Hochschule Kaiserslautern
University of Applied Sciences
Informatik und Mikrosystemtechnik
Zweibrücken

Bericht Studienprojekt SS 24

Bachelor-Studiengang (PO Version 2018)

IT Analyst

Titel:

Kürzeste Wege auf allgemeinen Graphen

Shortest paths on general graphs

vorgelegt von: Alexander Wiltz

vorgelegt am: 09. August 2024

vorgelegt bei: Prof. Dr. Jörg Hettel

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Alexander Wiltz, geboren am 06.03.1987 in Saarlouis, ehrenwörtlich, dass ich meine Studienarbeit mit dem Titel: „Kürzeste Wege auf allgemeinen Graphen“ selbstständig und ohne fremde Hilfe angefertigt und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe. Die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen habe ich innerhalb der Arbeit gekennzeichnet. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Überherrn, 6. August 2024

. . . . . .

(Alexander Wiltz)

Zusammenfassung

Diese Arbeit setzt sich mit kürzesten Wegen auf allgemeinen Graphen auseinander.

Der Fokus liegt auf den allgemein bekannten Wegfinder-Algorithmen in den gängigen Anwendungsgebieten. Dabei werden die wichtigsten Vor- und Nachteile erörtert und anhand von Beispielen beschrieben und weitestgehend erklärt.
Jeder behandelte Algorithmus hat eine informelle Beschreibung und wird mit Pseudocode beschrieben. Weiterhin wird auf die Zeitkomplexität mit ihren jeweiligen Bedingungen eingegangen.

Zur Vervollständigung der Ausarbeitung sind Implementierungen in JAVA-Code mit ausführbaren Beispielen mit Verweisen auf die betreffenden Algorithmen vorhanden.

Abstract

This thesis deals with shortest paths on general graphs.

The focus is on the generally known path finding algorithms in common application areas. The most important advantages and disadvantages are discussed and described and explained as far as possible using examples.

Each algorithm covered has an informal description and is described using pseudo code. Furthermore, the time complexity with its respective conditions is discussed.

To complete the present elaboration, there are implementations in JAVA code with executable examples with references to the algorithms concerned.

Inhaltsverzeichnis

1 Einleitung

Die Navigation ist die Kunst zu Wasser, zu Land und in der Luft, das Fahr- bzw. Flugzeug sicher zum gewünschten Zielpunkt zu steuern. Dieser Aufgabe gehen zwei geometrische Aufgaben voraus: das Feststellen der momentanen Position und die Ermittlung der Route zum gewünschten Ziel.

Wo es sich früher um das Folgen einer Route mit Karte und Kompass gehandelt hat, sind es heutzutage GPS-Geräte die zur Navigation eingesetzt werden. Aber allgemein kann die Navigation deutlich abstrakter betrachtet werden, in dem die Bewegung von Teilchen, Elementen oder Daten untersucht bzw. bestimmt werden. Das Thema ist sehr vielschichtig und erfordert je nach Anforderung komplexe Berechnungen, die teils von Computersystemen gelöst werden müssen.

Beispielsweise ist das Problem des Handlungsreisenden (auch Botenproblem) ein kombinatorisches Optimierungsproblem des Operations Research und der theoretischen Informatik. Die Aufgabe besteht darin, eine Auswahl mehrerer Orte so zu wählen, dass kein Ort doppelt oder gar mehrfach besucht wird, die gesamte Reisestrecke möglichst kurz und der Startort gleich der des Zielortes ist. (Weiteres unter: `https://algorithms.discrete.ma.tum.de/graph-games/tsp-game/index_de.html`)

Oft gibt es viele mögliche Wege zu einem definierten Ziel zu gelangen. Aber ob es der kürzeste oder der optimalste Weg ist, lässt sich auf den ersten Blick nicht immer feststellen.

Mit genau diesen Problemstellungen befasst sich die Graphentheorie. Dort werden kürzeste Pfade in einem gegebenen Graph bzw. einem Netzwerk gesucht und berechnet. In der Informatik ist die Wegfindung die algorithmengestützte Suche nach dem oder den optimalen Wegen von einem gegebenen Startpunkt zu einem oder mehreren Zielpunkten.

1.1 Ziel der Arbeit

Ziel ist es, einen grundlegenden Überblick über Kürzeste Wege Algorithmen und die mathematischen Beschreibungen aufzuzeigen und inwiefern die gängigen Algorithmen zum Lösen der mathematischen Herausforderungen mittels Software umgesetzt werden können.

1.2 Motivation

Die Motivation zu dieser Arbeit resultiert aus dem Interesse an dem Thema ”Netzwerke und Graphen” aus der Mathematik. Im Modul ”Diskrete Mathematik” für Informatiker, wurde das Thema Graphen und Fragestellungen dazu behandelt. Das Prinzip des ”Haus vom Nikolaus”[2] und dem ”Königsberger Brückenproblem”[3] von Leonhard Euler (1707-1783, Mathematiker) haben die Neugierde geweckt, sich grundlegend mit der Frage zu beschäftigen: Was sind Graphen und wie funktionieren sie?

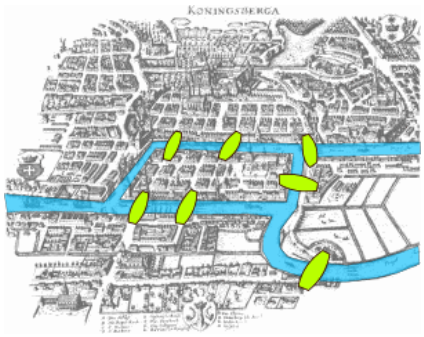

Abbildung 1.1: Königsberger Brücken [1]

Reduziert man Abbildung 1.1 auf eine schematische Darstellung, so ergibt sich für einen Graphen ein einfaches Modell, wobei die Linien die Brücken symbolisieren und die Punkte die Orte, die damit verbunden werden:

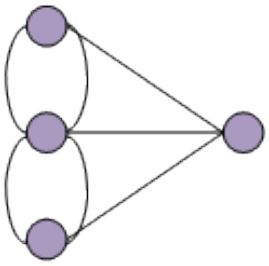

Abbildung 1.2: Königsberger Brücken als Graph

Die Frage des Brückenproblems ist, ob es möglich sei, einen Spaziergang so zu organisieren, dass man jede Brücke genau einmal überquert. Anders ausgedrückt: Besitzt der Graph einen Rundweg, in dem jede Brücke genau einmal vorkommt?

Bei näherer Betrachtung wird schnell klar, das sich Graphen auf viele alltägliche Probleme projizieren lassen und es aufgrund der Vielfalt von Interesse ist, deren Ansprüche und Aufgaben automatisiert, also mittels Software, darzustellen.

[1] https://mathepedia.de/Koenigsberger_Brueckenproblem.html, 18.05.2024
[2] https://de.wikipedia.org/wiki/Haus_vom_Nikolaus
[3] https://mathepedia.de/Koenigsberger_Brueckenproblem.html

1.3 Aufbau der Arbeit

Im nächsten Kapitel werden die Mathematischen Grundlagen zum Thema aufgegriffen und zum Verständnis der nachfolgenden Algorithmen beschrieben. Als weitere Grundlage werden die abstrakten Klassen der Knoten und der Graphen in JAVA beschrieben.

In den danach folgenden Kapiteln werden die behandelten Algorithmen von Dijkstra, Bellman-Ford und Floyd-Warshall, sowie dem A-Stern-Algorithmus beschrieben und anhand von Beispielen, alle notwendigen Schritte des jeweiligen Ablaufs zur Ermittlung eines kürzesten Pfades beschrieben. Neben einer informellen Beschreibung und dem Pseudocode, gibt es einen Abschnitt einer Implementierung in JAVA und der Verweis auf das entsprechende Repository in Github. Außerdem werden die Vor- und Nachteile umrissen und die Zeitkomplexität der Vollständigkeit halber angegeben.

Die JAVA-Code-Beispiele sind ebenfalls unter `https://github.com/alexander-wiltz/shortest-path-algorithms` abgelegt.

2 Grundlagen

Im folgenden Kapitel werden die notwendigen mathematischen Grundlagen zum Verständnis der weiteren Themen umrissen. Auf weitere Informationen zu den Begriffen wird in den jeweiligen Abschnitten verwiesen oder durch Angabe aller Informationsquellen im anhängenden Quellenverzeichnis.

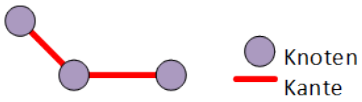

Abbildung 2.1: Darstellung eines Weges mit Knoten und Kanten

2.1 Kürzeste Wege finden

Kürzeste Wege finden, oder Pathfinding, ist in der Informatik die Suche nach dem optimalen Weg von einem Startpunkt zu einem oder mehreren Zielpunkten mit Hilfe von Algorithmen. Die Einsatzgebiete sind Vielfältig und reichen von Computerspielen, über Netzwerk-Flussanalysen und Routenplanung bis zu hin zu Infrastrukturlösungen.

In vielen, aber nicht unbedingt in allen Fällen, ist die Suche nach einem optimalen Weg zwischen zwei gegebenen Punkten auch die Suche nach der kostengünstigsten oder kürzesten Route. In der Praxis ist ein Pathfinding-Problem in den wenigsten Fällen die Suche nach der Luftlinie zwischen einem Start- und einem Zielpunkt. Häufig wird die Suche durch andere Faktoren bestimmt, wodurch andere Routen sinnvoller erscheinen.
In den meisten Fällen sind das nicht oder nur bedingt passierbare Hindernisse oder variable Kosten einer Route (Bodenbeschaffenheit, Geschwindigkeitsbegrenzungen, etc.)

Diese Faktoren werden Kosten genannt, welche in der Mathematik als Kantengewicht bezeichnet werden. Die Größe oder der Wert und der Nutzen wird durch die angegebene physikalische Größe bestimmt. Das können Längen, Zeiten oder elektrische Stromstärken sein.

Beispiel:
Eine Fahrtstrecke zwischen zwei Orten kann aufgrund des ausgebauten Straßennetzes über vielfache Weise ermittelt werden. Dabei kann dies über eine Autobahn (schnell) oder über eine Landstraße (kürzeste) erreicht werden. Was ist optimal?

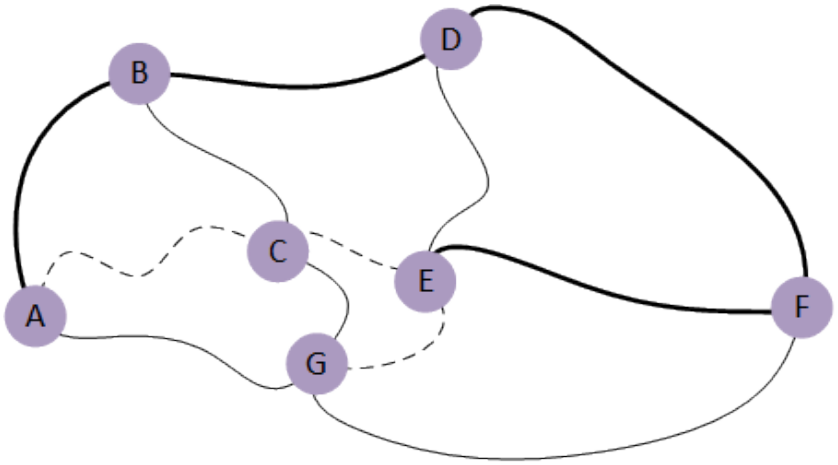

Abbildung 2.2: Fiktive Straßenkarte

Die mit Buchstaben in Abbildung 2.2 gekennzeichneten Kreise stellen Ortschaften dar.
Die dicken Linien stehen für Schnellstraßen, die dünneren für Landstraßen und die gestrichelten für Feldwege.
Allgemein werden weitere Parameter, wie z.B. Entfernungen, Zeiten oder Kosten benötigt, um konkrete Aussagen treffen zu können, welcher Weg oder über welche Abschnitte die optimale oder kürzeste Route von einem Startort zu einem Zielort führt. Der allgemeingültige Schluss einer möglichen Wegbestimmung ist, dass auf einer Landstraße weniger schnell gefahren werden kann und bei einem Weg über weniger schnell befahrbaren Wegen unter Umständen die gesamte Fahrzeit hoch sein könnte.

Diese Probleme werden in der Informatik als Shortest-Path-Probleme bezeichnet.

2.2 Greedy-Algorithmen

Greedy-Algorithmen, oder auch: gierige Algorithmen, bilden eine spezielle Klasse von Algorithmen in der Informatik. Sie finden oft schnell eine Lösung, während andere Algorithmen kein Ergebnis in endlicher Zeit liefern können. [1]

Oft sind die gefundenen Lösung jedoch nicht die optimale Lösung für ein Problem. Greedy-Algorithmen treffen immer nur die aktuell beste Entscheidung, ohne auf Vorherige oder auf Auswirkungen der Wahl zu achten.
Die Lösung wird einfach nach und nach zusammengesetzt. Dabei wird in jedem Schritt der Folgezustand ausgewählt, der zum Zeitpunkt der Betrachtung das beste Ergebnis verspricht. Ein mögliches Optimum wird dabei ignoriert, denn der Algorithmus ist in dem Sinne gierig, dass er einfach das nächstbeste Ergebnis bestimmt. Dieser Ansatz funktioniert auf einigen Problemen, die eigentlich deutlich schwieriger aussehen. Ein wichtiger Vorteil von Greedy Algorithmen ist, dass sie sehr schnell und einfach zu implementieren sind.

[1] https://de.wikipedia.org/wiki/Greedy-Algorithmus

Eine Anwendung eines Greedy-Algorithmus im täglichen Leben ist beispielsweise die Herausgabe von Wechselgeld.
Das Verfahren lautet:

Nimm jeweils immer die größte Münze unter dem Zielwert und ziehe sie von diesem ab. Mache genau so weiter bis der Zielwert gleich null ist.

Als Beispiel: Gebe 68 Cent mit den in der Realität (1, 2, 5, 10, 20, 50 Cent) vorhandenen Münzwerten zurück. $\longrightarrow$ 68 = 50 + 10 + 5 + 2 + 1
Der Algorithmus beginnt mit dem höchsten Münzwert und macht weiter, bis er den niedrigsten Münzwert erreicht hat. In diesem konkreten Fall ist das Ergebnis optimal, da von jeder möglichen Münze genau eine genommen werden kann und keine Münzen von niedrigem Wert vielfach und eine Größere ausgelassen.

Ein weiteres Beispiel:
Wird ein Münzvorrat für eine Rückgabe von 15 Cent vorgegeben und es stehen ausschließlich 11, 10, 5 und 1 Cent Münzen zur Verfügung, dann wäre das Optimum: 10 + 5. Durch die Vorgabe der zu verwendenden Münzen wird das Ergebnis zu: 15 = 11 + 1 + 1 + 1 + 1.

2.3 Grundlegende Begriffe

2.3.1 Graph

Ein Graph ist grundsätzlich eine Menge von Knoten und Kanten.

Definition:
Ein Graph $G = (V(G), E(G))$, besteht aus zwei endlichen Mengen, wobei V eine Menge von Knoten und E eine Menge von Kanten bezeichnet. Dabei ist:

- $V(G)$, die Knotenmenge des Graphen G, die eine nicht leere Menge von Elementen ist, die Knoten genannt werden. Ist $G = (V, E)$ ein gerichteter oder ein ungerichteter Graph, so nennt man ein Element $x \in V$ einen Knoten von G.
- $E(G)$, die Kantenmenge des Graphen G, die eine möglicherweise leere Menge von Elementen ist, die Kanten genannt werden. Ist $G = (V, E)$ ein ungerichteter Graph, so nennt man ein Element $[x, y] \in E$ (mit $x, y \in V$ die Kante von G; x, y heißen Endknoten.

Jede Menge $e \in E$ wird einem ungeordneten Paar von Knoten (u, v) mit $u, v \in V$ zugeordnet, die als Endknoten von e bezeichnet werden.
Wenn e eine Kante mit Endknoten u und v ist, sagt man, dass die Knoten u und v benachbart bzw. adjazent sind. [2]

[2]Handout "Diskrete Mathematik für Informatiker", Prof. Dr. Hettel, Hochschule Kaiserslautern, 2020

Ein Graph ist ein mathematisches Modell für eine netzartige Struktur. Das können Strukturen in Natur und Technik sein. Beispiele für Graphen sind:

- Soziale Netze
- Straßen- oder (U-)Bahn-Netze
- Energienetze
- Verwandschaftsbeziehungen, Stammbäume
- Computernetze
- elektrische Schaltungen
- chemische Verbindungen

In der Mathematik untersucht man lediglich die abstrakte Struktur. Die Art, die Lage und die Beschaffenheit bleibt dabei unberücksichtigt.

Viele algorithmische Probleme können auf Graphen zurückgeführt werden, wie auch viele graphentheoretische Probleme auf Algorithmen basieren. Dadurch ist die Graphentheorie auch in der Informatik bedeutungsvoll. Die Untersuchung von Graphen ist unter anderem ein Teil der Netzwerktheorie.

Gerichtete Graphen
In gerichteten Graphen (Digraphen) werden Kanten durch Pfeile anstelle von Linien zur Verdeutlichung ihrer zu durchlaufenden Richtung dargestellt. Der Pfeil zeigt dabei von seinem Anfangsknoten zu seinem Endknoten.
Ein Spezialfall sind orientierte Graphen: es gibt keine ungerichteten Kanten. Wenn eine Kante von Knoten A nach Knoten B existiert, dann kann nie die umgekehrte Kante von B nach A existieren.

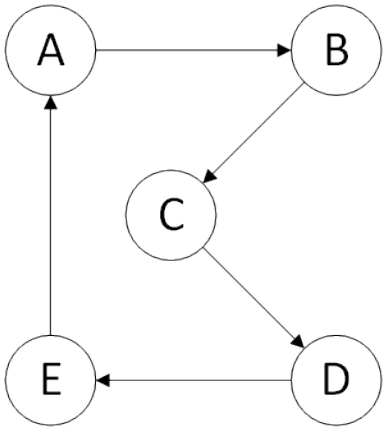

Abbildung 2.3: Gerichteter Graph

Ungerichtete Graphen
In ungerichteten Graphen werden die Verbindungen zwischen Knoten durch einfache Linien, also Kanten gekennzeichnet. Die Kanten haben keine Richtung. Jede Kante kann in beide Richtungen durchlaufen werden.

Eigenschaften von Graphen
Graphen können folgende Eigenschaften haben:

- zusammenhängend
  Ein Graph heißt zusammenhängend, wenn seine Knoten paarweise durch eine Kantenfolge verbunden sind.
- bipartit
  Ein einfacher Graph heißt bipartit oder paar, falls sich seine Knoten in zwei disjunkte

Teilmengen A und B aufteilen lassen($alsoA \cup B = VundA \cap B = \emptyset$), sodass jede Kante ein Ende in A und ein Ende in B hat und keine Kante innerhalb einer Teilmenge existiert.

- planar
  Ein Graph ist planar, wenn er auf einer Ebene, mit Punkten für die Knoten und Linien für die Kanten, dargestellt werden kann, sodass sich keine Kanten schneiden.
- eulersch
  Ein Graph, der in einem Zyklus, alle Kanten eines Graphen genau einmal enthält.
- hamiltonisch
  Ein Graph, bei dem in einem Zyklus, unabhängig seiner Reihenfolge, jeder Knoten genau einmal durchlaufen wird.

2.3.2 Knoten

Knoten, oder auch Ecken, sind der Teil eines Graphen, der mindestens Teil einer Kante ist. Ausnahmen bilden isolierte Knoten. Ein isolierter Knoten ist in einem ungerichteten Graphen ein Knoten ohne Nachbarn. Dieser Knoten ist vom Grad null. In einem gerichteten Graphen besitzt ein isolierter Knoten keine Vorgänger und Nachfolger und hat somit Eingangs- und Ausgangsgrad null.

Die Abkürzung V stammt aus dem Englischen: *vertex* für Knoten.

Grad Der Grad $d(v)$ eines Knoten v entspricht der Anzahl der mit v inzidenten Kanten, wobei jede Schlinge zwei Mal gezählt wird. Das bedeutet, der Grad von v gibt an, wie oft v ein Endknoten ist.

2.3.3 Kanten

Allgemein sind Kanten Teil eines Graphen und stellen die Verbindung zwischen genau zwei Knoten her.

Die Abkürzung E stammt aus dem Englischen: *edge* für Kante, oder Ecke.

Das **Kantengewicht** ist eine Angabe über den Wert einer Kante. Die Form wird durch die jeweilige Einheit definiert. Das Gewicht kann zum Beispiel eine Entfernung sein, eine Zeit oder ein elektrischer Strom. Unter Berücksichtigung des Kantengewichts können die Kosten zwischen Knoten ermittelt werden und somit die optimalen Wege nach definierten Kriterien zwischen Knoten.

2.3.4 Adjazenz

Der Begriff Adjazenz stammt aus dem Lateinischen und bedeutet Nachbarschaft.
Zwei Knoten, die durch eine Kante verbunden sind, oder zwei Kanten, die einen gemeinsamen Knoten besitzen, nennt man *benachbart* oder *adjazent*.

2.3.5 Inzidenz

Gehört ein Knoten zu einer Kante, so werden beide *inzident* genannt. Ein Knoten ist also inzident mit einer Kante, wenn der Knoten an wenigstens einem Ende der Kante liegt. Eine Kante ist inzident mit einem Knoten, wenn die Kante den Knoten an einem ihrer Enden hat.

2.3.6 Pfad

In der Graphentheorie werden mindestens zwei aufeinanderfolgende Knoten die mit einer Kante verbunden sind als Pfad (oder Weg) bezeichnet. Zwei aufeinanderfolgende Kanten, die mindestens einen gemeinsamen Knoten besitzen, werden Kantenzug genannt.

> **Definition:**
> Ein nichtleerer Graph W mit der Knotenmenge $\{x_1, x_2, \ldots, x_\mathrm{n}\}$ und der Kantenmenge $\{\{x_1, x_2\}, \{x_2, x_3\}, \ldots, \{x_\mathrm{n} - 1, x_\mathrm{n}\}\}$ mit $2 \leq n$ heißt Weg, wenn die Knoten x_i mit $1 \leq i \leq n$ paarweise verschieden sind. Auch ein Graph mit einer Knotenmenge $\{x_1\}$ (d. h. mit einem Knoten) und einer leeren Kantenmenge wird meistens als Weg (der Länge 0) bezeichnet. [18]

2.4 Bäume

Bäume sind wichtige Typen und grundlegende Bausteine von Graphen. Sie sind besonders geeignet zur Darstellung von Strukturen bzw. Abläufen, zum Beispiel für Suchen oder Sortieren.

> **Definition:**
> Ein Baum ist ein Graph, der zusammenhängend ist und keine Kreise enthält. Ein Graph, der nicht zusammenhängend ist, dessen Komponenten aber Bäume sind, heißt Wald. [17]

Eigenschaften von Bäumen
Ein zusammenhängender Graph G mit n Knoten ist genau dann ein Baum, wenn er eine (und damit alle) der folgenden Eigenschaften hat:

- G hat genau $n - 1$ Kanten
- Wird eine Kante entfernt, so ist der Restgraph nicht mehr zusammenhängend
- Zwischen zwei beliebigen Knoten gibt es genau einen Weg.

Wenn ein Baum, der alle n Knoten von einem zusammenhängendem Graph G enthält, dann heißt der Graph **aufspannender Baum** von G.

2.5 Datenstrukturen

Es gibt im wesentlichen zwei gebräuchliche Formen, Graphen am Computer darzustellen: die Adjazenzmatrix (auch Nachbarschaftsmatrix) und die Adjazenzliste (Nachbarschaftsliste). Es

greift praktisch jede algorithmische Lösung auf eine der beiden Darstellungsformen zurück. Eine weitere, weniger gebräuchliche Form ist die Inzidenzmatrix (auch Knoten-Kanten-Matrix).

2.5.1 Adjazenzmatrix

Eine Adjazenzmatrix speichert, welche Knoten eines Graphen durch eine Kante verbunden sind. Für jeden Knoten existiert eine Zeile und eine Spalte, woraus sich für n Knoten eine $n \times n$-Matrix ergibt. Ein Wert in der i-ten Zeile und j-ten Spalte gibt an, ob eine Kante von dem i-ten zu dem j-ten Knoten hat, also ob die Knoten verbunden sind. Ist der Wert eine 0 (*false*), gibt keine Verbindung durch eine Kante. Eine 1 (*true*) hingegen gibt an, dass eine Kante existiert.

Beispiel für ungerichtete Graphen:

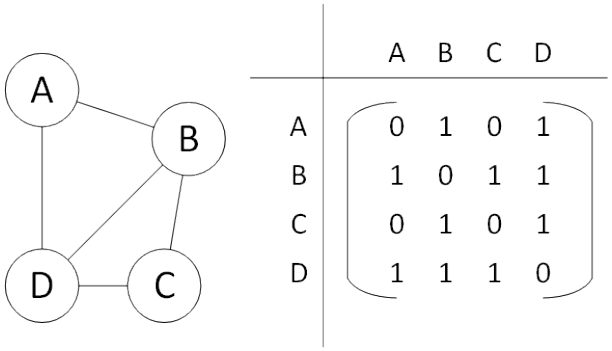

Abbildung 2.4: Beispiel einer Adjazenzmatrix mit einem ungerichteten Graph

Eine weitere Darstellungsform ist, wenn die Kanten ein Gewicht besitzen. Dann kann das Gewicht dieser Kante anstelle der einfachen Angabe, ob eine Verbindung besteht oder nicht, rücken. Anstelle von 0, für das nicht Vorhandensein einer Verbindung, kann auch davon ausgegangen werden, dass nicht bekannt ist, ob eine Verbindung besteht, daher ist auch eine Kennzeichnung durch ∞ gebräuchlich.

Beispiel für gerichtete Graphen:

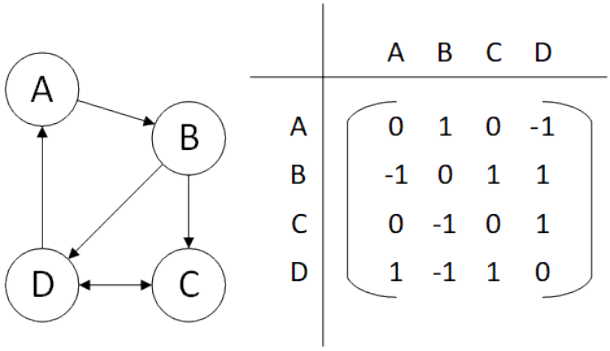

Abbildung 2.5: Beispiel einer Adjazenzmatrix mit einem gerichteten Graph

2.5.2 Adjazenzliste

In Adjazenzlisten (oder auch Nachbarschaftslisten) werden für jeden Knoten, alle ihre Nachbarn (in ungerichteten Graphen) beziehungsweise Nachfolger (in gerichteten Graphen) angegeben. Meistens wird für Algorithmen in einem Array für jeden Knoten eine einfach verkettete Liste aller Nachbarn gespeichert.

Beispiel:

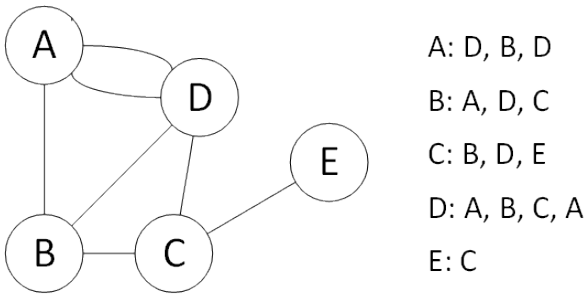

Abbildung 2.6: Beispiel einer Adjazenzliste

2.5.3 Inzidenzmatrix

Die Inzidenzmatrix eines Graphen zeigt die Beziehungen zwischen Knoten und Kanten eines Graphen auf. Wenn ein Graph n Knoten und m Kanten hat, so ist seine Inzidenzmatrix eine $m \times n$ - Matrix.
Einträge in einer Spalte, geben für eine Kante e, die zugehörigen Knoten v an. Die Einträge in einer Reihe, für einen Knoten v, geben die verbundenen Kanten e an.

Beispiel ungerichteter Graph:

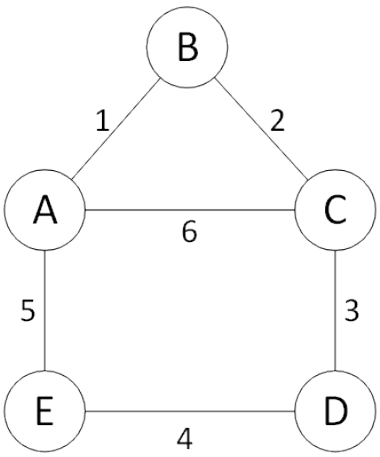

Abbildung 2.7: Beispiel für einen ungerichteten Graph

Matrix-Schreibweise:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Tabellarische Schreibweise:

	e_1	e_2	e_3	e_4	e_5	e_6
v_A	1	0	0	0	1	1
v_B	1	1	0	0	0	0
v_C	0	1	1	0	0	1
v_D	0	0	1	1	0	0
v_E	0	0	0	1	1	0

Tabelle 2.1: Beispiel einer Inzidenzmatrix für einen ungerichteten Graphen

Mit dem Wert 0 wird dargestellt, dass keine Verbindung zwischen einem Knoten und einer Kante besteht. Dementsprechend wird mit einer 1 eine Verbindung symbolisiert.

Beispiel gerichteter Graph:

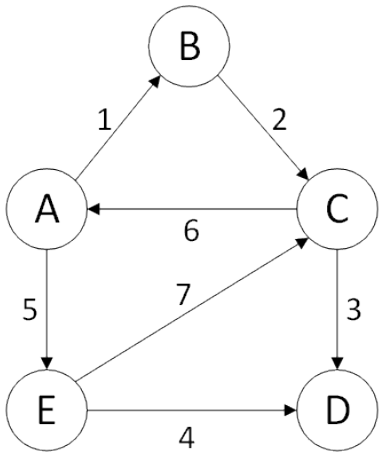

Abbildung 2.8: Beispiel für einen gerichteten Graph

Matrix-Schreibweise:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 1 \end{pmatrix}$$

Tabellarische Schreibweise:

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_A	1	0	0	0	1	-1	0
v_B	-1	1	0	0	0	0	0
v_C	0	-1	1	0	0	1	-1
v_D	0	0	-1	-1	0	0	0
v_E	0	0	0	1	-1	0	1

Tabelle 2.2: Beispiel einer Inzidenzmatrix für einen gerichteten Graphen

Mit dem Wert 0 wird dargestellt, dass keine Verbindung zwischen einem Knoten und einer Kante besteht. Das Vorzeichen bei einer bestehenden Verbindung gibt vor, ob es sich um einen Start- oder Endknoten handelt. Nach dieser Regel ist 1 als Startknoten (geht vom Knoten weg) zu bezeichnen und -1 als Endknoten (zeigt auf den Knoten).

2.6 Zeitkomplexität

Allgemein ist Komplexität eine quantitative Bewertung der Menge von Ressourcen (wie Zeit und Speicher), die benötigt werden, um eine bestimmte Aufgabe oder ein Problem zu lösen.

Zeitkomplexität hingegen ist die theoretische Berechnung der Rechenzeit, die ein Algorithmus benötigt, bezogen auf die Eingabegröße. Also die benötigte Zeit, um einen Algorithmus als Funktion der Länge einer Eingabe auszuführen. Es wird oft durch eine Funktion $T(n)$ ausgedrückt, wobei n die Eingabegröße und $T(n)$ die Zeit, die zur Ausführung des Algorithmus benötigt wird, repräsentiert.

In der theoretischen Informatik ist die Zeitkomplexität ausschlaggebend, um die Effizienz eines Algorithmus zu bestimmen. Sie dient zur Bestimmung, wie ein Algorithmus skaliert, wenn die Größe der Eingabedaten wächst.

In der Informatik wird die Landau-Notation (gebräuchlicher: O-Notation) als ein mathematischer Oberbegriff verwendet, um das Wachstum eines Algorithmus zu beschreiben. Die O-Notation beschreibt die obere Grenze für das Wachstum einer Funktion, wenn die Eingabegröße gegen unendlich wächst.

2.7 Optimierungsproblem

Ein Optimierungsproblem ist ein mathematisches Problem.
Das Ziel ist es, in einer Menge von Möglichkeiten die sinnvollste oder best möglichste eines Zielkriteriums zu bestimmen.

2.7.1 Single-source shortest path

Single-source shortest path (kurz: SSSP) befasst sich mit dem Problem, wie man einen kürzesten Weg zwischen einem Startknoten und allen anderen Knoten in einem Graphen bestimmt. Für nicht negative Gewichtsfunktionen kann der Dijkstra-Algorithmus oder der A*-Algorithmus verwendet werden. Der Bellman-Ford-Algorithmus wiederum kann für beliebige Gewichtsfunktionen die kürzesten Pfade zu allen übrigen Knoten eines Graphen berechnen.

2.7.2 Single-destination shortest path

Ziel des Single-destination shortest path (kurz: SDSP) ist die Bestimmung eines kürzesten Pfades zwischen einem Endknoten und allen anderen Knoten eines gegebenen Graphen. Das Problem kann durch die Umkehrung der Kantenrichtungen als SSSP beschrieben werden.

2.7.3 All-pairs shortest path

Die Variante All-pairs shortest path (kurz: APSP) befasst sich mit der Bestimmung der kürzesten Pfade zwischen allen Knotenpaaren eines Graphen. In Abhängigkeit der Gewichtsfunktion ist es effizienter, für jeden Knoten nacheinander das SSSP zu lösen oder spezialisierte Verfahren wie den Floyd-Warshall-Algorithmus oder den Min-Plus-Matrixmultiplikations-Algorithmus zu verwenden, die gleichzeitig für alle Paare kürzeste Pfade bestimmen.

2.8 Implementierungen in JAVA

Die in der Arbeit dargestellten Beispielimplementierungen benutzen eine abstrakte Knoten-Klasse, eine abstrakte Kanten-Klasse und eine allgemeine Graphenstruktur, die im folgenden beschrieben werden.

2.8.1 Abbildung von Knoten

Die abstrakte `GeneralNode`-Klasse, repräsentiert im allgemeinen alle Nodes, bzw. Knoten. Alle jeweiligen Nodes erben von dieser Klasse und benötigen teilweise zusätzliche Eigenschaften.

Generell hat jede Node bzw. jeder Knoten einen Namen, über den sie identifiziert werden kann.

```
1  @Setter
2  public abstract class GeneralNode<T> {
3      protected String name;
4
5      protected GeneralNode(String name) {
6          this.name = name;
7      }
8
9      @Override
10     public String toString() {
11         return this.name;
12     }
13 }
```

Listing 2.1: GeneralNode.java

2.8.2 Abbildung einer allgemeinen Kante

Es gibt für alle implementierten Algorithmen eine abstrakte und generelle `Edge`-Klasse mit der generelle für generelle `Node`-Klassen, um Kanten innerhalb eines Graphen darzustellen.

Jede Kante in der `GeneralEdge`-Klasse beinhaltet ein Ziel als `Node`-Objekt und das Kantengewicht.

```
1  @Getter
2  @AllArgsConstructor
3  public class GeneralEdge<T> {
4      protected T destination;
5      protected Integer weight;
6  }
```

Listing 2.2: GeneralEdge.java

2.8.3 Abbildung eines allgemeinen Graphen

Für alle implementierten Algorithmen existiert eine abstrakte und generelle `GeneralGraph`-Klasse mit der generellen `Node`-Klasse, die den jeweiligen Graphen repräsentieren. Jeder Graph enthält eine Map mit Nodes, bzw. Knoten und hat den mit ihnen verbundenen Kanten.

Der Methodenvorrat bietet mit `addNewNode()` das hinzufügen eines neuen Knotens zum Graphen, mit `addEgde()` die Möglichkeit eine einzelne, neue Kante hinzu zufügen, welche sinnvoll bei gerichteten Kanten ist, eine `addUndirectedEdge()`-Methode zum verbinden ungerichteter Kanten und die Ausgabe aller vorhandenen Nodes im Graphen und aller vorhandenen Edges, bzw. Kanten.

```
1  @Getter
2  public class GeneralGraph<T extends GeneralNode<T>> {
3
4      protected Map<T, List<GeneralEdge<T>>> edges = new HashMap<>();
5
6      /**
7       * Add a new node to build a graph
8       *
9       * @param node Node
10      */
11     public void addNewNode(final T node) {
12         edges.putIfAbsent(node, new ArrayList<>());
13     }
14
15     /**
16      * Use method when edge is directed to set a single direction
17      * from Node to Node
18      *
19      * @param node1  Source Node of Edge
20      * @param node2  Target Node of Edge
21      * @param weight Weight of given Edge
22      */
23     public void addEdge(final T node1, final T node2, final Integer weight) {
24         edges.putIfAbsent(node1, new ArrayList<>());
25         edges.putIfAbsent(node2, new ArrayList<>());
26         edges.get(node1).add(new GeneralEdge<T>(node2, weight));
27     }
28
29     /**
30      * Use method to simplify an edge implementation for undirected edges
31      *
32      * @param node1  First Node of Edge
33      * @param node2  Second Node of Edge
34      * @param weight Weight of Edge
```

```
35      */
36     public void addUndirectedEdge(
37             final T node1,
38             final T node2,
39             final Integer weight) {
40         addEdge(node1, node2, weight);
41         addEdge(node2, node1, weight);
42     }
43 
44     /**
45      * Get all edges which is linked to a node
46      *
47      * @param node Node on Edge
48      * @return List of Edges linked to a node
49      */
50     public List<GeneralEdge<T>> getEdgesByNode(final T node) {
51         return edges.getOrDefault(node, new ArrayList<>());
52     }
53 
54     /**
55      * Get all Edges in the whole graph
56      *
57      * @return List of all Edges
58      */
59     public List<GeneralEdge<T>> getAllEdges() {
60         List<GeneralEdge<T>> allEdges = new ArrayList<>();
61         for (Map.Entry<T, List<GeneralEdge<T>>> entry : edges.entrySet()) {
62             allEdges.addAll(entry.getValue());
63         }
64 
65         return allEdges;
66     }
67 
68     /**
69      * Get all Nodes contained in the whole graph
70      *
71      * @return Set of Nodes
72      */
73     public Set<T> getAllNodes() {
74         return edges.keySet();
75     }
76 }
```

Listing 2.3: Graph.java

3 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist nach seinem Erfinder Edsger W. Dijkstra, einem niederländischen Informatiker, benannt und löst das Problem der kürzesten Pfade für einen gegebenen Startknoten. Er berechnet den kürzesten Pfad zwischen einem Startknoten und dem Zielknoten (oder allen übrigen Knoten) in einem kantengewichteten Graphen. Vorausgesetzt der Graph ist nicht negativ. [5] [7]

3.1 Beschreibung

Die Idee des Algorithmus ist es, immer der Kante zu folgen, die die kürzeste Entfernung vom Startknoten zu sein scheint. Andere Kanten werden erst dann betrachtet, wenn alle kürzeren Entfernungen berücksichtigt wurden. Diese Art und Weise stellt sicher, dass bei Erreichen eines Knotens kein kürzerer Pfad zu ihm existiert. Eine berechnete Distanz zwischen einem Startknoten und einem besuchten Knoten wird gespeichert. Die kumulierten Distanzen zu noch nicht betrachteten Knoten können sich im Laufe des Algorithmus verändern, nämlich verringern. Dieses Schema wird so lange fortgesetzt, bis die Distanz zum Zielknoten berechnet wurde oder die Distanzen aller Knoten zum Startknoten bekannt sind. Die Knoten für die noch nicht die kürzeste Distanz gefunden.

3.2 Beispiel

Die Karte aus Abbildung 2.2 kann als Graph dargestellt werden. Orte werden zu Knoten, die Straßen zu Kanten und die benötigte fiktive Fahrzeit in Minuten wird als Kantengewicht angegeben. Die Befestigungen der Straße sind berücksichtigt.

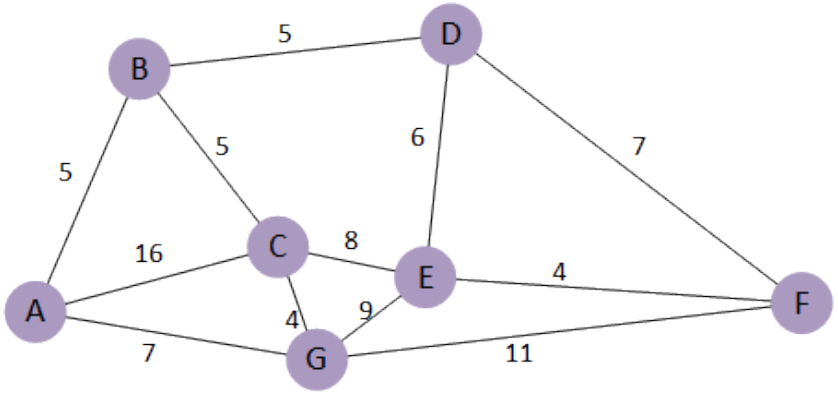

Abbildung 3.1: Fiktive Straßenkarte als gewichteter Graph

Mit Hilfe des erstellten Graph ist es möglich, diverse Wege bzw. Routen herauszulesen. Beispielhaft sind zwei mögliche Wege von A nach F eingezeichnet.

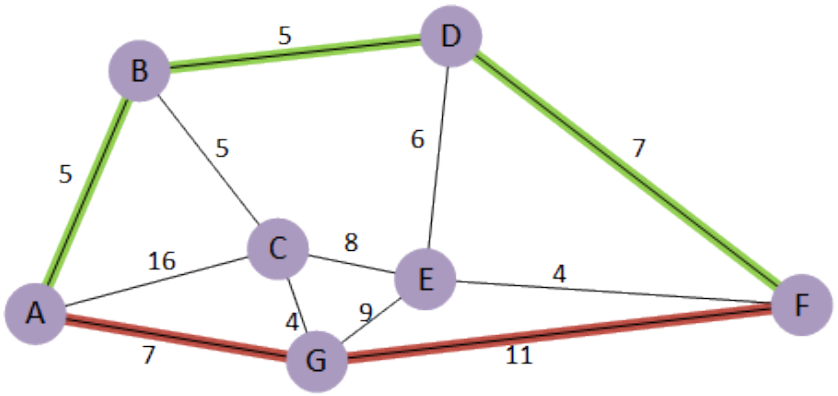

Abbildung 3.2: Fiktive Straßenkarte mit Wegen

Der Weg von A über B über D nach F ist aufgrund der Führung über eine Schnellstraße der schnellere Weg. Die Route von A über G nach F ist augenscheinlich kürzer, jedoch aufgrund der Fahrbahnbeschaffenheit langsamer. Damit ergeben sich folgende Fahrzeiten für $A \rightarrow B \rightarrow D \rightarrow F$ von 5 + 5 + 7 = 17 Minuten und für $A \rightarrow G \rightarrow F$ von 7 + 11 = 18 Minuten.

Der exemplarische Ablauf:
Im Grundsatz werden alle Knoten, vom Startknoten an, durchgehend auf ihre Entfernung geprüft und durch Probieren, der kürzeste Weg gesucht. Dabei wird jeweils die *Distanz* zum nächstgelegenen Knoten betrachtet und durch Bilden einer *Gesamtdistanz* die maximalen Kosten ermittelt. Alle Knoten werden in einer Tabelle erfasst und die Knoten, deren kürzester Weg gefunden wurde, werden aus der Tabelle entfernt.
Hinweis: Spalte ”erledigt” dient nur zum einfachen Lesen.

Der Startknoten steht an erster Stelle in der Tabelle und hat initial eine Distanz von 0. Andere Entfernung sind nicht bekannt und werden mit einer unendlichen Entfernung bewertet. Die Sortierung ist willkürlich, in diesem Beispiel folgerichtig dem Alphabet nach sortiert.

Knoten	**Vorgänger**	**Gesamtdistanz**	*erledigt*
A	-	0	
B	-	∞	
C	-	∞	
D	-	∞	
E	-	∞	
F	-	∞	
G	-	∞	

Tabelle 3.1: Dijkstra: Exemplarisch - Vorbereitung

Warteschlange: A; Erledigt: -; Ausgewählt: A; Nachfolger: B, C, G

Durch Betrachtung aller Nachbarn von Knoten A, werden die Knoten mit ihren jeweiligen Entfernungen **B(5)**, **C(16)** und **G(7)** entdeckt. Einträge in der Tabelle werden folgend erstellt und nach Entfernung sortiert:

Knoten	**Vorgänger**	**Gesamtdistanz**	*erledigt*
A	A	0	x
B	A	5	
G	A	7	
C	A	16	
D	-	∞	
E	-	∞	
F	-	∞	

Tabelle 3.2: Dijkstra: Exemplarisch - Schritt 1

Warteschlange: B, C, G; Erledigt: A; Ausgewählt: B; Nachfolger: C, D

Im nächsten Schritt wird Knoten B betrachtet: Nachbarn mit ihren Kosten sind C(5) und D(5). Die Distanz für Knoten C wird auf die mitgebrachten Kosten von B(5) + Kosten nach C(5) gesetzt, also 10. Ein Weg von A nach C über B ist ”günstiger” als von A direkt zu C. Darum wird der Wert aktualisiert. Der gleiche Ablauf für Knoten D mit: mitgebrachte Kosten B(5) + Kosten nach D(5), also 10. Für Knoten A wurde der kürzeste Weg gefunden (Startknoten) und fällt dadurch weg. Die Tabelle ändert sich zu:

Knoten	**Vorgänger**	**Gesamtdistanz**	*erledigt*
A	A	0	x
B	A	5	x
G	A	7	
C	B	10	
D	B	10	
E	-	∞	
F	-	∞	

Tabelle 3.3: Dijkstra: Exemplarisch - Schritt 2

Warteschlange: C, G, D; Erledigt: A, B; Ausgewählt: G; Nachfolger: C, E, F

Eine Betrachtung von Knoten G ergibt folgendes: Nachbarn mit den zugehörigen Kosten sind C(4), E(9) und F(11). Eine Entfernung zu C ergibt G(7) + C(4) = 11, für E ergibt G(7) + E(9) = 16 und für F G(7) + F(11) = 18. Die Distanz für C ist größer als die bereits ermittelte Distanz über B und wird deshalb nicht verändert. Nachbar A kann aufgrund seines Status als Startknoten außer acht gelassen werden.

Knoten	**Vorgänger**	**Gesamtdistanz**	*erledigt*
A	A	0	x
B	A	5	x
G	A	7	x
C	B	10	
D	B	10	
E	G	16	
F	G	18	

Tabelle 3.4: Dijkstra: Exemplarisch - Schritt 3

Warteschlange: C, D, E, F; Erledigt: A, B, G; Ausgewählt: C; Nachfolger: E, D

Knoten C wird betrachtet: Der von B kommende Weg ist der günstigste ermittelte, daher werden nur die Nachbarn G(4) und E(8) betrachtet. G hat Kosten in Höhe von 14 durch C(10) + G(4) und für E ergibt sich durch C(10) + E(8) ein Betrag von 18. E über G ist günstiger, bleibt unverändert. G über A ist ebenfalls günstiger. Die Tabelle ändert sich nicht.

Betrachtung des Knoten D: zu Nachbar E über D(10) + E(6) ergibt 16. Keine Veränderung zu der Tabelle 3.2. Nachbar F, das zu erreichende Ziel, über D(10) + F(7) ergibt eine Gesamtdistanz von 17. Der Wert ist niedriger als der zuvor ermittelte Wert von 18.

Knoten	**Vorgänger**	**Gesamtdistanz**	*erledigt*
A	A	0	x
B	A	5	x
G	A	7	x
C	B	10	x
D	B	10	x
E	G	16	
F	D	17	x

Tabelle 3.5: Dijkstra: Exemplarisch - Schritt 5

Warteschlange: D, E, F; Erledigt: A, B, C, D, G; Ausgewählt: D; Nachfolger: F

Der Vollständigkeit halber wird der Knoten E noch betrachtet, weil ein Algorithmus die Betrachtung ebenfalls in Erwägung zieht: lediglich F ist von E aus unbetrachtet und ermittelt E(16) + F(4) eine Gesamtdistanz von 20. F bleibt damit unverändert. Dazu die abschließenede Tabelle:

Knoten	**Vorgänger**	**Gesamtdistanz**	*erledigt*
A	A	0	x
B	A	5	x
G	A	7	x
C	B	10	x
D	B	10	x
E	G	16	x
F	D	17	x

Tabelle 3.6: Dijkstra: Exemplarisch - Schritt 6

Warteschlange: E; Erledigt: A, B, C, D, E, F, G; Ausgewählt: E; Nachfolger: F

Knoten F wird nicht mehr betrachtet, weil er als Zielknoten markiert ist. Ein Backtrace ergibt folgenden, bestimmten Weg:

$$\mathbf{F} \leftarrow \mathbf{D} \leftarrow \mathbf{B} \leftarrow \mathbf{A}$$

Ein Blick auf Tabelle 3.2 zeigt den Weg rückwärts, indem über Knoten F mit seinem Vorgänger D, dann der Knoten D mit seinem Vorgänger B betrachtet wird und Knoten B mit dem Vorgänger A, welcher der Startknoten ist.

Um den kürzeren Weg zu ermitteln unabhängig von der schnelleren Route, müssen die Kantengewichte gegen Entfernungsangaben ausgetauscht werden. Dann wird der rot-markierte Weg als Lösung ermittelt werden.

3.3 Informelle Beschreibung

Vorbereitung

1. Tabelle erstellen mit allen Knoten und den Attributen: Vorgänger und Gesamtdistanz
2. Gesamtdistanz des Startknotens auf 0 setzen und für alle anderen auf unendlich

Abarbeitung
Solange die Tabelle noch Knoten enthält, werden alle Elemente mit jeweils der kleinsten Gesamtdistanz genommen und folgendes gemacht:

1. Speichere, das Knoten betrachtet wurde
2. Prüfung: Ist das betrachtete Element der Zielknoten? Wenn ja, Backtrace über Vorgänger bis zum Start
3. Betrachte Nachbarknoten die noch in Tabelle sind:
   a) Kalkuliere Gesamtdistanz aus Summe der Gesamtdistanz des betrachteten Knotens und Distanz zum Nachbarelement
   b) Prüfe: Errechnete Gesamtdistanz kleiner als aktuelle Gesamtdistanz, dann überschreibe

3.4 Pseudocode

Für jeden Knoten im Graph wird die Distanz auf unendlich gesetzt und der Vorgänger auf unbekannt (null). Lediglich der Startknoten hat eine Distanz von 0. Die Menge Q enthält die Knoten, zu denen noch kein Weg gefunden wurde.

```
1 Methode initalisiereGraph(Graph, Starknoten, distanz[], vorgaenger[], Q):
2     fuer jeden Knoten m in Graph:
3         distanz[m] := unendlich
4         vorgaenger[m] := null
5     distanz[Startknoten] := 0
6     Q := Alle Knoten in Graph
```

Listing 3.1: Pseudocode Dijkstra: Graph initialisieren

Der Abstand vom Startknoten zum Knoten *m* wird dann geringer, wenn der Weg zu *m* über *n* kürzer als die bisherige Distanz ist. *n* wird zum Vorgänger vom betrachteten Knoten *m* auf dem kürzesten Weg und somit aktualisiert.

```
1 Methode distanzUpdate(n, m, distanz[], vorgaenger[]):
2     alternativ := distanz[n] + abstandZwischen(n, m)
3     wenn alternativ < distanz[m]:
4         distanz[m] := alternativ
5         vorgaenger[m] := n
```

Listing 3.2: Pseudocode Dijkstra: Knoten-Update

Ein kürzester Weg zwischen zwei Knoten wird gefunden, wenn der Algorithmus nach Zeile 5 der Dijkstra-Funktion abgebrochen wird, sofern *n* = Zielknoten ist. Der kürzeste Weg zu einem Zielknoten kann durch Iteration (Backtrace) über die Vorgänger ermittelt werden:

```
1 Funktion erstelleKuerzestenPfad(Zielknoten, vorgaenger[]))
2     Weg[] := [Zielknoten]
3     n := Zielknoten
4     solange vorgaenger[n] nicht null:
5         n := vorgaenger[n]
6         fuege n am Anfang von Weg[] ein
7     return Weg[]
```

Listing 3.3: Pseudocode Dijkstra: Kürzester Pfad

Zuerst werden alle Distanzen und Vorgänger für jeden Knoten im Graph initialisiert. So lange noch zu untersuchende Knoten in der Menge Q vorhanden sind, werden die Abstände gesucht. Für jeden gefundenen Nachbarn *m* eines Knoten *n* wird, sofern *m* sich noch in Q befindet, die Distanz aktualisiert.

```
1 Funktion Dijkstra(Graph, Startknoten):
2     initalisiereGraph(Graph, Starknoten, distanz[], vorgaenger[], Q)
3     solange Q nicht leer:
4         n := Knoten in Q mit kleinstem Wert in abstand[]
5         entferne n aus Q
6         fuer jeden Nachbarknoten m von n:
7             falls m in Q:
8                 distanzUpdate(n, m, distanz[], vorgaenger[])
9     return vorgaenger[]
```

Listing 3.4: Pseudocode Dijkstra: Dijkstra-Algorithmus

3.5 JAVA-Implementierung

Die Implementierung des Dijkstra-Algorithmus in JAVA beinhaltet eine `Node`-Klasse, die von der `GeneralNode`-Klasse erbt und eine Klasse für den Dijkstra-Algorithmus selbst.

3.5.1 Node.java

Für die Implementierung des Dijkstra-Algorithmus gibt es eine `Node`-Klasse, die von der abstrakten, generellen `Node`-Klasse erbt.

```
1 public class Node extends GeneralNode<Node> {
2     public Node(String name) {
3         super(name);
4     }
5 }
```

Listing 3.5: Dijkstra: Node.java

3.5.2 Dijkstra.java

Die Implementierung des Dijkstra-Algorithmus erfolgt nach dem Prinzip der informellen Beschreibung.

Die wichtigste Methode ist die `computeShortestPaths()`-Methode. Darin werden alle Pfade von einem angegeben Zielpunkt aus berechnet. Nachdem für alle Knoten im Graph alle Distanzen mit ∞ vorbelegt sind, wird für den Startknoten Kosten von 0 gesetzt. Nachdem die Listen vorbereitet sind, beginnt die eigentliche Abarbeitung des Graphen zur Bestimmung eines kürzesten Weges für die gegebenen Knoten für Start und Ziel.

Die Ergebnisse daraus werden in `shortestPath`-Liste gespeichert und können über die `showPath()`-Methode angezeigt werden.

```
1  public class Dijkstra {
2
3      private final Map<Node, Node> predecessors = new HashMap<>();
4      private final List<Node> shortestPath = new ArrayList<>();
5
6      public void computeShortestPaths(
7              @NonNull GeneralGraph<Node> graph,
8              Node start,
9              Node target) {
10         Map<Node, Integer> distances = new HashMap<>();
11         Set<Node> unvisitedNodes = new HashSet<>(graph.getAllNodes());
12
13         for (Node node : unvisitedNodes) {
14             distances.put(node, Integer.MAX_VALUE);
```

```
15          }
16          distances.put(target, 0);
17
18          while (!unvisitedNodes.isEmpty()) {
19              Node currentNode = getClosestDistances(unvisitedNodes, distances);
20              unvisitedNodes.remove(currentNode);
21
22              for (GeneralEdge<Node> adjacencyList :
23                      graph.getEdges(currentNode)) {
24                  if (unvisitedNodes.contains(adjacencyList.getDestination())) {
25                      Integer distance = distances.get(currentNode)
26                                          + adjacencyList.getWeight();
27                      if (distance < distances
28                                      .get(adjacencyList.getDestination())) {
29                          distances
30                              .put(adjacencyList.getDestination(), distance);
31                          predecessors
32                              .put(adjacencyList.getDestination(), currentNode);
33                      }
34                  }
35              }
36          }
37
38          Node currentNode = start;
39          if (predecessors.get(currentNode) == null) {
40              return;
41          }
42
43          shortestPath.add(currentNode);
44          while (predecessors.get(currentNode) != null) {
45              currentNode = predecessors.get(currentNode);
46              shortestPath.add(currentNode);
47          }
48
49          Collections.reverse(shortestPath);
50      }
51
52      public void showPath(Node start) {
53          System.out.println(printShortestPath(start));
54      }
55
56      private Node getClosestDistances(
57              Set<Node> unvisitedNodes,
58              Map<Node, Integer> distances) {
59          Node currentNode = null;
```

```
60          Integer closestDistance = Integer.MAX_VALUE;
61
62          for (Node node : unvisitedNodes) {
63              Integer currentDistance = distances.get(node);
64              if (currentDistance < closestDistance) {
65                  closestDistance = currentDistance;
66                  currentNode = node;
67              }
68          }
69          return currentNode;
70      }
71  }
```

Listing 3.6: Dijkstra.java

3.5.3 Beispielimplementierung

Das Beispiel ist exemplarisch als Unittest in das Projekt integriert. Der in dem Beispiel verwendete Graph, ist die programmatische Darstellung des Beispielgraphen aus Abbildung 3.1.

Zu Beginn wird eine Instanz der Klasse `GeneralGraph` implementiert und es werden alle vorhandenen Knoten als Instanzen einer Node instanziiert. Im Anschluss wird jeder Knoten über die `addNewNode()`-Methode zum Graphen hinzugefügt.

Nachdem alle Knoten des Graphen vorhanden sind, wird über die `addUndirectedEdge()`-Methode die zuvor erstellten `Node`-Objekte als ungerichtete Kante dem Graphen übergeben.

Mit einer Instanz der `Dijkstra`-Klasse werden alle Wege vom gegebenen Zielknoten mittels `computeShortestPaths()`-Methode zu allen vorhandenen Knoten berechnet, sofern vorhanden und in der `predecessors`-Map gespeichert.

Zum Anzeigen einer gewünschten Route, muss lediglich auf die `showPath()`-Methode zugegriffen werden.

```
1   public class DijkstraTestWithSampleGeneralGraph {
2
3       @Test
4       public void setGraph_StartNodeA_TargetNodeF_findWay() {
5           GeneralGraph<Node> graph = new GeneralGraph<>();
6           Node nodeA = new Node("A");
7           Node nodeB = new Node("B");
8           Node nodeC = new Node("C");
9           Node nodeD = new Node("D");
10          Node nodeE = new Node("E");
11          Node nodeF = new Node("F");
```

```
12          Node nodeG = new Node("G");
13
14          graph.addNewNode(nodeA);
15          graph.addNewNode(nodeB);
16          graph.addNewNode(nodeC);
17          graph.addNewNode(nodeD);
18          graph.addNewNode(nodeE);
19          graph.addNewNode(nodeF);
20          graph.addNewNode(nodeG);
21
22          graph.addUndirectedEdge(nodeA, nodeB, 5);
23          graph.addUndirectedEdge(nodeA, nodeC, 16);
24          graph.addUndirectedEdge(nodeA, nodeG, 7);
25          graph.addUndirectedEdge(nodeB, nodeC, 5);
26          graph.addUndirectedEdge(nodeB, nodeD, 5);
27          graph.addUndirectedEdge(nodeC, nodeE, 8);
28          graph.addUndirectedEdge(nodeC, nodeG, 4);
29          graph.addUndirectedEdge(nodeD, nodeE, 6);
30          graph.addUndirectedEdge(nodeD, nodeF, 7);
31          graph.addUndirectedEdge(nodeE, nodeF, 4);
32          graph.addUndirectedEdge(nodeE, nodeG, 9);
33          graph.addUndirectedEdge(nodeF, nodeG, 11);
34
35          Dijkstra dijkstra = new Dijkstra();
36          dijkstra.computeShortestPaths(graph,nodeA, nodeF);
37          dijkstra.showPath();
38      }
39  }
```

Listing 3.7: Dijkstra: Beispielimplementierung

Die Ausgabe aus dem Beispiel ist:

```
[F, D, B, A]
```

Listing 3.8: Dijkstra: Ausgabe

3.6 Vorteile

Die lineare Zeitkomplexität des Dijkstra-Algorithmus macht es einfach, den Algorithmus für große Probleme zu verwenden. Weiterhin ist der Algorithmus nützlich, wenn lediglich die kürzeste Entfernung bestimmt werden soll. Diese Berechnung findet beispielsweise in Routenplanungs-Werkzeugen Verwendung.

3.7 Nachteile

Es sind keine negativen Kantengewichte erlaubt. Ein wichtiger, nicht zu vernachlässigender Nachteil des Dijkstra-Algorithmus ist, dass für jede Anfrage einer Routenberechnung immer alle Richtungen aller Kanten berücksichtigt werden. Im oben angeführten Beispiel ist das eine zu vernachlässigende Tatsache, aber bei sehr großen Karten, wird die Laufzeit entsprechend hoch.

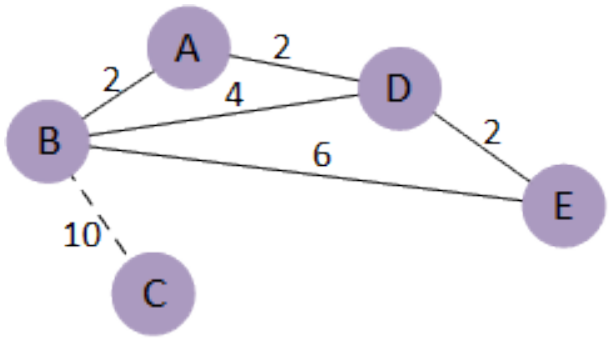

Abbildung 3.3: Ungeeigneter Graph

Wenn eine Route von E nach B betrachtet wird, sind die Kosten über jede Möglichkeit gleich groß. Ist das Ziel jedoch der Knoten C, gibt es keine Alternative zu einem Weg über Knoten B. Knoten B sei Startknoten und Knoten C sei Zielknoten, wird der Dijkstra-Algorithmus von Knoten B aus jeden Knoten über A, D und E prüfen, um dann zum Resultat $B \rightarrow C$ zu kommen.

3.8 Zeitkomplexität

Eine Abschätzung ist nur für Graphen, die keine negativen Kantengewichte enthalten, gültig. Weiterhin ist die Laufzeit des Algorithmus abhängig von der verwendeten Datenstruktur Q, in der Berechnung als T_{em} (”extract minimum”) bezeichnet, der Anzahl der Kanten $|E|$ und der Anzahl der Knoten $|V|$. [1]

Die Berechnung der Zeitkomplexität erfolgt schrittweise. Die Anzahl der Kanten m_{e} und die Anzahl der Knoten n_{v}:

[1] https://de.wikipedia.org/wiki/Dijkstra-Algorithmus#Zeitkomplexität

1. Startknoten in Tabelle eintragen ist konstant $\rightarrow O(1)$
2. Knoten aus Tabelle entnehmen in Abhängigkeit der Menge an Knoten und der verwendeten Datenstruktur $\rightarrow O(n_v \cdot T_{em})$
3. Prüfung, ob kürzester Weg bereits gefunden wurde für jede Kante und jeden Knoten. Verwendet man für diese Prüfung ein Set ist jeweils nur eine Kante zu prüfen, lineares Wachstum $\rightarrow O(m_e)$
4. Gesamtdistanz berechnen: erfolgt genau einmal pro Kante, da maximal einmal ein Weg zu einem Knoten gefunden wird $\rightarrow O(m_e)$
5. Eintrag in Tabelle, Abhängig von der Datenstruktur, bezeichnet mit T_i (”insert”) $\rightarrow O(n_v \cdot T_i)$
6. Update auf die Gesamtdistanz in der Tabelle pro Kante einmal. Auch hier besteht eine Abhängigkeit zur verwendeten Datenstruktur T_{dk} (”decrease key”). Für alle Kanten $\rightarrow O(m_e \cdot T_{dk})$

Daraus ergibt sich eine Summe von:

$$O(1) + O(n_v \cdot T_{em}) + O(m_e) + O(m_e) + O(m_e) + O(n_v \cdot T_i) + O(m_e \cdot T_{dk})$$

$O(1)$ kann vernachlässigt werden und $O(m_e)$ fallen weg. Durch Verkürzen des Terms ergibt sich:

$$O(n_v \cdot (T_{em} + T_i) + m_e \cdot T_{dk})$$

Bei der Implementierung werden alle Knoten betrachtet, dass heißt, es wird geprüft, ob von jedem Knoten eine Verbindung zu jedem anderen Knoten besteht. Daher ergibt sich für den Dijkstra-Algorithmus eine Zeitkomplexität von

$$O(n^2)$$

4 A-Stern-Algorithmus

Der A*-Algorithmus (”A Stern”) gehört zu den informierten Suchalgorithmen. Er berechnet einen kürzesten Pfad zwischen zwei Knoten in einem Graphen mit positiven Kantengewichten. Der Algorithmus gilt als Verallgemeinerung und Erweiterung des Dijkstra-Algorithmus, der in vielen Fällen aber auch umgekehrt auf den Dijkstra-Algorithmus reduziert werden kann.

Im Gegensatz zu uninformierten Suchalgorithmen verwendet der A*-Algorithmus eine Heuristik, um zielgerichtet zu suchen und damit die Laufzeit zu verringern. Der Algorithmus ist vollständig und optimal, was bedeutet, dass immer eine optimale Lösung gefunden wird, sofern eine existiert. [1] [3]

4.1 Beschreibung

Der A*-Algorithmus untersucht die Knoten zuerst, die wahrscheinlich schnell zum Ziel führen. Um den vielversprechendsten Knoten zu berechnen, wird allen Knoten x jeweils ein Wert $f(x)$ zugeordnet, der eine Abschätzung angibt, wie weit der Pfad vom Start zum Ziel unter Verwendung des betrachteten Knotens im optimalen Fall ist. Der Knoten mit den geringsten Kosten wird als nächster untersucht.

$$f(x) = g(x) + h(x)$$

Für einen Knoten x bezeichnet $g(x)$ die bisherigen Kosten vom Startknoten aus, um x erreichen zu können. $h(x)$ bezeichnet die geschätzten Kosten (Heuristik) von x bis zum Zielknoten. Die verwendete Heuristik darf die Kosten nie überschreiten. Für eine Wegsuche ist die euklidische Distanz eine geeignete Annahme: Die tatsächliche Strecke ist nie kürzer als die Luftlinie.

Der A*-Algorithmus ist vollständig, falls eine Lösung existiert, wird sie gefunden. Der Algorithmus ist optimal, es wird immer die optimale Lösung gefunden und wenn mehrere existieren, wird eine gefunden. Weiterhin ist der A*-Algorithmus optimal effizient, das heißt es gibt keinen anderen Algorithmus, der die Lösung unter Verwendung der gleichen Heuristik schneller findet.

4.2 Beispiel

Dieses Beispiel bezieht sich auf das vorangegangene Beispiel aus Abbildung 3.2, wurde jedoch mit anderen Kantengewichten versehen, die aufgrund der nachfolgenden Rechnungen näherungsweise an realistischen Ergebnissen sind. Es soll ein Weg zwischen Knoten A und F ermittelt werden.

4.2.1 Heuristik

Zur Messung von Entfernungen wird um den Beispielgraph exemplarisch ein Koordinatensystem zur Bestimmung der Entfernung gelegt:

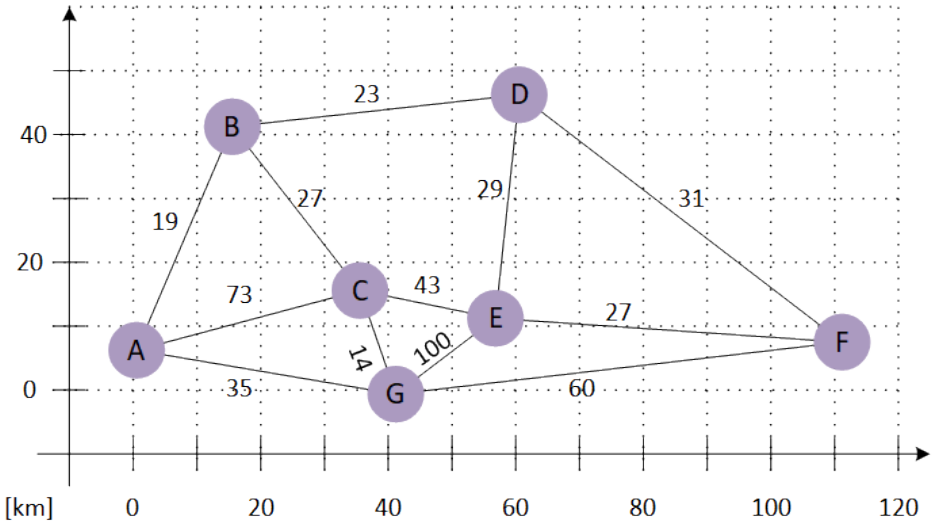

Abbildung 4.1: Fiktive Straßenkarte mit Koordinatensystem

Zur Bestimmung der euklidischen Distanz, muss die maximale Geschwindigkeit in diesem System ermittelt werden. Die Geschwindigkeit hilft beim Berechnen der näherungsweisen Bestimmung der mittleren Entfernung in Minuten der Orte untereinander.

Die Entfernung zwischen den Orten A-B und D-F wird mit Hilfe des Satz des Pythagoras bestimmt:

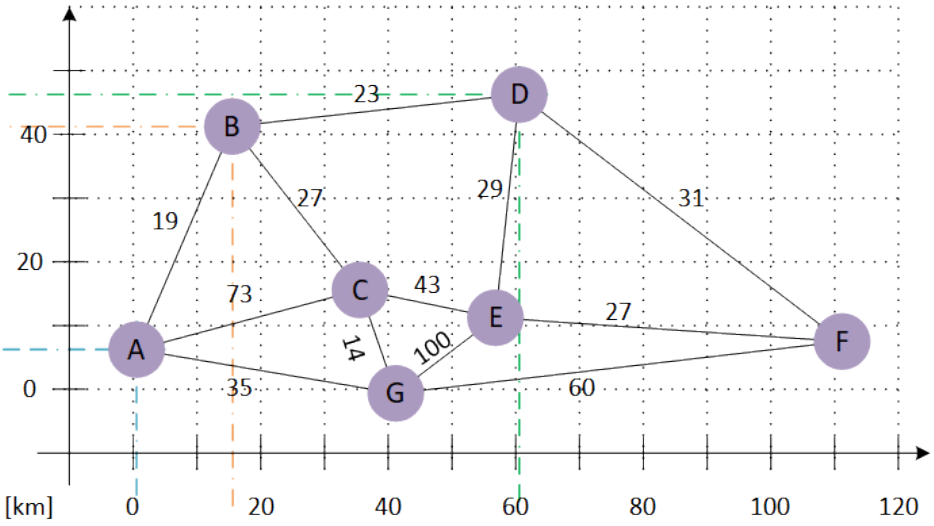

Abbildung 4.2: Fiktive Straßenkarte mit Koordinatensystem

Knoten	x	y
A	1,00	6,00
B	16,00	41,00
C	36,00	16,00
D	61,00	46,00
E	57,00	12,00
F	111,00	8,00
G	41,00	0,00

Tabelle 4.1: Ermittelte Koordinaten der Knoten

Für die Strecke AB:

$$s_{\text{AB}} = \sqrt{(16km - 1km)^2 + (41km - 6km)^2}$$
$$s_{\text{AB}} = \sqrt{(15km)^2 + (35km)^2} = 38,079km$$

Für die Strecke DF:

$$s_{\text{DF}} = \sqrt{(111km - 61km)^2 + (46km - 6km)^2}$$
$$s_{\text{DF}} = \sqrt{(50km)^2 + (40km)^2} = 64,031km$$

```
Entfernung: 38,079km
Kosten: 19min
Geschwindigkeit: 38,079km/19min = 2,0042km/min (~120,249km/h)
```

Listing 4.1: Berechnung mittlere Geschwindigkeit für Weg A-B

```
Entfernung: 62,801km
Kosten: 31min
Geschwindigkeit: 62,801km/31min = 2,0258km/min (~121,551km/h)
```

Listing 4.2: Berechnung mittlere Geschwindigkeit für Weg D-F

Die schnellstmögliche Geschwindigkeit (v_{max}) wird auf der Strecke *DF* erreicht. Das berechnen aller Geschwindigkeit ist in diesem Beispiel unerheblich, da die Karte initial so konstruiert wurde, das der bestimmte Abschnitt, der schnellste ist.

In modernen Navigationssystemen ist die schnellstmögliche Geschwindigkeit vorberechnet.

Mit $v_{\text{max}} = 2,0258km/min$ wird die euklidische Distanz von jedem Punkt der Karte zum Zielknoten F berechnet:

Knoten:	A	B	C	D	E	G
Distanz zu F in km:	110,018	100,568	75,425	62,801	54,148	70,456
Minimale Kosten in min:	54,309	49,644	37,232	31,001	26,729	34,779

Tabelle 4.2: A-Stern: Euklidische Distanz aller Knoten zum Zielknoten

Die minimalen Kosten pro Knoten entsprechen den kürzest möglichen Zeiten.

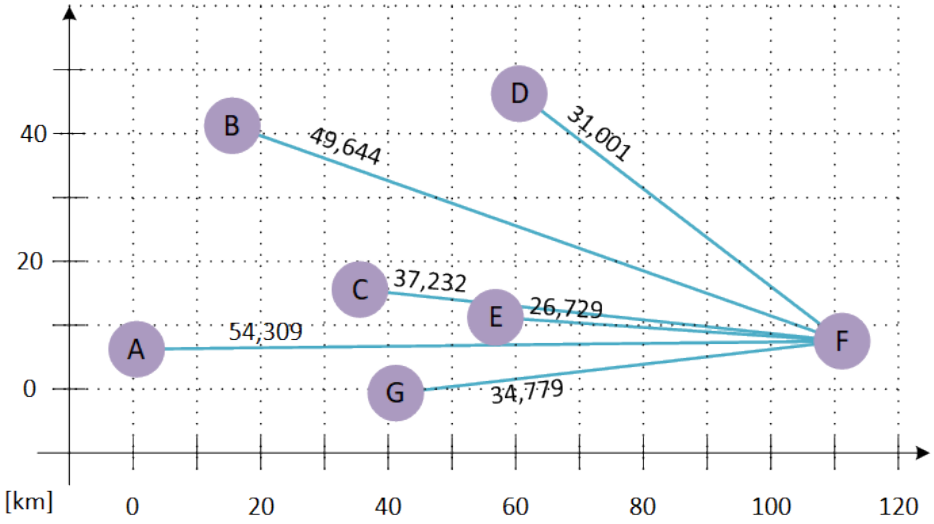

Abbildung 4.3: Fiktive Straßenkarte mit Koordinatensystem und berechneter euklidischer Distanz

4.2.2 Vorbereitung

Alle Knoten werden in einer Tabelle erfasst. Die Vorgänger-Knoten bleiben zunächst leer und als Gesamtkosten wird für den Startknoten der Wert 0 initial eingetragen. Alle anderen Knoten werden mit unendlich gefüllt.
Minimale Restkosten werden ebenfalls mit den zuvor berechneten Restkosten zum Zielknoten befüllt. Die Summe aller Kosten ist derzeit nicht bekannt und werden mit unendlich vorbefüllt. Ausgenommen der Startknoten, dessen Restkosten die errechneten Minimalkosten sind.

Die finale Tabelle stellt sich demnach folgendermaßen dar:

Knoten	**Vorgänger**	**Gesamtkosten vom Start**	**Minimale Restkosten zum Ziel**	**Summe aller Kosten**	*erledigt*
A	-	0	54,3	54,3	x
B	-	∞	49,6	∞	
C	-	∞	37,2	∞	
D	-	∞	31,0	∞	
E	-	∞	26,7	∞	
F	-	∞	0,0	∞	
G	-	∞	34,8	∞	

Tabelle 4.3: A-Stern: Exemplarisch - Vorbereitung

Erläuterung:
Kosten: Das Kantengewicht von einem Knoten zu seinem Nachbarn
Gesamtkosten: Die Summe aller Teilkosten vom Startknoten zu einem bestimmten Knoten über eventuelle Zwischenknoten
Restkosten: Mindestkosten berechnet durch die euklidische Distanz in Tabelle 4.2.1

4.2.3 Abarbeitung

Das Startelement A wird mit seinen Nachbarn B, C und G betrachtet. Die Kosten der Nachbarknoten sind derzeit unbekannt, werden nun auf die Kantengewichte von A an geändert. Die Gesamtkosten und die minimalen Restkosten zum Ziel werden addiert. Daraufhin ändert sich die Tabelle wie folgt:

Knoten	**Vorgänger**	**Gesamtkosten vom Start**	**Minimale Restkosten zum Ziel**	**Summe aller Kosten**	*erledigt*
A	A	0	54,3	54,3	x
B	A	19	49,6	68,6	
G	A	35	34,8	69,8	
C	A	73	37,2	110,2	
D	-	∞	31,0	∞	
E	-	∞	26,7	∞	
F	-	∞	0,0	∞	

Tabelle 4.4: A-Stern: Exemplarisch - Schritt 1

Die Knoten B,C und G wurden entdeckt. Diese können über den Knoten A in 19, 73 und 35 Minuten erreicht werden. Durch Addition auf die minimalen Restkosten des Ziels, braucht man mindestens 68, 110 oder 69 Minuten zum gewünschten Ziel.

Im nächsten Schritt wird der Knoten aus der Tabelle betrachtet, der als nächstes unbetrachtet ist: Knoten B. Die Nachbarn sind A, C und D. Knoten A gilt als erledigt, wird daher nicht mehr betrachtet. Die Gesamtkosten von Knoten D sind noch unendlich und werden dazu ausgerechnet. Gesamkosten vom Start für B sind 19, addiert mit den Kosten zu D ergeben 19 + 23 = 42.

Für den Knoten C wurde bereits ein Weg gefunden. Werden die Kosten über B nach C addiert, ergeben sich 19 + 27 = 46 Minuten. Das bedeutet, das ein Weg über B nach C günstiger ist, als ein direkter Weg von A nach c. Demnach wird dieser aktualisiert und Knoten B als betrachtet markiert.

Für die Tabelle bedeutet das:

Knoten	**Vorgänger**	**Gesamtkosten vom Start**	**Minimale Restkosten zum Ziel**	**Summe aller Kosten**	*erledigt*
A	A	0	54,3	54,3	x
B	A	19	49,6	68,6	x
G	A	35	34,8	69,8	
D	B	42	31,0	73,0	
C	B	46	37,2	83,2	
E	-	∞	26,7	∞	
F	-	∞	0,0	∞	

Tabelle 4.5: A-Stern: Exemplarisch - Schritt 2

Es erfolgt die Betrachtung des Knoten G, weil dieser als nächster als unbetrachtet in der Tabelle vorkommt. Die Nachbarn des Knoten sind A, C, E und F. A wird abermals ignoriert, da bereits erledigt. Die ermittelten Kosten nach C über A nach G sind 49 Minuten, somit höher als über B nach C. Es ändert sich nichts an C.
E ist bislang unbetrachtet, erhält als Vorgänger den Knoten G, mit Gesamtkosten von 35 + 100 Minuten und einer Summe von 161,729 Minuten.
Gleiches gilt für den Knoten F: Es ergeben sich 95 Minuten Gesamtkosten. Es gibt keine Restkosten, da F das definierte Ziel der Suche ist. Allerdings könnte ein kürzerer Weg existieren, da noch nicht alle Knoten vollständig abgearbeitet wurden.

Die Tabelle ändert sich:

Knoten	**Vorgänger**	**Gesamtkosten vom Start**	**Minimale Restkosten zum Ziel**	**Summe aller Kosten**	*erledigt*
A	A	0	54,3	54,3	x
B	A	19	49,6	68,6	x
G	A	35	34,8	69,8	x
D	B	42	31,0	73,0	
C	B	46	37,2	83,2	
F	G	95	0,0	95,0	
E	G	135	26,7	161,7	

Tabelle 4.6: A-Stern: Exemplarisch - Schritt 3

Die Betrachtung von D erschließt die Nachbarschaft zu B, E und F. B wird ignoriert und der Weg nach E untersucht. Die Gesamtkosten betragen 71 Minuten, welche kürzer sind als die zuvor ermittelten Kosten über G. Der Weg nach F reduziert sich über D ebenfalls auf 73 Minuten der Gesamtkosten ohne minimale Restkosten.
Das Ziel F wurde abermals erreicht. Die beiden offenen Knoten werden nach der Aktualisierung der Tabelle nicht mehr untersucht. Die Gesamtkosten zum Ziel sind bei Knoten C und Knoten E höher, sodass der bisher erreichte Wert von 73,000 nicht mehr verbessert werden kann.

Knoten	**Vorgänger**	**Gesamtkosten vom Start**	**Minimale Restkosten zum Ziel**	**Summe aller Kosten**	*erledigt*
A	A	0	54,3	54,3	x
B	A	19	49,6	68,6	x
G	A	35	34,8	69,8	x
D	B	42	31,0	73,0	x
F	D	73	0,0	73,0	
C	B	46	37,2	83,2	
E	D	71	26,7	97,7	

Tabelle 4.7: A-Stern: Exemplarisch - Schritt 4

Alle möglichen Routen wurden betrachtet und es gibt keine Kostenveränderung mehr nach dem Erfassen aller Werte für Tabelle 4.2.3. Der Backtrace des vollständig ermittelten Weges ergibt:

F ← D ← B ← A

4.3 Informelle Beschreibung

Vorbereitung

1. Tabelle erstellen mit allen Knoten, den entsprechenden Vorgängerknoten, die Gesamtkosten vom Start, den minimalen Restkosten zum Ziel und der Summe aller Kosten
2. Gesamtkosten des Startknotens auf 0 setzen und für alle anderen auf unendlich
3. Mittels Heuristik die minimalen Restkosten zum Ziel für alle Knoten bestimmen

Abarbeitung
Solange die Tabelle noch Knoten enthält, werden alle Elemente mit jeweils der kleinsten Kostensumme genommen und folgendes gemacht:

1. Prüfung: Ist das betrachtete Element der Zielknoten? Wenn ja, Backtrace über Vorgänger bis zum Start
2. Betrachte Nachbarknoten des entnommenen Elements die noch in Tabelle sind:
   a) Ermittle die Gesamtkosten vom Start als Summe der Gesamtkosten vom Start zum entnommenen Knoten addiert mit den Kosten vom entnommenen Knoten zum betrachteten Nachbarknoten
   b) Prüfe: Errechnete Gesamtdistanz kleiner als aktuelle Gesamtdistanz, dann
      i. Berechne für den Nachbarknoten die Summe aus den berechneten Gesamtkosten vom Start und den Restkosten zum Ziel
      ii. Setze als Vorgänger des Nachbarknotens den entnommenen Knoten
      iii. Setze für den Nachbarknoten die neu berechneten Gesamtkosten und die Kostensumme

4.4 Pseudocode

Die Openlist wird als Prioritätenwarteschlange deklariert und die Closedlist als Set.

```
1 deklariere openlist als PriorityQueue mit Knoten
2 deklariere closedlist als Set mit Knoten
```

Listing 4.3: Pseudocode A-Stern: Deklaration

Zu Beginn ist sowohl die Openlist als auch die Closedlist leer. Die Priorität beziehungsweise die Gesamtkosten des Startknotens sind unerheblich und der Startknoten wird der Openlist mit

seinen Gesamtkosten von 0 hinzugefügt. Die Schleife wird solange durchlaufen, bis entweder eine optimale Lösung gefunden wurde, oder festgestellt wurde, dass für die zu berechnenden Daten keine Lösung existiert.

Der Knoten mit den geringsten Kosten wird aus der OPENLIST entfernt. Dann wird geprüft, ob der aktuell betrachtete Knoten dem Zielknoten entspricht. Wenn ja, dann wurde ein Weg gefunden, wenn Nein, dann wird der Knoten der CLOSEDLIST hinzugefügt und alle nachfolgenden Knoten der OPENLIST hinzugefügt.

Die Wiederholung endet, wenn die OPENLIST leer ist.

```
1  Funktion a-star
2      setze in openlist startknoten[s] mit Kosten 0
3
4      wiederhole
5          aktuellerKnoten[m] := entferne Knoten mit Minimum Kosten aus openlist
6          wenn aktuellerKnoten[m] == zielKnoten[m] dann
7              return Graph
8
9          fuege zu closedlist aktuellerKnoten[m]
10         nachbarKnoten(aktuellerKnoten[m])
11     soalnge bis openlist leer ist
12
13     return leerer Graph
```

Listing 4.4: Pseudocode A-Stern: A-Stern

Überprüft alle Nachbarknoten eines gegebenen Knotens *m* und fügt sie der OPENLIST hinzu, wenn der Folgeknoten *n* zum ersten Mal entdeckt wurde, oder ein günstigerer Weg zu diesem Knoten gefunden wird.

Wenn sich der Folgeknoten eines zu untersuchenden Knoten bereits auf der CLOSEDLIST befindet, passiert nichts.

Die vorläufigen Kosten werden berechnet, in dem die Restkosten bis zum Ziel *z* mit den Kosten des aktuellen Knotens *m* addiert werden. Ist der Folgeknoten *n* bereits auf der OPENLIST, der neue Weg nicht optimaler als der bekannte Weg, dann passiert nichts.
Falls der nun gefundene Weg optimaler ist, werden die Kosten durch die vorher ermittelten vorläufigen Kosten ersetzt. Die Gesamtkosten werden in der OPENLIST aktualisiert beziehungsweise der Knoten mit den entsprechenden Kosten in die OPENLIST eingefügt.

```
1  Methode nachbarKnoten(m, z):
2      fuer jeden Nachfolger n von m
3          wenn closedlist n enthaelt dann
4              weiter
5
```

```
6           vorlaeufigeKosten := kosten[m] + distanz(m, z)
7           wenn n in openlist und vorlaeufigeKosten >= vorgaengerKosten dann
8               weiter
9
10          Vorgaenger k von n := m
11          vorgaengerKosten := vorlaeufigeKosten
12
13          gesamtkosten := vorlaeufigeKosten + distanz[n]
14          wenn openlist n enthaelt dann
15              aktualisiere fuer n die gesamtkosten in openlist
16          sonst
17              setze (n, gesamtkosten) in openlist
```

Listing 4.5: Pseudocode A-Stern: Nachbarknoten

4.5 JAVA-Implementierung

Die beispielhafte Implementierung des A-Stern-Algorithmus in JAVA erfolgt anhand der informellen Beschreibung aus Kapitel 4.3.

Die Besonderheit bei der Implementierung ist eine Erweiterung um eine Heuristik und eine eigene Klasse die die Distanzen der Nodes behandelt.

4.5.1 Node.java

Die `Node`-Klasse erbt von der `GeneralNode`-Klasse und wird um zwei zusätzliche Attribute, einer x-Koordinate und einer y-Koordinate. Diese Attribute werden für die Heuristik benötigt.

```
1  @Getter
2  public class Node extends GeneralNode<Node> {
3      private final Double x;
4      private final Double y;
5
6      public Node(String name, Double x, Double y) {
7          super(name);
8          this.x = x;
9          this.y = y;
10     }
11 }
```

Listing 4.6: A-Star: Node.java

4.5.2 NodeDistance.java

Die `NodeDistance`-Klasse ist eine Hilfsklasse die lediglich zum Speichern von Distanzen zu anderen Knoten ist. Die Klasse beinhaltet eine aktuelle Node und eine Distanz vom Typ `Double`.

```
1  @Getter
2  public class NodeDistance implements Comparable<NodeDistance> {
3      private final Node node;
4      private final Double distance;
5
6      public NodeDistance(Node node, Double distance) {
7          this.node = node;
8          this.distance = distance;
9      }
10
11    @Override
12     public int compareTo(NodeDistance other) {
13         return Double.compare(this.distance, other.distance);
14     }
15 }
```

Listing 4.7: A-Star: NodeDistance.java

4.5.3 Heuristic.java

Die `Heuristic`-Klasse bildet die Implementierung der Methoden ab, die zur Berechnung oder Verwendung der Heuristik benötigt werden.

In dieser Implementierung wird eine statische Variable `vMAX` verwendet, die die schnellstmögliche Geschwindigkeit des verwendeten Beispielgraphen repräsentiert. Die Berechnung erfolgt in Listing 4.2 anhand des Beispielgraphen.

Die Methode zur Berechnung der Heuristiken ist statisch, sodass bei der Verwendung keine eigene Klasse instanziiert werden muss.

Durch übergeben des Zielknotens in die `computeHeuristics()`-Methode, wird für jeden Knoten innerhalb des Graphen die euklidische Distanz zum Zielknoten mit der privaten Methode `computeEuclideanDistance()` berechnet und in einer `MAP` gespeichert.

```
1  public class Heuristic {
2
3      // average Speed calculated by Euclidean distance
4      // between node D and node f (distance divided by weight)
5      private static final Double vMAX = 2.0258;
6
7      /**
```

```
8       * Calculate costs from current node to given target node.
9       * Euclidean Distance multiplied with given maximum speed
10      *
11      * @param graph  Graph
12      * @param target Target Node
13      * @return Map with costs from all nodes in graph to target Node
14      */
15     public static Map<Node, Double> computeHeuristics(
16             GeneralGraph<Node> graph,
17             Node target) {
18         Map<Node, Double> heuristics = new HashMap<>();
19         Double distance = null;
20         Double cost = null;
21
22         for (Node currentNode : graph.getAllNodes()) {
23             if (currentNode.equals(target)) {
24                 heuristics.putIfAbsent(currentNode, 0.0);
25             }
26
27             distance = computeEuclideanDistance(currentNode, target);
28             cost = distance / vMAX;
29             heuristics.put(currentNode, cost);
30         }
31
32         return heuristics;
33     }
34
35     /**
36      * Computing the Euclidean Distance between two nodes
37      *
38      * @param from first node
39      * @param to   second node
40      * @return distance as double
41      */
42     private static Double computeEuclideanDistance(Node from, Node to) {
43         double distanceX = to.getX() - from.getX();
44         double distanceY = to.getY() - from.getY();
45         return Math.sqrt(distanceX * distanceX + distanceY * distanceY);
46     }
47 }
```

Listing 4.8: A-Star: Heuristic.java

4.5.4 AStar.java

Die Hauptklasse zur Berechnung eines kürzesten Weges mit dem A-Stern-Algorithmus ist anhand der informellen Beschreibung aus Kapitel 4.3 implementiert.

Eingangs werden alle Knoten mit Kosten von ∞ initialisiert und die euklidischen Distanzen von jedem Knoten im Graphen zum gegebenen Zielknoten berechnet und gespeichert. Danach werden die Kosten des Startknotens auf Kosten von 0 gesetzt und die unbestimmten Restkosten auf ∞.

Nun beginnt die eigentliche Berechnung aller Kosten im Graphen zum Ziel.

Die private `printShortestPath()`-Methode fügt den berechneten Weg in eine Liste ein, die anschließend rückwärts sortiert wird, um das Ergebnis für eine Ausgabe oder Weiterverwendung vorzubereiten.

Mittels `showPath()` kann das Ergebnis der Berechnung ausgegeben bzw. angezeigt werden.

```
1  public class AStar {
2
3      private final Map<Node, Node> predecessors = new HashMap<>();
4      private final List<Node> shortestPath = new ArrayList<>();
5
6      public void computeShortestPath(
7              GeneralGraph<Node> graph,
8              Node start,
9              Node target) {
10         Map<Node, Double> gScores = new HashMap<>();
11         Map<Node, Double> fScores = new HashMap<>();
12         PriorityQueue<NodeDistance> openSet = new PriorityQueue<>();
13
14         for (Node node : graph.getAllNodes()) {
15             gScores.put(node, Double.MAX_VALUE);
16             fScores.put(node, Double.MAX_VALUE);
17         }
18
19         Map<Node, Double> heuristics = Heuristic
20                                 .computeHeuristics(graph,target);
21         gScores.put(start, 0.0);
22         fScores.put(start, heuristics.getOrDefault(start, Double.MAX_VALUE));
23         openSet.add(new NodeDistance(start, fScores.get(start)));
24
25         while (!openSet.isEmpty()) {
26             NodeDistance current = openSet.poll();
27             Node currentNode = current.getNode();
28
```

```
29              if (currentNode.equals(target)) {
30                  printShortestPath(target);
31                  return;
32              }
33
34              for (GeneralEdge<Node> edge : graph.getEdges(currentNode)) {
35                  Node neighbour = edge.getDestination();
36                  Double tentativeGScore = gScores.get(currentNode)
37                                          + edge.getWeight();
38
39                  if (tentativeGScore < gScores.get(neighbour)) {
40                      predecessors.put(neighbour, currentNode);
41                      gScores.put(neighbour, tentativeGScore);
42                      fScores
43                          .put(neighbour,
44                              tentativeGScore + heuristics
45                              .getOrDefault(neighbour, Double.MAX_VALUE));
46                      openSet.add(
47                          new NodeDistance(neighbour, fScores.get(neighbour)));
48                  }
49              }
50          }
51
52      }
53
54      private void printShortestPath(Node currentNode) {
55          while (currentNode != null) {
56              shortestPath.add(currentNode);
57              currentNode = predecessors.get(currentNode);
58          }
59
60          Collections.reverse(shortestPath);
61      }
62
63      public void showPath() {
64          System.out.println(shortestPath);
65      }
66  }
```

Listing 4.9: A-Star: AStar.java

4.5.5 Beispielimplementierung

Die Beispielimplementierung ist exemplarisch als Unittest auf Grundlage des Beispielgraphes aus Abbildung 3.2 in das Projekt integriert.

Am Anfang wird eine Instanz der Klasse `GeneralGraph` implementiert und es werden alle vorhandenen Knoten als Instanzen einer Node instanziiert. Im Anschluss wird jeder Knoten über die `addNewNode()`-Methode zum Graphen hinzugefügt.

Nachdem alle Knoten des Graphen vorhanden sind, wird über die `addUndirectedEdge()`-Methode die zuvor erstellten `Node`-Objekte als ungerichtete Kante dem Graphen übergeben.

Mit einer Instanz der `AStern`-Klasse werden alle Wege vom gegebenen Zielknoten mittels `computeShortestPath()`-Methode zu allen vorhandenen Knoten berechnet, sofern vorhanden und in der `predecessors`-Map gespeichert.

Zum Anzeigen einer gewünschten Route, wird auf die `showPath()`-Methode zugegriffen.

```
1  public class AStarTestWithSampleGeneralGraph {
2
3      @Test
4      public void setGraph_StartNodeA_TargetNodeF_findWay() {
5          GeneralGraph<Node> graph = new GeneralGraph<>();
6          Node nodeA = new Node("A", 1.0, 6.0);
7          Node nodeB = new Node("B", 16.0, 41.0);
8          Node nodeC = new Node("C", 36.0, 16.0);
9          Node nodeD = new Node("D", 61.0, 46.0);
10         Node nodeE = new Node("E", 57.0, 12.0);
11         Node nodeF = new Node("F", 111.0, 8.0);
12         Node nodeG = new Node("G", 41.0, 0.0);
13
14         graph.addNewNode(nodeA);
15         graph.addNewNode(nodeB);
16         graph.addNewNode(nodeC);
17         graph.addNewNode(nodeD);
18         graph.addNewNode(nodeE);
19         graph.addNewNode(nodeF);
20         graph.addNewNode(nodeG);
21
22         graph.addUndirectedEdge(nodeA, nodeB, 19);
23         graph.addUndirectedEdge(nodeA, nodeC, 73);
24         graph.addUndirectedEdge(nodeA, nodeG, 35);
25         graph.addUndirectedEdge(nodeB, nodeC, 27);
26         graph.addUndirectedEdge(nodeB, nodeD, 23);
27         graph.addUndirectedEdge(nodeC, nodeE, 43);
28         graph.addUndirectedEdge(nodeC, nodeG, 14);
```

```
29          graph.addUndirectedEdge(nodeD, nodeE, 29);
30          graph.addUndirectedEdge(nodeD, nodeF, 31);
31          graph.addUndirectedEdge(nodeE, nodeF, 27);
32          graph.addUndirectedEdge(nodeE, nodeG, 100);
33          graph.addUndirectedEdge(nodeF, nodeG, 60);
34
35          AStar aStar = new AStar();
36          aStar.computeShortestPath(graph, nodeA, nodeF);
37          aStar.showPath();
38      }
39  }
```

Listing 4.10: A-Star: Beispiel

Die Ausgabe aus dem Beispiel ist:

```
[A, B, D, F]
```

Listing 4.11: A-Star: Ausgabe

4.6 Vorteile

Im Vergleich zum Dijkstra-Algorithmus zeigt das vorherige Beispiel, dass die Knoten C und E noch betrachtet worden wären. Durch die eingebundene Heuristik wurden diese möglichen Routen jedoch zuvor eliminiert und eine Routenberechnung war deutlich schneller und effizienter. Der A-Stern-Algorithmus hat also in dem konkreten Beispiel zwei Schritte gespart.

Allgemein betrachtet findet ein A-Stern-Algorithmus immer den kürzesten Pfad, sofern eine optimale Heuristik verwendet wird. Der Algorithmus hat eine ausgepsrochen hohe Effizienz, da er im Vergleich zu uninformierten Algorithmen eine Heuristik verwendet, um vielversprechende Pfade zu priorisieren. Weiterhin ist der A-Stern-Algorithmus vielseitig anpassbar und kann dadurch für verschiedene Problemstellungen eingesetzt werden, in dem lediglich die Heuristik entsprechend gewählt wird.

4.7 Nachteile

Nachteile des A-Stern-Algoithmus sind vor allem seine Abhängigkeit von Heuristik und sein Speicherbedarf. Die Qualität der Heuristik ist maßgebend für die Effizienz und die Genauigkeit des Algorithmus. Eine schlechte Heuristik kann demnach zu ineffizienten oder suboptimalen Pfaden führen. Üblicherweise speichert der Algorithmus alle bekannten Knoten in einer Prioritätenwarteschlange (Openlist). Bei besonders großen Graphen kann der Speicherbedarf erheblich sein.

4.8 Zeitkomplexität

Zur Bestimmung der Zeitkomplexität für den A-Stern-Algorithmus, wird die Komplexität der verwendeten PriorityQueue benötigt.

Laut der offiziellen JAVA-Dokumentation[1] ist die Komplexität für

- POLL(): $T_{\text{extractMinimum}} = O(\log n)$
- OFFER(): $T_{\text{insert}} = O(\log n)$
- REMOVE() und OFFER(): $T_{\text{decreaseKey}} = O(n) + O(\log n) = O(n)$

Durch das Einsetzen der Parameter aus der Dokumentation in die allgemeine Formel $O(n \cdot (T_{\text{em}} + T_{\text{i}}) + m \cdot T_{\text{dk}}$ ergibt sich folgende Berechnung:

$$O(n \cdot (\log n + \log n) + m \cdot n)$$

für $\log n + \log n = 2 \cdot \log n$ kann aufgrund der Konstante gekürzt werden.

$$O(n \cdot \log n + m \cdot n)$$

Für den Sonderfall $m \in O(n)$ (die Anzahl der Kanten ist ein Vielfaches der Anzahl der Knoten), kann die Formel zu $O(n \cdot \log n + n^2)$ vereinfacht werden. Neben dem quadratischen Anteil n^2 kann der quasilineare Anteil $n \cdot \log n$ vernachlässt werden. Es bleibt:

$$O(n^2) \text{ für } m \in O(n)$$

Der Einsatz einer PriorityQueue führt also zu quadratischem Aufwand, einer deutlich schlechteren Komplexitätsklasse als quasilinearer Aufwand.

[1] `https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/PriorityQueue.html`, abgerufen 26.07.2024

5 Bellman-Ford-Algorithmus

Der Algorithmus von Bellman und Ford ist ein Suchalgorithmus der Graphentheorie, der den kürzesten Weg in einem kantengewichteten Graphen berechnet. Im Gegensatz zum Dijkstra-Algorithmus und dem A-Stern-Algorithmus kann der Bellman-Ford-Algorithmus auch mit negativen Kantengewichten umgehen.

Der Bellman-Ford-Algorithmus berechnet, wie die anderen Kürzeste-Wege-Algorithmen auch, die günstigsten Wege von einem Startknoten zu allen anderen Knoten im Graphen. Dabei arbeitet er iterativ und verbessert schrittweise die Schätzung der Kosten, bis die korrekten Werte gefunden sind. [2] [4]

5.1 Beschreibung

Der Bellman-Ford-Algorithmus ähnelt dem Algorithmus von Dijkstra. Der Unterschied ist, dass im Gegensatz zum Dijkstra-Algorithmus keine Knoten priorisiert werden, sondern dass der Iteration aller Kanten gefolgt wird und die Gesamtkosten vom Startknoten zum Zielknoten aktualisiert werden, sofern diese eine Verbesserung gegenüber des Aktualzustands bedeuten. Aufgrund der Untersuchung aller vorhandenen Knoten ist die Anzahl der Iteration maximal $n - 1$.

5.2 Beispiel

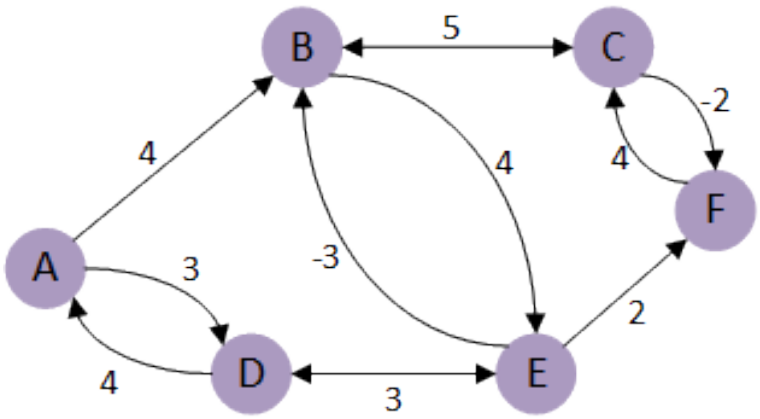

Abbildung 5.1: Fiktive Karte als gerichteter und gewichteter Graph

Als Beispiel dient ein fiktiver Graph, wie aus Abbildung 5.1. Mit einer Vorgabe der Suche nach einer Route von Knoten A zum Knoten E, fände der Dijsktra-Algorithmus lediglich die Möglichkeit von

$$\mathbf{A \rightarrow D \rightarrow E \rightarrow F}$$

mit Gesamtkosten von 5 + 4 + 2 = 11. Eine mögliche Route von

$$\mathbf{A \rightarrow B \rightarrow C \rightarrow F}$$

würde wegen des negativen Kantengewichts nicht in Betracht gezogen, obwohl dessen Gesamtkosten lediglich 5 + 7 - 3 = 9 betragen.

Als Vorbereitung wird eine Tabelle nach folgendem Schema erstellt:

Knoten	Vorgänger	Gesamtkosten vom Start
A	-	0
B	-	∞
C	-	∞
D	-	∞
E	-	∞
F	-	∞

Tabelle 5.1: Bellman-Ford: Exemplarisch - Vorbereitung

Erläuterung:
Kosten: Das Kantengewicht von einem Knoten zu seinem Nachbarn
Gesamtkosten: Die Summe aller Teilkosten vom Startknoten zu einem bestimmten Knoten über eventuelle Zwischenknoten

Die Tabelle wird nacheinander abgearbeitet und jeder Vorgängerknoten eines aktuell betrachteten Knoten untersucht. Die Iteration wird $n - 1$ mal durchgeführt. Im konkreten Beispiel demnach für 6 Knoten also 5 Iterationen. Es gibt beim Bellman-Ford-Algorithmus keine Priorisierung, was bedeutet, dass alle Knoten der Reihe nach mit ihren jeweiligen Verbindungen abgearbeitet werden.

5.2.1 Iteration 1

Iteration 1 wird exemplarisch vollständig durchgeführt.

Kante A-B
Die Summe der Gesamtkosten vom Start zu A betragen 0, da A selbst der Startknoten ist. Die Gesamtkosten für den Knoten B errechnen sich durch die Addition der Kosten von A und der Durch die Betrachtung der Kante(A, B) wird für Knoten B der Vorgänger A eingetragen, mit dem errechneten Gesamtgewicht von 4.

Kante A-D
Die Gesamtkosten von Start zu Knoten A betragen weiterhin 0. Addiert mit den Gesamtkosten der Kante(A, D) mit dem Betrag 3, ergeben sich Gesamtkosten von 3, mit dem Vorgänger A.

Kante B-C
Für Kante(B, C) werden die zuvor ermittelten Gesamtkosten von B über A mit den Kosten nach Kante C addiert. Das ergibt Gesamtkosten von $4 + 5 = 9$. Als Vorgänger wird für Kante C die Kante B notiert.

Kante B-E
Knoten E hat Gesamtkosten von 8 aus dem Ergebnis von Knoten B mit 4 und dem Kantengewicht von 4. Dem Knoten E wird ein Vorgänger B zugewiesen.

Kante C-B
Knoten C hat ermittelte Kosten über Knoten B mit Gesamtkosten von 9. Das Kantengewicht nach Kante B beträgt 5. Daraus resultieren Gesamtkosten von 14 für Knoten B. Diese Kosten sind allerdings höher als die Kosten von Knoten A nach Knoten B. Es wird nichts aktualisiert.

Kante C-F
Die Gesamtkosten für den Knoten F ergeben sich aus den Kosten der Kante C von 9 mit dem Kantengewicht der Kante(C, F) mit dem Betrag von -2. $9 + (-2) = 7$ für die Gesamtkosten für den Knoten F. Damit hat F den Vorgänger C und die Kosten von 7.

Kante D-A
Der Knoten A wird ermittelt über das Gesamtkosten von 3 aus dem Knoten A über D und dem Kantengewicht der Kante(D, A) von 4. Die Gesamtkosten betragen 7. Keine Aktualisierung.

Kante D-E
Die bereits ermittelten Gesamtkosten mit dem Betrag 3 von Knoten D, werden mit dem Kantengewicht der Kante(D, E) addiert, was Gesamtkosten von $3 + 3 = 6$ ergeben. Die nun berechneten Kosten von Knoten E sind geringer als die aus Kante(B, E). Somit wird Knoten E aktualisiert. Der Vorgänger von Knoten E ist der der Knoten D.

Kante E-B
Mit Gesamtkosten von 6 für den Knoten E und dem zu addierenden Kantengewicht der Kante (E, B) von -3, ergeben sich Gesamtkosten von $6 + (-3) = 3$ den Knoten B. Eine Aktualisierung findet statt, weil die nun ermittelten Kosten geringer sind, als die zuvor für B ermittelten Kosten über die Kante(A, B) von 4.

Kante E-D
Ermittelte Kosten von 6 für Knoten E, addiert mit dem Kantengewicht der Kante(E, D) von 3 ergibt Gesamtkosten für den Knoten D von 9. Das ist mehr als die zuvor ermittelten Kosten für Knoten D über Knoten A. Es findet keine Aktualisierung statt.

Kante E-F
Die derzeitigen Gesamtkosten von E betragen 6. Addiert mit dem Kantengewicht von 2 über die Kante(E, F) ergeben für die Kante F Kosten von 8. Das ist mehr als das zuvor ermittelte Ergebnis über die Kante(C, F) von 7.

Kante F-C
Die derzeitigen Gesamtkosten von F betragen 7. Addiert mit dem Kantengewicht von 4 über die Kante(F, C) ergeben sich für die Kante C Kosten von 11. Das ist mehr als das zuvor ermittelte Ergebnis über die Kante(B, C) von 9. Keine Aktualisierung.

Zur Übersicht, eine aktualisierte Tabelle:

Knoten	Vorgänger	Gesamtkosten vom Start
A	-	0
B	E	3
C	B	9
D	A	3
E	D	6
F	C	7

Tabelle 5.2: Bellman-Ford: Exemplarisch - Iteration 1

Am Ende der Iteration wurden alle Kanten genau einmal betrachtet und eine Route mit Gesamtkosten von 28 nach F ermittelt.
Der Backtrace von E nach A ergibt: $F \rightarrow C \rightarrow B \rightarrow E \rightarrow D \rightarrow A$ mit Kosten von $7 + 9 + 3 + 6 + 3 + 0 = 28$.

Da möglicherweise über alternative Wege ein kürzerer Weg gefunden werden kann, wird die gesamte Iteration wiederholt.

5.2.2 Iteration 2

Iteration 2 wird nicht vollständig vorgeführt.

Kante A-B und Kante A-D
Kante A hat jeweils aufgrund ihres Status als Startknoten Gesamtkosten von 0. Durch Addition des jeweiligen Kantengewichts ergeben sich für Knoten B Kosten von 4 und für Knoten D Kosten von 3. Es ändert sich nichts.

Kante B-C und Kante B-E
Für Kante B werden jeweils Kosten von 3 erhoben und für Knoten C mit dem Kantengewicht für Kante(B, C) von 5 addiert und für Knoten E mit einem Kantengewicht für Kante(B, E) von 4 addiert. Knoten C hat demnach Kosten von 8 und Knoten E Kosten von 7. Auch hier bleibt die Tabelle unverändert.

Kante C-B und Kante C-F
Knoten C hat ermittelte Gesamtkosten von 8. Addiert mit den jeweiligen Kantengewichten ergeben sich für Knoten B $8 + 5 = 13$, für Knoten F $8 + (-2) = 6$. Keine Verbesserung, demnach auch keine Änderung.

Kante D-A und Kante D-E
Knoten D hat Gesamtkosten von 3. Über die jeweiligen Kanten hat Knoten A ein Gesamtgewicht von $3 + 4 = 7$ und Knoten E ein Gesamtgewicht von $3 + 3 = 6$. Knoten A hat bereits ein Gesamtgewicht von 0, ist demnach teurer und wird nicht aktualisiert und Knoten E hat ebenfalls ein bereits gleichermaßen eingetragenes Gesamtgewicht von 6 über Knoten D.

Kante E-B und Kante E-D und E-F
Kante E mit Gesamtkosten von 6, addiert mit dem Kantengewicht über Kante(E, B) von -4 ergeben $6 + (-3) = 3$ für den Knoten B. Keine Verbesserung. Für Knoten D werden über die Kante(E, D) $6 + 3 = 9$ ermittelt. Ebenfalls keine Verbesserung. Knoten F hat Gesamtkosten von $6 + 2 = 8$ über die Kante(E, F). Keine Aktualisierung.

Kante F-C
Kante F mit Gesamtkosten von 6, addiert mit dem Kantengewicht über Kante(F, C) von 4 ergeben 10 für den Knoten C. Keine Verbesserung.

Die Tabelle ändert sich zu:

Knoten	Vorgänger	Gesamtkosten vom Start
A	-	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	6

Tabelle 5.3: Bellman-Ford: Exemplarisch - Iteration 2

Am Ende der Iteration wurden alle Kanten genau einmal betrachtet und eine Route mit Gesamtkosten von 6 nach F ermittelt.
Der Backtrace von E nach A ergibt: $F \leftarrow C \leftarrow B \leftarrow E \leftarrow D \leftarrow A$ mit Kosten von $(-2) + 5 + (-3) + 3 + 3 + 0 = 6$.

Da sich nach der zweiten Iteration eine weitere Verbesserung eingestellt hat, wird eine dritte Iteration durchgeführt um zu prüfen, ob in den folgenden Iterationen weitere Verbesserungen gefunden werden können.

5.2.3 Iteration 3

Iteration 3 wird nicht vollständig vorgeführt.

Kante A-B und Kante A-D
Kante A hat jeweils aufgrund ihres Status als Startknoten Gesamtkosten von 0. Durch Addition des jeweiligen Kantengewichts ergeben sich für Knoten B Kosten von 4 und für Knoten D Kosten von 3. Es ändert sich nichts.

Kante B-C und Kante B-E
Für Kante B werden jeweils Kosten von 3 erhoben und für Knoten C mit dem Kantengewicht für Kante(B, C) von 5 addiert und für Knoten E mit einem Kantengewicht für Kante(B, E) von 4 addiert. Knoten C hat demnach Kosten von 8 und Knoten E Kosten von 7. Auch hier bleibt die Tabelle unverändert.

Kante C-B und Kante C-F
Knoten C hat ermittelte Gesamtkosten von 8. Addiert mit den jeweiligen Kantengewichten ergeben sich für Knoten B $8 + 5 = 13$, für Knoten F $8 + (-2) = 6$. Keine Verbesserung, demnach auch keine Änderung.

Kante D-A und Kante D-E
Knoten D hat Gesamtkosten von 3. Über die jeweiligen Kanten hat Knoten A ein Gesamtgewicht von $3 + 4 = 7$ und Knoten E ein Gesamtgewicht von $3 + 3 = 6$. Knoten A hat bereits ein Gesamtgewicht von 0, ist demnach teurer und wird nicht aktualisiert und Knoten E hat ebenfalls ein bereits gleichermaßen eingetragenes Gesamtgewicht von 6 über Knoten D.

Kante E-B und Kante E-D und E-F
Kante E mit Gesamtkosten von 6, addiert mit dem Kantengewicht über Kante(E, B) von -4 ergeben $6 + (-3) = 3$ für den Knoten B. Keine Verbesserung. Für Knoten D werden über die Kante(E, D) $6 + 3 = 9$ ermittelt. Ebenfalls keine Verbesserung. Knoten F hat Gesamtkosten von $6 + 2 = 8$ über die Kante(E, F). Keine Aktualisierung.

Kante F-C
Kante F mit Gesamtkosten von 6, addiert mit dem Kantengewicht über Kante(F, C) von 4 ergeben 10 für den Knoten C. Keine Verbesserung.

Die Tabelle bleibt:

Knoten	Vorgänger	Gesamtkosten vom Start
A	-	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	6

Tabelle 5.4: Bellman-Ford: Exemplarisch - Iteration 3

Am Ende der Iteration wurden alle Kanten genau einmal betrachtet und eine Route mit Gesamtkosten von 6 nach F ermittelt.
Der Backtrace von E nach A ergibt: $F \leftarrow C \leftarrow B \leftarrow E \leftarrow D \leftarrow A$ mit Kosten von $(-2) + 5 + (-3) + 3 + 3 + 0 = 6$.

Nach der dritten Iteration wurden keine weiteren Verbesserungen festgestellt. Folglich kann an dieser Stelle die Untersuchung beendet werden.

Es ist sinnvoll, eine Möglichkeit zu implementieren, der die Ergebnisse nach den Iterationen vergleicht, um möglicherweise Laufzeit zu sparen. Bricht der Algorithmus nicht ab, würden bei diesem konkreten Beispiel drei weitere Durchläufe stattfinden, die ebenfalls keine Verbesserungen finden werden.

5.2.4 Backtrace

Aus der Tabelle 5.2.3 kann der Graph abgelesen werden. Der kürzeste Weg ergibt sich per Backtrace somit aus:

$$\mathbf{A} \rightarrow \mathbf{D} \rightarrow \mathbf{E} \rightarrow \mathbf{B} \rightarrow \mathbf{C} \rightarrow \mathbf{F}$$

.

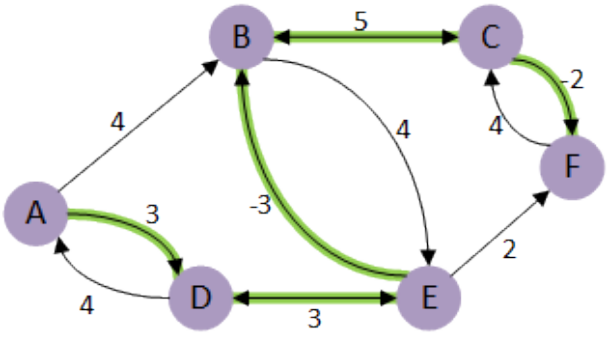

Abbildung 5.2: Fiktive Karte mit ermittelter Route

Ein Vorteil der zuvor erzeugten Tabelle und der Betrachtung aller Knoten im Graph, es kann dynamisch von jedem Knoten zu jedem beliebigen Knoten eine Route mittels Backtrace gefunden werden. Das gilt jedoch allgemein, da der Graph erst endet, wenn im gesamten Graph keine weiteren Kostenreduktionen mehr gefunden werden.

5.3 Negative Zyklen

Negativer Zyklus bedeutet, dass von einem Knoten aus, derselbe Knoten über einen Pfad mit negativen Gesamtkosten wieder erreicht wird.

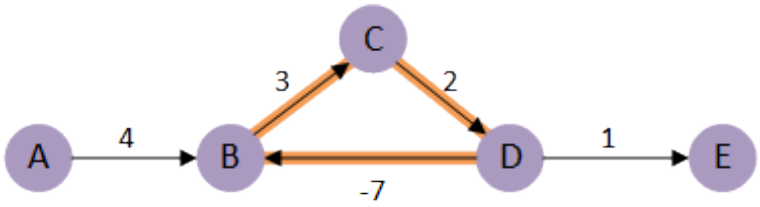

Abbildung 5.3: Beispiel negativer Zyklus

Im Beispiel-Graph Abbildung 5.3 hat der zyklische Pfad $B \rightarrow C \rightarrow D \rightarrow B$ Gesamtkosten von $3 + 2 + (-7) = -2$.

Problematik
Ein Zyklus kann beliebig oft gekreuzt werden. Das bedeutet, dass mit jeder Iteration die Gesamtkosten auf allen durchlaufenen Knoten weiter reduziert werden.

Bei erster Betrachtung des Beispiels aus Abbildung 5.3 ergibt sich eine logische Route von Knoten A nach Knoten E von $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ mit Gesamtkosten von $4 + 3 + 2 + 1 = 10$.

Durch genauere Betrachtung können die Gesamtkosten der Route von Knoten A nach Knoten E durch einen ”Umweg” reduziert werden, indem der Pfad über
$A \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ gewählt wird. Die Gesamtkosten betragen: $4 + 3 + 2 + (-7) + 3 + 2 + 1 = 8$. Einmaliges Durchlaufen ergibt eine Reduktion von 2.

Ein weiterer Zyklus reduziert die Kosten um weitere 2. Fünf maliges Durchlaufen erzeugen also Kosten von 0. Folglich endet der Zyklus niemals und findet keine kürzeste Route für den gegebenen Graph.

Identifikation
Im Abschnitt 5.1, der Beschreibung des Bellman-Ford-Algorithmus, wird erläutert, dass ein zu untersuchender Graph maximal $n - 1$ Iterationen durchlaufen muss, um einen kürzesten Weg zu finden. Führt der Algorithmus eine weitere Iteration durch, in dem geprüft wird, ob sich

an einem beliebigen Knoten die Gesamtkosten weiter reduzieren, handelt es sich um einen negativen Zyklus.

5.4 Informelle Beschreibung

Vorbereitung

1. Tabelle erstellen mit allen Knoten, den entsprechenden Vorgängerknoten und den Gesamtkosten vom Start
2. Gesamtkosten des Startknotens auf 0 setzen und für alle anderen auf unendlich

Abarbeitung
Führe folgende Anweisungen $n - 1$ mal durch, dabei ist n die Anzahl der Knoten

1. Für jede Kante des Graphen:
   a) Summe berechnen aus Gesamtkosten zum Startknoten und Kantengewicht
   b) Wenn die Summe niedriger als aktuelle Gesamtkosten des Kanten-Endknotens, setze Vorgänger des Endknotens auf Startknoten und die Gesamtkosten des Endknotens auf die zuvor berechnete Summe
2. Wenn in einer Iteration keine Änderungen vorgenommen wurden, Abarbeitung beenden

Wenn der Algorithmus nicht beendet wurde, prüfe auf negative Zyklen

1. Für jede Kante des Graphen:
   a) Summe berechnen aus Gesamtkosten zum Startknoten und Kantengewicht
   b) Wenn die Summe niedriger als aktuelle Gesamtkosten des Kanten-Endknotens, beende den Algorithmus wegen eines entdeckten negativen Zyklus

5.5 Pseudocode

G bezeichnet einen gewichteten Graphen mit der Kantenmenge E und einer Knotenmenge V. Das `Gewicht` beschreibt das Gewicht einer Kante zwischen zwei Knoten $[u, v]$. Der Startknoten s, von dem die kürzesten Wege zu allen anderen Knoten aus berechnet werden und die Anzahl aller Knoten n in V.

Nach Beendigung der Ausführung des Bellman-Ford-Algorithmus kann man dem Ergebnis entnehmen, ob G ein Zyklus negativer Länge ist. Ist der Graph positiv, enthält `Distanz` für jeden Knoten seinen Abstand zum Startknoten s.

Durch `Vorgänger` wird ein Baum aufgespannt definiert, der die vom Startknoten s ausgehenden Wege in Form eines Baums speichert.

Ausgehend von einem beliebigen Knoten in der Knotenmenge V, kann man den kürzesten Weg rückwärts ermitteln, bis man durch die gegebenen `Vorgänger` den Startknoten s erreicht hat.

```
Funktion Bellman-Ford:
    fuer jeden Knoten v in V
          distanz[v] := unendlich
          vorgaenger[v] := null
    distanz[Startknoten] := 0

    wiederhole fuer die Menge n aller Knoten in V weniger 1 mal
        fuer jede Kante [u,v] in E
            wenn (distanz[u] + gewicht[u,v]) < distanz[v] dann
                distanz[v] := distanz[u] + gewicht[u,v]
                vorgaenger[v] := u

    return distanz, vorgaenger
```

Listing 5.1: Pseudocode Bellman-Ford: Bellman-Ford-Algorithmus

5.6 JAVA-Implementierung

Die beispielhafte Implementierung des Bellman-Ford-Algorithmus in JAVA folgt der informellen Beschreibung aus Kapitel 5.4.

5.6.1 Node.java

Für die Implementierung des Bellman-Ford-Algorithmus gibt es eine `Node`-Klasse, die von der abstrakten, generellen `Node`-Klasse erbt.

```
public class Node extends GeneralNode<Node> {
    public Node(String name) {
        super(name);
    }
}
```

Listing 5.2: Bellman-Ford: Node.java

5.6.2 BellmannFord.java

Die wichtigste Methode ist die Implementierung der `computeShortestPath()`-Methode. Sie bildet den Kern der Wegberechnung beim Bellman-Ford-Algorithmus.

Durch initiales Setzen der Distanz für jeden Knoten im Graph und der Zuweisung der Gesamtkosten von 0 für den Startknoten, wird die Abarbeitung vorbereitet.

Für jede Kante innerhalb des Graphen wird die Summe der Gesamtkosten zum Startpunkt und dem Kantengewicht berechnet und überprüft, ob die aktuellen Gesamtkosten höher sind als

die berechnete Summe. Wenn dem so ist, wird der Vorgänger des Endknotens als Startknoten gesetzt und die Gesamtkosten des Endknotens als die zuvor bestimmte Summe.

Nachdem alle Iterationen abgeschlossen sind, wird auf negative Zyklen geprüft.

Mit der `showPath()`-Methode kann das Ergebnis der Berechnung ausgegeben werden.

```
1  public class BellmanFord {
2
3      private Map<Node, Node> predecessors = new HashMap<>();
4      private List<Node> shortestPath = new ArrayList<>();
5
6      public void computeShortestPath(
7              GeneralGraph<Node> graph,
8              Node start,
9              Node target) {
10         Map<Node, Integer> distances = new HashMap<>();
11
12         for (Node node : graph.getAllNodes()) {
13             distances.put(node, Integer.MAX_VALUE);
14         }
15         distances.put(start, 0);
16
17         List<GeneralEdge<Node>> edges = graph.getAllEdges();
18
19         for (int i = 1; i < graph.getAllNodes().size(); i++) {
20             for (GeneralEdge<Node> edge : edges) {
21                 Node u = getSourceNode(edge, graph);
22                 Node v = edge.getDestination();
23                 if (distances.get(u) != Integer.MAX_VALUE
24                     && distances.get(u)
25                     + edge.getWeight() < distances.get(v)) {
26                     distances.put(v, distances.get(u) + edge.getWeight());
27                     predecessors.put(v, u);
28                 }
29             }
30         }
31
32         // Negative cycle detection
33         for (GeneralEdge<Node> edge : edges) {
34             Node u = getSourceNode(edge, graph);
35             Node v = edge.getDestination();
36             if (distances.get(u) != Integer.MAX_VALUE
37             && distances.get(u)
38             + edge.getWeight() < distances.get(v)) {
```

```
39                  System.out.println("Graph␣contains␣a␣negative-weight␣cycle");
40                  return;
41              }
42          }
43
44          printShortestPath(target);
45      }
46
47      private Node getSourceNode(
48              GeneralEdge<Node> edge,
49              GeneralGraph<Node> graph) {
50          for (Node node : graph.getAllNodes()) {
51              for (GeneralEdge<Node> e : graph.getEdgesByNode(node)) {
52                  if (e == edge) {
53                      return node;
54                  }
55              }
56          }
57
58          return null;
59      }
60
61      private void printShortestPath(Node currentNode) {
62          while (currentNode != null) {
63              shortestPath.add(currentNode);
64              currentNode = predecessors.get(currentNode);
65          }
66
67          Collections.reverse(shortestPath);
68      }
69
70      public void showPath() {
71          System.out.println(shortestPath);
72      }
73  }
```

Listing 5.3: Bellman-Ford: BellmanFord.java

5.6.3 Beispielimplementierung

Die Beispielimplementierung ist exemplarisch als Unittest auf Grundlage des Beispielgraphes aus Abbildung 5.2 in das Projekt integriert.

Zuerst wird eine Instanz der Klasse `GeneralGraph` instanziiert und alle Knoten im Graph werden als Instanz einer `Node` eingefügt. Danach wird jeder Knoten über die `addNewNode()`-

Methode zum Graphen hinzugefügt.

Nachdem alle Knoten des Graphen implementiert sind, wird über die `addEdge()`-Methode jedes `Node`-Objekt mit einer gerichteten Kante dem Graphen mit ihrem jeweiligen Endknoten und dem Kantengewicht übergeben.

Mit einer Instanz der `BellmanFord`-Klasse werden alle Wege vom gegebenen Zielknoten mit der `computeShortestPath()`-Methode zu allen vorhandenen Knoten berechnet, sofern vorhanden.

Angezeigt kann die berechnetet Route über einen Zugriff auf die `showPath()`-Methode.

```
1  public class BellmanFordTestWithSampleGeneralGraph {
2
3      @Test
4      public void setGraph_StartNodeA_findWay() {
5          GeneralGraph<Node> graph = new GeneralGraph<>();
6          Node nodeA = new Node("A");
7          Node nodeB = new Node("B");
8          Node nodeC = new Node("C");
9          Node nodeD = new Node("D");
10         Node nodeE = new Node("E");
11         Node nodeF = new Node("F");
12
13         graph.addNewNode(nodeA);
14         graph.addNewNode(nodeB);
15         graph.addNewNode(nodeC);
16         graph.addNewNode(nodeD);
17         graph.addNewNode(nodeE);
18         graph.addNewNode(nodeF);
19
20         graph.addEdge(nodeA, nodeB, 4);
21         graph.addEdge(nodeA, nodeD, 3);
22         graph.addEdge(nodeB, nodeC, 5);
23         graph.addEdge(nodeB, nodeE, 4);
24         graph.addEdge(nodeC, nodeB, 5);
25         graph.addEdge(nodeC, nodeF, -2);
26         graph.addEdge(nodeD, nodeA, 4);
27         graph.addEdge(nodeD, nodeE, 3);
28         graph.addEdge(nodeE, nodeB, -3);
29         graph.addEdge(nodeE, nodeD, 3);
30         graph.addEdge(nodeE, nodeF, 2);
31         graph.addEdge(nodeF, nodeC, 4);
32
33         BellmanFord bellmanFord = new BellmanFord();
34         bellmanFord.computeShortestPath(graph, nodeA, nodeF);
```

```
35          bellmanFord.showPath();
36      }
37  }
```

Listing 5.4: Bellman-Ford: Beispiel

Die Ausgabe aus dem Beispiel ist:

```
[A, D, E, B, C, F]
```

Listing 5.5: Bellman-Ford: Ausgabe

5.7 Vorteile

Vorteile des Bellman-Ford-Algorithmus sind im Gegensatz zum Dijkstra-Algorithmus und dem A-Stern-Algorithmus, dass er mit negativen Kantengewichten umgehen kann. Das ist vor allem nützlich, wenn Gewinne oder finanzielle Kosten bei z.B. Transportrouten berücksichtigt werden sollen.

5.8 Nachteile

Ein großer Nachteil des Bellman-Ford-Algorithmus ist, dass er sehr langsam ist. Er benötigt immer das Quadrat aus der Anzahl der Knoten multipliziert mit der Anzahl der Kanten in einem Graph. Negative Kanten müssen nach der Betrachtung immer ausgeschlossen werden, um unendliche Iterationen zu vermeiden. Sollte ein gegebener Graph ausschließlich positive Kantengewichte enthalten, ist es sinnvoller einen anderen Algorithmus zu nutzen, der deutlich effizienter mit den zu suchenden Pfaden umgeht.

5.9 Zeitkomplexität

Die Laufzeit des Bellman-Ford-Algorithmus ist in $O(n \cdot m)$, wobei n die Anzahl der Knoten und m die Anzahl der Kanten im Graphen sind.
Will man die kürzesten Wege von jedem Knoten zu jedem anderen Knoten finden, so muss man den Algorithmus für jeden Startknoten einmal anwenden, insgesamt also n-mal. Die Laufzeitkomplexität dafür beträgt folglich $O(n^2 \cdot m)$.

6 Floyd-Warshall-Algorithmus

Der Algorithmus von Floyd und Warshall, oder auch Floyd-Warshall-Algorithmus, ist nach Robert Floyd und Stephen Warshall benannt. In Floyds Version findet der Algorithmus die kürzesten Pfade zwischen allen Paaren von Knoten eines Graphen und berechnet deren Länge (APSP, all-pairs shortest path, siehe Kapitel 2.7.3) und in Warshalls Version findet der Algorithmus die transitive Hülle eines Graphen. [6] [8]

6.1 Beschreibung

Der Floyd-Warshall-Algorithmus basiert auf dem Prinzip der dynamischen Programmierung, das heißt das zum algorithmischen Lösen eines Optimierungsproblems eine Aufteilung in Teilprobleme stattfindet und Zwischenresultate systematisch gespeichert werden.

Außerhalb von Graphen ist der Algorithmus von Floyd und Warshall besonders relevant für:

1. **Verkehrsplanungen**: In der Verkehrsplanung können kürzeste Wege zwischen verschiedenen Standorten berechnet werden.
2. **Flugrouten**: Fluggesellschaften verwenden den Algorithmus zum Ermitteln von kürzesten Flugverbindungen zwischen Städten. Weiterhin werden Distanzen und Zwischenstopps berücksichtigt.
3. **Netzwerkverbindungen**: Der Algorithmus kann verwendet werden, um die kürzesten Verbindungen zwischen Computern in einem Netzwerk zu finden. Dies ist wichtig für die Effizienz und Zuverlässigkeit von Netzwerken.
4. **Genetik und Bioinformatik**: In der Bioinformatik kann der Floyd-Warshall-Algorithmus verwendet werden, um ähnliche Sequenzen in DNA oder Proteinsequenzen zu finden. Dies hilft bei der Identifizierung von Genen und Proteinen.
5. **Spieleentwicklungen**: In Videospielen kann der Algorithmus verwendet werden, um Pfade für Charaktere oder KI-Gegner zu berechnen. Dies ermöglicht realistische Bewegungen und Interaktionen im Spiel.
6. **Zeitplanungen**: Der Algorithmus kann verwendet werden, um optimale Zeitpläne für Aufgaben oder Projekte zu erstellen. Dies ist nützlich in der Projektmanagement- und Produktionsplanung.

Der Algorithmus ist in zwei Teile untergliedert. Floyds Teil sucht die kürzesten Wege zwischen allen Paaren von Start- und Zielknoten. Der Teil von Warshall berechnet die ”transitive Hülle”. Die transitive Hülle erweitert einen Graphen um Kanten zwischen allen indirekt verbundenen Knotenpaaren. Wenn ein Graph zwei Kanten hat – eine von A nach B und eine von B nach C –

dann vervollständigt die transitive Hülle den Graphen um die Kante von A nach C (da ein Weg von A nach C über B existiert).

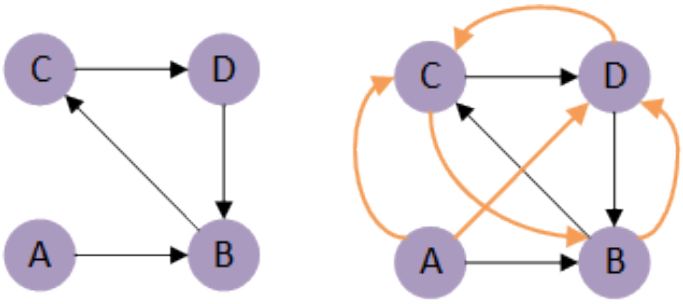

Abbildung 6.1: Beispiel-Graph mit transitiver Hülle

Nachfolgende Abbildung zeigt ein Beispiel mit vier Knoten. Links ist der Ausgangsgraph und rechts dessen transitive Hülle. Die orangefarbenen Pfeile stellen die indirekten Verbindungen dar.

Existiert ein kürzester Weg zwischen zwei Knotenpaaren, dann gehört dieses Knotenpaar auch in die transitive Hülle und umgekehrt. Um alle Verbindungen zu finden wird n mal (n = Anzahl der Knoten) über alle Kanten iteriert.

6.2 Beispiel

Eine beispielhafte Abarbeitung des Floyd-Warshall-Algorithmus an einem fiktiven Graph mit fünf Knoten und verschiedenen Kanten:

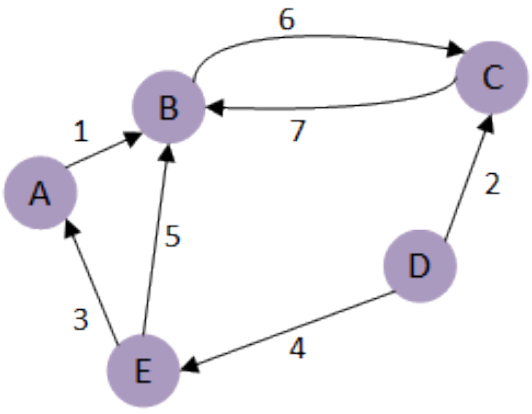

Abbildung 6.2: Fiktive Karte als gewichteter Graph

Vorbereitung
Zur Vorbereitung wird eine $n \times n$-Matrix erstellt. n steht für die Anzahl der Knoten im Graph. In diese Matrix werden für jedes Knotenpaar (i, j) ihr jeweiliges Gewicht von i nach j eingetragen, sofern dieses existiert. Unbekannte Verbindungen werden mit ∞ gekennzeichnet und die Entfernung von Knoten zu sich selbst mit 0.

	A	**B**	**C**	**D**	**E**
A	0	1	∞	∞	∞
B	∞	0	6	∞	∞
C	∞	7	0	∞	∞
D	∞	∞	2	0	4
E	3	5	∞	∞	0

Tabelle 6.1: Floyd-Warshall: Exemplarisch - Vorbereitung

Die Tabelle bzw. Matrix gibt Aufschluss über die Verbindungen. Die Zeilen beziehen sich auf den Startknoten der Kante und die Spalte auf den Endknoten der Kante.
Am Beispiel: *"Das Kantengewicht der Kante von Knoten E nach Knoten B beträgt 5"*

6.2.1 Iteration 1 - Indirekte Verbindungen über Knoten A

Im ersten Durchlauf werden für alle Knotenpaare (i, j) die eingetragenen Kosten des direkten Weges mit den Kosten des indirekten Weges von i nach j über Knoten A verglichen. Das bedeutet, dass die Kosten von i nach A, addiert mit den Kosten von A nach j erhoben werden, sofern ein solcher Weg existiert. Sind die Kosten des indirekten Weges geringer, als die des direkten Weges, so werden diese in der Matrix ersetzt.

Knotenpaare werden übersprungen, wenn $i = j$, $i = A$ oder $j = A$. Die Entfernung zu sich selbst ist immer 0. Wenn der Start- oder Zielknoten bereits A ist, gibt es keinen direkten Weg über A.

Den Anfang bildet das Knotenpaar (B, C). Die aktuellen Kosten betragen laut Matrix 6. Von B nach A (Spalte A bei Zeile B ist ∞) ist kein Weg bekannt. In diesem Schritt gibt es folglich keinen kürzeren Weg über Knoten A. Wege über (B, D) und (B, E) bieten ebenfalls keine kürzere Alternative.

Das Betrachten der Zeilen C und D liefern das gleiche Ergebnis: Es gibt keinen kürzeren Weg für (C, B), (C, D), (C, E), (D, B), (D, C), (D, E). In beiden Zeilen stehen für Spalte A aktuell jeweils Kosten von ∞.

Das Knotenpaar (E, B) hingegen hat in der Spalte A bei Zeile E einen Eintrag. Die Kosten von E nach A betragen 3. Die Kosten von A nach B betragen 1. Die Summe aus beiden Teilen ergibt 3 + 1 = 4. Das Resultat ist, dass die Kosten von E nach B über Knoten A geringer sind, als der direkte Weg. Ein kürzer Weg wurde gefunden.

Die Tabelle wird aktualisiert:

	A	B	C	D	E
A	0	1	∞	∞	∞
B	∞	0	6	∞	∞
C	∞	7	0	∞	∞
D	∞	∞	2	0	4
E	3	4	∞	∞	0

Tabelle 6.2: Floyd-Warshall: Exemplarisch - Iteration 1 - Erste Aktualisierung

Die nächsten Knotenpaare die betrachtet werden, sind das Paar (E,C) und (E, D). Von A nach C und von A nach D ist derzeit kein Weg bekannt, also kein indirekter Weg $E \rightarrow A \rightarrow C$ und kein indirekter Weg $E \rightarrow A \rightarrow D$.

Alle Knotenpaare wurden in diesem Schritt betrachtet.

6.2.2 Iteration 2 - Indirekte Verbindungen über Knoten B

In der zweiten Iteration werden wie zuvor, alle Knotenpaare (i, j) der bereits eingetragenen Kosten mit den Kosten von i nach j über B verglichen. Die derzeitigen Kosten werden aus der Matrix abgelesen.

Das erste Knotenpaar das betrachtet wird, ist das Paar (A, C). Bislang ist kein Weg von A über B und von B nach C bekannt, da in der Spalte C ∞ eingetragen ist. Die Kosten von A nach B betragen 1. Die Kosten von B nach C betragen 6. Die Summe ist folglich 7 und $7 < \infty$. Die Tabelle wird wieder aktualisiert:

	A	B	C	D	E
A	0	1	7	∞	∞
B	∞	0	6	∞	∞
C	∞	7	0	∞	∞
D	∞	∞	2	0	4
E	3	4	∞	∞	0

Tabelle 6.3: Floyd-Warshall: Exemplarisch - Iteration 2 - Erste Aktualisierung

Für das Knotenpaar (A, D) und (A, E) sind ebenfalls keine Wege bekannt. Ein Weg von A nach B ist bekannt, allerdings existiert weder ein bekannter Weg von B nach D, noch von B nach E. Die Kosten bleiben ∞.

Knotenpaare (C, A), (C, D) und (C, E): Ein Weg von Knoten C nach Knoten B ist mit Kosten von 7 bekannt, allerdings sind keine Wege von Knoten B nach Knoten A, Knoten D oder E

bekannt. Keine Veränderung der Matrix.

Knotenpaare (D, A), (D, C) und (D, E): Kein Weg von D nach B ist bekannt. Weitere Untersuchungen sind hinfällig.

Untersuchung der Knotenpaare (E, A), (E, C) und (E, D): Ein Weg von Knoten E nach Knoten B ist mit Kosten von 4 (In Iteration 1 ermittelt) bekannt. Es gibt keinen Weg von Knoten B nach Knoten A. Allerdings gibt es einen Weg von Knoten B nach Knoten C. Die Kosten betragen laut Tabelle 6. Die Gesamtkosten für einen indirekten Weg $E \rightarrow B \rightarrow C$ betragen 10. In Zeile E und Spalte C wird der Wert 10 eingetragen. Ein Überprüfung des letzten Knotenpaares (E, D) zeigt, dass kein Weg von Knoten B nach Knoten D führt.

	A	**B**	**C**	**D**	**E**
A	0	1	7	∞	∞
B	∞	0	6	∞	∞
C	∞	7	0	∞	∞
D	∞	∞	2	0	4
E	3	4	10	∞	0

Tabelle 6.4: Floyd-Warshall: Exemplarisch - Iteration 2 - Zweite Aktualisierung

Iteration 2 ist beendet. Die niedrigsten Kosten für Knotenpaare dessen Wege über die Knoten A und B führen sind nun bekannt.

6.2.3 Iteration 3 - Indirekte Verbindungen über Knoten C

Die dritte Iteration folgt dem gleichen Schema wie zuvor, jedoch für mögliche indirekte Wege über Knoten C. Die aktuelle Tabelle befindet sich für diese Iteration am Ende.

Die Knotenpaare von A werden betrachtet: (A, B), (A, D), (A, E). Von Knoten A nach Knoten C ist ein Weg mit Kosten von 7 bekannt. Außerdem ist ein Weg von Knoten C nach Knoten B mit Kosten von 7 bekannt. Ein indirekter Weg $A \rightarrow C \rightarrow B$ hat Gesamtkosten von $7 + 7 = 14$. Der Weg ist allerdings teurer bzw. länger als der bereits erfasste Weg von 1. Es findet keine Aktualisierung statt.
Für die beiden anderen Knotenpaare existiert kein indirekter Weg über Knoten C, da weder ein Weg von Knoten C nach Knoten D, noch von Knoten C nach Knoten E führt.

Knotenpaare (B, A), (B, D), (B, E): Die Kosten von B nach C betragen 6, von C existiert allerdings weder ein Pfad zu A, zu D oder zu E. Somit gibt es in dieser Iteration keinen der Wege $B \rightarrow C \rightarrow A$, $B \rightarrow C \rightarrow D$ und $B \rightarrow C \rightarrow E$.

Knotenpaare (D, A), (D, B), (D, E): Es existiert ein Pfad von D nach C, allerdings keiner von C nach A, somit auch keiner von D via C nach A. Es existiert aber ein Pfad von C nach B, also auch ein Weg $D \rightarrow C \rightarrow B$. Die derzeitigen Kosten sind unendlich, werden also aktualisiert zu

2 + 7 = 9.
Es ist kein Weg von Zwischenknoten C nach E bekannt, folglich auch kein Weg von D via C nach E.

Knotenpaare (E, A), (E, B), (E, D): Ein Weg von E nach C ist bekannt. Da derzeit keine Wege von C nach A oder von C nach D bekannt sind, werden diese nicht weiter betrachtet. Ein Weg von C nach B ist ebenfalls bekannt. Die Kosten betragen von Knoten E nach Knoten C 10 und die Kosten von Knoten C nach Knoten B 4. Die Gesamtkosten für $E \rightarrow C \rightarrow B$ ergeben 14. Da $14 > 4$, werden die Kosten in der Matrix nicht aktualisiert.

	A	**B**	**C**	**D**	**E**
A	0	1	7	∞	∞
B	∞	0	6	∞	∞
C	∞	7	0	∞	∞
D	∞	9	2	0	4
E	3	4	10	∞	0

Tabelle 6.5: Floyd-Warshall: Exemplarisch - Iteration 3

Iteration 3 ist beendet. Die niedrigsten Kosten für alle Knotenpaare sind bekannt, wenn auch indirekte Wege über Knoten C – und damit auch über A und B – zugelassen werden.

6.2.4 Iteration 4 - Indirekte Verbindungen über Knoten D

Iteration 4 kann ausgelassen werden, da es keinen Weg von einem beliebigen Knoten nach Knoten D gibt.

6.2.5 Iteration 5 - Indirekte Verbindungen über Knoten E

Das Betrachten der Matrix zeigt, dass kein Weg von den Knoten A, B, C nach Zwischenknoten E führt. Diese Paare müssen nicht untersucht werden.

Knotenpaare (D, A), (D, B) und (D, C): Die Kosten von Knoten D nach Knoten E betragen 4.
Die Kosten von Zwischenknoten E nach Knoten A betragen 3. Ein Weg $D \rightarrow E \rightarrow A$ kostet also 4 + 3 = 7. Matrix wird aktualisiert.
Von Zwischenknoten E nach Knoten B werden Kosten in Höhe von 4 erhoben. Addiert mit den Kosten von D nach E, entstehen Gesamtkosten von 8. Die nun ermittelten Kosten sind geringer als die Iteration 3 ermittelten Kosten für das Knotenpaar (D, B).
Ein Weg $D \rightarrow E \rightarrow C$ erzeugt Kosten von 4 + 10 = 14, was teurer ist als 2. Für diesen Pfad findet keine Veränderung statt.

Die finale Tabelle zeigt folgende Ergebnisse:

	A	B	C	D	E
A	0	1	7	∞	∞
B	∞	0	6	∞	∞
C	∞	7	0	∞	∞
D	7	8	2	0	4
E	3	4	10	∞	0

Tabelle 6.6: Floyd-Warshall: Exemplarisch - Iteration 5

Die fünfte und letzte Iteration ist beendet und das Ziel des Algorithmus ist damit erreicht. Die niedrigsten Kosten für alle Knotenpaare sind bekannt, sofern auch indirekte Pfade über alle Knoten zugelassen werden. Eine sorgfältige Betrachtung der gefundenen, indirekten Pfade gibt Aufschluss über tatsächliche ”Abkürzungen”:

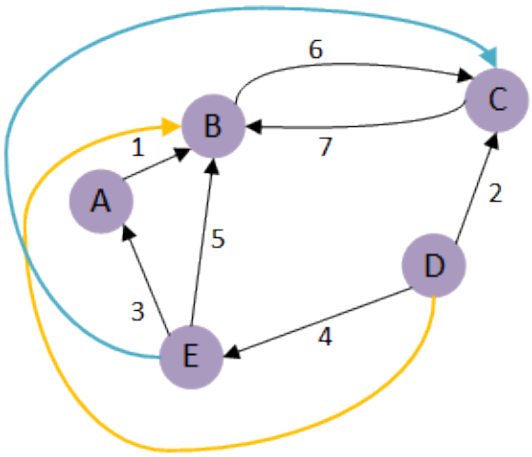

Abbildung 6.3: Teilergebnis Graph - Floyd-Warshall

Ein indirekter Weg aus Iteration 2 (blau) zeigt, dass der tatsächliche Weg nicht etwa $E \rightarrow B \rightarrow C$ ist, sondern $E \rightarrow A \rightarrow B \rightarrow C$ und ein Weg aus Iteration 5 (orange) nicht $D \rightarrow E \rightarrow B$ sondern $D \rightarrow E \rightarrow A \rightarrow B$.

6.2.6 Bestimmung der kürzesten Pfade

In den Iterationen wurden lediglich die Kosten der kürzesten Pfade zwischen zwei Knoten berechnet. Es gibt eine Bestimmung der Pfade selbst, also über welche Zwischenknoten eine Route führen wird.

Durch das Erweitern des Algorithmus ist eine Bestimmung einer Route möglich. Eine sogenannte Nachfolgermatrix ($n \times n$ - Matrix) wird angelegt. In dieser Matrix wird für jedes

Knotenpaar (i, j) initial der jeweilige Endknoten j eingetragen, sodass ein Weg von i nach j über den Nachfolger j selbst läuft. Sobald aber ein kürzerer Weg über einen Zwischenknoten k gefunden wird, wird in der Matrix an der Position (i, j) der Wert der Matrix an der Position (i, k) eingetragen. Das bedeutet, dass der Weg von i nach j nun über denselben Nachfolger führt wie der Weg von i nach k. Der Nachfolger kann k selbst sein, aber auch ein weiterer Zwischenknoten auf dem kürzesten Weg zu k.

Beispiel:

	A	**B**	**C**	**D**	**E**
A	-	B	-	-	-
B	-	-	C	-	-
C	-	B	-	-	-
D	-	-	C	-	E
E	A	B	-	-	-

Tabelle 6.7: Floyd-Warshall: Exemplarisch - Nachfolgermatrix Initial

Iteration 1:

	A	**B**	**C**	**D**	**E**
A	-	B	-	-	-
B	-	-	C	-	-
C	-	B	-	-	-
D	-	-	C	-	E
E	A	A	-	-	-

Tabelle 6.8: Floyd-Warshall: Exemplarisch - Nachfolgermatrix Iteration 1

Iteration 2:

	A	**B**	**C**	**D**	**E**
A	-	B	B	-	-
B	-	-	C	-	-
C	-	B	-	-	-
D	-	-	C	-	E
E	A	A	A	-	-

Tabelle 6.9: Floyd-Warshall: Exemplarisch - Nachfolgermatrix Iteration 2

Für das Knotenpaar (E, C) wird der Knoten A eingetragen statt der Knoten B, weil der tatsächliche Weg über $E \longrightarrow A \longrightarrow B \longrightarrow C$ führt.

Iteration 3:

	A	**B**	**C**	**D**	**E**
A	-	B	B	-	-
B	-	-	C	-	-
C	-	B	-	-	-
D	-	C	C	-	E
E	A	A	A	-	-

Tabelle 6.10: Floyd-Warshall: Exemplarisch - Nachfolgermatrix Iteration 3

Iteration 4:

	A	**B**	**C**	**D**	**E**
A	-	B	B	-	-
B	-	-	C	-	-
C	-	B	-	-	-
D	-	C	C	-	E
E	A	A	B	-	-

Tabelle 6.11: Floyd-Warshall: Exemplarisch - Nachfolgermatrix Iteration 4

Iteration 5:

	A	**B**	**C**	**D**	**E**
A	-	B	B	-	-
B	-	-	C	-	-
C	-	B	-	-	-
D	E	E	C	-	E
E	A	A	A	-	-

Tabelle 6.12: Floyd-Warshall: Exemplarisch - Nachfolgermatrix Iteration 5

Der gesuchte Weg sei: *Startknoten D, Zielknoten B*

Aus der Matrix kann gelesen werden:

- Zeile D, Spalte B: Der direkte Nachfolger von D auf dem Weg zu B ist: E
- Zeile E, Spalte B: Der direkte Nachfolger von E auf dem Weg zu B ist: A
- Zeile A, Spalte B: Der direkte Nachfolger von A auf dem Weg zu B ist: B (Zielknoten erreicht)

Der vollständige kürzeste Pfad lautet also:

D → E → A → B

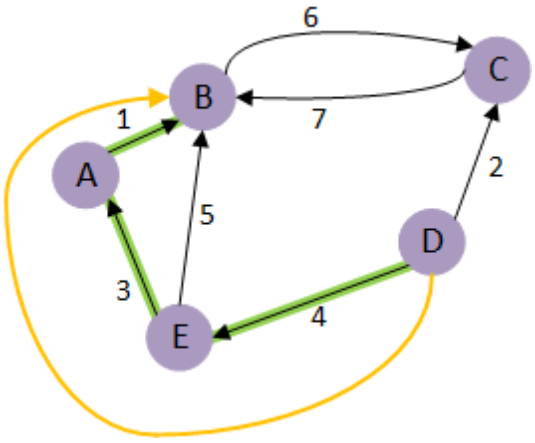

Abbildung 6.4: Karte mit Ergebnis

Das Ergebnis deckt sich mit dem gefundenen indirekten Weg aus Iteration 2 (auch in Abbildung 6.3 dargestellt).

6.3 Negative Zyklen

Negative Zyklen von beliebigen Knoten, führen dazu, dass die Kosten von diesem Knoten zu sich selbst negativ sind. Der Algorithmus von Floyd und Warshall erlaubt das Ablesen aller Kosten zu sich selbst über die Matrix-Diagonale.

	A	B	C	D	E
A	**0**	1	7	∞	∞
B	∞	**0**	6	∞	∞
C	∞	7	**0**	∞	∞
D	7	8	2	**0**	4
E	3	4	10	∞	**0**

Tabelle 6.13: Floyd-Warshall: Matrix-Diagonale

Befinden sich ausschließlich die Werte 0 in der Diagonale, existierten keine negativen Zyklen. Wäre im Gegensatz dazu, mindestens ein Wert negativ, wäre ein negativer Zyklus erkannt und der Algorithmus sollte mit einer Fehlermeldung beendet werden.

6.4 Informelle Beschreibung

Vorbereitung

1. Kostenmatrix der Größe $n \times n$ (n ist die Anzahl der Knoten) erstellen.
2. Für jedes Knotenpaar (i, j) die Kosten des direkten Pfades von i nach j vorbelegen, sofern existent, sonst unendlich.
3. Für die Diagonalen Nullen eintragen.

Vorbereitung Nachfolgermatrix

1. Nachfolgermatrix der Größe $n \times n$ erstellen.
2. Für jedes Knotenpaar (i, j) den Wert j eintragen.

Abarbeitung
Führe folgende Iteration n mal aus; k sei dabei der Schleifenzähler und stehe für den Zwischenknoten:

1. Für jedes Knotenpaar (i, j):
   a) Summe der Kosten des Weges $i \rightarrow k$ berechnen und die Kosten des Weges $k \rightarrow j$ berechnen.
   b) Ist die Summe kleiner als die Kosten des Weges $i \rightarrow j$, dann:
      i. trage die niedrigeren Kosten in Zeile i, Spalte j der Kostenmatrix ein
      ii. kopiere in der Nachfolgermatrix den Wert aus Feld (i, k) ins Feld (i, j).

Prüfe, ob auf der Diagonalen der Kostenmatrix eine negative Zahl vorkommt. Wenn ja, beende den Algorithmus mit Fehlerausgabe.

6.5 Pseudocode

```
Funktion Floyd-Warshall:
    fuer alle (i, j) in Knoten:
        wenn i == j, dann:
            kostenmatrix[i][j] := 0
            nachfolgermatrix[i][j] := 0
        oder wenn i oder j == unendlich, dann:
            kostenmatrix[i][j] := unendlich
            nachfolgermatrix[i][j] := unendlich
        sonst:
            kostenmatrix[i][j] := kantengewicht
            nachfolgermatrix[i][j] := j

    fuer alle k bis n-1:
        wenn distanz[i,k] oder distanz[k,j] nicht unendlich, dann:
            wenn distanz[i,k] + distanz[k,j] < kostenmatrix[i][j], dann
                kostenmatrix[i][j] := distanz[i,k] + distanz[k,j]
                nachfolgermatrix[i][j] := k

    fuer alle i, j in kostenmatrix[i][j]:
        wenn (i == j) < 0:
            Stop: Fehlerausgabe
```

Listing 6.1: Pseudocode Floyd-Warshall: Floyd-Warshall-Algorithmus

6.6 JAVA-Implementierung

Die beispielhafte Implementierung des Algorithmus von Floy und Warshall in JAVA folgt der informellen Beschreibung aus Kapitel 6.4.

6.6.1 Node.java

Für die Implementierung des Algorithmus von Floyd und Warshall gibt es eine `Node`-Klasse, die von der abstrakten, generellen `Node`-Klasse erbt.

```
public class Node extends GeneralNode<Node> {
    public Node(String name) {
        super(name);
    }
}
```

Listing 6.2: Floyd-Warshall: Node.java

6.6.2 FloydWarshall.java

Die `computeShortestPath()`-Methode in der `FloydWarshall`-Klasse ist der Kern der Integration vom Floyd-Warshall-Algorithmus.

Initial werden beide benötigten Matrizen und allen Distanzen mit ∞, sowie dem Startknoten mit `0` aufgebaut. Weiterhin wird in der Nachfolgermatrix jeder Nachfolger initial mit `null` gefüllt.

Im nächsten Schritt wird in der Distanzmatrix für jeden Knoten der bekannte Nachbar mit seinem Kantengewicht geschrieben und für die Nachfolgermatrix der entsprechende bekannte Nachfolgerknoten.

Ab hier beginnt die Abarbeitung nach dem in der informellen Beschreibung aus Abschnitt 6.4 erklärten Prinzip.

Nachdem alle Iterationen abgeschlossen sind, kann die Distanzmatrix mittels der `printDistanceMatrix()`-Methode ausgegeben werden, oder aber mit der `showPath()`-Methode kann das Ergebnis der Berechnung ausgegeben werden.

```
1  public class FloydWarshall {
2
3      private final Map<Node, Map<Node, Integer>> distances = new HashMap<>();
4      private final Map<Node, Map<Node, Node>> matrix = new HashMap<>();
5      private final List<Node> shortestPath = new ArrayList<>();
6
7      public void computeShortestPath(
8              GeneralGraph<Node> graph,
9              Node start,
10             Node target) {
11         Set<Node> nodes = graph.getAllNodes();
12         for (Node u : nodes) {
13             distances.put(u, new HashMap<>());
14             matrix.put(u, new HashMap<>());
15             for (Node v : nodes) {
16                 if (u.equals(v)) {
17                     distances.get(u).put(v, 0);
18                 } else {
19                     distances.get(u).put(v, Integer.MAX_VALUE);
20                 }
21                 matrix.get(u).put(v, null);
22             }
23         }
24
25         for (Node u : nodes) {
26             for (GeneralEdge<Node> edge : graph.getEdgesByNode(u)) {
```

```
27                distances.get(u).put(edge.getDestination(), edge.getWeight());
28                matrix.get(u).put(edge.getDestination(), edge.getDestination());
29            }
30        }
31
32        for (Node k : nodes) {
33            for (Node i : nodes) {
34                for (Node j : nodes) {
35                    if (distances.get(i).get(k) != Integer.MAX_VALUE
36                    && distances.get(k).get(j) != Integer.MAX_VALUE) {
37                        int newDist = distances.get(i).get(k)
38                            + distances.get(k).get(j);
39                        if (newDist < distances.get(i).get(j)) {
40                            distances.get(i).put(j, newDist);
41                            matrix.get(i).put(j, matrix.get(i).get(k));
42                        }
43                    }
44                }
45            }
46        }
47
48        Node current = start;
49        while (current != null) {
50            shortestPath.add(current);
51            if (current.equals(target)) {
52                break;
53            }
54            current = matrix.get(current).get(target);
55        }
56    }
57
58    public void printDistanceMatrix() {
59        Set<Node> nodes = distances.keySet();
60        System.out.print("␣|␣");
61        for (Node node : nodes) {
62            System.out.print(node + "␣␣␣|␣");
63        }
64        System.out.println();
65
66        for (Node u : nodes) {
67            System.out.print(u + "|␣");
68            for (Node v : nodes) {
69                if (distances.get(u).get(v) == Integer.MAX_VALUE) {
70                    System.out.print("INF␣|␣");
71                } else {
```

```
72                      StringBuilder value = new StringBuilder();
73                      value.append(distances.get(u).get(v).toString());
74                      value.append("␣".repeat(Math.max(0, 3 - value.length())));
75                      value.append("␣|␣");
76                      System.out.print(value);
77                  }
78              }
79
80              System.out.println();
81          }
82      }
83
84      public void showPath() {
85          System.out.println(shortestPath);
86      }
87  }
```

Listing 6.3: Floyd-Warshall: FloydWarshall.java

6.6.3 Beispielimplementierung

Die Beispielimplementierung ist exemplarisch als Unittest auf Grundlage des Beispielgraphes aus Abbildung 6.2 in das Projekt integriert.

Anfangs wird über die `GeneralGraph`Klasse eine neue Instanz instanziiert und alle Knoten im Graph werden als Instanz einer `Node` instanziiert. Danach wird jeder Knoten über die `addNewNode()`-Methode dem Graphen hinzugefügt.

Nachdem alle Knoten des Graphen implementiert sind, wird über die `addEdge()`-Methode jedes `Node`-Objekt mit einer gerichteten Kante dem Graphen mit ihrem jeweiligen Endknoten und dem Kantengewicht übergeben.

Über eine Instanz der `FlyodWarshall`-Klasse werden alle Wege vom gegebenen Zielknoten mit der `computeShortestPath()`-Methode zu allen vorhandenen Knoten berechnet, sofern vorhanden.

Die `printDistanceMatrix()`-Methode zeigt die Distanzmatrix in der Konsole an und die `showPath()`-Methode den berechneten kürzesten Weg.

```
1  public class FloydWarshallTestSampleGeneralGraph {
2
3      @Test
4      public void setGraphAndStartNodeA_findWay() {
5          GeneralGraph<Node> graph = new GeneralGraph<>();
6          Node nodeA = new Node("A");
```

```
7           Node nodeB = new Node("B");
8           Node nodeC = new Node("C");
9           Node nodeD = new Node("D");
10          Node nodeE = new Node("E");
11
12          graph.addNewNode(nodeA);
13          graph.addNewNode(nodeB);
14          graph.addNewNode(nodeC);
15          graph.addNewNode(nodeD);
16          graph.addNewNode(nodeE);
17
18          graph.addEdge(nodeA, nodeB, 1);
19          graph.addEdge(nodeB, nodeC, 6);
20          graph.addEdge(nodeD, nodeC, 7);
21          graph.addEdge(nodeD, nodeC, 2);
22          graph.addEdge(nodeD, nodeE, 4);
23          graph.addEdge(nodeE, nodeA, 3);
24          graph.addEdge(nodeE, nodeB, 5);
25
26          FloydWarshall floydWarshall = new FloydWarshall();
27          floydWarshall.computeShortestPath(graph, nodeD, nodeB);
28          floydWarshall.printDistanceMatrix();
29          floydWarshall.showPath();
30      }
31  }
```

Listing 6.4: Floyd-Warshall: Beispiel

Die Ausgabe aus dem Beispiel ist:

```
 | E   | D   | B   | A   | C   |
E| 0   | INF | 4   | 3   | 10  |
D| 4   | 0   | 8   | 7   | 2   |
B| INF | INF | 0   | INF | 6   |
A| INF | INF | 1   | 0   | 7   |
C| INF | INF | INF | INF | 0   |
```

Listing 6.5: Floyd-Warshall: Ausgabe Matrix

```
[D, E, A, B]
```

Listing 6.6: Floyd-Warshall: Ausgabe Ergebnis

6.7 Vorteile

Der Algorithmus von Floyd und Warshall löst das ”All-pairs shortest path” Problem, indem der Algorithmus für alle Paare von Knoten in einem gerichteten Graph den kürzesten Weg findet. In der Praxis ist das von besonderer Relevanz, wenn in einem Netzwerk für alle Paare von Knoten, der kürzeste Verbindungsweg ermittelt werden muss. Außerdem kann der Floyd-Warshall-Algorithmus im Gegensatz zum Dijkstra- und A-Stern-Algorithmus mit Kanten negativen Gewichts umgehen.

6.8 Nachteile

Bei besonders großen Graphen kann eine Implementierung des Floyd-Warshall-Algorithmus zu Laufzeit- und Speicherproblemen führen. Die ist aufgrund der Speicherung einer Matrix die die Abstände aller Knoten beinhaltet.
Gleichzeitig führt das Speichern aller Paare und deren Informationen zur Ineffizienz, weil möglicherweise nicht alle berechneten Informationen von Relevanz sind.
Der Algorithmus kann zwar mit negativen Kanten umgehen, erkennt allerdings negative Zyklen nicht, was zur Folge haben kann, dass falsche Ergebnisse geliefert werden.

6.9 Zeitkomplexität

Die Laufzeit des Floyd-Warshall-Algorithmus liegt in $O(n^3)$, weil für die drei Variablen k, i, j die Werte von 1 bis n durchlaufen werden müssen. Dabei ist n die Anzahl der Knoten eines Graphen.

7 Fazit

Die wichtigsten Algorithmen zur Bestimmung eines kürzesten Weges und die mathematischen Erfordernisse wurden anhand von ausführlichen Beispielen dargestellt. Die jeweiligen Abschnitte zu Pseudocode und der informellen Beschreibung werden helfen, die vorgestellten Algorithmen auch in anderen Programmiersprachen zu implementieren und geben Aufschluss über deren generellen Ablauf.

Alle bisher vorgestellten Pathfinding-Algorithmen finden den kürzesten Weg von einem Ausgangsknoten zu einem Zielknoten (oder zu allen anderen Knoten eines Graphen).

Dijkstra priorisiert die Suche dabei nach Gesamtkosten vom Ausgangsknoten. A* priorisiert zusätzlich nach geschätzten Kosten bis zum Ziel. Und Bellman-Ford priorisiert gar nicht, kann dafür mit negativen Kantengewichten umgehen. Floyd-Warshall hingegen findet die kürzesten Wege zwischen allen Paaren von Start- und Zielknoten (Floyds Variante).

Der Algorithmus von Floyd-Warshall ist vom Typ ein APSP-Algorthmus (All-pairs shortest path). Das bedeutet, der Algorithmus befasst sich mit der Bestimmung der kürzesten Pfade zwischen allen Knotenpaaren eines Graphen. Die anderen behandelten Algorithmen, Dijsktra, A-Stern und Bellman-Ford sind vom Typ SSSP-Algorithmen (Single-source shortest path). Diese Algorithmen bestimmen den kürzesten Weg zwischen einem Startknoten und allen anderen Knoten im Graphen.

Zu Beachten ist, dass der Algorithmus von Dijsktra und der A-Stern-Algorithmus nicht mit negativen Kanten umgehen können. Die Wahl des richtigen Algorithmus ist also grundsätzlich vom Anwendungsgebiet abhängig. Wird zusätzlich die Zeitkomplexität betrachtet, ist der Algorithmus von Floyd-Warshall schneller als der Bellman-Ford und der A-Stern-Algorithmus aufgrund seiner Heuristik effizienter als der Dijkstra-Algorithmus.

	Typ	negative Kanten	Ansatz	Datenstruktur	Zeit-komplexität (n: Knoten, m: Kanten	Anwendung
Dijkstra Algorithmus	SSSP	Nein	Greedy Algorithmus	-	$O(n^2)$	Effizient um kürzeste Pfade von einer Quelle zu finden
A-Stern Algorithmus	SSSP	Nein	Greedy Algorithmus	PriorityQueue	$O(n \cdot \log n + m \cdot n)$ (mit PriorityQueue)	Wie Dijkstra nur schneller und effizienter
Bellman-Ford Algorithmus	SSSP	Ja	Greedy Algorithmus	-	$O(n^2)$	Arbeitet wie Dijkstra, allerdings begrenzt durch Anzahl maximaler Iterationen
Floyd-Warshall Algorithmus	APSP	Ja	Dynamische Programmierung	Mehrdimensionale Arrays	$O(n^3)$	Findet Pfade zwischen allen Paaren von Knoten

Tabelle 7.1: Vergleich der Algorithmen

Laut Tabelle 7 ist der A-Stern-Algorithmus der effiziente Algorithmus. Mit einer Komplexität von $O(n \cdot \log n + m \cdot n)$, ist der A-Stern-Algorithmus der schnellste. Der wichtigste Nachteil ist, dass der Algorithmus nicht mit negativen Kanten umgehen kann und eine Heuristik benötigt wird. Sofern jedes Kantengewicht e in einem Graphen G $e \geq 0$ ist, sollte eine Version des A-Stern zur Bestimmung eines kürzesten Pfades in Betracht gezogen werden.

Um die Laufzeit einer A-Stern-Implementierung zu verbessern, kann statt der Verwendung einer *PriorityQueue* auch ein *TreeSet* verwendet werden. Die Laufzeit verändert sich dann zu $O(n \cdot \log n)$ für $m \in O(n)$. Das liegt an den Methoden REMOVE() und ADD(), die die Kosten für $T_{\text{decreaseKey}}$ auf $T_{\text{decreaseKey}} = O(\log n) + O(\log n) = O(\log n)$ Kosten verringen. [1]

Ist jedoch mindestens eine Kante in einem zu untersuchenden Graphen negativ, so sollte laut Tabelle eine Version des Bellman-Ford-Algorithmus verwendet werden. Mit einer Zeitkomplexität von $O(n^2)$ ist der Algorithmus von Bellman und Ford Zeitintensiv, allerdings deutlich schneller als der Algorithmus von Floyd und Warshall.

Es gibt eine Vielzahl von Algorithmen die Probleme kürzester Wege betrachten und in den vorangehenden Kapiteln keine Erwähnung finden. Um eine ideale Möglichkeit zur Lösung für ein spezifisches Problem zu finden, sollten auch andere Algorithmen oder Kombinationen aus Algorithmen betrachtet werden.

Es gibt eine Bibliothek, Namens `https://networkx.org/`, das in der Programmiersprache Python entwickelt wurde.
Diese Bibliothek beinhaltet bereits diverse Kürzeste-Wege-Algorithmen [2].

Es ist zu erwarten, dass nach wie vor an den Problemen kürzester Wege, z.B. dem Problem des Handlungsreisenden, geforscht wird.

[1] `https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/TreeSet.html`, 26.07.2024
[2] `https://networkx.org/documentation/latest/reference/algorithms/shortest_paths.html`, 26.07.2024

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Quellenverzeichnis

[1] *A*-Algorithmus*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=A*-Algorithmus&oldid=241111126.

[2] *Bellman-Ford-Algorithmus*. Stand: 02.08.2024. URL: https://de.wikipedia.org/w/index.php?title=Bellman-Ford-Algorithmus&oldid=241982171.

[3] *Der A*-Algorithmus*. Stand: 17.05.2024. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-a-star/index_de.html.

[4] *Der Bellman-Ford-Algorithmus*. Stand: 03.08.2024. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-bellman-ford/index_de.html.

[5] *Der Dijkstra-Algorithmus*. Stand: 17.05.2024. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-dijkstra/index_de.html.

[6] *Der Floyd-Warshall-Algorithmus*. Stand: 03.08.2024. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_de.html.

[7] *Dijkstra-Algorithmus*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Dijkstra-Algorithmus&oldid=243279594.

[8] *Floyd-Warshall-Algorithmus*. Stand: 03.08.2024. URL: https://de.wikipedia.org/w/index.php?title=Algorithmus_von_Floyd_und_Warshall&oldid=245119866.

[9] *Graph (Graphenthoerie)*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Graph_(Graphentheorie)&oldid=242940052.

[10] *Graphentheorie*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Graphentheorie&oldid=244561292.

[11] *Greedy-Algorithmus*. Stand: 20.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Greedy-Algorithmus&oldid=240097417.

[12] *Kante (Graphenthoerie)*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Kante_(Graphentheorie)&oldid=242940053.

[13] *Knoten (Graphenthoerie)*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Knoten_(Graphentheorie)&oldid=228123966.

[14] Krumke, Sven Oliver und Noltemeier, Hartmut. *Graphentheoretische Konzepte und Algorithmen*. Bd. 2. Vieweg-Teubner, 2009. ISBN: 978-3-8348-0629-1. DOI: 10.1007/978-3-8348-9592-9.

[15] *Kürzester Pfad*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=K%C3%BCrzester_Pfad&oldid=239946565.

[16] Prof. Dr.-Ing. Krischke, André und Dipl.Math. techn. Röpcke, Helge. *Graphen und Netzwerktheorie*. Carl Hanser Verlag, 2015. ISBN: 978-3-446-43229-1. URL: %7Bhttps://www.hanser-elibrary.com/isbn/9783446432291%7D.

[17] Teschl, Gerald und Teschl, Susanne. *Mathematik für Informatiker. Band 1: Diskrete Mathematik und Lineare Algebra*. Bd. 3. Springer-Verlag, 2008, S. 411–461. ISBN: 978-3-540-77432-7. DOI: 10.1007/978-3-540-77432-7.

[18] *Weg (Graphentheorie*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Weg_(Graphentheorie)&oldid=221379790.