon)$-approximation for the original problem. If we chose $S \coloneqq \varepsilon C_{\text{opt}}(G)/n$ and solved the RSP problem for G_S^+ exactly, the runtime complexity would be $\mathcal{O}\left(nm(1+1/\varepsilon)\right)$ as obtained from equation (5.7).

Unfortunately, we cannot use this approximation directly because we need to know $C_{\text{opt}}(G)$ to be able to compute the scaling factor S. But if we were able

to determine a lower bound L for $C_{\text{opt}}(G)$ satisfying $L \geq C_{\text{opt}}(G)/d$ for some constant d, we could use $S \coloneqq \varepsilon L/n$ and still obtain a $(1+\varepsilon)$-approximation with the same runtime complexity. Assuming that we have an algorithm ConstantBounds which computes such a constant-approximation lower bound (possibly together with an upper bound), algorithm 2 shows how to obtain the desired $(1+\varepsilon)$-approximation.

Algorithm 2 Algorithm finding a $(1+\varepsilon)$-approximation for the RSP problem

function ApproxRSP($G = (V, E, c, r), s \in V, t \in V, R \in \mathbb{R}_0^+, \varepsilon \in \mathbb{R}^+$)
  $(L, U) \coloneqq$ ConstantBounds(G, s, t, R)
  $S \coloneqq \varepsilon \cdot L/n$
  $(p', c') \coloneqq$ ExactRSP(G_S^+, s, t, R)
  return $(p', C_G(p'))$
end function

5.2 A Linear Approximation

As a first step towards a constant approximation we will derive an efficiently computable linear approximation algorithm for the RSP problem. At first, we can sort the edges of G by their cost in ascending order. This can be done in time $\mathcal{O}(m \log m)$. Let $e_1, \ldots, e_m$ be these sorted edges. We are then able to compute the graphs $G_i \coloneqq (V, E_i, c, r)$ where $E_i \coloneqq \{e_1, \ldots, e_i\}$. For any i, we can check in time $\mathcal{O}(m + n \log n)$ whether there is a path from s to t in G_i whose resource consumption is no more than R by executing Dijkstra's algorithm on (V, E_i, r). Thus, using binary search, we can find the smallest i for which such a path exists in time $\mathcal{O}((m + n \log n) \log m)$ — let us call it i^*.

Ignoring the trivial case $s = t$, we now know that any solution path p to the problem cannot only use edges with lower cost than $c(e_{i^*})$. Therefore, $L \coloneqq c(e_{i^*})$ is a lower bound for $C_{\text{opt}}(G)$. On the other hand, we know that there is a solution to the original problem only consisting of edges from E_{i^*}. It follows that $U \coloneqq n \cdot c(e_{i^*})$ is an upper bound for $C_{\text{opt}}(G)$. This is summarized in algorithm 3. Since the runtime of this algorithm is in $\mathcal{O}(nm)$ and thus does not affect the overall time complexity, we will neglect it in further considerations.

5.3 A Constant Approximation

To find a constant approximation, we would like to get an approximation with known approximation quality using a value $c' = C_{\text{opt}}(G_S^+)$ for some scaling

Algorithm 3 Algorithm finding a linear approximation for the RSP problem

function LINEARBOUNDS($G = (V, E, c, r), s \in V, t \in V, R \in \mathbb{R}_0^+$)
  $(e_1, \ldots, e_m) \coloneqq$ edges sorted by ascending cost
  Find i^* by using binary search with Dijkstra on $(G_i)_{1 \leq i \leq m}$
  return $(c(e_{i^*}), n \cdot c(e_{i^*}))$
end function

factor S. Using (5.2) and (5.4), we see that $Sc' \geq C_{\text{opt}}(G)$ and

$$\frac{Sc'}{C_{\text{opt}}(G)} \leq \frac{Sc'}{Sc' - Sn} = \frac{c'}{c' - n} \ . \tag{5.9}$$

For example, if $c' \geq 2n$, we have $c'/(c' - n) \leq 2$ and thus Sc' is a 2-approximation upper bound for $C_{\text{opt}}(G)$.

As we have shown at the end of section 5.1, all we have to do to find a constant approximation is to find a scaling factor S such that $c' = C_{\text{opt}}(G_S^+)$ is at least $2n$. To limit the computation time needed for computing that optimum, we also require $c' \leq 5n$. Both requirements are certainly fulfilled if $(C_{\text{opt}}(G)/S) \in [2n, 4n]$ (this follows from the inequalities in section 5.1). Given lower and upper bounds (L, U) for $C_{\text{opt}}(G)$, we know that this is true for some S where

$$\frac{L}{2n} \leq S \leq \frac{U}{2n} \ . \tag{5.10}$$

Because $4n = 2 \cdot 2n$, it suffices to consider a subset of all possible values for S such that the quotient of two consecutive values in this subset does not exceed 2 (i. e., the gaps between the values are not too big). Thus, we only need to consider the values

$$S_i \coloneqq \frac{U}{2n} \cdot 2^{-i} \tag{5.11}$$

for $i \in \{0, \ldots, \lceil \log_2(U/L) \rceil\}$. Executing the statement

$$(p', c') \coloneqq \text{EXACTRSP}(G_{S_i}^+, s, t, R, 5n) \tag{5.12}$$

lets us distinguish three cases in time $\mathcal{O}(mn)$:

- If $c' < 2n$, we know that S_i is too big.
- If $c' = \infty$, we know that S_i is too small.
- If $2n \leq c' \leq 5n$, we know that $L \coloneqq S_i c'/2$ is a lower bound and $U \coloneqq S_i c'$ is an upper bound for $C_{\text{opt}}(G)$.

Thus, we can again use binary search to find a number i^* such that S_{i^*} is a desired scaling factor. Because we know that $i^* \in \{0, \ldots, \lceil \log_2(U/L) \rceil\}$, this computation runs in time $\mathcal{O}(mn \log\log(U/L))$. Algorithm 4 shows the structure of this approach.

Algorithm 4 Algorithm finding a constant approximation for the RSP problem from arbitrary lower and upper bounds

function ConstBounds($G = (V, E, c, r), s \in V, t \in V, R \in \mathbb{R}_0^+, L \in \mathbb{R}^+, U \in \mathbb{R}^+$)
    Compute values S_i from L and U
    Find i^* by using binary search on S_i
    $S \coloneqq S_{i^*}$
    $(p', c') \coloneqq$ ExactRSP(G_S^+, s, t, R, ∞)
    return $(Sc'/2, Sc')$
end function

5.4 Previous Solution

The algorithm by Lorenz and Raz [3] uses the result of the linear approximation from section 5.2 as bound inputs for the constant approximation algorithm shown in section 5.3. This leads to a runtime in $\mathcal{O}(nm \log\log n)$, yielding an overall time complexity of

$$\mathcal{O}\left(nm\left(\frac{1}{\varepsilon} + \log\log n\right)\right) \tag{5.13}$$

for the $(1+\varepsilon)$-approximation scheme.

6 Improved FPTAS for RSP

In this section, we want to propose a $\mathcal{O}(\log n)$-approximation for the RSP problem with a time complexity of $\mathcal{O}(nm)$ which directly leads to an improved FPTAS with a runtime complexity of $\mathcal{O}(nm(1/\varepsilon + \log\log\log n))$. Furthermore, we introduce a variant of this approximation which can find a 2-approximation for RSP in time $\mathcal{O}(nf_G(n, m))$.

6.1 Basic Algorithm

In section 5.3, we have seen that by analyzing $G_{S_i}^+$ for different values of i, we can find a 2-approximation for $C_{\text{opt}}(G)$. Now we will consider the same

values S_i, $i \in \{0, \ldots, \lceil \log_2(U/L) \rceil\}$, but instead analyze $G^-_{S_i}$ because it has lower edge cost. This enables us to run a linear search on the S_i-values in reasonable time. The concrete algorithm is desribed as algorithm 5. This algorithm contains a parameter b to trade off approximation quality against runtime complexity. It traverses the values S_i until the graph $G^-_{S_i}$ has an optimal cost $> b$. Based on b, U and the last value of i, it then returns a valid approximation for $C_{\text{opt}}(G)$, as we will prove in the next section.

Algorithm 5 Algorithm using linear search to find bounds for the RSP problem

```
1: function LSBounds(G = (V, E, c, r), s ∈ V, t ∈ V, R ∈ ℝ₀⁺, b ∈ ℕ⁺)
2:     (L, U) ≔ LinearBounds(G, s, t, R)
3:     Compute values S_i from L and U
4:     for i = 0, 1, 2, ... do
5:         S ≔ S_i
6:         (p', c') ≔ ExactRSP(G_S^-, s, t, R, b)
7:         if c' = ∞ then
8:             if i = 0 then
9:                 return (Sb, U)
10:            else
11:                return (Sb, 2S(b + n))
12:            end if
13:        end if
14:    end for
15: end function
```

6.2 Analysis

First, we want to show that the bounds found by this algorithm are indeed correct. Obviously, U is a correct upper bound as we have explained in section 5.2. When the algorithm finds that $c' = \infty$, we know that $C_{\text{opt}}(G^-_S) > b$. Thus, we have

$$b \cdot S < C_{\text{opt}}(G^-_S) \cdot S \leq C_{\text{opt}}(G) \ . \tag{6.1}$$

Finally, if the algorithm reaches line 11, the value of c' in the previous iteration must have been at most b. Using the bounds from section 5.1, we obtain

$$b \geq C_{\text{opt}}(G^-_{S_{i-1}}) \geq \left(\frac{C_{\text{opt}}(G)}{S_{i-1}} - n \right) \ . \tag{6.2}$$

Since we know that $S_{i-1} = 2S_i$, it follows that

$$C_{\text{opt}}(G) \leq (b + n) S_{i-1} = 2(b + n) S_i \ . \tag{6.3}$$

Now that we have shown the correctness of the algorithm, we can analyze the quality of the approximation, concretely the quotient U/L, for arbitrary values of b.

Lemma 3. *For the values U and L returned by Algorithm 5, the estimation $U/L \in \mathcal{O}(1 + n/b)$ holds.*

Proof. We have to distinguish two cases:

- The algorithm terminates in line 9. Then,

$$\frac{U}{L} = \frac{U}{S_0 b} = \frac{U \cdot 2n}{Ub} = \frac{2n}{b} \ . \tag{6.4}$$

- The algorithm terminates in line 11. In this case, we obtain

$$\frac{U}{L} = \frac{2S(b+n)}{Sb} = 2 + \frac{2n}{b} \ . \tag{6.5}$$

We can therefore conclude that $U/L \in \mathcal{O}(1 + n/b)$. □

The last part of our analysis concerns the time complexity of the algorithm. By Lemma 2, we know that the runtime of $\textsc{ExactRSP}(G', s, t, R, b)$ lies in $\mathcal{O}(f_G(n, m) \cdot (1 + \min\{b, C_{\text{opt}}(G')\}))$. Because of this, we can bound the runtime $h(i)$ of iteration i from above by

$$h(i) \leq A f_G(n, m) \cdot (1 + \min\{b, C_{\text{opt}}(G^-_{S_i})\}) + D \tag{6.6}$$

for some constants $A, D \in \mathbb{R}^+$. Let i^* be the value of i in the last iteration. If $b \leq n$, we can derive the estimation

$$i^* \leq \lceil \log_2 n \rceil \tag{6.7}$$

similar to our analysis of the binary search bounds in section 5.3.[2]

Now, we can take our final step towards the running time of the algorithm.

Lemma 4. *If $b \leq n$, the time complexity of algorithm 5 is*

$$\mathcal{O}(f_G(n, m) \cdot (1 + b + \log n)) \ .$$

[2]Note that the assumption $b \leq n$ was only made for simplicity. If we dropped the assumption, we could derive $i^* \leq \left\lceil \log_2 \left(\frac{b+n}{2} \right) \right\rceil$ and Lemma 4 would still hold.

Proof. Obviously, $T(i^*) \coloneqq \sum_{i=0}^{i^*} h(i)$ describes the runtime of the loop in the algorithm. If $i^* = 0$, we have

$$T(i^*) = \sum_{i=0}^{i^*} h(i) \leq Af_G(n,m) \cdot (1+b) + D \in \mathcal{O}\left(f_G(n,m) \cdot (1+b+\log n)\right) \ .$$

If instead $i^* > 0$, we know that the previous scaling factor S_{i^*-1} led to a graph with optimal cost less than or equal to b. Since for all $e \in E$

$$c^-_{2S}(e) = \left\lfloor \frac{c(e)}{2S} \right\rfloor \leq \frac{1}{2} \left\lfloor \frac{c(e)}{S} \right\rfloor \ , \tag{6.8}$$

we conclude $C_{\text{opt}}(G^-_{S_{j-1}}) \leq C_{\text{opt}}(G^-_{S_j})/2$ for every $j > 0$. Iterating this inequality, we obtain

$$C_{\text{opt}}(G^-_{S_{i^*-1-k}}) \leq 2^{-k} C_{\text{opt}}(G^-_{S_{i^*-1}}) \leq 2^{-k} b \tag{6.9}$$

for every $0 \leq k \leq i^* - 1$. Using the shorthand $F \coloneqq f_G(n,m)$, this leads us to the estimation

$$\begin{aligned}
T(i^*) &= \sum_{i=0}^{i^*} h(i) \\
&\leq \left(\sum_{i=0}^{i^*-1} AF(1 + C_{\text{opt}}(G^-_{S_i})) + D \right) + (AF(1+b) + D) \\
&\overset{(6.9)}{\leq} \left(\sum_{i=0}^{i^*-1} AF 2^{-(i^*-1-i)} b \right) + (AF + D) \cdot i^* + (AF \cdot (1+b) + D) \\
&\leq AFb \left(\sum_{j=0}^{i^*-1} 2^{-j} \right) + (AF + D)(1 + i^*) + AFb \\
&\leq AFb \left(\sum_{j=0}^{\infty} 2^{-j} \right) + (AF + D)(1 + i^*) + AFb \\
&= 3AFb + (AF + D)(1 + i^*) \\
&\overset{(6.7)}{\leq} 3AFb + (AF + D)\left(2 + \log_2(n)\right) \\
&\in \mathcal{O}\left(f_G(n,m) \cdot (1 + b + \log n)\right) \ ,
\end{aligned}$$

which proves the claim. □

6.3 Variants of the algorithm

Because we are able to choose the parameter b in the algorithm above freely, there are several possibilities of usage. At first, we want to present our improved general FPTAS.

Corollary 5. *A $\mathcal{O}(\log n)$-approximation for RSP can be computed with a runtime in $\mathcal{O}(nm)$. Using this approximation as an input for the constant approximation algorithm from section 5.3, the overall algorithm has a time complexity of $\mathcal{O}(nm(1/\varepsilon + \log\log\log n))$.*

Proof. We will show that algorithm 5 computes the desired $\mathcal{O}(\log n)$-approximation with the mentioned complexity when using

$$b \coloneqq \max\left\{1, \left\lfloor \frac{n}{\log n} \right\rfloor\right\} \ . \tag{6.10}$$

Using Lemma 3, we see that

$$\frac{U}{L} \in \mathcal{O}(1 + n/b) = \mathcal{O}(1 + \log n) = \mathcal{O}(\log n) \ . \tag{6.11}$$

According to Lemma 4, the runtime of the algorithm lies in

$$\begin{aligned}\mathcal{O}(f_G(n,m) \cdot (1 + b + \log n)) &\subseteq \mathcal{O}\left((m + n\log n) \cdot \left(1 + \frac{n}{\log n} + \log n\right)\right) \\ &= \mathcal{O}\left(n\left(\frac{m}{\log n} + n\right)\right)\end{aligned}$$

and thus, since $m \geq n - 1$, also in $\mathcal{O}(nm)$. □

The complexity of this algorithm can be improved in any case where $f_G(n,m) \in \mathcal{O}(m)$, i. e. for graphs where the exact algorithm with zero-cost edges runs in time $\mathcal{O}(m \cdot (1 + C_{\text{opt}}(G)))$. As mentioned in section 4.2, these graphs comprise directed acyclic graphs and, assuming constant-time multiplication, undirected graphs where $\text{im}(r) \subseteq \mathbb{N}^+$.

Corollary 6. *A $\mathcal{O}(1)$-approximation for RSP can be computed with a runtime in $\mathcal{O}(n \cdot f_G(n,m))$. Using this approximation, the overall algorithm has a time complexity of $\mathcal{O}(nm(1/\varepsilon + f_G(n,m)/m))$.*

Proof. We will show that algorithm 5 computes the desired $\mathcal{O}(1)$-approximation with the mentioned complexity when using

$$b \coloneqq n \ . \tag{6.12}$$

By Lemma 3, we have

$$\frac{U}{L} \in \mathcal{O}(1 + n/b) = \mathcal{O}(1 + 1) = \mathcal{O}(1) \ . \tag{6.13}$$

According to Lemma 4, the runtime of the algorithm lies in

$$\begin{aligned}\mathcal{O}(f_G(n,m) \cdot (1 + b + \log n)) &\subseteq \mathcal{O}(f_G(n,m) \cdot (1 + n + \log n)) \\ &= \mathcal{O}(n \cdot f_G(n,m)) \ .\end{aligned}$$

□

7 Remarks

Throughout this thesis, we use the notation $\mathcal{O}(f)$ for functions $f : \mathbb{R}^n \to \mathbb{R}$, which we assume to be defined as

$$\mathcal{O}(f) \coloneqq \{g : \mathbb{R}^n \to \mathbb{R} \mid \exists A, D \in \mathbb{R} : g \leq Af + D\} \; . \tag{7.1}$$

Furthermore, we have some more ideas that might help to further improve the runtime of our algorithm:

- If the time needed to compute one row of the dynamic programming scheme could be improved to $\mathcal{O}(m)$ for graphs with zero-cost edges, we would instantly get the desired $\mathcal{O}(mn(1 + 1/\varepsilon))$ runtime for the approximation scheme.

- The runtime of this row computation only has to be improved if we have $n/\log n \leq C_{\text{opt}}(G_S^-) \leq n$ and $m \in o(n \log n)$. Both assertions indicate that there are possibly fewer and/or smaller cycles in the graph generated for Dijkstra's algorithm than for denser graphs and higher scaling factors.

- Since the zero-cost edges do not change during one execution of the dynamic programming scheme, we might be able to speed up the SSSP computation by doing a precomputation at the beginning of the dynamic programming scheme. This precomputation may take up to $\mathcal{O}(mn/\log\log n)$ time. In addition, one could try to exploit the fact that in successive iterations in the linear search algorithm, no zero-cost edges are ever added to the scaled graph.

- We only have to apply Dijkstra's algorithm to the strongly connected components of the graph from section 4.2. The acyclic part of its structure can be handled efficiently.

- After having computed the $\mathcal{O}(\log n)$-approximation, one might want to establish randomized rounding to obtain a Monte-Carlo approximation algorithm. Unfortunately, although it might work for "average" graphs, there are counterexamples where it is unlikely to find a good approximation.

8 Summary

In this thesis, we were able to improve the runtime of the approximation scheme to $\mathcal{O}(nm(1/\varepsilon + \log\log\log n))$. Unfortunately, we did not find a possibility to reduce the runtime complexity to the somewhat more aesthetic

$\mathcal{O}(nm(1/\varepsilon + 1))$ for all graphs, although we enlarged the types of graphs for which this complexity can be achieved. As we indicated in the last section, many ideas can be found to modify our algorithm. Assessing their use requires precise analysis and often brings up even more variants to be reviewed. The restricted shortest path problem thus remains an interesting challenge for future research.

References

[1] F. Ergun, R. Sinha, and L. Zhang. An improved FPTAS for restricted shortest path. *Information Processing Letters*, 83(5):287–291, 2002.

[2] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations research*, 17(1):36–42, 1992.

[3] D. H. Lorenz and D. Raz. A simple efficient approximation scheme for the restricted shortest path problem. *Operations Research Letters*, 28(5):213–219, 2001.

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

[5] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

A German summary

Das „Restricted Shortest Path“-Problem ist ein NP-hartes bikriterielles Optimierungsproblem. Eingabe ist ein Graph $G = (V, E, c, r)$, dessen Kanten durch nichtnegative Funktionen c und r ein Kosten- und ein Ressourcenwert zugewiesen ist. Ziel ist es, den bezüglich der Gesamtkosten $C_G(p)$ günstigsten Pfad p von $s \in V$ nach $t \in V$ zu finden, dessen Ressourcensumme $R_G(p)$ einen Grenzwert R nicht übersteigt. Dabei bezeichnet n die Anzahl der Knoten und m die Anzahl der Kanten des Graphen. Mithilfe von dynamischer Programmierung kann ein solcher optimaler Pfad $P_{\text{opt}}(G)$ gefunden werden, falls der Graph ganzzahlige Kostenwerte besitzt. Die Laufzeit dieses Lösungsverfahrens hängt linear von der Anzahl m der Kanten von G und den Kosten $C_{\text{opt}}(G) = C_G(P_{\text{opt}}(G))$ des optimalen Pfads ab. Um die Abhängigkeit von den Kosten des Pfades zu vermeiden, kann die Kostenfunktion eines Graphen G skaliert und gerundet werden. Dabei wird abhängig vom Skalierungsgrad eine hinsichtlich der Pfadkosten mehr oder weniger gute Approximation für den optimalen Pfad gefunden. Die Hauptkomplexität des Problems steckt darin, einen geeigneten Skalierungsfaktor zu finden.

Der Algorithmus von Lorenz und Raz findet zuerst auf einfache Weise eine $\mathcal{O}(n)$-Approximation für $C_{\text{opt}}(G)$. Anschließend werden basierend auf dieser Approximation $\mathcal{O}(\log n)$ potenzielle Skalierungsfaktoren mit binärer Suche durchsucht, bis ein geeigneter Skalierungsfaktor gefunden wird. Da jeder Suchschritt eine Zeitkomplexität von $\mathcal{O}(nm)$ hat, liegt die Gesamtkomplexität für diese Suche bei $\mathcal{O}(nm \log\log n)$. Eine $(1+\varepsilon)$-Approximation kann damit mit einer Zeitkomplexität von $\mathcal{O}(nm(1/\varepsilon + \log\log n))$ erhalten werden.

Der neue Ansatz, der in dieser Bachelorarbeit vorgestellt wird, ist in einer Zeit in $\mathcal{O}(nm)$ in der Lage, die Anzahl der mit binärer Suche zu durchsuchenden Skalierungsfaktoren auf $\mathcal{O}(\log\log n)$ und für manche Graphen sogar auf 0 einzuschränken. Damit verringert sich die Laufzeit des Approximationsalgorithmus von $\mathcal{O}(nm(1/\varepsilon + \log\log n))$ auf $\mathcal{O}(nm(1/\varepsilon + \log\log\log n))$ bzw. $\mathcal{O}(nm(1/\varepsilon + 1))$. Dieser Ansatz basiert auf vier Ideen:

- Es wird eine lineare Suche im Gegensatz zur binären Suche verwendet. Diese muss zwar mehr Skalierungsfaktoren durchsuchen, benötigt für die meisten Faktoren aber erheblich weniger Zeit, sodass sich letztendlich in der Abschätzung eine geometrische Reihe ergibt.

- Um diese Laufzeitreduktion durch die lineare Suche gewährleisten zu können, werden die Kosten beim Skalieren abgerundet statt aufgerundet.

- Die durch das Abrunden auftretenden Kanten mit Nullkosten müssen im exakten Lösungsalgorithmus speziell behandelt werden, dafür wird

im Allgemeinen der Algorithmus von Dijkstra verwendet.

- Da der Algorithmus von Dijkstra die Zeitkomplexität für Graphen mit wenigen Kanten erhöht, können bei diesen Graphen nur die billigeren Skalierungsfaktoren mit dieser Methode durchsucht werden. Es verbleiben dann $\mathcal{O}(\log \log n)$ Skalierungsfaktoren, die mit der bereits bekannten Suchtechnik durchsucht werden können.

In dieser Bachelorarbeit werden das Problem und der Algorithmus von Lorenz und Raz ausführlich vorgestellt. Außerdem wird der erwähnte neue Ansatz zur Verbesserung der Laufzeit vorgestellt und ausführlich analysiert.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, 04.02.2016

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature:

Stuttgart, 04.02.2016