Fachbereich Informatik

Masterthesis

Kürzeste Wege in dynamischen Graphen

Eingereicht am:

29. Februar 2016

Eingereicht von:

Nicolas Mönch
Langbehnstraße 19b
22761 Hamburg
Tel.: (0157) 54 38 17 18
E-Mail: its101379@fh-wedel.de

Referent:

Prof. Dr. Sebastian Iwanowski
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Telefon: (041 03) 80 48-63
E-Mail: iw@fh-wedel.de

Zweitgutachter:

Prof. Dr. Gerd Beuster
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Telefon: (041 03) 80 48-38
E-Mail: gb@fh-wedel.de

Inhaltsverzeichnis

Abbildungsverzeichnis IV

Listingsverzeichnis VI

Abkürzungsverzeichnis VII

1. Einleitung 1
- 1.1. Motivation . . . 1
- 1.2. Ziel . . . 2
- 1.3. Aufbau der Arbeit . . . 2

2. Grundlagen 4
- 2.1. Graphentheorie . . . 4
  - 2.1.1. Graphen . . . 4
  - 2.1.2. Gewurzelte Bäume . . . 5
  - 2.1.3. SSSP-Problem auf statischen Graphen . . . 6
  - 2.1.4. SSSP-Problem auf dynamischen Graphen (DSP-Problem) . . . 7
  - 2.1.5. Notationen . . . 8
- 2.2. Datenstrukturen . . . 9

3. Statische Algorithmen 12
- 3.1. Algorithmus von Bellman und Ford . . . 13
- 3.2. Algorithmus von D'Esopo-Pape . . . 16
- 3.3. Algorithmus von Dijkstra . . . 17
- 3.4. Framework für statische Algorithmen . . . 19

4. Semidynamische Algorithmen 27
- 4.1. Kategorisierung von Gewichtsänderungen . . . 28
  - 4.1.1. Erhöhung des Gewichts von Kanten im Baum . . . 28

4.1.2. Verminderung des Gewichts von Kanten im Baum 29
4.1.3. Erhöhung des Gewichts von Kanten außerhalb des Baums 30
4.1.4. Verminderung des Gewichts von Kanten außerhalb des Baums . . 30
4.2. Anpassung des Frameworks für dynamische Graphen 31
4.2.1. Erste Stufe der Anpassung 32
4.2.2. Zweite Stufe der Anpassung 38
4.3. Ball-and-String Algorithmus 45

5. Volldynamische Algorithmen **56**
5.1. Herleitung 56
5.2. Dynamic SWSF-FP-Algorithmus 60

6. Prototypische Implementierung **66**
6.1. Entwicklungskonfiguration 67
6.2. Implementierung der Komponenten 68
6.3. Anwendungsdemonstration 73

7. Experimente **77**
7.1. Zufällig generierte Graphen 77
7.1.1. Experimente anderer Autoren 78
7.1.2. Eigene Experimente 80
7.1.3. Vergleich der Ergebnisse 88
7.2. Straßennetzabbildende Graphen 89

8. Abschlussbetrachtung **91**
8.1. Zusammenfassung 91
8.2. Fazit 93
8.3. Ausblick 93

Literaturverzeichnis **95**

A. Inhalt der beiliegenden CD **97**

Eidesstattliche Erklärung **98**

Abbildungsverzeichnis

3.1. Beispielgraph G (Bellman-Ford) 14
3.2. Beispielgraph G (Bellman-Ford) mit SPT 15
3.3. Beispielgraph G (Dijkstra) 18
3.4. Beispielgraph G (Dijkstra) mit SPT 19
3.5. Beispielgraph G (Framework) 22
3.6. Beispielgraph G (Framework) mit SPT 26

4.1. Beispielgraph mit SPT - Erhöhung im Baum 28
4.2. Beispielgraph mit SPT - Verminderung im Baum 29
4.3. Beispielgraph mit SPT - Erhöhung außerhalb des Baums 30
4.4. Beispielgraph mit SPT - Verminderung im Baum 31
4.5. Beispielgraph G (dynamische Framework - Erhöhungen) 34
4.6. Beispielgraph G (dynamische Framework - Erhöhungen, Lösung) 35
4.7. Beispielgraph G (dynamische Framework - Verminderung) 37
4.8. Beispielgraph G (dynamische Framework - Verminderung, Lösung) 38
4.9. Beispielgraph G (2. Stufe d. Anpassung - Ausgangsgraph) 41
4.10. Beispielgraph G (2. Stufe d. Anpassung - Ergebnisgraph) 44
4.11. ”Ball–and–String”-Modell 47
4.12. Beispielgraph G (Ball-and-String–Algorithmus - Ausgangsgraph) 53

5.1. Beispielgraph G (dynamischer SWSF-FP-Algorithmus) 63
5.2. Beispielgraph G (dynamischer SWSF-FP-Algorithmus - Lösung) 65

6.1. Klassendiagramm Graphenmodellierung 69
6.2. Klassendiagramm Warteschlangenmodellierung 70
6.3. Klassendiagramm Algorithmen 71
6.4. Anwendungsdemonstration: 1. Tab 74
6.5. Anwendungsdemonstration: 2. Tab 76

7.1. Experimente - Erhöhung: variable Graphengröße 81
7.2. Experimente - Erhöhung: variable Kantenanzahl 82
7.3. Experimente - Erhöhung: variable Anzahl der Änderungen 83
7.4. Experimente - Verminderung: variable Graphengröße 84
7.5. Experimente - Verminderung: variable Kantenanzahl 84
7.6. Experimente - Verminderung: variable Anzahl der Änderungen 85
7.7. Experimente - Gemischte Änderungen: variable Graphengröße 86
7.8. Experimente - Gemischte Änderungen: variable Kantenanzahl 87
7.9. Experimente - Gemischte Änderungen: variable Anzahl der Änderungen . 87

Listingsverzeichnis

3.1. Algorithmus von Bellman und Ford 13
3.2. Algorithmus von Dijkstra 17
3.3. Algorithmenframework Narváez 21

4.1. Initialisierungsroutine des statischen Frameworks 32
4.2. Anpassung Initialisierungsroutine - Erhoehung 33
4.3. Anpassung Initialisierungsroutine - Verminderung 36
4.4. Hauptroutine 39
4.5. Anpassung Nachfolgerfunktion 40
4.6. Ball-and-String-Algorithmus: Initialisierungsroutine - Erhöhung 50
4.7. Ball-and-String-Algorithmus: Initialisierungsroutine - Verminderung . . . 51
4.8. Ball-and-String-Algorithmus: Hauptphase 52

5.1. Dynamischer SWSF-FP-Algorithmus 61

Abkürzungsverzeichnis

SSSP Single Source Shortest Path

SPT Shortest Path Tree

DSP Dynamic Shortest Path

FIFO First In First Out

SWSF Strict Weakly Superior Function

SWSF-FP Strict Weakly Superior Function – Fixed Point

1 Einleitung

1.1. Motivation

Angewendet auf verschiedene Szenarien verfolgen Navigationssysteme und Router in Netzwerken das gleiche Ziel. Sie arbeiten auf einem Graphen, der die zugrunde liegende Umwelt abstrahiert, etwa das Verkehrsnetz im Falle der Navigationssysteme oder ein Netzwerk bestehend aus Routern und angeschlossenen Systeme, welche miteinander verbunden sind. Ihre Aufgabe ist es, ausgehend von einem spezifizierten Startpunkt, einen Endpunkt auf dem optimalen Pfad zu erreichen und so Verkehrsteilnehmer oder Datenpakete auf dem kürzesten Pfad durch den Graphen zu leiten. Navigationssysteme und Router sind nicht die einzigen Anwendungen für dieses Problem. Es können viele weitere insbesondere im Bereich der Rechnernetze gefunden werden, wie beispielsweise Sensoren- und mobile Ad-hoc-Netze, in denen sich die Netztopologie häufig durch Positionsänderungen der Teilnehmer ändert.

Die Bestimmung des optimalen Pfad wird in der Graphentheorie als Problem der kürzesten Pfade (Single Source Shortest Path (SSSP)-Problem) bezeichnet. Bekannteste Vertreter zur Lösung des Problems sind die Algorithmen von Dijkstra, und Bellman und Ford. Diese

Algorithmen haben gemeinsam, dass sie auf statischen, sich nicht ändernden, gewichteten Graphen arbeiten. Die erwähnten Anwendungsszenarien hingegen unterliegen häufigen, dynamischen Änderungen. Im Falle der Navigationssysteme kann sich die Verkehrsdichte auf den Straßen in kürzestem Zeitraum ändern. Ein Unfall auf einer stark frequentierten Straße führt umgehend zu einem Stau, welcher die Fahrzeit um ein Vielfaches erhöhen kann. Aber auch weniger drastische Ereignisse, wie zum Beispiel die höhere Verkehrsdichte zu bestimmten Uhrzeiten, können die Verwendung von Ausweichrouten lukrativer werden lassen. Im Anwendungsfall der Router gilt ähnliches. Es gibt Tageszeiten, zu denen Leitungen stärker belastet werden. Systeme können sekündlich ausfallen oder dem Netz neu hinzugefügt werden. Dadurch ändert sich die Topologie des Netzes und des abstrahierten Graphen stark. Dies kann zuvor berechnete Ergebnisse in kürzester Zeit beeinträchtigen oder gar falsch werden lassen.

Aus diesem Grund muss hinterfragt werden, wie auf dynamische Änderungen der Graphen effizient zu reagieren ist. Ein naheliegender Ansatz besteht darin, die Berechnungen mit statischen Algorithmen von Grund auf neu durchzuführen. Die Motivation dieser Arbeit liegt darin, durch die Verwendung voriger Berechnungsergebnisse schneller aktualisierte kürzeste Pfade zu bestimmen.

1.2. Ziel

Das Ziel dieser Arbeit ist es, einen Überblick über vorhandene Algorithmen zu schaffen, welche auf dynamische Änderungen der Graphen reagieren und unter Zuhilfenahme der zuvor berechneten Ergebnisse aktuell gültige optimale Pfade bestimmen. Die Algorithmen werden in ihrer Funktionsweise analysiert und untereinander, sowie mit statischen Algorithmen für das Problem verglichen. Sämtliche vorgestellte Verfahren werden experimentell in unterschiedlichen Anwendungsszenarien getestet und daraus abgeleitet Aussagen über ihre Anwendbarkeit getroffen.

1.3. Aufbau der Arbeit

Im 2. Kapitel werden zunächst graphentheoretische Grundlagen, sowie Notationen und Datenstrukturen beschrieben. Mit Hilfe dieser kann das SSSP-Problem auf dynamischen Graphen formal und somit exakter beschrieben werden. Kapitel 3 schafft einen Über-

blick über Algorithmen für die statische Variante des Problems. Die darauf folgenden Kapitel 4 und 5 beschäftigen sich mit Algorithmen, welche speziell für die dynamische Variante des Problems ausgelegt wurden. Dabei handelt es sich zum Teil um angepasste statische, als auch neu entwickelte dynamische Algorithmen. Das 6. Kapitel stellt die im Rahmen dieser Arbeit entstandene prototypische Implementierung vor, mithilfe derer Experimente durchgeführt wurden. Kapitel 7 befasst sich mit der Auswertung eigener und fremder Experimente. Abschließend werden die Ergebnisse der Arbeit in Kapitel 8 zusammengefasst, gefolgt von einem Fazit und einem Ausblick.

2

Grundlagen

Dieser Abschnitt enthält Grundlagen, welche zum Verständnis der formalen Aufgabenstellung und der in Kapitel 3, 4 und 5 vorgestellten Algorithmen notwendig sind. Er ist untergliedert in einen Abschnitt zum Thema Graphentheorie, in dem spezielle Arten und Formen von Graphen, sowie auf ihnen aufbauende Probleme erläutert und Notationen eingeführt werden, sowie einen Abschnitt, in dem verwendete Datenstrukturen beschrieben werden.

2.1. Graphentheorie

2.1.1. Graphen

Graphen sind Strukturen aus Ecken (Vertices) und Kanten (Edges). Es wird zwischen ungerichteten und gerichteten Graphen unterschieden.

Ungerichtete Graphen

Ein ungerichteter Graph G besteht aus einem Paar (V, E). V ist eine Menge von Knoten und E eine Menge von Kanten. Eine Kante $e = \{v_1, v_2\}$ mit $v_1, v_2 \in V$ repräsentiert eine ungerichtete Verbindung zwischen zwei Endknoten v_1 und v_2. Ungerichtete Kanten können sowohl von v_1 nach v_2 als auch in entgegengesetzter Richtung traversiert werden (vgl. [IW14, Kapitel 7.1]).

Gerichtete Graphen

Ein gerichteter Graph oder Digraph (*directed Graph*) G besteht aus einer Struktur (V, E) mit einer Knotenmenge V und einer Kantenmenge E. Der Unterschied zu einem ungerichteten Graphen besteht darin, dass eine Kante e der Menge E ein geordnetes Paar der Form (v_1, v_2) ist. Die Reihenfolge der Endknoten ist wichtig, eine gerichtete Kante $e = (v_1, v_2)$ kann nur von v_1 nach v_2 traversiert werden. v_1 wird dabei als Anfangsknoten oder tail bezeichnet, v_2 als Endknoten oder head (vgl. [IW14, Kapitel 7.1]).

Gewichtete Graphen

Ist ein Graph (gerichtet oder ungerichtet) kantengewichtet, so existiert zusätzlich eine Gewichtsfunktion

$$w : E \to \mathbb{N} \text{ (oder } \mathbb{R}, \mathbb{R}^{\geq 0}) \tag{2.1}$$

Sie weist jeder Kante aus E ein Gewicht zu. Ein Graph kann auch knotengewichtet sein. Knotengewichtete Graphen sind im Rahmen dieser Arbeit nicht relevant. Im Folgenden erwähnte gewichtete Graphen sind immer kantengewichtet (vgl. [IW14, Kapitel 7.2.2]).

2.1.2. Gewurzelte Bäume

Gewurzelte Bäume sind eine spezielle Form von Graphen. Ein Graph ist ein gewurzelter Baum mit einer Wurzel, wenn er die folgenden zwei Eigenschaften erfüllt (vgl. [IW14, Kapitel 7.3]):

- zusammenhängend - Jeder Knoten des Graphen kann von der Wurzel aus über eine Folge von gerichteten Kanten erreicht werden.

- kreislos - Kein Knoten kann sich selbst über eine Folge von gerichteten Kanten erreichen.

Gewurzelte Bäume sind im Rahmen dieser Arbeit von besonderer Bedeutung. Sie werden die berechneten Informationen über kürzeste Pfade enthalten. Ein solcher Baum wird im Folgenden als kürzeste-Wege-Baum oder Shortest Path Tree (SPT) bezeichnet.

2.1.3. SSSP-Problem auf statischen Graphen

Das Problem des kürzesten Pfad oder shortest path problem beschäftigt sich damit, kürzeste Pfade in Graphen zu ermitteln und ist folgendermaßen definiert:
Gegeben ist ein gewichteter Digraph mit der Gewichtsfunktion

$$G = (V, E),\ w : E \to \mathbb{R} \tag{2.2}$$

Das Gewicht oder die Länge eines Pfads $p = v_0, v_1, .., v_k$ ist gleich der Summe der Gewichte der enthaltenen Kanten:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{2.3}$$

Die Länge des kürzesten Pfads zwischen zwei Knoten u, v ist folgendermaßen definiert:

$$\delta(u, v) = \begin{cases} min\{w(p : u \overset{p}{\rightsquigarrow} v)\} & \text{falls ein Pfad von } u \text{ nach } v \text{ existiert} \\ \infty & \text{sonst} \end{cases} \tag{2.4}$$

Ein kürzester Pfad zwischen den Knoten u, v ist definiert als ein beliebiger Pfad p mit

$$w(p) = \delta(u, v) \tag{2.5}$$

Es können mehrere kürzeste Pfade zwischen zwei Knoten existieren, die Länge eines kürzesten Pfads hingegen ist eindeutig [CH05].
Wird dieses Problem auf die Ermittlung aller kürzesten Wege bezogen auf einen bestimmten Startpunkt reduziert, so wird es als SSSP-Problem bezeichnet. Zur Speicherung aller kürzesten Pfade ausgehend von einem Startpunkt in einem Graphen G kann ein SPT verwendet werden (vgl. [CO01, Kapitel 24]).

2.1.4. SSSP-Problem auf dynamischen Graphen (DSP-Problem)

Das dynamische SSSP-Problem (DSP-Problem) besteht darin, kürzeste Pfade auf Graphen zu berechnen, welche sich zu bestimmten, diskreten Zeitpunkten ändern können. Diese Änderungen betreffen die Gewichtungen der Kanten des Graphen. Eingeschlossen sind das Einfügen und Löschen von Kanten, indem die Längen der Kanten von ∞ auf einen konkreten Wert dekrementiert, bzw. von einem konkreten Wert auf den Wert ∞ inkrementiert werden.

Im Vergleich zum SSSP-Problem kann bei dieser Art von Problem auf vorherige Berechnungen und Ergebnisse zurückgegriffen werden. Algorithmen, welche das Problem lösen, ohne eine komplette Neuberechnung auf dem veränderten Graphen auszuführen, werden *dynamic single-source shortest-path - Algorithmen* genannt [RB09][JG12].

Es ist folgendermaßen definiert:
Gegeben ist ein gewichteter Digraph mit einer Gewichtsfunktion

$$G = (V, E),\ w : E \to \mathbb{R}$$

Des Weiteren sind der alte SPT T und eine Menge von Kantenänderungen bekannt:

$$\epsilon = \{\langle e_i, \tau \rangle \mid e_i \in E \wedge \tau_i \in \mathbb{R}\} \tag{2.6}$$

Der Wert einer Kantenänderung τ_i der Kante e_i kann negativ und auch größer als das ursprüngliche Gewicht der Kante sein. Dadurch ist es möglich, dass eine Kante, welche innerhalb des alten Graphen eine positives Gewicht besaß, nach Berücksichtigung der Änderung negativ gewichtet ist.

Mit Hilfe des alten Graphen G und der Menge der Kantenänderungen ϵ ersteht der neue Graph G' mit der Gewichtsfunktion w':

$$w'(e_i) = \begin{cases} w(e_i) + \tau_i & \forall \langle e_i, \tau \rangle \in \epsilon \\ w(e_i) & \forall \langle e_i, \tau \rangle \notin \epsilon \end{cases} \tag{2.7}$$

In dieser Arbeit werden ausschließlich Graphen betrachtet, welche über nichtnegative Gewichtsfunktionen verfügen.

$$w : E \to \mathbb{R}^{\geq 0}$$

Daraus ergibt sich für Kantenänderungen folgender Wertebereich:

$$\epsilon = \{\langle e_i, \tau \rangle \mid e_i \in E \wedge -w(e_i) \leq \tau_i < \infty\} \tag{2.8}$$

Die Wahl des Wertebereichs stellt somit explizit sicher, dass das Gewicht einer Kante nach Durchführung der Kantenänderungen nicht negativ ist.

Die berechnete Lösung eines DSP, die sämtliche kürzeste Pfade zwischen dem Start- und allen anderen Knoten enthält, kann in Form eines SPT gespeichert werden.

2.1.5. Notationen

Im Folgenden werden Notationen beschrieben, welche zur Beschreibung der graphentheoretischen Zusammenhänge verwendet werden.

- $V(G)$ sei die Menge der Knoten in G.
- $E(G)$ sei die Menge der Kanten in G.
- Sei $e = (u, v) \in E$ eine Kante:
  - $t(e)$ sei der Anfangsknoten (*tail*) der Kante,
  - $h(e)$ sei der Endknoten (*head*) der Kante.
- Sei $u \in V$, so ist
  - $Out_u = \{e \mid e \in E \wedge t(e) = u\}$ die Menge der von u ausgehenden Kanten,
  - $In_u = \{e \mid e \in E \wedge h(e) = u\}$ die Menge der in u eingehenden Kanten,
  - $c(u) = \{v \mid v = h(e) \wedge e \in Out_u\}$ die Menge der Kinderknoten,
  - $p(u) = \{v \mid v = t(e) \wedge e \in In_u\}$ die Menge der Elternknoten.
- p_{uv} sei ein Pfad, eine Folge von Kanten, von u nach v in G.
- v ist von u *erreichbar*, wenn ein Pfad p_{uv} existiert.
- $des(G, u)$ bzw. $des(u)$ enthält alle von u erreichbaren Knoten (inklusive u). Sie werden als *Nachfolger* von u bezeichnet.
- p_{uv} wird als kürzester Pfad SP_{uv} bezeichnet genau dann, wenn kein kürzerer Pfad p^*_{uv} existiert.

- d_{uv} sei die kürzeste Distanz von Knoten u nach v.
- T_s beschreibt den SPT mit Wurzelknoten s in G in dem gilt: $\forall v \in des(s) : T_s$ enthält einen SP_{sv}.
- $spp_{T_s}(v)$ (*shortest path parent*) sei der Elternknoten von v in T_s.
- $spc_{T_s}(v)$ (*shortest path children*) sei die Menge der Kinderknoten von v in T_s.

2.2. Datenstrukturen

Im Folgenden Datenstrukturen beschrieben, welche zur Beschreibung der Algorithmen verwendet werden.

Graphen

Graphen werden im Kontext der Beschreibung der Algorithmen folgendermaßen definiert:

- ein Graph G besteht aus einer Menge von Knoten V und einer Menge von Kanten E.
- ein Knoten v besitzt einen Schlüssel id, sowie eine Menge eingehender Kanten $In(v)$ und ausgehender Kanten $Out(v)$.
- eine Kante e besitzt einen Schlüssel id.
- eine Funktion $w : E \to \mathbb{R}^{\geq 0}$ weist jeder Kante e einen Wert $w(e)$ zu.

SPT

Ein Shortest Path Tree T_s wird durch eine Menge zusätzlicher Knoteninformationen repräsentiert, welche über die Knotenschlüssel identifiziert werden. Die zusätzlichen Informationen aux enthalten folgende Einträge

- $spp(v)$
- $spc(v)$

- d_v

Warteschlange

Sämtliche Algorithmen arbeiten mit Warteschlangen. Einträge der Warteschlangen haben die Form $\langle v, data, key \rangle$ mit

- einem Knoten v.
- Hilfsdaten $data$, welche vom Algorithmus abhängig und optional sind.
- einem Schlüssel key, nachdem Einträge in der Warteschlange optional sortiert werden können. Als Werte sind auch geordnete Paare $\langle value2, value1 \rangle$ zulässig. Ausschlaggebend für deren Ordnung ist zunächst der Wert von $value2$, danach der Wert von $value1$. Gleiche Werte sind zulässig, deren Ordnung ist beliebig.

Warteschlangen werden über folgende Operationen verwaltet:

- $ENQUEUE(q, \langle v, data, key \rangle)$ fügt einer Warteschlange q einen Eintrag hinzu. Verfügt q bereits über einen Eintrag mit Knoten v, so wird dieser genau dann ersetzt, wenn der Schlüssel key des neuen Eintrags kleiner ist.
- $EXTRACTMIN(q)$ extrahiert den kleinsten Eintrag aus q.
- $REMOVE(q, v)$ entfernt Eintrag mit Knoten v aus q.
- $CONTAINS(q, v)$ signalisiert, ob ein Eintrag mit Knoten v in q enthalten ist.
- $PEEK(q, v)$ gibt den Eintrag für Knoten v aus q zurück, entfernt ihn jedoch nicht.

Die Implementierung von Warteschlange kann über unterschiedliche Datenstrukturen mit unterschiedlichen Operationslaufzeiten erfolgen. Im Rahmen dieser Arbeit werden Implementierungen durch lineare (einfach bzw. doppelt verkettet) und sortierte Listen, sowie binäre und Fibonacci-Halden betrachtet. Die Laufzeiten der vorgestellten Algorithmen sind auch von der gewählten Implementierungsvariante der Warteschlange abhängig. Tabelle 2.1 zeigt die Laufzeiten der Operationen in Abhängigkeit der Implementierung und der Anzahl der Listenelemente n. Bei der Verwendung von Halden wird vorausgesetzt, dass Knoten und Einträge Referenzen aufeinander besitzen. Das kleinste Element einer FIFO-Liste ist das erste Elemente an der ersten Position.

Operation	lineare Liste (FIFO)	sortierte Liste	binäre Halde	Fibonacci-Halde
$ENQUEUE$	$\mathcal{O}(n)$ bzw. $\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(1)$
$EXTRACTMIN$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
$REMOVE$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
$CONTAINS$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$PEEK$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Tabelle 2.1.: Operationslaufzeiten

Ist es im Folgenden eindeutig, welche Warteschlange q verwendet wird, so kann der Parameter der Warteschlange in Operationsaufrufen entfallen.

3

Statische Algorithmen

In diesem Kapitel werden grundlegende statische Algorithmen präsentiert, welche das Single Source Shortest Path (SSSP) für einen definierten Startpunkt auf statischen Graphen lösen. Im Detail werden die Algorithmen von Bellman und Ford, D'Esopo-Pape und Dijkstra vorgestellt und ihre Vorgehensweise an Beispielen verdeutlicht.

Der A*-Algorithmus wird im Rahmen dieser Arbeit nicht näher betrachtet, da dieser mit einer erweiterten Heuristik arbeitet. Diese muss zu jedem Zeitpunkt der Berechnung einige Konsistenzbedingungen erfüllen, welche aus folgendem Grund nur schwer mit dem DSP-Problem vereinbar sind. Eine der Konsistenzbedingungen der Heuristik besagt, dass Distanzen niemals überschätzt werden dürfen. Das DSP-Problem erlaubt jedoch, dass jedes Kantengewicht auf den Wert 0 dekrementiert wird. Die einzige Heuristik, die ohne weitere Anpassungen immer zulässig wäre, ist somit jene, welche für jeden Knoten den Wert 0 schätzt.

Im letzten Abschnitt wird ein Framework für die eingeführten, statischen Algorithmen vorgestellt, welches zur Implementierung der einzelnen Algorithmen nur geringfügig angepasst werden muss.

3.1. Algorithmus von Bellman und Ford

Der Algorithmus von Bellman und Ford löst das SSSP-Problem im generellen Fall. Das bedeutet, dass die Gewichte der Kanten auch negativ sein dürfen. Gegeben sind ein Graph G, ein Startknoten $s \in V(G)$, sowie eine Gewichtsfunktion $w : E(G) \to \mathbb{R}$. Der Algorithmus berechnet die kürzesten Distanzen zu allen anderen Knoten des Graphen G und hat einen booleschen Rückgabewert, der beschreibt, ob der Graph einen negativen Zyklus enthält. Ist dies der Fall, so kann es keine Lösung auf diesem Graphen geben und der Algorithmus terminiert mit einem negativen Rückgabewert (vgl. [CO01, Kapitel 24.1]).

```
1  Boolean BELLMAN-FORD (G, s, w)
2      // Initialisierung Distanzarray d, Elternarray p
3      for (Vertex v : V)
4          d[v] = INF;
5          p[i] = null;
6      d[s] = 0;
7      // Hauptschleife
8      for (i = 1; i<|V|; i++)
9          for (Edge e : E)
10             if (d[v] > d[u] + w(u,v))
11                 d[v] = d[u] + w(u,v);
12                 p[v] = u;
13     // Ueberpruefung negative Zyklen
14     for (Edge e : E)
15         if (d[v] > d[u] + w(u,v))
16             return FALSE;
17     return TRUE;
```

Listing 3.1: Algorithmus von Bellman und Ford

Der Algorithmus initialisiert die Distanzen aller Knoten $v \in V(G)$ mit $d_v = \infty$ mit Ausnahme der des Startknotens s ($d_s = 0$). Die Hauptschleife wird $(n-1)$-mal ausgeführt, wobei n die Anzahl der Knoten ($|V|$) sei.

In jedem Schritt wird für jede Kante $e = (u, v) \in E(G)$ untersucht, ob eine Distanzverbesserung für den Knoten v durch Erfüllung der Gleichung $d[v] > d[u] + w(u, v)$ gegeben ist. Ist dies der Fall, so wird die Distanz $d[v] = d[u] + w(u, v)$ und u als Vorgänger von v ($spp(v) = u$) gespeichert. Dieser Vorgang wird als Kantenentspannung (*relax*) bezeichnet. Auf diese Weise wird in einer Runde i der kürzeste Pfad zwischen s und jedem anderen

Knoten bestimmt, welcher maximal i Kanten enthält. Im schlechtesten Fall kann ein kürzester Pfad (ohne negative Zyklen) $n-1$ Kanten enthalten, sodass dieser nach ebenso vielen Iterationen berechnet ist.

Abschließend untersucht der Algorithmus, ob der Graph negative Zyklen enthält. Dies ist genau dann der Fall, wenn für zwei Knoten u, v nach $n-1$ Iterationen der Hauptschleife gilt, dass $d[v] > d[u] + w(u, v)$ und eine Verbesserung durch Hinzunahme weiterer Kanten möglich ist. In diesem Fall terminiert der Algorithmus mit negativem Rückgabewert.

Beispiel

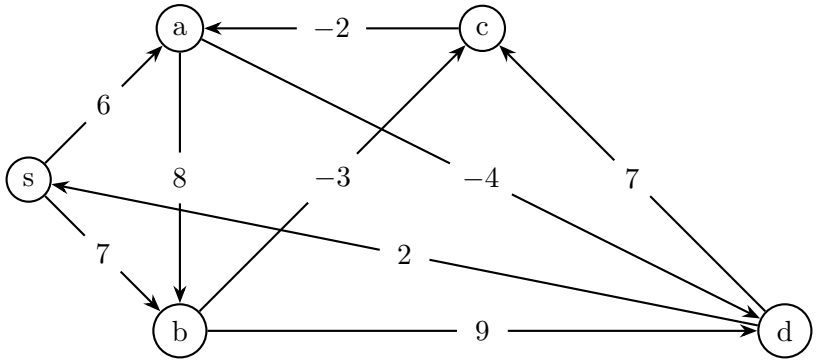


Abbildung 3.1.: Beispielgraph G (Bellman-Ford)

Iteration	erfolgreiche Kantenentspannungen	d_v
1	$(s, a), (s, b)$	$d_a = 6, d_b = 7$
2	$(a, d), (b, c)$	$d_d = 2, d_c = 4$
3	(c, a)	$d_a = 2$
4	(a, d)	$d_d = -2$

Tabelle 3.1.: Ablauf Bellman-Ford auf einem Beispielgraph

Tabelle 3.1 verdeutlicht den Ablauf des Algorithmus. In jeder Iteration wird für jede Kante und deren Bewertung überprüft, ob eine Distanzverbesserung erreicht werden kann. In der ersten Iteration sind die berechneten Distanzwerte für die Knoten $a = 6$ und $b = 7$ kleiner als ihr Initialwert ∞. Gleiches gilt für die Knoten c und d in der zweiten Iteration. In der dritten Iteration trifft der Algorithmus für Knoten a eine weitere Verbesserung an,

da bei der Überprüfung der Kante (c, a) ein Distanzwert $4 - 2 = 2$ (< 6) berechnet wird. Analog geschieht dies für Kante (a, d) in Iteration 4. Nach Ablauf der Hauptschleife wird für jede Kante erneut überprüft, ob eine erfolgreiche Kantenentspannung möglich ist. Da dies nicht der Fall ist, terminiert der Algorithmus mit dem Rückgabewert *TRUE* und die Ergebnisse befinden sich in d und p. Der resultierende SPT ist in Abbildung 3.2 kenntlich gemacht.

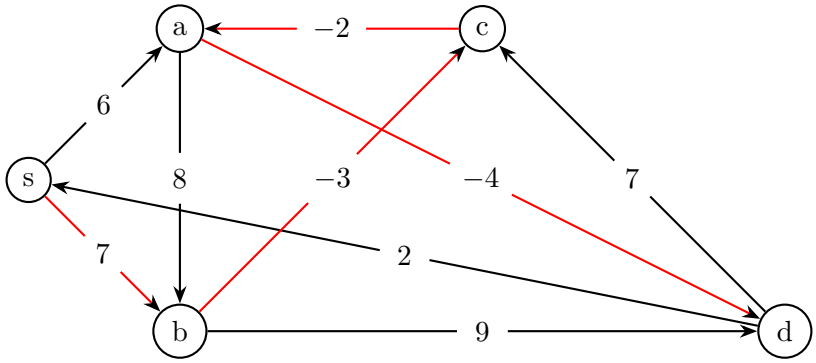


Abbildung 3.2.: Beispielgraph G (Bellman-Ford) mit SPT

Der Korrektheitsbeweis des Algorithmus kann [CO01, Kapitel 24.1] entnommen werden. Gegeben sei ein Graph $G = (V, E)$ mit $n = |V|$ Knoten und $m = |E|$ Kanten. Die Laufzeit des Algorithmus von beträgt $\mathcal{O}(n * m)$, denn

- die Laufzeit der Initialisierungsphase beträgt $\theta(n)$, da die Schleife n-mal ausgeführt wird und die Laufzeit des Schleifenrumpf konstant ist,
- die Laufzeit der Hauptphase beträgt $\mathcal{O}(n * m)$, da die äußere Schleife über n-1 Knoten und die innere Schleife über m Kanten iteriert und der Schleifenrumpf der inneren Schleife konstante Laufzeit aufweist,
- die Laufzeit der Überprüfungsphase beträgt $\mathcal{O}(m)$, da die Schleife m-mal ausgeführt wird und Laufzeit des Schleifenrumpf konstant ist.

Variante mit einer Warteschlange

In diesem Abschnitt soll eine Variante des Algorithmus vorgestellt werden, welche die Hauptschleife durch eine Iteration über eine Warteschlange (*queue*) ersetzt [DB92]. Diese Variante benötigt weniger Iterationen, ist jedoch auf Problemstellungen begrenzt, bei der die Werte der Gewichtsfunktion $w : E \to \mathbb{R}^{\geq 0}$ nicht negativ sind. Die ursprüngliche

Variante von Bellman und Ford überprüft in jeder der $n-1$ Schleifeniterationen für alle Kanten, ob diese entspannt werden können. In dieser Variante wird der Warteschlange q im Initialisierungsprozess ein Eintrag mit dem Startknoten s hinzugefügt. Der Algorithmus terminiert, wenn die Warteschlange q keine Einträge enthält. Solange q Einträge enthält, wird folgendermaßen vorgegangen:

- entnehme den ersten Eintrag v aus q
- überprüfe für jede ausgehende Kante $e \in Out(v)$, ob diese entspannt werden kann. Ist dies der Fall, so entspanne e und füge der Warteschlange q einen Eintrag für Knoten $h(e)$ an letzter Position hinzu, sofern kein Eintrag für diesen vorhanden ist.

Die Warteschlange arbeitet somit nach dem First In First Out (FIFO)-Prinzip, welches besagt, dass der älteste Eintrag als erstes aus der Warteschlange entnommen wird. Entsprechend der Deklaration des Gewichtsfunktion (mit nichtnegativem Wertebereich) kann bei dieser Variante kein negativer Zyklus im Graphen G enthalten sein, sodass diese Überprüfung zu Abschluss des Algorithmus nicht getätigt werden muss.

3.2. Algorithmus von D'Esopo-Pape

Der Algorithmus von D'Esopo-Pape kann als modifizierte Variante des Algorithmus von Bellman und Ford mit Warteschlangen betrachtet werden. Das Vorgehen des Algorithmus unterscheidet sich lediglich in der Implementierung der Warteschlange q. Anstelle des FIFO-Prinzips findet beim Hinzufügen eines Eintrags eine Unterscheidung statt, ob ein Eintrag für einen Knoten v bereits zu einem früheren Zeitpunkt vorhanden war und extrahiert wurde. Ist dies der Fall, so wird der neue Eintrag entgegen dem FIFO-Prinzip an erste Stelle der Warteschlange hinzugefügt. Auf diese Weise wird versucht, Distanzverbesserungen bereits betrachteter Knoten frühestmöglich zu verarbeiten und unnötige Folgeberechnungen zu vermeiden. Dieser Sachverhalten wird in Abschnitt 3.4 anhand eines Beispiels näher erläutert.

Im schlechtesten Fall besitzt dieser Algorithmus exponentielles Laufzeitverhalten [KE81]. In Untersuchungen konnte jedoch gezeigt werden, dass der Algorithmus in Graphen mit geringer Anzahl an Kanten in der Regel schneller ist als beispielsweise der Algorithmus von Bellman und Ford [DG79].

3.3. Algorithmus von Dijkstra

Der Algorithmus von Dijkstra löst das SSSP-Problem auf einem gewichteten Graphen $G = (V, E)$, in dem die Gewichtsfunktion $w : E \to \mathbb{R}^{\geq 0}$ keine negativen Werte zuweist.

```
1  Void DIJKSTRA (G, s, w)
2      // Warteschlange q, sortiert Eintraege nach vorlaeufig
           berechneter Distanz
3      // Menge calculated, berechnete Knoten
4      // Initialisierung Distanzarray d, Elternarray p,
           Warteschlange q
5      for (Vertex v : V)
6          d[v] = INF;
7          p[i] = null;
8          q.enqueue(v, d[v]);
9      d[s] = 0;
10     q.enqueue(s, 0);
11     calculated = {};
12     // Hauptschleife
13     while (q.size != 0)
14         v1 = q.extractmin();
15         calculated += v1;
16         for (Edge e : Out(v1)) // e = (v1,v2)
17             if (d[v2] > d[v1] + w(v1,v2))
18                 d[v2] = d[v1] + w(v1,v2);
19                 p[v2] = v1;
```

Listing 3.2: Algorithmus von Dijkstra

Der Algorithmus benutzt eine Hilfsmenge (im Pseudocode *calculated*), in welcher Knoten gespeichert werden, deren kürzeste Wege endgültig berechnete sind. Zunächst werden die Distanzen aller Knoten $v \in V(G)$ mit $d_v = \infty$ mit Ausnahme des Startknotens s mit $d_s = 0$ und die Elterneinträge mit dem Wert *null* initialisiert. Außerdem wird für jeden Knoten ein Eintrag mit dessen Distanz als Schlüssel in die Warteschlange eingefügt. Die Menge *calculated* ist zunächst leer.

Solang die Warteschlange nicht leer ist, wird der Eintrag mit kleinstem Schlüssel aus dieser entfernt und der enthaltene Knoten v in die Menge *calculated* eingefügt. Die Menge

der von v ausgehenden Kanten Out_v wird bestimmt und für jede dieser Kanten eine Kantenentspannung (vgl. 3.1) durchgeführt.

Beispiel

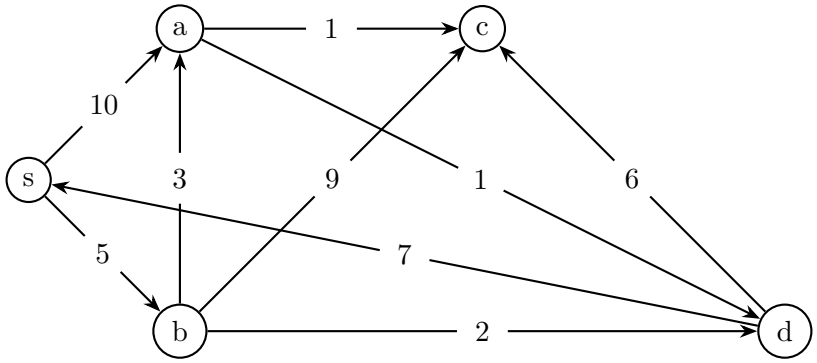


Abbildung 3.3.: Beispielgraph G (Dijkstra)

#	(v, d_v)	q	*calculated*
0	$(s,0),(a,\infty),(b,\infty),(c,\infty),(d,\infty)$	$\underline{s}, a, b, c, d$	$\{\}$
1	$(s,0),(a,10),(b,5),(c,\infty),(d,\infty)$	$\underline{b}, a, c, d$	$\{s\}$
2	$(s,0),(a,8),(b,5),(c,14),(d,7)$	$\underline{d}, a, c$	$\{s,b\}$
3	$(s,0),(a,8),(b,5),(c,14),(d,7)$	$\underline{a}, c$	$\{s,b,d\}$
4	$(s,0),(a,8),(b,5),(c,9),(d,7)$	$\underline{c}$	$\{s,b,d,a\}$
5	$(s,0),(a,8),(b,5),(c,9),(d,7)$	*empty*	$\{s,b,d,a,c\}$

Tabelle 3.2.: Ablauf Dijkstra auf einem Beispielgraph

Tabelle 3.2 verdeutlicht den Ablauf des Algorithmus. Das Resultat des Initialisierungsprozesses ist der Zeile 0 dargestellt. Die Warteschlange q enthält Einträge für alle Knoten. Der Eintrag für den Knoten s weist den kleinsten Schlüssel (0) auf, wird q im ersten Schleifendurchlauf entnommen und der Menge *calculated* hinzugefügt. Die ausgehenden Kanten (s, a) und (s, b) werden erfolgreich entspannt, die Distanz- und Elterneinträge entsprechend verändert. Die aktualisierte Warteschlange q enthält als kleinstes Element Knoten b mit der Distanz $d_b = 5$ (Zeile 1). Das beschrieben Vorgehen wird wiederholt, bis die Warteschlange keine Einträge enthält und die kürzesten Pfade zu allen Knoten

berechnet wurden. Die Ergebnisse befinden sich im Distanzvektor d, der resultierende SPT ist in Abbildung 3.4 kenntlich gemacht.

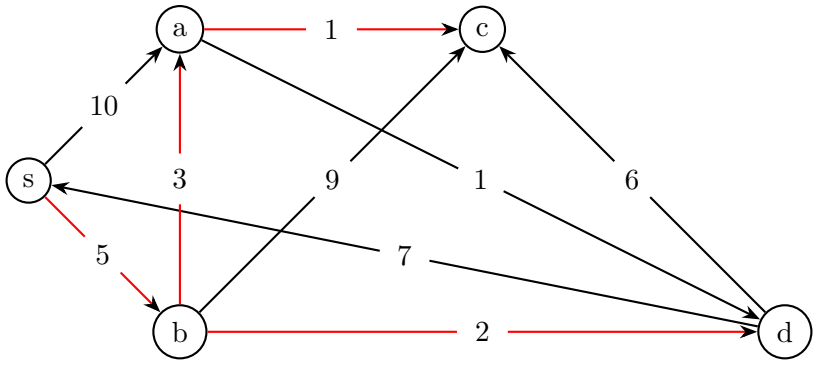


Abbildung 3.4.: Beispielgraph G (Dijkstra) mit SPT

Der Korrektheitsbeweis des Algorithmus kann [CO01, Kapitel 24.3] entnommen werden. Gegeben sei ein Graph $G = (V, E)$ mit $n = |V|$ Knoten und $m = |E|$ Kanten. Die Laufzeit des Algorithmus ist von der Implementierung der Warteschlange abhängig. Angenommen, die Warteschlange wird mit Hilfe eines Arrays implementiert, in dem linear nach dem kleinsten Element gesucht wird, so beträgt die Laufzeit des Algorithmus $\mathcal{O}(n^2)$, denn

- die Laufzeit der Initialisierungsphase beträgt $\mathcal{O}(n)$, da die Schleife n-mal ausgeführt wird und die Laufzeit des Rumpfs konstant ist,
- die Laufzeit der Hauptphase beträgt $\mathcal{O}(n^2)$, da die Warteschlange zu Beginn n Einträge enthält und pro Iteration einen Eintrag entfernt, und während jeder Iteration linear in ihr nach dem kleinsten Element gesucht werden muss ($\mathcal{O}(n)$).

Durch die Verwendung von binären Halden (binary heap, $\mathcal{O}(m * \lg n)$) kann die Laufzeit in Graphen verbessert werden, in denen die Anzahl der Kanten nicht den Grenzwert $|V|^2/\lg|V|$ überschreitet. Fibonacci Halden (Fibonacci heap, $\mathcal{O}(n * \lg n + m)$) liefern die beste theoretische Laufzeit, in der Praxis spricht jedoch der hohe Verwaltungsaufwand gegen eine Verwendung [CO01, Kapitel 24.3].

3.4. Framework für statische Algorithmen

Im diesem Abschnitt wird ein von Paolo Narváez in [NA00] beschriebener Basisalgorithmus vorgestellt, welcher als Framework für statische Algorithmen dienen soll. Die

bereits vorgestellten Algorithmen von Bellman und Ford, D'Esopo-Pape und Dijkstra können einzig durch unterschiedliche Implementierungen der verwendeten Warteschlange implementiert werden. Ein weiterer Vorteil des Algorithmus ist es, dass die Berechnung kürzester Pfade auf dynamischen Graphen und somit die Lösung des DSP-Problems mittels weniger Anpassungen möglich ist (siehe Kapitel 4.2).

Im Gegensatz zu den bereits beschriebenen Implementierungen verwendet das Algorithmenframework während der Berechnung explizit ein Baum $\hat{T}_s$, welcher bei Terminierung des Algorithmus den SPT repräsentiert. Die Eltern- und Distanzinformationen der Knoten sind in $\hat{T}_s$ enthalten. Elemente der Warteschlange q haben für einen Knoten v die Form $\{v, (p_v, d_v)\}$.

Der Algorithmus verwendet eine Funktion *des(Vertex v)* (Zeile 14), welche eine Menge von Nachfolgern von v im Baum T_s unter Optimierungsbedingungen bestimmt. Im statischen Fall ist die Funktion folgendermaßen definiert:

$$des(Vertex\ v) = \{v\}$$

Bei jedem Aufruf wird die Menge, welche lediglich das Element v enthält, $\{v\}$ ermittelt.

```
1  Void NARVAEZ (G, s, w)
2      // Initialisierung Distanzarray d, Elternarray p, enthalten in
           SPT T
3      for (Vertex v : V)
4          d[v] = INF;
5          p[v] = null;
6      q.enqueue({s, (null, 0)});
7      // Hauptschleife
8      while (q.size != 0)
9          {v, (p_new, d_new)} = q.extractmin();
10         delta = d_new - d[v];
11         if (delta < 0)
12             p[v] = p_new;
13             // Bestimmung der Nachfolger von v in T
14             descendants = des(v);
15             for (Vertex c : descendants)
16                 // Distanzupdate
17                 d[c] += delta;
18             for (Edge (u, v) : Out(descendants))
19                 newdist = d[u] + w(u,v)
20                 if (d[v] > newdist)
21                     q.enqueue({v, (u, newdist)})
```

Listing 3.3: Algorithmenframework Narváez

Das Algorithmenframework von Narvaéz wird in Listing 3.3 dargestellt.

In der Initialisierungsphase werden zunächst die Distanz- und Elternwerte der Knoten mit Initialwerten belegt und ein Eintrag für den Startknoten s mit leerem Elternwert *null* und der Distanz 0 der Warteschlange q hinzugefügt.

Die Hauptschleife des Algorithmus terminiert, wenn q keine Einträge enthält. Im Rumpf der Schleife wird zunächst ein Eintrag aus q extrahiert. Für den im Eintrag enthaltenen Knoten v wird die Differenz der im SPT gespeicherten Distanz $d[v]$ und der im Eintrag enthaltenen Distanz d_{new} gebildet und in δ gespeichert. Ist $\delta < 0$, so ist die Distanz d_{new} geringer als die ursprünglich gespeicherte Distanz, und der Schleifendurchlauf wird fortgesetzt. Der Elternknoten von v wird durch p_{new}, dem Wert des Eintrags, ersetzt. Anschließend wird die Menge der Nachfolger $descendants = \{v\}$ gebildet und für die gespeicherte kürzeste Distanz um den Wert δ reduziert. Für jede Kante e der Menge der

aus v ausgehenden Kanten Out_v wird sequentiell bestimmt, ob die Distanz $d_v+w(e)$ kleiner der für den Endknoten der Kante gespeicherten Distanz $d_h(e)$ ist, und der Endknoten somit über die ausgehende Kante über eine geringere Distanz erreichbar ist. Ist dies der Fall, so wird q ein Eintrag für $h(e)$ mit $\{h(e), (v, d_v + w(e))\}$ erstellt.

Bei näherer Betrachtung ähnelt dieser Algorithmus der vorgestellten Implementierung des Dijkstra-Algorithmus, einzig der Zeitpunkt des Hinzufügens der Warteschlangeneinträge ist verschieden. Auf diese Weise können mit Hilfe unterschiedlicher Implementierungen der Warteschlange die bereits vorgestellten Algorithmen implementiert werden:

- Bellman und Ford mithilfe einer FIFO-Warteschlange,
- D'Esopo-Pape mithilfe der modifizierten FIFO-Warteschlange, welche bereits extrahierte Einträge bei Einfüge-Operationen an erster Stelle platziert,
- Dijkstra mithilfe einer Vorrangwarteschlange (*priority queue*), sodass diese immer den Wert mit dem kleinsten Schlüssel extrahiert.

Beispiel

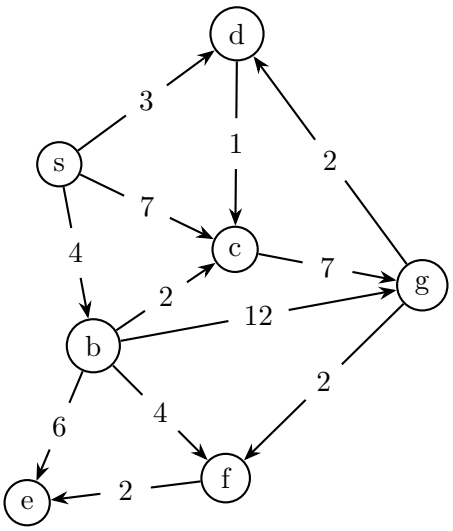


Abbildung 3.5.: Beispielgraph G (Framework)

#	(v, d_v) (wenn $d_v \neq \infty$)	q	Out(desc)
0		{s,(null, 0)}	{b,c,d}
1	(s,0)	{b,(s,4)}, {c,(s,7)}, {d,(s,3)}	{c,e,f,g}
2	(s,0), (b,4)	{c,(b,6)}, {d,(s,3)}, {e,(b,10)}, {f,(b,8)}, {g,(b,16)}	{g}
3	(s,0), (b,4), (c,6)	{d,(s,3)}, {e,(b,10)}, {f,(b,8)}, {g,(c,13)}	{c}
4	(s,0), (b,4), (c,6), (d,3)	{e,(b,10)}, {f,(b,8)}, {g,(c,13)}, {c,(d,4)}	{}
5	(s,0), (b,4), (c,6), (d,3), (e,10)	{f,(b,8)}, {g,(c,13)}, {c,(d,4)}	{e}
6	(s,0), (b,4), (c,6), (d,3), (e,10), (f,8)	{g,(c,13)}, {c,(d,4)}	{d}
7	(s,0), (b,4), (c,6), (d,3), (e,10), (f,8), (g,13)	{c,(d,4)}	{g}
8	(s,0), (b,4), (c,4), (d,3), (e,10), (f,8), (g,13)	{g,(c,11)}	{d}
9	(s,0), (b,4), (c,4), (d,3), (e,10), (f,8), (g,11)		

Tabelle 3.3.: Ablauf Framework (Bellman und Ford) auf einem Beispielgraph

#	(v, d_v) (wenn $d_v \neq \infty$)	q	Out(desc)
0		{s,(null, 0)}	{b,c,d}
1	(s,0)	{b,(s,4)}, {c,(s,7)}, {d,(s,3)}	{c,e,f,g}
2	(s,0), (b,4)	{c,(b,6)}, {d,(s,3)}, {e,(b,10)}, {f,(b,8)}, {g,(b,16)}	{g}
3	(s,0), (b,4), (c,6)	{d,(s,3)}, {e,(b,10)}, {f,(b,8)}, {g,(c,13)}	{c}
4	(s,0), (b,4), (c,6), (d,3)	{c,(d,4)}, {e,(b,10)}, {f,(b,8)}, {g,(c,13)}	{g}
5	(s,0), (b,4), (c,4), (d,3)	{e,(b,10)}, {f,(b,8)}, {g,(c,11)}	{}
6	(s,0), (b,4), (c,4), (d,3), (e,10)	{f,(b,8)}, {g,(c,11)}	{e}
7	(s,0), (b,4), (c,4), (d,3), (e,10), (f,8)	{g,(c,11)}	{d}
8	(s,0), (b,4), (c,4), (d,3), (e,10), (f,8), (g,11)		

Tabelle 3.4.: Ablauf Framework (D'Esopo-Pape) auf einem Beispielgraph

#	(v, d_v) (wenn $d_v \neq \infty$)	q	Out(desc)
0		{s,(null, 0)}	{b,c,d}
1	(s,0)	{d,(s,3)}, {b,(s,4)}, {c,(s,7)}	{c}
2	(s,0), (d,3)	{b,(s,4)}, {c,(d,4)}	{c,e,f,g}
3	(s,0), (b,4), (d,3)	{c,(d,4)}, {f,(b,8)}, {e,(b,10)}, {g,(b,16)}	{g}
4	(s,0), (b,4), (c,4), (d,3)	{f,(b,8)}, {e,(b,10)}, {g,(b,11)}	{e}
5	(s,0), (b,4), (c,4), (d,3), (f,8)	{e,(b,10)}, {g,(b,11)}	{}
6	(s,0), (b,4), (c,4), (d,3), (e,10), (f,8)	{g,(b,11)}	{d}
7	(s,0), (b,4), (c,4), (d,3), (e,10), (f,8), (g,11)		

Tabelle 3.5.: Ablauf Framework (Dijkstra) auf einem Beispielgraph

Gegeben ist der Graph aus Abbildung 3.5. Der Ablauf der Berechnung für alle drei Implementierungen wird in den Tabellen 3.3, 3.4 und 3.5 für alle Algorithmen des

Frameworks veranschaulicht. Das Resultat des Initialisierungsprozesses ist jeweils in Zeile 0 dargestellt. Die Warteschlange enthält zu Beginn nur einen Eintrag für den Startknoten s. Dieser wird in der ersten Iteration der Hauptschleife entnommen und verarbeitet. Eine Distanzverbesserung für den Eintrag ist vorhanden ($\Delta < 0$), sodass dessen Distanz- und Elterninformationen mit den Werten aus dem Eintrag aktualisiert werden.

Für die Endknoten der aus s ausgehende Kanten $Out(s)$ werden nach Überprüfung der Distanzen $d[s] + w(e), e \in \{(s,b),(s,c),(s,d)\}$, welche kleiner ∞ sind, neue Einträge erstellt und der Warteschlange q hinzugefügt. Die Algorithmen von Bellman und Ford und D'Esopo-Pape fügen Einträge in der Reihenfolge ein, in der sie im Programmablauf erstellt werden (im Beispiel wird erst die Kante (s,b) verarbeitet, gefolgt von (s,c) und (s,d)). Der Algorithmus von Dijkstra benutzt eine Vorrangwarteschlange, welche die Einträge nach ihrem Schlüssel, der Distanz zum Knoten s sortiert. Folglich wird im ersten Schritt der Eintrag für der Knoten c extrahiert, während die anderen Algorithmen den Eintrag für Knoten b verarbeiten.

Der Algorithmus von Bellman und Ford fügt in jeder Iteration alle ausgehenden Kanten der Warteschlange q hinzu oder aktualisiert vorhandene Einträge, sofern die Distanzüberprüfung feststellt, dass die Distanz mittels der betrachteten Kante verbessert werden kann. Der Algorithmus terminiert, sofern alle Einträge der Warteschlange verarbeitet wurden und diese leer ist. Der Tabelle 3.3 ist zu entnehmen, dass der Warteschlange mehrfach Einträge für einige Knoten (c und g) entnommen werden. Während für den Knoten c durch den zweiten Eintrag der Elternknoten verändert wird (von b nach d), bleibt dieser für den Knoten g gleich. Das liegt daran, dass der zweite Eintrag für Knoten c (Elternknoten von g) erst nach der Verarbeitung des ersten Eintrags für g extrahiert wird.

Diesen Effekt vermeidet der Algorithmus von D'Esopo-Pape, indem ein Eintrag an der ersten Stelle der Warteschlange eingefügt wird, wenn bereits ein Eintrag für den gleichen Knoten extrahiert wurde. Tabelle 3.4 verdeutlicht den Ablauf. In Iteration 3 wird eine ausgehende Kante zu c ermittelt und ein Eintrag in die Warteschlange an erster Stelle hinzugefügt, sodass dieser in Iteration 4 extrahiert wird. Eine doppelte Bearbeitung von Einträgen für den Knoten g wird dadurch vermieden. Der Algorithmus terminiert nach 8 Iterationen, und kann somit im Vergleich zum Algorithmus von Bellman und Ford eine Ausführung einsparen.

Der Algorithmus von Dijkstra extrahiert immer den Knoten, welcher den geringsten Distanzeintrag aufweist. Er stellt somit sicher, dass die extrahierte Distanz auch im

folgenden Verlauf nicht unterboten werden kann und somit final berechnet ist. Folglich terminiert der Algorithmus nach derselben Anzahl an Iterationen, welche der Graph an Knoten ausweist. In diesem Beispiel sind es 7 Iterationen für die 7 Knoten des Graphen, eine Iteration weniger als beim Algorithmus von D'Esopo-Pape. Der resultierende SPT kann Abbildung 3.6 entnommen werden.

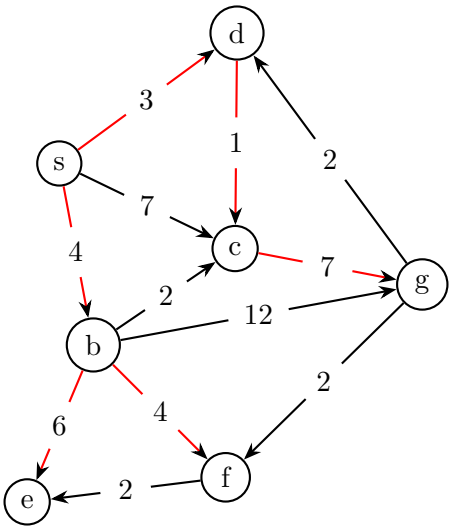


Abbildung 3.6.: Beispielgraph G (Framework) mit SPT

4

Semidynamische Algorithmen

Dieses Kapitel beschäftigt sich mit semidynamischen Algorithmen zur Lösung des DSP. Die Gewichte für Kanten auf dynamischen Graphen können inkrementiert oder dekrementiert werden. Semidynamische Algorithmen zeichnen sich dadurch aus, dass sie lediglich eine Art der Gewichtsänderung (Erhöhungen oder Verminderungen) je Durchlauf verarbeiten können und daher iterativ für beide Änderungsarten ausgeführt werden müssen. Der Verbund von zwei semidynamischen Algorithmen zur Verarbeitung verschiedenartiger Änderungen wird im Folgenden als zusammengesetzter volldynamischer Algorithmus bezeichnet. In einem solchen Verbund können verschiedene semidynamische Algorithmen beliebig kombiniert werden. Volldynamische Algorithmen (siehe Kapitel 5) hingegen benötigen nur eine Ausführung.

In Abschnitt 4.1 werden verschiedene Arten der Gewichtsänderung von Kanten respektive ihrer Zugehörigkeit zum SPT diskutiert. Im folgenden Abschnitt 4.2 wird das statische Framework mithilfe weniger Anpassungen insofern modifiziert, dass es gleichartige dynamische Veränderungen der Graphen verarbeiten kann, ohne die Lösung von Grund auf neu berechnen zu müssen. Im Abschnitt 4.3 wird ein eigens für das DSP-Problem entwickelter

Algorithmus, welcher auf dem Modell der ”Balls and Strings” basiert, vorgestellt und diskutiert.

4.1. Kategorisierung von Gewichtsänderungen

Gegeben sei ein Graph $G = (V, E)$ mit einer Gewichtsfunktion $w : E \to \mathbb{R}^{\geq 0}$, ein bereits für G berechneter SPT T_s und eine Menge $\epsilon = \{\langle e_i, \tau \rangle \mid e_i \in E \wedge \tau_i \in \mathbb{R}\}$ von Paaren, welche die Gewichtsänderungen der Kanten repräsentieren. Eine Gewichtsänderung einer einzelnen Kante kann bezogen auf T_s in eine der vier folgenden Kategorien eingeordnet werden:

4.1.1. Erhöhung des Gewichts von Kanten im Baum

Von dieser Art der Gewichtsänderung können nur Knoten betroffen sein, deren eigene kürzeste Pfade die veränderte Kante im SPT enthalten. Es muss überprüft werden, ob die Kante mit der inkrementierten Gewichtung weiterhin Teil des kürzesten Pfads ist und sich lediglich die Distanz vergrößert, oder ein Pfad über andere Kanten zu einer kürzeren Distanz führt. Knoten, welche diese Kante nicht auf ihrem kürzesten Pfad enthalten, sind von dieser Änderung nicht betroffen.

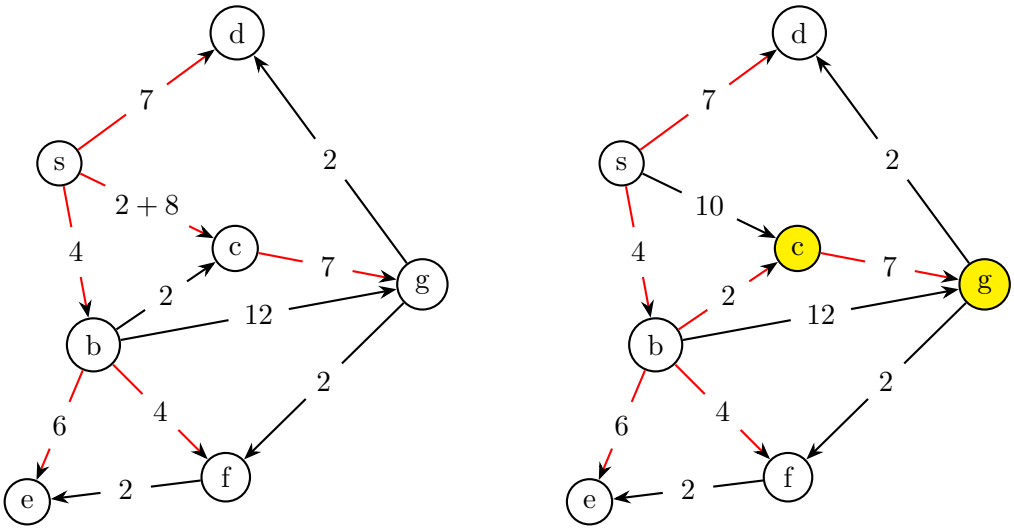


Abbildung 4.1.: Beispielgraph mit SPT - Erhöhung im Baum

Abbildung 4.1 verdeutlicht die Auswirkungen. Die Kante (s, c) ist Teil des SPT und wird um 8 Einheiten auf 10 erhöht. Die kürzesten Pfade zu c und g enthalten die veränderte Kante, sodass diese gegebenenfalls durch Hinzunahme anderer Kanten über einen kürzeren Pfad erreicht werden können. Dies ist der Fall unter Verwendung der Kanten (s, b) und (b, c), sodass die Distanzen $d_c = 6$ und $d_g = 13$ betragen.

4.1.2. Verminderung des Gewichts von Kanten im Baum

Von dieser Art der Gewichtsänderungen können alle Knoten betroffen sein, für die Pfade in G existieren, welche sie mit dem Startknoten über die betroffene Kante verbinden. Knoten, welche die veränderte Kante im kürzesten Pfad enthalten, werden zwangsläufig ihre Distanz zum Startknoten um die Gewichtsänderung der Kante dekrementieren. Für aus diesen Knoten ausgehende Kanten muss überprüft werden, ob für die am Kopf dieser Kanten liegenden Knoten und gegebenenfalls deren Nachfolger mit dem neuen Gewicht der Kante neue kürzeste Pfade gefunden werden können.

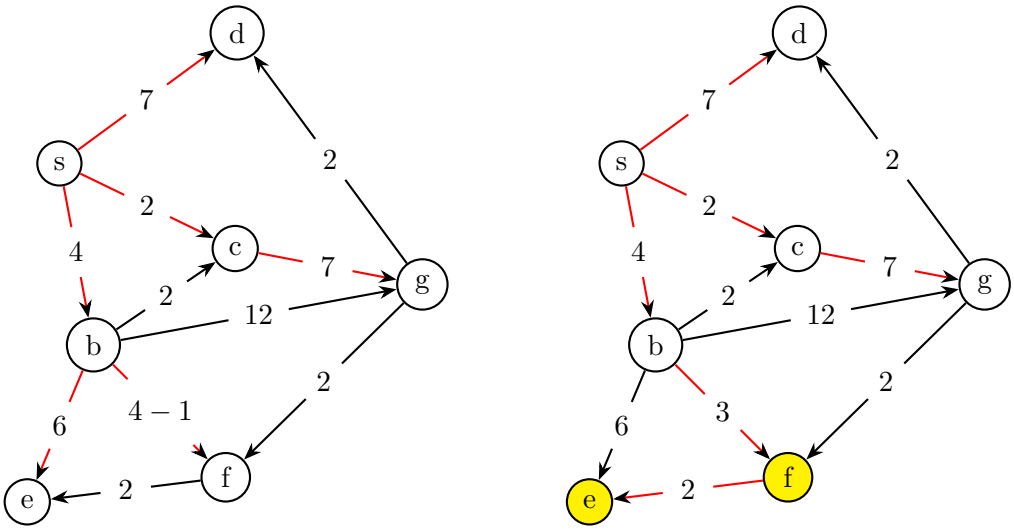


Abbildung 4.2.: Beispielgraph mit SPT - Verminderung im Baum

In Abbildung 4.2 wird das Gewicht der Kante (b, f), welche im SPT enthalten ist, um eine Einheit reduziert. Dies führt dazu, das neben der Distanzänderung $d_f = 7$ auch ein neuer kürzester Pfad für den Knoten e, welcher Nachfolger von f im Graphen und nicht im Ursprungs-SPT ist, gefunden werden kann.

4.1.3. Erhöhung des Gewichts von Kanten außerhalb des Baums

Die Erhöhung des Gewichts einer Kante außerhalb des Baums kann keine Änderung des SPT zur Folge haben, da Pfade über diese Kante bereits vor der Gewichtsänderung eine zu hohe Distanz aufwiesen, und diese durch die Änderung noch weiter ansteigt.

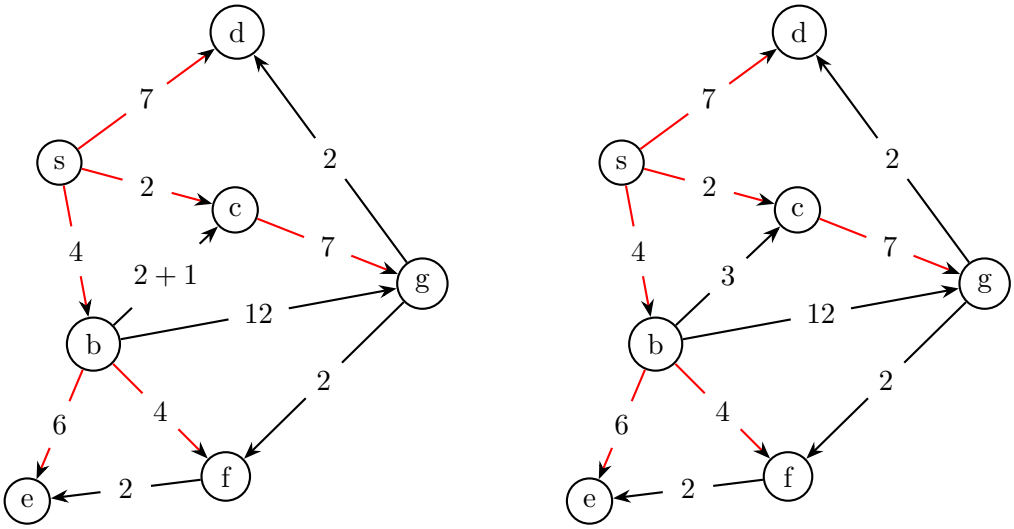


Abbildung 4.3.: Beispielgraph mit SPT - Erhöhung außerhalb des Baums

Abbildung 4.3 zeigt eine Erhöhung des Gewichts der Kante (b, c), welche sich nicht im SPT befindet. Die Erhöhung des Gewichts wird übernommen, am SPT findet jedoch keine Änderung statt.

4.1.4. Verminderung des Gewichts von Kanten außerhalb des Baums

Die Verminderung des Gewichts einer Kante außerhalb des Baum kann dazu führen, dass ein neuer kürzester Pfad für den Kopfknoten der Kante gefunden wird. Ist dies der Fall, so wird dieser Pfad übernommen und alle Nachfolger des Knotens im SPT angepasst. Für alle aus der Menge der angepassten Knoten ausgehenden Kanten muss überprüft werden, ob ihre Kopfknoten sowie deren Nachfolger ebenfalls mit Hilfe der veränderten Kante über neue kürzeste Pfade erreicht werden können.

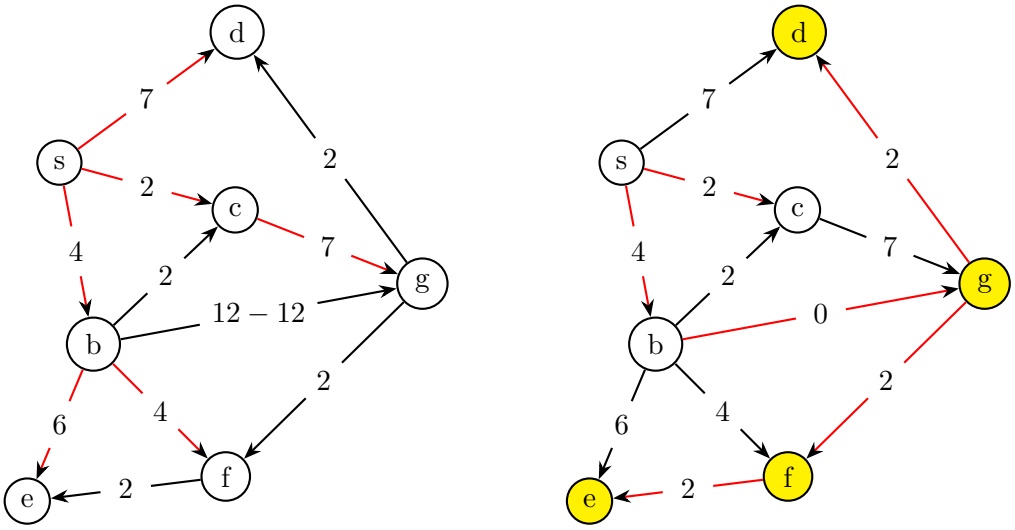


Abbildung 4.4.: Beispielgraph mit SPT - Verminderung im Baum

Das Beispiel in Abbildung 4.4 verdeutlicht das Verhalten. Die Verminderung des Gewichts der Kante (b, g) resultiert in einem neuen kürzesten Pfad von s nach g über b. Die Reduzierung der Distanz d_g führt dazu, dass sich die kürzesten Pfade für die Knoten d, e, f ändern, indem sie g traversieren.

4.2. Anpassung des Frameworks für dynamische Graphen

In diesem Abschnitt wird das Framework für statische Algorithmen aus Kapitel 3.4 insofern angepasst, dass es das DSP-Problem lösen kann. In Abschnitt 4.1 wurden die verschiedenen Möglichkeiten für Gewichtsveränderungen der Kanten respektive ihrer Lage in kürzesten Pfaden kategorisiert und das jeweilige Vorgehen für die Berechnung eines aktualisierten SPT erläutert. Paolo Narváez beschreibt in [NA00] zwei Stufen der Anpassung, durch die der Algorithmus diese Verfahren umsetzt. Die erste Stufe der Anpassung findet ausschließlich im Initialisierungsprozess des Algorithmus statt. Ziel ist es, nach dem zuvor beschriebenen Vorgehen die Menge der von den Änderungen (direkt) betroffenen Knoten zu bestimmen (vgl. 4.1) und für diese Einträge in die Warteschlange q hinzuzufügen. Diese werden anschließend in der Hauptschleife des Algorithmus in derselben Art und Weise wie im statischen Fall verarbeitet und somit in den SPT mit aktualisierten Elternknoten und Distanzwerten eingegliedert. Die zweite Stufe der

Anpassung baut auf der ersten auf und ändert die Arbeitsweise der Hauptschleife, indem die Funktion zur Bestimmung der Nachfolger innerhalb des SPT modifiziert wird.

4.2.1. Erste Stufe der Anpassung

Die erste Stufe der Anpassung konzentriert sich ausschließlich auf die Initialisierungsroutine des Algorithmus. Zusätzliche Parameter der dynamischen Version sind der SPT und die Menge der Paare von Kanten $e \in E$ und deren Änderungswerten θ. Über diese Menge wird iteriert, wobei in jedem Durchlauf betroffene Knoten lokalisiert und entsprechende Einträge in die Warteschlange eingetragen werden. Bei der Bearbeitung werden gleichartige Änderungen (Inkrementierung oder Dekrementierung) unterschiedlich behandelt. Verfügt die Menge der Änderungspaare über verschiedenartige Änderungen, so müssen verschiedene Arten des Algorithmus sequentiell ausgeführt werden. Diese Arten werden im Folgenden vorgestellt. Sie ersetzen die Initialisierungsroutine der statischen Version (abgebildet in Listing 4.1).

```
1    // Initialisierung Distanzarray d, Elternarray p, enthalten in
         SPT T
2    for (Vertex v : V)
3        d[v] = INF;
4        p[v] = null;
5    q.enqueue({s, (null, 0)});
```

Listing 4.1: Initialisierungsroutine des statischen Frameworks

Erhöhung der Kantengewichte

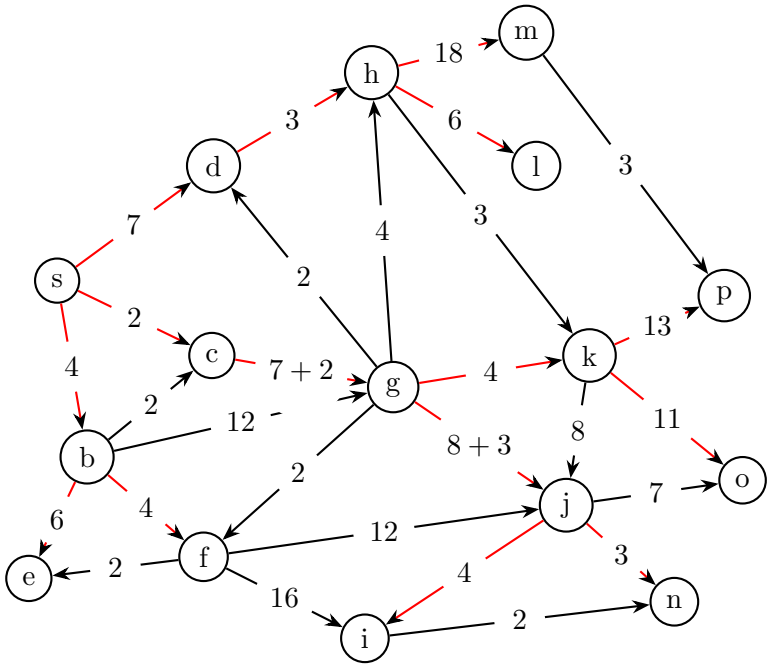
Die Initialisierungsroutine für Kantenerhöhungen ist in Listing 4.2 abgebildet.

```
1  Void Initialize (G, T, w, eps)
2      // Menge der Paare <Kante, Aenderungswert> eps
3      for ({Edge e, int theta} : eps)
4          w(e) += theta;
5          if (T.containsEdge(e))
6              descendants = des(h(e));
7              for (Vertex v : descendants)
8                  d[v] += theta;
9              for (Edge e2 : In(descendants))
10                 if (d[h(e2)] > d[t(e2)] + w(e2))
11                     q.enqueue({h(e2), (t(e2), d[t(e2)]+w(e2))});
```

Listing 4.2: Anpassung Initialisierungsroutine - Erhoehung

Die Menge der Paare aus Kanten e und Änderungswerten der Gewichte $\theta > 0$ wird in dem Parameter ϵ übergeben. Im ersten Schritt wird die Gewichtsfunktion aktualisiert, gefolgt von einer Überprüfung, ob die Kante e in dem ebenfalls übergebenen alten SPT T enthalten ist. Wie bereits in 4.1 beschrieben, haben Erhöhungen des Gewichts für Kanten, welche nicht in T enthalten sind, keinen Einfluss auf die weitere Berechnung, sodass keine weitere Bearbeitung des Eintrags stattfindet. Ist die Kante e Teil des Baums, so werden seine Nachfolgerknoten in T bestimmt und ihre Distanzen um den Änderungswert θ inkrementiert. Anschließend werden alle Eingangskanten der Nachfolgerknoten bestimmt und für diese überprüft, ob die kürzeste Distanz des Endknotens über die Eingangskante verbessert werden kann. Sofern diese Überprüfung erfolgreich ist, wird ein Eintrag für den Knoten der Warteschlange q hinzugefügt, welche in der Hauptschleife des Algorithmus wie zuvor im statischen Fall verarbeitet wird.

Beispiel



v	$d_{old}[v]$
s	0
b	4
c	2
d	7
e	10
f	8
g	9
h	10
i	21
j	17
k	13
l	16
m	28
n	20
o	24
p	26

Abbildung 4.5.: Beispielgraph G (dynamische Framework - Erhöhungen)

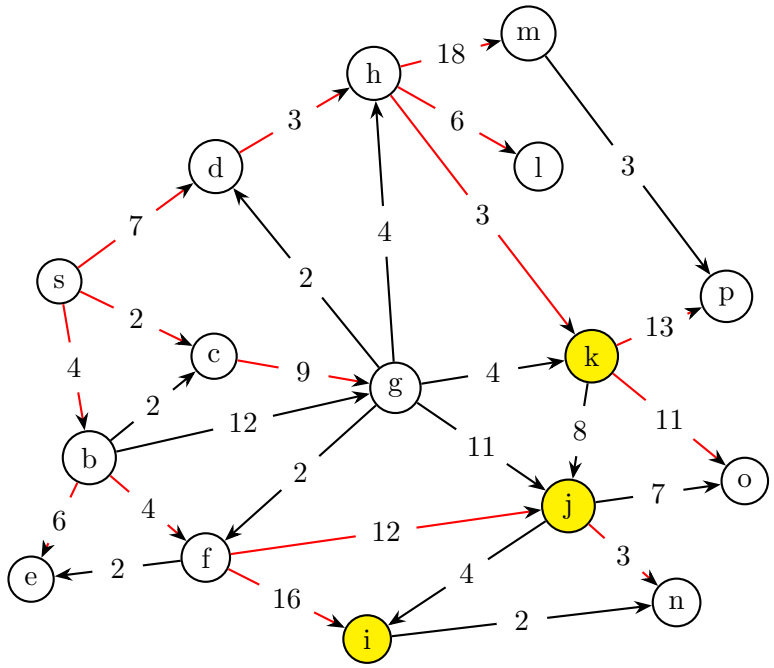
#	e, θ	descendants	In(descendants)	Einträge für q
1	((c,g),2)	{g,k,p,o,j,i,n}	{(c,g), (h,k), (m,p), (f,i), (f,j)}	{k,(h,13)}
2	((g,j),3))	{j,i,n}	{(g,j), (h,j), (f,j), (f,i)}	{j, (f,20)}, {i, (f,24)}

Tabelle 4.1.: Ablauf Initialisierungsroutine - Erhöhung

Tabelle 4.1 verdeutlicht das Vorgehen der Initialisierungsroutine auf dem Graphen G (4.5), für den der in rot eingezeichnete SPT vorhanden ist und die Kanten (c, g) und (g, j) inkrementiert werden sollen. Nachfolger der Kante in T, sowie die Menge der eingehenden Kanten können dort entnommen werden. Im ersten Durchlauf wird ein Eintrag für den Knoten k in die Warteschlange q, im zweiten Durchlauf für Knoten j und i, eingefügt. In der Hauptschleife werden diese äquivalent zur Arbeitsweise im statischen Fall (vgl. 3.3) verarbeitet. Im Detail bedeutet dies, dass unter der Bedingung einer Distanzverbesserung folgende Schritte stattfinden:

- der Elternknoten und die Distanz aus dem Eintrag werden in den SPT übernommen,
- für alle Endknoten ausgehender Kanten wird ein Eintrag in q hinzufügt, sofern eine Distanzverbesserung über diese Kante erreicht werden kann.

Der resultierende neue SPT ist in Abbildung 4.6 enthalten.



v	$d[v]$
s	0
b	4
c	2
d	7
e	10
f	8
g	11
h	10
i	24
j	20
k	13
l	16
m	28
n	23
o	24
p	26

Abbildung 4.6.: Beispielgraph G (dynamische Framework - Erhöhungen, Lösung)

Verminderung der Kantengewichte

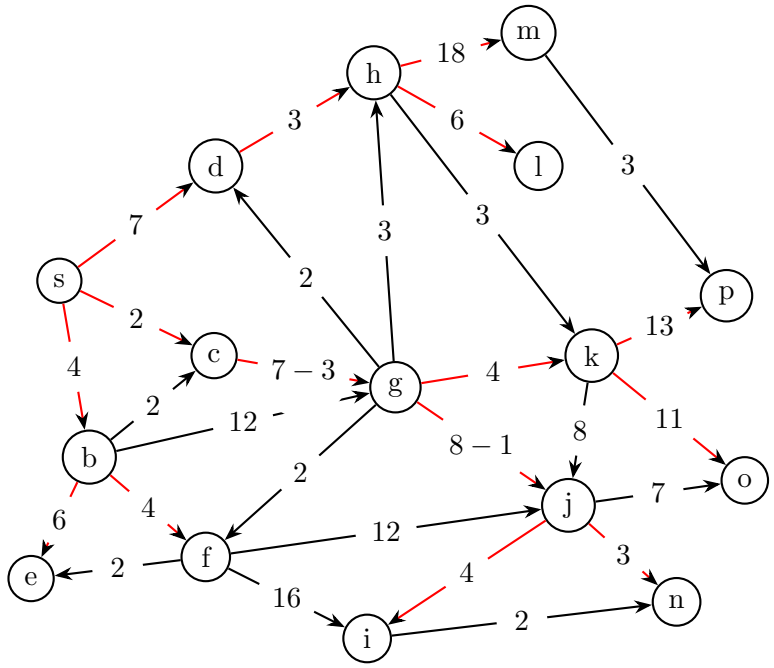
Die Initialisierungsroutine für Kantenverminderungen ist in Listing 4.3 abgebildet.

```
1     // Verminderung Kantengewichtung
2     for ((Edge e, int theta) : eps)
3         w(e) += theta
4         delta = (d[t(e)] + w(e)) - d[h(e)];
5         if (delta < 0)
6             p[h(e)] = t(e);
7             descendants = des(h(e));
8             for (Vertex v : descendants)
9                 d[v] += delta;
10            for (Edge e2 : Out(descendants))
11                newdist = d[t(e)] + w(e2);
12                if (d[h(e)] > newdist)
13                    q.enqueue({h(e2), (t(e2), newdist)});
```

Listing 4.3: Anpassung Initialisierungsroutine - Verminderung

Auch im Fall der Verminderung der Kantengewichte wird zunächst die Gewichtsfunktion aktualisiert. Ein δ wird bestimmt, welches den Wert θ annimmt, sofern die Kante e innerhalb des SPT enthalten ist, oder anzeigt, ob eine Distanzverbesserung für den Knoten $h(e)$ vorliegt, wenn die Kante in T integriert wird. Ist dies der Fall, so ist $\delta < 0$, anderenfalls ist die Initialisierungsphase für die Kante abgeschlossen. Da unter Umständen ein neuer kürzester Pfad für den Knoten $h(e)$ gefunden werden konnte ($\delta < 0$), wird das Elternattribut überschrieben. Es werden die Nachfolger in T bestimmt und mit δ verrechnet. Für die aus dieser Menge ausgehenden Kanten wird überprüft, ob deren Endknoten über eine neue kürzesten Distanz erreicht werden können und es werden gegebenenfalls Einträge für sie in die Warteschlange q eingefügt.

Beispiel



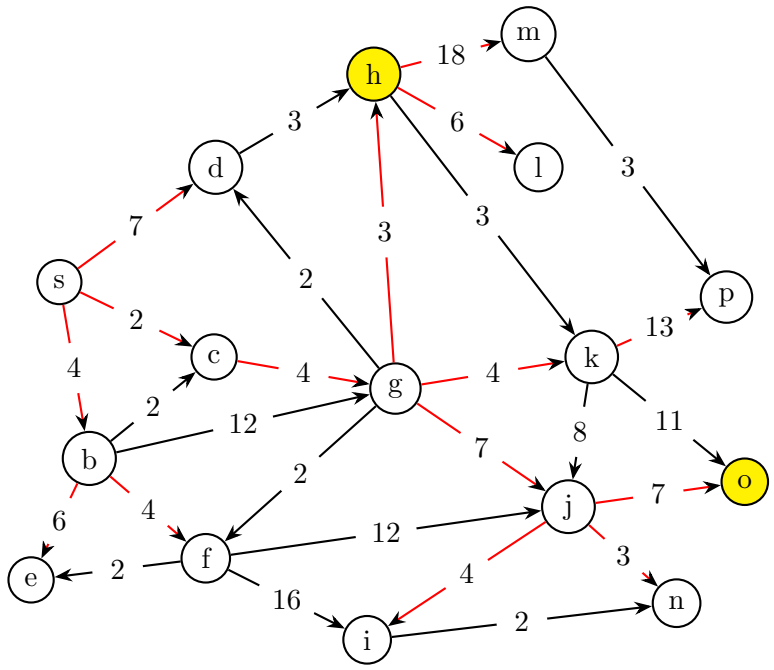
v	$d_{old}[v]$
s	0
b	4
c	2
d	7
e	10
f	8
g	9
h	10
i	21
j	17
k	13
l	16
m	28
n	20
o	24
p	26

Abbildung 4.7.: Beispielgraph G (dynamische Framework - Verminderung)

#	$e, theta$	descendants	Out(descendants)	Einträge für q
1	((c,g),-3)	{g,k,p,o,j,i,n}	{(g,f), (g,h), (g,d)}	{h, (g,9)}
2	((g,j),-1))	{j,i,n}	{(j,o)}	{o, (j,20)}

Tabelle 4.2.: Ablauf Initialisierungsroutine - Verminderung

Tabelle 4.2 verdeutlicht das Vorgehen der Initialisierungsroutine auf dem Graphen G (Abbildung 4.7), für den der in rot eingezeichnete SPT vorhanden ist und die Kanten (c, g) und (g, j) dekrementiert werden sollen. Nachfolger der Kante in T, sowie die Menge der eingehenden Kanten können dort entnommen werden. Im ersten Durchlauf wird q ein Eintrag für den Knoten h, im zweiten Durchlauf für Knoten o, hinzugefügt. In der Hauptschleife werden diese wie bereits für den statischen Fall beschrieben verarbeitet. Der resultierende neue SPT ist in Abbildung 4.8 enthalten.



v	$d[v]$
s	0
b	4
c	2
d	7
e	10
f	8
g	6
h	9
i	17
j	13
k	10
l	15
n	16
o	20
p	23

Abbildung 4.8.: Beispielgraph G (dynamische Framework - Verminderung, Lösung)

4.2.2. Zweite Stufe der Anpassung

Ein Nachteil der ersten Stufe der Anpassung liegt darin, dass pro Iteration der Hauptschleife lediglich ein Knoten verarbeitet wird. Dies ist begründet durch die Definition der Funktion zur Bestimmung der Nachfolger

$$des(vertex) = \{vertex\}$$

Die resultierende Ergebnismenge enthält ausschließlich den übergebenen Knoten. Die Hauptschleife aktualisiert die Informationen des Knoten und erstellt neue Warteschlangeneinträge für weitere Knoten, welche über aus v ausgehende Kanten mit einer kürzeren Distanz erreichbar sind.

Ziel der neuen Definition der Funktion ist es, möglichst viele Nachfolgeknoten zu bestimmen. Es sollen lediglich die Knoten nicht bearbeitet werden, für die Einträge in der Warteschlange q mit einem kleineren Distanzeintrag existieren. Diese werden im späteren Verlauf der Hauptschleife ohnehin bearbeitet, sodass eine sofortige Bearbeitung

zwangsläufig im späteren Verlauf überschrieben wird. Nachfolgeknoten, welche über einen größeren Distanzeintrag in q verfügen, sollen hingegen verarbeitet und die entsprechenden Einträge aus q entfernt werden.

Die Bearbeitung der Einträge findet auf dieselbe Art wie zuvor statt, durch die Wahl der Nachfolgemenge werden jedoch mehrere Knoten pro Schleifendurchlauf bearbeitet. Einstiegspunkt der Funktion *des* zur Bestimmung der Nachfolger ist Zeile 8 der Hauptroutine des Frameworks (siehe Listing 4.4). Am Quellcode der Hauptroutine des Frameworks findet keine direkte Änderung statt.

Zusätzlich dazu, dass mit der zweiten Stufe der Anpassung pro Iteration mehrere Knoten verarbeitet werden können, wird auf diese Weise auch garantiert, dass die minimale Anzahl an Veränderungen der Elternbeziehung stattfindet, denn eine Kante aus dem alten SPT wird nur entfernt, wenn sie im neuen SPT nicht vorhanden sein kann.

```
1   // Hauptschleife
2   while (q.size != 0)
3       {v, (p_new, d_new)} = q.extractmin();
4       delta = d_new - d[v];
5       if (delta < 0)
6           p[v] = p_new;
7           // Bestimmung der Nachfolger
8           descendants = des(v); // Hier findet die Aenderung
                statt
9           for (Vertex c : descendants)
10              // Distanzupdate
11              d[c] += delta;
12          for (Edge (u, v) : Out(descendants))
13              newdist = d[u] + w(u,v)
14              if (d[v] > newdist)
15                  q.enqueue({v, (u, newdist)})
```

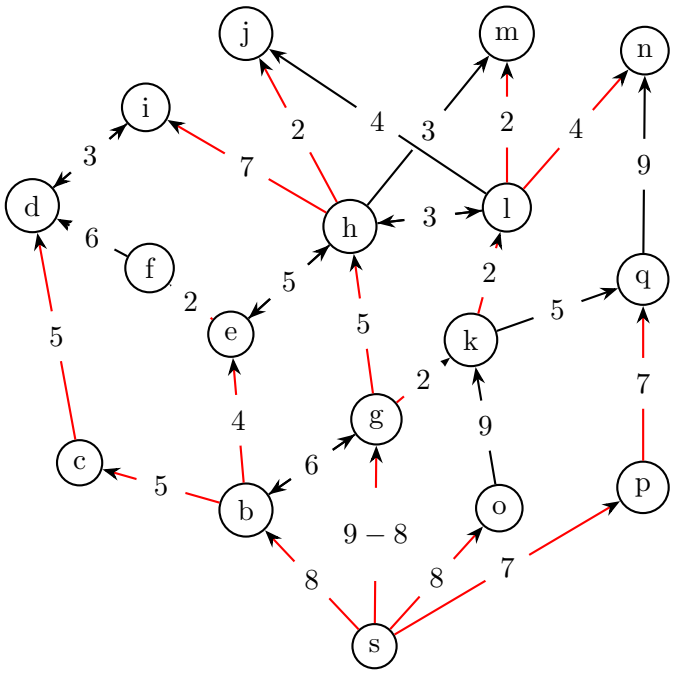
Listing 4.4: Hauptroutine

```
1  // Funktion zur Bestimmung der Kinder, welche verarbeitet werden
       sollen
2  Set<Vertex> des(v)
3      // Initialisierung: temporaere Warteschlange K, Ergebnismenge
           N
4      N = {};
5      K.enqueue(v);
6      // Schleife zur Bestimmung der relevanten Nachbarn
7      while (K.size != 0)
8          k = K.extractmin();
9          N += k;
10         for (Vertex u : spc(k)) //  Iteration ueber die Kinder
11                                 //  von k im Baum T ()
12             if (q.contains(u))
13                 {v1, (v2, qdistance)} = q.peek(u);
14                 if (d[u] + delta <= qdistance)
15                     q.remove(u);
16                     K.enqueue(u);
17             else
18                 K.enqueue(u);
19     return N;
```

Listing 4.5: Anpassung Nachfolgerfunktion

Die neue Funktion zur Bestimmung der Nachfolger ist in Listing 4.5 definiert: Die Nachfolger werden in der Menge N gespeichert. Zusätzlich wird eine Hilfswarteschlange K (nach dem FIFO-Prinzip) verwendet, welche einfache Knoten als Einträge enthält. Solange diese nicht leer ist, wird pro Durchlauf ein Eintrag entfernt. Die direkten Nachfolgeknoten werden bestimmt und für diese überprüft, ob sie enthaltene Einträge in der Warteschlange q vorhanden sind. Ist dies der Fall und die Distanz des Eintrags ist größer der Summe der aktuellen Distanz und δ, so wird der Eintrag aus q entfernt und der Hilfswarteschlange K hinzugefügt. Anderenfalls findet lediglich eine Hinzunahme für den Knoten in K statt.

Beispiel



v	$d_{old}[v]$	$d'[v]$ (n.Init)
s	0	0
b	8	8
c	13	13
d	18	18
e	12	12
f	14	14
g	9	1
h	14	6
i	21	13
j	16	8
k	11	3
l	13	5
m	15	7
n	17	9
o	8	8
p	7	7
q	14	14

Abbildung 4.9.: Beispielgraph G (2. Stufe d. Anpassung - Ausgangsgraph)

#	e, θ	descendants	Out(descendants)	Einträge für q
1	((s,g),-8)	{h,i,j,k,l,m,n}	{(g,b), (h,e), (i,d), (k,q)}	{b,(g,7)}, {e,(h,11)}, {d,(i,16)}, {q,(k,8)}

Tabelle 4.3.: Ablauf Initialisierungsroutine - 2. Stufe d. Anpassung

q	Δ	descendants	Out(descendants)	Einträge für q
{b,(g,7)}, {q,(k,8)}, {e,(h,11)}, {d,(i,16)}	-1	{b}	{(b,c), (b,e)}	{c,(b,12)}, ({e,(b,11)})
{q,(k,8)}, {e,(h,11)}, {c,(b,12)}, {d,(i,16)}	-6	{q}	{(q,n)}	
{e,(h,11)}, {c,(b,12)}, {d,(i,16)}	-1	{e}	{(e,f), (e,h)}	{f,(e,13)}
{c,(b,12)}, {f,(e,13)}, {d,(i,16)}	-1	{c}	{(c,d)}	({d,(c,17)})
{f,(e,13)}, {d,(i,16)}	-1	{f}	{(f,d)}	
{d,(i,16)}	-2	{d}	{(d,i)}	

Tabelle 4.4.: Ablauf Hauptschleife - 1. Stufe d. Anpassung

q	Δ	desc	Out (descendants)	q-Eintrag gelöscht	Einträge für q
{b,(g,7)}, {q,(k,8)}, {e,(h,11)}, {d,(i,16)}	-1	{b,c,e,f}	{(b,g), (c,d), (f,d)}	{e,(h,11)}	({d,(c,17)})
{q,(k,8)}, {d,(i,16)}	-6	{q}	{(q,n)}		
{d,(i,16)}	-2	{d}	{(d,i)}		

Tabelle 4.5.: Ablauf Hauptschleife - 2. Stufe d. Anpassung

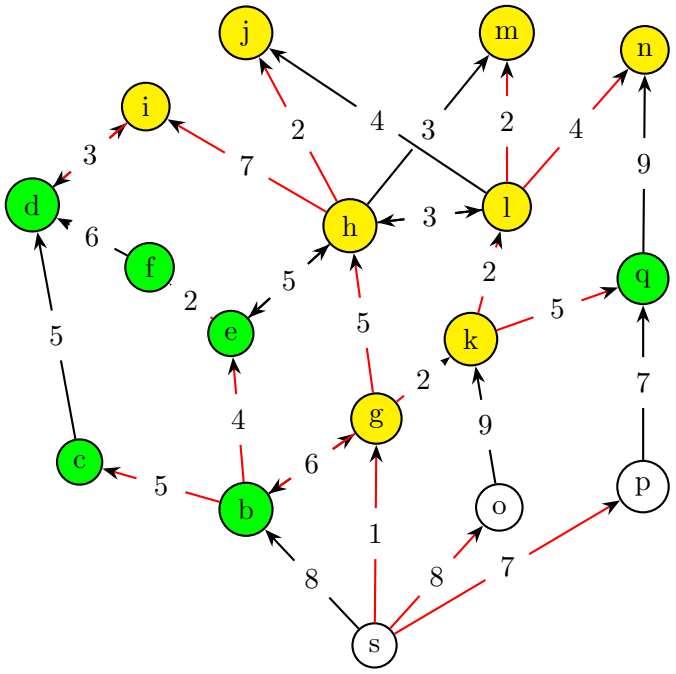
K	N	children(v)	q-Eintrag gelöscht	Einträge für K
b	{}	{c,e}	e	c, e
c, e	{b}	{d}		
e	{b, c}	{f}		f
f	{b, c, e}	{d}		
	{b, c, e, f}			

Tabelle 4.6.: Ablauf Nachfolgerfunktion - 2. Stufe d. Anpassung - 1. Aufruf

Das folgende Beispiel soll die Arbeitsweise der zweiten Stufe der Anpassung verdeutlichen und gleichzeitig die Vorteile im Vergleich zur ersten Stufe aufweisen. Für den Beispielgraph G aus Abbildung 4.9 soll das Gewicht der Kante (s, g) um 8 Einheiten reduziert werden. Die Tabelle in 4.9 enthält die Distanzwerte aller Knoten vor und nach der Initialisierungsphase des Algorithmus. Tabelle 4.3 enthält den Ablauf der Initialisierungsroutine. Sie iteriert einmalig, da nur das Gewicht einer Kante vermindert wurde, resultiert jedoch in der Änderung der Distanzen von 6 Nachfolgeknoten im SPT und zur Erstellung von 4 Einträgen, welche in die Warteschlange q eingefügt werden.

Tabelle 4.4 zeigt den Ablauf der Hauptschleife bei Verwendung der ersten Stufe der Anpassung, Tabelle 4.5 hingegen zeigt die Arbeitsweise der zweiten Stufe jeweils mit dem Dijkstra-Algorithmus als konkrete Implementierung. Tabelle 4.6 enthält die Arbeitsschritte der Nachfolgerfunktion exemplarisch für den ersten Aufruf mit Knoten b. Diese bestimmt zunächst die direkten Nachfolger c und e. Für den Knoten e wird der Eintrag aus q entfernt, da die Distanz mittels der Traversierung des Knoten b in gleichem Maße verringert werden kann wie bei der Traversierung des Knoten e, für welche ein Eintrag in q enthalten ist. Der Knoten d wird zwei Mal (als Nachfolger von c und f) überprüft, jedoch findet keine Aufnahme in die Hilfswarteschlange K statt, da in q ein Eintrag mit geringerer Distanz enthalten ist. Mit Hilfe der Nachfolgerfunktion wird bereits in der ersten Iteration der Hauptschleife die vierelementige Menge $\{b, c, e, f\}$ zur Verbesserung bestimmt und verarbeitet. Die Implementierung der ersten Stufe der Anpassung verarbeitet, wie auch der originale Dijkstra-Algorithmus, einen Knoten pro Iteration und benötigt somit sechs Schleifendurchläufe zur Bestimmung des neuen SPT. Die Implementierung der zweiten Stufe hingegen kann durch die Bestimmung der Menge im ersten Schritt drei Iterationen einsparen und terminiert somit nach drei Durchläufen.

Abbild 4.10 enthält den resultierenden SPT. Knoten, welche in der Initialisierungsphase bearbeitet wurden, sind gelb markiert. Grün markierte Knoten wurden hingegen in der Hauptphase bearbeitet. Die Tabelle enthält die Distanzen aller Knoten zum Zeitpunkt nach der Initialisierung (2. Spalte), sowie nach der Ausführung des Algorithmus (3. Spalte).



v	$d'[v]$ (n.Init)	$d'[v]$ (n.Alg)
s	0	0
b	8	7
c	13	12
d	18	16
e	12	11
f	14	13
g	1	1
h	6	6
i	13	13
j	8	8
k	3	3
l	5	5
m	7	7
n	9	9
o	8	8
p	7	7
q	14	8

Abbildung 4.10.: Beispielgraph G (2. Stufe d. Anpassung - Ergebnisgraph)

Korrektheit und Laufzeit

Der Korrektheitsbeweis des Algorithmus kann [NA00] entnommen werden. Gegeben sei ein Graph $G = (V, E)$ mit $n = |V|$ Knoten und $m = |E|$ Kanten, sowie ein SPT für den Ausgangsgraphen und eine Menge an Kantenänderungen ϵ. Ferner sei δ_d die minimale Anzahl an Knoten, dessen Distanz- oder Elternwerte geändert werden müssen und δ_{pd} die minimale Anzahl der Knoten, dessen Distanz- und Elternwerte geändert werden müssen. Diese Parameter sind nicht vom benutzen Algorithmus, sondern von

der Graphentopologie, dem SPT und ϵ abhängig und beschreiben die minimale Menge an Informationsänderungen zur Berechnung eines aktualisierten SPT. Des Weiteren sei D_{max} die maximale Anzahl der aus den Knoten ausgehenden Kanten in G. Die Laufzeit der einzelnen dynamischen Algorithmen, welche durch dieses Framework implementiert werden können, beträgt für die erste Stufe der Anpassung im Fall von

- Bellman-Ford: $\mathcal{O}(D_{max} * \delta_d^2)$,
- D'Esopo-Pape: keine polynomielle Laufzeit,
- Dijkstra (mit linearer Liste): $\mathcal{O}(\delta_d^2 + D_{max} * \delta_d)$,
- Dijkstra (mit binärer Halde): $\mathcal{O}(D_{max} * \delta_d * \lg \delta_d)$,
- Dijkstra (mit Fibonacci): $\mathcal{O}(\delta_d * \lg \delta_d + D_{max} * \delta_d)$,

und für die zweite Stufe der Anpassung im Fall von

- Bellman-Ford: $\mathcal{O}(D_{max} * \delta_d^3)$,
- D'Esopo-Pape: keine polynomielle Laufzeit,
- Dijkstra (mit linearer Liste): $\mathcal{O}(\delta_{pd} * \delta_d + \gamma * D_{max} * \delta_d)$,
- Dijkstra (mit binärer Halde): $\mathcal{O}(\gamma * D_{max} * \delta_d * \lg \delta_d)$,
- Dijkstra (mit Fibonacci): $\mathcal{O}(\delta_d * \lg \delta_d + D_{max} * \delta_d)$.

Die Beweise der einzelnen Laufzeiten können [NA00] entnommen werden.

4.3. Ball-and-String Algorithmus

Dieser Abschnitt befasst sich mit dem Ball-and-String-Algorithmus, welchen Paolo Narváez in [NA01] vorgestellt hat. Der Algorithmus verdankt seinen Namen dem sogenannten "Ball-and-String"-Modell (deutsch Modell der Bälle und Ketten), in welchem die Entstehung eines SPT durch ein physikalisches Experiment interpretiert werden kann.

In diesem Modell werden Knoten als Bälle und Kanten zwischen diesen als unelastische Ketten repräsentiert. Die Länge der Ketten ist anhängig von dem Gewicht, welches der Kante durch die Gewichtsfunktion zugewiesen wird. Der Ball, welcher den Startknoten repräsentiert, wird an einer Stelle fixiert. Ziel ist es, die restlichen Bälle in derselben Richtung möglichst weit vom Startknoten zu entfernen. Ihre Auslenkung wird jedoch

durch die Ketten begrenzt: Die maximale Auslenkung eines Balls ist erreicht, wenn mindestens auf einem Pfad, bestehend aus Bällen und Ketten, alle Ketten gespannt sind. Dieser Pfad ist der kürzeste, auf dem der Ball vom Startball aus erreicht werden kann. In einem physikalischen Experiment würde der Startball beispielsweise an der Decke eines Raums fixiert werden. Die restlichen Bälle werden aus gleicher Höhe gleichzeitig (in Richtung Boden) fallengelassen. Ein Ball fällt solange, bis jeweils eine Verbindung vom Startball aus zu ihm nur noch gespannte Ketten enthält. Da die Ketten unelastisch sind, verharrt der Ball an dieser Position. Wie beim SSSP-Problem ist es ebenso in diesem Modell möglich, dass mehrere Pfade zu einem Ball ausschließlich gespannte Ketten enthalten. Ist dies der Fall, so existieren mehrere kürzeste Pfade für denselben Ball. Zur Bestimmung eines gültigen SPT muss für alle Bälle, für die mehrere solcher Pfade vorhanden sind, eine Auswahl eines einzigen Pfads vorgenommen werden. Werden alle ausgewählten kürzesten Pfade in einem Baum zusammengefügt, so entsteht ein SPT für den durch das Modell repräsentierten Graphen.

Der statische Algorithmus von Dijkstra kann ebenfalls mit Hilfe dieses Modells interpretiert werden, da dieser die Distanzen der Knoten des Graphen in derselben Reihenfolge berechnet, wie die Bälle innerhalb des Modells eine feste Position einnehmen. Im Initialzustand befinden sich alle Bälle an derselben Position und der Startknoten s wird verankert. Die restlichen Bälle werden fallen gelassen. Der Algorithmus von Dijkstra tritt in die Hauptschleife und berechnet, welcher Knoten die kürzeste Distanz zum Startknoten besitzt. Im Modell ist dies der Zeitpunkt, an dem sich die erste Kette spannt. Der Ball erreicht seine finale Position und besitzt somit eine feste Distanz.

In der folgenden Iteration bestimmt der Algorithmus den Knoten, welcher die kürzeste Distanz zum Startknoten besitzt und noch nicht berechnet wurde. Im Modell geschieht dies wiederum zum Zeitpunkt, an dem sich eine zweite Kette spannt. Diese kann nur mit mindestens einem der fixierten Bälle (Startball oder erster verankerter Ball) aufgrund der Schwerkraft verbunden sein. Das im Modell beobachtete Verhalten wiederholt sich, bis alle Bälle eine feste Position einnehmen und entspricht genau der Berechnungsreihenfolge des Algorithmus von Dijkstra.

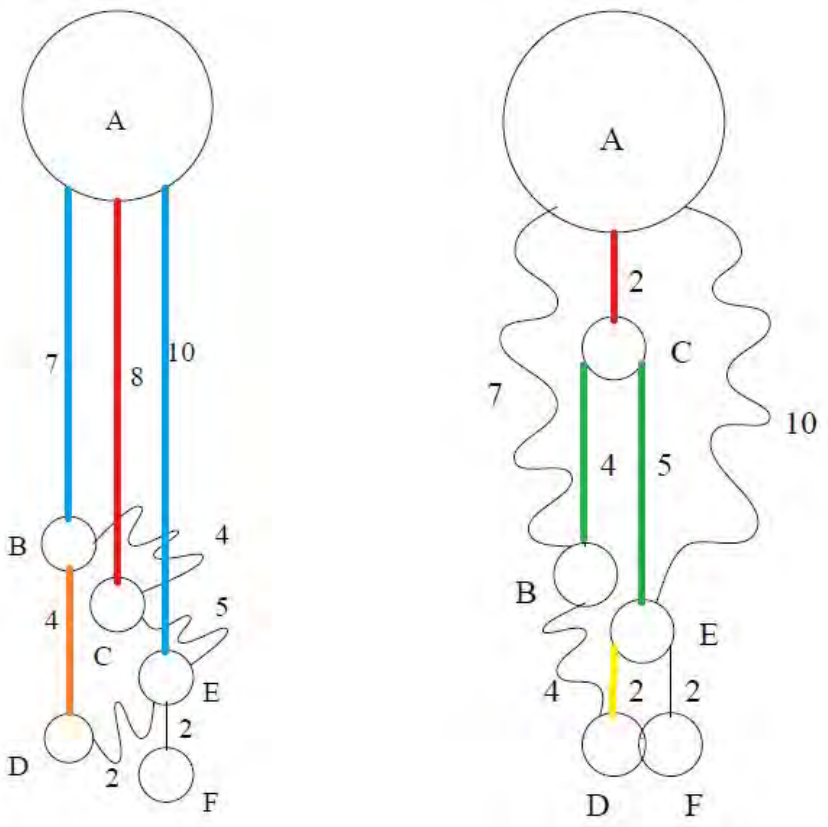


Abbildung 4.11.: ”Ball–and–String”-Modell (Quelle: [NA01])

Man stelle sich das in Abbildung 4.11 links aufgezeigte Modell im Initialzustand vor. Alle Bälle befinden sich auf der Höhe des Balls A, die Ketten sind entspannt. Der Ball A ist fixiert, die restlichen Bälle werden gleichzeitig losgelassen. Durch die Schwerkraft fallen sie nach unten in Richtung Boden. Der Fall eines Balls wird gestoppt, wenn eine Folge von Ketten gespannt ist. Dies tritt als erstes für den Ball B ein, da er über die kürzeste Kette zu A verfügt. Diesen Sachverhalt untersucht der Algorithmus von Dijkstra in der ersten Iteration und berechnet die endgültige Distanz für den Knoten, welcher durch den Ball B repräsentiert wird. Im Modell wird im Folgenden zunächst die Kette (A, C) gespannt und der Ball verharrt an seiner Position. Der Algorithmus von Dijkstra bestimmt diesen Ball in der zweiten Iteration, da dieser die kürzeste Gesamtdistanz der unberechneten Knoten zu A aufweist. Die Beobachtungen des Modells und Berechnungsschritte des Algorithmus wiederholen sich simultan zueinander für die restlichen Bälle bzw. Knoten, bis die maximale Auslenkung der Bälle im Modell erreicht ist und der Algorithmus kürzeste Distanzen für alle Knoten bestimmt hat.

Ebenso können grundlegende Beobachtungen für das Verhalten bei Änderungen der Kantenlängen aus dem beschriebenen und in 4.11 veranschaulichten Modell entnommen werden. Sei s ein Startball, der an einer festen Position (beispielsweise an der Decke eines Raums) verankert wird. Die restlichen Bälle werden auf der Höhe des Startballs losgelassen und fallen soweit wie möglich in Richtung des Bodens des Raums. Wird die Länge einer gespannten Kette erhöht, so wird der Ball am unteren Ende der Kette (und mit ihm gegebenenfalls weitere) solange Richtung Boden fallen, bis mindestens eine der mit dem Ball verbundenen Ketten erneut gespannt ist. Wird die Länge einer Kette vermindert, so wird der Ball hingegen gegebenenfalls durch die Kette nach oben gezogen. Auch in diesem Fall kann dieser Vorgang Auswirkungen auf weitere Bälle haben.

Abbildung 4.11 stellt ein solches "Ball–and–String"-Modell dar. Gerade Linien repräsentieren gespannte, kurvige Linien entspannte Ketten. Die Länge der Kette zwischen den Bällen A und C (rot) wird um 6 Einheiten gekürzt, sodass Ball C angehoben wird. In Folge der Erhöhung spannen sich die (grünen) Ketten (C, B) und (C, E), welche somit in die kürzesten Pfade der Kantenendknoten aufgenommen werden. Sie ersetzen die (blauen) Kanten zwischen den Bällen A und B (links) und A und E (rechts). Betrachtet man die Veränderung der Kettenlänge als kontinuierlichen, dynamischen Prozess (der Ball wird in einer gleichförmigen Bewegung angehoben), so wird die Anspannung der Kette (C, E) und die Entspannung der Kette (A, E) zu dem Zeitpunkt beginnen, ab dem der Ball C um 3 Einheiten erhöht ist. Dieser Prozess findet für den Ball B erst nach 5 Einheiten statt. Die Distanzverminderung für Ball E beträgt 3 Einheiten, für Ball B 1 Einheit: Je größer die Distanzverminderung eines Balles ist, desto früher bewegt er sich. Ein ähnlicher Vorgang ist für den Ball D durch die Ketten (B, D) (orange) und (E, D) (gelb) zu beobachten.

Der Ball-and-String-Algorithmus hat das Ziel, die Dynamik der Bälle bei einer Änderung der Kettenlänge zu simulieren. Die Verarbeitungsreihenfolge der betroffenen Knoten ist im Gegensatz zu beispielsweise der dynamischen Version des Algorithmus von Dijkstra nicht abhängig von der neu berechneten Gesamtdistanz, sondern es wird zunächst jeweils der Knoten ausgewählt, für den die minimale Distanzerhöhung bzw. maximale Distanzverminderung erreicht werden kann.

Der Algorithmus kann in eine Initialisierungsphase und eine Hauptphase unterteilt werden. Jeder Knoten erhält zusätzlich zu den Distanz- und Elternattributen ein Statusattribut, welches die Werte *verankert* (*anchored*) und *schwebend* (*floating*) annehmen kann. In einem endgültig berechneten SPT sind alle Knoten verankert. Der Algorithmus verwendet ebenfalls eine Warteschlange q, welche über die bekannten Operationen *ENQUEUE,*

EXTRACTMIN und *REMOVE* verfügt, jedoch Schlüsselwerte der Form $\langle value1, value2 \rangle$ enthalten. Primär ausschlaggebend für die Reihenfolge ist $value2$. Bei identischen Werten $value2$ zweier Einträge ist die Reihenfolge von $value1$ abhängig. Angenommen die Warteschlange enthält einen Eintrag für den Knoten v. Soll ein weiterer Eintrag für den selben Knoten hinzugefügt werden, so ersetzt dieser den vorhanden nur, wenn dessen zusammengesetzter Schlüssel kleiner ist.

Neben der Hilfsfunktion *des(v)*, welche alle Nachfolger des Knotens v im SPT T ermittelt, wird eine weitere Hilfsfunktion *des_anchored(v)* verwendet. Diese bestimmt alle Nachfolger von v in T, welche über Pfade erreicht werden können, entlang derer ausschließlich Knoten mit dem Statuswert *anchored* enthalten sind.

Initialisierungsphase

Die Initialisierungsphase des Algorithmus ist unterteilt in jeweils eine Fallbehandlung für Erhöhungen und Verminderungen der Gewichte der Kanten.

Erhöhung der Kantengewichte

Auf Grundlage der Überlegungen aus Abschnitt 4.1 wird in der Initialisierungsphase in Listing 4.6 für Erhöhungen von Kantengewichten die Menge der potentiell betroffenen Knoten bestimmt. Dazu werden die Nachfolger des Endknotens der Kante bestimmt, dessen Gewicht erhöht wurde. Diese Menge bildet einen Unterbaum innerhalb des SPT. Dieser wird durch das Zurücksetzen des Elternattributs der Wurzel aus dem SPT entnommen. Ihrem Status wird der Wert *floating* zugeordnet.

```
1     // Hilfsmenge N
2     N = 0;
3     // Erhoehung Kantengewichtung
4     for ((Edge e, int theta) : eps)
5         w(e) += theta
6         if (t(e) == spp(h(e)))
7             spp(h(e)) = null;
8             N2 = des_anchored(h(e));
9             for (Vertex v : N2)
10                status(v) = floating;
11            N += N2;
12    for (Edge e : In(N))
13        if (status(t(e)) == anchored)
14            newdist = d[h(e)] + w(e);
15            olddist = d[t(e)];
16            q.enqueue({h(e), (t(e), newdist, olddist-newdist)})
```

Listing 4.6: Ball-and-String-Algorithmus: Initialisierungsroutine - Erhöhung

Die Menge N enthält nach Ausführung der ersten Schleife sämtliche durch die Änderungen betroffene Knoten. Für sie erreichende Kanten $In(N)$ wird überprüft, ob die Anfangsknoten verankert sind. Ist dies der Fall, so wird q ein Eintrag für den Endknoten $h(e)$ mit der neuen Distanz und deren Differenz zur alten Distanz erstellt. Durch die Bedingung, dass der Status des Anfangsknotens verankert sein muss, werden nur Einträge für solche Knoten erstellt, welche sich an der Grenze zu bereits verankerten Knoten befinden.

Verminderung der Kantengewichte

Listing 4.7 zeigt die Initialisierungsphase für den Fall, dass Gewichte der Kanten vermindert werden sollen.

```
1     // Verminderung Kantengewichtung
2     for ((Edge e, int theta) : eps)
3         w(e) += theta
4         newdist = d[t(e)] + w(e);
5         if (newdist < d[h(e)])
6             q.enqueue({h(e), (t(e), newdist, newdist-olddist)});
```

Listing 4.7: Ball-and-String-Algorithmus: Initialisierungsroutine - Verminderung

Verglichen mit dem Fall der Gewichtserhöhung ist es bei der Verminderung nicht möglich, die Menge der betroffenen Knoten zum Zeitpunkt der Initialisierung zu bestimmen. Die Routine überprüft lediglich, ob die Änderung des Gewichts zu einer Distanzverbesserung des Endknotens der betroffenen Kante führt. Ist dies der Fall, so wird ein Eintrag für diesen erstellt und in die Warteschlange hinzugefügt.

Hauptphase

Nach der Initialisierungsphase folgt die Hauptphase in Form einer Schleife über die Einträge der Warteschlange, welche für alle Arten von Gewichtsveränderungen gleich vorgeht.

```
1     // Hilfsmenge N
2     N = {};
3     // Hauptschleife
4     while (q.size != 0)
5         {v, (p_new, d_new, delta)} = q.extracmin();
6         // neue Elternbeziehung setzen
7         spc(p_new) += v;
8         spc(p[v]) -= v;
9         spp(v) = p_new;
10        N = des(v);
11        // Schleife ueber alle Nachfolger in T
12        for (Vertex w : N)
13            d[w] += delta;
14            status(w) = anchored;
15            if (q.contains(w))
16                {w, (p_new2, d_new2, delta2)} = q.peek(w);
17                if (p_new != p_new2)
18                    q.remove(w);
19        // Schleife ueber aus N ausgehende Kanten
20        for (Edge e : Out(N))
21            olddist = d[h(e)];
22            newdist = d[t(e)] + w(e);
23            if ((olddist > newdist) || (status(h(e)) == floating))
24                q.enqueue({h(e),
25                    (t(e), newdist, newdist-olddist)});
```

Listing 4.8: Ball-and-String-Algorithmus: Hauptphase

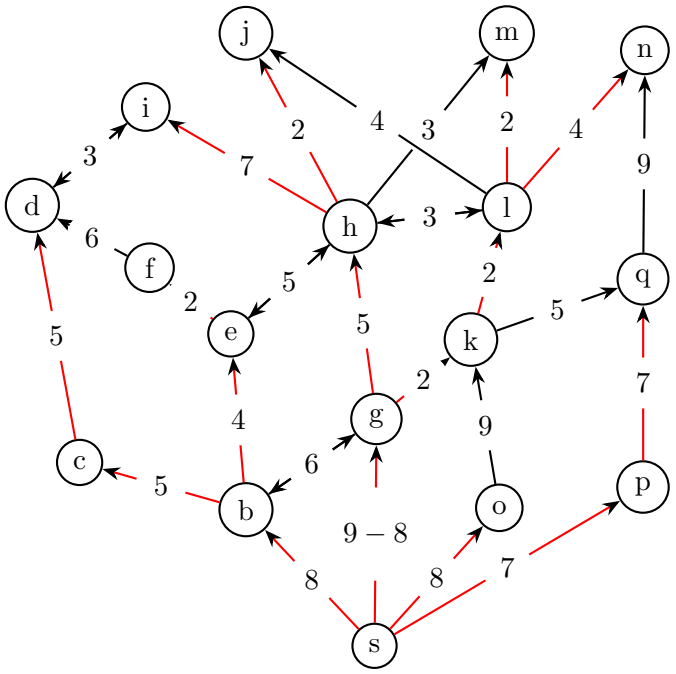
In der Hauptphase des Algorithmus wird in jedem Schleifendurchlauf der Eintrag, welcher den kleinsten Schlüssel aufweist, aus der Warteschlange q extrahiert und in die Elternbeziehungen des enthaltenen Knotens aktualisiert. Die Nachfolger des Knotens werden ermittelt, ihre Distanzen mit δ verrechnet und ihr Status auf *anchored* gesetzt. Außerdem wird für jeden Nachfolger überprüft, ob ein ihm zugehöriger Eintrag in q enthalten ist. Sofern dies der Fall ist und der entsprechende Eintrag einen anderen Elternknoten enthält, wird dieser entfernt. Dies geschieht, da der Eintrag kein kleineres δ aufweisen kann. Anderenfalls wäre er bereits extrahiert worden. Der Eintrag würde die Distanz auf einem anderen Pfad entweder um einen geringeren Wert vermindern (Gewichtsverminderung) oder um einen größeren Wert erhöhen (Gewichtserhöhung) und ist somit nicht optimal.

Abschließend werden die aus der Menge der Nachfolger ausgehenden Kanten bestimmt und für sie überprüft, ob mindestens eine der folgenden Eigenschaften gilt:

- die Distanz des Endknotens kann durch die Verwendung der Kante verringert werden, oder
- der Status des Endknotens ist *floating*, der bestehende Distanzwert ist somit nicht aktuell und muss neu berechnet werden.

Ist mindestens eine der Eigenschaften erfüllt, so wird ein neuer Eintrag für den Endknoten $h(e)$ der Kante e erstellt.

Beispiel



v	$d[v]$
s	0
b	8
c	13
d	18
e	12
f	14
g	9
h	14
i	21
j	16
k	11
l	13
m	15
n	17
o	8
p	7
q	14

Abbildung 4.12.: Beispielgraph G (Ball-and-String–Algorithmus - Ausgangsgraph)

q	descendants	q-Eintrag gelöscht	Out(desc)	Einträge für q
{g,(s,1,-8)}	{g, h, i, j, k, l, m, n}		{(g,b), (h,e), (i,d), (k,q)}	{q,(k,8,-6)}, {d,(i,16,-2)}, {b,(g,7,-1)}, {e,(h,11,-1)}
{q,(k,8,-6)}, {d,(i,16,-2)}, {b,(g,7,-1)}, {e,(h,11,-1)}	{q}		{(q,n)}	
{d,(i,16,-2)}, {b,(g,7,-1)}, {e,(h,11,-1)}	{d}		{(d,i)}	
{b,(g,7,-1)}, {e,(h,11,-1)}	{b, c, e, f}	{e,(h,11,-1)}	{(b,g), (c,d), (e,h), (f,d)}	

Tabelle 4.7.: Ablauf Hauptschleife - Ball-and-String–Algorithmus

Die Warteschlange q enthält nach der Initialisierungsphase lediglich den Eintrag $\{g, (s, 1, -8)\}$. Tabelle 4.7 veranschaulicht die Iterationen der Hauptphase. Zunächst wird der einzige Eintrag aus q extrahiert und der enthaltene Knoten g aktualisiert. Die Funktion $des(g)$ ermittelt als Nachfolger die Menge $\{g, h, u, j, k, l, m, n\}$, für welche die Distanzwerte aktualisiert werden. Da q zu diesem Zeitpunkt keine Einträge enthält, erfolgt kein Löschvorgang. Die aus der Menge der Nachfolger ausgehenden Kanten werden ermittelt. Bei potentiellen Distanzverbesserungen für die Endknoten der Kanten werden neue Einträge erstellt und in q eingefügt. In der ersten Iteration geschieht dies für die Knoten q, d, b und e, welche in der Warteschlange nach den Differenzen zur alten Distanz sortiert werden. Dadurch wird im nächsten Durchlauf der Knoten q verarbeitet. In der vierten Iteration wird der Knoten e als Nachfolger von b verarbeitet. Es existiert jedoch ein Eintrag für diesen Knoten in q, welcher infolgedessen gelöscht wird, da die selbe Distanzverbesserung bereits erreicht werden konnte. Die Hauptphase terminiert nach dieser Iteration, der resultierende SPT ist äquivalent zu dem des Beispiels der zweiten Phase der Anpassung (Abbildung 4.10).

Korrektheit und Laufzeit

Der Korrektheitsbeweis des Algorithmus kann [NA01] entnommen werden. Gegeben sei ein Graph $G = (V, E)$ mit $n = |V|$ Knoten und $m = |E|$ Kanten, sowie ein SPT für den Ausgangsgraphen und eine Menge an Kantenänderungen ϵ. Ferner sei δ_c die Anzahl der echten Teilbäume im alten SPT, welche mindestens einen inneren Knoten enthalten. δ_n sei die Anzahl der Knoten innerhalb der Teilbäume und δ_e die Anzahl der Kanten, welche in diesen Knoten starten. Die Laufzeit des Ball-and-String–Algorithmus in Abhängigkeit der Warteschlangenimplementierung beträgt

- Linearen Liste: $\mathcal{O}(\delta_e + \delta_c * \delta_n)$,
- Binären Halde: $\mathcal{O}(\delta_e * \lg(\delta_n))$,
- Fibonacci Halde: $\mathcal{O}(\delta_e + \delta_n * \lg(\delta_n))$.

Der Beweis der Laufzeiten kann [NA01] entnommen werden.

5

Volldynamische Algorithmen

Dieses Kapitel beschäftigt sich mit einem volldynamischen Algorithmus zur Lösung des DSP. Volldynamische Algorithmen zeichnen sich dadurch aus, dass sie im Unterschied zu semidynamischen Algorithmen in einem Durchlauf eine Menge Änderungen beiden Typs, also sowohl Erhöhungen als auch Verminderungen, verarbeiten können.

Abschnitt 5.1 beschreibt die theoretische Herleitung, welche dem volldynamischen SWSF-FP-Algorithmus zugrunde liegt. Dieser wird im abschließenden in Abschnitt 5.2 vorgestellt und diskutiert.

5.1. Herleitung

Dieser Abschnitt befasst sich mit dem dynamischen Strict Weakly Superior Function – Fixed Point (SWSF-FP)-Algorithmus, welchen Ganesan Ramalingam und Thomas Reps in [RR92] vorgestellt haben. Der Algorithmus basiert auf der von Donald Knuth in [DK76] definierten Generalisierung des Algorithmus von Dijkstra und löst das $\text{DSP}_{>0}$. Darin wird das $\text{SSSP}_{\geq 0}$ mithilfe einer kontextfreien Grammatik beschrieben.

Diese verwendet eine überliegende, monoton steigende Funktion (*superior function*)

$$g : R_+^k \to R_+$$

mit der Eigenschaft, dass

$$\forall x_1..x_k : g(x_1, .., x_k) \geq max(x_1, .., x_k) \tag{5.1}$$

gilt. $R_+ = \{R^+ \cup \infty\}$ enthält die Menge der positiven reellen Zahlen (inklusive 0) und positiv ∞.

Beispiele für eine solche *superior function* mit 2 Parametern $k = 2$:

- $g(x, y) = max(x, y)$
- $g(x, y) = x + y$
- $g(x, y) = max(1, x) * max(1, y)$

Die Funktionskomposition (theoretisch auch unterschiedlicher *superior functions*) kann genutzt werden, um eine *superior function* mit weiteren Parametern zu erstellen. Beispiele für $k = 3$:

- $g(x, y, z) = g(x, g(y, z)) = max(x, max(y, z))$
- $g(x, y, z) = g(x, g(y, z)) = x + (y + z)$
- $g(x, y, z) = g(x, g(y, z)) = max(1, x) * (max(1, y) * max(1, z))$

Alle Produktionen der Grammatik haben die Form

$$Y \to g(X_1, .., X_k)$$

$Y, X_1, .., X_k$ sind Nichtterminalsymbole und g, Klammern und Kommas sind Terminalsymbole. Ist $k = 0$, so wird das Nichtterminalsymbol direkt auf eine nichtnegative Zahl abgebildet.

Beispiel für eine solche Grammatik:

- Menge der Nichtterminalsymbole: $\{A, B, C\}$
- Menge der Terminalsymbole: $\{(,), ', ', a, b, c, d, e, f\}$

- Produktionen:
  - $A \to a$, $A \to b(B, C)$
  - $B \to c(a)$, $B \to d(A, C, A)$
  - $C \to e$, $C \to f(B, A)$
- überliegende Funktionen:
  - $a = 4$; $b(x, y) = max(x, y)$; $c(x) = x + 1$
  - $d(x, y, z) = x + max(y, z)$; $e = 9$; $f(x, y) = 0.5 * (x + y + max(x, y))$

Die Menge der Ketten von Terminalsymbolen für ein Nichtterminalsymbol Y über dem Terminalalphabet T wird durch

$$L(Y) = \{\alpha | \alpha \in T^* \wedge Y \to^* \alpha\}$$

beschrieben. Jede dieser Ketten repräsentiert eine verkettete Ausführung der überliegenden Funktion g, sodass sie ausgewertet einen reelen Wert $val(\alpha)$ besitzt.

Für die Beispielgrammatik und das darin enthaltene Nichtterminalsymbol A hat die Menge der Ketten beispielsweise folgenden Wert:
$L(A) = \{a, b(c(a), e), b(c(a), f(c(a), a)), ...\} \Rightarrow \{4, 9, 7, ...\}$

Für jedes Nichtterminalsymbol Y ist der kleinste Wert folgendermaßen definiert:

$$m(Y) = min\{val(\alpha) | \alpha \in L(Y)\}$$

Im zuvor beschriebenen Beispiel gilt $m(A) = min\{4, 9, 7...\}$.

Für die Anwendung des Algorithmus von Dijkstra wird nun für jeden Knoten v_i des Graphen G ein Nichtterminalsymbol C_i, sowie eine Produktion

$$C_i \to g_{ij}(C_j)$$

mit der entsprechenden überliegenden Funktion

$$g_{ij}(x) = d_{ij} + x$$

eingeführt. d_{ij} beschreibt das Gewicht der Kanten $(v_i, v_j) \in E(G)$. Eine weitere Produktion für den Startknoten s bzw. das entsprechende Nichtterminalsymbol C_0 wird hinzugefügt

$$C_0 \to 0$$

In dieser Anwendung beschreibt $L(Y)$ die Pfade des Graphen vom Startknoten zu einem durch Y repräsentierten Knoten und $m(Y)$ die Distanz des kürzesten Pfads.

Der generalisierte Algorithmus berechnet $m(Y)$ für alle Nichtterminalsymbole Y und speichert diese $\mu[Y]$. Die Menge D enthält Knoten, für die $\mu[Y]$ berechnet wurde. Vorläufige Ergebnisse von Distanzberechnungen werden in $\nu[Y]$ gespeichert. Der Ablauf des Algorithmus:

1. $D = \{\}$
2. Stop, wenn $\forall Y : Y \in D$
3. $\forall Y \notin D : \nu[Y] =$
   $min\{g(\mu[X_1], .., \mu[X_k]) | Y \to g(X_1, .., X_k)$ ist eine Produktion, $\{X_1, .., X_k\} \subseteq D\}$
4. Suche Minimum aus $\nu[Y], Y \notin D$ und setze $\mu[Y] = \nu[Y]$
5. Füge Y in D ein

Der SWSF-FP-Algorithmus stellt weitere Anforderungen an die Eigenschaften der überliegenden Funktion. Diese muss strikt (*strict*) sein, sodass die Ungleichung (5.1) echt größer ist. Des Weiteren soll für jedes $i \in [1, k]$

$$g(x_1, .., x_i, .., x_k) \leq x_i \to g(x_1, .., x_i, .., x_k) = g(x_1, .., \infty, .., x_k)$$

gelten (*weakly*).

Sei Q eine Menge von k Gleichungen für k Unbekannte $x_1, .., x_k$ und die i-te Gleichung wird beschrieben als

$$x_i = g_i(x_1, .., x_k)$$

Erfüllt die überliegende Funktion g zuvor beschriebenen Eigenschaften, so wird die Gleichung als SWSF-Gleichung bezeichnet. Es kann gezeigt werden, dass jede Gleichung dieser Art einen maximalen Fixpunkt besitzt. Ziel des SWSF-FP-Problems ist es, diese für alle Gleichungen zu bestimmen. Das $SSSP_{>0}$ ist ein Spezialfall des SWSF-FP. Die Einschränkung auf positive Kantengewichte resultiert aus der Forderung der Strikt-

heitseigenschaft an die Funktion. Die detaillierte Herleitung des Problems kann [RR92] entnommen werden.

5.2. Dynamic SWSF-FP-Algorithmus

Der Algorithmus zur Lösung des dynamischen SWSF-FP-Problems führt für jeden Knoten des Graphen einen Kosistenzzustand ein. Die Auswertung der SWSF-Gleichung $g_i(d[v_1], .., d[v_k])$ wird als $rhs(x_i)$ (*right hand side*) bezeichnet. In der speziellen Anwendung des DSP-Problem ist diese folgendermaßen definiert:

$$rhs(v) = \begin{cases} min_{(u,v)\in E}\{d[u] + w(u,v)\} & v \neq s \\ 0 & v = s \end{cases}$$

Ist die für den Knoten v_i gespeicherte Distanz $d[x_i] = rhs(x_i)$, so ist der Knoten konsistent. Ist die Distanz größer bzw. kleiner, so wird er als über- bzw. unterkonsistent bezeichnet. Der Unterschied zwischen dem dynamischen SWSF-FP-Algorithmus und dem Algorithmus von Dijkstra liegt darin, dass der Algorithmus von Dijkstra lediglich konsistente und überkonsistente Knoten verarbeiten muss: Nach der Initialisierungsphase gilt für jeden Knoten $v_i, i \neq 0$: $d[v_i] = \infty$. In der Hauptphase werden die Knoten iterativ in einen konsistenten Zustand gebracht, indem die kürzesten Distanzen mittels der überliegenden Funktion bestimmt werden. Im dynamischen Fall kann es hingegen geschehen, dass das Gewicht einer Kante erhöht wird und in Folge dessen $d[x_i] < rhs(x_i)$ ist. Der SWSF-FP-Algorithmus verwendet zur Verarbeitung inkonsistenter Knoten eine Vorrangwarteschlange q. Der Schlüssel eines Eintrags für einen Knoten v ist bestimmt durch $min(d[v], rhs(v))$.

```
1  Void DynamicSWSF-FP(G, T, w, eps)
2      // Warteschlange q
3      q = 0;
4      // Initialisierung
5      for ((Edge e, int theta) : eps)
6          w(e) += theta;
7          if (theta > 0)
8              if (t(e) == spp(h(e)))
9                  if (d[h(e)] != rhs(h(e)))
10                     q.enqueue(h(e), min(d[h(e)], rhs(h(e))));
11         if (theta < 0)
12             if (d[h(e)] > rhs(h(e)))
13                 q.enqueue(h(e), d[t(e)] + w(e))
14     // Hauptphase
15     while (q.size != 0)
16         v = q.extractmin();
17         // v ist ueberkonsistent
18         if (rhs(v) < d[v])
19             d[v] = rhs(v);
20             p[v] = rhsParent(v);
21             for (Vertex u : Out(v))
22                 if (rhs(u) != d[u])
23                     q.enqueue(u, min(rhs(u), d[u]));
24                 else
25                     q.remove(u);
26         // v ist unterkonsistent
27         else
28             d[v] = inf;
29             for (Vertex u : (Out(v) + v))
30                 if (rhs(u) != d[u])
31                     q.enqueue(u, min(rhs(u), d[u]));
32             else
33                     q.remove(u);
```

Listing 5.1: Dynamischer SWSF-FP-Algorithmus

Der Algorithmus kann in einer Initialisierungsphase und eine Hauptphase unterteilt werden. In der Initialisierungsphase werden für die Endpunkte $h(e)$ der Kanten, deren Gewichte erhöht wurden und die sich im SPT befinden, ein aktueller rhs-Wert bestimmt. Wenn dieser sich von $d[h(e)]$ unterscheidet, so wird ein Eintrag in die Warteschlange q

eingefügt. Für die Endpunkte der Kanten, deren Gewichte vermindert wurden, werden Einträge in q eingefügt, wenn die Gewichtsreduzierung zu einer Verbesserung der Distanz des Endknoten führt.

In der Hauptphase unterscheidet der Algorithmus in seiner Verarbeitung, ob ein extrahierter Knoten über- oder unterkonsistent ist. Ist ein Knoten überkonsistent, so wird der Distanzwert des Knotens mit dem rhs-Wert belegt, der Wert für den Elternknoten aktualisiert (im Pseudocode durch die Funktion *rhsParent(v)* ermittelt), und bestimmt, ob diese Verbesserung Auswirkungen auf die nachfolgenden Knoten im Graphen hat. Ist dies der Fall, so wird für diese ein Eintrag in q hinzugefügt. Ist der rhs-Wert des Nachfolgers gleich der bereits gespeicherten Distanz, so wird ein möglicherweise in der Warteschlange enthaltener Eintrag gelöscht. Ist ein Knoten hingegen unterkonsistent, so wird dieser künstlich überkonsistent gemacht, indem die gespeicherte Distanz auf den Maximalwert erhöht wird. Für diesen Knoten und dessen Nachfolger werden neue rhs-Werte bestimmt. Unterscheiden sich diese von der gespeicherten Distanz, so werden neue Einträge für die entsprechenden Knoten erstellt, anderenfalls werden möglicherweise vorhandene Einträge gelöscht. Der Algorithmus terminiert, wenn die Warteschlange q keine weiteren Einträge enthält.

Beispiel

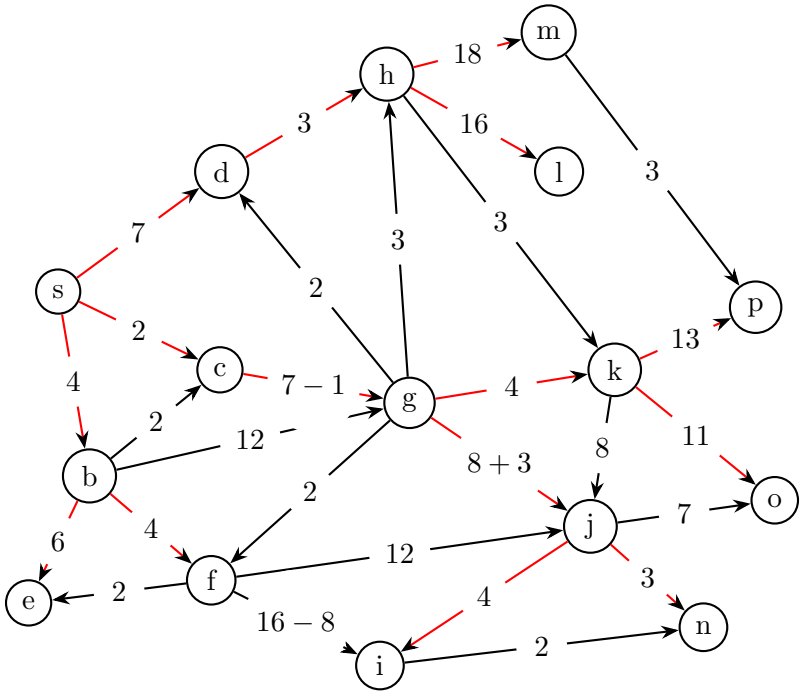


Abbildung 5.1.: Beispielgraph G (dynamischer SWSF-FP-Algorithmus)

#	e, θ	$d[h(e)]$	$rhs(h(e))$	Einträge für q
1	((c,g),-1)	9	$min(8, 16)$ =8	{g, 8}
2	((g,j),3))	17	20	{j, 17}
3	((f,i),-8))	21	16	{i, 16}

Tabelle 5.1.: Ablauf Initialisierungsroutine - dynamischer SWSF-FP-Algorithmus

Tabelle 5.1 verdeutlicht die Arbeitsweise der Initialisierungsphase des SWSF-FP-Algorithmus am Beispiel des Graphen aus Abbildung 5.1. Das Gewicht der Kante (c, g) soll um eine Einheit vermindert werden. Dadurch ergibt sich der rhs-Wert 8 für den Knoten g bei Traversierung des Knoten c. Da dieser kleiner als die ursprüngliche Distanz $d[g] = 9$ ist, wird ein Eintrag mit dem Schlüssel $min(8, 9) = 8$ in die Warteschlange q eingefügt. Das Gewicht der Kante (g, j) wird um 3 Einheiten erhöht. Der rhs-Wert beträgt für den Endknoten bei Verwendung des ursprünglichen Pfads 20. Ein Eintrag mit dem Schlüssel

$min(20, 17) = 17$ wird erstellt. Entsprechend wird ein Eintrag $\{i, 16\}$ in der dritten Iteration hinzugefügt.

q	$rhs(v) < d[v]$	Out(desc)	q-Eintrag gelöscht	Einträge für q
{g,8}, {i,16}, {j,17}	true	{d,f,j,k,h}	n.V.	{j,17}*, {k,12}
{k, 12}, {i,16}, {j,17}	true	{o,p}		{o,23}, {p,25}
{i,16}, {j,17}, {o,23}, {p,25}	true	{n}		{n,18}
{j,17}, {n,18}, {o,23}, {p,25}	false	{i,n,o}	n.V.	{j, 17}, {n,18}* {o,23}*
{j,17}, {n,18}, {o,23}, {p,25}	true	{i,n,o}	n.V.	{n,18}* {o,23}*
{n,18}, {o,23}, {p,25}	true	{}		
{o,23}, {p,25}	true	{}		
{p,25}	true	{}		

Tabelle 5.2.: Ablauf Hauptschleife - dynamischer SWSF-FP-Algorithmus

Tabelle 5.2 verdeutlicht den Ablauf der Hauptschleife des Algorithmus für das Beispiel. In der ersten Iteration wird der Eintrag $\{g, 8\}$ extrahiert. Der Knoten g ist somit überkonsistent, der Distanzwert wird mit dem Wert 8 überschrieben. Für die Nachfolger d, f und h sind keine Einträge in q enthalten, sodass diese auch nicht gelöscht werden. Für die Knoten j und k werden neue Einträge erstellt, wobei ein äquivalenter Eintrag für j bereits vorhanden ist. In der zweiten und dritten Iteration wird der Vorgang für die Knoten k und i wiederholt. In der vierten Iteration wird der Knoten j extrahiert. Dieser ist unterkonsistent, da der Distanzwert kleiner als der rhs-Wert ist ($17 < 19$). Der Distanzwert wird auf ∞ erhöht und ein Eintrag für j in q hinzugefügt. Mögliche Einträge für n und o sind bereits mit dem gleichen Schlüssel vorhanden. Die restlichen Iterationen verarbeiten überkonsistente Knoten nach dem beschriebenen Prinzip aus Iteration 1. Die Hauptschleife terminiert nach 8 Iterationen, das Ergebnis kann Abbildung 5.2 entnommen werden.

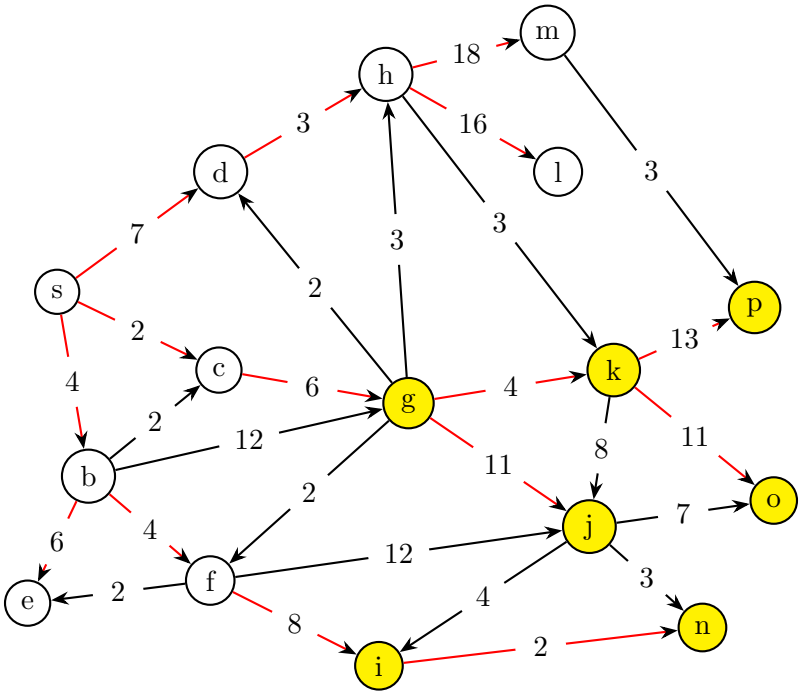


Abbildung 5.2.: Beispielgraph G (dynamischer SWSF-FP-Algorithmus - Lösung)

6

Prototypische Implementierung

Die bisher erfolgten Betrachtungen auf die Algorithmen zur Bestimmung der kürzesten Pfade und Distanzen auf Graphen, welche sich dynamisch verändern, sind theoretischer Natur. Ziel dieser Arbeit ist es jedoch auch, die Funktionalität der Algorithmen zu demonstrieren und ihre praktische Verwendbarkeit mithilfe von Laufzeitmessungen zu bewerten. Hierzu wurde eine prototypische Implementierung vorgenommen.

Es soll eine Anwendung entwickelt werden, welche das DSP-Problem mit sämtlichen der in Kapitel 3, 4 und 5 vorgestellten Algorithmen lösen kann. Die Anwendung soll über eine grafische Benutzeroberfläche steuerbar sein, in welcher der Benutzer einerseits die Möglichkeit hat, sich die Arbeitsweise der Algorithmen an konkreten Beispielen demonstrieren zu lassen. Der Benutzer kann einen der beschriebenen Algorithmen, sowie Parameter für einen Graphen (Anzahl der Knoten und Kanten) und die Anzahl der auf den Graphen auszuführenden Änderungen festlegen. Mithilfe dieser Parameter erstellt das Programm einen zufällig generierten Graphen und eine Menge von Änderungen, welche auf diesem ausgeführt werden sollen. Hat der Benutzer einen statischen Algorithmus ausgewählt, so soll zunächst der generierte Graph visualisiert werden. Ist der ausgewählte Algorithmus nicht statisch, so soll zusätzlich der SPT des Ausgangsgraphen berechnet und

innerhalb der Visualisierung kenntlich gemacht werden. Anschließend soll der neue SPT für den Graphen unter Berücksichtigung der Menge der Änderungen mithilfe des Algorithmus bestimmt und Änderungen des SPT deutlich gemacht werden. Zusätzlich sollen wichtige Ereignisse bei der Ausführung des Algorithmus in textueller Form protokolliert und dem Benutzer präsentiert werden. Andererseits soll der Benutzer die Möglichkeit haben, Laufzeitmessungen auf den Algorithmen auszuführen. Der Benutzer kann hierzu Parameter für die verwendeten Graphen, die Anzahl der Änderungen und Durchläufe, sowie die Art der Änderungen (Erhöhungen, Verminderungen oder gemischte Änderungen) bestimmen. Mithilfe der Parameter soll das Programm für jeden Durchlauf einen Graphen, sowie eine Menge von Änderungen der ausgewählten Art erstellen, und Lösungen mithilfe der Algorithmen bestimmen. Die durchschnittlichen Ausführungszeiten der Algorithmen sollen bestimmt und dem Benutzer in textueller Form präsentiert werden.

Wenngleich Laufzeitmessungen mithilfe des Programms möglich sind, handelt es sich um prototypische Implementierungen der verwendeten Algorithmen, welche noch weiter verbessert werden können. Insbesondere die Implementierung des volldynamischen SWSF-FP-Algorithmus kann noch weiter optimiert werden. Ansätze zur Optimierung können beispielsweise [RB09] entnommen werden. Ziel dieser Implementierung ist es somit nicht, das Laufzeitverhalten der Algorithmen abschließend zu bewerten. Vielmehr soll dem Benutzer durch die Implementierung die Arbeitsweise der Algorithmen verdeutlicht und die Laufzeitmessungen als Indiz für die Anwendbarkeit der Algorithmen wahrgenommen werden. Die Implementierung kann ebenso als Grundlage für weitere Arbeiten auf dem Gebiet der Bestimmung der kürzesten Pfade in dynamischen Graphen dienen.

In Abschnitt 6.1 wird zunächst die verwendete Entwicklungskonfiguration vorgestellt, gefolgt von einer Beschreibung der implementierten Komponenten des Prototyps in Abschnitt 6.2. Abschließend wird die Funktionsweise der entstandene Anwendung anhand kleiner Beispiele in Abschnitt 6.3 demonstriert.

6.1. Entwicklungskonfiguration

Im Folgenden werden kurz die eingesetzte Programmiersprache und die verwendeten Bibliotheken beschrieben. Die Implementierung wurde mittels der objektorientierten Programmiersprache Java unter der Verwendung der Version 6 (Java SE 6) durchgeführt. Die Programmiersprache Java wurde vor allem gewählt, da eine große Anzahl vielseitiger, freier Bibliotheken existiert, welche unterstützend zur Entwicklung des Prototypen genutzt

werden können. Die verwendeten Bibliotheken werden folgend kurz benannt und ihre Verwendungsmöglichkeiten beschrieben.

GraphStream

GraphStream ist eine Java – Bibliothek zur Modellierung und Analyse von dynamischen Graphen. Im Rahmen dieser Arbeit wurde die Bibliothek zur Visualisierung der betrachteten Graphen und der in ihnen enthaltenen SPTs verwendet. Die Bibliothek bietet insbesondere die Möglichkeit, die Lage der Knoten des Graphen automatisch so zu bestimmen, dass der Graph möglichst lesbar ist. Weitere Informationen und eine Möglichkeit zum Download stehen unter [GS] zur Verfügung.

Swing

Swing ist eine Java – Bibliothek zur Erstellung von grafischen Benutzeroberflächen, welche im Rahmen dieser Arbeit genutzt wurde. Weitere Informationen stehen unter [SW] zur Verfügung.

JUnit

JUnit ist ein Framework zum Testen von Programmen. Im Rahmen dieser Arbeit wurde es für das Testen der Funktionalität der Algorithmen verwendet. Weitere Informationen und eine Möglichkeit zum Download stehen unter [JU] zur Verfügung.

Die Implementierung wurde unter Verwendung von *Eclipse* ([EC]) als Entwicklungsumgebung durchgeführt.

6.2. Implementierung der Komponenten

Dieser Abschnitt beschreibt die Implementierung aller Komponenten des Prototypen.

Knoten und Kanten

Knoten werden durch die Klasse *Vertex* dargestellt. Knoten verfügen über einen Integer-Schlüssel *id*.

Kanten sind Instanzen der Klasse *Edge*. Sie verfügen neben dem Integer-Schlüssel *id* über die Attribute *head* und *tail*, welche die Anfangs- und Endknoten der Kante enthalten.

Graphen

Graphen werden innerhalb des Prototypen durch Instanzen der Klasse *IntMatrixGraph* repräsentiert.

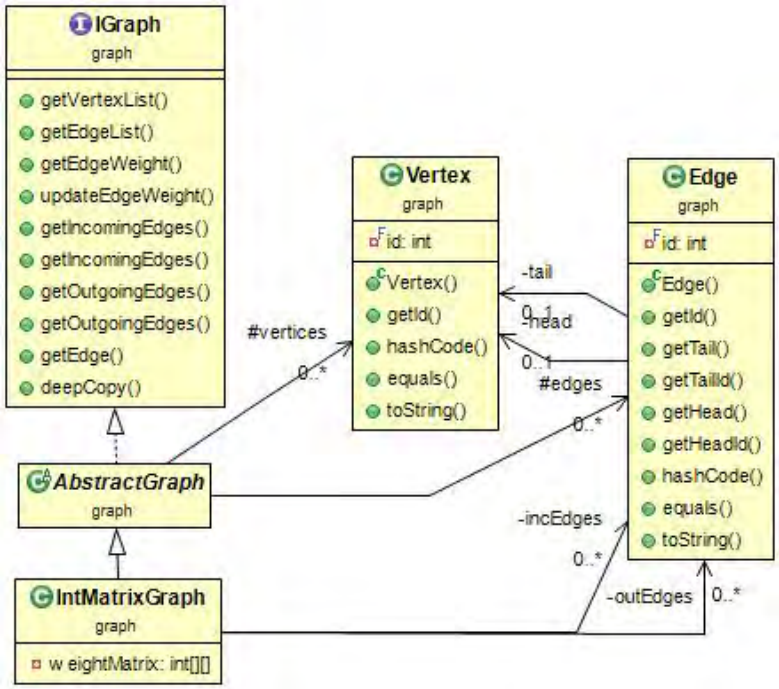


Abbildung 6.1.: Klassendiagramm Graphenmodellierung

Diese erhält bei der Erzeugung jeweils eine Liste von Knoten und Kanten, sowie eine Map, welche jeder Kante der Liste ein Integer-Gewicht zuweist. Die interne Repräsentation der Daten wird mithilfe einer Adjazenzmatrix realisiert, welche für jeden Knoten der Liste eine Zeile bzw. Spalte enthält und als Werte die Gewichte der Kanten zwischen den Knoten enthält. Existiert keine Kante zwischen zwei Knoten, so ist der Wert an dieser Stelle der maximale Wert des Datentyps Integer. Zudem werden für jeden Knoten Listen mit ein-

und ausgehenden Kanten gespeichert. Der Zugriff auf die Daten des Graphen ist durch die Schnittstelle *IGraph* beschrieben, welche jede konkrete Graphenklasse implementieren muss.

Das Klassendiagramm in Abbildung 6.1 verdeutlicht die Interaktion der Komponenten und beschreibt die Zugriffsmöglichkeiten auf diese.

SPT

SPT's werden durch die Klasse *SPT* dargestellt. Ein SPT wird entgegen der Namensgebung intern mithilfe von Hashtabellen realisiert. Hashtabellen eignen sich besonders aufgrund der effizienten Implementierbarkeit der Operationen zur Suche, zum Einfügen und zum Löschen von Elementen, welche für diese Anwendung besonders relevant sind. Ein SPT speichert für jeden Knoten die Eltern- und eine Liste von Nachfolgeknoten, sowie die kürzeste Distanz.

Warteschlangen

Zugriffe auf Warteschlangen werden über die Schnittstelle *IQueue* definiert.

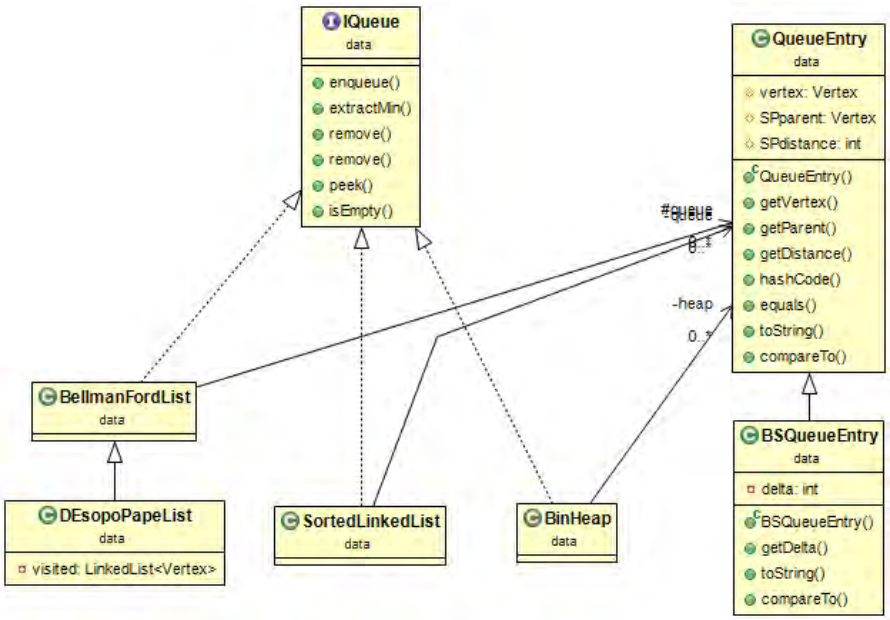


Abbildung 6.2.: Klassendiagramm Warteschlangenmodellierung

Innerhalb des Prototypen existieren vier verschiedene Implementierungen. Der Algorithmus von Bellman und Ford verwendet die *BellmanFordList*, welche mit Hilfe einer Liste nach dem FIFO-Prinzip arbeitet. Der Algorithmus von D'Esopo-Pape arbeitet mit der *DEsopoPapeList*, welche nach dem in Kapitel 3.2 beschrieben Prinzip agiert. Die restlichen Algorithmen arbeiten mit sortierten Warteschlangen. Diese werden realisiert durch Instanzen der Klassen *SortedLinkedList*, welche die Sortierung innerhalb einer Liste vornimmt, und *BinHeap*, welche die Sortierung mittels einer binären Halde implementiert. Einträge der Warteschlange sind Instanzen der Klasse *QueueEntry* bzw. *BSQueueEntry*. Diese verfügen über Knoten- und Elternattribute, sowie den Schlüsseleintrag Distanz. Der Ball–and–String-Algorithmus verwendet einen zusammengesetzten Schlüssel bestehend aus der Distanz und einem Wert δ. Einträge dieser Form werden durch Instanzen der Klasse *BSQueueEntry* repräsentiert, welche über ein zusätzliches Integer-Attribut δ (delta) verfügen.

Algorithmen

Algorithmen müssen die Schnittstelle *IAlgorithm* implementieren, welche Operationen zur Ausführung bereitstellt.

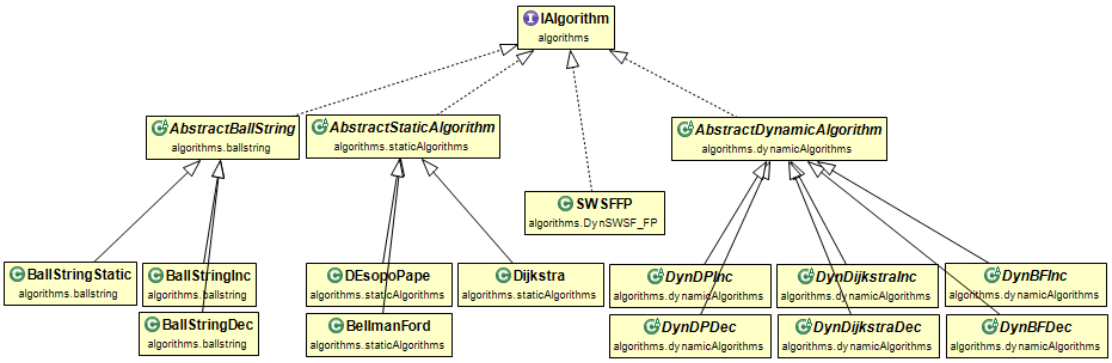


Abbildung 6.3.: Klassendiagramm Algorithmen

Das Klassendiagramm in Abbildung 6.3 zeigt die innerhalb des Prototyps implementierten Algorithmen. Klassen, die auf *Inc* enden, beschreiben Varianten der Algorithmen, welche das DSP-Problem für Erhöhungen der Kantengewichte lösen können. Analog beschreiben Klassen, welche auf *Dec* enden, Varianten von Algorithmen, die das DSP-Problem für Verminderungen der Kantengewichte lösen. Das statische Algorithmenframework (3.4) wurde in der abstrakten Klasse *AbstractStaticAlgorithm* implementiert, welche von den konkreten statischen Algorithmen beerbt wird. Analog wurde das dynamische Algorith-

menframework in der abstrakten Klasse *AbstractDynamicAlgorithm* implementiert. Die von ihr erbenden Klassen (z.B. *DynDijkstraInc*) sind ebenfalls abstrakt. Konkrete Implementierungen erben von diesen und unterscheiden sich in den zwei Stufen der Anpassung, sodass jede der abstrakten Klassen von zwei konkreten Klassen beerbt wird, welche aus Platzgründen nicht in der Abbildung enthalten sind. Des Weiteren sind Klassen für den Ball–and–String-Algorithmus und den dynamischen SWSF-FP Algorithmus vorhanden.

Hilfskomponenten

Dieser Abschnitt beschreibt Hilfskomponenten, welche im Rahmen der prototypischen Implementierung entstanden sind.

Graphgenerator

Die Generierung zufälliger Graphen wird in der Klasse *Graphgenerator* realisiert. Der Generator erwartet die Anzahl an Knoten und Kanten, sowie das maximale Gewicht einer Kante als Eingabeparameter und erzeugt einen zusammenhängenden Graphen, in dem zufällig Kanten mit ebenfalls zufällig gewähltem Gewicht eingefügt wurden. Der Generator ist ebenfalls für die Erstellung zufälliger Änderungen der Gewichte der Kanten unter Einhaltung der Konsistenzbedingungen zuständig.

Graphviewer

Die Visualisierung der Graphen und enthaltenen SPTs erfolgt mithilfe des Graphviewers, welcher in der gleichnamigen Klasse realisiert ist. Dieser konvertiert den Graphen in eine Form, welche die Graphenbibliothek *GraphStream* verarbeiten kann, und steuert die Anzeige.

TestUtility

Die TestUtility stellt Methoden zum automatisierten Testen der Algorithmen und vereinfachten Erstellung von Testszenarien bereit, welche unter anderem im Rahmen der Laufzeitmessung zum Einsatz kommen.

GUI

Die graphische Benutzeroberfläche wird in der Klasse *GUI* realisiert. Die Oberfläche ist in zwei Tabs unterteilt.

Im ersten Tab, dem Tab zur Visualisierung der Graphen und wichtiger Ereignisse der Algorithmenausführung, können Parameter bestimmt und Algorithmen mit diesen ausgeführt werden. Die Graphen und Änderungen werden mithilfe des Graphengenerators und des Graphenviewers erstellt und visualisiert. Die Algorithmen werden mit den Graphen und Änderungen ausgeführt und protokollieren dabei wichtige Ereignisse. Nach Abschluss der Berechnung des neuen SPT durch den Algorithmus wird die Visualisierung aktualisiert und das Ereignis-Protokoll ausgegeben.

Im zweiten Tab, dem Tab zur Ausführung von Zeitmessungen, können ebenfalls Parameter ausgewählt werden. Mithilfe der TestUtility (und des Graphengenerators) werden Testfälle erstellt. Die TestUtility bestimmt die Zeit, welche die Algorithmen zur Bearbeitung der Testfälle durchschnittlich benötigen. Diese wird schlussendlich durch die GUI ausgegeben.

6.3. Anwendungsdemonstration

Nachdem die einzelnen Bestandteile der prototypischen Implementierung vorgestellt wurden, dient dieser Abschnitt der Demonstration der Anwendung anhand von Beispielen. Hierbei stehen nicht die zuvor in der Theorie vermittelten Algorithmen im Vordergrund. Stattdessen soll die Funktionalität kurz vorgestellt und das Zusammenspiel der Komponenten näher erläutert werden.

Visualisierung

Dieser Abschnitt befasst sich mit der Funktionalität und den Arbeitsabläufen des ersten Tabs der grafischen Benutzeroberfläche.

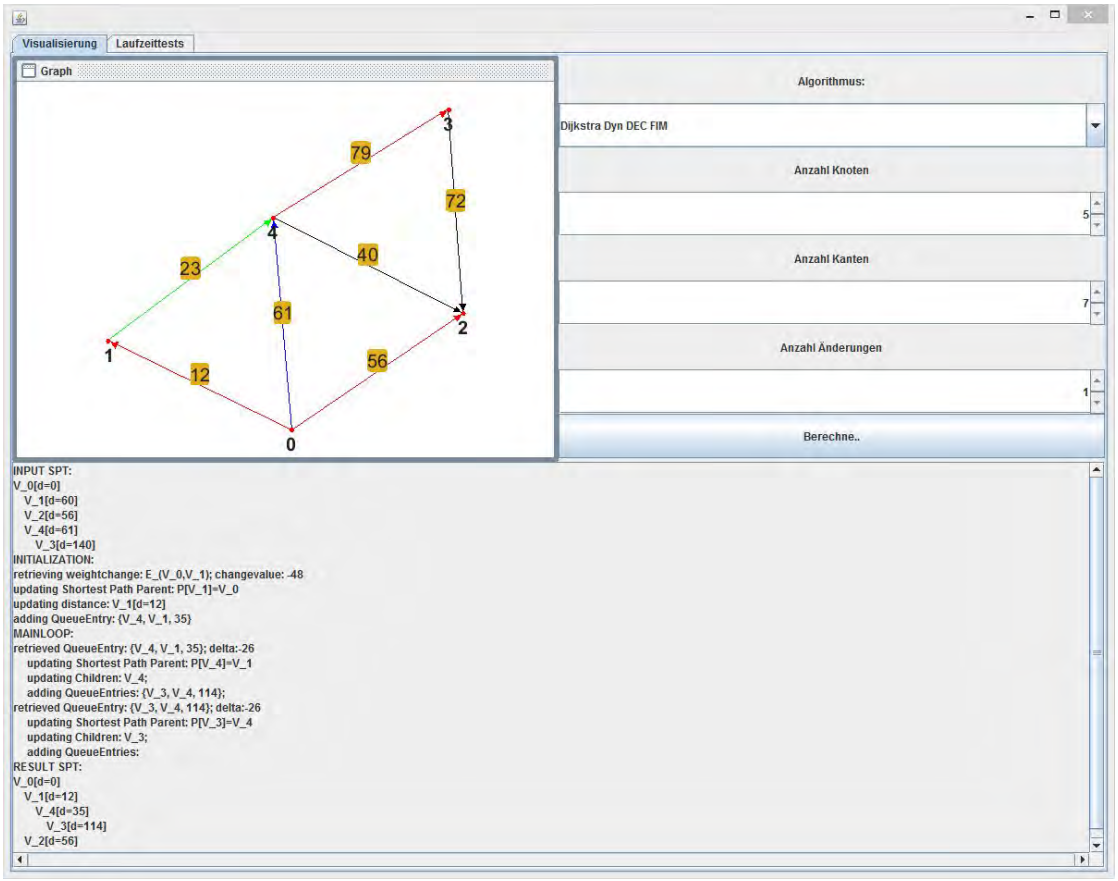


Abbildung 6.4.: Anwendungsdemonstration: 1. Tab

Abbildung 6.4 zeigt die Ergebnisausgabe nach der Ausführung eines einfachen Beispiels. Die Parametereingabe befindet sich in der oberen rechten Ecke: Der Benutzer wählt einen der vorgestellten Algorithmen, sowie die Anzahl der Knoten, Kanten und Änderungen aus. In diesem Beispiel wurde die dynamische Variante des Algorithmus von Dijkstra in erster Anpassungsstufe für die Verminderung von Kantengewichten ausgewählt. Ferner soll der Graph 5 Knoten und 7 Kanten enthalten und eine Gewichtsverminderung vorgenommen werden. Durch die Betätigung des ”Berechne”-Knopfs finden intern folgende Aktionen statt:

- Unter Verwendung eines Graphengenerators wird ein neuer Graph mit den Parametern für die Anzahl der Knoten und Kanten erzeugt.
- Der Graphengenerator wird ebenfalls zur Erstellung einer Kantenänderung verwendet. Die Art der Änderung (Verminderung oder Erhöhung) ist bestimmt durch die

Wahl des Algorithmus. Diese sind in unterschiedlichen Varianten je Änderungsart auswählbar.

- Wenn der Benutzer einen dynamischen Algorithmus ausgewählt hat, so wird ein SPT des erstellten Graphen benötigt. Dieser wird mithilfe eines beliebigen statischen Algorithmus bestimmt.
- Mithilfe des Graphviewers, welcher die Anzeige der linken oberen Ecke steuert, wird der Graph visualisiert. Bei der Nutzung eines dynamischen Algorithmus wird der zuvor berechnete SPT ebenfalls (rot) kenntlich gemacht.
- Eine Instanz des ausgewählten Algorithmus wird erzeugt und ausgeführt.
- Der durch den Algorithmus berechnete SPT wird gespeichert. Außerdem wird der Ereignis-Log, welcher wichtige Berechnungsschritte der Ausführung protokollierte, in der unteren Hälfte der Anzeige ausgegeben. Der Ereignis-Log enthält den Ein- und Ausgabe-SPT, sowie alle Operationen, welche die Warteschlange oder den SPT veränderten, in textueller Form und unterteilt die Ausführung in Initialisierungs- und Hauptphase.
- Der Titel des betätigten Knopfs wird in ”Zeige Ergebnis” geändert.

Durch die Betätigung des ”Zeige Ergebnis”-Knopfs wird der zuvor durch den Algorithmus berechnete und gespeicherte SPT an den Graphviewer übergeben. Dieser ist dafür zuständig, den SPT im Graphen kenntlich zu machen, sofern ein statischer Algorithmus ausgewählt wurde, oder dessen Anzeige im Falle eines dynamischen Algorithmus im Graphen zu aktualisieren. Hierzu werden Kanten, welche aus dem SPT entfernt wurden, blau markiert. Hinzugefügte Kanten hingegen werden in grüner Farbe angezeigt.

Laufzeittests

Dieser Abschnitt befasst sich mit der Funktionalität und den Arbeitsabläufen des zweiten Tabs der grafischen Benutzeroberfläche.

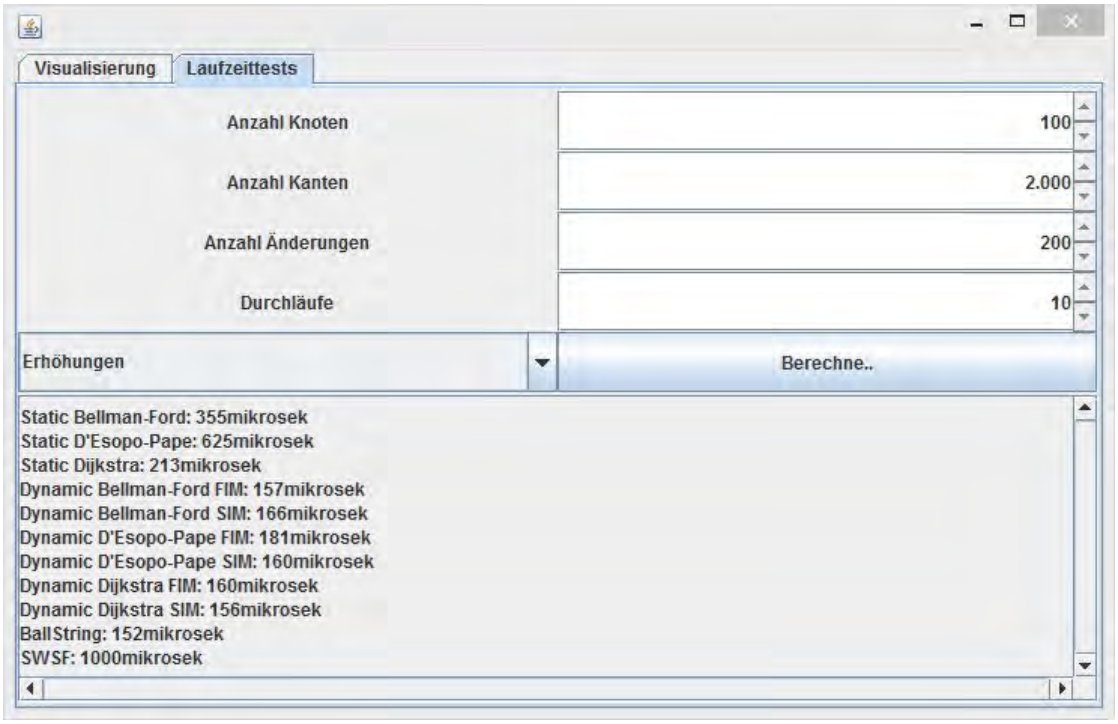


Abbildung 6.5.: Anwendungsdemonstration: 2. Tab

Abbildung 6.5 zeigt die Ergebnisausgabe nach der Ausführung eines beispielhaften Laufzeittests. Die Parametereingabe befindet sich innerhalb der oberen Hälfte des Fensters: Der Benutzer bestimmt Werte für die Anzahl der Knoten, Kanten, Änderungen und Durchläufe, sowie die Art der Änderungen. Durch die Betätigung des ”Berechne..”-Knopfs finden intern folgende Aktionen statt:

- Die Ausführung der Tests wird an die TestUtility delegiert.
- Diese erzeugt unter Verwendung des Graphengenerators für jeden Durchlauf einen Graphen mit der entsprechenden Anzahl an Knoten und Kanten.
- Die zu testenden Algorithmen ergeben sich aus dem Parameter Änderungsart. Sämtliche Algorithmen, die die Änderungsart verarbeiten können, werden getestet.
- Jeder dieser Algorithmen wird mit Hilfe der TestUtility und je einer Kopie der erzeugten Graphen getestet. Aus den Ausführungszeiten wird ein Mittelwert gebildet.
- Die Mittelwerte werden nach Abschluss aller Tests in textueller Form in der unteren Hälfte des Fensters ausgegeben.

7

Experimente

Dieses Kapitel befasst sich mit der experimentellen Evaluierung der beschriebenen Algorithmen. In Abschnitt 7.1 werden zu diesem Zweck künstliche, zufällig generierte Graphen verwendet. Mithilfe dieser Graphen wird die Arbeitsweise der Algorithmen für beliebige Eingaben verglichen. Abschnitt 7.2 hingegen verwendet Graphen, welche reale Straßennetze abbilden. Diese sind sehr speziell und zeichnen sich besonders dadurch aus, dass die Anzahl der Kanten im Graphen in Relation zur Anzahl der Knoten gering ist.

7.1. Zufällig generierte Graphen

Dieser Abschnitt beschäftigt sich mit Experimenten, bei denen die Algorithmen auf zufällig generierten Graphen arbeiten. In Unterabschnitt 7.1.1 werde Experimente anderer Autoren vorgestellt. Außerdem wurden unter Verwendung der prototypischen Implementierung eigene Experimente durchgeführt, welche in Unterabschnitt 7.1.2 vorgestellt werden. Im abschließenden Unterkapitel 7.1.3 werden die Ergebnisse beider Experimentreihen verglichen.

7.1.1. Experimente anderer Autoren

Paolo Narváez et al. untersuchen in [NA01] das Verhalten der einzelnen Algorithmen des angepassten Frameworks für dynamische Graphen (Kapitel 4.2) und des Ball–and–String-Algorithmus (Kapitel 4.3) im Hinblick auf die Anzahl der Vergleichsoperationen, welche ein Algorithmus zur Lösung des DSP-Problems benötigt.

Betrachtet werden zufällig generierte ungerichtete Graphen unterschiedlicher Größe, welche Netzwerke repräsentieren sollen. An jedem Knoten liegen durchschnittlich sieben Kanten an. Für jede betrachtete Netzwerkgröße ($|V| < 1500$) werden 40 Graphen generiert, auf denen jeweils 175 Kanten einzeln hinzugefügt und gelöscht werden sollen. Für jeden Algorithmus und jede Netzwerkgröße wird ein Mittelwert bezüglich der Anzahl der Vergleichsoperationen pro Kantenänderung gebildet. Die Untersuchung zeigt, dass die dynamischen Algorithmen den statischen im Falle einer einzelnen Kantenänderung deutlich überlegen sind: Die Anzahl der Vergleichsoperationen kann mithilfe der dynamischen Algorithmen um den Faktor 10^3 reduziert werden. Benötigt ein statischer Algorithmus zur Berechnung des SPT in einem Graphen mit einer Knotenanzahl $|V| \geq 1000$ mindestens 10000 Vergleiche, so berechnet ein dynamischer Algorithmus (ausgenommen der dynamischen Variante von D'Esopo-Pape) einen aktualisierten SPT mithilfe von $30 - 60$ Vergleichen.

Unter den dynamischen Algorithmen benötigt der Ball–and–String-Algorithmus die wenigsten Vergleiche zur Berechnung des Ergebnisses, gefolgt von den Algorithmen von Dijkstra und Bellman und Ford. Der Algorithmus von D'Esopo-Pape benötigt mit großem Abstand die meisten Vergleiche. Ein Beispiel, in dem eine wichtige Kante aus dem Graph entfernt wird und die Auswirkung auf den SPT sehr groß ist, verdeutlicht die Stärken und Schwächen der verschiedenen Algorithmen. Während der Ball–and–String-Algorithmus, sowie der Algorithmus von Dijkstra die minimale Anzahl an Kanten während der Berechnung besuchen, werden diese von den anderen Algorithmen mehrfach besucht. Auch die Anzahl der Extraktionen von Einträgen aus der Warteschlange, und somit die Anzahl der Iterationen der Hauptschleife, sowie die Anzahl an neu erstellten Verknüpfungen innerhalb des Baums, unterscheiden sich stark. Der Ball–and–String-Algorithmus weist auch in diesen Fällen die minimale Anzahl an Operationen auf.

Edward Chan et al. untersuchen in [CH05] sowohl das zeitliche Verhalten, als auch die Anzahl der verwendeten Operationen unterschiedlicher Algorithmen u.a. auf zufällig generierten, dichten Graphen. Untersucht werden der SWSF-FP-Algorithmus, der Ball–

and–String-Algorithmus, sowie dynamische und statische Varianten des Algorithmus von Dijkstra. Letzterer wird lediglich als Referenzwert genutzt.

Zunächst wird der Einfluss der Größe der Gewichtsveränderung einer Kante untersucht. Im Fall der Erhöhung der Kantengewichte ist dieser gering, da die Algorithmen zunächst betroffene Knoten innerhalb des SPT lokalisieren und verarbeiten. Ob ein Knoten betroffen ist, ist davon abhängig, ob sich eine veränderte Kante auf dem zuvor berechneten kürzesten Pfad befindet; die Größe der Gewichtsveränderung hingegen ist irrelevant. Im Fall der Verminderung der Kantengewichte ist der Einfluss größer. Die Menge der betroffenen Knoten kann nicht initial bestimmt werden, sondern erweitert sich mit fortlaufender Bearbeitung des Algorithmus. Je größer die Dekrementierung des Kantengewichts ist, desto eher sind Knoten, welche direkt oder indirekt mit der Kante verbunden sind, betroffen. Betroffene Knoten müssen verarbeitet werden und erhöhen somit den Aufwand des Algorithmus zur Berechnung des neuen SPT. Zusammengefasst ist der Einfluss von Gewichtsveränderungen auf dynamische Algorithmen relativ gering. Werden die Gewichte der Kanten jedoch um einen sehr hohen prozentualen Anteil reduziert, so steigt die Laufzeit und die Anzahl der Operationen für alle dynamischen Algorithmen gleichermaßen stark an.

Bei der Untersuchung des Verhaltens der Algorithmen bei Erhöhung der Gewichte der Kanten in zufällig generierten Graphen schneidet der dynamische Algorithmus von Dijkstra, unabhängig des prozentualen Anteils der veränderten Kanten an der Gesamtanzahl der Kanten, geringfügig besser ab als der Ball-and-String-Algorithmus. Der acsSWSF-FP-Algorithmus ist mit Abstand der langsamste der verglichenen dynamischen Algorithmen, da dieser jeden betroffenen Knoten mehrfach besucht. Verglichen mit der statischen Variante des Algorithmus von Dijkstra ist die dynamische Variante für die getesteten Größen performanter, sofern der Anteil der veränderten Kanten gemessen an der Gesamtanzahl der Kanten einen Schwellwert von ungefähr 10% nicht überschreitet.

Wird das Verhalten der Algorithmen bei der Verminderung der Gewichte der Kanten betrachtet, so fällt das Ergebnis sehr ähnlich aus. Die dynamische Variante des Algorithmus von Dijkstra ist in diesem Fall deutlich performanter als die restlichen Algorithmen. Auch im Vergleich zum statischen Algorithmus von Dijkstra ist die dynamische Variante performanter, sofern der prozentuale Anteil der veränderten Kanten an der Gesamtkantenzahl gering ist. Ein genauer Schwellwert ist nicht abzuleiten, jedoch ist dieser deutlich größer als 10%. Dieses Verhalten lässt sich ebenso für gemischte Veränderungen der Gewichte der Kanten beobachten.

Reinhard Bauer et al. untersuchen in [RB09] verschiedene Versionen des SWSF-FP-Algorithmus. Diese werden mit der statischen Variante des Algorithmus von Dijkstra und dem dynamischen Algorithmus von Bellman und Ford im Fall einzelner Veränderungen und Dijkstra im Fall mehrerer Veränderungen in der ersten Stufe der Anpassung verglichen. Diese erwiesen sich in Vortests jeweils als schnellste Algorithmen des Frameworks für das spezielle Anwendungsgebiet.

Getestet wurden die Algorithmen unter anderem auf speziellen zufällig generierten Graphen. Grid-Graphen sind Graphen, in dem Knoten auf einem zweidimensionalen Feld angeordnet werden. Benachbarte Knoten sind durch Kanten verbunden, welchen ein zufälliges Gewicht zugewiesen wird. Im Fall einzelner Veränderungen ist der dynamische Algorithmus von Bellman und Ford allen Versionen des SWSF-FP-Algorithmus überlegen. Die in dieser Arbeit beschriebenen Variante benötigt mehr als die doppelte Zeit zur Berechnung des aktualisierten SPT. Sollen mehrere Gewichte der Kanten verändert werden, so ist der dynamische Algorithmus verglichen mit dem SWSF-FP-Algorithmus in der beschriebenen Form nur leicht schneller; verbesserte Versionen des SWSF-FP-Algorithmus können dieses Problem schneller lösen.

7.1.2. Eigene Experimente

Dieser Abschnitt stellt die Resultate eigener Experimente vor, welche mithilfe der prototypischen Implementierung durchgeführt werden. Das Kriterium für den Vergleich und die Bewertung der Algorithmen ist die durchschnittlich aufgewandte Ausführungszeit. Unterteilt in die unterschiedlichen Veränderungsarten der Kantenbewertungen werden Experimente für verschiedene Parameterbelegungen durchgeführt. Folgende Parameter werden im Folgenden evaluiert:

- Anzahl der Knoten der Graphen
- Anzahl der Kanten der Graphen im Verhältnis zur Anzahl der Knoten
- Anzahl der Änderungen im Verhältnis zur Anzahl der Kanten

Die Parameter für die Anzahl der Kanten und Änderungen werden jeweils als prozentualer Wert zur Bezugsgröße angegeben. Die Ausführungszeiten der dynamischen Algorithmen werden mit der des statischen Algorithmus von Dijkstra verglichen. Durch die Durchführung der Experimente soll eine Erkenntnis darüber erlangt werden, unter welchen Bedingungen die dynamischen Algorithmen den statischen überlegen sind. Es werden

Experimente vorgestellt, in denen jeweils ein Parameter variabel und die restlichen fix sind.

Erhöhung der Kantengewichte

Dieser Abschnitt befasst sich mit Experimenten, in denen die Gewichte der Kanten erhöht werden.

Zunächst werden Experimente auf Graphen verschiedener Größe durchgeführt.

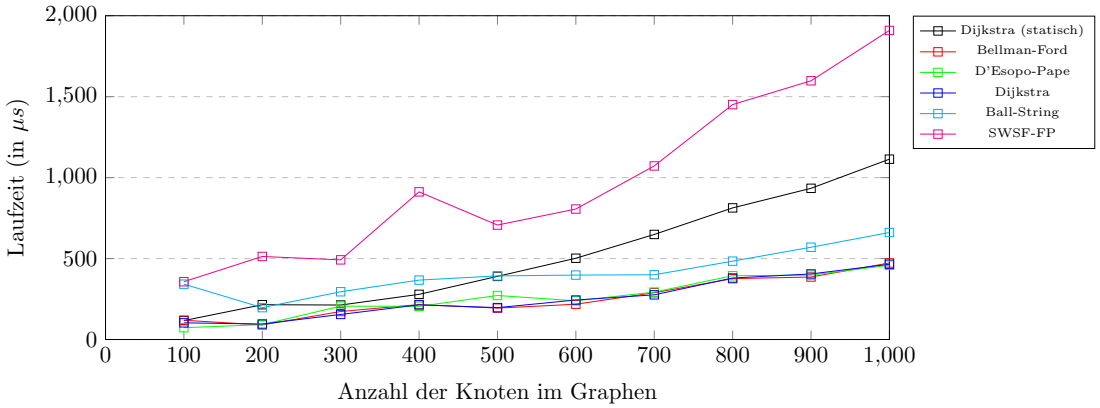


Abbildung 7.1.: Experimente - Erhöhung: variable Graphengröße

Abbildung 7.1 zeigt die Ergebnisse des Experiments. Für jede Graphengröße wurden 100 Beispielgraphen erstellt, auf denen die Algorithmen mit einer entsprechenden Menge an Änderungen angewandt wurden. Die restlichen Parameter wurden folgendermaßen ausgewählt: Anzahl der Kanten $= 3 *$ (Anzahl der Knoten), Anzahl der Änderungen $=$ $0.1 *$ (Anzahl der Kanten). Aus den Ausführungszeiten berechnete Mittelwerte sind der Abbildung zu entnehmen. Die dynamischen Algorithmen des Frameworks schneiden in dieser Testreihe unabhängig der Größe der Graphen am besten ab und weisen durchgängig nahezu identische Ausführungszeiten auf. Der Ball-and-String-Algorithmus ist in diesem Experiment stets etwas langsamer. In kleinen Graphen ist der statische Algorithmus von Dijkstra in etwa gleich schnell im Vergleich zu den semidynamischen Algorithmen. Mit zunehmender Graphengröße erhöht sich der Faktor, um den die semidynamischen Algorithmen dem statischen überlegen sind. So berechnen die Algorithmen des Frameworks beispielsweise ab einer Graphengröße von 500 Knoten das Ergebnis mindestens doppelt

so schnell wie der statische Algorithmus. Der volldynamische SWSF-FP-Algorithmus schneidet in diesem Experiment am schlechtesten ab.

Nachfolgend werden Ergebnisse von Experimenten vorgestellt, bei denen der Parameter Anzahl der Kanten variabel ist.

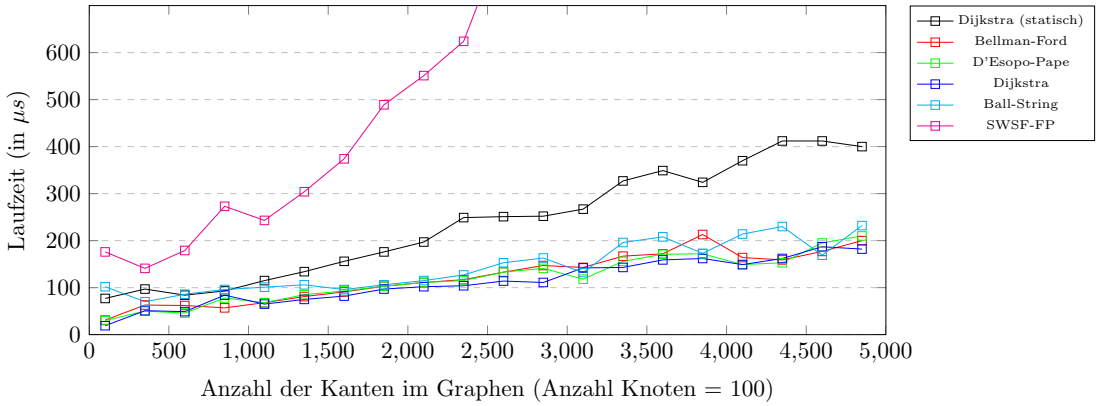


Abbildung 7.2.: Experimente - Erhöhung: variable Kantenanzahl

In Abbildung 7.2 wird das Ergebnis des Experiments dargestellt. Die getesteten Graphen besaßen 100 Knoten. Für jede getestete Anzahl an Kanten wurden 100 Beispielgraphen erstellt und mithilfe der Algorithmen und generierten Änderungsmengen der Größe $0.1 * ($ Anzahl der Kanten $)$ verarbeitet. Die dynamischen Algorithmen des Frameworks, sowie der Ball-and-String-Algorithmus schneiden in diesem Experiment sehr ähnlich ab. Ihre Ausführungszeiten sind deutlich schneller im Vergleich zu denen des statischen Dijkstra-Algorithmus, die Differenz wächst mit zunehmender Kantenanzahl. Ähnlich zum ersten Experiment schneidet auch hier der dynamische SWSF-FP-Algorithmus am schlechtesten ab.

Die Auswirkungen unterschiedlicher Werte für den dritten Parameter, die Anzahl der Veränderungen im Verhältnis zur Anzahl der Kanten, sollen im letzten Experiment für Gewichtserhöhungen festgestellt werden.

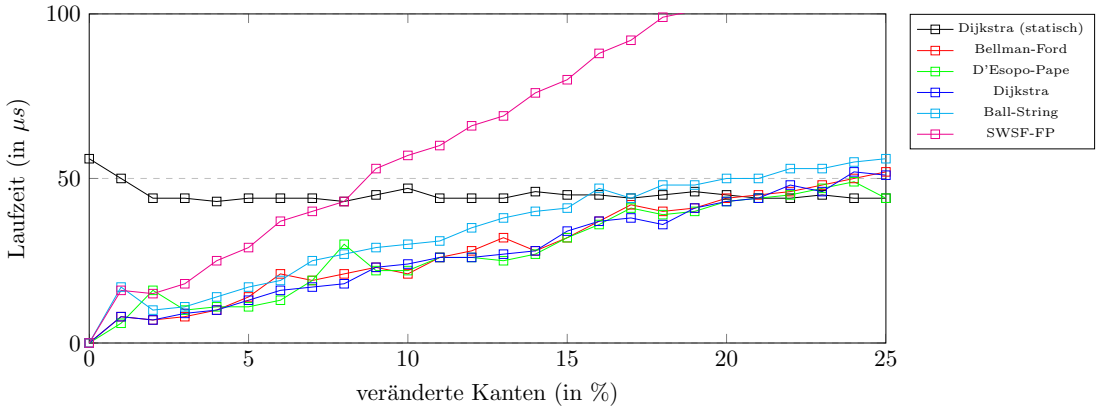


Abbildung 7.3.: Experimente - Erhöhung: variable Anzahl der Änderungen

Abbildung 7.3 zeigt die Ergebnisse des Experiments. In diesem Fall wurde der Test für jeden getesteten Prozentsatz veränderter Kanten 1000-fach wiederholt. Der Vergleich der Ergebnisse unter den dynamischen Algorithmen fällt ähnlich aus wie in den Experimenten zuvor. Die Algorithmen des Frameworks sind abermals geringfügig schneller als der Ball-and-String-Algorithmus. Die SWSF-FP-Algorithmus ist auch in diesem Fall deutlich langsamer. Der statische Algorithmus von Dijkstra ist unabhängig des Prozentsatzes veränderter Kanten immer in etwa gleich schnell. Das liegt daran, dass der Algorithmus das Ergebnis jedes mal komplett neu berechnet. Es können Grenzwerte bestimmt werden, die anzeigen, bis zu welchem Prozentsatz sich die Verwendung dynamischer Algorithmen profitiert und ab wann eine komplette Neuberechnung schneller ist. Beim SWSF-FP-Algorithmus liegt dieser Grenzwert bei 8%, beim Ball-and-String-Algorithmus bei 17%, und bei den Algorithmen des Frameworks bei 20%.

Verminderung der Kantengewichte

Dieser Abschnitt befasst sich mit Experimenten, in denen die Gewichte der Kanten dekrementiert werden.

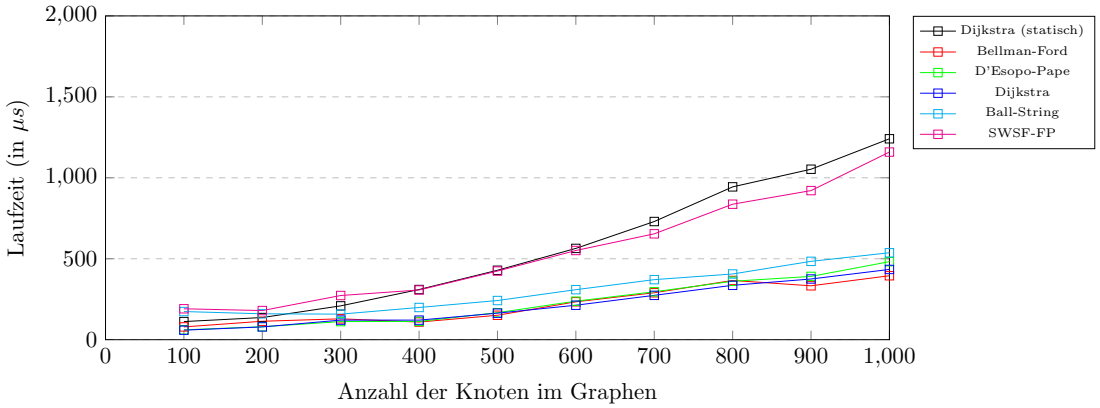


Abbildung 7.4.: Experimente - Verminderung: variable Graphengröße

Abbildung 7.4 zeigt die Ergebnisse des Experiments. Die Parameter des Experiments sind identisch zu denen des Tests für Gewichtserhöhungen (Anzahl der Kanten: 3∗ Anzahl der Kanten, Anzahl der Änderungen: 0.1∗ Anzahl der Kanten). Die semidynamischen Algorithmen schneiden in diesem Experiment erneut am besten ab. Der Ball-and-String-Algorithmus ist etwas langsamer als die Algorithmen des Frameworks, welche nahezu gleichauf liegen. Auffällig im Vergleich zum Experiment für Gewichtserhöhungen ist die deutliche Verbesserung der Laufzeit des SWSF-FP-Algorithmus. Dies ist auf die unterschiedliche Behandlung von Erhöhungen und Verminderungen innerhalb des Algorithmus zurückzuführen, wobei sich die Behandlung von Verminderungen zumindest innerhalb der im Rahmen dieser Arbeit durchgeführten Experimente als effizienter erweist.

Nachfolgend werden Ergebnisse von Experimenten vorgestellt, bei denen der Parameter Anzahl der Kanten variabel ist.

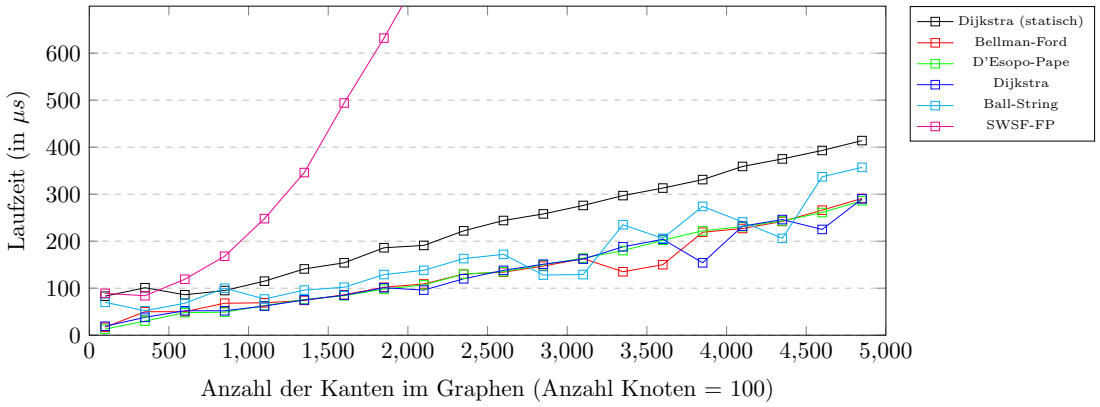


Abbildung 7.5.: Experimente - Verminderung: variable Kantenanzahl

In Abbildung 7.5 wird das Ergebnis des Experiments dargestellt. Die getesteten Graphen besaßen 100 Knoten und es wurden je Durchlauf das Gewicht von 10% der Kanten dekrementiert. Das Ergebnis ist auf den ersten Blick nahezu identisch zu dem Ergebnis des Experiments mit Gewichtserhöhungen, allerdings ist der Abstand zwischen den semidynamischen und dem statischen Algorithmus deutlich geringer im Falle der Gewichtsverminderungen. Die Differenz der Ausführungszeiten zwischen statischem und semidynamischen Algorithmen wächst mit der Anzahl der Kanten. Der volldynamische SWSF-FP-Algorithmus schneidet auch in diesem Experiment am schlechtesten ab.

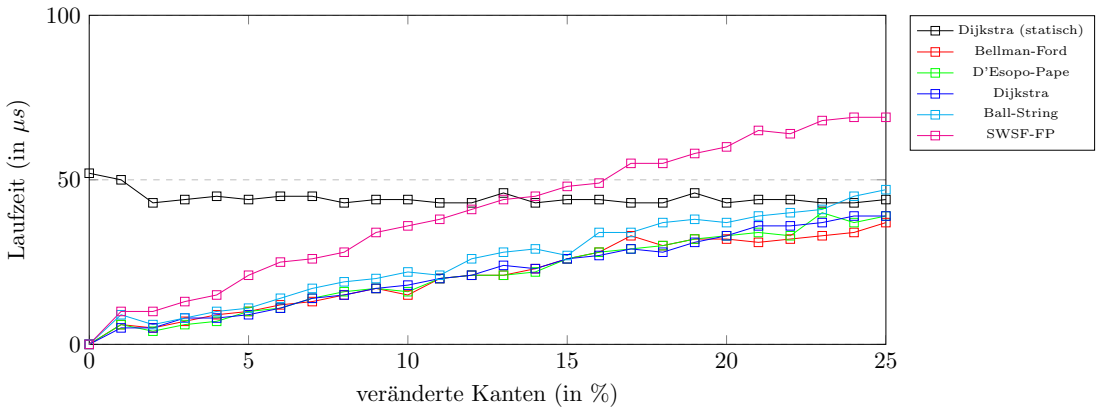


Abbildung 7.6.: Experimente - Verminderung: variable Anzahl der Änderungen

Auch die in Abbildung 7.6 präsentierten Ergebnisse des Experiments ähneln ihrem Pendant für Kantengewichtserhöhungen. Allerdings haben sich die Grenzwerte, bis zu denen ein dynamischer Algorithmus schneller als der statische Algorithmus von Dijkstra ist, verschoben. Der Grenzwert für den SWSF-FP-Algorithmus liegt in diesem Fall bei 13%, der des Ball-and-String-Algorithmus bei 23%. Der Grenzwert der Algorithmen des Frameworks liegt etwa bei 26-28%.

Gemischte Änderungen der Kantengewichte

In diesem Abschnitt werden Ergebnisse von Experimenten vorgestellt, in denen gemischte Änderungen auf die Kantengewichte ausgeführt wurden. Die Experimente wurden unter anderen mit zusammengesetzten volldynamischen Algorithmen durchgeführt. Nachfolgend werden die gewählten Kombinationen mit den entsprechenden Kürzeln in Klammern, welche in den Legenden der Abbildungen verwendet werden, aufgelistet, wobei der erstgenannte Algorithmus Gewichtserhöhungen und der zweite Gewichtsverminderungen

bearbeitet. Der dynamische Dijkstra wurde mit der zweiten Stufe der Anpassung verwendet.

- Dijkstra - Dijkstra (Dijk-Dijk)
- Ball-and-String - Ball-and-String (BS-BS)
- Dijkstra - Ball-and-String (Dijk-BS)
- Ball-and-String - Dijkstra (BS-Dijk)

Das erste Experiment evaluiert den Parameter Anzahl der Knoten eines Graphen im Fall der gemischten Veränderungen der Kantengewichte.

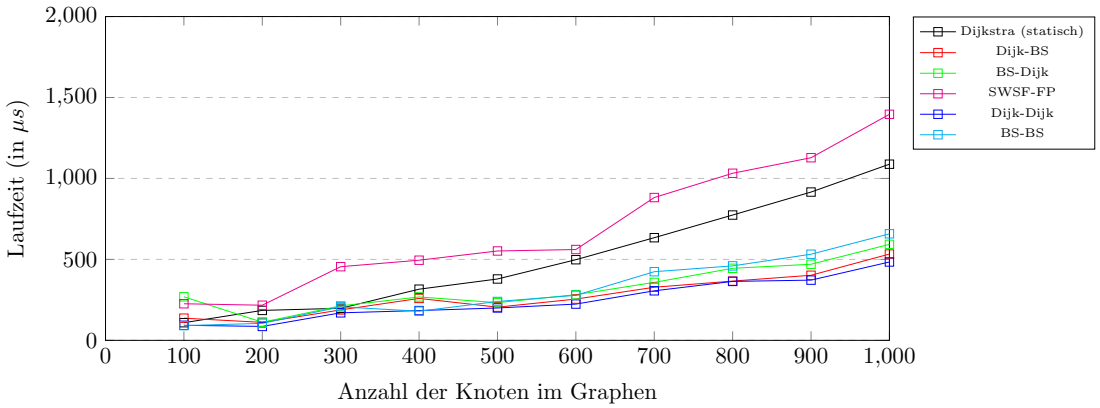


Abbildung 7.7.: Experimente - Gemischte Änderungen: variable Graphengröße

Abbildung 7.7 zeigt die Ergebnisse des Experiments. Die übrigen Parameter wurden mit den gleichen Werten belegt wie in den Experimenten für Erhöhungen und Verminderungen der Gewichte. Die Kombinationen der semidynamischen Algorithmen schneiden in diesem Experiment am besten ab. Die iterative Ausführung dieser verlangsamt die Berechnung nur gering, die Ausführungszeiten sind ähnlich im Vergleich zu denen der Experimente für Kantenänderungen einer Art. Am schnellsten schneidet die Kombination der dynamischen Algorithmen von Dijkstra für Erhöhungen und Verminderungen ab. Diese waren auch in den Experimenten für Kantenänderungen einer Art am schnellsten. Auch die Ausführungszeit des volldynamischen SWSF-FP-Algorithmus liegt in etwa im Mittel der Ausführungszeiten der vorangegangenen Experimente. Durch die einmalige Ausführung des Algorithmus kann somit keine erkennbare Verbesserung festgestellt werden. Der Algorithmus ist im Vergleich zum statischen Algorithmus von Dijkstra etwas langsamer.

Im zweiten Experiment wird erneut der Einfluss des Parameters Anzahl der Kanten im Graphen untersucht.

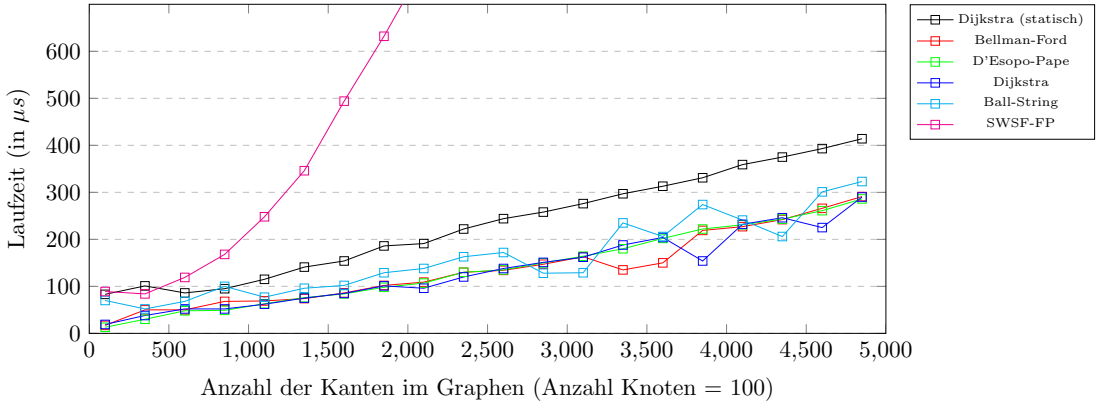


Abbildung 7.8.: Experimente - Gemischte Änderungen: variable Kantenanzahl

Die Ergebnisse, dargestellt in Abbildung 7.8, weisen ähnliche Tendenzen auf wie die Experimente für Kantenänderungen der anderen Änderungsarten. Auch in diesem Fall schneiden die Kombinationen semidynamischer Algorithmen, insbesondere unter Verwendung des dynamischen Algorithmus von Dijkstra, am besten ab. Unabhängig der Anzahl der Kanten berechnen diese das Ergebnis schneller als der statische Algorithmus von Dijkstra. Der volldynamische SWSF-FP-Algorithmus schneidet auch in diesem Experiment am schlechtesten ab.

Das letzte Experiment befasst sich mit der Erhöhung der Anzahl der Kantenänderungen.

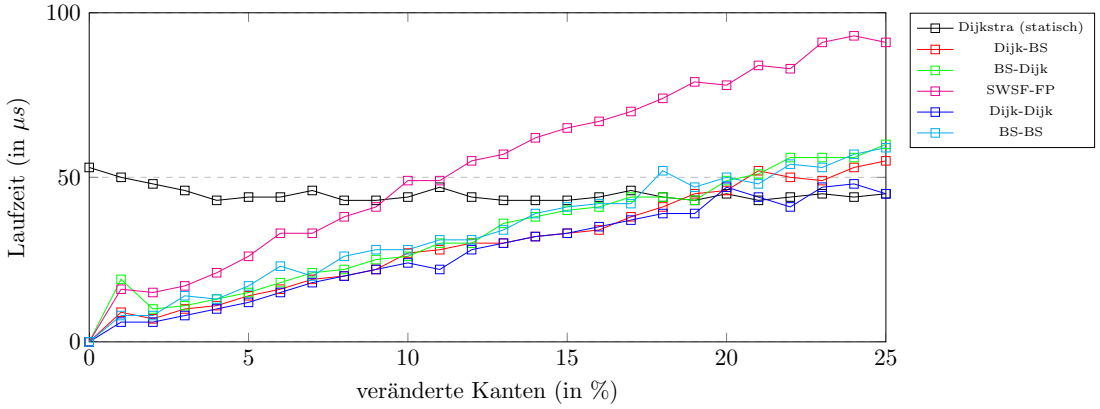


Abbildung 7.9.: Experimente - Gemischte Änderungen: variable Anzahl der Änderungen

Abbildung 7.9 zeigt die Ergebnisse des Experiments. Das Abschneiden der einzelnen Algorithmen untereinander entspricht dem der vorherigen Experimente. Die Grenzwerte, ab denen der statische Algorithmus von Dijkstra die Berechnung schneller abschließt, liegen zwischen den experimentell ermittelten Werten für Kantenänderungen einer Art. Der Grenzwert für den volldynamischen SWSF-FP-Algorithmus liegt bei 10%. Die Grenzwerte für die Kombinationen der semidynamischen Algorithmen liegen im Bereich von 18% bis 21%, und somit auch zwischen denen der Experimente für Änderungen einer Art.

7.1.3. Vergleich der Ergebnisse

Die Ergebnisse der Untersuchungen von Paolo Narvaéz et al., welche in 7.1.1 beschrieben wurden, konnten durch die eigenen Experimente teilweise bestätigt werden. Paolo Narvaéz verglich die Anzahl der Operationen statischer und semidynamischer Algorithmen bei der Berechnung eines neuen SPT für eine einzelne Kantenänderung. Dabei wurde festgestellt, das sich diese bei Benutzung eines semidynamischen Algorithmus um den Faktor 10^3 reduziert haben. Das Kriterium zur Beurteilung der Algorithmen war in den eigenen Experimente nicht die Anzahl der Operationen, sondern die Laufzeit der Algorithmen. Trotzdem kann die Aussage von Narvaéz bestätigt werden. Ist die Anzahl der veränderten Kantengewichte gering, so konnten deutliche Unterschiede in der Laufzeit zwischen statischen und dynamischen Algorithmen festgestellt werden. Dies gilt unabhängig der Art der Änderung und wird durch die Ergebnisse in den Abbildungen 7.3, 7.6 und 7.9 bestätigt.

Edward Chan et al. untersuchten die semidynamischen Algorithmen Ball-and-String und Dijkstra, sowie den volldynamischen SWSF-FP-Algorithmus. Getestet wurde unter anderem der Einfluss der Parameter Anzahl der Änderungen. Das Resultat des Experiments ist, wie auch bei den eigenen Experimenten, dass der dynamische Algorithmus von Dijkstra unabhängig der Kantenänderungsart am besten und nur etwas besser als der Ball-and-String-Algorithmus abschneidet. Der volldynamische SWSF-FP-Algorithmus hingegen ist wie in den eigenen Experimenten deutlich langsamer. Die Ergebnisse der Experimente unterscheiden sich jedoch in den festgestellten Grenzwerten, ab denen die Berechnung des statischen Algorithmus von Dijkstra schneller ist. Im Fall der Gewichtserhöhungen ermittelten Chan et al. einen Grenzwert von 10%, während dieser bei den eigenen Experimenten bei etwa 20% lag. Der Grenzwert steigt in beiden Experimenten für die anderen Änderungsarten ähnlich an.

Die Ergebnisse der Experimente von Reinhard Bauer et al. können mithilfe der eigenen Experimente nur bedingt bestätigt werden. Zwar ist der SWSF-FP-Algorithmus in beiden Experimentreihen langsamer als Repräsentanten des dynamischen Algorithmenframeworks, jedoch ist der Abstand bei Bauer et al. deutlich geringer, sodass einige Verbesserungen sogar dafür sorgen, dass dieser Ergebnisse schneller berechnen kann. Diese Entwicklung kann durch die eigenen Experimente nicht bestätigt werden. Einer der Gründe dafür ist die prototypische Implementierung, welche an einigen Stellen optimiert werden kann.

7.2. Straßennetzabbildende Graphen

Edward Chan et al. untersuchen in [CH05] sowohl das zeitliche Verhalten als auch die Anzahl der verwendeten Operationen unterschiedlicher Algorithmen auf Graphen verschiedener Größe ($|V|_{min} = 1000, |V|_{max} = 15000$), welche reale Straßennetze abbilden.

Bei der Untersuchung des Verhaltens der Algorithmen bei Erhöhung der Gewichte der Kanten in straßennetzabbildenden Graphen schneidet der Ball-and-String-Algorithmus, unabhängig des prozentualen Anteils der veränderten Kanten an der Gesamtanzahl der Kanten am besten ab. Er ist dabei nur geringfügig besser als die dynamische Variante des Algorithmus von Dijkstra, welche bei den Experimenten auf zufällig generierten Graphen am schnellsten war. Der SWSF-FP-Algorithmus ist auch auf straßennetzabbildenden Graphen mit Abstand der langsamste der verglichenen dynamischen Algorithmen, da dieser jeden betroffenen Knoten mehrfach besucht. Verglichen mit der statischen Variante des Algorithmus von Dijkstra ist die dynamische Variante für die getesteten Größen performanter, sofern der Anteil der veränderten Kanten einen Schwellwert von circa $2\% - 4\%$ nicht überschreitet. Dieser Schwellwert ist verglichen zu dem der Experimente mit zufällig generierten Graphen deutlich geringer.

Wird das Verhalten der Algorithmen bei der Verminderung der Gewichte der Kanten betrachtet, so fällt das Ergebnis deutlich anders aus. Die dynamische Variante des Algorithmus von Dijkstra ist in diesem Fall deutlich performanter als die restlichen Algorithmen. Der Ball-and-String-Algorithmus schneidet bei diesem Test erstmalig und deutlich am schlechtesten ab. Im Vergleich zum statischen Algorithmus von Dijkstra ist die dynamische Variante performanter, sofern nicht mehr als etwa 10% der Gewichte der Kanten verändert werden.

Für gemischte Veränderungen der Gewichte der Kanten wird ein weiterer Algorithmus, ein zusammengesetzter volldynamischer Algorithmus, eingeführt und getestet. Dieser besteht aus dem Ball-and-String-Algorithmus zur Bestimmung der Lösung für Erhöhungen der Gewichte der Kanten und der dynamischen Variante des Algorithmus von Dijkstra zur Lösung der Problems für Verminderungen der Kantengewichte. Bei der Untersuchung des Verhaltens der Algorithmen für gemischte Veränderungen schneidet der zusammengesetzte Algorithmus am besten und nur geringfügig besser als die dynamische Variante des Algorithmus von Dijkstra ab. Der Grenzwert zum statischen Algorithmus liegt auch in diesem Fall bei nur etwa 2% und somit deutlich unter dem ermittelten Wert für zufällig generierte Graphen.

Reinhard Bauer et al. untersuchen in [RB09] ebenfalls die bereits in Abschnitt 7.1 beschriebenen Algorithmen auf straßennetzabbildenden Graphen. Auf diesen ist die dynamische Variante des Algorithmus von Bellman und Ford bei Änderung des Gewichts einer einzelnen Kante den anderen Algorithmen überlegen. Werden jedoch mehrere Änderungen der Gewichte vorgenommen, so ist dieser im Vergleich zum ursprünglichen SWSF-FP-Algorithmus nur leicht schneller. Verbesserte Versionen des SWSF-FP-Algorithmus können das Ergebnis hingegen schneller berechnen.

8

Abschlussbetrachtung

In dem vorliegende Kapitel findet eine abschließende Betrachtung dieser Arbeit statt. Die Inhalte und Ergebnisse werden kurz zusammengefasst und ein Fazit aus den erworbenen Erkenntnissen gezogen. Abschließend wird ein Ausblick gegeben, welcher mögliche Ansätze zur Verbesserung aufweist und Anknüpfungspunkte für weitere Arbeiten auf dem Themengebiet bietet.

8.1. Zusammenfassung

Im Rahmen dieser Arbeit hat eine intensive Beschäftigung mit dem Themengebiet der Bestimmung kürzester Wege in dynamischen Graphen stattgefunden. Dieses ist eng verwandt mit dem der Bestimmung der kürzesten Wege in statischen Graphen, welches auch als Single Source Shortest Path (SSSP)-Problem bekannt ist und oft Gegenstand der Forschung war. Bekannte Algorithmen zur Lösung des Problems wurden von Richard Bellman und Lester Ford, sowie Edsger W. Dijkstra entwickelt.

In dieser Arbeit werden diese und der Algorithmus von D'Esopo-Pape in Kapitel 3 vorgestellt und zu einem Framework zusammengefasst, in welchem alle Algorithmen durch die unterschiedliche Implementierungsform einer Warteschlange realisiert werden. Diese dienen nicht nur als Referenz für Algorithmen zur Lösung des Problems der Bestimmung kürzester Wege in dynamischen Graphen (DSP-Problem). Vielmehr werden in Kapitel 4 Anpassungen thematisiert, welche die Algorithmen insofern modifizieren, dass diese das dynamische Problem unter der Einschränkung lösen können, dass alle Änderungen gleichartig (entweder Erhöhungen oder Verminderungen) sind. Im selben Kapitel wird der Ball-and-String-Algorithmus vorgestellt, welcher Ähnlichkeiten zum dynamischen Algorithmus von Dijkstra aufweist, sich jedoch insbesondere durch das Sortierungskriterium der Warteschlange von ihm unterscheidet. Für ihn gilt dieselbe Einschränkung bezüglich der Gleichartigkeit der Änderungen. Sollen die Algorithmen verschiedenartige Änderungen verarbeiten, so müssen sie für jeden Änderungstypen iterativ ausgeführt werden.

Zur Vermeidung dieser Problematik wurde der volldynamische SWSF-FP-Algorithmus entwickelt und wird in Kapitel 5 vorgestellt. Dieser kann verschiedenartige Änderungen innerhalb einer Ausführung verarbeiten. Er basiert auf einer Generalisierung des Algorithmus von Dijkstra mithilfe einer kontextfreien Grammatik und kann verschiedenartige Änderungen unter anderem durch die Einführung neuer Zustände der Knoten und einer zustandsabhängigen Bearbeitung dieser verarbeiten.

Im Rahmen dieser Arbeit ist eine prototypische Implementierung zur Veranschaulichung der Arbeitsweise der Algorithmen einerseits und zur Messung der Laufzeiten andererseits entstanden. Diese wird in Kapitel 6 unter Betrachtung der einzelnen Komponenten beschrieben und ihre Funktionsweise anhand einzelner Anwendungsbeispiele demonstriert.

In Kapitel 7 werden Experimente mit den eingeführten Algorithmen durchgeführt. Als Grundlage zur Ausführung der Experimente dienen zwei Arten von Graphen:

- zufällig generierte Graphen werden algorithmisch und mithilfe eines Zufallszahlengenerators beliebig erstellt.
- straßennetzabbildende Graphen repräsentieren reale Straßennetze, welche auf Grundlage von Karten manuell erstellt wurden.

Es werden sowohl Experimente anderer Autoren für beide Graphenarten, als auch Experimente, welche mithilfe der prototypischen Implementierung auf zufällig generierten Graphen durchgeführt wurden, vorgestellt. Die Experimente zeigen, dass dynamische

Algorithmen Vorteile gegenüber statischen haben, sofern die Anzahl der Kantenänderungen im Verhältnis zur Anzahl der Kanten im Graphen einen bestimmten Grenzwert nicht überschreitet. Der Grenzwert ist von der Art des Graphen abhängig. Auf zufällig generierte Graphen konnten höhere Grenzwerte (je nach Autor und Änderungsart im Bereich 10-20%) erreicht werden als auf straßennetzabbildenden (etwa 2%). In den eigenen Experimenten schnitt die dynamische Variante des Algorithmus von Dijkstra am besten ab. Der volldynamische SWSF-FP-Algorithmus schnitt hingegen am schlechtesten ab. In Experimenten anderer Autoren konnten jedoch durch einige Modifikationen des Algorithmus bessere Ergebnisse erzielt werden.

8.2. Fazit

Die vorliegende Arbeit verschafft einen Überblick über Algorithmen, welche das DSP-Problem lösen. Es wurden sowohl Algorithmen vorgestellt, welche aus dem Umfeld des SSSP-Problem stammen und lediglich für DSP-Problem modifiziert wurden, als auch solche, welche für das dynamische Problem konzipiert wurden (z.B. Ball-and-String-Algorithmus). Die Ideen und Arbeitsweisen der Algorithmen wurden beschrieben, analysiert und durch Beispiele veranschaulicht.

Die Ergebnisse der Experimente, welche mithilfe der prototypischen Implementierung durchgeführt wurden, zeigen, dass dynamische Algorithmen zur Lösung des DSP-Problems ihren statischen Pendants überlegen sind, sofern die Anzahl der Änderungen einen bestimmten Grenzwert nicht überschreiten. Der schlechteste experimentell festgestellte Grenzwert von 2% lässt jedoch einigen Freiraum für die Anwendung solcher Algorithmen: In einem Straßennetz mit beispielsweise 1000 Kreuzungen und 3000 Verbindungsstraßen dürften sich die erwarteten Fahrzeiten für bis zu 60 Straßen zeitgleich verändern, ehe ein statischer Algorithmus die kürzesten Distanzen schneller berechnen kann als sein dynamisches Gegenüber.

8.3. Ausblick

Der volldynamische SWSF-FP-Algorithmus schnitt in der von Thomas Reps und Ganesan Ramalingam vorgestellten Form in den Experimenten aufgrund häufiger Neuberechnungen von Distanzen verglichen mit den anderen Algorithmen relativ schlecht ab. Reinhard Bauer

et al. gelang es in [RB09] den Algorithmus zu optimieren und konnten damit die Laufzeit soweit verbessern, dass der optimierte Algorithmus in einigen Experimenten am besten abschnitt. Diese Optimierungen wurden im Rahmen dieser Arbeit nicht implementiert, sind jedoch ein interessanter Anhaltspunkt für weitere Untersuchungen.

Neben der Implementierung des SWSF-FP-Algorithmus kann der Prototyp auch an anderen Stellen weiter optimiert werden. In der Entwurfsphase der Software wurde darauf geachtet, die Komponenten so leicht austauschbar wie möglich zu machen. So können weitere Implementierungen für die verwendete Datenstrukturen einfach ausprobiert und mithilfe der Experimente evaluiert werden. Auch der verwendete Graphengenerator ist leicht austauschbar, sodass dieser beispielsweise durch eine Implementierung ersetzt werden kann, welche Graphen generiert, die Straßennetzen besonders ähnlich sind.

Im Rahmen dieser Arbeit wurden nur Experimente auf zufällig generierten Graphen durchgeführt. Eine Integration von beispielsweise OpenStreetMap-Daten oder Abbildungen von Netzwerken zur Erstellung von Graphen würde es ermöglichen, die Experimente im realen Umfeld durchzuführen und die Anwendbarkeit der Algorithmen abschließend zu bewerten.

Literaturverzeichnis

[IW14] Sebastian Iwanowski, Rainer Lang. *Diskrete Mathematik mit Grundlagen*, Springer Vieweg, Wiesbaden, Deutschland, 2014, ISBN 978-3-658-07130-1

[CO01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, 2nd Edition, The MIT Press, Cambridge, Massachusetts, USA, 2001, ISBN 978-0262032933

[CH05] Edward P.F. Chan, Yaya Yang, *Shortest Path Trees Computation in Dynamic Graphs*, University of Waterloo, Waterloo, Ontario, Kanada, 2005, https://cs.uwaterloo.ca/research/tr/2005/CS-2005-14.pdf, Abruf 28.02.2016

[RB09] Reinhard Bauer, Dorothea Wagner. *Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study*, Karlsruhe Institute of Technology, Germany. Aus *Experimental Algorithms*, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings, S. 51-62, ISBN 978-3-642-02010-0, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.165.2037&rep=rep1&type=pdf, Abruf 28.02.2016

[JG12] J. Gao, Q. Zhao. *Dynamic Shortest Path Algorithms for Hypergraphs*, University of New York, USA, 2012, http://arxiv.org/pdf/1202.0082.pdf, Abruf 28.02.2016

[DB92] Dimitri P. Bertsekas. *A Simple and Fast Label Correcting Algorithm for Shortest Paths*, Networks, Vol. 23, S. 703-709, 1993, http://www.mit.edu/~dimitrib/SLF.pdf, Abruf 28.02.2016

[KE81] A. Kershenbaum. *A Note on Finding Shortest Path Trees*, Networks, Vol. 11, S. 399-400, 1981

[DG79] R. Dial, F. Glover, D. Karney, D. Klingman. *A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees*,

Networks, Vol. 9, S. 215-248, 1979

[NA00] Paolo Narváez, Kai-Yeung Siu, Hong-Yi Tzeng. *New Dynamic Algorithms for Shortest Path Tree Computation*, IEEE/ACM Transactions on Networking, Vol. 8, No. 6, 2000, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4731&rep=rep1&type=pdf, Abruf 28.02.2016

[NA01] Paolo Narváez, Kai-Yeung Siu, Hong-Yi Tzeng. *New Dynamic SPT Algorithm based on a Ball-and-String Model*, IEEE/ACM Transactions on Networking, Vol. 9, No. 6, S. 706-718, 2001, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6310, Abruf 28.02.2016

[RR92] Ganesan Ramalingam, Thomas Reps. *An Incremental Algorithm for a Generalization of the Shortest-Path Problem*, University of Wisconsin, Madison, USA, 1992, ftp://ftp.cs.wisc.edu/pub/techreports/1992/TR1087.pdf, Abruf 28.02.2016

[DK76] Donald E. Knuth. *A Generalization of Dijkstra's Algorithm*, Stanford University, Kalifornien, USA, 1976, http://cis.upenn.edu/~lhuang3/knuth77.pdf, Abruf 28.02.2016

[GS] GraphStream – A Dynamic Graph Library, http://graphstream-project.org/, Abruf 28.02.2016

[SW] Java Swing, http://docs.oracle.com/javase/tutorial/uiswing/index.html, Abruf 28.02.2016

[JU] JUnit, http://junit.org/, Abruf 28.02.2016

[EC] Eclipse, https://eclipse.org/, Abruf 28.02.2016

A

Inhalt der beiliegenden CD

Die wichtigsten Verzeichnisse auf der beiliegenden CD und deren Inhalt sind in der folgenden Übersicht aufgelistet:

- **Ausarbeitung**
  - **MA-Moench** – Masterthesis als PDF-Datei
  - **Quellen**
- **Implementierung**
  - **DynamicRouting** – Aktueller Entwicklungsstand der prototypischen Implementierung als ausführbares Java Archive
  - **doc** – Klassendokumentation des Prototypen
  - **src** – Quellcode des Prototypen
  - **Benutzerhandbuch** – Handbuch als PDF-Datei

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum Nicolas Mönch