Informatik
Hauptcampus
HOCHSCHULE TRIER

Verifikation von Algorithmen zur Pfadplanung

Verification of Pathfinding Algorithms

Kremp Nicolas

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Christof Rezk-Salama, Dipl.-Inf.

Trier, den 12.11.2020

Vorwort

Herzlichen Dank an meine Familie und alle Freunde, die mich während der Studienzeit mental wie physisch unterstütz haben und mir ermöglicht haben, diesen Weg zu gehen. Besonderen Dank gilt meinem Bruder, der mir über die gesamte Zeit beigewohnt und aktiv mit mir gelernt und geforscht hat, sowie meinen Eltern und Großeltern, die mir in schwierigen Lebenslagen immer Halt gaben. Es sei auch ein Dank an den betreuenden Professor ausgerichtet, der mir interessante Themengebiete und Ansätze aufzeigen konnte, um eine Arbeit mit Nachdruck zu verfolgen und auszuarbeiten.

Kurzfassung

Diese Arbeit verifiziert Algorithmen der Pfadplanung in verschiedenen Anwendungsbereichen. Das sieben Brücken Problem von Kaliningrad führte zur Entwicklung der Graphentheorie. Eine Verifikation der Algorithmen die derartige Probleme lösen, wird in der Arbeit durch das Betrachten der Laufzeitklasse und die Ermittlung der Laufzeiten auf ein und dem selben Rechner stattfinden. Die ausgewählten Algorithmen können angepasst und verbessert werden um somit auf ein Problem abgestimmt zu werden. Zur Erhebung empirischer Daten wurde im Rahmen dieser Arbeit ein Projekt implementiert. Die Ergebnisse des Projektes werden vorgestellt und diskutiert, um einen projektspezifischen Algorithmus zu verifizieren. Dadurch soll klar werden, was ausschlaggebend ist, um einen Algorithmus verifizieren zu können. Dabei wird sich herausstellen, dass es mit allen Algorithmen möglich ist die Problemstellung des Projektes zu lösen. Der A*-Algorithmus ereicht dabei die beste Laufzeit und seine Implementierung eignet sich am besten. In der Zusammenfassung werden das Arbeitsumfeld wie die Ergebnisse kritisch bewertet und festgestellt, wieso sich in den Teilgebieten des Routing, der Navigation, der Tomographie, dem Stock-Marketing und der Spielentwicklung die verwendeten Algorithmen eignen.

This thesis verifies algorithms of path planning in different application areas. The Kaliningrad seven bridge problem led to the development of graph theory. A verification of the algorithms that solve such problems will take place in the work by looking at the runtime class and determining the runtimes on one and the same computer. The selected algorithms can be adapted and improved in order to be tailored to a problem. A project was implemented as part of this work to collect empirical data. The results of the project are presented and discussed in order to verify a project-specific algorithm. This should make it clear what is crucial in order to be able to verify an algorithm. It will turn out that it is possible to solve the problem of the project with all algorithms. The A * algorithm achieves the best runtime and its implementation is best suited. In the summary, the work environment and the results are critically assessed and it is determined why the algorithms used are suitable in the sub-areas of routing, navigation, tomography, stock marketing and game development.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

1.1 Entstehung und Geschichte

Die Aufgabe der Pfadplanung ist es, einen Weg zwischen zwei Punkten zu finden, der beliebigen Anforderungen entspricht. Das heißt, dass der Weg beispielsweise durch bestimmte Wegpunkte führen muss, möglichst natürlich verlaufen soll oder höchstens ein festgelegter Aufwand entstehen darf, um den Weg zu beschreiten. Es kann ebenfalls verknüpfte Anforderungen an den gesuchten Weg geben. Im wissenschaftlichen Sinne befasste sich Leonard Euler 1736 erstmals mit der Aufgabe der Pfadplanung. Die Frage entstand aufgrund der Topologie der Stadt Kaliningrad, die bis 1945 Königsberg hieß. Sie wird vom Fluss Pregolja in vier Stadtgebiete geteilt, was in Abbildung 1.1 skizziert ist [Sch09, S.317-358].

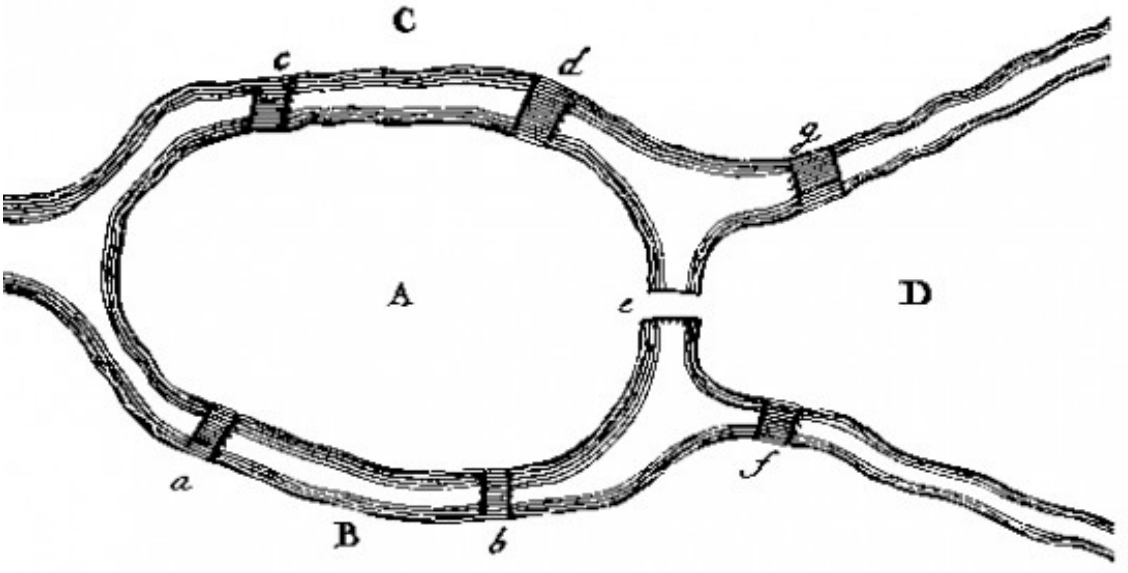

Abb. 1.1. Der Fluss Pregolja und die Stadtgebiete A bis D mit den Brücken a bis g [Sch09, Abbildung aus Kapitel 15.]

Daraus entstand die Frage, ob es einen Weg geben würde, der einen Rundgang durch die Stadt ermöglicht, dabei jede Brücke genau einmal überquert und wieder am Startpunkt endet. Euler erkannte, dass das gestellte Problem auf ein mathematisches Problem zurückzuführen ist. Er fand eine allgemeine Lösung, indem er die Repräsentationsform wechselte. Den Beweis, dass es keinen solchen Weg geben würde, führt Euler anhand eines Graphen und er stütz ihn darauf, dass A notwendigerweise eine gerade Anzahl von Brücken benötigt, aber eine ungerade Anzahl von Brücken hat. Eine Nachbildung des Diagramms, das Euler zur Lösung der Aufgabe erstellte, ist in Abbildung 1.2 dargestellt [Sch09, S.317-358].

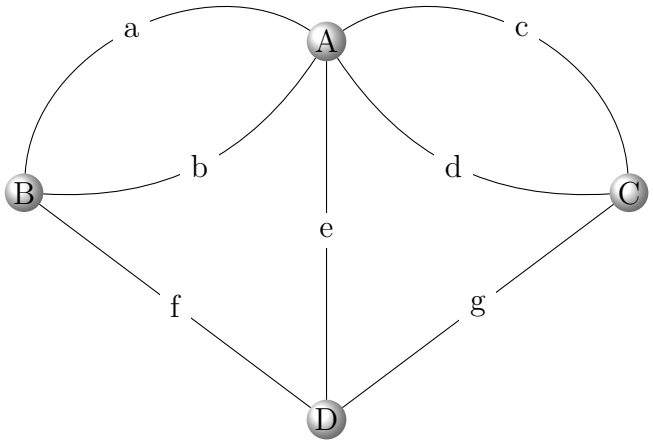

Abb. 1.2. Zugehöriger Graph für die sieben Brücken von Königsberg [Sch09, Anlehnung an Abbildung aus Kapitel 15]

Die aus der Lösung entstandenen Erkenntnisse hatten keinen signifikanten wissenschaftlichen Wert, doch die Ansätze die Euler zur Lösung verwendete, waren der Grundsatz zur Disziplin der Graphentheorie. Die Graphentheorie ist eine Repräsentationsform, welche sich in vielen Bereichen, insbesondere der Informatik, zur Lösung der verschiedenen Probleme eignet. So bietet sie beispielsweise effiziente Lösungen bei der Navigation innerhalb einer Datenbank, beim Data-Mining, der Modellierung von Netzwerken sowie der Planung von kürzesten Routen und Wegen [Sch09, S.317-358].

1.2 Grundlagen der algorithmischen Pfadplanung

Um ein Problem mathematisch oder algorithmisch zu berechnen, bedarf es einer Repräsentationsform. Diese Arbeit nutzt vorwiegend die Mengendarstellung und das Modell der Graphen. Die mathematische Definition 1.1 ist dazu nötig, die Schritte des der Algorithmen in Kapitel 3 nachvollziehen zu können.

Definition 1.1. *Graph*

1. *Ein gerichteter Graph ist ein Paar G = (V,E), wobei V eine nicht leere endliche Menge und* $E \subset V \times V \setminus \{(v,v)|v \in V\}$ *ist. V heißt die Menge der Knoten und E heißt die Menge der gerichteten Kanten. Ein Knoten w heißt benachbart oder adjazent zu v, wenn* $(v,w) \in E$*. v heißt Anfangspunkt und w heißt Endpunkt der Kante (v,w).*
2. *Bei einem Graphen besitzen die Kanten keine Richtung.* $E \subset \mathcal{P}_2(V)$*, der Menge der zweielementigen Teilmengen von V. Die Knoten* $v, w \in V$ *sind benachbart oder adjazent, wenn* $\{v,w\} \in E$*, v und w heißen Endpunkte der Kante* $e = \{v,w\}$*. Die Kante* $\{v,w\}$ *heißt inzident zu v und zu w.*
3. *Sei* $v \in V$*.* $U(v) := \{w \in V|w$ *adjazent zu* $v\}$ *heißt Umgebung von v. Die Anzahl* $|U(v)|$ *der Elemente von U(v) heißt Grad von v, kurz deg(v). Die Anzahl der Knoten* $|V|$ *heißt Ordnung von G. [Kne19, S. 218, 219]*

Bemerkungen:

1. $|V|$ *und* $|E|$ messen die Größe eines Graphen. Wir geben Aufwandsabschätzungen in Abhängigkeit von $|V|$ *und* $|E|$ an.
2. Einem gerichteten Graphen kann man einen (ungerichteten) Graphen — den unterliegenden Graphen — zuordnen, indem einer Kante (v,w) die Kante $\{v,w\}$ zugeordnet wird. Den einem Graphen zugeordneten gerichteten Graphen erhalten wir, wenn wir jede Kante $\{v,w\}$ durch die Kanten (v,w) und (w, v) ersetzen.

Es ist möglich an einer Kante zwischen zwei Knoten ein Gewicht zu notieren. Dieses Gewicht kann einen Aufwand darstellen, welcher nötig ist, um einen Weg zu beschreiten. Existiert ein solches Gewicht, heißt der Graph gewichtet, ansonsten ungewichtet. Bei den gerichteten Graphen kann es zwischen zwei Knoten auch zwei verschiedene Kanten geben. Dies dient dazu, verschiedene Gewichtungen in der entgegengesetzten Richtung zu notieren [Kne19, S. 251].
Um die Laufzeit der Algorithmen zu verbessern, kann auch eine Adjazenzmatrix gespeichert werden. Diese Matrix gibt für jedes Knotenpaar an, ob eine Kante zwischen den Knoten existiert. Gibt es eine Kante zwischen zwei Knoten, so ist der Eintrag in der Adjazenzmatrix, auf den Wert '1' zu setzen, ansonsten auf '0' [OW12, S. 591].

Ein solcher mathematisch definierter Graph kann ohne weiteren Berechnungsaufwand auch visuell dargestellt werden. In Abbildung 1.3 sind die vorangehend beschriebenen Eigenschaften veranschaulicht.

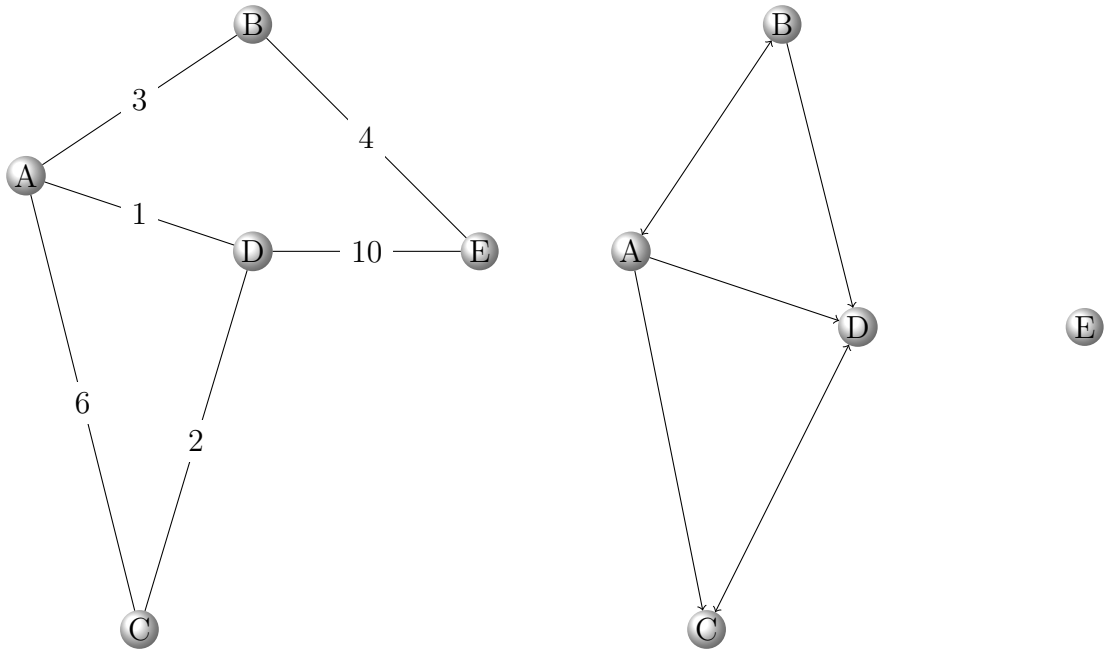

Abb. 1.3. links: Ein ungerichteter, gewichteter, zusammenhängender Graph ; rechts: zwei gerichtete, ungewichtete Teilgraphen (siehe Definition 1.2). Quelle: eigene Darstellung.

Auf diesen Grundlagen aufbauend ist eine erste Definition eines Pfades oder Weges möglich, siehe Definition 1.2. Dabei ist zu beachten, dass ein Weg nur existieren kann, solange die Start- und Zielknoten auch mit dem restlichen Graphen verbunden sind. Dies kann beispielsweise beim *preprocessing*, vor der eigentlichen Kalkulation, mit der Adjazenzmatrix geprüft werden.

Definition 1.2. *Zusammenhang und Wege*[1]
Ein Graph $G = (V, E)$ heißt zusammenhängend, wenn es für alle Knotenpaare $v, w \in V$ mit $v \neq w$ einen Weg in G von v nach w gibt. Die Knotenfolge $u_0, \ldots, u_k \in V$ heißt dabei ein Weg (der Länge k) von u_0 nach u_k, wenn für alle $0 \leq i < k$ gilt $(u_i, u_i + 1) \in E$ (bzw. im ungerichteten Fall $\{ u_i, u_i + 1 \} \in E$). Ein Weg $u_0, \ldots, u_k \in V$ heißt knotendisjunkt, wenn alle Knoten paarweise verschieden sind ($u_i \neq u_j$ für alle $0 \leq i < j \leq k$), und kantendisjunkt, wenn jede Kante nur einmal durchlaufen wird ($(u_i, u_i + 1) \neq (u_j, u_j + 1)$ für alle $0 \leq i < j < k$ bzw. im ungerichteten Fall $\{ u_i, u_i + 1 \} \neq \{ u_j, u_j + 1 \}$).

Der *kürzeste Weg* ist der Weg, bei dem die Summe der Kantengewichte minimal ist [Bri20, S. 293]. Ganz gleich was die numerischen Werte verkörpern.
Das *Shortest Path Problem* lässt sich aus den Grundlagen der Graphentheorie ableiten. Dieses Problem kann in mindestens vier Teilprobleme eingeteilt werden. Das eigentliche Problem ist das *Single Source Shortest Path Problem* (SSSP), bei dem die Instanz ein Graph ist und der *kürzeste Weg* von einem Startknoten zu allen anderen Knoten im Graphen betrachtet wird. Invers dazu ist das *Single Destination Shortest Path Problem* (SDSPP oder SDSP), bei dem von allen Knoten im Graphen der *kürzeste Weg* hin zu einem Zielknoten betrachtet wird. Ein weiteres Teilproblem ist das *Single Pair Shortest Path Problem* (SPP). Das letzte Teilproblem ist das *All Pairs Shortest Path Problem* (APSP). Bei den letzten beiden Problemen werden die *kürzesten Wege* zwischen ein oder mehreren Knotenpaaren gesucht [KN12, S. 169, 170].
Die Eingabeinstanz ist dabei meist ein ungerichteter und gewichteter Graph, lediglich die Fragestellung ändert sich. In Fällen, bei denen es eine Rolle spielt, in welche Richtung ein Weg beschritten wird, kann ein gerichteter Graph benutzt werden. Damit stehen die grundlegenden Gemeinsamkeiten fest. Im Kapitel 3 wird auf die Algorithmen, Dijkstra, A*, Bellman-Ford und Floyd-Warshall näher eingegangen.

[1] Weicker, Algorithmen und Datenstrukturen, Springer Fachmedien Wiesbaden 2013 [WW13, S. 8,9]

1.3 Das Problem der Verifikation

Da jeder der genannten Algorithmen eine andere Vorgehensweise hat und unterschiedliche Prinzipien verfolgt werden, muss zur Verifikation der Algorithmen festgehalten werden, welchen Zweck das Eingabeproblem verfolgt. Das heißt, dass die Algorithmen an der Laufzeit gemessen und verglichen werden können, jedoch berücksichtigt werden muss, wie gut sie sich eignen, eine Problemstellung zu lösen. So ist beispielsweise in Erwägung zu ziehen, ob es das Ziel ist, die Kosten zu jedem Knoten im Graphen stetig zu kennen und aktuell zu halten oder ob es ausreicht, den Teilgraphen zu kennen, der den kürzesten Weg beinhaltet.

Im Rahmen dieser Arbeit ist zur Erhebung empirischer Daten ein Projekt implementiert worden. Das Projekt ist ein Spielfeld, welches durch einen Graphen ein Gitternetz realisiert. Es enthält eine Figur, die sich über das Spielfeld bewegen kann und dabei Hindernisse berücksichtigt. Um den kürzesten Weg zu ermitteln, werden die Algorithmen aus Kapitel 3 verwendet und das Ergebnis verglichen. Im Kapitel 4 werden einige Aufgabengebiete der Pfadplanung untersucht und festgestellt, welche Algorithmen sich für das jeweilige Gebiet durchgesetzt haben. Der Aufbau des Projektes sowie die genaue Funktionsweise und die hervorgegangenen Ergebnisse werden in Kapitel 5 beschrieben. Die daraus gewonnen Ergebnisse werden verwendet um in Kapitel 6 zu einem Entschluss zu kommen. Es wird erörtert, warum sich die Algorithmen für das jeweilige Aufgabengebiet eignen und ob sie ersetzt werden könnten. Es folgt eine Zusammenfassung und ein kurzer Ausblick auf zukünftige Fragestellungen im abschließenden Kapitel 7.

Unter der Überschrift Verwandte Arbeiten in Kapitel 2 sind einige Arbeiten erwähnt, die das Grundwissen vertiefen können oder aber maßgeblich zur Lösung einzelner Probleme der Graphentheorie und der Pfadplanung beigetragen haben. Des Weiteren ist festzustellen, dass sich diese Arbeit stark an der Situation, wie sie im Projekt gegeben ist, orientiert und die Erhebung der Daten zur Laufzeit nur für dieses Programm gültig sind. Im Allgemeinen gelten die ermittelten Laufzeitklassen der Algorithmen nach der O-Notation. Eine Berücksichtigung zur Überwachung der Threads auf dem ausführenden Rechner wurde nicht umgesetzt. Die erhobenen Durchschnittswerte sind in diesen Fall Näherungswerte, die die Aussagen zu den Laufzeitklassen untermauern sollen.

2

Verwandte Arbeiten

In diesem Kapitel wird auf Arbeiten, die ähnliche Themen vorstellen, referenziert. Durch einige dieser Arbeiten kann das Grundverständnis vertieft oder Erkenntnisse in nahe liegenden Themengebieten gewonnen werden. Es ist nicht zwingend erforderlich, die vorgestellenten Arbeiten zu kennen, um den Sinn dieser Arbeit nachvollziehen zu können.

Das Buch „Algorithmen und Datenstruckturen“ von Helmut Knebel beinhaltet wichtige Informationen zur Graphentheorie Kapitel 1.2 und erklärt die Grundlagen dieser [Kne19].

In dem Artikel von Dijkstra 1959 „A note on two problems in connexion with graphs“ sind Notizenn zur Ermittlung der minimalen Kantenlänge innerhalb eines Graphen notiert. Der Artikel hilft die Eingangsfrage der Pfadplanung zu verstehen [D$^+$59].

Um die O-Notation genauer zu betrachten, ist das Buch zur Zahlentheorie von Paul Bachmann vorgeschlagen [Bac04]. Die O-Notation wird im Verlauf der Arbeit genutzt, um die Laufzeit der Algorithmen untereinander zu vergleichen.

Der Artikel „On the Design of Link-State Routing Protocol for Connection-Oriented Networks“ von M. Sivabalan und H. T. Mouftah beschäftigt sich mit Routingprotokollen sowie der Kommunikation innerhalb eines Netzwerkes [SM01]. Dieser Artikel gibt aufschlussreiche Informationen, um das Gebiet des Routing in Kapitel 4.1 zu vertiefen.

Die Studienarbeit des Institutes für Verkehrswirtschaft, Straßenwesen und Städtebau der Universität Hannover analysiert Routensuchverfahren und stellt Anforderungen an Routensuchverfahren im Verkehrswesen auf, die bei der Navigation Kapitel 4.2 anfallen [Ros96].

In der Arbeit von H. Fuchs „Optimal Surface Reconstruction from Planar Conturs“ sowie in der Arbeit “Surfaces form Conturs“ von David Meyers und Shelly Skinner „Surface form Conturs“ sind wichtige Merkmale und die Vorgehensweise bei der Oberflächenrekonstruktion wiedergegeben. Diese finden beispielsweise in der Computertomographie Kapitel 4.3 Verwendung [FKU77], [MSS92]. Des Weiteren behandelt die Diplomarbeit von Björn Labitzke die Kernprobleme der Volumenvisualisierung bildgebender Verfahren wie Computer- oder Magnetresonanztomographie [Lab09].

Die beiden Artikel „Identifying Market Arbitrage Opportunity using Pathfinding Algorithm“ und „Currency Arbitrage Detection“ führen Gedanken zur Pfadplanung im Stockmarketing Kapitel 4.4 mithilfe des Bellman-Ford Algorithmus an [DI19], [MJS20].

Der von der Victoria University in Melbourne veröffentlichte Artikel beschäftigt sich mit der Darstellung unterschiedlicher Repräsentationsformen der Graphen in einem Computerspiel. Dieser Artikel verdeutlicht den Unterschied zwischen der Darstellung des Spielfeldes als Graph oder als Nav-Mesh und dient zum Veranschaulichung Vorgehensweise der Pfadplanung in Computerspielen siehe Kapitel 4.5 [CS11].

Die im Folgenden aufgeführten Arbeiten und Ausarbeitungen beschäftigen sich mit der Pfadplaung unter Verwendung verschiedener Algorithmen und der Berücksichtigung von unüberwindbaren Hindernissen. Die Masterarbeit „Vergleich von Pathfinding-Algorithmen“ von Heiko Waldschmidt und die Ausarbeitungen, „Algorithmen zur Pfadplanung“ von Martin Helmreich, Sebastian Prokop, Chris Schwemmer sowie „Übersicht über die Pfadplaung“ von Benjamin Lange und „Pathfinding-Algorithmen in verschiedenen Spielegenres“ von Andreas Hofmann, dienten zur Inspiration und Strukturierung dieser Bachelor Arbeit [Hei08], [MH05], [Lan16], [Hof13].

3

Verwendete Algorithmen

3.1 Dijkstra

Der Dijkstra-Algorithmus wurde nach dem Informatiker Edsger Wybe Dijkstra benannt und bezieht sich auf eine Veröffentlichung aus dem Jahre 1959 [D$^+$59], [Dör17, S.136]. Der Algorithmus gehört zu den Sogenannten *Greedy-Algorithmen* (z.Dt. gierig). Die *Greedy-Algorithmen* erzielen bei jedem Iterationsschritt eine maximale Teillösung. Eine getroffene *Greedy-Auswahl* ist unumkehrbar, welches der Grund für eine effiziente Laufzeit ist [DPV06, S.139], [WW13, S.117], [CLRS09, S.414]. Eine Optimalität der Gesamtlösung ist jedoch nicht gewährleistet [Dör17, S.144].
Die Breitensuche ist das iterative Besuchen der Knoten von einem beliebigen Startknoten s in einem gerichteten oder ungerichteten Graphen $G = (V, E)$ [Dör17, S.129], [CLRS09, S.596], [KN12, S.161]. Dazu wird der Graph von s aus betrachtet und in Schichten eingeteilt, die jeweils die Entfernung zum Startknoten s widerspiegeln, siehe Definition 3.1. Der Breitensuche-Algorithmus terminiert, wenn alle Knoten durchlaufen wurden [DPV06, S.116].

Definition 3.1. *Breitensuche*[1]

1. *Initialisierung des Startknoten $s = 0$ in der First-In First-Out (FIFO) Warteschlange Q einfügen und Distanz $d(u) = \infty$ für alle Knoten $u \in V$ ohne s*
2. *Solange $Q \neq \emptyset$ werden folgende Schritte wiederholt:*
3. *Entnehme Knoten u aus Q*
4. *Wiederhole für alle adjazenten Knoten v von $u \in V$ folgende Schritte:*
   (a) Prüfe ob die Bedingung $d(v) = \infty$ zutrifft
   (b) Wahr: Knoten v wird in Q eingefügt und Distanz $d(v) = d(u)+1$ gesetzt
   (c) Falsch: Wiederhole Schritt 4

[1] Angelehnt an Cormen Thomas et al, Introducing to Algorithms, The MIT Press 2009 [CLRS09, S.595] und Krumke Sven Oliver und Hartmut Noltemeier, Graphentheoretische Konzepte und Algorithmen, Vieweg+Treubner Verlag Wiesbaden 2012 [KN12, S.161].

Um das *Single Source Shortest Path Problem* effizient zu lösen, verwendet der Dijkstra-Algorithmus das Verfahren der Breitensuche [DPV06, S.121], [CLRS09, S.658]. Die Eingabe des Algorithmus ist ein Graph $G = (V, E)$ und ein beliebiger Startknoten s. Der Dijkstra-Algorithmus sucht einen Pfad mit geringstmöglichen Kosten zu allen anderen Knoten in G [OW12, S.406]. Existiert kein Pfad von s zu einem Knoten v, so ist die Distanz d des Knotens ∞ (dh. $d(s, v) = \infty$). Die Pfadsuche gelingt nur dann, wenn keine negativen Kantenkosten c existieren (d.h. $c \in \mathbb{R}^+$) [Dör17, S.139]. Ist ein Knoten v über einen anderen Knoten u mit geringeren Kosten erreichbar, wird die Distanz d aktualisiert (dh. $d(s, u)+c(u, v) \leq d(s, v) : \forall c(u, v) \in E$) [Dör17, S.136]. In Abbildung 3.1 wird die *Pseudocode-Implementierung* des Dijkstra-Algorithmus mit Hilfe einer Prioritätswarteschlange (engl. *Priority Queue*) umgesetzt:

```
1     Dijkstra:
2
3     //Initialisierung
4     for u in V:
5         d(u)= unendlich
6         prev(u)=nil
7     d(s)=0
8
9     //H wird als Prioritaetswarteschlange initialisiert
10    while H is not empty:
11        u=deletemin(H)
12
13        for all edges(u,v) in E:
14
15            if d(v)>d(u)+c(u,v):
16                d(v)=d(u)+c(u,v)
17                prev(v)=u
18                decreasekey(H,v)
```

Abb. 3.1. Pseudocode des Dikstra-Algorithmus zum lösen des *Shortest-Path Problem* in Anlehnung an [DPV06, S.120,121].

Bei der Initialisierung wird über eine Schleife die Distanz d aller Knoten auf ∞ und der Vorgänger aller Knoten auf nil gesetzt. Die Distanz von Startknoten s wird auf den Wert '0' gesetzt, damit dieser zuerst betrachtet wird. In der Prioritätswarteschlange H werden alle Knoten-Wert Paare geordnet nach Distanz hinterlegt z.B. {(∞, u), (∞, v), (0, s),... } [DPV06, S.121]. Die while Schleife läuft solange bis H leer ist. Folgende Schritte werden also wiederholt: Der Knoten mit der kleinsten Distanz wird aus der Prioritätswarteschlange H entfernt, dieser Vorgang wird im Pseudocode als *deletemin* Operation bezeichnet. Für jede Kante (u, v) aus der Kantenmenge E wird verglichen, ob der Knoten v mit geringeren Kosten erreicht werden kann. Sollte ein Knoten v über einen anderen Knoten u mit geringeren Kosten erreichbar sein, wird die Distanz d aktualisiert, dies gilt auch für alle unbesuchten Knoten mit Distanz = ∞. Der Vorgänger des Knotens v wird auf den Knoten u gesetzt, da dieser über einen kostengünstigeren Pfad erreichbar ist. Im Anschluss wird der Distanzwert des Knotens v in der Prioritätswarteschlange über die *decreasekey* Operation aktualisiert [DPV06, S.120, 121] [CLRS09, S.505].

Die Laufzeit des Algorithmus hängt davon ab, wie die Prioritätswarteschlange implementiert ist [CLRS09, S.661]. So kann der Algorithmus zum Beispiel eine Prioritätswarteschlange als Array, Binary Heap oder als Fibonacci-Heap realisiert werden [DPV06, S.125]. Die Art der Implementierung hängt von verschiedenen Faktoren ab. Die Array-Implementierung zum Beispiel hat eine Laufzeit von $O(|V|^2)$, jedoch wird diese Implementierung bei Graphen mit Anzahl der Kanten $|E|$ kleiner als die Anzahl der Knoten $|V|$ verwendet. Dies liegt daran, dass die *decreasekey* Operationen in konstanter Laufzeit, also $O(1)$ ausgeführt werden kann. Die Suche des minimalsten Wertes der *deletemin* Operation liegt jedoch bei $O(|V|)$. Die Suche nach dem Knoten mit minimalem Wert ist durch die Heap-Datenstruktur unkompliziert und führt dazu, dass die *deletemin* Operation sowohl bei dem Binary Heap als auch bei dem Fibonacci-Heap in der Laufzeit $O(log|V|)$ realisierbar ist [DPV06, S.125]. Die *decreasekey* Operationen der Binary Heap-Implementierung belaufen sich auf $O(log|V|)$ und bei der Fibonacci-Heap Variante, bei einer amortisierten Betrachtung der Laufzeit, auf $O(1)$. Der Fibonacci-Heap besitzt somit eine Laufzeit $O(|V|log|V| + |E|)$ [DPV06, S.125] [OW12, S.426,427].

3.2 A*

Der erstmals von Peter Hart, Nils Nilsson und Bertram Raphael beschiebene A*-Algorithmus ist eine Erweiterung des oben beschriebenen Dijkstra-Algorithmus [HP66]. Damit zählt der Algorithmus ebenfalls zu den *single source shortest path algorithms*. Die folgenden zwei Unterkapitel beschäftigen sich mit den Grundprinzipien des A*-Algorithmus und möglichen Erweiterungen, die angewandt werden können, wenn der Graph sich während der Berechnung ändern könnte [Ste97] [Ert13] [KL02].
Die Idee, die dem A*-Algorithmus zugrunde liegt, ist zu schätzen, welche Knoten im Graphen für den kürzesten Weg relevant sind und diese zu betrachten. Um das Prinzip umzusetzen wird eine Heuristik verwendet [Dör17, Kapitel 4.5.2]. Die Schätzfunktion oder Heuristik, ist der grundlegende Unterschied zum Dijkstra - Algorithmus. Dadurch wird aus dem *Greedy-Algorithmus* ein zielgerichteter Algorithmus. Für die folgende Bewertungsfunktion $f(v)$ gilt: [2]

- $g(v)$, die Kosten vom Startknoten s zum Knoten v,
- $h(v)$, die geschätzten Kosten von Knoten v zum Zielknoten t,
- $f(v)$, die Kosten vom Startknoten s zum Zielknoten t über v:

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

Zusammengefasst sind die Kosten eines Knoten v die bisherigen Kosten addiert mit den geschätzten Restkosten des Knoten v zum Zielknoten t. Hierbei ist $h(v)$ die Heuristik. Die Wahl der Heuristik ist ausschlaggebend für die Laufzeit des

[2] Dörn Sebastian, Programmieren für Ingenieure und Naturwissenschaftler, Springer Fachmedien Wiesbaden 2017 [Dör17, S. 148]

Algorithmus und in Abhängigkeit der Anwendung zu wählen. Da es unterschiedliche Heuristiken wie beispielsweise die *Manhattan Heuristik*, die *Diagonal Heuristik* oder die *Euklidische Distanz* gibt, die situationsbedingt unterschiedlich genau schätzen, muss eine Heuristik zulässig sein [Ert13, S. 109]. Im Folgenden sind drei der bekanntesten Heuristiken notiert.

1. Manhattan Heuristik
$$h(v) = (|v.x - t.x| + |v.y - t.y|)$$
2. Diagonal Heuristik
$$h(v) = max\{|v.x - t.y|, |v.y - t.y|\}$$
3. Eukliedische Distanz
$$h(v) = \sqrt{(v.x - t.x) \quad + (v.y - t.y)}$$

Überschätzt eine Heuristik die Kosten, geht die Optimalität des A*-Algorithmus verloren [Ert13, S. 110] [HP66, S. 102], die bereits von den Erfindern bewiesen wurde [HP66, S. 103, 104]. Ist die Heuristik festgelegt, kann der A*-Algorithmus iterativ mit folgenden Schritten eine Menge speichern und bearbeiten:[3]

1. Bestimmung aller Nachbarknoten $N_G(s)$ von Startknoten s
2. Speicherung der Menge $N_G(s)$ in einer Liste W
3. Berechnung des f -Wertes für jeden Knoten in W
4. Wiederholung der folgenden Schritte:
   (a) Bestimmung des Knotens $v \in W$ mit minimalem f -Wert
   (b) Falls v der Zielknoten t ist $\Rightarrow$ kürzester Weg ist gefunden
   (c) Hinzufügen der Elemente von $N_G(v)$ in die Liste W
   (d) Berechnung des f -Wertes für jeden Knoten in $N_G(v)$
   (e) Entfernung des Knotens v aus W

Ist der A* mit einem Fibonacci-Heap implementiert ist er, mit einer Laufzeit von $O(|V|\ log|V|)$, performanter als der Dijkstra Algorithmus, der entsprechend implementiert in $O(|V|\ log|V|+|E|)$ liegt. Ohne Fibonacci-Heap implementiert liegt der A* zwar in der Laufzeitklasse $O(|V|^2)$, nähert sich aber in den seltensten Fällen seiner asymptotischen oberen Schranke, dem *worstcase* an.

Auf der Folgeseite ist der Ablauf des A*-Algorithmus in Abbildung 3.2 Graphisch dargestellt anhand eines Graphen und der Veränderung in jedem Schritt.

[3] Dörn Sebastian, Programmieren für Ingenieure und Naturwissenschaftler, Springer Fachmedien Wiesbaden 2017 [Dör17, S. 149], Darstellung in Abbildung 3.2

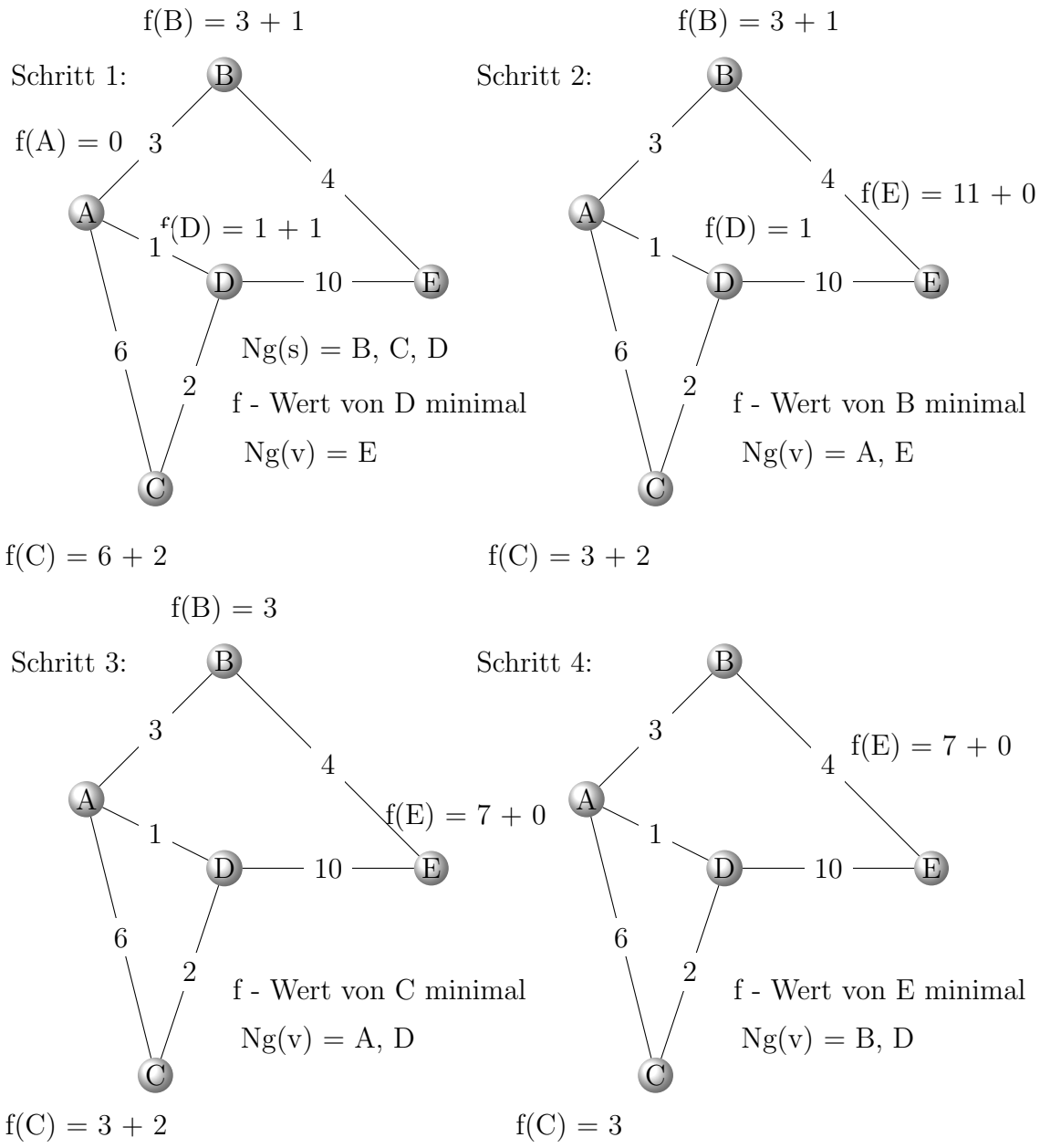

Abb. 3.2. Schritt 1: Startknoten ist A und Zielknoten E, Nachbarn werden in Liste W übernommen, A aus W entfernt. Schritt 2: E wird in W übernommen, D entfernt. Schritt 3: B wird aus W entfernt. Schritt 4: C wird aus W entfernt, Letzter Knoten ist Zielknoten E. Quelle: Eigene Darstellung.

3.3 Bellman-Ford

Der Bellman-Ford Algorithmus (im Folgenden als BF abgekürzt), ist ein weiterer auf Graphen basierender *single source Shortest path algorithm*. Er wurde von Richard Bellman und Lester Ford entwickelt, um sich mit dem Problem der Findung des kürzesten Weges zu befassen. Bellman hat die Problemstellung in seiner Arbeit mit einer Suche nach dem kürzesten Weg zwischen zwei Städten, welche in einem Netz aus weiteren Städten liegen, erläutert [Bel58, S.2].

Der Algorithmus baut auf einem gerichteten und gewichteten Graphen auf. Ein Vorteil gegenüber dem in Kapitel 3.1 erläuterten Dijkstra-Algorithmus oder dem A*-Algorithmus ist, dass es bei Bellman-Ford auch negative Kantengewichtungen geben kann. Es wird auch erkannt, ob der verwendete Graph einen negativen Zyklus hat [KN12, S.184]. Auf der Implementierungsebene liegt bei dem BF-Algorithmus das Prinzip der dynamischen Programmierung zugrunde [Bel58, S.2,S.3]. Die dynamische Programmierung wird genutzt, um Optimierungsproblemen in Teilprobleme zu teilen und systematisch Zwischenergebnisse zu speichern. Der Ansatz der dynamischen Programmierung ist bei dem BF-Algorithmus gut zu erkennen [Bel58, S.3,S.4]. Statt von dem Startknoten ausgehend den günstigsten Weg zu finden, wird werden die Kosten eines jeden Knotens zu jedem weiteren mit den Kantenkosten addiert. Dies wird dann gespeichert und an den Startknoten weitergegeben.

Das genau Funktionsprinzip des Algorithmus und der Graph, der die besagten Zyklen erkennt, wird im Folgenden erläutert. Zu Beginn wird von einem Startknoten ausgehend eine Liste für die Kosten zu den im Netz vorhandenen Knoten mit $+\infty$ initialisiert. Die Initialisierung mit $+\infty$ bedeutet, dass noch kein Weg zu diesem Knoten bekannt ist. Der Algorithmus wählt nach der Initialisierung einen Knoten, welcher sich in dem Netz befindet, aus. Es gibt für die Auswahl des Knotens keine bestimmte Reihenfolge. Der Algorithmus relaxiert die Kanten, die von dem ausgewählten Knoten ausgehen. Das Relaxieren von Kanten ist das erneute Ausrechnen der Kosten zu dem Nachbarknoten des ausgewählten Knotens [DMS14, S.263]. Dazu werden die Kosten, die benötigt werden, um den ausgewählten Knoten zu erreichen, mit den Kosten zum nächsten Nachbarknoten addiert. Ist die Summe kleiner als die Kosten zuvor, werden die Kosten zu dem Nachbarknoten aktualisiert. Dieser Schritt wird $N - 1$ mal ausgeführt. Dabei ist zu beachten, dass mit N die Anzahl der Knoten im Graph gemeint ist [Bel58, S.3]. Die in der Abbildung 3.3 gezeigten Graphen veranschaulichen diese Schritte.

Nach $N - 1$ Durchläufen hat der Algorithmus den kürzesten Weg gefunden, sofern kein negativer Kreis in dem Graphen vorhanden ist. Um einen negativen Kreis vollständig zu erkennen, müssen nach dem $(N - 1)$ten Durchlauf ein letztes Mal alle Kanten relaxiert werden. Wenn sich in diesem Durchlauf die Kosten eines Knotens ändern, wird der Eintrag für die Kosten des Knotens auf $-\infty$ gesetzt [DMS14, S.263,S.264].

Das Ersetzen der Kosten auf den Wert $-\infty$ dient dazu, verifizieren zu können, welche Knoten in den negativen Kreis eingebunden sind beziehungsweise von dem Kreis beeinflusst werden. Der Bellman-Ford erkennt zwar negative Kreise, kann aber keinen kürzesten Weg finden, wenn ein solcher Kreis existiert. Der Grund dafür ist, dass sich die Kosten in dem Kreis immer verringern und es mit jedem weiteren Durchlauf einen neuen kürzesten Pfad gibt. Nachdem der Bellman-Ford alle Durchläufe beendet hat, gibt dieser als Ergebnis eine Liste mit den Kosten zu jedem Knoten in dem Netz ausgehend von dem Startknoten zurück. Der Bellman-Ford-Algorithmus hat eine Laufzeit von $O(N * M)$. Das entsteht dadurch, dass bei jedem Durchlauf die N Knoten und deren M Kanten Bearbeitet werden müssen [DMS14, S.185, S.186].

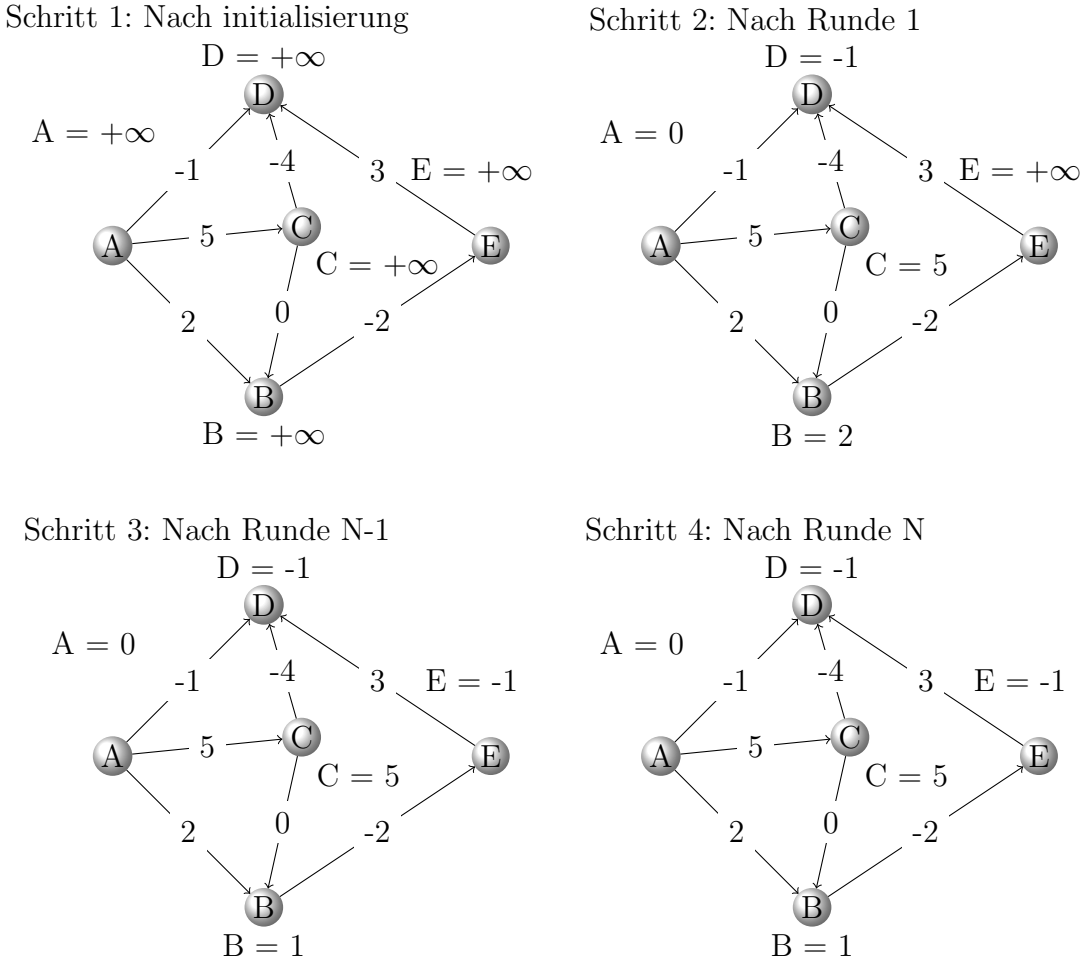

Abb. 3.3. Schritt 1: Zeigt die Kosten zu den Knoten nach der Initialisierung. Schritt 2: Zeigt die Kosten nach dem Durchlauf 1. Schritt 3: Zeigt die Kosten nach dem Durchlauf $N - 1$. Schritt 4: Zeigt die Kosten nach dem Durchlauf N, hier ist zu sehen, dass kein Knoten die Kosten $-\infty$ hat, also ist kein negativer Zyklus vorhanden.

Für den Algorithmus gibt es einige Modifikationen. Eine dieser Modifikationen ermöglicht es dem Bellman-Ford-Algorithmus negative Zyklen schon während der Durchläufe zu erkennen. Das Grundverfahren dieses modifizierten Bellman-Ford bleibt gleich. Der Unterschied zum Grundalgorithmus ist, dass er um einen zusätzlichen Schritt erweitert wird. Die Idee hinter diesem Schritt ist es, dass der Grundalgorithmus unterbrochen wird, sobald ein Knoten eine Kante mit negativer Gewichtung relaxiert. Des Weiteren wird geprüft, ob dieser Knoten zu einem negativen Zyklus gehört. Das genaue Verfahren kann in der Arbeit von W. Domschke nachgelesen werden [Dom73, S. 6].

3.4 Floyd-Warshall

Der Floyd-Warshall Algorithmus ist nach den Informatikern Robert Floyd und Stephen Warshall benannt. Der Algorithmus setzt sich aus zwei Teilen zusammen, dem Teil von Floyd [Flo62] und dem Teil von Warshall [War62]. Beide Versionen wurden 1962 vorgestellt, zusammen ist der Algorithmus auch unter dem Namen *Tripple-Algorithmus* bekannt. Floyds Teil, siehe Abbildung 3.4, berechnet den APSP (all-pairs shortest path) für einen Graphen. Wir gehen von folgender Betrachtung aus:

Falls der kürzeste Weg von u nach v durch w geht, dann sind die enthaltenen Teilpfade von u nach w und von w nach v schon minimal. Unter der Annahme die kürzesten Wege zwischen allen Knotenpaaren, die nur über Knoten mit Index kleiner als k führen, sind schon bekannt, werden alle kürzesten Wege über Knoten mit Index höchstens k gesucht. Dadurch ergeben sich für einen Pfad von u nach v zwei Möglichkeiten: Entweder der Weg geht über den Knoten k, dann setzt er sich zusammen aus schon bekannten Pfaden von u nach k und von k nach v, oder es ist der schon bekannte Weg von u nach v über Knoten mit Index kleiner als k. Angenommen, der Graph ist gegeben durch seine Gewichtsmatrix w. $w[i, j]$ ist das Gewicht der Kante von i nach j, falls eine solche Kante existiert. Falls es keine Kante von i nach j gibt ist $w[i, j]$ unendlich. Dann kann die Matrix d der kürzesten Distanzen durch folgendes Verfahren bestimmt werden:

```
1
2     Algorithmus von Floyd:
3
4       For all i,j : d[i,j] = w[i,j]
5       For k = 1 to n
6           For all Pairs i,j
7               d[i,j] = min(d[i,j], d[i,k] + d[k,j])
```

Abb. 3.4. Pseudocode des Floyd-Algorithmus zum lösen des *APSP (all-pairs shortest path) Problem*. Quelle: Eigene Darstellung.

Da der Algorithmus auf Grundlage einer Matrix rechnet und keine Liste implementiert wie vergleichsweise der Dijkstra oder der A*-Algorithmus, bietet er sich an, wenn beispielsweise Vertexdaten bereits nach Zeile und spalte gespeichert wurden. Eine solche Matrix kann effizient durch eine Matrixmultiplikation errechnet werden. Doch der Algorithmus benötigt weitere Schritte, um den kürzesten Weg in einem Graphen zu finden, denn er speichert nur die kleinste Distanz zwischen den einzelenen Knotenpaaren. Mit dem Teil von Warshall lässt sich die transitive Hülle berechnen, siehe Abbildung 3.5. Der Algorithmus wird um die Information der Adjazenzmatrix ergänzt und folgendermaßen abgeändert:

```
1
2     Algorithmus von Warshall:
3
4      For k = 1 to n
5          For i = 1 to n
6              If d[i,k] = 1
7                  For j = 1 to n
8                      If d[k,j] = 1
9                          d[i,j] = 1
```

Abb. 3.5. Pseudocode des Warshall-Algorithmus zum bestimmen der transitiven Hülle. Quelle: Eigene Darstellung.

In der Zeile (9) wird $d[i, j]$ auf 1 gesetzt, genau dann, wenn ein Pfad von i nach k und ein Pfad von k nach j über Kanten mit Index kleiner als k existiert. Der Floyd-Algorithmus funktioniert auch, wenn die Kanten ein negatives Gewicht eingetragen haben. Zyklen mit negativer Länge werden (anders als beim Bellman-Ford-Algorithmus) nicht erkannt und verfälschen das Ergebnis. Im Nachhinein sind negative Zyklen dennoch erkennbar durch negative Werte auf der Hauptdiagonalen der Distanzmatrix. Da es zu numerischen Problemen kommen kann, wenn dies erst am Ende der Berechnung geprüft wird, sollte die Diagonale nach jedem mal, wenn in Zeile (7) ein Element geändert wurde, geprüft werden [Hou10]. Die Laufzeit des Floyd-Warshall-Algorithmus liegt in $O(n^3)$, weil über die drei Variablen k, i, j für die Werte von 1 bis n iteriert werden muss. Dabei ist n Anzahl der Knoten des Graphen. Um den Ablauf des Algorithmus an einem Graphen zu veranschaulichen, wird in Abbildung 3.6 ein Beispiel angeführt und die zugehörige Tabelle findet sich in Abbildung 3.7.

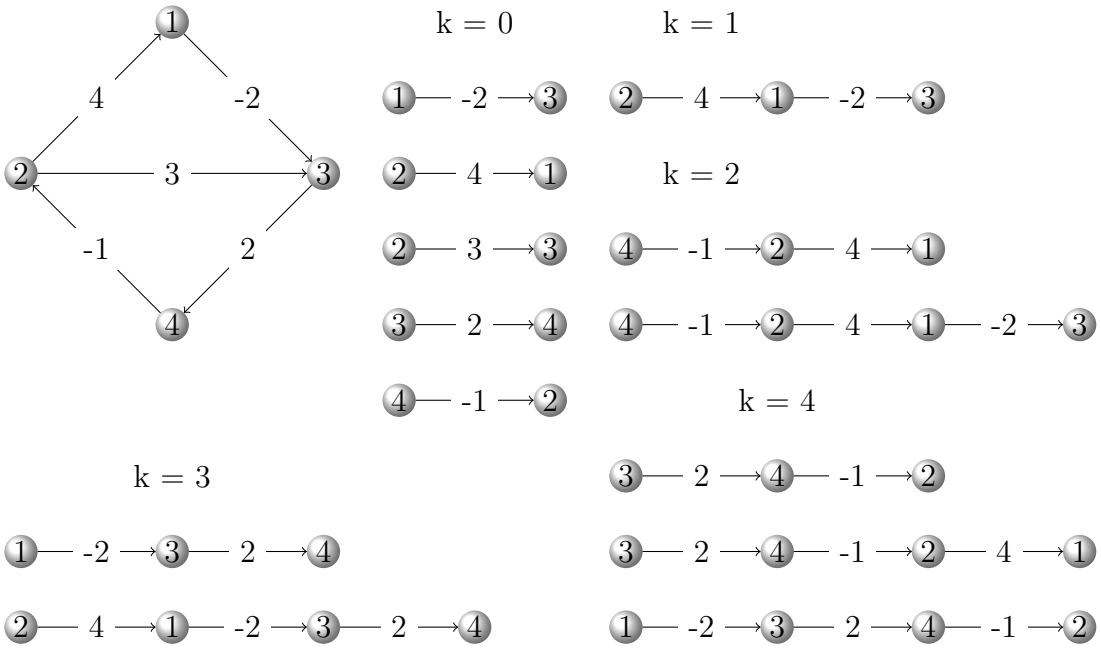

Abb. 3.6. Der Algorithmus wird auf dem Graphen links mit vier Knoten ausgeführt, für jedes $k = 1$ bis n. Quelle: Eigene Darstellung.

k = 0

i \ j	1	2	3	4
1	0	∞	−2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	−1	∞	0

k = 1

i \ j	1	2	3	4
1	0	∞	−2	∞
2	4	0	**2**	∞
3	∞	∞	0	2
4	∞	−1	∞	0

k = 2

i \ j	1	2	3	4
1	0	∞	−2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	**3**	−1	**1**	0

k = 3

i \ j	1	2	3	4
1	0	∞	−2	**0**
2	4	0	2	**4**
3	∞	∞	0	2
4	3	−1	1	0

k = 4

i \ j	1	2	3	4
1	0	**−1**	−2	0
2	4	0	2	4
3	**5**	**1**	0	2
4	3	−1	1	0

Abb. 3.7. Zeigt die Tabelle zu Abbildung 3.6 und die daraus hervorgehende Matrix nach jeder Iteration von $k = 0$ bis $k = n$.

4

Anwendungsgebiete

Dieses Kapitel gibt Aufschluss über einige wichtige Teilgebiete, in denen die Algorithmen, die das SSSP lösen, zum Einsatz kommen. Im Folgenden werden, die verschiedenen Anforderungen heraus gearbeitet, um Erkennen zu können, welche Algorithmen nützlich und welche in dem Einzelfall nicht zu gebrauchen sind. Es kann auch auf einem Teilgebiet einen alternativen Algorithmus geben, der nicht die beste Lösung erzielt, aber weniger aufwendig zu implementieren ist oder weniger Speicher benötigt.

4.1 Routing

In einem Netzwerk aus Rechnern findet die Kommunikation über eine Verbindung statt. Jeder Rechner ist mit einem Knotenpunkt verbunden, um Informationen in das Netz einzustreuen. Dabei kann das Netz verschiedene Topologien aufweisen. Bei einer *peer-to-peer Verbindung* (P2P) sind beispielsweise alle Rechner gleichberechtigt, was den Unterschied zu einer Ring-, Bus- oder Sterntopologie darstellt. Baumtopologien werden in Netzwerken eher selten verwendet. Das Ziel in einem Netzwerk ist es, idealerweise ein *fully connected* Netzwerk herzustellen, was in der Praxis aber sehr schwer ist. Die verschiednen Topologien sind im Buch von André Hilbig dargestellt [Hil16, S.10 Abbildung 2.1]. Ein Knoten innerhalb des Netzwerkes ist ein Router. Er ermöglicht den Einstieg in ein Netzwerk und ermöglicht es, die Datenpakete mithilfe eines Protokolls weiterzuleiten [Hil16, S.16-25]. Die Routen, die das Protokoll vorgibt, können beim *statischen Routing* fest vorgegeben sein. Andernfalls wird der kürzeste Weg mithilfe einer Routingtabelle ermittelt. Die verwendeten Protokolle unterscheiden sich zwischen Interior Gateway Protocols (IRP), die innerhalb einer Domäne bzw. eines autonomen Systems eingesetzt werden und Exterior Gateway Protocols (ERP), die zwischen Domänen eingesetzt werden [BNMB20, S.196]. Zu der ersten Gruppe gehören Routing-Protokolle mit Distance-Vector-Algorithmen wie das Gateway to Gateway Protocol (GGP), das Routing Information Protocol (RIP), das Interior Gateway Routing Protocol (IGRP) und das Enhanced Interior Gateway Routing Protocol (EIGRP), mit Link-State-Algorithmen arbeiten das Intermediate System to Intermediate System Protocol (IS-IS) und Open Shortest Path First (OSPF).

Zu den Distanz-Vector Algorithmen zählt der Bellman-Ford-Algoritmus und zu den Link-State-Algorithmen der Dijkstra-Algorithmus [SM01] [D$^+$59] [Bel58] [BNMB20]. Der A*-Algorithmus sowie der Floyd Warshall Algorithmus wird in keinem dieser Protokolle verwendet. Da es beim Routing nötig ist, Informationen über das gesamte Netzwerk zu haben.

4.2 Navigation

Routensuchverfahren zählen mit zu den grundlegenden Algorithmen, wenn es um die Verkehrsplanung geht. Bereits bei der Lösung des *Traveling Sailsmen Problem* (TSP) verwendet Kruskal Pfadplanungs-Algorithmen [Kru56]. Aus den Aufgaben die ein Routensuchverfahren im Verkehrswesen hat, geben sich letztlich zwei hervorstechende Anforderungen [Ros96, S.3]. Zwar kann durch den Straßenverlauf und der daraus gegebenen Wegstrecke für ein Teilstück eine Vereinfachung vorgenommen werden, dennoch ist das entstehende Netz meist sehr groß [DK03, S.177 Abbildung 3: Aproximation für Teilstücke]. Deshalb ist ein Algroithmus gefordert, der eine möglichst schnelle Berechnungszeit aufweist. Dabei ist es nicht notwendig den kompletten Graphen zu kennen, sondern nur das Teilstück, welches den kürzesten Weg enthält. Die zweite Anforderung ergibt sich daraus, dass auch die Verkehrslage, Umleitungen und eventuelle Staus berücksichtigt werden müssen oder aber gegebenenfalls die Kosten des Weges durch Autobahnmaut oder erhöhte Benzinkosten mit in Betracht gezogen werden [Ros96, S.3]. Die Abstraktion von einem Verkehrsnetz auf einen Graphen kann problemlos erfolgen wie auch die Abbildung auf [Ros96, S.11] zeigt. Generell kommen zur Pfadplanung im Verkehrswesen, der durch Moor veränderte und auf Dijkstra basierende Algorithmus zum Einatz. Aber auch der A*-Algorithmus und verschiedene Matrixmultiplikationsalgorithmen wie beispielsweise der von Floyd und Warshall entwickelte Algorithmus, in optimierter Form. [Ros96, S.20-44].

4.3 Tomographie

Die Computertomographie ist ein bildgebendes Verfahren, in dem es darum geht aus MR CISS Volumendaten ein Bild zu erzeugen. Dazu kann, unter anderem, ein Emissions- Absorbtions Modell zu Grunde liegen, welches die Ausbreitung von Licht im Raum oder innerhalb eines Volumens berücksichtigt. Im Wesentlichen wird zwischen direkten und indirekten Verfahren unterschieden. Unter den indirekten Verfahren gibt es die *Surface Segmentation* und die Verfahren zu Isoflächen. Die Verfahren zu *Surface Segmentation* sind GPU-basierte Verfahren der Volumenvisualisierung. Sie nutzen das Objektraum- (2D oder 3D Texture Slicing) oder das Bildraum-Verfahren (3D Texture Slicing / GPU Raycasting) [PB13]. Soll aus einem Volumendatensatz eine Oberfläche extrahiert werden, die keine Isofläche ist, kommt eine Konturbasierte Segmentierung zum Einsatz. Die drei Kernprobleme hierbei sind das Korrespondenz-, Tiling- und das Branching Problem [Lab09, S.6]. Gerade das Tiling Problem, die Rekonstruktion der Mantelfläche zwischen zweier

korrespondierenden Schichten, lässt sich als Problem der Graphensuche formulieren. Daher liegt es nahe, eine effiziente Lösung mit Hilfe eines Pfadplanungsalgorithmus zu bestimmen [MSS92, S.233] [BG93, S.11] [BCL96, Kapitel 2.1]. Hierzu liefert H. Fuchs interessante wie effiziente Ansätze und Verbesserungen in Anlehnung an Matrixmultiplikations-Algorithmen wie den Algorithmus von Floyd und Warshall[FKU77].

4.4 Stock-Marketing

Stock-Marketing stellt zumeist schnelle Transaktionen beispielsweise an der Börse dar. Hierbei kann es sein, dass ein Geschäft mit Wertverlust angestrebt wird, um bei der nächsten Transaktion doch wieder einen positiven Ertrag zu bekommen. Auch dieses Problem verschiedener Transaktionen oder dem Wechsel zwischen verschiedenen Währungen kann als Graph dargestellt werden. Allerdings müssen hierbei auch negative Kanten in Betracht gezogen werden [DI19] [MJS20]. Ein einfaches Beispiel hierfür ist ein Taxifahrer, der eine Fahrt annimmt, bei der er keinen Fahrgast auf dem Rückweg hat. So hat er Ausgaben für Benzin und die Fahrzeugabnutzung, aber er transportiert keinen Fahrgast. Somit sind seine Kosten positiv und sein Ertrag negativ. Die *Currency Arbitrage Detection* ist eine Handelsstrategie, die einen risikofreien Gewinn erzielt und dabei diese Möglichkeit in Betracht zieht. Dieser Gewinn wird erzielt, indem die Chancen in den Unterschieden der Unternehmen in einem klar definierten und klaren Markt oder in den verschiedenen Marktformen genutzt werden. Es ist der Prozess, einen Gewinn zu erzielen, der risikofrei ist, wenn die verschiedenen günstigen Bedingungen für Preis- und Kostenunterschiede desselben Unternehmens berücksichtigt werden, unabhängig davon, ob die Waren allein oder in identischen Kombinationen bewertet werden. Aufgrund solcher Preisinkonsistenzen und -schwankungen kann sich auf dem Markt ein harmloser und risikofreier Status verdient werden, und es besteht eine gewisse Zuverlässigkeit und Gewissheit, dass er möglicherweise mehr als seine risikofreie Rendite verdient. Der Forex-Markt ist der höchste und liquideste Markt der Welt. Ein hochliquider Markt weist auf einen geringeren Spread, ein hohes Volumen und einen hohen Umsatz hin. Hierzu wird oft ein Bellman-Ford basiertes Modell verwendet [DI19][BCL96].

4.5 Spielentwicklung

In der Welt der Computerspiele wird die Pfadplanung häufig genutzt, um KI Elemente zu steuern. Das heißt, verschiedene NPC (non-player-charakter) oder aber der Spieler selbst muss sich den Weg durch unbekanntes Terrain suchen. Steuert der Spieler eine Figur per *point and click*, so muss der Spielercharakter automatisch den überwindbaren Weg entlang laufen. Zwei Verfahren haben sich etabliert, um den Weg für Charakter zu errechnen. Die Repräsentation des Graphen über ein Wegpunkt Netz oder Graph und die Repräsentation über Navigations Meshes. Der Unterschied liegt darin das Navigations Meshes die gesamte Karte in begehbare Zonen aufteilen und der direkte Weg zum Zonenrand gesucht wird [CS11, S.127 Abbildung 3 und S.129 Abbildung 7].

Oftmals ist dennoch ein Weg gesucht, der etwas *natürlicher* und nicht geradlinig verläuft. Deshalb gibt es dynamische Ansätze sowie die Art der Implementierung, die einen *natürlicheren* Verlauf des Weges erzielen. Da der A*-Algorithmus nicht angewendet werden kann, wenn sich der Graph zur Laufzeit verändert, muss auf die *dynamische Programmierung* zurückgegriffen werden. Antony Stentz entwickelte den D*-Algorithmus *dynamic A**. Der D*-Algorithmus merkt sich die Berechnungen von nicht betroffenen Knoten und ist somit optimal für sich ändernde Graphen [Ste97, Kapitel 2, 2.2 und 3.0]. Die Ansätze für die Algorithmen IDA* *Iterative Deepening A** [Ert13, Abbildung 6.10 und Kapitel 6.2.3] und LPA* *Lifelong Planing A** stammen von Steven Koenig und Maxim Likhachev [KL02, S. 467 - 483]. Deren Arbeiten beschreiben einen iterativen Ansatz, in dem die Suchtiefe innerhalb eines Baumes[1] solange erhöht wird, bis der Zielknoten gefunden ist. In diesem Anwendungsbereich ist es nicht nötig bzw. manchmal nicht möglich den vollständigen Graph zu kennen oder aktuell zu halten. Es reicht auch hier aus, den Teilgraphen zu kennen, der den kürzesten Weg beinhaltet.

[1] Priese Lutz, Theoretische Informatik, Springer Berlin Heidelberg 2018 [PE18, S. 24] (Baum: spezielle Form eines Graphen)

5

Erhebung empirischer Daten

Zur Erhebung der empirischen Daten wurde ein Projekt in Unity umgesetzt. Die verschiedenen in Kapitel 3 aufgeführten Algorithmen wurden implementiert, um das SSSP für die Umgebung des Projektes zu berechnen. Die höhere Programmiersprache die hierbei verwendet wurde, ist C#, da das die in Unity, einer Spiel-Engine des Unternehmens Unity Technologies, üblicherweise verwendete Programmiersprache ist. Im Folgenden werden der Projektaufbau, die Implementierung und die erhobenen Ergebnisse diskutiert.

5.1 Projektbeschreibung

Das Projekt ist eine Art Spiel. Es besteht im wesentlichen aus einem Gitternetz und dem Playercharakter siehe Abbildung 5.1.

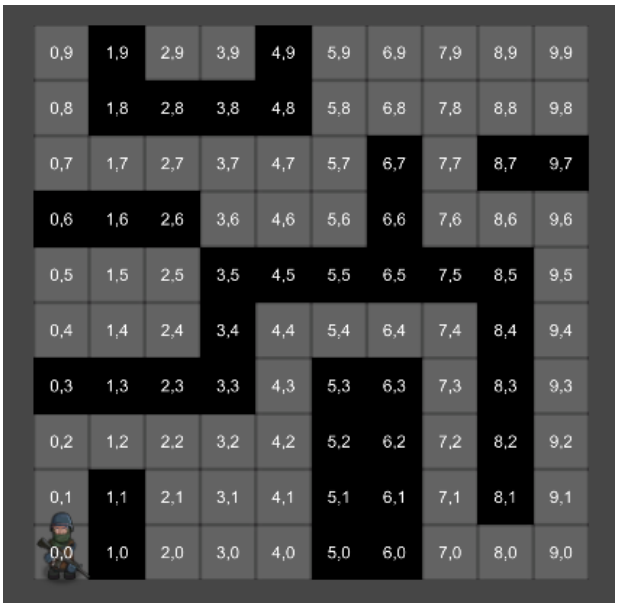

Abb. 5.1. Spielfeld mit Playercharakter und einem Labyrinth aus Hindernissen (schwarz) und Pfaden (grau). Quelle: Eigene Darstellung.

Die betretbaren Bereiche lassen sich mit der Steuerung der Maus als unpassierbar markieren. Dazu wird die rechte Maustaste benutzt. Auf der linken Maustaste kann dem Charakter der Befehl gegeben werden, von der Gitterposition (v, w) zur Gitterposition (x, y) zu laufen. Die kosten für das Bewegen belaufen sich auf 10 Punkte für gerades Laufen in die 4er Nachbarschaft (horizontal, vertikal) und 14 Punkten für diagonales Laufen in die 8er (mit Diagonalfelder) Nachbarschaft. Diese Kosten sind davon abgeleitet, dass die Diagonale u in einem Quadrat immer genau $\sqrt[2]{u} + u$ beträgt. Um speziell die Unterschiede zwischen dem Dijkstra-Algorithmus und dem A*-Algorithmus zu zeigen, ist bei diesen beiden Algorithmen eine optische Zusatzfunktion mit eingebaut. Diese zeigt das Vorgehen der Algorithmen auf den Rastern an und gibt, sobald die Berechnung endet, den errechneten Weg grün markiert aus. Siehe Abbildung 5.2.

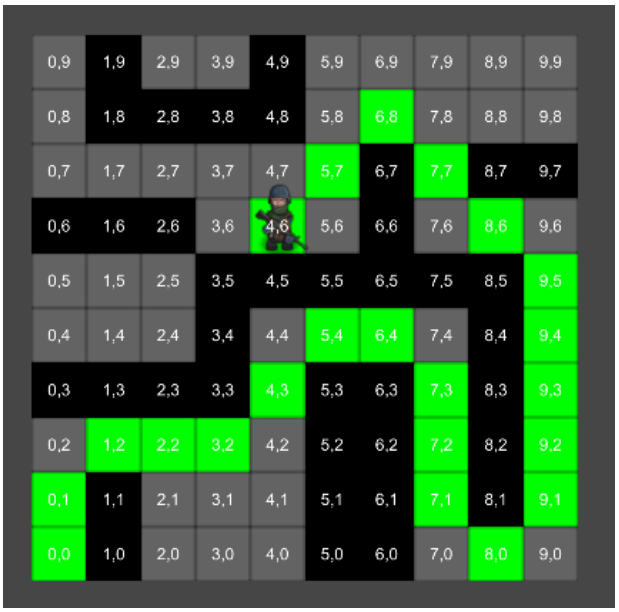

Abb. 5.2. Kürzester Weg (in grün) nach Berechnung von Startpunkt$(0, 0)$ zu Endpunkt$(4, 6)$ mit Dijkstra oder A*. Quelle: Eigene Darstellung.

Der Hauptsächliche unterschied der beiden Algorithmen Dijkstras und des A* liegt in der Verwendung einer Heuristik. Die Heuristik macht aus dem *Greedy Algorithmus* von Dijkstra den zielgerichteten A*-Algorithmus. Während der Dijkstra, wie in Abbildung 5.3 zu sehen ist alle günstigen, in Frage kommenden Knoten untersucht, betrachtet der A* nur die, bei denen auch die geschätzten Restkosten geringer sind als bei den übrigen Knoten.

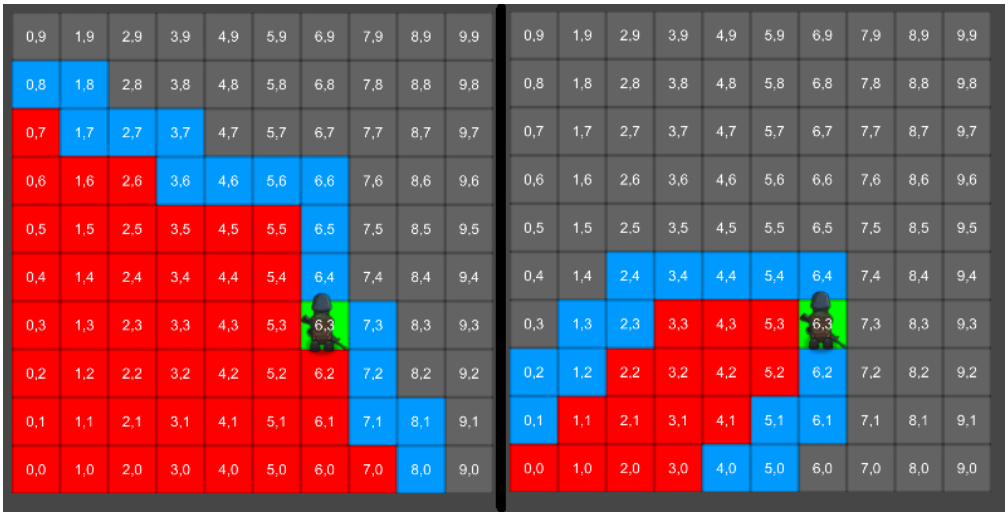

Abb. 5.3. Unterschied zwischen *Greedy Algorithmus* links und zielgerichtetem Algorithmus rechts. Die Roten Knoten sind die in der geschlossenen Liste, die blauen die Knoten in der offenen Liste und der grüne der welcher gerade untersucht wird. Quelle: Eigene Darstellung.

Hierbei sind die roten Felder bereits untersuchte die blauen, noch zu untersuchende und das grüne Feld, das welches gerade untersucht wird. Auf der Abbildung 5.3 ist deutlich zu erkennen, dass der Dijkstra wesentlich mehr Knoten untersucht bis er zum Ergebnis kommt als der A*. Deshalb bietet der A*-Algorithmus im *average case* eine durchschnittlich bessere Laufzeit.

Das Spielfeld ist variabel programmiert, es kann ganz einfach mehr oder weniger Knoten enthalten oder auch eine nicht quadratische Struktur aufweisen. Länge und Breite können also frei gewählt werden. Zur Untersuchung der Laufzeit werden später die Anzahl der Knoten im Graphen um ein Vielfaches erhört, das hat zur Folge, dass sich die Unterschiede in der Laufzeit besser abzeichnen sollen.

Zusätzlich wurden Buttons unter dem Spielfeld hinzugefügt, um die Pfadplanung mit den unterschiedlichen Algorithmen durchzuführen. Die Buttons setzen codeintern einen String und entscheiden, welcher Algorithmus benutzt wird. Der Reload-Button setzt die Szene komplett zurück und entfernt auch die visuelle Darstellung des errechneten Weges siehe Abbildung 5.4.

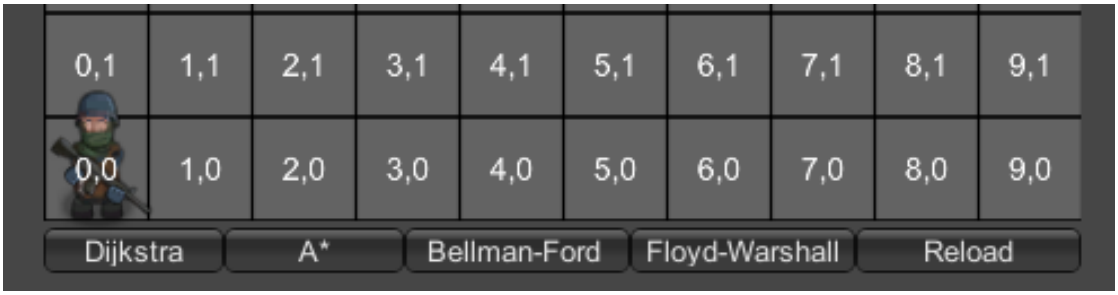

Abb. 5.4. Hinzugefügte Buttons um den Algorithmus zu wählen. Quelle: Eigene Darstellung.

5.2 Implementierung

Das gesammte Programm besteht aus fünf C# Skripten. In der Game Scene befinden sich die Objekte GameHandler, der CharakterPathfindingMovementHandler, die Objekte für Debug und die visuelle Veranschaulichung so wie das Testingobjekt, welches das ausführende Testing Skript abgehängt bekommen hat. Das Grid Skript als Teilkomponente definiert eine eigene Klasse, die einen Array aus Gridobjects verwaltet. Diese Klasse kennt ihre Höhe und ihre Breite, die Zellengröße der einzelnen Grids, die sie in dem Array abspeichert, und die Position in den Weltkoordinaten. Dazu verfügt die Klasse Grid über die dazugehörigen setter und getter Methoden, um an die Attribute zu gelangen. Und einen Konstruktor der Höhe, Breite, Zellgröße und ein Delegate als Parameter erwartet.

Die Klasse PathNode hat die Attribute für die x und y Koordinate sowie die f,g und h Kosten, die später für die Heuristk benötigt werden, ein boolean, der angibt, ob die PathNode begehbar ist und eine PathNode, welche auf den vorgänger zeigen kann. Außerdem kennt die Klasse PathNode dass Grid in dem sie sich befindet. Ebenfalls verfügt die Klasse über die Methoden, um den boolean zu setzen und die Kosten zu berechnen. Sie verfügt auch über einen Konstruktor, der ein Gridobjekt sowie Höhe und Breite des späteren Grids als Parameter erwartet.

Die Klasse Pathfinding ist mit ihrer *open* und *closedList* und den neun Attributen die komplexeste, in ihr finden auch die Berechnungen mit Hilfe der Algorithmen aus Kapitel 3 statt. Zwei konstante ganzzahlige Werte legen die Kosten für gerades und diagonales laufen fest. Ein String legt innerhalb der Klasse die Entscheidung zwischen den Algorithmen fest. Die Stopwatch Klasse aus der Collections Bibliothek misst die Zeit, die nach Aufruf der Methode FindPath vergeht, bis der Pfad gefunden wurde. Der Konstruktor der Klasse erwartet zwei ganzzahlige Werte für Höhe und Breite. Die Funktion FindPath ist überladen. Sie kann einmal mit den Vectoren der Mauszeigerpossition aufgerufen werden und einmal mit x und y Koordinate des Start- und Endknotens. Innerhalb der Methode FindPath startet der Timer und es wird eine Liste erstellt. Start und Endknoten werden gespeichert. Für jede Position wird je nach Höhe und Breite Iterativ ein GridObject PathNode erzeugt. Die Gesammtkosten der PathNodes werden auf unendlich und der Vorgänger auf null gesetzt siehe Abbildung 5.5.

```
64
65      public List<PathNode> FindPath(int startX, int startY, int endX, int endY) {
66          timer.Start();
67
68          allList = new List<PathNode>();
69
70          PathNode startNode = grid.GetGridObject(startX, startY);
71          PathNode endNode = grid.GetGridObject(endX, endY);
72
73          if (startNode == null || endNode == null)
74          {
75              // Invalid Path
76              timer.Stop();
77              return null;
78          }
79
80          for (int x = 0; x < grid.GetWidth(); x++)
81          {
82              for (int y = 0; y < grid.GetHeight(); y++)
83              {
84                  PathNode pathNode = grid.GetGridObject(x, y);
85
86                  pathNode.gCost = 99999999;
87                  pathNode.CalculateFCost();
88                  pathNode.cameFromNode = null;
89                  allList.Add(pathNode);
90              }
```

Abb. 5.5. Codeausschnitt Zeilen 64-90. Quelle: Eigene Darstellung.

Im Fall dass der String zur Wahl der Algorithmen das Keyword *floyd* oder *bellman* enthält, wird eine einfache Distranzmatrix aufgestellt deren Kosten für die Nachbarknoten den Kosten für die Bewegung entspricht. Alle anderen werden auf Unendlich, die Diagonale auf null gesetzt siehe Abbildung 5.6.

```
92      if (algo.Equals("floyd") || algo.Equals("bellman"))
93      {
94          graph = new int[grid.GetWidth() * grid.GetHeight(), grid.GetWidth() * grid.GetHeight()];
95
96          for (int i = 0; i < (grid.GetWidth() * grid.GetHeight()); i++)
97              for (int j = 0; j < (grid.GetWidth() * grid.GetHeight()); j++)
98              {
99                  PathNode node = allList[i];
100                 PathNode node2 = allList[j];
101                 List<PathNode> neighbourList = GetNeighbourList(node);
102
103                 int firstX = node.x;
104                 int fistY = node.y;
105
106                 int secX = node2.x;
107                 int secY = node2.y;
108
109                 if (neighbourList.Contains(node2))
110                     if (Math.Abs(firstX - secX) == 1 && Math.Abs(fistY - secY) == 1)
111                         graph[i, j] = 14;
112                     else
113                         graph[i, j] = 10;
114                 else
115                     graph[i, j] = INF;
116
117                 if (i == j)
118                     graph[i, j] = 0;
119             }
```

Abb. 5.6. Codeausschnitt Zeilen 92-119. Quelle: Eigene Darstellung.

Falls der String zur Wahl der Algorithmen das Keyword *dij* oder *aStar*, enthält werden die offene und geschlossene Liste angelegt. Der Startknoten hat die Kosten null, die Heuristischen kosten werden zugewiesen und die F-Kosten gesetzt. Danach fährt der Algorithmus mit oder ohne Berücksichtigung der heuristischen Funktion, wie in Kapitel 3 beschrieben, mit der Bearbeitung der Listen fort. Nicht erreichbare Knoten werden sofort der geschlossenen Liste hinzugefügt, siehe Abbildung 5.7.
Werden die Pfade mit dem A*-Algorithmus berechnet werden die Liste und die Heuristik verwendet, werden die Pfade mit dem Dijkstra-Algorithmus berechnet, läuft im Prinzip der A* Programmcode mit geschätzten Restkosten null für jeden Knoten. Damit sind alle Knoten *gleichweit* vom Zielknoten entfernt und der Algorithmus wird *greedy*. In den Zeilen 169 - 189 werden die Berechnungen und Operationen von Djikstra bzw. dem A* ausgeführt. In der Schleife in Zeilen 220 - 242 die Berechnungen des Bellman-Ford und in der Methode in Zeilen 348 - 365 die Berechnungen des Floyd-Warshall Algorithmus. Dies ist in Abbildung 5.8 auf den folgesenden Seiten zu sehen.

```
134         if (algo.Equals("aStar") || algo.Equals("dij"))
135         {
136             openList = new List<PathNode> { startNode };
137             closedList = new List<PathNode>();
138 
139             startNode.gCost = 0;
140             startNode.hCost = CalculateDistanceCost(startNode, endNode);
141             startNode.CalculateFCost();
142 
143             PathfindingDebugStepVisual.Instance.ClearSnapshots();
144             PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, startNode, openList, closedList);
145 
146             while (openList.Count > 0)
147             {
148                 PathNode currentNode = GetLowestFCostNode(openList);
149                 if (currentNode == endNode)
150                 {
151                     // Reached final node
152                     PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, currentNode, openList, closedList);
153                     PathfindingDebugStepVisual.Instance.TakeSnapshotFinalPath(grid, CalculatePath(endNode));
154                     return CalculatePath(endNode);
155                 }
156 
157                 openList.Remove(currentNode);
158                 closedList.Add(currentNode);
159 
160                 foreach (PathNode neighbourNode in GetNeighbourList(currentNode))
161                 {
162                     if (closedList.Contains(neighbourNode)) continue;
163                     if (!neighbourNode.isWalkable)
164                     {
165                         closedList.Add(neighbourNode);
166                         continue;
167                     }
```

Abb. 5.7. Erstellung der Listen, initialisierung der Kosten in Falle Dijkstra und A* Zeilen 134-167. Quelle: Eigene Darstellung.

Die Klasse Testing ist in der Scene an ein testObject angehängt und erbt von MonoBehaviour. Damit verfügt sie über die Update Methode von Unity und kann Aktionen per Frame ausführen. In diesem Fall wird sie benötigt, um die Interaktion mit der Maus zu steuern. In der Methode Start werden zu Beginn des Spieles die Aufrufe einmalig ausgeführt. Hier werden die Objekte Pathfinding mit der gewünschten Gridgröße und das Debug-Skript so wie die visuelle Veranschaulichung aufgerufen. Die Klasse stellt in dem Projekt eine Schnittstelle zur Gameengine und gleichzeitig die *main-methode* dar. Die Algorithmen wurden im Rahmen dieses Projektes ohne Erweiterungen und Verbesserungen implementiert. Der Dijkstra bzw. A*-Algorithmus wurde daher nicht mit einer Heap oder Fibonacci-Heap Struktur angelegt.

```
169                     int tentativeGCost = currentNode.gCost + CalculateDistanceCost(currentNode, neighbourNode);
170                     if (tentativeGCost < neighbourNode.gCost)
171                     {
172                         neighbourNode.cameFromNode = currentNode;
173                         neighbourNode.gCost = tentativeGCost;
174                         if (algo.Equals("aStar"))
175                         {
176                             neighbourNode.hCost = CalculateDistanceCost(neighbourNode, endNode);
177                         }
178                         else if (algo.Equals("dij"))
179                         {
180                             neighbourNode.hCost = 0;
181                         }
182
183                         neighbourNode.CalculateFCost();
184
185                         if (!openList.Contains(neighbourNode))
186                         {
187                             openList.Add(neighbourNode);
188                         }
189                     }
220             for (int n = 0; n <= allList.Count - 1; n++)
221                 for (int i = 0; i <= allList.Count - 1; i++)
222                 {
223                     List<PathNode> neighbourList = GetNeighbourList(allList[i]);
224                     for (int j = 0; j < neighbourList.Count; j++)
225                     {
226                         if (allList[i].gCost + graph[i, allList.IndexOf(neighbourList[j])] < neighbourList[j].gCost)
227                         {
228                             neighbourList[j].gCost = allList[i].gCost + graph[i, allList.IndexOf(neighbourList[j])];
229                             neighbourList[j].cameFromNode = allList[i];
230                         }
231                     }
232                 }
233
234             foreach (PathNode wayPoint in allList)
235                 foreach (PathNode wayPointToo in GetNeighbourList(wayPoint))
236                 {
237                     if (wayPoint.gCost + graph[allList.IndexOf(wayPoint), allList.IndexOf(wayPointToo)] < wayPointToo.gCost)
238                     {
239                         timer.Stop();
240                         return null; // negative circle
241                     }
242                 }
243
248             while (shortestPath[shortestPath.Count - 1] != startNode && pathLegnthReached < grid.GetWidth() * grid.GetHeight())
249             {
250                 List<PathNode> nieghbours = GetNeighbourList(shortestPath[shortestPath.Count - 1]);
251
252                 foreach (PathNode nieghbour in nieghbours)
253                 {
254                     if (graph[from, allList.IndexOf(nieghbour)] <= costs)
255                     {
256                         shortestPath[shortestPath.Count - 1].cameFromNode = nieghbour;
257                         costs = graph[from, allList.IndexOf(nieghbour)];
258                     }
259                 }
260
261                 if (!shortestPath.Contains(shortestPath[shortestPath.Count - 1].cameFromNode))
262                     shortestPath.Add(shortestPath[shortestPath.Count - 1].cameFromNode);
263
264                 pathLegnthReached++;
265             }
```

Abb. 5.8. Zeilen 169 - 189 Djikstra und A*, Zeilen 220 - 242 Bellman-Ford, Zeilen 348 - 365 Floyd-Warshall Algorithmus. Quelle: Eigene Darstellung.

5.3 Ergebnisse des Projektes

Die Algorithmen wurden am privaten Rechner unter möglichst gleichen Bedingungen durchgeführt, um Annäherungswerte an die tatsächliche Laufzeit zu bekommen. Der Prozessor des Rechners ist ein Intel(R) Core(TM) i7-6700K mit 4.00 GHz, das System ist Windows 10 mit 32,0 GB installiertem Arbeitsspeicher. Die verwendete Graphikkarte ist eine Nvidea GForce GTX 1070. Dazu wurde das Spielfeld mit einer unterschiedlichen Topologie aufgerufen. Ein 10 mal 10 Netz mit 100 Knoten ($n = 100$), ein 15 mal 15 Netz mit 225 Knoten ($n = 225$), ein 20 mal 20 Netz mit 400 Knoten ($n = 400$) und ein 30 mal 30 Netz mit 900 Kno-

ten ($n = 900$). Die Laufzeit wurde mit der oben erwähnten Stopwatch Klasse für gleichlange Wege im Graphen gemessen. Es wurde ein möglichst langer Weg, so wie wesentlich kürzere Wege untersucht. Außerdem wurde ein Spezialfall betrachtet, der ein Problem für den A*-Algorithmus darstellt, das *U-Sahpe Problem* .

Der Dijkstra Algorithmus hatte im Graphen mit 100 Knoten Laufzeiten von 2-15 Millisekunden (ms) erreicht, im Graphen mit 225 Knoten bereits eine Laufzeit von 1,02 - 1,05 Sekunden (s), beim Test mit 400 Knoten eine Laufzeit von 5-8 Sekunden und beim Graphen mit 900 Knoten größer als 20 Sekunden. Daran war zu sehen, das bei doppelter Knotenanzahl nicht die doppelte Zeit, sondern ein Vielfaches an Zeit benötigt wurde. Außerdem hatte die Streckenlänge Einfluss auf die Laufzeit, da der Algorithmus beendet wird, sobald der Weg gefunden ist.

Bei den Testläufen mit dem A* ergaben sich die Laufzeiten wie folgt. Bei $n = 100$ erreichte er 1 - 5 ms, bei $n = 225$ 44-83 ms, 1,34-2 s bei $n = 400$ und eine Laufzeit von mehr als 5 Sekunden bei $n = 900$. Auch hier war mehr als das Doppelte an Zeit nötig für doppelt so viele Knoten. Im Schnitt war der A*-Algorithmus tatsächlich schneller als der Dijkstra. Nur in einem Spezialfall, den wir später noch näher Betrachten wollen, war der A* weniger effizient. Darüber hinaus war auch hier die Weglänge ausschlaggebend für die Laufzeit, da auch dieser Algorithmus terminiert, sobald der kürzeste Weg gefunden wurde.

Bei Auswahl des Bellman-Frod-Algorithmus mit $n = 100$ wurde eine Laufzeit von 9-15 ms, bei $n = 225$ eine von 85 ms- 1,26 s, bei $n = 400$ eine von 2-3 s und bei $n = 900$ eine von größer als 25 Sekunden erreicht. Der getestete Algorithmus macht keinen Unterschied zwischen kürzeren und längeren Wegen, da er immer über alle Knoten iteriert und den gesammten Graphen berechnet. In Rahmen dieses Projektes erzielt der Bellman-Ford also durchschnittlich höhere Laufzeiten, da er mehr Informationen berechnet als nötig sind, um den kürzesten Weg zu bestimmen.

Der Floyd-Warshall-Algorithmus nutzt als einziger Algorithmus eine Matrix für die Wegekosten. Die Matrix beinhaltet am Anfang die Kosten für existierende Kanten, alle nicht direkt verknüpften Knoten bekommen die $+\infty$ Einträge. Für jeden Schritt und somit jede Iterationstiefe werden alle daraus errechenbaren Einträge aktualisiert. Daraus lassen sich dann die kosten von Start- zu Endknoten ermitteln. Der endgültige Weg wird dann über die Matrix bestimmt. So erreichte der Floyd-Warshall-Algorithmus bei $n = 100$ eine Laufzeit von 9- 15 ms, bei $n = 225$ eine von 44-83 ms, bei $n = 400$ eine von 2-3 s und bei $n = 900$ eine Laufzeit von mehr als 25 Sekunden. Auch dieser Algorithmus berechnet die komplette Matrix und ermittelt den Weg nach der Kostenmatrix. So wird also ebenfalls das gesammte Netz errechnet, bevor der Algorithmus terminiert.

Im Rahmen des Projektes ist aufgefallen, dass der Floyd-Warshall im Vergleich zu dem Bellman-Ford trotz einer wesentlich höheren Laufzeitklasse ($O(n*m)$ für den Bellman-Ford und $O(n^3)$ für den Floyd-Warshall) nicht wesentlich höhere Zeiten benötigte. Das liegt daran, dass in genau diesem Fall die Anzahl der Knoten n immer kleiner gleich der Kanten m ist. Denn jeder Knoten hat in unserem Beispiel mindestens drei und maximal 8 Kanten. Für die Kantengröße m, in diesem Fall, nähert sich die Laufzeit des Bellman-Frod also der des Floyd-Warshall an.

Im letzten Abschnitt dieses Kapitels wird auf das *U-Sahpe* Problem hingewiesen. Da der A*-Algorithmus ein zielgerichteter Algorithmus ist, kann dieser sich in einem U-Förmigen Hinderniss *verlaufen*. Ist dies der Fall wird annähernd eine *worst-case* Laufzeit erreicht und der A* ist nicht mehr unbedingt schneller als der Dijkstra. Dies ist in Abbildung 5.9 veranschaulicht.

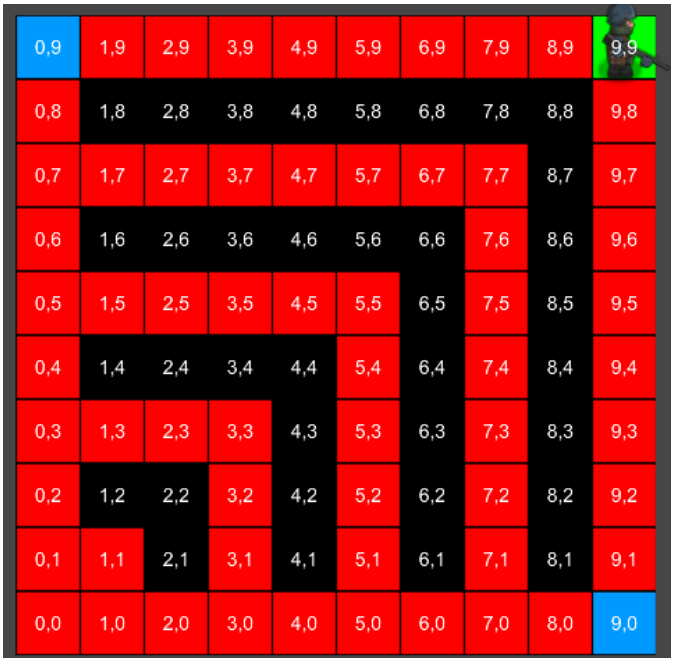

Abb. 5.9. Fall A*, fast alle Knoten müsssen betrachtet werden. Die Heuristik hat kaum Einfluss. Es wird eine Laufzeit erreicht die sich dem Dijkstra annähert. Quelle: Eigene Darstellung.

Innerhalb dieses Projektes hat das *U-Shape* Problem sogar dafür gesorgt, dass der Dijkstra Algorithmus eine um eine Millisekunde schnellere Laufzeit hatte. Beide Algorithmen sind mit der selben Liste implementiert. Dieser Unterschied ist als *Abweichung* hinzunehmen. Die Messungen wurden mit den immer gleichen Ergebnissen mehrfach durchgeführt und die angegebenen Laufzeiten über den Durchschnitt von mindestens zehn Versuchen gebildet.

6

Verifikation der Algorithmen

Die vorausgegangen Kapitel 3 und Kapitel 5 geben aufschlussreiche Informationen über die vier untersuchten Algorithmen, ihrer Laufzeit und ihrer Unterschiede. Um zu rechtfertigen, wo ein Algorithmus nützlich zum Einsatz kommt, reicht es nicht nur ein Attribut dieser zu betrachten. Die Anforderungen an die Lösung des Problemes sind genau so ausschlaggebend wie die Laufzeit. Im konkreten Fall des Projektes, in dem ein Graph das Gitternetz repräsentiert und die Kantenkosten alle gleich und fix sind, steht fest dass der A*-Algorithmus sich am Besten eignete. Das Problem *Hindernisse zu erschaffen* war mit jedem der 4 Algorithmen möglich. Sei es durch das nicht Betrachten des Knotens oder durch wiederholtes Setzen des $+\infty$ Wertes. Weil der Dijkstra und der A* aber die Suche beenden konnten sobald der nächste untersuchte Knoten der Zielknoten ist, haben diese zwei Algorithmen Laufzeitvorteile aufgewiesen. Dabei war der A* am effizientesten. Wegen seiner dynamischen Ansätze, IDA*/LPA* und D* eignet sich der A*-Algorithmus besonders gut für die Anwendung mit Graphen und Nav-Meshes (siehe Kapitel 4.5). Der Dijkstra belegt in dem Versuch den zweiten Platz, da er ebenfalls wie der A* funktioniert und mit einer Liste, aber ohne die Heuristik arbeitet. Auch der Dijkstra ist nicht schwieriger zu implementieren als der A*-Algorithmus. Im Bezug auf die Laufzeit und die Komplexität bei der Implementierung haben die beiden anderen Algorithmen Bellman-Ford und Floyd-Warshall ähnlich schlecht abgeschnitten. Hier gab es zusätzlich zur höheren Laufzeiten noch einige Komplikationen bei der Implementierung zu lösen. Beispielsweise musste aus der Kostenmatrix, die nur die Information der kürzesten Wege zwischen Knoten A und Knoten B enthält, eine Liste gefasst werden, um den kompletten Weg von Startknoten S zu Zielknoten T zu ermitteln, da der Floyd-Warshall-Algorithmus nur die Information für Knotenpaare speichert, denn er lößt nur das APSP. Auch das *herausnehmen der Knoten* war relativ aufwendig zu implementieren bei den Algorithmen Bellman-Ford und Floyd-Warshall.

Es bleibt dennoch zu beachten, dass das Ergebnis dieses Projektes keine Regel darstellt. Das heißt, ändern sich die Anforderungen, die durch das Problem entstehen, kann ein anderer Algorithmus sich besser eignen. Die beiden Algorithmen Dijkstra und A* können beispielsweise keine kürzesten Wege berechnen, wenn negative Kantenkosten existieren. Ein gutes Beispiel hierfür ist das *Stock-Marketing* (siehe Kapitel 4.4) oder aber in der Navigation, siehe 4.4. Würde in einem Spiel das Erklimmen eines Hanges mit einem höheren Aufwand verbunden werden und müsste dies mit einer negativen Kante dargestellt werden, so kann weder der Dijkstra noch der A* den Graphen berechnen und es müsste auf eine Alternative zurück gegriffen werden. So könnte allerdings modeliert werden, dass das Erklimmen des Hanges zwar mehr Aufwand in Anspruch nimmt, aber im Nachhinein jedoch den schnelleren Weg darstellt. In so einem Fall könnte auf den Bellman-Ford-Algorithmus nicht verzichtet werden. Auch der Floyd-Warshall hat seine Daseinsberechtigung, denn er kann relativ unkompliziert angewandt werden, wenn die Ausgangsdaten bereits in der Datenstruktur einer Matrix vorliegen. Das kommt häufig vor, wenn Punktkoordinaten verwendet werden und der kürzeste Weg zwischen diesen Punkten gesucht ist. Beispielsweise bei der rekonsturktion der Mantelfäche zwischen zwei korrespondierenden Schichten innerhalb von Volumendaten in der medizinischen Computergrafik (siehe Kapitel 4.3). In Netzwerken wiederum ist es wichtig den kompletten Graphen zu kennen. Obwohl der Dijkstra-Algorithmus beendet werden kann, sobald der Zielknoten gefunden wurde, macht es Sinn, ihn beim Routing zu verwenden, auch der Bellman-Ford-Algorithmus hat hier seine Vorteile (siehe Kapitel 4.1). Die Navigation für das Verkehrswesen ähnelt wieder dem Vorgehen einer AI oder KI in einem Computerspiel beim Beschreiten eines Weges, daher ist hier wieder der A*-Algorithmus sehr nützlich (siehe Kapitel 4.2).

Ebenfalls steht fest, dass die Abstraktion des Eingangsproblems so wie die Wahl der Implementierung eine Rolle spielt, wie die Lösung angegangen und wie effizient das Problem gelöst werden kann. Ein gut durchdachtes Preprocessing und die Vorsortierung durch eine Heap-Datenstruktur kann die Laufzeit beispielsweise weiter verringern. Große Graphen, die nicht vollständig oder nur teilweise verknüpft sind, können mit der Prüfung der Adjazenzmatrix schneller berechnet werden. Die endgültige Wahl des Algorithmus, der angewandt werden soll, bleibt fallabhängig und es gibt keine allgemeine Lösung zu dieser Wahl.

7

Zusammenfassung und Ausblick

Zusammengefasst kann gesagt werden, dass jeder der vier Algorithmen seine Daseinsberechtigung hat und nicht *out of date* ist. Die Erweiterungen und alternativen Implementierungen der Algorithmen zeigen, das der Grundsatz der jeweiligen Algorithmen nicht ersetzbar ist, dennoch kann weiter auf den Basisalgorithmen aufgebaut und diese verbessert werden. Diese Arbeit geht zunächst auf die Entstehung der Graphentheorie und die Grundlagen dieser ein. Die Aufgabe der Verifikation eines Algorithmus in seinem Teilgebiet wird formuliert (Einleitung und Problemstellung). Arbeiten, die maßgeblich zur Problemlösung und zum Verständnis der einzelnen Teilgebiete, auf denen die Pfadplanung angewandt wird, werden vorgestellt (Verwandte Arbeiten) bevor die untersuchten Algorithmen näher erläutert werden (Verwendete Algorithmen). Weiterhin gibt die Arbeit einen besseren Einblick in die einzelnen Teilgebiete und weist dabei auf die spezifische Problemstellung hin, die in dem jeweiligen Teilgebiet existiert (Anwendungsgebiete). Für die (Erhebung empirischer Daten) wurde im Umfang der Bachelorarbeit ein Projekt angelegt, das auf eines der Teilgebiete zugeschnitten, Ergebnisse liefert und präsentiert. Im vorletzten Kapitel wird das Ergebnis der Arbeit untersucht und Paralellen oder Differenzen zu anderen Problemen festgestellt (Verifikation der Algorithmen).

Wie in Kapitel 1.3 erwähnt, wurde innerhalb des Projektes nicht berücksichtigt, dass der ausführende Rechner während der Berechnung der Wege auch andere Kalkulationen ausführt. Um diesem Problem entgegenzuwirken, wurden zu einen alle nicht verwendeten Programme, auf die der Nutzer Einfluss nehmen kann, geschlossen oder zurückgestellt. Außerdem wurden die Abschnittsmessungen in Kapitel 5 mehrfach durchgeführt und der Durchschnittswert gebildet. In Einzelfällen wich das Ergebnis dem Erwartungswert sehr stark ab, deshalb wurden einige Messungen verworfen.

Die aus dem Projekt gewonnenen Ergebnisse sind nur teilweise zufriedenstellend. Beispielsweise hatten in einigen Fällen, wie schon in Kapitel 5.3 erwähnt, die Algorithmen von Bellman-Ford und Floyd-Warshall nahezu identische Laufzeiten. Das liegt zum einen daran, dass die Knotenmenge der durchgeführten Berechnungen doch relativ gering ist und zum anderen die Kantenmenge gegenüber der Knotenmenge um ein Vielfaches höher ist.

Ein weiteres Problem, das sich auffallend auf die Laufzeit auswirkte, ist die Verwendung von Funktionen, die die C# Bibliothek bereitstellt. Beispielsweise war die Implementierung des Bellman-Ford-Algorithmus unter Verwendung der *foreach schleife* erheblich langsamer als das anlegen einer Liste und der Zugriff per Index. Deshalb muss ebenfalls berücksichtigt werden, dass die Algorithmen unterschiedlich implementiert werden mussten. Da die Algorithmen auf Unterschiedliche weise die Kosten für gerades und diagonales Laufen auf dem Spielfeld bestimmen, ist auch dies ein weiterer Störfaktor, der sich nicht einfach vermeiden ließ. Zudem nimmt die visuelle Veranschaulichung, die zur Beleuchtung des *U-Sahpe Problemes* implementiert wurde, ebenfalls Berechnungszeit in Anspruch. Das Programm rechnet signifikant länger als die Zeit die es zur Berechnung ausgibt. Die ermittelte Zeit ist trotz des bekannten Problemes korrekt, durch das überdachte setzen der *Stopwatch Funktion* ist die Zeit, die benötigt wird, um die visuelle Verarbeitung bereit zu stellen, nicht mit in die Messung eingeflossen.

Das Projekt ist abgeschlossen und das Ergebnis steht fest, doch können in Zukunft weitere Projekte in anderen Teilgebieten angestrebt werden, um die dort verwendeten Algorithmen zu verifizieren. Die zusammengetragenen Informationen weisen jedoch darauf hin, dass es zwar möglich ist, eine Alternative innerhalb eines Teilgebietes zu finden, die nicht wesentlich schlechter oder gleich effizient ist, dennoch gibt es dafür keinen offensichtlichen Grund. Möglicherweise geben zukünftige Verbesserungen wie die des A* auf die Erweiterung D* Anlass, in Zukunft eine Untersuchung zu wiederholen. Eine alternative Implementierung der einzelnen Algorithmen kann ebenfalls lohnend in Betracht gezogen werden, um eventuelle Störfaktoren zu beheben.

Literaturverzeichnis

Bac04. Bachmann, Paul: *Analytische Zahlentheorie*. In: Meyer, Wilhelm Franz (Herausgeber): *Encyklopädie der Mathematischen Wissenschaften mit Einschluss ihrer Anwendungen*, Seiten 636–674. Viweg+Teubner Verlag, Wiesbaden, 1904.

BCL96. Bajaj, Chandrajit L, Edward J Coyle und Kwun-Nan Lin: *Arbitrary topology shape reconstruction from planar cross sections*. Graphical models and image processing, 58(6):524–543, 1996.

Bel58. Bellman, Richard: *On a routing problem*. Quarterly of applied mathematics, 16(1):87–90, 1958.

BG93. Boissonnat, Jean-Daniel und Bernhard Geiger: *Three-dimensional reconstruction of complex shapes based on the Delaunay triangulation*. In: *Biomedical Image Processing and Biomedical Visualization*, Band 1905, Seiten 964–975. International Society for Optics and Photonics, 1993.

BNMB20. Bök, Patrick-Benjamin, Andreas Noack, Marcel Müller und Daniel Behnke: *Computernetze und Internet of Things*. Springer Fachmedien Wiesbaden, Wiesbaden, 2020.

Bri20. Briskorn, Dirk: *Graphentheorie*. In: Briskorn, Dirk (Herausgeber): *Operations Research*, Band 43, Seiten 291–401. Springer Berlin Heidelberg, Berlin, Heidelberg, 2020.

CLRS09. Cormen, Thomas, Charles Leiserson, Ronald Rivest und Clifford Stein: *Introduction to Algorithms*. The MIT Press, 2009. Third Edition.

CS11. Cui, Xiao und Hao Shi: *A*-based pathfinding in modern computer games*. International Journal of Computer Science and Network Security, 11(1):125–130, 2011.

D+59. Dijkstra, Edsger W et al.: *A note on two problems in connexion with graphs*. Numerische Mathematik, 1(1):269–271, 1959.

DI19. Dan Informatika, Sekolah Teknik Elektro: *Identifying Market Arbitrage Opportunity using Pathfinding Algorithm*. Institut Teknologi Bandung Indonesia, 2019.

DK03. Duckham, Matt und Lars Kulik: *"Simplest" paths: automated route selection for navigation.* In: *International Conference on Spatial Information Theory*, Seiten 169–185. Springer, 2003.

DMS14. Dietzfelbinger, Martin, Kurt Mehlhorn und Peter Sanders: *Algorithmen und Datenstrukturen.* Springer Verlag, 2014. Kapitel 10: Kürzeste Wege.

Dom73. Domschke, Wolfgang: *Zwei Verfahren zur Suche negativer Zyklen in bewerteten Digraphen.* Computing, 11(2):125–136, 1973.

Dör17. Dörn, Sebastian: *Programmieren für Ingenieure und Naturwissenschaftler.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

DPV06. Dasgupta, Sanjoy, CH Papadimitriou und UV Vazirani: *Algorithms.* McGraw-Hill Education (book), 2006.

Ert13. Ertel, Wolfgang: *Grundkurs Künstliche Intelligenz.* Springer Fachmedien Wiesbaden, Wiesbaden, 2013.

FKU77. Fuchs, Henry, Zvi M Kedem und Samuel P Uselton: *Optimal surface reconstruction from planar contours.* Communications of the ACM, 20(10):693–702, 1977.

Flo62. Floyd, Robert W: *Algorithm 97: shortest path.* Communications of the ACM, 5(6):345, 1962.

Hei08. Heiko, Waldschmidt: *Vergleich von Pathfinding-Algorithmen.* Universität Kassel, 2008.

Hil16. Hilbig, André: *Kompetenzen in der Informatik zur Prävention von Cybermobbing.* Springer Fachmedien Wiesbaden, Wiesbaden, 2016.

Hof13. Hofmann, Andreas: *Pathfinding-Algorithmen in verschiedenen Spielegenres.* Fachhochschule Technikum Wien, 2013.

Hou10. Hougardy, Stefan: *The Floyd–Warshall algorithm on graphs with negative cycles.* Information Processing Letters, 110(8-9):279–281, 2010.

HP66. Hart Peter, Nilsson Nils, Raphael Bertram: *A formal basis for the heuristic determination of minimum cost paths.* IEEE Transactions of Science and Cybernetics, 1966.

KL02. Koenig, Sven und Maxim Likhachev: *D^* lite.* American Association for Artificial Intelligence, 2002.

KN12. Krumke, Sven Oliver und Hartmut Noltemeier: *Graphentheoretische Konzepte und Algorithmen.* Vieweg+Teubner Verlag, Wiesbaden, 2012.

Kne19. Knebl, Helmut: *Algorithmen und Datenstrukturen.* Springer Fachmedien Wiesbaden, Wiesbaden, 2019.

Kru56. Kruskal, Joseph B.: *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* Proceedings of the American Mathematical Society, 7(1):48, 1956.

Lab09. Labitzke, Björn: *Erzeugung von Tetraedergittern f ur die Volumendeformation.* Universität Siegen, 2009.

Lan16. Lange, Benjamin: *Übersicht über die Pfadplanung.* Universität Koblenz-Landau, 2016.

MH05. Martin Helmreich, Sebastian Prokop, Chris Schwemmer: *Algorithmen zur Pfadplanung*. Universität Erlangen, 2005.

MJS20. Mahajan, Pranit, Vinayak Jadhav und Yagnesh Salian: *Currency Arbitrage Detection*. International Journal of Innovative Science and Research Technology, 2020.

MSS92. Meyers, David, Shelley Skinner und Kenneth Sloan: *Surfaces from contours*. ACM Transactions On Graphics (TOG), 11(3):228–258, 1992.

OW12. Ottmann, Thomas und Peter Widmayer: *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, Heidelberg, 2012.

PB13. Preim, Bernhard und Charl P Botha: *Visual computing for medicine: theory, algorithms, and applications*. Newnes, 2013.

PE18. Priese, Lutz und Katrin Erk: *Theoretische Informatik*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

Ros96. Rose, Martin: *Kritische Analyse von Routensuchverfahren*. Institut für Verkehrswirtschaft, Straßenwesen und Städtebau, Universität Hannover, 1996.

Sch09. Schubert, Matthias: *Mathematik für Informatiker*. Vieweg+Teubner, Wiesbaden, 2009.

SM01. Sivabalan, M und HT Mouftah: *On the Design of Link-State Routing Protocol for Connection-Oriented Networks*. Journal of Network and Systems Management, 9(2):223–242, 2001.

Ste97. Stentz, Anthony: *Optimal and efficient path planning for partially known environments*. In: *Intelligent Unmanned Ground Vehicles*, Seiten 203–220. Springer, 1997.

War62. Warshall, Stephen: *A theorem on boolean matrices*. Journal of the ACM (JACM), 9(1):11–12, 1962.

WW13. Weicker, Karsten und Nicole Weicker: *Algorithmen und Datenstrukturen*. Springer Fachmedien Wiesbaden, Wiesbaden, 2013.

A

Glossar

SSP	Shortest Path Problem
SSSPP	Single Scource Shortest Path Problem
SSSP	Single Scource Shortest Path
SDSPP	Single Destination Shortest Path Problem
SDSP	Single Destination Shortest Path
APSPP	All Pairs Shortest Path Problem
APSP	All Pairs Shortest Path
BF	Bellman-Ford
P2P	Peer-to-Peer
IRP	Interior Gateway Protocol
ERP	Exterior Gateway Protocol
GGP	Gateway to Gateway Protocol
RIP	Routing Information Protocol
IGRP	Interior Gateway Routing Protocol
EIRGP	Enhanced Interior Gateway Routing Protocol
IS-IS	Intermediate System to Intermediate System Protocol
OSPF	Open Shortest Path First
TSP	Traveling Salesman Problem
MR	Magnet Resonanz
MRT	Magnet Resonanz Tomographie
CISS	Centre for Intelligence and Security Studies
GPU	Graphics Processing Unit
AI	Artificial Intelligence
KI	Künstliche Intelligenz
NPC	Non-Player-Charakter
IDA	Iterative Deepening A*
LPA	Lifelong Planing A*

B

Erklärung der Kandidatin / des Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

Datum	Unterschrift der Kandidatin / des Kandidaten