k-Shortest Path: Average-Case Analysis and Practical Improvements

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe – Universität

in Frankfurt am Main

von

Alexander Schickedanz

aus Offenbach am Main

Frankfurt (2023)

(D 30)

vom Fachbereich 12 der
Johann Wolfgang Goethe – Universität als Dissertation angenommen.

Dekan: Prof. Dr. M. Möller

Gutachter: Prof. Dr. U. Meyer, Asst. Prof. Dr. D. Ajwani

Datum der Disputation: 06.10.2023

Acknowledgements

First and foremost, I would like to thank my advisor, Ulrich Meyer. He not only guided and supported me during the whole process, he also introduced me to Deepak Ajwani. Deepak Ajwani in turn I am extremely grateful for bringing me into k-shortest path and supporting me on the way. This endeavor would not have been possible without both of them.

I am also sincerely grateful to Manuel Penschuck for the many valuable discussions over the years – both scientific and private – as well as the helpful feedback for this thesis. Special thanks goes to my close friends Kathrin Lehmer, Jonathan Weinberger, and Katharina Beck for proofreading my thesis and tolerating my lack of time.

During the past six years I had the pleasure of working with the great people of the "third floor" especially Mario Holldack, Hannes Seiwert, Hung Tran, David Hammer, and Claudia Gressler. It was a fantastic and exciting time.

Last but not least, I am grateful to my family and friends for their support and encouragement all along.

Alexander Schickedanz
October 18, 2023

This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) under grant ME 2088/4-2.

Deutsche Zusammenfassung

Das Kürzeste-Wege-Problem ist eines der klassischen und fundamentalen Probleme in der theoretischen Informatik. Eine Verallgemeinerung ist das sogenannte k-Kürzeste-Wege-Problem. Dabei werden in einem Graphen G mit n Knoten und m Kanten die k kürzesten Wege zwischen zwei Knoten s und t in nicht-absteigender Reihenfolge nach der Gesamtlänge gesucht. Wir beschäftigen uns ausschließlich mit der Variante des k-Kürzeste-Wege-Problems, in der nur kreisfreie Wege betrachtet werden, d. h. kein Knoten auf einem solchen Weg darf mehrfach besucht werden.

Auf englisch “loopless” oder “simple” genannt.

Das k-Kürzeste-Wege-Problem taucht in vielen Anwendungen auf wie z. B. Routing in verschiedenen Netzwerken [38, 36, 78, 6, 64, 34, 61] und vielfältigen anderen Optimierungsproblemen wie z. B. die Konstruktion virtueller Netzwerke [69], Chip-Layout Optimierung [37, 71], Optimierung von Gruppentestungen [7] und Optimierung von Datenbankanfragen [68].

Einer der bekanntesten k-Kürzeste-Wege-Algorithmen ist Yens Algorithmus [76]. Ausgehend vom i-ten kürzesten Weg P_i, wird für jeden Knoten auf dem Weg eine kürzeste Verzweigung berechnet und diese in einer Liste von Kandidaten gespeichert. Eine Verzweigung ist hier ebenfalls ein Weg von s nach t, der bis zum Verzweigungsknoten mit dem Weg, von dem er abzweigt, übereinstimmt. Vom Verzweigungsknoten aus wird dann eine andere Kante verwendet als im ursprünglichen Weg P_i. Für den restlichen Verlauf der Verzweigung gibt es keine weiteren Einschränkungen, d. h. die Verzweigung kann im Anschluss wieder Kanten verwenden, die bereits in P_i verwendet wurden, solange die Verzweigung kreisfrei bleibt. Der $i + 1$ kürzeste Weg wird dann aus der Liste von Kandidaten ausgewählt, die sowohl die neuen Kandidaten des i-ten Wegs als auch die übrigen Kandidaten aus den ersten $i - 1$ kürzesten Wege enthält. Yens Algorithmus kommt auf eine Worst-Case-Komplexität von $\mathcal{O}(kn \cdot \mathrm{spc}(n, m))$, wobei $\mathrm{spc}(n, m)$ die Laufzeit des verwendeten Kürzeste-Wege-Algorithmus ist. Die Komplexität ergibt sich daraus, dass für jeden Knoten auf jedem der k Wege eine Verzweigung berechnet werden muss und schlimmstenfalls jeder Weg $\Theta(n)$ Knoten hat.

Yens Algorithmus: ☞ *Abschnitt 3.2.1*

Seit Yen seinen Algorithmus 1971 vorgestellt hat, wurden viele Detailverbesserungen und Heuristiken [57, 47, 46, 63, 25, 74] entwickelt, um den Algorithmus in der Praxis zu beschleunigen. Eine der neusten Verbesserungen ist Fengs Algorithmus [24]. Feng unterteilt die Knoten im Graph in drei Kategorien:

Fengs Algorithmus: ☞ *Abschnitt 3.2.2*

- Rote Knoten: Knoten, die eine Verzweigung nicht besuchen darf, da sonst ein Kreis geschlossen wird.
- Gelbe Knoten: Knoten, deren kürzester Weg zum Zielknoten durch einen roten Knoten führt.
- Grüne Knoten: Alle übrigen Knoten.

Kürzeste Wege müssen nicht eindeutig sein. Im Baum kürzester Wege wird durch die SSSP Implementierung beliebig einer ausgewählt.

Feng hat weiter gezeigt, dass kürzeste Verzweigungen zunächst nur aus roten, dann nur aus gelben und zum Schluss nur aus grünen Knoten bestehen. Dank dieser Eigenschaft muss man den grünen Teil der Verzweigung nicht durch einen Kürzeste-Wege-Algorithmus berechnen lassen. Stattdessen kann man diesen aus einem einmalig

vorberechneten Baum kürzester Wege abfragen. Feng ersetzt also alle Kanten zu grünen Knoten durch sog. Expresskanten, die direkt zum Zielknoten führen. Die Kürzeste-Wege-Berechnung lässt sich so auf die Menge der gelben Knoten einschränken. Außerdem kann die Kürzeste-Wege-Berechnung komplett übersprungen werden, wenn der nächste Nachbar des Verzweigungsknotens ein grüner Knoten ist. Zusätzlich berechnet Fengs Algorithmus eine neue Kantengewichtsfunktion, welche die Kürzeste-Wege-Berechnung Richtung Zielknoten begünstigt, während alle Wege zwischen Start und Zielknoten ihre relative Länge zueinander behalten. Fengs Algorithmus liefert in der Praxis bessere Laufzeiten als Yens Algorithmus, führt aber leider nicht zu einer Verbesserung der Worst-Case-Komplexität.

Unser Beitrag

Average-Case Analyse: ☞ Kapitel 4

In Kapitel 4 analysieren wir deshalb die Average-Case-Komplexität, da diese die in der Praxis zu beobachtenden Laufzeiten oftmals besser widerspiegelt als die Worst-Case-Komplexität. Wir zeigen in den Sätzen 4.5 und 4.7 die Average-Case-Komplexität von Yens Algorithmus, sowie in Satz 4.9 die Average-Case-Komplexität von Fengs Algorithmus.

Verbesserte Heuristiken und empirische Vergleiche: ☞ Kapitel 5

Zusätzlich zur Average-Case Analyse schlagen wir auch praktische Verbesserungen an den Heuristiken vor, welche wir in Kapitel 5 ausführlich beschreiben und auf diversen Graphklassen verschiedener Größen empirisch vergleichen. Dabei zeigen wir, dass unsere neuen Heuristiken, vor allem das erweiterte Überspringen von Kürzesten-Wege-Berechnungen, die Laufzeiten deutlich verringern können.

Parallelisierung der k-shortest path Algorithmen: ☞ Kapitel 6

Zuletzt beleuchten wir zwei Strategien zur Parallelisierung von Yens und Fengs Algorithmen auf Multicore Systemen und vergleichen diese in Kapitel 6.

Average-Case Analyse

Die Kantengewichte müssen einer Zufallsverteilung folgen, die zum einen nur nicht-negative Werte annehmen kann und zum anderen eine positive Dichte im Wert Null hat (siehe Annahmen (A1) und (A2) auf Seite 31).

Wir zeigen in Theorem 4.5, dass Yens Algorithmus auf $\mathcal{G}(n,p)$ Zufallsgraphen [30] mit wenigstens logarithmischem Durchschnittsgrad und zufälligen Kantengewichten mit hoher Wahrscheinlichkeit in $\mathcal{O}(k \cdot \log(n) \cdot \mathrm{spc}(n,m))$ Zeit läuft. Der Beweis nutzt aus, dass kürzeste Wege in Zufallsgraphen mit hoher Wahrscheinlichkeit zwei Eigenschaften haben. Einerseits ist ihre Länge nach oben beschränkt, andererseits haben kurze Wege nur wenige Kanten. Wir zeigen dann, dass es wenigstens $\Omega(n)$ Wege von s nach t gibt, deren Länge in der gleichen Größenordnung wie die des kürzesten Wegs liegen. Auch wenn die im Beweis konstruierten Wege nicht unbedingt die k kürzesten sind, beschränken sie doch deren Länge. Die k kürzesten Wege sind also kurz genug, dass sie mit hoher Wahrscheinlichkeit maximal logarithmisch viele Kanten haben. Da Yens Algorithmus für jeden Knoten einen neuen Kandidaten berechnen muss, folgt daraus eine Average-Case-Komplexität von $\mathcal{O}(k \cdot \log(n) \cdot \mathrm{spc}(n,m))$.

Auf Zufallsgraphen mit einem kleineren Durchschnittsgrad $\alpha = o(\log n)$ und uniform verteilten Kantengewichten über $[0;1]$ zeigen wir in Theorem 4.7 eine etwas schwächere Schranke für die Average-Case-Komplexität von Yens Algorithmus. Da der Graph insgesamt deutlich weniger Kanten hat, haben die Pfade mit hoher Wahrscheinlichkeit einen Faktor $\mathcal{O}\left(\frac{\log n}{\alpha}\right)$ mehr Kanten. Dies schlägt sich direkt in der Average-Case-Komplexität von $\mathcal{O}\left(k \cdot \frac{\log^2 n}{\alpha} \cdot \operatorname{spc}(n, m)\right)$ nieder. Der Beweis ist analog zum Beweis für Theorem 4.5.

Die Average-Case-Komplexität für Yens Algorithmus gilt auch für Fengs Algorithmus, da Fengs Algorithmus im Wesentlichen eine Heuristik für Yens Algorithmus ist. Im Falle von ungewichteten Graphen zeigen wir jedoch in Theorem 4.9 sogar eine verbesserte Average-Case-Komplexität von $\mathcal{O}(k \cdot \operatorname{spc}(n, m))$ solange k eine Konstante ist. Dafür argumentieren wir, dass die Größe des Teilgraphen, auf dem die kürzeste Verzweigung berechnet wird, exponentiell wächst und in Folge die Gesamtkomplexität zur Berechnung all dieser kürzesten Wege nur linear in der Größe der Gesamteingabe ist. Da wir in diesem Fall ungewichtete Graphen mit einem Durchschnittsgrad von $\Omega(\log n)$ voraussetzen, analysieren wir die Struktur des Baums der Breitensuche. Wir beweisen, dass alle Teilbäume, die an Knoten einer bestimmten Ebene hängen, mit hoher Wahrscheinlichkeit etwa gleich viele Knoten beinhalten bis auf einen Faktor der vom Durchschnittsgrad abhängt.

Auf ungewichteten Graphen berechnet die Breitensuche die kürzesten Wege von einem Startknoten zu allen anderen Knoten.

Analyse des Baums der Breitensuche:
☞ *Lemma 4.12*

Wir konnten experimentell demonstrieren, dass die Teilgraphen aus gelben Knoten auch im Falle von gewichteten Graphen exponentiell wachsen. Die Ergebnisse legen also nahe, dass Theorem 4.9 auch für gewichtete Graphen gilt. Der mathematische Beweis ist jedoch noch offen.

Unsere Verbesserte Heuristiken

Die von Feng vorgestellten Optimierungen basieren auf der von ihm beschriebenen Knotenfärbung. Die Berechnung der Knotenfärbung benötigt $\mathcal{O}(n+m)$ Zeit und $\mathcal{O}(n)$ Speicher. Wir zeigen in Kapitel 5, dass die Optimierungen auch angewandt werden können, ohne die Knotenfärbung explizit zu berechnen.

- Verwendet man einen Kürzeste-Wege-Algorithmus, der die Berechnung stoppt, sobald der kürzeste Weg zum Zielknoten gefunden ist, z. B. Dijkstras Algorithmus, in Kombination mit der Kantengewichtsfunktion, die Feng beschrieben hat, besucht der Kürzeste-Wege-Algorithmus eine ähnliche Anzahl Knoten, als wäre er auf die gelben Knoten eingeschränkt. Insbesondere zeigen wir, dass im Median nur wenige hundert Knoten erkundet werden.

- Feng zeigt, wie man Kürzeste-Wege-Berechnungen überspringen kann, wenn der nächste Nachbar des Verzweigungsknotens grün ist. Wenn die Knotenfärbung vorab nicht berechnet wurde, kann dennoch von einem einmalig vorberechneten Baum kürzester Wege den kürzesten Weg zum Zielknoten betrachtet werden. Falls dieser Weg keinen Kreis schließt, wäre der Knoten grün gefärbt worden und die Berechnung kann übersprungen werden.

  Der nächste Nachbar muss nicht eindeutig sein. Falls es mehrere Nachbarn mit gleicher Distanz gibt, betrachtet Fengs Algorithmus einen beliebigen, abhängig von der Implementierung.

  Ob die Verzweigung kreisfrei ist, kann in Zeit $\mathcal{O}(x \log x)$ überprüft werden, wobei x die Anzahl der Knoten ist, die die Verzweigung besucht. Dieser Zeitaufwand steht in Konkurrenz zu der Zeit, die für die Knotenfärbung benötigt wird. Insbesondere wenn die Verzweigungen nur wenige Knoten umfassen, kann es schneller sein, auf Kreisfreiheit zu prüfen, als die Knotenfärbung zu berechnen.

Die Heuristiken, mit denen Kürzeste-Wege-Berechnungen übersprungen werden, setzen immer einen vorberechneten Baum aller kürzesten Wege zum Zielknoten voraus. Eine kürzeste Verzweigung lässt sich so in konstanter Zeit abfragen. Auch hier gilt, dass die kürzesten Wege nicht eindeutig sein müssen und im Falle mehrere kürzeste Wege zwischen zwei Knoten ein beliebiger durch die Implementierung ausgewählt wird.

Wir erweitern außerdem die Heuristik, Kürzeste-Wege-Berechnungen zu überspringen, auf zwei Arten:

- Sowohl Yens als auch Fengs Algorithmus pflegen eine Liste mit Kandidaten für die k kürzesten Wege. Sobald die Liste k Kandidaten enthält, kann die Länge des längste Kandidatenpfads dazu verwendet werden, Kürzeste-Wege-Berechnungen zu überspringen. Wenn die kürzeste Verzweigung zwar nicht kreisfrei, aber dafür länger als der längste Kandidat in der Liste ist, ist auch die kürzeste kreisfreie Verzweigung bereits zu lang, um als Kandidat in Frage zu kommen.

- Das Überspringen Kürzester-Wege-Berechnungen lässt sich in beiden Fällen, kreisfreie Verzweigungen und zu lange Verzweigungen, von der kürzesten Verzweigung auf die zweit kürzeste Verzweigung verallgemeinern. Dazu muss zunächst die kürzeste Verzweigung berechnet werden. Für jeden Knoten dieser Verzweigung muss dann eine weitere berechnet werden.

  Alle diese Verzweigungen können wieder von einem Baum kürzester Wege bezogen werden. Die Anzahl der Verzweigungen hängt jedoch von der Anzahl der Knoten auf der kürzesten Verzweigung ab.

  Prinzipiell lässt sich das Verfahren auch auf die i-kürzeste Verzweigung verallgemeinern. Allerdings wird der Aufwand für die Berechnung zunehmend größer, während die Erfolgschancen kleiner werden, wie unsere Experimente nahe legen.

Es ergeben sich so insgesamt 14 Varianten von Yens und Fengs Algorithmus mit verschiedenen Kombinationen von Optimierungen, die wir in Kapitel 5 miteinander vergleichen. Dazu haben wir alle Algorithmen in C++ implementiert. Der Code ist unter der Open Source Lizenz GPLv3 auf GitHub[1] verfügbar. Wir vergleichen die Algorithmen auf $\mathcal{G}(n,p)$ Zufallsgraphen mit konstantem und logarithmischem Durchschnittsgrad sowie auf Grid-Graphen jeweils in verschiedenen Größen von einer Millionen bis hin zu 256 Millionen Knoten. Die Kantengewichte sind in allen Fällen unabhängig und uniform über dem Intervall $[0;1]$ verteilt.

Zunächst stellen wir fest, dass auf $\mathcal{G}(n,p)$ Zufallsgraphen zwischen 96 % und 100 % aller Kürzeste-Wege-Berechnungen übersprungen werden können, während es auf Grid-Graphen immer noch mehr als 94 % im Median sind. Etwa 90 % der übersprungenen Berechnungen auf $\mathcal{G}(n,p)$ Graphen lassen sich auf kreisfreie kürzeste Verzweigungen zurückführen. Weitere 5 % bis 7 % können mit Hilfe der Länge der kürzesten Verzweigung übersprungen werden. Auf die zweitkürzeste Verzweigung entfallen nur noch etwa 0,5 %.

Tabelle aller Algorithmen und deren Features: ☞ *Tabelle 5.1*

Vergleicht man die Laufzeiten, ergibt sich folgendes Bild: Auf $\mathcal{G}(n,p)$ Graphen mit konstantem Durchschnittsgrad und uniform verteilten Kantengewichten über $[0;1]$ sind die Algorithmen Yen-gs-l und Yen-gs2-l um etwa einen Faktor fünf schneller als Fengs Algorithmus. Dabei verwendet Yen-gs-l alle Optimierungen von Fengs Algorithmus,

[1] https://doi.org/10.5281/zenodo.7713239

jedoch ohne die Knotenfärbung zu berechnen. Zusätzlich überspringt Yen-gs-l Kürzeste-Wege-Berechnungen, wenn die kürzeste Verzweigung bereits zu lang ist. Yen-gs2-l überspringt zusätzlich Kürzeste-Wege-Berechnungen sollte die zweitkürzeste Verzweigung kreisfrei oder zu lang sein.

Auf $\mathcal{G}(n, p)$ Graphen mit logarithmischem Durchschnittsgrad sind Yen-gs-l und Yen-gs2-l ebenfalls schneller als Fengs Algorithmus jedoch mit kleinerem Abstand.

Das gleiche Verhalten sieht man auch auf Grid-Graphen. Hier sind Yen-gs-l und Yen-gs2-l sogar um einen Faktor 40 schneller als Fengs Algorithmus. Jedoch lässt sich hier auch Fengs Algorithmus selbst um einen Faktor von bis zu 13 beschleunigen, wenn Kürzeste-Wege-Berechnungen auch bei zu langen kürzesten Verzweigungen übersprungen werden.

Ergänzend vergleichen wir die Algorithmen auf dem Graph des sozialen Netzwerks Orkut und dem europäischen Straßennetz miteinander. Da der Orkutgraph im original ungewichtet ist, haben wir zusätzlich fünf verschiedene Gewichtsfunktionen verwendet. Auf allen außer dem ungewichteten Graph zeigt sich, dass Algorithmen, die Kürzeste-Wege-Berechnungen anhand der Verzweigungslängen überspringen, schneller laufen als das jeweilige Pendant ohne diese Heuristik. Anders als auf den synthetischen Graphen zeigt sich aber auf dem Orkutgraph, dass vor allem bei Gewichtsfunktionen, die längere kürzeste Wege erzwingen, Varianten von Fengs Algorithmus schneller laufen als die von Yens Algorithmus. Auf dem europäischen Straßennetz ist Yen-gs2-l zwischen vier und acht mal schneller als Fengs Algorithmus. Ähnlich wie auf dem Orkutnetzwerk, ist auf den synthetischen Graphen Feng-gs2-l bis zu doppelt so schnell wie Yen-gs2-l und somit etwa um einen Faktor 16 schneller als Fengs Algorithmus. Feng-gs2-l verwendet die gleichen Optimierungen wie Yen-gs2-l, berechnet aber zusätzlich die Knotenfärbung, welche sowohl das Überspringen von Kürzeste-Wege-Berechnungen als auch ausgeführte Kürzeste-Wege-Berechnungen beschleunigen kann. Das gleiche Verhalten sehen wir auch, wenn als Kantengewichte statt der Distanz die Fahrzeit verwendet wird. Hier sind die Unterschiede in den Laufzeiten sogar noch etwas größer.

Gewichtsfunktionen, die längere kürzeste Wege erzwingen sind z.B. solche die Kanten zu Knoten mit hohem Grad ein besonders hohes Gewicht zuordnen. Insbesondere sind solche gewichte nicht mehr uniform auf $[0; 1]$ *verteilt.*

Parallelisierung von Yens und Fengs Algorithmus

Abschließend beschreiben wir in Kapitel 6 zwei Möglichkeiten, Yens und Fengs Algorithmus zu parallelisieren.

- Da alle Verzweigungen vom i-ten kürzesten Weg unabhängig voneinander berechnet werden können, ist eine parallele Berechnung dieser möglich. Diese Form der Parallelisierung ist nur erfolgreich, wenn für hinreichend viele Verzweigungen die Kürzeste-Wege-Berechnung nicht übersprungen wird. Wir haben bereits gezeigt, dass je nach Graphklasse mehr als 94 % der Kürzeste-Wege-Berechnungen übersprungen werden können. Da in $\mathcal{G}(n, p)$ Graphen kürzeste Wege mit hoher Wahrscheinlichkeit nur logarithmisch viele Kanten haben, bleiben nur einzelne Berechnungen übrig, weshalb wir hier keine Beschleunigung durch die Parallelisierung beobachten können. In Grid-Graphen haben kürzeste Wege mit hoher Wahrscheinlichkeit $\Theta(\sqrt{n})$ viele Kanten, d. h. es bleiben genug Verzweigungen übrig, für die eine Kürzeste-Wege-Berechnung ausgeführt werden muss, was sich entsprechend in den Laufzeiten widerspiegelt. Fengs Algorithmus lässt sich so mit 16 Threads um einen Faktor von acht beschleunigen. Yen-gs2-l lässt sich mit vier Threads um einen Faktor von zwei beschleunigen. Dies steigt jedoch auch bei 16 Threads nicht über einen Faktor von 3, da deutlich mehr Kürzeste-Wege-Berechnungen übersprungen werden als bei Fengs Algorithmus.

- Eine andere Variante besteht darin, einen parallelen Kürzeste-Wege-Algorithmus wie z. B. Δ-Stepping [51] zu verwenden. Auf allen von uns untersuchten Graphen, wurden jedoch nur wenige hundert Knoten bei der Kürzeste-Wege-Berechnung erkundet, weshalb wir hier keine Beschleunigung durch Parallelisierung demonstrieren konnten.

- Prinzipiell lassen sich die beiden Varianten, parallele Verzweigungen und parallele Kürzeste-Wege-Algorithmen, auch gleichzeitig verwenden. Dies lohnt sich jedoch nur, wenn zum einen jede Variante einzeln eine Beschleunigung bewirkt und zum anderen das eingesetzte System mehr Prozessorkerne hat, als von einer Variante alleine ausgenutzt werden können.

Insgesamt konnten wir demonstrieren, dass das parallele Berechnen von Verzweigungen prinzipiell funktioniert, jedoch sind die Optimierungen aus Kapitel 5 bereits so gut, dass nur wenig parallelisierbare Berechnungen übrig bleiben. Sollten die Heuristiken auf einer speziellen Eingabe weniger gut funktionieren, profitiert man stärker von einer Parallelisierung, wie unsere Experimente zeigen.

Abstract

The single-source shortest-path problem is a fundamental problem in computer science. We consider a generalization of the shortest-path problem, the k-shortest path problem. Let G be a directed edge-weighted graph with n nodes and m edges and s, t be two fixed nodes. The goal is to compute k paths $P_1, \ldots, P_k$ between two fixed nodes s and t in non-decreasing order of their length such that all other paths between s and t are at least as long as the k^{th} path P_k. We focus on the version of the k-shortest path problem where the paths are not allowed to visit nodes multiple times, sometime referred to as k-shortest simple path problem.

The probably best known k-shortest path algorithm is Yen's algorithm [76]. It has a worst-case time complexity[2] of $\mathcal{O}(kn \cdot \operatorname{spc}(n, m))$, where $\operatorname{spc}(n, m)$ is the complexity of the single-source shortest-path algorithm used as a subroutine. In case of Dijkstra's algorithm $\operatorname{spc}(n, m)$ is $\mathcal{O}(m + n \log n)$. One of the more recent improvements of Yen's algorithm is by Feng [24]. Even though Feng's algorithm is much faster in practice, it has the same worst-case complexity as Yen's algorithm.

Yen's algorithm:
☞ *Section 3.2.1*

Feng's algorithm:
☞ *Section 3.2.2*

The main results presented in this thesis are upper bounds on the average-case of Yen's and Feng's algorithm, as well as practical improvements and a parallel implementation of Yen's and Feng's algorithms including these improvements. The implementation is publicly available under GPLv3 open source license[3].

We show in our analysis that Yen's algorithm has an average-case complexity of $\mathcal{O}(k \log(n) \cdot \operatorname{spc}(n, m))$ on $\mathcal{G}(n, p)$ graphs with at least logarithmic average-degree and random edge weights following a distribution with certain properties. On $\mathcal{G}(n, p)$ graphs with constant to logarithmic average-degree and uniform random edge-weights over $[0; 1]$, we show an average-case complexity of $\mathcal{O}\left(k \cdot \frac{\log^2 n}{np} \cdot \operatorname{spc}(n, m)\right)$. Feng's algorithm has an even better average-case complexity of $\mathcal{O}(k \cdot \operatorname{spc}(n, m))$ on unweighted $\mathcal{G}(n, p)$ graphs with logarithmic average-degree and for constant values of k. We further provide evidence that the same holds true for $\mathcal{G}(n, p)$ graphs with uniform random edge-weights over $[0; 1]$.

Average-case results:
☞ *Theorems 4.5, 4.7, and 4.9*

Details about the assumptions on the weight distributions can be found on page 31.

On the practical side, we suggest new heuristics to prune even more single-source shortest-path computations than Feng's algorithm and evaluate all presented algorithms on $\mathcal{G}(n, p)$ and Grid graphs with up to 256 million nodes. We demonstrate speedups by a factor of up to 40 compared to Feng's algorithm.

Sequential improvements:
☞ *Chapter 5*

Finally we discuss two ways to parallelize the suggested algorithms and evaluate them on grid graphs showing speedups by a factor of 2 using 4 threads and by a factor of up to 8 using 16 threads, respectively.

Parallelizations of Yen's and Feng's algorithm:
☞ *Chapter 6*

[2]In our context, time complexity refers to the unit-cost RAM model.
[3]https://doi.org/10.5281/zenodo.7713239

Contents

Introduction

Computing shortest paths between two nodes in a graph with n nodes and m edges is one of the most fundamental optimization problems which appears in many applications. In this thesis, we consider a generalization of the shortest path problem, the k-shortest path problem, k-SP for short, where the goal is to compute k paths $P_1, \dots, P_k$ between two fixed nodes s and t in non-decreasing order of their length such that all other paths between s and t are at least as long as the k^{th} path P_k. The k-shortest path problem comes in two variants:

- Allowing loops, meaning that a path can visit a node multiple times.
- Disallowing loops. We call paths without loops *loopless* or *simple*.

We only consider the loopless variant of the k-shortest path problem in this thesis, which is why we do not explicitly mention that the loopless variant is meant in later chapters. We further focus on the k-shortest simple path problem on *directed* graphs with edge-weights.

The length of these paths or the number of short paths can reveal deeper semantic insights into the relationship between nodes and the underlying graph structure. As such, the k-shortest path problem appears as a subproblem in many applications like link prediction and recommendation systems. In settings with ill-defined or hard-to-optimize constraints, one approach can be to compute several paths and choose a valid option among them based on other criteria by another algorithm or even by a human expert.

1.1 Applications

An obvious application for k-shortest path is to find alternative paths in networks. The k-shortest path problem appears in optical mesh network routing to optimize the traffic throughput [38], in telecommunication network routing used in a learning algorithm to improve the quality of service [36], routing in jellyfish topology networks, used in compute clusters, with HPC workloads [78, 6]. It can also be found in concurrent entanglement routing in quantum networks [64], and in evacuation routing to optimize police resource allocation [34]. However, without further restrictions, k-shortest path is usually not suitable to find multiple alternative paths on a road network. The k shortest paths will most likely overlap by a lot and thus will probably not be considered true alternative paths by a user. For this kind of application, one can look, e.g., at the k-shortest path problem with limited overlap [14]. Another option is to compute the

k-shortest path for a bigger k and filter paths using a similarity measure, as it is done in [61] for stochastic routing optimized for autonomous driving.

Yet another application of k-shortest path concerns link predictions [4, 43] in networks with small diameters where most nodes are connected via short paths and so the number of short paths between two nodes seems to be more relevant than the actual length.

It has further use-cases in creating virtual networks while keeping the underlying physical networks robust [69].

In computational linguistics, k-shortest path can be used for summary generation and multi-sentence compression [26, 8]. Here multiple shortest-paths in a word-graph are generated and filtered for certain properties.

In bioinformatics, it can be used to infer regulatory paths in gene networks [65], to identify causal relations between genes. For gene recognition, it can be used in a learning algorithm to predict gene structure [15].

In computer vision, k-shortest path can be used for multi-object tracking, e.g., to track multiple players in sports analysis [44].

There are also applications of k-shortest path algorithms in chip design to optimize wire length and pin-count on electrowetting-on-dielectric chips [37] and for full-chip static electrostatic discharge verification [71], for predicting chemical reactions in solid-state materials [48], in optimal group testing [7], in ranked enumeration of conjunctive database queries [68], and many more.

While some of these applications require the variant of k-shortest path allowing cycles, other applications require the loopless variant. Some machine learning applications may benefit from both the variants as different features.

1.2 Related Work

Average-case analysis: ☞ *Chapter 4*

Practical improvements: ☞ *Chapter 5*

Parallelization: ☞ *Chapter 6*

In this thesis, we focus on the loopless version of the k-shortest path problem introduced by Clarke et al. [17] in 1963. More precisely, we focus on an analysis of the average-case complexity of an exact algorithm by Yen [76], a more recent optimization of it by Feng [24], as well as further practical improvements and parallelizations.

From an asymptotic worst-case complexity perspective the best algorithm for the directed edge-weighted case is by Gotthilf and Lewenstein [31]. It achieves a worst-case complexity of $\mathcal{O}(kn(m + n \log\log n))$ by solving the replacement path problem[1] $\mathcal{O}(k)$ times on a graph with n nodes and m edges. Yet, as noted by Feng [24], the algorithm does not seem to be practical. For directed unweighted graphs, Roditty and Zwick [59] proved an upper bound of $\tilde{\mathcal{O}}(km\sqrt{n})$ using a Monte Carlo algorithm, a randomized algorithm allowing for a small probability to have an incorrect output.

$\tilde{\mathcal{O}}$ is similar to $\mathcal{O}$ notation but also ignores logarithmic factors.

Williams and Williams [75] showed that of a list of graph problems either all have algorithms running in $\mathcal{O}\left(n^{3-\varepsilon}\right)$ for a $\varepsilon > 0$, or none of them has. This list of graph problems includes the all-pairs shortest paths problem as well as the replacement

[1]Given a shortest path from s to t the replacement path problem asks for each edge e on that path for a shortest s–t-path that avoids the edge e.

path problem and the second-shortest path problem on edge-weighted directed graphs. Agarwal and Ramachandran [2] showed a similar result for the sparse cases of a list of graph problems including the replacement path problem and the second-shortest path problem on directed edge-weighted graphs suggesting that they cannot be solved in polynomial less than $\tilde{\mathcal{O}}(nm)$ time. Lawler [42] proved that the k-shortest simple path problem can be solved in $\mathcal{O}\big(kn^3\big)$ on directed edge-weighted graphs even if there are negative edge weights.

Yen's algorithm runs in $\mathcal{O}(kn(m + n \log n))$ time in the worst-case using Dijkstra's algorithm [19] with Fibonacci heaps [27] as a subroutine. Although we cannot expect asymptotic worst-case improvements for sparse directed edge-weighted graphs due to the results just mentioned, many practical improvements have been found in the last years. For example, Perko [57] demonstrated how to efficiently implement Yen's algorithm for large values of k in terms of memory usage. Martins et al. [47] generalized Yen's algorithm for the constrained k-shortest simple path problem and later Martins et al. [46] presented an improved implementation of Yen's algorithm. Sedeño-Noda [63], Feng [25], and Wen et al. [74] also worked on improving runtimes and memory consumption of Yen's algorithm for certain graph classes.

Hershberger et al. [35] presented an algorithm that uses a fast branching technique reporting to be up to eight times faster than Yen's algorithm. The fast branching can fail to find the correct branch in which case a slower exhaustive subroutine had to be used. Thus, the algorithm does not come with asymptotic improvements. Vanhove and Fack [70] suggested a k-shortest path algorithm that uses a precomputed shortest-path tree to avoid shortest path computations but in turn only finds an approximation of the k^{th}-shortest path. Feng [24] proposed a node-classification heuristic to improve the average-case runtimes of Yen's algorithm by pruning the size of the graph for shortest path computations and allowing to skip some shortest path computations completely. Kurz and Mutzel [41] presented an algorithm that is not based on Yen's algorithm and showed in experiments that their algorithm is faster than Feng's algorithm by about a factor of 8 to 25 depending on graph class and size. Until then Feng claimed to have the fastest k-shortest simple path algorithm in practice. Chen et al. [12] recently suggested a new k-shortest path algorithm that updates and maintains a shortest path tree rooted at the target node claiming to be faster than the current state of the art algorithm. They report to be about 500 times faster than Yen's algorithm but there is no comparison to Feng's algorithm or the algorithm by Kurz and Mutzel. In Chapter 5, we describe some variants of Yen's and Feng's algorithm with some practical improvements. In our experiments on sparse graphs, we observe speedups by a factor of 40 to 500 compared to Yen's algorithm and by a factor of 4 to 40 compared to Feng's algorithm, respectively. Unfortunately, we did not manage to include a direct comparison to the algorithms of Kurz and Mutzel as well as Chen et al. A fair comparison between all the recently suggested k-shortest path algorithms is a huge task on its own as we describe in Chapter 7.

Feng's algorithm:
☞ Section 3.2.2

Kurz and Mutzel used random graphs from the ninth DIMACS implementation challenge including $\mathcal{G}(n, m)$ graphs with up to 10000 nodes and grid graphs with up to 160000 nodes.

There are only a few papers on parallelizing k-shortest path algorithms. One recent attempt of parallelizing Yen's algorithm is by Singh and Singh [66] achieving a speedup by a factor of 6 compared to a sequential implementation. The parallelization utilizes GPUs by Nvidia using Compute Unified Device Architecture (CUDA) mainly based on a parallel version of Dijkstra's algorithm. Another work by Yu et al. [77] focuses on a distributed algorithm for the k-shortest path problem on graphs with dynamically changing edge weights. We have parallelized our proposed algorithms to work on multi-core shared memory systems. In our experiments, we see a speedup of a factor of two on four threads and up to a factor of eight using 16 threads on grid graphs depending on the algorithm variant.

Parallelization: ☞ *Chapter 6*

Using a parallel SSSP algorithm is a simple way to make use of parallelism in most k-shortest path algorithms. We focus on the Δ-stepping algorithm by Meyer and Sanders [49]. Implementations of this algorithm have been found to be among the fastest on massively parallel computing environments [45], GPUs [72], multicores [53], and distributed architectures [32, 20, 56]. Δ-stepping manages edges in a list of buckets of a fixed width controlled by the parameter Δ. There is also a rather new algorithm called radius-stepping by Blelloch et al. [11] which uses an adaptive bucket width. Since radius-stepping also requires a preprocessing step that introduces new edges to the graph, we do not consider it for our implementation.

Δ-Stepping: ☞ *Section 2.3.2*

For *undirected* weighted graphs, Katoh et al. [40] presented a k-shortest simple path algorithm with an asymptotic worst-case running time of $\mathcal{O}(k(m + n \log n))$ which is by a factor of n faster than the best known algorithm for directed graphs. Hadjiconstantinou and Christofides [33] show an efficient implementation of the KIM algorithm on dense graphs.

KIM algorithm: ☞ *Section 3.2.4*

Even though it is out of scope for this thesis, we want to mention that there is also work on approximation algorithms for the k-shortest simple path problem [9, 28], distance oracles for the replacement path problem [18, 10], all-pair k-shortest simple path algorithms [1], the k-shortest path algorithms that allow for loops [21, 5, 4] including parallel algorithms [60], and top-k temporal closeness in temporal networks [55].

Distance oracles are precomputed data structures to answer distance queries fast.

Temporal closeness is a measurement for the relevance of a node in a temporal network.

Finally, we want to mention the k-shortest hyperpath problem [54, 23] which can be solved using an adapted version of Yen's algorithm working on hypergraphs where edges connect multiple source nodes to a single target node.

1.3 Outline

The thesis is organized as follows. First we briefly define all necessary preliminaries in Chapter 2. Chapter 3 then contains all the basics about the k-shortest path problem as well as the details on Yen's algorithm, Feng's algorithm and a short overview of the KIM algorithm by Katoh et al. [40]. Chapter 4 follows with a detailed average-case time complexity analysis of Yen's algorithm which also holds for Feng's algorithm showing an average-case time complexity of $\mathcal{O}(k \log(n) \cdot \operatorname{spc}(n, m))$ on directed edge-weighted graphs with n nodes and $m = \Omega(n \log n)$ edges. For sparse directed edge-weighted $\mathcal{G}(n, p)$ graphs with $m = \Theta(n)$, we show an average-case time complexity of $\mathcal{O}\left(k \cdot \frac{\log^2 n}{np} \cdot \operatorname{spc}(n, m)\right)$. In both cases $\operatorname{spc}(n, m)$ denotes the worst-case complexity of the SSSP algorithm in use. We further prove an even better average-case complexity

Yen's average-case complexity: ☞ *Theorem 4.5*

Feng's average-case complexity: ☞ *Theorem 4.9* of $\mathcal{O}(km)$ for Feng's algorithm for a constant k on directed unweighted graphs. We also provide evidence that the same result should hold on edge-weighted graphs as well. Chapter 5 is about implementation details and practical improvements of Yen's and Feng's algorithm which result in several new algorithm variants which are evaluated in that chapter. The thesis then closes with Chapter 6 containing details on how we parallelized the algorithms described in the previous chapters including some implementation details and an empirical performance comparison.

The code of all variants of Yen's and Feng's algorithm listed in Table 5.1 is publicly available on GitHub under an open source license at

https://doi.org/10.5281/zenodo.7713239

It is written in C++20 in a modular way such that SSSP algorithms and some data structures can easily be replaced or extended in future experiments.

Preliminaries 2

2.1 Graphs

Let a graph $G = (V, E)$ be a pair of two sets V and $E \subset V \times V$. We interpret the elements of V as *nodes* and the elements of E as *edges*. There are two major types of graphs:

- Directed graphs. We denote edges $(u, v) \in E$ as a pair.
- Undirected graphs. For each edge $(u, v) \in E$ the edge (v, u) is also in E. We denote undirected edges as a set $\{u, v\}$ to indicate that edges in both directions are present in the graph.

We mostly consider *weighted graphs* which are graphs with an additional edge-weight function $\mathrm{d}(e)$ for each edge $e \in E$ and we define $\mathrm{d}(e) := \infty$ for all $e \notin E$. On undirected graphs $\mathrm{d}((u, v)) = \mathrm{d}((v, u))$ holds for all edges $\{u, v\} \in E$. This does not have to be true on directed graphs even if both directions of an edge are present.

A table of notations can be found in Appendix A.1.1 on page 83.

A path in a graph is a tuple of nodes $P = (v_1, \ldots, v_r)$, such that the edges $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, r-1$ exist in the graph.

We extend the notation of edge-weights to the length of a path. In order to not confuse the length of a path and the number of its hops, we will call the sum of a path's edge-weights the weight-length denoted by $\mathrm{d}(\cdot)$

$$\mathrm{d}(P) = \mathrm{d}(v_1, \ldots, v_r) = \sum_{i=1}^{r-1} \mathrm{d}(v_i, v_{i+1})$$

with $|P| := r$ and the number of its edges the hop-length denoted by $\overline{\mathrm{d}}(\cdot)$.

Especially when talking about the number of edges of a path, the edges are sometimes called hops.

If the actual nodes on the shortest path are not relevant, we denote the shortest path in terms of weight by $u \dashrightarrow v$ and by $u \twoheadrightarrow v$ for a shortest path in terms of hops. In this context we denote a single edge as $u \to v$. Even though the shortest path has not to be unambiguous, the shortest distance is well-defined in both cases. So we write $\mathrm{d}\big(u \dashrightarrow v\big)$ for the weight-length of the shortest path in terms of weight and $\overline{\mathrm{d}}\big(u \twoheadrightarrow v\big)$ for the hop-length of the shortest path in terms of hops from u to v. If there exists no path from u to v in G, we define $\mathrm{d}\big(u \dashrightarrow v\big) := \infty$ and $\overline{\mathrm{d}}\big(u \twoheadrightarrow v\big) := \infty$.

Following this notation, we denote the diameter of a graph with

$$\mathrm{Diam}(G) := \max_{s,t \in V}\big\{\mathrm{d}\big(s \dashrightarrow t\big)\big\} \qquad \text{and} \qquad \overline{\mathrm{Diam}}(G) := \max_{s,t \in V}\big\{\overline{\mathrm{d}}\big(s \twoheadrightarrow t\big)\big\}$$

in terms of edge-weights and hops, respectively.

Given two paths $P_1 = (u_1, \dots, u_r)$ and $P_2 = (v_1, \dots, v_{r'})$, we denote the joined path P by $P_1 \circ P_2$ defined as follows.

$$P_1 \circ P_2 := \begin{cases} (u_1, \dots, u_r, v_2, \dots, v_{r'}) & \text{if } u_r = v_1 \\ (u_1, \dots, u_r, v_1, \dots, v_{r'}) & \text{if } u_r \neq v_1 \text{ and } (u_r, v_1) \in E \end{cases}$$

2.2 Breadth First Search and Depth First Search

Breadth First Search, BFS for short, and Depth First Search, DFS for short, are two of the most basic graph traversal algorithms. Being at a node u with neighbors $v_1, \dots, v_k$, BFS first visits all neighbors of $v_1, \dots, v_k$ before visiting their neighbors starting with the neighbors of v_1 and ending with the neighbors of v_k. DFS visits first v_1 and its neighbors before visiting the other neighbors of u. The process recurses without revisiting nodes. Algorithm 1 shows how to implement both BFS and DFS in a sequential way. The only difference is that BFS uses a queue to organize the nodes to visit next while DFS uses a stack. Note that the order of the neighbors is not defined by the algorithm and only determined by their order in the adjacency list.

Both algorithms can be used to find all nodes in a graph that can be reached from a given source node. If the particular order does not matter, DFS can be faster due to a better cache locality of the stack, which is why we use DFS instead of BFS in Algorithm 4.

On an unweighted graph, BFS can be used as SSSP algorithm, since it visits the nodes in order of non-decreasing hop-distance. Since both BFS and DFS have a worst-case running time of $\mathcal{O}(n + m)$, BFS is preferred over a general SSSP algorithm like, e.g., Dijkstra's algorithm.

Algorithm 1: Breadth First Search / Depth First Search

```
Input  : Graph $G = (V, E)$, edges are stored as adjacency lists, the start node $u \in V$.
visited[$v$] ← FALSE  $\forall\ v \in V$          // initialize array of visited nodes
L ← $\{u\}$                                       // L is a queue for BFS and a stack for DFS
while L.isNotEmpty() do
    $v$ ← L.pop()                                 // store the next element in $v$ and remove it from L
    if not visited[$v$] then                      // If $v$ was not visited yet ...
        visited[$v$] ← TRUE                       // ... mark it as visited and ...
        for $w \in E[v]$ do                       // ... store all its neighbors to visit in the future.
            L.push($w$)
```

2.3 Single-Source Shortest-Path Algorithms

k-shortest path algorithms:
☞ Chapter 3

The k-shortest path algorithms throughout this thesis have to compute many shortest deviation paths in order to find the relevant k paths. They all use a single-source shortest-path algorithm, SSSP algorithm for short, as a subroutine to fulfill this task. However, the k-shortest path algorithms do not require it to be a specific SSSP algorithm. So in order to keep it simple one can always think of the well known Dijkstra's algorithm.

Even though one could use any SSSP algorithm, some optimizations require the following features:

k-shortest path optimizations: ☞ Sections 5.1 and 5.2

- Stop the computation as soon as the distance and path to a target node is settled. This is sometimes referred to as single-pair shortest-path problem, SPSP for short.
- Stop the computation as soon as it is settled that the target node is further away than a certain distance.

Note that the actual used SSSP algorithm is not the focus of the optimizations we consider in this thesis. There are other options like, e.g., Radius Stepping [11]. If needed, one could always interchange the SSSP algorithm by a faster algorithm or an algorithm specialized for a certain graph type.

2.3.1 Dijkstra's algorithm

Dijkstra's algorithm [19] is a label-setting algorithm computing the shortest path in a directed, edge-weighted graph. Starting with a source node s, Dijkstra's algorithm maintains a priority queue of nodes where the node with the shortest tentative distance to the source node s has the highest priority. In each step the node with the highest priority gets settled and its out-edges get relaxed, updating the distances to the connected nodes. See Algorithm 2 for details.

Algorithm 2: Dijkstra's SSSP algorithm

```
Input  : Graph $G = (V, E)$, distances $\mathrm{d}(u, v)\ \forall\ (u, v) \in E$, the source node $s \in V$.
Output : Distances tentDist$[v]$ and parent nodes tentDist$[v]$ for each node $v \in V$.
1  tentDist$[v] \leftarrow \infty\ \forall\ v \in V$                  // the array of tentative distances
2  tentDist$[s] \leftarrow 0$                                      // set the source nodes distance to zero
3  parent$[v] \leftarrow$ NONE $\forall\ v \in V$                    // the array of parent nodes
4  parent$[s] \leftarrow s$                                        // mark the source node as root
5  P $\leftarrow \{s\}$                                            // the priority queue of nodes to be relaxed
6  while P.isNotEmpty() do
7      $v \leftarrow$ P.pop()                                      // get the node with the smallest tentative distance
8      forall $(v, w) \in E$ do                                    // Iterate over all out-edges of the settled node $v$
9          if tentDist$[w] >$ tentDist$[v] + \mathrm{d}(v, w)$ then   // if the edge improves the distance...
10             tentDist$[w] \leftarrow$ tentDist$[v] + \mathrm{d}(v, w)$    // ...relax the edge...
11             parent$[w] \leftarrow v$     // ...remember which node realizes the tentative distance...
12             if $w \in P$ then
13                 P.decreaseKey($w$)    // ...and update the position of node $w$ in the PQ...
14             else
15                 P.push($w$)           // ...or insert the node $w$ into the PQ for the first time
```

Algorithm 2 computes both distances and an array of parent nodes that realize these distances. Tracking the array of parent nodes back to the source node results in a shortest path. If one is only interested in distances and not the actual paths, the lines 3, 4, and 11 can be deleted.

If the priority queue is implemented using Fibonacci heaps [27], Dijkstra's algorithm has a worst-case time complexity of $\mathcal{O}(m + n \log n)$.

2.3.2 Δ-Stepping

Since we later want to look into parallelizing k-shortest path algorithms, we use Δ-stepping by Meyer [50] as the SSSP algorithm in all our implementations and experiments. It also has a linear average-case time complexity which fits in the picture with our average-case analysis of k-shortest path algorithms.

Average-case analysis of k-shortest path algorithms: ☞ *Chapter 4*

Given a graph $G = (V, E)$ with edge weights $\mathrm{d}(e) > 0$ for each edge $e \in E$, a source node $s \in V$, and a parameter $\Delta > 0$, called the bucket width, Δ-stepping works roughly speaking like Dijkstra's algorithm [19], but eligible nodes are organized in a bucket list, where the i^{th} bucket contains all nodes v with a tentative distance $\mathrm{tent}(v)$ with $i \cdot \Delta \leq \mathrm{tent}(v) < (i + 1) \cdot \Delta$ in no particular order. In addition, edges e are distinguished into light ($\mathrm{d}(e) < \Delta$) and heavy ($\mathrm{d}(e) \geq \Delta$) edges. When it comes to processing the i^{th} bucket, this is done in phases.

1. For each light out-edge e of all nodes in the i^{th} bucket a relax-request is stored in a list and all nodes from the i^{th} bucket are removed from the current bucket and stored in a buffer.

2. After all relax-requests are generated, the requests are executed in parallel, updating the tentative distances and adding nodes where the tentative distance changed to their respective bucket. This could reintroduce nodes to the current bucket, so the first two phases need to be repeated as long as the current bucket is not empty.

3. When the current bucket is empty, for each node from the buffer all heavy out-edges e are relaxed once in parallel. In this phase nodes cannot be reintroduced into the current bucket, since the new tentative distances are at most

$$\mathrm{dist}(v) + \mathrm{d}(e) \geq i \cdot \Delta + \Delta = (i + 1)\Delta$$

   After all heavy edges are relaxed, the buffer is cleared and the next bucket can be processed.

At the beginning of the third phase the distances of all nodes in the buffer are already settled. The algorithm stops if all buckets are empty.

Δ-stepping runs in $\mathcal{O}(|V| + |E|)$ average time, when edge weights are chosen uniformly at random over $[0; 1]$ and Δ is chosen to be $\Delta = 1/\max\{\deg(v) : v \in V\}$ as shown in [50]. In order to achieve the linear average-case running time, Δ-stepping uses an adaptive bucket-splitting that is used to handle the case when to many nodes are in a single bucket. In our implementation we only use the simple version without the adaptive bucket-splitting described in [50].

2.4 Probabilistic Preliminaries

Throughout this thesis we will give some bounds on the outcome of random events. For a random event A, $\mathbb{P}[A]$ denotes the probability for A to happen. $\mathbb{E}[X]$ denotes the expected value of a random variable X. We say that a bound that depends on a parameter n holds *with high probability*, short *whp.*, if it holds with probability at least $1 - o(n^{-\varepsilon})$ for a constant $\varepsilon > 0$. If it holds only with probability $1 - o(1)$, we say it holds *almost surely*.

2.4.1 Chernoff Bounds

We will use the well-known Chernoff bounds [13] in the following form.

Lemma 2.1. Let $X = \sum_{i=1}^{n} X_i$ be the sum of n independent, not necessary identically distributed, random variables over $[0; 1]$. Then

$$\mathbb{P}[X < \mathbb{E}[X] - \varepsilon\mathbb{E}[X]] = \mathbb{P}[X < (1-\varepsilon)\mathbb{E}[X]] \leq \exp\left(-\frac{\varepsilon^2\mathbb{E}[X]}{2}\right) \tag{2.1}$$

holds for any $0 < \varepsilon < 1$ and equivalent to that for any $a < \mathbb{E}[X]$

$$\mathbb{P}[X < \mathbb{E}[X] - a] = \mathbb{P}\left[X < \left(1 - \frac{a}{\mathbb{E}[X]}\right)\mathbb{E}[X]\right] \leq \exp\left(-\frac{a^2}{2\mathbb{E}[X]}\right) \tag{2.2}$$

Similarly, it holds for any $\varepsilon > 0$:

$$\mathbb{P}[X > \mathbb{E}[X] + \varepsilon\mathbb{E}[X]] = \mathbb{P}[X > (1+\varepsilon)\mathbb{E}[X]] \leq \exp\left(-\frac{\min\left\{\varepsilon, \varepsilon^2\right\}\mathbb{E}[X]}{3}\right) \tag{2.3}$$

◀

2.4.2 Stochastic Dominance

Given two random variables X and Y distributed according to distribution functions F_X and F_Y, respectively. We say that Y *stochastically dominates* X, denoted as $X \leq_{\text{st}} Y$, if

$$\mathbb{P}[X > t] \leq \mathbb{P}[Y > t] \tag{2.4}$$

holds for all $t \in \mathbb{R}$. This can also be interpreted as

$$\mathbb{P}[Y \leq t] \leq \mathbb{P}[X \leq t]\,. \tag{2.5}$$

$X \leq_{\text{st}} Y$ also implies $\mathbb{E}[X] \leq \mathbb{E}[Y]$. One example with $X \leq_{\text{st}} Y$ is binomial distributed random variables $X \sim \textsc{Bin}[t_1, p]$ and $Y \sim \textsc{Bin}[t_2, p]$ with $t_1 < t_2$, where $\textsc{Bin}[n, p]$ is the binomial distribution with n independent experiments and success probability p.

3 The k-Shortest Path Problem

This chapter is an introduction to the k-shortest path problem itself as well as Yen's and Feng's algorithm, which we analyze further in Chapter 4. Even though we include the pseudocode here in order to explain the algorithms as clearly as possible, the details on the implementations as well as new algorithmic improvements can be found in Chapter 5 where we have a closer look at sequential running times. Details on the parallelization can be found in Chapter 6.

3.1 General Notation

The k-shortest path problem, short k-SP problem, is a generalization of the well known shortest path problem.

Given a directed graph $G = (V, E)$ with non-negative edge weights $\mathrm{d}(e)$ for each edge $e \in E$, we are interested in finding the k shortest simple paths, in terms of weight, between two fixed nodes s and t such that all other paths from s to t are at least as long as the k shortest. We call a path $P = (v_1, \dots, v_r)$ *simple* or *loopless* if $v_i \neq v_j$ holds for all $1 \leq i < j \leq r$.

We are only interested in the loopless case. In contrast to the loopy variant, algorithms are often more complex for the loopless variant.

Let $P_1, \dots, P_k$ be the k shortest paths with $P_i \neq P_j$ for all $1 \leq i < j \leq k$ in non-decreasing order by their weight-length. We denote the i^{th}-shortest path as

$$P_i = \left(v_1^{(i)}, \dots, v_{|P_i|}^{(i)}\right)$$

with $v_1^{(i)} = s$ and $v_{|P_i|}^{(i)} = t$ for all $1 \leq i \leq k$. We also define

A table of notations can be found in Appendix A.1.1 on page 83.

$$\mathrm{R}_i(j) \coloneqq \left(v_1^{(i)}, \dots, v_j^{(i)}\right), \qquad \mathrm{S}_i(j) \coloneqq \left(v_j^{(i)}, \dots, v_{|P_i|}^{(i)}\right)$$

to be the prefix path to the j^{th} node and the suffix path from the j^{th} node of P_i, respectively. Keep in mind that, even though all the k shortest paths are pairwise different, they are not necessarily edge disjoint.

The k shortest paths can be viewed as a hierarchy of deviating paths. We say path P_j, with $j \geq 2$, deviates at node $\mathrm{dev}(P_j) = v_{\mathrm{di}(P_j)}^{(j)} \in V$, called the *deviation node* from path P_i at *deviation node index* $\mathrm{di}(P_j)$, with $i < j$ if P_i is the longest path in terms of edge-weights having the same prefix path $\mathrm{R}_i(\mathrm{di}(P_j))$ from s to $\mathrm{dev}(P_j)$ as P_j and P_i itself deviated not after $\mathrm{dev}(P_j)$ from the path it deviated from. We call $P_i = \mathrm{par}(P_j)$ the parent path of P_j (see Figure 3.1 for an illustration). For simplicity Figure 3.1 does not show that paths can join before the target node t. This is possible and not even uncommon depending on the structural properties of the graph.

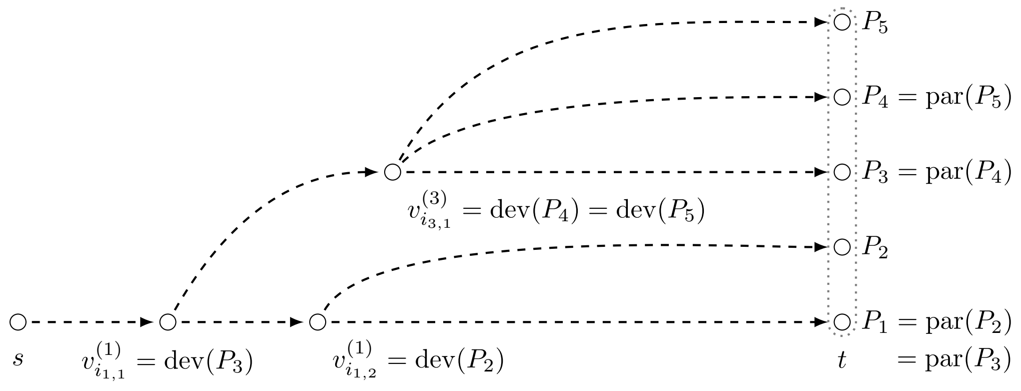

P_5
$P_4 = \mathrm{par}(P_5)$
$P_3 = \mathrm{par}(P_4)$
$v^{(3)}_{i_{3,1}} = \mathrm{dev}(P_4) = \mathrm{dev}(P_5)$
P_2
$P_1 = \mathrm{par}(P_2)$
$= \mathrm{par}(P_3)$
s
$v^{(1)}_{i_{1,1}} = \mathrm{dev}(P_3)$
$v^{(1)}_{i_{1,2}} = \mathrm{dev}(P_2)$
t

Figure 3.1: *An illustration of the relation between k shortest paths. Path P_3 deviates from P_1 and not P_2 because P_2 deviates after $\mathrm{dev}(P_3)$. Paths P_4 and P_5 deviate from the same node but we say that P_4 is the parent of P_5 since P_4 is longer, in terms of weight, than P_3.*

3.2 Deviation Based k-Shortest Path Algorithms for Directed Graphs

In this thesis we focus on the deviation based k-shortest path algorithms by Yen and Feng for directed graphs. From a high level perspective these algorithms do the following on a given directed graph $G = (V, E)$, with source node $s \in V$ and target node $t \in V$:

1. Create an empty list C of candidate paths.
2. Compute the shortest path in terms of edge-weights from s to t in G and add it to the candidate list C.
3. For i in $1, \dots, n$:
   (a) Remove the shortest path in the candidate list C and call it P_i.
   (b) For each node $v^{(i)}_j$, $j \neq |P_i|$, on path P_i compute the shortest deviation path $P'_{i,j}$ from $v^{(i)}_j$ to t such that $P_{i,j} = \mathrm{R}_i(j) \circ P'_{i,j}$ is neither in C nor one of the first i shortest paths $P_1, \dots, P_i$.
   (c) Add $P_{i,j}$ as a new candidate to C.

Section 3.2.1 about Yen's algorithm goes into detail on how to compute deviations without finding paths multiple times. Section 3.2.2 about Feng's algorithm then adds heuristics to reduce the overall computational work.

3.2.1 Yen's k-Shortest Path Algorithm

Starting with the shortest path, Yen's algorithm [76] computes a set of candidates for the $(i+1)^{\text{th}}$-shortest path by computing deviations from the i^{th}-shortest path by temporarily removing edges from the graph that would lead to already found paths.

3.2.1.1 Detailed Description

Be given a graph $G = (V, E)$ with edge weights $\mathrm{d}(e)$ for each edge $e \in E$, two nodes $s, t \in V$, and $k \in \mathbb{N}_{>0}$.

Algorithm 3: Yen's Algorithm

```
Input  : Directed weighted graph $G$, nodes $s$ and $t$, the number $k$ of paths to compute
Output : The $k$ shortest $s$–$t$-paths $P_1, \dots, P_k$ sorted by increasing length
1  $P \leftarrow$ SSSP$(G, s, t)$                    // compute the shortest path from $s$ to $t$ in $G$
2  $C \leftarrow \{(P, 1, \{\})\}$                    // priority queue of candidates
3  for $i \leftarrow 1, \dots, k$ do
4      $(P_i, \mathrm{di}(P_i), D_i) \leftarrow C$.POPMIN()    // the shortest path among the candidates
5      if $i = k$ then return
6      for $j \leftarrow \mathrm{di}(P_i), \dots, |P_i| - 1$ do    // compute all deviations from $i^{\text{th}}$-shortest path
           /* compute the temporary graph $G_j^{(i)}$; $D_{i,j}$ is used as defined in (3.1) */
7          $G_j^{(i)} \leftarrow G$.REMOVEALLEDGESTO$(v_1^{(i)}, \dots, v_j^{(i)})$
8          $G_j^{(i)} \leftarrow G_j^{(i)}$.REMOVEEDGES$(D_{i,j})$
           /* compute the shortest path in the reduced graph $G_j^{(i)}$ */
9          $P \leftarrow (v_1^{(i)}, \dots, v_j^{(i)}) \circ$ SSSP$(G_j^{(i)}, v_j^{(i)}, t)$
           /* add the new candidate path $P$, the deviation index, */
           /* and the set of already used edges to the candidate list */
10         $C$.PUSH$((P, j, D_{i,j}))$
```

The algorithm first computes the shortest path $P_1 = (v_1^{(1)}, \dots, v_{|P_1|}^{(1)})$ using an arbitrary single-source shortest-path algorithm, e.g., Dijkstra's algorithm. Let then $C = \{(P_1, 1, \{\})\}$ be the set of candidates for the k shortest paths.

We use Δ-stepping in our implementations.

For $i = 1, \dots, k$ Yen's algorithm chooses the shortest path $(P_i, \mathrm{di}(P_i), D_i)$ that is currently in the set of candidates C, with $\mathrm{di}(P_i)$ being the index of the deviation node $\mathrm{dev}(P_i)$ where P_i deviated from its parent path, and D_i being the set of edges that was already used when deviating from $\mathrm{dev}(P_i)$ while having the same prefix path $\mathrm{R}_i(\mathrm{di}(P_i))$. It computes new candidates as follows:

As an intuition, the sets D_i are used to remember which paths are already found and which edges cannot be used in order to find other paths.

For $j = \mathrm{di}(P_i), \dots, |P_i| - 1$ let $G_j^{(i)} = (V_j^{(i)}, E_j^{(i)})$ be the graph induced from G by removing all nodes $v_1^{(i)}, \dots, v_{j-1}^{(i)}$ along with their incident edges. For $j = \mathrm{di}(P_i)$ all edges from D_i also get removed. Then it computes the shortest path $c_j^{(i)}$ from $v_j^{(i)}$ to t in $G_j^{(i)}$ using a single-pair shortest-path algorithm and adds $(\mathrm{R}_i(j) \circ c_j^{(i)}, j, D_{i,j})$ to the set of candidates C, with

$$D_{i,j} = \begin{cases} \left\{ \left(v_j^{(i)}, v_{j+1}^{(i)} \right) \right\} \cup D_i & : \ j = \mathrm{di}(P_i) \\ \left\{ \left(v_j^{(i)}, v_{j+1}^{(i)} \right) \right\} & : \ j \neq \mathrm{di}(P_i) \end{cases} \tag{3.1}$$

The pseudocode can be found in Algorithm 3. In order to keep it simple, the pseudocode does not contain exception handling, e.g., the case that there are less than k paths in total or if there is no deviation found.

In the worst-case Yen's algorithm computes for each hop on each of the first $k - 1$ shortest paths a deviation candidate. Since simple s–t-paths in a graph with n nodes have at most $\mathcal{O}(n)$ hops, this results in a total of at most $\mathcal{O}(kn)$ deviations to compute. Let

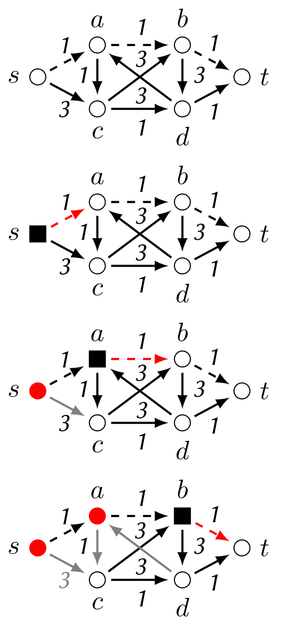


Figure 3.2: The directed graph G with edge weights. The dashed edges are part of the shortest path P_1. The square nodes are the respective deviation nodes with red nodes and edges being temporarily removed from the graph when computing the deviations.

$\mathrm{spc}(n, m)$ be the worst-case time complexity of the SSSP algorithm in use, then the worst-case complexity of Yen's algorithm is $\mathcal{O}(kn \cdot \mathrm{spc}(n, m))$, which is $\mathcal{O}(kn(m + n \log n))$ if Dijkstra's algorithm with Fibonacci heaps is used [27].

We show in Chapter 4 that the average-case complexity of Yen's algorithm is much better than its worst-case complexity.

3.2.1.2 A Small Example

Given the directed graph G visualized in Figure 3.2, we want to compute the $k = 3$ shortest paths from s to t. Yen's algorithm first computes the shortest path $P_1 = (s, a, b, t)$. Since it is not a deviation path, we set the deviation index $\mathrm{di}(P_1) = 1$ and the set of forbidden edges to $D_1 = \{\}$. Now, Yen's algorithm computes a deviation path for each node but the last on the shortest path (see Fig. 3.2).

- Deviating from node s: $D_{1,1} = \{(s, a)\} \cup D_1$ contains only the out-edge of s, since D_1, the set of previously used edges, is empty by definition. So the edge (s, c) has to be used in the shortest deviation. This gives us the candidate path $c_{1,1} = (s, c, d, t)$ and we store $(c_{1,1}, \mathrm{di}(c_{1,1}) = 1, D_{1,1})$ in the candidate list C with a total weight-length of $\mathrm{d}(c_{1,1}) = 5$.

- Deviating from node a: $D_{1,2} = \{(a, b)\}$ contains also only a single edge because a is not the deviation node of P_1. This forces the SSSP algorithm to choose the edge (a, c) resulting in the deviation path $c_{1,2} = (s, a, c, d, t)$. We store $(c_{1,2}, \mathrm{di}(c_{1,2}) = 2, D_{1,2})$ in the candidate list C with a total weight-length of $\mathrm{d}(c_{1,2}) = 4$.

- Deviating from node b: Similar to the last deviation, $D_{1,3} = \{(b, t)\}$ contains only a single edge and we get $c_{1,3} = (s, a, b, d, t)$ and store $(c_{1,3}, \mathrm{di}(c_{1,3}) = 3, D_{1,3})$ in the candidate set C with a weight-length of $\mathrm{d}(c_{1,1}) = 6$.

Now all deviations of P_1 are computed yielding $P_2 = c_{1,2}$ as the second-shortest path with $\mathrm{di}(P_2) = 2$ and $D_2 = \{(a, b)\}$ while $c_{1,1}$ and $c_{1,3}$ remain in the candidate list C. In order to compute the third-shortest path Yen's algorithm computes the candidates from the second-shortest path. Since the deviation node index is $\mathrm{di}(P_2) = 2$, Yen's algorithm does not compute a deviation from node s and instead starts with node a (see Fig. 3.3).

- Deviating from node a: The set of already used out-edges $D_{2,2} = \{(a, c)\} \cup D_2$ contains all out-edges of a, which is why we cannot find a deviation path here.

- Deviating from node c: With $D_{2,3} = \{(c, d)\}$ we find the shortest deviation path to be $c_{2,3} = (s, a, c, b, t)$ and add $(c_{2,3}, \mathrm{di}(c_{2,3} = 3, D_{2,3}))$ as a new candidate to C with a length of $\mathrm{d}(c_{2,3}) = 6$.

- Deviating from node d: The only out-edge of node d leads to a which is at that time temporarily removed from the graph as it would introduce a loop to any found deviation paths. Thus we cannot find any deviation paths.

After all deviations are computed, $P_3 = c_{1,1}$ is the shortest path in the candidate list C with a length of $\mathrm{d}(c_{1,1}) = 5$. Note that the third-shortest path deviated from the first shortest path and not from the second one. Since we found all $k = 3$ shortest paths, Yen's algorithm stops here.

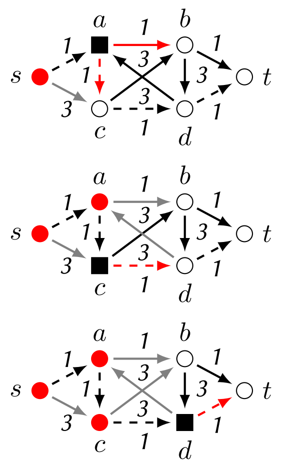


Figure 3.3: The dashed edges are part of the second-shortest path P_2. The square nodes are the respective deviation nodes with red nodes and edges are temporarily removed from the graph when computing the deviations.

3.2.1.3 Correctness

Assume that Yen's algorithm correctly computed the first i shortest paths. Let $P_l = \mathrm{par}(P_i)$ be the parent path of P_i. In order to compute the $(i+1)^{\mathrm{th}}$-shortest path, the algorithm computes deviations in addition to the deviations from the other paths that are already in the list of candidates. We have to check three cases of deviations.

1. Deviations at nodes $v_j^{(i)}$ with $j < \mathrm{di}(P_i)$. The prefixes $\mathrm{R}_i(j)$ are identical to the prefix paths $\mathrm{R}_l(j)$ of the parent path of P_i. So all deviations from those nodes would lead to paths that were already computed from P_l or even earlier and thus are skipped by Yen's algorithm.

2. The node $v_j^{(i)}$ with $j = \mathrm{di}(P_i)$ is the deviation node, where P_i deviated from its parent path P_l (see Figure 3.4a). To compute a deviation it is not enough to remove only the edge $e_j = \left(v_j^{(i)}, v_{j+1}^{(i)}\right)$, because this would result in finding the parent path P_l or an even shorter path again. So in addition to e_j all edges need to be removed, that were already used by P_l and its predecessors at this deviation node. All these edges are collected in D_i.

3. Deviations at nodes $v_j^{(i)}$ with $j > \mathrm{di}(P_i)$ (see Figure 3.4b). For these nodes the prefix path $\mathrm{R}_i(j)$ is unique among the first i shortest paths and thus only the edge $\left(v_j^{(i)}, v_{j+1}^{(i)}\right)$ has to be removed.

In addition to the edges in $D_{i,j}$ all in-edges to the nodes $v_1^{(i)}, \ldots, v_j^{(i)}$ get removed to prevent loops, since these are the nodes on the prefix path $\mathrm{R}_i(j)$ and thus cannot be visited again without closing a loop. The reduced graph $G_j^{(i)}$ still contains all paths from $v_j^{(i)}$ to t that do not visit a node in the prefix path $\mathrm{R}_i(j)$, except for those which have already been computed before. Starting with the shortest path from s to t computed in the full graph G, the correctness of Yen's algorithm follows by induction.

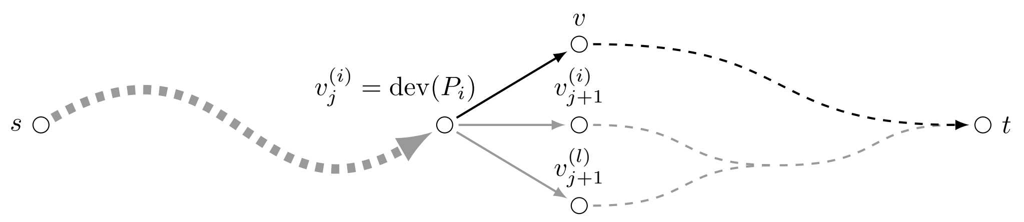
3.2.2 Feng's k-Shortest Path Algorithm

Feng's algorithm works just like Yen's algorithm but adds additional strategies to reduce the average complexity. Namely these are:

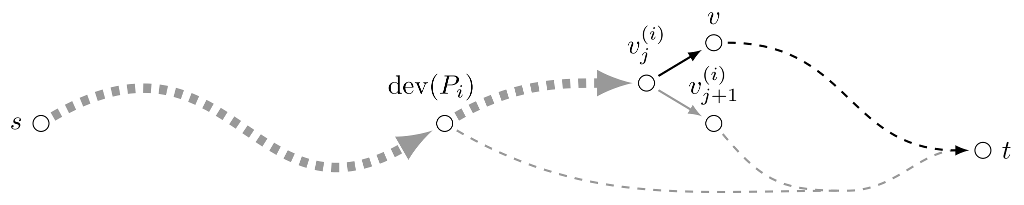
- (**guiding**) A preprocessing step, that computes an improved edge weight function to guide the single-source shortest-path algorithm into the right direction.

- (**coloring**) A node coloring to reduce the size of the graphs $G_j^{(i)}$ on which the single-pair shortest-path algorithm needs to be executed on.

- (**skipping**) Attempt to skip calls to an SSSP algorithm.



(a) Deviating from P_i at the node where P_i deviated from its parent path P_l. Since P_i deviated at $\mathrm{dev}(P_i)$ from P_l, there are at least two edges that cannot be used again with the given prefix path.



(b) Deviating from nodes after the deviation node. Since the edge used after the deviation node $\mathrm{dev}(P_i)$ makes the prefix path unique, only the edge $\left(v_j^{(i)}, v_{j+1}^{(i)}\right)$ cannot be used.

Figure 3.4: Visualization of how candidate paths deviate from their parent path P_i at the deviation node of P_i (Fig. 3.4a) and at other nodes (Fig. 3.4b). Nodes on the prefix path $\mathrm{R}_i(j)$ (gray bold dashed arrow) cannot be visited since this would introduce a loop. The gray edges must also not be used since this would rediscover P_i itself or an even shorter path.

The pseudocode of Feng's algorithm can be found in Algorithm 5 at the end of the section.

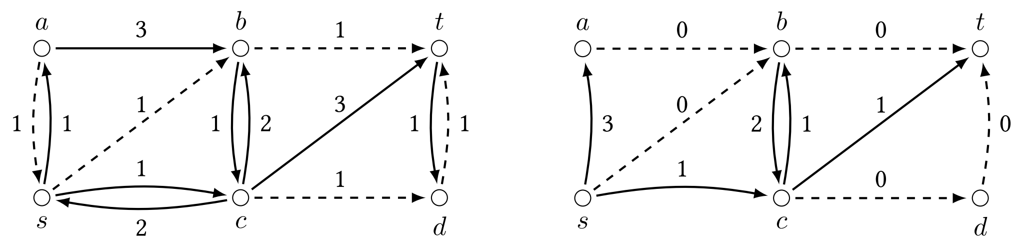
3.2.2.1 The Reverse Shortest Path Tree

G' is the graph obtained from G by removing all in-edges of s and out-edges of t as well as inverting the direction of all other edges.

All three strategies use the reverse shortest path tree T. Given the directed input graph $G = (V, E)$ with a weight function d, a source node $s \in V$, and a target node $t \in V$, the reverse shortest path tree is the shortest path tree T computed on the graph $G' = (V, E')$ with $E' := \{(u, v) : (v, u) \in E, u \neq s, v \neq t\}$ and weight function $\mathrm{d}'(u, v) := \mathrm{d}(v, u)$ for each edge $(u, v) \in E'$. The reverse shortest path tree T contains for each node $v \in V$ a shortest path from t to v in G' that does not go through s. These paths are equivalent to the shortest paths in G from v to t that do not go through s. Removing shortest paths that go through s from the graph is a simple optimization because such path can never be loopless since all s–t-paths already start at node s. However, T still contains a shortest path from s to t, so computing the reverse shortest path tree saves us computing the shortest s–t-path separately.

The shortest path from s to t does not go through s but starts in s.

We want to point out that shortest paths do not have to be unique. If multiple shortest paths from v to t exist, the reverse shortest path tree contains only one of them. Which one is arbitrary and decided by the implementation of the SSSP algorithm in use.



(a) Graph G with edge weights according to d. (b) Graph G^* with edge weights according to d^*.

Figure 3.5: *A graph G (Fig. 3.5a) and the graph G^* (Fig. 3.5b) generated from G by the graph preprocessing described in Section 3.2.2.2. Dashed edges are part of the reverse shortest path tree rooted in t.*

3.2.2.2 Preprocessing the Graph

Feng's algorithm precomputes an additional graph

$$G^* = (V, E^*) \quad \text{with} \quad E^* := \{(u,v) \in E \; : \; u \neq s, v \neq t\} \tag{3.2}$$

G' and G^* contain the same edges but in G' all directions are reversed while edges in G^* point in the same direction as in the original graph G.

removing all in-edges of s and out-edges of t from G. Since all s–t-paths start with the node s, an in-edge of s can never be used without introducing a loop. A similar argument holds for the out-edges of t. So the removal of these edges does not change the result in any way. However, it removes all paths from the graph, that go through the source node s and so cannot be part of the k shortest simple paths. Then it uses the reverse shortest path tree T to compute an improved edge weight function $\mathrm{d}^*(u,v)$ defined as

$$\mathrm{d}^*(u,v) := \begin{cases} \mathrm{d}(u,v) + \mathrm{d}(v \overset{G^*}{\dashrightarrow} t) - \mathrm{d}(u \overset{G^*}{\dashrightarrow} t) & : \mathrm{d}(v \overset{G^*}{\dashrightarrow} t) < \infty \\ \infty & : \mathrm{d}(v \overset{G^*}{\dashrightarrow} t) = \infty \end{cases} \tag{3.3}$$

The edge-weight function $\mathrm{d}(\cdot)$ is defined in G but since $E^* \subseteq E$ holds, it also applies to G^*. The notation $\overset{G^*}{\dashrightarrow}$ indicates that we talk about a shortest path within G^* and not within G even though we use the edge-weight function d.

for each edge $(u,v) \in E^*$ as a replacement for the given weight function $\mathrm{d}(\cdot)$. The weight function $\mathrm{d}^*(\cdot)$ reduces the weight of edges leading toward the target node, and increases the weight of edges leading away from it. Figure 3.5 shows an example of how the preprocessing affects shortest paths and edge weights.

Lemma 3.1. Given a directed graph $G = (V, E)$ with a non-negative edge-weight function d, a source node $s \in V$, and target node $t \in V$. Then in the graph G^* as defined in Equation (3.2), the edge weight function $\mathrm{d}^*(u,v)$ as defined in Equation (3.3) has the following properties:

a) $\mathrm{d}^*(u,v) \geq 0$ holds for all edges $(u,v) \in E^*$.

b) Let P be an arbitrary path from u to t in G^*. Then $\mathrm{d}^*(P) = \mathrm{d}(P) - \mathrm{d}(u \overset{G^*}{\dashrightarrow} t)$ holds for all nodes $u \in V$.

c) $\mathrm{d}^*\big(u \dashrightarrow t\big) = 0$ holds for all nodes $u \in V$.

d) Let P_1 and P_2 be two different paths from u to t in G^* with $\mathrm{d}(P_1) \leq \mathrm{d}(P_2)$. Then $\mathrm{d}^*(P_1) \leq \mathrm{d}^*(P_2)$ holds.

e) Let $(u_1, u_2, \ldots u_r, t)$ be a shortest path from u_1 to t in G^*. Then $\mathrm{d}^*(u_1, u_2) = 0$ holds. ◀

Note that the shortest paths $u \xrightarrow{G^*} t$ are defined by the weight function d but restricted to only use edges in G^*. However, Lemma 3.1 implies that shortest paths in terms of the original weight function d are also shortest paths in terms of the new weight function d^*.

Proof. Property b) directly implies property c) by choosing $P = u \xrightarrow{G^*} t$. Property b) also implies property d), since the lengths of all paths from u to t are shifted by the same value $\mathrm{d}(u \xrightarrow{G^*} t)$. Property e) follows directly from properties a) and c). So we are left to show the first two properties.

a) Assume $\mathrm{d}^*(u, v) < 0$ would hold. This translates to $\mathrm{d}(u, v) + \mathrm{d}\big(v \xrightarrow{G^*} t\big) < \mathrm{d}\big(u \xrightarrow{G^*} t\big)$ which means that the path $(u, v, \ldots, t)$, with a detour over v to t, is shorter than the shortest path from u to t. This is a contradiction, so $\mathrm{d}^*(u, v) \geq 0$ needs to hold.

b) Let $P = (u_1, \ldots u_r, u_{r+1} = t)$ be an arbitrary path from u_1 to t, then $\mathrm{d}^*(P)$ can be written as

We say the length of the shortest path from a node to itself has a length of zero.

$$\begin{aligned}
\mathrm{d}^*(P) &= \sum_{i=1}^{r} \mathrm{d}^*(u_i, u_{i+1}) \\
&= \sum_{i=1}^{r} \Big(\mathrm{d}(u_i, u_{i+1}) + \mathrm{d}\big(u_{i+1} \xrightarrow{G^*} t\big) - \mathrm{d}\big(u_i \xrightarrow{G^*} t\big)\Big) \\
&= \sum_{i=1}^{r} \mathrm{d}(u_i, u_{i+1}) + \sum_{i=2}^{r+1} \mathrm{d}\big(u_i \xrightarrow{G^*} t\big) - \sum_{i=1}^{r} \mathrm{d}\big(u_i \xrightarrow{G^*} t\big) \\
&= \mathrm{d}(P) + \mathrm{d}\big(t \xrightarrow{G^*} t\big) - \mathrm{d}\big(u_1 \xrightarrow{G^*} t\big) \\
&= \mathrm{d}(P) - \mathrm{d}\big(u_1 \xrightarrow{G^*} t\big)
\end{aligned}$$ □

By Lemma 3.1 d) the preprocessing step preserves the relative order of all s–t-paths according to their lengths, and the shortest path has a weight-length of zero. So the resulting graph of this precomputation G^*, used by Feng's algorithm, can also be used as an input by any other k-SP algorithm, such as Yen's algorithm, without any change to the algorithm. We compare Yen's algorithm using the preprocessing with Feng's algorithm later in Chapter 6.

3.2.2.3 Node Coloring

Feng [24] showed that the nodes of G fall into three categories depending on the current deviation node $v_j^{(i)}$:

1. Red nodes: Nodes that must not be visited anymore in order to avoid loops. These nodes are $v_1^{(i)}, \ldots, v_j^{(i)}$, the prefix nodes of the ith-shortest path.

2. Yellow nodes: Nodes u where the shortest path from u to t in the reverse shortest path tree goes through at least one of the red nodes $v_1^{(i)}, \ldots, v_j^{(i)}$.

3. Green nodes: All nodes that are not yellow or red.

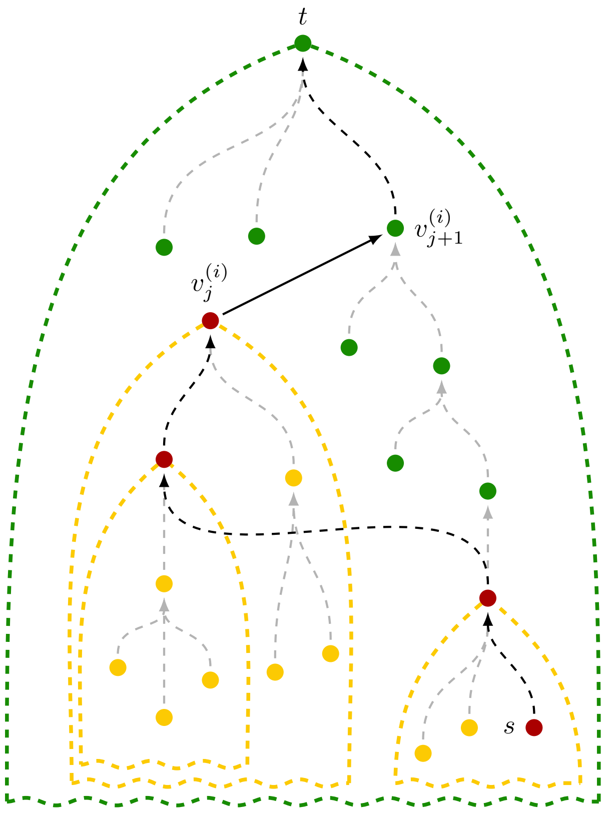


Figure 3.6: *An example for the node coloring described by Feng, when deviating from $v_j^{(i)}$. The i^{th} shortest path P_i runs along the black dashed line. The gray arrows are paths within the reverse shortest path tree T rooted at t, while P_i can use edges that are not part of T. Note that s has no yellow subtree since all in-edges are removed from the graph as described in Section 3.2.2.2.*

We refer to Figure 3.6 for an example. When deviating from the i^{th}-shortest path P_i at node $v_j^{(i)}$ the shortest path uses only yellow nodes until it hits the first green node u. From there on it only uses green nodes.

Lemma 3.2. When deviating from the i^{th}-shortest path P_i at node $v_j^{(i)}$ the shortest deviation path has the form

$$c_j^{(i)} = P_r \circ P_y \circ P_g$$

where P_r consists of only red nodes, P_y of only yellow nodes and P_g of only green nodes. ◀

Proof. The red prefix path is the prefix $\mathrm{R}_i(j)$ of the path P_i we currently deviate from. Since these nodes are already part of the deviation path, they must not be visited later because this would introduce a loop. So the shortest path from the deviation node $v_j^{(i)}$ to t consists of some yellow and some green nodes. Let g be the first green node on that path. By the definition of green nodes we know that the shortest path from g to t does not go through a red node. So this needs to hold for all nodes on the shortest path $g \dashrightarrow t$. If it would hit a yellow node y, the shortest path from y to t would pass a red node r by definition of yellow nodes and thus the shortest path from g to t would also go through r. In this case g could not have been green in the first place.

Thus the deviation path cannot go through any red or yellow nodes after hitting the first green node. So after the all-red prefix path comes a yellow subpath that could be empty followed by an all-green suffix path. □

The yellow subpath needs to be computed by an SSSP algorithm within the induced yellow subgraph $Y_j^{(i)} = (V_j^{(i)}, E_j^{(i)})$, which consists of the nodes $v_j^{(i)}$, t, and all yellow nodes with respect to $v_j^{(i)}$. The yellow graph contains all induced edges $(u,v) \in E$ with $u, v \in Y_j^{(i)}$ as well as an express edge (v,t) for each edge $(v,g) \in E$ with $v \in Y_j^{(i)}$ and a green node g with $\mathrm{d}(v,t) = \mathrm{d}(v,g) + \mathrm{d}\big(g \dashrightarrow t\big)$ or $\mathrm{d}^*(u,t) = \mathrm{d}^*(u,v)$, respectively.

The yellow graph needs to be recomputed for each deviation node and the path it is on. This can be done by a depth first search from each of the red nodes on the reverse shortest path tree T, which in total takes $\mathcal{O}(n)$ time due to the following Lemma:

Note that the sets Y_i are not whole subtrees but a specific subset of nodes. Have a look at Figure 3.6 for an illustration.

Lemma 3.3. Let $r_1 \neq r_2$ be two red nodes and let Y_1 and Y_2 be the sets of yellow nodes such that r_i is the first red node on the shortest path from $y \in Y_i$ to t according to reverse shortest path tree for $i = 1, 2$. Then $Y_1 \cap Y_2 = \{\}$ holds. ◀

Proof. Since the color of a node is determined by a DFS traversal from a red node in the reverse shortest path tree, each yellow node has a unique first red node on the shortest path to t. Thus Y_1 and Y_2 cannot share any nodes. □

By Lemma 3.3 the yellow graph $Y_j^{(i)}$ can be computed by adding the yellow nodes hanging from $v_j^{(i)}$ to the yellow graph $Y_{j-1}^{(i)}$. So computing the node coloring can be done in $\mathcal{O}(n)$ time in total by reusing the yellow graphs. In order to maintain the express edges we need to check for each express edge (u,t) if its original head node changed from green to either yellow or red. If so, the express edge needs to be removed and the original edge has to be restored. This takes $\mathcal{O}(m)$ time per deviation, but is still dominated by the time needed for the SSSP computation.

After the candidate path is computed within the yellow subgraph, the express edge that leads from a yellow node directly to the target node t, is expanded to the corresponding shortest path, which can be pulled from the reverse shortest path tree T.

Algorithm 4 shows the pseudocode to compute the yellow graph, when deviating from the i^{th}-shortest path at the j^{th} node.

3.2.2.4 Skipping SSSP Computations

According to Lemma 3.2, no SSSP computation is needed if the closest neighbor of the deviation node $v_j^{(i)}$ is green. When deviating at the source node s, SSSP computations can always be skipped since s has no in-edges after the preprocessing step and thus all neighbors are green.

Lemma 3.4. Given a directed graph $G = (V, E)$ with a non-negative edge weight function d, a source node s, and a target node $t \in V$. Let $G^* = (V, E^*)$ be defined as in Equation (3.2) and d^* as in Equation (3.3). Then all deviations at node s can be computed without the need to call an SSSP algorithm using the reverse shortest path tree T only. ◀

Proof. All deviations at the source node s are of the form $s \to u \dashrightarrow t$, where u is a neighbor of s. The shortest paths $u \dashrightarrow t$ do not pass through s since s has no in-edges

Algorithm 4: Subroutine computeYellowGraph

```
Input  : Directed graph $G = (V, E)$, edge weight function $\mathrm{d}$, reverse shortest path tree $T$,
         set of edges to remove $D_{i,j}$, path $P_i = \left(v_1^{(i)}, \dots, v_{|P_i|}^{(i)}\right)$, deviation node index $j$.
Output : The yellow graph $Y_j^{(i)}$, weight function $\mathrm{d}_y$, express edge mapping $M$
1  $E' \leftarrow E \setminus D_{i,j}$
   /* set of yellow nodes $Y$ */
   /* at the beginning all nodes are implicitly green, therefore $Y$ is empty */
2  $Y \leftarrow \{\}$
3  $R \leftarrow \left\{v_1^{(i)}, \dots, v_j^{(i)}\right\}$                // set of red nodes
4  $S \leftarrow \{\}$            // empty stack used for the DFS traversal of $T$
5  for $l \leftarrow 1, \dots, j$ do                          // find the yellow nodes
6      $S$.pushMultiple($T$.getNeighborsOf($v_l^{(i)}$))
7      while $S$ is not empty do
8          $v \leftarrow S$.pop()
           /* the coloring stops if a red node is discovered */
           /* yellow nodes cannot be rediscoverd since $T$ is a tree */
9          if $v \notin R$ then
10             $Y \leftarrow Y \cup \{v\}$
11             $S$.pushMultiple($T$.getNeighborsOf($v$))
12 $X \leftarrow \{\}$                                   // set of express edges
13 $M \leftarrow \{\}$                // the map from express edges to original edges
14 $\mathrm{d}_y \leftarrow \mathrm{d}$
15 foreach $(u, v) \in \{(u, v) \in E' : u \in R \cup Y, v \notin R \cup Y\}$ do      // Add the express egeds
16     $X \leftarrow X \cup \{(u, t)\}$
       /* check if the new express edge is shorter than the old express edge */
       /* we assume $\mathrm{d}_y(u, t) = \infty$ if not defined otherwise */
17     if $\mathrm{d}(u, v) < \mathrm{d}_y(u, t)$ then
18         $M[u] \leftarrow v$
19         $\mathrm{d}_y(u, t) \leftarrow \mathrm{d}(u, v)$
20 $Y_j^{(i)} \leftarrow \left(Y \cup \left\{v_j^{(i)}, t\right\}, \{(u, v) \in E' : u, v \in Y\} \cup X \cup \left\{(u, v) \in E' : u = v_j^{(i)}\right\}\right)$
```

within G^*. So these paths can simply be pulled from the reverse shortest path tree T without any extra computations. $\square$

3.2.2.5 Correctness and Complexity of Feng's Algorithm

Lemmas 3.1, 3.2, and 3.4 show that none of the additional work Feng's algorithm does changes or removes any s–t-paths in the graph. So the correctness follows from the correctness of Yen's algorithm.

The preprocessing is dominated by a full run of an SSSP algorithm. Computing the yellow graph takes only linear time in the number of edges. So both are dominated by the calls of the running time for the SSSP algorithm computing the actual deviation path. Even though Feng's algorithm has a lower average-case complexity than Yen's

Implementation details: ☞ *Section 5.3*

Algorithm 5: Feng's Algorithm

```
Input  : Directed weighted graph $G$, nodes $s$ and $t$, the number $k$ of paths to compute
Output : The $k$ shortest $s$–$t$-paths $P_1, \dots, P_k$ sorted by increasing length.
1  $G^* \leftarrow G$.removeAllInEdgesOf($s$).removeAllOutEdgesOf($t$)
2  $T \leftarrow$ computeReverseSSSPTree($G^*, t$)        // get the shortest path from $s$ to $t$ from $T$
3  $P \leftarrow T$.getShortestPathFrom($s$)
4  $C \leftarrow \{(P, 1, \{\})\}$                         // priority queue of candidates
5  for $i \leftarrow 1, \dots, k$ do
6      $(P_i, \mathrm{di}(P_i), D_i) \leftarrow C$.popMin()     // the shortest path among the candidates
7      if $i = k$ then return
8      for $j \leftarrow \mathrm{di}(P_i), \dots, |P_i| - 1$ do  // compute all deviations from $i^{\mathrm{th}}$-shortest path
           /* see Algorithm 4 for details; $D_{i,j}$ is defined in (3.1); $\mathrm{d}^*$ as defined in (3.3)   */
9          $Y_j^{(i)}, \mathrm{d}_y, M \leftarrow$ computeYellowGraph($G^*, \mathrm{d}^*, T, D_{i,j}, P_i, j$)
10         $u \leftarrow Y_j^{(i)}$.getClosestNeighbor($v_j^{(i)}$)
           /* attempt to skip the SSSP computation   */
11         if $u$ is green then        // edges to green nodes point directly to $t$ in $Y_j^{(i)}$
12             $P \leftarrow \mathrm{R}_i(j) \circ T$.getShortestPathFrom($u$)
13         else
               /* compute the shortest path in the yellow graph $Y_j^{(i)}$   */
14             $P \leftarrow \mathrm{R}_i(j) \circ$ sssp($Y_j^{(i)}, \mathrm{d}_y, v_j^{(i)}, t$)
               /* the last edge is always an express edge   */
15             $M$.mapExpressEdgeToPath($P, T$)
           /* add the new candidate path $P$, the deviation index, and the set of   */
           /* already used edges to the candidate list   */
16         $C$.push(($P, j, D_{i,j}$))
```

algorithm, they both have the same worst-case complexity. In Chapter 4 we analyze the average-case complexity of Yen's and Feng's algorithms.

3.2.3 Variants Between Yen's and Feng's Algorithm

The three optimizations used by Feng's algorithm (graph preprocessing, node coloring, and skipping SSSP computations) can mostly be used independently and give reason to take a closer look into three algorithms, that are to some extend hybrids between Yen's and Feng's algorithm. Namely these are:

- Yen-s: Yen's algorithm as described in Section 3.2.1, but also attempting to skip SSSP computations. In order to do such attempts, Yen-s needs the reverse shortest path tree as described in Section 3.2.2.1. From there it can pull the shortest paths as needed. Contrary to Feng's algorithm, Yen-s cannot check if the neighbor is green since it does not compute the yellow graph. Instead it needs to make sure that the path it gets from the reverse shortest path tree does not intersect with the prefix path $\mathrm{R}_i(j)$ when deviating at the j^{th} node from the i^{th}-shortest path.

- Yen-g: Yen's algorithm as described in Section 3.2.1 but using the preprocessing described in Section 3.2.2.2 beforehand.
- Yen-gs: Just like Yen-s as described before but using the preprocessing described in Section 3.2.2.2 in addition.

We discuss these algorithms and further heuristic improvements in Chapter 5.

3.2.4 The KIM Algorithm

Yen's algorithm as well as Feng's algorithm, presented in Sections 3.2.1 and 3.2.2, work on both directed and undirected graphs. However, there is also an algorithm by Katoh, Ibaraki, and Mine [40] specialized for undirected graphs further referred to as KIM. The KIM algorithm exploits properties of undirected graphs and achieves a worst-case time complexity of $\mathcal{O}(k \cdot \mathrm{spc}(n, m))$ for computing k shortest paths on an undirected edge-weighted graph with n nodes and m edges, where $\mathrm{spc}(n, m)$ is the complexity of the used SSSP algorithm. Compared to the time complexity of Yen's and Feng's algorithm, the time complexity of KIM is better by a factor of n. So on first glance it looks like KIM should always be preferred on undirected graphs, but, as we see later in Chapter 4, the respective average-case time complexities of Yen's and Feng's algorithm are far better than their worst-case complexity. We further show in Chapter 5 that on $\mathcal{G}(n, p)$ graphs an SSSP algorithm only has to explore a tiny fraction of the graph and most of the SSSP computations can be skipped completely which makes some versions of Yen's and Feng's algorithm competitive to KIM as we show in Section 5.4.3.

Since the KIM algorithm is not the main focus of this thesis, we keep this section short and refer to [40] for more details and proofs.

3.2.4.1 Finding the Second-Shortest Path

The KIM algorithm is based on Lemma 3.5.

Lemma 3.5. [40, Lemma 2.1] Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, let P_1 be the shortest path from s to t. Furthermore let $T(s)$ and $T(t)$ be the SSSP trees rooted in s and t, respectively. If G contains simple paths from s to t that do not contain the prefix path $\mathrm{R}_1(\alpha)$ of P_1, let P be a shortest of them. Then P is either of Type 1 or Type 2:

The prefix path $\mathrm{R}_1(\alpha)$ contains the first α nodes of the path P_1 as defined on p. 13.

Type 1 P is of the form $s \dashrightarrow u \dashrightarrow t$, where $s \dashrightarrow u$ only uses edges of $T(s)$, $u \dashrightarrow t$ only uses edges of $T(t)$, and $s \dashrightarrow u$ deviates from P_1 at the β^{th} node with $\beta < \alpha$.

Type 2 P is of the form $s \dashrightarrow u \rightarrow v \dashrightarrow t$, where $u \rightarrow v$ is an edge that is neither in $T(s)$ nor $T(t)$ and $s \dashrightarrow u$ deviates from P_1 at the β^{th} node with $\beta < \alpha$.

◀

Shortest paths of Type 1 can be found by iterating over all nodes in G that are not in P_1 and looking for the smallest combined distance in $T(s)$ and $T(t)$ in $\mathcal{O}(n)$

Comparison between KIM and Yen's algorithm: ☞ *Section 3.2.4.5*

time. Similarly, shortest paths of Type 2 can be found by iterating over all edges that are not in P_1 in $\mathcal{O}(m)$ time. So in total the second-shortest path can be computed in $\mathcal{O}(n) + \mathcal{O}(m) + \mathcal{O}(\mathrm{spc}(n,m)) = \mathcal{O}(\mathrm{spc}(n,m))$, where computing both of the shortest path trees $T(s)$ and $T(t)$ takes $\mathcal{O}(\mathrm{spc}(n,m))$ time each.

FSP algorithm: ☞ *Algorithm 6*

Katoh et al. use a subroutine called FSP in order to find a shortest path in the setting of Lemma 3.5.

Algorithm 6: Subroutine FSP

```
Input  : $G = (V, E), s, t \in V, \mathrm{R}_1(\alpha)$
Output : The shortest $s$–$t$-path $P$ in $G$ that does not contain the prefix $\mathrm{R}_1(\alpha)$.
/* compute both SSSP trees storing for each node $u$ the respective distance $\mathrm{d}(s \dashrightarrow u)$  */
/* and $\mathrm{d}(t \dashrightarrow u)$, the predecessor nodes $F_s(u)$ and $F_t(u)$ of $u$ in $s \dashrightarrow u$ and $t \dashrightarrow u$, */
/* respectively, the indexes $\xi(u)$ and $\zeta(u)$ where $s \dashrightarrow u$ and $t \dashrightarrow u$ deviate first from */
/* $P_1$, respectively, and the set of nodes $S(u)$, such that $s \dashrightarrow u$ is a prefix of the  */
/* shortest path $s \dashrightarrow v$ for each node $v \in S(u)$.  */
1  $T(s) \leftarrow$ SSSP($G$, $s$)
2  $T(t) \leftarrow$ SSSP($G$, $t$)
3  $H \leftarrow \{\}$      // a node (Type 1) or an edge (Type 2) that results in the shortest deviation
4  $L \leftarrow \infty$      // the length of the deviation defined by $H$
5  $N \leftarrow \{s\}$      // a stack of nodes to be processed initialized with the source node $s$
6  while $N \neq \{\}$ do      // starting with $s$ explore the nodes in DFS order
7      $u \leftarrow N$.POP()      // store the top element in $u$ and remove it from the stack
8      if $\xi(u) < \zeta(u)$ then      // Type 1
9          $D \leftarrow \mathrm{d}(s \dashrightarrow u) + \mathrm{d}(t \dashrightarrow u)$
10         if $D < L$ then
11             $L \leftarrow D, H \leftarrow u$
12     else if $\xi(u) = \zeta(u)$ then      // Type 2
13         foreach $v \in \{v : \{u, v\} \in E\} \setminus S(u)$ with $\xi(v) < \alpha$ do
14             $D \leftarrow \mathrm{d}(s \dashrightarrow u) + \mathrm{d}(u, v) + \mathrm{d}(t \dashrightarrow v)$
15             if $D < L$ then
16                 $L \leftarrow D, H \leftarrow (u, v)$      // the direction of the edge matters
17     if $\xi(u) \leq \zeta(u)$ then
18         foreach $v \in S(u)$ with $\xi(v) < \alpha$ do
19             $N$.PUSH($v$)
20 $P \leftarrow \{\}$      // initialize the result as empty
21 if $H = u^*$ is a node then      // $H$ is a node (Type 1)
22     $P \leftarrow (L, \xi(u^*), s \dashrightarrow u^* \dashrightarrow t)$
23 else if $H = (u^*, v^*)$ is an edge then      // $H$ is an edge (Type 2)
24     $P \leftarrow (L, \xi(u^*), s \dashrightarrow u^* \rightarrow v^* \dashrightarrow t)$
```

The while-loop and the stack (lines 5–19) are referred to as the recursive subroutine SEP in [40].

3.2.4.2 Finding the Third-Shortest Path

For the third-shortest path there are now three possible types of deviations from the second-shortest path to be considered. Let $\alpha = \mathrm{di}(P_2)$ be the index where P_2 deviated from P_1. Then there are three candidates for the third-shortest path:

1. P_a deviates from P_2 at the $(\alpha+1)^{\text{th}}$ node or later meaning P_a contains $\mathrm{R}_2(\alpha+1)$ as prefix. This deviation can be computed by removing the nodes $v_1^{(2)}, \ldots, v_\alpha^{(2)}$ from G and using $v_{\alpha+1}^{(2)}$ as source node while prohibiting the suffix path $\mathrm{S}_2(\alpha+1)$. Let P be the shortest path from $v_{\alpha+1}^{(2)}$ to t, then $P_a = \mathrm{R}_2(\alpha+1) \circ P$ is the first candidate.

2. P_b deviates from P_1 at the α^{th} node or later but cannot contain $\mathrm{R}_2(\alpha+1)$ as a prefix. This deviation can be computed by removing the nodes $v_1^{(2)}, \ldots, v_{\alpha-1}^{(2)}$ and the edge $\left\{v_\alpha^{(2)}, v_{\alpha+1}^{(2)}\right\}$ from G and using $v_\alpha^{(1)}$ as source node while prohibiting the suffix path $\mathrm{S}_1(\alpha)$. Let P be the shortest path from $v_\alpha^{(1)}$ to t, then $P_b = \mathrm{R}_1(\alpha) \circ P$ is the second candidate.

3. P_c deviates from P_1 before the α^{th} node. The shortest path from s to t is then P_c, the third candidate.

The third-shortest path is now the shortest one in terms of weight of the three candidate paths P_a, P_b, and P_c. Each of these candidates can be computed by FSP on the respective graphs with appropriate source and target nodes.

3.2.4.3 Finding the k^{th}-shortest Path

Computing the third-shortest path can now be generalized to the k^{th}-shortest path. The $(k-1)^{\text{th}}$-shortest path does only introduce three new candidates of the types described for the third-shortest path. Thus computing these three new candidates will still only require $\mathcal{O}(\mathrm{spc}(n,m))$ time. The generalization is done by restricting each of the three deviations P_a, P_b, and P_c to a subgraph of G that only allows for new deviation candidates. In order to do so, the algorithm needs to keep track of the ranges of deviation node indexes where new deviations can branch off from the j^{th} path. These index ranges are stored in sets W_j for $1 \leq j \leq k$ with

$$W_j := \{\mathrm{di}(P_j)+1, |P_j|\} \cup \{\mathrm{di}(P_i) \ : \ \text{for all } P_i \text{ deviated from } P_j.\}\,.$$

The algorithm also needs to keep track of the edges used by the deviation nodes to prevent finding the same paths over and over again. The edges are stored in sets $B_j(\alpha)$ which are all empty at the beginning of the algorithm and get updated over time. KIM then runs on subgraphs $G_j(\alpha)$ where the nodes $v_1^{(j)}, \ldots, v_{\alpha-1}^{(j)}$ and their incident edges are removed.

In order to keep this section short we spare further details on the general case and only show the pseudocode in Algorithm 7. More details and proofs can be found in [40]. As for the previously described algorithms, the pseudocode does not contain any error handling in case a deviation is not found or there exist less than k simple paths from s to t in total.

Algorithm 7: KIM algorithm

```
Input  : undirected weighted graph $G$, nodes $s$ and $t$, the number $k$ of paths to compute
Output : the $k$ shortest $s$–$t$-paths $P_1, \dots, P_k$ sorted by increasing length.
1  $P_1 \leftarrow$ SSSP($G$,$s$,$t$)                // can be stopped as soon as the shortest path is found
2  $W_1 \leftarrow \{1, |P_1|\}$
3  $P \leftarrow \mathrm{FSP}(G, s, t, P_1)$            // compute the second-shortest path
4  $C \leftarrow \{(P, 1, \mathrm{di}(P))\}$            // the candidate list
5  for $i \leftarrow 2, \dots, k$ do
6      $(P_i, j, \alpha) \leftarrow C$.popMin()         // the shortest path among the candidates
7      if $i = k$ then return
8      $W_i \leftarrow \{\alpha + 1, |P_i|\}$
9      $B_j(\alpha) = B_j(\alpha) \cup \left\{v^{(i)}_{\alpha+1}\right\}$   // we assume all $B_j(\alpha)$ to be initialized as empty set
       /* compute deviation from $P_i$ after $\alpha$ */
10     if $\alpha + 1 \neq |P_i|$ then
11         $P_a \leftarrow \mathrm{R}_i(\alpha+1) \circ \mathrm{FSP}(G_i(\alpha+1), v^{(i)}_{\alpha+1}, t, \mathrm{S}_i(\alpha+1))$
12         $C \leftarrow C \cup \{(P_a, i, \mathrm{di}(P_a))\}$
       /* compute deviation from $P_j$ from the $\alpha^{\text{th}}$ node on but before the $\gamma^{\text{th}}$ node */
13     $\gamma \leftarrow \min\{\beta \in W_j \ : \ \alpha + 1 \leq \beta \leq |P_j|\}$
14     $\overline{G} \leftarrow G_j(\alpha)$.removeEdges($\left\{\left\{v^{(j)}_{\alpha}, v\right\} \ : \ v \in B_j(\alpha)\right\}$)
15     $\overline{R} \leftarrow (v^{(j)}_{\alpha}, \dots, v^{(j)}_{\gamma})$
16     $P_b \leftarrow \mathrm{R}_j(\alpha) \circ \mathrm{FSP}(\overline{G}, v^{(j)}_{\alpha}, t, \overline{R})$
17     $C \leftarrow C \cup \{(P_b, j, \mathrm{di}(P_b))\}$
       /* compute deviation from $P_j$ from the $\gamma^{\text{th}}$ node on but before the $\alpha^{\text{th}}$ node */
18     if $\alpha \notin W_j$ then
19         $W_j \leftarrow W_j \cup \{\alpha\}$
20         $\gamma \leftarrow \max\{\beta \in W_j \ : \ \mathrm{di}(P_j) + 1 \leq \beta < \alpha\}$
21         $\overline{G} \leftarrow G_j(\gamma)$.removeEdges($\left\{\left\{v^{(j)}_{\gamma}, v\right\} \ : \ v \in B_j(\gamma)\right\}$)
22         $\overline{R} \leftarrow (v^{(j)}_{\gamma}, \dots, v^{(j)}_{\alpha})$
23         $P_c \leftarrow \mathrm{R}_j(\gamma) \circ \mathrm{FSP}(\overline{G}, v^{(j)}_{\gamma}, t, \overline{R})$
24         $C \leftarrow C \cup \{(P_c, j, \mathrm{di}(P_c))\}$
```

We use $\alpha = \mathrm{di}(P_i)$ as an abbreviation.

3.2.4.4 KIM Cannot be Modified for Directed Graphs

At first glance one might think that the KIM algorithm can also be used for directed graphs if the reverse shortest path tree is used for $T(t)$. Unfortunately this is not possible as the example in Figure 3.7 shows. Here the second-shortest path P_2 with

$$P_2 = (s, a, b, e, f, g, c, d, t)$$

does not satisfy Lemma 3.5 because it uses two edges, (e, f) and (f, g), that are neither in $T(s)$ nor in $T(t)$, with $T(t)$ being the reverse shortest path tree in this case. All paths covered by Lemma 3.5 contain a loop, e.g., a Type 1 path C_1 with

$$C_1 = s \dashrightarrow e \dashrightarrow t = (s, a, b, e, a, b, c, d, t)$$

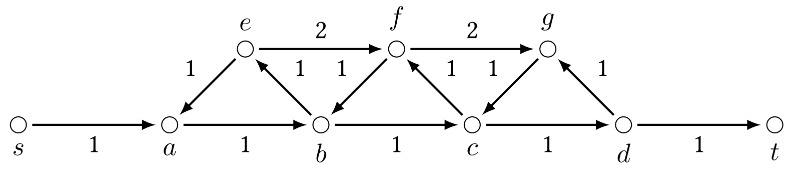


Figure 3.7: Example graph showing why KIM does not work on directed graphs. The second-shortest simple path $(s, a, b, e, f, g, c, d, t)$ does not satisfy Lemma 3.5.

contains the loop (a, b, c, a) and the Type 2 path C_1 with

$$C_2 = s \dashrightarrow e \rightarrow f \dashrightarrow t = (s, a, b, e, f, b, c, d, t)$$

contains the loop (b, e, f, b).

3.2.4.5 Comparison with Yen's Algorithm

The good worst-case complexity of $\mathcal{O}(k \cdot \mathrm{spc}(n, m))$ of the KIM algorithm comes from the fact that for each of the k shortest paths at most three new candidates need to be computed requiring two full SSSP computations each. In contrast, Yen's algorithm computes one candidate for the next shortest path for each node of the k^{th}-shortest path between the deviation node and the target node, which can be up to $\mathcal{O}(n)$ many candidates. However, Yen's algorithm needs only one SSSP computation per candidate path and each SSSP computation can be stopped as soon as the distance to the target node is settled. Being able to stop SSSP computations early can make a huge difference in practical running times as we show in Section 5.1.

So there might be graphs where the k shortest paths are expected to have sufficiently few hops each, in which case Yen's or Feng's algorithm might actually be faster than KIM even though KIM has a much better worst-case complexity. In Section 5.4.3 we present a runtime comparison between KIM and variants of Yen's and Feng's algorithm suggesting that there is only a constant factor in the average-case complexities of KIM and variants of Yen's and Feng's algorithm, respectively. In our experiments, KIM was more than an order of magnitude slower than some of these variants.

4 Average-case Analysis of k-Shortest Path Algorithms

In this chapter we show that the number of hops of the k shortest paths are bounded whp. by $\mathcal{O}(\log n)$ on dense $\mathcal{G}(n,p)$ graphs with an at least logarithmic average degree and by $\mathcal{O}\left(\frac{\log^2 n}{np}\right)$ on sparse $\mathcal{G}(n,p)$ graphs with an at least constant average degree. Using this we can show that the average-case time complexity of Yen's algorithm is $\mathcal{O}(k \cdot \log(n) \cdot \mathrm{spc}(m,n))$ on directed edge-weighted graphs with n nodes and $\Omega(n \log n)$ edges. On sparse $\mathcal{G}(n,p)$ graphs the average-case time complexity is $\mathcal{O}\left(k \cdot \frac{\log^2 n}{np} \cdot \mathrm{spc}(n,m)\right)$. These results also hold for Feng's algorithm.

Average-case complexity of Yen's algorithm:
☞ *Theorems 4.5 and 4.7*

We further show that on directed unweighted $\mathcal{G}(n,p)$ graphs with $\Omega(n \log n)$ edges, Feng's algorithm has an even better average-case complexity of $\mathcal{O}(k\,\mathrm{spc}(n,m))$ for constant values of k. In addition, we provide some evidence that the same should hold on edge-weighted graphs, too.

Average-case complexity of Feng's algorithm:
☞ *Theorem 4.9*

We will utilize some results of Priebe [58] and Meyer [51] who showed upper bounds on the diameter of random graphs and on the number of hops of short paths which we briefly recap in Section 4.2.

4.1 Graph Model and Assumptions

Let $\mathcal{D}(n,p,F)$ be the set of directed, edge-weighted graphs $G = (V,E)$ with $|V| = n$ nodes following the $\mathcal{G}(n,p)$ model, where each edge $(u,v) \in V \times V$ exists with independent probability p, and edge-weights distributed according to a distribution function F. By Chernoff bounds, such graphs have $\Theta\left(n^2 p\right)$ edges whp. Such graphs can contain edges of the form (v,v) called self-loops. However, such edges cannot be part of a shortest path and thus are not relevant in practice.

$\mathcal{G}(n,p)$ graphs:
☞ *Appendix A.2.1.1*

Each edge-weight is drawn at random and independently according to a common distribution function F. We assume the distribution function F to be independent of the graph. In order to get useful results we assume F to have the following properties:

In our experiments we use the uniform random distribution over $[0;1]$.

(A1) F is concentrated on $[0;+\infty)$ and $F(0) = 0$ holds.

(A2) $F'(0)$, the derivative of F also called *density function*, exists and is strictly positive.

Assumption (A1) is needed to make sure that edge weights are non-negative while assumption (A2) allows for a linear Taylor-approximation

$$F(x) = F'(0) \cdot x + o(x) > 0 \qquad \text{for} \qquad x \searrow 0 \tag{4.1}$$

for x close to zero. From the Taylor-approximation (4.1) we can conclude that there exists a constant $\varepsilon_0 < \frac{1}{1.2 \cdot F'(0)}$ such that

$$0.9 \cdot F'(0) \cdot x \leq F(x) \leq 1.1 \cdot F'(0) \cdot x \tag{4.2}$$

holds for all $x < \varepsilon_0$. Priebe uses the bounds in Equation (4.2) in the proofs of Lemma 4.1 and Lemma 4.2 in his PhD thesis [58] and we also use them later in the proof of Lemma 4.6.

4.1.1 Edge Weight Distributions Fulfilling the Assumptions

Assumption (A1) is barely any restriction in the context of edge weights, since it enforces non-negative edge weights and a positive probability for non-zero edge weights. Assumption (A2), however, requires a positive density at zero. This rules out, e.g., uniform distributions over $[a; b]$ with $a, b \in \mathbb{R}$ and $0 \lneq a < b$ but still allows for many probability distributions like

Note that we do not need $F'(0) > 0$ to assure positive probability for zero-weight edges. Instead we need this to allow for a general and easy to use approximation of the probability for edges with small weights.

- Uniform distribution over $[0; b]$ for $b \in \mathbb{R}_{>0}$
- Beta distribution $B(\alpha, \beta)$ with $\alpha, \beta \geq 1$
- Triangular distribution over $[0; b]$
- Exponential distribution

4.2 Recap of Results on Short Path Properties

We utilize some results of Priebe [58]. He showed upper bounds on the diameter of random graphs and on the number of hops on short paths.

Lemma 4.1. [58, Lemma 3.4] Let (G, d) be a directed graph with edge weights $\mathrm{d}(\cdot)$ drawn uniformly at random from $\mathcal{D}(n, p, F)$, where F satisfies (A1) and (A2). If $\frac{np}{\log n}$ is sufficiently large, then the diameter $\mathrm{Diam}(G)$ is bounded by $\mathcal{O}\left(\frac{\log n}{np}\right)$ whp. ◀

Recall that $\mathrm{Diam}(G)$ denotes the diameter in terms of edge-weights and $\mathrm{Diam}(G)$ is defined to be infinity on disconnected graphs.

In order to prove Lemma 4.1 one observes that the diameter $\mathrm{Diam}(G)$ of the graph G can be viewed as

$$\mathrm{Diam}(G) = \max\{\mathrm{Diam}(G, s) \ : \ s \in V\}$$

with $\mathrm{Diam}(G, s) := \max\left\{\mathrm{d}\left(s \dashrightarrow v\right) \ : \ v \in V\right\}$ being the weight-length of the longest of the shortest paths from s to any other node v. We also observe that $\mathrm{Diam}(G, s)$ are identical distributed for all $s \in V$, thus for any x

$$\mathbb{P}[\mathrm{Diam}(G) > x] \leq \sum_{v \in V} \mathbb{P}[\mathrm{Diam}(G, v) > x] = n\,\mathbb{P}[\mathrm{Diam}(G, s) > x]$$

holds. Now it is enough to prove that $\mathrm{Diam}(G, s)$ is bounded by $\mathcal{O}\left(\frac{\log n}{np}\right)$ for a random but fixed node $s \in V$. Instead of showing the bound for the longest path in terms of weight in the shortest path tree rooted in s, we use the so called spanning arborescence rooted in s.

Given a graph $G = (V, E)$, the spanning arborescence $T(s) = \left(\bigcup_i V^{(i)}, E'\right)$, where $V = \bigcup_i V^{(i)}$ is a partition of the set of all nodes V, is a special spanning tree approximating a shortest path tree. Assuming that the adjacency list of each node is sorted by increasing edge weights, the v-rank of a node w is the rank of w in the adjacency list of v.

$T(s)$ is constructed in stages. In the zeroth stage, $V^{(0)}$ consists only of the node s while E' is empty. In the first stage $V^{(1)}$ consists of node u with s-rank 1 and the corresponding edge (s, u) is added to E'. In the i^{th} stage, for all nodes $v_j \in V^{(j)}$ with $0 \leq j < i$, $V^{(i)}$ consists of all nodes $w \in V$ with v_j-rank of $i - j$ as long as the edge (v_j, w) does not close a cycle in $\left(\bigcup_{l=0}^{i} V^{(l)}, E'\right)$ and the corresponding edges (v_j, w) are added to E'. This process ends as soon as $V = \bigcup_i V^{(i)}$ holds or no more nodes can be added.

Note that $V^{(i)}$ can contain multiple nodes for $i \geq 2$.

Frieze and Grimmett [29, Theorem 5.1] showed that the construction of $T(s)$ stops after $\mathcal{O}(\log n)$ stages whp. for graphs in $\mathcal{D}(n, p, F)$.

Lemma 4.1 and the spanning arborescence can then be used to show Lemma 4.2.

Lemma 4.2. Let (G, d) be a directed graph with edge weights $\mathrm{d}(\cdot)$ drawn uniformly at random from $\mathcal{D}(n, p, F)$, where F satisfies (A1) and (A2). If $\frac{np}{\log n}$ is sufficiently large, then all shortest paths in G consist of $\mathcal{O}(\log n)$ edges whp. [58, Lemma 3.10]. ◀

The proof of Lemma 4.2 actually shows that it is highly unlikely for paths with a small weight-length to have many edges which then also holds for the shortest paths under the given conditions. Hence, Lemma 4.2 can also be used for all paths of limited weight-length and not only for the shortest paths between two nodes.

Corollary 4.3. Let (G, d) be a directed graph with edge weights $\mathrm{d}(\cdot)$ drawn uniformly at random from $\mathcal{D}(n, p, F)$, where F satisfies (A1) and (A2) and let $\mathrm{Diam}(G)$ be the diameter of G. If $\frac{np}{\log n}$ is sufficiently large, then paths of weight-length $\mathcal{O}(\mathrm{Diam}(G))$ consist of $\mathcal{O}(\log n)$ edges whp. ◀

With Lemma 4.1 in mind, Corollary 4.3 only holds for paths of constant length.

Meyer [51] further improved the analysis for a special case:

Lemma 4.4. Let (G, d) be a directed graph with edge weights $\mathrm{d}(\cdot)$ drawn uniformly at random from $\mathcal{D}(n, p, F)$, where F is the uniform distribution over $[0; 1]$. There exists a constant $c^* > 1$ such that for $np > 3c^*$ the maximum shortest-path weight-length is bounded by $\mathcal{O}\left(\frac{\log n}{np}\right)$ whp. [51, Theorem 10] ◀

Lemma 4.4 yields the same result as Lemma 4.1 for $p = \Omega\left(\frac{\log n}{n}\right)$. But it also provides an upper bound on weight-length of short paths in sparse $\mathcal{G}(n, p)$ graphs with a constant average-degree. Note that Lemma 4.4 considers the maximum shortest-path length and not the diameter of the graph. For a constant average-degree, $\mathcal{G}(n, p)$ graphs are not strongly connected. However, Karp [39] shows that in $\mathcal{G}(n, p)$ graphs with $np > 1$ the set of nodes $V(s) \subset V$ reachable from a source node s is either $|V(s)| = \mathcal{O}\left(\frac{\log n}{np}\right)$ or $|V(s)| = \Theta(n)$. In contrast to Lemma 4.1, Lemma 4.4 was only shown for uniform random edge-weights over $[0; 1]$.

4.3 Average-Case Analysis of Yen's Algorithm

$\mathrm{spc}(n, m)$ is the average-case complexity of the SSSP algorithm in use.

In this section, we prove in Theorem 4.5 an average-case complexity of Yen's algorithm to be $\mathcal{O}(k \cdot \log(n) \cdot \mathrm{spc}(n, m))$ for graphs with at least logarithmic average-degree using Lemma 4.6. After that, we show an average-case complexity of $\mathcal{O}\left(k \cdot \frac{\log^2 n}{np} \cdot \mathrm{spc}(n, m)\right)$ for graphs with at least constant average-degree, as stated in Theorem 4.7 using Lemma 4.8.

Yen's algorithm: ☞ Section 3.2.1

Theorem 4.5. For random directed graphs $G = (V, E) \in \mathcal{D}(n, p, F)$, with $|V| = n$, $|E| = m$, F satisfying (A1) and (A2), and $p = \Omega\left(\frac{\log n}{n}\right)$, the average-case time complexity of Yen's k-SP algorithm with $k = \mathcal{O}(n)$ is $\mathcal{O}(k \log(n) \cdot \mathrm{spc}(n, m))$, where $\mathrm{spc}(n, m)$ is the worst-case time complexity of computing a single-source shortest-path in G. ◀

Reminder: P_i and the path it deviates from are identical up to the deviation node $\mathrm{dev}(P_i)$, where P_i deviates from P_j. So for all nodes up to $\mathrm{dev}(P_i)$ no new candidates need to be computed. The sets $D_{i,l}$ are defined in Equation (3.1).

Proof. The work done by Yen's algorithm depends on the number of deviations computed. Assume we already have found the ith-shortest path P_i. This means Yen's algorithm has already computed the shortest deviations from each node of each of the the first $i - 1$ paths to the target node t. To compute the $(i + 1)$th-shortest path, Yen's algorithm is now going to compute a deviation path for each node from $\mathrm{dev}(P_i)$ to t. In order to compute one such deviation path at the lth node with $l \geq \mathrm{di}(P_i)$ it removes all edges from the graph that would lead to paths it already found which is at most $\mathcal{O}(m)$ edges. This can be done in $\mathcal{O}(n + m)$ time by iterating over all edges and mark in-edges of nodes $v_1^{(i)}, \ldots, v_l^{(i)}$ on the prefix path of P_i and all out-edges of $v_l^{(i)}$ stored in $D_{i,l}$ as removed. After that it computes the shortest path from $\mathrm{dev}(P_i)$ to t on the remaining graph, which takes $\mathcal{O}(\mathrm{spc}(n, m))$ time.

Lemma 4.6 shows that the number of hops of the ith-shortest path is $\mathcal{O}(\log n)$ whp., for $i \leq k = \mathcal{O}(n)$ proving the bound of $\mathcal{O}(k \log(n) \cdot \mathrm{spc}(n, m))$ for the average-case complexity of Yen's algorithm. □

The bound can be simplified using Dijkstra's algorithm with Fibonacci heaps [27] having a worst-case complexity of $\mathcal{O}(m + n \log n)$. Due to our choice of p, the graph has at least $m = \Omega(n \log n)$ edges whp. and thus the total time complexity to compute one deviation path simplifies to $\mathcal{O}(m)$ whp. Hence, the overall average-case complexity of Yen's algorithm simplifies to $\mathcal{O}(km \log n)$.

Lemma 4.6. For random directed graphs $G = (V, E) \in \mathcal{D}(n, p, F)$ with $|V| = n$, $|E| = m$, $p = \Omega\left(\frac{\log n}{n}\right)$, F satisfying (A1) and (A2), and random but fixed nodes $s, t \in V$ the ith-shortest path from s to t consists of $\mathcal{O}(\log n)$ hops whp. for $i = \mathcal{O}(n)$. ◀

Proof. Using Corollary 4.3, we only need to show that there are enough short paths in terms of weight between the two nodes s and t. Even if the paths we are about to construct, are not the actual k shortest paths, they yield an upper bound on the weight-length for k shortest paths and so Corollary 4.3 applies showing that the k shortest paths also only have $\mathcal{O}(\log n)$ hops whp.

For the purpose of this analysis, we can think about the generation of G as a two step process:

1. Draw two graphs $G_s = (V_s, E_t)$ and $G_t = (V_t, E_t)$ at random and independently from $\mathcal{D}(\frac{n}{2}, p, F)$ with $V_s \cap V_t = \{\}$. We will then choose $s \in V_s$ and $t \in V_t$.

2. For all pairs $(v_s, v_t) \in V_s \times V_t$ add edges (v_s, v_t) independently at random to a new edge set E_{st} with probability p and analogously for $E_{ts} \subset V_t \times V_s$.

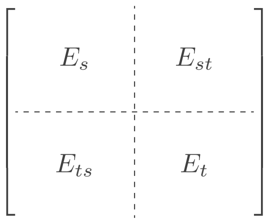


Figure 4.1: Scheme of the adjacency matrix of a $\mathcal{G}(n,p)$ graph G as described in the proof of Lemma 4.6.

The graph $G = (V, E)$ with $V = V_s \cup V_t$ and $E = E_s \cup E_t \cup E_{st} \cup E_{ts}$ is a random graph from $\mathcal{D}(n, p, F)$. Note that p is the same for all three graphs. In order for Corollary 4.3 to hold for the graphs G_s and G_t, the value $\frac{\frac{n}{2}p}{\log \frac{n}{2}}$ needs to be sufficiently large because both graphs only have halve the number of nodes of the full graph G each. Corollary 4.3 then automatically holds for G, too, since $\frac{np}{\log n}$ is approximately twice as big as $\frac{\frac{n}{2}p}{\log \frac{n}{2}}$.

Finally we need to make sure that there are enough short paths from s to t. Due to our lower bound on p, Lemma 4.1 holds for G_s and G_t and thus the diameters $\mathrm{Diam}(G_s)$ and $\mathrm{Diam}(G_t)$ are finite whp. meaning that G_s and G_t are strongly connected. So for each node $u \in G_s$ there is a path from s to u of length at most $\mathrm{Diam}(G_s)$. Additionally, for each node $u \in G_s$ there are $\frac{n}{2}$ potential edges (u, v) with $v \in G_t$. Let I_u be the indicator variable that at least one of these edges exists and has a weight of at most $\overline{d} = \frac{1}{npF'(0)}$. Due to $\overline{d} = o(1)$, the probability for a single such edge to exist follows from (4.1) to be

$$pF\left(\overline{d}\right) = p \cdot \left(F'(0) \cdot \overline{d} + o(\overline{d})\right) = \frac{1}{n} + o\left(\frac{1}{nF'(0)}\right)$$

which is $pF\left(\overline{d}\right) \geq \frac{0.9}{n}$ for big enough n by (4.2). The probability for $I_u = 1$ can now be bounded by

$$\mathbb{P}[I_u = 1] \geq 1 - (1 - \tfrac{0.9}{n})^{\frac{n}{2}} \geq 1 - e^{-\frac{0.9}{2}} > \frac{1}{4}$$

holds and thus $0 < \mathbb{E}[I_u] < 1$ follows. With that it follows by Chernoff bounds that $\sum_{u \in G_s} I_u = \Theta(n)$ holds whp. which implies that there are $\Theta(n)$ paths from s to t with length bounded by $\mathrm{Diam}(G_s) + \mathrm{Diam}(G_t) + \overline{d} \leq 3\,\mathrm{Diam}(G)$. They are not necessarily the actual k shortest paths, but we showed that there are enough such short paths to provide an upper bound on the *weight-length* of the k^{th}-shortest path. Since they are all short enough, Corollary 4.3 holds and each of them has $\mathcal{O}(\log n)$ hops whp. □

Theorem 4.5 only holds for $\mathcal{G}(n,p)$ graphs with $p = \Omega\left(\frac{\log n}{n}\right)$ meaning they have at least an expected logarithmic average degree. For sparser graphs, we show a slightly weaker average-case complexity in Theorem 4.7.

Theorem 4.7. For random directed graphs $G = (V, E) \in \mathcal{D}(n, p, F)$, with $|V| = n$, $|E| = m$, F being the uniform distribution over $[0; 1]$, $p > \frac{3c^*}{n}$ and $p = \mathcal{O}\left(\frac{\log n}{n}\right)$, the average-case complexity of Yen's k-shortest path algorithm with $k = \mathcal{O}(n)$ is $\mathcal{O}\left(k \cdot \frac{\log^2 n}{np} \cdot \mathrm{spc}(n, m)\right)$, where $\mathrm{spc}(n, m)$ is the worst-case complexity of computing a single-source shortest-path in G. ◀

The constant c^* is the same as in Lemma 4.4.

Proof. The proof works analogously to the proof of Theorem 4.5. Lemma 4.8 shows that the number of hops on each of the k shortest paths can be bounded by $\mathcal{O}\left(\frac{\log^2 n}{np}\right)$ whp., so computing a shortest path for each node on each of the k shortest paths takes at most $\mathcal{O}\left(k \cdot \frac{\log^2 n}{np} \cdot \mathrm{spc}(n,m)\right)$ time in total whp. □

We believe that Theorem 4.7 holds for more general edge-weight distributions. But in order to prove this, all results Theorem 4.7 is based on need to be proven for more general edge-weight distributions first.

For $p = \Theta\left(\frac{\log n}{n}\right)$ both Theorem 4.5 and Theorem 4.7 yield the same average-case complexity of Yen's algorithm. But keep in mind that Theorem 4.7 is only proven for uniform random edge-weights over $[0;1]$ while Theorem 4.5 holds for more general edge-weight distributions.

Lemma 4.8. For random directed graphs $G = (V,E) \in \mathcal{D}(n,p,F)$ with $|V| = n$, $|E| = m$, $p > \frac{3c^*}{n}$, $p = \mathcal{O}\left(\frac{\log n}{n}\right)$, F being the uniform distribution over $[0;1]$, and random but fixed nodes $s,t \in V$ the i^{th}-shortest path from s to t consists of $\mathcal{O}\left(\frac{\log^2 n}{np}\right)$ hops whp. for $i = \mathcal{O}(n)$ if t is reachable from s. ◀

Proof. Just like in the proof of Lemma 4.6, we think of G as follows.

1. Draw two graphs $G_s = (V_s, E_t)$ and $G_t = (V_t, E_t)$ at random and independently from $\mathcal{D}(\frac{n}{2}, p, F)$. We will then choose $s \in V_s$ and $t \in V_t$.
2. For all pairs $(v_s, v_t) \in V_s \times V_t$ add edges (v_s, v_t) independently at random to a new edge set E_{st} with probability p and analogously for $E_{ts} \subset V_t \times V_s$.

The graph $G = (V,E)$ with $V = V_s \cup V_t$ and $E = E_s \cup E_t \cup E_{st} \cup E_{ts}$ is a random graph from $\mathcal{D}(n,p,F)$.

According to Karp [39], the set of nodes $V(s) \subset V$ reachable from a source node s is either small with $|V(s)| = \mathcal{O}\left(\frac{\log n}{np}\right)$ or giant with $|V(s)| = \Theta(n)$. We assume t to be reachable from s, so $t \in V(s)$. If $V(s)$ is small, the number of hops on every simple path is bounded by $|V(s)|$. In this case we have nothing more to show. So now we assume $V(s)$ to be giant.

As in the proof of Lemma 4.6, we show that there are enough s–t-paths. From Lemma 4.4 we know, that the weight-length of the shortest paths from s to any node $u \in V_s \cap V(s)$ can be bounded by $\mathcal{O}\left(\frac{\log n}{np}\right)$ whp. The same holds for the weight-length of the shortest paths from any node $v \in V_t \cap V(s)$ to t. The proofs by Karp [39] can also be used to show that $V_s \cap V(s)$ as well as $V_t \cap V(s)$ have both $\Theta(n)$ nodes. Thus, by Chernoff bounds, we know that there are $\Theta(n)$ edges $(u,v) \in (V_s \cap V(s)) \times (V_t \cap V(s))$ whp. Unlike in the proof of Lemma 4.6, we cannot only use short edges due to the lower bound on p. So instead we use edges of any weight. Since we assume F to be the uniform distribution over $[0;1]$, an edge can have at most a weight of 1.

$a \dashrightarrow b$ denotes a shortest path from node a to node b in terms of edge-weights.

Now we know that whp. G contains $\Theta(n)$ paths of the form $s \dashrightarrow u \to v \dashrightarrow t$ with $u \in V_s$ and $v \in V_t$. The weight-length of these paths is bounded by

$$\mathrm{d}(s \dashrightarrow u) + \mathrm{d}(u,v) + \mathrm{d}(v \dashrightarrow t) = \mathcal{O}\left(\frac{\log n}{np}\right) + 1 + \mathcal{O}\left(\frac{\log n}{np}\right) = \mathcal{O}\left(\frac{\log n}{np}\right).$$

Recall that, like for the proof of Lemma 4.6, these are not necessarily the k shortest paths. But they give us an upper bound on the length of the k shortest paths which we now can use to get an upper bound on the number of hops on the k shortest paths.

The paths we just constructed are not short enough for Corollary 4.3 to apply. So we split up the paths into $\Theta\left(\frac{\log n}{np}\right)$ subpaths such that each subpath has a small enough constant weight-length. For each of these subpaths Corollary 4.3 does now apply showing that each subpath consists of $\mathcal{O}(\log n)$ hops whp. So in total all paths with a weight-length of $\mathcal{O}\left(\frac{\log n}{np}\right)$ have $\mathcal{O}\left(\frac{\log^2 n}{np}\right)$ hops whp. □

4.4 Average-Case Analysis of Feng's Algorithm

Feng's algorithm: ☞ *Section 3.2.2*

Feng's algorithm is essentially a heuristic on top of Yen's algorithm, which does not improve the worst-case complexity, but could drastically reduce the computation time since it prunes a lot of nodes from the graph the shortest paths are computed on. If the yellow graphs computed by Feng's algorithm would consist of all nodes in the graph, then Feng's algorithm would be identical to Yen's algorithm. Thus Theorems 4.5 and 4.7 also hold for Feng's algorithm. Currently, we cannot show a better average-case complexity for the weighted case, but in the *unweighted* case we prove in Section 4.4.1 that Feng's algorithm has an even better average-case complexity for computing the k^{th}-shortest path for $k = \Theta(1)$. Although we cannot prove the same in the weighted case, we provide some evidence that the same results should hold in Section 4.4.2.

4.4.1 Theoretical Analysis on Unweighted $\mathcal{G}(n, p)$ Graphs

Theorem 4.9. For random directed, *unweighted* graphs $G = (V, E)$ following the $\mathcal{G}(n, p)$ model, with $|V| = n$, $|E| = m$, and $p = \Omega\left(\frac{\log n}{n}\right)$, the average-case complexity of Feng's algorithm for computing the k^{th}-shortest path is $\mathcal{O}(m)$ for $k = \Theta(1)$. ◀

Proof. Feng's algorithm consists of three main steps.

i) Computing the reverse shortest path tree $\Gamma = \Gamma(t)$ from all nodes to t: The computation happens only once and the shortest path from s to t is part of the result. The reverse shortest path tree can be computed in $\mathcal{O}(n + m)$ time by a BFS since we assume G to be unweighted.

ii) Computing the yellow graphs: This is done on the i^{th}-shortest path P_i for each node between $\operatorname{dev}(P_i)$ and t. The yellow graph $Y_j^{(i)}$ can be decomposed as $\Gamma\left(v_j^{(i)}\right) \cup Y_{j-1}^{(i)}$, where $\Gamma\left(v_j^{(i)}\right)$ is the subtree of Γ hanging from $v_j^{(i)}$, meaning that we do not need to recompute $Y_{j-1}^{(i)}$. This can be done in $\mathcal{O}\left(m + \sum_{j=1}^{|P_i|-1} \left|Y_j^{(i)}\right|\right)$ time for the $(i+1)^{\text{th}}$-shortest path.

iii) Computing the shortest path from the deviation node to t in the yellow graphs: In the unweighted case, this can be done using BFS in $\mathcal{O}\left(m + \sum_{j=1}^{|P_i|-1} \left|Y_j^{(i)}\right|\right)$ time for the $(i+1)^{\text{th}}$-shortest path.

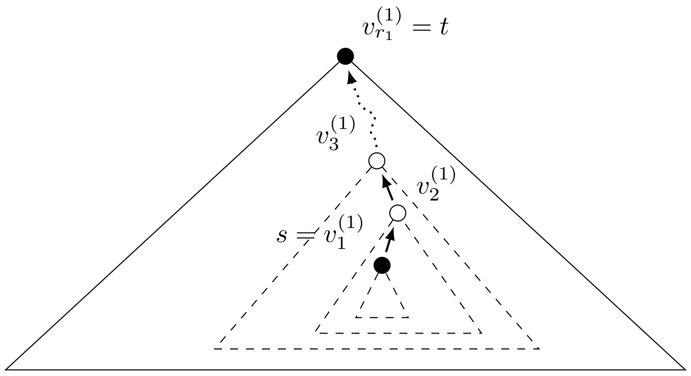


Figure 4.2: *Visualization of the shortest path in the reverse BFS tree Γ including the nested subtrees $\Gamma(s)$, $\Gamma\left(v_2^{(1)}\right)$ and $\Gamma\left(v_3^{(1)}\right)$.*

Thus the overall complexity for Feng's algorithm to compute the $(i+1)^{\text{th}}$-shortest path, for $i < k$, is $\mathcal{O}\left(m + \sum_{j=1}^{|P_i|-1} \left|Y_j^{(i)}\right|\right)$. We show in Lemma 4.10 that

$$\sum_{j=1}^{|P_i|-1} \left|Y_j^{(i)}\right| = \mathcal{O}(n+m)$$

holds for $i < k = \Theta(1)$ whp., which proves the average complexity of Feng's algorithm to be $\mathcal{O}(m)$. □

We published a special case of this Lemma in [62]. The proof we provided was lacking some important details. But as we show here, the Lemma holds even in a more general version.

Lemma 4.10. For random directed, *unweighted* graphs $G = (V, E)$ following the $\mathcal{G}(n,p)$ model, with $np > 2\log n$ and $|E| = m$, the combined size of the yellow graphs is $\sum_{j=1}^{|P_i|-1} \left|Y_j^{(i)}\right| = \mathcal{O}(n+m)$ whp. for the i^{th}-shortest path with $i < k$ and $k = \Theta(1)$. ◀

Proof. First, we show that $\sum_{j=1}^{l} \left|Y_j^{(1)}\right| = \mathcal{O}(n+m)$ holds for the second-shortest path. Then we use this result to show that it also holds for a constant k.

Let $Y_j^{(i)} = \left(V_j^{(i)}, E_j^{(i)}\right)$ and $\left|Y_j^{(i)}\right| = \left|V_j^{(i)}\right| + \left|E_j^{(i)}\right|$. We show that $\sum_{j=1}^{|P_1|} \left|V_j^{(1)}\right| = \mathcal{O}(n)$ holds whp. From this we argue that $\sum_{j=1}^{|P_1|} \left|E_j^{(1)}\right| = \mathcal{O}(m)$ also holds whp.

In the case of an unweighted graph, the yellow graphs $Y_j^{(1)}$ are the subtrees of the reverse BFS tree $\Gamma = \Gamma(t)$ rooted at t where all edges are oriented *towards* the root.

Consider the j^{th} node $v_j^{(1)}$ on the shortest path $P_1 = \left(v_1^{(1)}, \dots, v_{r_1}^{(1)}\right)$ from s to t where $r_1 = |P_1|$ is the number of nodes of P_1. Notice that the index of the nodes of P_1 starts at the source, thus the node $v_j^{(1)}$ is in the $(r_1 - j)^{\text{th}}$ layer of the reverse BFS tree Γ and $v_{r_1} = t$ is the only node in layer 0.

Figure 4.3 shows a schematic of this definition.

Let $\hat{\Gamma}_i$ and $\check{\Gamma}_i$ be defined as

$$\hat{\Gamma}_i := \bigcup_{j<i} \Gamma_j \qquad \text{and} \qquad \check{\Gamma}_i := \bigcup_{j>i} \Gamma_j = V \setminus (\hat{\Gamma}_i \cup \Gamma_i). \tag{4.3}$$

As we will se from Lemma 4.11 that the size of the i^{th} layer is at least $|\Gamma_i| = \Omega\left((np)^i\right)$ whp. for all $i \le i_0$, where i_0 is the biggest index such that $\left|\hat{\Gamma}_{i_0}\right| = \left|\bigcup_{j=0}^{i_0-1} \Gamma_j\right| < \sqrt{n}$ holds. So the same is true for $\left|\hat{\Gamma}_i\right|$ and $\left|V \setminus \hat{\Gamma}_i\right| = \Theta(n)$ follows for all $i \le i_0$.

i_0 is calculated as such during the proof of Lemma 4.11.

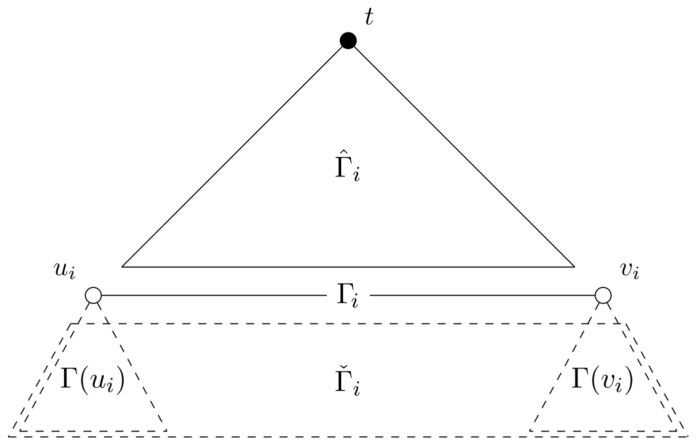


Figure 4.3: *A diagram of the situation in a BFS tree rooted in a node t looking at a layer i where $u_i \in \Gamma_i$ is the first node visited in the layer and $v_i \in \Gamma_i$ is the last one.*

Given that the sizes of subtrees $\Gamma(u)$, $\Gamma(v)$ hanging from nodes $u, v \in \Gamma_i$, with $i < i_0$, differ at most by a factor of $\Theta(np)$, as will be shown in Lemma 4.12, the total number of nodes in a single subtrees hanging from a node in layer i can be bounded as follows. Let X_i be the size of the biggest subtree hanging from a node in layer i. For $i < i_0$ a total of $\Theta(n)$ nodes are distributed to the subtrees hanging from the nodes in layer i. The smallest subtree must have $\Theta\left(\frac{X_i}{np}\right)$ nodes by Lemma 4.12. Assuming that all subtrees except for the biggest ones are as small as possible, we can bound the size of the biggest subtree by

$$\Theta(n) = X_i + \Omega\big((np)^i\big) \cdot \Theta\left(\frac{X_i}{np}\right) = X_i \cdot \Omega\big((np)^{i-1}\big)\,. \tag{4.4}$$

From (4.4) it now follows that

$$X_i = \mathcal{O}\left(\frac{n}{(np)^{i-1}}\right) \tag{4.5}$$

is an upper bound of the size of the subtrees hanging from nodes in layer $i < i_0$.

Now we split up the sum of all subtrees along the shortest path $P_1 = \left(v_1^{(1)}, \ldots, v_{r1}^{(1)}\right)$ into three parts.

- The whole BFS tree as it is the subtree hanging from the target node. It trivially contains $\mathcal{O}(n)$ nodes.
- Big trees hanging from nodes that are between one and $i_0 - 1$ hops away from the target node t. Even if we assume that all these trees are always as big as possible their size is bounded as in (4.5).
- Small trees hanging from nodes at least i_0 hops away from the target node t. Each of these trees is smaller than the smallest of the big trees. We do not know an exact bound but we can just say that they have the same size as the smallest of the big trees which is $\mathcal{O}\left(\frac{n}{(np)^{i_0-1}}\right)$. We can assume that $i_0 \geq 2$ holds. Otherwise $np > \sqrt{n}$ would hold whp. in which case the BFS tree has only a constant depth and the sum of the sizes of a constant number of subtrees is clearly bounded by

Recall that the target node t is the root of the BFS tree since the BFS tree is a reverse shortest path tree in our case.

The Lemma assumes $np > 2\log n$.

$\mathcal{O}(n)$. So assuming the smallest i_0 and choosing np as small as possible, the sizes of the small subtrees are bounded by $\mathcal{O}\left(\frac{n}{\log n}\right)$. Since the diameter of the given $\mathcal{G}(n,p)$ graphs are $\log n$ whp. by Lemma 4.2, we have at most that many small subtrees.

Now we can bound the sum of the sizes of all subtrees as follows:

$$\sum_{i=1}^{r_1}\left|\Gamma\left(v_i^{(1)}\right)\right| = \underbrace{\sum_{i=1}^{r_1-i_0}\left|\Gamma\left(v_i^{(1)}\right)\right|}_{\text{small subtrees}} + \underbrace{\sum_{i=r_1-i_0+1}^{r_1-1}\left|\Gamma\left(v_i^{(1)}\right)\right|}_{\text{big subtrees}} + \left|\Gamma\left(v_{r_1}^{(1)}\right)\right|$$

$$= \mathcal{O}(\log n)\cdot\mathcal{O}\left(\frac{n}{\log n}\right) + n\sum_{i=1}^{i_0}\mathcal{O}\left(\frac{1}{(np)^{i-1}}\right) + \mathcal{O}(n) = \mathcal{O}(n)$$

This concludes the proof for the second-shortest path.

For the k^{th}-shortest path we look at the edge (u,v) used to deviate from the $(k-1)^{\text{th}}$-shortest path. Such a deviation edge cannot follow a path in the BFS tree and so the subtree $\Gamma(u)$ is not included in $\Gamma(v)$. Both u and v could be on layer 1 and thus $|\Gamma(u)| = \Theta(n)$ and $|\Gamma(v)| = \Theta(n)$ can hold simultaneously. Since only one such edge

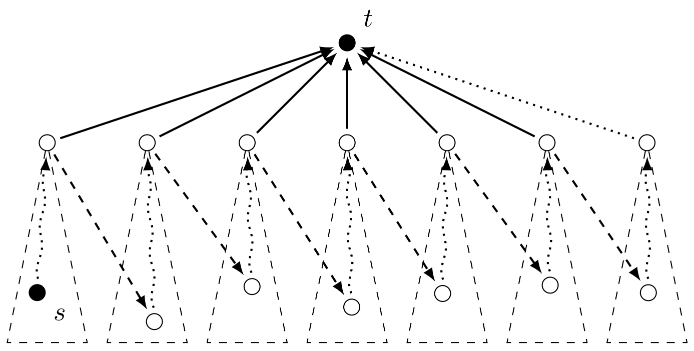


Figure 4.4: Visualization of k^{th}-shortest path in the reverse BFS tree. The k^{th}-shortest path follows the dotted and dashed edges and subpaths. The solid and dotted edges and subpaths are part of the BFS tree, while dashed edges are not part of the BFS tree.

is introduced when going from the $(k-1)^{\text{th}}$ to the k^{th}-shortest path, the k^{th}-shortest path can only have k such edges. All other edges follow the BFS tree and the sizes of the hanging trees sum up to at most $\mathcal{O}(n)$, just as for the second-shortest path, totaling up to

$$\sum_{i=1}^{r_k}\left|\Gamma\left(v_i^{(k)}\right)\right| = \mathcal{O}(n) + k\cdot\mathcal{O}(n) = \mathcal{O}(n)\,.$$

Now that we have bounded the combined number of nodes in the yellow graphs, we can bound the total number of edges in all yellow graphs combined by $\mathcal{O}\left(n^2p\right) = \mathcal{O}(m)$ whp. since the number of adjacent edges of any node is bounded by Chernoff bounds to be at most $\mathcal{O}(np)$. $\square$

Since the average-case complexity of Feng's algorithm depends on the sizes of the yellow subgraphs, is not sufficient to show that there exist enough paths from s to t with certain properties, like we did for Yen's algorithm. Instead one needs to argue that the combined sizes of all yellow subtrees is small enough. Since s–t-paths could jump multiple times between two yellow subtrees, Lemma 4.10 only holds for constant values of k.

Lemma 4.11. Let $G = (V, E)$ be a random directed, *unweighted* graph following the $\mathcal{G}(n, p)$ model, with $np \geq c \log n$ for a constant $c > 2$. Given a random but fixed node t let Γ_i be the set of nodes with a minimal hop-distance i to t.
Then the following holds whp.:

Minimal hop-distance of i also means that the nodes are in the i^{th} BFS layer.

a) $|\Gamma_i| = \Omega\big((np)^i\big)$ for all $1 \leq i \leq i_0 = \frac{1}{2}\frac{\log n}{\log(np)}$ [16, Lemma 8]

b) $|\Gamma_i| \geq \sqrt{n}$ for $i_0 < i < i_1$ where i_1 is the smallest index such that

$$\left|\bigcup_{i=0}^{i_1} \Gamma_i\right| > n - 2\sqrt{n} \tag{4.6}$$

◀

Proof. Part a) can be shown by induction over i. Starting with $i = 1$, Γ_1 consists of all nodes u such that (u, t) is an edge in G. The expected number of such edges is $(n-1)p$. By Chernoff bounds we show that $|\Gamma_1| \geq c_1(np)^1$ holds whp. for $c_1 = 1 - \sqrt{\frac{2}{c}} - \varepsilon$ with $\varepsilon > 0$.

Here we use the Chernoff bounds as stated in (2.1).

$$\begin{aligned}
\mathbb{P}[|\Gamma_1| \leq (c_1+\lambda)(n-1)p] &= \mathbb{P}[|\Gamma_1| \leq (1-(1-c_1-\lambda))(n-1)p] \\
&\leq \exp\left(-\frac{(1-c_1-\lambda)^2(n-1)p}{2}\right) \\
&\leq \exp\left(-\frac{(1-c_1-\lambda)^2 c(n-1)\log n}{2n}\right) \\
&= \exp\left(-\log n \cdot \left(\sqrt{\frac{2}{c}} + \varepsilon - \lambda\right)^2 \cdot \frac{c(n-1)}{2n}\right) \\
&= n^{-\frac{(1-c_1-\lambda)^2 c(n-1)}{2n}} \\
&= o\big(n^{-1}\big)
\end{aligned} \tag{4.7}$$

Equation (4.7) is true for all positive λ and ε such that

We use the parameter λ to find a constant factor such that the bounds we want to show hold.

$$0 < \lambda < \varepsilon - \sqrt{\frac{2}{c}}\left(\sqrt{\frac{n}{n-1}} - 1\right) \quad \text{and} \quad 0 < \varepsilon < 1 - \sqrt{\frac{2}{c}}$$

holds. So $|\Gamma_1| \geq (c_1+\lambda)np$ holds with probability at least $1 - o\big(n^{-1}\big)$.

Let $N_i = \bigcup_{j=0}^{i} \Gamma_j$ be the set of nodes with a hop-distance to t of at most i. For $i > 1$, the number of edges (u, v) with $u \in V \setminus N_{i-1}$ and $v \in \Gamma_{i-1}$ is distributed according to $\textsc{Bin}[t_i, p]$ with $t_i = |\Gamma_{i-1}| \cdot (n - |N_{i-1}|)$. But some edges could have the same source

Keep in mind that we construct a tree where all edges point to the root.

node, meaning that the size of Γ_i is at most the number of edges from $V \setminus N_{i-1}$ to Γ_{i-1} and could be smaller. Given Γ_{i-1} and Γ_i, the number of edges from Γ_i to Γ_{i-1} would be distributed following $\textsc{Bin}[|\Gamma_{i-1}| \cdot |\Gamma_i|, p]$ if Γ_i were independent from Γ_{i-1}. So $|\Gamma_i|$ stochastically dominates a random variable $X_i \sim \textsc{Bin}[t'_i, p]$ with

$$t'_i < |\Gamma_{i-1}| \cdot (n - |N_{i-1}|) - |\Gamma_{i-1}| \cdot |\Gamma_i| = |\Gamma_{i-1}| \cdot (n - |N_i|).$$

Let now $t'_i = |\Gamma_{i-1}| (n - \sqrt{n})$. Then $|\Gamma_i|$ stochastically dominates X_i as long as

$$|N_i| < \sqrt{n} \tag{4.8}$$

holds and we have

$$\mathbb{P}\Big[|\Gamma_i| \le \mathbb{E}[X_i] - \sigma\sqrt{\mathbb{E}[X_i]}\Big] \le \mathbb{P}\Big[X_i \le \mathbb{E}[X_i] - \sigma\sqrt{\mathbb{E}[X_i]}\Big] \le \exp\left(-\frac{\sigma^2}{2}\right)$$

for $\mathbb{E}[X_i] = t'_i p$ and $\sigma < \sqrt{t'_i p}$. This implies that

$$\begin{aligned}
|\Gamma_2| &\ge |\Gamma_1| \cdot \left(n - \sqrt{n}\right) p - \sigma\sqrt{|\Gamma_1| \cdot \left(n - \sqrt{n}\right) p} \\
&\ge (c_1 + \lambda) np \left(n - \sqrt{n}\right) p - \sigma\sqrt{(c_1 + \lambda) np \left(n - \sqrt{n}\right) p} \\
&= (c_1 + \lambda)(np)^2 \left(1 - n^{-\frac{1}{2}}\right) - \sigma\sqrt{(c_1 + \lambda)(np)^2(1 - n^{-\frac{1}{2}})} \\
&= (c_1 + \lambda)(np)^2 \left(1 - n^{-\frac{1}{2}} - \sigma\frac{\sqrt{1 - n^{-\frac{1}{2}}}}{\sqrt{(c_1 + \lambda)(np)^2}}\right) \\
&\ge (c_1 + \lambda)(np)^2 \left(1 - n^{-\frac{1}{2}} - \frac{\sigma}{\sqrt{c_1 (np)^2}}\right)
\end{aligned}$$

holds with probability at least

$$\left(1 - o\left(n^{-1}\right)\right)\left(1 - \exp\left(-\frac{\sigma^2}{2}\right)\right) = 1 - o\left(n^{-1}\right) - \exp\left(-\frac{\sigma^2}{2}\right).$$

By induction on $i > 2$ we get

$$|\Gamma_i| \ge (c_1 + \lambda)(np)^i \prod_{j=2}^{i} \left(1 - n^{-\frac{1}{2}} - \frac{\sigma}{\sqrt{c_1 (np)^j}}\right)$$

with probability at least

$$1 - o\left(n^{-1}\right) - i \exp\left(-\frac{\sigma^2}{2}\right).$$

Using $\sigma = \sqrt{5 \log n}$ and with $i < \log n$ finally results in

$$\begin{aligned}
|\Gamma_i| &\geq (c_1 + \lambda)(np)^i \prod_{j=2}^{i} \left(1 - n^{-\frac{1}{2}} - \frac{\sigma}{\sqrt{c_1 (np)^j}}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - in^{-\frac{1}{2}} - \frac{\sigma}{\sqrt{c_1}} \sum_{j=2}^{i} \sqrt{(np)^{-j}}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - in^{-\frac{1}{2}} - \frac{\sigma}{np\sqrt{c_1}} \sum_{j=2}^{i} \sqrt{(np)^{-j-2}}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - in^{-\frac{1}{2}} - \frac{\sigma}{np\sqrt{c_1}} \sum_{j=0}^{\infty} \sqrt{(np)^{-j}}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - in^{-\frac{1}{2}} - \frac{\sigma}{np\sqrt{c_1}} \cdot \frac{1}{1 - (np)^{-\frac{1}{2}}}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - in^{-\frac{1}{2}} - \frac{\sigma}{np\sqrt{c_1}} \cdot \frac{\sqrt{c \log n}}{\sqrt{c \log n} - 1}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - \frac{i}{\sqrt{n}} - \frac{\sqrt{5 \log n}}{c \log(n) \sqrt{c_1}} \cdot \frac{\sqrt{c \log n}}{\sqrt{c \log n} - 1}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - \frac{\log n}{\sqrt{n}} - \sqrt{\frac{5}{c \cdot c_1}} \cdot \frac{1}{\sqrt{c \log n} - 1}\right) \\
&\geq (c_1 + \lambda)(np)^i \left(1 - \mathcal{O}\left(\frac{1}{\sqrt{\log n}}\right)\right) \\
&\geq c_1 (np)^i
\end{aligned}$$

for n large enough. As mentioned before, $|\Gamma_i| \geq c_1(np)^i$ holds for all i with $|N_i| < \sqrt{n}$ (4.8). Let i_0 be the maximal value for i such that

$$\sqrt{n} \geq |N_{i_0}| = \left|\bigcup_{j=0}^{i_0} \Gamma_j\right| = \sum_{j=0}^{i_0} |\Gamma_j| \geq \sum_{j=0}^{i_0} c_1 (np)^j = c_1 \cdot \frac{(np)^{i_0+1} - 1}{np - 1}$$

holds. From this we calculate i_0 to be as follows:

Note that $\frac{1}{c_1} > 1$ is a constant and for n big enough $\left(1 - \frac{1}{np}\right) \cdot \frac{1}{c_1} > 1$ holds.

$$\begin{aligned}
\frac{\log\left(\frac{\sqrt{n} \cdot (np-1)}{c_1} + 1\right)}{\log(np)} - 1 &\geq \frac{\log\left(\frac{\sqrt{n} \cdot (np-1)}{c_1}\right)}{\log(np)} - 1 \\
&\geq \frac{\log\left(\sqrt{n} \cdot np \cdot \left(1 - \frac{1}{np}\right) \cdot \frac{1}{c_1}\right)}{\log(np)} - 1 \\
&= \frac{\frac{1}{2}\log(n) + \log(np) + \log\left(\left(1 - \frac{1}{np}\right) \cdot \frac{1}{c_1}\right)}{\log(np)} - 1 \\
&\geq \frac{1}{2} \frac{\log n}{\log(np)} + 1 + o(1) - 1 \geq \frac{1}{2} \frac{\log n}{\log(np)} = i_0
\end{aligned}$$

This concludes Part a).

We show Part b) by contraposition. Assume $|\Gamma_i| < \sqrt{n}$ for one i with $i_0 < i < i_1$. The BFS layers partition V into three disjoint sets

$\hat{\Gamma}_i$ and $\check{\Gamma}_i$ are defined as in (4.3). See Figure 4.3 for a diagram of the partition.

$$\Gamma_i, \qquad \hat{\Gamma}_i = \bigcup_{j=0}^{i-1} \Gamma_j, \qquad \check{\Gamma}_i = V \setminus (\Gamma_i \cup \hat{\Gamma}_i), \tag{4.9}$$

such that there exist no edges (u, v) with $u \in \hat{\Gamma}_i$ and $v \in \check{\Gamma}_i$. By the definition of i_1 we know $\left|\check{\Gamma}_i\right| > 2\sqrt{n}$. The probability for such a partition can be bounded as follows using Stirling's approximation:

i_1 was defined in (4.6) as the smallest index such that $\hat{\Gamma}_{i_1} > n - 2\sqrt{n}$ holds.

$$\begin{aligned}
&\binom{n}{|\Gamma_i|} \cdot \binom{n - |\Gamma_i|}{\left|\check{\Gamma}_i\right|} \cdot \binom{n - |\Gamma_i| - \left|\check{\Gamma}_i\right|}{\left|\hat{\Gamma}_i\right|} \cdot (1-p)^{\left|\check{\Gamma}_i\right| \cdot \left|\hat{\Gamma}_i\right|} \\
&= \binom{n}{|\Gamma_i|} \cdot \binom{n - |\Gamma_i|}{\left|\check{\Gamma}_i\right|} \cdot (1-p)^{\left|\check{\Gamma}_i\right| \cdot \left|\hat{\Gamma}_i\right|} \\
&\le \binom{n}{|\Gamma_i|} \cdot \binom{n}{\left|\check{\Gamma}_i\right|} \cdot (1-p)^{\left|\check{\Gamma}_i\right| \cdot \left|\hat{\Gamma}_i\right|} \\
&\le \binom{n}{\sqrt{n}} \cdot \binom{n}{\left|\check{\Gamma}_i\right|} \cdot (1-p)^{\left|\check{\Gamma}_i\right| \cdot (n - \sqrt{n} - \left|\check{\Gamma}_i\right|)} \\
&\le \left(\frac{ne}{\sqrt{n}}\right)^{\sqrt{n}} \cdot \left(\frac{ne}{\left|\check{\Gamma}_i\right|}\right)^{\left|\check{\Gamma}_i\right|} \left(1 - \frac{c \log n}{n}\right)^{n\left|\check{\Gamma}_i\right|\left(1 - \frac{\sqrt{n}}{n} - \frac{\left|\check{\Gamma}_i\right|}{n}\right)} \\
&\le \left(\frac{ne}{\sqrt{n}}\right)^{\sqrt{n}} \cdot \left(e\sqrt{n}\right)^{\left|\check{\Gamma}_i\right|} \cdot e^{-c\log(n)\left|\check{\Gamma}_i\right|\left(1 - \frac{\sqrt{n}}{n} - \frac{\left|\check{\Gamma}_i\right|}{n}\right)} \\
&\le \exp\left(\left(\frac{1}{2}\log(n) + 1\right)\sqrt{n} + \left(\frac{1}{2}\log(n) + 1\right)\left|\check{\Gamma}_i\right| - c\log(n)\left|\check{\Gamma}_i\right|\left(1 - \frac{3}{\sqrt{n}}\right)\right) \\
&= \exp\left(\left(\frac{1}{2}\log(n) + 1\right) \cdot \left(\sqrt{n} + \left|\check{\Gamma}_i\right|\right) - c\log(n)\left|\check{\Gamma}_i\right|\left(1 - \frac{3}{\sqrt{n}}\right)\right) \\
&\le \exp\left(\frac{3}{2}\left|\check{\Gamma}_i\right|\left(\frac{1}{2}\log(n) + 1\right) - c\log(n)\left|\check{\Gamma}_i\right|\left(1 - \frac{3}{\sqrt{n}}\right)\right) \\
&= \exp\left(\left|\check{\Gamma}_i\right|\left(\frac{3}{2}\left(\frac{1}{2}\log(n) + 1\right) - c\log(n)\left(1 - \frac{3}{\sqrt{n}}\right)\right)\right) \\
&= \exp\left(\left|\check{\Gamma}_i\right|\left(\frac{3}{4}\log(n) + \frac{3}{2} - c\log(n)\left(1 - \frac{3}{\sqrt{n}}\right)\right)\right) \\
&= \exp\left(\left|\check{\Gamma}_i\right|\left(\frac{3}{2} + \log(n) \cdot \left(\frac{3}{4} - c + \frac{3c}{\sqrt{n}}\right)\right)\right) \\
&= \left(e^{\frac{3}{2}} \cdot n^{\frac{3}{4} - c + \frac{3c}{\sqrt{n}}}\right)^{\left|\check{\Gamma}_i\right|}
\end{aligned} \tag{4.10}$$

With $c > 2$ and $\frac{3c}{\sqrt{n}} < 1$ we find the exponent to be $\frac{3}{4} - c + \frac{3c}{\sqrt{n}} < -1$ for all $n > 576$. Thus (4.10) simplifies to

$$\left(e^{\frac{3}{2}} \cdot n^{\frac{3}{4} - c + \frac{3c}{\sqrt{n}}}\right)^{\left|\check{\Gamma}_i\right|} < \left(\frac{e^{\frac{3}{2}}}{n}\right)^{\left|\check{\Gamma}_i\right|} = o\left(n^{-\left|\check{\Gamma}_i\right|}\right)$$

for n big enough. Summing over all possible values for $\left|\check{\Gamma}_i\right| > 2\sqrt{n}$ we end up with a total probability bound of

$$\sum_{x=2\sqrt{n}}^{n} o\left(n^{-x}\right) < n \cdot o\left(n^{-2\sqrt{n}}\right) = o\left(n^{-2\sqrt{n}+1}\right)$$

for a partition described in (4.9) to exist. In turn this means that Γ_i contains at least $\sqrt{n}$ with probability $1 - o\left(n^{-2\sqrt{n}+1}\right)$ for n big enough. □

Lemma 4.12. Let $G = (V, E)$ be a random directed, *unweighted* graph following the $\mathcal{G}(n, p)$ model, with $np \geq c \log n$ for a constant $c > 2$. Let Γ_i be the i^{th} layer of the reverse BFS tree $\Gamma = \Gamma(t)$ rooted in a random but fixed node t. Then for all nodes $u, v \in \Gamma_i$ the sizes of subtrees $\Gamma(u)$ and $\Gamma(v)$ of Γ differ by a factor of at most $\mathcal{O}(np)$. ◀

Proof. During construction of the reverse BFS tree Γ, the first nodes visited in layer i are likely to have more children than the last nodes in this layer, since nodes from layer $i+1$ with multiple out-edges to nodes in layer i will connect to the first node discovering them. Let i_2 be the first layer with $\left|\hat{\Gamma}_{i_2} \cup \Gamma_{i_2}\right| = \Omega\left(\frac{1}{p}\right)$, so $\left|\check{\Gamma}_i\right| = \Theta(n)$ holds for all $i < i_2$, where

$\hat{\Gamma}_i$ *and* $\check{\Gamma}_i$ *are defined as in* (4.3). *See Figure 4.3 for a diagram of the partition.*

$$\hat{\Gamma}_i = \bigcup_{j<i} \Gamma_j \qquad \text{and} \qquad \check{\Gamma}_i = \bigcup_{j>i} \Gamma_j = V \setminus (\hat{\Gamma}_i \cup \Gamma_i).$$

Let u_i be the first node processed in layer i and v_i be the last one. Then the number of children $\mathrm{nc}(u_i)$ and $\mathrm{nc}(v_i)$ can be bounded whp. using Chernoff bounds by

$$\begin{aligned} |\mathrm{nc}(u_i)| &< (1+\delta_1)\left|\check{\Gamma}_i\right| p && (4.11)\\ |\mathrm{nc}(v_i)| &> (1-\delta_2)\left(\left|\check{\Gamma}_i\right| - |\Gamma_i| \cdot |\mathrm{nc}(u_i)|\right) p \\ &> (1-\delta_2)\left(\left|\check{\Gamma}_i\right| - |\Gamma_i| \cdot (1+\delta_1)\left|\check{\Gamma}_i\right| p\right) p \\ &= (1-\delta_2)\left|\check{\Gamma}_i\right| p (1 - \underbrace{(1+\delta_1)|\Gamma_i| p}_{x}) && (4.12) \end{aligned}$$

for constants $0 < \delta_1 < 1$ and $0 < \delta_2 < 1$ as long as $|\Gamma_i| = o\left(\frac{1}{p}\right)$ holds. In this case x, the reduction due to the loss of possible child nodes, is bounded by $o(1)$, so its effect is negligible.

This changes in layer i_2 where $\left|\hat{\Gamma}_{i_2} \cup \Gamma_{i_2}\right| = \Omega\left(\frac{1}{p}\right)$ holds for the first time because we expect each node in $\check{\Gamma}_{i_2}$ to have $\Omega(1)$ edges to $\hat{\Gamma}_{i_2} \cup \Gamma_{i_2}$. We cannot argue that all these edges go from $\check{\Gamma}_{i_2}$ to Γ_{i_2}, since both sets are not fully randomly selected. So instead we argue that two arbitrary sets of nodes without any edges between them cannot be too big.

We assume $p \geq \frac{c \log n}{n}$.

Given an arbitrary subset $B \subset V$ with $|B| = \Omega\left(\frac{n}{\log n}\right)$, we can bound the size of a set $A \subset V$ with $A \cap B = \{\}$ such that there exists no edge $(a, b) \in A \times B$ in E as follows:

$$\begin{aligned}\mathbb{P}[\nexists\,(a,b)\;:\;a\in A, b\in B] &= (1-p)^{|A||B|}\\ &< \left(1-\frac{\log n}{n}\right)^{|A|\cdot\Omega\left(\frac{1}{p}\right)} = \left(1-\frac{\log n}{n}\right)^{|A|\cdot\Omega\left(\frac{n}{np}\right)}\\ &< e^{-\log(n)\cdot|A|\cdot\Omega\left(\frac{1}{np}\right)} = n^{-|A|\cdot\Omega\left(\frac{1}{np}\right)}\end{aligned}$$

So from $|A|\cdot\Omega\left(\frac{1}{np}\right) = \Omega(1)$ we see that such a set A with $|A| = \Omega(np)$ nodes does not exist whp. and thus $|A| = o(np)$ follows.

This means that, given $\left|\hat{\Gamma}_{i_2}\cup\Gamma_{i_2}\right| = \Omega\left(\frac{1}{p}\right)$, we know that $\left|\hat{\Gamma}_{i_2+1}\cup\Gamma_{i_2+1}\right| = \Theta(n)$ and $\left|\check{\Gamma}_{i_2+1}\right| = o(np)$ holds whp. because the set of nodes without edges to $\hat{\Gamma}_{i_2}\cup\Gamma_{i_2}$ cannot have more than $\Omega(np)$ nodes as we just showed. So either during processing layer i_2 or layer i_2+1 the set of nodes not yet connected to the BFS tree shrinks from $\Theta(n)$ nodes to $o(np)$ nodes. The consequence is that at the first nodes processed in the layer have $\mathcal{O}(np)$ neighbors while the last nodes processed have only a constant number of neighbors or none at all in the BFS tree whp.

In summary the nodes in each layer have a similar number of successor nodes in the BFS tree up to the layer where the set of unconnected nodes runs out of nodes. In this layer the first processed nodes have up to $\mathcal{O}(np)$ neighbors while the last ones have only a constant number or none at all. The remaining $o(np)$ nodes will most likely not only connect to the first processed node in the last layer but even if they do, this would only double the number of nodes in the subtree which is covered by the factor of $\mathcal{O}(np)$ Lemma 4.12 claims. Keep in mind that a subtree always contains the root itself and thus have at least one node.

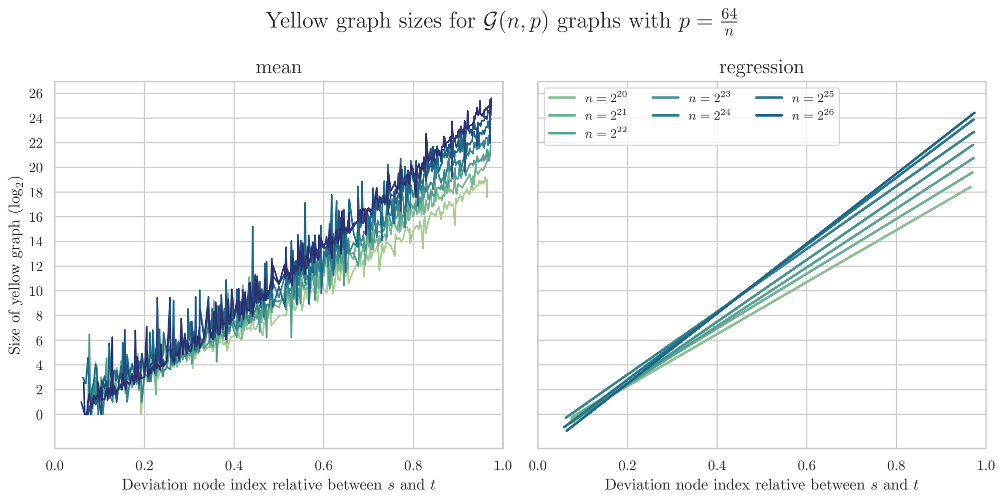
So we have shown that for all nodes $u, v\in\Gamma_i$ with $i\leq i_2$ the subtrees $\Gamma(u)$ and $\Gamma(v)$ differ at most by a factor of $\mathcal{O}(np)$ in size whp. $\square$

4.4.2 Empirical Evidence on Weighted $\mathcal{G}(n,p)$ Graphs

We want to stress that the result of Theorem 4.9 only holds for *unweighted* graphs following the $\mathcal{G}(n,p)$ model. It seems to be much harder to prove a similar result for the weighted case. Instead we provide some empirical evidence that even in the *weighted* case the yellow graphs grow exponentially.

We computed on several $\mathcal{G}(n,p)$ graphs the $k = 50$ shortest paths for 20 random pairs of source and target nodes. For each computed deviation, we computed the size of the yellow graph hanging from each of the deviation nodes. In order to make the sizes comparable, the index of the deviation nodes $i\in\{0,r\}$ were normalized to an relative index between $i'\in[0;1]$ by dividing i by the number of hops r of the deviation path. Figure 4.5 contains all these sizes except for the source node, which would always have zero nodes in the yellow graph, and the target node, which we never deviate from.

Figure 4.5 (top) shows that the size of the yellow graph grows exponentially for directed, weighted $\mathcal{G}(n,p)$ graphs with $p = \frac{64}{n}$. For $2^{20}\leq n\leq 2^{26}$ this p is of order of $\Theta\left(\frac{\log n}{n}\right)$ as in Theorem 4.9. However, the yellow graphs also grow exponentially



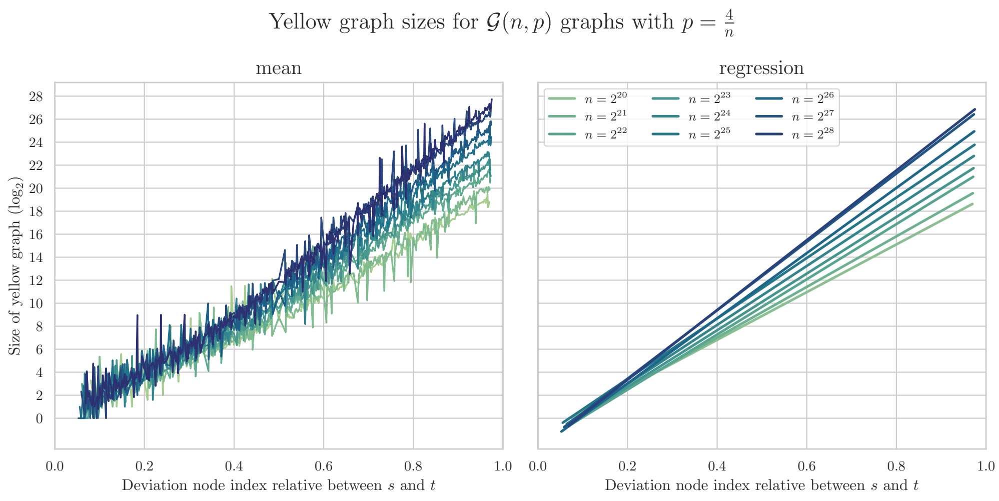


Figure 4.5: *Sizes of yellow graphs per deviation node index relative to the number of nodes on the respective shortest path. Averaged over 20 runs computing the $k = 50$ shortest paths on $\mathcal{G}(n,p)$ graphs with $p = \frac{64}{n}$ (top) and $p = \frac{4}{n}$ (bottom) as well as uniform random edge weights over $[0;1]$. We observe an exponential growth of the yellow graphs in both cases.*

for the very sparse case of $p = \frac{4}{n}$ as Figure 4.5 (bottom) shows. So according to the empirical evidence, the Theorem 4.9 does not only hold for unweighted graphs but also for weighted $\mathcal{G}(n,p)$ graphs with $p = \Omega\left(\frac{\log n}{n}\right)$ as well as weighted $\mathcal{G}(n,p)$ graphs with $p \geq \frac{4}{n}$. We also want to point out that Theorem 4.9 only holds for constant values of k. In the experiments we used $k = 50$ which is about a logarithmic in the number of nodes of the graphs we used. This may indicates that Theorem 4.9 also holds for some $k = \omega(1)$. However, constant and logarithmic values are hard to distinguish even on graphs that barely fit in memory.

5 Comparison of Sequential k-Shortest Path Algorithms

After analyzing the average-case complexity of Yen's and Feng's algorithm in Chapter 4, this chapter discusses how the two algorithms compare in practice and also how the other variants of Yen's algorithm fit into the picture. Before we turn to the comparison of the sequential performance of the algorithms in Section 5.4.2, we take a closer look at two novel heuristic improvements in Sections 5.1 and 5.2 as well as some minor optimizations and implementation details in Section 5.3.

The algorithms we consider in our comparisons are Yen's and Feng's algorithms each with a certain subset of algorithmic improvements, namely:

- Graph preprocessing (G), sometimes called *guiding*, denoted by "G" in the algorithm name.
- Skipping an SSSP computation by a loopless shortest deviation (SD) or second-shortest deviation (SSD) pulled from the reverse SSSP tree denoted in the algorithm name by "S" and "S2", respectively.
- Skipping an SSSP computation by a shortest deviation (SDL) or second-shortest deviation (SSDL) that is already too long to be considered as a candidate for the k^{th}-shortest path denoted by "L".

Skipping SSSP computations:
☞ Section 5.2

For clarity Table 5.1 contains a full list of algorithms and their respective features. Note that all algorithms always use the early stopping optimization described in Section 5.1 which is why it is not accounted for in the algorithm names.

Algorithm / Feature	Yen	Yen-S	Yen-S-L	Yen-S2	Yen-S2-L	Yen-G	Yen-GS	Yen-GS-L	Yen-GS2	Yen-GS2-L	Feng-GS	Feng-GS-L	Feng-GS2	Feng-GS2-L
ES	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G						✓	✓	✓	✓	✓	✓	✓	✓	✓
SD		✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
SDL			✓		✓			✓		✓		✓		✓
SSD				✓	✓				✓	✓			✓	✓
SSDL					✓					✓				✓
Y											✓	✓	✓	✓

Table 5.1: *All algorithm variants and their respective features. Early stopping (ES) is always used and mentioned just for completeness which is why it is not reflected in the algorithm names. Yellow graph (Y) is an exclusive feature to variants of Feng's algorithm.*

We refer to Feng's algorithm in its basic form as Feng-gs to underline that it uses the graph preprocessing and skips SSSP computations if the shortest deviation pulled from the SSSP tree is loopless, even though this is redundant since Feng's algorithm is based on these features.

Throughout this chapter we present the results of several experiments. In order to focus on the relevant information, we moved some of the metadata to Appendix A.2.3.

5.1 Stop SSSP Computations as Early as Possible

There are two situations that allow us to stop an SSSP computation early, from now on referred to as *early stopping*.

1. Since, in most cases, we are only interested in the shortest path to the target node t, we can stop the SSSP computation as soon as the distance to t is settled. For Δ-stepping this is the case after t was in the latest processed bucket.

Any upper limit for the path length would work.

2. If k, the number of paths to compute, is known and the candidate list contains already k candidates, we can use the length of the k^{th} candidate to stop the SSSP computation as soon as we see that t is further away than the k^{th} candidate is long.

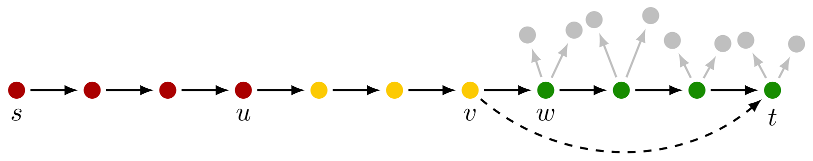


Figure 5.1: Example showing the i^{th}-shortest path deviating at node u. The node coloring is according to Feng. When the graph preprocessing is used, all edges between green nodes have a weight of zero. In the yellow graph the edge (v, w) is replaced by the express edge (v, t).

Graph Preprocessing:
☞ Section 3.2.2.2

Stopping the SSSP computation early is especially powerful when used in combination with the graph preprocessing used by Feng's algorithm. Using the yellow graph analysis in Section 3.2.2.3 and Lemma 3.1 e), we can see that as soon as the SSSP algorithm settles a node w that would be colored green, as shown in Figure 5.1, all nodes on the shortest path from w to t have the same distance from the source node as w since all edges between green nodes have a weight of zero due to the graph preprocessing. Thus, these nodes, including the target node, are settled next and the SSSP computation can be stopped. While settling the green nodes distances, all out-edges are relaxed, including the gray edges shown in Figure 5.1. These relaxations are skipped when the yellow graph is used by replacing the edge (v, w) by an express edge (v, t). So if the the average degree is small enough and the number of green nodes on the shortest path to t is also small, the fraction of explored nodes when using the graph preprocessing without the yellow graph should be comparable to when the yellow graph is used. If neither the graph preprocessing nor the yellow graph is used, all gray edges are relaxed,

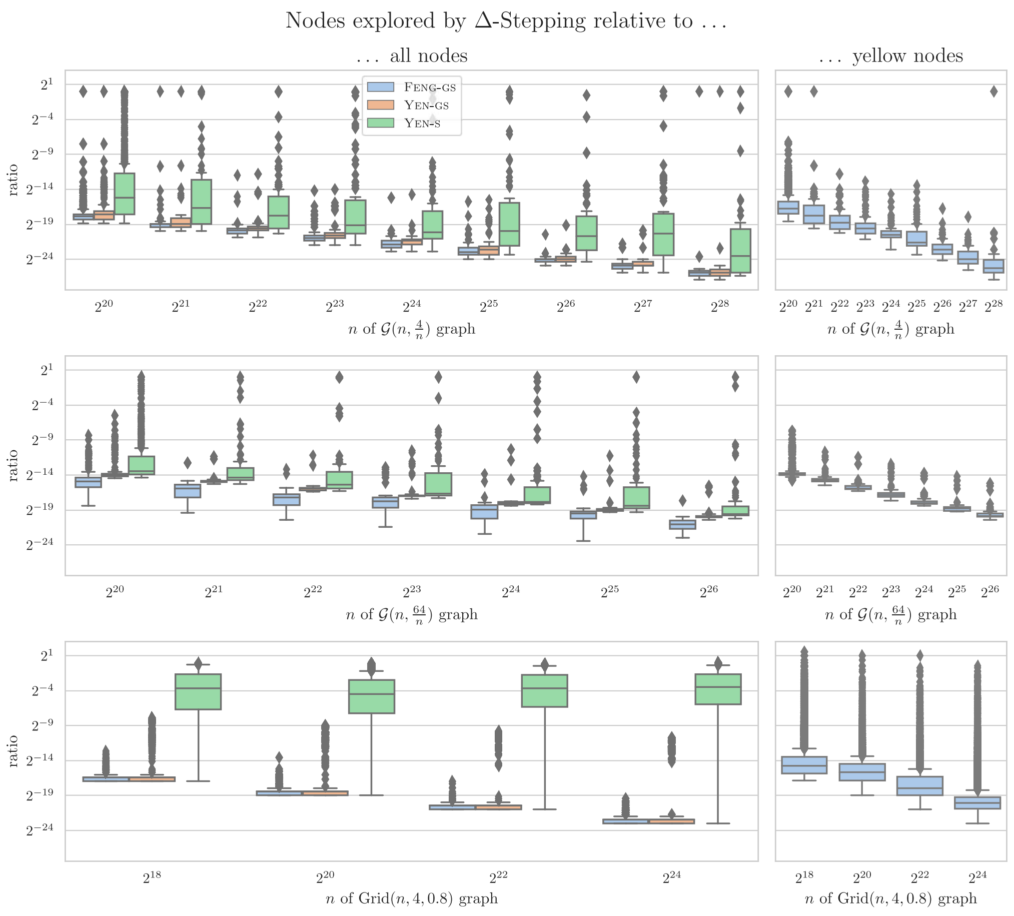


Figure 5.2: *Ratio of explored nodes by Δ-stepping ($\Delta = 0.01$) relative to the total number of nodes (left) and relative to the number of yellow nodes (right) on $\mathcal{G}(n,p)$ and $\mathrm{Grid}(n,r,p)$ graphs with uniform random edge weights over $[0;1]$. Using early stopping, all algorithms only explore a tiny fraction of the graph. Yen-gs explores about as many nodes as Feng-gs even though Feng-gs uses the yellow graph to limit the number of explored nodes. In fact, even on the yellow graph only a fraction of the nodes gets explored.*

some of the gray nodes' distances might be settled and even more edges get relaxed and thus a much bigger fraction of the graph gets explored.

Figure 5.2 confirms that on sparse $\mathcal{G}(n,p)$ graphs with $p = \frac{4}{n}$ the number of explored nodes when using early stopping together with the graph preprocessing is comparable to the number of explored nodes when using the yellow graph. In both cases only a few hundred nodes are explored in most of the SSSP calls independent of the number of nodes in the graph. Without the graph preprocessing on $\mathcal{G}(n,p)$ graphs less than one thousandth of the nodes are explored by the SSSP algorithm but still a lot more nodes compared to when graph preprocessing is used. This suggest that we can achieve a similar performance in the SSSP computation by just using the graph preprocessing without the need to maintain the yellow graph.

Note that the yellow graph is designed to limit the number of nodes an SSSP computation is performed on in order to spend as little time as possible on computing irrelevant paths and distances. Using the yellow graph, only yellow nodes can be explored because all edges (v,w) between a yellow node v and a green node w are replaced by an express edge (v,t) and thus green nodes other than the target node t are unreachable. However, even then stopping the SSSP computation early allows to explore only a fraction of the yellow nodes, about $\frac{1}{2^{17}}$ for $\mathcal{G}(n,p)$ graphs with about $n = 2^{20}$ nodes and about a factor of $\frac{1}{2^{24}}$ for $\mathcal{G}(n,p)$ graphs with $n = 2^{28}$ nodes and $p = \frac{4}{n}$, as Figure 5.2 also shows.

The yellow graph also contains a single red node (the deviation node) and a single green node (the target node).

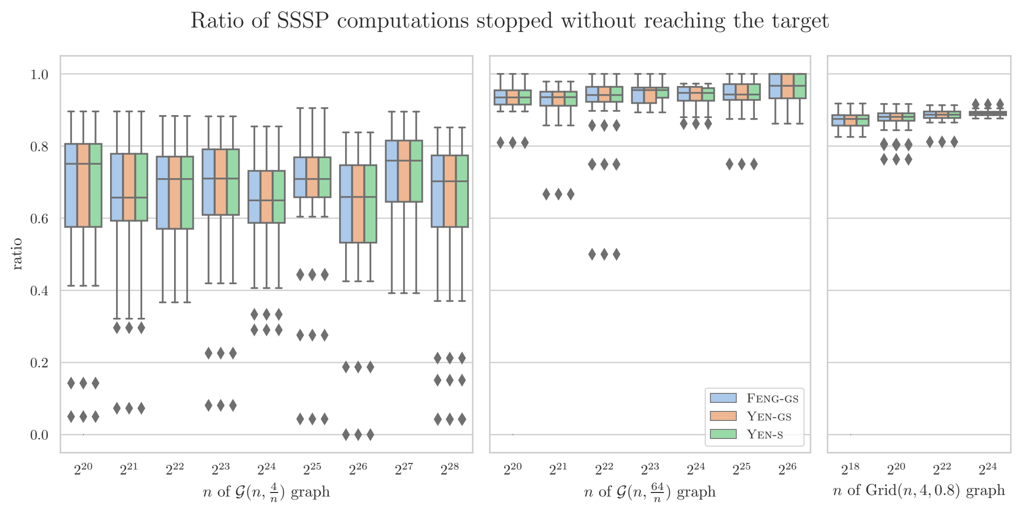


Figure 5.3: *The ratio of SSSP computations stopped before settling the distance to the target node for $\mathcal{G}(n,p)$ and* Grid(n,r,p) *graphs with uniform random edge weights over* $[0;1]$. *Most SSSP computations can be stopped early; over 60% on sparse $\mathcal{G}(n,p)$ graphs and about 90% on denser $\mathcal{G}(n,p)$ as well as* Grid(n,r,p) *graphs.*

We have a similar picture for denser $\mathcal{G}(n,p)$ graphs with $p = \frac{64}{n}$ and for Grid(n,r,p) graphs with $r = 4$ and $p = 0.8$ with minor differences. For denser $\mathcal{G}(n,p)$ graphs using the yellow graph much less nodes get explored compared to if only the graph preprocessing is used. This is due to the higher average degree of the nodes that are skipped by the express edges. On grid graphs the graph preprocessing is about as good as the yellow graphs when it comes to the number of explored nodes. But we see that without graph preprocessing a lot more nodes get explored, more than $\frac{1}{2^4}$ of all nodes in most cases. So here early stopping with graph preprocessing makes an even bigger difference than for the sparse $\mathcal{G}(n,p)$ graphs.

When used with the graph preprocessing, early stopping seems to make the yellow graph rather an analysis tool than an actual temporary graph representation. Our experiments in Section 5.4.2 show that on our synthetic graphs the yellow graph does slow down the algorithms in most cases. But we see in Section 5.4.4 that this is not true for all graphs.

One advantage the yellow graph could also have over the full graph with preprocessing is that it can be implemented in a way that the number of nodes gets actually reduced. If it is implemented in such a way, it could in turn allow for a smaller memory footprint reducing time for memory allocation and possible cache misses. However, such an implementation would also require a map between yellow graph nodes and original graph nodes to be maintained. We actually did this in [3] but since then implemented a simpler and faster version described in Section 5.3, so our current implementation of SSSP tree and Δ-stepping does not benefit from a potentially small yellow graph in terms of memory usage.

While stopping the SSSP computation as soon as the target's distance is settled will prune the work of settling most distances to nodes further away than the target node t, stopping when noticing that the target node itself is too far away will prune even more work if it happens early enough. Figure 5.3 shows that in median more than 60% of the SSSP computations were stopped before settling the target nodes distance on sparse

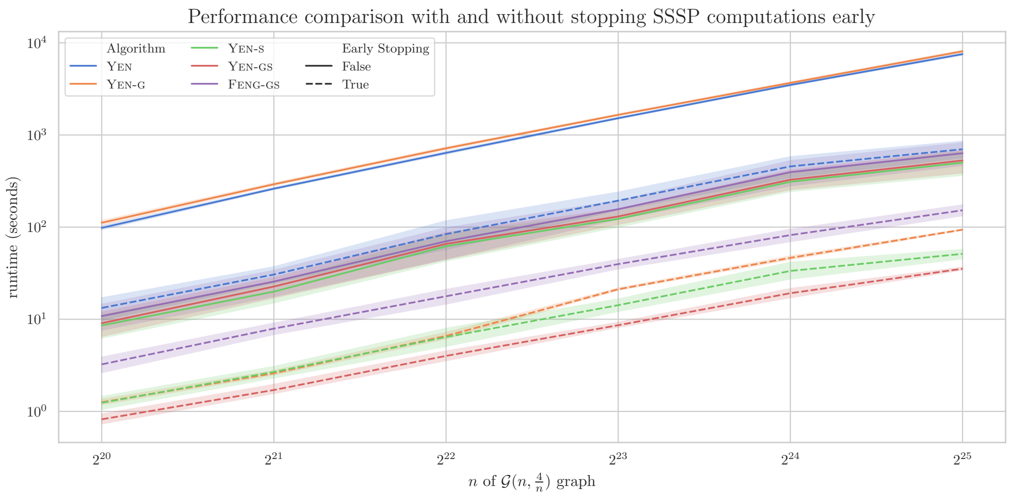


Figure 5.4: *Runtimes of the algorithms* Yen*,* Yen-G*,* Yen-S*,* Yen-GS*, and* Feng-GS *with and without early stopping of SSSP computations.* Yen-S*,* Yen-GS*, and* Feng-GS *have a very similar runtime. Early stopping speeds up all algorithms, most by about an order of magnitude,* Yen-G *by about two orders of magnitude.*

$\mathcal{G}(n,p)$ graphs and close to 90% on denser $\mathcal{G}(n,p)$ and $\text{Grid}(n,r,p)$ graphs. This is, as expected, the same for all examined algorithms because all experiments were performed on the same set of random inputs and the respective distances are only shifted by the graph preprocessing such that Δ-stepping stops after processing the same respective bucket in all cases. Note that this data is measured on the SSSP computations that were not skipped completely. Only for Yen-S there are two minor differences on two instances of the denser $\mathcal{G}(n,p)$ graphs. These come from the missing preprocessing resulting in a slightly different graphs and potentially other skipped SSSP computations.

SSSP Skipping: ☞ *Section 5.2*

Since it costs almost no time to check if the SSSP algorithm can be stopped early, this optimization is always part of our implementations and thus is not reflected in the algorithm names. To demonstrate its impact, we did a small performance comparison between the five basic algorithms Yen, Yen-G, Yen-S, Yen-GS, and Feng-GS. As Figure 5.4 shows, all algorithms benefit highly from using an SSSP algorithm with early stopping. Yen speeds up by about an order of magnitude when using early stopping and becomes comparable with Yen-S, Yen-GS, and Feng-GS without early stopping while Yen-G becomes about two orders of magnitude faster. Yen-S and Yen-GS also speed up by about an order of magnitude, for Feng-GS it is about a factor of three. Figure 5.4 also shows that Yen with early stopping is about as fast as Feng-GS without early stopping. We want to stress the fact that Yen-G with early stopping is even faster than Feng-GS with early stopping besides the fact that Yen-G is not using SSSP skipping like Feng-GS is. This shows the power of early stopping in combination with graph preprocessing on one side and the costs of maintaining a yellow graph on the other side.

Algorithm features: ☞ *Table 5.1*

5.2 Skipping SSSP Computations

Even better than stopping SSSP computations as early as possible is skipping the whole SSSP computation completely. But in contrast to early stopping this optimization is not entirely free since it requires to construct a reverse graph where all edges are inverted as well as to compute a full SSSP tree on the reverse graph to be able to look up shortest

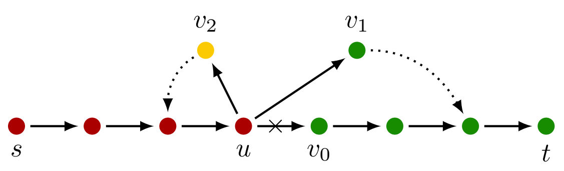


Figure 5.5: Illustration of what deviating at node u can look like. Solid arrows are edges, dotted arrows are paths going only through nodes of the same color as the node they start at. The edge (u, v_0) cannot be used since it was already used before. The node colors are according to Feng's node coloring described in Section 3.2.2.3. However, the coloring is not required in order to be able to skip SSSP computations. If graph preprocessing is used, the edges (u, v_1) and (u, v_2) carry the total weight of the shortest paths $u \rightarrow v_1 \dashrightarrow t$ and $u \rightarrow v_2 \dashrightarrow t$, respectively, and all other edges on the respective shortest paths have weights zero. If $\mathrm{d}(u, v_1) < \mathrm{d}(u, v_2)$, the SSSP computation can be skipped since the deviation is loopless. Otherwise the shortest path goes through a red node and introduces a loop. If the graph preprocessing is not used, $\mathrm{d}(u, v_1) > \mathrm{d}(u, v_2)$ could hold while $u \rightarrow v_1 \dashrightarrow t$ is shorter than $u \rightarrow v_2 \dashrightarrow t$. In this case all valid neighbors need to be checked for their distance to the target node.

paths fast. Skipping SSSP computations by looking at the shortest deviation path pulled from a precomputed reverse SSSP tree was introduced by Feng [24] in the context of Feng's algorithm as described in Section 3.2.2.4 in terms of the yellow graph. We used SSSP skipping without the yellow graph to improve the running time of Yen's algorithm resulting in the algorithm OptYen we described in [3].

OptYen from [3] is here called Yen-s due to a broader spectrum of optimizations.

Skipping SSSP computations works basically as follows: We pull the shortest deviation path p from a precomputed reverse SSSP tree rooted in the target node t. If the path p is simple, we can add it directly to the candidate list without calling an SSSP algorithm that would compute exactly the same path. We call this *skip by shortest deviation* (SD). If p is not simple, we can compare its length with the k^{th} candidate in the list at that moment. If p, the shortest possible deviation, is already longer, in terms of weight, than the k^{th} candidate, the shortest *simple* path will be so, too. So even though we have not found a valid candidate, we can skip the SSSP computation because we already know that its result would be a path that is too long to be considered as a candidate. We call this *skip by shortest deviation length*, SDL for short. Note that skipping by shortest deviation length requires either k to be known or a threshold for the length to be given. Details on skipping SSSP computations can be found in Section 5.2.1.

We always assume k to be part of the input.

The concept of skipping SSSP computations can be extended to the second-shortest deviation and in fact to any j^{th}-shortest deviation. But we only look into skipping SSSP computations by first and second-shortest deviation since the skips become much more complicated while simultaneously yielding less room for improvements, as we show in Section 5.2.3.

Skip by second-shortest deviation: ☞ Section 5.2.2

5.2.1 Skip by Shortest Deviation: Details

Note that the closest neighbors include only neighbors that can be reached via edges that are still allowed. See Section 3.2.1 for details.

If the closest neighbor v, in terms of edge weight, of the deviation node u is green, the shortest path from v to the target node t does not go through a red node and thus can be pulled from the reverse shortest path tree as the new candidate. In case of Feng's algorithm, it is enough to check the closest neighbor and its path to be loopless, since Feng's algorithm uses the graph preprocessing and all edges on the shortest path from green nodes v to t have a weight of zero. So the total weight-length of the shortest path from u to t is the weight of the edge (u, v). If the out-edges of u are sorted by their weight, the algorithm has to check only a single edge, which makes this the least expensive version of skipping an SSSP computation.

If the shortest deviation path p we get from the reverse SSSP tree is not loopless, we can still use it. Since all k-SP algorithms we consider do maintain a list of candidates and there are most likely many more than k candidates, we can compare p to the k^{th} candidate c. If p is already longer than c, we can also skip the SSSP computation since a valid deviation path would be even longer than p and thus cannot be one of the k shortest paths.

See Figure 5.5 for an illustration of SSSP skipping.

The same skips are still possible if we do not have the node coloring. If the graph preprocessing is still used, let v be the closest neighbor, in terms of weight, to the deviation node u. We then need to pull the shortest path from v to t from the reverse shortest path tree, and check if the path contains any loops. This can be done by sorting the nodes on the path and checking if any node appears more than once. The time cost of this check depends on the number of hops of the path and in turn on the graph structure. In worst-case this check takes $\mathcal{O}(n \log n)$. However, as we showed in Chapter 4 shortest paths can have much less than n hops and thus checking a path to be loopless can be done in $\mathcal{O}(\log(n) \cdot \log\log(n))$ on $\mathcal{G}(n, p)$ graphs whp.

If skipping is used without the graph preprocessing, the algorithm cannot just use the shortest out-edge, in terms of weight, but needs to find the out-edge (u, v) such that $\mathrm{d}(u, v) + \mathrm{d}\big(v \dashrightarrow t\big)$ is minimal. Since the shortest distance $\mathrm{d}\big(v \dashrightarrow t\big)$ is stored in the reverse shortest path tree, computing $\mathrm{d}(u, v) + \mathrm{d}\big(v \dashrightarrow t\big)$ only takes additional constant time for each out-edge. The total time to compute all these distances is then linear in the out-degree of the deviation node.

Note that both the node coloring and the weight function used by the graph preprocessing do not change whether an SSSP computations can be skipped. But the preprocessing step we use does also remove u–t-paths that pass through s, so we expect that more SSSP computations can be skipped when the preprocessing is used. However, in our considered settings it is a very rare event that shortest deviations would go through s and get removed by the graph preprocessing. So we do not see any better chance for skipping SSSP computations if graph preprocessing is used in out experiments.

SSSP skip statistics: ☞ Section 5.2.3

5.2.2 Skip by Second-Shortest Deviation

The concept of skipping SSSP computations by the shortest deviation can be extended to the second-shortest deviation if the shortest deviation p is not loopless and not already too long to be considered as a candidate. In order to compute the second-shortest deviation, the algorithm needs to compute a deviation path from each node of the shortest deviation p between the deviation node u and the first red node v_3 which

Deviating from the shortest deviation: ☞ Figure 5.6

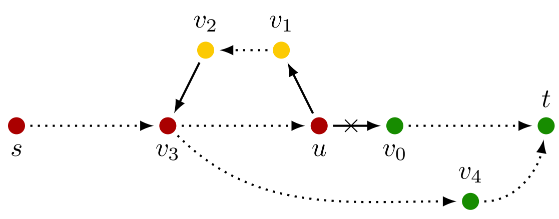


Figure 5.6: Illustration of what a shortest deviation pulled from the reverse shortest path tree looks like if it is not loopless and thus cannot be used to skip an SSSP computation. Solid arrows are edges and dotted arrows are subpaths. The node coloring, here only for clarity, is according to Feng's node coloring. An actual coloring is not required. The subpaths $s \rightsquigarrow v_3$ and $v_3 \rightsquigarrow u$ go only through red nodes, $v_1 \rightsquigarrow v_2$ goes only through yellow nodes. The path $v_3 \rightsquigarrow v_4$ goes through red and yellow nodes while v_4 is the first green node it visits. The path $p = (s, \ldots, v_3 \ldots, u, v_0, \ldots, t)$ is the path we want to compute a loopless deviation from and $d_1 = (s \ldots, u, v_1, \ldots, v_2, v_3, \ldots, v_4, \ldots, t)$ is the shortest deviation pulled from the reverse SSSP tree. In order to find the second-shortest deviation, only the nodes $u, v_1, \ldots, v_2$ need to be considered since deviations from d_1 at other nodes would include the node v_3 and thus again not be loopless.

introduces the loop, not including v_3 itself as shown in Figure 5.6. Deviations from the node v_3 onwards will always contain the same loop as the shortest deviation and thus will never be considered to become a candidate for the k shortest paths.

Computing the second-shortest deviation can be done just like described before by choosing for each deviation node u' the path p' such that $\mathrm{d}(u', v') + \mathrm{d}\big(v' \dashrightarrow t\big)$ is minimal. If the graph preprocessing is not used, all out-edges of each deviation node need to be checked for this. Note that we do not need to check each deviation to be loopless but only the shortest one. Again, just like for the shortest deviation, if the second-shortest deviation is not loopless, its length can be compared to the current k^{th} candidate to maybe skip the SSSP computation.

Feng's node coloring: ☞ *Section 3.2.2.3*

Implementation details: ☞ *Section 5.3*

Note that all nodes between v_1 and v_2 as shown in Figure 5.6 are yellow by their definition, since the shortest path in the reverse SSSP tree passes through a red node. This is important to be able to deviate from them in case Feng's algorithm is used. If there would be green nodes between v_1 and v_2, they would not necessarily be part of the yellow graph depending on the implementation.

Algorithm 8 combines the described steps for skipping an SSSP computation by the first and second-shortest deviation as well as their respective length. It is implemented in a generic way and can be used with and without graph preprocessing. However, if graph preprocessing is used and edges are sorted by length, the for-loops iterating over out-edges can be optimized away. An SSSP computation only needs to be carried out if Algorithm 8 returns FALSE.

Iterating over each out-edge of up to each node at most one time mean to iterate over at most each edge once.

The overall complexity of Algorithm 8 depends on the time to check for a path to be loopless in lines 7 and 20 as well as the number of hops on the shortest deviation we iterate over in line 14. As stated before, checking for a path to be loopless takes up to $\mathcal{O}(n \log n)$ by sorting the nodes on the path and looking for duplicate nodes. If the graph preprocessing is not used and the edges are not sorted by weight, all out-edges

Algorithm 8: Subroutine ATTEMPTSSSPSKIP

```
    Input  : Path $P_i = (v_1^{(i)}, \dots, v_{|P_i|}^{(i)})$ to deviate from, the index $j$ where to deviate from $P_i$,
             the temporary graph $G_j^{(i)}$, the reverse shortest path tree $T$, the list of candidates $C$
    Output : TRUE if the SSSP computation can be skipped, else FALSE.
    /* check the shortest deviation                                                                  */
 1  $L \leftarrow \infty$, $P \leftarrow$ NONE
 2  for $\left(v_j^{(i)}, u\right) \in G_j^{(i)}$ do            // the edge $\left(v_j^{(i)}, v_{j+1}^{(i)}\right)$ does not exist in $G_j^{(i)}$
 3      $P' \leftarrow \mathrm{R}_j(P_i) \circ T$.GETSHORTESTPATH($u$)
 4      if $\mathrm{d}(P') < L$ then
 5          $L \leftarrow \mathrm{d}(P')$, $P \leftarrow P'$
 6  if $L = \infty$ then return TRUE                                   // no deviation path exists at all
 7  if $P$ is loopless then                                            // a valid deviation was found
 8      $C$.PUSH(($P, j, D_{i,j}$))         // Add the path to the candidate list. $D_{i,j}$ is defined in (3.1).
 9      return TRUE
10  else                                                                // the shortest deviation has a loop
        /* If the length of $P = (v_1, \dots, v_{|P|})$ is longer than the current $k^{\text{th}}$ candidate      */
        /* length, we can also skip the SSSP computation. If there is no $k^{\text{th}}$ candidate yet,   */
        /* the function returns $\infty$.                                                               */
11      if $L \geq C$.GETLENGTHOFCANDIDATE($k$) then return TRUE
        /* $P$ is short enough so we look for the second-shortest deviation                             */
12      $J \leftarrow$ the index of the node that introduces the loop to $P$
13      $L \leftarrow \infty$, $Q \leftarrow$ NONE
14      for $j' \leftarrow j, \dots, J-1$ do
15          for $(v_{j'}, u) \in G_j^{(i)}$ with $u \neq v_{j'+1}$ do                // $G_j^{(i)}$ is the same as before
16              $Q' \leftarrow \mathrm{R}_{j'}(P) \circ T$.GETSHORTESTPATH($u$)
17              if $\mathrm{d}(Q') < L$ then
18                  $L \leftarrow \mathrm{d}(Q')$, $Q \leftarrow Q'$
19      if $L = \infty$ then return TRUE                               // No second deviation path exists
20      if $Q$ is loopless then                                        // A valid second deviation was found
21          $C$.PUSH(($Q, j, D_{i,j}$))  // Add the path to the candidate list. $D_{i,j}$ is defined in (3.1).
22          return TRUE
23      else if $L \geq C$.GETLENGTHOFCANDIDATE($k$) then   // the second-shortest deviation is
          too long
24          return TRUE      // we found an invalid path, but a valid path must be even longer
25      else
            /* The two shortest deviations contain loops and are shorter than the $k^{\text{th}}$          */
            /* candidate so we need an SSSP call to find the shortest loopless deviation.       */
26          return FALSE
```

of the respective deviation node need to be checked if they are part of the shortest deviation. Let σ be the maximum out-degree in the graph. In worst-case Algorithm 8 has to iterate over up to σ neighbors of the deviation node to find the shortest deviation and up to $\mathcal{O}(m)$ deviations of the shortest deviation to find the second-shortest deviation.

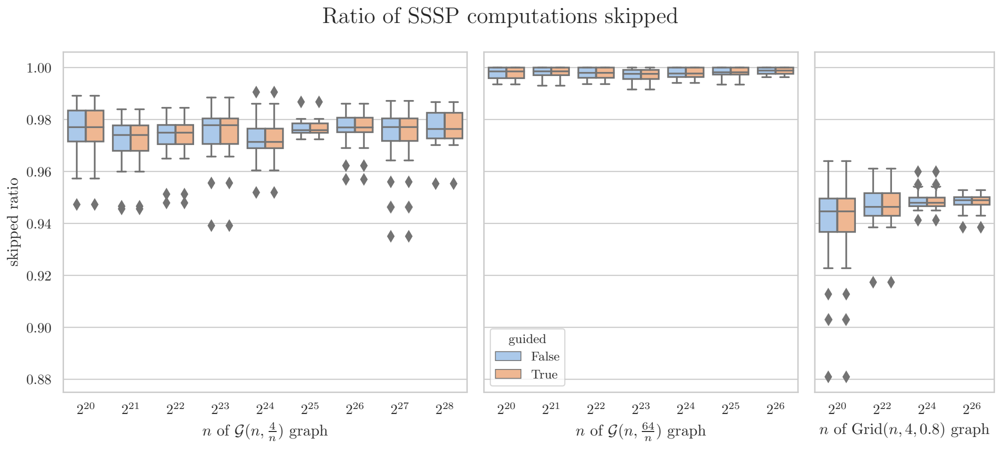


Figure 5.7: *The ratio of skipped SSSP computations to the total number of computed candidates on $\mathcal{G}(n,p)$ and $\mathrm{Grid}(n,r,p)$ graphs with uniform random edge weights over $[0;1]$. On all tested graph types, 94% to 100% of the SSSP computations could be skipped.*

Overall Algorithm 8 can take up to $\mathcal{O}(m + n \log n)$ time in worst-case to find both, the first and second-shortest deviation, and check them to be loopless. Recall that only those two need to be checked to be loopless, not all $\mathcal{O}(m)$ deviations. This gets better when used with graph preprocessing and edges sorted by weight. It then only takes up to $\mathcal{O}(n + n \log n)$ time in worst-case, since only one out-edge needs to be checked for the shortest deviation at each node. Finally, when used with node coloring it takes at most $\mathcal{O}(n)$ time, since we can check both paths to be simple in constant time. Keep in mind that the SSSP computation that is attempted to be skipped takes itself up to $\mathcal{O}(m + n \log n)$ time. So Algorithm 8 does not affect the overall worst-case time complexity of the k-SP algorithms.

Number of hops on k shortest paths: ☞ Lemma 4.6

As we showed in our average-case analysis in Chapter 4, the k shortest paths have only $\mathcal{O}(\log n)$ edges whp. This renders the average-case running time of Algorithm 8 to be $\mathcal{O}(\sigma \log(n) + \log(n) \log \log(n))$ whp. on $\mathcal{G}(n,p)$ graphs with random edge-weights.

As briefly mentioned before, the concept of skipping SSSP computations could be extended further to the third- or any j^{th}-shortest deviation. But from the third-shortest deviation onwards this would require all the bookkeeping that is done by Yen's and Feng's algorithm to ensure not to find candidates multiple times. This includes an extra list of candidates as well as a way to manage all the edges that cannot be used when deviating from a certain path at a certain node which is done by a temporary graph in Yen's algorithm. This would then also cost much more time exceeding the worst-case complexity of the SSSP computation that it is supposed to skip. However, as Figure 5.7 shows, most SSSP computations can already be skipped using the first- and second-shortest deviations or their respective length.

5.2.3 Evaluation of SSSP Computation Skipping Types

By the law of large numbers, $\mathcal{G}(n,m)$ graphs and $\mathcal{G}(n,p)$ graphs with $m = pn^2$ behave in a similar way if the number of nodes is big enough.

In our paper [3] we reported that less than 10% of the SSSP computations can be skipped on average using the shortest deviation on *undirected* $\mathcal{G}(n,m)$ graphs with $m = 4n$ and between 16% to 18% on average on *undirected* $\mathcal{G}(n,m)$ graphs with $m = 128n$ both with uniform random edge weights over $[0;1]$.

When we look at sparse *directed* $\mathcal{G}(n, p)$ graphs the situation is much better. Figure 5.7 shows that for sparse *directed* $\mathcal{G}(n, p)$ graphs with $p = \frac{4}{n}$ and uniform random edge weights over $[0; 1]$, over 97% of the SSSP computations can be skipped, independently of the graph sizes, using one of the skips described in Sections 5.2.1 and 5.2.2. Most SSSP skips can be done just by the shortest deviation (over 90% in median) and its length (about 5% in median), as shown in Figure 5.8. The second-shortest deviations and their respective lengths lead only to 0.1% to 0.3% of the SSSP skips combined on $\mathcal{G}(n, p)$ graphs with $p = \frac{4}{n}$. However, this has to be seen in the context of only about 5% of SSSP computation not already skipped by the shortest deviations and their respective lengths. So this is really about 2% to 6% of the SSSP computations that otherwise could not be skipped by the shortest deviation or its length. Note that skipping via second-shortest deviation can only happen if an SSSP computation could not be skipped already via the shortest deviation. On denser $\mathcal{G}(n, p)$ graphs with $p = \frac{64}{n}$, we see a similar distribution of SSSP computation skips to the four skip types. On grid graphs, skipping SSSP computations via shortest deviation length has a much higher impact. About 42% of SSSP computation skips are accounted for by this type of skip in the median.

Also note that the implementation checks first if a deviation is loopless and if so counts the skip as “skip by shortest deviation” even if the candidate list does not store the deviation because of its length. These skips are also the SSSP computations skipped by Feng’s algorithm as described in [24] and by OptYen described in [3]. Skips by the second-shortest deviation are labeled in the same order.

On denser directed $\mathcal{G}(n, p)$ graph with $p = \frac{64}{n}$, skipping SSSP computation works even better with more than 99% and in some cases all SSSP computations skipped.

For $\mathcal{G}(n, p)$ graphs with $n \leq 2^{26}$, as we use them, $p = \frac{64}{n}$ is in the same order as $\frac{\log n}{n}$.

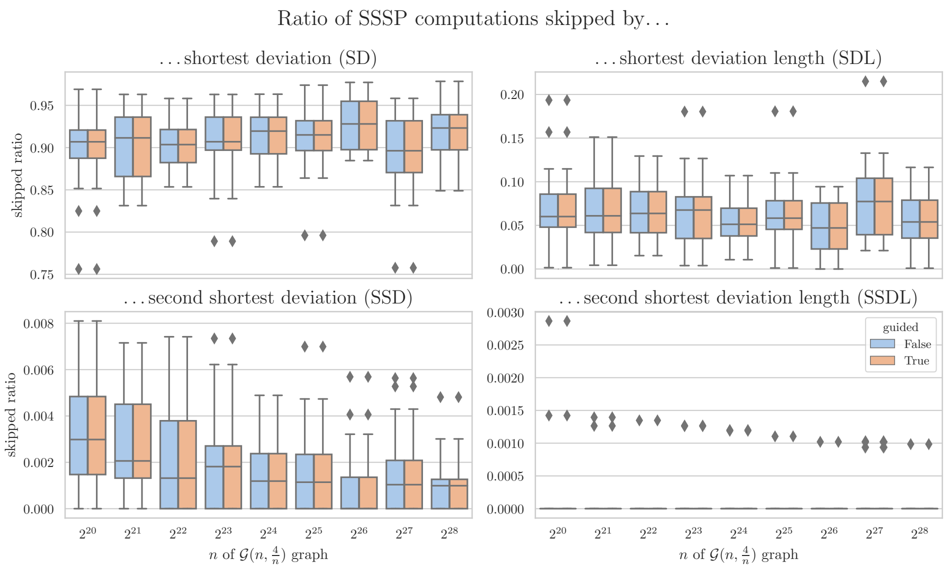
As mentioned in Section 5.2.5, the plots do not show any difference between runs with and without graph preprocessing, even though SSSP skipping could theoretically benefit from the graph preprocessing.

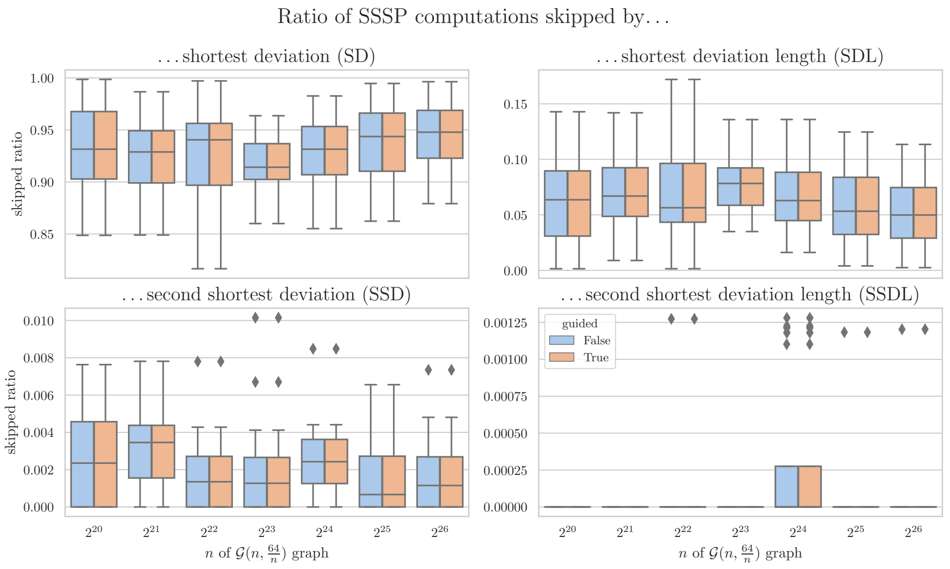
5.2.4 Skipping SSSP Computations on Directed and Undirected Graphs

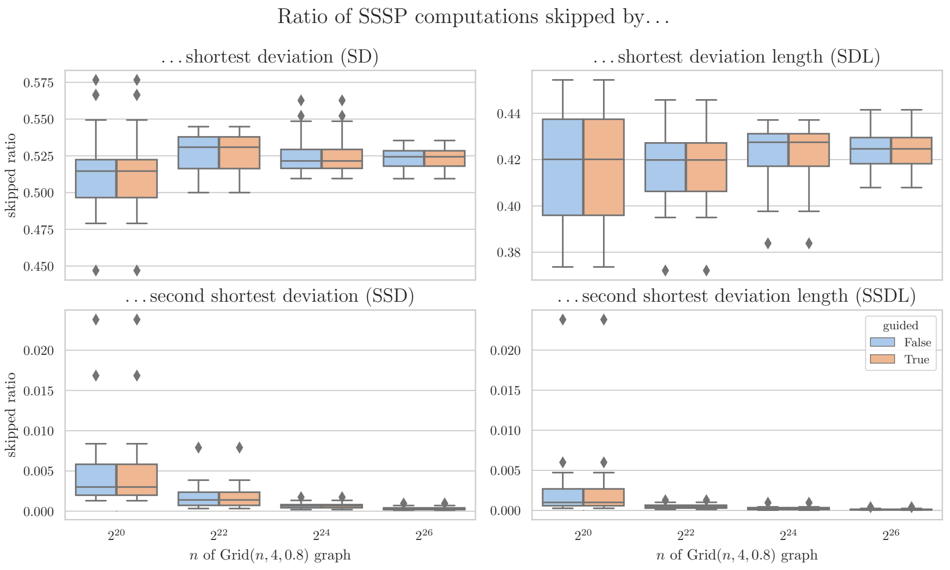
As previously discussed and reported in [3], we found that much less SSSP computations can be skipped on *undirected* graphs. To fill the gap between directed and undirected graphs, we looked at q-directed $\mathcal{G}(n, p)$ graphs, where each edge (u, v) has a chance of $1 - q$ to be undirected. This means that in addition to an edge (u, v) the edge (v, u) with $\mathrm{d}(u, v) = \mathrm{d}(v, u)$ does also exist with probability $1 - q$. To keep keeping the average degree at np we change the probability for an edge (u, v) to be present from p to $p' = \frac{p}{2-q}$. So an 1-directed $\mathcal{G}(n, p)$ graph is a classic directed $\mathcal{G}(n, p')$ graph with $p' = p$ while a 0-directed $\mathcal{G}(n, p)$ graph is an undirected $\mathcal{G}(n, p')$ graph with $p' = \frac{p}{2}$.

Figure 5.9 shows for the SSSP skips via shortest deviation (SD) and 0-directed $\mathcal{G}(n, p)$ graphs a similar ratio of skipped SSSP computations as we reported in [3]. The more unlikely it is for edges to be undirected, the better is the ratio of SSSP computations skipped by the shortest deviation. It also shows that close to 50% of SSSP computations can be skipped on undirected graphs using the shortest deviation length (SDL). This rapidly shrinks for q-directed $\mathcal{G}(n, p)$ graphs with $q \geq 0.8$. The main reason for this

Figure 5.8: *The ratio of skipped SSSP computations to the total number of computed candidates on* $\mathcal{G}(n,p)$ *and* $\mathrm{Grid}(n,r,p)$ *graphs with uniform random edge weights over* $[0;1]$. *This figure shows the same data as Figure 5.7 split up into the four separate skip types. On* $\mathcal{G}(n,p)$ *graphs, by far the most SSSP computations are skipped due to SD with a ratio of over 90%. On* $\mathrm{Grid}(n,r,p)$ *graphs less than 60% of SSSP computations were skipped by SD but more than 40% were be skipped by SDL.*







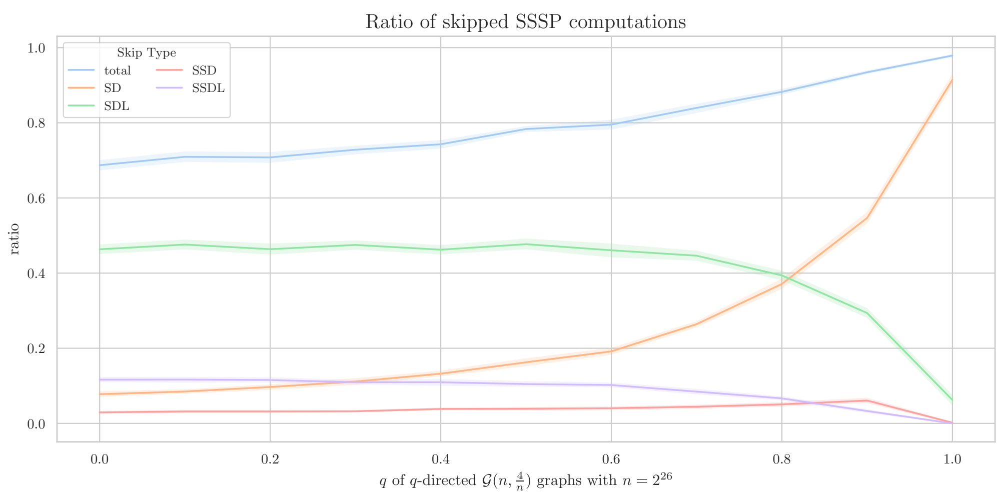


Figure 5.9: *The ratio of SSSP computations that can be skipped via shortest deviation (SD), length of shortest deviations (SDL), second-shortest deviation (SSD), and length of second-shortest deviation (SSDL) for q-directed $\mathcal{G}(n, p)$ graphs.*

is, that the paths become more likely to be loopless and thus get labeled differently. Finally Figure 5.9 also shows that the second-shortest deviation length (SSDL) can be used to skip some SSSP computations, although such paths can only rarely be used as a candidate (SSD). This is even worse for directed graphs.

The reason for the shortest deviation being much less likely to be loopless on undirected graphs than it is on directed graphs, is that on undirected graphs every edge $\{u, v\}$ can be traversed in both directions. So the lighter an edge is, the more likely it is part of the shortest not necessarily loopless deviation. But it is also likely that the shortest path from v to the target node t leads right back to the deviation node u since we already know that its shortest path to t is even shorter than the deviation, otherwise we would not deviate from it.

5.2.5 Skipping SSSP Computations with and without Graph Preprocessing

As mentioned in Section 3.2.2.2, the graph preprocessing does not only change the edge weights while preserving the relative order of s–t-paths but also removes in-edges to the source node s.

Sometimes, we call the graph preprocessing sometimes guiding *for short.*

Let v be a node such that the shortest v–t-path P goes through s. This path cannot be loopless and thus the SSSP computation could only be skipped because of the weight-length of P. After the graph preprocessing, the shortest path p' from v to t cannot pass through s anymore, since all in-edges of s have been removed. The new shortest path p' has now a chance to be loopless and is also at least as long as P was. So if it is still not loopless, it has a higher chance to be already too long to be used as a candidate. This argument carries over to the second-shortest deviation and so on.

So it is theoretically more likely to skip an SSSP computation when guiding is used compared to when it is not. However, we do not see this in our experiments since it is a very rare event in the graphs we considered.

5.3 Implementation Details and Minor Optimizations

Before we compare the performance of the k-shortest path algorithms, we take a brief look into some implementation details. As for basically all algorithms, their efficient implementations differ greatly from their theoretical description. We do not want to go into too much detail, since the engineering of the code is not the focus of this thesis. So these are just some of the most relevant details.

5.3.1 Graphs

Δ-stepping details: ☞ Section 2.3.2

E.g. a graph with 2^{28} nodes and 2^{32} edges requires about 53 GB of memory without padding bytes and 68 GB with padding bytes.

Graphs are stored as two arrays, one containing all edges grouped by tail node, sorted by increasing weight and one containing the start indexes of its light and heavy out-edges for each node according to the given Δ used by Δ-stepping as 8 byte integers. Each edge contains a 4 byte value for its head node and a 8 byte value for its weight. So a graph with n nodes and m edges requires a total of at least $16n + 12m$ bytes of memory. However, depending on the architecture of the hardware, the C++ compiler will most likely add padding bytes for optimal performance. Thus the graph requires about $16(n + m)$ bytes in total.

One could reduce the 8 byte indexes to 4 bytes if the graph has less than 2^{32} edges in total. Since the biggest $\mathcal{G}(n, p)$ graph we consider has about 2^{32} edges in expectation we decided to used 8 byte indexes for all graphs in order to minimize side effects on the runtimes. We use 8 byte values (double precision) for edges weights to avoid casting values between datatypes because we use 8 byte floating point values for path lengths. Using only 4 byte values (single precision) for path lengths leads to wrong path orders due to rounding errors when paths have a lot of hops like on $\mathrm{Grid}(n, r, p)$ graphs.

5.3.2 Temporary and Yellow Graphs

For each deviation path computed by an SSSP algorithm a temporary or yellow graph must be created such that only loopless deviations can be found. This requires removing some nodes or at least all of their in-edges as well as some out-edges of the deviation node.

However, instead of creating a full temporary graph with the described properties, it is enough to store a bit field to mark nodes removed from the temporary graph. While iterating over the out-edges of a node the iterator checks whether the target node is flagged as removed and if so skips the edge. The deviation node itself is the only node in the graph where some of the out-edges are removed while the head node is still in the graph. In this case the iterator can just check whether the edge is in a list of forbidden edges. This is a quite slow method but since only a single node (the deviation node) of the whole graph is affected, this case is the first to be handled and it does not affect the overall performance. This way the temporary graph only uses $\Theta(n)$ additional bit compared to $\Theta(n + m)$ memory for a complete new graph or $\Theta(m)$ memory for a bit field that marks edges instead of nodes. It can be computed in $\Theta\big(\overline{\mathrm{d}}(P)\big)$ time, when deviating from path P. The temporary graph can also be updated and reused for the

The hop-length $\overline{\mathrm{d}}(P)$ of P varies between graph classes. For $\mathcal{G}(n, p)$ graphs $\overline{\mathrm{d}}(P)$ is $\mathcal{O}(\log n)$ whp. while it is $\mathcal{O}(\sqrt{n})$ on $\mathrm{Grid}(n, r, p)$ graphs.

next deviation node by marking only a single node and supplying a new set of forbidden edges for the new deviation node.

For the yellow graph it is similar. Instead of computing a new graph it is enough to store an array that holds the color of each node. The express edges can be generated on the fly whenever an edge to a green node is visited. If an SSSP algorithm like Dijkstra or Δ-stepping is used, the SSSP computation can be stopped as soon as the distance to the target node is settled. In this case only a fraction of all express edges needs to be generated. Also the node coloring can easily be updated to be reused for the next deviation node. Similar to the temporary graph, the yellow graph needs only $\Theta(n)$ additional memory if implemented this way.

Stopping SSSP computation early: ☞ *Section 5.1*

Note that even though the temporary graph and the yellow graph have the same additional memory bound of $\Theta(n)$, the simpler temporary graph only stores $\Theta(n)$ bits while the yellow graph needs to be able to differentiate three states. This could be stored in two bits per node but handling pairs of bits is slower than handling a whole byte which is why in our implementation bytes are used and thus yellow graphs need about eight times more memory than temporary graphs. However, a graph with about 256 million nodes will only need about 256 MB of memory in addition compared to about 32 MB for a simple temporary graph.

Handling pairs of bits is about 3% to 5% slower than using a whole byte according to a micro benchmark we did. In addition an array of bits is not thread-safe.

5.3.3 SSSP Tree

The SSSP tree stores an array containing for each node a distance as a 8 byte double and its predecessor to the target node as a 4 byte integer, totaling to $12n$ bytes. So the memory requirement is independent from the number of edges in the graph. However, it is also independent from the number of nodes that are actually visited by Δ-stepping. So even if Δ-stepping only visits a tiny fraction of the nodes as we saw in Section 5.1, the whole SSSP tree needs to be allocated and initialized. This is where one could benefit from implementing the yellow graph in a way that a new graph is built with less nodes instead of adding a coloring array on top of the original graph whenever the yellow graph is small enough.

The compiler will most likely add padding bits so the real memory usage will probably be $16n$ bytes.

5.3.4 Δ-Stepping

The main reason for implementing our own version of Δ-stepping is that we need to be able to stop the computation if some conditions are met.

Δ-stepping details: ☞ *Section 2.3.2*

We implemented a simple version of Δ-stepping without adaptive bucket-splitting in case that it gets too full. This optimization seems not to be necessary when doing experiments on random graphs and would make the implementation more complicated. In order to keep the number of buckets as small as possible, we only have $b = \left\lceil \frac{d_{\max}}{\Delta} \right\rceil + 1$ many buckets with $d_{\max}$ being the biggest edge weight in the graph. So when processing bucket i and relaxing an edge (u, v) with weight d we insert v into bucket $i + \frac{d}{\Delta} < i + b - 1$. Since we already cleared all buckets before the i^{th} bucket, we can reuse them by inserting nodes into bucket $i + \frac{d}{\Delta} \mod b$ which is at most $i + b - 1 \mod b = i - 1$.

In all our experiments we used $\Delta = 0.01$ which we empirically found to be a good choice. Since this thesis is not about optimizing Δ, we do not claim that this is the optimal choice. But against the recommendation of choosing $\Delta = \frac{1}{\max\{\deg(v)\,:\,v \in V\}}$ [49] we found that a smaller value works better in our case, especially in context of stopping SSSP computations early.

5.3.5 Code

Parallelization of k-SP algorithms: ☞ *Chapter 6*

We implemented all algorithms mentioned in Table 5.1 and data structures used by these algorithms in C++20 with OpenMP 4.5 for parallelization using the GNU Compiler Collection[1] in version 10.3. The code is published at

https://doi.org/10.5281/zenodo.7713239

under GNU General Public License v3.0. The code is written in a modular way in order make the SSSP algorithm and some data structures easily exchangeable in future research projects.

5.4 Experimental Performance Comparison

In this section we present the results of some runtime experiments comparing the variants of Yen's and Feng's algorithm.

5.4.1 Experimental Setup

List of algorithms: ☞ *Table 5.1*

We ran all previously described algorithms on two types of graphs of varying sizes. For one we use two sets of random $\mathcal{G}(n, p)$ graphs. One set with an average degree of $np = 4$, corresponding to a small constant, and the number of nodes ranging from $n = 2^{20}$ to $n = 2^{28}$. For the other set of $\mathcal{G}(n, p)$ graph we used an average degree of $np = 64$, between two and four times the logarithmic number of nodes, while the number of nodes range from $n = 2^{20}$ to $n = 2^{26}$. This is to ensure that our empirical analysis and our theoretical average-case analysis in Chapter 4 are both done in similar setup. The other graph type we use is grid graphs with the number of nodes ranging from $n = 2^{18}$ to $n = 2^{26}$. The key difference between $\mathcal{G}(n, p)$ graphs and grid graphs is their expected diameter in terms of number of hops, since this is most relevant for the running time of Yen's and Feng's algorithm as well as variants of these algorithms.

Average-Case Results: ☞ *Theorems 4.5, 4.7, and 4.9*

To run the experiments, we generated one graph of each type and size. We then executed several runs drawing each time an independent pair of nodes as source and target nodes. For each pair of nodes we compute the $k = 50$ shortest paths.

We chose $k = 50$ as it lies in a range that many applications [15, 26, 65, 8, 38, 69, 44, 48, 72] seem to use. There are also applications requiring bigger values for k. However, this introduces the problem of storing and handling more shortest paths as well as equally many candidate paths. Optimizing the handling of candidate paths is a research subject on its own [63, 25, 74] which we want to avoid at this point. An approximately

[1] https://gcc.gnu.org/

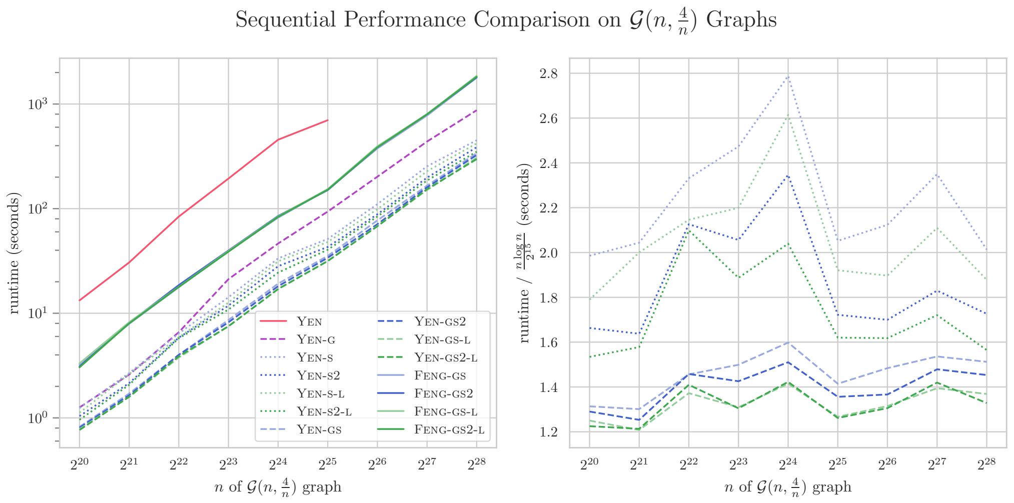


Figure 5.10: *Sequential runtimes of all algorithm variants showing variants of Feng's algorithm as solid lines, variants of Yen's algorithm with guiding as dashed lines and without guiding as dotted lines, respectively. If skipping by deviation length is used the line is colored green and blue if it is not. All algorithms use early stopping.*

logarithmic k, as we chose it, also allows to experiment on bigger graphs since the runtime highly depends on k.

Our experiments ran on machines of the Goethe-HLR[2] compute cluster. The machines contained 196 GB of internal memory and two Intel Xeon Skylake Gold 6148 each having 20 physical cores and 20 Hyper-Threading cores. For the experiments on the sequential algorithms, we locked the executable to one of the physical cores to avoid runtime artifacts from core switching. We ran our experiments on multiple machines all having an identical hard- and software setup in order to keep the total time manageable.

5.4.2 Comparing Sequential Performance

For the sequential performance comparison we looked into the running times of all algorithms listed in Table 5.1. Figure 5.10 shows the running times on sparse $\mathcal{G}(n, p)$ graphs with $p = \frac{4}{n}$. We see that YEN is by far the slowest, which is why we did not run it on the full data set. The other algorithms form essentially four groups:

1. Variants of Feng's algorithm (solid lines). They are all about equal in speed while being about five times faster than YEN. However, maintaining the yellow graph costs time which makes this group the second slowest.

2. Variants of Yen's algorithm without guiding but with SSSP skipping (dotted lines). This group is the second fastest. The right side of Figure 5.10 shows that the variants with skipping SSSP computations by the second deviation is faster than only skipping them by the shortest deviation and it gets even faster when SSSP skipping by the deviation length is also turned on, making them about three times faster than the variants of Feng's algorithm and about an order of magnitude faster than YEN.

3. Variants of Yen's algorithm with guiding and SSSP skipping (dashed lines). This group is the fastest one being about 20% to 40% faster than the respective variant

[2]https://csc.uni-frankfurt.de/wiki/doku.php?id=public:service:goethe-hlr

without guiding. Within this group, variants skipping SSSP computations by deviation length are faster than versions only skipping SSSP computations if the deviation is loopless.

4. Yen-g forms a group on its own (violet line). Even though it is much faster than all variants of Feng's algorithm, it is also in most cases much slower than variants of Yen's algorithm using any sort of SSSP skipping. This behavior is expected since still every SSSP computation needs to be carried out including resource allocations even if only a fraction of the graph is explored.

Comparing variants of Yen's algorithm skipping SSSP computations both with and without guiding, we notice that without guiding, Yen-s-l is slower than Yen-s2 while with guiding it is the other way around meaning Yen-gs-l is faster than Yen-gs2. That Yen-gs-l is faster than Yen-gs2 is supported by our data on the number of SSSP computations that can be skipped discussed in Section 5.2. However, it is possible that many of the SSSP computations skipped by SDL can also be skipped by SSD which is probably why we see such a difference between Yen-gs and Yen-gs2 in some cases. This would also be true if guiding is used in addition but with guiding a non-skipped SSSP computation costs much more time which could be a reason why we see this effect without guiding but not when guiding is used.

We also point out that Yen-s, in [3] called OptYen, is the slowest of both of the fastest groups, with about a factor of 1.5 to 1.9 slower than the new algorithm variants Yen-gs-l and Yen-gs2-l.

The running times on $\mathcal{G}(n,p)$ graph with logarithmic average degree shown in Figure 5.11 (top), are ordered similar compared to the running times on $\mathcal{G}(n,p)$ graphs with a constant average degree. However, the relative runtime differences are smaller. Since the overall running time is much slower for $p = \frac{64}{n}$ and the graph files are quite big, we only tested graphs with up to 2^{26} nodes having close to a billion edges. For $\mathcal{G}(n,p)$ graphs with $p = \frac{4}{n}$ we tested up to $n = 2^{28}$ nodes which then also have about a billion edges.

On grid graphs the situation is a little bit different than on $\mathcal{G}(n,p)$ graphs. We use $\mathrm{Grid}(n,r,p)$ graphs with $n = n_w \cdot n_h$ nodes where n_w is the width of the grid and n_h the height such that $n_w = r \cdot n_h$ holds. Each node has out-edges to the nodes above, below, left, and right to it each with probability p. Specifically, we used $r = 4$ and $p = 0.8$. In order not to draw any advantages from the indexing, in terms of cache efficiency, node indexes are randomized after the graph is generated. Each generated edge gets a uniform random edge weight from $[0;1]$.

Shortest paths in grid graphs have $\Theta(\sqrt{n})$ hops in expectation. This means that the k-shortest path algorithms have to compute a lot more deviations than on graphs with a smaller diameter like $\mathcal{G}(n,p)$ graphs. For comparison, as Figure 5.11 shows for $n = 2^{26}$ nodes running times of up to 28 hours to compute the 50th-shortest path on grid graphs, while it takes less than 7 minutes on sparse $\mathcal{G}(n,p)$ graphs with roughly the same number of nodes and edges.

Figure 5.11 (bottom) shows that Yen-g is already more than an order of magnitude

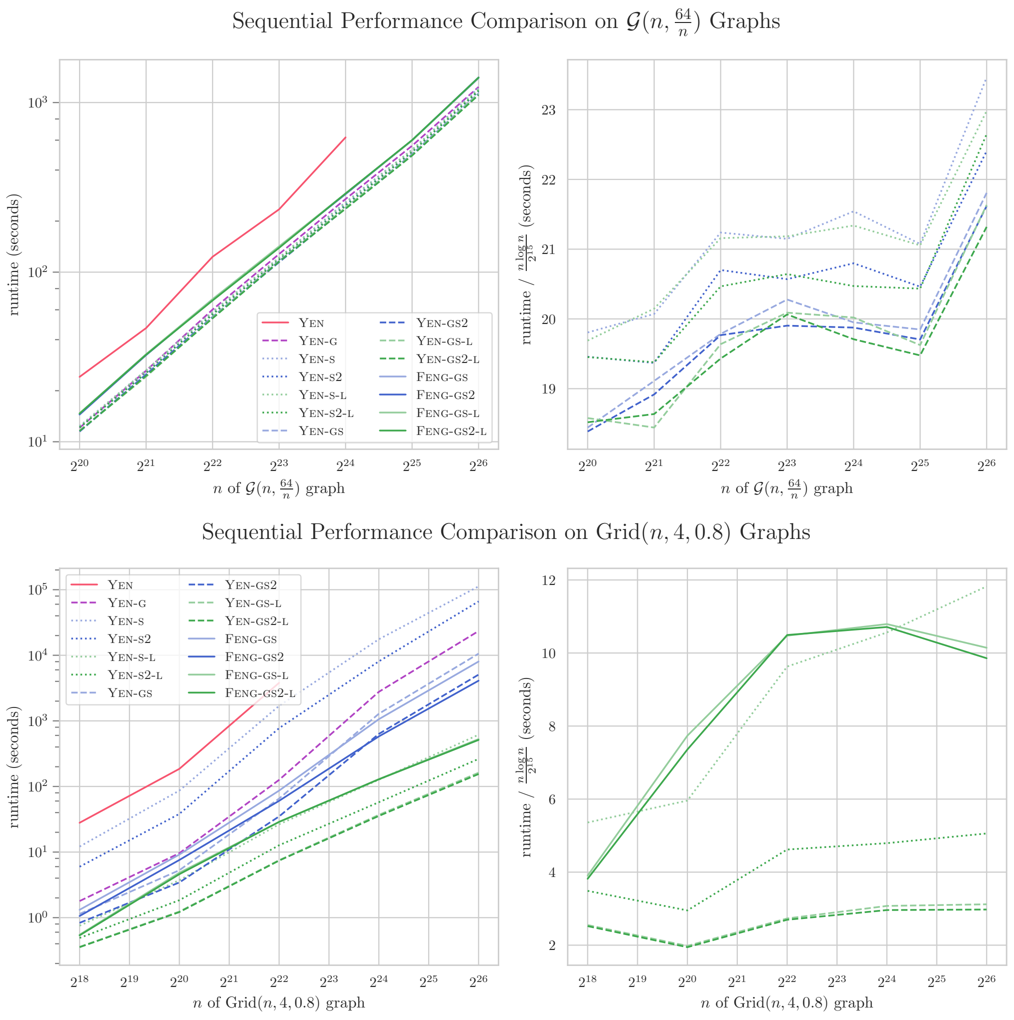


Figure 5.11: *Sequential runtimes of all algorithm variants on denser $\mathcal{G}(n, p)$ graphs with $p = \frac{64}{n}$ (top) and* Grid(n, r, p) *graphs (bottom) showing variants of Feng's algorithm as solid lines, variants of Yen's algorithm with guiding as dashed lines and without guiding as dotted lines, respectively. If skipping by deviation length is used the line is colored green and blue if it is not. All algorithms use early stopping.*

faster than Yen. Skipping SSSP computations by shortest and second-shortest deviation, Yen-s and Yen-s2 respectively, without the graph preprocessing is slower than Yen-g by a factor of at least 4.8 and 2.8, respectively. A major speedup comes from skipping SSSP computations by the length of the shortest and second-shortest deviation. On the two biggest Grid(n, r, p) graphs, algorithm variants using SSSP skipping by length are at least a factor of 4.5 to 140 faster than the respective variant without using SSSP skipping by length. Particularly noteworthy is Yen-s2-l which is more than 250 times faster than Yen-s2 on average on Grid(n, r, p) graphs with $n = 2^{26}$ nodes. When both SSSP skipping by length and graph preprocessing is used the speedup by using the second-shortest deviation is almost negligible, similar to the results on $\mathcal{G}(n, p)$ graphs. As for the $\mathcal{G}(n, p)$ graphs, Yen-gs-l and Yen-gs2-l are the two fastest algorithms, both up to a factor of three faster than Feng's algorithm using the same respective optimizations and about a factor of 10 to 50 faster than Feng's algorithm Feng-gs without further optimizations on bigger graphs.

5.4.3 Comparing Sequential Performance on Undirected Graphs

In Section 3.2.4 we described the KIM algorithm. It is especially designed for undirected graphs and utilizes the properties of undirected graphs in order to achieve a worst-case complexity of $\mathcal{O}(k \cdot \mathrm{spc}(n, m))$, where $\mathrm{spc}(n, m)$ is the worst-case complexity of computing an SSSP tree on a graph with n nodes and m edges. On the i^{th}-shortest path, the KIM algorithm computes at most six SSSP trees in order to compute additional deviation candidates for the next shortest path. But in contrast to Yen's algorithm, the SSSP computations in KIM compute a full SSSP tree each while in Yen's algorithm the SSSP computations can be stopped as soon as the distance to the target node is settled. We showed in Section 5.1 that only a few nodes need to be explored by the SSSP algorithm. We further showed in Section 5.2.3 that almost all SSSP computations can be skipped on directed graphs and about 70% on undirected $\mathcal{G}(n, p)$ graphs when skipping SSSP computations by deviation length. This motivates a direct comparison of the KIM algorithm with some of the variants of Yen's and Feng's algorithm.

We only compare variants of Yen's and Feng's algorithm that are using the SSSP skip by deviation length since Figure 5.9 shows that this type of skip has the biggest impact on undirected graphs. We also add Yen-s and Feng-gs to the comparison since we had both algorithms in a similar comparison in [3].

Figure 5.12 shows that Yen-gs2-l prunes enough work to be about a factor 55 faster than the KIM algorithm on $\mathcal{G}(n, p)$ graphs with $p = \frac{4}{n}$. On $\mathrm{Grid}(n, r, p)$ graphs it is even better, running about 70 to 127 times faster than the KIM algorithm.

The worst-case complexity would suggest a factor of $\mathcal{O}(n)$ between the algorithms. The average-case analysis still suggests a factor of $\mathcal{O}(\log n)$ between Yen's algorithm and the KIM algorithm. However, we can see in Figure 5.12 that the variants of Yen's and Feng's algorithm on one side and the KIM algorithm on the other side seem to differ only by a constant factor. On the grid graphs, we see that there is in fact a non-constant factor between these algorithms when we look at the runtimes of Yen-s and Feng-gs. But for the other algorithm variants the non-constant factor does not show up in the plot.

We point out that we did not reimplement the KIM algorithm and used a C++ implementation from [3] which is why the code of KIM is not in the code repository of this thesis.

5.4.4 Experiments on Real World Graphs

We want to close the chapter on sequential runtime comparisons of k-shortest path algorithm by showing experiments on real world graphs. For this, we chose the Orkut social network [52] as a graph with a small diameter and the European road network as a graph with a larger diameter. The instance of the European road network was made available by PTV group[3]. It is not publicly available but can be obtained on request for research purposes from KIT[4].

[3]https://ptvgroup.com

[4]https://i11www.iti.kit.edu/resources/roadgraphs.php

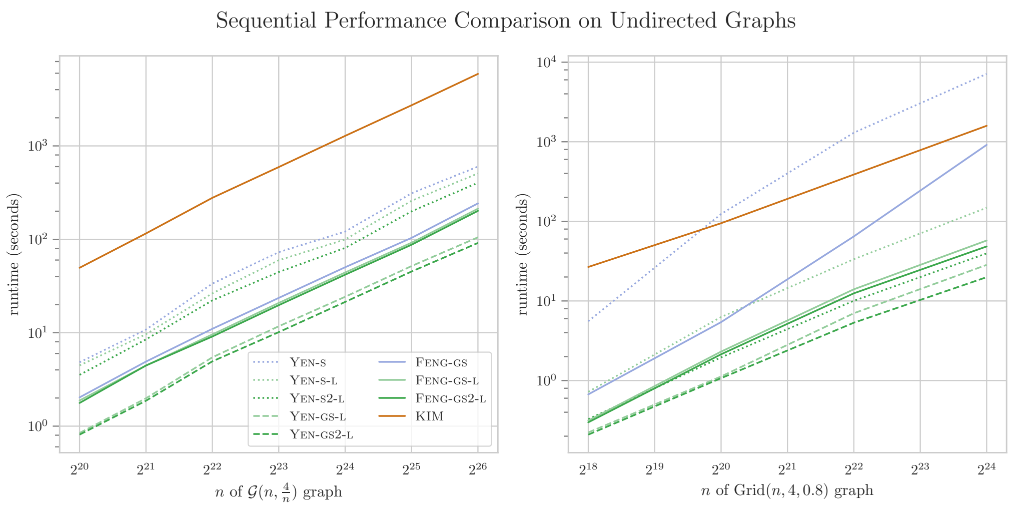


Figure 5.12: *Sequential runtimes of some variants of Yen's and Feng's algorithm as well as the KIM algorithm on undirected $\mathcal{G}(n, p)$ and grid graphs. We see that the variants using SSSP skipping by deviation length outperform the KIM algorithm by one to two orders of magnitude.*

The Orkut network contains about 3 million nodes denoting users and about 117 million edges denoting links between users. The graph is originally unweighted, so in addition to the unweighted graph, we also used five other types of edge-weights:

- Uniform random edge-weights over $[0; 1]$, which we also used in most other experiments.
- The sum of the out-degree of the nodes connected by an edge as well as the inverse of the sum.
- The product of the out-degree of the nodes connected by an edge as well as the inverse of the product.

We normalized all weights to be between zero and one by dividing all edge-weights by the heaviest weight in the graph.

With sum and product of node-degrees, edges between high degree nodes become heavy. This reduces the chance that such edges are part of shortest paths and in turn shortest paths are likely to have more edges. With the inverse of sum and product of node-degrees, edges between high degree nodes are lighter than edges between low degree nodes which should make it more likely for shortest paths to also have only a few edges.

The European road network has about 18 million nodes and about 42.5 million edges. The graph comes with two different sets of edge-weights:

- The physical distance in meters.
- The travel time in seconds assuming a certain average speed depending on the road type varying between 130 km/h on fast motorways down to 10 km/h on gravel roads.

Again, we normalized all edge-weights to fit in the $[0; 1]$ interval by dividing all weight by the biggest edge-weight.

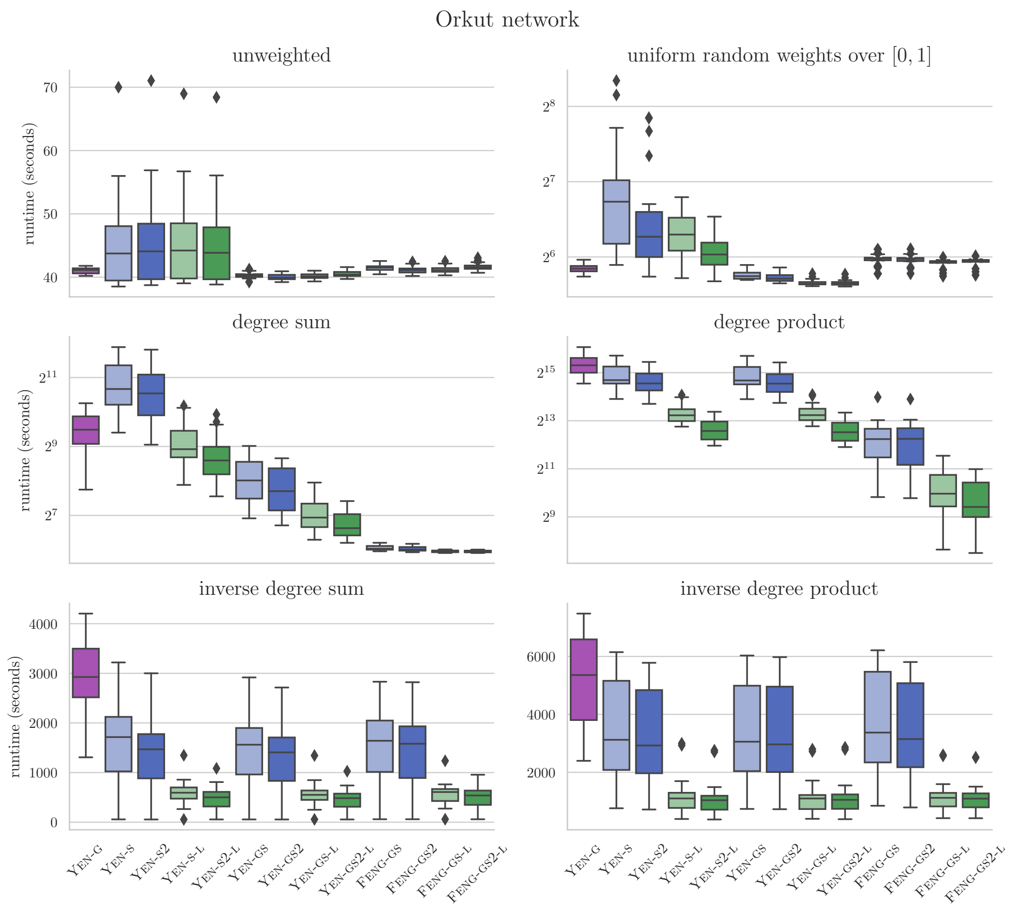


Figure 5.13: *Runtimes of various k-shortest path algorithms on the Orkut social network with six different types of weight assigned to the edges.*

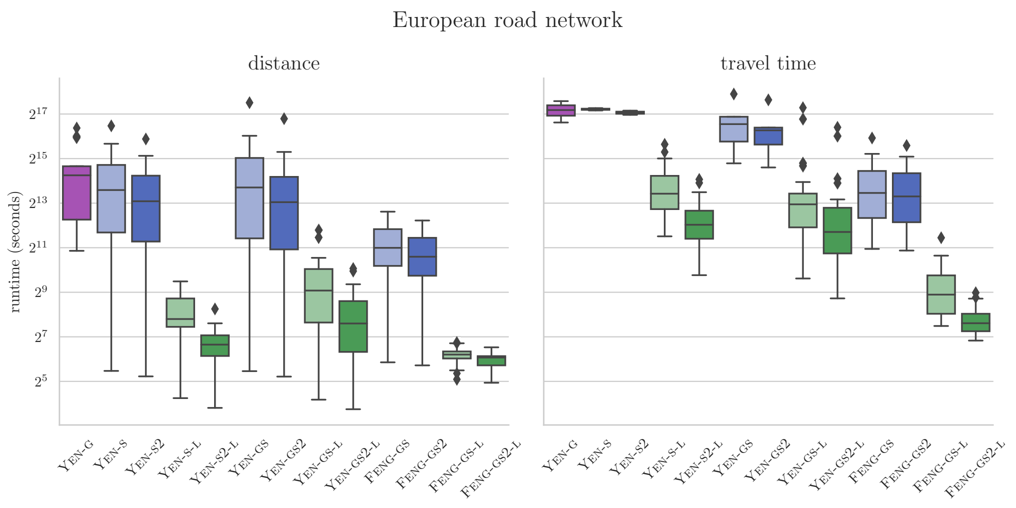


Figure 5.14: *Runtimes of various k-shortest path algorithms on the European road network with distances and travel time as edge weights, respectively.*

In Figure 5.13, we see heavily varying runtimes of the several algorithm depending on the edge weights. On the unweighted graph and the graph with random edge weights, algorithms without the graph preprocessing are by far the slowest while variants of Feng's algorithm are the second fastest and variants of Yen's algorithm using the graph preprocessing are the fastest. On graphs with sum and product of node-degrees as edge-weights, variants of Feng's algorithm are the fastest. On graphs with the inverse of the sum and the product of node-degrees as edge-weights, we see no clear difference between variants of Yen's and Feng's algorithm. On all graphs but the unweighted graph, algorithms skipping SSSP computations by deviation length (green) are always faster compared to the respective algorithms version not using that feature (blue). Keep in mind that we did not adjust the bucket size of Δ-stepping for the non-random egde-weights and instead used $\Delta = 0.01$ as for all other experiments. This could result in higher overall running times of all k-SP algorithms but does not affect the relative performance between them.

Sum and product of node-degrees as edge-weights cause shortest paths to have more edges.

On the European road network, we see that algorithms skipping SSSP computations by deviation length have a clear advantage over their counterparts that do not use this optimization. Figure 5.14 shows speedups of factors 23 to 84 between these variants on distance edge-weights. On travel time edge-weights, we see similar results although the runtimes are overall slower. Here the algorithms FENG-GS-L and FENG-GS2-L stand out, being about 280 and 700 times faster than YEN-S2, respectively. To stress this result a little bit more, FENG-GS2-L took under five minutes to compute the $k = 50$ shortest paths using the travel time edge-weights while Feng's algorithm took well over two hours and YEN-G, YEN-S, and YEN-S2 took over one and a half day. We want to point out, that the latter three algorithms did not finish all runs due to runtimes of over 48 hours of some runs, at which point we canceled the runs.

Feng's algorithm is denoted by FENG-GS in the plots.

The results on the European road network are consistent with the results on grid graphs in the sense that the algorithms skipping SSSP computations by the length of the shortest deviations outperform all other algorithms. But on the road network variants of Feng's algorithm, FENG-GS-L and FENG-GS2-L, are the fastest while they are up to a factor of three slower than YEN-GS-L and YEN-GS2-L on grid graphs. The experiment on the European road network with distances as edge-weight also show that the algorithm YEN-S2-L is faster than YEN-GS2-L where the only difference is that YEN-S2-L does not use the graph preprocessing.

5.4.5 Conclusion

All our experiments show that both Yen's and Feng's algorithm benefit from our improved heuristics to skip SSSP computations. Particularly skipping SSSP computation by the length of non-simple deviations, SDL and SSDL, speeds up the running times both on synthetic graphs as well as on real world graphs, especially on graphs with a larger diameter like grid graphs or road networks.

Heuristics: ☞ Sections 5.2.1 and 5.2.2

We also demonstrated that computing the node coloring of Feng's algorithm can be slower than just relying on the graph preprocessing combined with stopping the SSSP computation as soon as the target node is found. We see this behavior on $\mathcal{G}(n,p)$ graphs, where the k shortest paths have only $\mathcal{O}(\log n)$ hops whp. The node coloring does not only limit the nodes explored by the SSSP algorithm but also allows to skip an SSSP computation in constant time. Without the node coloring skipping SSSP computations takes linear time in the number of hops of the current shortest path. This is why we see that on $\mathcal{G}(n,p)$ graphs all versions of Feng's algorithm are slower than the corresponding versions of Yen's algorithm. On grid graphs where the k shortest paths have $\Theta(\sqrt{n})$ hops whp., we see the advantage of the node coloring. On the Orkut network the algorithms skipping SSSP computations by deviation length are always faster than the corresponding ones not using this feature except on the unweighted graph. We also see that the node coloring is only beneficial on the graphs with sum and product of out-degrees as edge-weights. Using these edge-weight functions, shortest paths have more edges. This fits well in the picture with the European road network. Here the algorithms using the node coloring are always faster than the corresponding algorithm without the node coloring.

Results on $\mathcal{G}(n,p)$ and $\mathrm{Grid}(n,r,p)$ graphs: ☞ *Figures 5.10 and 5.11*

In summary our experiments demonstrated that our new heuristics, namely SSD, SDL, and SSDL, can speed up Yen's and Feng's algorithms significantly. They show that on graphs with small diameter it is even faster not to compute Feng's node coloring and use Yen-gs2-l instead. On graphs with larger diameters, our new heuristics without the node coloring can still outperform Feng's algorithm. However, combining our new heuristics with the node coloring can be even faster as the experiment on the Orkut network and the European road network shows.

Results on the real world graphs: ☞ *Figures 5.13 and 5.14*

If nothing is know about the input graph, Yen-gs2-l would be a good choice since it is the fastest algorithm on all the synthetic graphs. However, Feng-gs2-l would be also worth a try, since it is the fastest algorithm on the European road network, where it is about 17 times faster than Yen-gs2-l with travel times as edge weights. On the synthetic graphs, Yen-gs2-l is faster than Feng-gs2-l by up to a factor of seven. All other algorithms mentioned are probably not relevant in practice and only show the influence of the individual optimizations on the performance.

6 Empirical Comparison of k-Shortest Path Algorithms on Multicores

In Chapter 4 we showed that the average-case time complexity of Yen's algorithm and Feng's algorithm are much better than their worst-case complexity we briefly described in Chapter 3. We then presented in Chapter 5 multiple variants of both algorithms using additional heuristics to speed up the sequential performance. It turned out that we can skip most SSSP computations completely and on the few remaining SSSP computations only a tiny fraction of the nodes needs to be explored by the SSSP algorithm.

Algorithm variants: ☞ *Table 5.1*

In addition to these optimizations on sequential performance we also parallelized the presented algorithms. In this chapter, we discuss the parallelization variants, implementation details, and present results of the performance experiments.

6.1 Parallelization Strategies

We consider three strategies to parallelize the k-shortest path algorithms.

- **Parallel Shortest Paths**: The easiest way to parallelize any k-SP algorithm is to use a parallel SSSP algorithm. We use the parallel Δ-stepping algorithm. There are other options, e.g., Radius-Stepping [11], but Δ-stepping does not require a preprocessing of the graph, aside from partitioning the edges by weight into light and heavy edges.

Details on Δ-stepping: ☞ *Sections 2.3.2 and 6.2.2*

- **Parallel Deviations**: While all deviations of the i^{th}-shortest path need to be computed before the deviations of the $(i+1)^{\text{th}}$-shortest path, the order of the deviations of the i^{th}-shortest path is arbitrary and the deviations are independent of each other. This allows to compute them in parallel using a sequential SSSP algorithm.
- **Mixed Strategy**: Instead of using only one of the strategies above, we can use both at the same time. Nested parallelism has no benefit over non-nested parallelism but if none of the two other strategies can utilize all available threads by itself, the mixed strategy could be a good option.

Which parallelization strategy works best depends on the expected number of SSSP computations and the number of nodes explored during an SSSP computation. As we observed in Section 5.1 for $\mathcal{G}(n,p)$ and $\mathrm{Grid}(n,r,p)$ graphs, only a tiny fraction of the nodes is explored by the SSSP algorithm. So the overhead from using a parallel SSSP algorithm might be bigger than the speedup due to the parallel execution.

On $\mathcal{G}(n,p)$ graphs with $n = 2^{24}$, the paths have about 20 hops each. So most of the time, we do not have more than a single SSSP computation.

On $\mathrm{Grid}(n, r, p)$ graphs with $n = 2^{24}$ nodes, the paths have about 2^{12} hops each. So still about 250 (6% of 2^{12}) SSSP computations need to be carried out.

For parallel deviations we have a similar situation on $\mathcal{G}(n, p)$ graphs. We saw in Section 5.2 that most of the SSSP computations can be skipped completely on $\mathcal{G}(n, p)$ ($> 96\%$) and $\mathrm{Grid}(n, r, p)$ (92% – 96%) graphs. On $\mathcal{G}(n, p)$ graphs this leaves only limited potential to compute deviations in parallel, since the number of deviations computed on each shortest path is only logarithmic whp. On $\mathrm{Grid}(n, r, p)$ graphs we also only have about 6% of the SSSP computations to be actually carried out, but since the paths have $\Theta(\sqrt{n})$ hops whp., there are still enough SSSP computations left to execute. So in Section 6.4 we will use grid graphs to ensure that we can see effects of the parallelization.

6.2 Implementation Details on Parallelization

In order to get an efficient parallelization of our algorithm, we need to avoid frequent use of slow locking mechanisms, which can be utilized to prevent multiple threads writing simultaneously to the same memory address. Instead, we use additional data structures to separate the data used by the individual threads described in more detail in the following two sections.

6.2.1 Implementation

Notes on the code:
☞ Section 5.3.5

In addition to the sequential versions of the algorithms in Table 5.1 we implemented parallel versions using OpenMP 4.5 that part of the GNU Compiler Collection 10.3[1]. We reused as much code as possible but added separate parallel implementations of all functions that would introduce unnecessary overhead when used with just a single thread.

6.2.2 Implementation Details on Δ-Stepping Parallelization

We use Δ-stepping as the SSSP algorithm as described in Section 2.3.2. Here we want to point out some implementation details concerning the parallelization and the overhead they come with.

Δ-stepping uses a bucket list to organize nodes according to their tentative distances. Each bucket has a so called bucket width of Δ, i.e., the i^{th} bucket contains nodes with a tentative distance between $i \cdot \Delta$ and $(i + 1) \cdot \Delta$. Within the buckets the nodes are not sorted in any way. The out-edges of the nodes in the current bucket are relaxed in two phases:

1. Light edges e with an edge weight of $\mathrm{d}(e) < \Delta$. These edges might introduce new nodes to the current bucket or even reintroduce nodes to the bucket.

2. Heavy edges e with an edge weight of $\mathrm{d}(e) \geq \Delta$. These edges cannot introduce nodes to the current bucket. Thus they only have to be relaxed once for every node that was in the bucket even if it was reinserted multiple times into the bucket as long as heavy edges are relaxed after all light edges have been relaxed.

[1]https://gcc.gnu.org/

Within each phase, the order of the relaxations is arbitrary and can be carried out in parallel. In order to prevent multiple threads updating the same nodes distance at the same time, we split up the buckets and use a mapping to assign nodes to threads. Let τ be the number of threads. A bucket B_i is then split up into τ subbuckets $B_{i,0}, \ldots, B_{i,\tau-1}$ where each node v is assigned to bucket $B_{i,h(v)}$ by a hash function h. Since we already look at random graphs, we used $h(v) = v \mod \tau$ as a hash function. With this each thread can maintain its own part of the bucket independently and we expect all subbuckets to be about evenly filled since we run it on random graphs with random node IDs. Relaxing all out-edges of nodes in the current bucket right away would possibly lead to multiple threads updating the same nodes distance, so instead each node stores a relaxation request in a request list R. The request list R consists of τ^2 sublists $R_{i,j}$. This way the i^{th} thread can store for each edge $e = (u, v)$ the relaxation request in the sublist $R_{i,h(v)}$ without interfering with other threads. After all threads have placed all their requests, the i^{th} thread executes all requests in the sublists $R_{0,i}, \ldots, R_{\tau-1,i}$. This way each thread relaxes only edges that target nodes the thread is assigned to.

$h(v) = v \mod \tau$ will probably not assign the nodes evenly on non-random node indexes. For non-random node indexes another hash function should be used. Computing $\mod \tau$ is also rather slow in practice if τ is not a power of 2. Alternatively one could randomize the node indexes, but this could result in more cache misses.

Relaxing heavy edges works the same way. The algorithm remembers the nodes that were once scanned in the current bucket by storing them in a node cache. This cache consists of an array containing flags indicating if a node is already in the cache as well as a subcache for each thread, to store the actual nodes. The vector of flags is needed to be able to quickly look up if a node is in the cache, while the subcaches are needed to quickly iterate over the stored nodes only without iterating over all other nodes too.

The array of flags stores actually 8 bit integers instead of booleans since arrays of booleans are not thread-safe in C++. In order to minimize the memory of Boolean vectors, C++ stores them as bigger integers and maps the index in the array to a bit in one of these integers. So concurrent write operations on different indexes close enough to be in the same underlying integer could overwrite each other.

With this implementation we have a lock free parallelization only using barriers between phases of the algorithm. However, compared to the sequential implementation we also have some overhead depending on the number of threads.

1. Memory overhead due to the thread-safe node cache, which increases the memory footprint by a factor of 8.
2. Computational overhead due to the node-to-thread assignment using the hash function. This could also be solved by generating a random mapping once and storing it in an array. But it would still take time to look up values in an array at random positions with potential cache misses as well as additional $\Theta(n)$ memory.
3. Initializing and maintaining $\Theta(\tau^2)$ arrays. Spreading requests over many different arrays can lead to a higher memory consumption and additional cache misses. This gets more expensive with the number of threads.

If there is enough work to do, the speedup due to parallelization is bigger than the time loss due to the overheads.

On $\mathcal{G}(n, p)$ and $\mathrm{Grid}(n, r, p)$ graphs only a tiny fraction of the nodes is explored during an SSSP computation, as we showed in Section 5.1, i.e., we cannot expect a parallel Δ-stepping to scale well on these graphs and we see in fact the expected behavior in Figure 6.1. But there might be graphs, graph classes, or edge-weight distributions where a lot more nodes need to be explored.

6.2.3 Implementation Details on k-Shortest Path Parallelization

All deviations of the i^{th}-shortest path can be computed independently of each other. But for each deviation a slightly different temporary graph $G_j^{(i)}$ in case of Yen's algorithm or yellow graph $Y_j^{(i)}$ in case of Feng's algorithm is needed. For this reason each thread needs to maintain its own temporary graph $G_j^{(i)}$ or yellow graph $Y_j^{(i)}$, respectively. As described in Section 5.3, we implemented temporary and yellow graphs by additional arrays on top of the original graph to mark nodes as removed from the graph or storing a nodes color, respectively. Even though the additional array comprises only n bits, or n bytes for the yellow graph, respectively, it scales linearly with the number of treads. This can be expensive in terms of memory if a lot of threads are used.

During the execution the deviations are assigned dynamically to the threads. We saw in Section 5.2 that for most of the deviations the respective SSSP computation can be skipped. Since we do not know up front for which deviation the SSSP computation can be skipped, we use the slower method of assigning jobs dynamically to a thread whenever it has finished its last one. This way we get a better load balancing. The alternative would be assigning the threads to the deviations at the beginning, risking that a single thread needs to execute multiple SSSP computations while other threads skip all SSSP computations and in turn wait most of the time.

6.3 Experimental Setup

Hyper-Threading cores have their own cache but share the ALU with a physical core. So either a physical core or its Hyper-Threading core can compute at the same time.

In Chapter 5 we presented a sequential comparison of 14 algorithm variants in total. For our parallel comparison we exclude Yen, Yen-g, Yen-s, Yen-s2, Yen-s-l, and Yen-s2-l as the other algorithms ran consistently faster.

Section 5.2 showed that on $\mathcal{G}(n,p)$ instances graph preprocessing and SSSP skipping both prune almost all of the computational work that could be parallelized which is why we ran the parallelization experiments on $\mathrm{Grid}(n,r,p)$ graphs only.

Non-Uniform Memory Access, NUMA for short, is used in multi processor setups. It basically means that different CPUs in the machine have direct access to only certain chunks of the memory thus access to other chunks is slower.

We consider a shared memory setting where all processing units have access to the full memory. Multiple processing units can read simultaneously from the same memory address. But only one processing unit can write to a memory address at a time.

Our experiments ran on machines of the Goethe-HLR[2] compute cluster. The machines contain 196 GB of internal memory and two Intel Xeon Skylake Gold 6148 each having 20 physical cores and 20 Hyper-Threading cores. Although each machine had two processors, we locked the experiments to use only one CPU in order to avoid NUMA effects in the results which also halves the available RAM. We ran our experiments on multiple machines all having an identical hard- and software setup in order to keep the total time manageable. Note that we did not tune the implementations to any specific technical details on the described machines.

[2]https://csc.uni-frankfurt.de/wiki/doku.php?id=public:service:goethe-hlr

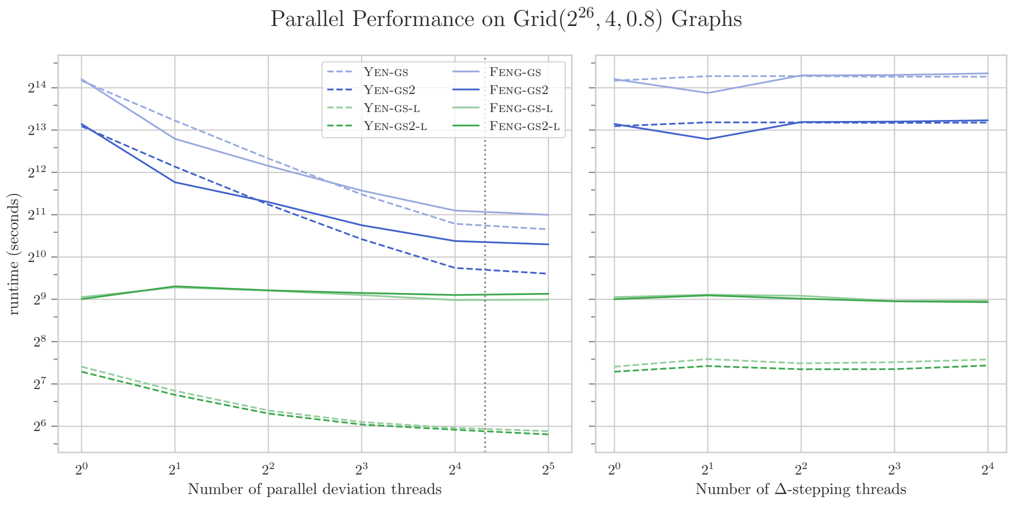


Figure 6.1: *Runtimes of several algorithm variants using multiple threads for deviations (left) and Δ-stepping (right). The vertical dotted line indicates the number of physical cores of the processor. As expected, we see a speedup for parallel deviations but not for parallel SSSP computations.*

6.4 Experimental Results

Figure 6.1 shows that parallel deviations work better for algorithm variants that skip less SSSP computations like Yen-GS, Yen-GS2, Feng-GS, and Feng-GS2. For these algorithm variants the running time almost halves when doubling the number of threads. For variants skipping SSSP computations by deviation length, much less SSSP computations are required and thus the parallel deviations scale not quite as well for algorithm variants Yen-GS-L and Yen-GS2-L. But still the runtime can be sped up from about 3.2 minutes using a single thread to just under a minute using 16 threads. For the variants Feng-GS-L and Feng-GS2-L we cannot see any speedup at all. This is due to the little potential for speedup by parallel deviations while simultaneously maintaining the yellow graphs takes longer than maintaining only a temporary graph like in the variants of Yen's algorithm. Keep in mind that the machines we used only have 20 physical cores which is why we see a slightly weaker speedup for 32 cores, where 12 Hyper-Threading cores are used.

As expected we observe no speedup when using parallel Δ-stepping, since we expect to only explore a few hundred nodes of the graph. So there is not enough work left to be parallelized. On the contrary we even see for Yen-GS-L and Yen-GS2-L that they become slightly slower when parallel Δ-stepping is used with 16 threads due to the reasons discussed in Section 6.2.2.

Since we do not see a speedup when using parallel Δ-stepping, we did not include experiments with the mixed strategy mentioned before.

7 Conclusion

7.1 Summary

7.1.1 Theoretical Results

In Chapter 4, the theoretical part of this thesis, we prove in Theorem 4.5 that the average-case complexity of Yen's algorithm is $\mathcal{O}(k \log(n) \cdot \operatorname{spc}(n, m))$ on $\mathcal{G}(n, p)$ graphs with at least logarithmic average-degree and a variety of edge-weight distributions following some assumptions. We get a slightly weaker average-case complexity in Theorem 4.7 of $\mathcal{O}\Big(k \cdot \frac{\log^2 n}{np} \cdot \operatorname{spc}(n, m)\Big)$ for Yen's algorithm on sparse graphs with a constant average-degree but only for uniform random edge-weights over $[0; 1]$. At the core of both proofs, we confirm that enough short s–t-paths exist in terms of weight and from this we can show that enough short s–t-paths exist in terms of hops.

$\operatorname{spc}(n, m)$ is the average-case complexity of the SSSP algorithm in use.

Assumptions on edge-weight distributions can be found on page 31.

Both results also hold for Feng's algorithm since it is basically a heuristic on top of Yen's algorithm. However, we prove in Theorem 4.9 an even better average-case complexity of $\mathcal{O}(k \cdot \operatorname{spc}(n, m))$ for Feng's algorithm on unweighted $\mathcal{G}(n, p)$ graphs with a logarithmic average-degree and constant values of k. For the proof, we analyze the size of subtrees of the reverse BFS tree rooted in the target node t. We show that the sizes of the subtrees hanging from nodes within the same BFS level differ only by a logarithmic factor whp.

We close Chapter 4 by providing empirical evidence that Theorem 4.9 should also hold on weighted $\mathcal{G}(n, p)$ graphs with uniform random edge-weights over $[0; 1]$ and logarithmic as well as constant average-degree.

7.1.2 Empirical Results and Practical Improvements

In Chapter 5, the practical part of the thesis, we suggest several new heuristics to improve the sequential runtime of Yen's and Feng's algorithms.

First, we observe in Section 5.1 that an SSSP algorithm like Δ-stepping only explores a tiny fraction of the graph in order to find a shortest deviation path when the graph preprocessing described by Feng is used. This allows to stop the SSSP computations early in case it needs to be executed at all. In Section 5.2.3 we demonstrate that most SSSP computations can be skipped completely by pulling a shortest deviation path from a precomputed reverse shortest path tree. We also describe how to extend this approach in order to also pull the second-shortest path from the reverse shortest-path tree trying to skip even more SSSP computations. In total, our experiments show that about 94% of

SSSP computations can be skipped on Grid graphs, about 98% on sparse $\mathcal{G}(n,p)$ graphs with a constant average-degree, and over 99% on $\mathcal{G}(n,p)$ graphs with a logarithmic average-degree.

All weighted graphs in our experiments had uniform random edge-weights over $[0;1]$.

We implemented[1] all algorithms and heuristics presented in Chapter 5 and evaluated them in Section 5.4 on $\mathcal{G}(n,p)$ and Grid graphs as well as some real world graphs. Our experiments show that our new heuristics significantly speedup Yen's and Feng's algorithms. On Grid graphs we see a speedup by a factor of up to 40 compared to Feng's algorithm.

The practical part closes with Chapter 6 about parallelizing the described k-shortest path algorithms. We consider two strategies for parallelization: Computing deviation paths in parallel and using a parallel SSSP algorithm. For the parallel deviations we demonstrate speedups of a factor two up to eight on a CPU with 16 cores on Grid graphs. However, all the optimizations described in the sequential part lead to the SSSP algorithm exploring only a few hundred nodes if it is not skipped entirely. So we do not see a speedup when using a parallel SSSP algorithm.

7.2 Future Work and Open Questions

We want to point out some open questions in the k-shortest path domain.

7.2.1 Extended Average-Case Complexity Analysis

Average-case analysis: ☞ *Chapter 4*

Regarding our average-case analysis, we believe that Theorem 4.7, on the average-case complexity of Yen's algorithm on sparse graphs, holds not only for uniform random edge-weights over $[0;1]$ but also for more general edge-weight distributions as used in Theorem 4.5. To close this gap many proofs from [51] need to be extended for more general edge-weight distributions. In Theorem 4.9, we show the improved average-case complexity of Feng's algorithm for a constant k on *unweighted* graphs. Furthermore, we provide empirical evidence that the same result should hold for weighted graphs as well. However, proving that seems to require deeper insights into the structure of shortest-path trees.

Algorithm variants: ☞ *Chapter 6 and Table 5.1*

Alongside the average-case analysis, we also suggest some algorithm variants using heuristics to prune off most of the k-shortest path subroutine computations. Using these heuristics, it could be possible to prove an even better average-case complexity.

7.2.2 Smoothed Complexity Analysis

Performance experiments: ☞ *Section 5.4*

We observe in our experiments that all considered algorithms behave differently depending on the number of edges on the k shortest paths. Therefore, it seems worth taking a look at the smoothed complexity of the k-shortest path problem. Smoothed analysis is a field introduced by Spielman and Teng [67] in order to bring theoretical complexity analysis and performance closer together in practice. The smoothed analysis identifies

[1]The implementation is publicly available at https://doi.org/10.5281/zenodo.7713239.

relevant parameters of input data the overall complexity depends on. The analysis is done with respect to these parameters.

7.2.3 Extensive Practical Comparison

A field we could not fully cover is a complete comparison between all the current state of the art k-shortest path algorithms. This is a huge task since there are at least four algorithms, namely Feng's algorithm [24], the algorithm by Kurz and Mutzel [41], the algorithm by Chen et al. [12], and at least one of the new algorithms we presented in Chapter 5. Some of our algorithms run faster on certain undirected graphs than the KIM algorithm beside its worst-case complexity being nearly a factor of n better than Yen's and Feng's algorithm. Therefore, even though these algorithms are made for directed graphs, they should be compared to the KIM algorithm [40] on undirected graphs as well. In order to make a fair comparison, all these algorithms need to be implemented with a similar level of code quality and optimization. We support this by making our implementations publicly available under an open source license. The comparison then needs to be done on multiple graph classes with various sizes since we already saw in our comparison that no one single algorithm performs best on all graph classes. In addition, the underlying data structures to represent the resulting paths partially depend on the value of k used and have different advantages and disadvantages in terms of memory consumption and runtimes. While we use smaller k values in the order of $\log n$, some other authors use k values in the range of $\sqrt{n}$ in their experiments. Even though both are valid choices, differences in the size of k make it even harder to compare algorithms across multiple papers. All algorithms should also use a common SSSP algorithm whenever appropriate. For some of the k-shortest path algorithms the SSSP algorithm needs to support some features that could reduce performance in k-shortest path algorithms not requiring these features. Such functions could be stopping the SSSP computation when the shortest path to the target node is found or when it is certain that the target node is too far away to be relevant. In our case, the choice of Δ-stepping as default SSSP routine introduces an additional parameter that could make comparisons even harder. Finally, such a comparison should also take parallelism into account. The variants of Yen's algorithm we show in Chapter 5 can be parallelized rather easily. However, we did not check if the same technique can be used for the other state of the art algorithms or if these algorithms would profit more from a parallel SSSP algorithm or a completely different technique. All these factors make a meaningful comparison a long and extensive task on its own which is why we decided to leave this for future work.

Comparison on undirected graphs:
☞ *Section 5.4.3*

Discussion of our choice for the size of k:
☞ *Section 5.4.1*

Appendix

A

A.1 List of Notations

If not stated otherwise the following notations are used.

A.1.1 General Notation

$\mathrm{d}(e), \mathrm{d}(u, v)$	p. 7	The weight of an edge $e = (u, v)$.
$\mathrm{d}(P)$	p. 7	The weight-length of a path P.
$\overline{\mathrm{d}}(P)$	p. 7	The hop-length of a path P.
$\lvert P \rvert$	p. 7	The number of nodes on a path. $\overline{\mathrm{d}}(P) = \lvert P \rvert - 1$ holds.
$u \to v$	p. 7	An edge from node u to node v as an alternative to (u, v).
$u \dashrightarrow v$	p. 7	A shortest path from node u to node v in terms of weight.
$u \twoheadrightarrow v$	p. 7	A shortest path from node u to node v in terms of hops.
$P \circ Q$	p. 8	Join to paths P and Q. We use this notation loosely, so if the head node of P is not the same as the tail node of Q the two paths are joined by the respective edge.

A.1.2 k-Shortest Path Related Notation

P_i	p. 13	The i^{th}-shortest path.
$v_j^{(i)}$	p. 13	The j^{th} node of the i^{th}-shortest path.
$\mathrm{R}_i(j)$	p. 13	Prefix of the i^{th}-shortest path up to and including the j^{th} node. $\mathrm{R}_i(j) = \left(v_1^{(i)}, \ldots, v_j^{(i)}\right)$
$\mathrm{S}_i(j)$	p. 13	Suffix of the i^{th}-shortest path from the j^{th} node on. $\mathrm{S}_i(j) = \left(v_j^{(i)}, \ldots, v_{\lvert P_i \rvert}^{(i)}\right)$
$\mathrm{par}(P_i)$	p. 13	Parent path of the i^{th}-shortest path P_i. That is the path P_i deviated from.So if $P_l = \mathrm{par}(P_i)$ holds then $\mathrm{R}_i(\mathrm{di}(P_i)) = \mathrm{R}_l(\mathrm{di}(P_i))$ and $\mathrm{R}_i(\mathrm{di}(P_i) + 1) \neq \mathrm{R}_l(\mathrm{di}(P_i) + 1)$ holds, too.
$\mathrm{dev}(P_i)$	p. 13	The node at which P_i deviated from its parent path.
$\mathrm{di}(P_i)$	p. 13	The index of the node at which P_i deviated from it parent path. So $v_{\mathrm{di}(P_i)}^{(i)} = \mathrm{dev}(P_i)$ holds.

A.2 Experiments

This section contains some additional details about the experiments presented throughout the thesis.

A.2.1 Graph Classes

We mostly use $\mathcal{G}(n, p)$ and $\mathrm{Grid}(n, r, p)$ graphs for our experiments. This section contains some details about these graph types. The code we used to generate the graphs for our experiments is also part of the code repository.

A.2.1.1 $\mathcal{G}(n, p)$ and $\mathcal{G}(n, m)$ Graphs

The $\mathcal{G}(n, p)$ model introduced by Gilbert [30] describes a random graph with exactly n nodes where each of the $n \cdot (n-1)$ possible edges has a probability of p to exist. This model is closely related to the $\mathcal{G}(n, m)$ model introduced by Erdős and Rényi [22] where exactly m of the $n \cdot (n-1)$ possible edges are chosen. For $n \to \infty$ both models are identical. Both models also work for undirected graphs.

In our experiments we only use $\mathcal{G}(n, p)$ graphs both directed and undirected depending on the respective setting. We also introduce q-directed graphs for $0 \leq q \leq 1$, where each (u, v) has a chance of $1 - q$ to be undirected, meaning that the edge (v, u) also exists in the graph and, if the graph is weighted, both edges share the same edge weight. So 0-directed graphs are the same as undirected graphs and 1-directed graphs are directed graphs complying to the respective model. Note that q-directed graphs for $0 < q < 1$ are also directed graphs but do not fully comply with the respective model. However, they form sort of a bridge between directed and undirected graphs in terms of the properties of shortest paths.

In our experiments we use graphs from $n = 2^{20}$ nodes to $n = 2^{28}$ nodes. For such sizes $\mathcal{G}(n, p)$ graphs look statistically all the same which is why we did not generate new random graphs for each experiment. Instead we generated one $\mathcal{G}(n, p)$ graph of each size and only chose random source and target nodes.

$\mathcal{G}(n, p)$ graphs are widely used for average-case analysis and probabilistic existence proofs. However, they also have some properties like a logarithmic diameter as Priebe [58] showed, so even though there is a positive probability for any graph with a certain number of nodes and edges to be drawn, we will see graphs with logarithmic diameters whp.

A.2.1.2 $\mathrm{Grid}(n, r, p)$ Graphs

An other important graph type is grids, which we use to simulate e.g., road networks or other real world graphs with a large diameter. The $\mathrm{Grid}(n, r, p)$ graphs we use are two dimensional grids consisting of n nodes layed out in n_r rows and n_c columns such that both $\frac{n_r}{n_c} = r$ and $n_r \cdot n_c = n$ holds. Each node has potential out-edges to the four neighboring nodes directly to the left, right, top, and below each with probability p. The top row is not connected to the bottom row and the same holds for the left most and right most column.

A.2.2 Hardware and Software

All performance experiments were done on a subset of servers of the Goethe-HLR compute cluster at Goethe – University Frankfurt. All servers used were identical in terms of hardware and software. Each server had two Intel Xeon Gold 6148 (Skylake) CPUs having 20 physical cores and 20 additional hyper threading cores and 196 GB of RAM. All servers run Scientific Linux 7.6.

We implemented all code in C++20 with OpenMP 4.5 for parallelization using the GNU Compiler Collection[1] in version 10.3 to compile our code. Even though we only used CPUs by Intel, the code is not specialized to work best with these exact CPUs whatsoever.

The code is publicly available at

https://doi.org/10.5281/zenodo.7713239

under GLPv3 open source license. Besides the k-shortest path algorithms the repository contains code to generate $\mathcal{G}(n, p)$ and $\mathrm{Grid}(n, r, p)$ graphs as well as the raw data of our experiments and the code to create the plots.

A.2.3 Metadata of the Experiments

All experiments we did followed a similar pattern. We created for each graph class and size a single random graph and reduced it to the largest connected component. This removes only a very small amount of nodes from the graph and in turn saves us a lot of error handling for the case that the target node is not reachable. We then generated for each graph class and size twenty random source and target nodes to run all the algorithms on to address for varying results due to the random choices of source and target node as well as smaller variations due to the machines and other software running on them. The only exception are the runtimes we show on the European road network in Section 5.4.4, where we attempted twenty runs but some of the slowest did not finish due to extremely high running times. Since all algorithms ran on the same set of source-target pairs, the results show sometimes similar bumps in the plots for different algorithms.

The machines were exclusively used for the experiment but still runs the operation system and management processes.

Even though we not always explicitly mentioned it, we always calculated $k = 50$ paths as discussed in Section 5.4.1.

A.2.4 Plot Types

All plots we showed were created using seaborn [73] and Jupyter[2]. We use mostly two types of plots to present the results of our experiments, namely line plots and box plots.

[1]https://gcc.gnu.org/

[2]https://jupyter.org/

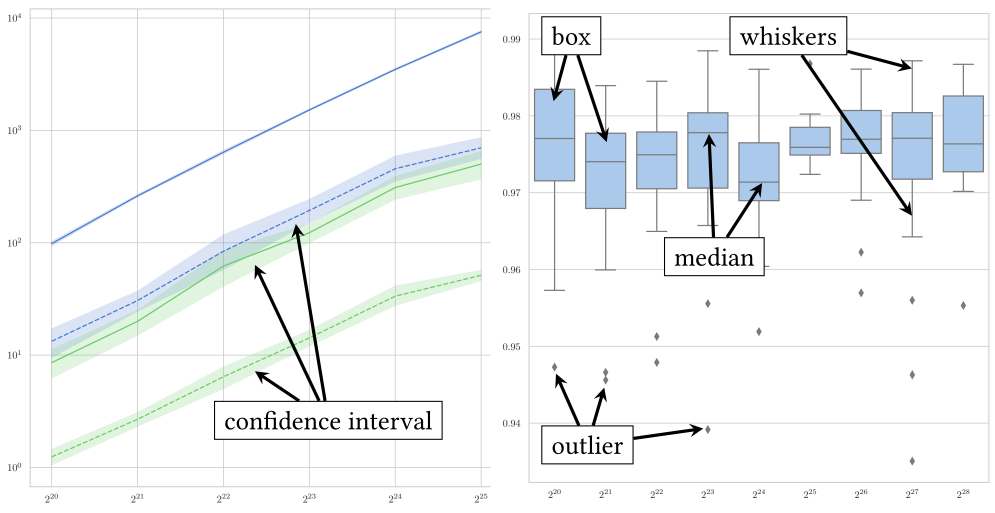


Figure A.1: Example line plot (left) and box plot (right).

A.2.4.1 Line Plots

Our line plots always show the mean of the respective data set. In some line plots, in addition to the line itself, an area around each line in the same color is shown. These areas show the 95% confidence interval for the shown line computed from the data. See Figure A.1 for an example. We omit this area in most plots since it makes the plot hard to read when there are too many lines.

A.2.4.2 Box Plots

The box plots convey much more information than the line plots. The lower and upper end of the box denot the 25% and 75% quantile of the underlying data, the line in the box denotes the median. The whiskers extend to 1.5 times the inter quantile range above and beyond the upper and lower end of the box, respectively. All data points further away are considered outliers represented as diamond shapes. See Figure A.1 for an example.

Bibliography

[1] U. Agarwal and V. Ramachandran. Finding k Simple Shortest Paths and Cycles. In S.-H. Hong, editor, *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ISAAC.2016.8.

[2] U. Agarwal and V. Ramachandran. Fine-grained complexity for sparse graphs. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 239–252, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3188745.3188888.

[3] D. Ajwani, E. Duriakova, N. Hurley, U. Meyer, and A. Schickedanz. An empirical comparison of k-shortest simple path algorithms on multicores. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*, pages 78:1–78:12. ACM, 2018. doi:10.1145/3225058.3225075.

[4] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29(1), Feb. 2015. doi:10.1609/aaai.v29i1.9154.

[5] H. Aljazzar and S. Leue. K^*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, 175(18):2129–2154, 2011. doi:10.1016/j.artint.2011.07.003.

[6] Z. ALzaid, S. Bhowmik, and X. Yuan. Multi-path routing in the jellyfish network. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 832–841, 2021. doi:10.1109/IPDPSW52791.2021.00124.

[7] H. Aprahamian, D. R. Bish, and E. K. Bish. Optimal risk-based group testing. *Management Science*, 65(9):4365–4384, 2019. doi:10.1287/mnsc.2018.3138.

[8] S. Banerjee, P. Mitra, and K. Sugiyama. Multi-document abstractive summarization using ilp based multi-sentence compression. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1208–1214. AAAI Press, 2015.

[9] A. Bernstein. *A Nearly Optimal Algorithm for Approximating Replacement Paths and k Shortest Simple Paths in General Graphs*, pages 742–755. doi:10.1137/1.9781611973075.61.

[10] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 101–110, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1536414.1536431.

[11] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan. Parallel shortest paths using radius stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 443–454, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2935764.2935765.

[12] B. Y. Chen, X.-W. Chen, H.-P. Chen, and W. H. Lam. Efficient algorithm for finding k shortest paths based on re-optimization technique. *Transportation Research Part E: Logistics and Transportation Review*, 133:101819, 2020. doi:10.1016/j.tre.2019.11.013.

[13] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *The Annals of Mathematical Statistics*, 23(4):493 – 507, 1952. doi:10.1214/aoms/1177729330.

[14] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser. Alternative routing: k-shortest paths with limited overlap. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2820783.2820858.

[15] J. S. Chuang and D. Roth. Gene recognition based on dag shortest paths. *Bioinformatics*, 17:S56–S64, 06 2001. doi:10.1093/bioinformatics/17.suppl_1.S56.

[16] F. Chung and L. Lu. The diameter of sparse random graphs. *Advances in Applied Mathematics*, 26(4):257 – 279, 2001. doi:10.1006/aama.2001.0720.

[17] S. Clarke, A. Krikorian, and J. Rausen. Computing the n best loopless paths in a network. *Journal of the Society for Industrial and Applied Mathematics*, 11(4):1096–1102, 1963. doi:10.1137/0111081.

[18] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM Journal on Computing*, 37(5):1299–1318, 2008. doi:10.1137/S0097539705429847.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959. doi:10.1007/BF01386390.

[20] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. *Single-source shortest paths with the parallel boost graph library*, pages 219–248. 07 2009. doi:10.1090/dimacs/074/09.

[21] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998. doi:10.1137/S0097539795290477.

[22] P. Erdős and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290 – 297, 1959.

[23] R. Fagerberg, C. Flamm, R. Kianian, D. Merkle, and P. F. Stadler. Finding the k best synthesis plans. *Journal of Cheminformatics*, 10(1):19, Apr 2018. doi:10.1186/s13321-018-0273-z.

[24] G. Feng. Finding k shortest simple paths in directed graphs: A node classification algorithm. *Networks*, 64(1):6–17, 2014. doi:10.1002/net.21552.

[25] G. Feng. Improving space efficiency with path length prediction for finding k shortest simple paths. *IEEE Transactions on Computers*, 63(10):2459–2472, 2014. doi:10.1109/TC.2013.136.

[26] K. Filippova. Multi-sentence compression: Finding shortest paths in word graphs. In *Proceedings of the 23rd International Conference on Computational Linguistics*, COLING '10, pages 322–330, USA, 2010. Association for Computational Linguistics.

[27] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[28] A. Frieder and L. Roditty. An experimental study on approximating k shortest simple paths. *ACM J. Exp. Algorithmics*, 19, apr 2015. doi:10.1145/2630068.

[29] A. Frieze and G. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10(1):57–77, 1985. doi:10.1016/0166-218X(85)90059-9.

[30] E. N. Gilbert. Random Graphs. *The Annals of Mathematical Statistics*, 30(4):1141 – 1144, 1959. doi:10.1214/aoms/1177706098.

[31] Z. Gotthilf and M. Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Information Processing Letters*, 109(7):352–355, 2009. doi:10.1016/j.ipl.2008.12.015.

[32] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 01 2005.

[33] E. Hadjiconstantinou and N. Christofides. An efficient implementation of an algorithm for finding k shortest simple paths. *Networks*, 34(2):88–101, 1999. doi:10.1002/(SICI)1097-0037(199909)34:2<88::AID-NET2>3.0.CO;2-1.

[34] Y. He, Z. Liu, J. Shi, Y. Wang, J. Zhang, and J. Liu. k-shortest-path-based evacuation routing with police resource allocation in city transportation networks. *PLOS ONE*, 10(7):1–23, 07 2015. doi:10.1371/journal.pone.0131962.

[35] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Trans. Algorithms*, 3(4):45–es, nov 2007. doi:10.1145/1290672.1290682.

[36] S. Hoceini, A. Mellouk, and Y. Amirat. K-shortest paths q-routing: A new qos routing algorithm in telecommunication networks. In P. Lorenz and P. Dini, editors, *Networking - ICN 2005*, pages 164–172, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/978-3-540-31957-3_21.

[37] M. Ibrahim, C. Salama, M. W. El-Kharashi, and A. Wahba. *Pin-Count and Wire Length Optimization for Electrowetting-on-Dielectric Chips: A Metaheuristics-Based Routing Algorithm*, pages 271–294. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-20071-2_10.

[38] D. J. Ives, P. Bayvel, and S. J. Savory. Routing, modulation, spectrum and launch power assignment to maximize the traffic throughput of a nonlinear optical mesh network. *Photonic Network Communications*, 29(3):244–256, Jun 2015. doi:10.1007/s11107-015-0488-0.

[39] R. M. Karp. The transitive closure of a random digraph. *Random Structures & Algorithms*, 1(1):73–93, 1990. doi:10.1002/rsa.3240010106.

[40] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982. doi:10.1002/net.3230120406.

[41] D. Kurz and P. Mutzel. A Sidetrack-Based Algorithm for Finding the k Shortest Simple Paths in a Directed Graph. In S.-H. Hong, editor, *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49:1–49:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ISAAC.2016.49.

[42] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972. doi:10.1287/mnsc.18.7.401.

[43] A. Lebedev, J. Lee, V. Rivera, and M. Mazzara. Link prediction using top-k shortest distances. In A. Calì, P. Wood, N. Martin, and A. Poulovassilis, editors, *Data Analytics*, pages 101–105, Cham, 2017. Springer International Publishing.

[44] Q. Liang, W. Wu, Y. Yang, R. Zhang, Y. Peng, and M. Xu. Multi-player tracking for multi-view sports videos with improved k-shortest path algorithm. *Applied Sciences*, 10(3), 2020. doi:10.3390/app10030864.

[45] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. *An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances*, pages 23–35. 2007. doi:10.1137/1.9781611972870.3.

[46] E. Martins and M. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1:121–133, 06 2003. doi:10.1007/s10288-002-0010-2.

[47] E. Martins, M. Pascoal, and J. Santos. Deviation algorithms for ranking shortest paths. *Int. J. Found. Comput. Sci.*, 10:247–262, 09 1999. doi:10.1142/S0129054199000186.

[48] M. J. McDermott, S. S. Dwaraknath, and K. A. Persson. A graph-based network for predicting chemical reaction pathways in solid-state materials synthesis. *Nature Communications*, 12(1):3097, May 2021. doi:10.1038/s41467-021-23339-x.

[49] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. 1998 European Symposium on Algorithms. doi:10.1016/S0196-6774(03)00076-2.

[50] U. Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 797–806, USA, 2001. Society for Industrial and Applied Mathematics.

[51] U. Meyer. *Design and analysis of sequential and parallel single-source shortest-paths algorithms*. PhD thesis, Saarland University, Saarbrücken, Germany, 2002. URL: http://scidok.sulb.uni-saarland.de/volltexte/2004/207/index.html.

[52] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[53] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522739.

[54] L. R. Nielsen, K. A. Andersen, and D. Pretolani. Finding the k shortest hyperpaths. *Computers & Operations Research*, 32(6):1477–1497, 2005. doi:10.1016/j.cor.2003.11.014.

[55] L. Oettershagen and P. Mutzel. Computing top-k temporal closeness in temporal networks. *Knowledge and Information Systems*, 64(2):507–535, Feb 2022. doi:10.1007/s10115-021-01639-4.

[56] T. Panitanarak and K. Madduri. Performance analysis of single-source shortest path algorithms on distributed-memory systems. In *SIAM Workshop on Combinatorial Scientific Computing (CSC)*, page 60, 2014.

[57] A. Perko. Implementation of algorithms for k shortest loopless paths. *Networks*, 16(2):149–160, 1986. doi:10.1002/net.3230160204.

[58] V. Priebe. *Average-case complexity of shortest-paths problems.* PhD thesis, Saarland University, Saarbrücken, Germany, 2001. doi:10.22028/D291-25863.

[59] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Automata, Languages and Programming*, pages 249–260, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[60] E. Ruppert. Finding the k shortest paths in parallel. *Algorithmica*, 28(2):242–254, Oct 2000. doi:10.1007/s004530010038.

[61] A. Samara. *Stochastic routing optimized for autonomous driving*. PhD thesis, Mannheim, 2021. URL: https://madoc.bib.uni-mannheim.de/60206/.

[62] A. Schickedanz, D. Ajwani, U. Meyer, and P. Gawrychowski. Average-case behavior of k-shortest path algorithms. In L. M. Aiello, C. Cherifi, H. Cherifi, R. Lambiotte, P. Lió, and L. M. Rocha, editors, *Complex Networks and Their Applications VII - Volume 1 Proceedings The 7th International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2018, Cambridge, UK, December 11-13, 2018*, volume 812 of *Studies in Computational Intelligence*, pages 28–40. Springer, 2018. doi:10.1007/978-3-030-05411-3_3.

[63] A. Sedeño-Noda. An efficient time and space k point-to-point shortest simple paths algorithm. *Applied Mathematics and Computation*, 218(20):10244–10257, 2012. doi:10.1016/j.amc.2012.04.002.

[64] S. Shi and C. Qian. Concurrent entanglement routing for quantum networks: Model and designs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 62–75, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405853.

[65] Y.-K. Shih and S. Parthasarathy. A single source k-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics*, 28(12):i49–i58, 06 2012. doi:10.1093/bioinformatics/bts212.

[66] A. P. Singh and D. P. Singh. Implementation of k-shortest path algorithm in gpu using cuda. *Procedia Computer Science*, 48:5–13, 2015. International Conference on Computer, Communication and Convergence (ICCC 2015). doi:10.1016/j.procs.2015.04.103.

[67] D. Spielman and S.-H. Teng. Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 296–305, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/380752.380813.

[68] N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, 13(9):1582–1597, May 2020.

[69] Y. Urayama and T. Tachibana. Virtual network construction with k-shortest path algorithm and optimization problems for robust physical networks. *International Journal of Communication Systems*, 30(1):e2958, 2015. e2958 dac.2958. doi:10.1002/dac.2958.

[70] S. Vanhove and V. Fack. An effective heuristic for computing many shortest path alternatives in road networks. *International Journal of Geographical Information Science*, 26(6):1031–1050, 2012. doi:10.1080/13658816.2011.620572.

[71] B. Viale, M. Fer, L. Courau, P. Galy, B. Jacquier, J. Lescot, and B. Allard. An automated tool for chip-scale esd network exploration and verification. In *2016 38th Electrical Overstress/Electrostatic Discharge Symposium (EOS/ESD)*, pages 1–10, 2016. doi:10.1109/EOSESD.2016.7592551.

[72] Y. Wang, Q. Liu, H. Ren, X. Ma, L. Liu, W. Wang, and J. Zhang. Optimizing multi-criteria k-shortest paths in graph by a natural routing genotype-based genetic algorithm. In *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 341–345, 2018. doi:10.1109/ICIEA.2018.8397739.

[73] M. L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. doi:10.21105/joss.03021.

[74] Q. Wen, R. Chen, L. Nai, L. Zhou, and Y. Xia. Finding top k shortest simple paths with improved space efficiency. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems*, GRADES'17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3078447.3078460.

[75] V. V. Williams and R. R. Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5), aug 2018. doi:10.1145/3186893.

[76] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971. doi:10.1287/mnsc.17.11.712.

[77] Z. Yu, X. Yu, N. Koudas, Y. Liu, Y. Li, Y. Chen, and D. Yang. Distributed processing of k shortest path queries over dynamic road networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 665–679, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389735.

[78] X. Yuan, S. Mahapatra, W. Nienaber, S. Pakin, and M. Lang. A new routing scheme for jellyfish and its performance with hpc workloads. SC '13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2503210.2503229.