Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Berechnung kürzester Wege mit negativen Kantenkosten

Patrick Holtz

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Dr. rer. nat. Claudius Proissl
Beginn am:	4. Oktober 2023
Beendet am:	4. April 2024

Kurzfassung

Das Kürzeste-Wege-Problem ist ein elementares Problem der Graphentheorie. Während sich für Graphen mit positiven Kantenkosten schon lange nahe-lineare Algorithmen wie der Dijkstra-Algorithmus etabliert haben, sind im Allgemeinen für Graphen mit beliebigen, also auch negativen Kantenkosten, nur Algorithmen mit schlechterer Komplexität bekannt. Bekannte Beispiele hierfür sind der Bellman-Ford Algorithmus oder der Goldberg-Algorithmus, die in $\mathcal{O}(nm)$ bzw. $\mathcal{O}(m\sqrt{n}\log W)$ laufen. Der 2022 vorgestellte Algorithmus von Bernstein et al. verspricht mit $\mathcal{O}(m\log^8(n)\log W)$ erstmals eine nahe-lineare Komplexität, die die bestehenden Algorithmen in der Theorie verbessert.

In dieser Arbeit wird Bernsteins Algorithmus vereinfacht und von Grund auf erklärt und eine praktische Implementierung vorgestellt. Außerdem wird durch Laufzeittests untersucht, wie Bernsteins Algorithmus im Vergleich mit ausgewählten bestehenden Algorithmen abschneidet. Die Tests erfolgen auf drei verschiedenen Graphklassen.

Es konnte festgestellt werden, dass Bernsteins Algorithmus auf zufällig generierten Graphen mit etwa 10^6 Knoten um ein Vielfaches langsamer ist als eine effiziente Implementierung von Bellman-Ford (SPFA). Trotzdem konnte gezeigt werden, dass die nahe-lineare Komplexität erreicht wird. Somit ergibt sich bei der Anwendung auf praktische Probleme für Bernsteins Algorithmus kein Laufzeitvorteil, weshalb er die anderen Algorithmen erst bei weitaus größeren Eingabegraphen zeitlich überbieten kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

APSP-Problem All-Pair Shortest Path Problem. 18

DAG Directed Acyclic Graph. 17

DFS Depth First Search. 19

LDD Low Diameter Decomposition. 37

SCC Strongly Connected Component. 17

SPFA Shortest Path Faster Algorithm. 23

SSSP-Problem Single-Source Shortest Path Problem. 18

1 Einleitung

Die Bestimmung kürzester Pfade in Graphen ist seit vielen Jahrzehnten ein Forschungsgebiet der theoretischen Informatik. Das Problem findet Einsatz in zahlreichen Bereichen beginnend bei der Routenplanung in Verkehrsnetzen oder Rechnernetzen bis hin zur Untersuchung von sozialen Netzwerken.
Die Forschung fächert sich in viele Teilgebiete auf, abhängig von den Anforderungen an den Eingabegraphen und an das Ergebnis. Für Graphen mit positiven Kantenkosten hat sich unter der Verwendung von Fibonnaci Heaps Dijkstras Algorithmus mit einer asymptotischen Komplexität von $\mathcal{O}(n \cdot \log(n) + m)$ bewährt [Dij59], [FT87]. Dabei stellen n und m die Anzahl der Knoten und Kanten dar.
Im Gegensatz dazu stellt sich das Problem der Findung kürzester Wege unter Einbeziehung negativ gewichteter Kanten als aufwendiger heraus. Der bekannte aus den 1960er Jahren stammende Bellman-Ford Algorithmus [Bel58], [For56] schafft es, dieses Problem in $\mathcal{O}(n \cdot m)$ zu lösen. Allerdings ist die Laufzeit je nach Implementierung quadratisch in der Anzahl der Knoten und bei dichten Graphen sogar kubisch, da dann $m \approx n^2$ gilt. Daher wurden seither immer wieder neue Methoden vorgestellt, die das Kürzeste-Wege-Problem schneller lösen sollten. Jedoch stellen viele dieser Methoden bestimmte Anforderungen an den Eingabegraphen oder funktionieren nur in parallelen Systemen. Zusätzlich war es lange ungewiss, ob es sogenannte nahe-lineare Algorithmen für dieses Problem gibt, die also in $\mathcal{O}(n^{1+\epsilon})$ laufen, wobei $\epsilon > 0$ beliebig klein ist.
Bernstein et. al. konnten diese Aussage bestätigen, indem sie einen inkrementellen Algorithmus vorstellen, der kürzeste Pfade in Graphen mit ganzzahligen Kantenkosten von einem Ursprungsknoten zu allen anderen Knoten in $\mathcal{O}(m \cdot \log^8(n) \cdot \log W)$ löst und dabei auch negativ-gewichtete Zyklen erkennt. Hierbei ist W eine weitere Konstante, die im Verlauf dieser Arbeit noch vorgestellt wird.

Ziel dieser Arbeit ist es, herauszufinden, wie der Algorithmus von Bernstein, der bisher nur als Pseudocode vorliegt, praktisch implementiert werden kann und wie sich die verbesserte Komplexität gegenüber bestehenden Algorithmen auf dessen Laufzeit auswirkt. Darüber hinaus wird eine von Grund aufbauende und verständliche Erklärung des Bernstein-Algorithmus präsentiert.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen zu Graphen: Hier werden Grundlagen zu Graphen, dem Kürzesten-Wege-Problem sowie zu weiteren, nicht direkt das Kürzeste-Wege-Problem betreffende Algorithmen, vorgestellt, die in darauffolgenden Kapiteln vorausgesetzt werden.

Kapitel 3 – Verwandte Arbeiten: Gibt einen Überblick über bestehende Algorithmen zur Bestimmung kürzester Wege in Graphen mit beliebigen Kantenkosten. Die zum späteren Vergleich herangezogenen Algorithmen werden dabei detaillierter beschrieben.

Kapitel 4 – Bernstein-Algorithmus: Dieses Kapitel dient einer verständlichen und praxisorientierten Erklärung des Algorithmus von Bernstein et. al. [BNWN22]

Kapitel 5 – Implementierung: Hier werden Details zur Implementierung des zuvor beschrieben Algorithmus von Bernstein geklärt und Optimierungen vorgeschlagen.

Kapitel 6 – Experimente: Führt Laufzeitmessungen auf drei verschiedenen Graphklassen durch und vergleicht die Resultate des Bernstein-Algorithmus mit denen des Bellman-Ford Algorithmus und Goldberg-Algorithmus.

Kapitel 7 – Zusammenfassung und Ausblick: Fasst die Ergebnisse zusammen und bietet einen Ausblick für weitere Schritte in der Wissenschaft.

2 Grundlagen zu Graphen

In diesem Kapitel werden Grundlagen zu Graphen vorgestellt, die in darauffolgenden Kapiteln vorausgesetzt werden. Außerdem werden Algorithmen zur Lösung von Teilproblemen, die von später vorgestellten Algorithmen zur Bestimmung kürzester Pfade verwendet werden, erläutert.

2.1 Graphentheorie

Ein gewichteter, gerichteter Graph G ist ein Tupel (V, E, w) mit Knotenmenge V und Kantenmenge E bestehend aus Tupeln $e = (u, v)$ mit $u, v \in V$. Die Kostenfunktion $w : E \to \mathbb{R}$ definiert für jede Kante ein Kantengewicht. Die Begriffe Gewicht und Kosten sind dabei als Synonyme anzunehmen.
Wir definieren außerdem die Knoten- bzw. Kantenanzahl als $n := |V|$ und $m := |E|$.
Ein *Pfad* ist ein Tupel von Knoten $(v_1, v_2, ..., v_k)$, wobei aufeinanderfolgende Knoten im Tupel durch eine Kante $(v_i, v_{i+1}) \in E$ im Graphen verbunden sein müssen. Die Länge $w(P)$ eines Pfades P wird beschrieben durch die Summe der Kantengewichte dieser Kanten. Wir verwenden die Notation $\mathrm{dist}_G(s, t)$ für die Länge des kürzesten Pfades von s nach t, d.h. des Pfades mit minimaler Länge.
Ein *Zyklus* ist ein Pfad mit der Eigenschaft $v_1 = v_k$. Wenn ein Graph keinen Zyklus enthält, wird er als azyklisch bezeichnet. In unserem Fall verwenden wir auch die Bezeichnung Directed Acyclic Graph (DAG), da wir mit gerichteten Graphen arbeiten.
Darüber hinaus sprechen wir von einem *Vorgänger* $x \in V$ eines Knoten $u \in V$, wenn eine Kante $(x, u) \in E$ existiert. Genauso ist x *Nachfolger* eines solchen Knotens, wenn eine Kante $(u, x) \in E$ existiert.
Wenn wir von einer positiven Kante sprechen, ist derjenigen Kante ein Kantengewicht von $w(e) \geq 0$ zuzuordnen. Dementsprechend wird eine Kante mit $w(e) < 0$ als negative Kante bezeichnet. Die Menge aller negativer Kanten nennen wir E^{neg}.
Ein Graph $G' = (V', E', w)$ ist Teilgraph von G, wenn $V' \subseteq V$ und $E' \subseteq E$. Mit der Formulierung $G \setminus E_1$ ist der Graph G gemeint, aus dem die Kantenmenge $E_1 \subseteq E$ entfernt wurde. Ähnlich beschreibt $G[V_1]$ den durch V_1 induzierten Teilgraphen von G also den Teilgraphen, der nur aus der Knotenmenge $V_1 \subseteq V$ besteht, und nur jene Kanten beibehält, die auch zwischen Knoten aus V_1 liegen.
Eine stark zusammenhängende Komponente, englisch Strongly Connected Component (SCC), ist ein induzierter Teilgraph $G[S]$, für den gilt: Für zwei beliebige Knoten u, v aus S existiert

in $G[S]$ ein Pfad von u nach v und von v nach u. Zudem muss ein SCC maximal sein, d.h. es darf keine größere Knotenmenge $R \supset S$ geben, für die die Eigenschaft auch erfüllt ist. Der Durchmesser eines SCCs ist der längste aller kürzesten Pfade zwischen zwei beliebigen Knoten innerhalb des SCCs.
Für SCCs gilt zudem folgendes Lemma:

Lemma 1 (DAG-Kanten)
Komprimiert man jeweils alle SCCs eines gerichteten Graphen G zu einem Knoten, so bildet der entstandene Graph G' einen DAG.

Proof: Nehme man an, G' bilde keinen DAG. Dann gäbe es einen Zyklus in G'. Sei also mit $(v_1, v_2, ..., v_k, v_1)$ ein solcher Zyklus gegeben, der die Knotenmenge $S = \{v_1, ..., v_k\}$ umfängt. Jedoch bildet nun der induzierte Graph $G'[S]$ einen SCC. Dies widerspricht jedoch der Annahme, da SCCs maximal sein müssen und somit dieser SCC schon vorher hätte zu einem Knoten komprimiert werden müssen. □

Unter einer topologischen Sortierung versteht man eine Anordnung der Knotenmenge V eines Graphen als Tupel $(v_{k_1}, ..., v_{k_n})$, sodass für jeden Knoten $v_i \in V$ seine Nachfolger später in der Anordnung erscheinen als er selbst. Es ist darauf hinzuweisen, dass eine topologische Sortierung nur bei azyklischen Graphen möglich ist.

2.2 Kürzeste-Wege-Problem

Es gibt verschiedene Variationen des Problems des kürzesten Pfades, englisch *Shortest Path Problem*, die sich je nach Gegebenheiten des Graphs und Anforderungen an das Resultat unterscheiden.
Varianten des Kürzester-Wege-Problems ergeben sich beispielsweise aus folgenden Fragestellungen:

- Arbeitet man mit gewichteten Graphen?
- Arbeitet man mit gerichteten Graphen?
- Sind negative Kantenkosten erlaubt?
- Sucht man kürzeste Pfade von einem bestimmten Knoten zu allen anderen Knoten (Single-Source Shortest Path Problem (SSSP-Problem)) oder sucht man kürzeste Pfade von jedem Knoten zu allen anderen Knoten (All-Pair Shortest Path Problem (APSP-Problem))?

In dieser Arbeit arbeiten wir ausschließlich mit gerichteten und gewichteten Graphen mit beliebigen Kantenkosten und beschränken uns auf das SSSP-Problem.
Da wir auch mit negativen Kantenkosten arbeiten, kann es passieren, dass ein Graph einen

negativen Zyklus enthält. Dies führt jedoch dazu, dass nicht definierbare kürzeste Pfade entstehen können, weil ein kürzester Pfad, der am negativen Zyklus entlang führt, durch Iteration über diesen Zyklus beliebig oft verkürzt werden kann. Daher ist es eine meist hinzugefügte Anforderung, in diesem Fall eine Fehlermeldung oder alternativ mindestens einen negativen Zyklus zurückzugeben.
Zusammengefasst lassen sich diese Anforderungen in Definition 2.2.1 ausdrücken.

Definition 2.2.1 (Kürzester Pfad Problem)
***Eingabe**: Graph $G = (V, E, w)$ und ein Knoten $s \in V$*
***Aufgabe**: Gebe eine Fehlermeldung zurück, wenn ein negativer Zyklus existiert. Finde ansonsten für jeden Knoten v einen Pfad von s nach v mit minimaler Länge.*

2.3 Bestimmung einer topologischen Sortierung

In Abschnitt 2.1 wird beschrieben, was eine topologische Sortierung ist. Definition 2.3.1 fasst dies noch einmal zusammen.

Definition 2.3.1 (Topologische Sortierung)
***Eingabe**: DAG (V, E, w)*
***Aufgabe**: Finde eine Anordnung $(v_{k_1}, ..., v_{k_n})$ der Knotenmenge V, sodass für alle Nachfolger v_{k_j} eines Knotens v_{k_i} gilt: $j > i$*

Es gilt nun, einen effizienten Algorithmus zu beschreiben, der eine topologische Sortierung herstellt. Für dieses Problem findet man in der Literatur zahlreiche Algorithmen. Da es in dieser Arbeit um sequentielle Algorithmen zur Findung kürzester Pfade geht, möchte ich mich auch bei diesem Problem auf sequenzielle Algorithmen beschränken.
Zwei etablierte Algorithmen sind der Algorithmus von Kahn [Kah62] und ein Algorithmus basierend auf einer Tiefensuche Depth First Search (DFS), der von Tarjan [Tar76] stammt. Beide haben eine asymptotische Laufzeit von $\mathcal{O}(n + m)$.
Kahns Algorithmus führt eine Liste von Knoten, die keine eingehenden Kanten haben. Da es sich beim Eingabegraphen um einen DAG handelt, muss die Liste initial nicht-leer sein.
In jeder Iteration wird ein Knoten u dieser Liste entnommen, an die topologische Sortierung angehängt, und die ausgehenden Kanten (u, v) werden betrachtet und aus dem Graphen entfernt. Wenn v keine anderen Vorgänger außer u hat, weiß man, dass v in der topologischen Sortierung direkt nach u folgen kann, wodurch v in die Liste eingefügt werden kann. Ansonsten muss es noch andere Knoten geben, die vorher der topologischen Sortierung angehängt werden müssen.
Da dieser Algorithmus jedoch eine initiale Liste der Knoten ohne eingehende Kanten verlangt, die jedoch nicht immer ohne Überprüfung aller Knoten verfügbar ist, stellen wir im Folgenden den auf der Tiefensuche basierenden Algorithmus 2.1 vor.
Die Funktion VISIT(u) hat die Nachbedingung, dass alle Nachfolgerknoten von u nach Aufruf die richtige topologische Sortierung besitzen. Dies wird dadurch gewährleistet, dass auf

Algorithmus 2.1 DFS Topologische Sortierung

```
order ← empty stack (LIFO) used for topological sorting
toDo ← list of all nodes
while toDo not empty do
    extract a node u out of toDo
    VISIT(u)
end while
return order
function VISIT(u)
    if not toDo contains u then
        return
    end if
    for all outgoing edges (u, v) do
        VISIT(v)
    end for
    order.push(u)
    toDo.remove(u)
end function
```

Nachfolgeknoten v ein rekursiver Aufruf von VISIT gestartet wird und erst nach Abschluss aller Rekursionsaufrufe der Knoten u an den Kopf der `order` Liste angehängt wird. Dadurch müssen Nachfolgerknoten schon zuvor angehängt worden sein.
Da bei einem beliebigen Knoten gestartet wird, muss nun sichergestellt werden, dass alle Knoten abgearbeitet werden. Hierfür wird eine `toDo`-Liste geführt, die alle noch nicht bearbeiteten Knoten enthält und man iteriert über übrig gebliebenen Einträge von `toDo`. Am Ende jedes VISIT-Aufrufs wird der entsprechende Knoten aus `toDo` entfernt, sodass die while-Schleife definitiv terminiert.
Die asymptotische Laufzeit von $\mathcal{O}(n + m)$ kommt dadurch zustande, dass jeder Knoten genau einmal betrachtet wird und für jeden Knoten alle ausgehenden Kanten genau einmal betrachtet werden, also in der Summe alle Kanten genau einmal betrachtet werden. Es ist jedoch darauf hinzuweisen, dass diese Laufzeit nur dann erfüllt ist, wenn alle Operationen auf den vorhandenen Datenstrukturen in konstanter Zeit ausgeführt werden.

2.4 Bestimmung der SCCs eines Graphen

In diesem Abschnitt geht es darum, einen Algorithmus zu beschreiben, der SCCs in einem Graphen bestimmt. Unser Interesse richtet sich außerdem auf die Bestimmung aller Kanten, die zwei SCCs verbinden.

Definition 2.4.1 (SCC Bestimmung)
***Eingabe**: Graph (V, E, w)*
***Aufgabe**: Bestimme Mengen $S_1, ..., S_k \subseteq V$, sodass $\forall 1 \leq i \leq k : G[S_i]$ ist ein SCC. Bestimme zusätzlich die Kantenmenge $E' \subseteq E$ mit der Eigenschaft:*

$$(u, v) \in E' \iff \forall\, 1 \leq i \leq k : (u \in S_i \implies v \notin S_i) \tag{2.1}$$

Dank Tiefensuche bieten sich einige Algorithmen an, die in linearer Zeit laufen. Einer der verbreitetsten Algorithmen stammt von R. Tarjan [Tar72], der SCCs in $\mathcal{O}(n + m)$ bestimmt. Dieser wird nun vorgestellt und an die zusätzliche Anforderung angepasst.
Zuerst definieren wir die Menge `sepEdges` der Verbindungskanten zweier SCCs.
Wie bei der topologischen Sortierung beginnt die Tiefensuche bei einem beliebigen Knoten. Die bei der Suche gefundenen Knoten werden in einen Stack eingefüllt und dieser Reihenfolge nach mit einem `index` nummeriert. Um Zusammenhangskomponente zu erkennen, wird zusätzlich für jeden Knoten der niedrigste Index des Knotens gespeichert, der mit dem aktuellen Knoten über den bisher gefundenen Suchbaum einen Zyklus bildet. Diesen Index nennen wir `lowLink`. Eine maximale Zusammenhangskomponente ist dann gefunden, wenn der `index` des bearbeiteten Knotens u auch dem `lowLink`-Knoten entspricht. Dann ist dieser Knoten die Wurzel eines neuen SCCs, weil nach Definition von `lowLink` kein Zyklus zu einem tieferen Knoten im Stack existiert. Sobald ein solcher SCC erkannt wird, werden solange Knoten aus dem Stack entnommen, bis man beim aktuellen Knoten u ankommt, wobei alle entnommenen Knoten einschließlich des aktuellen Knotens ein SCC bilden.
Es gibt nun drei Fälle, die zu beachten sind, wenn bei der Tiefensuche die ausgehenden Kanten (u, v) eines Knotens u betrachtet werden.

- Wenn der Zielknoten v noch nie besucht wurde, wird per Tiefensuche ein rekursiver Aufruf auf dem Knoten gestartet. Da über den nachfolgenden Teilbaum ein Zyklus mit einem schon im Stack befindlichen Knoten gebildet werden könnte, wird `u.lowLink` auf das Minimum des aktuellen Wertes und des Wertes von `v.lowLink` gesetzt. Wenn außerdem der Zielknoten v nach der Rekursion nicht mehr auf dem Stack ist, bedeutet dies, dass er zuvor in einen SCC eingefügt wurde. Also kann in diesem Fall die Kante (u, v) zur Menge `sepEdges` hinzugefügt werden.
- Wenn der Zielknoten v schon besucht wurde und sich im Stack befindet, haben wir einen Zyklus gefunden und setzen daher `u.lowLink` auf das Minimum des aktuellen Wertes und des Index von v.
- Wenn der Zielknoten v schon besucht wurde, sich aber nicht auf dem Stack befindet, so muss er zu einem schon bearbeiteten SCC gehören. Wir wissen also, dass die Kante (u, v) zwei verschiedene SCCs verbindet und fügen sie in die Menge der `sepEdges` ein.

Der hier in Worten beschriebene Algorithmus ist in Algorithmus 2.2 dargestellt.

Algorithmus 2.2 Tarjan SCC Algorithmus

```
SCCs ← ∅ set of SCCs
stack ← depth first search stack
sepEdges ← separator edges
toDo ← list of all nodes
index ← 0
while toDo not empty do
    extract a node u out of toDo
    VISIT(u)
end while
function VISIT(u)
    toDo.remove(u)
    u.index ← index
    u.lowLink ← index
    index ← index +1
    stack.push(u)
    u.onStack ← true
    for all outgoing edges (u, v) do
        if v ∈ toDo then
            VISIT(v)
            if not v.onStack then
                sepEdges ∪ {(u, v)}
            end if
            u.lowLink ← min(u.lowLink, v.lowLink)
        else if v.onStack then
            u.lowLink ← min(u.lowLink, v.index)
        else
            sepEdges ∪ {(u, v)}
        end if
    end for
    if u.lowLink = u.index then
        scc ← ∅ new scc
        repeat
            next ← stack.pop()
            scc ∪ {next}
        until next = u
        SCCs.add(scc)
    end if
end function
```

3 Verwandte Arbeiten

3.1 Bellman-Ford Algorithmus

Der wohl bekannteste Kürzester-Wege Algorithmus für beliebige Kantenkosten ist der Bellman Ford Algorithmus, benannt nach R. Bellman und L. Ford [Bel58], [For56]. Dessen Idee ist simpel. Für jeden Knoten wird die Distanz $d(v)$ vom Ursprungsknoten s gespeichert, wobei initial für alle Knoten $d(v) = \infty$ und $d(s) = 0$ gesetzt wird. Man betrachte nun wiederholt alle Kanten (u, v) und prüft, ob über die Kante eine Verbesserung der Distanz des Zielknotens $d(u) + w(u, v) < d(v)$ vorliegt. Wenn dies der Fall ist, wird die Distanz dementsprechend angepasst.
Nach der i-ten Iteration der Überprüfung aller Kanten ist gewährleistet, dass die Distanz zu jedem Knoten u höchstens so hoch ist wie die Länge des kürzesten Pfades von s nach u, der maximal i Kanten enthält, falls ein solcher Pfad existiert. Denn wenn ein kürzester Pfad von s nach u mit maximal i Kanten existiert, so kann es zwar einen kürzeren Pfad mit mehr als i Kanten geben, die Distanz zu u darf aber nach Induktion über die Iterationen nicht größer sein als die Länge dieses Pfades.
Da ein kürzester Pfad im Graphen maximal $n-1$ Kanten enthält, müssen alle Kanten $n-1$-mal überprüft werden, bis die Distanzen unverändert bleiben.
Aufgrund dieser Feststellung kann eine n-te Iteration über alle Kanten dafür genutzt werden, negative Zyklen zu erkennen. Denn wenn in der n-ten Iteration eine Veränderung der Distanzen beobachtet werden kann, muss dies auf einen negativen Zyklus zurückzuführen sein.
Zusätzlich zum Beschriebenen speichert Algorithmus 3.1 für jeden Knoten den Vorgänger $p(v)$ im kürzesten Pfad nach v, welcher bei jeder Verbesserung einer Distanz angepasst wird. Damit lässt sich nach Durchlauf des Algorithmus nicht nur die Länge des kürzesten Pfades, sondern auch der Pfad selbst, bestimmen.
Die asymptotische Komplexität von Bellman-Ford beträgt offensichtlich $\mathcal{O}(n \cdot m)$. Nun gibt es einige Verbesserungen, die in der Praxis zu besseren Laufzeiten führen.

Der von E. Moore in [Moo59] beschriebene und später inoffiziell zu Shortest Path Faster Algorithm (SPFA) benannte Algorithmus verwendet dasselbe Prinzip wie Bellman-Ford, basiert aber auf der Tatsache, dass viele Kanten mit hoher Wahrscheinlichkeit unnötigerweise überprüft werden. Beispielsweise werden in der ersten Iteration von Bellman-Ford alle Kanten betrachtet, obwohl nur die ausgehenden Kanten vom Startknoten relevant sind.
Daher wird eine Queue verwendet, die die Randknoten speichert, für die die Distanz ungewiss ist. Bei einem Update einer Distanz wird der Zielknoten in die Queue eingefügt. So werden

Algorithmus 3.1 Bellman-Ford-Algorithmus

```
d(v) ← ∞   ∀v ∈ V \ {s}
d(s) ← 0
p(v) ← unset   ∀v ∈ V
for i ← 0; i < n − 1; i ← i + 1 do
    for all (u, v) ∈ E do
        if d(u) + w(u, v) < d(v) then
            d(v) ← d(u) + w(u, v)
            p(v) ← u
        end if
    end for
end for
for all (u, v) ∈ E do
    if d(u) + w(u, v) < d(v) then
        return Error: Negative cycle found.
    end if
end for
return d(v), p(v)   ∀v ∈ V
```

Kanten, die nicht zum Ergebnis beitragen, nicht bearbeitet. In den meisten Anwendungsfällen verbessert sich die Laufzeit von SPFA gegenüber Bellman-Ford, die Worst-Case Komplexität bleibt aber bestehen (vgl. [CGR94]).

Eine Erweiterung von Pape [Pap74], bekannt unter dem Namen D' Escopo-Pape-Algorithmus kommt durch die Verwendung einer Deque zustande. Bei einem Update einer Distanz wird der Knoten ans Ende oder an den Anfang eingefügt je nachdem, ob der Knoten schon einmal in der Deque gespeichert wurde oder nicht. Während die Laufzeit womöglich dadurch im Durchschnitt weiter gesteigert wird, konnte allerdings gezeigt werden, dass die Worst-Case Komplexität exponentiell ist [SW81].

3.2 Verwendung von Potenzialfunktionen

Ein häufig verwendetes Hilfsmittel sind Potenzialfunktionen $\phi : V \to \mathbb{R}$, die von der Knotenmenge auf eine reelle Zahl abbilden. Eine neue Kostenfunktion w_ϕ bildet sich aus der alten Kostenfunktion und einer Potenzialfunktion und ist für eine Kante $e = (u, v)$ folgendermaßen definiert:

$$w_\phi(e) = w(e) + \phi(u) - \phi(v) \quad \forall e \in E \tag{3.1}$$

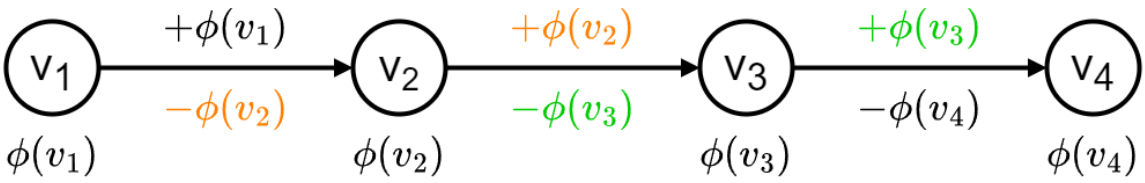


Abbildung 3.1: Auswirkung einer Potenzialfunktion auf die Pfadlänge

Lemma 2
Für zwei Potenzialfunktionen ϕ und θ gilt:

$$(w_\phi)_\theta(e) = w_{\phi+\theta}(e)$$

Definition 3.2.1
*Eine Kantenmenge E nennen wir **behoben**, wenn eine Potenzialfunktion ϕ gefunden ist, sodass gilt: $\forall e \in E : w_\phi(e) \geq 0$*
*Eine negative Kante e (mit $w(e) < 0$) gilt als **eliminiert**, wenn eine Potenzialfunktion ϕ gefunden ist, sodass $w_\phi(e) \geq 0$.*

Wir definieren einen neuen Graphen $G_\phi = (V, E, w_\phi)$ und sprechen vom potenzierten Graphen. Dieser bringt zwei nützliche Eigenschaften mit sich:

1. Sei $p = (v_1, ..., v_n)$ ein beliebiger Pfad in G und sei w die Summe der Kantenkosten der enthaltenen Kanten. Dann ist die resultierende Pfadlänge nach Anwendung einer beliebigen Potenzialfunktion definiert durch $w_\phi = w + \phi(v_1) - \phi(v_n)$.
   Das liegt daran, dass sich die Potenziale in der Mitte des Pfades gegenseitig aufheben, wie Abbildung 3.1 veranschaulicht.
   Handelt sich es bei dem Pfad um einen Zyklus, so bleibt folglich die Pfadlänge gleich.
2. Eine Folgerung aus diesen Aussagen ist, dass auch die kürzesten Pfade in G_ϕ erhalten bleiben, da alle Pfadlängen von einem Knoten s zu einem Knoten t um denselben Wert $\phi(s) - \phi(t)$ verschoben wurden. Außerdem ist ein negativer Zyklus in G auch in G_ϕ vorhanden.

Mit diesen Informationen lässt sich ein Kürzester-Wege-Algorithmus für negative Kantenkosten definieren. Dafür nehmen wir an, dass es einen Algorithmus FixEdges(G) gibt, der eine Potenzialfunktion ϕ zurückgibt, die alle negativen Kanten in E eliminiert.

Hierbei ist zu beachten, dass zwar die kürzesten Pfade, die Dijkstra generiert, mit den kürzesten Pfaden in G übereinstimmen, die Distanzen jedoch durchaus unterschiedlich sein können. Um die Distanz eines kürzesten Pfades in G zu erhalten, muss also vom durch Dijkstra berechneten Pfad das Potenzial des Startknotens subtrahiert und das Potenzial des Zielknotens addiert werden.

Algorithmus 3.2 Kürzester Weg via Potenzialfunktion

Input: $G = (V, E, w)$
Output: Kürzester Weg Baum
$\phi \leftarrow$ FixEdges(G)
$w_\phi \leftarrow$ applyPriceFunction(w, ϕ)
return $Dijkstra(G_\phi)$

Da Dijkstras Algorithmus in $\mathcal{O}(n \cdot log(n) + m)$ [Dij59], [FT87] läuft und Bellman-Ford eine Worst-Case-Laufzeit von $\mathcal{O}(n \cdot m)$ hat, bringt dieser Algorithmus auch nur dann einen Vorteil mit sich, wenn FixEdges schneller als Bellman-Ford und am Besten in nahe-linearer Zeit läuft. Wir werden anhand von Goldbergs und Bernsteins Algorithmus zwei Implementierungen sehen, die die Anforderungen von FixEdges erfüllen.

3.3 Skalierungsalgorithmen

Das nun vorgestellte Konzept der Skalierung von Parametern wurde erstmals von Edmonds und Karp [EK72] in Bezug auf das Min-Cost-Flow-Problem vorgestellt. Garbow [Gar85] verwendet das Konzept, um einen effizienten Algorithmus für das Maximum-Weight-Matching Problem und durch Reduktion schließlich auch für das Kürzeste-Wege-Problem mit beliebigen Kantenkosten vorzustellen. Der Algorithmus läuft dabei in $\mathcal{O}(n^{3/4} m \log W)$, wobei W der Betrag des kleinsten negativen Kantengewichts ist und ganzzahlige Kantenkosten gefordert sind.
Allgemein lässt sich ein Skalierungsalgorithmus auf folgende Weise rekursiv beschreiben (vgl. [Gar85]):

> Given a network problem, halve the numeric parameters (i.e., a number l becomes $\lfloor l/2 \rfloor$) and solve this scaled-down problem recursively. Double the solution to get a near-optimum solution on the original network. Then transform the near-optimum solution to optimum.

Im Bezug auf das Kürzeste-Wege-Problem sind diese Parameter die Kantenkosten.
Da Garbow in [Gar85] wie schon gesagt das Kürzeste-Wege-Problem auf das Maximum-Weight-Matching-Problem für bipartite Graphen reduziert, befindet sich der Skalierungsbestandteil im Algorithmus von Maximum-Weight-Matching, auf den ich an dieser Stelle aber nicht genauer eingehen möchte. Allerdings werde ich kurz die Erweiterung auf das Kürzeste-Wege-Problem andeuten. Garbow stellt ein Schema vor, aus einem Eingabegraphen einen bipartiten Graph zu erzeugen. Er zeigt, dass die sog. doppelten Werte $y(i)$ für alle Knoten i, die beim Maximum-Weight-Matching entstehen, die Eigenschaft $\forall (u, v) \in E : y(u) + w(u, v) \geq y(v)$ erfüllen. Durch Verwendung dieser Funktion als Potenzialfunktion auf den Kantenkosten ($w_y(u, v) = w(u, v) + y(u) - y(v)$) wird erreicht, dass $w_y(u, v)$ nach Definition von y nicht-negativ ist, wodurch Dijkstra angewendet werden kann. Somit handelt es sich bei dieser

Methode um eine Variante des Algorithmus 3.2.
Goldbergs Algorithmus, der in Abschnitt 3.4 ausführlich vorgestellt wird, verwendet auch das Skalierungsprinzip und kommt auf eine Laufzeit von $\mathcal{O}(m\sqrt{n}\log(W))$.
Im Jahr 2017 wurde von Cohen et al. [CMSV17] ein weiterer Skalierungsalgorithmus vorgestellt. Dieser löst das Minimum-Cost Flow Problem, b-Matching Problem auf bipartiten Graphen und mithilfe derselben oben angesprochenen Reduktion von Garbow auch das Kürzeste-Wege Problem in $\mathcal{O}(m^{10/7}\log W)$ und bietet eine der besten asymptotische Laufzeiten für Graphen mit wenigen Kanten (sog. sparse graphs).

3.4 Goldberg-Algorithmus

Goldberg stellte im Jahr 1995 einen weiteren Skalierungsalgorithmus [Gol95] vor, der mit einer Laufzeit von $\mathcal{O}(m\sqrt{n}\log(W))$ bisherige Algorithmen unterbietet. Ich beziehe mich bei der Vorstellung auch auf die Notizen von U. Zwick [Zwi08], der weitere Details aufklärt.
Der Grundgedanke ist es, eine Potenzialfunktion zu finden, die negative Kanten eliminiert, sodass anschließend Dijkstra angewendet werden kann (siehe Abschnitt 3.2). Da es sich als einfacher erweist, dieses Problem für größere negative Kantenkosten zu lösen, skaliert man zuerst die Kosten herunter, genauer gesagt definiert man Kosten $w'(e) = \lceil w(e)/2 \rceil$.
Es kann nun gezeigt werden, dass eine Potenzialfunktion ϕ', die negative Kanten in w' eliminiert, dazu verwendet werden kann, eine Potenzialfunktion ϕ zu definieren, die negative Kanten in w eliminiert (vgl. [Zwi08]). Dazu wählt man für alle Knoten $u \in V$ Potenziale $\phi(u) = 2 \cdot \phi'(u)$:

$$\begin{aligned} w_\phi(u,v) &= w(u,v) + \phi(u) - \phi(v) \\ &\geq (2w'(u,v) - 1) + 2\phi'(u) - 2\phi'(v) \end{aligned}$$

Mit $w'(u,v) + \phi'(u) - \phi'(v) \geq 0$:

$$w_\phi(u,v) \geq -1$$

Es bleibt nun einzig übrig, einen Algorithmus Refine zu finden, der eine Potenzialfunktion findet, die in einem Graphen mit Kantenkosten $w(e) \geq -1$ alle negativen Kanten eliminiert. Wenn ein negativer Zyklus in diesem Graphen gefunden wird, so handelt es sich auch um einen negativen Zyklus im Originalgraphen.
Der Skalierungsalgorithmus ist in Abschnitt 3.3 dargestellt. Recurse sorgt dafür, dass die Gewichte in $\mathcal{O}(\log W)$ Rekursionen herunterskaliert werden und aus einer Potenzialfunktion ϕ', die negative Kanten im skalierten Graphen $G' = (V, E, w')$ eliminiert, eine Potenzialfunktion zu definieren, die negative Kanten im gegebenen Graphen G eliminiert.

Algorithmus 3.3 Goldberg Skalierung $(G = (V, E, w))$

return RECURSE(w)
function RECURSE(w)
  `min` $\leftarrow$ minimal weight in w
  if `min` < -1 **then**
    $w'(e) \leftarrow \lceil w(e)/2 \rceil \quad \forall e \in E$
    $\phi' \leftarrow$RECURSE(w')
    $\phi \leftarrow$REFINE($w_{\phi'}$) // $\forall e \in E : w_{\phi'}(e) \geq -1$
    return $2 \cdot \phi' + \phi$ // $(G_{2\cdot\phi'})_\phi = G_{2\cdot\phi'+\phi}$
  else
    return REFINE(w)
  end if
end function

3.4.1 Eliminierung negativer Kanten für Eingabekosten $w(e) \geq -1$

Der nun erläuterte Algorithmus REFINE zur Eliminierung negativer Kanten in einem Eingabegraphen mit Gewichten $w(e) \geq -1$ läuft in $\mathcal{O}(m\sqrt{n})$. Dadurch ergibt sich für den gesamten Algorithmus automatisch eine Laufzeit von $\mathcal{O}(m\sqrt{n}\log W))$.
Zuerst wird der Graph $G^- = (V, E^-, w)$ definiert, der nur aus der Teilmenge von Kanten $E^- \subseteq E$ besteht für die gilt: $w(e) \leq 0$.

Graph-Komprimierung

Im weiteren Verlauf des Algorithmus soll der Graph azyklisch sein, um negative Kanten einfacher zu eliminieren. Man beachte, dass in G^- nur entweder negative Zyklen oder Zyklen der Länge 0 existieren können.
Indem man auf dem komprimierten Graphen arbeitet, der Knoten eines SCCs zu einem neuen Knoten zusammenfasst, sind nach Lemma 1 Zyklen ausgeschlossen. Das ist zulässig, da die Anwendung von Potenzialfunktionen auf dem komprimierten Graphen keine Längen von Zyklen des Originalgraphen verändert. Die Berechnung der SCCs kann dabei in linearer Zeit erfolgen (vgl. Algorithmus 2.2). Währenddessen können auch negative Zyklen erkannt werden, da Kanten in G^-, die innerhalb von SCCs verlaufen, keine negativen Gewichte haben dürfen. Es sei jedoch zu beachten, dass an dieser Stelle nicht unbedingt alle negativen Zyklen erkannt werden, da der Test nicht auf G erfolgt. Mit G^- ist von nun an der komprimierte Graph G^- gemeint, der aber auch die positiven Kanten beinhaltet.

Grundidee zur Eliminierung negativer Knoten

Sei im Folgenden ein negativer Knoten ein Knoten mit mindestens einer negativen eingehenden Kante. Lemma 1 zeigt, dass der komprimierte Graph azyklisch ist. Man macht sich diese Tatsache zu Nutze, da man so einen negativen Knoten v eliminieren kann ohne neue negative Kanten zu generieren, indem man für v und alle von v aus erreichbaren Knoten das Potenzial auf -1 setzt. Mehrere negative Knoten können also nacheinander eliminiert werden, indem sukzessiv die Potenziale der genannten Knoten um 1 verringert werden.
Es ist einfach zu zeigen, dass dieses Verfahren auf G^- nicht dazu führen kann, dass in G ehemalige positive Kanten negativ werden.
Da man jedoch im Worst-Case Szenario n negative Knoten eliminieren und jeweils für $\mathcal{O}(m)$ Kanten das Potenzial verringern muss, ergäbe sich dadurch eine Laufzeit von $\mathcal{O}(n \cdot m)$, was keine Verbesserung zu Bellman Ford liefern würde.

Gleichzeitige Eliminierung von Knoten

Um mehrere negative Knoten gleichzeitig zu eliminieren, behilft man sich eines Tricks. Man füge in G^- einen Hilfsknoten s und für jeden Knoten in V Kanten (s, v) mit Gewicht 0 ein (siehe Definition 4.1.1). Die Knoten des Graphen werden nun in Ebenen L_i einsortiert, die folgendermaßen definiert sind:

$$L_i = \{v \in V : \mathrm{dist}_{G_s^-}(s, v) = -i\}$$

Man nehme nun an, man habe aktuell k negative Knoten und betrachte den Index D der höchsten Ebene, also die negativste auffindbare Distanz eines Knotens von s.
Es gibt nun zwei Fälle die getrennt voneinander behandelt werden müssen:

1. Wenn $D \leq \sqrt{k}$, muss es eine Ebene geben, die mindestens $\sqrt{k}$ negative Knoten enthält. Das liegt daran, dass ansonsten alle Ebenen weniger als $\sqrt{k}$ negative Knoten enthalten würden, es aber maximal $\sqrt{k}$ Ebenen gibt. Damit wäre die Summe der negativen Knoten kleiner als k was ein Widerspruch ist.
2. Wenn $D > \sqrt{k}$, muss es einen Pfad in G^- geben, der mindestens $\sqrt{k}$ negative Knoten enthält. Das folgt aus Definition von D und der Tatsache, dass für alle Kanten $w(e) \geq -1$ gilt.

Der erste Fall ist einfach zu behandeln. Um die negativen Knoten einer Ebene L_i zu eliminieren, verringert man einfach das Potenzial aller Knoten der Ebenen $L_{\geq i} = \bigcup_{j \geq i} L_j$ um 1. Dies kann in $\mathcal{O}(m)$ geschehen. Die Argumentation funktioniert gemäß der allgemeinen Beschreibung zur Eliminierung negativer Knoten in Abschnitt 3.4.1.
Für den zweiten Fall zitiere ich [Gol95] sinngemäß:

> Sei P der zu behandelnde Pfad in G^- und seien $d'(v)$ die Indizes der Ebenen, in denen die Knoten des Pfades eingeordnet sind. Für Knoten, die nicht Teil des Pfades sind, sei $d'(v) = 0$. Definiere eine neue Kostenfunktion $w'(e) := \max(0, w(e))$. Es wird ein Hilfsgraph G_s aus G erzeugt durch Hinzunahme eines Hilfsknotens s mit Kanten zu allen Knoten (s, v) und zugehörigem Gewicht $w'(s, v) = n + d'(v)$.
> Berechne jetzt auf dem Hilfsgraphen kürzeste Weglängen d in Bezug auf die Kostenfunktion w'. Eine neue Potenzialfunktion ϕ', die die negativen Knoten des Pfades eliminiert und keine neuen negativen Kanten erzeugt, erweitert die alte Potenzialfunktion ϕ durch: $\phi'(v) = \phi(v) + d(v) - n$.

Auch dieser Fall einschließlich der kürzesten Weglängenbestimmung kann in linearer Zeit berechnet werden, da bei der Kürzesten-Wege-Berechnung die Weglängen durch n begrenzt sind. Für den Beweis zu dieser Prozedur verweise ich auf das Paper [Gol95].
Die finale Laufzeit $\mathcal{O}(m\sqrt{n})$ von Refine kommt also dadurch zustande, dass pro Iteration von der aktuellen Anzahl negativer Knoten k mindestens $\sqrt{k}$ Stück eliminiert werden, wodurch $\mathcal{O}(\sqrt{n})$ Iterationen benötigt werden. Jede Iteration hat dabei einen Aufwand von $\mathcal{O}(m)$.

3.5 Weitere Arbeiten

Es gibt noch zahlreiche weitere Arbeiten und Algorithmen für das SSSP-Problem, die für beliebige Kantenkosten funktionieren und sich auf bestimmte Themenfelder konzentrieren:
Im Jahr 2005 stellten Yuster und Zwick [YZ05] einen SSSP-Algorithmus vor, der mithilfe von Matrixmultiplikationen kürzeste Pfadlängen bestimmt. Mit einer Laufzeit von $\mathcal{O}(W \cdot n^\omega)$ wird dadurch in der Theorie Goldbergs Algorithmus auf dichten Graphen (Graphen mit vielen Kanten im Verhältnis zur Knotenanzahl) übertroffen. Dabei ist $\omega < 2,376$ ein Exponent, der bei der Matrixmultiplikation entsteht. Es ist jedoch anzumerken, dass nicht die Pfade selbst berechnet werden und dass davon ausgegangen wurde, dass keine negativen Zyklen existieren.
Ein im Jahr 2022 veröffentlichter Algorithmus für das Minimum-Cost Flow Problem [CKL+22], der in $m^{1+o(1)}$ läuft, reduziert das SSSP-Problem und zeigt somit, dass nahe-lineare Zeiten für das Kürzeste-Wege-Problem möglich sind.

Auf der Teilmenge der planaren Graphen lassen sich noch bessere Laufzeiten finden. Eine der besten Komplexitäten wird mit $\mathcal{O}(n \log^3 n)$ im Algorithmus von Fakcharoenphol und Rao [FR01] erzielt.
Eine weitere Spezialisierung ist die Frage nach dynamischen Kürzester-Wege Algorithmen. Hierbei geht man davon aus, dass man die kürzesten Pfade für einen Graphen schon berechnet hat, und möchte nun nach der Ausführung bestimmter Graphoperationen wie dem Einfügen oder Löschen einer Kante oder der Veränderung eines Kantengewichtes die kürzesten Pfade neu berechnen, ohne den Berechnungsprozess von Grund auf zu wiederholen. Es kann gezeigt werden, dass dynamische SSSP-Problem-Algorithmen aufwendiger als dynamische APSP-Problem-Algorithmen sind (vgl. [RZ04]). Interessante Arbeiten stammen von Roddity et al. [RZ04] und Frigioni et al. [FMSN98]. Es stellt sich aber als schwierig heraus, die Arbeiten

bezüglich Laufzeit miteinander zu vergleichen, da oft unterschiedliche Anforderungen und Bedingungen formuliert werden.
Des Weiteren fällt eine Rubrik von Arbeiten unter die parallelen Algorithmen, die vorhandene sequentielle Kürzester-Wege Algorithmen für die parallele Programmierung anpassen oder entsprechend neue Algorithmen formulieren. Dabei wird versucht, die Arbeit so auf die Prozessoren aufzuteilen, dass die Komplexität des kritischen Pfades die Komplexität der Summe aller Instruktionen über die Prozessoren hinweg, unterbietet. Der kritische Pfad ist die längste Folge von Instruktionen, die nicht weiter parallelisiert werden kann. Eine aktuelle Arbeit hierzu stammt von Cao et al. [CFR22], welche auf Goldbergs Algorithmus basiert und im kritischen Pfad eine Komplexität von $\mathcal{O}(n^{5/4+o(1)} \log W)$ erreicht.

4 Bernstein-Algorithmus

In Abschnitt 3.2 wurde ein allgemeiner Ansatz vorgestellt, mithilfe von Potenzialfunktionen kürzeste Pfade zu berechnen, indem alle negativen Kanten eliminiert werden und anschließend der Dijkstra-Algorithmus auf dem äquivalenten Graphen ausgeführt wird.
Eine Umsetzung neben Goldbergs Algorithmus wird im Algorithmus von Bernstein et al. [BNWN22] vorgestellt. In diesem Kapitel geht es darum, diesen von Grund auf zu erklären. Dabei werden zunächst die wesentlichen Ideen aufgezeigt, die Hintergrund des Bernstein-Algorithmus sind. Anschließend wird die Funktionsweise des Algorithmus erklärt.
Abgesehen von Abschnitt 4.6 wird davon ausgegangen, dass im Eingabegraphen G keine negativen Zyklen vorliegen. Außerdem gelten zwei Vorbedingungen für den Eingabegraphen:

1. Für alle Kanten e gilt: $w(e) \geq -1$
2. Jeder Knoten hat konstant viele ausgehende Kanten.

Aus der zweiten Voraussetzung folgt, dass $m = \mathcal{O}(n)$, weshalb m und n in Komplexitätsaussagen gleichbedeutend sind. Man könnte auch Graphen mit linear vielen ausgehenden Kanten erlauben, indem jeder Knoten mit k ausgehenden Kanten durch k Knoten ersetzt werden, die dann zusammen über Kanten mit Gewicht 0 verbunden sind und einen SCC bilden. So kann jedem neuen Knoten eine der ausgehenden Kanten zugeteilt werden.
Der Algorithmus lässt sich auf beliebige ganzzahlige Kantenkosten erweitern, indem man Goldbergs Skalierungsalgorithmus, der in Algorithmus 3.3 vorgestellt wurde, anwendet.
Die Version des Bernstein-Algorithmus, die die Abwesenheit negativer Zyklen voraussetzt, hat eine Laufzeit von $\mathcal{O}(m \log^5(n) \log W)$. Wenn der Eingabegraph einen negativen Zyklus enthält, terminiert diese Version nicht.
Die Erweiterung dieses Algorithmus, die in Abschnitt 4.6 angedeutet wird, erkennt einen negativen Zyklus und gibt ihn zurück. Diese Version hat eine Laufzeit von $\mathcal{O}(m \log^8(n) \log W)$.

Wichtig Alle Informationen, die in diesem Kapitel übermittelt werden, stammen aus der Arbeit von Bernstein [BNWN22] und werden hier nur in einem anderen Erklärungsansatz geschildert, es sei denn es ist ausdrücklich darauf hingewiesen, dass sie aus einer anderen Quelle stammen.

4.1 Grundideen

4.1.1 Dijsktra vereint mit Bellman Ford

Ein elementarer Bestandteil des Bernstein-Algorithmus basiert auf dem nun folgenden Lemma 3, wofür zuerst die Definition des Graphen G_s benötigt wird. Die Idee dahinter stammt aus Johnsons Algorithmus [Joh77].

Definition 4.1.1 (Graph G_s)
Graph G_s entsteht aus $G = (V, E, w)$ durch Hinzunahme eines Knotens s, der Hilfsquelle genannt wird. Zusätzlich werden für alle Knoten $v \in V$ Kanten (s, v) mit jeweiligem Kantengewicht $w(s, v) = 0$ hinzugefügt.

Lemma 3 (Johnson Shift)
Sei $G = (V, E, w)$ ein gerichteter Graph ohne negative Zyklen und sei s die Hilfsquelle in G_s. Man definiere eine Potenzialfunktion $\phi(v) = \mathrm{dist}_{G_s}(s, v)$ für alle Knoten $v \in V$.
Dann gilt für alle Kanten $e \in E$: $w_\phi(e) \geq 0$

Proof: Für eine Kante (u, v) gilt:

$$\mathrm{dist}_{G_s}(s, v) \leq \mathrm{dist}_{G_s}(s, u) + w(u, v)$$

Daraus folgt:

$$\begin{aligned} w_\phi(u, v) &= w(u, v) + \phi(u) - \phi(v) \\ &= w(u, v) + \mathrm{dist}_{G_s}(s, u) - \mathrm{dist}_{G_s}(s, v) \\ &\geq w(u, v) + \mathrm{dist}_{G_s}(s, u) - (\mathrm{dist}_{G_s}(s, u) + w(u, v)) \\ &\geq 0 \end{aligned}$$

□

Um die in Lemma 3 erforderlichen Distanzen $\mathrm{dist}_{G_s}(s, v)$ zu berechnen, wird der Algorithmus *Dijkstra+BF* verwendet, der, wie sich später herausstellen wird, in der Laufzeit eine vorteilhafte Eigenschaft gegenüber anderen Algorithmen wie Bellman-Ford bietet. Dieser ist in Algorithmus 4.1 dargestellt.
Dijkstra+BF iteriert über zwei Phasen. In der ersten Phase, die auf Dijkstra basiert, wird versucht, Distanzen zu Knoten über ausschließlich nicht-negative Kanten zu verbessern. Jeder Knoten, der in dieser Phase bearbeitet wird, wird für die zweite Phase zur Bearbeitung markiert. Die zweite Phase wiederum behandelt ausschließlich negative Kanten und versucht Distanzen darüber zu verbessern. Danach wiederholt sich der Prozess solange, bis keine Veränderungen mehr stattfinden.
An dieser Stelle verwenden wir einen Trick, um den Eingabegraphen G nicht aufgrund des neuen Knotens s verändern zu müssen, denn dies würde später in der Implementierung

Algorithmus 4.1 Dijkstra+BF

```
Q ← priority queue (as min-heap) contains all nodes in V        // Initialisierungstrick
d(u) ← 0 for all nodes u ∈ V
Every node is marked
while Q not empty do
    while Q not empty do                                        // Dijkstra Phase
        Extract minimal node u out of Q
        Mark u
        for all non-negative outgoing edges (u, v) do
            if d(u) + w(u, v) ≤ d(v) then
                d(v) ← d(u) + w(u, v)
                if v ∉ Q do Q.push(v)
            end if
        end for
    end while
    for all marked nodes u do                                   // Bellman Ford Phase
        for all negative outgoing edges (u, v) do
            if d(u) + w(u, v) ≤ d(v) then
                d(v) ← d(u) + w(u, v)
                if v ∉ Q do Q.push(v)
            end if
        end for
        Unmark u
    end for
end while
return d(u)   ∀u ∈ V
```

zusätzlichen Aufwand bedeuten. Um also trotzdem die Distanzen für den in der Realität nicht vorhandenen Ursprungsknoten s zu berechnen, führen wir gedanklich die erste Iteration der Dijkstra-Phase durch und stellen fest, dass jeder Knoten $v \in V$ in den Min-Heap Q eingefügt wird. Somit kann dies abgekürzt werden, indem initial alle Knoten des Eingabegraphen G in den Min-Heap Q eingefügt und markiert werden und alle Distanzen auf 0 gesetzt werden.

Es wird nun angedeutet, warum die berechneten Distanzen korrekt sind und wie die Laufzeit zustande kommt. Für die ausführlichen Korrektheits- und Laufzeitbeweise beziehe ich mich auf die Urheber des Algorithmus [BNWN22]. Es kann gezeigt werden, dass für eine Kante (u, v), die die aktuellen Distanzen verändern könnte d.h. $d(u) + w(u, v) < d(v)$, entweder der Knoten u im Min Heap Q gespeichert ist oder markiert ist. Aus der Tatsache, dass der Algorithmus nur dann terminiert, wenn Q leer ist und kein Knoten mehr markiert sind, folgt, dass nach der Terminierung keine Kante mehr existiert, die die berechneten Distanzen verändern könnte. Dies impliziert unmittelbar die Korrektheit.
Um die Laufzeit des Algorithmus zu verstehen, bedarf es einer weiteren Definition:

Definition 4.1.2 (Funktion $\eta_G(v)$)
$\eta_G(v)$ beschreibt, die minimale Anzahl von negativen Kanten der kürzesten Pfade von s nach v auf dem Graphen G_s. Formal:

$$\eta_G(v) = \begin{cases} \infty & \text{dist}_{G_s}(s,v) = -\infty \\ \min\{|E^{neg}(G) \bigcap P|\} : P \text{ ist die Kantenmenge eines kürzesten Pfades von } s \text{ nach } v \text{ in } G_s & \text{sonst} \end{cases}$$

Der kürzeste Pfad, der diese minimale Anzahl negativer Kanten bietet, wird als $P_G(v)$ definiert. Schließlich wird definiert: $\eta(G) = \max_{v \in V} \eta_G(v)$

Eine Aussage, die an dieser Stelle nicht gezeigt wird, ist, dass die ausgehenden Kanten eines Knoten v in maximal $\eta_G(v)$ Durchläufen der Dijkstra-Phase genau einmal betrachtet werden. Weil jeder Knoten nach Voraussetzung des Algorithmus $\mathcal{O}(1)$ viele ausgehende Kanten hat, verursacht somit jeder Knoten maximal $\mathcal{O}(\eta_G(v))$ viele Einfügeoperationen. Bei der Initialisierung müssen zudem alle Knoten eingefügt werden. Also können alle Knoten insgesamt $\mathcal{O}(n + \sum_{v \in V} \eta_G(v))$ mal in Q eingefügt werden. Zusammen mit dem Aufwand des Einfügens in den Min Heap ergibt sich die Gesamtlaufzeit von

$$\mathcal{O}(log(n) \cdot (\sum_{v \in V} \eta_G(v) + n)).$$

Eine wichtige Anmerkung ist, dass im Falle von negativen Zyklen im Eingabegraphen ein Knoten v existiert, sodass $\eta_G(v) = -\infty$, wodurch dieser Algorithmus unendlich lange läuft. Wenn nun aber $\eta(G) \in \mathcal{O}(n)$ gilt, ergäbe sich eine Laufzeit von $\mathcal{O}(log(n) \cdot n^2)$, was langsamer als Bellman Ford ist. So kommt die Idee auf, die Anzahl negativer Kanten auf kürzesten Pfaden im Graphen vor der Ausführung von *Dijsktra+BF* zu minimieren.

4.1.2 Allgemeines Vorgehen

Eine Möglichkeit, die negativen Kanten auf kürzesten Pfaden zu reduzieren, ergibt sich aus Lemma 1. Durch die Berechnung der SCCs im Graphen kann man folglich die Kantenmenge E in Teilmengen E^{DAG} und E^{SCC} aufteilen, wobei die Kanten der ersten Menge zwei verschiedene SCCs verbinden, während die der zweiten Menge innerhalb von SCCs liegen. Wir bezeichnen $H = \bigcup_i G[V_i]$ als den Teilgraphen, der alle SCCs V_i vereint und damit nur Kanten E^{SCC} enthält.
Für das weitere Vorgehen wird die Definition des verschobenen Graphen G^B benötigt.

Definition 4.1.3 (Graph G^B und $G^B_{\geq 0}$)
Graph $G^B = (V, E, w^B)$ entsteht aus $G = (V, E, w)$ durch Definition einer neuen Kostenfunktion:

$$w^B(e) = \begin{cases} w(e) + B & w(e) < 0 \\ w(e) & \text{sonst} \end{cases}$$

Graph $G^B_{\geq 0}$ wiederum entsteht aus G^B durch das Aufrunden aller negativer Kantenkosten auf 0.

Für die Kanten E^{DAG} lässt sich ein linearer Algorithmus auf dem komprimierten Graphen aus Lemma 1 beschreiben, der eine Potenzialfunktion bestimmt, sodass die negativen Kanten dieser Teilmenge eliminiert werden. Dieser ist in Abschnitt 4.4 erklärt.
Somit blieben für die Ausführung von *Dijkstra+BF* nur noch potenzielle negative Kanten in E^{SCC} übrig, wodurch wir direkt auf H arbeiten könnten. Da diese Menge noch immer groß ist und die Anzahl negativer Kanten auf kürzesten Pfaden nicht beschränkt ist, wird eine weitere Technik benötigt, um die Anzahl negativer Kanten auf kürzesten Pfaden zu verkleinern.

Wie später gezeigt wird, beeinflusst der Durchmesser von SCCs die Anzahl negativer Kanten auf kürzesten Pfaden innerhalb der SCCs von H^B, wenn er in Abhängigkeit von B angegeben wird. Dabei erzielen kleine Durchmesser bessere Werte für $\eta(H^B)$. Um die Durchmesser von SCCs zu beschränken wird Low Diameter Decomposition (LDD) verwendet.
LDD wählt auf Eingabe eines Graphen G mit positiven Kantenkosten und eines Durchmessers D eine Teilmenge von Kanten $E^{rem} \subseteq E$ aus, sodass alle SCCs in $G \setminus E^{rem}$ den Durchmesser D haben. Die genaue Spezifikation ist in Abschnitt 4.1.3 beschrieben.

Bernstein et al. erreichen mit ihrer Wahl des Durchmessers, die in Abschnitt 4.3 genauer erläutert wird, dass die maximale Anzahl negativer Kanten auf kürzesten Pfaden innerhalb von H^B halbiert wird. Dabei sei B zunächst eine beliebige positive Zahl. Wenn wir nun *Dijkstra+BF* auf H^B aufrufen, ergebe sich schon ein Laufzeitvorteil zu zuvor, da $\eta(H^B) \leq \eta(G)/2$. Allerdings entstehen zwei neue Probleme:

1. In der neu entstandenen Teilmenge E^{rem} befinden sich potenziell negative Kanten, die auch eliminiert werden müssen.
2. Durch die Verschiebung der Kantenkosten in H^B werden die kürzesten Pfade verändert. Also bedeutet das Auflösen der negativen Kanten in H^B durch *Dijkstra+BF* nicht gleichzeitig das Auflösen der negativen Kanten in H.

Das erste Problem kann gelöst werden, indem dafür gesorgt wird, dass die Wahrscheinlichkeit $P(e \in E^{rem})$ gering ist. Somit könnten wir, nachdem negative Kanten in E^{SCC} und E^{DAG} durch zwei Potenzialfunktionen ϕ_1 und ϕ_2 eliminiert wurden, *Dijsktra+BF* auf dem Graphen $G_{\phi_1+\phi_2}$ ausführen, um die restlichen Kanten E^{rem} zu eliminieren.

Bernstein et al. lösen das zweite Problem, indem sie von Grund auf mit dem Graph G^B arbeiten und dafür sorgen, dass alle negativen Kanten in G^B eliminiert werden. Wenn nämlich $\forall e \in E : w^B(e) \geq 0$ gilt, gilt für den Originalgraphen $w(e) \geq -B$. Der Algorithmus, der diese Eigenschaft herstellt, wird *ScaleDown* genannt. Die Zahl $\eta(H^B)$ wird letztendlich durch einen rekursiven Aufruf von *ScaleDown* auf H^B immer weiter halbiert, bis sie schließlich so klein ist, dass *Dijkstra+BF* effizient aufgerufen werden, um die restlichen negativen Kanten zu eliminieren. Danach gilt also in H^B die Eigenschaft $w(e) \geq 0$ und in H wie gewollt $w(e) \geq -B$.
Sodass die Eigenschaft $w(e) \geq -B$ einen Vorteil bringt, muss B so gewählt werden, dass

die Kantenkosten vor der Ausführung von *ScaleDown* kleiner als $-B$ sein können. Bernstein verwendet die Vorbedingung $w(e) \geq -2B$. So können Stück für Stück die Kantenkosten vergrößert werden, bis sie sehr nahe bei 0 liegen. Wenn das geschehen ist, werden wir zeigen, dass eine kleine Addition auf alle Kantengewichte, die die letzten negativen Kanten entfernt, keine kürzesten Pfade verändert.

4.1.3 Spezifikation von Low Diameter Decomposition und ScaleDown

LDD entfernt Kanten E^{rem} aus dem Eingabegraphen G, sodass der übrige Graph $G \setminus E^{rem}$ nur in SCCs zerlegt werden kann, deren Durchmesser durch einen Wert D beschränkt ist. Hier ist nur ein schwacher Durchmesser gefordert, d.h. die kürzesten Pfade zwischen zwei Knoten des SCCs dürfen auch über Knoten, die nicht Teil des SCCs sind, gehen und somit Kanten in E^{rem} verwenden.
Wie schon zuvor angedeutet, ist unsere zusätzliche Anforderung an LDD, die Anzahl der entfernten Kanten zu beschränken, sodass *Dijkstra+BF* effizient aufgerufen werden kann.
Formal bedeutet das:

Definition 4.1.4 (Low Diameter Decomposition)
***Eingabe**: Graph $G = (V, E, w)$ mit positiven Kantengewichten und ein positiver Durchmesser D*
***Ausgabe**: Eine Kantenmenge $E^{rem} \subseteq E$ sodass gilt:*

- *Für alle SCCs in $G \setminus E^{rem}$ gilt: Für zwei beliebige Knoten eines gleichen SCCs gilt $\text{dist}_G(u, v) \leq D$ und $\text{dist}_G(v, u) \leq D$*
- *Die Wahrscheinlichkeit, dass eine Kante in E^{rem} enthalten ist, beträgt $P[e \in E^{rem}] = \mathcal{O}(\frac{w(e) \cdot log^2(n)}{D} + n^{-10})$*

Als nächstes definieren wir die Spezifikationen von *ScaleDown*.

Definition 4.1.5 (ScaleDown($G = (V, E, w), \Delta, B$))
***Eingabe**: Graph $G = (V, E, w)$, positive Parameter $Delta$ und B*
***Vorbedingungen**:*

- *Die Kantenkosten sind beschränkt durch: $w(e) \geq -2B \quad \forall e \in E$*
- *Die kürzesten Pfade in G^B enthalten maximal Δ negative Kanten*
  Formal: $\eta(G^B) \leq \Delta$

***Ausgabe**: Eine Potenzialfunktion ϕ, sodass gilt: $w_\phi(e) \geq -B \quad \forall e \in E$*

Algorithmus 4.2 SPMain($G = (V, E, w), s$)

$w(e) \leftarrow w(e) \cdot n$
$B \leftarrow$ round up n to the next power of 2
$\phi_0(v) = 0 \quad \forall v \in V$
for $i = 1$ to $t := \log_2(B)$ **do**
    $\psi_i \leftarrow$ ScaleDown($G_{\phi_{i-1}}, \Delta = n, B/2^i$)
    $\phi_i = \phi_{i-1} + \psi_i$
end for
$w' = w_{\phi_t} + 1, G' = (V, E, w')$
Dijkstra(G', s)

4.2 Hauptroutine SPmain

SPmain ist die Hauptroutine des Bernstein-Algorithmus und eine Variation des allgemeinen Algorithmus 3.2 mit den zu Beginn des Kapitels genannten Einschränkungen. Er hat also die Aufgabe, für einen Graphen mit Kantenkosten $w(e) \geq -1$ iterativ eine Potenzialfunktion zu finden, die die negativen Kantenkosten immer weiter vergrößert bis schließlich alle negativen Kanten eliminiert sind und Dijkstra angewendet werden kann. Für jede Iteration wird die Funktion ScaleDown verwendet. Da wir mit ganzzahligen Werten rechnen wollen, wird zuerst eine Skalierung der Kosten vorgenommen.
Das gleichmäßige Multiplizieren der Kostenfunktion w mit einem konstanten Faktor auf allen Kanten bringt keine Veränderung der kürzesten Pfade mit sich. Im Gegensatz dazu können die kürzesten Wege durch gleichmäßige Addition einer Konstanten auf die Einträge verändert werden. Man kann aber das Eintreten des letzteren Falles über die Wahl des Summanden einschränken. Hierzu hilft folgendes Extrembeispiel als Intuition:
Sei P ein kürzester Pfad von u nach v der Länge $d := \text{dist}_G(u, v)$, der über alle n Knoten in V

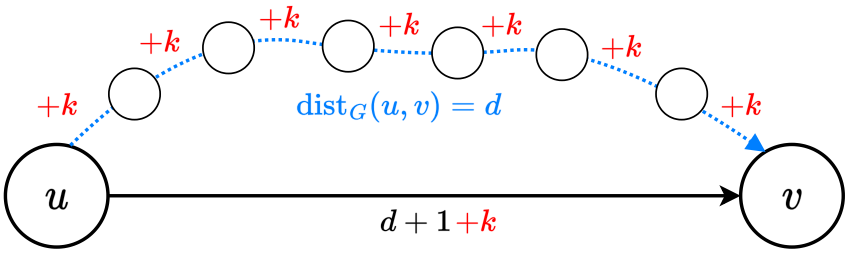


Abbildung 4.1: Auswirkung einer gleichmäßiger Addition von k auf kürzeste Pfade

geht und sei P' ein weiterer Pfad, der nur die Kante (u, v) abläuft und die Länge $w(P) = d + 1$

hat (siehe Abbildung 4.1). Damit das Aufsummieren einer Konstante k den kürzesten Pfad nicht zu P' verändert, muss folgende Ungleichung erfüllt sein:

$$w(P') = d + 1 + k > d + (n-1) \cdot k = w(P) \tag{4.1}$$
$$1 + k > n \cdot k - k \tag{4.2}$$
$$1 > n \cdot k - 2 \cdot k = k \cdot (n-2) \tag{4.3}$$
$$k < \frac{1}{n-2} \tag{4.4}$$

Daraus folgt, dass wenn die Eingabekosten mit n multipliziert werden, eine anschließende Addition von $k = 1$ auf alle Gewichte keine kürzesten Pfade verändert. Denn dann beträgt $|P'| = nd + n + 1$ und somit bleibt Ungleichung 4.4 mit $k < n/(n-2)$ erfüllt.
Nun ist das Vorgehen klar: Die Kantenkosten werden initial mit n multipliziert und anschließend durch *ScaleDown* mithilfe von Potenzialfunktionen soweit vergrößert, bis $w_\phi(e) \geq -1$ gilt. Mit einer finalen Addition um 1 sind alle negativen Kanten eliminiert und es kann Dijkstra angewendet werden, um die kürzesten Pfade zu bestimmen. SPMain ist in Algorithmus 4.2 dargestellt.
Anhand der Nachbedingung von `ScaleDown` ist zu erkennen, dass die for-Schleife eine Potenzialfunktion ϕ_t erzeugt, sodass $w_{\phi_t} \geq -1$ gilt, denn in jeder Iteration der for-Schleife erzeugt ScaleDown aus Kosten $w_{\phi_{i-1}} \geq -B/2^{i-1}$ neue Kosten $w_{\phi_{i-1}+\psi i} \geq -B/2^i$. Da ein Graph ohne negative Zyklen maximal $\Delta = n$ negative Kanten haben kann, ist die Vorbedingung von ScaleDown auch erfüllt. In SPMain werden $\mathcal{O}(\log n)$ Aufrufe des Algorithmus ScaleDown gestartet, die jeweils eine Laufzeit von $\mathcal{O}(m \log^4 m)$ haben. Da Dijkstra eine untergeordnete Laufzeit hat, kommt somit für SPMain eine Laufzeit von $\mathcal{O}(m \log^5 m)$ zustande. Zusammen mit Goldbergs Skalierungsalgorithmus ergibt sich für einen Graphen mit beliebigen ganzzahligen Kantenkosten die Laufzeit $\mathcal{O}(m \log^5 m \log(W))$, wobei W der Betrag des kleinsten negativen Kantengewichts ist.

4.3 Nebenroutine ScaleDown

Nun folgt die Umsetzung der in Definition 4.1.5 beschriebenen Spezifikation von *ScaleDown*. In Abschnitt 4.1.2 wurde hergeleitet wie der grobe Ablauf von *ScaleDown* funktioniert. Zusammengefasst hat der Eingabegraph G von *ScaleDown* die Eigenschaft $w(e) > -2B$. Es wird nun der Graph G^B erzeugt und alle negativen Kanten dieses Graphen werden durch eine Potenzialfunktion ϕ eliminiert. Danach gilt für alle Kanten $e \in E$ in G^B : $w_\phi(e) \geq 0$ und in G^B gilt $w_\phi(e) \geq -B$ wie gewollt. Die Eliminierung der Kanten erfolgt durch Aufrufen von LDD, was die Kantenmenge E in drei Teilmengen aufteilt, deren negative Kanten nacheinander eliminiert werden:

- Die Menge der entfernten Kanten E^{rem}, die direkt vom Algorithmus LDD zurückgegeben werden.

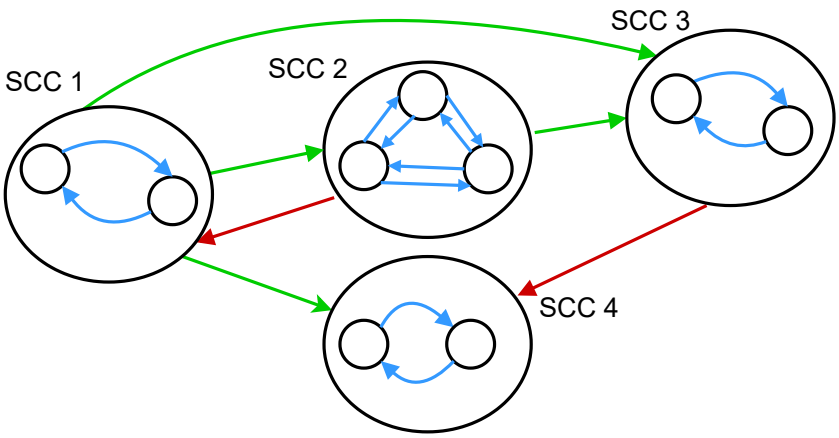


Abbildung 4.2: E^{rem}: rote Kanten, E^{SCC}: blaue Kanten, E^{DAG}: grüne Kanten

- Die Menge der Kanten E^{SCC}, die sich innerhalb von SCCs des Graphen $G \setminus E^{rem}$ befinden.
- Die Menge der Kanten E^{DAG}, die zwei verschiedene SCCs in $G \setminus E^{rem}$ verbinden.

Abbildung 4.2 veranschaulicht diese Kantenklassen. Der Algorithmus für ScaleDown ist in 4.3 dargestellt. Da für LDD ein Eingabegraph mit ausschließlich positiven Kanten benötigt wird, wird hier $G^B_{\geq 0}$ mitgegeben. Anhand folgender Ungleichungen sieht man, dass trotzdem auch für Graph $G \setminus E^{rem}$ die Durchmesser der SCCs durch $\Delta B/2$ beschränkt sind:

$$\text{dist}_G(u, v) \leq \text{dist}_{G^B_{\geq 0}}(u, v) \leq \Delta B/2$$

Mithilfe dessen kann nun Theorem 1 bewiesen werden:

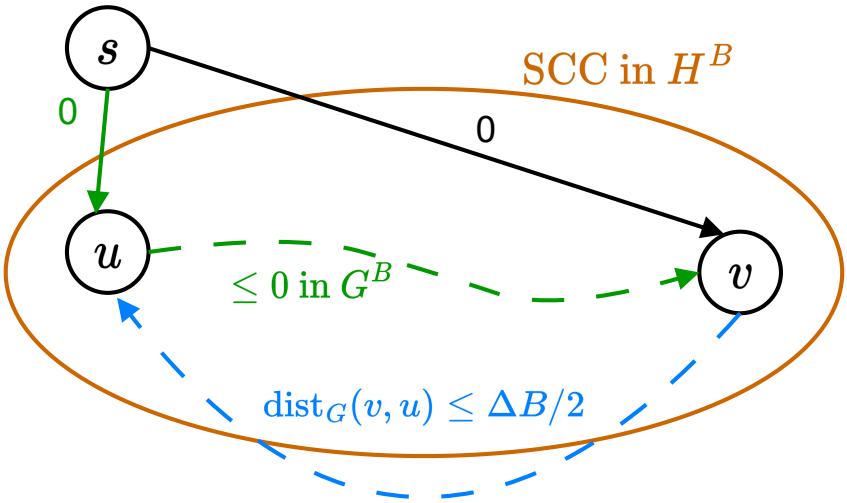


Abbildung 4.3: Darstellung des Pfades $P_{H^B}(v)$ (grün) und weiterer Einschränkungen

Theorem 1
Nach der Ausführung von LDD gilt für Graph H^B: $\eta(H^B) \leq \Delta/2$

Proof: Es sei v ein beliebiger Knoten in H^B. Wir betrachten nun den Pfad $P_{H^B}(v)$ (vgl. Definition 4.1.2), der aus einer Kante (s, u) und einem Pfad P von u nach v besteht (siehe Abbildung 4.3).
Nach vorheriger Beobachtung gilt: $\text{dist}_G(v, u) \leq \Delta B/2$
Da $P_{H^B}(v)$ ein kürzester Pfad von s nach v ist und jeder Knoten von s aus mit einer Kante von Gewicht 0 erreichbar ist, gilt $\text{dist}_{H_s^B}(s, v) \leq 0$ und folglich auch $\text{dist}_{G^B}(u, v) \leq 0$.
Weil für G^B nur B auf die negativen Kanten in G addiert wird, kann die Länge von P in G nicht größer sein als $-B \cdot \eta_{H^B}$, also $\text{dist}_G(u, v) \leq -B \cdot \eta_{H^B}$.
Dadurch, dass wir voraussetzen, dass keine negativen Zyklen in G existieren, muss gelten:

$$\begin{aligned} \text{dist}_G(u, v) + \text{dist}_G(v, u) &\geq 0 \\ -B \cdot \eta(H^B) + \Delta B/2 &\geq 0 \\ \eta(H^B) &\leq \Delta/2 \end{aligned}$$

□

Statt nun aber *Dijkstra+BF* auf dem Graphen H^B aufzurufen, starten wir einen rekursiven Aufruf von *ScaleDown*, um $\eta(H^B)$ noch weiter zu verkleinern. Erst wenn in einem *ScaleDown*-Aufruf $\eta(G^B) \leq \Delta \leq 2$ gilt, wird *Dijkstra+BF* ausgeführt. In diesem Fall kann *Dijktra+BF* offensichtlich in $\mathcal{O}(n \log n)$ gelöst werden.
Es bleibt also nur noch die Frage nach der Laufzeit des anderen *Dijktra+BF*-Aufrufs, der übrig gebliebene negative Kanten in E^{rem} eliminiert. Folgendes Lemma kann mithilfe der von LDD gegebenen Wahrscheinlichkeit $P[e \in E^{rem}]$ gezeigt werden:

Lemma 4
Nach der Auführung von LDD gilt für alle Knoten $u \in V$: $E[|P_{G^B}(v) \cap E^{rem}|] = \mathcal{O}(\log^2 m)$

Mithilfe dessen kann die finale Laufzeit des zweiten *Dijkstra+BF*-Aufrufs geklärt werden. Weil vor der Eliminierung negativer Kanten in E^{rem} schon alle anderen negativen Kanten eliminiert wurden, gilt für den Erwartungswert von $E[\eta_{G^B_{\phi_2}}]$:

$$E[\eta_{G^B_{\phi_2}}] \leq E[|P_{G^B}(v) \cap E^{rem}|] = \mathcal{O}(\log^2 m)$$

Eingesetzt in die Laufzeit von *Dijktra+BF* ergibt sich:

$$\mathcal{O}((m + E[\eta_{G^B_{\phi_2}}]) \cdot \log m) = \mathcal{O}(m \cdot \log^3 m)$$

Da die nicht-rekursive Laufzeit eines *ScaleDown*-Aufrufs $\mathcal{O}(m \cdot \log^3 m)$ beträgt und $\log(\Delta)$ rekursive Aufrufe durchgeführt werden, beträgt die Gesamtkomplexität von *ScaleDown*:

$$\mathcal{O}(m \log^3 m \log \Delta)$$

Algorithmus 4.3 ScaleDown($G = (V, E, w), \Delta, B$)

if $\Delta \leq 2$ **then return** DIJKSTRA+BF(G^B)
end if
$E^{rem} \leftarrow$ LDD($G^B_{\geq 0}, \Delta B/2$)
// *Bestimmung der SCCs* $V_1, V_2, ...$ *in* $G^B \setminus E^{rem}$ *mit Mengen der jeweils ausgehenden Kanten*
// *pro SCC* $E_1, E_2, ...$
$V_1, V_2, ...$ and $E_1, E_2, ... \leftarrow$ TARJANSCC($G^B \setminus E^{rem}$)
$H \leftarrow \bigcup_i G[V_i]$ // graph of all SCCs combined
$\phi_1 \leftarrow$ SCALEDOWN($H, \Delta/2, B$)
$\phi_2 \leftarrow$ FIXDAGEDGES($G^B_{\phi_1} \setminus E^{rem}, (V_1, V_2, ...), (E_1, E_2, ...)$)
$\phi_2 \leftarrow \phi_2 + \phi_1$ // value-wise addition
$\phi_3 \leftarrow$DIJKSTRA+BF($G^B_{\phi_2}$)
return $\phi_2 + \phi_3$

4.4 Auflösen negativer DAG-Kanten

Lemma 1 sagt aus, dass der Graph, der durch Komprimierung der SCCs eines Eingabegraphen zu einzelnen Knoten entsteht, ein DAG ist. Im Folgenden wird ein Algorithmus beschrieben, der mögliche negative DAG-Kanten auflöst. Dazu wird eine topologische Sortierung des komprimierten Graphen benötigt (siehe Abschnitt 2.4.1).
Die Eingabe ist also ein Graph $G = (V, E, w)$ und eine Menge $V_1, V_2, ..., V_k$, wobei für alle $1 \leq i \leq k$ $G[V_i]$ SCCs in G sind. Es seien außerdem für jeden SCC V_i die ausgehenden Kanten E_i bekannt mit der Eigenschaft:

$$(u, v) \in E_i \iff u \in V_i \wedge v \notin V_i$$

Die Idee ist Folgende: Betrachte man zwei Knoten u und v, die durch eine negative Kante (u, v) verbunden sind, und deren Potenzial $\phi(v)$ gegeben ist. Um eine negative Kante (u, v) zu beheben, muss das Potenzial $\phi(u) \geq \phi(v) - w(u, v)$ gewählt werden. Dies leitet sich aus der Ungleichung $w_\phi = w(u, v) + \phi(u) - \phi(v) \geq 0$ ab. Da jedoch damit Vorgängerknoten von u mit unverändertem Potenzial eventuell negativ gemacht werden können, müssen die Knoten nach der umgekehrten topologischen Sortierung abgearbeitet werden.
Außerdem wird im Falle von mehreren ausgehenden Kanten eines Knoten das Gewicht der minimalen negativen Kante betrachtet, weil ein geeignetes Potenzial für diese minimale Kante auch höher gewichtete negative Kanten eliminiert.
Das Resultat ist in Algorithmus 4.4 dargestellt.

Algorithmus 4.4 FixDagEdges($G, \mathcal{P} = (V_1, ..., V_k), \mathcal{D} = (E_1, ..., E_k)$)

$\phi \leftarrow 0$ undefinierte Potenzialfunktion
$Q = (q_1, ..., q_k) \leftarrow$ topologicalSort($G, \mathcal{P}, \mathcal{D}$)
price $\leftarrow 0$
for $i \leftarrow k;\ i > 0;\ i \leftarrow i - 1$ **do**
    min $\leftarrow 0$
    for all $e \in E_{q_i}$ **do**
        if $w(e) \leq$ min **then**
            min $\leftarrow w(e)$
        end if
    end for
    price $\leftarrow min$
    for all $u \in V_{q_i}$ **do**
        $\phi(u) \leftarrow$ price
    end for
end for
return ϕ

4.5 Low Diameter Decomposition

In Abschnitt 4.1.3 wurden schon die Anforderungen von LDD erläutert. Nun folgt die Erklärung eines Algorithmus, der diese Anforderungen erfüllt. Zuerst benötigen wir jedoch das Verständnis von sogenannten Bällen um einen Knoten herum:

Definition 4.5.1 (Bälle und Randkanten)
Sei ein Graph $G = (V, E, w)$ und eine Länge D gegeben.
Für einen beliebigen Knoten $u \in V$ ist der eingehende Ball $\mathrm{Ball}^{\mathrm{in}}_G(u, D)$ eine Teilmenge von V mit folgender Eigenschaft:

$$\mathrm{Ball}^{\mathrm{in}}_G(u, D) = \{v \in V : \mathrm{dist}_G(v, u) \leq D\}$$

In anderen Worten: Der Ball enthält alle Knoten, deren kürzeste Pfade zu u durch die Länge D beschränkt sind.
Außerdem sei $\mathrm{boundary}(\mathrm{Ball}^{\mathrm{in}}_G(u, D)) \subseteq E$ die Menge der Randkanten des Balls:

$$\mathrm{boundary}(\mathrm{Ball}^{\mathrm{in}}_G(u, D)) = \{(x, y) | x \notin \mathrm{Ball}^{\mathrm{in}}_G(u, D) \wedge y \in \mathrm{Ball}^{\mathrm{in}}_G(u, D)\}$$

Abbildung 4.4 veranschaulicht diese beiden Definitionen.
Nach gleichem Prinzip wird der ausgehende Ball $\mathrm{Ball}^{\mathrm{out}}_G(u, D)$ definiert, bei dem kürzeste Pfade betrachtet werden, die von u zu einem anderen Knoten hinführen.

$$\mathrm{Ball}^{\mathrm{out}}_G(u, D) = \{v \in V : \mathrm{dist}_G(u, v) \leq D\}$$

Auch hier werden entsprechende Randkanten $\mathrm{boundary}(\mathrm{Ball}^{\mathrm{out}}_G(u, D)) \subseteq E$ definiert:

$$\mathrm{boundary}(\mathrm{Ball}^{\mathrm{out}}_G(u, D)) = \{(x, y) | x \in \mathrm{Ball}^{\mathrm{out}}_G(u, D) \wedge y \notin \mathrm{Ball}^{\mathrm{out}}_G(u, D)\}$$

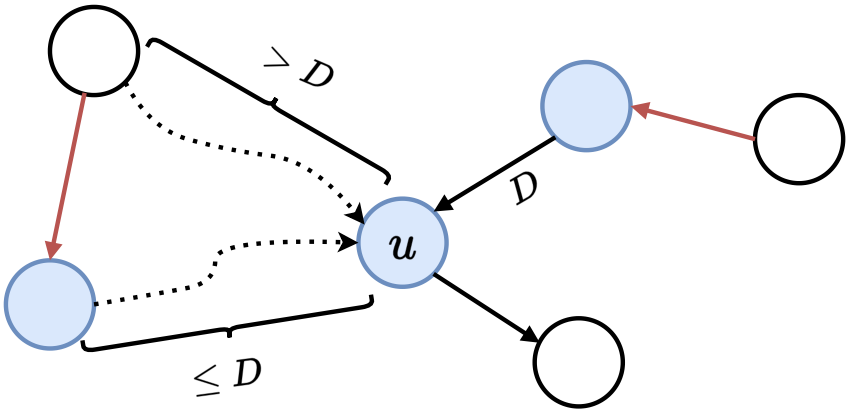


Abbildung 4.4: $\text{Ball}^{\text{in}}_G(u, D)$: Menge aller blauen Knoten, $\text{boundary}(\text{Ball}^{\text{in}}_G(u, D))$: Menge aller rote Kanten

4.5.1 Prinzip

Unter G_0 wird im weiteren Verlauf der Eingabegraph eines Aufrufs von LDD bezeichnet. G ist anfangs G_0 und wird im Laufe des Algorithmus durch Herausschneiden von Teilgraphen verkleinert.
Der Algorithmus besteht aus zwei Phasen. In der ersten Phase werden alle Knoten in eine der Kategorien **in-light**, **out-light** oder **heavy** eingeteilt. Dabei gelten die Eigenschaften in Lemma 5, die später noch gezeigt werden:

Lemma 5 (Ball Kardinalitäten)

- *Wenn u* ***in-light*** *ist, gilt mit hoher Wahrscheinlichkeit:* $|\text{Ball}^{\text{in}}_{G_0}(u, D/4)| \leq 0.7 \cdot |V(G_0)|$
- *Wenn u* ***out-light*** *ist, gilt mit hoher Wahrscheinlichkeit:* $|\text{Ball}^{\text{out}}_{G_0}(u, D/4)| \leq 0.7 \cdot |V(G_0)|$
- *Wenn u* ***heavy*** *ist, gilt mit hoher Wahrscheinlichkeit:* $|\text{Ball}^{\text{in}}_{G_0}(u, D/4)| > 0.5 \cdot |V(G_0)|$ *und* $|\text{Ball}^{\text{out}}_{G_0}(u, D/4)| > 0.5 \cdot |V(G_0)|$

Phase 2 hat die Aufgabe, aus G so lange Knoten zu entfernen, bis nur noch als **heavy** kategorisierte Knoten übrig bleiben. Genauer gesagt werden für alle **light** markierten Knoten Bälle erzeugt und deren Randkanten zur Menge E^{rem} hinzugefügt. Dadurch wird erreicht, dass jeder dieser Bälle mit keinem Knoten von außerhalb einen SCC bilden kann. Denn um einen Zyklus mit einem Knoten innerhalb des Balls und außerhalb zu bilden, wird zwangsläufig eine der Randkanten benötigt.
Nun stellt sich die Frage, wie die Durchmessereigenschaft für die Knoten innerhalb eines Balls erfüllt wird. Um dieses Problem zu lösen, führen Bernstein et al. einen rekursiven Aufruf auf dem induzierten Teilgraphen des Balls auf. Dieser Aufruf stellt dann sicher, dass die Durchmesser der SCCs innerhalb dieses Teilgraphen durch D beschränkt sind. Die Rekursion wird also solange fortgesetzt, bis entweder keine **light**-Knoten mehr in den Teilgraphen zu finden sind oder die Graphgröße $|V(G)| = 1$ erreicht ist. In letzterem Fall ist der verbleibende Knoten

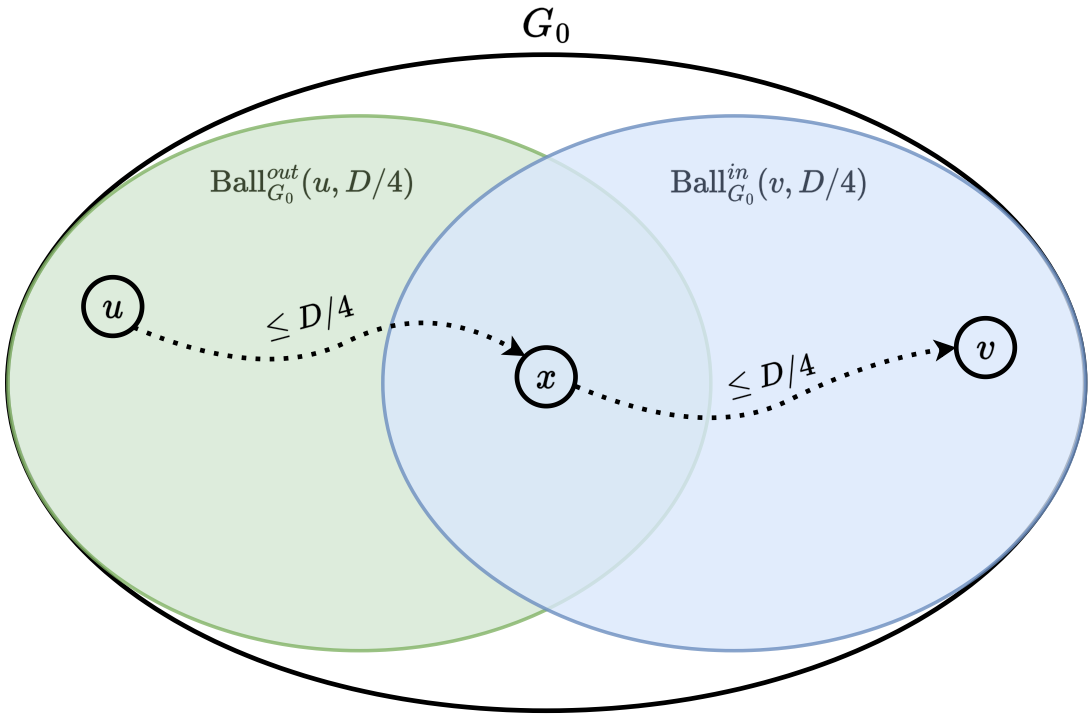


Abbildung 4.5: Durchmesserbeschränkung von **heavy**-Knoten

automatisch **heavy**, da dessen Ballkardinalität immer $1 > 0.5|V(G_0)|$ entspricht. Es wird dabei eine logarithmische Rekursionstiefe erzielt, weil nach Lemma 5 bekannt ist, dass ein **light**-Ball maximal 70% der ursprünglichen Knotenanzahl eines Aufrufs von LDD entspricht.

Es gilt nun noch zu zeigen, dass für alle **heavy**-Knoten, die nach Phase 2 in G übrig bleiben, die Durchmessereigenschaft auch erfüllt ist.
Für als **heavy** kategorisierte Knoten gilt bekanntlich $|\text{Ball}^{in}_{G_0}(u, D/4)| > 0.5 \cdot |V(G_0)|$ und $|\text{Ball}^{out}_{G_0}(u, D/4)| > 0.5 \cdot |V(G_0)|$. Daraus folgt jedoch, dass die Schnittmenge eines ausgehenden Balls von u mit dem eingehenden Ball von v nicht-leer sein muss. Also muss ein beliebiger Knoten $x \in V(G_0)$ in der Schnittmenge existieren, für den gilt:

$$\text{dist}_{G_0}(u, x) \leq D/4 \wedge \text{dist}_{G_0}(x, v) \leq D/4$$

Damit ist in der Summe der kürzeste Pfad von u nach v (und nach gleicher Argumentation von v nach u) in G_0 durch $D/2$ beschränkt (siehe Abbildung 4.5).
Da jetzt die erste Eigenschaft von Low Diameter Decomposition (Definition 4.1.4) geklärt ist, bleibt noch die Frage nach der zweiten Eigenschaft, die Wahrscheinlichkeit einzuschränken, dass eine Kante Teil von E^{rem} ist. Dafür betrachten wir zuerst die Details zu Phase 1 und zeigen insbesondere Lemma 5.

4.5.2 Phase 1

Da man vermeiden möchte, für jeden Knoten den zugehörigen eingehenden und ausgehenden Ball zu berechnen, um dessen Kardinalität zu bestimmen und die Eigenschaften in Lemma 5 zu erfüllen, behilft man sich mit dem Trick, dies nur für eine kleine Menge S von $k = c \cdot log(n)$ zufällig ausgewählten Knoten zu tun. Mithilfe der Bälle $\text{Ball}^{\text{in}}_G(s_i, D/4)$ und $\text{Ball}^{\text{out}}_G(s_i, D/4)$ für alle $s_i \in S$ lassen sich für jeden Knoten $u \in V$ die Schnittmengen $\text{Ball}^{\text{in}}_G(u, D/4) \bigcap S$ und $\text{Ball}^{\text{out}}_G(u, D/4) \bigcap S$ berechnen. Um erstere zu berechnen, muss für jedes s_i geschaut werden, ob u in dessen ausgehenden Ball enthalten ist. Für letztere müssen dementsprechend die eingehenden Bälle der ausgewählten Knoten s_i durchsucht werden.
Es bleibt nun zu zeigen, wie die nun berechneten Schnittmengen dazu beitragen, Aussagen über die tatsächliche Kardinalität der Bälle aller Knoten zu treffen. Dazu beweisen wir Theorem 2. Theorem 2 impliziert Lemma 5. Der Beweis erfordert Hoeffdings Ungleichung:

Lemma 6 (Hoeffdings Ungleichung (Abweichung))
Seien $X_1, ..., X_k$ unabhängige Zufallsvariablen mit Werten $\in \{0, 1\}$ und sei $X_{avg} = 1/k \cdot \sum_{i=1}^{k} X_i$. Dann gilt für ein $1 > t > 0$: $\mathbb{P}(X_{avg} < \mathbb{E}[X_{avg} - t]) \leq e^{-2kt^2}$

Theorem 2 (Ball Kardinalitäten)
Sei $u \in V$ ein beliebiger Knoten und $p = 1 - 1/n^{20}$ die Wahrscheinlichkeit für folgende Eigenschaften:

- *Wenn $|\text{Ball}^{\text{in}}_G(u, D/4) \bigcap S| \leq 0.6 \cdot k$ gilt, ist u **in-light** und es gilt mit Wahrscheinlichkeit p: $|\text{Ball}^{\text{in}}_{G_0}(u, D/4)| \leq 0.7 \cdot |V(G_0)|$*
- *Wenn $|\text{Ball}^{\text{out}}_G(u, D/4) \bigcap S| \leq 0.6 \cdot k$ gilt, ist u **out-light** und es gilt mit Wahrscheinlichkeit p: $|\text{Ball}^{\text{out}}_{G_0}(u, D/4)| \leq 0.7 \cdot |V(G_0)|$*
- *Wenn $|\text{Ball}^{\text{in}}_G(u, D/4) \bigcap S| > 0.6 \cdot k$ und $|\text{Ball}^{\text{out}}_G(u, D/4) \bigcap S| > 0.6 \cdot k$ gilt, ist u **heavy** und es gilt mit Wahrscheinlichkeit p: $|\text{Ball}^{\text{in}}_{G_0}(u, D/4)| > 0.5 \cdot |V(G_0)|$ und $|\text{Ball}^{\text{out}}_{G_0}(u, D/4)| > 0.5 \cdot |V(G_0)|$*

Proof: Wir zeigen die erste Aussage. Die Beweise für die beiden anderen erfolgen nach gleichem Schema.
Kontraposition: Wenn für einen Knoten u $|\text{Ball}^{\text{in}}_{G_0}(u, D/4)| > 0.7 \cdot |V(G_0)|$ gilt, dann gilt mit Wahrscheinlichkeit $p = 1/n^{20}$: $|\text{Ball}^{\text{in}}_G(u, D/4) \bigcap S| \leq 0.6 \cdot k$.
Für den Erwartungswert der Schnittmenge gilt:

$$\mathbb{E}[|\text{Ball}^{\text{in}}_G(u, D/4) \bigcap S|] = \sum_{i=1}^{k} \mathbb{P}(s_i \in \text{Ball}^{in}_G(u, D/4))$$

Da $\forall s_i \in S : \mathbb{P}(s_i \in \text{Ball}^{\text{in}}_G(u, D/4)) > 0.7$ (weil alle s_i zufällig gewählt sind)

$$\mathbb{E}[|\text{Ball}^{\text{in}}_G(u, D/4) \bigcap S|] > 0.7 \cdot k$$

Indem man für die Hoeffdings Ungleichung $X_i = 1 \iff s_i \in \mathrm{Ball}^{\mathrm{in}}_G(u, D/4)$ und $t = 0.1$ wählt, ist erkenntlich, dass ein Wert von $|\mathrm{Ball}^{\mathrm{in}}_G(u, D/4) \bigcap S|] > 0.7 \cdot k$ sehr unwahrscheinlich ist. Damit ist die Kontraposition gezeigt. □

4.5.3 Algorithmus & Details

Es wird nun der Algorithmus 4.5 gezeigt, so wie er in [BNWN22] vorgestellt wird. Es ist in den Zeilen $4 - 15$ die Phase 1 und in Zeilen $16 - 25$ Phase 2 zu erkennen. Es fallen nun 3 Details auf, die bisher noch nicht angesprochen wurden.

1. Die Durchmesser der Bälle, die in Phase 2 berechnet werden, werden aus einer geometrischen Verteilung entnommen (Zeile 17).
2. In Zeilen $20 - 22$ befindet sich eine Abfrage, bei dessen Eintreten die Kantenmenge des Gesamtgraphen G_0 zurückgegeben wird.
3. In Zeilen $26 - 29$ findet eine weitere Abfrage statt, bei der auch die gesamte Kantenmenge zurückgegeben wird.

Zuerst zum dritten Punkt: Wir haben zuvor gezeigt, dass die Durchmessereigenschaft für **heavy**-Knoten, die nach Phase 2 in G übrig sind, erfüllt ist. Dieser Beweis baute jedoch auf der Annahme auf, dass die dritte Implikation aus Theorem 2 sicher eintritt. Da aber tatsächlich die Implikation mit einer Wahrscheinlichkeit von $p = 1/n^{20}$ für einen Knoten nicht zutrifft, muss der umgekehrte Fall auch betrachtet werden. In diesem Fall ist die Ungleichung $\mathrm{dist}_{G_0}(x, y) \leq D/2$ für zwei beliebige **heavy**-Knoten x und y nicht immer erfüllt also muss es nach Ende von Phase 2 einen Knoten in G geben, der nicht Teil des Balls $\mathrm{Ball}^{in}_{G_0}(u, D/2)$ oder $\mathrm{Ball}^{out}_{G_0}(u, D/2)$ ist. Anders ausgedrückt gilt entweder $V(G) \not\subseteq \mathrm{Ball}^{in}_{G_0}(u, D/2)$ oder $V(G) \not\subseteq \mathrm{Ball}^{out}_{G_0}(u, D/2)$. Indem die Kantenmenge $E^{rem} = E(G_0)$ zurückgegeben wird, ist jeder Knoten in $G_0 \setminus E^{rem}$ ein einzelner SCC und deren Durchmesser sind definitiv $\leq D$.

Der zweite Punkt entsteht aus einem ähnlichen Grund heraus wie der Dritte, denn auch hier möchte man sicherstellen, dass die Implikation für **light**-Knoten aus Lemma 5 zutreffen. Die Überprüfung nach $R > D/4$ kommt also daher, dass für größere Durchmesser die Ungleichung $|\mathrm{Ball}^{*}_G(u, R) \bigcap S| \leq 0.6 \cdot k$ nicht sichergestellt werden kann. Die zweite Überprüfung dient schließlich dazu, den unwahrscheinlichen Fall, dass die Implikationen nicht zutreffen, abzufangen. Wie in Punkt 3 wird daher auch hier die Kantenmenge des Eingabegraphen zurückgegeben.

Die Tatsache, dass für den Durchmesser der Bälle in Phase 2 die geometrische Verteilung mit der angegeben Trefferwahrscheinlichkeit von $p = \min(1, 80 \cdot \log_2(n)/D)$ gewählt wurde, dient dem Zweck, die Zielwahrscheinlichkeit von $P[e \in E^{rem}] = \mathcal{O}(\frac{w(e) \cdot log^2(n)}{D} + n^{-10})$ zu erreichen, wobei hiermit der erste Summand entsteht. Der zweite Summand entsteht durch die zuvor erläuterten Überprüfungen, die die Misserfolgswahrscheinlichkeit von Theorem 2

Algorithmus 4.5 LDD($G = (V, E, w), D$)

1: $G_0 \leftarrow G$
2: $E^{rem} \leftarrow \emptyset$
3: $k \leftarrow c \log_2(n)$
4: $S \leftarrow \{s_1, ..., s_k\}$ // s_i sind unabhängig und zufällig ausgewählte Knoten aus V
5: Berechne für alle $s_i \in S$ die Bälle $\text{Ball}^{in}_{G_0}(s_i, D/4)$ und $\text{Ball}^{out}_{G_0}(s_i, D/4)$
6: Berechne mithilfe der Bälle von zuvor für alle $v \in S$ die Schnittmengen $\text{Ball}^{in}_{G_0}(v, D/4) \cap S$ und $\text{Ball}^{out}_{G_0}(s_i, D/4) \cap S$ (siehe Abschnitt 4.5.2)
7: **for all** $u \in V$ **do**
8: **if** $|\text{Ball}^{in}_{G_0}(v, D/4) \cap S| \leq 0.6 \cdot k$ **then**
9: Markiere u als **in-light**
10: **else if** $|\text{Ball}^{out}_{G_0}(v, D/4) \cap S| \leq 0.6 \cdot k$ **then**
11: Markiere u als **out-light**
12: **else**
13: Markiere u als **heavy**
14: **end if**
15: **end for**
16: **while** $u \in G : u$ ist $*$**-light** wobei $* \in \{in, out\}$ **do**
17: Bestimme eine Probe $R \sim \text{Geo}(p)$ mit $p = \min(1, 80 \cdot \log_2(n)/D)$
18: $\text{Ball} \leftarrow \text{Ball}^{*}_{G}(u, R)$
19: $E^{rem} \leftarrow E^{rem} \cup \text{boundary}(\text{Ball})$
20: **if** $R > D/4$ oder $|\text{Ball}| > 0.7 \cdot |V(G_0)|$ **then**
21: **return** $E(G_0)$
22: **end if**
23: $E^{rem} \leftarrow E^{rem} \cup$ LDD($G[\text{Ball}], D$)
24: $G \leftarrow G \setminus \text{Ball}$
25: **end while**
26: $u \leftarrow$ zufälliger Knoten in G
27: **if** $V(G) \not\subseteq \text{Ball}^{in}_{G_0}(u, D/2)$ oder $V(G) \not\subseteq \text{Ball}^{out}_{G_0}(u, D/2)$ **then**
28: **return** $E(G_0)$
29: **end if**
30: **return** E^{rem}

abfangen. Für die ausführliche Mathematik dahinter verweise ich auf das Paper von Bernstein [BNWN22].

4.5.4 Komplexität

Um die Bälle der gewählten Knoten S in Phase 1 zu berechnen, werden $k = \mathcal{O}(\log(n))$ Aufrufe des Dijkstra Algorithmus benötigt, der jeweils eine Laufzeit von $\mathcal{O}(|V(G)| \log(n) + |E(G)|)$ hat. Die Schnittmengen lassen sich wie zuvor beschrieben in linearer Zeit berechnen. Also hat Phase 1 eine Komplexität von $\mathcal{O}(|V(G)| \log^2(n) + |E(G)| \log(n))$.
In Abschnitt 4.5.1 haben wir erläutert, dass die Rekursionstiefe des Algorithmus $\mathcal{O}(log(n))$ beträgt. Dadurch, dass jeder Ball, der in Phase 2 generiert wird und rekursiv bearbeitet wird, direkt aus dem Graphen entfernt wird, kann jeder Knoten auch nur $\mathcal{O}(log(n))$ mal Teil eines Aufrufs von LDD sein. Außerdem hat das Entfernen der Knoten zur Folge, dass bei den Ballberechnungen in Phase 2 nie ein Knoten mehrmals betrachtet werden muss. Weil die Stichprobenahme aus der geometrischen Verteilung in $\mathcal{O}(log(n))$ und bei $\mathcal{O}(|V(G)|)$ vielen Aufrufen der while-Schleife in $\mathcal{O}(|V(G)| \log(n))$ stattfinden kann, wird die nicht-rekursive Komplexität von Phase 2 von der der Ballberechnungen übertroffen, welche $\mathcal{O}(|V(G)| \log(n) + |E(G)|)$ beträgt. Auch die Überprüfung am Ende eines Aufrufs spielt eine untergeordnete Rolle, da nur zwei Dijkstra-Ausführungen benötigt werden.
Die nicht-rekursive Laufzeit eines Aufrufs, die von Phase 1 überschattet wird, zusammen mit der schon genannten Tatsache, dass jeder Knoten Teil von $\mathcal{O}(log(n))$ rekursiven Aufrufen ist, ergibt eine Gesamtkomplexität von $\mathcal{O}(n \log^3(n) + m \log^2(n))$.

4.6 Negative Zyklen

Bisher wurde angenommen, dass im Eingabegraphen von Bernstein keine negativen Zyklen vorliegen. Wir wissen, dass im Falle von negativen Zyklen der Teilalgorithmus *Dijsktra+BF* nie terminiert (siehe Abschnitt 4.1.1). Daher kann der gesamte Algorithmus auch nicht terminieren. Mit dem Bernstein-Algorithmus ist im Folgenden der in diesem Kapitel erklärte Algorithmus gemeint, der mithilfe Goldbergs Skalierungsframework für beliebige ganzzahlige Kantenkosten funktioniert. Dieser hat somit die Komplexität $\mathcal{O}(m \log^5(n) \log(W))$. Bernstein et al. beschreiben nun einen Monte-Carlo Algorithmus, der auf dem Bernstein-Algorithmus basiert und ihn erweitert:

- Wenn der Eingabegraph negative Zyklen enthält, wird immer eine Fehlermeldung zurückgegeben.
- Wenn der Eingabegraph keine negativen Zyklen enthält, werden mit hoher Wahrscheinlichkeit die kürzesten Pfade zurückgegeben aber sonst auch eine Fehlermeldung.

Die Erkennung negativer Zyklen erfolgt dadurch, dass $C \cdot \log(n)$ viele Aufrufe des Algorithmus gestartet werden, die alle für die Zeit $2T$ gestoppt werden. Hierbei ist C eine hohe Konstante und T die erwartete Laufzeit des Bernstein-Algorithmus. Wenn nach dieser Zeit in keinem der Aufrufe ein Ergebnis vorliegt, wird eine Fehlermeldung zurückgegeben. Im umgekehrten Fall zählt das erste Ergebnis eines Aufrufs als korrekt und wird zurückgegeben. Mithilfe der Markow-Ungleichungen kann die Wahrscheinlichkeit, dass keiner der Aufrufe in der geforderten Zeit terminiert, obwohl keine negativen Zyklen vorliegen, minimiert werden.
Aufgrund der $C \cdot \log(n)$-fachen Ausführung des Bernstein-Algorithmus, der keine negativen Zyklen erkennt, beträgt die Komplexität des erweiterten Monte-Carlo-Algorithmus $\mathcal{O}(m \log^6(n) \log(W))$. Mithilfe des Monte-Carlo Algorithmus konstruieren Bernstein et al. einen Las-Vegas Algorithmus, der zudem auf Kosten eines zusätzlichen Faktors $\mathcal{O}(\log^2(n))$ in der Gesamtkomplexität einen gefundenen negativen Zyklus zurückgibt.

Allerdings ist diese Vorgehensweise in der Praxis fragwürdig. Zum Einen ist die tatsächliche Laufzeit eines Aufrufs des Bernstein-Algorithmus abhängig von vielen Parametern wie der Größe des Graphen bis hin zur Schnelligkeit der Hardware. Außerdem können die Laufzeiten selbst bei gleichen Voraussetzungen stark schwanken, wenn beispielsweise der Prozess des Algorithmus nicht die volle Ressourcenzuteilung erhält.
Zum Anderen erfordern $C \cdot log(n)$ viele Aufrufe des Bernstein-Algorithmus deutlich mehr Zeit. Man könnte dies zwar durch Parallelisierung verbessern, allerdings würden bei großen Graphen sehr viele Prozessoren benötigt.
Aufgrund dieser Problematik beschränken wir uns bei der Implementierung und den Laufzeittests auf die Voraussetzung, dass keine negativen Zyklen im Eingabegraphen vorliegen.

5 Implementierung

In diesem Kapitel wird die Implementierung des Bernstein Algorithmus, der im letzten Kapitel erläutert wurde, diskutiert. Es wird hierbei mit einem objektorientierten Ansatz gearbeitet. Die finale Implementierung erreicht dabei die angegebene asymptotische Laufzeit von $\mathcal{O}(m \log^5(n) \log W)$ des Bernstein-Algorithmus.

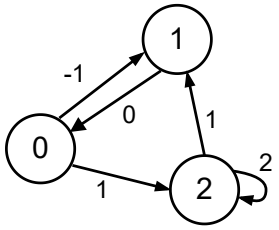
5.1 Graph Datenstruktur

Die Wahl der Graph-Datenstruktur hat einen massiven Einfluss auf die Laufzeit eines beliebigen Graphalgorithmus, da sie darüber bestimmt, wie groß der Aufwand für eine bestimmte Graphoperation ist. Daher ist es sinnvoll, zuerst die Anforderungen an die Datenstruktur zu formulieren.
In unserem Fall hat die Abfrage von ausgehenden Kanten eines Knotens hohe Priorität, da dies ein fester Bestandteil von Suchalgorithmen und auch Kürzeste-Pfade Algorithmen ist. Außerdem wird bei der Berechnung von eingehenden Bällen in LDD eine schnelle Abfrage eingehender Kanten in einen Knoten benötigt.

Um diese beiden zentralen Operationen in konstanter Zeit zu lösen, stellen wir folgende Datenstruktur für einen Graph $G = (V, E, w)$ vor: Es sei `edges` ein Array von Kanten, wobei jede Kante $e \in E$ folgende Informationen speichert:

- eindeutige Identifikationsnummer (ID)
- Ursprungsknoten (source)
- Zielknoten (target)
- Kantengewicht $w(e)$

Die Identifikationsnummer ID jeder Kante entspricht dabei der Indexposition in `edges`. Außerdem werden die Kanten so sortiert, dass alle ausgehenden Kanten eines bestimmten Knoten nebeneinander in der Liste vorliegen. Dadurch entsteht eine Nummerierung der Knoten, die in Zukunft für die Identifikation eines Knotens verwendet wird.
Die Kantensortierung bietet den Vorteil, dass über das Speichern einer einzigen Kanten-ID, welche Offset genannt wird, über die ausgehenden Kanten eines bestimmten Knotens iteriert werden kann. Genauer formuliert speichert die Liste `offsets` für jeden Knoten $u \in V$ die ID der ersten Kante e, für die `e.source = u` gilt. Die letzte Kante, die den Ursprungsknoten u hat,



edges			
ID	source	target	w
0	0	1	-1
1	0	2	1
2	1	0	0
3	2	1	1
4	2	2	2

offsets	
ID	offset
0	0
1	2
2	3
3	5

pred	
ID	pred
0	2
1	0, 3
2	1, 4

Abbildung 5.1: Graph Repräsentation anhand eines Beispielgraphen

hat dementsprechend die ID: `offset[u+1]-1`. Um diese Formel auch für den letzten Knoten sicherzustellen, fügen wir einen $n + 1$-ten Eintrag mit Offset m ein.
Für die Abfrage eingehender Kanten benötigt es eine doppelt-verschachteltes Array `pred` (Predecessors), die für jeden Knoten die IDs der eingehenden Kanten angibt.
Abbildung 5.1 gibt eine Repräsentation der genannten Listen für einen Minigraphen an.

5.2 Hauptalgorithmus Bernstein

In den Pseudocodes der Teilroutinen des Bernstein Algorithmus sind oft Neuzuweisungen und Additionen von Potenzialfunktionen vorzufinden. Da jedoch jede dieser Operationen einen Aufwand von $\mathcal{O}(m)$ mit sich bringt, stellt sich die Frage nach einer effizienteren Methode.
Man kann leicht feststellen, dass die Potenziale nur in FixDAGEdges und in Dijkstra+BF verändert werden. Dafür helfen zwei Beobachtungen:

- In SPMain werden die resultierenden Potenzialfunktionen mehrerer ScaleDown-Aufrufe addiert.
- Die resultierende Potenzialfunktion eines ScaleDown-Aufrufs setzt sich aus der Summe der Ergebnisse von FixDAGEdges, Dijkstra+BF und dem rekursiv erzeugten Rückgabewert von ScaleDown zusammen (siehe Abbildung 5.2).

Daher bietet es sich an, eine globale Potenzialfunktion `prices` in Form eines Arrays zu führen und in den Funktionen FixDAGEdges und Dijkstra+BF die Ergebnisse auf `prices` zu addieren. Somit muss nur einmal der Aufwand für die Erstellung einer solchen Liste aufgebracht werden und nur pro Aufruf der beiden genannten Funktionen die Addition von Potenzialen durchgeführt werden. Folglich hat scaledown keinen Rückgabewert mehr und die restlichen Additionen von Potenzialen in spmain und scaledown fallen weg.

Eine weitere Feststellung ist, dass oft Aufrufe auf Graphen G_ϕ oder G^B gestartet werden, die im Vergleich zum Eingabegraphen G nur eine veränderte Kostenfunktion besitzen. Anstatt

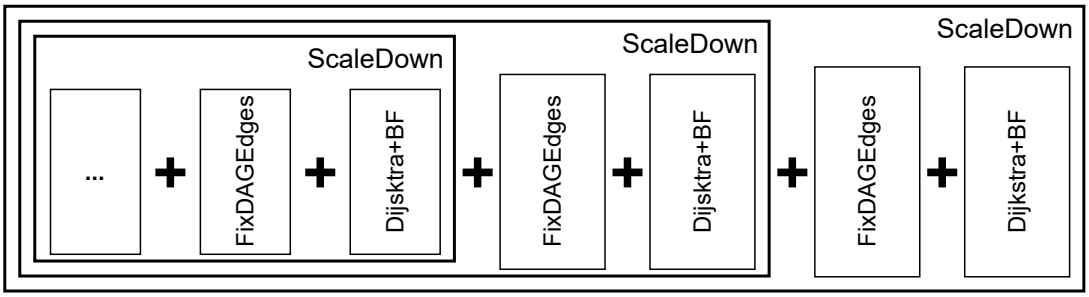


Abbildung 5.2: Zusammensetzung der Potenziale eines ScaleDown-Aufrufs

Listing 5.1 Rückgabe eines Kantengewichts im Graphen G_ϕ^B

```
EdgeWeight getEdgeWeight(EdgeID i){
      const Edge& edge = graph.getEdge(i);
      EdgeWeight weight = edge.weight + prices[edge.source] - prices[edge.target]
      if (weight < 0)
            return weight + B;
      return weight;
}
```

hierfür eine neue Liste von Kosten zu erstellen oder gar den ganzen Graphen zu kopieren, schlagen wir einen ähnlichen Ansatz wie für die Potenzialfunktionen vor.
Eine Abfrage der Kantenkosten findet in den Funktionen LDD, FIXDAGEDGES und DIJKSTRA+BF statt. In allen Fällen sollen die zum Zeitpunkt des Aufrufs aktuellen Potenziale auf die originale Kostenfunktion angewendet werden und anschließend um B erhöht werden. Indem B als globale Variable geführt wird, die nur in SPMAIN verändert wird, kann das geforderte Gewicht einer Kante mit ID i über die Funktion `getEdgeWeight(i)`, dargestellt in Listing 5.1, abgefragt werden. B wird also wie in Algorithmus 4.2 initialisiert, aber nun am Anfang jeder Iteration der for-Schleife halbiert. Die for-Schleife wird dementsprechend abgebrochen, wenn $B = 1$ erreicht ist.

5.2.1 Repräsentation eines Teilgraphen $G \setminus E'$

Ein weiteres Problem in SCALEDOWN ergibt sich durch den rekursiven Aufruf und den Aufruf von FIXDAGEDGES, denn hierbei wird nicht nur auf einem Graphen mit veränderten Kosten sondern auch mit einer Teilmenge von Kanten $G \setminus E'$ gearbeitet. Nun gäbe es zwei Möglichkeiten, dies zu implementieren:
Die erste Möglichkeit ist, einen neuen Teilgraphen zu erstellen, indem eine Transformation durchgeführt wird, bei der die entsprechenden Kanten aus den Datenstrukturen herausgenommen werden. Dafür ergäbe sich allerdings eine neue Indizierung der Kanten, weshalb eine Abbildung der Kantenindizes vom neuen Teilgraphen zum Ursprungsgraphen benötigt würde. Außerdem muss in den Datenstrukturen nach den Kanten E' gesucht werden, was Aufwand $\mathcal{O}(m)$ erfordert. Durch die Wahl einer `offset`-Liste zur Repräsentation ausgehender Kanten

würde zusätzlich eine Sortierung der zu entfernenden Kanten benötigt, wodurch Aufwand $\mathcal{O}(m \log(m))$ anfiel. Das liegt daran, dass das Entfernen einer Kante (u, v) alle Offsets der folgenden Knoten $> u$ verändern kann.
Die zweite Möglichkeit ist, mit sogenannten Masken zu arbeiten, die darüber bestimmen, ob eine Kante im Teilgraphen enthalten ist oder nicht. Diese Masken können als Liste mit booleschen Einträgen oder als Bitmasken, welche zusätzlich schnelle Bit-Operationen erlauben, gespeichert werden. Der Vorteil ist, dass die Konstruktion einer Maske viel einfacher erfolgen kann als die Konstruktion eines neuen Teilgraphen. Zudem ist ein geringerer Speicherverbrauch nötig. Auf der anderen Seite müssen bei Operationen, die nur einen Teilgraphen betreffen, auch die Kanten des Gesamtgraphen betrachtet werden und über eine Abfrage der Maske ausgeschlossen werden.
Bei der Laufzeitanalyse von *ScaleDown* wurde ignoriert, dass *FixDAGEdges* und der rekursive Aufruf *ScaleDown* auf Teilgraphen aufgerufen wird. Stattdessen wurden immer die Größen n und m des Eingabegraphen von Bernsteins Algorithmus verwendet. Da die Maske zum Ausschluss von Kanten immer die Größe m hat, wirkt sich diese Methode nicht auf die asymptotische Komplexität aus.
Letztendlich habe ich mich aufgrund der einfacheren Konstruktion für die zweite Option entschieden.

Nun stellt sich die Frage, wie der Einbau von Bitmasken in ScaleDown funktioniert.
Dafür geben wir ScaleDown statt eines Graphen eine Bitmaske `edgeMask` als Parameter mit, wobei Bit $i = 0$ genau dann gilt, wenn Kante i im Teilgraph enthalten ist. Diese wird in LDD als Parameter weitergegeben, um dort nicht zugehörige Kanten auszuschließen. LDD gibt nun eine neue Bitmaske `removedEdges` basierend auf `edgeMask` zurück, für die Bit $i = 1$ genau dann, wenn i in E^{rem} ist oder nicht im Teilgraphen enthalten ist.
Zur Berechnung der SCCs wird Tarjans Algorithmus, der in Abschnitt 2.4 erklärt wurde, in Kombination mit der Bitmaske `removedEdges` verwendet. Eine besuchte Kante i wird darin dementsprechend ignoriert, wenn `removedEdges[i]==1` gilt.
Der nächste Schritt ist nun, eine Bitmaske zu finden, die die SCC-Kanten E^{SCC} von allen anderen Kanten maskiert. Dazu formen wir die Menge `sepEdges` aus Algorithmus 2.2 in eine Bitmaske um, wobei Bit $i = 1$ gdw. $i \in E^{DAG}$. Diese wird mit `removedEdges` initialisiert. Es ist einfach zu erkennen, dass dadurch am Ende von Tarjans Algorithmus für alle Kanten gilt: $e \in E^{SCC} \iff$ `sepEdges = 0`. Damit erfüllt `sepEdges` genau die Anforderung für die Bitmaske des rekursiven Aufrufs von ScaleDown.
Abbildung 5.3 zeigt, wie die Masken für einen ScaleDown-Aufruf aussehen könnten.
Für den Algorithmus Dijsktra+BF kann die Bitmaske `edgeMask` verwendet werden. Übrig bleibt die Frage nach der Umsetzung von FixDAGEdges, da dort auf einem komprimierten Graphen gearbeitet wird, der SCCs zu einzelnen Knoten zusammenfasst.

Bit	0	1	2	3	4	5	6	7
edgeMask	0	0	0	0	0	0	**1**	**1**
removedEdges	0	**1**	0	0	**1**	0	**1**	**1**
sepEdges	**1**	**1**	0	0	**1**	**1**	**1**	**1**

Abbildung 5.3: Beispiel für Bitmasken in einem ScaleDown-Aufruf

5.2.2 Komprimierung von SCCs zu Knoten

In Abschnitt 4.4 wurde der Algorithmus FixDAGEdges vorgestellt, der negative Kanten E^{DAG}, die zwischen unterschiedlichen SCCs in einem Graphen $G \setminus E^{rem}$ liegen, eliminiert. Dazu müssen jedoch SCCs zu einzelnen Knoten komprimiert werden. In der Praxis muss kein neuer Graph erstellt werden, denn man kann über die ausgehenden Kanten eines SCCs iterieren. Dazu erfordert es allerdings eine geeignete Datenstruktur. Zwar haben wir im erweiterten Tarjan Algorithmus die Menge E^{DAG} der Kanten zwischen SCCs bestimmt, aber nicht den einzelnen SCCs ihre ausgehende Kanten zugewiesen.
Eine simple Möglichkeit, die ausgehenden Kanten eines SCCs zu speichern ist, während Tarjans Algorithmus für jeden Knoten nach Einfügen in einen SCC die SCC-Nummer zwischenzuspeichern. Wir erstellen nun eine doppelt verkettete Liste, die für jeden SCC eine Liste der ausgehenden Kanten speichert, wobei für diese Kanten statt des tatsächlichen Zielknotens die Nummer des Ziel-SCC gespeichert wird. Nachdem der Tarjan-Algorithmus beendet ist, kann nun über alle gefundenen Kanten E^{DAG} iteriert werden und mithilfe des Mappings von Knoten zu SCC die Kanten in die Datenstruktur eingefüllt werden.

5.3 Implementierung von LDD

Eine der zentralen Fragen bei der Implementierung von LDD ist, wie das Entfernen eines Balls aus dem Graphen $G \setminus \mathrm{Ball}_G(v, R)$ realisiert werden kann, denn im Gegensatz zum Hauptalgorithmus von Bernstein werden hierbei auch Knoten entfernt.
Eine Möglichkeit dies zu lösen wäre die Graph-Datenstruktur nach Kanten (u, v) zu durchsuchen, wobei u oder v ein zu entfernender Knoten ist, um diese dann aus der Datenstruktur zu entfernen. Jedoch wird dazu ein Aufwand von $\mathcal{O}(m)$ benötigt.
Es wurde sich deshalb für einen ähnlichen Ansatz wie bei der Herausnahme einzelner Kanten entschieden. Es wird für jeden Aufruf eine Mengendatenstruktur `nodes`, implementiert als Hashtabelle, verwendet, die die Knoten des Teilgraphen speichert. Diese erlaubt es, schnell über alle Knoten des Teilgraphen zu iterieren und zu prüfen, ob ein Knoten Teil des Teilgraphen ist. Jedoch stellte sich in ersten Laufzeittests heraus, dass letztere Prüfung langsamer ist als erwartet. Daher wird zusätzlich ein globales Array verwendet, um diese Prüfung über

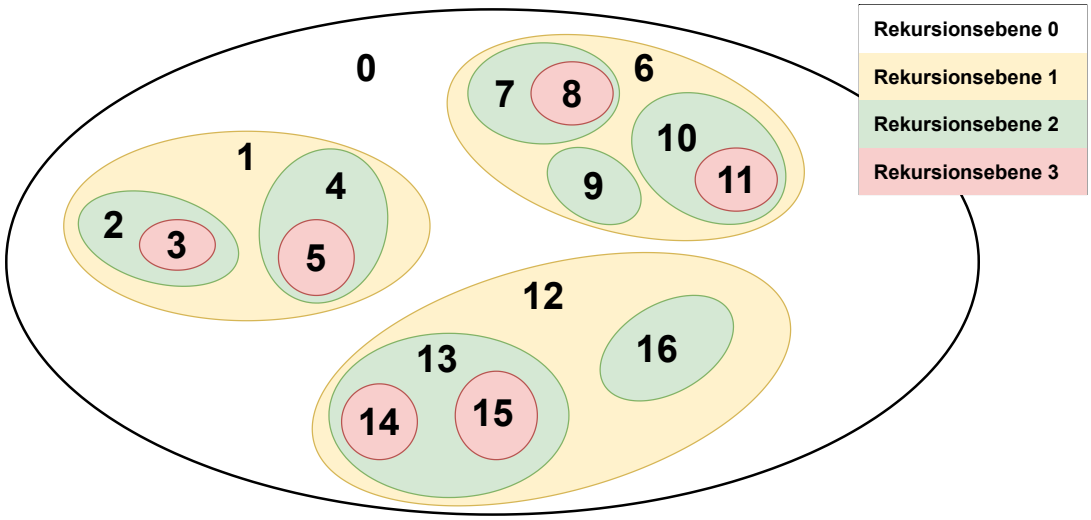


Abbildung 5.4: Vergabe der Rekursionsnummern für `nodeMap`

eine einzige Abfrage zu gewährleisten. Dieses Array `nodeMap` speichert für jeden Knoten des Gesamtgraphen die aktuelle Nummer des Rekursionsaufrufs LDD(G'), zu dem der Knoten dazugehört (siehe Abbildung 5.4). Da in einem Aufruf von LDD ein Knoten nur in maximal einem folgenden Rekursionsaufruf teilnehmen kann, kann mithilfe der Nummer eindeutig bestimmt werden, ob ein Knoten Teil eines Rekursionsaufrufs ist oder nicht.
Das Entfernen eines Knotens aus dem Graphen stellt sich nun als einfach heraus, da er nur aus `nodes` gelöscht werden muss. Die verkleinerte Menge `nodes` wird schließlich für die Durchmesser-Überprüfung der übrig gebliebenen heavy-Knoten am Ende von LDD benötigt.

5.3.1 Ballberechnung

Wie in Abschnitt 4.5.4 angedeutet, erfolgt die Berechnung eines Balls $\mathrm{Ball}^*_G(v, D)$ für $* \in \{in, out\}$ über den Dijkstra-Algorithmus. Hierbei wird ein Min-Heap als Prioritätswarteschlange verwendet, um Knoten mit kurzen Distanzen zuerst abzuarbeiten. Dies hat zur Folge, dass die Berechnung eines Balls statt in $\mathcal{O}(m + n \log(n))$ mithilfe eines Fibonacci-Heaps nun in $\mathcal{O}((m + n) \cdot \log n)$ erfolgt. Jedoch wirkt sich das nicht auf die asymptotische Komplexität von LDD aus, da die nicht-rekursive Laufzeit nun zwar $\mathcal{O}(n \log^2(n) + m \log^2(n))$ beträgt, was unter Einbezug der Voraussetzung, dass jeder Knoten konstant viele ausgehende Kanten hat, zu $\mathcal{O}(n \log^2(n))$ reduziert werden kann.
Die Berechnung eingehender Bälle funktioniert gleich wie die ausgehender Bälle mit dem Unterschied, dass für einen Knoten eingehende statt ausgehender Kanten und Startknoten statt Zielknoten betrachtet werden.
Um zu vermeiden, eine Datenstruktur `distances` zum Speichern der aktuellen Distanzen für

alle Knoten des Gesamtgraphen anzulegen, wird hier eine Map verwendet, die zu Beginn nur die Distanz für den Startknoten v speichert.
Wenn ausgehende (eingehende) Kanten betrachtet werden, wird zuerst überprüft, ob die Kante Teil der `edgeMask` aus Abschnitt 5.2.1 ist und ob der Zielknoten (Startknoten bei eingehenden Bällen) zum Teilgraphen gehört. Letzteres geschieht mithilfe der `nodeMap`. Nur Knoten, die über zulässige Kanten verbesserte Distanzen, die auch kleiner als der Durchmesser D sind, erhalten, werden in `distances` und die Prioritätswarteschlange eingefügt. Dabei gilt für alle Knoten, die keinen Eintrag in `distances` haben, eine Distanz von unendlich.
Die Menge der Knoten des Balls `nodes` wird schließlich aufgebaut, indem bei jedem Herausnehmen eines Knotens aus der Prioritätswarteschlange dieser Knoten in `nodes` eingefügt wird.

Die Randkanten eines Balls werden berechnet, indem alle zulässigen Kanten, die während des Dijkstra-Algorithmus bearbeitet werden, als ID zwischengespeichert werden. Über eine einfache Überprüfung, ob der Zielknoten (Startknoten bei eingehenden Bällen) nicht Teil der `distances`-Map ist, kann eine Kante der Menge der Randkanten zugeordnet werden.

5.3.2 Weitere Optimierungen

Wir erinnern uns, dass LDD, wie in Abschnitt 5.2.1 beschrieben, eine Bitmaske `removedEdges` zurückgibt. Da im gesamten Algorithmus von LDD nur Randkanten eines Balls in der Bitmaske `removedEdges` aktiviert werden, kann die Implementierung optimiert werden, indem schon während der Berechnung der Randkanten in Bällen, die innerhalb der while-Schleife erstellt werden, die entsprechenden Kanten in `removedEdges` aktiviert werden.

In der Menge S der zufällig ausgewählten Knoten können Knoten mehrfach enthalten sein. Dies kann vermehrt dann auftreten, wenn der Teilgraph in einem Aufruf von LDD nur wenige Knoten enthält. Daher ist es eine weitere Optimierung, für mehrfach vorkommende Knoten $s_i \in S$ die Bälle $\mathrm{Ball}^*_{G_0}(s_i, D/4)$ nur einmal zu berechnen. Das kann über den Einsatz einer Map erfolgen, die für jeden unterschiedlichen Eintrag von S die Anzahl der Vorkommen dieses Eintrags in S speichert. Diese Anzahl muss nun bei der anschließenden Berechnung der Schnittmengen $\mathrm{Ball}^*_{G_0}(s_i, D/4) \cap S$, wie sie in Abschnitt 4.5.2 erklärt wurde, berücksichtigt werden.

5.3.3 Asymptotische Komplexität

Offen bleibt noch die Frage, ob sich die Komplexität von LDD durch die Wahl der Datenstrukturen in LDD verändert.
Bei der Ballberechnung muss für jede ausgehende bzw. eingehende Kante eines Knoten mithilfe der `edgeMask` und `nodeMap` geprüft werden, ob die Kante Teil des Teilgraphen ist. Da der Algorithmus konstant viele ausgehende Kanten von einem Knoten voraussetzt, verändert die Überprüfung aller ausgehender Kanten des Eingabegraphen nicht die Komplexität der

ausgehenden Bälle. Da im Durchschnitt ein Knoten auch konstant viele eingehende Kanten hat, trifft dies auch für eingehende Bälle zu. Dadurch, dass wir eine Map für die Speicherung der berechneten Distanzen verwenden, die anfangs nur den Startknoten enthält, kann die Initialisierung des Dijkstra-Algorithmus in konstanter Zeit erfolgen.
Für das Entfernen eines Balls der Größe k aus der Hashtabelle `nodes` wird durchschnittlich ein in k linearer Aufwand benötigt, wodurch sich dies auch nicht auf die asymptotische Laufzeit auswirkt, da insgesamt jeder Knoten nur einmal entfernt wird.

5.4 Alternative zur Goldberg-Skalierung

In Kapitel 4 wurde erklärt, dass Bernsteins Algorithmus, der Kantenkosten $w(e) \geq -1$ voraussetzt, mithilfe Goldbergs Skalierungsframework (siehe 3.3) für beliebige Kosten erweitert werden kann. Allerdings ist die Annahme, dass die in diesem Algorithmus verwendete Funktion *Refine* eine Potenzialfunktion zurückgibt, die die Kantenkosten für Eingabegraphen mit $w(e) \geq -1$ eliminiert. Die Hauptroutine SPMain aus Bernsteins Algorithmus erfüllt dies nicht, da zum einen Potenzialfunktionen für initial skalierte Kantenkosten gefunden werden und zum anderen am Ende die Konstante 1 auf alle Kantengewichte addiert werden. Zwar könnte man über einige Umformungen, bei denen die Potenziale auch rationale Zahlen annehmen, die Eigenschaft von *Refine* erreichen, jedoch gibt es eine bessere Möglichkeit, die auch in dieser Implementierung des Bernstein-Algorithmus verwendet wird.
Anstatt die for-Schleife in SPMain $\log(B)$ mal aufzurufen, wobei B die nächst-größere Zweierpotenz von n ist, setzen wir nun B auf die nächst-größere Zweierpotenz von nW. W sei dabei folgendermaßen definiert:

$$W = \max(1, -\min_{e \in E}(w(e)))$$

Die for-Schleife wird somit $\lceil \log(nW) \rceil = \lceil (\log(n) + \log(W)) \rceil$ mal aufgerufen.

Um zu zeigen, dass diese Vorgehensweise korrekt ist, zeigen wir zuerst, dass der Induktionsanfang erfüllt ist. Nach der Multiplikation aller Kanten mit n gilt $w(e) \geq -Wn$. Der erste ScaleDown-Aufruf nimmt das Argument $B/2 = Wn/2$ entgegen. Somit ist die Vorbedingung $w(e) \geq -2 \cdot Wn/2 = -Wn$ erfüllt.
Die ersten $\lceil \log(W) \rceil$ Iterationen der for-Schleife bezwecken dementsprechend, dass der initiale Zustand vor der Veränderung von SPMain zutrifft, d.h. für $i = \lceil \log(W) \rceil$ gilt $w_{\phi_i}(e) \geq -n$. Die nachfolgenden $\lceil \log(n) \rceil$ Iterationen sorgen dafür, dass für $i = \lceil \log(nW) \rceil$ die Eigenschaft $w_{\phi_i}(e) \geq -1$ gilt, sodass final alle Kosten um 1 erhöht werden können.

Somit verbessert unsere Implementierung sogar die Komplexität von Bernstein, da für den vereinfachten Algorithmus, der die Abwesenheit negativer Zyklen voraussetzt, statt $\mathcal{O}(m \log^5(n) \log W)$ eine Komplexität von $\mathcal{O}(m \log^4(n) \log(Wn))$ zustande kommt.

6 Experimente

Das Ziel dieses Kapitels ist es, mithilfe der beschriebenen Implementierung des Bernstein-Algorithmus einen Laufzeitenvergleich mit bestehenden Algorithmen durchzuführen. Da der Fokus auf den Bernstein-Algorithmus gelegt wird, berücksichtigen wir die in Kapitel 4 genannten Einschränkungen, d.h.

1. Die Kostenfunktion w bildet auf ganzzahlige Werte ab.
2. Jeder Knoten hat konstant viele ausgehende Kanten. Daraus folgt: $m = \mathcal{O}(n)$

Außerdem setzen wir voraus, dass der Eingabegraph keine negativen Zyklen enthält. Dadurch ist die explizite Suche danach, welche für den Bernstein-Algorithmus in Abschnitt 4.6 dargelegt wurde, nicht erforderlich.
Die Experimente erfolgen nacheinander auf drei Klassen von Graphen:

- RAND-Graphen: Dies sind nach gleichem Schema zufällig generierte Graphen. Genauere Details folgen in Abschnitt 6.1.1.
- Straßengraphen aus dem deutschen Straßennetz
- BAD-Graphen: Zufällig generierte Graphen, die für SPFA eine hohe Komplexität hervorrufen.

Für den Vergleich kommen vier Algorithmen zum Einsatz, die in Tabelle 6.1 aufgelistet und zusammengefasst sind. Es ist wichtig anzumerken, dass die jeweiligen Komplexitäten nur unter den zuvor genannten Bedingungen zutreffen.
Alle in diesem Kapitel vorgestellten Experimente wurden auf einem System mit einem *Intel i5 12400* Prozessor und *16 GB* Arbeitsspeicher durchgeführt. Aufgrund von Hardwarebeschränkungen wurde bei den Experimenten keine bis eine Wiederholung auf demselben Eingabegraphen durchgeführt.

6.1 Randomisierte (RAND) Graphen

6.1.1 Graphgenerierung

Die Generierung der randomisierten Graphen basiert auf der Anzahl der Knoten n. Dem Zufallsgenerator liegt eine Gleichverteilung zu Grunde.

Name	Beschreibung	Komplexität
Bernstein-Algorithmus (BERN)	Die in Kapitel 5 vorgestellte Implementierung des Bernstein-Algorithmus.	$\mathcal{O}(m \log^4 n \log nW)$
Goldberg-Algorithmus (GOLD)	Der in Abschnitt 3.4 erklärte Algorithmus von Goldberg.	$\mathcal{O}(m\sqrt{n} \log W)$
Bellman-Ford-Algorithmus (BF)	Die Implementierung des Bellman-Ford-Algorithmus via zwei verschachtelter for-Schleifen. Mit einer quadratischen Laufzeit unter den genannten Bedingungen dient dieser Algorithmus dazu zu zeigen, dass die zuvor genannten Algorithmen asymptotisch schneller sind.	$\mathcal{O}(n \cdot m)$
SPFA	Eine Verbesserung des Bellman-Ford-Algorithmus, die häufig zu sehr schnellen Laufzeiten führt.	$\mathcal{O}(n \cdot m)$

Tabelle 6.1: Verwendete Algorithmen für den Vergleich

Für jeden Knoten werden zufällig zwischen 2 und 10 ausgehende Kanten generiert. Der Zielknoten jeder dieser Kanten wird dabei zufällig unter der Menge aller Knoten V bestimmt. Jede Kante erhält zunächst ein zufälliges Kantengewicht zwischen 0 und 100. Um negative Kantenkosten hinzuzufügen und das Erstellen negativer Zyklen zu verhindern, verwenden wir eine zufällig generierte Potenzialfunktion ϕ, die jedem Knoten ein ganzzahliges Potenzial zwischen 0 und 1000 zuweist. Indem man nun für jede Kante $e = (u, v)$ Kosten $w_\phi(e) = w(e) + \phi(u) - \phi(v)$ definiert, werden Kosten im Intervall von -1000 bis 1100 erzeugt. Nach Beobachtung 2 in Abschnitt 3.2 können durch diese Definition keine negativen Zyklen entstehen.
Durch die in Abschnitt 5.4 erklärte Vorgehensweise kann Bernsteins Algorithmus auf Graphen mit beliebigen ganzzahlige Kantenkosten angewendet werden. Um den Mehraufwand, der für größere W betrieben werden muss, zu umgehen, runden wir in diesem Abschnitt alle negativen Kantenkosten auf -1 auf. Auch dadurch können keine neuen negativen Zyklen entstehen. In Abschnitt 6.4 wird untersucht, wie der Algorithmus auf Graphen mit kleineren negativen Kantenkosten abschneidet.
Da es durch diese Generierung vorkommen kann, dass eine Vielzahl von Knoten von einem bestimmten Knoten aus nicht erreichbar ist, wurde für jeden Graph ein fester Startknoten gewählt. Dadurch konnte durch eine initiale Überprüfung sichergestellt werden, dass für alle Testgraphen von diesem Knoten aus eine Erreichbarkeit der Gesamtknotenmenge von über 90% zutrifft. Dies schließt große Laufzeitausbrüche in SPFA aus.

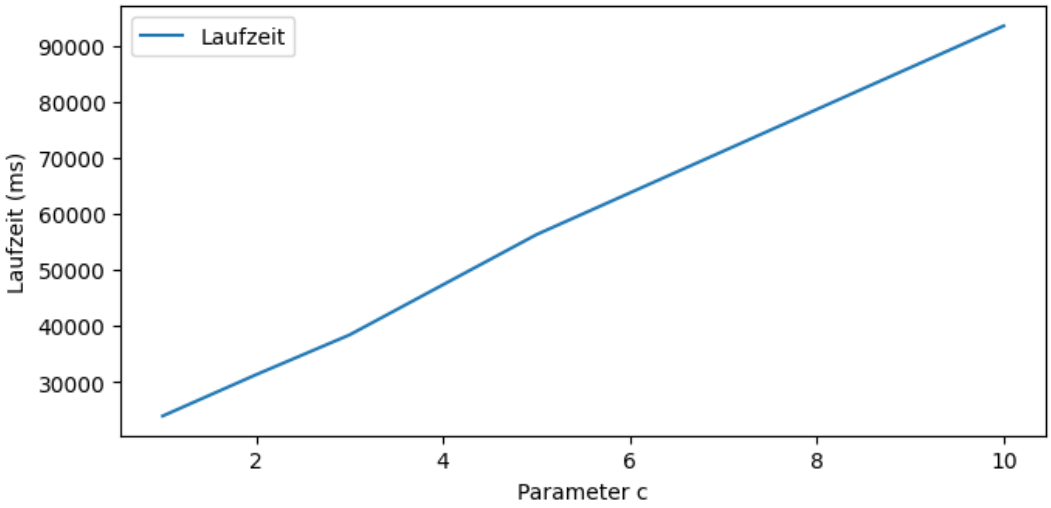


Abbildung 6.1: Einfluss von Parameter c auf die Laufzeit

6.1.2 Parametereinstellung von c

Zuerst muss begutachtet werden, wie der Parameter c in LDD, der die Anzahl der zufällig ausgewählten Knoten k beeinflusst, sich auf die Laufzeit des Bernstein-Algorithmus auswirkt. Ein höherer Parameter sorgt in der Theorie für eine geringere Abbruchwahrscheinlichkeit bei den beiden Überprüfungen der Ballkardinalität in LDD. Für das Experiment verwenden wir einen RAND-Graphen mit 10^5 Knoten und testen Werte zwischen 1 und 10.
Abbildung 6.1 veranschaulicht, dass die Laufzeit linear mit der Zunahme des Parameters c steigt. Das liegt daran, dass die Anzahl der generierten Bälle auch linear zunimmt. Wie wir später noch sehen werden, ist die Ballgenerierung ein Bottleneck des Bernstein-Algorithmus.
Daher wurde die Entscheidung getroffen, den Parameter $c = 1$ zu verwenden, um die Laufzeiten so gering wie möglich zu halten.

6.1.3 Vergleich

Nun gilt es, die Laufzeiten des Bernstein-Algorithmus auf RAND-Graphen mit denen der drei anderen Algorithmen zu vergleichen. Alle Algorithmen wurden auf jedem Graphen zweimal durchlaufen und der Mittelwert der beiden Durchläufe wird als Laufzeit aufgetragen.
Abbildung 6.2 zeigt die Ergebnisse in einem Diagramm mit logarithmischer x- und y-Skala. Getestet wurde auf Graphen im Bereich zwischen $n = 10^3$ und $n \approx 10^6$. Zur Einordnung der Kurven wurden zusätzlich zwei gestrichelte Kurven für jeweils eine lineare Steigung und eine quadratische Steigung der Laufzeit hinzugefügt. Die y-Achsenverschiebung dieser beiden Kurven ist dabei zu ignorieren, d.h. nur die Steigungen sind zu beachten.
Es fallen folgende Beobachtungen auf:

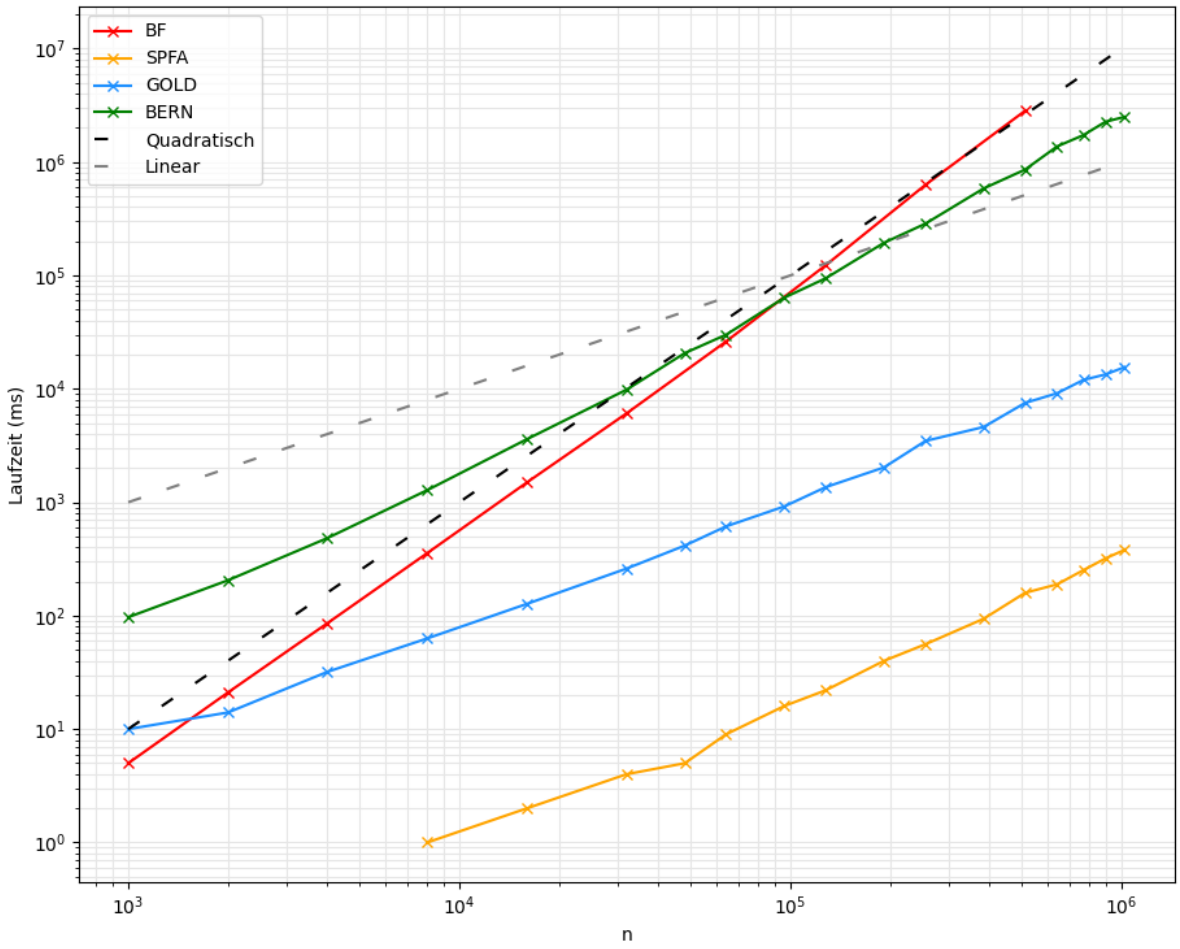


Abbildung 6.2: Laufzeiten-Vergleich auf RAND-Graphen

- Die Laufzeit des simplen Bellman-Ford-Algorithmus (BF) steigt in etwa quadratisch mit der Anzahl der Knoten, wie zu erwarten war.
- Alle Kurven der anderen Algorithmen steigen schwächer als die quadratische Kurve aber stärker als die lineare Kurve.
- SPFA ist in allen Messungen schneller als BF, wie zu erwarten war.
- GOLD ist ab ca. 1500 Knoten schneller als BF. BERN hingegen übertrumpft BF erst ab ca. 90000 Knoten.
- SPFA ist in allen Testgraphen am schnellsten. Danach folgt der Goldberg-Algorithmus, der jedoch im Messbereich durchschnittlich in etwa 56 mal langsamer ist als SPFA. Auf dem dritten Platz folgt der Bernstein-Algorithmus, der wiederum zwischen 10 und 160 mal langsamer als GOLD ist.

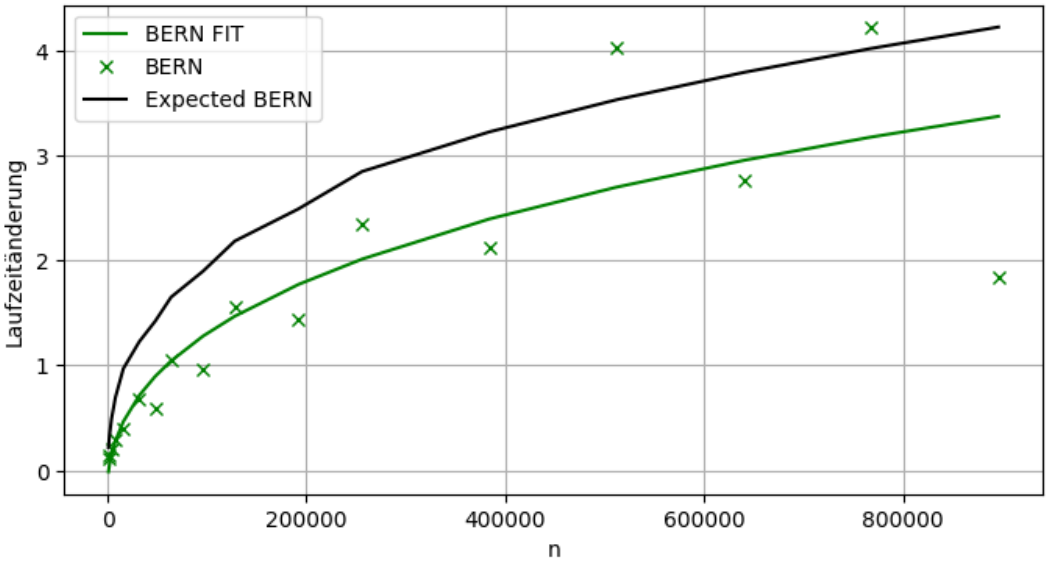


Abbildung 6.3: Vergleich der Steigungen auf RAND-Graphen

- Die Steigung von SPFA ist größer als die von GOLD. Das bedeutet, dass auf weitaus größeren Graphen SPFA von GOLD eingeholt werden kann. Jedoch ähnelt die Steigung von SPFA in diesem Messintervall der Steigung von BERN.

6.1.4 Steigungsverhalten

Um einen genaueren Blick auf die Steigungen von BERN zu werfen, werden in Abbildung 6.3 die Steigungen zwischen zwei aufeinander folgenden Datenpunkten aufgetragen. Die Kurve „Expected BERN“ gibt dabei die Ableitungsfunktion der erwarteten Komplexitätsfunktion $c \cdot n \log(n)^5$ wieder und dient somit der Einordnung. Die Kurve „BERN FIT“ bildet eine passende Funktion zu den einzelnen Datenpunkten, indem sie die Parameter a, b und c der Funktion $f(n) = a \cdot \log(n)^b + c$ berechnet.
Unter Einbezug von lokalen Laufzeitschwankungen kann erkannt werden, dass in unserer Implementierung des Bernstein-Algorithmus die Steigungsänderung stetig abnimmt. Das bedeutet, dass dieser Ansatz einen klaren Vorteil gegenüber Algorithmen mit quadratischer Laufzeit wie Bellman-Ford hat, da bei Letzteren die Laufzeitänderung linear zunimmt.

6.1.5 Laufzeitaufteilung

Da beobachtet werden konnte, dass der Bernstein-Algorithmus im Vergleich zu den anderen Algorithmen im verwendeten Messintervall deutlich langsamer ist, stellt sich die Frage, ob und welche Bestandteile des Algorithmus besonders große Anteile der Laufzeit beanspruchen.

Graph	$n = 64000$		$n = 128000$	
Bestandteil	Zeit (ms)	Anteil	Zeit	Anteil
Gesamt	30911	100%	87248	100%
Dijkstra+BF	1746	5.6%	4430	5.1%
TarjanSCC	4506	14.6%	10150	11.6%
LDD	21290	68.9%	63633	73.0%
LDD, davon Ball	13648	44.2%	44855	51.4%
FixDAGEdges	2336	7.6%	6492	7.4%

Tabelle 6.2: Teillaufzeiten des Bernstein-Algorithmus auf RAND-Graphen

Dafür wurden die einzelnen Laufzeiten folgender Teilroutinen in einem Durchlauf auf RAND-Graphen mit 64000 und 128000 Knoten gemessen:

- Dijkstra+BF
- SCC-Bestimmung (erweiterter Tarjan-Algorithmus)
- LDD
- Ball-Berechnungen (Bestandteil von LDD)
- FixDAGEdges

In Tabelle 6.2 ist zu erkennen, dass LDD den größten Anteil der Gesamtlaufzeit beansprucht. Diese Tatsache ist hauptsächlich den Ballberechnungen, welche Bestandteil von LDD sind, geschuldet. Der Grund dafür ist, dass für jede Ballberechnung der Dijkstra-Algorithmus ausgeführt werden muss und in einem Durchlauf von LDD durchschnittlich in etwa n Bälle erstellt werden. Für größere Graphen scheint der Anteil der Ballberechnungen und somit LDD sogar größer zu werden.
Die anderen Bestandteile von Bernsteins Algorithmus spielen eine untergeordnete Rolle. Ein kleiner Ausreißer kann bei der SCC-Berechnung verzeichnet werden, was man damit erklären kann, dass die zusätzliche Berechnung ausgehender Kanten eines SCC einen gewissen Mehraufwand zum Tarjan-Algorithmus erfordert.

6.1.6 Überspringen von LDD

Auch wenn die angegebene Komplexität von $\mathcal{O}(m \cdot \log^5(m) \log W)$ durch das Überspringen von LDD nicht gewährleistet werden kann, stellt sich trotzdem die Frage, wie sich das Überspringen auf die Laufzeit auswirkt. Um dies umzusetzen, werden nun automatisch keine Kanten als entfernt markiert, wodurch die resultierenden SCCs beliebige Durchmesser haben. Somit ist die Rekursion in ScaleDown auch unnötig, weshalb der Rekursionsaufruf auf dem Teilgraphen H der SCCs durch einen Aufruf *Dijsktra+BF* auf diesem Teilgraphen ersetzt wird. Die Ergebnisse sind in Abbildung 6.4 dargestellt.

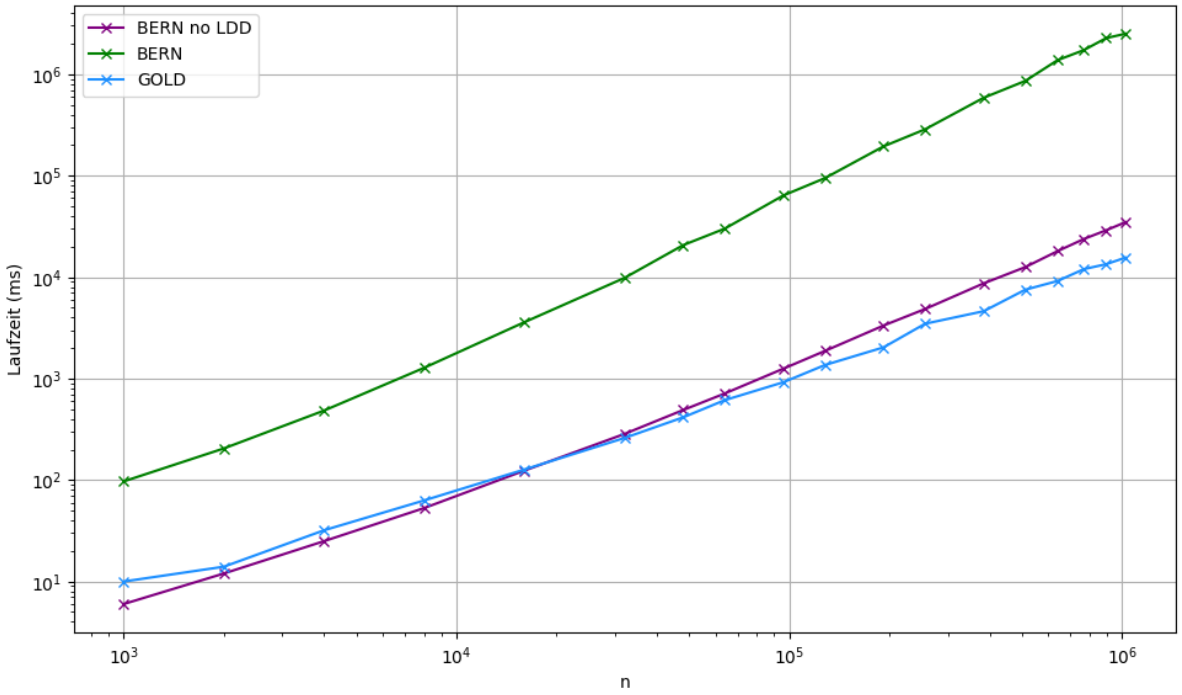


Abbildung 6.4: Laufzeitenvergleich unter Ausnahme von LDD

Unter Berücksichtigung der Erkenntnis aus vorherigem Abschnitt, dass LDD den Großteil der Laufzeit von Bernsteins Algorithmus beansprucht, ist es offensichtlich, dass die Kurve von „BERN no LDD“ tiefer positioniert ist. Tatsächlich ist BERN ohne LDD ähnlich schnell wie GOLD. Allerdings kann man klar erkennen, dass BERN ohne LDD stärker steigt als GOLD. Dadurch, dass die Kurve von BERN sogar leicht stärker steigt als die von „BERN no LDD“, zeigt auch, dass LDD in diesem Messintervall keinen wirklichen Mehrwert zur Laufzeit beiträgt. Ein Grund, der diese Beobachtung erklären könnte, ist, dass „BERN no LDD“ zum Eliminieren des Großteils der Kanten Dijksta+BF verwendet. Dijkstra+BF weist gewisse Ähnlichkeiten zu SPFA auf, welcher, wie in Abschnitt 6.1.3 gezeigt, auf RAND-Graphen deutlich schneller als die quadratische Worst-Case Komplexität ist.

6.2 Straßengraphen

In einem weiteren Experiment werden Straßengraphen aus dem deutschen Straßennetz verwendet, welche eine andere Struktur aufweisen als die zufällig generierten Graphen aus vorherigem Abschnitt. Um auch hier keine negativen Zyklen zu erzeugen, wird auf die ursprüngliche Kostenfunktion wie zuvor eine zufällig generierte Potenzialfunktion mit Werten von 0 bis 1000 angewendet. Negative Kantenkosten werden auf -1 aufgerundet.
Zum Einsatz kommt das Straßennetz vom Bezirk Stuttgart und Mecklenburg-Vorpommern

(MV). Der Stuttgart-Graph kommt mit 1132113 Knoten und 2292887 Kanten, während der MV-Graph mit 644199 Knoten und 1305996 Kanten etwa halb so groß ist. Der Anteil negativer Kanten beträgt auf dem MV-Graph in etwa 26% und auf dem Stuttgart-Graph etwa 32%.
Tabelle 6.3 zeigt die gemessenen Laufzeiten. Die Hierarchie der schnellsten Algorithmen bleibt bestehen. SPFA läuft jedoch auf diesen Straßengraphen knapp 50% langsamer als auf ähnlich großen RAND-Graphen. Für GOLD und BERN zeichnet sich ein umgekehrtes Bild ab. GOLD liefert hier in etwa 3 mal schnellere Laufzeiten, BERN sogar 4 mal schnellere Laufzeiten als auf RAND-Graphen. Gründe sind die geringeren Anteile negativer Kanten an der Gesamtkantenanzahl auf Straßengraphen und auch das geringere Verhältnis von etwa $2:1$ zwischen Kanten und Knoten.

Graph	SPFA [ms]	GOLD [ms]	BERN [ms]
MV ($n = 644199$)	277	3056	293642
Stuttgart ($n = 1132113$)	624	5388	636112

Tabelle 6.3: Laufzeiten auf Straßengraphen

6.3 Konstruierter Graph

In einem weiteren Experiment soll herausgefunden werden, ob es Graphen gibt, für die Goldbergs Algorithmus und bestenfalls Bernsteins Algorithmus in der hier verwendeten Eingabegraphen-Größenordnung von $< 10^6$ Knoten schneller sind als SPFA. Hierzu konstruieren wir einen „Worst-Case"-Graphen, für den SPFA möglichst langsam läuft. Wir möchten darauf hinweisen, dass hierbei die Bernsteins Voraussetzung, dass Eingabegraphen konstant viele ausgehende Kanten aus einem Knoten haben, ignoriert wird. In der Theorie kann somit nicht die Komplexität von Dijkstra+BF gewährleistet werden.

Definition 6.3.1 (BAD Graph)
Seien alle Knoten von 0 *bis* $n-1$ *durchnummeriert. Es werden Kanten* $(0, i) \quad \forall 2 \leq i \leq n-1$ *hinzugefügt. Außerdem werden Kanten* $(i, i+1) \quad \forall 0 \leq i \leq n-2$ *hinzugefügt. Alle Kanten erhalten das Gewicht* -1.

Der BAD-Graph ist in Abbildung 6.5 veranschaulicht. Gesucht sind nun kürzeste Pfade ausgehend von Knoten 0. Es ist einfach zu erkennen, dass für alle $i \in V$ die Distanz zu Knoten i genau $-i$ beträgt.
Die Laufzeit von SPFA auf diesem Graph hängt davon ab, in welcher Reihenfolge ausgehende Kanten abgearbeitet werden. Da wir das Worst-Case Szenario hervorrufen wollen, sorgen wir dafür, dass für Knoten 0 zuerst die ausgehenden Kanten mit größerer Zielknotennummer betrachtet werden. Dadurch werden über diese Kanten nie final korrekte Distanzen berechnet. Wenn also eine Kante $(0, i)$ mit $i > 1$ betrachtet wurde, wird i und danach alle nachfolgenden

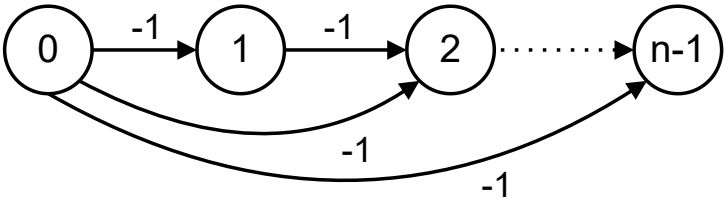


Abbildung 6.5: BAD Graph

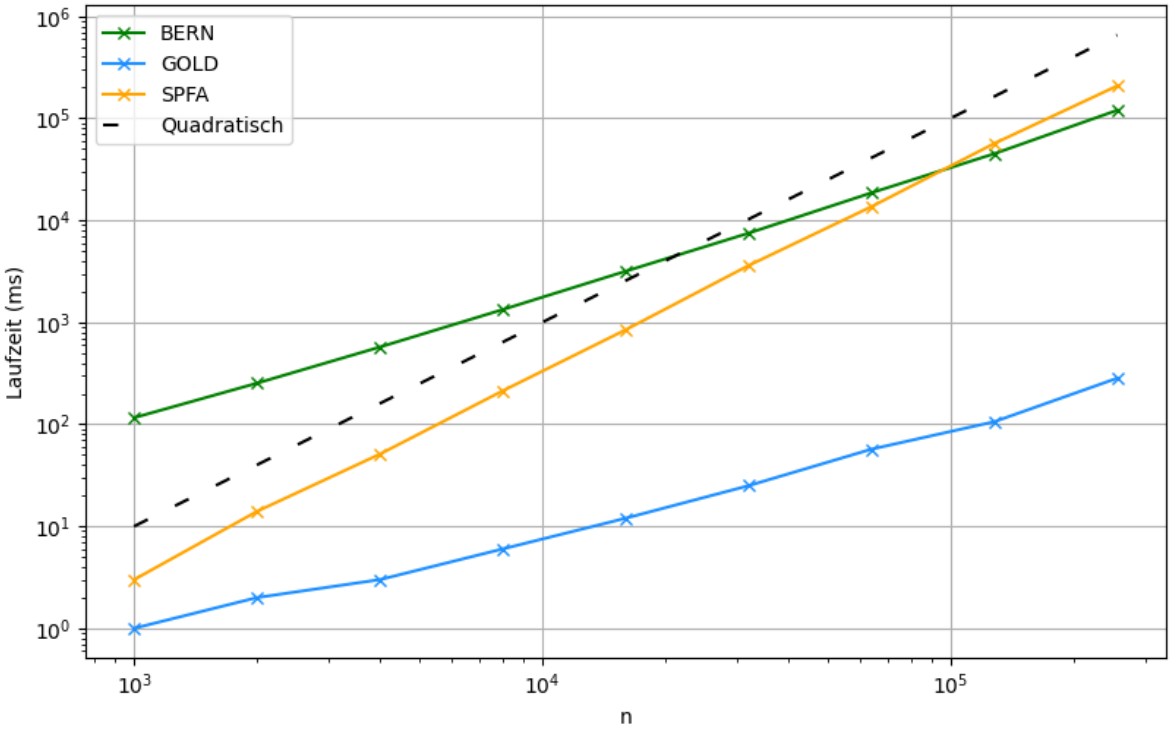


Abbildung 6.6: Laufzeitenvergleich auf BAD Graphen

Knoten $j > i$ in die Queue eingefügt. Jedoch sind alle in Folge dessen berechneten Distanzen nicht final korrekt. Erst wenn die Kante $(0, 1)$ betrachtet wird, können die finalen Distanzen bestimmt werden. Damit entsteht ein quadratischer Aufwand.

Abbildung 6.6 zeigt nun die Ergebnisse der Messungen. Wie angenommen, steigt die Kurve von SPFA quadratisch. Tatsächlich ist Bernsteins Algorithmus somit ab einer Graphgröße von etwa 10^5 Knoten schneller als SPFA. Jedoch ist Goldbergs Algorithmus trotzdem auf BAD-Graphen durchschnittlich etwa 265 mal schneller als BERN. Das liegt daran, dass die Knoten in einem Pfad angeordnet sind, die in einer einzigen Iteration eliminiert werden (siehe Abschnitt 3.4.1). Insgesamt kann also bestätigt werden, dass Graphen existieren, für die GOLD und BERN schon bei kleinerer Knotenanzahl schneller sind als SPFA.

Graph	GOLD [ms]	BERN [ms]
$n = 64000, w(e) \geq -1$	632	31174
$n = 64000, w(e) \geq -10$	1734	56340
$n = 64000, w(e) \geq -100$	4004	77740
$n = 128000, w(e) \geq -1$	1497	98910
$n = 128000, w(e) \geq -10$	3962	164712
$n = 128000, w(e) \geq -100$	7442	226103

Tabelle 6.4: Laufzeiten auf Straßengraphen

6.4 Einfluss des maximalen negativen Kantengewichts W

In allen vorherigen Experimenten wurde auf Graphen gearbeitet, deren Kantenkosten größer oder gleich -1 sind. Für die beiden Skalierungsalgorithmen GOLD und BERN ergibt sich ein zusätzlicher von W abhängiger Faktor in der Komplexität, wenn der Eingabegraph beliebige negativen Kosten aufweist. Daher werden die Algorithmen GOLD und BERN in einem weiteren Experiment auf RAND-Graphen zweier Größen getestet, die genauso wie in Abschnitt 6.1.1 konstruiert werden, aber deren negative Kantenkosten auf drei verschiedene Werte aufgerundet werden. Dadurch entstehen Graphen mit Eigenschaften $w(e) \geq -1$, $w(e) \geq -10$ und $w(e) \geq -100$.

Anhand der Messwerte in Tabelle 6.4 ist zu erkennen, dass sich bei beiden Graphgrößen höhere Werte für W negativ auf die Laufzeit auswirken. Jedoch steigen die Laufzeiten von GOLD bei größerem W stärker an als bei BERN. Das ist mit der Tatsache zu erklären, dass GOLD den Skalierungsansatz von Goldberg verwendet, der einen zusätzlichen Faktor $\log(W)$ in der Komplexität hervorruft. Währenddessen ist in unserer Implementierung von Bernstein W Teil eines anderen $\log$-Faktors in der Komplexität, d.h. hier ist nur ein kleinerer Anstieg von $\log(n)$ zu $\log(nW)$ zu vermerken.

7 Zusammenfassung und Ausblick

Ziel dieser Bachelorarbeit war es, eine Implementierung des Bernstein-Algorithmus (BERN) zu entwickeln und die Laufzeiten des Algorithmus mit anderen, bestehenden SSSP-Problem-Algorithmen zu vergleichen. Für den Vergleich wurden der Goldberg-Algorithmus (GOLD) und zwei Varianten des Bellman-Ford Algorithmus (BF und SPFA) herangezogen. Es wurden Eingabegraphen dreier Graphklassen verwendet, die aus einer Knotenanzahl von 10^3 bis $\sim 10^6$ bestanden.

Wir konnten eine Implementierung produzieren, deren asymptotische Komplexität die von Bernstein erreicht und sogar auf $\mathcal{O}(m \log(n)^4 \log(nW))$ verbessert. Anhand eines Steigungdiagramms konnte belegt werden, dass BERN in den Experimenten eine nahe-lineare Laufzeit erreicht, wie sie bei der Vorstellung der Arbeit von Bernstein [BNWN22] angegeben wurde. Damit ist BERN auf RAND-Graphen ab ca. 90000 Knoten schneller als die einfachste Implementierung BF des Bellman-Ford-Algorithmus.
Allerdings ergeben sich einige Nachteile, die die praktische Anwendbarkeit des Algorithmus im Vergleich zu Goldberg und SPFA, der die besten Ergebnisse erzielen konnte, deutlich einschränken:

- Auf RAND-Graphen ist BERN in unseren Experimenten selbst bei einem Graphen mit 10^6 Knoten in etwa 6500-mal langsamer als SPFA, welcher der schnellste getestete Algorithmus ist. Auch im Vergleich mit Goldbergs Algorithmus ist Bernstein in unserem Messintervall zwischen 10 und 160 mal langsamer. Auf Straßengraphen zeichnet sich ein ähnliches Bild ab, wobei BERN und GOLD hier besser abschneiden. Damit BERN an die Laufzeiten von SPFA herankommt, wären Eingabegraphen in nicht absehbaren, höheren Größenordnungen nötig.
- In den Laufzeittests wurden zuerst nur Graphen mit Kantenkosten $w(e) \geq -1$ verwendet. Auch wenn alle getesteten Graphen einschließlich BERN für beliebige ganzzahlige Kantenkosten funktionieren, sind GOLD und BERN für kleinere minimale negative Kantenkosten W noch langsamer, weil ein zusätzlicher von W abhängiger Faktor die Laufzeit verschlechtert. Dies konnte in einem separaten Experiment belegt werden. Die Varianten des Bellman-Ford Algorithmus hingegen sind robust gegen diesen Wert W.
- Getestet wurde auch nur die reduzierte Version des Bernstein-Algorithmus, die die Abwesenheit negativer Zyklen voraussetzt. Zwar stellten Bernstein et al. eine Erweiterung vor, um die Existenz negativer Zyklen zu erkennen. Jedoch wurde in Abschnitt 4.6 diskutiert, dass diese Erweiterung, die für den Monte-Carlo-Algorithmus einen zusätzliche Laufzeitfaktor $\log(n)$ und für den Las-Vegas Algorithmus einen zusätzlichen Faktor $\log^3(n)$

erfordert, in der Praxis unpraktikabel ist.
Im Gegensatz dazu bieten BF, SPFA und GOLD Ansätze an, negative Zyklen zu erkennen, ohne dass die Komplexität und damit auch die tatsächliche Laufzeit verschlechtert wird.

Trotzdem wurde anhand der BAD-Graphen gezeigt, dass es Szenarien geben kann, in denen SPFA schon ab Graphen mit 10^5 Knoten langsamer ist als BERN. Die BAD-Graphen wurden jedoch absichtlich so konstruiert, dass die Laufzeit von SPFA maximiert wird. In praktischen Anwendungsfällen werden Graphen in seltensten Fällen eine solch spezielle Struktur aufweisen.

Somit kann zusammengefasst werden, dass der Bernstein-Algorithmus zwar implementiert werden kann, sodass eine nahe-lineare Zeit erreicht wird, aber für praktische Anwendungsfälle zu langsam ist, um eine sinnvolle Alternative zu bestehenden Algorithmen wie SPFA anzubieten.

Ausblick

Da in den Experimenten gezeigt werden konnte, dass der Teilalgorithmus „Low Diameter Decomposition“ den weitaus größten Bestandteil der Gesamtlaufzeit von Bernsteins Algorithmus einnimmt, ist es eine weitere Forschungsfrage, ob und wie ein effizienterer Algorithmus für LDD beschrieben werden kann, der dieselben Anforderungen erfüllt.

Neben dieser Arbeit wurde im Jahr 2023 von Bringmann et al. eine Überarbeitung des Bernstein-Algorithmus [BCF23] präsentiert, die mit $\mathcal{O}(m \log^2(n) \log(nW) \log\log n)$ weniger $\log$-Faktoren und damit eine weitaus bessere Komplexität verspricht als Bernsteins Algorithmus. In einer weiteren Untersuchung könnte also untersucht werden, wie die überarbeitete Version im Vergleich mit Bernsteins Version und anderen SSSP-Problem-Algorithmen abschneidet.

Literaturverzeichnis

[BCF23] K. Bringmann, A. Cassis, N. Fischer. „Negative-Weight Single-Source Shortest Paths in Near-Linear Time: Now Faster!“ In: *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2023, S. 515–538 (zitiert auf S. 72).

[Bel58] R. Bellman. „On a routing problem“. In: *Quarterly of applied mathematics* 16.1 (1958), S. 87–90 (zitiert auf S. 15, 23).

[BNWN22] A. Bernstein, D. Nanongkai, C. Wulff-Nilsen. „Negative-weight single-source shortest paths in near-linear time“. In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, S. 600–611 (zitiert auf S. 16, 33, 35, 48, 50, 71).

[CFR22] N. Cao, J. T. Fineman, K. Russell. „Parallel Shortest Paths with Negative Edge Weights“. In: *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 2022, S. 177–190 (zitiert auf S. 31).

[CGR94] B. V. Cherkassky, A. V. Goldberg, T. Radzik. „Shortest paths algorithms: Theory and experimental evaluation“. In: *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. 1994, S. 516–525 (zitiert auf S. 24).

[CKL+22] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, S. Sachdeva. „Maximum flow and minimum-cost flow in almost-linear time“. In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, S. 612–623 (zitiert auf S. 30).

[CMSV17] M. B. Cohen, A. Mądry, P. Sankowski, A. Vladu. „Negative-weight shortest paths and unit capacity minimum cost flow in õ (m 10/7 log w) time*“. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2017, S. 752–771 (zitiert auf S. 27).

[Dij59] E. W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische mathematik* 1.1 (1959), S. 269–271 (zitiert auf S. 15, 26).

[EK72] J. Edmonds, R. M. Karp. „Theoretical improvements in algorithmic efficiency for network flow problems“. In: *Journal of the ACM (JACM)* 19.2 (1972), S. 248–264 (zitiert auf S. 26).

[FMSN98] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. „Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights“. In: *European Symposium on Algorithms*. Springer. 1998, S. 320–331 (zitiert auf S. 30).

[For56] L. R. Ford. „Network flow theory“. In: (1956) (zitiert auf S. 15, 23).

[FR01] J. Fakcharoenphol, S. Rao. „Planar graphs, negative weight edges, shortest paths, and near linear time“. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE. 2001, S. 232–241 (zitiert auf S. 30).

[FT87] M. L. Fredman, R. E. Tarjan. „Fibonacci heaps and their uses in improved network optimization algorithms“. In: *Journal of the ACM (JACM)* 34.3 (1987), S. 596–615 (zitiert auf S. 15, 26).

[Gar85] H. N. Garbow. „Scaling algorithms for network problems“. In: *Journal of Computer and System Sciences* 31.2 (1985), S. 148–168 (zitiert auf S. 26).

[Gol95] A. V. Goldberg. „Scaling algorithms for the shortest paths problem“. In: *SIAM Journal on Computing* 24.3 (1995), S. 494–504 (zitiert auf S. 27, 29, 30).

[Joh77] D. B. Johnson. „Efficient algorithms for shortest paths in sparse networks“. In: *Journal of the ACM (JACM)* 24.1 (1977), S. 1–13 (zitiert auf S. 34).

[Kah62] A. B. Kahn. „Topological sorting of large networks“. In: *Communications of the ACM* 5.11 (1962), S. 558–562 (zitiert auf S. 19).

[Moo59] E. F. Moore. „The shortest path through a maze“. In: *Proc. of the International Symposium on the Theory of Switching*. Harvard University Press. 1959, S. 285–292 (zitiert auf S. 23).

[Pap74] U. Pape. „Implementation and efficiency of Moore-algorithms for the shortest route problem“. In: *Mathematical programming* 7 (1974), S. 212–222 (zitiert auf S. 24).

[RZ04] L. Roditty, U. Zwick. „On dynamic shortest paths problems“. In: *Algorithms–ESA 2004: 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004. Proceedings 12*. Springer. 2004, S. 580–591 (zitiert auf S. 30).

[SW81] D. R. Shier, C. Witzgall. „Properties of labeling methods for determining shortest path trees“. In: *JOURNAL OF RESEARCH of the National Bureau of Standards* 86.3 (1981), S. 317 (zitiert auf S. 24).

[Tar72] R. Tarjan. „Depth-first search and linear graph algorithms“. In: *SIAM journal on computing* 1.2 (1972), S. 146–160 (zitiert auf S. 21).

[Tar76] R. E. Tarjan. „Edge-disjoint spanning trees and depth-first search“. In: *Acta Informatica* 6.2 (1976), S. 171–185 (zitiert auf S. 19).

[YZ05] R. Yuster, U. Zwick. „Answering distance queries in directed graphs using fast matrix multiplication“. In: *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*. IEEE. 2005, S. 389–396 (zitiert auf S. 30).

[Zwi08] U. Zwick. „Lecture notes for Änalysis of Algorithms": Computing shortest paths and detecting negative cycles“. In: (Juli 2008) (zitiert auf S. 27).

Alle URLs wurden zuletzt am 30. 03. 2024 geprüft.

Erklärung

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

Ort, Datum, Unterschrift