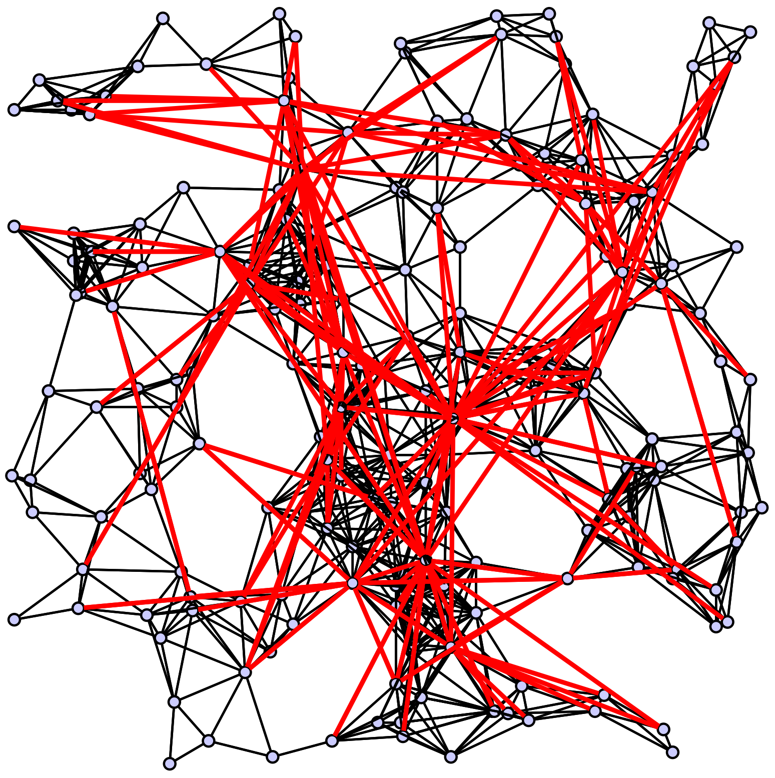
Heuristic Algorithms for the Shortcut Problem

Diplomarbeit von Andrea Schumm
Thesis of Andrea Schumm

August 18, 2009



Supervisors: Prof. Dr. Dorothea Wagner
and Dipl.-Math. Reinhard Bauer

Institute of Theoretical Informatics
Universität Karlsruhe (TH)

Danksagung

In erster Linie möchte ich mich bei Reinhard Bauer für die großartige Betreuung während der letzten Monate bedanken. Ebenso danke ich Frau Prof. Dr. Dorothea Wagner für die Möglichkeit, meine Abschlussarbeit an ihrem Lehrstuhl zu schreiben, sowie für die Annahme und Bewertung dieser Arbeit.

Für die moralische Unterstützung, die mir zuteil wurde, sowie für abschließendes Korrekturlesen bin ich André sehr dankbar.

Ganz besonders möchte ich auch meiner Familie dafür danken, dass ich mich in allen Phasen meines Studiums und auf meinem sonstigen Lebensweg auf ihre volle Unterstützung verlassen konnte.

Deutsche Zusammenfassung

Kürzeste-Wege-Probleme tauchen in vielen Anwendungsgebieten auf, zum Beispiel im Kontext der Routenplanung. Die Netzwerke, die dort verwendet werden, sind typischerweise so groß, dass die Benutzung von Standardalgorithmen zur Berechnung des kürzesten Weges zwischen zwei Punkten zu inakzeptablen Anfragezeiten führt.

In den letzten Jahren wurde eine Vielzahl an Techniken entwickelt, um diese Anfragezeiten zu verringern. Viele dieser *speed-up Techniken* benutzen dabei *Shortcuts*. Shortcuts sind zusätzliche Kanten, deren Länge der Distanz der Endknoten entspricht. Abhängig von der konkreten Technik werden verschiedene Strategien verwendet, um eine Menge von Shortcuts zu bestimmen, diese sind jedoch ausnahmslos heuristisch. Die Hauptaufgabe dieser Shortcuts ist dabei, den Suchraum zu reduzieren, indem beispielsweise die Anzahl der Kanten eines kürzesten Weges verringert wird.

Vor kurzem hat dies zu der Idee geführt, die Bestimmung einer optimalen Menge an Shortcuts als ein eigenständiges Problem zu betrachten. Das Ziel dabei ist, die durchschnittliche Anzahl der Kanten auf kürzesten Wegen im Graphen zu minimieren, indem eine feste Anzahl an Shortcuts eingefügt wird. Das entsprechende kombinatorische Problem ist das Best Shortcut Problem(BSP). Die Untersuchung dieses Problems ist ein erster Ansatz, um die empirischen Ergebnisse, die mit Shortcuts erzielt wurden, durch einen theoretischen Hintergrund zu bereichern. Darüber hinaus stellt das BSP eine interessante algorithmische Fragestellung als ein eigenständiges Problem dar.

Wie in [BDDW09] bewiesen wurde, ist das BSP NP-hart. Dort wurde auch ein Approximationsalgorithmus vorgestellt, dessen Güte von der Anzahl der Shortcuts abhängt. Dieser Algorithmus fügt sukzessive Shortcuts in den Graphen ein, indem er jeweils das BSP eingeschränkt auf einen Shortcut in den modifizierten Graphen löst.

Im ersten Teil dieser Arbeit beschäftigen wir uns mit der Frage, wie die Laufzeit dieses Teilproblems verringert werden kann. Als Hauptergebnis dieses Abschnittes geben wir einen Algorithmus an, der das BSP eingeschränkt auf einen Shortcut mit einer Laufzeit in $\Theta(n^3)$ löst. Verglichen mit dem Brute-Force Ansatz, der in [BDDW09] verwendet wird, ist das eine deutliche Verbesserung. Diese theoretischen Überlegungen bestätigen wir mit einer experimentellen Auswertung unter Verwendung verschiedener Graphklassen.

Da eine Laufzeit in $\Theta(n^3)$ für größere Graphen immer noch zu viel ist, betrachten wir weiterhin Heuristiken, um einzelne Shortcuts mit hoher Qualität zu finden. Dazu geben wir verschiedene Möglichkeiten an, die Zentralitätsindizes Betweenness und Stress für Paare von Knoten zu generalisieren. Dadurch erzielen wir Schätzungen für die Qualität einzelner Shortcuts. Falls kürzeste Wege im zugrunde liegenden Graphen eindeutig sind, sind diese Schätzungen exakt. Des weiteren geben wir Algorithmen an, die diese generalisierten Zentralitätsindizes mit einer Laufzeit in $O(n \cdot (n \log n + m))$ berechnen.

Da die Größe von Straßennetzwerken typischerweise nur Algorithmen mit fast-linearen Laufzeiten erlaubt, kann in diesen Netzwerken nicht einmal die Qualität einer gegebenen Lösung vollständig auswertet werden. Da die Zielsetzung dieser Arbeit ist, hochqualitative Heuristiken für das BSP als eigenständiges Problem zu entwickeln, beschränken

wir unsere Experimente auf kleine Graphen. Die Lösungen, die wir auf diese Weise erhalten, bieten dabei Anhaltspunkte für die generelle Struktur optimaler Lösungen. Im abschließenden experimentellen Teil dieser Arbeit werten wir die Ergebnisse aus, die mit dem Greedy-Algorithmus aus [BDDW09] und verschiedenen lokalen Suchstrategien in Verbindung mit den Algorithmen aus dem ersten Teil erzielt werden können.

Contents

Nomenclature

$\lvert P\rvert$	Hop-length of a path P	11
$\bar{G}$	The reverse graph of G	12
$\Delta_i(s,b)$	$\lvert\{t \in P^+(s,b) \mid o_b(s,t) = i\}\rvert$	35
η_{st}	Number of hop-minimal shortest s-t-paths	11
$\eta_{wz}(x,y)$	Number of hop-minimal shortest w-x-y-z-paths	12
$\eta_{xz}(y)$	Number of hop-minimal shortest x-y-z-paths	12
$\mathrm{In}_s(v)$	The set of inneighbours of v in D_s	13
$\mathrm{len}(P)$	Length of a path P	11
$\mathrm{Out}_s(v)$	The set of outneighbours of v in D_s	13
σ_{st}	Number of shortest s-t-paths	11
$\sigma_{st}(e)$	Number of shortest s-t-paths that use $e \in E$	44
$\sigma_{wz}(x,y)$	Number of shortest w-x-y-z-paths	12
$\sigma_{xz}(y)$	Number of shortest x-y-z-paths	12
$\mathrm{dist}(s,t)$	Distance from s to t	11
$C_B(a,b)$	Pair betweenness centrality of $(a,b) \in V \times V$	46
$C_B(e)$	Betweenness centrality of an edge e	44
$C_B(v)$	Betweenness centrality of a node v	31
$C_r(s,b)$	$\sum_{i=0}^{r} \Delta_i(s,b)$	35
$C_S(a,b)$	Pair stress centrality of $(a,b) \in V \times V$	46
$C_S(e)$	Stress centrality of an edge e	44
$C_S(v)$	Stress centrality of a node v	44

1 Introduction

One of the basic issues in graph theory is to find shortest paths in a graph. The probably most prominent algorithm in this context is the one introduced by E. W. Dijkstra in 1959.

Shortest-paths problems occur in many fields of application. For example, if we aim to travel from Karlsruhe to Munich as fast as possible by car or train, this corresponds to a shortest path query. In the context of route planning or timetable information, algorithms typically have to deal with huge networks. Unfortunately, applying Dijkstra's algorithm to these networks leads to unacceptably high query times.

In the last years, a variety of techniques have been developed to shorten point-to-point query times by using data precomputed in advance. A lot of these *speed-up techniques* make use of *shortcuts* in some manner. Shortcuts are additional edges, whose length corresponds to the distance between the end-nodes. The strategy which shortcuts are inserted depends on the particular speed-up technique, but always, heuristics are used to answer this question. The overall purpose of inserting shortcuts is to reduce the search space, for example to decrease the number of edges on a shortest path.

Recently, this gave rise to the idea of considering the question of how to find optimal shortcut assignments as a problem of its own. Here, the aim is to minimize the average number of edges on an edge-minimal shortest path by inserting a given number of shortcuts. Limiting the number of shortcuts is motivated by the desire to keep the memory requirement low. The corresponding combinatorial problem is the BEST SHORTCUT PROBLEM(BSP) as introduced in [BDDW09]. The analysis of the BSP is a first approach to enhance the empirical results on using shortcuts by a theoretical analysis. In addition to that, the BSP turned out to be interesting as a self-contained problem.

Related work. In connection with speed-up techniques, there is a vast amount of publications. To get a recent overview of the results, consider [DSSW09] and [WW07]. Some results concerning the benefit of externally computed shortcuts on the speed-up techniques *arc-flags* and *reach* can be found in [BDW08].

As the main topic of this thesis, the BEST SHORTCUT PROBLEM, is very new, there is not much related work. In [BDDW09], the BSP is introduced and shown to be NP-hard. Further, two approximation algorithms for graphs with unique shortest paths are described and a fast way to stochastically evaluate shortcut assignments is given.

Our Contribution. In this thesis, we revisit the greedy approximation algorithm introduced in [BDDW09]. Basically, this algorithm successively adds shortcuts to a graph. By developing a fast algorithm to solve the BSP restricted to one shortcut, we are able to reduce the asymptotical time complexity of the greedy strategy from $O(c \cdot n^3 \cdot (n \log n + m))$ to $O(c \cdot n^3)$, where c denotes the number of shortcuts we aim to insert. As the running time of this algorithm is still infeasible for larger graphs, we additionally study the question of how to find single, high-quality shortcuts with less time complexity. To this end, we develop different heuristics to estimate the quality of shortcuts. If shortest paths in the underlying graph are unique, these estimations are exact. Moreover, we show how to compute these estimations with a time complexity in $O(n \cdot (n \log n + m))$. Both of these algorithms are easy to understand and implement in practice.

In the experimental part of this thesis, we evaluate strategies to find high-quality solutions for the BSP as a self-contained topic. These shortcut assignments can be used to get some clues about the structure of optimal solutions. As the size of road networks typically only allows algorithms with slightly super-linear time complexity, this does not even permit to evaluate a given shortcut assignment completely. Therefore, our experiments are restricted to rather small graphs. We compare different local search strategies in combination with the algorithms of the first part and evaluate the respective shortcut assignments found.

Overview

In the following, we briefly outline how this thesis is organized:

Chapter 2: The second chapter starts with some basic graph theoretical notation. Additionally, we introduce some definitions in the special context of the topic of this thesis. We then state a formal definition of the BSP and revisit some basic results taken from [BDDW09]. Furthermore, we describe how Dijkstra's algorithm can be modified to compute some extra information and conclude with two basic algorithms to sum up values in shortest-paths dags.

Chapter 3: This chapter is devoted to algorithms that solve the BSP restricted to one shortcut. We start with describing our first approaches to speed up the brute-force algorithm by pruning the search space and conclude with an algorithm that evaluates all shortcuts simultaneously with a time complexity in $\Theta(n^3)$.

Chapter 4: Here, we examine the question which heuristics can be used to find single shortcuts with high quality. To this end, we use the centrality measures stress and betweenness and show how they can be adapted to suit the BSP in a straightforward way. Using these centrality measures, we get an $O(n \cdot (n \log n + m))$-algorithm that finds optimal shortcuts if shortest paths are unique. Furthermore, in the general case, this algorithm can be used to estimate the quality of shortcuts under different aspects.

Chapter 5: In this chapter, we evaluate the algorithms developed to this point. We start with a quick overview of our test instances and evaluate the respective running times. Further, we compare the solutions found by using the shortcut ratings introduced in Chapter 4 among each other and with other simple heuristics.

Chapter 6: This chapter gives a brief overview of fundamental local search strategies and specifies how these can be adapted to the BSP. Moreover, we describe a measure that we will use to examine the structure of the search space of local search algorithms for the BSP.

Chapter 7: This chapter provides the results of the experiments taken in context with our local search algorithms. Furthermore, we examine basic properties of the search space and take a look at the assignments found by using local search.

Chapter 8: We summarize the results of this thesis and conclude with a brief outlook.

2 Fundamentals

2.1 Preliminaries

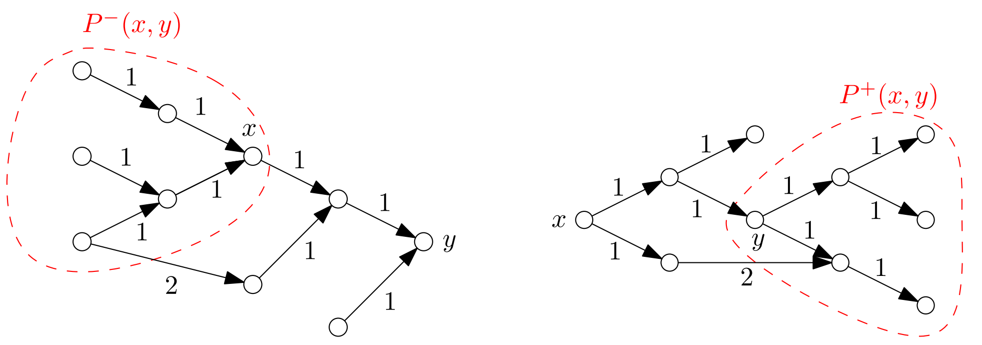
2.1.1 Common notation

Here and in the following, let $G = (V, E, \text{len})$ be a weighted, directed graph with $n = |V|$ vertices, $m = |E|$ edges and length function $\text{len} : E \to \mathbb{R}^+$. For two nodes v and w, we say that w is an *inneighbour* of v, if $(w, v) \in E$. Analogously, an *outneighbour* of v is a node w with $(v, w) \in E$. We call w a *neighbour* of v if it is an inneighbour or an outneighbour of v, or both. If v has no outneighbours, we call it a *sink*.

A *walk* with *source* s and *target* t (or shorter an *s-t-walk*) in G is a k-tuple of vertices $P = (s = u_0, u_1, \ldots, u_{k-1} = t)$, $k \geq 1$, with the property that for every i between 1 and $k - 1$, the edge (u_{i-1}, u_i) exists. P is called a *path*, if the nodes u_i are pairwise distinct. An *undirected walk* in G is a walk in the underlying undirected graph; more precisely, it is a sequence $P = (s = u_0, u_1, \ldots, u_{k-1} = t)$, $k \geq 1$, where for each i between 1 and $k - 1$ at least one of the edges (u_{i-1}, u_i) or (u_i, u_{i-1}) exists in the graph. The *length* of P is defined as $\text{len}(P) := \sum_{i=1}^{k-1} \text{len}(u_{i-1}, u_i)$, whereas the *hop-length* $|P|$ of P is the number of edges on P. An s-t-path is called a *shortest path* if its length is minimal among the lengths of all s-t-paths. Consequently, a shortest path P is called *hop-minimal*, if $|P|$ is minimal among all shortest s-t-paths. Given two vertices s and t, the distance $\text{dist}(s, t)$ from s to t is the length of a shortest s-t-path. Analogously, we denote by the hop-distance $h(s, t)$ from s to t the hop-length of a hop-minimal shortest path between s and t. Furthermore, let σ_{st} be the number of shortest s-t-paths and η_{st} the number of hop-minimal shortest s-t-paths.

A walk $P = (u_0, \ldots, u_{k-1})$ is said to be a *circuit*, if $u_0 = u_{k-1}$, $k \geq 2$ and $v_1, \ldots, v_k$ as well as the edges $(u_0, u_1), \ldots, (u_{k-2}, u_{k-1})$ are distinct. A graph without circuits is called *acyclic*. For directed, acyclic graphs, we use the abbreviation *dag*. For $s \in V$, the *shortest-paths dag* D_s grown from s is a subgraph that is obtained in the following way: We reduce the set of edges to those that are on at least one shortest path from s to another node in V and delete isolated nodes. It is easy to see that, as we assume edge lengths to be positive, these subgraphs are always acyclic.

A directed graph is called *strongly connected* if there exists an s-t-path for each (s, t) in $V \times V$. Consequently, we call it *weakly connected* if the underlying undirected graph is connected; that is, for each two nodes u and v there is an undirected path connecting



(a) Part of the shortest-paths dag grown from y in $\bar{G}$

(b) Part of the shortest-paths dag grown from x in G

Fig. 2.1: Illustration of the sets $P^-(x,y)$ and $P^+(x,y)$

u and v. Fuzzily, we call a simple graph *dense* if the number of its edges is close to the maximal number of edges whereas we characterize it as *sparse* if it has only few edges in relation to the number of nodes. The *reverse graph* $\bar{G} = (V, \bar{E}, \overline{\text{len}})$ is the one obtained from G by substituting each $(u,v) \in E$ by (v,u) and by defining $\overline{\text{len}}(v,u) := \text{len}(u,v)$. An undirected graph is a *tree*, if it is weakly connected and has no undirected circuits. Further, a directed graph G is called a *directed tree* if the underlying undirected graph is a tree.

2.1.2 Topic-related definitions

We define a *shortcut* (u,v) as an (additional) edge with the property that $\text{len}(u,v)$ equals the distance between u and v in G. If we add a set of shortcuts E' to the graph (we call E' a *shortcut assignment*), we get a supergraph $G' = (V, E \cup E', \text{len}')$ of G. In this notation, $\text{len}' : E \cup E' \to \mathbb{R}^+$ is given by $\text{len}'(u,v) = \text{dist}(u,v)$ for $(u,v) \in E'$ and $\text{len}'(u,v) = \text{len}(u,v)$ otherwise.

Moreover, we have to give some further notation related to shortest-paths dags that we will use throughout this thesis. For arbitrary x, y and $z \in V$, a *shortest x-y-z-path* is a shortest path from x to z with y on it. Analogously, a *shortest w-x-y-z-path* is a shortest path from w to z passing first x and then y. We denote by $\sigma_{xz}(y)$ the number of shortest x-y-z-paths and by $\eta_{xz}(y)$ the number of hop-minimal shortest x-y-z-paths. Similarly, $\sigma_{wz}(x,y)$ represents the number of shortest w-x-y-z-paths and $\eta_{wz}(x,y)$ the number of hop-minimal shortest w-x-y-z-paths.

Further, let $P^-(x,y) := \{s \in V \mid \exists \text{ shortest } s\text{-}x\text{-}y\text{-path}\}$ and $P^+(x,y) := \{t \in V \mid \exists \text{ shortest } x\text{-}y\text{-}t\text{-path}\}$ denote the sets of start- or end-nodes of shortest paths through x and y. It is easy to see that the nodes in $P^-(x,y)$ lie in a sub-dag of the shortest-paths dag grown from y in $\bar{G}$, whereas the nodes in $P^+(x,y)$ lie in a sub-dag of the shortest-paths dag grown from x in G. The nodes in these sub-dags can be found by performing depth-first or breadth-first search from x or y in the respective dags. This is illustrated in Figure 2.1.

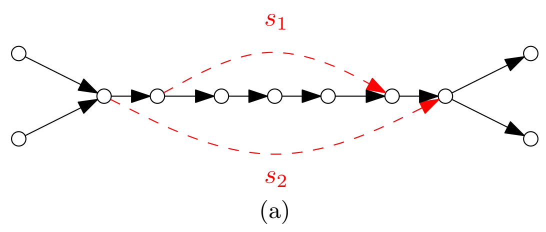


Fig. 2.2: Examples for shortcut assignments with two shortcuts (we assume the edge lengths in the original graph to be uniform): The one illustrated in (b) is better (with respect to the decrease in overall hop lengths), as a simple calculation shows.

Finally, we denote by $\mathrm{In}_s(v)$ the set of inneighbours of v in the shortest-paths dag D_s grown from s and similarly by $\mathrm{Out}_s(v)$ the set of outneighbours of v in D_s. We call w a *successor* of v in D_s, if $w \neq v$ and there exists a shortest s-w-v-path.

2.2 The Best Shortcut Problem

In this section, we describe the topic of this thesis, the Best Shortcut Problem as introduced in [BDDW09]. Moreover, we give a review on some elementary properties of the problem and outline a greedy, suboptimal algorithm for finding shortcut assignments.

2.2.1 Problem Definition and Complexity

Roughly speaking, the Best Shortcut Problem consists of adding a number of shortcuts to a graph, such that the expected number of edges that are contained in an edge-minimal shortest-path from a random node s to a random node t is minimal. Formally, the problem is defined as follows.

Definition 1 (Best Shortcut Problem (BSP)) Given a graph $G = (V, E, \mathrm{len})$ and a positive integer $c \in \mathbb{N}$, find a graph $G' = (V, E \cup E', \mathrm{len}')$ such that $|E'| \leq c$ and

$$w(E') := \sum_{s,t \in V} h(s,t) - \sum_{s,t \in V} h'(s,t)$$

is maximal, whereas $\mathrm{len}' : E \cup E' \rightarrow \mathbb{R}^+$ equals $dist(u, v)$ if $(u, v) \in E'$, equals $\mathrm{len}(u, v)$ otherwise, $h(s, t)$ denotes the hop distance in (V, E) and $h'(s, t)$ denotes the hop distance in $(V, E \cup E')$.

We call $w(E')$ the *decrease in overall hop-lengths* we get by inserting E' in G. Figure 2.2 shows a simple example for two shortcut assignments in the same graph. Taking a closer look at the shortcut assignment in Figure 2.2a, we see that it is not optimal; $w(s_1, s_2)$ is 66, whereas $w(s_3, s_4)$ is 72. The example could suggest that there exists always an optimal solution with shortcuts that do not overlap in the sense that the shortest paths they represent are disjoint. This is clearly not the case, as, if we just set the number of shortcuts c we want to insert high enough, an optimal solution contains all possible shortcuts.

In [BDDW09], Bauer et al. show that the Best Shortcut Problem is *NP*-hard by using a reduction from the Min Set Cover problem. Furthermore, they prove that, unless $P = NP$, there exists no polynomial-time algorithm that approximates BSP up to an additive constant.

2.2.2 The Basic Greedy Strategy

We now describe a greedy algorithm for finding shortcut assignments, that successively adds locally optimal shortcuts to the graph. More formally, we consider a sequence $s_1, \dots, s_c$ of shortcuts and a family of graphs G_i, where G_{i+1} is the result of adding the shortcut s_{i+1} to G_i. G_0 corresponds to the original graph G. We determine the s_i and G_i recursively by setting s_i to be a shortcut which maximizes $w(s)$ in G_i. Pseudocode for this approach is given in Algorithm 1.

Algorithm 1: GREEDY

```
Input: graph $G = (V, E, \mathrm{len})$, number of shortcuts $c$
Output: graph $G' = (V, E \cup E', \mathrm{len}')$ with shortcuts added
1 $G_0 = G := (V, E_0, \mathrm{len}_0)$;
2 forall $i = 1, 2, \dots, c$ do
3     $s_i := \mathrm{argmax}\{w(s) \text{ in } G_i \mid s \in V \times V\}$;
4     $E_i := E_{i-1} \cup \{s_i\}$;
5     $\mathrm{len}_i : E_i \to \mathbb{R}^+$, $\mathrm{len}_i(s_i) := dist(s_i)$ and $\mathrm{len}_i(u, v) := \mathrm{len}_{i-1}(u, v)$ for each
      $(u, v) \in E_{i-1}$, $(u, v) \neq s_i$;
6     $G_i := (V, E_i, \mathrm{len}_i)$
7
8 Output $G' = G_c$.
```

Bauer et al. show that, if G has the property that shortest paths between arbitrary nodes are unique, the output of the GREEDY-strategy is in fact a factor c-approximation of the BSP. Using no optimization, we can trivially determine the value of $w(s)$ in G_i by performing an all-pairs shortest paths computation in G. As we will see in Section 2.3.1, this can be accomplished in $O(n \cdot (n \log n + m))$ time. If we simply repeat this to evaluate all possible shortcuts in G_i, we get an overall time complexity in $O(c \cdot n^3 \cdot (n \log n + m))$. The first part of this thesis shows ways to decrease the costs for finding one optimal shortcut in arbitrary graphs, which can be used in a straightforward way to get lower time bounds for the GREEDY-algorithm.

Note that we cannot count on the uniqueness of shortest paths after several steps of the GREEDY-algorithm, even if the underlying unmodified graph has this property. Inserting an arbitrary shortcut in the graph results in an alternative shortest path for at least the end-nodes of the shortcut. Furthermore, one could ask the question if hop-minimal shortest paths are always unique after the insertion of shortcuts, if this is the

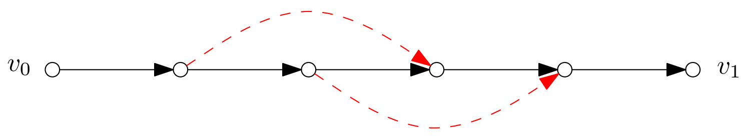


Fig. 2.3: This simple graph shows that the insertion of shortcuts destroys the property that hop-minimal shortest paths are unique (we assume the edge lengths in the underlying graph to be uniform)

case in the underlying graph. This does not hold, as a simple example given in Figure 2.3 illustrates.

In the previous section, we already saw that the shortcut assignment in Figure 2.2a is not optimal. A closer look at this assignment reveals that it is an assignment that could be found by the GREEDY-strategy. In the original graph, we see that $w(s_1) = 48$ is optimal among all possibilities. If we consider the graph we obtain, if we insert s_1 into G, $w(s_2) = 18$ is the highest value we get by considering all potential shortcuts. This illustrates that it is rather easy to construct graphs in which we get non-optimal results by the GREEDY-strategy, even if we add only few shortcuts to the graph.

2.3 The All-Pairs Shortest Paths Problem

As the shortcut problem is closely related to the problem of finding shortest paths in a graph, we give a brief overview of some fundamental algorithms for solving this task. The algorithms presented for finding one optimal shortcut require the computation of the distances and hop-distances of all pairs of nodes as a preprocessing step. This corresponds to the *all-pairs shortest paths problem*.

2.3.1 Dijkstra's algorithm

The presumably most prominent approach for solving shortest paths problems is the algorithm presented by E. W. Dijkstra in 1959 (see [Dij59]). Dijkstra's algorithm solves the *single-source all-targets shortest paths problem*, which is the problem of finding the distance from one arbitrarily chosen node s to all other nodes $t \in V$. As already mentioned, we assume the edge lengths to be nonnegative, which is crucial for the correctness and termination of the algorithm.

It is easy to see that we can solve the all-pairs shortest paths problem in G by simply solving the single-source-all-targets problem for all $s \in V$. Thus, we can trivially determine the distances between all nodes in V by n runs of Dijkstra's algorithm.

The base algorithm proceeds as follows. We store for each node in the graph a status marker indicating exactly one of the states *unvisited*, *visited* and *finished*. Additionally, we use a priority queue Q for storing the set of visited, but not yet finished nodes during the algorithm. The nodes in Q are keyed by the current distance label. The lower this

Algorithm 2: Dijkstra

```
Input: Graph $G = (V, E, \mathrm{len})$, node $s \in V$
Data Structure: Priority Queue $Q$ for nodes, keyed by the current distance labels
Output: $d(s,t)$ for all $t \in V$
forall $t \in V \setminus \{s\}$ do
    $d(s,t) := \infty$ ;
$d(s,s) := 0$ ;
Insert $s$ in $Q$ ;
while $Q \neq \emptyset$ do
    Remove minimal element $v$ from $Q$ ;
    Mark $v$ as finished ;
    forall $e := (v,w) \in E$ do
        if $w$ is not marked as finished then
            Mark $w$ as visited ;
            if $d(s,v) + len(e) < d(s,w)$ then
                $d(s,w) := d(s,v) + \mathrm{len}(e)$ ;
            if $w \notin Q$ then
                Insert $w$ in $Q$ ;
```

key is for a node, the higher is its priority in the queue.

In the beginning, all status markers are set to unvisited and the distances $d(s, v)$ for all nodes $v \in V$ are set to infinity. Then we begin the actual computation by setting $d(s, s)$ to zero, marking s as visited and inserting s in the priority queue. While there are visited nodes, the algorithm removes one of the nodes with minimal distance label from the queue, marks it as finished and then considers all outgoing edges e from this node. If the end-node w of e is not marked as finished, we test if we get a shorter path from s to w via e. If this is the case, we update the distance label of w with respect to s and insert w into Q if w was unvisited before. Pseudocode for Dijkstra's algorithm is given as Algorithm 2.

The correctness of Dijkstra's algorithm relies on the invariant that all nodes marked as finished already have their correct distance label assigned. A complete proof can be found in [Sch03].

Using no special data structures, Dijkstra's algorithm has a running time in $O(n^2)$. Fredman and Tarjan showed that Dijkstra's algorithm can be speeded up to $O(m + n \log n)$ by using a Fibonacci heap for the implementation of the priority queue (see [FT87]).

We should not forget to mention that Dijkstra's algorithm is not the only way to solve the all-pairs shortest paths problem. Another possibility is to use fast matrix multiplication, yielding lower computational complexity for dense graphs (see for example [CLRS01]). As most of the graph classes that we consider in the experiments are rather sparse, and as in the algorithms we use, the asymptotical time complexity is dominated by $\Theta(n^3)$ or $\Theta(n \cdot m)$ subroutines, we solve the problem of finding all distances and hop-distances in a graph by n applications of Dijkstra's algorithm throughout this thesis.

2.3.2 Modifying Dijkstra's algorithm to determine hop-distances and to count shortest s-t-paths

Later on, we will need some extra information about shortest paths beyond the distances in the graph. First, we will need a fast way to compute hop-distances between all pair (s, t) of nodes. Second, we are interested in the number of shortest and hop-minimal shortest paths, σ_{st} and η_{st} between arbitrary nodes s and t. In this section, we show how to adapt Dijkstra's algorithm to additionally compute these values. We just need some minor changes that do not increase the asymptotical time behaviour or the general proceeding of the algorithm. Algorithm 3 shows the pseudocode of these modifications.

To compute the hop-distances, we initialize the hop-distance labels to infinity for all t in $V \setminus \{s\}$ and to 0 for s. Next, if we find a shorter path to a node v, we additionally have to change the hop-distance for v to be the number of hops on the new (temporarily) shortest path. This is done in line 21. Finally, if we find an alternative shortest path to v, we have to test if it has fewer hops than the shortest s-v-paths found up to now. If this is true, we decrease the hop-distance of v, which we do in line 25.

In order to compute σ_{st}, we use the following lemma taken from [Bra01] to justify the correctness of our modifications.

Algorithm 3: HopAndPathCountingDijkstra

```
Input: Graph $G = (V, E, \mathrm{len})$, node $s \in V$
Data Structure: Priority Queue $Q$ for nodes, keyed by the current distance labels
Output: $\mathrm{dist}(s,t)$, $h(s,t)$, $\sigma_{st}$ and $\eta_{st}$ for all $t \in V$
forall $t \in V$ do
    $\mathrm{dist}(s,t) := \infty$ ;
    $h(s,t) := \infty$ ;
    $\sigma_{st} := 0$ ;
    $\eta_{st} := 0$ ;
$\mathrm{dist}(s,s) := 0$ ;
$h(s,s) := 0$ ;
$\sigma_{ss} := 1$ ;
$\eta_{ss} := 1$ ;
Insert $s$ in $Q$ ;
while $Q \neq \emptyset$ do
    Remove minimal element $v$ from $Q$ ;
    Mark $v$ as finished ;
    forall $e := (v,w) \in E$ do
        if $w$ is not marked as finished then
            Mark $w$ as visited ;
            if $dist(s,v) + len(e) < dist(s,w)$ then
                $\sigma_{sw} := \sigma_{sv}$ ;
                $\eta_{sw} := \eta_{sv}$ ;
                $\mathrm{dist}(s,w) := \mathrm{dist}(s,v) + \mathrm{len}(e)$ ;
                $h(s,w) := h(s,v) + 1$ ;
            if $dist(s,v) + len(e) = dist(s,w)$ then
                $\sigma_{sw} := \sigma_{sw} + \sigma_{sv}$ ;
                if $h(s,v) + 1 < h(s,w)$ then
                    $h(s,w) := h(s,v) + 1$ ;
                    $\eta_{sw} := \eta_{sv}$ ;
                if $h(s,v) + 1 = h(s,w)$ then
                    $\eta_{sw} := \eta_{sw} + \eta_{sv}$ ;

            if $w \notin Q$ then
                Insert $w$ in $Q$ ;
```

Lemma 1 (Combinatorial shortest-paths counting) *For $s \neq v \in V$:*

$$\sigma_{sv} = \sum_{u \in \mathrm{In}_s(v)} \sigma_{su}$$

This holds because sub-paths of shortest paths are also shortest paths. Therefore, removing the last edge on a shortest s-t-path, we get a shortest s-w-path for some w in $\mathrm{In}_s(v)$. For a complete proof, see [Bra01].

For the number of hop-minimal shortest paths, we get a similar result; the proof is analogous.

Lemma 2 (Combinatorial hop-minimal shortest-paths counting) *For $s \neq v \in V$:*

$$\eta_{sv} = \sum_{\substack{u \in \mathrm{In}_s(v) \\ h(s,u)+1=h(s,v)}} \eta_{su}$$

We proceed as follows: If, during the execution of Dijkstra's algorithm, a shorter path to a node v is found, all shortest paths found so far are not valid any more. Thus, we have to (re-)set the values for σ_{sv} and η_{sv} to the number of shortest or hop-minimal shortest paths to the start-node of the current edge. In line 22, we test, if we have found alternative shortest paths to v. If this is the case, we have to adjust σ_{sv} according to Lemma 1. Also, we have to check whether η_{sv} is consistent with Lemma 2, if we consider the new hop-distance between s and v. The correctness of the algorithm follows from Lemma 1 and Lemma 2.

2.4 Summing up values in shortest-paths dags

In the algorithms to solve the BSP with exactly one shortcut, we will frequently come across the same sub-problem. In short, we consider a shortest-paths dag D_s grown from a node s. Every node v in D_s has a value $f(v)$ assigned. We now want to add $f(v)$ to each predecessor of v in D_s. More precisely, as result, we aim to compute the values $f_{\mathrm{sum}}(v) = \sum_{w \in P^+(s,v)} f(w)$ for each v in D_s. The values $f(v)$ correspond to different measures. In the most simple setting, we initialize all values to 1, thus as result, we just get the exact cardinality of $P^+(s,v)$ for each v in D_s. If we have precomputed all distances and hop-distances in the graph, we can decide in $O(1)$ whether a node t is in $P^+(s,v)$. For this, we just have to test if $\mathrm{dist}(s,v) + \mathrm{dist}(v,t) = \mathrm{dist}(s,t)$. Thus, considering all pairs of nodes, we get a trivial $O(n^2)$-algorithm.

In this section, we will describe some variants of depth-first-search(DFS) that sum up intermediate results along the paths of shortest-paths dags. As result, we get upper and lower bounds for f_{sum} with a time complexity in $O(m)$.

We first consider the simple case that shortest paths in the underlying graph are unique, implying that all shortest-paths dags form trees. We show that under these assumptions,

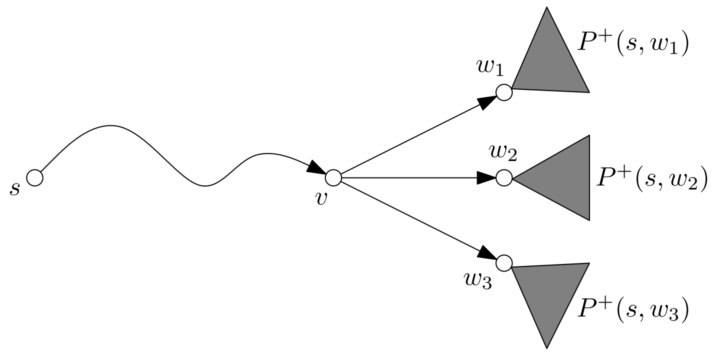


Fig. 2.4: Illustration to Lemma 3: In this case, the shortest-paths dag grown from s is a tree; the sets $P^+(s, w_i)$ are disjoint.

it is possible to determine the exact values of $f_{\text{sum}}(v)$ for one shortest-paths dag in $O(m)$ using DFS.

The first observation we make, is that $f_{\text{sum}}(v)$ obeys the following recursive equation, if we consider shortest-paths trees. Figure 2.4 illustrates the preconditions of the lemma.

Lemma 3 *Let D_s denote the shortest-paths dag grown from a node s in V. If D_s is a tree, then $f_{sum}(v)$ for any node v in D_s satisfies*

$$f_{sum}(v) = f(v) + \sum_{w \in \text{Out}_s(v)} f_{sum}(w)$$

Proof
As the sets $P^+(s, w_i)$ are disjoint, we get

$$\begin{aligned} f_{\text{sum}}(v) &= \sum_{w \in P^+(s,v)} f(w) \\ &= f(v) + \sum_{w \in \text{Out}_s(v)} \sum_{x \in P^+(s,w)} f(x) \\ &= f(v) + \sum_{w \in \text{Out}_s(v)} f_{\text{sum}}(w) \end{aligned}$$

■

Algorithm 4 exploits this relationship to compute all values for $f_{\text{sum}}(v)$ by essentially performing one DFS.

Corollary 1 If there is exactly one shortest path from s to each t in the shortest-paths dag D_s grown from s, then Algorithm 4 computes the exact values of $f_{\text{sum}}(v)$ for any node v in D_s.

Algorithm 4: SumValuesSuccessorsUpperBound

```
Data: shortest-paths dag $D_s = (V, E)$, start-node $s \in V$, $f(v)$ for all $v \in D_s$
Result: Upper bounds $f_{\text{sum,up}}(v)$ for the values of $f_{\text{sum}}(v)$
Init stack ;
forall $v \in D_s$ do
    $f_{\text{sum,up}}(v) := f(v)$ ;
mark($s$) ;
push$\big((\text{NULL}, s)\big)$ ;
while stack is not empty do
    $(p, v) :=$ topmost element in the stack ;
    if there is an unmarked outgoing edge $(v, w)$ then
        mark$\big((v, w)\big)$ ;
        if $w$ is unmarked then
            mark($w$) ;
            push$\big((v, w)\big)$ ;
        else
            Increase $f_{\text{sum,up}}(v)$ by $f_{\text{sum,up}}(w)$ ;

    else
        if $p \neq NULL$ then
            Increase $f_{\text{sum,up}}(p)$ by $f_{\text{sum,up}}(v)$ ;
        pop() ;


```

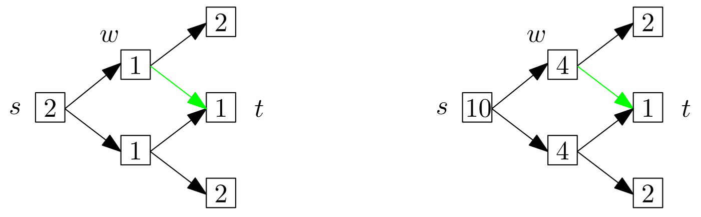


Fig. 2.5: Illustrates the problem that we get if the shortest-paths dag is not a tree: The left side shows an example of a dag with the corresponding values of $f(v)$ for each v, whereas on the right side the values $f_{\text{sum, up}}(v)$ are displayed

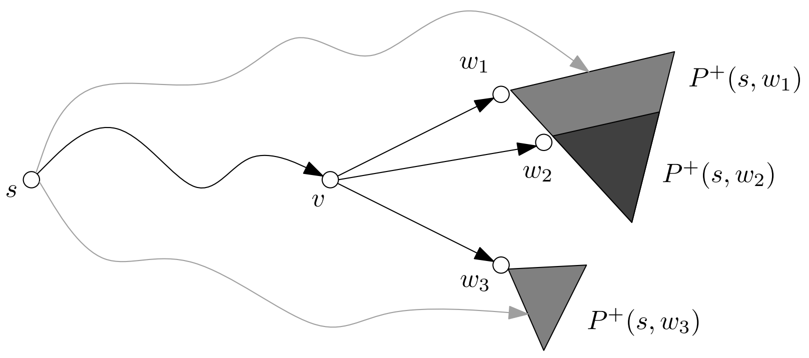


Fig. 2.6: Illustrates the general situation that we have in a shortest-paths dag: the sets $P^+(s, w_i)$ are not necessarily disjoint

Unfortunately, if we apply the same algorithm to general shortest-paths dags, the values of $f_{\text{sum, up}}(v)$ we compute are not equal to $f_{\text{sum}}(v)$. This is illustrated in Figure 2.5, where $f(t)$ is added to $f_{\text{sum}}(s)$ twice. Nevertheless, it is rather easy to see that we get upper bounds for $f_{\text{sum}}(v)$, if we apply Algorithm 4.

More precisely, we get the connection of the following lemma (for illustration, see Figure 2.6).

Lemma 4 *For the values $f_{\textit{sum, up}}(v)$ computed by Algorithm 4, we obtain:*

$$f_{\textit{sum, up}}(v) = \sum_{x \in P^+(s,v)} \sigma_{vx} \cdot f(x)$$

Proof
Let D_s denote the shortest-paths dag grown from s. As $\sum_{w \in P^+(s,v)} \sigma_{vw} \cdot f(w)$ equals $f(v)$ if v is a sink, the values for the sinks are already correctly initialized in the beginning. Thus, it remains to show that we get the correct results for the other nodes in D_s under

the assumption that all outneighbours in D_s get their proper values assigned. We get

$$\begin{aligned}
f_{\text{sum, up}}(v) &= f(v) + \sum_{w \in \text{Out}_s(v)} f_{\text{sum, up}}(w) \\
&= f(v) + \sum_{w \in \text{Out}_s(v)} \sum_{x \in P^+(s,w)} \sigma_{wx} \cdot f(x) \\
&= f(v) + \sum_{x \in P^+(s,v)\setminus\{v\}} \sum_{\substack{w \in \text{Out}_s(v): \\ x \in P^+(s,w)}} \sigma_{wx} \cdot f(x) \\
&= f(v) + \sum_{x \in P^+(s,v)\setminus\{v\}} f(x) \cdot \sum_{\substack{w \in \text{Out}_s(v): \\ x \in P^+(s,w)}} \sigma_{wx} \\
&= f(v) + \sum_{x \in P^+(s,v)\setminus\{v\}} f(x) \cdot \sum_{w \in \text{Out}_s(v)} \sigma_{vx}(w) \\
&= f(v) + \sum_{x \in P^+(s,v)\setminus\{v\}} f(x) \cdot \sigma_{vx} \\
&= \sum_{x \in P^+(s,v)} \sigma_{vx} \cdot f(x)
\end{aligned}$$

■

Therefore, the values $f_{\text{sum, up}}(v)$ are clearly upper bounds for $f_{\text{sum}}(v)$, as σ_{vx} is always greater or equal to 1 for each x in $P^+(s,v)$.

Conversely, if we want to determine lower bounds for $f_{\text{sum}}(v)$, we can use a very simular algorithm. Its pseudocode is given in Algorithm 5. Note that this algorithm is almost completely identical to the one we use for the upper bounds; we just changed two lines because we want to sum up just parts of the values for $f_{\text{sum, low}}(w)$ to its inneighbours. Additionally, we need as input the number of shortest paths from s to every other node in the dag, which can be determined with little overhead while building the shortest-paths dag (see Section 3).

Lemma 5 *For the values $f_{\textit{sum, low}}(v)$ computed by Algorithm 5, we obtain:*

$$f_{\textit{sum, low}}(v) = \sum_{x \in P^+(s,v)} \frac{\sigma_{sx}(v)}{\sigma_{sx}} \cdot f(x)$$

Proof
Analogous to the proof of Lemma 4, we get:

$$f_{\text{sum, low}}(v) = f(v) + \sum_{w \in \text{Out}_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot f_{\text{sum, low}}(w)$$

Algorithm 5: SumValuesSuccessorsLowerBound

```
Input: shortest-paths dag $D_s = (V, E)$, start-node $s \in V$, $f(v)$ and $\sigma_{sv}$ for all $v \in V$
Output: Lower bounds $f_{\text{sum,low}}(v)$ for the values of $f_{\text{sum}}(v)$
1  Init stack ;
2  forall $v \in V$ do
3      $f_{\text{sum,low}}(v) := f(v)$ ;
4  mark($s$) ;
5  push$\big((\text{NULL}, s)\big)$ ;
6  while stack is not empty do
7      $(p, v) :=$ topmost element in the stack ;
8      if there is an unmarked outgoing edge $(v, w)$ then
9          mark$\big((v, w)\big)$ ;
10         if $w$ is unmarked then
11             mark($w$) ;
12             push$\big((v, w)\big)$ ;
13         else
14             Increase $f_{\text{sum,low}}(v)$ by $\frac{\sigma_{sv}}{\sigma_{sw}} \cdot f_{\text{sum,low}}(w)$ ;
15
16     else
17         if $p \neq NULL$ then
18             Increase $f_{\text{sum,low}}(p)$ by $\frac{\sigma_{sv}}{\sigma_{sw}} \cdot f_{\text{sum,low}}(v)$ ;
19         pop() ;
20
21
```

Furthermore, we initialize $f_{\text{sum, low}}(v)$ to $f(v)$, which is already the correct value for the sinks. So, we only need to show that $f_{\text{sum, low}}(v)$ equals the desired value under the assumption that we have already computed the correct values for the outneighbours of v in D_s. This is true, as

$$\begin{aligned}
f_{\text{sum, low}}(v) &= f(v) + \sum_{w \in \text{Out}_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot f_{\text{sum, low}}(w) \\
&= f(v) + \sum_{w \in \text{Out}_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sum_{x \in P^+(s,w)} \frac{\sigma_{sx}(w)}{\sigma_{sx}} \cdot f(x) \\
&= f(v) + \sum_{w \in \text{Out}_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sum_{x \in P^+(s,w)} \frac{\sigma_{sw} \cdot \sigma_{wx}}{\sigma_{sx}} \cdot f(x) \\
&= f(v) + \sum_{x \in P^+(s,v) \setminus \{v\}} \frac{\sigma_{sv}}{\sigma_{sx}} \cdot f(x) \cdot \sum_{\substack{w \in Out_s(v): \\ x \in P^+(s,w)}} \sigma_{wx} \\
&= f(v) + \sum_{x \in P^+(s,v) \setminus \{v\}} \frac{\sigma_{sv}}{\sigma_{sx}} \cdot f(x) \cdot \sum_{w \in Out_s(v)} \sigma_{vx}(w) \\
&= f(v) + \sum_{x \in P^+(s,v) \setminus \{v\}} \frac{\sigma_{sv} \cdot \sigma_{vx}}{\sigma_{sx}} \cdot f(x) \\
&= \sum_{x \in P^+(s,v)} \frac{\sigma_{sx}(v)}{\sigma_{sx}} \cdot f(x)
\end{aligned}$$

■

As $\frac{\sigma_{sx}(v)}{\sigma_{sx}}$ is always less or equal 1, the values $f_{\text{sum, low}}(v)$ are clearly lower bounds for $f_{\text{sum}}(v)$. Furthermore, we see that, just as with the upper bounds, we get the correct values for $f_{\text{sum}}(v)$ if we assume shortest paths to be unique. In this case, $\frac{\sigma_{sx}(v)}{\sigma_{sx}}$ is either 1, if x is in $P^+(s,v)$, or 0 otherwise.

Note that this last algorithm for the lower bound is closely related to the fast algorithm for betweenness centrality proposed by Brandes (see [Bra01]). This is no coincidence, as we will need this variant of DFS to compute the (generalized) betweenness centrality for pairs of nodes.

If we consider not all shortest paths from a given start-node s but only shortest paths that are hop-minimal, we can proceed analogously by just considering the edges in D_s that are on at least one hop-minimal shortest path from s to another node in V. Figure 2.7 shows an example of a shortest-paths dag and highlights the edges that lie on hop-minimal shortest paths (these form a sub-dag of D_s). The algorithms for the upper and lower bounds can be modified using just the edges that are in this reduced shortest-paths dag in a straightforward way. Furthermore, the values for σ_{st} have to be replaced by η_{st} to reflect that we only consider hop-minimal shortest paths. The results we get are the same as with the whole shortest-paths dag, if we replace σ_{st} by η_{st} again.

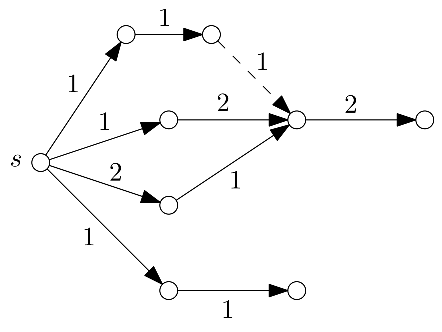


Fig. 2.7: Example for a shortest-paths dag: solid edges are the ones that lie on at least one hop-minimal path starting at s

Note that all results could as easily be obtained by a breadth-first-traversal or by exploiting the topological order in the dag. As the complexity is independent from the method chosen, we focus exclusively on depth-first-traversal, as this is the way we chose in the implementation of some subroutines of our algorithms.

3 Algorithms to Determine One Optimal Shortcut

In this chapter, we will examine the question of how to find exactly one optimal shortcut efficiently without evaluating each possible shortcut using Dijkstra's algorithm.

The contribution of this section is twofold. First, we improve the brute-force algorithm heuristically by reducing the search space, both with and without additional space overhead. Second, we describe a $\Theta(n^3)$ algorithm for finding one optimal shortcut using shared intermediate results.

3.1 Preliminaries

Before we present our algorithms, we have to introduce some further notation. First, let $P(a, b) := \{(s, t) \in V \times V \mid \exists \text{ shortest } s\text{-}a\text{-}b\text{-}t\text{-path}\}$ be the set of pairs of nodes such that there exists a shortest s-t-path that could benefit from the insertion of a shortcut between a and b. Note that $P(a, b)$ is a subset of $P^-(a, b) \times P^+(a, b)$, but in general, the two sets are not equal. Figure 3.1 illustrates this case.

From here on, let $G = (V, E)$ always denote a strongly connected, directed graph. Further, let a and b be arbitrary but fixed nodes in V and $G' = (V, E \cup \{(a, b)\})$ the supergraph of G that we get by inserting a shortcut from a to b. In this context, the notation $h'(s, t)$ indicates the hop-distance between s and t in G'.

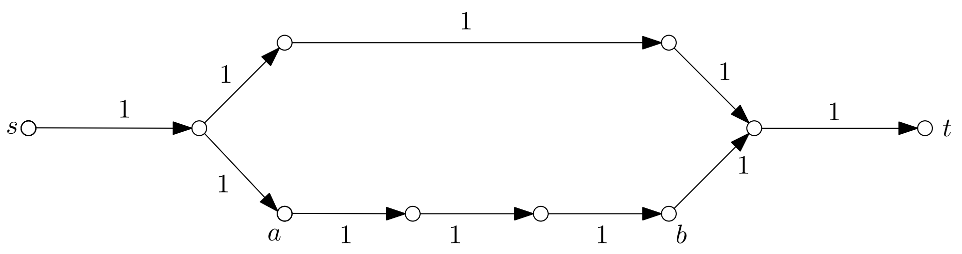


Fig. 3.1: In this example, s is clearly in $P^-(a, b)$ and t in $P^+(a, b)$, but (s, t) is not in $P(a, b)$

Finally, we recall the following crucial observation concerning shortest paths that we will use heavily during the rest of this thesis.

Lemma 6 (Bellman criterion) *A node $v \in V$ lies on a shortest path between vertices s, $t \in V$, if and only if $dist(s,t) = dist(s,v) + dist(v,t)$.*

3.2 Heuristic speed-up with linear space complexity

This section describes our first approach to speed up the brute-force algorithm for finding one optimal shortcut. The idea is to find upper bounds for $w(a,b)$ that are easier to compute than $w(a,b)$ itself.

3.2.1 General proceeding

The first algorithm that we use to heuristically improve the running time for finding one optimal shortcut is outlined as follows. As already mentioned, we will use upper bounds for $w(a,b)$ to exclude shortcuts from being completely evaluated if they are provably non-optimal. Pseudocode for this general approach is given in Algorithm 6. We modify the brute-force algorithm from Section 2.2.2 in two ways. First, we determine for each pair of nodes, if $w(a,b)$ can be greater than the overall decrease in hop-length of the best shortcut found so far using upper bounds. Second, we consider the pairs of nodes in a special order that causes shortcuts that are likely to have high values for $w(a,b)$ to be evaluated in the beginning. The idea behind this is obvious: If we find good shortcuts early, more pairs of nodes can be excluded using the upper bounds.

Of course, this approach makes only sense, if we are able to compute reasonably close upper bounds with less time complexity than is needed to compute the exact value of $w(a,b)$ by solving APSP. Similarly, we have to determine the order in which we consider shortcuts without causing considerable time and space overhead. How this can be done is subject of the next sections.

Algorithm 6: BestShortcutHeuristically

```
   Input: Strongly connected graph $G = (V, E, len)$
   Output: argmax$\{w(s) \mid s \in V \times V\}$
1  DecreaseWithBestSolution := 0 ;
2  forall $(a,b) \in \{(a,b) \in V \times V\}$ with decreasing importance do
3      if $upperbound(a,b) > DecreaseWithBestSolution$ then
4          Determine $w(a,b)$ solving APSP ;
5          if $w(a,b) > DecreaseWithBestSolution$ then
6              DecreaseWithBestSolution := $w(a,b)$ ;
7              $s := (a,b)$ ;
8
9
10 return $s$ ;
```

3.2.2 Upper bounds for the decrease in overall hop length

Our first approach uses the sets $P^-(a,b)$ and $P^+(a,b)$ to determine an upper bound for $w(a,b)$. As $w(a,b) = \sum_{s,t \in V} h(s,t) - h'(s,t)$, we have to consider only the pairs $(s,t) \in V \times V$ for which $h(s,t)$ and $h'(s,t)$ differ. The following lemma states that if there is no shortest s-a-b-t-path, then the hop-distance between s and t cannot be decreased by inserting a shortcut between a and b. This result is rather trivial, but as we can reuse the lemma later on, we give a detailed proof.

Lemma 7 *Let a and b denote arbitrary but fixed nodes. Let $G' = (V, E \cup \{(a,b)\} len')$ be the graph that results from the addition of a shortcut from a to b and h' the hop-distance in G'. For all s, $t \in V$, for which no shortest s-a-b-t path exists, we have*

$$h(s,t) = h'(s,t)$$

Proof
Let p denote a hop-minimal shortest s-t-path in G'. Assume that p contains the edge $e = (a,b)$. Then, as e is a shortcut in G, there is a shortest s-a-b-t path in G, which leads to a contradiction. Therefore, as G and G' differ only in the existence of e, p is also a path in G. Thus, as $h(s,t)$ is always greater or equal to $h'(s,t)$, it follows that $h(s,t) = h'(s,t)$. ■

Additionally, $h(s,t) - h'(s,t)$ is bounded by $h(a,b) - 1$. If we put all these observations together, we can estimate $w(a,b)$ as in the following lemma. Figure 3.2 illustrates the idea.

Lemma 8

$$w(a,b) \leq |P^-(a,b)| \cdot |P^+(a,b)| \cdot (h(a,b) - 1)$$

Proof

$$\begin{aligned}
w(a,b) &= \sum_{s,t \in V} \big(h(s,t) - h'(s,t)\big) \\
&= \sum_{s,t \in P(a,b)} \big(h(s,t) - h'(s,t)\big) + \sum_{s,t \notin P(a,b)} \underbrace{\big(h(s,t) - h'(s,t)\big)}_{=0 \text{ (Lemma 7)}} \\
&\leq \sum_{s,t \in P(a,b)} \big(h(a,b) - 1\big)) \\
&= |P(a,b)| \cdot \big(h(a,b) - 1\big) \\
&\leq |P^+(a,b)| \cdot |P^-(a,b)| \cdot \big(h(a,b) - 1\big)
\end{aligned}$$

■

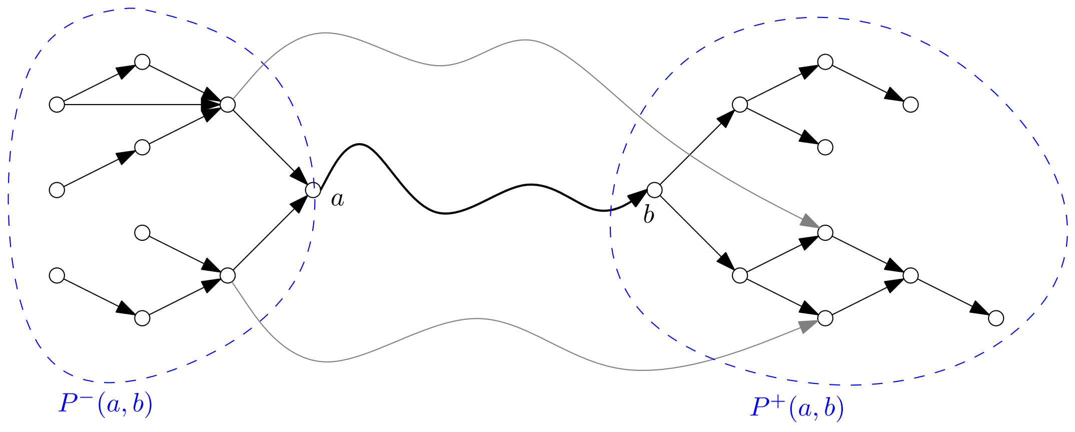


Fig. 3.2: Illustration to Lemma 8

Of course, upper bounds are only valuable if they can be determined faster than the exact values for $w(a,b)$. If we use a simple way to compute $w(a,b)$ like solving APSP with n runs of Dijkstra's algorithm, we need $O(n(n \log n + m))$ time. In contrast to that, the cardinalities of $P^+(a,b)$ and $P^-(a,b)$ can be determined easily by a depth-first search from b in D_a or from a in $\overline{D_b}$, respectively. Thus, we have to solve the single-source shortest paths problem two times, followed by two depth-first traversals, yielding an overall time complexity in $O(n \log n + m)$, whereas the space requirement stays linear in the size of the graph. The value for $h(a,b)$ is a “side product” of solving the single-source shortest paths problem for a, if we use a modified Dijkstra that also determines the hop-distances as described in Section 2.3.2.

Therefore, if we get an upper bound for $w(a,b)$ that is worse than the exact decrease in overall hop length of the best shortcut found so far, we can omit the evaluation of the current shortcut, saving a factor of n by considering this shortcut. On the other hand, if this is not the case, the additional time complexity for computing the bound is clearly dominated by the time needed to solve APSP. Thus, in the worst case, the running time should not be much higher than without using these bounds, while we have a good chance to decrease the overall running time considerably, if the bounds prove to be able to exclude a large percentage of the shortcuts without solving APSP.

3.2.3 Centrality measures for nodes

Centrality indices have been introduced to quantify the “importance” or centrality of nodes in a given graph or network. This is a rather fuzzy definition and there are a lot of different ways to measure the importance of nodes suited for certain applications. The research on this topic dates back to the 1950s, where simple measures as the degree centrality were first introduced.

In this section, we will give a brief description of the centrality measures we use to preselect start- and end-nodes of shortcuts heuristically with the purpose of determining shortcuts (a, b) that are likely to have high values for $w(a, b)$.

The most simple centrality is the *degree centrality* $C_D(v)$. As we deal with directed graphs, we have to specify which kind of degree we are using. We choose the sum of the out- and indegree and get the following definition:

$$C_D(v) = d^+(v) + d^-(v)$$

This is a local measure, as the degree centrality of a node is only determined by its neighbours. It is nonetheless interesting whether such a simple and easy to compute centrality index is suitable to additionally decrease the time to compute one optimal shortcut.

As the values $w(a, b)$ are determined by the shortest paths through a and b, we now focus on two simple centrality measures based on the set of shortest paths in G. The first one is the *betweenness centrality* $C_B(v)$ introduced by Anthonisse in 1971 (see [Ant71]) and Freeman in 1977 (see [Fre77]). Let δ_{st} denote the fraction of shortest paths between s and t that contain a node v:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$$

The ratios $\delta_{st}(v)$ can be interpreted as the probability that v is on a shortest path between s and t chosen uniformly at random among all shortest s-t-paths. With this at hand, we can define the betweenness centrality of a node v as

$$C_B(v) = \sum_{s \neq v \in V} \sum_{t \neq v \in V} \delta_{st}(v)$$

In 2001, Brandes proposed an algorithm to compute the betweenness centrality for all nodes in a weighted graph with a time complexity in $O(n \cdot (n \log n + m))$ and with linear space requirement (see [Bra01]).

The third centrality index that we use is also based on shortest paths. It was introduced by Gutman in 2004 (see [Gut04]) and is called the *reach centrality* $r(v)$. To define this measure, we first have to introduce the notion of the *reach* $r(v, P)$ of a node v on a path $P = (s = u_1, \ldots, v = u_i, \ldots, t = u_l)$ containing v. We define

$$r(v, P) := \min\{len(P_1 = (s = u_1, \ldots, v = u_i), len(P_2 = (v = u_i, \ldots, t = u_l)\}$$

as the minimum of the distances between s and v and v and t following P. Now we can define the reach of v in G by

$$r(v) = \max\{r(v, Q) \mid \text{Q is a shortest path containing } v\}$$

As mentioned in [Gut04], reach centralities for all nodes in a graph can be computed using an APSP-algorithm with an overall time complexity in $O(n \cdot (n \log n + m))$.

3.2.4 Notes on the complexity of using node centrality to preselect shortcuts

Computing the centrality indices betweenness or reach can be accomplished in $O(n \cdot (n \log n + m))$ time and linear space. If we use the degree criterion to measure the importance of nodes, we need $O(m)$ time. The most natural way to exploit node centralities would be to use a generalized centrality measure for pairs of nodes, for example the sum of the node centrality indices. Using this, we could build a list of all possible shortcuts and sort them according to decreasing pair centrality. This would take $O(n^2 \log n)$ time with an arbitrary, efficient sorting algorithm, being dominated by the computation of the bounds for all n^2 shortcuts. But storing the list of all possible shortcuts would cause a space complexity in $O(n^2)$. To avoid this, we use a much simpler approach: We sort the nodes according to decreasing centrality and assign an index to them that corresponds to the position in the sorted list. Then, we enumerate all pairs of nodes (s, t) in increasing order of the sums of the indices of s and t in the list. That means, we consider the shortcuts in the order

$$(L[0], L[0]), (L[0], L[1]), (L[1], L[0]), (L[0], L[2]), \ldots.$$

This can be done in linear time and with linear space. Thus, the additional overhead caused by using the mentioned preselection criteria is dominated by the computation of the upper bounds. So, all in all, in the worst case, we cannot exclude any shortcuts in advance, but we do not add considerable time consumption to the base algorithm, whereas we save a factor of n in the evaluation of each shortcut s that can be dismissed according to its upper bound for $w(s)$. Detailed pseudocode of the whole algorithm is given in the appendix as Algorithm 14.

3.3 Heuristic speed-up with additional space consumption

In this section, we will study the question of how to reduce the time complexity for pruning by allowing an increased memory consumption in $O(n^2)$. We get an algorithm that is very similar to the one in the previous section. The difference is that we compute the upper bounds for all $w(a, b)$ in advance, with a time complexity in $O(n \cdot (n \log n + m))$. Unfortunately, the upper bounds that we get are not as tight as the ones in the previous section, because we cannot compute the correct cardinalities of $P^+(a, b)$ and $P^-(a, b)$ but have to use upper bounds instead. To do this, we use Algorithm 4 from Page 25. For each possible start-node a of a shortcut we build the shortest-paths dag D_a. Then, we assign to every node b in the dag the value $f(b) = 1$. According to Lemma 4, the values $f_{\text{sum,up}}$ that the algorithm computes with this initialization equal $\sum_{x \in P^+(a,b)} \sigma_{bx}$. Therefore, we get upper bounds $\overline{|P^+(a, b)|}$ for $|P^+(a, b)|$ for all (a, b) in $V \times V$. The upper bounds for $|P^-(a, b)|$ can be determined analogously. The pseudocode of the whole algorithm can be found in the appendix as Algorithm 15.

If we compare this algorithm to the one in the previous section, it is not easy to say, which one performs better with respect to computation time. Algorithm 15 is able to compute the upper bounds with an overall time complexity in $O(n\cdot(n\log n+m))$, whereas Algorithm 14 needs $O(n^2 \cdot (n \log n + m)$ time to compute the bounds all in all. On the other hand, the bounds used in Algorithm 14 are tighter and potentially able to exclude more shortcuts in advance.

In general, we could guess that Algorithm 15 is better suited, if shortest paths are "nearly unique", as in this case, the values for $|P^+(a,b)|$ and $\overline{|P^+(a,b)|}$ (as well as $|P^-(a,b)|$ and $\overline{|P^-(a,b)|}$) almost coincide. The question which algorithm to use also heavily depends on the question, which ratio of the shortcuts can be excluded using the upper bounds and on the size of the considered graph. This has to be evaluated experimentally, which is done in Section 5.

3.4 A $\Theta(n^3)$-Algorithm for Finding one Optimal Shortcut

In this section, we describe a fast algorithm for solving the BSP with exactly one shortcut. To this end, we first consider the special case that shortest paths in the underlying graph are unique. Then we outline a simple algorithm for finding one optimal shortcut in this case and point out the problems we get when we try to apply the same algorithm to general graphs. To overcome these problems, we take a closer look at how the insertion of a shortcut can decrease the hop-distance between two nodes. Using these results, we describe an algorithm that computes the values $w(a,b)$ for all nodes a and b in the graph with a running time in $\Theta(n^3)$.

3.4.1 Special case: Unique shortest paths

For graphs with unique shortest paths, we observe that every shortest path is already a hop-minimal shortest path. Hence, in this case, for every pair of nodes for which a shortest s-a-b-t-path exists, the following equation holds:

$$h(s,t) - h'(s,t) = h(a,b) - 1.$$

This simplifies our problem a lot. For the remainder of this section, we assume all distances in the graph to be precomputed using n runs of Dijkstra's algorithm. Our aim is to compute $w(a,b)$ for each pair of nodes (a,b). Exploiting the above equation, we get $w(a,b) = (h(a,b) - 1) \cdot |P(a,b)|$. If we would consider all pairs of nodes and test if

$$\mathrm{dist}(s,a) + \mathrm{dist}(a,b) + \mathrm{dist}(b,t) = \mathrm{dist}(s,t),$$

we could determine $|P(a,b)|$ in $O(n^2)$ running time for fixed a and b. Thus, altogether, we would stay in time $O(n^4)$.

Instead of that, we consider all shortest-paths dags one by one. In the shortest-paths dag D_s grown from s, we determine for every a and b the number of nodes t for which there exists a shortest s-a-b-t-path. For this, we exploit the fact that the existence of a shortest s-a-b-t-path is equivalent to the existence of a shortest s-a-b- and a shortest s-b-t-path.

Lemma 9 *If shortest paths are unique, then for each choice of nodes a and b in the graph, the following equation holds:*

$$w(a,b) = \sum_{s \in P^-(a,b)} (h(a,b) - 1) \cdot |P^+(s,b)|$$

Proof

$$\begin{aligned} w(a,b) = \sum_{s,t \in V} h(s,t) - h'(s,t) \qquad &= \sum_{(s,t) \in P(a,b)} h(a,b) - 1 \\ = \sum_{s \in P^-(a,b)} \sum_{t \in P^+(s,b)} h(a,b) - 1 &= \sum_{s \in P^-(a,b)} (h(a,b) - 1) \cdot |P^+(s,b)| \end{aligned}$$

■

In the first step, we determine $|P^+(s,b)|$ for each b in the dag. As we assume shortest paths to be unique, we can either apply one of the algorithms from Section 2.4 or use the precomputed distances. This yields a time complexity in $O(m)$ or $O(n^2)$, respectively. The latter approach is also feasible in the case of non-unique shortest paths.

In the second step, we test for each possible start-node of a shortcut a and each b, if there exists a shortest s-a-b-path using the Bellman criterion. If this is the case, we add $(h(a,b) - 1) \cdot |P^+(s,b)|$ to $w(a,b)$. The values $w(a,b)$ have to be initialized to zero in the beginning.

We repeat this process for every shortest-paths dag. The correctness of this algorithm follows from Lemma 9. Precomputing the distances and hop-distances in the graph can be done in $O(n \cdot (n \log n + m))$ time, while the overall time complexity for evaluating each dag is in $\Theta(n^3)$. In summary, for the special case that shortest paths are unique, we can easily compute the values $w(a,b)$ for all a and b in $\Theta(n^3)$.

3.4.2 Additional notation

If we try to generalize this approach to work with arbitrary graphs, we encounter several problems. First, in the general case, the existence of a shortest s-a-b-t-path does not force $h(s,t)$ to decrease by $h(a,b) - 1$ if we insert a shortcut between a and b. As illustrated in Figure 3.3, it is possible that the insertion of a shortcut decreases the hop-distance between s and t by less than $h(a,b) - 1$ or not at all.

We now introduce two terms that reflect this observation. The first one is the *offset* of b with respect to s and t, which we define as

$$o_b(s,t) := h(s,b) + h(b,t) - h(s,t).$$

Informally, the offset is a measure for the increase of the hop-distance between s and t, if we restrict ourselves to shortest paths via b (see Figure 3.3 for an example).

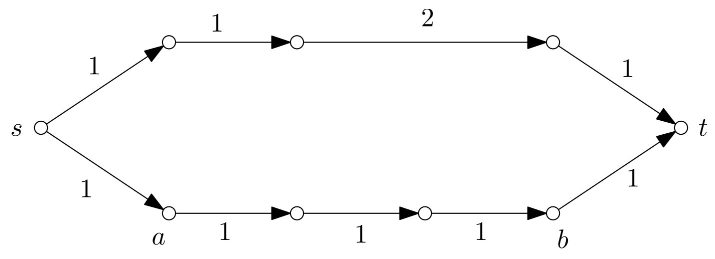


Fig. 3.3: An example for the case that the number of hops on the (sole) s-a-b-t-path is greater than $h(s,t)$, but inserting a shortcut between a and b decreases $h(s,t)$. The offset of b with respect to s and t equals 1 and the offset of b with respect to s and b equals 0. Thus, the values for $\Delta_0(s,b)$ and $\Delta_1(s,b)$ are both 1.

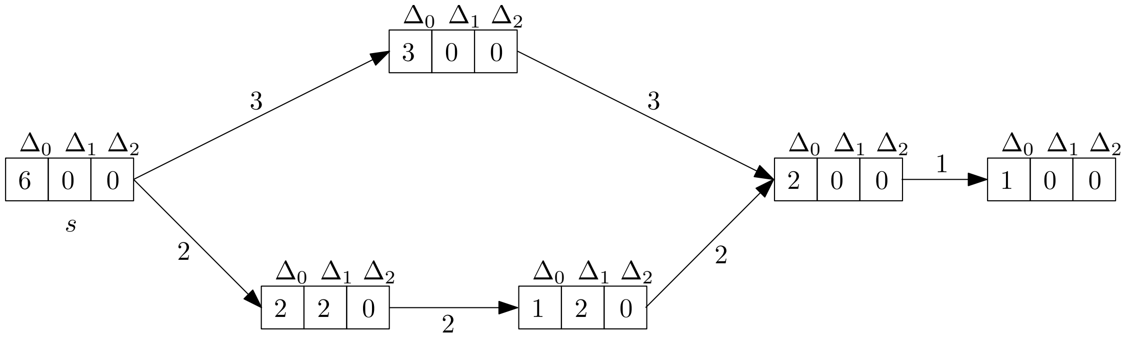


Fig. 3.4: A shortest-path dag in G grown from node s : for every node b in the dag, the corresponding values for $\Delta_i(s,b)$ for $i \in \{0,1,2\}$ are shown

Further, we divide the nodes t in $P^+(s,b)$ in equivalence classes with respect to o_b. As we will see later on, the number of nodes in these equivalence classes is of particular interest. Accordingly, we define

$$\Delta_i(s,b) := |\{t \in P^+(s,b) \mid \; o_b(s,t) = i\}|.$$

To get the idea behind this definition, we have a second look at the graph in Figure 3.3. If we insert a shortcut from a to b, this decreases the overall hop length restricted to the shortest paths starting at s by $(h(a,b)-1)\cdot\Delta_0(s,b)+(h(a,b)-2)\cdot\Delta_1(s,b)=3$. As another example, Figure 3.4 shows a shortest-path dag grown from a node s and for every node b in the dag the corresponding values for $\Delta_i(s,b)$. If we consider general graphs, the $\Delta_i(s,b)$ will play a similar role as $|P^+(s,b)|$ did in the algorithm that we used for the case of unique shortest paths.

As we can see later on, the algorithm we propose makes use of partial (weighted) sums of the $\Delta_i(s,b)$ for fixed s and b in V. So, for convenience, we introduce two further

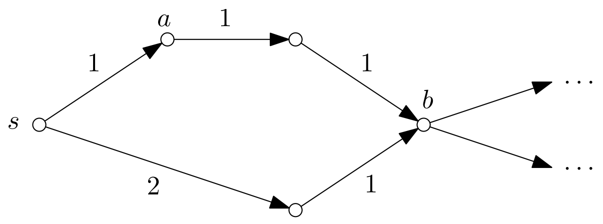


Fig. 3.5: An example for the case that a is a predecessor of b in D_s, but on no hop-minimal shortest s-a-b-path

abbreviations

$$C_r(s,b) := \sum_{i=0}^{r} \Delta_i(s,b)$$

and

$$D_r(s,b) := \sum_{i=0}^{r} i \cdot \Delta_i(s,b).$$

The second problem we possibly come across is, that no hop-minimal shortest s-b-path containing a has to exist, even if there is a shortest s-a-b-path. As with the offset, this means that the insertion of a shortcut between a and b can decrease the hop-distance between s and some t in $P^+(s,b)$, but by less than $h(a,b)-1$. We define the *potential gain* $g_s(a,b)$ of a shortcut from a to b with respect to s as

$$g_s(a,b) := h(s,b) - h(s,a) - 1$$

This is an upper bound for the decrease of the hop-distance between s and any t in the graph. Note that $g_s(a,b)$ equals $h(s,b) - h'(s,b)$, if there exists a shortest s-a-b-path in G and if $g_s(a,b)$ is nonnegative.

3.4.3 Expressing the decrease in hop-distance in terms of gain and offset

Here, we will state the accurate relationship between gain, offset and the decrease in hop-distance for arbitrary pairs of nodes.

As before, we denote by a and b two arbitrary but fixed nodes in the graph, that will represent the start- and end-node of a shortcut. We then consider two arbitrary nodes s and t and observe, how the hop-distance between s and t changes due to the insertion of a shortcut between a and b. To this end, we examine three different cases: First, we consider the case that there exists a shortest s-a-b-t-path and $g_s(a,b)$ is greater than $o_b(s,t)$ (Lemma 10). Then, we continue with the case that there is a shortest s-a-b-t-path, but $g_s(a,b)$ is less than or equal to $o_b(s,t)$ (Lemma 11). For the sake of completeness, we terminate with the case that there is no shortest s-a-b-t-path.

If there is a shortest s-a-b-t-path, we can easily compute the decrease in $h(s,t)$ using precomputed hop-distances by determining the maximum of $h(s,t) - h(s,a) - 1 - h(b,t)$

and zero. Finding an alternative equation for $h(s,t) - h'(s,t)$ using gain and offset might seem unintuitial. The overall idea behind this is to use intermediary results depending on three nodes that shortcuts with the same end-node have in common.

Lemma 10 *For all s, $t \in V$ with the properties that there is a shortest s-a-b-t-path in G and $o_b(s,t) < g_s(a,b)$:*

$$h(s,t) - h'(s,t) = g_s(a,b) - o_b(s,t)$$

Proof
Let p denote a hop-minimal shortest path between s and t in G'. Assume that p does not contain the edge $e = (a,b)$.

$$\begin{aligned}
\Rightarrow \qquad h'(s,t) &= h(s,t) \\
\Rightarrow \qquad o_b(s,t) &= h(s,b) + h(b,t) - h(s,t) \\
&= h(s,b) + h(b,t) - h'(s,t) \\
&= h(s,b) - h(s,a) - 1 + 1 + h(s,a) + h(b,t) - h'(s,t) \\
&= \underbrace{h(s,b) - h(s,a) - 1}_{=g_s(a,b)} + \underbrace{h(s,a) + 1 + h(b,t) - h'(s,t)}_{\geq 0,\ \exists \text{sh. } s\text{-}a\text{-}b\text{-}t\text{-path}} \\
&\geq g_s(a,b)
\end{aligned}$$

This is contradicts our assumptions. Therefore, without loss of generality, we can assume that e is on p.

$$\begin{aligned}
\Rightarrow \qquad h'(s,t) &= h'(s,a) + 1 + h'(b,t) \\
&= h(s,a) + 1 + h(b,t) \\
\Rightarrow \qquad h(s,t) - h'(s,t) &= h(s,t) - h(s,a) - 1 - h(b,t) + h(s,b) - h(s,b) \\
&= h(s,b) - h(s,a) - 1 + h(s,t) - h(b,t) - h(s,b) \\
&= \underbrace{h(s,b) - h(s,a) - 1}_{g_s(a,b)} - \underbrace{(h(s,b) + h(b,t) - h(s,t))}_{=o_b(s,t)} \\
&= g_s(a,b) - o_b(s,t)
\end{aligned}$$

■

The next lemma deals with the case that $g_s(a,b)$ is less or equal than $o_b(s,t)$ and states that under this condition, the hop-distance between s and t does not decrease if we insert a shortcut between a and b.

Lemma 11 *For all s and $t \in V$ with $o_b(s,t) \geq g_s(a,b)$:*

$$h(s,t) = h'(s,t)$$

Proof

$$\begin{aligned} & o_b(s,t) \geq g_s(a,b) \\ \iff \quad & h(s,b) + h(b,t) - h(s,t) \geq h(s,b) - h(s,a) - 1 \\ \iff \quad & h(s,a) + 1 + h(b,t) \geq h(s,t) \end{aligned}$$

From this, it follows that every hop-minimal shortest s-t-path in G' using $e = (a,b)$ has more or the same number of hops than a hop-minimal shortest s-t-path in G. As G is a subgraph of G' and $h'(s,t)$ is always less or equal to $h(s,t)$, we conclude that $h(s,t) = h'(s,t)$.

■

The last case is somewhat trivial, we just have have to recall that if there is no shortest s-a-b-t-path, then the hop-distance between s and t can not be decreased by inserting a shortcut between a and b. This is exactly what Lemma 7 from Section 3.2.2 states.

3.4.4 The main algorithm

As we have dealt with all the relevant possible ways a, b and t can be located with respect to each other in the shortest-path dag grown from s by now, we are ready to present an alternative equation for $w(a,b)$ using partial sums of the $\Delta_i(s,b)$.

Lemma 12

$$w(a,b) = \sum_{\substack{s \in P^-(a,b) \\ g_s(a,b) > 0}} \Big(g_s(a,b) \cdot C_{g_s(a,b)-1}(s,b) - D_{g_s(a,b)-1}(s,b) \Big)$$

Proof

$$
\begin{aligned}
w(a,b) &= \sum_{s,t\in V} \big(h(s,t) - h'(s,t)\big) \\
&= \sum_{(s,t)\in P(a,b)} \big(h(s,t) - h'(s,t)\big) + \sum_{(s,t)\notin P(a,b)} \underbrace{\big(h(s,t) - h'(s,t)\big)}_{=0 \text{ (Lemma 7)}} \\
&= \sum_{\substack{(s,t)\in P(a,b)\\ o_b(s,t)<g_s(a,b)}} \big(h(s,t) - h'(s,t)\big) + \sum_{\substack{(s,t)\in P(a,b)\\ o_b(s,t)\geq g_s(a,b)}} \underbrace{\big(h(s,t) - h'(s,t)\big)}_{=0 \text{ (Lemma 11)}} \\
&\overset{\text{Lemma 10}}{=} \sum_{\substack{(s,t)\in P(a,b)\\ o_b(s,t)<g_s(a,b)}} h(s,b) - h(s,a) - 1 - o_b(s,t) \\
&\overset{o_b(s,t)\geq 0}{=} \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \sum_{i=0}^{g_s(a,b)-1} \sum_{\substack{t\in P^+(s,b)\\ o_b(s,t)=i}} \underbrace{h(s,b) - h(s,a) - 1}_{=g_s(a,b)} - i \\
&= \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \sum_{i=0}^{g_s(a,b)-1} \Delta_i(s,b)\cdot\big(g_s(a,b) - i\big) \\
&= \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \Big(g_s(a,b)\cdot \sum_{i=0}^{g_s(a,b)-1} \Delta_i(s,b) - \sum_{i=0}^{g_s(a,b)-1} \big(i\cdot\Delta_i(s,b)\big)\Big) \\
&= \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \Big(g_s(a,b)\cdot C_{g_s(a,b)-1}(s,b) - D_{g_s(a,b)-1}(s,b)\Big)
\end{aligned}
$$

■

Lemma 12 gives us a possibility to evaluate all potential shortcuts together by considering the shortest-path dags in the graph one by one. For each $s \in V$ we determine the set of shortcuts (a,b) with the property that s is in $P^-(a,b)$. Subsequently, for each shortcut with start-node a and end-node b in this set, we determine the overall decrease in hop lengths that we would get by inserting it, restricted to the shortest-path dag grown from s. If we sum up these decreases for all shortest-path dags, we get the correct value for $w(a,b)$. Using this approach, we can share intermediary results that shortcuts with the same end-node have in common, which considerably reduces computational complexity.

The algorithm we propose is outlined as follows. First, we precompute all distances and hop-distances in the graph using n runs of Dijkstra's algorithm. From this point on, we consider all shortest-path dags successively. Let us denote the respective root-node be

s. In a first step, we compute all relevant $\Delta_i(s, b)$. The pseudocode for this is given in Algorithm 7.

Algorithm 7: ComputeDeltas

```
Input: Strongly connected graph $G = (V, E)$, node $s \in V$, precomputed distances
       and hop-distances in $G$
Output: $\Delta_i(s, b)$ for all $b \in V$, $i \in [0, \dots, n-1]$
1  forall $t \in V$ do
2      for $i \in [0, \dots, n-1]$ do
3          Set $\Delta_i(s, b) := 0$ ;
4
5
6  forall $b, t \in V$ do
7      if there exists a shortest $s$-$b$-$t$-path in $G$ then
8          Compute $j := o_b(s, t)$ ;
9          Increment $\Delta_j(s, b)$ by one ;
10
11
```

Lemma 13 *Algorithm 7 computes the correct values of $\Delta_i(s, b)$ for all $b \in V$ and $i \in [0, \dots, n-1]$ and has a time complexity in $\Theta(n^2)$ while consuming $\Theta(n^2)$ space.*

Proof
Recall that $\Delta_i(s, b) = |\{t \in P^+(s, b) \mid o_b(s, t) = i\}|$. The correctness of the results follows directly from this definition.

As we have precomputed all distances in the graph, the condition in line 7 can be tested in constant time using the Bellman criterion. Similarly, as all hop-distances are given, computing $o_b(s, t)$ takes constant time. For fixed s, we just have to to store n^2 values $\Delta_j(s, b)$, thus, altogether, the time and space complexity is in $\Theta(n^2)$. ■

In the second step we determine partial sums of the $\Delta_i(s, b)$; we proceed as in Algorithm 8.

Lemma 14 *Algorithm 8 computes the correct values for $C_r(s, b)$ and $D_r(s, b)$ for all $b \in V$ and $r \in [0, \dots, n-1]$ and has a time complexity in $\Theta(n^2)$ while consuming $\Theta(n^2)$ space.*

Proof
Recall that $C_r(s, b) = \sum_{i=0}^{r} \Delta_i(s, b)$. This is the same as $\sum_{i=0}^{r-1} \Delta_i(s, b) + \Delta_r(s, b) = C_{r-1}(s, b) + \Delta_r(s, b)$ for $r > 0$. So we can easily compute the partial sums recursively, which is exactly how the algorithm works. The same holds for the computation of the $D_r(s, b)$.

As we have precomputed all relevant values, it is easy to see that the assertions concerning time and space complexity are correct. ■

Algorithm 8: ComputePartialSums

```
Input: Strongly connected graph $G = (V, E)$, node $s \in V$, the values $\Delta_i(s, b)$ for all
       $b \in V$, $i \in [0, \dots, n-1]$
Output: $C_r(s, b)$ and $D_r(s, b)$ for all $b \in V$, $r \in [0, \dots, n-1]$
forall $b \in V$ do
    Set $C_0(s, b) = \Delta_0(s, b)$ and $D_0(s, b) = 0$ ;
    for $r \in [1, \dots, n-1]$ do
        Set $C_r(s, b) = C_{r-1}(s, b) + \Delta_r(s, b)$ ;
        Set $D_r(s, b) = D_{r-1}(s, b) + r \cdot \Delta_r(s, b)$ ;
```

With these sub-algorithms, we are finally able to formulate the main result of this chapter, a $\Theta(n^3)$-algorithm for computing $w(a, b)$ for all a, $b \in V$. The pseudocode for this is given as Algorithm 9.

Algorithm 9: DecreaseInOverallHopLengths

```
Input: Strongly connected graph $G = (V, E)$
Output: The values $w(a, b)$ for all $a, b \in V$
forall $v \in V$ do
    Distances($v$), HopDistances($v$) = HopcountingDijkstra($G$, $v$) ;
forall $a, b \in V$ do
    Set $w(a, b) := 0$ ;
forall $s \in V$ do
    $\Delta_i(s, b)$ := ComputeDeltas($G$, $s$, Distances, HopDistances) ;
    $C_r(s, b)$, $D_r(s, b)$ := ComputePartialSums($G$, $s$, $\Delta_i(s, b)$) ;
    forall $a, b \in V$ do
        if *there exists a shortest s-a-b-path* then
            if $g_s(a, b) > 0$ then
                Increment $w(a, b)$ by $g_s(a, b) \cdot C_{g_s(a,b)-1}(s, b) - D_{g_s(a,b)-1}(s, b)$ ;
```

Proposition 1 Algorithm 9 is correct and has a time complexity in $\Theta(n^3)$ while consuming $\Theta(n^2)$ space.

Proof
The correctness of the algorithm follows directy from Lemma 12. Using Lemma 13 and Lemma 14, we know that computing the relevant values $\Delta_i(s, b)$, $C_r(s, b)$ and $D_r(s, b)$ for all nodes s and b in the graph is in $\Theta(n^3)$, if we use the algorithms introduced above.

Again, evaluating the condition in Line 9 takes constant time if we use the Bellman criterium. As we have precomputed all hop-distances and the values for the partial sums, incrementing $w(a, b)$ by the respective amount is in $O(1)$. Thus, altogether, we get a time complexity in $\Theta(n^3)$.

To get along with $O(n^2)$ memory, we have to be a little bit more careful. If we memorize all the values for $\Delta_i(s, b)$, $C_r(s, b)$ and $D_r(s, b)$ we compute during the algorithm, we need $\Theta(n^3)$ space. But note that in each execution of the loop starting at line 5, we only need the respective values for the current node s. Thus, at the end of the loop, we can free the memory needed by these intermediary results; so all in all, we need only $\Theta(n^2)$ space. ■

4 Pair Centrality Indices

Until now, we tried to reduce computational complexity for finding one optimal shortcut. Our main result was an algorithm that evaluates all potential shortcuts together with a time complexity in $\Theta(n^3)$. Compared with the time to solve this task with a brute-force algorithm, this is a major improvement.

Nonetheless, if we consider larger graphs, a time complexity in $\Theta(n^3)$ can be prohibitive. From this arises the question, whether there are heuristics for finding one reasonably good shortcut with less time complexity. The idea is to rate shortcuts using centrality indices and to pick the shortcut rated best as heuristic for the BSP restricted to one shortcut.

4.1 Betweenness and stress for nodes and edges

In Section 3.2.3 we already described node betweenness and gave a brief justification why this measure can be useful to estimate the quality of shortcuts. There, we rated shortcuts best if the betweenness values of the (non-adjacent) end-nodes were maximal. Another centrality index related to shortest paths is the *stress centrality* $C_S(v)$ of a node v:

$$C_S(v) = \sum_{s,t \in V} \sigma_{st}(v) \qquad \text{(stress centrality of a node } v\text{)}$$

Betweenness and stress can be applied to edges in the following way:

$$C_S(e) = \sum_{s,t \in V} \sigma_{st}(e) \qquad \text{(stress centrality of an edge } e\text{)}$$

$$C_B(e) = \sum_{s,t \in V} \frac{\sigma_{st}(e)}{\sigma_{st}} \qquad \text{(betweenness centrality of an edge } e\text{)}$$

Here, $\sigma_{st}(e)$ denotes the number of shortest s-t-paths that use an edge e. To compute edge betweenness, some minor changes to Brandes' algorithm suffice.

In many cases, node betweenness performs tolerably well as heuristic for the BSP restricted to few shortcuts. On the other hand, Figure 4.1 shows an extreme example of

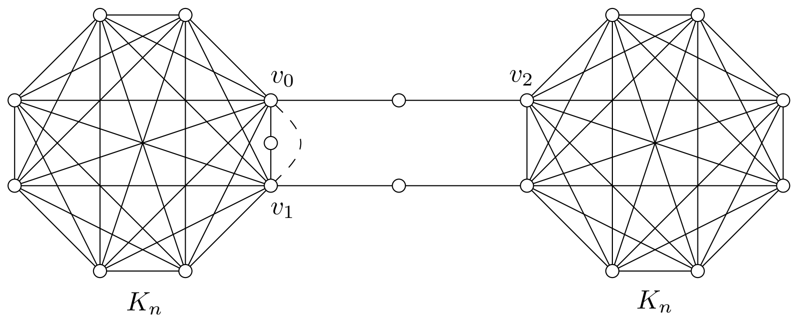


Fig. 4.1: An example for a case, where node betweenness is inappropriate: The shortcut from v_0 to v_1 is clearly not even close to optimal, as the comparison with a shortcut from v_0 to v_2 reveals. As $C_B(v_1)$ is slightly greater than $C_B(v_2)$ (due to the node between v_0 and v_2), we would prefer the shortcut from v_0 to v_1 if using node betweenness of the end-nodes as heuristic. The same holds for the stress indices.

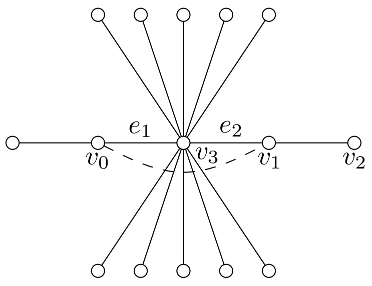


Fig. 4.2: An example for a case, where edge betweenness is inappropriate: The shortcut displayed as dashed is clearly and considerably suboptimal, as the comparison with a shortcut from v_3 to v_2 reveals. Nonetheless, e_1 and e_2 are the edges with the highest edge betweenness indices in the graph. The same holds for the stress indices.

a shortcut that would be considerably overestimated if using node betweenness or node stress.

Taking a closer look at this graph, we see why this happens. There are in fact a lot of shortest paths through v_0 and a lot of shortest paths through v_1. But very few of them use *both* nodes. Thus, barely no shortest path exists that could profit from a shortcut between v_0 and v_1.

In this graph, the use of the betweenness of the edges between v_0 and v_1 would be much more appropriate to estimate the usefulness of a shortcut between v_0 and v_1. But this does not really solve the problem. Consider for example the graph in Figure 4.2, which is an example for a situation where edge betweenness fails to give proper ratings. The problem with edge betweenness is, that shortest paths can “bend off” between the end-nodes of a shortcut.

4.2 Generalization of centrality indices to pairs of nodes

From this arises the question whether there is a way to generalize the aforementioned centrality indices to avoid these problems. The following definitions of *pair betweenness* and *pair stress* are the straightforward result of these considerations.

$$C_S(a,b) = \sum_{s,t \in V} \sigma_{st}(a,b) \qquad \text{(pair stress centrality of nodes } a \text{ and } b\text{)}$$

$$C_B(a,b) = \sum_{s,t \in V} \frac{\sigma_{st}(a,b)}{\sigma_{st}} \qquad \text{(pair betweenness centrality of nodes } a \text{ and } b\text{)}$$

One might think of these centrality measures as a kind of "path betweenness" or "path stress" of the shortest paths between a and b.

The difference of our centrality indices for pairs of nodes to the ones defined for edges is that we do not restrict ourselves to pairs of adjacent nodes. Returning to the problem of finding good shortcuts, the idea behind these centrality indices is obvious: the usefulness of a shortcut s between a and b is closely related to the number of shortest paths that can be shortened (in terms of the number of edges on it) by s. For fixed a and b, this equals the number of shortest s-a-b-t-paths $C_S(a,b)$.

On the other hand the betweenness centrality takes into account that the hop-distance between two nodes s and t does not automatically decrease, if there is a shortest s-t-path that can benefit from a shortcut. But, if every shortest s-t-path is a shortest s-a-b-t-path, this is guaranteed. In this case, $\sigma_{st}(a,b)/\sigma_{st}$ takes the maximum value 1. If there are other shortest s-t-paths, we add only smaller values to the betweenness of (a,b), as the probability of a decrease of $h(s,t)$ by less than $h(a,b)-1$ increases.

Note that if a and b are adjacent and the edge between a and b forms a unique shortest a-b-path, the pair betweenness of (a,b) equals the edge betweenness of $e=(a,b)$. The same holds for the stress indices. Another interesting property of $C_B(a,b)$ and $C_S(a,b)$ is the fact, that we can convert these measures to $w(a,b)$ by multiplying with $h(a,b)-1$, if shortest paths in the underlying graph are unique.

Lemma 15 *If shortest paths in G are unique, then:*

$$w(a,b) = (h(a,b)-1) \cdot C_S(a,b) = (h(a,b)-1) \cdot C_B(a,b)$$

Proof
As there is exactly one shortest s-t-path, $\sigma_{st}(a,b)$ is 1, if there is a shortest s-a-b-t-path, and 0 otherwise. Thus, $C_S(a,b)$ equals $|P(a,b)|$. The same holds for $C_B(a,b)$. Hence, it remains to show that $w(a,b)$ equals $|P(a,b)| \cdot (h(a,b)-1)$.

As we assume shortest paths to be unique, every shortest s-t-path is a hop-minimal shortest s-t-path as well. It follows, that

$$\begin{aligned} w(a,b) &= \sum_{s,t\in V} h(s,t) - h'(s,t) = \sum_{(s,t)\in P(a,b)} h(s,t) - h'(s,t) \\ &= \sum_{(s,t)\in P(a,b)} h(a,b) - 1 = (h(a,b)-1)\cdot |P(a,b)| \end{aligned}$$

■

Thus, we can expect to get a rather good estimation for $w(a,b)$ by multiplying the centrality measures with $h(a,b)-1$, if shortest paths in the underlying graph are "nearly unique". Moreover, we get upper bounds for $w(a,b)$ that are tighter than the ones we used in Algorithm 15, if we use the stress centrality.

Lemma 16

$$w(a,b) \leq (h(a,b)-1)\cdot \frac{1}{\sigma_{ab}} \cdot C_S(a,b) \leq (h(a,b)-1)\cdot \left(\sum_{t\in P^+(a,b)} \sigma_{bt}\right)\cdot\left(\sum_{s\in P^-(a,b)} \sigma_{sa}\right)$$

Proof

$$\begin{aligned} w(a,b) &= \sum_{s,t\in V} h(s,t) - h'(s,t) = \sum_{(s,t)\in P(a,b)} h(s,t) - h'(s,t) \\ &\leq \sum_{(s,t)\in P(a,b)} h(a,b) - 1 = (h(a,b)-1)\cdot |P(a,b)| \end{aligned}$$

As for every tuple of nodes (s,t) in $P(a,b)$, there exist at least σ_{ab} shortest s-a-b-t-paths, we get

$$w(a,b) \leq (h(a,b)-1)\cdot \frac{1}{\sigma_{ab}} \cdot C_S(a,b)$$

Furthermore, we get

$$\begin{aligned} (h(a,b)-1)\cdot \frac{1}{\sigma_{ab}} \cdot C_S(a,b) &= (h(a,b)-1)\cdot \sum_{(s,t)\in P(a,b)} \sigma_{sa}\cdot\sigma_{bt} \\ &\leq (h(a,b)-1)\cdot \sum_{s\in P^-(a,b)} \sum_{t\in P^+(a,b)} \sigma_{sa}\cdot\sigma_{bt} \\ &= (h(a,b)-1)\cdot \left(\sum_{s\in P^-(a,b)} \sigma_{sa}\right)\cdot\left(\sum_{t\in P^+(a,b)} \sigma_{bt}\right) \end{aligned}$$

■

One problem with the two introduced pair centralities is that they tend to be rather stable under the insertion of shortcuts, which might be unwanted if we use the centralities to measure the usefulness of shortcuts. Therefore, we consider two further pair centralities, that are closely related to pair betweenness and pair stress but restricted to hop-minimal shortest paths. We call them *hop-minimal pair betweenness* $C_{\text{HB}}(a,b)$ and *hop-minimal pair stress* $C_{\text{HS}}(a,b)$:

$$C_{\text{HS}}(a,b) = \sum_{s,t \in V} \eta_{st}(a,b) \qquad \text{(hop-minimal pair stress of nodes } a \text{ and } b\text{)}$$

$$C_{\text{HB}}(a,b) = \sum_{s,t \in V} \frac{\eta_{st}(a,b)}{\eta_{st}} \qquad \text{(hop-minimal pair betweenness of nodes } a \text{ and } b\text{)}$$

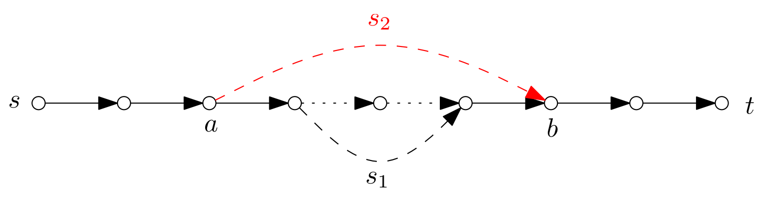
It is easy to see that the hop-minimal variants do not differ from the original pair centralities, if shortest paths are unique. Thus, using Lemma 15, we can easily convert the hop-minimal pair centralities to $w(a,b)$ in this case.

Each of the introduced pair centralities has advantages and disadvantages if used to measure the quality of shortcuts. Before we take a closer look at these, we will specify how exactly we plan to use pair centralities to rate shortcuts. To exploit the property to get exact estimations for $w(a,b)$, if shortest paths are unique, we multiply pair centralities with the hop-distance between the two nodes. Further, as there seems to be no reason to prefer shortcuts (a,b) with multiple shortest paths between a and b, we divide our pair stress centralities by σ_{ab} or η_{ab}, respectively. Thus, we get the following shortcut ratings:

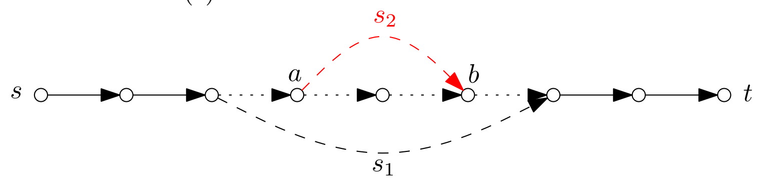
$$\begin{aligned}
\text{rate}_{\text{B}}(a,b) &= (h(a,b)-1) \cdot C_B(a,b) \\
\text{rate}_{\text{S}}(a,b) &= (h(a,b)-1) \cdot \frac{1}{\sigma_{ab}} \cdot C_S(a,b) \\
\text{rate}_{\text{HB}}(a,b) &= (h(a,b)-1) \cdot C_{\text{HB}}(a,b) \\
\text{rate}_{\text{HS}}(a,b) &= (h(a,b)-1) \cdot \frac{1}{\eta_{ab}} \cdot C_{\text{HS}}(a,b)
\end{aligned}$$

Figure 4.3 shows some simple examples of how shortcuts can be located with respect to each other. To get an idea of how the different shortcut ratings work, we take a look at the following table. Here, we compare the values that are added to the different shortcut ratings concerning the paths between s and t. More precisely, we give

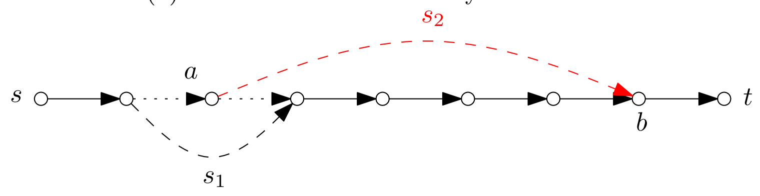
$\sigma_{st}(a,b)/\sigma_{ab} \cdot (h(a,b)-1)$	for pair stress,
$\sigma_{st}(a,b)/\sigma_{st} \cdot (h(a,b)-1)$	for pair betweenness,
$\eta_{st}(a,b)/\eta_{ab} \cdot (h(a,b)-1)$	for hop-minimal pair stress and
$\eta_{st}(a,b)/\eta_{st} \cdot (h(a,b)-1)$	for hop-minimal pair betweenness.



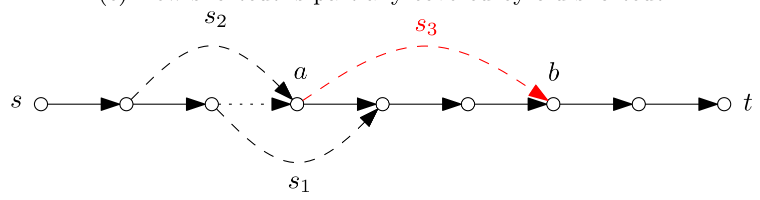
(a) New shortcut covers old shortcut



(b) New shortcut is covered by old shortcut



(c) New shortcut is partially covered by old shortcut



(d) Several hop-minimal shortest s-t-paths

Fig. 4.3: Example situations to illustrate the advantages and disadvantages of different pair centrality measures. In the underlying graph, there is exactly one shortest s-t-path. The shortcut displayed in red is the one to be evaluated heuristically using centrality measures. Black shortcuts have been already inserted. Edges displayed as dotted are on no hop-minimal shortest s-t-path.

The closer these values are to $h(s,t) - h'(s,t)$ (for all (s,t) in $V \times V$), the better we can use the respective centrality measure to estimate $w(a,b)$.

	$h(s,t) - h'(s,t)$	$C_B(a,b)$	$C_S(a,b)$	$C_{\text{HB}}(a,b)$	$C_{\text{HS}}(a,b)$
Fig. 4.3a)	2	2	2	2	2
Fig. 4.3b)	0	1/2	1	0	0
Fig. 4.3c)	3	2	4	0	0
Fig. 4.3d)	2	4/3	4	1	2

In the first example, all measures are equal to the decrease in hop-distance between s and t. In the second case, we overestimate this decrease if we use the original variants of pair betweennes, while in the third case, we underestimate it if we use the hop-minimal variants. Using the variants of pair betweenness, we tend to get underestimations, while with pair stress, we frequently observe overestimations. This illustrates that it is not easy to say in advance, which variant of pair betweenness is best suited to provide shortcut ratings. We will examine this question experimentally in Section 5.3.2.

Finally, analogous to the upper bounds for $w(a,b)$ that we get using $C_S(a,b)$, $C_{\text{HB}}(a,b)$ gives us a tool to determine lower bounds for $w(a,b)$.

Lemma 17

$$w(a,b) \geq (h(a,b)-1) \cdot C_{HB}(a,b)$$

Proof

$$\begin{aligned}
w(a,b) &= \sum_{s,t \in V} h(s,t) - h'(s,t) \\
&= \sum_{\substack{(s,t)\in P(a,b):\\ h(s,a)+h(a,b)+h(b,t)=h(s,t)}} \underbrace{(h(s,t)-h'(s,t))}_{=h(a,b)-1} + \sum_{\substack{(s,t)\in P(a,b):\\ h(s,a)+h(a,b)+h(b,t)>h(s,t)}} (h(s,t)-h'(s,t)) \\
&\geq \sum_{\substack{(s,t)\in P(a,b):\\ h(s,a)+h(a,b)+h(b,t)=h(s,t)}} (h(a,b)-1) \\
&\geq \sum_{\substack{(s,t)\in P(a,b):\\ h(s,a)+h(a,b)+h(b,t)=h(s,t)}} (h(a,b)-1) \cdot \frac{\eta_{st}(a,b)}{\eta_{st}} \\
&= (h(a,b)-1) \cdot \sum_{s,t\in V} \frac{\eta_{st}(a,b)}{\eta_{st}} \\
&= (h(a,b)-1) \cdot C_{\text{HB}}(a,b)
\end{aligned}$$

■

4.3 Fast algorithms to determine pair centrality indices

In the previous section, we saw how the proposed centrality indices for shortcuts can be used to estimate the decrease in overall hop-length. This is of course only relevant, if there is a way to compute the centrality indices faster than our $\Theta(n^3)$-algorithm to determine one optimal shortcut. For sparse graphs, this is the case, as we can use the ideas of Brandes' fast algorithm for node betweenness to compute betweenness for pairs of nodes as well. In contrast to the base algorithm, our algorithm works in two phases: In the first phase, we consider all shortest-paths dags in G to compute intermediary results. This corresponds to Brandes' algorithm. In the second phase, we consider the shortest-paths dags in $\bar{G}$ to sum up the intermediary results from the first phase.

Before we take a closer look at the details, we give an alternative characterization of $C_B(a,b)$, which we will need to justify the correctness of our algorithm.

Lemma 18

$$C_B(a,b) = \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} \cdot \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}}$$

Proof

Clearly, $\sigma_{st}(a,b) = 0$, if there is no shortest s-a-b-t-path. On the other hand, if there is a shortest s-a-b-t-path, then the length of each concatenation of shortest s-a-, a-b- and b-t-paths equals the distance between s and t in the graph. Hence, in this case $\sigma_{st}(a,b)$ equals $\sigma_{sa} \cdot \sigma_{ab} \cdot \sigma_{bt}$. It follows, that

$$C_B(a,b) = \sum_{s,t \in V} \frac{\sigma_{st}(a,b)}{\sigma_{st}} = \sum_{s,t \in P(a,b)} \frac{\sigma_{sa} \cdot \sigma_{ab} \cdot \sigma_{bt}}{\sigma_{st}}$$

Recall that the existence of a shortest s-a-b-t-path is equivalent to the question whether $\mathrm{dist}(s,a) + \mathrm{dist}(a,b) + \mathrm{dist}(b,t)$ equals $\mathrm{dist}(s,t)$. This is true if and only if $\mathrm{dist}(s,b) + \mathrm{dist}(b,t) = \mathrm{dist}(s,t)$ and $\mathrm{dist}(s,a) + \mathrm{dist}(a,b) = \mathrm{dist}(s,b)$, which is equivalent to the condition that s is in $P^-(a,b)$ and t is in $P^+(s,b)$. Hence, we get

$$\sum_{s,t \in P(a,b)} \frac{\sigma_{sa} \cdot \sigma_{ab} \cdot \sigma_{bt}}{\sigma_{st}} = \sum_{\substack{s \in P^-(a,b) \\ t \in P^+(s,b)}} \frac{\sigma_{sa} \cdot \sigma_{ab} \cdot \sigma_{bt}}{\sigma_{st}} = \sum_{\substack{s \in P^-(a,b) \\ t \in P^+(s,b)}} \frac{\sigma_{sa} \cdot \sigma_{ab} \cdot \sigma_{sb} \cdot \sigma_{bt}}{\sigma_{sb} \cdot \sigma_{st}}$$

Like before, we know that if s is in $P^-(a,b)$ and t in $P^+(s,b)$, $\sigma_{sb}(a)$ equals $\sigma_{sa} \cdot \sigma_{ab}$ and $\sigma_{st}(b)$ equals $\sigma_{sb} \cdot \sigma_{bt}$. Thus,

$$\begin{aligned}
\sum_{\substack{s \in P^-(a,b) \\ t \in P^+(s,b)}} \frac{\sigma_{sa} \cdot \sigma_{ab} \cdot \sigma_{sb} \cdot \sigma_{bt}}{\sigma_{sb} \cdot \sigma_{st}} &= \sum_{\substack{s \in P^-(a,b) \\ t \in P^+(s,b)}} \frac{\sigma_{sb}(a)}{\sigma_{sb}} \cdot \frac{\sigma_{st}(b)}{\sigma_{st}} \\
&= \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} \cdot \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}}
\end{aligned}$$

■

With this result and the algorithms for summing up values in shortest-paths dags introduced in Section 2.4, it is easy to develop an algorithm that computes betweenness for all pairs of nodes efficiently.

In the first phase, for fixed s and arbitrary t, we compute $\sum_{t \in P^+(s,b)} \sigma_{st}(b)/\sigma_{st}$. These values correspond directly to the dependencies in Brandes' algorithm. In the second phase, we use the algorithm SumValuesDecessorsLowerBound to sum up fractions of the results from the first phase along the paths in $\overline{D_b}$. Pseudocode for this approach is given as Algorithm 10.

Algorithm 10: PairBetweenness

```
  Input: Strongly connected graph $G = (V, E)$
  Output: $C_B(a, b)$ for each $(a, b) \in V \times V$
1 forall $s \in V$ do
2     Build $D_s$ and compute $\sigma_{sb}$ for all $b \in V$ using HopAndPathCountingDijkstra
3     Compute $\sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}}$ for all $b \in V$ using SumValuesDecessorsLowerBound
4
5 forall $b \in V$ do
6     Build $\overline{D_b}$ and compute $\sigma_{ab}$ for all $a \in V$ using HopAndPathCountingDijkstra
7     Compute $C_B(a, b) = \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} \cdot \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}}$ using
      SumValuesDecessorsLowerBound
8
```

The correctness of the algorithm follows directly from Lemma 5 and Lemma 18. For each node, we have to build two shortest-paths dags, one in the original and one in the reverse graph. Using Dijkstra's algorithm, this causes an overall time consumption in $O(n \cdot (n \log n + m))$. Further, we call the algorithm SumValuesDecessorsLowerBound two times for each node. As this corresponds essentially to two depth-first-traversals, this does not increase the asymptotical time complexity, so all in all, we stay in $O(n \cdot (n \log n + m))$. Unfortunately, in contrast to Brandes' algorithm, we cannot restrict ourselves to linear space complexity, as all intermediary results of the first phase have to be stored to be available during the second phase.

Considering the stress centrality for pairs of nodes, we get a very similar result:

Lemma 19

$$C_S(a, b) = \sigma_{ab} \cdot \sum_{s \in P^-(a,b)} \sigma_{sa} \cdot \sum_{t \in P^+(s,b)} \sigma_{bt}$$

Proof

$$\begin{aligned}
C_S(a, b) &= \sum_{s,t \in V} \sigma_{st}(a, b) \\
&= \sum_{s,t \in P(a,b)} \sigma_{sa} \cdot \sigma_{ab} \cdot \sigma_{bt} \\
&= \sigma_{ab} \cdot \sum_{\substack{s \in P^-(a,b) \\ t \in P^+(s,b)}} \sigma_{sa} \cdot \sigma_{bt} \\
&= \sigma_{ab} \cdot \sum_{s \in P^-(a,b)} \sigma_{sa} \cdot \sum_{t \in P^+(s,b)} \sigma_{bt}
\end{aligned}$$

■

With this result, it is straightforward to adapt the algorithm we used to compute betweenness for pairs of nodes to compute our generalized stress centrality. We just have to use SumValuesDecessorsUpperBound instead of SumValuesDecessorsLowerBound and multiply the output $C_S(a, b)$ of the algorithm with the number of shortest paths between a and b for each pair of nodes (a, b).

Further, it is easy to see that these algorithms can be used to compute C_{HB} and C_{HS} as well, if we restrict ourselves to hop-minimal shortest paths as described in Section 2.4. Finally, it remains to mention that, like with node betweenness, the computational complexity reduces to $O(nm)$, if we consider unweighted graphs. In this case, we can replace Dijkstra's algorithm by a simple breadth-first-search.

5 First Experiments

In this chapter, we will evaluate the algorithms developed to this point with respect to computation time and solution quality.

Experimental Setup. All of the algorithms evaluated in this thesis are implemented in C++ and compiled with GCC 4.3, using optimization level 4 and unrolling of loops as compiler flags. The experiments in this chapter are performed on one core of an Intel Xeon E5430 clocked at 2.66 GHz. The machine has 6144 kB L2 Cache and 32 GB RAM.

5.1 Graph classes used

5.1.1 Road graphs

The first graph class we consider are graphs modelling road networks centered at Karlsruhe with varying size, provided by the PTV AG. We use graphs ranging from 100 to 5000 nodes and extract the biggest strongly connected component, respectively. Our overall aim is to add several shortcuts to the graph using the algorithms to determine one optimal shortcut as subroutines. Thus, we are also interested in the performance of these algorithms after several shortcuts already have been added to the original graphs. Therefore, in addition to the original test set, we consider graphs that are generated by adding approximately $\sqrt{n}$ and n shortcuts to the original graphs greedily.

Table 5.1 shows the number of edges and the average number of hop-minimal shortest paths (HMSP) and all shortest paths(SP) between two arbitrary nodes in the graph. The latter two properties are of interest, as they give a rough indicator for the hardness of the test instances for our heuristic algorithms. Note that here and in the following, only a subset of the graphs are listed; the corresponding values for the graphs omitted did not differ substantially to the results given.

As the time complexity of adding n shortcuts greedily is unacceptably high for larger graphs, we generate these modified graphs only for graphs with up to 1000 nodes.

5.1.2 Grid graphs

The second graph class we consider are *rectangular grid graphs.* By this, we denote graphs with a mapping of the nodes to lattice points of a rectangular grid with the property that there exists an edge between two nodes if and only if the corresponding lattice points are adjacent.

	original graphs			$\sqrt{n}$ shortcuts added			n shortcuts added		
$\|V\|$	$\|E\|$	HMSP	SP	$\|E\|$	HMSP	SP	$\|E\|$	HMSP	SP
102	241	1.00	1.00	251	1.00	1.72	341	1.02	13.10
171	403	1.00	1.00	417	1.00	2.03	603	1.10	58.75
491	1129	1.00	1.00	1151	1.00	3.57	1629	1.15	6735.24
970	2287	1.00	1.01	2318	1.00	3.05	3287	1.13	23968.00
1969	4609	1.00	1.00	4653	1.00	5.09	-	-	-
4894	11484	1.00	1.01	-	-	-	-	-	-

Table 5.1: Road graphs

Grid graphs are often used to evaluate algorithms and are interesting out of several reasons. They are easy to imagine and analyze and, in the unweighted case, the average number of (hop-minimal) shortest paths between two nodes is very high. To tune the average number of shortest paths, we choose integer valued edge weights uniformly at random out of a given interval with varying size. Moreover, the edge weights len are symmetric, that means $\text{len}(x, y) = \text{len}(y, x)$ for all pairs of adjacent nodes x and y in the graph. Furthermore, we choose the corresponding grids to be close to squares. Table 5.2 illustrate the impact of the range of edge weights on the average number of (hop-minimal) shortest paths.

		uniform weights		weights in [1,2]	
$\|V\|$	$\|E\|$	HMSP	SP	HMSP	SP
100	360	281.91	281.91	6.16	6.21
196	728	16151.20	16151.20	6.82	7.13
506	1934	31074800.00	31074800.00	21.76	22.92
992	3842	236549000.00	236549000.00	116.34	131.42
1980	7742	610058000.00	610058000.00	1536.80	1619.08
4970	19598	1115270000.00	1115270000.00	6185490.00	6293160.00

		weights in [1,5]		weights in [1,100]		weights in [1,1000]	
$\|V\|$	$\|E\|$	HMSP	SP	HMSP	SP	HMSP	SP
100	360	1.34	1.43	1.00	1.01	1.00	1.00
196	728	1.52	1.81	1.02	1.03	1.00	1.00
506	1934	2.11	2.59	1.01	1.02	1.00	1.00
992	3842	5.62	7.11	1.05	1.07	1.00	1.00
1980	7742	4.49	7.52	1.04	1.09	1.00	1.01
4970	19598	23.24	37.08	1.15	1.20	1.01	1.02

Table 5.2: Grid Graphs

5.1.3 $G(n,p)$-graphs

The third graph class we use are random $G(n,p)$-graphs according to the Erdõs-Rényi model. This means, we generate random graphs with n nodes and the property that an edge (x,y) between two arbitrary nodes x and y exists with probability p. As all the other graph classes considered are rather sparse, we use dense $G(n,p)$-graphs to illustrate how the running times of our algorithms depend on the density of the underlying graph. To avoid the effect that most of the edges are longer than the shortest path between the end-nodes, we consider unweighted graphs. Table 5.3 shows properties of the generated graphs. A density of k percent means that we set the parameter p to $k/100$.

	appr. 5% density			appr. 40% density		
$\|V\|$	$\|E\|$	HMSP	SP	$\|E\|$	HMSP	SP
100	466	2.23	2.23	3952	9.72	9.72
200	1970	3.51	3.51	15848	19.24	19.24
500	12204	8.81	8.81	99760	48.08	48.08
1000	49640	11.36	11.36	397888	95.56	95.56
2000	199444	7.76	7.76	1598600	192.02	192.02
5000	1250692	11.94	11.94	9994926	479.92	479.92

Table 5.3: $G(n,p)$-graphs

5.1.4 Unit Disk graphs

The last class of graphs considered are *unit disk graphs*. Given n and m, a unit disk graph is generated by randomly assigning each of the n nodes to a point in the unit square of the Euclidean plain. Two nodes are connected by an edge in case their Euclidean distance is below a given radius. This radius is adjusted such that the resulting graph has approximately m edges. Unit disk graphs are commonly used to model the topology of ad-hoc wireless communication networks. According to this context, we set edge weights to the square of the distances between the end-nodes in the plane. As we choose the location of the points randomly, shortest paths in these graphs are nearly unique. Few exceptions occur due to discretization effects. We use graphs of average node degree 10 and 20. Elementary properties of the generated graphs can be found in Table 5.4.

av. Deg. 10				av. Deg. 20			
$\lvert V\rvert$	$\lvert E\rvert$	HMSP	SP	$\lvert V\rvert$	$\lvert E\rvert$	HMSP	SP
100	1000	1	1.00	100	2000	1	1.00
198	1998	1	1.00	200	4000	1	1.00
499	4998	1	1.00	500	10000	1	1.00
1000	10010	1	1.00	1000	20008	1	1.00
1998	20012	1	1.00	2000	39988	1	1.00
4997	49990	1	1.00	5000	99974	1	1.00

Table 5.4: Unit Disk graphs

5.2 Running times for Finding One Optimal Shortcut

5.2.1 Comparison of preselection criteria

In this section, we will compare the running time of our heuristic improved versions of the brute-force algorithm that use bounds under different preselection criteria (see Section 3.2.3). As an example, Table 5.5 shows the results for our road graphs with degree, reach and betweenness as strategies to preselect promising shortcuts. For comparison reasons, we also give the running times of the versions without preselection criteria (called None). Both with precomputed and non-precomputed bounds, we see that the running times

	non-precomputed bounds				precomputed bounds			
$\lvert V\rvert$	N	D	R	B	N	D	R	B
102	0.46	0.44	0.45	0.44	0.07	0.05	0.06	0.05
171	2.18	2.05	2.09	2.04	0.41	0.28	0.30	0.26
278	11.59	10.57	11.29	10.66	3.97	2.97	3.40	2.89
392	27.63	27.32	28.33	27.38	6.20	5.81	6.55	5.79
491	47.93	46.89	49.11	46.58	5.10	4.53	6.28	4.55
591	82.43	81.75	86.46	81.97	7.62	6.29	9.93	6.63
684	167.78	163.08	168.09	162.60	51.56	46.30	49.92	46.61
781	265.21	268.49	270.09	256.76	91.33	94.04	94.07	84.23
876	353.89	359.24	361.55	345.29	110.19	114.47	114.21	99.56
970	423.52	441.84	448.58	420.00	92.23	109.33	111.11	86.91

Table 5.5: Running times (in seconds) of heuristic algorithms to determine one optimal shortcut using bounds on road graphs centered at Karlsruhe. Preselection criteria used are: None(N), Degree(D), Reach(R) and Betweenness(B)

differ only slightly. Altogether, with reach and degree, there seems to be no improvement compared to the version without preselection. In contrast to that, using betweenness, we could reduce the running times for finding one optimal shortcut with every test instance.

The running times for the other graph classes show similar behaviour.

All in all, the running times do not differ much due to different preselection criteria and for the majority of graphs, betweenness yields the best results. Therefore, to compare our heuristic algorithms to the unoptimized version and our $\Theta(n^3)$-algorithm, we always regard solely the results for preselection using betweenness.

5.2.2 Running times for exact algorithms

In this section, we will examine the running times of our algorithms to solve the BSP restricted to one shortcut. Figure 5.1 shows the results the particular strategies achieved in road graphs centered at Karlsruhe with and without added shortcuts. Running times over 2000 seconds remained unconsidered. The first observation is that the bounds we used to prune the search space for the brute-force algorithm proved to be valuable. Using these, we were able to evaluate graphs with up to 2500 nodes, while for the unoptimized brute-force algorithm, even graphs with only 400 nodes led to inacceptably long running times. We achieve the best results with the original graphs without shortcuts. This coincides with the fact that in these graphs, shortest paths are nearly unique. Intuitively, our bounds should be tighter in this case. If we compare the (rough) precomputed bounds to the (tighter) non-precomputed bounds, we see that the former lead to better performance in the original graphs and the latter to better performance in the graphs with shortcuts. Recalling the fact that the bounds are equal if shortest paths are unique, this confirms our theoretical results. Compared to the other algorithms, our $\Theta(n^3)$-algorithm is unbeatable fast. With this algorithm, we are able to evaluate graphs with up to 2500 nodes in under 260 seconds.

The experiments with grid graphs confirmed these results (see Figure 5.2). The lower the average number of shortest paths, the better is the performance of our algorithms using bounds. As with the road graphs, if shortest paths are nearly unique (edge weights in $[1, \ldots, 1000]$), the precomputed bounds achieve better results than the non-precomputed bounds.

We get similar results evaluating unit disk graphs (see Figure 5.3). The hardness of $G(n, p)$-graphs for our algorithms with bounds is not this closely related to the average number of shortest paths. First, we see that the unoptimized brute-force algorithm has a worse performance than with the other graph classes. This coincides with theoretical considerations, as the running time of APSP increases with the number of edges in the graph. The fact that our bounds work astonishingly well in the graphs with density 0.4 has a very simple reason: In these graphs, there are no shortest paths that contain more than two edges. Thus, both bounds are trivially tight.

As a last remark concerning the algorithms using bounds, we take a closer look at the performance in road graphs with shortcuts added in advance and grid graphs with uniform edge weights. Interestingly, the performance is equally poor, although the average number of shortest paths in the road graphs is much lower than with the grid graphs. This could indicate that graphs with shortcuts pose an especially hard challenge to these algorithms.

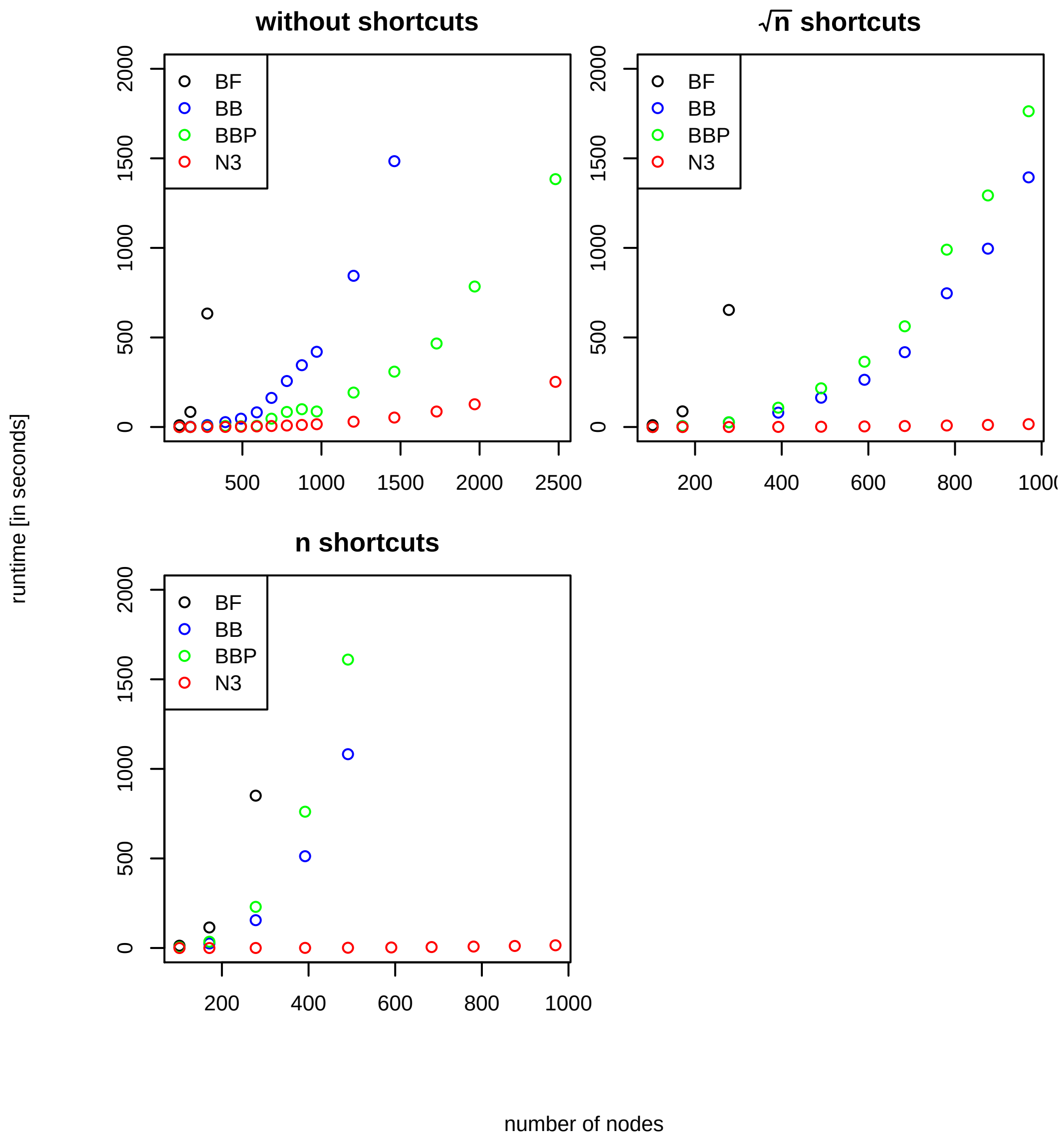


Fig. 5.1: Running times for different algorithms to solve the BSP restricted to one shortcut in road graphs centered at Karlsruhe: brute-force(BF), brute-force using (non-precomputed) bounds and preselection according to betweenness(BB), brute-force using precomputed bounds and preselection according to betweenness(BBP) and $\Theta(n^3)$-algorithm(N3)

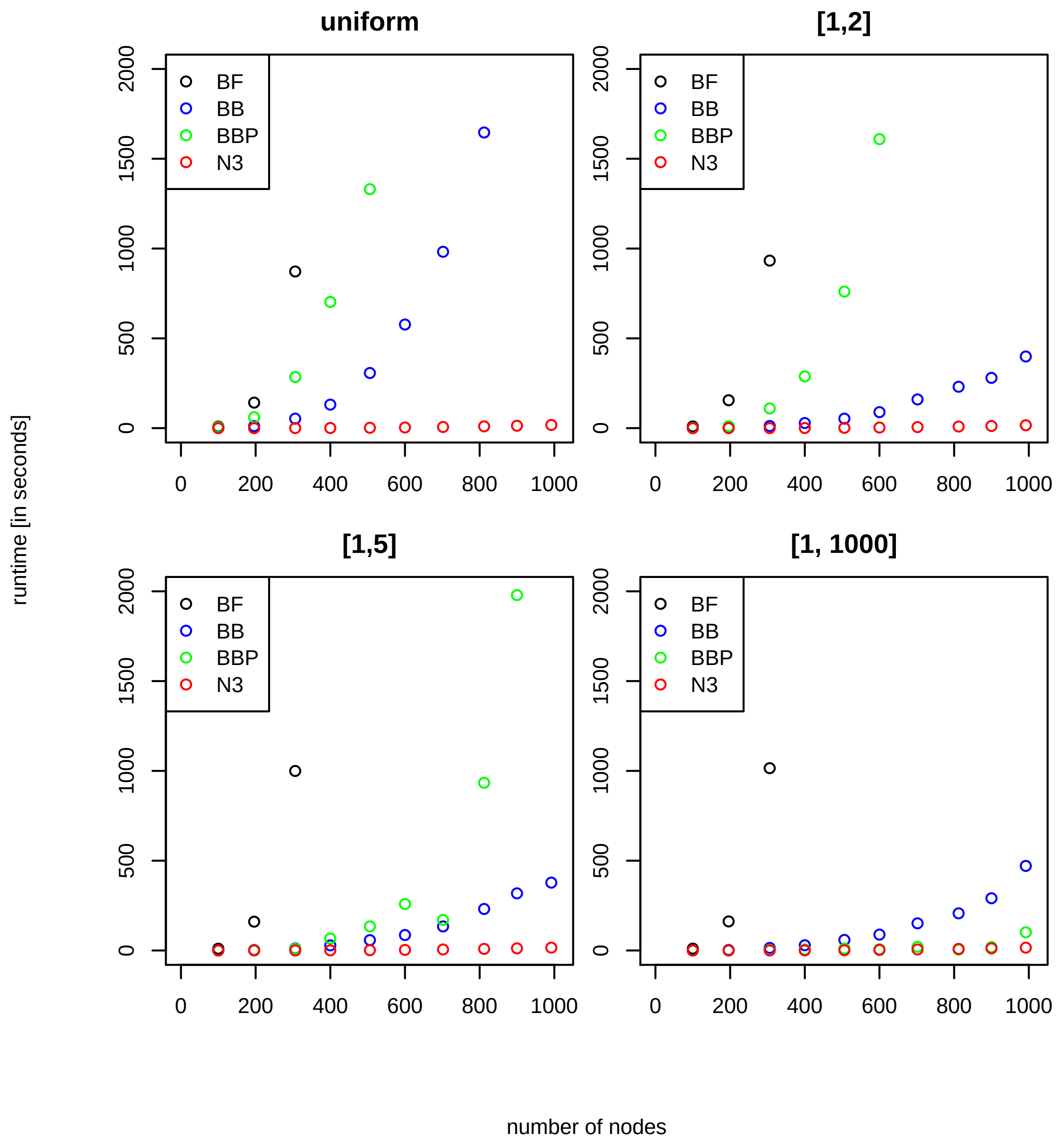


Fig. 5.2: Running times for different algorithms to solve the BSP restricted to one shortcut in grid graph with varying scope of edge weights: brute-force(BF), brute-force using (non-precomputed) bounds and preselection according to betweenness(BB), brute-force using precomputed bounds and preselection according to betweenness(BBP) and $\Theta(n^3)$-algorithm(N3)

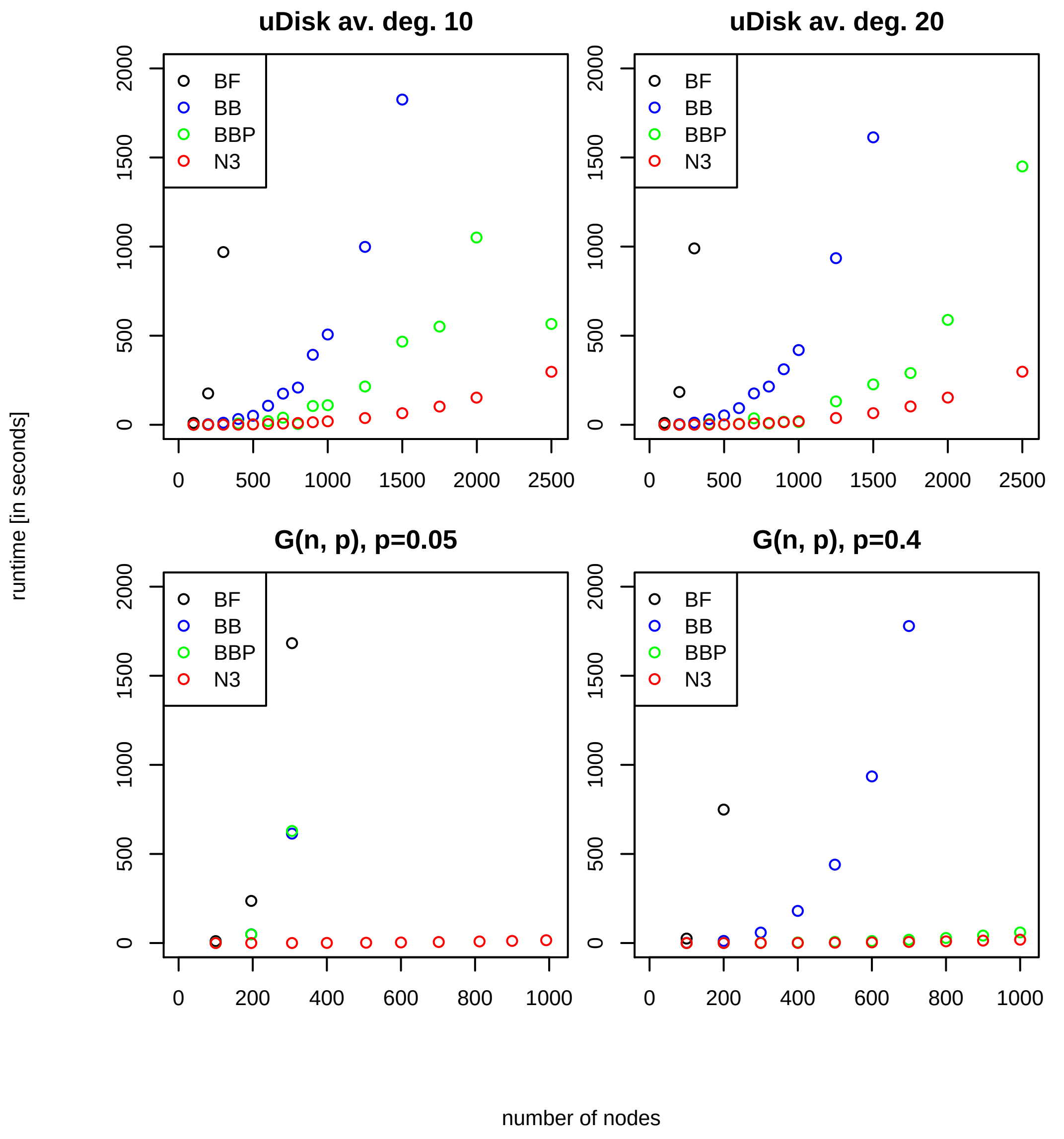


Fig. 5.3: Running times for different algorithms to solve the BSP restricted to one shortcut in unit disk graphs and $G(n, p)$-graphs: brute-force(BF), brute-force using (non-precomputed) bounds and preselection according to betweenness(BB), brute-force using precomputed bounds and preselection according to betweenness(BBP) and $\Theta(n^3)$-algorithm(N3)

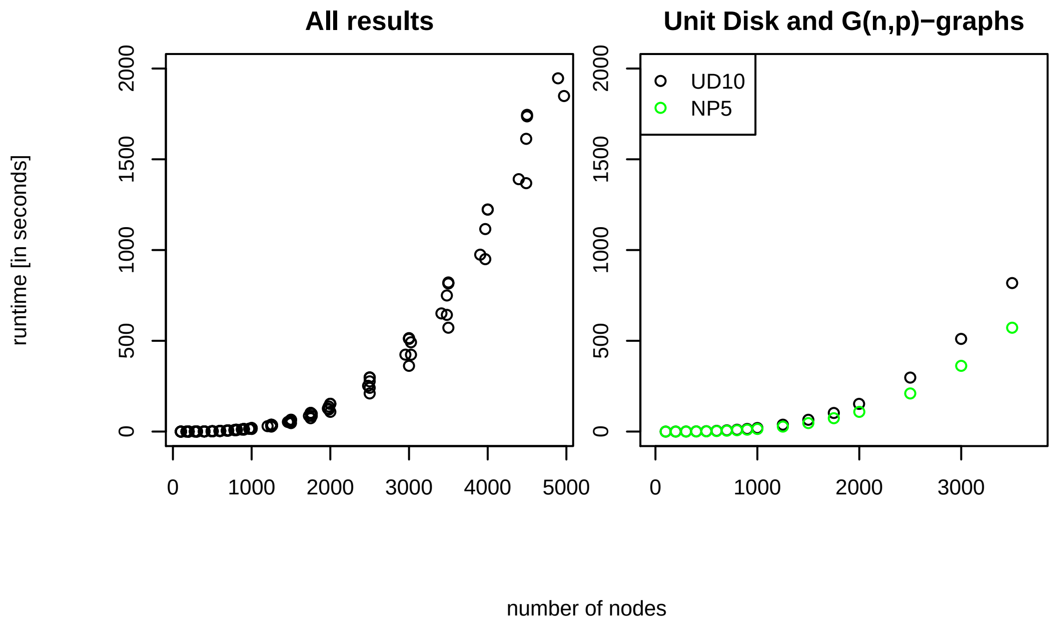


Fig. 5.4: Running time of $\Theta(n^3)$-algorithm on different graph classes: On the left, we see the results of all graphs considered, on the right, we compare unit disk graphs with average degree 10 to $G(n, p)$-graphs with appr. 5% density

We now take a closer look at the performance of our $\Theta(n^3)$-algorithm that clearly outperforms the other strategies. In the left part of Figure 5.4, we see all results of this algorithms for different graph classes depending on the number of nodes in the graph. The first observation is that there seems to be no fundamental differences between the particular classes considered. Closer analysis shows that the curves we get if we restrict ourselves to one graph class are rather smooth while the running times for different classes and comparable number of nodes differ to a constant factor. The largest differences arise when comparing unit disk graphs with average degree 10 to $G(n, p)$-graphs with 5% density (the running times for the latter are approximately 26% lower). This might seem unintuitial at first glance, as the only obvious influence of the particular graph structure on the performance of our $\Theta(n^3)$-algorithm arises in the preprocessing step, when we solve APSP using Dijkstra's algorithm. But this should be harder for dense graphs. Closer analysis showed that another effect has a much greater influence on the performance. If we take a look at the pseudocode again, we see that there are some calculations that are only executed if one particular node is on a shortest path between two other nodes. As shortest paths in unit disk graphs are rather long, the probability for this is a lot higher than in the $G(n, p)$-graphs considered. This outweighs the additional overhead for solving APSP in the latter by far.

Conclusion. In summary, by pruning the search space using bounds, we were able to get considerable improvements concerning computation time. Depending on the structure of the graph, especially the average number of (hop-minimal) shortest paths, either of the two kinds of bounds is preferable. Nonetheless, our $\Theta(n^3)$-algorithm clearly outperforms the other algorithms, especially in the case of non-unique shortest paths.

5.3 Heuristics for finding single shortcuts

In this section, we evaluate the quality of single shortcuts found by different heuristics. We will focus on our pair centrality indices and compare the results obtained with these with shortcuts ranked best according to our preselection criteria and with the optimum. As we consider single shortcuts, this optimum can be found using our $\Theta(n^3)$-algorithm.

5.3.1 Running times

In a first step, we will compare the running times of the different heuristics among one another and to the running time of our $\Theta(n^3)$-algorithm. Table 5.6 shows the results we obtain if we use the heuristics in road graphs centered at Karlsruhe. Clearly, all algorithms aside from degree and computing the optimum have running times in the same complexity class. Among these, the algorithms to compute bounds and pair centralities are to a constant factor slower than reach and betweenness. If we consider the graph with 4894 nodes, all heuristics find solutions with less than 2% of the time needed to compute an optimal shortcut. These results were confirmed by the other sparse graph classes, namely grid graphs and unit disk graphs.

$\|N\|$	Deg	Reach	Bet	Bound	PB	HMPB	PS	HMPS	Opt
102	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.02
171	0.00	0.01	0.01	0.02	0.02	0.03	0.02	0.03	0.07
491	0.00	0.05	0.05	0.21	0.22	0.22	0.21	0.21	1.64
970	0.00	0.20	0.22	0.87	0.89	0.90	0.88	0.89	15.81
1969	0.00	0.88	0.95	3.66	3.75	3.75	3.75	3.80	127.07
4894	0.00	5.83	6.12	23.44	24.10	23.90	24.11	23.92	1946.13

Table 5.6: Running times (in secondes) of different heuristics in road graphs centered at Karlsruhe. From left to right: degree, reach, node betweenness, upper bounds, pair betweenness, hop-minimal pair betweenness, pair stress, hop-minimal pair stress, $\Theta(n^3)$-algorithm (for comparison)

If we consider very dense graphs, we do not obtain any improvements in the running time using heuristics instead of computing the optimum. The only exception is the degree criterium. This is illustrated in Table 5.7 and coincides with theoretical considerations concerning asymptotical running times.

$\lvert N\rvert$	Deg	Reach	Bet	Bound	PB	HMPB	PS	HMPS	Opt
100	0.00	0.01	0.01	0.04	0.04	0.04	0.04	0.04	0.02
500	0.00	1.34	1.63	3.31	3.68	3.69	3.50	3.53	1.98
1000	0.02	12.67	18.45	26.17	29.95	30.91	29.94	29.16	18.86
3000	0.13	373.57	1255.40	723.68	832.70	834.55	799.14	797.04	513.65

Table 5.7: Running times of different heuristics on $G(n, p)$-graphs with 40% density. From left to right: degree, reach, node betweenness, upper bounds, pair betweenness, hop-minimal pair betweenness, pair stress, hop-minimal pair stress, $\Theta(n^3)$-algorithm (for comparison)

5.3.2 Quality

Concerning the quality of single shortcuts found by different heuristics, the first graphs we consider are road graphs. Figure 5.5 shows the results for the unmodified graphs. First, we see that the algorithm using hop-minimal pair betweenness always found an optimal solution. The same holds for the other pair centralities. As for these shortcut ratings, we get provably optimal results if shortest paths are unique, this confirms our theoretical results. The other heuristics yield worse solutions. Thereby, except for two graphs, the results obtained by using node betweenness are (in most instances by far) the best among these.

Figure 5.5 also illustrates the results for the same graphs and approximately $\sqrt{n}$ shortcuts added in advance. A comparison of the solution quality obtained by different pair centralities can be found in Figure 5.6. For these graphs, the hop-minimal variants of the pair centrality indices yield by far the best results, while the “conventional” variants have much lower performance. A striking observation is that the quality of the shortcuts found by using degree is astonishingly high. This can be explained by properties of high-quality shortcut assignments, which we will discuss in Section 7.4.

Unfortunately, if we add n shortcuts to the graph, the quality of the shortcuts found using hop-minimal pair centralities decreases. In this setting, the performance of the pair centrality indices is approximately the same as with betweenness, bounds and degree. Anyhow, on the whole, the quality of the shortcuts found with these heuristics seems to be better than with just random picking or reach.

The experiments in grid graphs confirmed these observations. The performance of node betweenness, pair betweenness and pair stress decreases with increasing average number of shortest paths in the graph. As an example, Figure 5.6 shows the results obtained by different pair centralities in grid graphs with edge weights in $[1, 5]$. Again, using pair centralities, we get optimal and nearly optimal solutions. Interestingly, for these graphs, pair betweenness seems to work a lot better than pair stress. As shortest paths are nearly unique in our unit disk graphs, using pair centralities, we get optimal results for all instances.

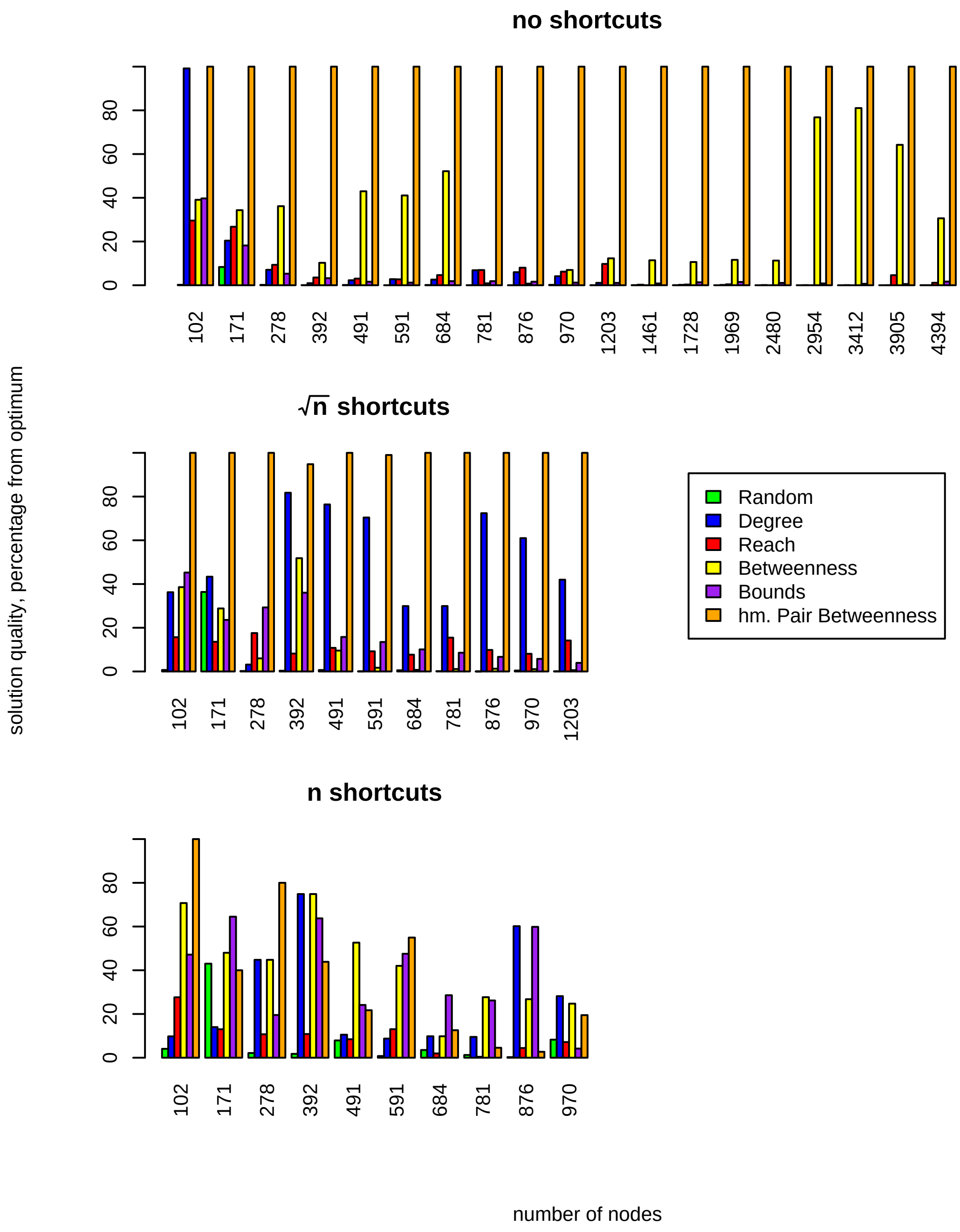


Fig. 5.5: Solution qualitiy of single shortcuts in road graphs with and without shortcuts inserted in advance, different heuristics

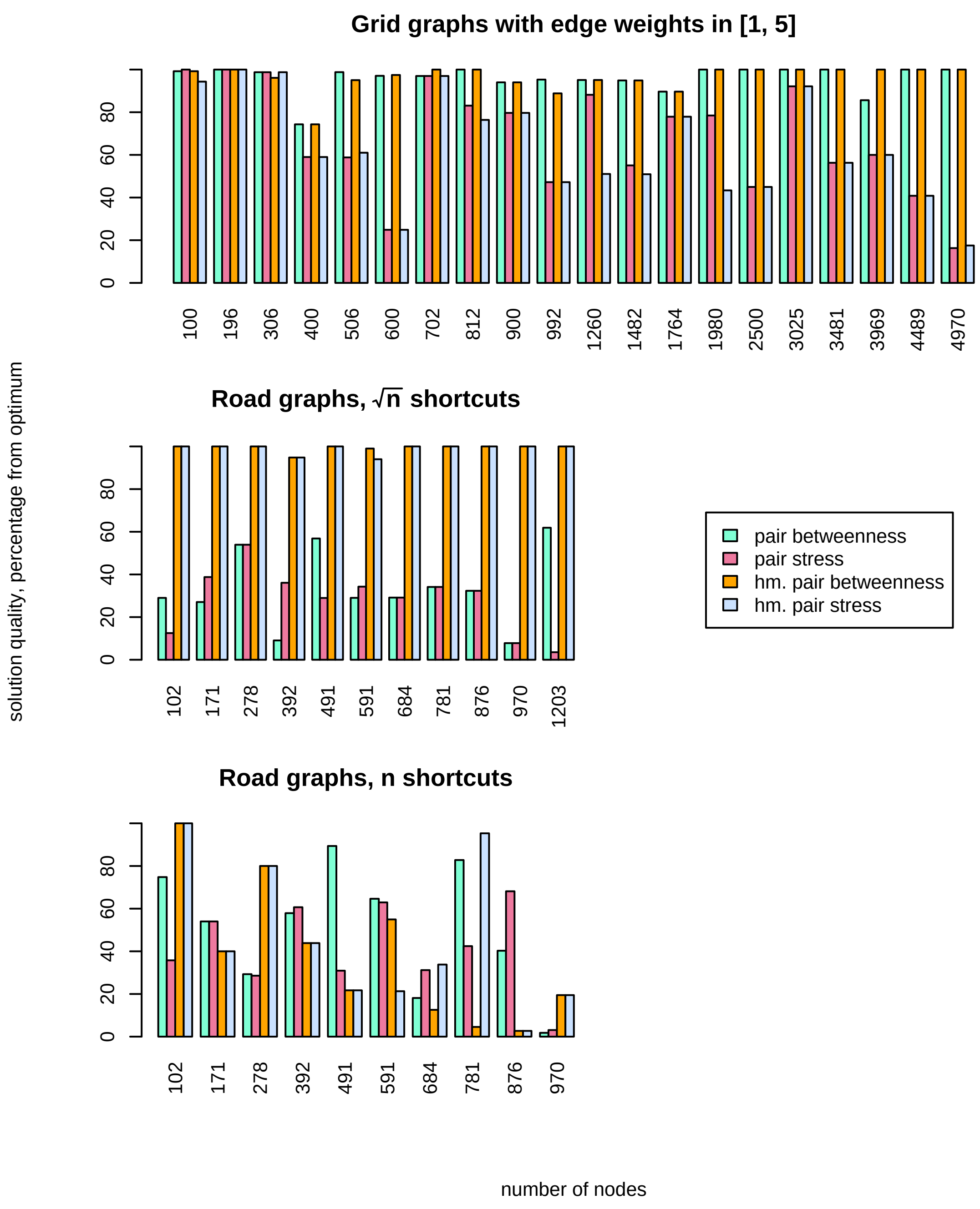


Fig. 5.6: Solution quality of single shortcuts in grid graphs and road graphs with shortcuts inserted in advance, pair centralities

Conclusion. Summarizing the results, on the whole, hop-minimal pair betweenness seems to be the best choice for rating shortcuts, especially if we aim to add a set of shortcuts to a graph.

6 Local Search

In this chapter, we give a brief introduction to local search and show how the concept of local search can be applied to the BSP. We then introduce four basic local search strategies and describe in detail, how to use them to develop heuristic algorithms for the BSP. The last section is devoted to a measure that is an indicator for the suitability of local search to solve a given combinatorial problem under a given modeling of search space and evaluation function.

6.1 General concept and application to the BSP

For an introduction to the idea of local search and basic local search strategies, see [HS05]. Local search algorithms play an important role in the context of *combinatorial problems*. For a combinatorial problem, a potential solution can typically be partitioned into solution components. For example, if we consider the BSP, every combination of c distinct shortcuts is a potential solution to our problem. Formally, a stochastic local search algorithm is defined as follows (see [HS05]):

Definition 2 (Stochastic Local Search Algorithm) Given a (combinatorial) problem Π, a stochastic local search algorithm for solving an arbitrary problem instance $\pi \in \Pi$ is defined by the following components:

- the *search space* $S(\pi)$ of the instance π, which is a finite set of *candidate solutions*
- a *set of (feasible) solutions* $S'(\pi) \subseteq S(\pi)$
- a *neighbourhood relation* on $S(\pi)$, $N(\pi) \subseteq S(\pi) \times S(\pi)$
- a finite *set of memory states* $M(\pi)$, which, in the case of SLS algorithms that do not use memory, may consist of a single state only
- an *initialization function* $\text{init}(\pi) : \{\emptyset\} \to \mathcal{D}(S(\pi) \times M(\pi))$, which specifies a probability distribution over initial search positions and memory states
- a *step function* $\text{step}(\pi) : S(\pi) \times M(\pi) \to \mathcal{D}(S(\pi) \times M(\pi))$ mapping each search position and memory state onto a probability distribution over its neighbouring search positions and memory states
- a *termination predicate* $\text{terminate}(\pi) : S(\pi) \times M(\pi) \to \mathcal{D}(\{T, F\})$ mapping each search position and memory state to a probability distribution over truth values

(T = true, F = false), which indicates the probability with which the search is to be terminated upon reaching a specific point in the search space and memory state

In the above, $\mathcal{D}(S)$ denotes the set of probability distributions over a given set S, where formally, a probability distribution $D \in \mathcal{D}(S)$ is a function $D : S \rightarrow \mathbb{R}_0^+$ that maps elements of S to their respective probabilities.

Further, we use an *evaluation function* $g(\pi)(s) : S(\pi) \rightarrow \mathbb{R}$ to rate candidate solutions. This is used as guidance towards good solutions and it will always coincide with the objective function characterizing the BSP, that is $g(\pi)(s) = w(s)$, the decrease in overall hop-lengths we aim to maximize. Typically, the step function used heavily relies on the value of the evaluation function of the current candidate solution and its neighbours.

In the context of the local search strategies we use, the concept of *local maxima* is of great importance.

Definition 3 (Local Maximum) Given a search space S, a neighbourhood relation $N \subseteq S \times S$ and an evaluation function $g : S \mapsto \mathbb{R}$, a *local maximum* is a candidate solution $s \in S$ such that for all $s' \in N(s)$, we have $g(s) \geq g(s')$.

Given an instance $\pi = (G, c)$ of the BSP, we map the corresponding components of local search to the following objects:

- *Search space* $S(\pi)$: The set of all possible shortcut assignments with c shortcuts in G.
- *(Feasible) solutions* $S'(\pi)$: The same as $S(\pi)$; every assignment with c shortcuts is considered feasible.
- *Neighbourhood relation*: Depending on the local search strategy; except for Variable Neighbourhood Descent, the standard 1-exchange neighbourhood is used: two shortcut assignments are considered as neighbours if they differ in at most one shortcut
- *Set of memory states* $M(\pi)$: Depending on the local search strategy
- *Initialization function init*(π): Two possibilities can be combined with every local search strategy:
  - *Random initialization*: To determine an initial candidate solution, we choose c shortcuts uniformly at random. To avoid unnecessary multiple edges, we sample without replacement and discard loops and shortcuts between nodes that are already connected by an edge that forms a shortest path. It is easy to see that these shortcuts do not contribute to decreasing the hop-distances in G.
  - *Greedy initialization*: As initial candidate solution, we use the output of the GREEDY-algorithm with c shortcuts in G.
- *Step function step*(π): Depending on the local search strategy; roughly, we use the following two variants (called *pivoting rules*) as components:

  - *Best Improvement*: we select a neighbour that achieves maximal improvement in the evaluation function among all neighbours of the current candidate solution
  - *Random Order First Improvement*: in each search step we determine a random order in which we evaluate the neighbours of our current candidate solution s; the first neighbour s' with $g(s') > g(s)$ becomes our new candidate solution.
- *termination predicate terminate*(π): We use two very simple termination criteria:
  - *cut-off time*: We specify a maximal number r of runs of our $\Theta(n^3)$-algorithm and terminate after we have reached this bound.
  - *local maximum*: If we consider simple local search algorithms without escape strategies, we terminate after we have reached a local maximum as candidate solution. In this case, the step function is undefined as there is no neighbour that improves the evaluation function.

6.2 Neighbourhood Pruning

Taking a closer look at the time needed to evaluate a complete 1-exchange-neighbourhood without optimization, we see that this is a very expensive task. The number of assignments in each neighbourhood is $c \cdot (n^2 - c + 1)$, as each of the c shortcuts in the current candidate solution can be replaced by $n^2 - c + 1$ possible shortcuts (we consider the reflexive variant of the neighbourhood relation, where each assignment is a neighbour of itself). Thus, if we determine the evaluation function for each assignment in the neighbourhood by solving APSP, we have an overall time complexity in $\Theta(c \cdot (n^2 - c + 1) \cdot n \cdot (n \log n + m))$

Therefore, we do not explicitely evaluate all neighbours, or, more precisely, we evaluate them in groups using our $\Theta(n^3)$-algorithm. This means, we consider just those shortcut assignments that are the result of deleting one of the shortcuts in the current candidate solution s and refilling greedily. Clearly, a most improving neighbour s' of s is among these assignments. Thus, if we use Best Improvement to determine our next candidate solution, we can restrict ourselves to this subset of our actual neighbourhood without changing the step function at all. Similarly, if we do not find an assignment that improves the current solution quality among these neighbours, we know that the current search position constitutes a local maximum. Using this idea, we are able to evaluate complete 1-exchange-neighbourhoods in $\Theta(c \cdot n^3)$ time.

If we use Random Order First Improvement, the step function changes, as we consider only a subset of $\{s' \mid (s, s') \in N(\pi) \text{ and } w(s') > w(s)\}$. Similarly, if we evaluate larger neighbourhoods using this idea, we consider only a subset of the neighbourhood, as deleting k shortcuts and refilling greedily does not always lead to a most improving neighbour of s. Nonetheless, the set of shortcut assignments that can thus be obtained forms a subset of neighbours that is likely to achieve good solution qualities.

Another heuristic to prune the neighbourhood arises from the possibility to use the shortcut ratings we introduced in Section 4.3 instead of our $\Theta(n^3)$-algorithm. Using this idea, we can evaluate 1-exchange-neighbourhoods in $O(c \cdot n \cdot (n \log n + m))$. The downside of this approach is that, if we consider just the neighbours of s that arise of deleting one shortcut and refilling with the shortcut with maximum rating, there is no guarantee that we find a most improving neighbour. As a complexity of $\Theta(c \cdot n^3)$ in each search step is infeasible for larger graphs, we can nonetheless try to improve the solution quality using this pruned neighbourhood.

6.3 Fundamental local search strategies

In this section, we consider four basic local search strategies. Unless specified otherwise, we use 1-exchange neighbourhoods.

One of the simplest local search strategies used is *Iterative Improvement.* The general proceeding is given as Algorithm 11. We iteratively select a neighbour of our current candidate solution s with better solution quality until we reach a local maximum. As remaining degrees of freedom, we can choose between random and GREEDY initialization and between Best Improvement and Random Order First Improvement as step function.

Algorithm 11: Iterative Improvement (II)

```
1 Determine initial candidate solution $s$
2 while $s$ is not a local optimum do
3     Choose a neighbour $s'$ of $s$ such that $g(s') > g(s)$
4     $s := s'$
5
```

The downside of Iterative Improvement lies in the fact that as soon as we reach a local maximum, the search gets stuck and there is no possibility to reach better solutions. The probably most basic escape strategy one can think of is to pick a new initial candidate solution and apply a new run of Iterative Improvement as soon as we reach a local maximum. We terminate the search process after a given cut-off time. We call this variant of local search *Iterative Improvement Random Restart.* Theoretically, we have the same degrees of freedom as with simple Iterative Improvement. But it is easy to see that there should be at least some diversification in the search process. If we use the deterministic variant of Iterative Improvement with GREEDY initialization and Best Improvement as step function, we will always reach the same local maximum and therefore never achieve any improvements after the first phase of Iterative Improvement.

Another strategy to avoid getting stuck in local maxima uses larger neighbourhoods. The idea behind this is the fact that a local maximum under the standard 1-exchange neighbourhood does not have to be a local maximum under k-exchange neighbourhoods with $k > 1$. As an extreme example, if we consider c-exchange neighbourhoods with c representing the number of shortcuts in a candidate solution, each neighbourhood represents

Algorithm 12: Variable Neighbourhood Descent (VND)

```
1  Determine initial candidate solution $s$
2  $i := 1$
3  repeat
4      Choose a most improving neighbour $s'$ of $s$ in $N_i$
5      if $g(s') > g(s)$ then
6          $s := s'$
7          $i := 1$
8      else
9          $i := i + 1$
10
11 until $i > i_{max}$
```

the whole search space. This leads to the idea of *Variable Neighbourhood Descent*, that successively scans larger neighbourhoods until it reaches a candidate solution with higher quality than the current local maximum (see Algorithm 12). As the size of k-exchange neighbourhoods increases rapidly with k, we use neighbourhoods that are pruned in two ways: First, as already mentioned, we consider just those neighbours that arise from deleting k shortcuts and refilling greedily. Second, if the size of this pruned neighbourhood exceeds a parameter r, we randomly choose k shortcuts in the current candidate solution to be replaced and repeat this until we have evaluated (at most) r neighbouring assignments. If we do not find an improving solution among these, we increase k by 1. Concerning initialization strategy and pivoting rule we have the same choices as with Iterative Improvement.

The last local search strategy we apply is *Tabu Search* (see Algorithm 13). In contrast to Variable Neighbourhood Descent, Tabu Search sometimes accepts worsening search steps to escape from local maxima. To avoid cycling search behaviour, aspects of the search history are used to restrict the current neighbourhood. In the context of the BSP, we choose the following approach: If, during a search step, a shortcut sc is replaced by another, sc is set tabu for the next tt search steps, with tt being a tuning parameter. In each step, we just consider those neighbours of our current candidate solution s that are the result of deleting one shortcut in s and replacing it with the shortcut that is optimal among all shortcuts that are not currently set tabu. If none of these neighbouring assignments improves the current solution quality, we jump to the assignment that is the least worsening. Note that the tabu tenure tt can be respected without much additional overhead, as our $\Theta(n^3)$-algorithm not only determines the shortcut sc with maximum $w(sc)$, but evaluates all possible shortcuts simultaneously.

Furthermore, we weaken the tabu tenures by using a criterion called *aspiration.* This means that we override the tabu status of a candidate solution if it leads to an improvement in the best solution found.

Again, we have the choice between random and greedy initialization and best and Random Order First Improvement as pivoting rule. Note that the latter only differ if there is a admissible neighbour that improves the current solution quality.

Algorithm 13: Tabu Search (TS)

```
1 Determine initial candidate solution $s$
2 while termination criterion is not satisfied do
3     Determine set $N$ of non-tabu neighbours of $s$
4     Choose a best improving solution $s'$ in $N$
5     Update tabu attributes based on $s$ and $s'$
6     $s := s'$
7
```

Last but not least, as mentioned in the previous section, we can combine all of these search strategies with heuristics to prune neighbourhoods that are faster than the computation of single, optimal shortcuts. In particular, we can use hop-minimal pair betweenness to replace each computation of single, optimal shortcuts by choosing a shortcut with maximum rating. The results thus obtained will be discussed in Section 7.2.3.

6.4 Fitness-distance correlation

One question arising in the context of local search algorithms is how well the search landscape under the chosen neighbourhood relation is suited for local search. In this context, the distribution (and number) of local maxima plays an important role. On the other hand, it is of interest if the evaluation function provides means to guide local search algorithms towards good (or optimal) solutions. A strong correlation between the evaluation function value of candidate solutions and their distance to the closest global optimum is an indicator for the latter property. The existence of such a correlation is often referred to as a *big valley structure* (see [Boe96]). The term big valley results from the intuition that an optimal solution is located at the bottom of a valley and surrounded by a large number of local minima whose quality deteriorates with increasing distance to the optimum. In the context of maximization problems, the term *massif central* is more intuitive. The *fitness-distance correlation coefficient* as introduced by Jones and Forest in 1995 (see [JF95]) aims to measure this property.

Definition 4 (Fitness-Distance Correlation Coefficient) Given a candidate solution $s \in S$, let $g(s)$ be the value of the evaluation function of s, and let $d(s)$ be the distance of s to the closest global optimum. Given fitness-distance pairs $(g(s), d(s))$ for all $s \in S$, the *fitness-distance correlation coefficient (FDC coefficient)* is defined as

$$\rho_{\mathrm{fdc}}(g,d) := \frac{\mathrm{Cov}(g,d)}{\sigma(g)\cdot\sigma(d)} = \frac{\langle g(s)\cdot d(s)\rangle - \langle g(s)\rangle\cdot\langle d(s)\rangle}{\sqrt{\langle g^2(s)\rangle - \langle g(s)\rangle^2}\cdot\sqrt{\langle d^2(s)\rangle - \langle d(s)\rangle^2}},$$

where $\mathrm{Cov}(g, d)$ denotes the covariance of fitness-distance pairs $(g(s), d(s))$ over all $s \in S$; $\sigma(g)$ and $\sigma(d)$ are the respective standard deviations of the evaluation function and the distance values for all $s \in S$; and $\langle g(s)\rangle$, $\langle g^2(s)\rangle$, $\langle d^2(s)\rangle$, $\langle g(s) \cdot d(s)\rangle$ denote the averages of $g(s)$, $g^2(s)$, $d^2(s)$ and $g(s) \cdot d(s)$, respectively, over all candidate solutions $s \in S$.

By definition, the minimum and maximum values of the FDC-coefficient are -1 and 1. These extreme values indicate a perfect linear correlation between fitness and distance. If we consider maximization problems, a value of $\rho_{\mathrm{fdc}}(g, d)$ close to -1 suggests that the evaluation function provides sufficient guidance towards globally optimal solutions.

As the complete evaluation of the FDC coefficient would require to evaluate all candidate solutions in the search space, this is obviously infeasible for all but the smallest instance sizes. Thus, instead of computing the exact value of $\rho_{\mathrm{FDC}}(g, d)$, we give an estimation based on a sample of m candidate solutions $\{s_1, \ldots, s_m\}$ with an associated set of fitness-distance pairs $\{(g_1, d_1), \ldots, (g_m, d_m)\}$. The estimate r_{FDC} of ρ_{FDC} is then computed as

$$r_{\mathrm{FDC}} := \frac{\widehat{\mathrm{Cov}}(g, d)}{\hat{\sigma}(g) \cdot \hat{\sigma}(d)},$$

where $\widehat{\mathrm{COV}}(g, d)$ denotes the sample covariance of the pairs (g_i, d_i) and $\hat{\sigma}(g)$ and $\hat{\sigma}(d)$ the sample standard deviations of $G := \{g_1, \ldots, g_m\}$ and $D := \{d_1, \ldots, d_m\}$, respectively.

Instead of taking random samples that are uniformly distributed over the search space, it is typically more interesting to evaluate samples of local maxima (see for example [Boe96]). The idea behind this is the observation that efficient SLS methods consider mainly rather good candidate solutions. For instance, such samples can be obtained using multiple runs of (a randomized version of) Iterative Improvement.

In the context of the BSP, there is a second reason for us to choose this approach: As there are typically a lot more potential shortcuts than the number c of shortcuts in each candidate solution, the probability that two arbitrarily chosen assignments have some shortcuts in common is rather low. As this corresponds to maximum distance between the assignments, samples of local maxima seem to be more interesting.

Finally, as we do not know globally optimal solutions for our test instances, we use the distance to solutions that are best among all solutions found by our local search algorithms instead of the distance to global maxima. In Section 7.3, we will use fitness-distance correlation coefficients to analyze the search space of instances of the BSP using 1-exchange-neighbourhoods.

7 Experimental Results Concerning Local Search

This chapter provides the results of the experiments taken in context with local search algorithms. The experiments in this chapter are performed on one core of an Intel Xeon E5430 clocked at 2.66 GHz, equipped with 12 MB L2 Cache and 16 GB RAM.

7.1 Preliminaries

7.1.1 Test instances

As most of our algorithms are randomized, we will have to run single algorithms on the same test instances rather often to get significant results regarding performance and solution quality. Similarly, as there are a lot of variants of particular local search algorithms, this increases the time needed for a profound evaluation. Therefore, we restrict ourselves to few, rather small graphs. We consider four classes of graphs:

- road graphs centered at Karlsruhe with 171 and 491 nodes, respectively: `ka171`, `ka491`
- grid graphs with edge weights chosen uniformly in $[1, \ldots, 1000]$ and 196 and 506 nodes: `grid196`, `grid506`
- unit disk graphs with average degree 10 and 198 and 499 nodes: `uDisk10_198`, `uDisk10_499`
- unit disk graphs with average degree 20 and 200 and 500 nodes: `uDisk20_200`, `uDisk20_500`

These graphs form a subset of the graphs considered in the evaluation of the algorithms for finding one optimal shortcut. To get a rough idea of the quality of solutions found by our heuristics, the following table lists the sum of all hop-distances in the respective graphs:

instance	sum of hop-distances	instance	sum of hop-distances
`ka171`	336.065	`ka491`	4.854.990
`grid196`	387.872	`grid506`	4.349.470
`uDisk10_198`	497.860	`uDisk10_499`	5.104.758
`uDisk20_200`	481.782	`uDisk20_500`	4.961.860

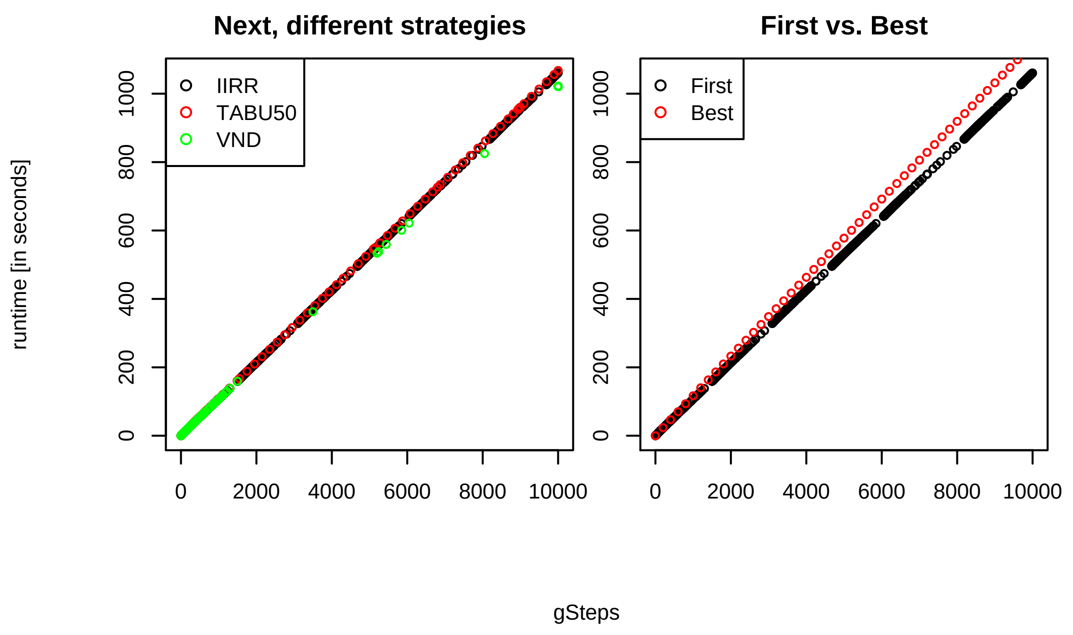


Fig. 7.1: Overall running time after gSteps executions of $\Theta(n^3)$-algorithm (gSteps) for different local search strategies on `uDisk10_198` with 200 shortcuts. In the left, we compare Iterative Improvement Random Restart(IIRR), Tabu Search with tabu tenure 50 (TABU50) and Variable Neighbourhood Descent with a maximum of 1000 neighbours to be considered in each neighbourhood (VND). All variants use Random Order First Improvement and random initialization. The right side shows the corresponding values for Iterative Improvement Random Restart with Best Improvement (Best) and Random Order First Improvement (First).

7.1.2 Running time versus number of computations of single, optimal shortcuts

In this section, we will examine the correlation between running times of our different local search algorithm and the number of times our $\Theta(n^3)$-algorithm is called during the search process (we call this number of calls the number of *gSteps*. In Figure 7.1, corresponding runtime/gSteps-pairs are illustrated. Clearly, there seems to be a linear correlation between these two measures. Further, the average runtime per gStep does not vary substantially if we compare different local search strategies (left side of Figure.7.1). Interestingly, if we compare the running times of the versions with Best Improvement to the ones with Random Order First Improvement, the average running time per gStep is lower in the latter case. On the other hand, if we compare Iterative Improvement, Tabu Search and Variable Neighbourhood Descent with Best Improvement, again, there seems to be no substantial differences among these. Closer analysis shows that this effect has nothing to do with the particular implementation of the respective local search steps. Instead, the running time of our $\Theta(n^3)$-algorithm decreases under the insertion of shortcut assignments with high quality. This can be explained by the fact that the probability that for arbitrary nodes s, a, b in the graph, $g_s(a, b)$ is greater than zero decreases, which saves some calculations. With Random Order First Improvement, we find good solutions a lot quicker. This will be discussed in detail in the next section.

Altogether, there seems to be a close relationship between running time and number of gSteps. Therefore, motivated by [AO96], for the comparison of our local search algorithms that use the $\Theta(n^3)$-algorithm, we measure running times in gSteps. The idea is to get results that are reproducible, independent from implementation and runtime environments and to gain deeper insights in the behaviour of the respective algorithms. However, for the comparison of these algorithms with other heuristics, we will use concrete running times.

Concerning our test instances with about 200 nodes, roughly, we get the following correspondence between running time and number of gSteps:

instance	running time per gStep	instance	running time per gStep
`ka171`	$\approx$ 72.7 ms	`uDisk10_198`	$\approx$ 114.4 ms
`grid196`	$\approx$ 104.9 ms	`uDisk20_200`	$\approx$ 122.7 ms

7.2 Performance of local search algorithms

7.2.1 Finding good solutions fast

In this section, we compare different variants of Iterative Improvement with respect to the time needed to reach a local maximum, the quality of the local maxima found and the diversification of the search process. The latter is of especial importance if we use Iterative Improvement as part of higher-level local search algorithms that depend on reinitialization mechanisms.

Figure 7.2 shows how the average solution quality evolves over time using different kinds of Iterative Improvement. Here, we use the test instances `ka200` and `grid200` with 200 shortcuts each.

The most striking observation is, that the variant of Iterative Improvement using Best Improvement and random initialization takes a lot more time to converge than the other strategies. If we take a closer look at how the solution assignments of the particular test runs develop over time, we see that the variants with random initialization have one property in common. We start with a random assignment of shortcuts that are commonly far from optimal. Roughly, in the first phase of Iterative Improvement, we replace most of these shortcuts greedily, while in the second phase we improve this solution by further local search steps. With Best Improvement, each search step takes 200 gSteps, while with Random Order First Improvement, in the beginning, the probability that an arbitrarily chosen shortcut to replace increases the solution quality by a significant amount is rather high.

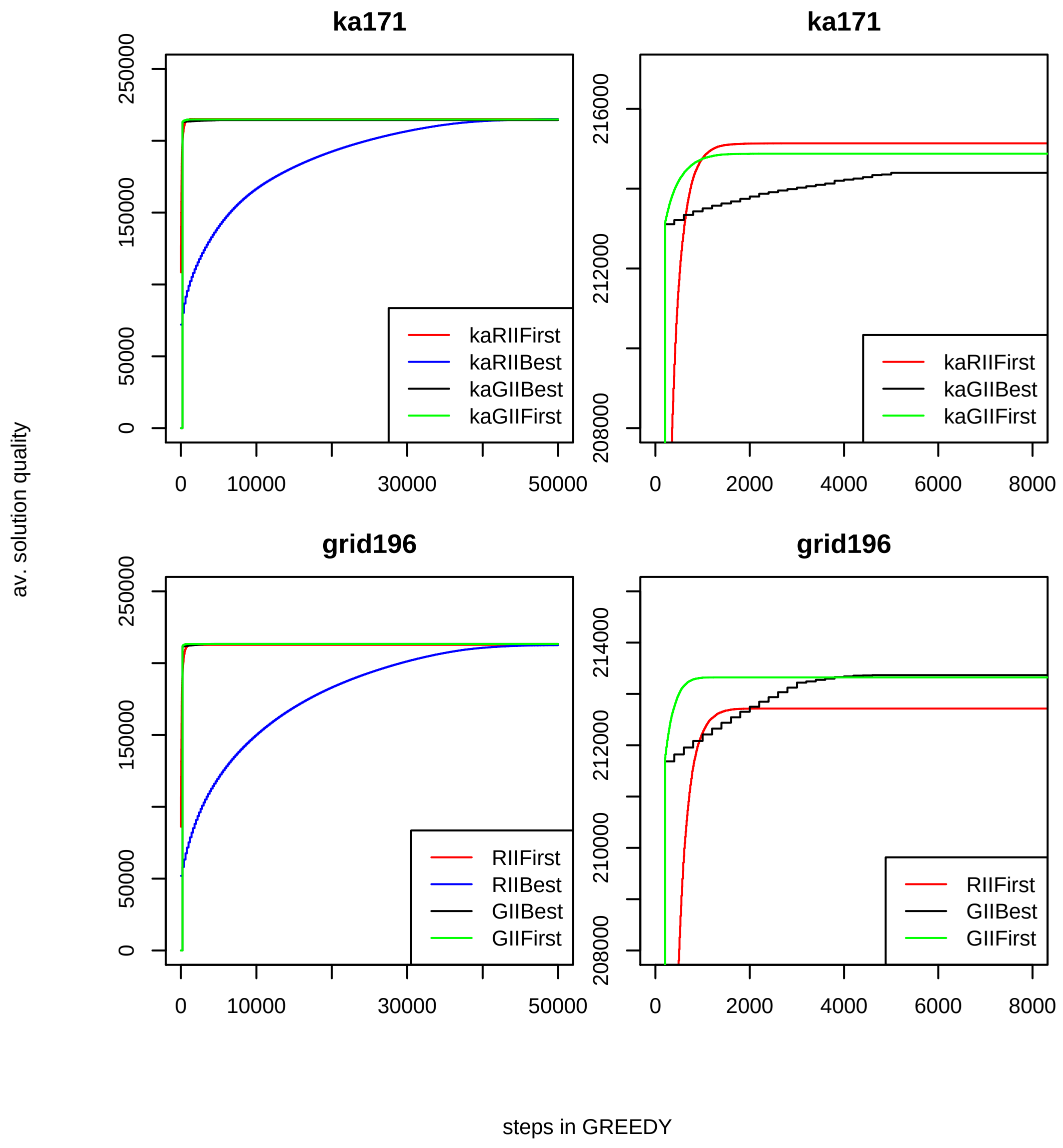


Fig. 7.2: Running times and average solution qualities (over 200 runs) for different variants of Iterative Improvement and different scales: Random initialization and Random Order First Improvement(RIIFirst), Random initialization and Best Improvement (RIIBest), Greedy initialization and Random Order First Improvement(GIIFirst), Greedy initialization and Best Improvement(GIIBest)

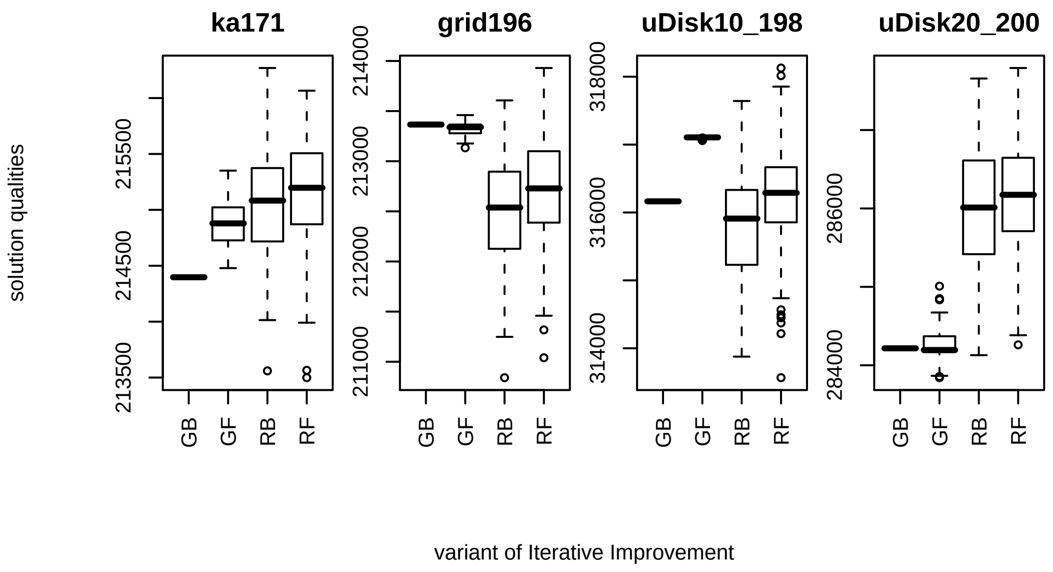


Fig. 7.3: Distribution of solution qualities for different variants of Iterative Improvement (over 200 runs each): random initialization and Random Order First Improvement (RF), random initialization and Best Improvement (RB), greedy initialization and Random Order First Improvement(GF), greedy initialization and Best Improvement(GB)

Considering the average solution qualities the algorithms converge to (visible in the graphs at the right side), we see that the algorithms seem to differ slightly. But if we compare the results for `ka171` and `grid196`, this seems to depend on the test instances rather than the algorithms itself. To compare the particular solution qualities of the local maxima found, we take a closer look at Figure 7.3.

First, we see that both random initialization and Random Order First Improvement create some diversification in the search process that leads to different local maxima. Second, there seems to be no general way to predict which variant of Iterative Improvement leads to the best local maxima without considering specific test instances. The variant with random initialization and Best Improvement seems to perform slightly worse than the variant with Random Order First Improvement. This can be explained by the use of a cut-off time of 50000 gSteps. Test runs that didn't reach a local optimum after this time (between 12% and 27% of the runs with random initialization and Best Improvement) stay unconsidered. Intuitively, one could expect that these test runs lead to better solution qualities, which would explain the slightly worse performance.

Conclusion. Summarizing, the question of which variant of Iterative Improvement to use depends on the application. If we just want to find a good solution as fast as possible, we should use the variant with greedy initialization and Random Order First Improvement, as this version is likely to reach a local minimum after comparatively little time. If we want to use Iterative Improvement as part of Iterative Improvement Random Restart, the variant with random initialization and Random Order First Improvement seems to be the best choice, as this version provides sufficient diversification and finds local maxima adequately fast.

7.2.2 Using escape strategies

In this section, we will compare the performance of Tabu Search, Variable Neighbourhood Descent and simple Iterative Improvement Random Restart.

Reasonable Configurations. If we consider all degrees of freedom, there are a lot of possible combinations between pivoting rule, initialization strategy and algorithm-specific parameters (as tabu tenure and maximal size of the neighbourhood to be considered).

Thus, we restrict ourselves to some of these combinations. First, as already mentioned, for Iterative Improvement Random Restart, the variant with random initialization and Random Order First Improvement seems to be by far the best choice. Preliminary experiments showed that concerning the performance of the other local search strategies, the initial search position or rather the first local maximum found is of great importance. Therefore, if we choose greedy initialization, this influences the solution quality of the best found assignment in a way that is very specific to the test instance. As our goal is to compare the mean behaviour of the particular search strategies (independent from the initial search position), we choose random initialization instead and perform a set of sample runs. Recalling the results of the previous section, in this case, Best Improvement would lead to unacceptably long running times until we reach rather good solutions. Hence, we consider only the variants with random initialization and Random Order First Improvement in our experiments.

Note that if we just aim to use Tabu Search or Variable Neighbourhood Descent to reach a good solution once, we can use greedy initialization as well, as this reduces the time to find the first local maximum. Compared to the cut-off times we used in our experiments, this is negligible.

For Tabu Search, we have to tune the tabu tenure tt (in terms of search steps). As comparing the long-time behaviour of Tabu Search with 200 shortcuts under different values for tt is a very expensive task – we have to perform a multitude of test runs to get significant results – in the first step we consider only 14 shortcuts. Figure 7.4 shows the resulting performance with tabu tenures 5, 10 and 50. Using a tabu tenure of 50, we get the best results (even for test instance `uDisk20_20`, the average solution quality in the end is slightly better than with the other choices of tt). Analyzing some particular runs with lower tabu tenure, we frequently observe cyclic behaviour. Some test runs with 200 shortcuts and different tabu tenures confirmed that a tabu tenure of 50 seems to be a sensible choice for this case as well. Therefore, for the experiments with Tabu Search and 200 shortcuts, we use a tabu tenure of 50.

Similarly, for our variant of Variable Neighbourhood Descent, we have to specify the maximum number of neighbours r in each neighbourhood that will be evaluated. After some preliminary considerations and test runs, we set r to 1000.

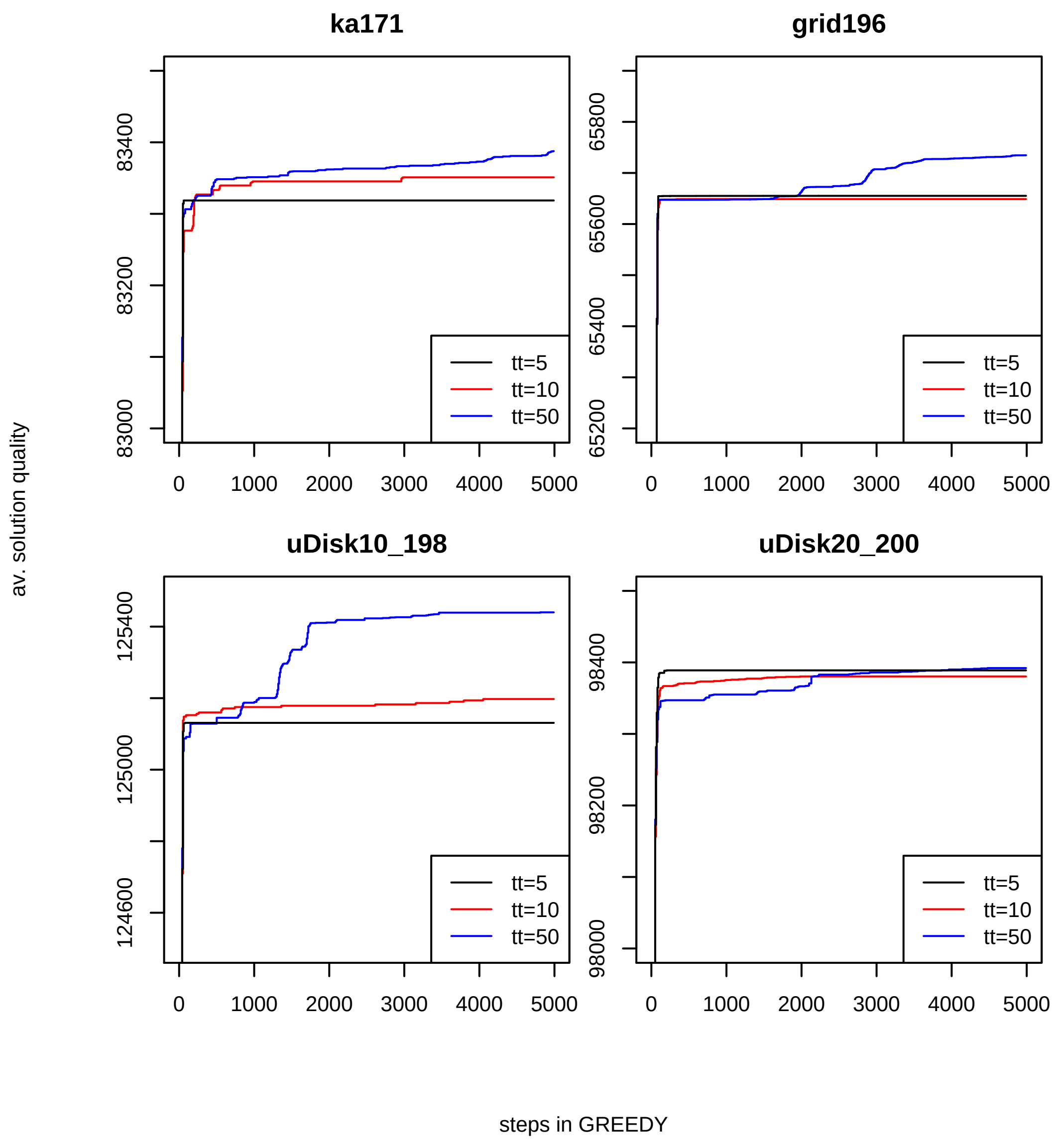


Fig. 7.4: Average long-time behaviour (over 200 runs) of tabu search on different test instances with 200 nodes and 14 Shortcuts depending on tabu tenure *tt* – random initialization and next improving neighbour strategy were used

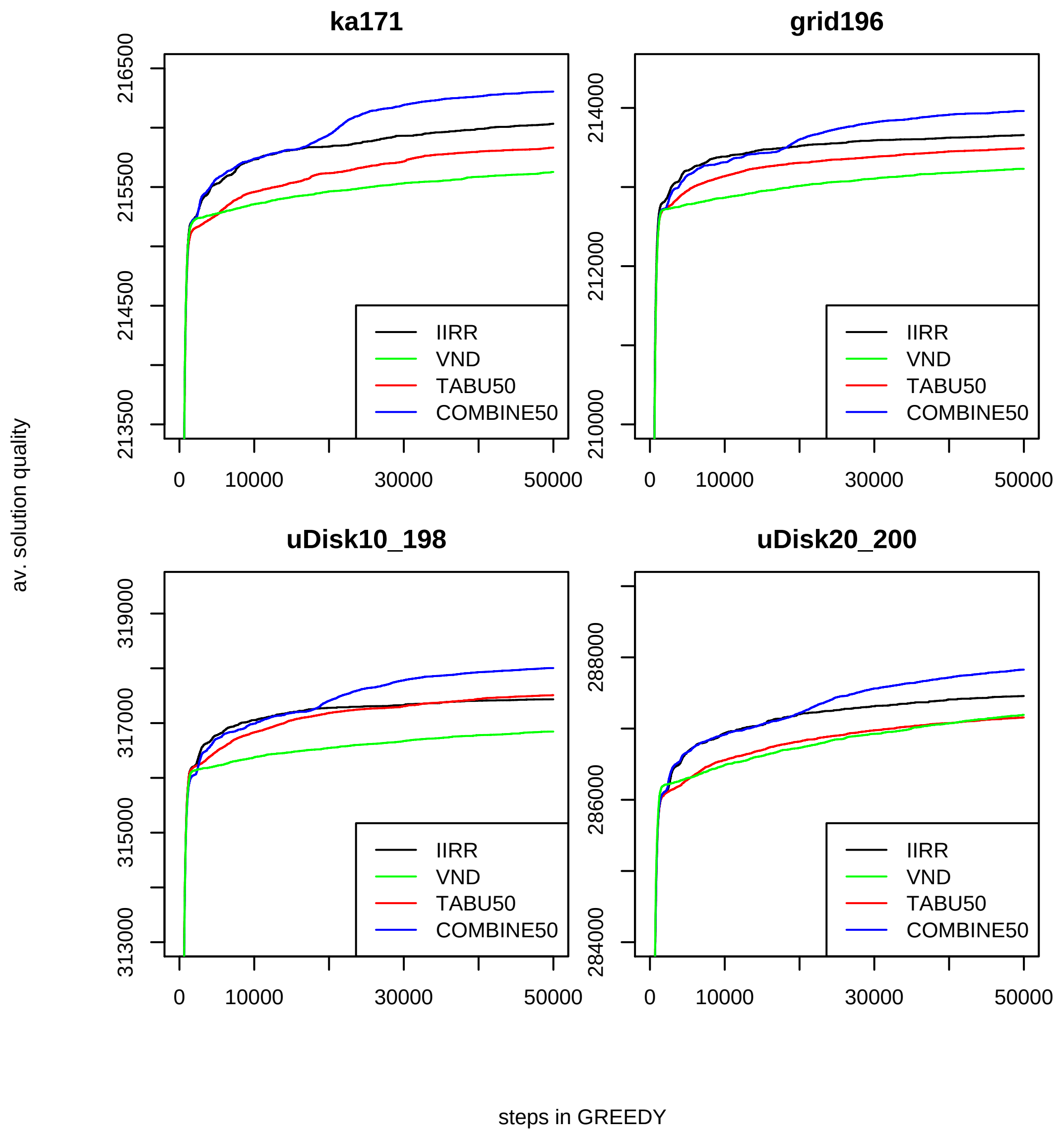


Fig. 7.5: Average long-time behaviour (over 100 runs) of different local search strategies on test instances with 200 nodes and 200 shortcuts : Iterative Improvement Random Restart (IIRR), Variable Neighbourhood Descent with maximum 1000 neighbour samples (VND), Tabu Search with tabu tenure 50 (TABU50) and Combine Search with tabu tenure 50 – each algorithm in the variant that uses the next-improving neighbour strategy

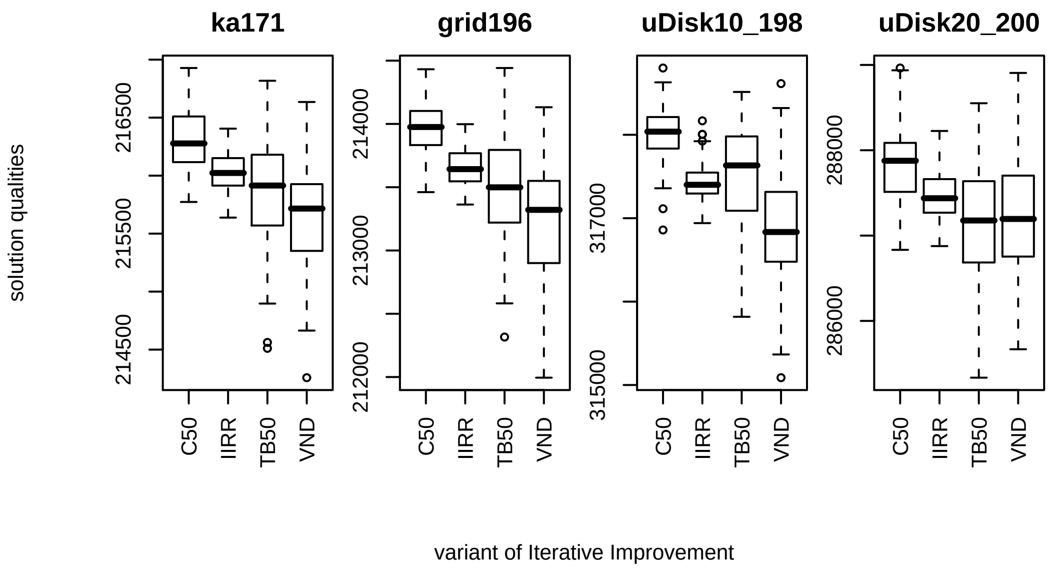


Fig. 7.6: Distribution of solution qualities for different local search strategies (over 100 runs each) on test instances with 200 shortcuts: Iterative Improvement Random Restart (IIRR), Tabu Search with tabu tenure 50 (TB50), Variable Neighbourhood Descent (VND) and Combine Search with tabu tenure 50 (C50)

Results. Figure 7.5 shows the development of the average solution quality of Iterative Improvement Random Restart, Tabu Search and Variable Neighbourhood Descent under the parameters specified above. The first observation we make is, that none of the search strategies seems to stagnate over the considered time interval. If we compare the different search strategies, simple Iterative Improvement Random Restart achieves the best results, which might be surprising at first glance. But if we have a closer look at the number of gSteps we need to find a local maximum from an arbitrary search position (typically between 800 and 2000 for the instance `ka171`), we see that this corresponds to comparatively little computational time. In contrast to that, if we use for example Tabu Search, one single search step typically already corresponds to 200 gSteps. Thus, Tabu Search seems to need too much time to find better solutions to compete with simply sampling local maxima in the considered time interval.

Figure 7.6 shows the distribution of the corresponding solution qualities of the particular runs after 50000 search steps. Interestingly, the solution qualities of Tabu Search and Variable Neighbourhood Descent scatter over a much wider range. Despite the worse mean behaviour of Tabu Search and Variable Neighbourhood Descent, the best solutions were found by some runs of these local search strategies. This can be explained by the fact that the performance of Tabu Search and Variable Neighbourhood Descent depends a lot on the initial search position.

Enhancements. This leads to the idea of *Combine Search.* Combine Search is a combination between Iterative Improvement Random Restart and Tabu Search. Thereby, we spend one third of the overall time on searching a local maximum with reasonably high solution quality using Iterative Improvement and use this optimum as initial search position for Tabu Search. Figure 7.5 and 7.6 show the mean behaviour and distribution of the resulting solution qualities for Combine Search with tabu tenure 50. Regarding our test instances, Combine Search shows the best behaviour among all search strategies considered.

7.2.3 Using maximum pair betweenness instead of $\Theta(n^3)$-algorithm

Here, we will compare the results we obtain if we replace each computation of single, optimal shortcuts by determinining a shortcut that is rated best according to hop-minimal pair betweenness. We consider the greedy strategy, Iterative Improvement and the local search strategies evaluated in the previous section. In particular, we take a look at the development of solution qualities of our different local search strategies in combination with shortcut ratings.

Approximately 200 nodes

In the first step, we consider Iterative Improvement with random initialization and Random Order First Improvement. The following table illustrates the results for our test instances with approximately 200 nodes and 14 and 200 shortcuts, respectively. We give the average running time (in seconds) of Iterative Improvement using our $\Theta(n^3)$-algorithm (II) and using shortcut ratings according to hop-minimal pair betweenness (II-APPROX). Moreover, we compare the average quality of the assignments thus obtained. In the context of II, these assignments constitute true local maxima, whereas with II-APPROX, this is not guaranteed. The results were sampled over 200 runs of Iterative Improvement each.

For comparison, we also list the quality of the best assignments found by our local search algorithms (BEST). As we optimized the implementation of Iterative Improvement after the experiments, the respective running times for II are obtained using the results of Section 7.1.2 and the number of gSteps.

instance	shortcuts	BEST	II		II-APPROX	
		qual	∅ time	∅ qual	∅ time	∅ qual
`ka171`	14	83508	3.11	83301	1.08	82471
	200	216547	105.38	215137	51.46	214125
`grid196`	14	65744	3.84	64260	1.65	64304
	200	214442	150.69	212715	74.58	211579
`uDisk10_198`	14	125555	4.51	125139	1.98	125096
	200	318801	167.17	316201	75.47	314440
`uDisk20_200`	14	98400	5.30	98200	2.17	98181
	200	288964	176.31	286158	78.13	284918

The first observation we make, is that the mean quality of the assignments does not differ substantially. In particular, the results obtained by using shortcut ratings are always less than one percent worse than the assignments found using single, optimal shortcuts. For instance `grid196` and 14 shortcuts, we even get a better result in combination with hop-minimal pair betweenness. Using shortcut ratings, we can decrease running times by more than 50 percent compared to the version that depends on our $\Theta(n^3)$-algorithm.

Furthermore, Figure 7.7 illustrates the long-time behaviour of our local search strategies in combination with shortcut ratings. Like before, we use a cut-off time of 50000 calculations of shortcut ratings, which corresponds to approximately 35-50 minutes, depending on the particular test instance. The development of average solution qualities is very similar to the results of the previous section. In particular, none of the search strategies seems to stagnate over the considered time interval. The solution qualities are worse than using our $\Theta(n^3)$-algorithm, but not significantly. As the calculation of shortcut ratings is possible for larger graphs, this offers an opportunity to use local search strategies in these instances as well.

Approximately 500 nodes

The following table lists the solution quality of assignments found using the greedy strategy in our test instances with approximately 500 nodes and 500 shortcuts. We show the running times and solution quality of greedy in combination with our $\Theta(n^3)$-algorithm (GREEDY) and in combination with shortcut ratings according to hop-minimal pair betweenness (GREEDY-APPROX).

instance	shortcuts	GREEDY		GREEDY-APPROX	
		time	qual	time	qual
`ka491`	22	33.12	1682223	4.66	1682223
	500	726.41	3663346	123.16	3640261
`grid506`	22	34.21	927160	5.75	927160
	500	792.86	2921768	148.51	2906936
`uDisk10_499`	22	39.85	1144576	5.99	1144576
	500	865.73	3632104	145.41	3615623
`uDisk20_500`	22	39.51	1146318	5.96	1146318
	500	851.56	3493348	150.17	3486303

Again, the quality of the results obtained does not differ much. Considering shortcut assignments with approximately $\sqrt{n}$ shortcuts, we even get the same quality if we replace the computation of single, optimal shortcuts by shortcut ratings. For the experiments with 500 shortcuts, despite the partially discouraging results in Section 5.3.2, the quality decreases by less than one percent if we consistently add shortcuts with maximum rating.

Summarizing, we do not get significantly worse results when using shortcut ratings instead of optimal shortcuts, while saving over 80 percent running time in all of the test instances considered in this section.

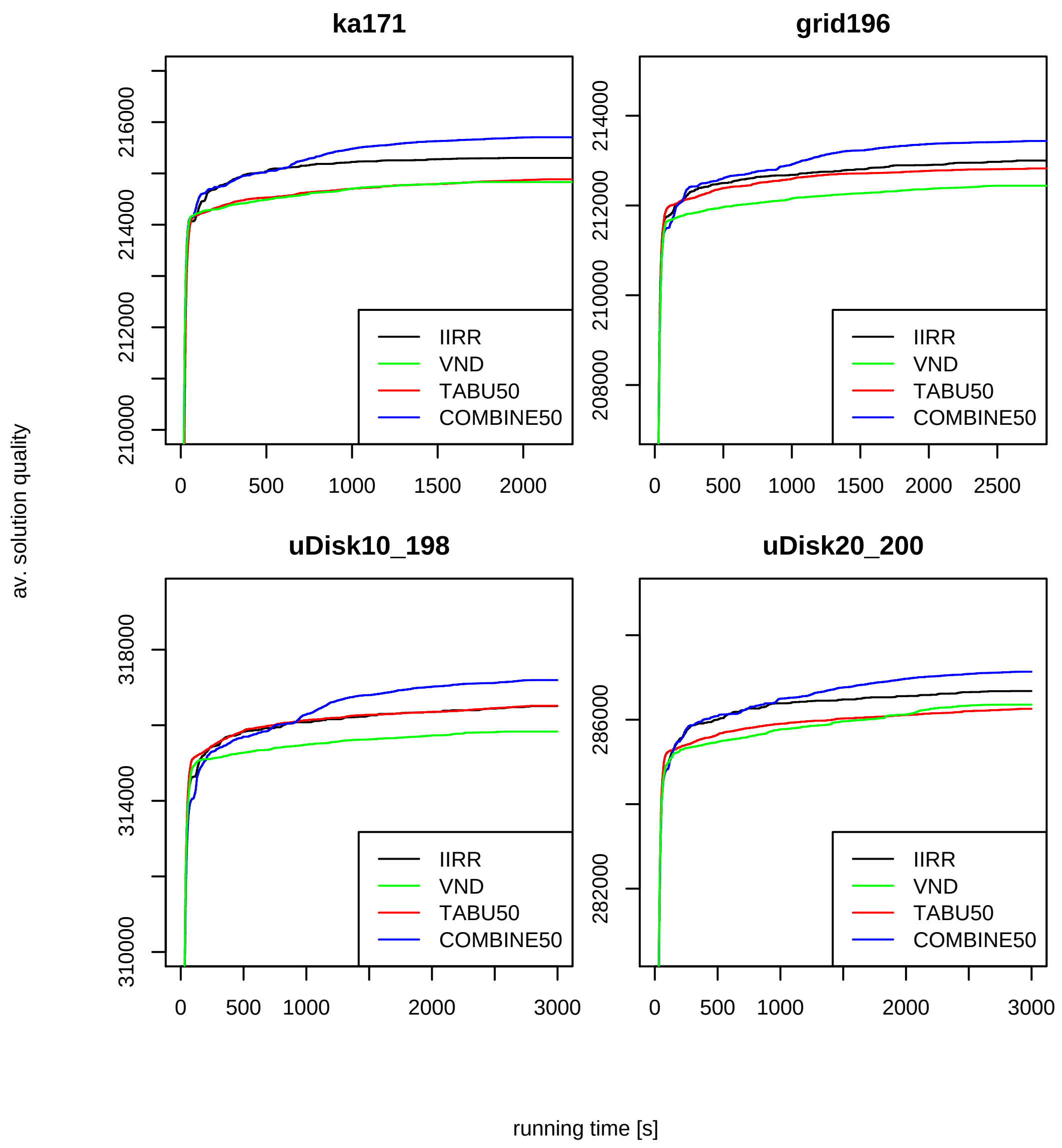


Fig. 7.7: Average long-time behaviour (over 50 runs) of different local search strategies on test instances with 200 nodes and 200 shortcuts using shortcut ratings according to hop-minimal pair betweenness : Iterative Improvement Random Restart (IIRR), Variable Neighbourhood Descent with maximum 1000 neighbour samples (VND), Tabu Search with tabu tenure 50 (TABU50) and Combine Search with tabu tenure 50 – each algorithm in the variant that uses the next-improving neighbour strategy

7.2.4 Comparison with trivial bounds and static rating

As a last aspect concerning the solution quality obtained by the greedy strategy and by our local search algorithms, we will try to put the results in a greater context. To this end, we use simple upper bounds for the quality of shortcut assignments and compare the assignments found in our experiments with assignments obtained by using simple (static) heuristics.

The most natural way to get an idea of the quality of solutions found by heuristics would be to compare the results with an optimal solution. As we did not find a way to compute optimal solutions for our test instances within reasonable time bounds, we use upper bounds instead. Let $(a_0, b_0), \ldots, (a_{c-1}, b_{c-1})$ denote the c shortcuts with the highest values for $(h(a_i, b_i) - 1) \cdot |P(a_i, b_i)|$. Then,

$$B_1(G) = \sum_{i=0}^{c-1} |P(a_i, b_i)| \cdot (h(a_i, b_i) - 1)$$

is an upper bound for $w(S)$ for any shortcut assignment S with c shortcuts. This is true, as, for all shortcuts (a, b) in S, $|P(a, b)| \cdot (h(a, b) - 1)$ is an upper bound for the decrease in overall hop-lengths we get by inserting a shortcut between a and b, independent from the other shortcuts in S.

As this approach does not yield reasonably tight upper bounds for our test instances with approximately n shortcuts, we also use another simple bound for the quality of shortcut assignments. We determine $H = \sum_{s,t \in V} h(s, t)$, which is a trivial upper bound for $w(S)$. This can be slightly improved by using the fact that the hop-distance between each pair of distinct nodes that is not linked by an edge or a shortcut is at least 2. Thus, we obtain

$$B_2(G) = H - 2(n(n-1) - (m + c)) - m - c$$

as another upper bound for the decrease in overall hop-lengths. Concretely, we compare the following bounds and algorithms:

- BOUND: The respective minimum of $B_1(G)$ and $B_2(G)$
- BEST: The quality of the best assignment found by our local search algorithms
- GREEDY: The quality we get by using the greedy strategy
- STAT-PB: The quality we get by inserting the c (non-adjacent) shortcuts with the highest hop-minimal pair betweenness in the underlying graph. Deviating from the ratings we used up to now, we do not multyply the betweenness values with the hop-distance between the end-nodes. This is motivated by the thought that hop-distances in the graph change considerably during the insertion of shortcuts.
- STAT-B: The quality of the assignment found by inserting the c (non-adjacent) shortcuts (a_i, b_i) with the highest values for the minimum of the node betweenness of a and b

The results can be found in Table 7.1. First, we see that the qualities obtained by STAT-PB and STAT-B are significantly lower than with the greedy strategy and local search.

instance	shortcuts	BOUND	BEST	GREEDY	STAT-PB	STAT-B
ka171	14	160480	83508	82931	27085	30130
	200	278528	216547	213111	118583	95164
grid196	14	101100	65744	64110	26874	23454
	200	312360	214442	211686	120930	82847
uDisk10_198	14	186892	125555	121786	24222	28266
	200	422046	318801	313172	150038	134354
uDisk20_200	14	160886	98400	95582	31484	28620
	200	406382	288964	282856	134314	74704
ka491	22	4374961	-	1682223	390396	414700
	500	4375439	-	3663346	1956094	1152926
grid506	22	1550450	-	927160	218652	199285
	500	3840844	-	2921768	1418464	852536
uDisk10_499	22	2070150	-	1144576	242576	253679
	500	4613252	-	3632104	1503058	1129797
uDisk20_500	22	2526782	-	1146318	183004	263096
	500	4473360	-	3493348	1468832	1067382

Table 7.1: Comparison of the quality of shortcut assignments found using different heuristics and simple upper bounds for the decrease in overall hop-lengths

Moreover, for our test instances with approximately n shortcuts, the results obtained by using hop-minimal pair betweenness are better than the ones using minimal node betweenness. The improvements obtained by using local search instead of the greedy strategy always correspond to less than three percent of the solution quality. Compared to the rough upper bounds we used, the solution qualities found by local search and the greedy strategy seem to be reasonably high. In particular, for the test instances we considered, the greedy strategy achieved much better solution qualities than guaranteed by the theoretical analysis in [BDDW09].

7.3 Fitness-distance analysis and number of shortcuts

In this section, we will evaluate fitness-distance correlations and examine the influence of the number of shortcuts on the quality of solutions found by the GREEDY-strategy.

Fitness-distance correlation. For the analysis of the former, we have to choose between different versions of Iterative Improvement to obtain a set of local maxima samples. As already mentioned, the version with random initialization and Random Order First Improvement provides the most diversification, while the mean time of reaching a local maximum stays moderately low. Thus, to sample local maxima, we perform 500 runs of Iterative Improvement with random initialization and Random Order First Improvement on our test instances with approximately 200 nodes. The following table lists the resulting values for the number of different local maxima found and the fitness-distance correlation of the sample local maxima with 14 and 200 shortcuts each.

	14 shortcuts		200 shortcuts	
instance	l. m.	FDC coeff.	l. m.	FDC coeff.
ka171	11	−0.7644288	476	−0.66995
grid196	18	−0.9211459	488	−0.724163
uDisk10_198	17	−0.5761276	469	−0.7133596
uDisk20_200	11	−0.5943998	492	−0.4988985

The fitness distance correlation coefficient of all configurations is rather high, indicating a search landscape and neighbourhood relation that provides good conditions for higher-level local search mechanisms. The experiments with 14 shortcuts yield very few local maxima, while the experiments with 200 shortcuts confirmed the results on the great diversification of this variant of Iterative Improvement on test instances with approximately n shortcuts.

Figure 7.8 shows the corresponding fitness-distance plots for the test instances with 200 shortcuts. The (linear) correlation between fitness and distance indicated by the corresponding correlation coefficient is clearly visible. Another striking observation is, that the quality of the local maxima found does not differ much. The quality of the local maxima with the worst fitness is less than 2 percent away from the respective quality of the best solution found. This coincides with the observation that the solutions found by our higher-level local search algorithms are not substantially better than the solutions of simple Iterative Improvement.

Taking a closer look at the range of distances of local maxima from the best solution found, we see that the closest local maximum is approximately 40 search steps from the best solution and the farthest local maximum almost 160 search steps. Here, the local maxima for grid200 are substantially closer to the best solution than the ones for uDisk20_200. Clearly, the maximum possible distance between two candidate solutions is 200. From this arises the question if, for example for instance grid200, there exists a kind of “core assignment” that consists of a set of shortcuts that all local maxima have in common. Closer analysis shows that this is not the case. The first 100 local maxima have 4 shortcuts in common, but there is no shortcut that is contained in all 488 local maxima. The same holds for the other test instances.

Number of shortcuts. Next, we take a closer look at the solution quality of the assignment found by the GREEDY-strategy depending on the desired number of shortcuts to insert. Figure 7.9 shows the development of the solution quality we get when adding the assignment found. As might be expected, the first 500 shortcuts are responsible for a great part of the decrease in overall hop-length while the following shortcuts improve the solution by less and less. The red line marks the first shortcut that decreases the overall hop-length by 1. This means that at this point, the hop-distance between arbitrary nodes in the graph is at least 2 (as if this is not the case, a better shortcut exists). From this

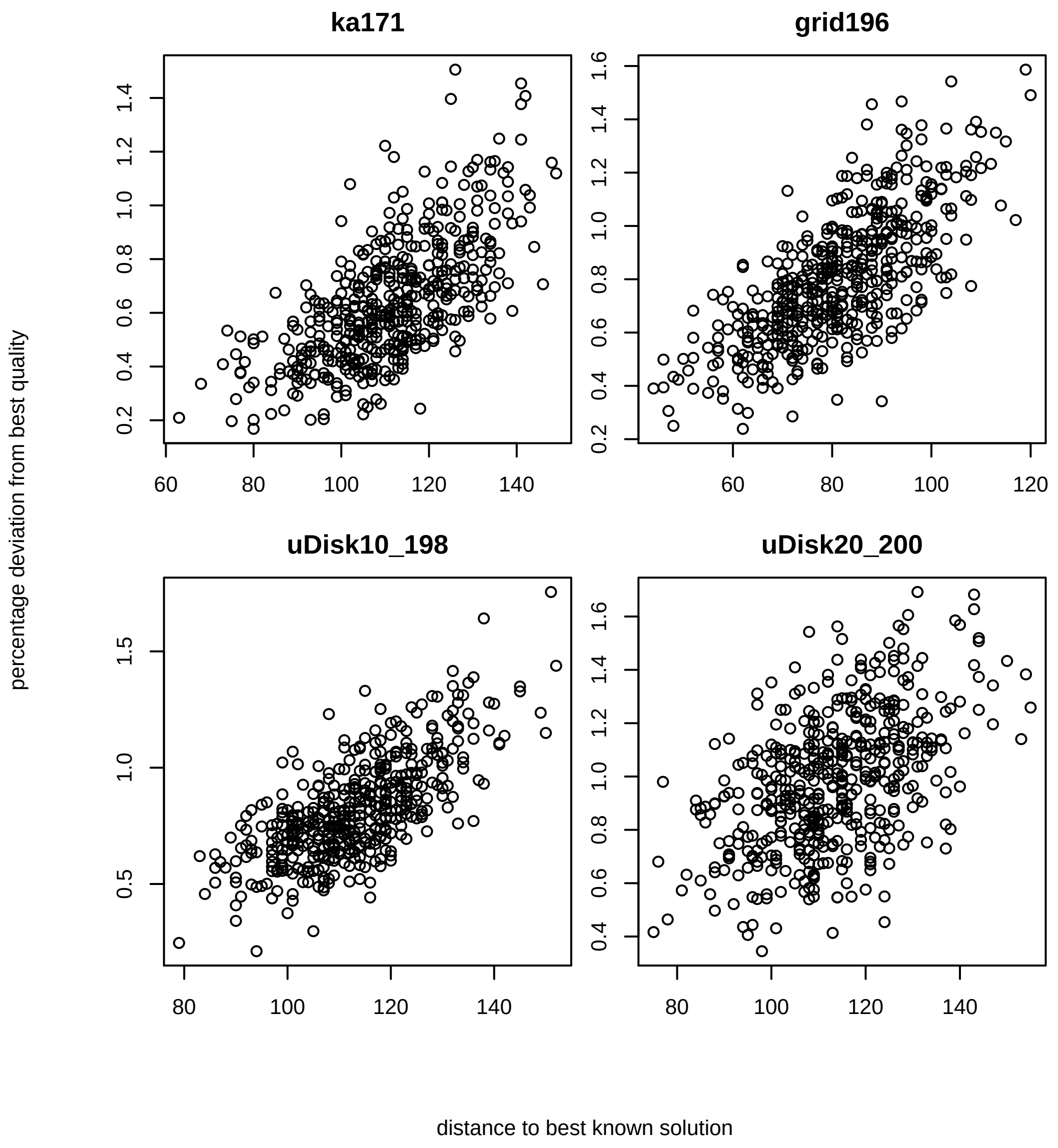


Fig. 7.8: Fitness-distance plots for local maxima found by 500 runs of Iterative Improvement with random initialization and Random Order First Improvement on different test instances with 200 shortcuts

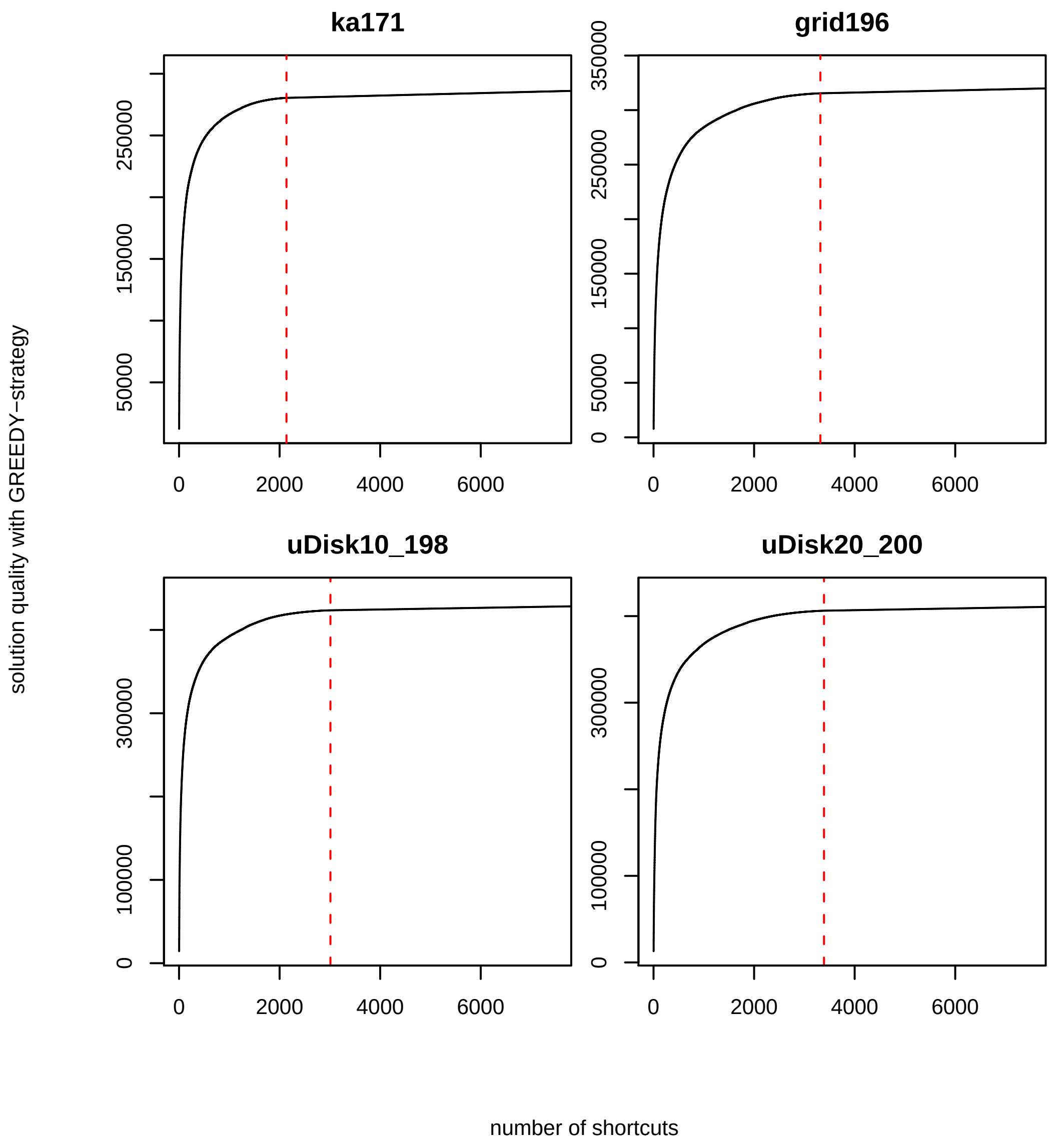


Fig. 7.9: Solution quality of GREEDY-strategy depending on the number of shortcuts in the resulting assignment. The red, dashed line marks the first shortcut that shortens solely the hop-distance between its end-nodes.

point on, the graph becomes linear until each pair of nodes is either linked by an edge or a shortcut.

7.4 Properties of high-quality shortcut assignments

In this section, we will take a look at the assignments found by our local search algorithms. Figure 7.10 shows the shortcut assignment S with maximum $w(S)$ under all assignments found in test instance `ka171` with 200 shortcuts. For the sake of clarity, edges and shortcuts are shown without direction.

The perhaps most striking observation is the existence of "super-nodes" that are incident to a great number of shortcuts. For example, the "super-node" situated a little below the middle of the illustration is incident to 50 shortcuts. This effect is even more obvious in the assignment found in the grid graph shown in Figure 7.11. The astonishingly good behaviour of the degree criterion in graphs with shortcuts added greedily that we observed in Section 5.3.2 can be explained by this property.

Furthermore, there are only few shortcuts that link nodes with low hop-distance in the underlying graph. This can explain the poor performance of the reach criterion in Section 5.3.2, as nodes with high reach tend to be accumulated close to each other in the center of the graphs we considered.

To get an idea how "super-nodes" could be characterized independent from concrete shortcut assignments, we take a look at Figure 7.12, where node betweenness values for the test instance `grid196` are illustrated. If we compare these values to the assignment in Figure 7.11, we see that nodes that are incident to a lot of shortcuts have high betweenness. Conversely, not all nodes with high betweenness are "super-nodes". This could be explained by the fact that a lot of nodes with high betweenness values are located rather close to each other in the center of the graph. Thus, linking all nodes with high betweenness would lead to shortcuts that correspond to paths with very few edges in the underlying graph.

The assignments found in the other graphs considered confirm these results. Figure 7.13 shows the number of shortcuts nodes are incident to in graphs with approximately 500 nodes and 500 shortcuts. Assignments for the test instances `uDisk10_198` and `uDisk20_200` can be found in the appendix.

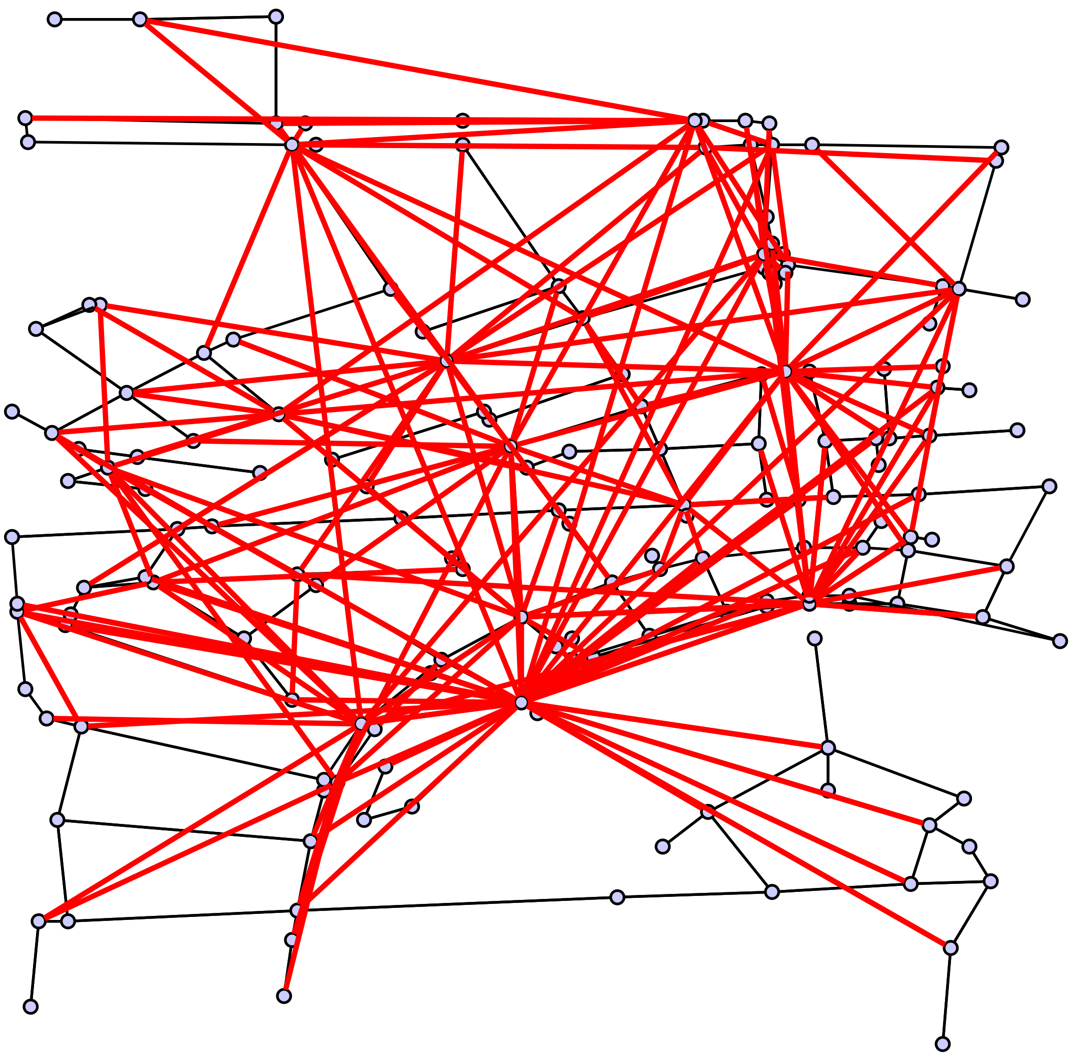


Fig. 7.10: Best assignment found with our local search algorithms for road graph centered in Karlsruhe with 171 nodes and 200 shortcuts (`ka171`). This solution was found by a run of Combine-Search after appr. 48 minutes.

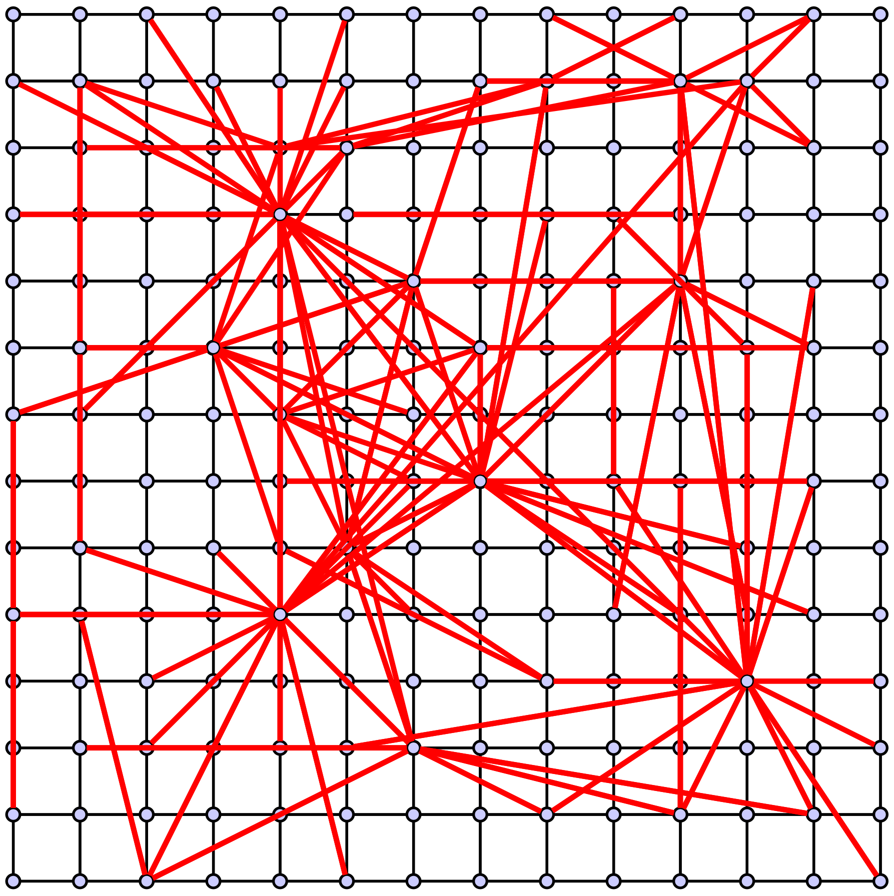


Fig. 7.11: Best assignment found with our local search algorithms for grid graph with 196 nodes, edge weights in $[1, \ldots, 1000]$ and 200 shortcuts (`grid196`). This solution was found by a run of Tabu-Search after appr. 57 minutes.

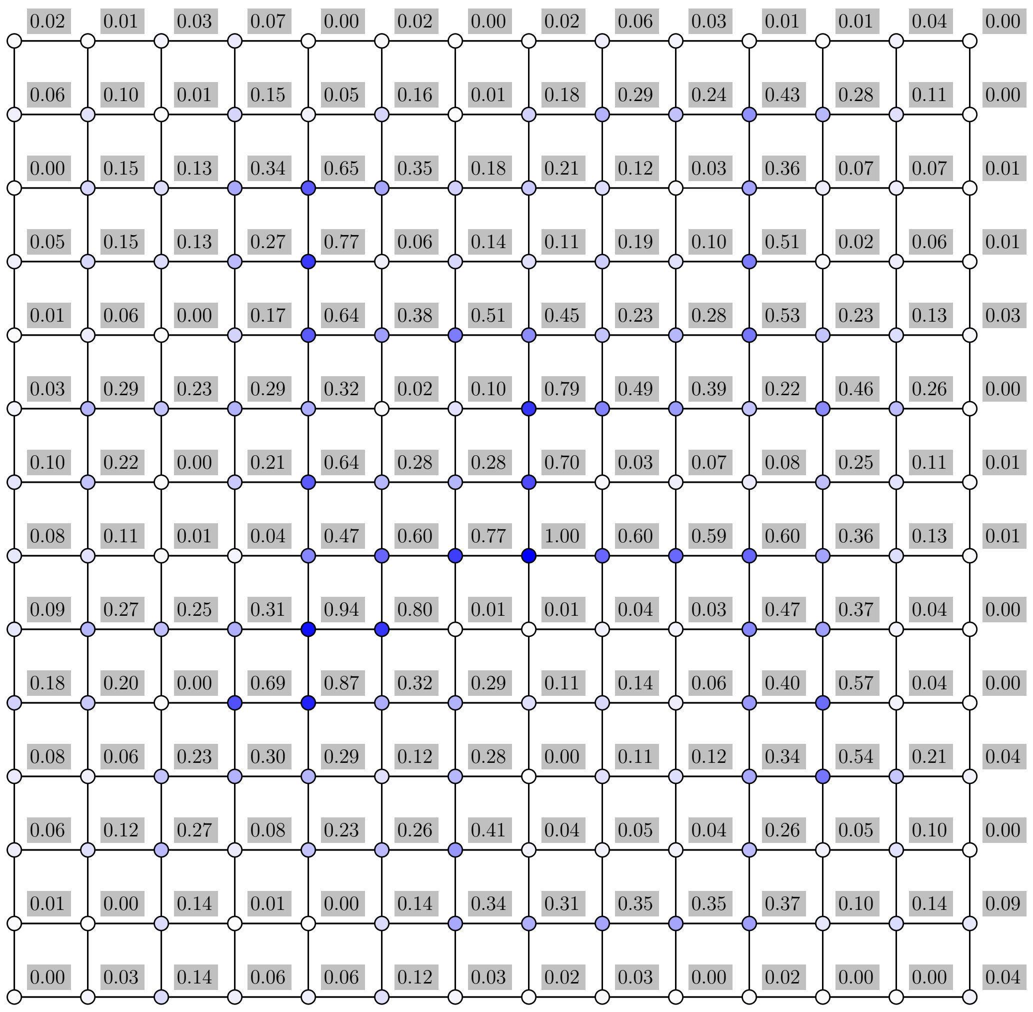


Fig. 7.12: (Normalized) node betweenness centrality for grid graph with 196 nodes, edge weights in $[1, \dots, 1000]$ and 200 shortcuts (`grid196`)

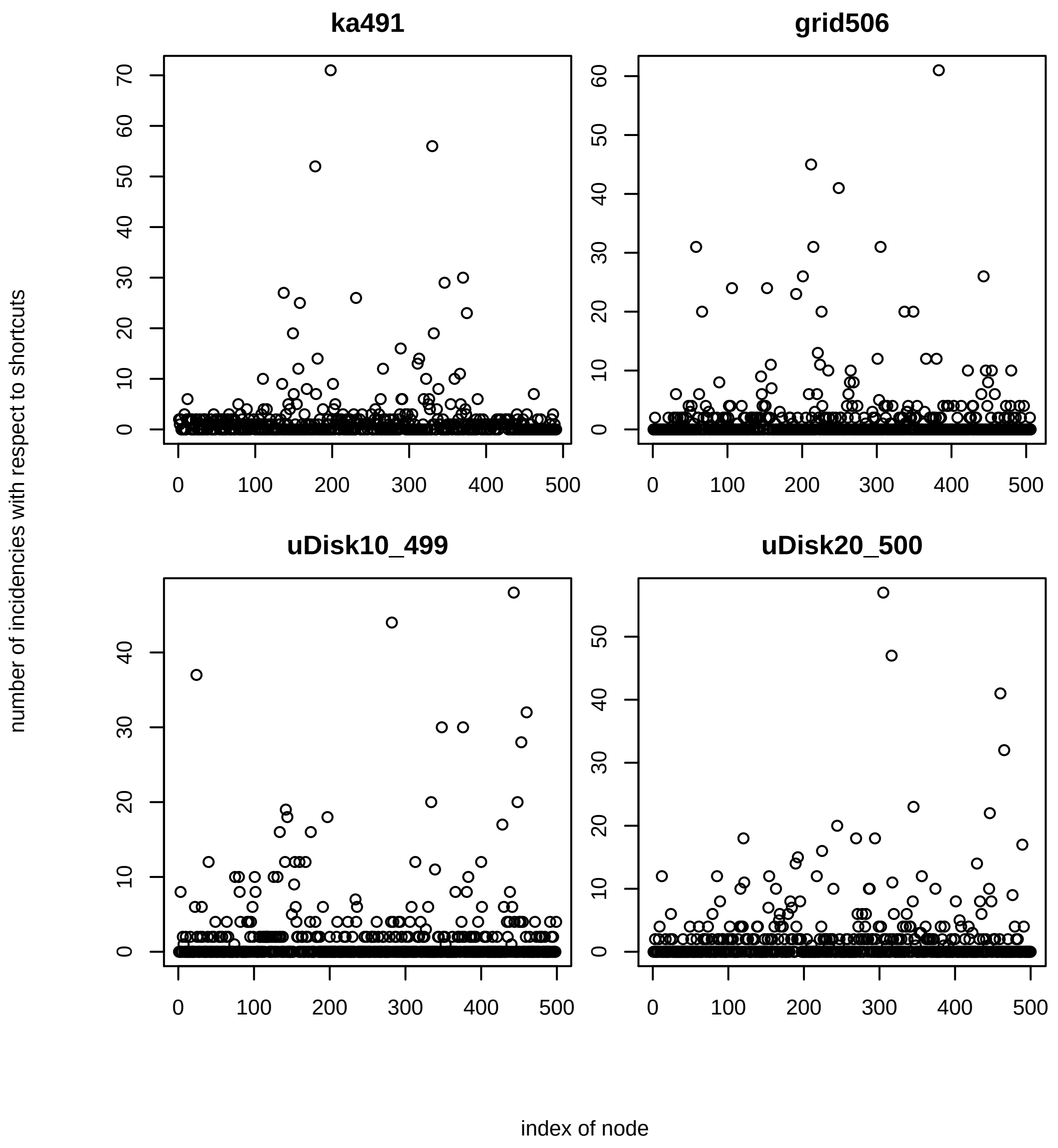


Fig. 7.13: Number of incidencies of nodes with respect to shortcuts for different test instances with 500 shortcuts and assignment found by Iterative Improvement with GREEDY-initialization and next improving neighbour strategy

8 Final Remarks

Conclusion. This thesis concentrated on the Best Shortcut Problem(BSP), which consists of adding a fixed number of shortcuts to a graph, such that the expected number of edges that are contained in an edge-minimal shortest path between two nodes is minimal. Our main aim was to develop high-quality heuristics for the BSP as a self-contained topic, focusing on the greedy algorithm introduced in [BDDW09] and different local search strategies.

In the first part, we developed algorithms to solve the BSP restricted to one shortcut faster than with brute force. As a first approach, we used two kinds of upper bounds for the quality of single shortcuts to prune the search space. Both of these bounds proved to be able to decrease the running time for finding single, optimal shortcuts considerably. These results could be improved by developing an algorithm that evaluates all shortcuts simultaneously with a time complexity in $\Theta(n^3)$, which is a major improvement compared to the time complexity of the brute-force approach. To confirm this theoretical analysis, we conducted experiments using different graph classes.

Additionally, as a time complexity in $\Theta(n^3)$ is still infeasible for larger graphs, we proposed heuristics to give estimations on the quality of single shortcuts that can be determined with less computational overhead. To this end, we stated different generalizations of node betweenness and node stress to pairs of nodes that are especially suited to the BSP. In the special case that shortest paths in the underlying graph are unique, the estimations obtained by these pair centralities are exact. Moreover, based on the ideas behind Brandes' algorithm, we outlined algorithms to determine pair centralities with a time complexity in $O(n \cdot (n \log n + m))$.

In the second part, we used the algorithms developed to this point in combination with the greedy strategy and with different local search algorithms. For the former, the resulting solution quality turned out to be far better than guaranteed by theoretical analysis. Regarding local search, we adapted the strategies Iterative Improvement, Tabu Search and Variable Neighbourhood to the BSP. Using extensive experimental evaluation, we examined the question, whether these strategies could be successfully applied to the BSP. With respect to our test instances, all of these strategies turned out to achieve further improvements concerning solution quality. In this context, a combination between Iterative Improvement with random restarts and Tabu Search showed the best behaviour. We further analyzed the search space of local search under 1-exchange neighbourhoods using a sample of local maxima, confirming the general suitability of the BSP for local search algorithms.

Moreover, concerning the greedy strategy and local search, we evaluated the solutions obtained by using shortcuts rated best according to pair centralities instead of using single, optimal shortcuts. Our experiments suggested that the solution quality thus obtained is only slightly worse than with the base algorithms. This gives us an opportunity to use these strategies in larger graphs as well.

We concluded our work with studying the impact of the number of shortcuts on the solution quality obtained with the greedy strategy. Moreover, we took a closer look at particular shortcut assignments with high quality and stated some observations concerning the structure of these solutions, especially the existence of “super-nodes” that are incident to a great number of shortcuts.

Outlook. From the theoretical point of view, the perhaps most interesting open question is, if there exists a polynomial constant factor approximation algorithm for the Best Shortcut Problem. As the BSP is proven to be NP-hard, there is little hope on finding optimal solutions for larger graphs in general. Nonetheless, an important task would be to compute optimal solutions for small graphs within feasible time bounds. To assess the quality of heuristic algorithms for the BSP and in combination with branch-and-bound approaches, the knowledge of tighter upper bounds on the quality of shortcut assignments would also be very valuable.

All of the heuristics developed and evaluated in this thesis are restricted to rather small graphs. Hence, we cannot use our algorithms to compute shortcut assignments for graph sizes that are of interest in the context of speed-up techniques. Thus, another future task is to develop heuristics that are able to deal with very large networks. Based on the results of this thesis, it would be interesting to develop an algorithm that stochastically evaluates our pair centralities, as these proved to be very valuable to estimate the quality of shortcuts. Moreover, an algorithm to dynamically update pair centralities under the insertion and deletion of shortcuts may also be future work. In combination with the stochastic approach to estimate the measure function proposed in [BDDW09], this could be used by local search algorithms for larger graphs. Another approach would be to evaluate further properties of the assignments found by our heuristics for small graphs. These properties could be used to develop heuristics to find similar assignments with less time complexity.

Finally, concerning local search strategies, there are a lot of promising approaches that could be adapted and evaluated. For example, a very recent local search strategy that seems to be promising in the context of the BSP is *Dialectic Search* as introduced by Kadioglu and Sellmann in [KS09].

A Additional Material

Algorithm 14: BestShortcutUpperBounds1

```
Input: Strongly connected graph $G = (V, E, len)$, Centrality measure crit
Output: argmax$\{w(s) \mid s \in V \times V\}$
1  Compute node centralities according to crit
2  DecreaseWithBestSolution := 0
3  $s := 0$
4  $L := (v \in V)$
5  Sort $L$ according to decreasing centrality of $v$
6  for $i \in [0, \ldots, 2n-2]$ do
7      forall $(a, b) \in \{(a, b) \in V \times V \mid index(L, a) + index(L, b) = i\}$ do
8          Build $D_a$ and determine $h(a, b)$ using HopAndPathCountingDijkstra
9          Determine $|P^+(a, b)|$ using DepthFirstSearch in $D_a$
10         Build $\overline{D_b}$ using HopAndPathCountingDijkstra in $\bar{G}$
11         Determine $|P^-(a, b)|$ using DepthFirstSearch in $\overline{D_b}$
12         upperbound := $|P^+(a, b)| \cdot |P^-(a, b)| \cdot (h(a, b) - 1)$
13         if $upperbound > DecreaseWithBestSolution$ then
14             Determine $w(a, b)$ solving APSP
15             if $w(a, b) > DecreaseWithBestSolution$ then
16                 DecreaseWithBestSolution := $w(a, b)$
17                 $s := (a, b)$
18
19
20
21 return $s$
```

Algorithm 15: BestShortcutUpperBounds2

```
Input: Strongly connected graph $G = (V, E, len)$, Centrality measure crit
Output: argmax$\{w(s) \mid s \in V \times V\}$
1  Compute node centralities according to crit
2  forall $v \in V$ do
3      $f(v) := 0$
4  forall $a \in V$ do
5      Build $D_a$ and determine $h(a, b)$ using HopAndPathCountingDijkstra in $G$
6      Determine upper bounds $f_{\text{sum,up}}(b)$ using SumValuesSuccessorsUpperBound in
       $D_a$
7      Set $\overline{|P^+(a, b)|} := f_{\text{sum,up}}(b)$
8  forall $b \in V$ do
9      Build $\overline{D_b}$ using HopAndPathCountingDijkstra in $\bar{G}$
10     Determine upper bounds $f_{\text{sum,up}}(a)$ using SumValuesSuccessorsUpperBound in
       $\overline{D_b}$
11     Set $\overline{|P^-(a, b)|} := f_{\text{sum,up}}(a)$
12 DecreaseWithBestSolution := 0
13 $s := 0$
14 $L := (v \in V)$
15 Sort $L$ according to decreasing centrality of $v$
16 for $i \in [0, \ldots, 2n - 2]$ do
17     forall $(a, b) \in \{(a, b) \in V \times V \mid index(L, a) + index(L, b) = i\}$ do
18         upperbound := $\overline{|P^+(a, b)|} \cdot \overline{|P^-(a, b)|} \cdot (h(a, b) - 1)$
19         if $upperbound > DecreaseWithBestSolution$ then
20             Determine $w(a, b)$ solving APSP
21             if $w(a, b) > DecreaseWithBestSolution$ then
22                 DecreaseWithBestSolution := $w(a, b)$
23                 $s := (a, b)$
24
25
26
27 return $s$
```

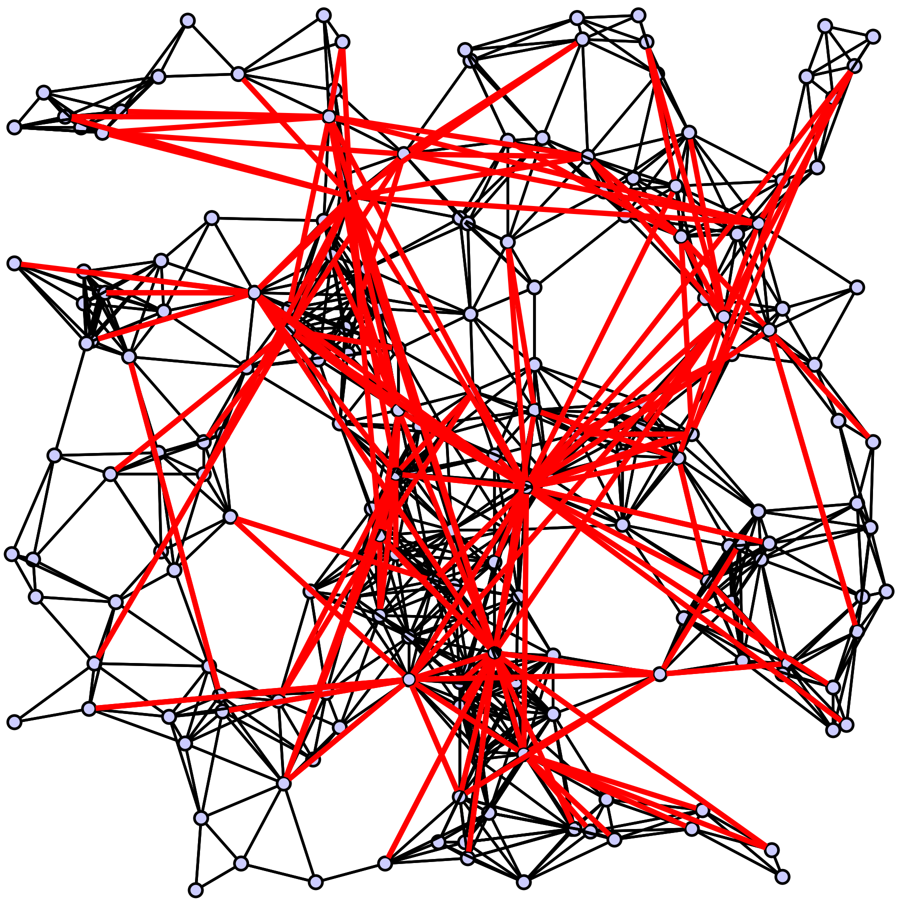


Fig. A.1: Best assignment found with our local search algorithms for unit disk graph with 198 nodes, average degree 10 and 200 shortcuts (`uDisk10_198`). This solution was found by a run of Combine-Search after appr. 89 minutes

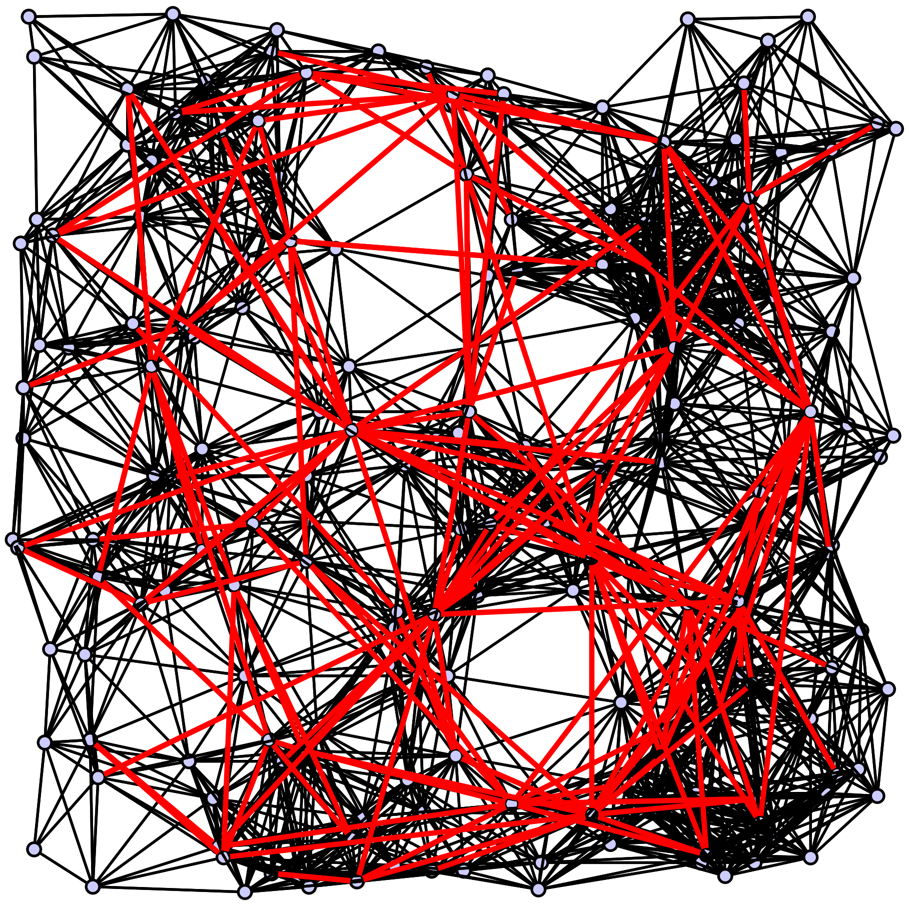


Fig. A.2: Best assignment found with our local search algorithms for unit disk graph with 200 nodes, average degree 20 and 200 shortcuts (`uDisk20_200`). This solution was found by a run of Combine-Search after appr. 90 minutes

Bibliography

[Ant71] Jacob M. Anthonisse. The rush in a directed graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, 2e Boerhaavestraat 49 Amsterdam, Oct 1971.

[AO96] Ravindra K. Ahuja and James B. Orlin. Use of representative operation counts in computational testing of algorithms. *Informs Journal of Computing*, 8(3):318–330, 1996.

[BDDW09] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, and Dorothea Wagner. The Shortcut Problem – Complexity and Approximation. In *Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, volume 5404 of *Lecture Notes in Computer Science*, pages 105–116. Springer, January 2009.

[BDW08] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Impact of Shortcuts on Speedup Techniques. Technical Report 2008-10, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2008.

[BE05] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations.* Lecture Notes in Computer Science. Springer-Verlag, 2005.

[Boe96] Kenneth Dean Boese. *Models for iterative global optimization.* PhD thesis, Los Angeles, CA, USA, 1996.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[Bra08] Ulrik Brandes. On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. *Social Networks*, 30(2):136–145, 2008.

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

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

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

[Fre77] Linton Clarke Freeman. A Set of Measures of Centrality Based Upon Betweeness. *Sociometry*, 40:35–41, 1977.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

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

[HS05] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search – Foundations and Applications*. Elsevier, 2005.

[JF95] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 184–192, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[KS09] Serdar Kadioglu and Meinolf Sellmann. Dialectic search. September 2009. To appear in the proceedings of the 15th Inter. Conference on the Principles and Practice of Constraint Programming (CP-2009).

[R D08] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[Sch03] Alexander Schrijver. *Combinatorial Optimization*. Springer-Verlag, 2003.

[WW07] Dorothea Wagner and Thomas Willhalm. Speed-Up Techniques for Shortest-Path Computations. In *Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS'07)*, volume 4393 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2007.