INSTITUT FÜR INFORMATIK
Fachbereich Informatik und Mathematik

Masterarbeit

Suche nach schnellsten Wegen in einem Verkehrsverbund mit verschiedenen Verkehrsmitteln unter Verwendung der A^*-Methode

Christos Terzis

Abgabe am 29.09.2016

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz und Softwaretechnologie

Erklärung

gemäß Ordnung für den Masterstudiengang Informatik §24 Abs.12

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Frankfurt am Main, 29.09.2016 Christos Terzis

Danksagung

Zunächst möchte ich mich an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben. Die Arbeit wäre ohne die Unterstützung und den Rat einer Vielzahl von Menschen nicht möglich gewesen.

Ganz besonders gilt dieser Dank Herrn Prof. Dr. Manfred Schmidt-Schauß, der meine Arbeit und somit auch mich betreut hat. Durch stetig kritisches Hinterfragen und konstruktive Kritik gab er mir wertvolle Hinweise mit auf dem Weg. Er stand nicht nur mit wissenschaftlichem Rat und fachlichen Diskussionen zur Seite, sondern auch mit moralischer Unterstützung und Motivation. Er hat mich stets sehr gut betreut und dazu gebracht, über meine Grenzen hinaus zu denken.
Vielen Dank für die Geduld und Mühen.

Ich danke dem Institut für Informatik für die Möglichkeit, diese Arbeit am Lehrstuhl für Künstliche Intelligenz und Softwaretechnologie an der Goethe-Universität anfertigen zu können.

Nicht zuletzt gebührt meinen Eltern und meinem Bruder Dank, die während meines Lebens und vor allem während des Studiums emotional immer an meiner Seite standen.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

LISTINGS

TABELLENVERZEICHNIS

KAPITEL
EINS

Einleitung

1.1 Aufgabenstellung

In dieser Arbeit befassen wir uns mit dem folgenden Suchproblem:
Es ist ein zusammenhängender Graph (gerichtet oder ungerichtet) mit m Knoten und n Kanten $c_1, \ldots, c_n$ mit nicht negativen Kantengewichten, ein Startknoten S, ein Zielknoten Z und eine *heuristische Funktion* h gegeben, die die Kosten von einem bereits erreichten Knoten N bis zum nächsten Zielknoten abschätzt. In dieser Suche wird jeder Kante c_i Kosten von c zugewiesen. Der Pfad vom Startknoten bis zum Zielknoten über die besuchten Knoten mit den Zeiten bildet die Dauer der Suche. Das Ziel ist es, einen Weg zu finden, sodass die maximale Gesamtzeit über die Knoten, also vom Startknoten zum Zielknoten, minimiert wird.
Ein einfacher Suchalgorithmus ist der A^*-Algorithmus, der oft in Routenplaner oder Computerspielen Anwendung findet. Der A^*-Algorithmus basiert auf drei Komponenten, nämlich g, h und f. $g(n)$ bezeichnet eine Bewertung für die Zustände, wobei $g(n)$ die bisher geringsten Kosten zur Erreichung des Zustands n angibt. $h(n)$ ist eine heuristische Funktion zur Schätzung der Restkosten von einem bereits erreichten Knoten N bis zum nächsten Zielknoten. Die letzte Komponente f ist eine Bewertungsfunktion $f(n) = g(n)+h(n)$, die zur Auswahl des zu expandierenden Zustandes dient.
Der A^*-Algorithmus expandiert dann den Knoten, der die geringste Bewertung $f(n)$ besitzt. Dabei werden die Knoten in zwei Mengen, der *Open*-Menge, d.h. eine Liste von allen Positionen direkt adjazent zu Gebieten, die bereits erkundet wurden und der *Closed*-Menge, dem Stammsatz aller Positionen, die vom Algorithmus untersucht wurde, gespeichert. Der A^*-Algorithmus besteht aus zwei Variationen, dem Baum-Such-Verfahren und dem Graph-Such-Verfahren abhängig von den Eigenschaften von h, die sich im Aktualisierung des minimalen Weges bis zu einem Knoten während des Ablaufs unterscheiden. Beim Baum-Such-Verfahren findet keine Aktualisierung statt und besitzt keine Menge expandierender Knoten *(Closed-List)* und somit kann dann hier der gleiche Knoten mehrfach in der *Open*-Menge mit verschiedenen Wegen stehen.
Bei Graphsuche hat man immer den aktuell bekannten minimalen Weg, und somit den Knoten nur einmal in einer Liste untersuchender Knoten *(Open-List)*. Wir be-

trachten die Graphsuche.
Man unterscheidet zwischen zwei Eigenschaften von Heuristiken beim A^*-Algorithmus, die der Unterschätzung und die der Monotonie. Eine heuristische Funktion ist unterschätzend, wenn sie die Kosten unterschätzt, d.h. wenn für jeden Knoten $N : h(N) \leqslant c^*(N)$ gilt, wobei c die Kosten des optimalen Weges von N bis zum nächsten Zielknoten Z ist. Für die Monotonie muss der Fall der Unterschätzung gelten und dass die geschätzten Kosten von einem Knoten k nicht größer sein dürfen als die tatsächlichen Kosten zu einem Nachfolgeknoten k' plus die geschätzten Kosten dieses Knotens. Die Monotonie ist somit eine stärkere Eigenschaft als die Unterschätzung. Sie unterscheiden sich nur in dem Punkt, dass ein bereits expandierter Knoten mit einer niedrigeren Bewertung $f(n)$ ein zweites Mal während der Suche erreicht wird, bei einem A^*-Algorithmus mit Monotoniebedingung jedoch niemals auftreten kann. Der A^*-Algorithmus ist mit monotoner Heuristik unter Verwendung von Graphsuche optimal. Eine passende und einfache Heuristik für eine Suche hierbei wäre die Luftlinie. Die tatsächliche Entfernung ist nie kürzer als die direkte Verbindung.
Wir interessieren uns für die Suche nach den schnellsten Wegen in einem Verkehrsverbund mit verschiedenen Verkehrsmitteln unter Verwendung der A^*-Methode. Hierbei geht es darum, dass jeder Knoten aus Bahnhöfen und Flughäfen in Deutschland besteht, die wiederum in drei Arten von Kanten eingeteilt werden: 1. langsame, die mit dem Auto zurückgelegt werden, 2. schnelle, die mit dem Flugzeug zurückgelegt werden, und 3. mittelschnelle Kanten, auf denen eine Bahn fährt.
Die Verkehrsmittel, die zur Verfügung stehen, wären dann somit das Auto (Geschwindigkeit: v_A), die Bahn (Geschwindigkeit: v_B) und das Flugzeug (Geschwindigkeit: v_F). Ein Start- und Endknoten wird als Eingabe für den Algorithmus dienen. Das Ziel des Algorithmus ist es dann hierfür den schnellsten Weg bei Eingabe von zwei Knoten A und B zu finden, indem man eine gute monotone Schätzfunktion wählt. Wir werden später versuchen nachzuweisen, dass die heuristische Funktion, die wir für den Algorithmus entwickelt haben, unterschätzend und monoton ist.
Ein großer Bestandteil meiner Arbeit behandelt die Implementierung einer bestimmten Variante des A^*-Algorithmus, dessen heuristische Funktion unterschätzend und monoton ist. Dieser A^*-Algorithmus unterscheidet sich vom einfachen A^*-Algorithmus in der Art der Gewichtsfunktion, also sprich z.B. der kürzeste oder schnellste Weg zwischen zwei Knoten in einem Graphen. Wir betrachten den schnellsten Weg als Gewichtsfunktion. Unsere Gewichtsfunktion hat das Ziel, die schnellste Verbindung bei Eingabe eines Start- und Endknotens zu suchen. Wir untersuchen experimentell die erwartete Zeit des A^*-Algorithmus auf verschiedenen deterministisch generierten Instanzen. Dabei werden verschiedene Parameter für die Erstellung der Eingabeinstanz getestet. Weiterhin werden wir die Berechnungszeiten verschiedener Heuristiken und Anzahl expandierender Knoten ermitteln und mit einer Suche von Bahn/Flugzeug und Auto/Bahn/Flugzeug vergleichen. Dabei ist unsere Aufgabe zu achten, welche Heuristik die schnellste Laufzeit besitzt. Aufgrund von bereits untersuchten Experimenten wollen wir die Hypothese stützen, dass die Nullheuristik (Dijkstra-Algorithmus) bei der Suche nach dem schnellsten Weg langsam ist und die meisten Knoten im Vergleich zu allen anderen Heuristiken expandiert.
Hierfür werden die dabei entstandenen Ergebnisse ausgewertet und analysiert.

1.2 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in folgende Kapitel:

- Kapitel 2 befasst sich mit dem Grundlagenwissen, das in den folgenden Kapiteln vorausgesetzt wird. Wichtige Begriffe und Ergebnisse zu den Themen Suchproblem und Suchstrategien werden beschrieben.
- Kapitel 3 stellt informierte Suchstrategien vor und beschreibt verschiedene Algorithmen für kürzeste Wege und Heuristiken.
- Kapitel 4 handelt vom A^*-Algorithmus. Wir befassen uns mit der Idee und den Eigenschaften dieses Algorithmus.
- Kapitel 5 beschreibt die spezielle Variante des untersuchten Algorithmus. Wir erklären die untersuchten Heuristiken und gehen auf ihre Eigenschaften näher ein. Dann stellen wir Verwendung und bekannte Ergebnisse des speziellen A^*-Algorithmus vor.
- Kapitel 6 beschreibt die Versuche und das Ziel der Arbeit.
- Kapitel 7 befasst sich mit der Implementierung von der Suche nach schnellsten Wegen in einem Verkehrsverbund mit verschiedenen Verkehrsmitteln unter Verwendung des A^*-Algorithmus. Es wird genauer auf die Entwicklung der einzelnen Komponenten und auf die verwendeten Eingabeinstanzen eingegangen.
- Kapitel 8 stellt die experimentellen Ergebnisse der Variante des A^*-Algorithmus dar und vergleicht verschiedene Fälle und verändere Parameter miteinander.
- Kapitel 9 gibt eine Zusammenfassung und eine Diskussion der erarbeiteten Ergebnisse dieser Arbeit wieder.

KAPITEL
ZWEI

Grundlagen

2.1 Das Suchproblem

Wir betrachten das Problem, in dem ein zusammenhängender Graph (gerichtet oder ungerichtet) mit m Knoten und n Kanten $T_1, \ldots, T_n$ mit positiven Kantengewichten $c_1, ...c_n$, ein Startknoten S, ein Zielknoten Z und eine *heuristische Funktion* h gegeben sind. Die heuristische Funktion schätzt die Kosten von einem bereits erreichten Knoten N bis zum nächsten Zielknoten ab. In dieser Suche wird jeder Kante von zwei Knoten eine Kostenfunktion c zugewiesen, die angibt, wie groß die Entfernung zwischen diesen Punkten ist. Wir suchen einen Weg, um die maximale Gesamtzeit über die Knoten, also die Zeit, um vom Startknoten zum Zielknoten zu gelangen, zu minimieren. Dieses Suchproblem ist eine kleine Änderung zum klassischen Suchproblem, bei dem es darum geht, den kürzesten Weg von einem Startknoten zu einem Zielknoten zu finden.
Das klassische Suchproblem kann mithilfe effizienter Implementierung von Suchalgorithmen gelöst werden. Somit kann man einen effizienten Algorithmus erwarten, der optimal ist, d.h., dass die Lösung des Suchproblems der kürzeste Pfad zum Zielknoten ist. Bevor wir näher auf den effizienten Algorithmus eingehen, werden wir nun genauer die Definitionen von Problemen und ihren Lösungen mithilfe von Beispielen erläutern.

2.2 Definition eines Problems und Lösung

Ein Problem ist eine Ansammlung von Informationen, die ein Agent verwenden wird um zu entscheiden, was zu tun ist [RN95]. Ein Problem kann man in fünf Bestandteile zerlegen:

- einem Startzustand, in dem ein Agent anfängt.
- einer Menge von möglichen Aktionen, die einem Agenten bereitstehen.
- einem Überführungsmodell, das beschreibt, was eine Aktion erzeugt.

Zusammen definieren diese den Zustandsraum des Problems - die Menge aller Zustände, die von diesem Startzustand aus durch eine Folge von Aktionen erreichbar

sind. Der Zustandsraum erzeugt ein gerichtetes Netz oder Graphen, bei dem die Knoten Zustände sind und die Verbindungen zwischen den Knoten die Aktionen repräsentieren. Ein Pfad im Zustandsraum ist schlicht eine Abfolge von Aktionen, die einen Zustand in einen anderen überführt.

- Ein Zieltest, welcher beurteilt, ob ein bestimmter Zustand ein Zielzustand ist. Manchmal gibt es eine explizite Menge von möglichen Zielzuständen und der Test überprüft einfach, ob einer von ihnen erreicht worden ist. Hin und wieder wird das Ziel nicht als explizite Auflistung einer Menge von Zuständen spezifiziert, sondern als abstrakte Eigenschaft. Beispielsweise ist beim Schachspiel das Ziel, einen Zustand namens „Schachmatt“ zu erreichen, bei dem der gegnerische König beim nächsten Zug gefangen genommen werden kann - unabhängig davon, was der Gegner tut.

- eine Pfadkostenfunktion, die einem Pfad Kosten zuteilt. Die Kosten eines Pfades sind die Summe der Kosten der einzelnen Aktionen entlang des Pfades. Die Pfadkostenfunktion wird oft als g bezeichnet.

Zusammen definieren der Startzustand, die Menge der möglichen bereitstehenden Aktionen, der Zieltest und die Pfadkostenfunktion ein Suchproblem. Die Ausgabe eines Suchalgorithmus ist die Lösung für ein Problem, ein Pfad vom Startzustand zu einem Zustand, die den Zieltest erfüllt. Die Güte der Lösung wird anhand der Pfadkostenfunktion bewertet und eine optimale Lösung besitzt die geringsten Pfadkosten aller Lösungen [RN04].

2.3 Beispielprobleme

Der Umfang von Aufgabenumgebungen, die mit Problemen charakterisiert werden können ist enorm. Man kann zwischen Spielproblemen, die als Veranschaulichung oder Anwendung verschiedener problemlösender Methoden gedacht sind und Problemen der realen Welt, die dazu neigen viel schwerer zu sein und dessen Lösung tatsächlich wichtig ist, unterscheiden. In diesem Abschnitt werden wir Beispiele von beiden Problemen anführen. Spielprobleme können in einer kurzgefassten und exakten Beschreibung angegeben werden. Dies bedeutet, das diese leicht von verschiedenen Forschern verwendet werden können um die Leistung von Algorithmen zu vergleichen. Probleme der realen Welt hingegen neigen nicht zu einer vereinheitlichten, abgestimmten Beschreibung, aber es ist möglich, eine Annäherung der Formulierung zu skizzieren [RN04].

2.3.1 Das 8-Damen-Problem

Es geht um das folgende Problem: Wir haben 8 Damen und ein 8 x 8 Schachbrett mit abwechselnd schwarzen und weißen Quadraten. Die Damen werden auf dem Schachbrett gestellt. Jede Dame kann jede andere Dame, die auf der selben Zeile, Spalte oder Diagonalen platziert ist, angreifen. Das Ziel ist es die passende Platzierung der Damen auf dem Schachbrett so zu finden, dass keine Dame eine andere Dame angreifen kann. Abbildung 2.1 stellt einen Lösungsversuch dar, der mißlingt: Die Dame in der ganz rechten Spalte wird durch die Dame ganz oben links angegriffen [Kum08].

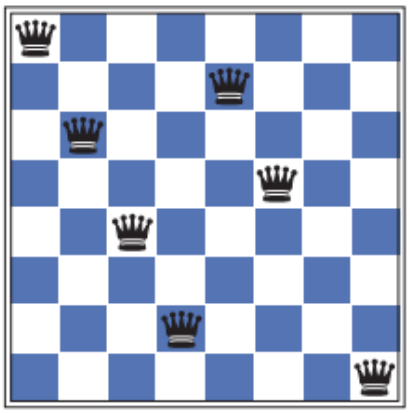


Abbildung 2.1: Ein möglicher Versuch einer Lösung für das 8-Damen-Problem [1]

Während der letzten drei Jahrzehnte wurde das Problem wie auch die ganze n-Damen-Familien-Problem im Kontext der Informatik diskutiert und wird als ein Beispiel für Backtrack Algorithmen, Generieren von Permutationen, Divide und Conquer Paradigmen, Methoden der Programmentwicklung, Bedingungserfüllungsprobleme, ganzzahlige Programmierung, Spezifikation und neuronale Netze verwendet [SD07].
Wir unterscheiden vorwiegend zwei Arten der Formulierung. Die inkrementelle Formulierung umfasst platzierende Damen nacheinander, wohingegen die vollständige Formulierung mit allen 8 Damen auf dem Brett beginnt und diese herumzieht. In beiden Fällen sind die Pfadkosten nicht von Interesse, weil nur der Endzustand zählt. Algorithmen vergleichen deshalb nur Suchkosten. Eine inkrementelle Formulierung könnte folgendermaßen aussehen:

- Zustände: Eine beliebige Anordnung von 0 bis 8 Damen auf dem Brett
- Ausgangszustand: keine Dame auf dem Brett
- Aktion: Hinzufügen einer Dame zu einem beliebigen Quadrat
- Überführungsmodell: gibt das Brett mit einer auf das spezifizierte Quadrat hinzugefügten Dame zurück

[1] Die Quelle für die Abbildungen ist `https://www.pearson-studium.de/kunstliche-intelligenz.html`

- Zieltest: 8 Damen auf dem Brett und keine davon wird angegriffen.
- Pfadkosten: 0

In dieser Formulierung müssen wir 64^8 mögliche Möglichkeiten untersuchen. Eine sinnvollere Wahl würde die Tatsache benutzen eine Damen genau dort nicht zu platzieren, wo bereits eine attackiert worden ist. Somit könnten wir folgendes probieren:

- Zustände: Anordnungen von 0 bis 8 Damen wobei keine Dame eine andere angreift.
- Aktionen: Platzieren einer Dame in der am äußerst links liegenden freien Spalte, sodass diese nicht durch eine andere Dame angegriffen wird.

Mit dieser Formulierung verkleinert sich der 8-Damen-Zustandsraum von 64^8 auf 2057 und die Lösungen lassen sich einfacher erörtern. Das ist eine enorme Veränderung, aber immer noch nicht zufriedenstellend, um das Problem brauchbar zu machen [RN95].
Eine übliche Methode für das Lösen dieses Problems besteht in dem Versuch, die Damen auf dem Brett im rechten Winkel stehend nacheinander zu legen. Falls eine Dame die neue eingeführte Dame bedroht, ziehen wir die Dame zurück und suchen nach einer anderen Position. Falls wir keine Lösung finden können, wählen wir eine Dame zum Entfernen, die bereits positioniert ist, weisen ihr eine andere Position zu, die bis jetzt noch nicht verwendet wurde und starten die Suche erneut. Die letzte Operation nennt man Backtrack, und die ganze Strategie bezeichnet man als ein Trial-and-Error-Algorithmus. Für diese Formulierung reduziert sich der 8-Damen-Zustandsraum von 2057 auf nur 92 und das Millionen-Damen-Problem lässt sich mit Leichtigkeit lösen [SD07].

2.3.2 Routensuche-Problem

Unter dem Routensuche-Problem versteht man die Berechnung und die Ausgabe eines Weges von einem Startpunkt zu einem Zielpunkt [Ern03]. Algorithmen für die Routensuche werden in einer Vielzahl von Anwendungen verwendet, wie beispielsweise Routenplanung in Computernetzwerken, automatische Reiseberatungssysteme und Planungssysteme für den Flugreiseverkehr. Im Folgenden betrachten wir als Beispiel die Probleme des Flugverkehrs, die von einer Webseite für die Reiseplanung erledigt werden müssen:

- Zustände: Jeder Zustand besteht aus einer Position (z.B. ein Flughafen) und einer aktuellen Zeit. Dieser Zustand muss zusätzliche Informationen über diese historischen Hinsichten erfassen, weil auch die Kosten einer Aktion (eines Flugabschnittes) von vorhergehenden Abschnitten, ihren Flugtarifen und ihrem Status als In- oder Auslandsflüge bedingt sein können.
- Ausgangszustand: wird durch die Abfrage des Benutzers genau bestimmt.
- Aktionen: Wähle einen willkürlichen Flug vom derzeitigen Ort in irgendeiner Klasse aus, welcher nach der momentanen Zeit startet und bei Notwendigkeit noch reichlich Zeit für die Beförderung innerhalb des Flughafens lässt.

- Übergangsmodell: Der Zustand nach Durchführung des Fluges enthält als aktuelle Position das Ziel des Fluges und als aktuelle Zeit die Ankunftszeit des Fluges.
- Zieltest: Erreichen wir innerhalb einer vom Nutzer vorgegebenen Zeit das Ziel?
- Pfadkosten: Diese sind von der Flugzeit, der Wartezeit, dem Flugpreis, Flugzeugtyp usw. abhängig.

Aufgrund von schwer durchsichtigen Tarifstrukturen von Fluggesellschaften benutzen geschäftliche Flugbuchungssysteme eine Problemformulierung solcher Art. Vielen Reisenden ist jedoch bewusst, dass nicht alle Flüge nach Plan verkehren. Die Lösung wäre ein System, das Alternativpläne berücksichtigt, wie z.B. Sicherungsreservierungen auf alternativen Flügen und zwar in einem Umfang, dass diese gemäß den Kosten und der Ausfallwahrscheinlichkeit des ursprünglichen Fluges begründet erscheinen lassen [RN04].

2.3.3 Traveling Salesman Problem

Eines der bekanntesten Problemstellungen in der kombinatorischen Optimierung der Informatik ist das Traveling Salesman Problem (TSP)(Problem des Handlungsreisenden). Darin soll ein Handlungsreisender eine gewisse Anzahl erreichbarer Städte so miteinander verbinden, dass die abgefahrene Gesamtstrecke die minimalste überhaupt ist. Der Handlungsreisende startet seine Fahrt an seinem Ausgangsort, besucht die anderen Orte genau einmal und gelangt dann wieder zu seinem Ausgangsort zurück [Obe07]. Das Problem wird als NP-hart bezeichnet, d.h. dieses Problem ist mindestens so schwer lösbar wie NP-vollständige Probleme und somit vermutlich nicht effizient lösbar. Somit kann man keinen effizienten Algorithmus erwarten, der für jede Instanz eine optimale Lösung ausgibt [Böl13]. Trotzdem wurde viel Arbeit betrieben, um die Fähigkeiten von TSP-Algorithmen zu verbessern. Diese Algorithmen werden nicht nur zum Lösen von Reiseplanung für Handelsreisende verwendet, sondern z.B. auch zur Optimierung der Fahrwege von Regalbediengeräten in Lagerhäusern.

2.4 Pathfinding

Alle Pathfinding-Algorithmen haben gemein, dass sie einen Graphen verwenden und darin kürzeste Wege prüfen. Die folgenden Unterabschnitten erklären Grundlagen zur Graphentheorie und Lösungen für die Suche.

2.4.1 Graphentheorie

Der Graph stellt ein Paar (V, E) endlicher Mengen mit $V \cap E = \emptyset$ dar. Dabei ist V die Menge der im Graph enthaltener Knoten (engl. vertex) und E die Menge der Kanten (engl. edge). Jede Kante $(K1, K2)$ stellt eine Verbindung zwischen den zwei Knoten $K1$ und $K2$ dar. Man unterscheidet zwischen gerichteten und ungerichteten Graphen. Ein gerichteter Graph besteht aus einer Menge E von gerichteten Kanten (mit Kantenmenge E, welches eine binäre Relation auf V, d.h. $E \subseteq V \times V$ ist;

Kante $(u, v) \in E$ ist von u nach v gerichtet, symbolisch $u \to v$; u heißt Kopf (engl. head), v heißt Ende (engl. tail)). Ein ungerichteter Graph ist ein Paar $G = (V, E)$, bestehend aus Kantenmenge E und ist eine symmetrische binäre Relation auf V, d.h. $(u, v) \in E \Leftrightarrow (v, u) \in E$ [Kos]. In der Arbeit werden wir uns primär auf den gerichteten Graphen beziehen. Bei einem gewichteten Graphen sind alle Kanten mit Kosten bzw. Zahlenwerte versehen. Der Wert repräsentiert vereinfacht gesagt die Distanz zwischen den verbundenen Knoten oder auch die Zeit, die es dauert von Knoten A nach Knoten B zu gelangen oder aber eine Kombination dieser Faktoren. Bei einem ungerichteten Graphen ist die Richtung für die Kosten irrelevant. Ein Weg bezeichnet eine Folge von Kanten, die von einem Startknoten S zu einem Zielknoten G führen. Die Weg-Kosten (S, G) sind die Summe der Kosten aller Kanten eines Weges. Kürzeste Wege von S nach G sind die Wege, für die gilt, dass es keinen Weg von S nach G mit geringeren Kosten gibt [GRRW10]. Pathfinding-Algorithmen suchen den kürzesten Weg.

2.4.2 Suche nach Lösungen

Wir haben gesehen, wie man ein Problem definiert. Jetzt müssen wir die Probleme lösen. Durch die Verwendung eines Operators wird aus einem gegebenen Zustand eine Menge anderer Zustände hervorgerufen. Diesen Ablauf nennt man Expansion eines Zustandes. Der Kern der Suche besteht darin, einen Zustand aus einer Menge auszuwählen und die anderen für einen eventuell späteren Einsatz aufzuschieben, wie etwa dann, wenn die getroffene Auswahl nicht zum Erfolg führt. Die Wahl des als nächsten zu expandierenden Zustands wird durch eine Suchstrategie definiert. Der Suchablauf kann als Aufstellung eines Suchbaums, der über den Zustandsraum gelegt wird, angenommen werden. Der Anfangszustand entspricht der Wurzel des Suchbaums, nämlich dem Suchknoten. Die Blätter des Baums sind Zustände, die keine Nachfolger im Baum besitzen, entweder weil sie keine hervorbringen oder sie noch nicht expandiert wurden. In jedem Schritt wird ein Blattknoten zur Expansion vom Suchalgorithmus bestimmt [Chea].

Datenstrukturen für Suchbäume und allgemeiner Suchalgorithmus

Ein Knoten v eines Suchbaums stellt eine Datenstruktur mit fünf Komponenten dar [Chea]:

- Der Zustand des Zustandsraums, dem v gleicht;
- der Knoten im Suchbaum, der diesen Knoten generiert hat (der Vaterknoten von v);
- dem Operator, der angewendet wurde um v zu generieren;
- die Zahl der Knoten auf dem Pfad von der Wurzel zu v (die Tiefe von v);
- die Pfadkosten des Pfades vom Anfangszustand bis zu dem v entsprechenden Zustand (traditionell mit $g(n)$ bezeichnet).

Der Datentyp Knoten[2]ist definiert durch

Datentyp KNOTEN
Komponenten: ZUSTAND,VATERKNOTEN,OPERATOR,TIEFE,PFADKOSTEN

Außerdem wird eine Datenstruktur zur Darstellung der Knoten benötigt, die zwar schon erstellt, aber noch nicht expandiert worden ist. Diese Knotenmenge nennt sich *Rand*.
Die Knotenmenge *Rand* wird als Liste implementiert. Auf einer Liste sind folgende Operationen festgelegt:

- MAKE-LIST(*Elemente*) erstellt eine Liste aus den gegebenen Elementen
- EMPTY?(*Liste*) gibt *true* zurück, wenn die Liste keine Elemente mehr hat
- REMOVE-FRONT(*Liste*) entfernt das erste Element aus der Liste und gibt es zurück
- LISTING-FN(*Elemente, Liste*) fügt eine Menge von Elementen in die Liste ein

Bei der Funktion LISTING-FN hängt es davon ab, wie die Elemente in die Liste eingefügt werden. Daraus erzeugen sich verschiedene Varianten des Suchalgorithmus. Der allgemeine Suchalgorithmus ist unter Verwendung dieser Funktionen in folgender Weise definiert:

```
function ALLGEMEINE-SUCHE(Problem, LISTING-FN) returns Lösung oder
Fehler
    nodes ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[Problem]))
    loop do
        if EMPTY?(nodes) then return Fehler
        node ← REMOVE-FRONT(nodes)
        if STATE(node) erfüllt ZIELPRÄDIKAT[Problem] then return node
        nodes ← LISTING-FN(nodes, EXPAND(node, OPERATORS[Problem]))
    end
```

Algorithmus 1 : Allgemeine Suchalgorithmus[2]

Leistungsbewertung für die Problemlösung

Die Leistung einer Suchstrategie lässt sich nach vier Kriterien bewerten [Chea]:

- Vollständigkeit: Findet die Suchstrategie garantiert eine Lösung wenn es eine gibt?
- Zeitbedarf: Wie lange dauert es bis die Suchstrategie eine Lösung zu finden?

[2]Die Quelle für den Algorithmus ist `https://www.tu-chemnitz.de/informatik/KI/scripts/ss06/KI_05-skr-2.doc`

- Speicherplatzbedarf: Wie viel Speicher benötigt die Suchstrategie für die Suche?
- Optimalität: Findet die Suchstrategie die beste Lösung, wenn es mehrere Lösungen gibt?

Was macht man nun, um eine Lösung zu finden? Am einfachsten ist es, irgendeinen zufälligen Zustand aus dem *Rand* herauszusuchen, eine Aktion darauf auszuüben und die sich daraus resultierenden Zustände dahingehend zu untersuchen, ob sie Zielzustände sind. „*Diese Strategie wäre das Stochern mit der Stange im Nebel, denn sie verläuft rein zufällig*“[Chea]. Damit kann man zwar auch ans Ziel gelangen, also wäre das Verfahren vollständig, aber der Zeitbedarf kann enorm groß werden, somit ist er nicht abschätzbar. Dieses Verfahren wäre garantiert nicht optimal.
Statt rein zufällig zu handeln, kann man die Suche strukturiert durchführen, aber ohne zu wissen, welche Richtung die geeignetste ist. Aus diesem Grund nennt man diese Strategien *blind* oder *uninformiert*. Sie sind aber strukturiert, d.h. sie arbeiten die Zustände nach einem bestimmten Plan ab.
Aus der Problembeschreibung kann man oft Hinweise dafür erhalten, welche Richtung für die Suche geeignet ist, d.h. in welcher Richtung man hoffen kann, möglichst schnell zu einem Zielknoten zu kommen. Diese Richtung sollte man vorziehen. Solche Hinweise werden in so genannten *heuristischen Funktionen* festgehalten, aus diesem Grund nennt man diese Strategien *heuristisch* oder *informiert* [Cheb].

2.4.3 Uninformierte Suchstrategien

In diesem Abschnitt werden wir Baum-Such-Verfahren zum Lösen von Problemen kennenlernen, ohne problemspezifische Informationen zu nutzen. Deswegen spricht man auch von *uninformierter Suche*. Man weiß nicht, wie weit auseinander die Lösung von einem gegebenen Zustand zu einen Zielzustand sein könnte.

Breitensuche

Die einfache klassische Form der Suche ist die Breitensuche (Breadth-first search), bei dem der Baum ebenenweise analysiert wird. Das ebenenweise Durchgehen eines Baumes wird auch als *military-order* genannt. [Pan]. Bei dieser Suche wird zunächst der Wurzelknoten expandiert, anschließend alle Tochterknoten des Wurzelknotens, dann deren Nachfolger usw. Generell werden immer zuerst alle Knoten auf Tiefe d des Suchbaums expandiert, bevor ein Knoten auf Tiefe $d + 1$ expandiert werden kann. Die Breitensuche wird mithilfe eines Aufrufs des Algorithmus ALLGEMEINE-SUCHE mit einer Listenfunktion, die die neuen Knoten am Ende der Liste einfügt, implementiert.

```
function BREITENSUCHE(Problem) returns eine Lösung oder Fehler
    return ALLGEMEINE-SUCHE(Problem, ENLIST-AT-END)
```

Algorithmus 2 : Algorithmus Breitensuche[2]

Diese uninformierte Suchstrategie erfüllt das Kriterium der Vollständigkeit, denn sie findet garantiert eine Lösung, falls eine vorhanden ist, und sie bekommt diejeni-

ge mit dem kürzesten Pfad heraus. Falls die Pfadkosten eine monotone wachsende Funktion der Tiefe des Suchbaums sind, dann ist die Breitensuche auch optimal.
Großer Nachteil dieser Strategie ist der hohe Zeit- und Speicherplatzbedarf von $O(b \cdot d)$. Zum Beurteilen des Zeit- und Speicherplatzbedarfs wird ein hypothetischer Zustandsraum angenommen, in dem jeder Zustand zu exakt b Folgezuständen expandiert werden kann. b ist der Verzweigungsfaktor *(branching factor)* der Zustände. Die Menge der insgesamt erstellten Knoten des Suchbaums ist eine Größe für den Zeit- und Speicherplatzbedarf. In der Tiefe 0 existiert ein Knoten (die Wurzel), in der Tiefe 1 b Knoten, in der Tiefe 2 b^2 Knoten usw., in der Tiefe d würden dann prinzipiell b^d Knoten existieren. Falls eine (die erste) Lösung in der Tiefe d existiert, dann ist die Menge der maximal zu erzeugenden Knoten $1+b+b^2+b^3+...+b^d = \frac{b^{d+1}-1}{b-1} = O(b^d)$
Von der Komplexität sind Zeit- und Speicherplatzbedarf gleich groß ($O(b^d)$), da im Worstcase-Fall alle Knoten der Tiefe d gespeichert werden müssen, im Bestfall die Knoten der Tiefe $d-1$ [Mue].

Tiefensuche

Bei der Tiefensuche wird immer ein Knoten auf der tiefsten Ebene des Suchbaums expandiert. Wenn keiner der Knoten auf der tiefsten Ebene expandierbar ist, wird bis zur letzten Verzweigung zurückgegangen (mittels *Backtracking*) und ein anderer Pfad verfolgt. Die Tiefensuche wird durch Aufruf des Algorithmus ALLGEMEINE-SUCHE mit einer Listenfunktion, die die neuen Knoten an den Anfang der Liste einfügt, implementiert

```
function TIEFENSUCHE(Problem) returns eine Lösung oder Fehler
    return ALLGEMEINE-SUCHE(Problem, ENLIST-AT-FRONT)
```

Algorithmus 3 : Algorithmus Tiefensuche[2]

Die Tiefensuche braucht viel weniger Speicherplatz als die Breitensuche, da in jeder Tiefe maximal b Knoten gespeichert wird und mindestens einen Knoten zusammen mit seinen Geschwisterknoten beinhaltet. Wenn m die größte im Suchbaum vorkommende Tiefe ist, dann werden $b \cdot m$ Speicherzellen gebraucht.
Die Rechenzeit der Tiefensuche ist, genau wie bei der Breitensuche, $O(b^m)$, weil im worst case Fall alle Knoten bis zur maximalen Tiefe m untersucht und besucht werden müssen. In der Realität kann die Rechenzeit niedriger sein, da nur ein Bereich des Zustandsraums untersucht werden muss, bis eine Lösung entdeckt ist.
Die Tiefensuche hat den Nachteil gegenüber der Breitensuche, dass sie bei unendlich tiefen Bäumen in eine Endlosschleife läuft, falls auf dem ganz linken Ast keine Lösung existiert. Somit wäre die Frage nach dem Herausfinden der optimalen Lösung geklärt. Die Tiefensuche ist also weder vollständig noch optimal [Ert09].

Tiefenbeschränkte Suche

Wenn die festgelegte Tiefenschranke k übertroffen wird, werden keine Nachfolger dieser Knoten mehr erstellt. Die Tiefensuche bekommt in diesem Fall jeden Zielknoten, der höchstens Tiefe k hat, heraus [uDDSb]. Die tiefenbeschränkte Suche kann mit

einem speziellen Algorithmus oder durch Aufruf von ALLGEMEINE-SUCHE zusammen mit Operatoren, die die Tiefe überwachen, implementiert werden. Der Zeit- und Speicherplatzbedarf ist nahezu wie bei der Tiefensuche; ist k der Tiefenschnitt, dann ist der Zeitbedarf $O(b^k)$ und der Speicherbedarf $O(b \cdot k)$.

Suche mit iterativer Vertiefung

Mit der iterativ vertiefenden Suche ist tiefenbeschränkte Suche mit variablem Tiefenschnitt gemeint. Hierbei werden schrittweise wachsende Tiefenschnitte 0, 1, 2, ... bestimmt und für jeden Wert die tiefenbeschränkte Suche ausgeführt. Mit folgender Funktion kann die Implementierung beschrieben werden:

```
function SUCHE-MIT-ITERATIVEM-VERTIEFEN(Problem) returns eine
Lösungsfolge oder Fehler
    inputs: Problem; eine Problembeschreibung

    for depth ← 0 to ∞ do
        if Tiefenbeschränkte-Suche(Problem, depth) erfolgreich then return ihr
        Ergebnis
    end
    return Fehler
```

Algorithmus 4 : Suche mit iterativem Vertiefen[2]

Diese Suche kombiniert Vorteile der Tiefen- und Breitensuche: sie ist vollständig und optimal, aber hat so viel Speicherplatz wie die Tiefensuche. Die Rechenzeit der Suche mit iterativem Vertiefen beträgt $O(b^d)$ und der Speicherplatzbedarf $O(b \cdot d)$. Somit stellt die Suche mit iterativem Vertiefen das beste blinde Suchverfahren dar. Es taugt für große Suchräume mit unbekannter Tiefe der Lösung [Chea].

KAPITEL

DREI

Informierte Suche

In den vorherigen Kapiteln wurde vorgestellt, dass die sogenannte uninformierte oder blinde Suche, zu der die Breiten- und Tiefensuche gehören, bei schwierigen Problemen unmöglich ist. Vergleichsweise stehen beim Schachspiel jedem Spieler in seinem Zug ungefähr 30 mögliche Züge zur Verfügung, die wiederum jeweils 30 weitere mögliche Züge aufweisen, etc. Um einen vollen Zug (Zug und Gegenzug) hervorzusehen, würde heißen, ungefähr 1000 Möglichkeiten zu beachten. Bei vier Zügen summiert sich die Zahl der zu berechnenden Züge in die Billionen. Bei 40 Zügen sind insgesamt 10^{120} Zusammenstellungen zu untersuchen. Das sind mehr Kombinationen, als es auf der Erde Atome gibt [MT14]. Es ist nicht vorauszusehen, dass ein System eine solche Kontrolle jemals vollständig durchführen können wird. Dieses Phänomen wird als kombinatorische Explosion definiert [Hau87]. Schon bei schmalen Suchbäumen lässt es die Notwendigkeit an Zeit und Rechenkapazität ins Unzählbare klettern.
Man erreicht einen Vorteil, wenn man vorher weiß, wo es sich zu suchen rentiert. Das verlangt aber eine gewisse Zusammenstellung: Ich lege den Haustürschlüssel nie in die Schublade, auf dem Schreibtisch, auf dem Fernseher oder in die Schuhkammer, also benötige ich ihn dort auch nicht zu suchen. Dieses Vorgehen nennt man Beschränkung des Suchraums. Noch intelligenter ist es, wenn man Hypothesen darüber aufstellen kann, wo man genau nachschauen muss: Ich habe die Tür wohl aufgeschlossen, jedoch den Schlüssel drinnen nicht abgelegt. Also muss er noch im Schloss stecken. So ein Verfahren wird als heuristische oder informierte Suche bezeichnet [Pea84]. Die KI-Wissenschaft hat unterschiedliche heuristische Suchverfahren entworfen, mit der Gemeinsamkeit, dass sie Wissen über das genaue Suchproblem von Gebrauch machen, um zu bestimmen, wo sie als nächstes suchen [Len02].

3.1 Informierte Suchstrategien

Bei informierten Suchstrategien beruht die Auswahl eines Knotens v zur Ausbreitung auf einer heuristischen Bewertungsfunktion $f(v)$. Diese Bewertungsfunktion kann verschiedene Sachverhalte mit berücksichtigen, beispielsweise Aussagen über den Knoten v beziehungsweise seinen Zustand, Aussagen über den Pfad vom Startknoten s nach v, Hinweise aus der Problem- und Zielbeschreibung sowie Wissen aus

dem Problembereich. Die Suchstrategie wird maßgeblich von der Bewertungsfunktion f entschieden. Im Folgenden werden verschiedene informierte Suchstrategien vorgestellt, die sich bis auf die Bewertungsfunktion f ähnlich sind [Brä].

3.1.1 Best-First-Suche

Das allgemeine Verfahren, das wir nun anschauen wollen, ist die sogenannte Bestensuche (Best-First). Die Bestensuche ist eine Instanz des TREE-SEARCH oder GRAPH-SEARCH Algorithmus und sucht sich einen Knoten auf Grundlage einer Evaluierungsfunktion bzw. Bewertungsfunktion $f(n)$ zur Expansion aus. Die Evaluierungsfunktion dient als Kostenabschätzung, sodass der Knoten mit der besten Bewertung zuerst expandiert wird. Der folgende Algorithmus implementiert diese Suche.

function BEST-FIRST-SUCHE(*Problem*, EVAL-FN) **returns** eine Lösungsfolge oder Fehler
  inputs: *Problem*; eine Problembeschreibung
    EVAL-FN ; eine Evaluierungsfunktion
  LISTING-FN ← eine Funktion, die Knoten mittels EVAL-FN ordnet
  return ALLGEMEINE-SUCHE(*Problem*, LISTING-FN)

Algorithmus 5 : Best-First-Suche[1]

Es existiert eine ganze Klasse von Best-First-Suchverfahren. Sie verwenden alle eine erwartete Kostenmenge für die Lösung und probieren dieses zu minimieren. Diese Kostenmenge muss die Kosten eines Pfads von einem Zustand zu dem am wenigsten entfernten Zielzustand vermerken.

- Komplexität:
  Sei b der Verzweigungsfaktor und d die maximale Tiefe der Lösung. Dann beträgt die Zeitkomplexität $T(n) = 1 + b + b^2 + ... + b^d \rightarrow O(b^d)$. Die Speicherkomplexität ist: $S(n) = T(n)$. Der Algorithmus ist nicht vollständig, da unendliche Pfade existieren, falls jeder einzelne Knoten als beste Lösung evaluiert wird. Die Optimalität ist abhängig von der Heuristik.

- Anwendungen:
  Web crawler (automatic indexer), Spiele [Nap]

3.1.2 Greedy-Suche

Eines der einfachsten Best-First Suchstrategien ist die Minimierung der geschätzten Kosten für das Erreichen des Ziels. Das heißt, der Knoten, dessen Zustand als dem Zielzustand am nächsten liegend eingeschätzt wird, wird immer als erstes expandiert. Für die meisten Probleme können die Kosten des Erreichens des Zieles von einem bestimmten Zustand abgeschätzt, jedoch aber nicht genau bestimmt, werden. Eine Funktion, die solche Kostenschätzungen berechnet nennt man heuristische Funktion

[1]Die Quelle für den Algorithmus ist `https://www.tu-chemnitz.de/informatik/KI/scripts/ss06/KI_05-skr-2.doc`

und wird in der Regel mit dem Buchstaben h angegeben. $h(n)$ sind die geschätzten Kosten des günstigsten Pfads von einem Zustand am Knoten n zum Zielzustand. Eine Best-First-Suche, die h zum Auswählen des nächsten Knoten zum Expandieren nutzt, wird als Greedy-Suche bezeichnet. Der folgende Algorithmus ist eine Implementierung der Greedy-Suche [Chea]:

function GREEDY-SEARCH(*Problem*) **returns** eine Lösung oder Fehler
  return BEST-FIRST-SEARCH(*Problem, h*)

Algorithmus 6 : Greedy-Suche[1]

Jede beliebige Funktion kann als heuristische Funktion genutzt werden. Es erfordert nur $h(n) = 0$, falls n ein Zielknoten ist.
Um uns ein Bild machen zu können, wie eine heuristische Funktion aussieht, müssen wir ein spezielles Problem wählen, weil heuristische Funktionen problem-spezifisch sind. Betrachten wir uns hierfür ein Routensuchproblem an, mit dem Ziel von Arad nach Bukarest zu gelangen. Die Landkarte von Rumänien auf Abbildung 3.1 veranschaulicht das Problem.

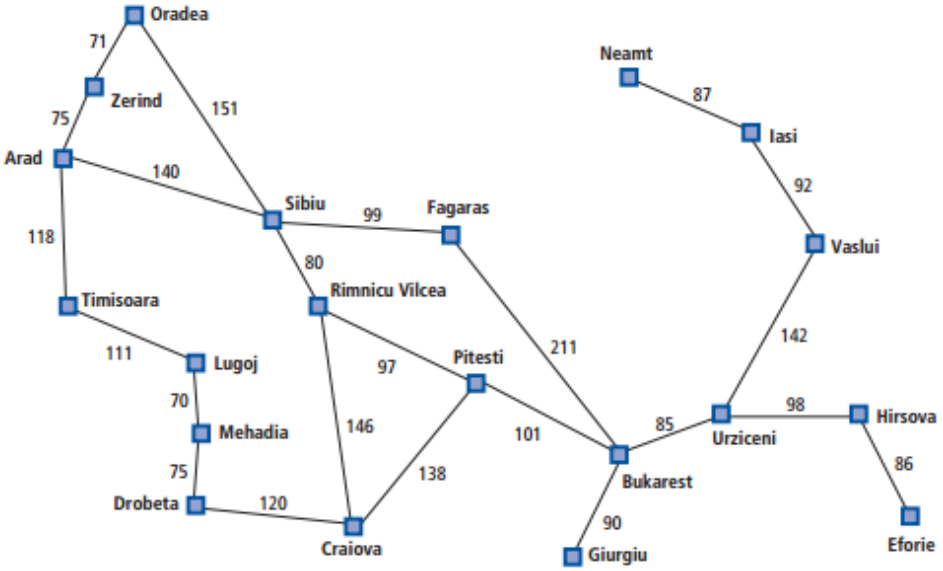


Abbildung 3.1: Eine einfache Straßenkarte von Rumänien [2]

Eine gute heuristische Funktion für die Routensuchprobleme wie diese ist die Luftliniendistanz zum Ziel. Diese bezeichnen wir wie folgt:

$$h_{LLD}(n) = \text{Luftliniendistanz zwischen } n \text{ und Zielort}$$

Wenn das Ziel Bukarest ist, dann benötigen wir die Luftliniendistanzen zu Bukarest, die in Abbildung 3.2 gekennzeichnet sind. Beispielsweise ist $h_{LLD}(Arad) = 366$. Es ist zu beachten, dass wir nur die Werte von h_{LLD} berechnen können, falls wir

[2] Stuart J. Russell and Peter Norvig, Künstliche Intelligenz. Ein moderner Ansatz

die Kartenkoordinaten der Städte in Rumänien wissen. Darüber hinaus braucht man eine gewisse Erfahrung, um zu wissen, dass h_{LLD} etwas über die tatsächlichen Straßendistanzen aussagt und damit eine sinnvolle Heuristik ausdrückt.

Arad	366	Mehadia	241
Bukarest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Abbildung 3.2: Luftliniendistanzen nach Bukarest [2]

Die Abbildung 3.6 stellt den Verlauf einer Greedy-Suche vor, um einen Pfad von Arad nach Bukarest zu finden. Mit der Luftliniendistanzheuristik ist der erste Knoten, der expandiert wird von Arad aus Sibiu, weil dieser näher zu Bukarest ist als Zerind oder Timisoara. Der nächste Knoten, der expandiert wird, wird Fagaras sein, weil dieser am nächsten ist. Fagaras wiederum erzeugt den Knoten Bukarest, welcher das Ziel ist. Für dieses spezielle Problem führt minimale Suchkosten durch: Es findet eine Lösung ohne dabei jemals einen Knoten zu expandieren, der nicht im Lösungsweg enthalten ist. Allerdings ist dieser nicht optimal: Der Weg, der über Sibiu und Fagaras nach Bukarest gefunden wird ist 32 km länger als der Weg über Rimnicu Vilcea und Pitesti. Dieser Weg wurde nicht gefunden, weil Fagaras auf Luftliniendistanz näher zu Bukarest ist als Rimnicu Vilcea, also wurde dieser als erstes expandiert. Man erkennt also, weshalb der Algorithmus als gierig bezeichnet wird - er bemüht sich in jedem Schritt, dem Ziel so nah wie möglich zu kommen.
Wenn man die Greedy-Suche mit der Tiefensuche vergleicht, ist erstere für Bäume sogar in einem endlichen Zustandsraum unvollständig. Angenommen wir haben das Problem von Iasi nach Fagaras zu kommen gegeben. Die Heuristik empfiehlt, Neamt als Erstes zu expandieren, da es am nächsten bei Fagaras liegt, aber dies ist jedoch eine Sackgasse. Die Lösung besagt, als Erstes Vaslui anzupeilen - ein Weg, der der Heuristik nach ursprünglich weiter vom Ziel entfernt liegt - und dann folgend nach Urziceni, Bukarest und Fagaras zu fahren. Aber der Algorithmus findet diese Lösung zu keiner Zeit. Wenn er nämlich Neamt expandiert, kehrt Iasi zurück in den Grenzbereich. Iasi ist näher an Fagaras als Vaslui und aus diesem Grund wird Iasi wieder expandiert, was zu einer Endlosschleife führt.
Im Worstcase ist die Zeit- und Platzkomplexität für die Greedy-Suche $O(b^m)$, wobei m die maximale Tiefe des Suchraumes bedeutet. Mithilfe einer guten Heuristikfunktion kann die Zeit- und Platzkomplexität aber erheblich vermindert werden. Wie groß diese Verminderung möglich ist, ist bedingt vom jeweiligen Problem und der

Qualität der Heuristik [RN04].

a) Der Ausgangszustand

Abbildung 3.3: Schritt 1: Die Ausgangslage [2]

b) Nach der Expansion von Arad

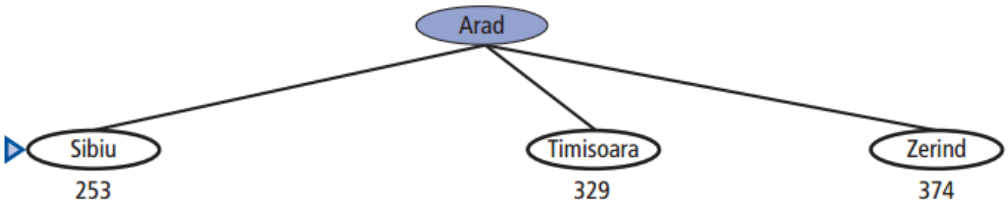


Abbildung 3.4: Schritt 2: Nach der Relaxation von Arad [2]

c) Nach der Expansion von Sibiu

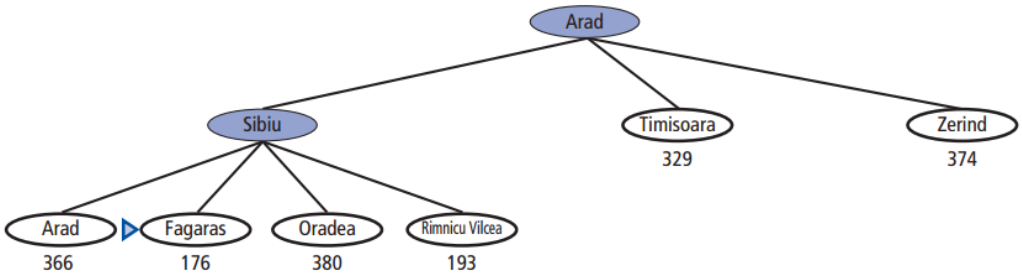


Abbildung 3.5: Schritt 3: Nach der Relaxation von Sibiu [2]

d) Nach der Expansion von Fagaras

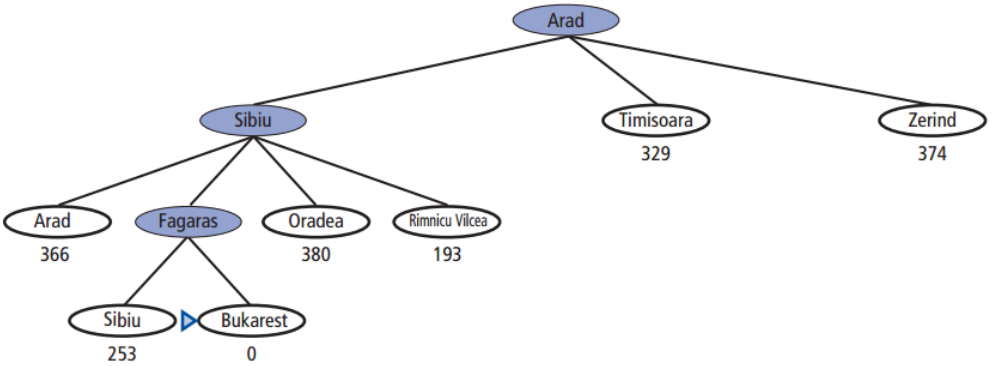


Abbildung 3.6: Schritt 4 in einer Greedy-Suche für Bukarest unter Verwendung der Luftliniendistanz zu Bukarest, wobei die Luftliniendistanz-Heuristik h_{LLD} eingesetzt wird. [2]

3.1.3 A^*-Suche

Wenn man die heuristische Funktion h und die Pfadkostenfunktion g addiert, bekommt man die Funktion $f(n) = g(n) + h(n)$ heraus, die die geschätzten Kosten der billigsten Lösung durch den Knoten n darstellt.
Man bezeichnet die Funktion h als zulässige Heuristik, wenn die Funktion das Merkmal besitzt, dass sie die Kosten eines Pfades bis zu einem Zielknoten nicht überschätzt. Allgemein erklärt: *Sind h‘ die tatsächlichen (aber nicht bekannten) Kosten, dann muss für h gelten: Für alle Knoten n: h(n) ≤ h‘(n). Wenn h zulässig ist, dann überschätzt auch die Funktion f niemals die Kosten der besten Lösung durch n.*
Eine Best-First-Suche unter Nutzung von f als Evaluierungsfunktion mit zulässigem h nennt man A^*-Suche. Der Algorithmus für die A^*-Suche ist folgender [Chea]:

function A^*-SUCHE(*Problem*) **returns** eine Lösung oder Fehler
  return BEST-FIRST-SUCHE(*Problem, g + h*)

Algorithmus 7 : A^*-Suche[1]

Eine heuristische Funktion heißt monoton, wenn deren Werte entlang der Pfade von der Wurzel zu einem Blatt im Suchbaum niemals geringer werden. Eine heuristische Funktion, die nicht monoton ist, kann in eine monotonen Funktion transformiert werden. Dies passiert wie folgt: *Ist n ein Knoten und n‘ einer seiner Nachfolger. Dann setze f(n’) = max(f(n), g(n’) + h(n’)).*
Diese Gleichung nennt man Pfadmaximierungsgleichung. Durch sie entwickeln sich heuristische Funktionen, die monoton sind.

Beispiel 1 *[uDDSb]: Weitere Beispiele für Graphen und Kostenfunktionen sind:*

Wenn $h(N) = 0$, dann ist der A^*-Algorithmus dasselbe wie die sogenannte Gleiche-Kostensuche (branch-and-bound [3])

Wenn $c(N_1, N_2) = k$ (k Konstante, z.B. 1), und $h(N) = 0$, dann entspricht der A^*-Algorithmus der Breitensuche.

Der Dijkstra-Algorithmus zum Finden optimaler Wege in einem Graphen entspricht (wenn man von der Speicherung des optimalen Weges absieht) der folgenden Variante: Man wählt $h(N) = 0$ und nimmt den A^*-Algorithmus.

3.1.4 Weitere informierte Suchstrategien

Es gibt bestimmte Problemtypen, wie das oben erwähnte 8-Damenproblem, bei dem der Lösungsweg irrelevant ist und einzig das Problem bestmöglichst zu lösen genügt. Dabei kennt man noch nicht explizit den Zielzustand, sondern wird nur mithilfe von Anforderungen dargestellt.
In solchen Fällen können iterative Verbesserungsalgorithmen benutzt werden. Hierfür wird mit einem zufälligen Knoten im Zustandsraum gestartet und nach vorteilhafteren Nachfolgern gesucht. Dabei gibt es zwei Klassen dieser Algorithmen, die man unterscheidet:

- Hill-climbing-Suche: Dieser Algorithmus rückt kontinuierlich in die Richtung von größeren Werten heran bzw. besseren Nachfolgern. Kommt man nun in die Situation, wo nun mehrere Nachfolger in Frage kommen, wird durch Zufall einer ausgewählt, d.h. die Suche läuft einen zufälligen Weg.
  Mängel dieser Strategie sind, dass die Suche auf einem lokalen Maximum oder auf einem Sattel stoppt. Eine Methode dieser Strategie kann gegen dieses Nachteil helfen. Man bezeichnet diese als *random-restart hill-climbing*. Er bearbeitet eine Folge von hill-climbing-Suchen von verschiedenen Anfangspunkten und speichert das beste Ergebnis. Wenn nun reichlich Wiederholungen ausgeführt werden, wird auch die optimale Lösung herausgefunden.

- Simulated annealing: Statt einem zufälligen Neustart, kann beim *simulated annealing* einige eintrübende Schritte bzw. einige Schritte zurückgesetzt werden. Damit kann ein anderer Weg herausgefunden werden, der zu einer besseren Lösung resultiert.
  Jeden Schritt wählt man zufällig aus, wenn die jetzige Situation besser wird, wird er ausgeführt. Falls die Situation sich aber verschlechtert, wird er mit einer bestimmten Wahrscheinlichkeit < 1 ausgeführt. Verstellbar ist diese Wahrscheinlichkeit durch zwei Parameter und sie ändert sich auch mit der Anzahl der zugelassenen falschen Schritten [Mue].

3.2 Heuristikfunktionen

Bislang haben wir erst ein Beispiel einer Heuristik gesehen und zwar die Luftliniendistanz für Routensuchprobleme. In diesem Kapitel werden wir uns die Heuristiken für das 8-Puzzle anschauen, um einen Aufschluss über Heuristiken im Allgemeinen

[3] Beim Branch-and-Bound-Algorithmus wird die Tiefensuche dann beendet, wenn der aktuelle Zustand schlechter ist als der bislang herausgefundene Zustand. Zur Realisierung der Beurteilung „besser“oder „schlechter“ muss eine Qualitätsfunktion festgelegt werden [HL13]

zu geben.
Das 8-Puzzle war eines der ersten Heuristik-Suchproblemen. Das Ziel dieses Puzzles ist es, die Platten horizontal oder vertikal in den leeren Platz zu schieben, bis die Anfangskonfiguration mit der Zielkonfiguration übereinstimmt (siehe Abbildung 3.7).

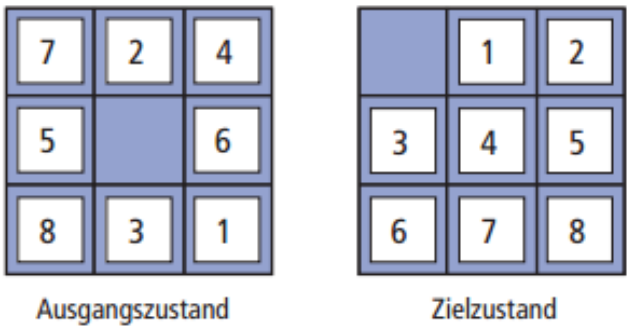


Abbildung 3.7: Beispiel für das 8-Puzzle. Die Lösung besteht aus 26 Schritte [1]

Vor allem der Schwierigkeitsgrad des 8-Puzzles macht es interessant. Eine typische Lösung handelt in 20 Schritten, obwohl dies natürlich vom Startzustand abhängt. Der Verzweigungsgrad geht gegen 3 (wenn das leere Feld in der Mitte ist, dann gibt es vier mögliche Spielzüge; wenn es in einer Ecke ist sind es zwei; und wenn es entlang einer Kante ist sind es drei). Das bedeutet, eine umfassende Suche bis zur Tiefe 22 würde etwa $3^{22} = 3.1 \cdot 10^{10}$ Anordnungen betrachten. Eine Graphsuche würde dies um einen Faktor von etwa 170.000 reduzieren, weil es nur $9!/2 = 181.440$ verschiedene Anordnungen existieren, die erreichbar sind. Das ist immer noch eine große Anzahl an Anordnungen. Also ist der nächste Punkt eine gute Heuristikfunktion zu finden. Wenn wir unter Benutzung von A^* die kürzesten Lösungen herausfinden wollen, benötigen wir eine Heuristikfunktion, die niemals die Anzahl der Schritte zum Ziel überschätzt. Im Folgenden werden zwei der gebräuchlicheren Ansätze erklärt:

- h_1 = die Anzahl der Felder, die sich in der falschen Position befinden . In Abbildung 3.7 ist keines der acht Felder in der Zielposition, also würde der Ausgangszustand $h_1 = 8$ haben. h_1 ist eine zulässige Heuristik, weil klar ist, dass jedes Feld, das sich nicht am richtigen Platz befindet, mindestens einmal bewegt werden muss.

- h_2 = die Summe der Distanzen der Felder von ihren Zielpositionen. Weil die Felder sich nicht diagonal bewegen können, berechnen wir die Distanz aus der Summe der horizontalen und vertikalen Distanzen. Dies wird manchmal auch City-Block-Distanz oder Manhattan-Distanz genannt. h_2 ist ebenso zulässig, weil jede Bewegung nur ein Feld um einen Schritt näher an das Ziel bewegen kann. Die 8 Felder im Startzustand ergeben eine Manhattan-Distanz von $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$.

Erwartungsgemäß, resultiert keiner dieser Werte eine Schätzung oberhalb der tatsächlichen Lösungskosten, die 26 betragen [RN04].

KAPITEL

VIER

Der A^*-Algorithmus

So genannte Agenten tauchen oft in Computerspielen auf: Agenten nennt man auch Spieler oder gehören in den Bereich der Künstlichen Intelligenz (KI) und sind rational und lassen sich wie Spieler steuern. Die KI oder der Spieler gibt oft dem Agenten den Auftrag, an einen bestimmten Ort bzw. zu einem Ziel zu gehen. Die Bestimmung des Weges vom Standort des Agenten zu seinem Ziel bezeichnet man als *Pathfinding* (deutsch: Wegsuche oder Pfadsuche). Das Ziel eines *Pathfinding*-Methode ist einen kürzesten Weg zu bestimmen.
Kürzeste Wege spielen sowohl in Computerspiele als auch in unserem Alltag eine große Rolle. In Computerspielen besitzt meist ein Spieler einen Agenten und befindet sich in großen Umgebungen und möchte sich dort bewegen können. Bei rechenintensiven Spielen wie Warcraft 3 möchte man dann schnelle Informationen über den kürzesten Weg erhalten, deren Start und Ziel sich weit entfernt in einem großen Suchraum befinden [Wal]. Im Alltag ist das nicht viel anders. Kürzeste Wege spielen bei uns im Alltag nämlich auch eine große Rolle, auch wenn uns nicht immer klar ist, dass wir die kürzesten Wege berechnen: Die Frage, wie man von der Küche am schnellsten ins Bad kommt, benötigt bereits eine Berechnung des kürzesten Wegs. Das ist aber ein triviales Beispiel. Wenn jedoch die Wege länger werden und die Anzahl der Möglichkeiten größer, wie z.B. „Wie komme ich morgens am schnellsten zur Uni oder auf die Arbeit? Welche S-Bahn-Verbindung nehme ich?“, dann ist es sehr schwer im Kopf den kürzesten Weg zu berechnen. Da fragt man schnell sein Navigationssystem oder die Fahrplanseite [uWFR].

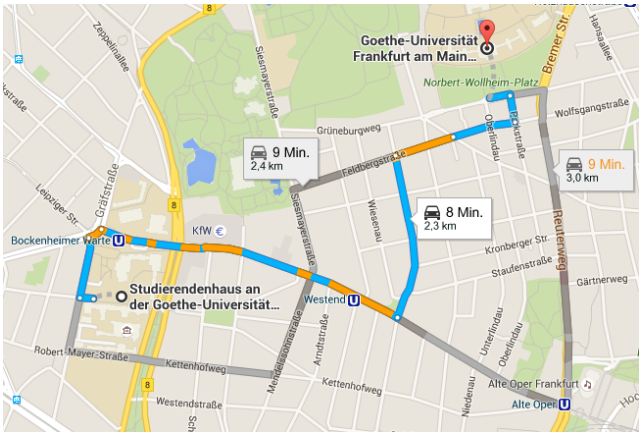


Abbildung 4.1: Kürzeste Wege von Uni Campus Bockenheim zum Uni Campus Westend [2]

Der A^*-Algorithmus ist ein Algorithmus, der kürzeste Wege berechnet. Wir haben ihn kurz im Kapitel der informierten Suche kennengelernt. Dieser Algorithmus nimmt in meiner Arbeit einen hohen Stellenwert ein.

4.1 Definition

In dieser Arbeit befassen wir uns mit der experimentellen Untersuchung eines Verkehrsverbunds unter Verwendung des A^*-Algorithmus. Der A^*-Algorithmus unter Berücksichtigung der aktuellen Pfadlänge in einer heuristischen Suche wurde von Hart, Nilsson und Raphael entworfen und 1968 in „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“ veröffentlicht [PEHR68]. Ein paar Jahre später hat man aufgrund einiger technischer Gegensätzlichkeiten eine weitere Veröffentlichung getätigt [PH72].
Inzwischen ist der A^*-Algorithmus bis heute immer wieder Thema wissenschaftlicher Veröffentlichungen, die sich sowohl mit den Eigenschaften wie der Optimalität und Vollständigkeit des Algorithmus als auch mit der Anwendung für Probleme in der Praxis befassen.
Nun ist der A^*-Algorithmus einer der berühmtesten, einfachsten und den meist benutzten heuristischen Algorithmen.

Der A^*-Algorithmus muss Suchprobleme lösen, die folgende Eingabe haben:

- einen Graphen mit einer Nachfolgerfunktion N_F, die zu jedem Knoten N die Nachfolger $N_F(N)$ ermittelt.
- einer reellwertige Kostenfunktion c für Kanten, sodass $c(N_1, N_2) \in R$ gerade die Kosten der Kante von Knoten N_1 zu Knoten N_2 bestimmt. Für einen Weg $N_1 \longrightarrow N_2 \longrightarrow ... \longrightarrow N_k$ bezeichne $c_W(N_1N_2...N_k)$ gerade die Summe der Kosten $\sum_{i=1}^{k-1} c(N_i, N_{i+1})$.

[2] Die Quelle für die Abbildungen ist `GoogleMaps`

- ein Startknoten S
- eine Schätzfunktion $h(.)$, der die Restkosten von einem bereits erreichten Knoten N bis zum nächsten Zielknoten abschätzt
- einen Test auf Zielknoten Z.

Das Ziel des A^*-Algorithmus ist die Ermittlung eines optimalen (d.h. mit minimalen Kosten) Weges vom Startknoten zu einem der Zielknoten [uDDSa].

4.1.1 Algorithmus A^*

Die Bewertung des zu expandierenden Knotens setzt sich aus den bereits verbrauchten Kosten $g(N)$ vom Start bis zu einem aktuellen Knoten N und der noch geschätzten Kosten $h(N)$ bis zu einem Zielknoten Z zusammen. Hierfür wird die Bewertungsfunktion $f(N) = g(N)+h(N)$ verwendet. Man unterscheidet zwei Methoden des A^*-Algorithmus und zwar das Baum-Such-Verfahren und das Graph-Such-Verfahren. Der Unterschied liegt in der Aktualisierung des minimalen Weges bis zu einem Knoten während des Ablaufs beim Graph-Such-Verfahren.

Datenstrukturen:

- Mengen von Knoten: *Open* und *Closed*
- Wert $g(N)$ für jeden Knoten (markiert mit Pfad vom Start zu N)
- Heuristik h, Zieltest Z und Kantenkostenfunktion c

Eingabe:

- $Open := \{S\}$, wenn S der Startknoten ist
- $g(S) := 0$, ansonsten ist g nicht initialisiert
- $Closed := \emptyset$

Algorithmus:

repeat
  Wähle N aus *Open* mit minimalem $f(N) = g(N) + h(N)$
  if $Z(N)$ **then**
    break; // Schleife beenden
  else
    Berechne Liste der Nachfolger $\mathsf{N} := NF(N)$
    Schiebe Knoten N von *Open* nach *Closed*
    for $N' \in \mathsf{N}$ **do**
      if $N' \in Open \cup Closed$ und $g(N) + c(N, N') > g(N')$ **then**
        skip // Knoten nicht verändern
      else
        $g(N') := g(N) + c(N, N')$; // neuer Minimalwert für $g(N')$
        Füge N' in *Open* ein und (falls vorhanden) lösche N' aus *Closed*;
      end-if
    end-for
  end-if
until $Open = \emptyset$;
if $Open = \emptyset$, **then** Fehler, kein Zielknoten gefunden
else N ist der Zielknoten mit $g(N)$ als minimalen Kosten
end-if

Algorithmus 8 : A^*-Algorithmus[3]

Beim Baum-Such-Verfahren ist keine *Closed*-Menge vorhanden. Aus diesem Grund kann der selbe Knoten mehrfach in der *Open*-Menge mit verschiedenen Wegen sein. Beim Graph-Such-Verfahren dagegen steht der Knoten nur einmal in der *Open*-

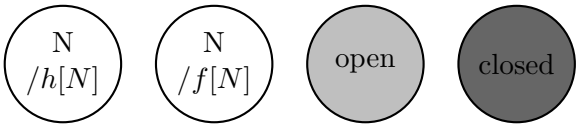
[3]Die Quelle für die Abbildungen ist `http://www.ki.informatik.uni-frankfurt.de/lehre/SS2016/KI/skript/skript-KI.pdf`

Menge und der aktuell bekannte minimale Weg ist immer bekannt.
Es ist zu berücksichtigen, dass der Zielknoten in der *Open*-Menge stehen kann, man diesen aber erst als Ziel feststellt, wenn der Zielknoten als zu expandierenden Knoten ausgewählt wird. Ebenfalls kann ein Knoten, der sich in der *Closed*-Menge befindet, wieder in die *Open*-Menge aufgenommen werden, weil ein kürzerer Weg ermittelt wird [uDDSa].

4.2 Beispiel

Wir betrachten das Routensuchproblem, bei dem es die Aufgabe ist, den kürzesten Weg von Düsseldorf nach Berlin zu finden. Die Kantengewichte entsprechen den Kosten (Entfernung in km), um von einer Stadt zur nächsten zu gelangen. Jeder Knoten besitzt die geschätzte Entfernung zum Zielknoten (Wert der h Funktion) oder die geschätzten Kosten der kostengünstigsten Lösung (Wert der f Funktion). Als Heuristik wird hier die Luftlinie benutzt (fiktiv). Wenn ein Knoten erreicht wurde, zeigt eine Kante auf den Vorgängerknoten.

Notation



Abkürzungen:

- Dd = Düsseldorf
- Os = Osnabrück
- Ko = Köln
- Br = Bremen
- H = Hannover
- Be = Berlin
- Er = Erfurt

Führt man den A^*-Algorithmus für dieses Beispiel mit Startknoten Düsseldorf (Dd) und Zielknoten Berlin (Be) aus, erhält man den Ablauf:

Am Anfang:
$Open : \{Dd\}$
$Closed : \{\emptyset\}$

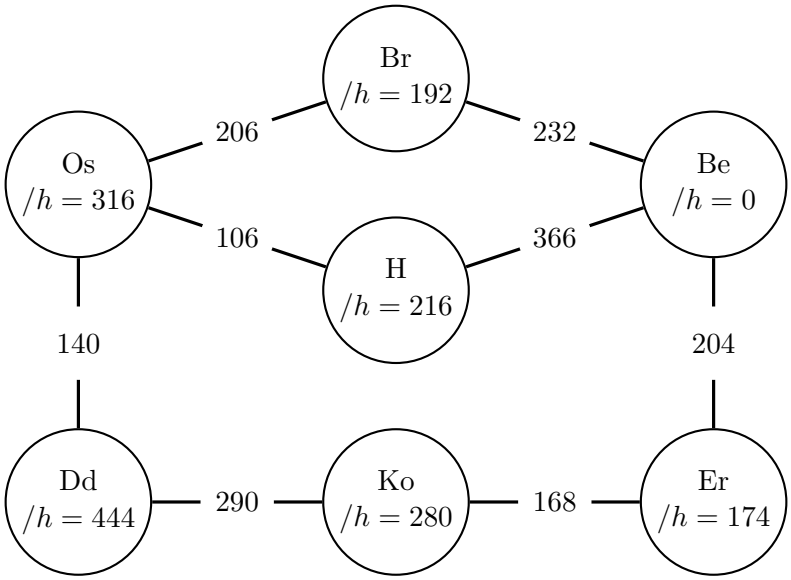


Abbildung 4.2: Landkarte

Nach Expansion von Dd:
$Open : \{Os, Ko\}$ mit $f(Os) = 456$ und $f(Ko) = 570$, da $f_{Os} = g(Dd) + c(Dd, Os) + h(Os) = 0+140+316 = 456$ und $f_{Ko} = g(Dd)+c(Dd, Ko)+h(Ko) = 0+290+280 = 570$.
$Closed : \{Dd\}$

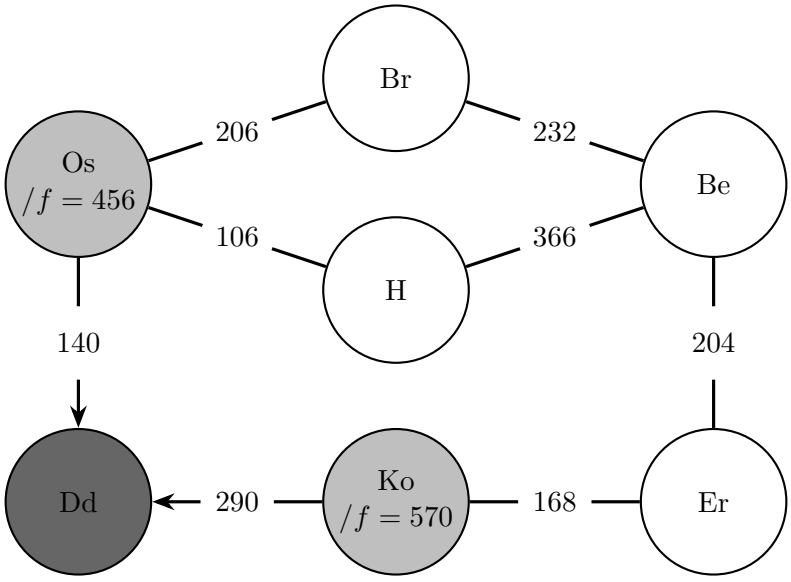


Abbildung 4.3: Schritt 1: Nach der Relaxation von Düsseldorf (Dd)

Nächster Schritt:
Da f_{Os} kleiner ist als f_{Ko}, wird Os als nächster Knoten expandiert.

Nach Expansion von Os:
$Open : \{Ko, Br, H\}$ mit $f(Ko) = 570, f(Br) = 538$ und $f(H) = 462$, da $f_{Ko} = g(Dd)+c(Dd, Ko)+h(Ko) = 0+290+280 = 570$, $f_{Br} = g(Os)+c(Os, Br)+h(Br) = 140+206+192 = 538$ und $f_H = g(Os)+c(Os, H)+h(H) = 140+106+216 = 462$.
$Closed : \{Dd, Os\}$

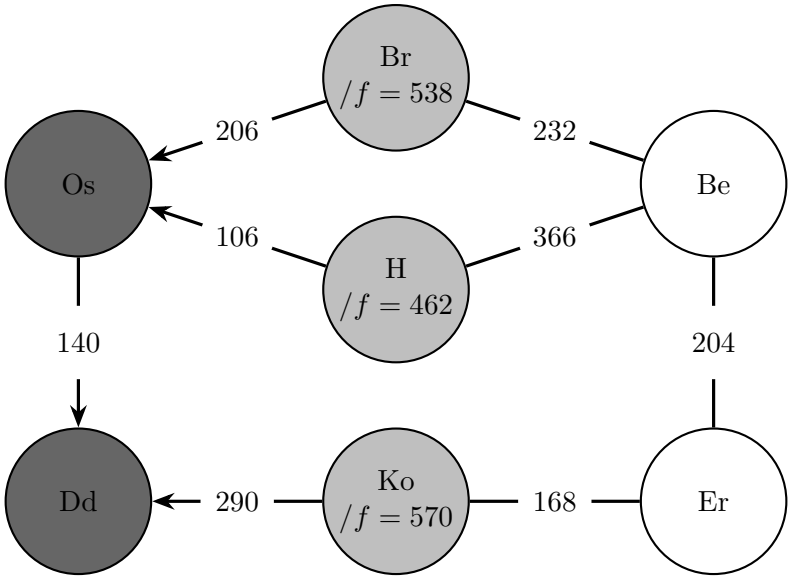


Abbildung 4.4: Schritt 2: Nach der Relaxation von Osnabrück (Os)

Nächster Schritt:
Da f_H kleiner ist als f_{Ko} und f_{Br}, wird H als nächster Knoten expandiert.

Nach Expansion von H:
$Open : \{Ko, Br, Be\}$ mit $f(Ko) = 570, f(Br) = 538$ und $f(Be) = 612$, da $f_{Ko} = g(Dd)+c(Dd, Ko)+h(Ko) = 0+290+280 = 570$, $f_{Br} = g(Os)+c(Os, Br)+h(Br) = 140+206+192 = 538$ und $f_{Be} = g(H)+c(H, Be)+h(Be) = 246+366+0 = 612$.
$Closed : \{Dd, Os, H\}$

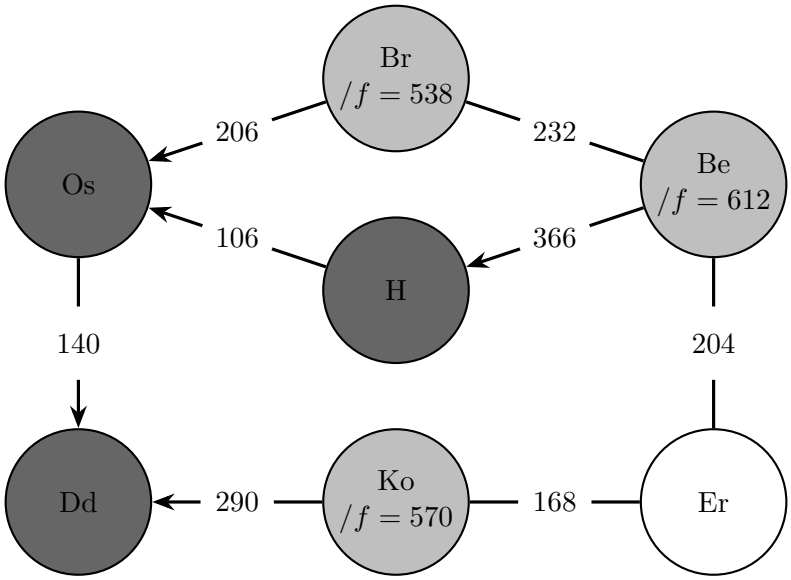


Abbildung 4.5: Schritt 3: Nach der Relaxation von Hannover (H)

Nächster Schritt:
Da f_{Br} kleiner ist als f_{Ko} und f_{Be}, wird Br als nächster Knoten expandiert.

Nach Expansion von Br:
Open : $\{Ko, Be\}$ mit $f(Ko) = 570$ und $f(Be) = 578$, da $f_{Ko} = g(Dd)+c(Dd, Ko)+h(Ko) = 0+290+280 = 570$ und $f_{Be} = g(Br)+c(Br, Be)+h(Be) = 346+232+0 = 578$. Nachdem nun also Br expandiert wird, ergibt sich nun für den Knoten Be ein kürzerer Weg über Br als über H davor.
Closed : $\{Dd, Os, H, Br\}$

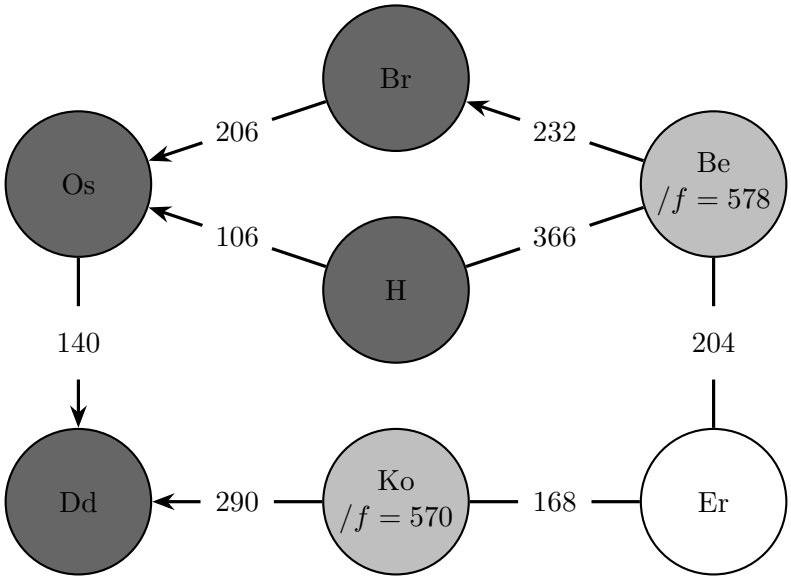


Abbildung 4.6: Schritt 4: Nach der Relaxation von Bremen (Br)

Nächster Schritt:
Da f_{Ko} kleiner ist als f_{Be}, wird Ko als nächster Knoten expandiert.

Nach Expansion von Ko:
$Open : \{Be, Er\}$ mit $f(Be) = 578 und f(Er) = 632$, da $f_{Be} = g(Br) + c(Br, Be) + h(Be) = 346 + 232 + 0 = 578$ und $f_{Er} = g(Ko) + c(Ko, Er) + h(Er) = 290 + 168 + 174 = 632$.
$Closed : \{Dd, Os, H, Br, Ko\}$

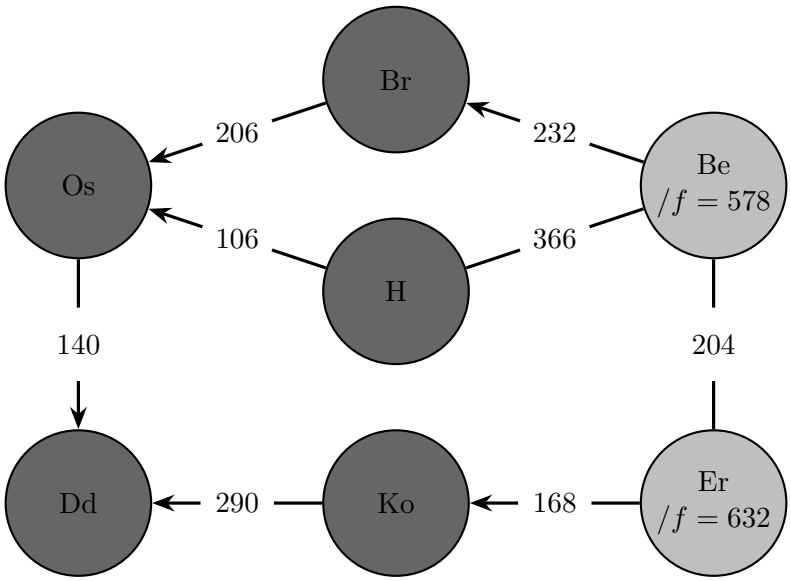


Abbildung 4.7: Schritt 5: Nach der Relaxation von Köln (Ko)

Nächster Schritt:
Da f_{Be} kleiner ist als f_{Er}, wird Be als nächster Knoten expandiert, welches das Ziel ist und somit ergibt sich als kürzester Weg: Düsseldorf (Dd) - Osnabrück (Os) - Bremen (Br) - Berlin (Be).

Finalzustand:

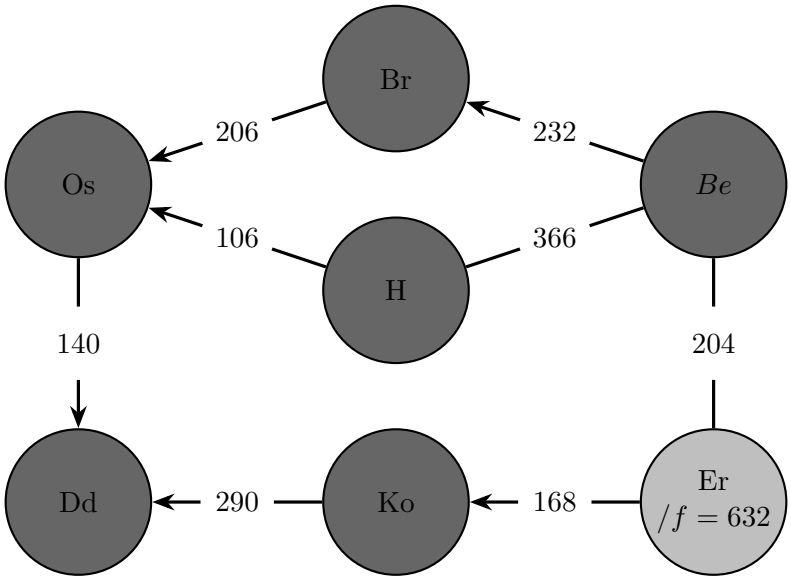


Abbildung 4.8: Ergebnis: kürzester Weg von Düsseldorf (Dd) nach Berlin (Be)

4.3 Eigenschaften

Notation

Im Folgenden stehen einige Definitionen, die wir zur Analyse der Eigenschaften benötigen [uDDSa]:
$g^*(N, N')$ = Kosten des optimalen Weges von N nach N'
$g^*(N)$ = Kosten des optimalen Weges vom Start bis zu N
$c^*(N)$ = Kosten des optimalen Weges von N bis zum nächsten Zielknoten Z.
$f^*(N) = g^*(N) + c^*(N)$
(Kosten des optimalen Weges durch N bis zu einem Zielknoten Z)

Definition 1 *: Eine Schätzfunktion $h(.)$ bezeichnet man als unterschätzend, gdw. $h(N) \leq c^*(N)$ für jeden Knoten gilt, also die Schätzfunktion h die Kosten unterschätzt [uDDSa].*

Das ist eine wichtige Eigenschaft des A^*-Algorithmus für die Bestimmung des kürzesten Weges in einem Graphen.

Definition 2 *: Voraussetzungen für den A^*-Algorithmus:*

1. *es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei $d = inf\{$ Kosten aller Wege von S zu einem Zielknoten $Z\}$.*
2. *Für jeden Knoten N gilt: $h(N) \leq c^*(N)$, d.h. die Schätzfunktion ist unterschätzend.*

3. *Für jeden Knoten N ist die Anzahl der Nachfolger endlich.*

4. *Alle Kanten kosten etwas: $c(N, N') \succ 0$ für alle N, N'.*

5. *Der Graph ist schlicht*
   (zwischen zwei Knoten gibt es höchstens eine Kante)[uDDSb].

Die Voraussetzung (1) der Endlichkeit der Knoten und (4) sind z.B. erfüllt, wenn es eine untere Schranke $\delta > 0$ für die Kosten einer Kante existiert (Beweise können in den meisten Einführungsbücher der Künstlichen Intelligenz gefunden werden).

Bemerkung 1 *: Der Dijkstra-Algorithmus gleicht dem A^* -Algorithmus , wenn man die Schätzfunktion $h(N) = 0$ setzt für alle Knoten N. Dieser besitzt die Aufgabe, einen Weg mit minimalen Kosten zwischen zwei Knoten eines gegebenen endlichen Graphen zu finden. Damit hat man immer eine Front (OPEN) der Knoten mit minimalen Wegen ab dem Starknoten [uDDSa].*

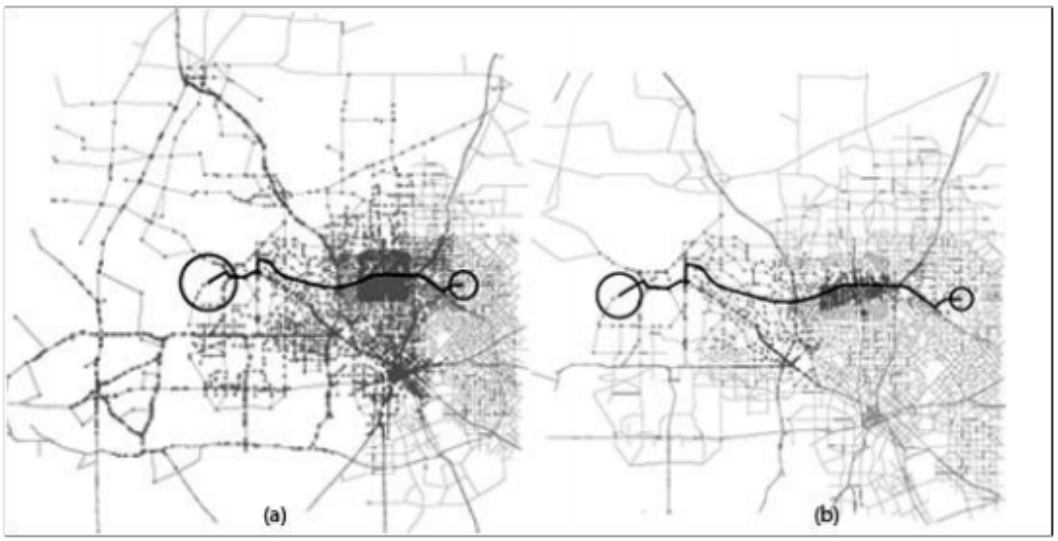


Abbildung 4.9: (a) Dijkstra's-Suche (b) A^*-Suche auf einem Straßennetz vom Dallas Ft-Worth Stadtgebiet [5]

Abbildung 4.9 zeigt, dass die A^*-Suche ergebniswirksamer ist mit der Anzahl besuchter Knoten. Folglich sucht der Dijkstra-Algorithmus wesentlich mehr Knoten als der A^* tut.

Definition 3 *: Wenn man zwei Schätzfunktionen h_1 und h_2 hat mit:*

a) h_1 und h_2 unterschätzen den Aufwand zum Ziel

b) für alle Knoten N gilt: $h_1(N) \leq h_2(N) \leq c^*(N)$

[5]Die Quelle für die Abbildungen ist `http://eprints.uthm.edu.my/7478/1/AMANI_SALEH_ALIJA.pdf`

dann nennt man h_2 besser informiert als h_1 [uDDSb].

Aus der soeben erwähnten Aussage kann man noch nicht ableiten, dass der A^*-Algorithmus sich zu h_2 sich besser verhält als zu h_1.
Erforderlich ist: Die Abweichung bei Sortierung der Knoten mittels f muss klein sein.
D.h. optimal wäre $f(k) \leqslant f(k') \Longleftrightarrow f^*(k) \leqslant f^*(k')$.
Dann entdeckt der Algorithmus zu h_2 höchstens die gleiche Anzahl von Knoten wie h_1 [Wel].
Aufgrund von Informiertheit gibt es keinen besseren Ansatz als den A^*-Algorithmus, in dem Sinne, dass keine andere Suchstrategie mit Zugang zum gleichen Maß von externen Wissen weniger Arbeit leisten kann als der A^*-Algorithmus und sich darauf verlassen kann, die optimale Lösung zu finden [SH06].

Definition 4 *: Eine Schätzfunktion $h(.)$ ist monoton, gdw.*
$h(N) \leq c(N, N') + h(N')$ für alle Knoten N und deren Nachfolger N' und wenn $h(Z) = 0$ ist für Zielknoten Z [uDDSb].

Lemma 1 *: Eine monotone Schätzfunktion $h(.)$ ist unterschätzend [uDDSb].*

Die Monotonie ist somit eine kräftigere Eigenschaft als die Unterschätzung. Sie unterscheiden sich nur in dem Punkt, dass ein bereits expandierter Knoten mit einer niedrigeren Bewertung $f(n)$ ein zweites Mal während der Suche erreicht wird, bei einem A^*-Algorithmus mit Monotoniebedingung jedoch niemals in Erscheinung treten kann. Der A^*-Algorithmus ist mit monotoner Heuristik unter Verwendung von Graphsuche optimal. Eine passende und einfache Heuristik für eine Suche hierbei wäre die Luftlinie. Die tatsächliche Entfernung ist nie kürzer als die direkte Verbindung [PXJW].

Die Monotonie der Schätzfunktion gleicht der Dreiecksungleichung in Geometrien und metrischen Räumen.
Die Monotonieeigenschaft gibt das Verhalten der Funktion f beim Expandieren wieder:
Wenn die Schätzfunktion monoton ist, dann nimmt der Wert von f monoton zu, wenn man einen Weg weiter expandiert.
Die Eigenschaften der A^*-Suche sind ebenso auch seine Vorteile. Die A^*-Suche ist:

- korrekt, d.h. das Suchergebnis, das die A^*-Suche übermittelt, ist eine Lösung des Suchproblems
- vollständig, d.h. wenn es eine Lösung des Suchproblems gibt, so wird diese auch gefunden
- optimal, d.h. dass die Lösung des Suchproblems der kürzeste Pfad zum Zielknoten darstellt [Bau]
- optimal effizient, „d.h. jeder andere optimale und vollständige Algorithmus, der dieselbe Heuristik benutzt, muss mindestens so viele Knoten anschauen wie A^*, um eine Lösung zu finden.“[Kön10]

Sein größter Nachteil ist, ähnlich wie bei der Breitensuche, seine belastende Speicherplatzkomplexität mit $O(Branchingfaktor^{Suchtiefe})$ [WS04]. Ein weiterer Nachteil ist seine Laufzeit, die immer abhängig von der Qualität der Heuristik ist. Bei einer höheren (niedrigeren) gewünschten Qualität ist auch eine höhere (niedrigere) Laufzeit zu erwarten. Es gibt perfekte Heuristiken, welche bewirken, dass der Algorithmus nur die Knoten überschreitet, die auch auf dem Lösungspfad liegen. Diese Heuristiken machen in der Praxis aber wenig Sinn, da der Aufwand für ihre Berechnung unproportional hoch ist. Als Gegenteil gibt es Heuristiken, welche kaum Rechenaufwand brauchen (vgl. erste Heuristik im Beispiel des 8-Puzzles dieser Arbeit 3.2), bei welchen der Algorithmus aber kaum schneller ist als der Dijkstra. Eine gute Heuristik erfasst den Mittelweg zwischen perfekter und schlechter Performanz. Unabhängig von der Heuristik (solange sie unterschätzend ist), verfügt A^* in jedem Fall über eine bessere Laufzeit als der Dijkstra-Algorithmus, welcher eine Komplexität von $O(n^2)$ besitzt. Unter Benutzung von Fibonacci-Heaps kann auch eine Laufzeit von $O(n \cdot log(n))$ erlangt werden [JL].

KAPITEL

FÜNF

Spezialisierung des A^*-Algorithmus

Es ist eine Spezialisierung des A^*-Algorithmus gegeben. Wir betrachten den A^*-Algorithmus mit der Gewichtsfunktion Zeit. Das bedeutet, wir interessieren uns nicht für den kürzesten Weg zweier Punkte, wie es in den meisten Lehrbüchern beschrieben wird, sondern für den schnellsten Weg zwischen diesen beiden. Hierfür haben wir ein Verkehrswegenetz mit den Verkehrsmitteln Auto, Bahn und Flugzeug modelliert. Die Knoten bestehen aus Straßen, Bahnhöfe und Flughäfen. Die Straße soll später einen Zufallspunkt darstellen. Die Kanten sind Verbindungen zwischen den Straßen, Bahnhöfen und Flughäfen. Es gibt drei Arten von Kanten: langsame Kanten, die mit dem Auto zurückgelegt werden, mittelschnelle Kanten, auf denen eine Bahn fährt und schnelle Kanten, auf denen ein Flugzeug fliegt. Da wir drei Verkehrsmittel haben, gibt es auch drei Geschwindigkeiten: Auto = v_A, Bahn = v_B und Flugzeug = v_F. Gesucht ist bei Eingabe von zwei Punkten die optimale Verbindung. Optimale Verbindungen können entweder die schnellste Verbindung sein oder die billigste, in unseren Fällen immer die schnellste. Das Suchverfahren, das angewendet werden soll, ist der A^*-Algorithmus mit verschiedenen Varianten, Schätzfunktionen und Parametern.

5.1 Varianten

Es gibt zwei Fälle, zwischen denen unterschieden wird. Der erste Fall besteht darin, mit der Bahn und dem Flugzeug den schnellsten Weg von Knoten zu Knoten zu suchen.
Beim zweiten Fall sind Start- und Endpunkt beliebig, also evtl. ohne irgendeine Kante. Die Annahme ist dann immer, dass es zu jedem Knoten eine Straße gibt. Hier gibt es folgendes Szenario: zum Knoten K wird der nächste (Luftlinie) Bahnhof bzw. Flugplatz gesucht. Diese Szenario wird ausprobiert und dokumentiert.
Außerdem werden auch Parameter wie z.B. Umsteigezeiten bzw. mittlere Zeiten zwischen Abfahrten verschiedener Verkehrsmittel auf die Suche angewandt und variiert.
Zu den einzelnen Fällen muss natürlich auch eine passende (unterschätzende und monotone) Heuristik gewählt werden, auf die wir im nächsten Abschnitt näher eingehen werden.

Weiterhin werden auch die mittleren Geschwindigkeiten der verschiedenen Verkehrsmittel geändert und getestet.

5.1.1 Multiplizierer

Ein gegebenes Liniennetz besteht aus etwa 150 Knoten (Standorte). Um den Algorithmus nun auf Herausforderungen zu stellen, werden mehrere Knoten dadurch erstellt, dass eine Kopie der Karte mit einer Erweiterung um einen bestimmten Faktor des Längen- und Breitengrades erstellt wird. Die Funktion des Kopierens („Multiplizierer“) dient zum Vereinfachen von Knotenerstellung. Der Aufwand für ganz Europa einzutragen wäre viel zu groß. Für den „Multiplizierer“ gilt: Zwischen den Netzen gibt es dann Verbindungen zu den Flughäfen, also kann man von jedem Flughafen eines Netzes zu jedem anderen Flughafen des selben Netzes oder eines anderen Netzes fliegen. Abbildung 5.1 veranschaulicht ein Beispiel wie ein Flughafen einen anderen Flughafen erreichen kann. Die Karte mit dem Wert 0 zeigt das Ausgangsnetz und die mit der 1 das gespiegelte bzw. kopierte Netz.

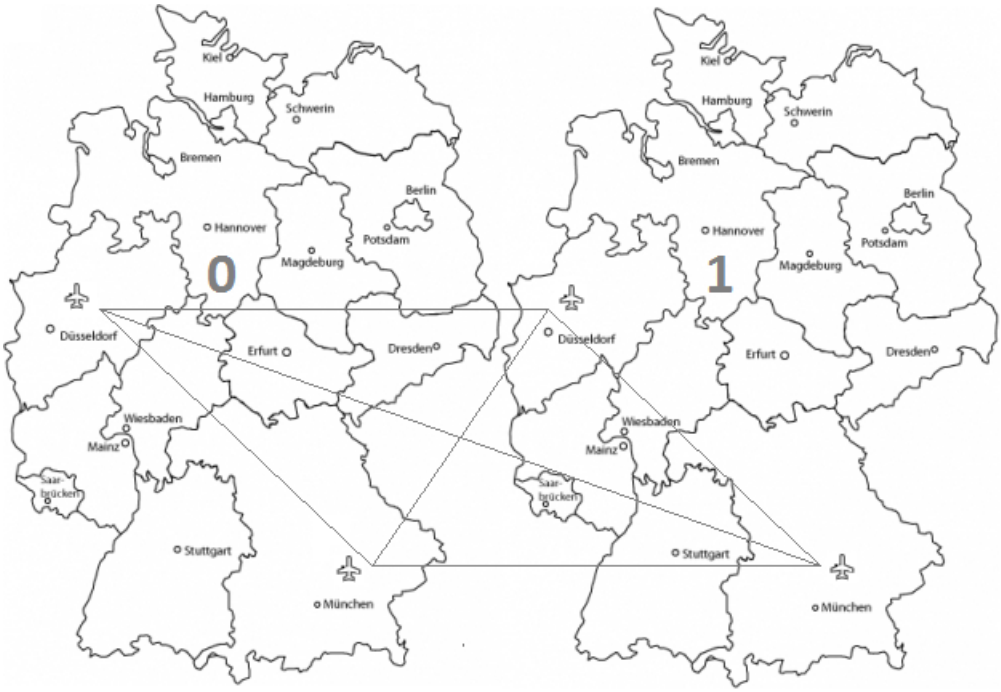


Abbildung 5.1: Spiegeln des Ausgangsnetzes von Deutschland [1]

[1] Die Quelle für die Abbildungen ist `http://www.fernwisser.de/2011/08/25/deutschlandkarte-eps-als-kostenloser-download/`

Werden mehrere Kopien erstellt, sieht die Karte dann folgendermaßen aus:

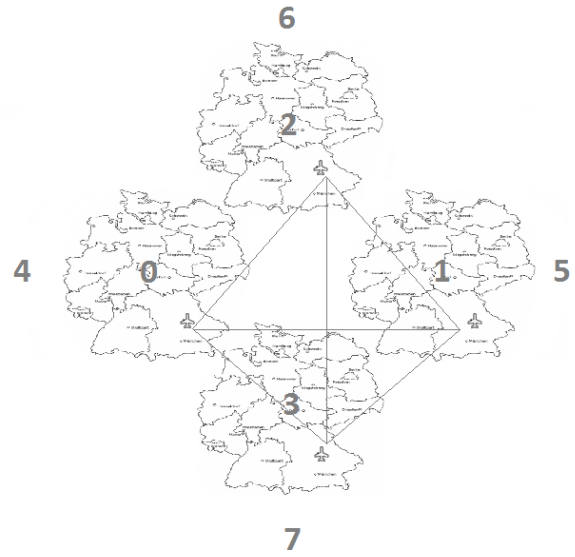


Abbildung 5.2: Mehrmaliges Spiegeln des Ausgangsnetzes Deutschlands [1]

Abbildung 5.2 zeigt ein Viermaliges Spiegeln des Ausgangsnetzes Deutschlands mit der entsprechenden Zahl der Kopie und wo sich dementsprechend die nächste Kopien befinden würden.

5.1.2 Wartezeit

Eine weitere Herausforderung für den A^*-Algorithmus ist es auch, wenn wir fixe Wartezeiten einbauen. Zu Beginn werden wir ohne Wartezeiten testen und danach vergleichen wir die Ergebnisse mit den fixen Wartezeiten. Fixe Wartezeiten wird es an jedem Flughafen und an jedem Großbahnhof geben. Die Großbahnhöfe sind: Frankfurt (M) Hbf, Berlin Hbf, Hannover, Köln Hbf, Stuttgart, München Hbf, Nürnberg, Hamburg Hbf, Düsseldorf Hbf und Essen und alle kopierten Bahnhöfe, die diesen Namen tragen, also z.B. München Hbf und München Hbf_1. Die Wartezeiten werden nicht nur in der Komponente g, sondern auch in h enthalten sein. Die genaue Zusammensetzung aus der Suche und der Wartezeit werden wir in der Implementierung erläutern. Natürlich werden diese Wartezeiten dann auch variiert, um zu sehen wie sehr sie den A^*-Algorithmus beeinflussen.

5.2 Heuristik

Wir haben im vorherigen Kapitel gesehen, wie wichtig Schätzfunktionen sind und welche Eigenschaften erfüllt werden müssen, und zwar muss eine Schätzfunktion h unterschätzend und monoton sein, damit der optimalste (kürzeste oder schnellste)

Weg gefunden werden kann. Für die Suche werden drei verschiedene Heuristiken verwendet und eine weitere Heuristik nur theoretisch besprochen.

- Luftlinienheuristik
- *Advanced* Heuristik
- Nullheuristik
- *Abstract* Heuristik

Im Folgenden werden einige Bezeichnungen aufgelistet, die wir zur Klärung der Heuristiken benötigen:
Mit $ll()$ bezeichnen wir den Luftlinienabstand. Es gilt wegen der Dreiecksungleichung: $ll(N, Z) < ll(N, N') + ll(N', Z)$. Da der echte Abstand stets größer als Luftlinie ist, gilt: $ll(N, N') \leq c(N, N')$, und mit der Bezeichnung $h(\cdot) := ll(\cdot)$ gilt $h(N) \leq c(N, N') + h(N')$.

- $ll(N, N_F)$ = berechnet die Luftlinie zwischen den Knoten N und den am nächsten Flughafen von N
- $ll(Z, Z_F)$ = berechnet die Luftlinie zwischen den Zielknoten Z und den am nächsten nächsten Flughafen von Z
- $ll(N, N_B)$ = berechnet die Luftlinie zwischen den Knoten N und den am nächsten Bahnhof von N
- $ll(Z, Z_B)$ = berechnet die Luftlinie zwischen den Zielknoten Z und den am nächsten Flughafen von Z
- $ll(N, Z)$ = berechnet die Luftlinie zwischen den Knoten N und den Zielknoten Z
- $ll(N_B, N_F)$ = berechnet die Luftlinie zwischen den am nächsten Bahnhof von N und den am nächsten Flughafen von N
- $ll(Z_B, Z_F)$ = berechnet die Luftlinie zwischen den am nächsten Bahnhof von Z und den am nächsten Flughafen von Z
- $ll(N, N_{FB})$ = berechnet die Luftlinie zwischen den am nächsten Standort (Flughafen oder Bahnhof) von N und den am nächsten Standort (Flughafen oder Bahnhof) von N
- $ll(Z, Z_{FB})$ = berechnet die Luftlinie zwischen den am nächsten Standort (Flughafen oder Bahnhof) von Z und den am nächsten Standort (Flughafen oder Bahnhof) von Z
- v_A = mittlere Geschwindigkeit des Autos
- v_B = mittlere Geschwindigkeit der Bahn
- v_F = mittlere Geschwindigkeit des Flugzeugs

Luftlinienheuristik:
$h_1(N) = ll(N, Z)/v_F$. D.h. nehme die Luftlinie direkt mit dem Flugzeug.

Advanced Heuristik:
Die *Advanced* Heuristik $h_2(N, Z)$ sieht folgendermaßen aus: Man sucht die nächsten Flughäfen N_F zu N und Z_F zu Z. Dann nimmt man an, dass die Flughäfen auf den Luftlinien liegen. Hierfür muss aber eine Fallunterscheidung gelten; einmal für die Variante vom Bahn und Flugzeug und einmal für die Variante vom Auto, Bahn und Flugzeug.

Die Heuristik $h_2(N, Z)$ für Bahn und Flugzeug sieht genauer folgendermaßen aus:

1. Wenn $ll(N, N_F) + ll(Z, Z_F) \geq ll(N, Z)$, dann $h_2(N, Z) = ll(N, Z)/v_B$. D.h. Bahnfahren ist schneller

2. Wenn $ll(N, N_F) + ll(Z, Z_F) < ll(N, Z)$, dann
   $h_2(N, Z) = (ll(N, N_F)+ll(Z, Z_F))/v_B+(ll(N, Z)-(ll(N, N_F)+ll(Z, Z_F)))/v_F$

Die Heuristik $h_2(N, Z)$ für Auto, Bahn und Flugzeug sieht genauer folgendermaßen aus:

1. Wenn $ll(N, N_B) + ll(Z, Z_B) \geq ll(N, Z)$ und $ll(N, N_F) + ll(Z, Z_F) \geq ll(N, Z)$, dann $h_2(N, Z) = ll(N, Z)/v_A$. D.h. Autofahren ist schneller

2. Wenn $ll(N, N_B) + ll(Z, Z_B) < ll(N, Z)$ oder $ll(N, N_F) + ll(Z, Z_F) < ll(N, Z)$, dann

   2.1 falls $ll(N, N_B) + ll(N_B, N_F) + ll(Z_B, Z_F) + ll(Z, Z_B) \geq ll(N, Z)$, dann $h_2(N, Z) = ll(N, Z)/v_A$, sonst $h_2(N, Z) = (ll(N, N_B) + ll(Z, Z_B))/v_A +$
   $((ll(N, N_F)-ll(N, N_B))+(ll(Z, Z_F)-ll(Z, Z_B)))/v_B+(ll(N, Z)-(ll(N, N_F)$
   $+ll(Z, Z_F)))/v_F$

   2.2 falls $ll(N, N_B)+ll(N_B, N_F)+ll(Z, Z_F) \geq ll(N, N_F)+ll(Z_B, Z_F)+ll(Z_B, Z)$, dann

      2.2.1 falls $ll(N, N_F) + ll(Z_F, Z_B) + ll(Z_B, Z) \geq ll(N, Z)$, dann $h_2(N, Z) = ll(N, Z)/v_A$ sonst $h_2(N, Z) = (ll(Z, Z_B)+ll(N, N_F))/v_A + ll(Z, Z_F)-$
      $ll(Z, Z_B)/v_B + (ll(N, Z) - (ll(N, N_F) + ll(Z, Z_F)))/v_F$

      2.2.2 falls $ll(N, N_B)+ll(N_B, N_F)+ll(Z_F, Z) \geq ll(N, Z)$, dann $h_2(N, Z) = ll(N, Z)/v_A$ sonst $h_2(N, Z) = (ll(N, N_B)+ll(Z, Z_F))/v_A+(ll(N, N_F)-$
      $ll(N, N_B))/v_B + (ll(N, Z) - (ll(N, N_F) + ll(Z, Z_F)))/v_F$

   2.3. falls $ll(N, N_B) + ll(Z_B, Z) \geq ll(N, N_F) + ll(Z_F, Z)$, dann

      2.3.1 falls $ll(N, N_B) + ll(Z_B, Z) \geq ll(N, Z)$, dann $h_2(N, Z) = ll(N, Z)/v_A$ sonst $h_2(N, Z) = (ll(N, N_B)+ll(Z, Z_B))/v_A+(ll(N, Z)-(ll(N, N_B)+$
      $ll(Z, Z_B)))/v_B$

      2.3.2 falls $ll(N, N_F) + ll(Z_F, Z) \geq ll(N, Z)$, dann $h_2(N, Z) = ll(N, Z)/v_A$ sonst $h_2(N, Z) = (ll(N, N_F)+ll(Z, Z_F))/v_A+(ll(N, Z)-(ll(N, N_F)+$
      $ll(Z, Z_F)))/v_F$

Die Reihenfolge ist so gewählt, damit wir 1. schnell ans Ziel gelangen und 2. damit wir so viele Verkehrsmittel ausschöpfen, um möglichst keine Lücke auf der Luftlinie zwischen N und Z zu besitzen.

Nullheuristik:
Die Nullheuristik $h_3(N)$ ist die Heuristik, bei der $h_3(N) = 0$ ist. Der Dijkstra-Algorithmus zum Finden optimaler Wege in einem Graphen entspricht (wenn man von der Speicherung des optimalen Weges absieht) dieser Variante.

Abstract Heuristik:
Die *Abstract* Heuristik $h_4(N, Z)$ ist eine abstrakte und vereinfachtere Heuristik als die *Advanced* Heuristik für den Fall Auto/Bahn/Flugzeug. Sie ist im Grunde genommen eine Verbindung aus der *Advanced* Heuristik von Bahn/Flugzeug und Auto/-Bahn/Flugzeug. Man sucht den nächsten Flughafen oder Bahnhof N_{FB} zu N und Z_{FB} zu Z. Dann nimmt man die Luftlinie direkt mit dem Flugzeug. Hierfür muss aber eine Fallunterscheidung gelten und zwar:

1. Wenn $ll(N, N_{FB}) + ll(Z, Z_{FB}) \geq ll(N, Z)$, dann $h_4(N, Z) = ll(N, Z)/v_A$. D.h. Autofahren ist schneller
2. Wenn $ll(N, N_{FB}) + ll(Z, Z_{FB}) < ll(N, Z)$, dann $h_4(N, Z) = (ll(N, N_{FB}) + ll(Z, Z_{FB}))/v_A + (ll(N, Z) - (ll(N, N_{FB}) + ll(Z, Z_{FB})))/v_F$

Besser informierte Heuristik:
Behauptung: Heuristik h_2 ist besser informiert als Heuristik h_1, h_3 und h_4. Heuristik h_4 ist besser informiert als Heuristik h_1 und h_3. Heuristik h_1 ist besser informiert als h_3.

5.2.1 Monotonie

Wir wissen, es reicht nicht aus eine Schätzfunktion nur aufzustellen. Wir müssen zeigen, dass diese unterschätzend und monoton ist.

Behauptung: Alle vier Heuristiken sind unterschätzend und monoton.

Beweis:

1) Luftlinienheuristik:
Klar ist, dass diese unterschätzend ist. Sie ist auch monoton: Der Weg von N über N' nach Z ist wegen der Dreiecksungleichung länger als die Luftlinie von N nach Z, also $ll(N, Z) \leq ll(N, N') + ll(N', Z)$ und da $ll(N, N') \leq c(N, N')$ gilt auch $ll(N, Z) \leq c(N, N') + ll(N', Z)$. Rechnet man die Zeit, dann gilt auch $ll(N, Z)/v_F \leq c(N, N')/v_F + ll(N', Z)/v_F \leq c(N, N')/v_B + ll(N', Z)/v_F$.

2) *Advanced* Heuristik:
Für Bahn/Flugzeug:
Sei N' der nächste Knoten. Dazu sei N'_F der nächste Flughafen. Wenn N selbst schon der Flughafen ist, dann ist die Rechnung einfach, also nehmen wir an, dass N kein

Flughafen ist. Es gilt, dass $ll(N, N_F) \leq ll(N, N') + ll(N', N'_F)$ da N_F der nächste Flughafen zu N ist. Bei gleichem Abstand N, N' ist $h_2(N', Z)$ minimal, wenn N' auf der Luftlinie N, Z liegt. Dann ist auf jeden Fall die Flugstrecke die in $h_2(N, Z)$ eingeht, länger als die die in $h_2(N', Z)$ eingeht. Es reicht somit aus, die Rechnung so zu machen, als sei N' und N'_F auf der Luftlinie N, Z, und erhält nach kurzer Rechnung die Abschätzung $h_2(N, Z) \leq c(N, N') + h_2(N', Z)$, also die Monotonie.

Für Auto/Bahn/Flugzeug:
zu 2.1/2.2.2): Sei N' irgendein nächster Punkt. Dazu sei N'_B der nächste Bahnhof. Es gilt, dass $ll(N, N_B) \leq ll(N, N') + ll(N', N'_B)$ da N_B der nächste Bahnhof zu N ist. Bei gleichem Abstand N, N' ist $h_2(N', Z)$ minimal, wenn N' auf der Luftlinie N, Z liegt. Dann ist auf jeden Fall die Bahnstrecke, die in $h_2(N, Z)$ eingeht, länger als die, die in $h_2(N'Z)$ eingeht. Sei N'_F der nächste Flughafen. Es gilt, dass $ll(N, N_F) \leq ll(N, N') + ll(N', N'_F)$, da N_F der nächste Flughafen zu N ist. Bei gleichem Abstand N, N' ist $h_2(N', Z)$ minimal, wenn N' auf der Luftlinie N, Z liegt. Dann ist auf jeden Fall die Flugstrecke die in $h_2(N, Z)$ eingeht, länger als die, die in $h_2(N', Z)$ eingeht. Es reicht somit aus, die Rechnung so zu machen, als sei N', N'_B und N'_F auf der Luftlinie N, Z, und erhält nach kurzer Rechnung die Abschätzung $h_2(N, Z) \leq c(N, N') + h_2(N', Z)$, also die Monotonie.

zu 2.2.1/2.3.2): Sei N' irgendein nächster Punkt. Dazu sei N'_F der nächste Bahnhof. Es gilt, dass $ll(N, N_F) \leq ll(N, N') + ll(N', N'_F)$ da N_F der nächste Bahnhof zu N ist. Bei gleichem Abstand N, N' ist $h_2(N', Z)$ minimal, wenn N' auf der Luftlinie N, Z liegt. Dann ist auf jeden Fall die Flugstrecke, die in $h_2(N, Z)$ eingeht, länger als die, die in $h_2(N'Z)$ eingeht. Es reicht somit aus, die Rechnung so zu machen, als sei N' und N'_F auf der Luftlinie N, Z, und erhält nach kurzer Rechnung die Abschätzung $h_2(N, Z) \leq c(N, N') + h_2(N', Z)$, also die Monotonie.

zu 2.3.1): Sei N' irgendein nächster Punkt. Dazu sei N'_B der nächste Bahnhof. Es gilt, dass $ll(N, N_B) \leq ll(N, N') + ll(N', N'_B)$ da N_B der nächste Bahnhof zu N ist. Bei gleichem Abstand N, N' ist $h_2(N', Z)$ minimal, wenn N' auf der Luftlinie N, Z liegt. Dann ist auf jeden Fall die Bahnstrecke, die in $h_2(N, Z)$ eingeht, länger als die, die in $h_2(N'Z)$ eingeht. Es reicht somit aus, die Rechnung so zu machen, als sei N' und N'_B auf der Luftlinie N, Z, und erhält nach kurzer Rechnung die Abschätzung $h_2(N, Z) \leq c(N, N') + h_2(N', Z)$, also die Monotonie.

3) Nullheuristik:
Behauptung: h_3 ist monoton. Klar ist, dass diese unterschätzend ist.
Sie ist auch monoton: Der Weg von N über N' nach Z ist immer länger als 0 aufgrund der Tatsache, dass wir immer positive Kantengewichte (Kosten) betrachten. Somit gilt $0 \leq c(N, N') + ll(N', Z)$.

4) *Abstract* Heuristik:
Sei N' der nächste Knoten. Dazu sei N'_{FB} der nächste Flughafen oder nächste Bahnhof. Es gilt, dass $ll(N, N_{FB}) \leq ll(N, N') + ll(N', N'_{FB})$ da N_{FB} der nächste Flughafen oder Bahnhof zu N ist. Bei gleichem Abstand N, N' ist $h_4(N', Z)$ minimal, wenn N'

auf der Luftlinie N, Z liegt. Dann ist auf jeden Fall die Flugstrecke, die in $h_4(N, Z)$ eingeht, länger als die, die in $h_4(N', Z)$ eingeht. Es reicht somit aus, die Rechnung so zu machen, als sei N' und N'_{FB} auf der Luftlinie N, Z, und erhält nach kurzer Rechnung die Abschätzung $h_4(N, Z) \leq c(N, N') + h_4(N', Z)$, also die Monotonie.

□

5.2.2 Besser informiert

Wir wollen nun noch die Eigenschaft der besser informierten Heuristik zueinander zeigen.

Behauptung: Heuristik h_2 ist besser informiert als Heuristik h_1, h_3 und h_4. Heuristik h_4 ist besser informiert als Heuristik h_1 und h_3. Heuristik h_1 ist besser informiert als Heuristik h_3.

Beweis:
a) h_2 sei die Heuristik für Bahn/Flugzeug
1) zu zeigen: $h_1 \leq h_2$
$\rightarrow$ 1.1): $ll(N, Z)/v_F \leq ll(N, Z)/v_B$
Unter der Voraussetzung, dass $v_F > v_B$ ist, ist diese Aussage korrekt.

$\rightarrow$ 1.2): $ll(N, Z)/v_F \leq (ll(N, N_F) + ll(Z, Z_F))/v_B$
$+ (ll(N, Z) - (ll(N, N_F) + ll(Z, Z_F)))/v_F$
$\rightarrow ll(N, Z)/v_F \leq (ll(N, N_F) + ll(Z, Z_F))/v_B + ll(N, Z)/v_F -$
$ll(N, N_F)/v_F - ll(Z, Z_F)/v_F$
$\rightarrow 0 \leq (ll(N, N_F) + ll(Z, Z_F))/v_B - ll(N, N_F)/v_F - ll(Z, Z_F)/v_F$
Unter der Voraussetzung, dass $v_F > v_B$ ist, ist diese Aussage korrekt.

Also ist h_2 besser informiert als h_1.

2) zu zeigen: $h_3 \leq h_2$
$\rightarrow$ 2.1): $0 \leq ll(N, Z)/v_B$
Klar.

$\rightarrow$ 2.2): $0 \leq (ll(N, N_F) + ll(Z, Z_F))/v_B + (ll(N, Z) - (ll(N, N_F) + ll(Z, Z_F)))/v_F$
Unter der Voraussetzung, dass $v_F > v_B$ ist, ist diese Aussage korrekt.

Also ist h_2 besser informiert als h_3.

3) zu zeigen: $h_4 \leq h_2$
$\rightarrow$ Bei jeder Betrachtung der Fälle von *Abstract* Heuristik und *Advanced* Heuristik ist die Heuristik von *Abstract* Heuristik kleiner gleich als die der *Advanced* Heuristik, da wir bei der *Abstract* Heuristik immer den nächsten Punkt von N und Z betrachten, egal, ob es ein Bahnhof oder Flughafen ist und dann die Luftlinie mit dem Flugzeug fliegen unter der Annahme, dass $v_F > v_B$ ist.

Also ist h_2 besser informiert als h_4.

4) zu zeigen: $h_4 \leq h_1$
$\rightarrow$ 4.1): $ll(N,Z)/v_F \leq ll(N,Z)/v_B$
Unter der Voraussetzung, dass $v_B > v_F$ ist, ist diese Aussage korrekt.

$\rightarrow$ 4.2): $ll(N,Z)/v_F \leq (ll(N,N_{FB}) + ll(Z,Z_{FB}))/v_B + (ll(N,Z)$
$- (ll(N,N_{FB}) + ll(Z,Z_{FB})))/v_F$
Unter der Voraussetzung, dass $v_B > v_F$ ist, ist diese Aussage korrekt.

Also ist h_4 besser informiert als h_1.

5) zu zeigen: $h_4 \leq h_3$
$\rightarrow$ 5.1): $0 \leq ll(N,Z)/v_B$
Klar.

$\rightarrow$ 5.2): $0 \leq (ll(N,N_{FB}) + ll(Z,Z_{FB}))/v_B + (ll(N,Z) - (ll(N,N_{FB})$
$+ ll(Z,Z_{FB})))/v_F$
Klar.

Also ist h_4 besser informiert als h_3.

6) zu zeigen: $h_3 \leq h_1$
$\rightarrow 0 \leq ll(N,Z)/v_F$
Klar.

Also ist h_1 besser informiert als h_3.

b) h_2 sei die Heuristik für Auto/Bahn/Flugzeug
1) zu zeigen: $h_1 \leq h_2$
$\rightarrow$ 1.1): $ll(N,Z)/v_F \leq ll(N,Z)/v_A$
Unter der Voraussetzung, dass $v_F > v_A$ ist, ist diese Aussage korrekt.

$\rightarrow$ 1.2): $ll(N,Z)/v_F \leq (ll(N,N_B) + ll(Z,Z_B))/v_A + ((ll(N,N_F) - ll(N,N_B)) +$
$(ll(Z,Z_F) - ll(Z,Z_B)))/v_B + (ll(N,Z) - (ll(N,N_F) + ll(Z,Z_F)))/v_F$
$\rightarrow ll(N,Z)/v_F \leq (ll(N,N_B)+ll(Z,Z_B))/v_A+((ll(N,N_F)-ll(N,N_B))+(ll(Z,Z_F)-$
$ll(Z,Z_B)))/v_B + ll(N,Z)/v_F - (ll(N,N_F)/v_F - ll(Z,Z_F)))/v_F$
$\rightarrow 0 \leq (ll(N,N_B) + ll(Z,Z_B))/v_A + ((ll(N,N_F) - ll(N,N_B)) + (ll(Z,Z_F)$
$- ll(Z,Z_B)))/v_B - ll(N,N_F)/v_F - ll(Z,Z_F)/v_F$
Unter der Voraussetzung, dass $v_A > v_B > v_F$ ist, ist diese Aussage korrekt.

$\rightarrow$ 1.3): $ll(N,Z)/v_F \leq (ll(Z,Z_B) + ll(N,N_F))/v_A + ll(Z,Z_F) - ll(Z,Z_B)/v_B +$
$(ll(N,Z) - (ll(N,N_F) + ll(Z,Z_F)))/v_F$
$\rightarrow ll(N,Z)/v_F \leq (ll(Z,Z_B)+ll(N,N_F))/v_A+ll(Z,Z_F)-ll(Z,Z_B)/v_B+(ll(N,Z)/v_F)$
$- (ll(N,N_F)/v_F - ll(Z,Z_F)))/v_F$
$\rightarrow 0 \leq (ll(Z,Z_B) + ll(N,N_F))/v_A + ll(Z,Z_F) - ll(Z,Z_B)/v_B - (ll(N,N_F)/v_F -$
$ll(Z,Z_F)))/v_F$

Unter der Voraussetzung, dass $v_A > v_B > v_F$ ist, ist diese Aussage korrekt.

$\rightarrow$ 1.4): $ll(N,Z)/v_F \leq (ll(N,N_B) + ll(Z,Z_F))/v_A +$
$(ll(N,N_F) - ll(N,N_B))/v_B + (ll(N,Z) - (ll(N,N_F) + ll(Z,Z_F)))/v_F$
$\rightarrow ll(N,Z)/v_F \leq (ll(N,N_B) + ll(Z,Z_F))/v_A + (ll(N,N_F) - ll(N,N_B))/v_B +$
$(ll(N,Z)/v_F - (ll(N,N_F)/v_F - ll(Z,Z_F)))/v_F$
$\rightarrow 0 \leq (ll(N,N_B) + ll(Z,Z_F))/v_A + (ll(N,N_F) - ll(N,N_B))/v_B - (ll(N,N_F)/v_F -$
$ll(Z,Z_F)))/v_F$
Unter der Voraussetzung, dass $v_A > v_B > v_F$ ist, ist diese Aussage korrekt.

$\rightarrow$ 1.5): $ll(N,Z)/v_F \leq (ll(N,N_B) + ll(Z,Z_B))/v_A + (ll(N,Z) -$
$(ll(N,N_B) + ll(Z,Z_B)))/v_B$
Unter der Voraussetzung, dass $v_A > v_B > v_F$ ist, ist diese Aussage korrekt.

$\rightarrow$ 1.6): $ll(N,Z)/v_F \leq (ll(N,N_F) + ll(Z,Z_F))/v_A + (ll(N,Z) -$
$(ll(N,N_F) + ll(Z,Z_F)))/v_F$
Unter der Voraussetzung, dass $v_A > v_B > v_F$ ist, ist diese Aussage korrekt.

Also ist h_2 besser informiert als h_1.

2) zu zeigen: $h_3 \leq h_2$
$\rightarrow$ 2.1): $0 \leq ll(N,Z)/v_A$
Klar.

$\rightarrow$ 2.2): $0 \leq (ll(N,N_B) + ll(Z,Z_B))/v_A + ((ll(N,N_F) - ll(N,N_B)) + (ll(Z,Z_F) -$
$ll(Z,Z_B)))/v_B + (ll(N,Z) - (ll(N,N_F) + ll(Z,Z_F)))/v_F$
Klar.

$\rightarrow$ 2.3): $0 \leq (ll(Z,Z_B) + ll(N,N_F))/v_A + ll(Z,Z_F) - ll(Z,Z_B)/v_B + (ll(N,Z) -$
$(ll(N,N_F) + ll(Z,Z_F)))/v_F$
Klar.

$\rightarrow$ 2.4): $0 \leq (ll(N,N_B) + ll(Z,Z_F))/v_A + (ll(N,N_F) - ll(N,N_B))/v_B + (ll(N,Z) -$
$(ll(N,N_F) + ll(Z,Z_F)))/v_F$
Klar.

$\rightarrow$ 2.5): $0 \leq (ll(N,N_B) + ll(Z,Z_B))/v_A + (ll(N,Z) - (ll(N,N_B) + ll(Z,Z_B)))/v_B$
Klar.

$\rightarrow$ 2.6): $0 \leq (ll(N,N_F) + ll(Z,Z_F))/v_A + (ll(N,Z) - (ll(N,N_F) + ll(Z,Z_F)))/v_F$
Klar.

Also ist h_2 besser informiert als h_3.

3) zu zeigen: $h_4 \leq h_2$
$\rightarrow$ Bei jeder Betrachtung der Fälle von *Abstract* Heuristik und *Advanced* Heuristik

ist die Heuristik von *Abstract* Heuristik kleiner gleich als die der *Advanced* Heuristik, da wir bei der *Abstract* Heuristik immer den nächsten Punkt von N und Z betrachten, egal, ob es ein Bahnhof oder Flughafen ist und dann die Luftlinie mit dem Flugzeug fliegen unter der Annahme, dass $v_F > v_B > v_A$ ist.

4) zu zeigen: $h_4 \leq h_1$
$\rightarrow$ 4.1): $ll(N,Z)/v_F \leq ll(N,Z)/v_A$
Unter der Voraussetzung, dass $v_A > v_F$ ist, ist diese Aussage korrekt.

$\rightarrow$ 4.2): $ll(N,Z)/v_F \leq (ll(N,N_{FB}) + ll(Z,Z_{FB}))/v_A + (ll(N,Z) - (ll(N,N_{FB}) + ll(Z,Z_{FB})))/v_F$
Unter der Voraussetzung, dass $v_A > v_F$ ist, ist diese Aussage korrekt.

Also ist h_4 besser informiert als h_1.

5) zu zeigen: $h_4 \leq h_3$
$\rightarrow$ 5.1): $0 \leq ll(N,Z)/v_A$
Klar.

$\rightarrow$ 5.2): $0 \leq (ll(N,N_{FB})+ll(Z,Z_{FB}))/v_A+(ll(N,Z)-(ll(N,N_{FB})+ll(Z,Z_{FB})))/v_F$
Klar.

Also ist h_4 besser informiert als h_3.

6) zu zeigen: $h_3 \leq h_1$
$\rightarrow 0 \leq ll(N,Z)/v_F$
Klar.

Also ist h_1 besser informiert als h_3. □

5.3 Verwandte Arbeiten

Eine Vielzahl klassischer Graph-Such-Algorithmen wurden zum Lösen des Problems des Berechnen der kürzesten Folge von Übergängen entwickelt, die einen Zielzustand aus einem Initialzustand auf einem gewichteten Graphen erreichen. Die zwei beliebtesten sind Dijkstra-Algorithmus ((Dikstra 1959) und A^* ((P. E. Hart und Raphael 1968)).
Goyal, Mogha, Luthra und Sangwan untersuchten, welche der kürzesten Pfadalgorithmen A^* oder Dijkstra am schnellsten auf realen Straßennetzen läuft. Sie stellten fest, dass der A^*-Algorithmus in vielen Szenarien halb so lange braucht, um den kürzesten Pfad zu finden als Dijkstra unter der Annahme, dass die heuristische Funktion monoton ist [AGS14].
Liang Dai hat sich ebenfalls mit dem schnellsten und kürzesten Pfad-Algorithmus für das Straßennetz beschäftigt. Er stellte eine Karte von Ottawa City mit über 26000 Knoten zusammen. Dieser Graph bestand aus vier Kategorien mit Unterteilung der Straßen. Die verschiedenen Straßentypen besaßen verschiedene maximale

Fahrgeschwindigkeiten. Das Ergebnis bestätigt, dass in einem Straßennetz der A^*-Algorithmus mit Euklidischer Distanz eine bessere Laufzeit erreichen kann als der Dijkstra [AGS05].
Ein weiterere interessante Untersuchung haben Chang, Tai, Yeh, Hsieh und Chang gemacht. Sie beschäftigten sich mit einem *vehicular-ad-hoc-network*- (VANET-) basierend auf A^* (VBA^*) Routenplanungsalgorithmus um die Route mit der kleinsten Fahrtzeit oder dem geringsten Kraftstoffverbrauch abhängig von zwei Echtzeit-Verkehrsinformationen, welche nicht in traditionellen GPS-Navigation-Applikationen verwendet wurden. Die erste Verkehrsinformation ist die aufgenommene Verkehrsinformation des Straßenabschnitts, das das Fahrzeug durchlaufen hat. Es ist der weitere Austausch zwischen Fahrzeugen über einen *IEEE 802.11p wireless link*. Die zweite Verkehrsinformation ist generiert durch Google Maps. Eine GPS-Navigation-Applikation ist dann auf der Android Plattform zur Realisierung VBA^* implementiert. Schließlich werden Simulationen für sechs Routenplanungsalgorithmen durch *VANET simulator The ONE* ausgeführt, in einem verkehrsreichen und einem verkehrsarmen Zeitraum. Kurz gesagt erreicht VBA^* erhebliche Kürzungen sowohl in durchschnittlicher Fahrtzeit als auch in Kraftstoffverbrauch der geplanten Route im Vergleich zu traditionellen Routenplanungsalgorithmen wie Dijkstra. In Zukunft werden umfangreiche Simulationen basierend auf realen Datenverkehr ausgeführt werden, um zu beobachten, ob VBA^* traditionelle Algorithmen übertrifft und inwieweit es dies tut [ICCC13].
Ansonsten werden meist viele Algorithmen zur Anwendung gebraucht wie D^*(Stentz 1995) oder *Real-Time Adaptive* A^* (Sun, Koenig, and Yeoh 2006), die Optimierungen des A^*-Algorithmus sind oder diesen gebrauchen um praxisorientierte Szenarien darzustellen, wenn der Planer mal keine vollständige Information besitzt [FBM10].

KAPITEL

SECHS

Experimentelle Untersuchung eines Verkehrsverbunds

Wir haben in folgenden Richtungen experimentiert:

1. Wir haben versucht mithilfe unserer Heuristiken zu zeigen, dass in jedem Fall diese Heuristiken den schnellsten Weg zwischen zwei Punkten aufzeigen. Dabei soll nicht nur der schnellste Weg zwischen A und B, sondern auch der schnellste Weg zwischen B und A untersucht werden. Der Hinweg und der Rückweg sollten in jedem Fall identisch sein. Das kann nur erreicht werden, wenn die Heuristiken unterschätzend und monoton sind. Hierfür haben wir zahlreiche verschiedene Eingabeinstanzen getestet. Die Instanzen wurden deterministisch oder zufällig je nach Fall erzeugt. Für gegebene Parameter wurde die minimale Zeit für die Suche ausgegeben. Außerdem hat uns die Berechnungszeit und die Expansion der Knoten interessiert. Also wie viel Zeit einer der jeweiligen Heuristiken für die Suche benötigt, bis die schnellste Route gefunden wurde und wie viele Knoten expandiert werden mussten. Wir haben für etwa 150, 1000 und 2000 Knoten folgende Eingabeinstanzen getestet, wobei eine Suche etwa wie folgt aussah: *suche*(Start, Ziel, v_B, v_F, v_A, Wartezeit, h):

   a) Start: Dieser Parameter sagt uns, wo wir starten. Dieser kann beliebige Koordinate besitzen für den Fall Auto/Bahn/Flugzeug oder ein fester Knoten für Bahn/Flugzeug sein.

   b) Ziel: Dieser Parameter sagt uns, wo wir enden. Dieser kann beliebige Koordinate besitzen für den Fall Auto/Bahn/Flugzeug oder ein fester Punkt sein für Bahn/Flugzeug.

   c) v_A: Gibt die mittlere Geschwindigkeit des Autos an

   d) v_B: Gibt die mittlere Geschwindigkeit der Bahn an

   e) v_F: Gibt die mittlere Geschwindigkeit des Flugzeugs an

   f) Wartezeit: Keine Umsteigezeiten oder fixe Umsteigezeiten; es wird getestet, wie sich die Suche auch mit fixen Umsteigezeiten verhält. Dabei haben Flughäfen die größten Wartezeiten, gefolgt von großen Bahnhöfen und den restlichen Bahnhöfen

g) h : Gibt uns die jeweilige Heuristik an: Nullheuristik, *Advanced* Heuristik, Luftlinienheuristik

2. Das Hauptziel dieser Arbeit ist es, zu demonstrieren, dass die *Advanced* Heuristik nicht nur einfach ist, sondern auch bei einem großen Liniennetz den schnellsten Weg zwischen zwei Orten in möglichst kurzer Berechnungszeit bestimmen kann und nicht viele Knoten im Vergleich zu den anderen Heuristiken expandieren muss. Zu diesem Zweck haben wir auch die Parameter wie Geschwindigkeit der Verkehrsmittel oder Wartezeiten verändert und seine durchschnittliche Berechnungszeit mit den anderen Heuristiken für unterschiedliche Start- und Endpunkte verglichen. Der Vergleich der Anzahl expandierender Knoten von *Advanced* Heuristik mit der Nullheuristik ist trivial, da die Suche mit der Nullheuristik aufgrund des nicht vorhanden seins von Heuristik ($h = 0$) durch Expandieren gleichermaßen in allen Richtungen sucht und die Suche mit der *Advanced* Heuristik den Bereich nur in Richtung des Ziels scannt. Jedoch liegt die Vermutung nahe, dass die *Advanced* Heuristik eine etwas hohe Berechnungszeit besitzen wird, dafür aber zielgerichteter sucht als die Nullheuristik, indem sie nach dem möglichst nächsten Flughafen sucht. Aus diesem Grund haben wir noch zum Vergleich die Luftlinienheuristik untersucht, welches vermuten lässt, dass diese auch wenige Knoten expandiert.

KAPITEL
SIEBEN

Implementierung

7.1 Werkzeuge

Für die Entwicklung dieses Projektes wurde die Entwicklungsumgebung *Luna Service Release 2* (4.4.2)[1] verwendet. Die verwendete Programmiersprache war *Java* in der Version 8 Update 31 (Build 1.8.0_31-b13)[2]. Neben Eclipse habe ich das Buildmanagement-Tool *Maven* [3] verwendet, um hauptsächlich möglichst schnell und einfach eigene und fremde Bibliotheken einbinden zu können. Als Visualisierung der Knoten und Kanten habe ich den Diagrammzeichner *yEd Graph Editor* [4] ausgewählt.
Das Graph Framework *JGraphT* [5] half mir Graphen darzustellen. *JGraphT* ist eine freie Java Graphenbibliothek, die mathematische graphentheoretische Objekte und Algorithmen bereitstellt. *JGraphT* unterstützt verschiedene Arten von Graphen, einschließlich gerichtete und ungerichtete Graphen, Graphen mit gewichteten und ungewichteten Kanten und vieles mehr.
Die Städte mit ihren Verbindungen werden in einer Datenbank erstellt. Diese werden mittels *phpmyadmin* [6] mithilfe einer grafischen Oberfläche meiner Daten und den darin enthaltenen Tabellen verwaltet. Welche genauen Attribute die Tabellen enthalten, wird noch in diesem Kapitel erläutert.
Das Verkehrsnetz setzt sich aus dem ICE-Netz der Deutschen Bahn von 2016 [7] und einer Übersichtskarte Deutschlands mit den Internationalen Flughäfen des Bundesamtes für Kartographie und Geodäsie [8] zusammen .

[1] https://eclipse.org/luna/
[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[3] https://maven.apache.org/
[4] https://www.yworks.com/products/yed/
[5] http://jgrapht.org/
[6] https://www.phpmyadmin.net/
[7] https://www.bahn.de/p/view/service/fahrplaene/streckennetz.shtml
[8] https://www.bkg.bund.de/DE/Produkte-und-Services/Shop-und-Downloads/Landkarten/Karten-Downloads/Themenkarten/Themenkarten-2013/Deutschlandkarte-Flughafen.html

7.2 Klassenübersicht

In diesem Abschnitt werden die Pakete und deren Inhalte, die im Rahmen dieser Masterarbeit erstellt und bearbeitet wurden, beschrieben. Im Projekt „Astern“ (Abbildung 7.1) befinden sich die von mir erstellten Pakete mit entsprechenden Javaklassen.

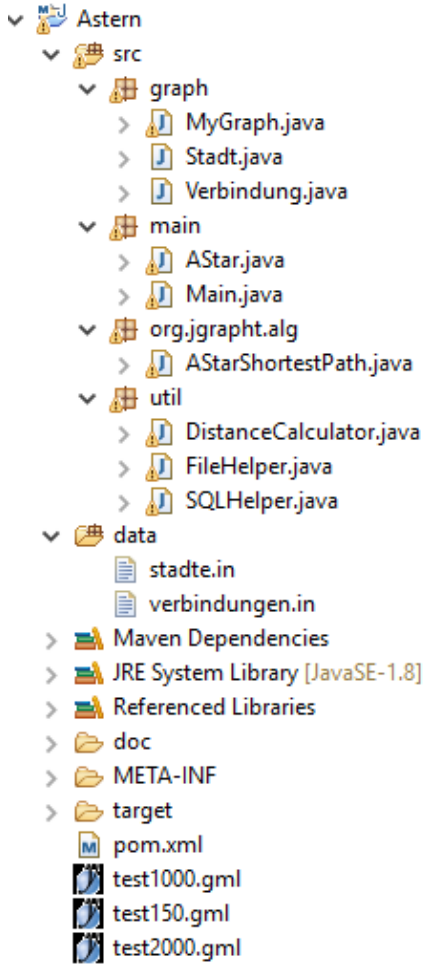


Abbildung 7.1: Das Projekt AStar mit den einzelnen Pakete und deren Klassen

Für die verschiedenen Aufgaben bzw. Bausteine des Grundgerüstes habe ich jeweils ein eigenes Paket angelegt. In den .gml Dateien befinden sich die Visualisierungen der verschiedenen Ausgangsnetze für die Suche.

7.3 Datenbankübersicht

Die Knoten und Verbindungen werden in einer Datenbank abgespeichert. Die Datenbank läuft auf einem *mysql wampserver*. Der Name der Datenbank ist „karte“ . Diese wird in zwei Tabellen unterteilt, in „standort“ und „zusammenhang“ . In „standort“ sind die ganzen Bahnhöfe und Flughäfen gespeichert, in „zusammenhang“ die

Verbindungen und Kosten zwischen den Bahnhöfen und Flughäfen.

7.4 Klassenbeschreibung

In diesem Absatz werden die erstellten Klassen beschrieben. Es werden Quellcodestücke gezeigt, die ich als besonders wichtig empfinde. Dieser Quellcode wird dann im Einzelnen näher besprochen und die Realisierung wird dem Leser verdeutlicht.

7.4.1 MyGraph

Listing 7.1: MyGraph.java Die MyGraph-Klasse

```
1  package graph;
2
3  public class MyGraph<V,E> extends AbstractBaseGraph<Stadt,
       DefaultWeightedEdge> implements DirectedGraph<Stadt,
       DefaultWeightedEdge>{
4
5      public MyGraph(){
6          super(new ClassBasedEdgeFactory<Stadt, DefaultWeightedEdge>(
               DefaultWeightedEdge.class), true, true);
7      }
8
9      protected MyGraph(EdgeFactory<Stadt, DefaultWeightedEdge> ef,
10             boolean allowMultipleEdges, boolean allowLoops) {
11         super(ef, allowMultipleEdges, allowLoops);
12     }
13 }
```

Die *MyGraph*-Klasse stellt unseren Graphen dar. Mithilfe der *MyGraph*()-Methode in Listing 7.1 wird ein neuer Graph erzeugt. Wir verwenden einen gerichteten Graphen, der abhängig von der angegebenen *Edge Factory* ist. Die *Edge Factory* setzt sich aus den Parametern einer Stadt und den gewichteten Kanten zusammen.

7.4.2 Stadt

Listing 7.2: Stadt.java Die Stadt-Klasse

```
1  package graph;
2
3  public class Stadt {
4
5      private String id;
6      private String name;
7      private double laengengrad;
8      private double breitengrad;
9      private String typ;
10     private int riesigbahnhof;
```

```
11
12      public Stadt(String id, String namen, double laenge, double
            breite,
13              String typ) {
14          this.id = id;
15          this.name = namen;
16          this.laengengrad = laenge;
17          this.breitengrad = breite;
18          this.setTyp(typ);
19      }
20
21      public Stadt(String name) {
22          this.name = name;
23      }
24
25      public Stadt() {
26      }
27
28  }
```

In der Klasse „Stadt“ werden die Bahnhöfe und die Flughäfen implementiert. In Listing 7.2 befinden sich die Attribute „id“, „Name“, „Laengengrad“, „Breitengrad“, „Typ“ und „Riesigbahnhof“ einer Station. Das Attribut „Typ“ dient zur Unterscheidung von Bahnhöfen und Flughäfen. „Riesigbahnhof“ sollte später dafür verwendet werden, um Bahnhöfen, die Großbahnhöfe sind, eine höhere Umsteigezeit als anderen Bahnhöfen zuzuordnen. Aufgrund von Zeitmangel wird es hierfür einen anderen Ansatz geben und zwar werden an jedem Bahnhof mögliche Wartezeiten (Umsteigezeiten) auftreten auf die wir jedoch noch später drauf eingehen werden. Der Konstruktor *public Stadt()*, der als Argument drei Strings und zwei Doubles erwartet, mit dem Parameter „id“, „name“, „laenge“, „breite“ und „typ“ dient zur Instanziierung einer Stadt (Station). Beim Aufruf dieses Konstruktors wird somit eine Stadt erstellt.

7.4.3 Verbindung

Listing 7.3: Verbindung.java Die Verbindung-Klasse

```
1  package graph;
2
3  public class Verbindung {
4
5      private Stadt stadt1;
6      private Stadt stadt2;
7      private double dist;
8      private double cost;
9      private double warteBahn;
10     private double warteFlug;
11
```

```
12      public Verbindung(Stadt stadt1, Stadt stadt2, double cost, double
            av,
13              double bv, double fv, double warteBahn, double warteFlug)
                    {
14          this.stadt1 = stadt1;
15          this.stadt2 = stadt2;
16          this.dist = cost;
17
18          this.warteBahn = (warteBahn / 60);
19          this.warteFlug = (warteFlug / 60);
20          this.cost = setcost(cost, av, bv, fv);
21      }
22
23      private double setcost(double cost, double av, double bv, double
            fv) {
24          String typeone = stadt1.getTyp();
25          String typetwo = stadt2.getTyp();
26
27          if (typeone.equals("Flughafen") && typetwo.equals("Flughafen")
                ) {
28              return (cost / fv) + warteFlug;
29          }
30
31          else if (typeone.equals("Auto") || typetwo.equals("Auto")) {
32              return (cost / av);
33          }
34
35          else {
36              return (cost / bv) + warteBahn;
37          }
38
39      }
40
41  }
```

Diese Java-Klasse beinhaltet die Kerneigenschaften von einer Verbindung. Diese Klasse erstellt die Verbindungen der Bahnhöfe und Flughäfen mit den Kosten. In Zeile 12 von Listing 7.3 wird am Konstruktor *Verbindung()* eine Verbindung aus zwei Knoten (Staedte genannt), dessen Kosten, Durchschnittsgeschwindigkeiten der Verkehrsmittel und die Wartezeiten in min von Bahn und Flugzeug, festgesetzt. Da uns in der Arbeit der schnellste Weg interessiert, werden die Kosten dann in Kosten (Entfernung) pro Durchschnittsgeschwindigkeit also in die Größe Zeit umgewandelt und zu den Kosten wird je nachdem, welcher Typ jeweils die Knoten in einer Verbindung sind, Wartezeiten zu den Kosten dazu addiert (Z. 27-37). Die Größe Zeit ist als Weg (Kosten) pro Geschwindigkeit (Durchschnittsgeschwindigkeit) gegeben.

7.4.4 AStar

Diese Java-Klasse beinhaltet im Wesentlichen die Implementierung der verschiedenen Heuristiken und die Wartezeiten an den verschiedenen Stationen des A^*-Algorithmus.

Listing 7.4: AStar.java Die AStar-Klasse (1)

```
1  package main;
2
3  public class AStar {
4
5      public enum HeuristicTyp {
6          SimpleHeuristic, AdvancedHeuristic, NullHeuristic
7      }
8
9      double vAuto = 0;
10     double vFlug = 0;
11     double vBahn = 0;
12     double warteBahn = 0;
13     double warteFlug = 0;
14     public static int expandnodenumbers = 0;
```

Listing 7.4 veranschaulicht die Heuristiken, die in den Tests behandelt werden, nämlich: die Luftlinienheuristik (*SimpleHeuristic*), die selbst entwickelte *Advanced* Heuristik (*AdvancedHeuristic*) und die Nullheuristik (*NullHeuristic*). Weiterhin werden die Geschwindigkeiten und Wartezeiten (Umsteigezeiten) deklariert und initialisiert. Die Variable *expandnodenumbers* dient zur Bestimmung der Anzahl an expandierenden Knoten in der jeweiligen Heuristik und wird zunächst auch deklariert und initialisiert .

Listing 7.5: AStar.java Die AStar-Klasse (2)

```
1  public GraphPath<Stadt, DefaultWeightedEdge> findPathBetween(
2              Graph<Stadt, DefaultWeightedEdge> graph, Stadt start,
                   Stadt end,
3              HeuristicTyp heuristicTyp) {
4
5          AStarShortestPath<Stadt, DefaultWeightedEdge> shortestPath =
               new AStarShortestPath<Stadt, DefaultWeightedEdge>(
6                  graph);
7
8          GraphPath<Stadt, DefaultWeightedEdge> result;
9
10         switch (heuristicTyp) {
11         case AdvancedHeuristic:
12             result = shortestPath.getShortestPath(start, end,
13                     new AdvancedHeuristic(graph.vertexSet()));
14             expandnodenumbers = shortestPath.numberOfExpandedNodes;
15             return result;
```

```
16          case SimpleHeuristic:
17              result = shortestPath.getShortestPath(start, end,
18                     new SimpleHeuristic());
19              expandnodenumbers = shortestPath.numberOfExpandedNodes;
20              return result;
21          case NullHeuristic:
22              result = shortestPath.getShortestPath(start, end,
23                     new NullHeuristic());
24              expandnodenumbers = shortestPath.numberOfExpandedNodes;
25              return result;
26          }
27          return null;
28      }
```

In Listing 7.5 soll der Weg in einem Graphen von einem Startpunkt zu einem Endpunkt unter Angabe der Heuristik gefunden werden. Dabei wird der schnellste Weg und die Anzahl an expandierenden Knoten unter Verwendung der Heuristik ausgerechnet, je nachdem, welche Heuristik gewählt wird.

Listing 7.6: AStar.java Die AStar-Klasse (3)

```
1   class NullHeuristic implements AStarAdmissibleHeuristic<Stadt> {
2
3           public double getCostEstimate(Stadt sourceVertex, Stadt
                targetVertex) {
4               return 0.0;
5           }
6       }
7
8       class SimpleHeuristic implements AStarAdmissibleHeuristic<Stadt>
            {
9           @Override
10
11          public double getCostEstimate(Stadt sourceVertex, Stadt
                targetVertex) {
12              return (DistanceCalculator.getLuftLinie(sourceVertex,
                    targetVertex) / vFlug);
13
14          }
15      }
```

Nun kommen wir genauer auf die Heuristiken des A^* zu sprechen. In Listing 7.6 wird eine Klasse *NullHeuristic* erstellt, die dazu dient, ihre geschätzten Kosten $h(N)$ bis zum Zielknoten für alle Knoten N auf 0 zu setzen. Die nächste Klasse ist die *SimpleHeuristic*. Bei dieser Heuristik wird die Luftlinie zwischen N und dem Zielknoten direkt mit dem Flugzeug genommen, d.h. man nimmt die Luftlinie direkt mit dem Flugzeug.

Listing 7.7: AStar.java Die AStar-Klasse (4)

```
1  class AdvancedHeuristic implements AStarAdmissibleHeuristic<Stadt> {
2
3          @Override
4      public double getCostEstimate(Stadt sourceVertex, Stadt
           targetVertex) {
5
6          Stadt n = sourceVertex;
7          Stadt z = targetVertex;
8          Stadt nf = DistanceCalculator.getNextFlughafen(n, alleStadte);
9          Stadt zf = DistanceCalculator.getNextFlughafen(z, alleStadte);
10         Stadt nb = DistanceCalculator.getNextHaltestelle(n, alleStadte
               );
11         Stadt zb = DistanceCalculator.getNextHaltestelle(z, alleStadte
               );
12
13         double dn_nb = DistanceCalculator.getLuftLinie(n, nb);
14         double dz_zb = DistanceCalculator.getLuftLinie(z, zb);
15         double dn_nf = DistanceCalculator.getLuftLinie(n, nf);
16         double dz_zf = DistanceCalculator.getLuftLinie(z, zf);
17         double distance = DistanceCalculator.getLuftLinie(n, z);
18         double dnb_nf = DistanceCalculator.getLuftLinie(nb, nf);
19         double dzb_z = DistanceCalculator.getLuftLinie(zb, z);
20         double dzf_z = DistanceCalculator.getLuftLinie(zf, z);
21         double dzf_zb = DistanceCalculator.getLuftLinie(zf, zb);
22     }
23     private double nimm(double punkt1, double punkt2, double v) {
24         return (punkt1 + punkt2) / v;
25     }
```

Als letztes haben wir noch die interessante Variante für die Schätzfunktion, nämlich die *AdvancedHeuristic*. Man sucht in Listing 7.7 die nächsten Bahnhöfe N_B (minimalste Distanz eines Bahnhofs zu N) und die nächsten Flughäfen N_F (minimalste Distanz eines Flughafens zu N) zu N (irgendein Knoten) und Z_B (minimalste Distanz eines Bahnhofs zu Z) und Z_F (minimalste Distanz eines Flughafens zu Z) zu Z (Zielknoten). Dann nimmt man an, dass zwischen den Bahnhöfen gefahren werden kann oder die Flughäfen auf der Luftlinien liegen. Um die jeweiligen Kosten zum Ziel abzuschätzen, definieren wir zunächst die Variablen, die wir für die Heuristik benötigen (Z.6-22). Die Variablen initialisieren den Start- und Endpunkt, die nächsten Bahnhöfe und Flughäfen und die Entfernungen zwischen den verschiedenen Punkten. Die *nimm*-Funktion dient als Vereinfachung zur Bestimmung der Gesamtzeit zweier Distanzen.

Listing 7.8: AStar.java Die AStar-Klasse (5)

```
1  double mf = (dn_nf + dz_zf);
2
3      if (mf >= distance) {
```

```
4         return distance / vBahn;
5     }
6
7     else {
8         return mf / vBahn + (distance - mf) / vFlug + 2
9                           * (warteFlug / 60);
10    }
```

Listing 7.8 behandelt den Fall Bahn/Flugzeug. Wir nehmen an, dass jeder Flughafen einen Bahnhof enthält und starten stets in einem Bahnhof. Daher besitzt die Bahn keine Wartezeit.

Listing 7.9: AStar.java Die AStar-Klasse (6)

```
1  boolean gibtAus = true;
2  //1
3  if (((dn_nb + dz_zb) >= distance) && ((dn_nf + dz_zf) >= distance)) {
4      if (gibtAus)
5          return distance / vAuto;
6  }
7  else {
8      //2
9      if (((dn_nb + dz_zb) < distance) || ((dn_nf + dz_zf) < distance)
10         //2.1
11     ){
12         if (gibtAus)
13             return nimm(dn_nb, dz_zb, vAuto)
14                    + nimm(dn_nf - dn_nb, dz_zf - dz_zb, vBahn)
15                    + ((distance - (dn_nf + dz_zf)) / vFlug)
16                    + (warteBahn / 60) + (warteFlug / 60);
17     }
18     else {
19         //2.2
20         if ((dn_nf + dzf_zb + dzb_z) <= (dn_nb + dnb_nf + dz_zf)
21         ){
22             //2.2.1
23             if (((dn_nf + dzf_zb + dzb_z) < distance)) {
24                 if (gibtAus)
25                     return nimm(dz_zb, dn_nf, vAuto)
26                            + ((dz_zf - dz_zb) / vBahn)
27                            + ((distance - (dn_nf + dz_zf)) / vFlug)
28                            + (warteBahn / 60)
29                            + (warteFlug / 60);
30             }
31             //2.2.2
32             else {
33                 if ((dn_nb + dnb_nf + dz_zf) < distance) {
34                     if (gibtAus)
```

```
35                      return nimm(dn_nb, dz_zf, vAuto)
36                              + ((dn_nf - dn_nb) / vBahn)
37                              + ((distance - (dn_nf + dz_zf)) / vFlug)
38                              + (warteBahn / 60)
39                              + (warteFlug / 60);
40              }
41          }
42      }
43      //2.3 und 2.3.1
44      else {
45          if (((dn_nf + dz_zf) < distance)
46                  && ((dn_nf + dz_zf) <= (dn_nb + dz_zb))) {
47                  return nimm(dn_nf, dz_zf, vAuto)
48                          + ((distance - (dn_nf + dz_zf)) / vFlug)
49                          + 2 * (warteFlug / 60);
50          }
51          //2.3 und 2.3.2
52          else {
53              if (((dn_nb + dz_zb) < distance)
54                  && ((dn_nb + dz_zb) < (dn_nf + dz_zf))) {
55                      return nimm(dn_nb, dz_zb, vAuto)
56                              + ((distance - (dn_nb + dz_zb)) / vBahn)
57                              + 2 * (warteBahn / 60);
58              }
59              else{
60
61                          return distance / vAuto;
62              }
63          }
64      }
65  }
66  }
67  }
68  return distance / vAuto;
```

Listing 7.9 implementiert den Fall Auto/Bahn/Flugzeug. Diese Klasse stellt die ganzen möglichen Fälle dar. Außerdem gehen wir in der Heuristik immer von zwei Wartezeiten aus. Sie werden so betrachtet, weil uns die Zeit interessiert. Wenn etwas länger braucht, dann wird sein Schätzwert auch groß. In den Fällen wo wir nur fliegen, warten wir zweimal für den Flug und addieren zweimal die Wartezeit des Flugs oben drauf; in den Fällen wo wir nur Bahn fahren, warten wir zweimal für die Bahn und addieren zweimal die Wartezeit der Bahn oben drauf. Sollte der Fall eintreffen wo wir sowohl fliegen als auch Bahn fahren, dann addieren wir sowohl einmal die Wartezeit des Flugs als auch die Wartezeit der Bahn oben drauf.

7.4.5 Main

Die Klasse Main besteht aus der *main*-Methode, mit der die Ausführung des Programms beginnt und andere wichtige Funktionen wie z.B. der *berechneMultiplierPosition*, der *suche*- und *calculateTime*-Funktion beinhaltet.

Listing 7.10: Main.java Die Main-Klasse (1)

```
1  package main;
2
3  public class Main {
4
5      public static double bahnV = 100;
6      public static double flugV = 1000;
7      public static double autoV = 80;
8      private static double warteBahn = 0;
9      private static double warteFlug = 0;
10     private static boolean randomStart = false;
11     private static boolean randomZiel = false;
12
13     static int max = 142;
14
15     private static HashSet<Stadt> closedstaedte;
```

In Listing 7.10 werden die Daten einiger wichtiger Parameter eingestellt. Die beiden Variablen „randomStart“ und „randomZiel“ geben an, ob es sich um einen zufälligen Start- und Endpunkt handeln soll. Der Parameter „max“ ist fest, da er unsere Anzahl an Stationen auf unserer Ausgangskarte angibt.

Listing 7.11: Main.java Die Main-Klasse (2)

```
1      static int multiplier = 1;
2      private static List<Stadt> stadte;
3      private static List<Verbindung> verbindunge;
4
5      static HashMap<String, Stadt> copyStadte = new HashMap<String,
           Stadt>();
6
7      public static void main(String[] args) throws IOException {
8
9          closedstaedte = new HashSet<Stadt>();
10         FileHelper fileHelper = new FileHelper();
11
12         stadte = fileHelper.getAllCitys();
13         verbindunge = fileHelper.getAllVerbindung(autoV, bahnV, flugV,
14                 warteBahn, warteFlug);
15
16         MyGraph<Stadt, DefaultWeightedEdge> graph = new MyGraph<Stadt,
               DefaultWeightedEdge>();
17         for (Stadt stadt : stadte) {
```

```
18            graph.addVertex(stadt);
19
20            for (int i = 1; i < multiplier; i++) {
21                Stadt copy = new Stadt(stadt.getId(),stadt.getName()+
                      "_" + i, berechneMultiplierPosition(i, "lon", stadt
                      ),berechneMultiplierPosition(i, "lat", stadt),
                      stadt.getTyp());
22                copyStadte.put(stadt.getName() + "_" + i, copy);
23                graph.addVertex(copy);
24                if (stadt.getTyp().equals("Flughafen")){
25                    graph.addEdge(stadt, copy);
26                    DefaultWeightedEdge e = graph.getEdge(stadt, copy);
27                    graph.setEdgeWeight(e, DistanceCalculator.
                          getLuftLinie(stadt, copy));
28
29                    graph.addEdge(copy,stadt);
30                    DefaultWeightedEdge ee = graph.getEdge(copy, stadt)
                          ;
31                    graph.setEdgeWeight(ee, DistanceCalculator.
                          getLuftLinie(copy, stadt));
32                }
33            }
34        }
35        for (Verbindung verbindung : verbindunge) {
36            Stadt st1 = verbindung.getStadt1();
37            Stadt st2 = verbindung.getStadt2();
38            graph.addEdge(st1, st2);
39            DefaultWeightedEdge e = graph.getEdge(st1, st2);
40            graph.setEdgeWeight(e, verbindung.getCost());
41            for (int i = 1; i < multiplier; i++) {
42                Stadt einfugstart = copyStadte.get(st1.getName() + "_"
                       + i);
43                Stadt einfugziel = copyStadte.get(st2.getName() + "_"
                      + i);
44                graph.addEdge(einfugstart, einfugziel);
45                DefaultWeightedEdge ev = graph.getEdge(einfugstart,
                      einfugziel);
46                graph.setEdgeWeight(ev, verbindung.getCost()*i);
47            }
48        }
49        if (multiplier > 1) {
50            verbindeAlleFlughaefen(stadte, graph);
51        }
52         FileWriter w;
53         GmlExporter<Stadt, DefaultWeightedEdge> exporter =
54         new GmlExporter<Stadt, DefaultWeightedEdge>();
```

```
55          exporter.setPrintLabels(GmlExporter.PRINT_VERTEX_LABELS);
56          w = new FileWriter("test150.gml");
57          exporter.export(w, graph);
```

In Listing 7.11 wird beschrieben, wie die Kopie bzw. die Spiegelung des Ausgangsliniennetzes funktioniert und dadurch werden die Knoten und Kanten erstellt. Eine kopierte Station besitzt als Namen den Namen der zu kopierenden Station mit einer Zahl, die von dem *multiplier* (Z.1) abhängig ist und wird um einen bestimmten Faktor im Längen- und Breitengrad verschoben. Der *multiplier* gibt an, wieviele Kopien der Karte gemacht werden, wobei erst bei *multiplier* > 1 angefangen wird Kopien und Verbindungen aller Flughäfen mit den kopierten Flughäfen zu erstellen. In der *main*-Methode werden zu Beginn alle Knoten und Kanten aus der „FileHelper“-Klasse gelesen, die vorher aus der Datenbank geladen wurde. Zu diesen Daten werden wiederum je nachdem, welchen Wert man dem *multiplier* gibt, eine kopierte Anzahl an Karten aus der Hauptkarte hinzugefügt. Dabei kann man wie bereits erwähnt festlegen, um welchen Faktor deren Längen- und Breitengrad erweitert werden soll. Die kopierten Karten werden die selbe Verbindungen untereinander besitzen wie aus der Hauptkarte.
Mithilfe des „GmlExporters“ (Z.60) wird unsere Karte visualisiert und lässt sich in der Desktop Applikation „yEd“ betrachten (siehe Anhang).

Listing 7.12: Main.java Die Main-Klasse (3)

```
1  for (int i = 0; i < 1; i++) {
2             Stadt rand = randomStart? generateRandomPoint() :
3             stadte.get(135);
4             Stadt rand1 = randomZiel? generateRandomPoint():
5             stadte.get(136) ;
6
7             suchAusgabe(graph, rand, rand1, HeuristicTyp.NullHeuristic
                  );
8             expandnum = 0;
9             suchAusgabe(graph, rand, rand1, HeuristicTyp.
                  SimpleHeuristic);
10            expandnum = 0;
11            suchAusgabe(graph, rand, rand1, HeuristicTyp.
                  AdvancedHeuristic);
12        }
13    }
14    private static void verbindeAlleFlughaefen(List<Stadt> stadte,
15            MyGraph<Stadt, DefaultWeightedEdge> graph) {
16        ArrayList<Stadt> flughaefen = new ArrayList<Stadt>();
17
18        for (Stadt x : stadte) {
19            if (x.getTyp().equals("Flughafen")) {
20                flughaefen.add(x);
21            }
22        }
```

```
23          for (Stadt y : flughaefen) {
24              for (int i = 0; i < flughaefen.size(); i++) {
25                  Stadt s = flughaefen.get(i);
26
27                  if (!y.equals(s)) {
28                      Verbindung v = new Verbindung(y, s,
29                              DistanceCalculator.getLuftLinie(y, s), autoV
                                    ,
30                              bahnV, flugV, warteBahn, warteFlug);
31                      verbindunge.add(v);
32                      graph.addEdge(y, s);
33                      DefaultWeightedEdge e = graph.getEdge(y, s);
34                      graph.setEdgeWeight(e,
35                              DistanceCalculator.getLuftLinie(y, s));
36
37                      Verbindung v1 = new Verbindung(s, y,
38                              DistanceCalculator.getLuftLinie(y, s), autoV
                                    ,
39                              bahnV, flugV, warteBahn, warteFlug);
40                      verbindunge.add(v1);
41                      graph.addEdge(s, y);
42                      DefaultWeightedEdge e1 = graph.getEdge(s, y);
43                      graph.setEdgeWeight(e1,
44                              DistanceCalculator.getLuftLinie(s, y));
45                  }
46              }
47          }
48      }
```

In Listing 7.12 befindet sich ein wesentlicher Inhalt des Projektes, wie z.B. der Start der Suche mit den verschiedenen Parameter.
In Zeile 2-5 wird dann eine Fallunterscheidung für den Start- und Endpunkt gemacht. Darin wird jeweils unterschieden, ob zufällig ein Punkt auf der Karte erstellt oder ein bestimmter Punkt aus der Datenbank ausgewählt wird. Um eine kopierte Karte zu erreichen, wird zwischen allen kopierten Flughäfen und den Ausgangsflughäfen eine Kante gegeben, dessen Kantengewicht der Luftliniendistanz entspricht.

Listing 7.13: Main.java Die Main-Klasse (4)

```
1  private static double berechneMultiplierPosition(int x, String wert,
2              Stadt std) {
3          int lat1 = 0;
4          int lat2 = 10;
5          int lat3 = -20;
6          int lat0 = 0;
7          int lon0 = 0;
8          int lon1 = 15;
9          int lon2 = 10;
```

```
10          int lon3 = 10;
11
12          int mult = (x / 4) + 1;
13
14          int bereich = x \mod 4;
15
16          switch (bereich) {
17
18          case 0:
19              if (wert.equals("lon") && (x > 0))
20                  return std.getLaengenGrad() + mult * lon1 * -1;
21              else
22                  return std.getBreitenGrad();
23          case 1:
24              if (wert.equals("lon") && (x > 0))
25                  return std.getLaengenGrad() + mult * lon1;
26              else
27                  return std.getBreitenGrad();
28          case 2:
29              if (wert.equals("lon") && (x > 0))
30                  return std.getLaengenGrad() + lon2;
31              else
32                  return std.getBreitenGrad() + mult * lat2;
33          case 3:
34              if (wert.equals("lon") && (x > 0))
35                  return std.getLaengenGrad() + lon3;
36              else
37                  return std.getBreitenGrad() + mult * lat3;
38          }
39          return 0;
40      }
41      public static GraphPath<Stadt, DefaultWeightedEdge> suche(
42              MyGraph<Stadt, DefaultWeightedEdge> graph, Stadt start,
                    Stadt ziel,
43              double vBahn, double vAuto, double vFlugzeug, double
                    warteBahn,
44              double warteFlug, HeuristicTyp heuristic) {
45          bahnV = vBahn;
46          flugV = vFlugzeug;
47          autoV = vAuto;
48
49          if (!graph.containsVertex(start))
50              graph.addVertex(start);
51          if (!graph.containsVertex(ziel))
52              graph.addVertex(ziel);
53          if (randomStart && !graph.containsEdge(start, ziel))
```

```
54              addEdgesForCar(graph, start, ziel);
55          if (randomZiel && !graph.containsEdge(ziel, start))
56              addEdgesForCar(graph, ziel, start);
57          AStar astar = new AStar();
58          if (!randomStart && !randomZiel)
59              astar.setFall(false);
60          else
61              astar.setFall(true);
62          astar.setGeschwindigkeiten(vAuto, vBahn, vFlugzeug);
63          astar.setWarteZeiten(warteBahn, warteFlug);
64          expandnum = astar.expandnodenumbers;
65          GraphPath<Stadt, DefaultWeightedEdge> path = astar.
                findPathBetween(
66                  graph, start, ziel, heuristic);
67          return path;
68      }
```

Die „berechneMultiplierPosition“ -Funktion (Listing 7.13, Z.21) beschreibt und berechnet die notwendigen Längen- und Breitengradwerte für die kopierten bzw. gespiegelten Karten nach dem bereits gezeigten Muster.
Die Suche nach dem schnellsten Weg wird mithilfe der *suche()*-Funktion in Zeile 41 mit den angegebenen Parameter erstellt.
In Zeile 49-52 wird ein Start- und Zielpunkt in den Graphen eingefügt und eine Kante jeweils zum nächsten Bahnhof und Flughafen erzeugt, falls der Start- und Zielpunkt kein Knoten im Graphen ist (Fall: Auto/Bahn/Flugzeug). Sollte außerdem der Start- oder Zielpunkt ein Zufälliger sein, dann wird eine Kante zu jeder anderen Station virtuell gezogen.

Listing 7.14: Main.java Die Main-Klasse (5)

```
1  public static double calculateTime(
2              MyGraph graph,
3              GraphPath<Stadt, DefaultWeightedEdge> path) {
4
5          double time = 0;
6
7          for (DefaultWeightedEdge edge : path.getEdgeList()) {
8              time += getCost(graph, edge);
9          }
10         return time;
11     }
12     private static Stadt generateRandomPoint() {
13      Random r = new Random();
14      double breiteMax = 55;
15      double breiteMin = 47;
16      double startrandomBreite = breiteMin + (breiteMax - breiteMin)
17      * r.nextDouble();
18      double laengeMax = 15;
```

```
19      double laengeMin = 6;
20      double startrandomlaenge = laengeMin + (laengeMax - laengeMin)
21      * r.nextDouble();
22      System.out.println("Zufallspunkt:"+ startrandomBreite + " " +
23      startrandomlaenge);
24      return new Stadt(Integer.toString(max++), "rand" + max,
25      startrandomlaenge, startrandomBreite, "Auto");
26      }
27     private static void addEdgesForCar(MyGraph<Stadt,
           DefaultWeightedEdge> graph, Stadt start, Stadt ziel) {
28         for (Stadt v : stadte) {
29             double gewicht = DistanceCalculator.getLuftLinie(start, v)
                   / autoV;
30             if (!graph.containsEdge(start, v) && !v.equals(ziel)){
31             graph.addEdge(start, v);
32             DefaultWeightedEdge e = graph.getEdge(start, v);
33             graph.setEdgeWeight(e, gewicht);
34
35             graph.addEdge(v, start);
36             DefaultWeightedEdge e2 = graph.getEdge(v, start);
37             graph.setEdgeWeight(e2, gewicht);
38             }
39         }
40     }
41     private static void workOnEdgesForONECar(MyGraph<Stadt,
           DefaultWeightedEdge> graph, Stadt start, Stadt ziel, String
           option) {
42
43         if(option.equals("add")){
44             double gewicht = DistanceCalculator.getLuftLinie(start,
                   ziel) / autoV;
45             if (!graph.containsEdge(start, ziel)){
46             graph.addEdge(start, ziel);
47             DefaultWeightedEdge e = graph.getEdge(start, ziel);
48             graph.setEdgeWeight(e, gewicht);
49
50             graph.addEdge(ziel, start);
51             DefaultWeightedEdge e2 = graph.getEdge(ziel, start);
52             graph.setEdgeWeight(e2, gewicht);
53             }
54         }
55         else {
56             if(option.equals("remove")){
57                 if(graph.containsEdge(start,ziel)){
58                     graph.removeEdge(start, ziel);
59                     graph.removeEdge(ziel,start);
```

```
60                  }
61              }
62          }
63      }
64      private static void suchAusgabe(MyGraph<Stadt,DefaultWeightedEdge
            > g, Stadt start, Stadt ziel, HeuristicTyp heuristic ){
65
66          GraphPath<Stadt,DefaultWeightedEdge> path = null;
67          double calctime = 0;
68          double midtime = 0.0;
69
70          long time = System.currentTimeMillis();
71          for (int i = 0; i < 5 ; i++){
72          path= suche(g, start, ziel, bahnV, autoV, flugV, warteBahn,
                warteFlug, heuristic);
73          time = System.currentTimeMillis() - time;
74          calctime = calculateTime(g, path);
75          midtime = midtime + time;
76          time = System.currentTimeMillis();
77      }
```

In Listing 7.14 werden unter Anderem die Berechnungszeit, ein Zufallspunkt, die Zeit des Weges und der Pfad erstellt.
Damit ein zufälliger Punkt generiert wird, wird die *generateRandomPoint*-Funktion implementiert. Der Definitionsbereich vom Längen- und Breitengrad des zufälligen Punktes richtet sich zunächst nach dem von Deutschland, da wir zunächst ein Verkehrsnetz von Deutschland betrachten. Der Längengrad ist zwischen 6. und 15. Längengrad Ost und 47. und 55. Breitengrad Nord. Je nachdem, wie groß der *multiplier* ist, wird dieser noch vergrößert.
Bei diesem Zufallsfall mit zufälligen Start- und Endpunkt beginnt man den Weg mit dem Auto. Dieses Auto soll vom Startpunkt jeden anderen Punkt erreichen können, genauso wie der Endpunkt jeden anderen Punkt. Genau das wird mit der *addEdgesForCar()*-Methode erreicht.
Die *workOnEdgesForONECar*-Funktion (Z.40) generiert einen direkten Weg für ein Auto vom Start zum Ziel mit dem Gewicht der Luftlinienentfernung durch die Autogeschwindigkeit.
Zu guter Letzt dient die *suchAusgabe*-Methode am Ende unter anderem für die durchschnittliche Berechnungszeit, die Zeit und den Weg der Suche.

7.4.6 AStarShortestPath

Diese Klasse implementiert den A^*-Algorithmus mit dem Ziel, den schnellsten Weg zu finden. Dabei werden verschiedene wichtige Funktionen erstellt, die zum schnellsten Weg zwischen zwei Punkten A und B beitragen. Fast der komplette Inhalt der Klasse wurde in einem *Open-Source*[9] gefunden und an unsere Aufgabenstellung an-

[9] https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/AStarShortestPath.java

gepasst.

Listing 7.15: AStarShortestPath.java Die AStarShortestPath-Klasse (1)

```
1  package org.jgrapht.alg;
2
3  public class AStarShortestPath<V, E> {
4      private final Graph<V, E> graph;
5
6      protected FibonacciHeap<V> openList;
7      protected Map<V, FibonacciHeapNode<V>> vertexToHeapNodeMap;
8
9      protected Set<V> closedList;
10
11     protected Map<V, Double> gScoreMap;
12
13     protected Map<V, E> cameFrom;
14
15     protected AStarAdmissibleHeuristic<V> admissibleHeuristic;
16
17     protected int numberOfExpandedNodes;
18
19     public AStarShortestPath(Graph<V, E> graph) {
20         if (graph == null) {
21             throw new IllegalArgumentException("Graph cannot be null!"
                   );
22         }
23
24         this.graph = graph;
25     }
26
27     private void initialize(AStarAdmissibleHeuristic<V>
           admissibleHeuristic) {
28         this.admissibleHeuristic = admissibleHeuristic;
29         openList = new FibonacciHeap<V>();
30         vertexToHeapNodeMap = new HashMap<V, FibonacciHeapNode<V>>();
31         closedList = new HashSet<V>();
32         gScoreMap = new HashMap<V, Double>();
33         cameFrom = new HashMap<V, E>();
34         numberOfExpandedNodes = 0;
35     }
```

In Listing 7.15 werden zu Beginn die Datenstrukturen initialisiert. Die *Closed-List* ist zu Beginn noch leer (Z.31).

Listing 7.16: AStarShortestPath.java Die AStarShortestPath-Klasse (2)

```
1  public GraphPath<V, E> getShortestPath(V sourceVertex, V targetVertex
       ,
2              AStarAdmissibleHeuristic<V> admissibleHeuristic) {
3          if (!graph.containsVertex(sourceVertex)
4                  || !graph.containsVertex(targetVertex)) {
5              throw new IllegalArgumentException(
6                      "Source or target vertex not contained in the graph
                           !");
7          }
8          this.initialize(admissibleHeuristic);
9          gScoreMap.put(sourceVertex, 0.0);
10         FibonacciHeapNode<V> heapNode = new FibonacciHeapNode<V>(
               sourceVertex);
11         openList.insert(heapNode, 0.0);
12         vertexToHeapNodeMap.put(sourceVertex, heapNode);
13
14         do {
15             FibonacciHeapNode<V> currentNode = openList.removeMin();
16
17             if (currentNode.getData().equals(targetVertex)) {
18
19                 return this.buildGraphPath(sourceVertex, targetVertex,
20                         currentNode.getKey());
21             }
22             expandNode(currentNode, targetVertex);
23             closedList.add(currentNode.getData());
24             HashSet<Stadt> closer = Main.getClosed();
25             closer.add((Stadt) currentNode.getData());
26         } while (!openList.isEmpty());
27
28         return null;
29     }
```

Mithilfe der *getShortestPath*-Methode in Listing 7.16 wird der kürzeste Pfad vom Start- zum Zielknoten berechnet und zurückgegeben. Dabei wird jedes Mal, wo die Methode aufgerufen wird, der Pfad wieder neu berechnet. Als nächstes wird die *while*-Schleife (Z.14-26) solange durchgegangen, bis die optimale Lösung entdeckt wurde oder festliegt, dass es keine Lösung gibt.

Listing 7.17: AStarShortestPath.java Die AStarShortestPath-Klasse (3)

```
1  private void expandNode(FibonacciHeapNode<V> currentNode, V endVertex
       ) {
2
3   ((Stadt)currentNode.getData()).getName() + " T>P: " + ((Stadt)
        currentNode.getData()).getTyp());
4          numberOfExpandedNodes++;
```

```
5
6           Set<E> outgoingEdges = null;
7           if (graph instanceof UndirectedGraph) {
8               outgoingEdges = graph.edgesOf(currentNode.getData());
9           } else if (graph instanceof DirectedGraph) {
10              outgoingEdges = ((DirectedGraph) graph).outgoingEdgesOf(
                    currentNode
11                      .getData());
12          }
13          for (E edge : outgoingEdges) {
14              V successor = Graphs.getOppositeVertex(graph, edge,
15                      currentNode.getData());
16              if ((successor.equals(currentNode.getData()))) {
17                  continue;
18              }
19              double gScore_current = gScoreMap.get(currentNode.getData
                    ());
20              double tentativeGScore = gScore_current + graph.
                    getEdgeWeight(edge);
21
22              if(closedList.contains(successor)){
23                  if(gScoreMap.get(successor)>tentativeGScore){
24                      closedList.remove(successor);
25                      vertexToHeapNodeMap.remove(successor);
26                      FibonacciHeapNode<V> heapNode = new
                            FibonacciHeapNode<V>(
27                              successor);
28                      openList.insert(heapNode, tentativeGScore);
29                      vertexToHeapNodeMap.put(successor, heapNode);
30                  }
31              }
32              if (!vertexToHeapNodeMap.containsKey(successor)
33                      || (tentativeGScore < gScoreMap.get(successor))) {
34                  cameFrom.put(successor, edge);
35                  gScoreMap.put(successor, tentativeGScore);
36
37                  double fScore = tentativeGScore
38                          + admissibleHeuristic.getCostEstimate(successor
                                ,
39                                  endVertex);
40                  Stadt ausgabe = (Stadt) successor;
41
42                  if (!vertexToHeapNodeMap.containsKey(successor)) {
43                      FibonacciHeapNode<V> heapNode = new
                            FibonacciHeapNode<V>(
44                              successor);
```

```
45                  openList.insert(heapNode, fScore);
46                  vertexToHeapNodeMap.put(successor, heapNode);
47              } else {
48
49                  if(fScore > vertexToHeapNodeMap.get(successor).
                        getKey()){
50                  openList.decreaseKey(vertexToHeapNodeMap.get(
                        successor), vertexToHeapNodeMap.get(successor).
                        getKey());}
51                  else{
52                      vertexToHeapNodeMap.remove(successor);
53                      FibonacciHeapNode<V> heapNode = new
                            FibonacciHeapNode<V>(
54                              successor);
55                      openList.insert(heapNode, fScore);
56                      vertexToHeapNodeMap.put(successor, heapNode);
57                      openList.decreaseKey(vertexToHeapNodeMap.get(
                            successor), fScore);
58                  }
59              }
60          }
61      }
62  }
```

Die *expandNode*-Methode aus Zeile 1 in Listing 7.17 kontrolliert alle Nachfolgerknoten und fügt sie der *Open-List* hinzu, falls entweder der Nachfolgeknoten erstmals oder ein besserer Weg zu diesem Knoten entdeckt wird.

Listing 7.18: AStarShortestPath.java Die AStarShortestPath-Klasse (4)

```
1  private GraphPath<V, E> buildGraphPath(V startVertex, V targetVertex,
2              double pathLength) {
3          List<E> edgeList = this.buildPath(targetVertex);
4
5          return new GraphPathImpl<V, E>(graph, startVertex,
               targetVertex,
6                  edgeList, pathLength);
7      }
8      private List<E> buildPath(V currentNode) {
9          if (cameFrom.containsKey(currentNode)) {
10             List<E> path = buildPath(Graphs.getOppositeVertex(graph,
11                     cameFrom.get(currentNode), currentNode));
12             path.add(cameFrom.get(currentNode));
13             return path;
14         } else {
15             return new ArrayList<E>();
16         }
17     }
```

```
18      public int getNumberOfExpandedNodes() {
19          return numberOfExpandedNodes;
20      }
21  }
```

In Listing 7.18 verfolgt die rekursive *buildPath*-Methode den Weg vom Zielknoten zum Startknoten über die Kanten zurück. Die *getNumberOfExpandedNodes*()-Methode gibt die Anzahl expandierender Knoten in der A^*-Suche aus seinem letzten Aufruf wieder. Ein Knoten wird dann expandiert, falls es aus der *Open-List* entfernt wird.

7.4.7 DistanceCalculator

Die „DistanceCalculator"-Klasse berechnet die Distanz zwischen zwei Punkten unter Angabe der Längen- und Breitengrade der beiden Punkte. Weiterhin werden in dieser Klasse die nächste Haltestelle und der nächste Flughafen eines Knotens bestimmt. Zur Entfernungsbestimmung in Listing 7.19 für die Städte wird einfach die Erdoberfläche als Ebene angesehen und zum Satz des Pythagoras gegriffen (Z.11-16). Die Konstante 111.3 ist dabei der Abstand zwischen zwei Breitenkreisen gemessen in km und 71.5 der durchschnittliche Abstand zwischen zwei Längenkreisen. Eine weitere Möglichkeit der Entfernungsberechnung, die genauer wäre, ist die *Haversine formula* (Flugdistanz nach der Großkreisentfernung). Wir werden die *Haversine formula* verwenden. In Java implementiert, sieht eine entsprechende Funktion, die die Distanz zwischen zwei GPS-Koordinaten - gegeben durch Längen- und Breitengrad - in Kilometern berechnet, wie ab Zeile 17 aus.

Listing 7.19: DistanceCalculator.java Die DistanceCalculator-Klasse (1)

```
1  package util;
2
3  public class DistanceCalculator {
4
5
6      public static double getLuftLinie(Stadt stadt1, Stadt stadt2) {
7          // if(stadt1 != null && stadt2 != null){
8          return distance(stadt1.getBreitenGrad(), stadt1.getLaengenGrad
               (),
9                  stadt2.getBreitenGrad(), stadt2.getLaengenGrad(), "K")
                       ;
10     }
11     private static double distance2(double lat1, double lon1, double
           lat2,
12             double lon2) {
13         double d = Math.sqrt(Math.pow((lon2 - lon1), 2)
14                 + Math.pow((lat2 - lat1), 2));
15         return d;
16     }
17     private static double distance(double lat1, double lon1, double
           lat2,
```

```
18              double lon2, String unit) {
19          double theta = lon1 - lon2;
20          double dist = Math.sin(deg2rad(lat1)) * Math.sin(deg2rad(lat2)
                )
21                  + Math.cos(deg2rad(lat1)) * Math.cos(deg2rad(lat2))
22                  * Math.cos(deg2rad(theta));
23          dist = Math.acos(dist);
24          dist = rad2deg(dist);
25          dist = dist * 60 * 1.1515;
26          if (unit == "K") {
27              dist = dist * 1.609344;
28          } else if (unit == "N") {
29              dist = dist * 0.8684;
30          }
31
32          return (dist);
33      }
34      private static double deg2rad(double deg) {
35          return (deg * Math.PI / 180.0);
36      }
37      private static double rad2deg(double rad) {
38          return (rad * 180 / Math.PI);
39      }
```

In Listing 7.20 berechnet und bestimmt die *getNextFlughafen()*-Methode den nächsten Flughafen von einer Stadt, indem er alle anderen Städte betrachtet und denjenigen mit der minimalsten Luftliniendistanz auswählt. Die *getNextHaltestelle()*-Methode funktioniert analog nur mit dem Unterschied, dass die nächste Haltestelle von einer Stadt bestimmt und berechnet wird.

Listing 7.20: DistanceCalculator.java Die DistanceCalculator-Klasse (2)

```
1       public static Stadt getNextFlughafen(Stadt stadt1, List<Stadt>
            alleStadte) {
2           if (stadt1.getTyp().equals("Flughafen"))
3               return stadt1;
4           DistanceCalculator linie = new DistanceCalculator();
5           double min = Double.MAX_VALUE;
6           Stadt minStadt = null;
7           for (Stadt stadt : alleStadte) {
8
9               if (stadt.getTyp().equals("Flughafen")
10                      && !stadt.getName().equals(stadt1.getName())) { //
11                  double output = linie.getLuftLinie(stadt1, stadt);
12                  if (output < min) {
13                      min = output;
14                      minStadt = stadt;
15                  }
```

```
16              }
17          }
18          return minStadt;
19      }
20      public static Stadt getNextHaltestelle(Stadt stadt1, List<Stadt>
            alleStadte) {
21          DistanceCalculator linie = new DistanceCalculator();
22          double min = Double.MAX_VALUE;
23          Stadt minStadt = null;
24          for (Stadt stadt : alleStadte) {
25              if (!stadt.getTyp().equals("Auto")
26                      && !Main.getClosed().contains(stadt)
27                      && !stadt1.getName().equals(stadt.getName())) { //
28                  double output = linie.getLuftLinie(stadt1, stadt);
29                  if (output < min) {
30                      min = output;
31                      minStadt = stadt;
32                  }
33              }
34          }
35
36          return minStadt;
37      }
38  }
```

7.4.8 SQLHelper

Die „SQLHelper“-Klasse dient als Schnittstelle mit der Datenbank. Mithilfe der *getAllCitys()*-Methode in Listing 7.21 werden alle Standorte aus der Datenbank hochgeladen, welches alle Bahnhöfe und Flughäfen beinhaltet. Sobald dann eine Verknüpfung mit der MySQL-Datenbank erstellt worden ist, werden die „Staedte“ durch einen einfachen *select* Befehl (Z.17-18) abgefragt und die ganzen „Staedte“ mit ihren Attributen und Datensätzen und als Liste von „Staedte“ mit den Namen „cities“ zurückgegeben.

Listing 7.21: SQLHelper.java Die SQLHelper-Klasse (1)

```
1   package util;
2
3   public class SQLHelper {
4       private static String myDriver = "org.gjt.mm.mysql.Driver";
5       private static String myUrl = "jdbc:mysql://127.0.0.1/karte";
6       private static HashSet<String> idsBahn;
7
8       public List<Stadt> getAllCitys() {
9           List<Stadt> cities = new ArrayList<Stadt>();
10          idsBahn = new HashSet<String>();
```

```
11          try {
12              Connection conn = DriverManager.getConnection(myUrl, "root
                    ", "");
13              Statement statement = null;
14
15              statement = conn.createStatement();
16
17              ResultSet resultSet = statement.executeQuery("Select *
                    from standort;");
18
19              while (resultSet.next()) {
20                  cities.add(new Stadt(resultSet.getString("id"),
                        resultSet
21                            .getString("name"), resultSet.getDouble("laenge
                                "),
22                            resultSet.getDouble("breite"), resultSet
23                                    .getString("Typ")));
24
25                  if (resultSet.getInt("riesigbahnhof") == 1)
26
27                      idsBahn.add(resultSet.getString("name"));
28              }
29              conn.close();
30              return cities;
31          } catch (Exception e) {
32              System.err.println("Got an exception! ");
33              System.err.println(e.getMessage());
34          }
35          return null;
36      }
```

In Zeile 1 in Listing 7.22 wird mit der *getCity()*-Funktion jeweils die aktuelle Stadt aus der Datenbank zurückgegeben. Diese wird auch mit einer bekanntlich einfachen Anfrage abgefragt (Z.10).

Listing 7.22: SQLHelper.java Die SQLHelper-Klasse (2)

```
1   public Stadt getCity(String name) {
2           try {
3
4               Connection conn = DriverManager.getConnection(myUrl, "root
                    ", "");
5               Statement statement = null;
6
7               statement = conn.createStatement();
8
9               ResultSet resultSet = statement
```

```
                    .executeQuery("Select * from standort where name =
                        \"" + name
                            + "\";");

            while (resultSet.next()) {
                return new Stadt(resultSet.getString("id"),
                        resultSet.getString("name"),
                        resultSet.getDouble("laenge"),
                        resultSet.getDouble("breite"),
                        resultSet.getString("Typ")// ,

                );
            }

            conn.close();
        } catch (Exception e) {
            System.err.println("Got an exception! ");
            System.err.println(e.getMessage());
        }
        return null;
    }
```

Die bisherigen Abfragen bezogen sich ausschließlich auf die Knoten. Nun benötigen wir die Abfragen nach den Kanten, also den Verbindungen. Dies geschieht mit der *getAllVerbindung()*-Funktion in Listing 7.23. Wir laden uns mit dieser Methode alle Verbindungen aus der Datenbank hoch und speichern diese als Liste. Die Verbindungen erhalten noch als Parameter die Geschwindigkeiten der Verkehrsmittel und die Wartezeiten der Bahn und des Flugzeugs.

Listing 7.23: SQLHelper.java Die SQLHelper-Klasse (3)

```
public List<Verbindung> getAllVerbindung(double av, double bv, double
    fv, double wBahn, double wFlug) {

        List<Verbindung> verbindungen = new ArrayList<Verbindung>();
        try {

            Connection conn = DriverManager.getConnection(myUrl, "root
                ", "");
            Statement statement = null;

            statement = conn.createStatement();

            ResultSet resultSet = statement
                    .executeQuery("Select * from zusammenhang;");

            while (resultSet.next()) {

```

```
16                verbindungen.add(new Verbindung(getCity(resultSet.
                      getString(
17                          "anfang").trim()), getCity(resultSet.getString(
                              "ende")
18                          .trim()), resultSet.getDouble("kosten"), av, bv
                              , fv, wBahn, wFlug));
19            }
20            conn.close();
21            return verbindungen;
22        } catch (Exception e) {
23            System.err.println("Got an exception! ");
24            System.err.println(e.getMessage());
25        }
26        return null;
27    }
28
29    public static HashSet<String> getBahnids() {
30        return idsBahn;
31    }
32 }
```

7.4.9 FileHelper

Listing 7.24: FileHelper.java Die FileHelper-Klasse

```
1  package util;
2
3  public class FileHelper {
4      private static String myDriver = "org.gjt.mm.mysql.Driver";
5      private static String myUrl = "jdbc:mysql://127.0.0.1/karte";
6
7      public List<Stadt> getAllCitys() {
8          List<String> lines = null;
9          try {
10             lines = FileUtils.readLines(new File("data/stadte.in"));
11         } catch (IOException e1) {
12             // TODO Auto-generated catch block
13             e1.printStackTrace();
14         }
15     return cities;
16     }
17     public List<Verbindung> getAllVerbindung(double av, double bv,
           double fv, double wBahn, double wFlug) {
18
19         List<Stadt>stadte = getAllCitys();
20
```

```
21          List<String> lines = null;
22          try {
23              lines = FileUtils.readLines(new File("data/verbindungen.in
                    "));
24          } catch (IOException e1) {
25
26              e1.printStackTrace();
27          }
28          return verbindungen;
29      }
```

Die „FileHelper“-Klasse dient ausschließlich zum Abspeichern der Daten aus der Datenbank. Diese Daten befinden sich dann im Ordner *data*. Diese Klasse wurde erstellt, damit man die MySQL-Datenbank *phpmyadmin* nicht aufzusetzen braucht.

7.5 Datenbankbeschreibung

In diesem Abschnitt werden die erstellten Daten der Datenbank beschrieben. In der Datenbankübersicht wurde bereits erwähnt, wo sich die Daten der Datenbank befinden. Nun geht es darum, zu erklären, wie die Daten in der Datenbank eingetragen sind. Wie bereits erwähnt wurde, sind die Daten unter der Datenbank „karte“ in zwei Tabellen unterteilt, und zwar einmal in die Tabelle „standort“ und einmal in die Tabelle „zusammenhang“. Die Tabelle „standort“ beinhaltet alle Bahnhöfe und Flughäfen (siehe 7.2). Sie besteht aus den Attributen „id“, „name“, „laenge“, „breite“, „Typ“ und „riesigbahnhof“. Die Attribute „breite“ und „laenge“ beziehen sich auf den Längen- und Breitengrad, „Typ“ aus Haltestelle oder Flughafen und „riesigbahnhof“, ob dieser Bahnhof ein Großbahnhof ist; dann ist er mit einer 1 gekennzeichnet, ansonsten mit einer 0. Die Unterscheidung wurde gemacht, um bestimmte Wartezeiten (Umsteigezeiten) den jeweiligen Bahnhöfen zuzuordnen.

id	name	laenge	breite	Typ	riesigbahnhof
1	Frankfurt (M) Hbf	8.662135	50.105001	Haltestelle	1
2	Berlin Hbf	13.369402	52.5250839	Haltestelle	1
3	Wolfsburg	10.7865461	52.4226503	Haltestelle	0
4	Hannover	9.7320104	52.3758916	Haltestelle	1
5	Minden	8.8949206	52.2964919	Haltestelle	0

Abbildung 7.2: Ausschnitt aus der Datenbank für Knoten

Die Bahnhöfe und Flughäfen wurden wie oben erwähnt erstellt (siehe 7.1). Die Längen- und Breitengrade der Städte wurden mithilfe der Google Maps Koordinaten [10] bestimmt. Welcher Bahnhof als „riesigbahnhof“ gekennzeichnet wurde, habe ich durch die Information der Reisenden täglich an diesen Bahnhöfen Deutschlands [11] definiert.

[10] http://www.gpskoordinaten.de/
[11] http://www.goeuro.de/bahnhoefe/

Kommen wir nun zu den Kanten. Diese sind in der Tabelle „zusammenhang“ gespeichert (siehe 7.3). Diese Tabelle besitzt die Attribute „id“, „anfang“, „ende“ und „kosten“. „anfang“ bezeichnet immer eine Stadt A und „ende“ die dazugehörige Kante zu einer anderen Stadt B. Dabei gibt es immer eine Kante zurück zu einer Stadt, also auch eine Kante von B nach A.

id	anfang	ende	kosten
1	Münster	Dortmund	56
2	Dortmund	Münster	56
3	Bremen	Osnabrück	123
4	Osnabrück	Bremen	123
5	Osnabrück	Münster	48

Abbildung 7.3: Ausschnitt aus der Datenbank für Kanten

Außerdem existiert bei den Flughäfen von jedem Flughafen eine Kante zu jedem anderen Flughafen und zurück. Die Kanten der Bahnen sind entsprechend aus dem beschriebenen Verkehrsnetz des ICE-Netzes. Weiterhin gibt es auch eine Kante zwischen jedem Flughafen und jedem nächstmöglichen Bahnhof (mit der kleinsten Luftliniendistanz) .
Die „kosten“ zwischen Bahnhöfen und Bahnhof/Flughafen sind aus dem Kursbuch [12] und der Trassenpreisauskunftssoftware der Deutschen Bahn [13]. Die Kosten zwischen Flughäfen entsprechen den Luftlinienentfernung [14]. Das Problem mit einem Flughafen, der sowohl Flughafen als auch Bahnhof ist, habe ich dadurch gelöst, das dazwischen eine Kante auch mit eine sehr geringen Kostenzahl existiert. Alle Zahlen sind etwas gerundet.

[12] http://kursbuch.bahn.de/hafas/kbview.exe/dn?rt=1&mainframe=search

[13] http://fahrweg.dbnetze.com/fahrweg-de/produkte/trassen/trassenpreise/trassenpreisauskunft_tpis.html

[14] http://www.gctoolbox.de/ger/tools/Abstand_und_Winkel_zwischen_zwei_Koordinaten_in_Luftlinie/distance.htm

KAPITEL

ACHT

Ergebnisse

Wir haben für das gesamte Experiment die Nullheuristik, *Advanced* Heuristik und Luftlinienheuristik verwendet, jeweils mit Parameter wie Umsteigezeiten oder Geschwindigkeiten bei einigen Fällen auch variiert. Für die verschiedenen Spaltennamen verwenden wir die in Kapitel 5 eingeführte Notation, wobei S stets den Startknoten definiert. Die Instanzen wurden stichprobenartig ausgewählt. Außer den Tests mit den verschiedenen Parameter wurde ohne Wartezeiten gerechnet.

8.1 Experimentelle Ergebnisse für Bahn/Flugzeug

Unsere Schätzung der erwarteten Berechnungszeit (in ms) für die jeweilige Instanz ist die durchschnittliche Berechnungszeit über eine hohe Anzahl an Durchläufen (5) des Weges. Für die Ergebnisse haben wir folgende Geschwindigkeiten festgelegt: $v_B = 100$ und $v_F = 1000$, wobei im Abschnitt der unterschiedlichen Parameter wir die Geschwindigkeit des Flugzeugs auch auf $v_F = 300$ gesetzt haben um zu schauen welchen Einfluss dieser mit sich bringt.

Ergebnisse für etwa 150 Knoten

Für das erste Experiment haben wir die Anzahl von Knoten auf etwa 150 gesetzt. Hierfür werden in den folgenden Tabellen die Ergebnisse der erwarteten Berechnungszeit in ms der Anzahl expandierenden Knoten für die verschiedenen Heuristiken dargestellt, die für die jeweilige Instanz resultieren. Wir erinnern uns daran, dass S den Startknoten, Z den Zielknoten, h_1 die Luftlinienheuristik, h_2 die *Advanced* Heuristik und h_3 die Nullheuristik bezeichnet.

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Ulm, Fulda	3,0	20,0	13,8	5,0	37,0	8,6
Waren, Naumburg	3,2	13,8	7,0	3,6	23,6	16,2
Darmstadt, Essen	3,0	20,2	9,6	3,2	20,0	7,6
Frankf. (M) Hbf, Berlin Hbf	0,0	20,0	6,0	3,0	15,6	6,0
München Hbf, Jena Paradies	7,2	40,4	9,8	2,4	13,8	9,0
Dresden F, Soest	6,2	37,0	10,6	3,2	20,2	7,6
Hamburg F, Halle	5,0	35,8	8,8	1,0	7,4	6,0
Frankfurt/M F, Rostock	7,6	39,0	12,4	0,8	14,0	6,6
Düsseldorf F, Mainz	3,0	17,0	4,6	1,2	15,8	6,0
Münster/Osnabrück F, Riesa	3,0	26,4	10,8	3,2	11,6	4,4
Hamburg F, Erfurt F	3,2	7,0	3,2	0,0	8,2	4,4
Dresden F, Frankfurt/M F	0,6	10,2	1,8	0,6	10,0	5,4
Stuttgart F, Bremen F	0,0	5,8	7,6	0,0	6,2	4,6
Berlin Tegel F, Köln/Bonn F	3,0	8,4	4,4	3,0	9,0	4,4
Saarbrücken F, Nürnberg F	0,0	7,0	6,4	5,0	157,6	5,4

Tabelle 8.1: Berechnungszeit: Bahn, Flugzeug für etwa 150 Knoten

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Ulm, Fulda	87	10	105	90	33	110
Waren, Naumburg	33	14	61	100	27	127
Darmstadt, Essen	25	10	49	32	14	49
Frankf. (M) Hbf, Berlin Hbf	15	10	48	14	8	33
München Hbf, Jena Paradies	74	26	94	25	9	62
Dresden F, Soest	110	56	127	23	13	55
Hamburg F, Halle	90	54	104	3	3	24
Frankfurt/M F, Rostock	147	60	152	8	6	27
Düsseldorf F, Mainz	16	6	34	4	3	11
Münster/Osnabrück F, Riesa	57	21	78	6	4	24
Hamburg F, Erfurt F	1	1	12	1	1	22
Dresden F, Frankfurt/M F	1	1	17	1	1	28
Stuttgart F, Bremen F	1	1	28	1	1	32
Berlin Tegel F, Köln/Bonn F	1	1	24	2	1	34
Saarbrücken F, Nürnberg F	1	1	10	60	1232	11

Tabelle 8.2: Anzahl expandierender Knoten: Bahn, Flugzeug für etwa 150 Knoten

Ablauf eines Ergebnis

Das folgende Beispiel gibt die *Open*- und *Closed*-Menge sowie den aktuellen Knoten für den Weg von München Hbf nach Nürnberg, nach jedem ersten, zweiten, vorletzten und letzten Iterationsschritt an:
a) Luftlinienheuristik:

	Exp. Knoten	Open	Closed
Am Anfang		{(München Hbf, 0.0)}	∅
1. Iteration	München Hbf	{(M-Ost, 0.1), (Augsburg, 0.63), (M-Pasing, 0.07), (Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Nürnberg, 1.71)}	{(München-Hbf, 0.0)}
2. Iteration	M-Pasing	{(M-Ost, 0.1), (Augsburg, 0.63), (Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Nürnberg, 1.71), (Augsburg, 0.54)}	{(München Hbf, 0.0) , (M-Pasing, 0.07) }
6. Iteration	Tutzing	{(Salzburg, 1.53), (Ingolstadt, 0.81), (Augsburg, 0.54), (Rosenheim, 0.65), (Hamburg Flughafen, 0.60099), (Bremen Flughafen, 0.563), (Hannover Flughafen, 0.481), (Berlin Schönefeld Flughafen, 0.46600), (Berlin Tegel Flughafen, 0.48), (Münster/Osnabrück Flughafen, 0.511), (Düsseldorf Flughafen, 0.486), (Leipzig/Halle Flughafen, 0.34300), (Köln/Bonn Flughafen, 0.43600), (Erfurt Flughafen, 0.29799), (Dresden Flughafen, 0.34100), (Frankfurt/Main Flughafen, 0.29900), (Saarbrücken Flughafen, 0.355), (Stuttgart Flughafen, 0.19299), (Nürnberg, 0.07999), (Murnau, 0.35)}	{(München Hbf, 0.0) , (M-Pasing, 0.07) , (M-Ost, 0.1) , (München Flughafen, 0.4) , (Nürnberg Flughafen, 0.537) , (Tutzing, 0.4)}
7. Iteration	Nürnberg	{(Salzburg, 1.53), (Ingolstadt, 0.81), (Augsburg, 0.54), (Rosenheim, 0.65), (Hamburg Flughafen, 0.60099), (Bremen Flughafen, 0.563), (Hannover Flughafen, 0.481), (Berlin Schönefeld Flughafen, 0.46600), (Berlin Tegel Flughafen, 0.48), (Münster/Osnabrück Flughafen, 0.511), (Düsseldorf Flughafen, 0.486), (Leipzig/Halle Flughafen, 0.34300), (Köln/Bonn Flughafen, 0.43600), (Erfurt Flughafen, 0.29799), (Dresden Flughafen, 0.34100), (Frankfurt/Main Flughafen, 0.29900), (Saarbrücken Flughafen, 0.355), (Stuttgart Flughafen, 0.19299), (Murnau, 0.35)}	{(München Hbf, 0.0) , (M-Pasing, 0.07) , (M-Ost, 0.1) , (München Flughafen, 0.4) , (Nürnberg Flughafen, 0.537) , (Tutzing, 0.4), (Nürnberg, 0.07999)}

Tabelle 8.3: *Open*- und *Closed*-Menge für die Luftlinienheuristik

b) *Advanced* Heuristik:

	Exp. Knoten	Open	Closed
Am Anfang		{(München Hbf, 0.0)}	∅
1. Iteration	München Hbf	{(M-Ost, 0.1), (Augsburg, 0.63), (M-Pasing, 0.07), (Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Nürnberg, 1.71)}	{(München Hbf, 0.0)}
2. Iteration	M-Ost	{(Augsburg, 0.63), (M-Pasing, 0.07), (Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Nürnberg, 1.71), (Rosenheim, 0.65), (München Flughafen, 0.30000)}	{(München Hbf, 0.0) , (M-Ost, 0.1) }
5. Iteration	Nürnberg Flughafen	{(Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Rosenheim, 0.65), (Augsburg, 0.54), (Hamburg Flughafen, 0.60099), (Bremen Flughafen, 0.563), (Hannover Flughafen, 0.481), (Berlin Schönefeld Flughafen, 0.46600), (Berlin Tegel Flughafen, 0.48), (Münster/Osnabrück Flughafen, 0.511), (Düsseldorf Flughafen, 0.486), (Leipzig/Halle Flughafen, 0.34300), (Köln/Bonn Flughafen, 0.43600), (Erfurt Flughafen, 0.29799), (Dresden Flughafen, 0.34100), (Frankfurt/Main Flughafen, 0.299000), (Saarbrücken Flughafen, 0.355), (Stuttgart Flughafen, 0.19299), (Nürnberg, 0.07999)}	{(München Hbf, 0.0) , (M-Ost, 0.1) , (M-Pasing, 0.07) , (München Flughafen, 0.4) , (Nürnberg Flughafen, 0.537)}
6. Iteration	Nürnberg	{(Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Rosenheim, 0.65), (Augsburg, 0.54), (Hamburg Flughafen, 0.60099), (Bremen Flughafen, 0.563), (Hannover Flughafen, 0.481), (Berlin Schönefeld Flughafen, 0.46600), (Berlin Tegel Flughafen, 0.48), (Münster/Osnabrück Flughafen, 0.511), (Düsseldorf Flughafen, 0.486), (Leipzig/Halle Flughafen, 0.34300), (Köln/Bonn Flughafen, 0.43600), (Erfurt Flughafen, 0.29799), (Dresden Flughafen, 0.34100) (Frankfurt/Main Flughafen, 0.299000), (Saarbrücken Flughafen, 0.355), (Stuttgart Flughafen, 0.19299)}	{(München Hbf, 0.0) , (M-Ost, 0.1) , (M-Pasing, 0.07) , (München Flughafen, 0.4) , (Nürnberg Flughafen, 0.537), (Nürnberg, 0.07999)}

Tabelle 8.4: *Open*- und *Closed*-Menge für die *Advanced* Heuristik

c) Nullheuristik:

	Exp. Knoten	Open	Closed
Am Anfang		{(München Hbf, 0.0)}	∅
1. Iteration	München Hbf	{(M-Ost, 0.1), (Augsburg, 0.63), (M-Pasing, 0.07), (Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Nürnberg, 1.71)}	{(München Hbf, 0.0)}
2. Iteration	M-Pasing	{(M-Ost, 0.1), (Augsburg, 0.63), (Tutzing, 0.4), (Salzburg, 1.53), (Ingolstadt, 0.81), (Nürnberg, 1.71), (Augsburg, 0.54)}	{(München Hbf, 0.0) , (M-Pasing, 0.07) }
8. Iteration	Augsburg	{(Salzburg, 1.53), (Ingolstadt, 0.81), (Rosenheim, 0.65), (Murnau, 0.35), (Hamburg Flughafen, 0.60099), (Bremen Flughafen, 0.563), (Hannover Flughafen, 0.481), (Berlin Schönefeld Flughafen, 0.46600), (Berlin Tegel Flughafen, 0.48), (Münster/Osnabrück Flughafen, 0.511), (Düsseldorf Flughafen, 0.486), (Leipzig/Halle Flughafen, 0.34300), (Köln/Bonn Flughafen, 0.43600), (Erfurt Flughafen, 0.29799), (Dresden Flughafen, 0.34100), (Frankfurt/Main Flughafen, 0.29900), (Saarbrücken Flughafen, 0.355), (Nürnberg, 0.07999), (Stuttgart, 0.19999), (Ulm, 0.86000), (Würzburg, 2.16)}	(München Hbf, 0.0) , (M-Pasing, 0.07) , (M-Ost, 0.1) , (Tutzing, 0.4) , (München Flughafen, 0.4) , {(Nürnberg Flughafen, 0.537) , (Stuttgart Flughafen, 0.593) , (Augsburg, 0.61000)}
9. Iteration	Nürnberg	{(Salzburg, 1.53), (Ingolstadt, 0.81), (Rosenheim, 0.65), (Murnau, 0.35), (Hamburg Flughafen, 0.60099), (Bremen Flughafen, 0.563), (Hannover Flughafen, 0.481), (Berlin Schönefeld Flughafen, 0.46600), (Berlin Tegel Flughafen, 0.48), (Münster/Osnabrück Flughafen, 0.511), (Düsseldorf Flughafen, 0.486), (Leipzig/Halle Flughafen, 0.34300), (Köln/Bonn Flughafen, 0.43600), (Erfurt Flughafen, 0.29799), (Dresden Flughafen, 0.34100), (Frankfurt/Main Flughafen, 0.29900), (Saarbrücken Flughafen, 0.355), (Stuttgart, 0.19999), (Ulm, 0.86000), (Würzburg, 2.16)}	{(München Hbf, 0.0) , (M-Pasing, 0.07) , (M-Ost, 0.1) , (Tutzing, 0.4) , (München Flughafen, 0.4) , (Nürnberg Flughafen, 0.537) , (Stuttgart Flughafen, 0.593) , (Augsburg, 0.61000) , (Nürnberg, 0.07999)}

Tabelle 8.5: *Open*- und *Closed*-Menge für die Nullheuristik

Ergebnisse für etwa 1000 Knoten

Für etwa 1000 Knoten ergaben sich folgende erwartete Berechnungszeiten und Anzahl expandierender Knoten:

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Frankf. (M) Hbf, Berlin Hbf_6	6,6	693,8	86,6	6,2	577,2	41,4
Hannover_4, F Flughafen Fernbahnhof_4	3,8	588,6	56,4	6,8	658,2	69,8
Greifswald_6, Hannover	4,6	588,6	38,2	137,8	1168,2	156,2
M-Pasing_5, Gotha_3	10,6	729,0	86,2	27,6	1034,6	107,6
$\text{Oldenburg (Oldb)}_3$, HH Dammtor_5	17,0	689,4	94,0	6,6	600,6	106,0
Stuttgart F_4, München F_1	6,2	523,4	40,2	3,2	530,8	33,8
Minden_1, München F_5	6,2	558,0	17,0	13,6	584,8	78,6
Nürnberg F_4, Köln Hbf_3	6,4	599,8	61,8	6,2	552,6	44,6
Herford_1, Stuttgart F_2	6,2	581,0	21,8	16,6	714,2	80,6
Frankfurt/M F_6, Dresden F_5	3,0	532,4	37,0	6,8	530,0	32,6
Minden_5, Frankf. (M) Hbf_5	8,0	648,4	58,0	79,6	1547,6	126,2
Heidelberg, Nürnberg F_4	8,2	578,8	18,0	11,0	598,4	71,2
Nürnberg F_3, Nürnberg F_2	3,0	530,6	34,0	3,0	516,8	30,0
$\text{Kassel-Wilhelmshöhe}_6$, Duisburg	8,2	633,4	46,8	143,6	1255,2	172,8
München F_1, Stuttgart_4	9,2	650,8	76,0	7,2	547,6	43,4

Tabelle 8.6: Berechnungszeit: Bahn, Flugzeug für etwa 1000 Knoten

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Frankf. (M) Hbf, Berlin Hbf_6	28	26	406	7	6	147
Hannover_4, F Flughafen Fernbahnhof_4	3	3	125	27	21	308
Greifswald_6, Hannover	18	15	173	1075	1049	1091
M-Pasing_5, Gotha_3	52	37	472	215	123	668
$\text{Oldenburg (Oldb)}_3$, HH Dammtor_5	97	31	523	58	12	536
Stuttgart F_4, München F_1	1	1	108	1	1	109
Minden_1, München F_5	6	3	40	68	7	342
Nürnberg F_4, Köln Hbf_3	17	10	248	3	3	107
Herford_1, Stuttgart F_2	11	7	55	134	33	435
Frankfurt/M F_6, Dresden F_5	1	1	81	1	1	89
Minden_5, Frankf. (M) Hbf_5	24	19	256	612	1154	825
Heidelberg, Nürnberg F_4	9	4	49	68	8	365
Nürnberg F_3, Nürnberg F_2	1	1	96	1	1	98
$\text{Kassel-Wilhelmshöhe}_6$, Duisburg	21	15	185	1054	937	1079
München F_1, Stuttgart_4	27	16	356	2	2	108

Tabelle 8.7: Anzahl expandierender Knoten: Bahn, Flugzeug für etwa 1000 Knoten

Ergebnisse für etwa 2000 Knoten

Für etwa 2000 Knoten ergaben sich folgende erwartete Berechnungszeiten und Anzahl expandierender Knoten:

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Frankf. (M) Hbf_{10}, Frankf.(M) Hbf	12,4	2237,6	101,8	16,8	2197,0	162,4
Frankf. (M) Hbf_{13}, Minden_4	74,8	3300,2	250,6	26,4	2301,8	180,4
München F_{11}, Züssow_{12}	427,8	5690,6	447,4	13,8	1673,8	96,0
Nürnberg F_{13}, München F	10,6	2079,2	95,8	12,4	1828,2	87,8
Rostock_9, Dresden Hbf_8	23,2	2396,0	181,4	434,0	5637,6	476,8
München F_{10}, M-Pasing_{10}	94,6	2357,0	266,8	0,4	41,6	2,4

Tabelle 8.8: Berechnungszeit: Bahn, Flugzeug für etwa 2000 Knoten

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Frankf. (M) Hbf_{10}, Frankf.(M) Hbf	4	3	276	29	20	553
Frankf. (M) Hbf_{13}, Minden_4	325	150	1105	43	36	657
München F_{11}, Züssow_{12}	2107	2948	2173	14	14	236
Nürnberg F_{13}, München F	1	1	211	1	1	202
Rostock_9, Dresden Hbf_8	58	44	646	2058	2526	2136
München F_{10}, M-Pasing_{10}	427	42	1179	3	3	4

Tabelle 8.9: Anzahl expandierender Knoten: Bahn, Flugzeug für etwa 2000 Knoten

Erkenntnisse

Die Suche mit den verschiedenen Instanzen gab immer den gleichen schnellsten Weg für alle Heuristiken heraus. Die Berechnungszeit steigt mit der Anzahl an Knoten, die man betrachtet, da viel mehr mögliche Wege in Betracht gezogen werden müssen. Unsere selbst entwickelte *Advanced* Heuristik braucht am längsten im Vergleich zu den anderen Heuristiken um den schnellsten Weg auszugeben. Das liegt daran, dass er viel mehr Rechenoperationen durchführen muss, um mögliche nächste Flughäfen zu finden. Im Schnitt expandiert die *Advanced* Heuristk mindestens so viele und wenige Knoten wie die Luftlinienheuristik. Das liegt daran, dass die *Advanced* Heuristik versucht nach Bedingung den Wert für jeden Knoten so zu addieren, sodass der möglichst nahebringende Fall zum Ziel ausgewählt wird. D.h. manche kurze Wege werden so schwer gewichtet, sodass sie bei Auswahl noch nicht zutreffen und der wirklich richtige Weg gewählt wird. Die Nullheuristik schnitt im Schnitt deutlich schlechter ab als die beiden anderen Heuristiken.

Ergebnisse für unterschiedliche Parameter

Nun wird die Geschwindigkeit des Flugzeugs mit $v_F = 1000$ und $v_F = 300$ variiert und fixe Umsteigezeiten (*warteBahn*=10 min, *warteFlug*=30 min) gesetzt. Aufgrund von Zeitmangel wurde wie schon erwähnt ein neuer Ansatz für die Wartezeiten (Umsteigezeiten) bestimmt (ursprünglich: Wartezeiten der Bahn nur an Großbahnhöfe). Daraus ergeben sich folgende Berechnungszeiten und Anzahl expandierender Knoten für etwa 150, 1000 und 2000 Knoten:

(S, Z)	v_F	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Ulm, Fulda	1000	5,0	17,2	9,4	5,4	16,6	9,2
München Hbf, Jena Paradies	1000	5,0	15,4	11,2	4,0	17,2	8,6
Darmstadt, Essen	300	0,0	13,0	7,0	4,2	19,0	6,4
Frankfurt/M F, Rostock	300	4,4	32,6	12,6	3,2	12,4	7,4
Frankf. (M) Hbf, Berlin Hbf_6	1000	29,6	802,4	90,8	6,4	572,0	36,8
Minden_1, München F_5	1000	3,2	561,2	13,8	13,8	552,6	60,0
Nürnberg F_4, Köln Hbf_3	300	6,2	554,8	54,0	3,2	530,6	17,0
München F_1, Stuttgart_4	300	6,2	552,0	60,0	3,8	530,4	32,0
Frankf. (M) Hbf_{10}, Frankf. (M) Hbf	1000	12,8	2248,4	90,0	29,0	2267,0	172,0
Frankf. (M) Hbf_{13}, Minden_4	300	25,0	2401,0	244,4	17,4	2152,4	204,4

Tabelle 8.10: Berechnungszeit: Bahn, Flugzeug für unterschiedliche Parameter und Knotenzahl

(S, Z)	v_F	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
Ulm, Fulda	1000	84	8	96	87	10	109
München Hbf, Jena Paradies	1000	83	10	95	45	8	63
Darmstadt, Essen	300	28	7	53	28	8	50
Frankfurt/M F, Rostock	300	126	36	150	12	7	34
Frankf. (M) Hbf, Berlin Hbf_6	1000	204	52	586	15	7	132
Minden_1, München F_5	1000	5	3	37	95	3	287
Nürnberg F_4, Köln Hbf_3	300	10	4	256	3	3	44
München F_1, Stuttgart_4	300	14	2	366	2	2	108
Frankf. (M) Hbf_{10}, Frankf. (M) Hbf	1000	11	3	241	73	31	722
Frankf. (M) Hbf_{13}, Minden_4	300	84	42	1010	33	23	866

Tabelle 8.11: Anzahl expandierender Knoten: Bahn, Flugzeug für unterschiedliche Parameter und Knotenzahl

Erkenntnisse

Die Wartezeiten der Verkehrsmittel und die Änderung der Geschwindigkeit des Flugzeugs beeinflussten die Suche nicht. Weiterhin ergaben sich im Schnitt für alle Heuristiken eine etwas höhere Berechnungszeit. Was erstaunlich war, war die Feststellung, dass die unterschiedlichen Parameter einen Einfluss auf die Anzahl expandierender Knoten hatten. Sowohl die Luftlinienheuristik als auch die *Advanced* Heuristik expandierten weniger Knoten, jedoch die *Advanced* Heuristik deutlich weniger. Eine weitere aber zu erwartende Änderung war die Gesamtzeit der Suche von Punkt S nach Punkt Z.

8.2 Experimentelle Ergebnisse für Auto/Bahn/Flugzeug

Die folgenden Tabellen stellen die erwartete Berechnungszeit und die Anzahl expandierender Knoten für die Verkehrsmittel Auto, Bahn und Flugzeug dar. Diese sind ähnlich aufgebaut wie beim ersten Experiment. Für die Ergebnisse haben wir folgende Geschwindigkeiten festgelegt: $v_A = 80$, $v_B = 100$ und $v_F = 1000$, wobei im Abschnitt der unterschiedlichen Parameter wir die Geschwindigkeit des Flugzeugs auch wie im ersten Fall auf $v_F = 300$ gesetzt haben um zu schauen welche Bedeutung hat.

Ergebnisse für etwa 150 Knoten

Für das zweite Experiment haben wir die Anzahl von Knoten auf etwa 150 gesetzt. Hierfür werden in den Tabellen die Ergebnisse der erwarteten Berechnungszeit in ms und die Ergebnisse der Anzahl expandierenden Knoten für die verschiedenen Heuristiken dargestellt, die für die jeweilige Instanz resultieren. Wir erinnern uns daran, dass S den Startknoten, Z den Zielknoten, h_1 die Luftlinienheuristik, h_2 die *Advanced* Heuristik und h_3 die Nullheuristik bezeichnet. rand_{143} und rand_{144} stellen die Zufallspunkte dar.

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Bad Hersfeld	7,8	72,8	25,0	7,8	13,2	16,0
rand_{143}, F Süd	8,6	63,2	15,6	8,4	17,2	13,6
rand_{143}, Oldenburg (Oldb)	5,6	56,8	12,8	7,6	19,6	10,4
rand_{143}, Eberswalde	7,6	61,8	13,2	2,4	13,0	7,8
rand_{143}, Dortmund	7,4	66,2	16,0	5,8	22,4	10,8
rand_{143}, rand_{144}	6,8	61,0	13,6	10,6	65,2	21,8
rand_{143}, rand_{144}	8,2	74,0	18,0	9,0	61,8	18,2
rand_{143}, rand_{144}	10,6	63,6	20,4	9,0	67,8	17,2
rand_{143}, rand_{144}	5,0	55,2	8,4	11,0	74,4	15,4
rand_{143}, rand_{144}	8,0	63,2	16,8	9,8	61,0	21,8
rand_{143}, Münster/Osnabrück F	6,2	58,0	13,0	3,4	12,4	8,6
rand_{143}, Düsseldorf F	6,4	63,0	11,2	4,2	18,4	9,0
rand_{143}, Leipzig/Halle F	8,0	77,8	18,0	5,2	11,0	11,2
rand_{143}, Köln/Bonn F	5,6	56,8	10,4	3,0	13,2	7,8
rand_{143}, Dresden F	4,8	56,8	9,8	2,6	9,2	8,4

Tabelle 8.12: Berechnungszeit: Auto, Bahn, Flugzeug für etwa 150 Knoten

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Bad Hersfeld	104	19	123	132	8	149
rand_{143}, F Süd	14	6	33	129	10	142
rand_{143}, Oldenburg (Oldb)	57	7	95	87	14	103
rand_{143}, Eberswalde	35	6	73	23	7	44
rand_{143}, Dortmund	73	22	87	98	19	124
rand_{143}, rand_{144}	17	5	36	134	5	161
rand_{143}, rand_{144}	69	6	94	95	6	110
rand_{143}, rand_{144}	59	7	77	56	6	90
rand_{143}, rand_{144}	8	2	8	13	2	15
rand_{143}, rand_{144}	62	5	92	82	5	105
rand_{143}, Münster/Osnabrück F	12	4	20	25	4	48
rand_{143}, Düsseldorf F	7	2	36	72	11	87
rand_{143}, Leipzig/Halle F	4	2	13	128	4	146
rand_{143}, Köln/Bonn F	2	2	10	25	4	50
rand_{143}, Dresden F	4	3	34	21	3	43

Tabelle 8.13: Anzahl expandierender Knoten: Auto, Bahn, Flugzeug für etwa 150 Knoten

Ergebnisse für etwa 1000 Knoten

Für etwa 1000 Knoten ergaben sich folgende erwartete Berechnungszeiten und Anzahl expandierender Knoten:

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Frankf. (M) Hbf_6	79,2	2923,8	160,4	154,4	530,6	173,6
rand_{143}, Stuttgart_3	76,0	2893,6	97,4	149,8	546,2	162,8
rand_{143}, Züssow_6	298,8	2204,6	337,0	163,2	618,4	183,2
rand_{143}, Basel Bad Bf_2	411,0	2806,8	286,0	161,4	520,4	173,4
rand_{143}, Frankf. (M) Hbf_4	77,0	2851,4	153,0	153,2	584,0	171,8
rand_{143}, rand_{144}	193,8	5097,2	366,8	770,8	4881,6	779,0
rand_{143}, rand_{144}	749,6	4497,6	751,8	521,2	3832,2	707,2
rand_{143}, rand_{144}	653,2	4433,8	841,6	832,0	3969,8	831,2
rand_{143}, rand_{144}	688,8	4599,2	838,2	523,2	4600,0	491,4
rand_{143}, rand_{144}	601,8	4498,6	468,4	509,0	4391,6	480,6
rand_{143}, München F_1	85,2	3933,0	108,0	245,0	1147,0	259,8
rand_{143}, Stuttgart F_4	113,4	3854,6	112,2	178,8	836,0	232,6
rand_{143}, Stuttgart F_1	76,4	2802,2	120,0	144,4	565,6	151,0
rand_{143}, Saarbrücken F_3	87,6	3094,6	123,8	142,2	506,0	163,2
rand_{143}, Dresden F_4	69,8	3293,0	112,6	149,4	575,0	163,2

Tabelle 8.14: Berechnungszeit: Auto, Bahn, Flugzeug für etwa 1000 Knoten

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Frankf. (M) Hbf_6	33	25	301	1127	5	1112
rand_{143}, Stuttgart_3	19	6	20	1128	2	1108
rand_{143}, Züssow_6	1288	1097	1395	1081	16	1095
rand_{143}, Basel Bad Bf_2	1347	805	1037	1134	18	1121
rand_{143}, Frankf. (M) Hbf_4	41	13	293	1096	3	1101
rand_{143}, rand_{144}	46	5	362	1447	4	1542
rand_{143}, rand_{144}	1080	2	1209	1484	2	1528
rand_{143}, rand_{144}	1435	7	1521	1992	9	1978
rand_{143}, rand_{144}	1851	21	2012	1086	21	1139
rand_{143}, rand_{144}	1217	11	1264	918	8	1099
rand_{143}, München F_1	27	12	89	1081	40	1094
rand_{143}, Stuttgart F_4	4	2	84	1124	2	1109
rand_{143}, Stuttgart F_1	4	2	138	1036	2	1075
rand_{143}, Saarbrücken F_3	9	4	78	1123	3	1108
rand_{143}, Dresden F_4	9	3	101	1122	9	1108

Tabelle 8.15: Anzahl expandierender Knoten: Auto, Bahn, Flugzeug für etwa 1000 Knoten

Ergebnisse für etwa 2000 Knoten

Für etwa 2000 Knoten ergaben sich folgende erwartete Berechnungszeiten und Anzahl expandierender Knoten:

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Frankf. (M) Hbf_1	246,0	10679,4	395,2	428,0	2084,0	493,8
rand_{143}, Hannover_8	280,0	10488,4	551,8	126,6	4192,2	534,4
rand_{143}, rand_{144}	1001,0	9636,2	1115,4	419,6	9520,8	761,6
rand_{143}, rand_{144}	813,0	10036,2	985,0	1446,6	7728,6	1483,2
rand_{143}, München F_{10}	242,8	10365,8	375,6	412,0	1993,8	466,0
rand_{143}, München F	240,0	10040,0	373,2	343,4	2012,8	428,6

Tabelle 8.16: Berechnungszeit: Auto, Bahn, Flugzeug für etwa 2000 Knoten

(S, Z)	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Frankf. (M) Hbf_1	21	11	305	1851	9	2070
rand_{143}, Hannover_8	60	38	701	271	1006	2238
rand_{143}, rand_{144}	1877	2	2151	436	2	1318
rand_{143}, rand_{144}	1587	10	1968	3811	18	3792
rand_{143}, München F_{10}	23	9	250	1847	2	2031
rand_{143}, München F	15	12	238	1362	6	1789

Tabelle 8.17: Anzahl expandierender Knoten: Auto, Bahn, Flugzeug für etwa 2000 Knoten

Erkenntnisse

Wie bereits im Fall davor wird bei allen Heuristiken der selbe schnellste Weg ausgegeben. Auch die erhöhte Berechnungszeit durch nun drei Verkehrsmittel ist klar. Interessant aber war zu sehen, dass die *Advanced Heuristik* in allen Tests bis auf einen, weniger Knoten expandiert hat als die Luftlinienheuristik. Bei dem einen Test hatten beide Heuristiken gleich viele Knoten expandiert. Großer Verlierer war wie zu erwarten die Nullheuristik.

Ergebnisse für unterschiedliche Parameter

Nun wird die Geschwindigkeit des Flugzeugs von $v_F = 1000$ zu $v_F = 300$ variiert und fixe Umsteigezeiten gesetzt. Daraus ergeben sich folgende Berechnungszeiten und Anzahl expandierender Knoten für etwa 150, 1000 und 2000 Knoten (In Klammern wird angegeben, welche Fall betrachtet wurde, sprich Knotenzahl:Test):

(S, Z)	v_F	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Dortmund	1000	6,2	59,4	15,8	6,4	12,4	9,4
rand_{143}, rand_{144} (K:150,T:8)	1000	6,2	59,4	21,8	6,2	68,8	15,6
rand_{143}, rand_{144} (K:150,T:9)	300	7,4	56,2	12,6	9,4	60,4	9,2
rand_{143}, Düsseldorf F	300	6,4	57,4	15,6	3,0	12,6	12,6
rand_{143}, Frankf. (M) Hbf_4	1000	75,6	2944,8	156,4	156,6	606,2	175,0
rand_{143}, rand_{144} (K:1000,T:7)	1000	255,2	2628,2	298,0	332,0	2636,8	372,4
rand_{143}, rand_{144} (K:1000,T:9)	300	347,2	2725,6	457,2	231,8	2977,6	311,4
rand_{143}, Stuttgart F_4	300	72,8	2769,6	108,6	146,6	529,8	171,4
rand_{143}, rand_{144} (K:2000,T:4)	1000	812,4	10620,6	943,8	1430,6	8089,4	1504,0
rand_{143}, rand_{144} (K:2000,T:3)	300	526,0	9628,6	972,6	332,6	10021,6	737,6

Tabelle 8.18: Berechnungszeit: Auto, Bahn, Flugzeug für unterschiedliche Parameter und Knotenzahl

(S, Z)	v_F	$h_1(S, Z)$	$h_2(S, Z)$	$h_3(S, Z)$	$h_1(Z, S)$	$h_2(Z, S)$	$h_3(Z, S)$
rand_{143}, Dortmund	1000	87	15	103	77	10	92
rand_{143}, rand_{144} (K:150,T:8)	1000	44	5	59	48	5	75
rand_{143}, rand_{144} (K:150,T:9)	300	6	2	7	8	2	11
rand_{143}, Düsseldorf F	300	15	5	62	42	3	86
rand_{143}, Frankf. (M) Hbf_4	1000	43	10	369	998	3	1036
rand_{143}, rand_{144} (K:1000,T:7)	1000	924	2	1063	1370	2	1442
rand_{143}, rand_{144} (K:1000,T:9)	300	1574	10	1941	749	3	1038
rand_{143}, Stuttgart F_4	300	4	2	84	1078	2	1083
rand_{143}, rand_{144} (K:2000,T:4)	1000	1331	6	1705	3688	12	3699
rand_{143}, rand_{144} (K:2000,T:3)	300	668	2	1909	120	2	1044

Tabelle 8.19: Anzahl expandierender Knoten: Auto, Bahn, Flugzeug für unterschiedliche Parameter und Knotenzahl

Erkenntnisse

Der Weg rand_{143} nach Dortmund ergab mit Wartezeiten einen anderen schnellsten Weg für alle Heuristiken als ohne Wartezeiten. Das bedeutet, dass es für die Wegbestimmung knappe Entscheidungen zwischen Flug und Bahn gegeben hat. Insgesamt führte auch dieser Versuch zum Ergebnis, dass die Luftlinienheuristik und die *Advanced* Heuristik bei verschiedenen Parameter weniger Knoten expandiert. Die Nullheuristik konnte leider auch hier zu keinem positiven Ergebnis kommen.

KAPITEL

NEUN

Zusammenfassung und Fazit

Wir haben die schnellste Verbindung in Verkehrssystemen mit verschiedenen Verkehrsmitteln unter der Verwendung des A^*-Algorithmus untersucht und die Berechnungszeit verschiedener monotoner Heuristiken mit heuristischen Erweiterungen verglichen. Der erste Fall untersuchte die Suche mit den zur Verfügung stehenden Verkehrsmittel Bahn und Flugzeug, bei dem man nur von Knoten zu Knoten gesucht hat. Der zweite Fall analysierte dasselbe, aber mit Auto, Bahn und Flugzeug. Hierbei waren Start- und Endpunkt beliebig. Die Annahme war dann immer, dass es eine Straße zu jedem Knoten gibt. Zum Knoten K wurde dann der nächste Bahnhof bzw. Flugplatz (Luftlinie) gesucht.
Um eine sinnvolle Suche durchzuführen, wurden viele Knoten erzeugt, indem eine Deutschlandkarte mit Bahnhöfen und Flughäfen mehrfach gespiegelt und verschiedene Parameter wie verschiedene Geschwindigkeiten und fixe Umsteigezeiten angewendet wurden, um den Einfluss auf die Suche zu testen.
Die untersuchten monotonen Heuristiken für die Suche nach der schnellsten Verbindung zwischen zwei Orten waren die Nullheuristik, wo man die Schätzfunktion $h(N)$ für alle Knoten N auf 0 setzt, die Luftlinienheuristik und die von mir selbst bestimmte Heuristik, die ich als *Advanced* Heuristik bezeichnet habe. Hierfür sucht man die nächsten Flughäfen N_F zu N und Z_F zu Z. Dann nimmt man an, dass die Flughäfen auf den Fluglinien liegen. Ebenfalls muss die Monotonie der *Advanced* Heuristik bewiesen werden.

Unsere Ergebnisse unterstützen die Vermutung, dass die Nullheuristik eine größere Anzahl an Knoten expandiert als die anderen Heuristiken. Jedoch besitzt sie wie vermutet keine sehr hohe Berechnungszeit, sondern ist meistens nur etwas langsamer als die Luftlinienheuristik. Das kann allerdings daran liegen, dass kein riesiges Netz von über 1000000 Knoten betrachtet wurde. Die *Advanced* Heuristik hat im Gegensatz zu den beiden anderen Heuristiken eine extrem hohe Berechnungszeit, die bei jedem Schritt auf der Suche nach dem nächsten Flughafen ist. Somit werden jedes Mal viele Knoten neu verglichen. Allerdings könnte man es für möglich halten, dass man die *Advanced* Heuristik besser implementieren kann, sodass er besonders auf einem riesigen Netz viel schneller rechnet als die Nullheuristik.
Dadurch, dass die Nullheuristik keine Heuristik besitzt, sucht der Algorithmus mit

dieser Heuristik durch gleichmäßige Expansion in jede Richtung, aber die beiden anderen Heuristiken scannen den Bereich nur in Richtung des Ziels und sind somit zielgerichteter. Wie man sich sicher vorstellen kann, hat diese Nullheuristik dadurch gewöhnlich zur Folge, dass eine viel größere Fläche untersucht wird, bevor das Ziel gefunden werden kann. Das macht die Nullheuristik langsamer als die restlichen Heuristiken. Aber alle haben ihre eigenen Vorteile, wie z.B. die Luftlinienheuristik und die *Advanced* Heuristik, die meistens bei gegebenen Start- und Zielpunkten verwendet werden, im Gegensatz dazu die Nullheuristik, die bei unbekannten Zielpunkten verwendet wird.

Die fixen Umsteigezeiten und die variablen Geschwindigkeiten haben die Suche soweit beeinflusst, dass die Suche an Laufzeit zugenommen hat, was selbstverständlich, jedoch die *Advanced* Heuristik seine Stärke gezeigt. Nämlich bei der Anzahl an expandierenden Knoten. Weiterhin haben die verschiedenen Parameter die Suche dazu gebracht, bei einem Testfall einen anderen schnellsten Weg auszuwählen.
Der A^* ist sowohl vollständig (findet einen Weg, falls einer existiert) und optimal (findet stets den kürzesten und schnellsten Weg), falls eine monotone Schätzfunktion verwendet wird.
Dass die *Advanced* Heuristik sowohl unterschätzend als auch monoton ist, konnten wir sowohl experimentell als auch schriftlich nachweisen (der schnellste Weg zusammen mit den anderen Heuristiken wurde stets gefunden).
Die Tatsache, dass die Luftlinienheuristik besser informiert als die Nullheuristik, aber dass zugleich die *Advanced* Heuristik besser informiert als die Luftlinienheuristik, konnten wir ebenfalls zeigen.
Wir haben dargelegt, dass die *Advanced* Heuristik nicht nur eine einfache Heuristik, sondern auch eine gute Schätzfunktion für den A^*-Algorithmus ist. Der A^*-Algorithmus ist unter Beachtung wichtiger Zwecke in der Lage andere Faktoren zu betrachten. Obwohl nur ein kleines Verkehrsnetz betrachtet wurde, könnte dieser A^*-Algorithmus auch seine Stärken auf viel größeren Netzen zeigen.
Außerdem bestätigen die Experimente, die auf realen Daten basieren, dass diese Heuristiken eine schnelle Berechnung auf schnelle Verbindungen erlauben. Allerdings muss man an dieser Stelle erwähnen, dass wir einfachheitshalber davon ausgegangen sind, dass z.B. die Distanz zwischen allen Punkten linear ist, jedoch sind Autobahnen in der realen Welt nicht linear. In manchen Grenzwert-Situationen kann die Verwendung der Autobahn zu langen Umwegen führen. Deshalb sollte man die Autobahnen als ein separates Netzwerk behandeln. Falls die in der Arbeit aufgezeigte und bewiesene Heuristik in der realen Welt Anwendung finden soll, müssen für die Auto- und Bahnstrecken separate Netzwerke implementiert werden, die nicht-lineare Faktoren wie Kurven berücksichtigen.

ANHANG
A

Anhang

A.1 Inhalt des Datenträgers

- „thesis.pdf“: digitale Version der Arbeit
- „Schreiben29092016“: LaTex Datei zusammen mit allen dafür benötigten Ressourcen
- „Astern“: enthält Quellcodestücke und eine Javadoc unter dem Pfad Astern/-doc
- „Ergebnisse“: Darin sind alle Versuchsergebnisse enthalten
- „HOWTOINSTALL“: Anleitung zum Starten des Programms
- „README“: genaue Beschreibung des Inhalts vom Datenträger
- „Karte“: In dem Ordner „Karte“ befinden sich das ICE-Liniennetz und die Übersichtskarte der Flughäfen, die die Standorte der Bahn und des Flugzeugs unsere Ausgangskarte ergaben. Aus der Übersichtskarte der Flughäfen wurden nur die blau markierten Flughäfen als Standorte des Flugzeugs unserer Ausgangskarte verwendet

A.2 Visualisierung

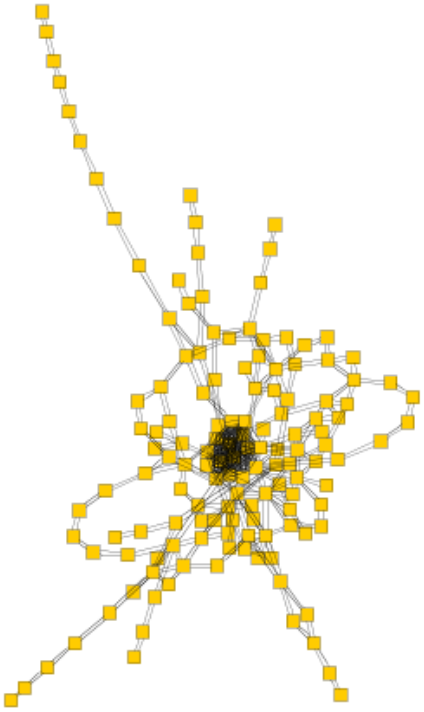


Abbildung A.1: Organisches Layout für etwa 150 Knoten

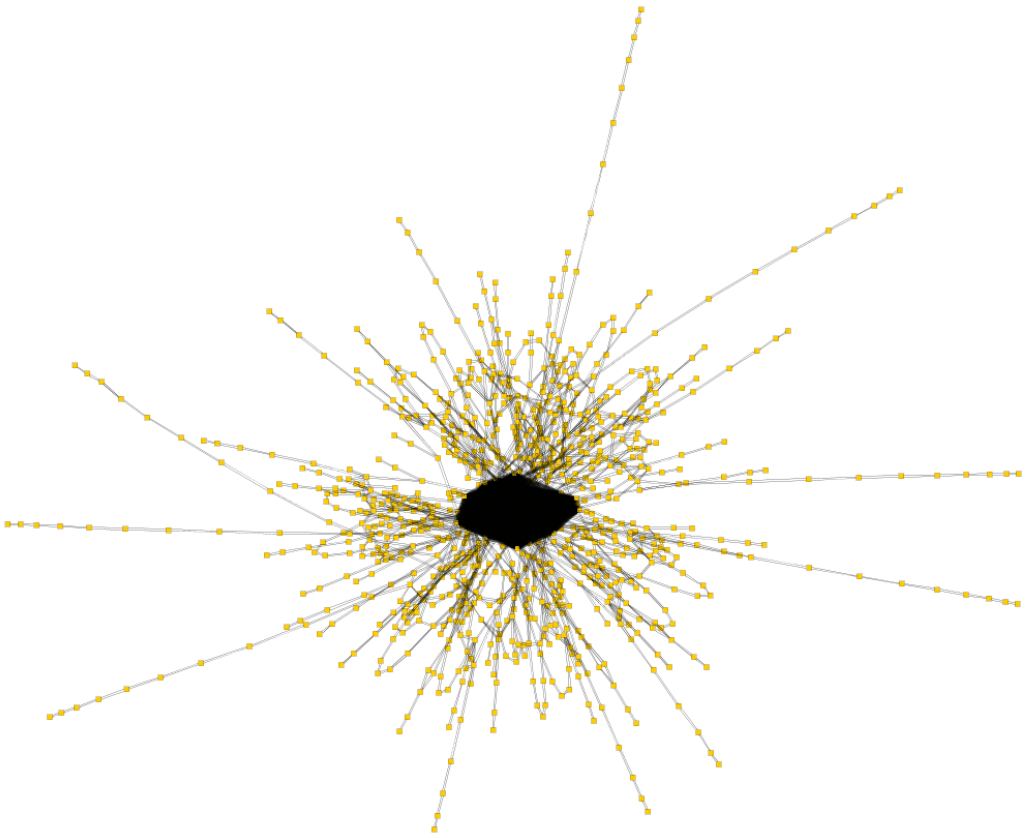


Abbildung A.2: Organisches Layout für etwa 1000 Knoten

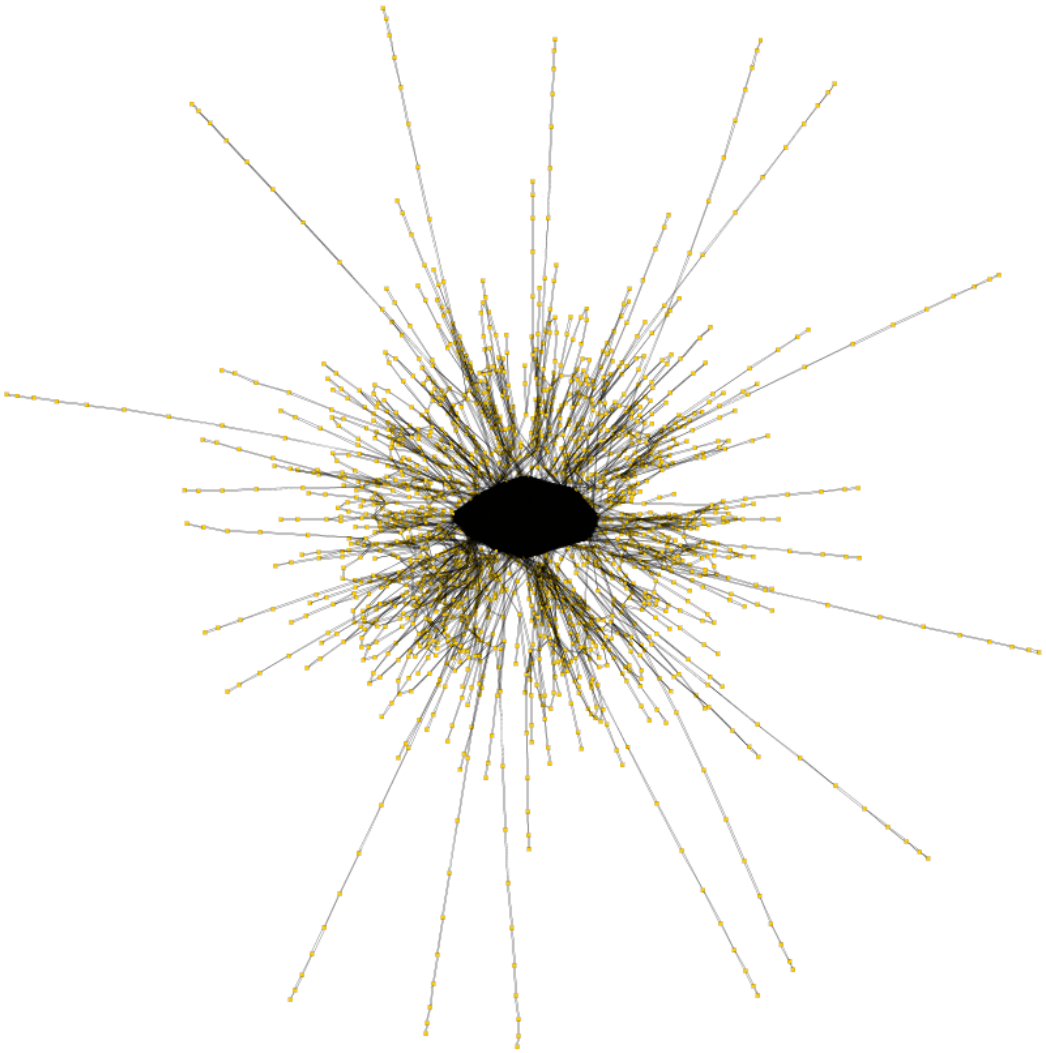


Abbildung A.3: Organisches Layout für etwa 2000 Knoten

LITERATURVERZEICHNIS

[AGS05] Rishabh Luthra Abhishek Goyal, Prateek Mogha and Ms. Neeti Sangwan. Fast shortest path algorithm for road network and implementation. *Carleton University School of Computer Science COMP 4905 HONOURS PROJECT Fall Term*, 2005.

[AGS14] Rishabh Luthra Abhishek Goyal, Prateek Mogha and Ms. Neeti Sangwan. Path finding: A* or dijkstra's? *International Journal of Innovative Trends in Engineering*, 2014.

[Bau] Stefan K. Baur. Was sie schon immer über spiele wissen wollten... `https://www2.informatik.uni-erlangen.de/EN/teaching/WS2005/GameAlgHS/download/Baur-AStar.pdf`. Zugriffsdatum: 29.09.2016.

[Böl13] A. Bölte. *Modelle und Verfahren zur innerbetrieblichen Standortplanung*. Physica-Schriften zur Betriebswirtschaft. Physica-Verlag HD, 2013.

[Brä] Christof Bräutigam. Einsatz heuristischer suchverfahren zur erzeugung eines akrostichons. `http://www.uni-weimar.de/medien/webis/teaching/theses/braeutigam_2012.pdf`. Zugriffsdatum: 29.09.2016.

[Chea] Technische Universitaet Chemnitz. Künstliche intelligenz in der schule. `https://www.tu-chemnitz.de/informatik/KI/scripts/ws0405/KI_Schule/KI-Schule-04-lehr-1.doc`. Zugriffsdatum: 29.09.2016.

[Cheb] Technische Universitaet Chemnitz. Künstliche intelligenz in der schule. `https://www.tu-chemnitz.de/informatik/KI/scripts/ws0405/KI_Schule/KI-Schule-04-lehr-1.pdf`. Zugriffsdatum: 29.09.2016.

[Ern03] I.C. Ernst. *Location Based Services in Deutschland: Lösungsmöglichkeiten und Anwendungsbeispiele*. Diplom.de, 2003.

[Ert09] W. Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Computational Intelligence. Vieweg+Teubner Verlag, 2009.

[FBM10] Mohamad El Falou, Maroua Bouzid, and Abdel-Illah Mouaddib. Dec-a*: A decentralized a* algorithm. *AAAI Technical Report*, 2010.

[GRRW10] F. Gurski, I. Rothe, J. Rothe, and E. Wanke. *Exakte Algorithmen für schwere Graphenprobleme*. eXamen.press. Springer Berlin Heidelberg, 2010.

[Hau87] John Haugeland. *Künstliche Intelligenz - Programmierte Vernunft?* Mcgraw-Hill Book Company Gmbh, 1987.

[HL13] D. Hartmann and K. Lehner. *Technische Expertensysteme: Grundlagen, Programmiersprachen, Anwendungen*. Springer Berlin Heidelberg, 2013.

[ICCC13] Feng-Han Yeh Dung-Lin Hsieh Ing-Chau Chang, Hung-Ta Tai and Siao-Hui Chang. A vanet-based formula route planning algorithm for travelling time- and energy-efficient gps navigation app. *Department of Computer Science and Information Engineering, National Changhua University of Education*, 2013.

[JL] Sanjay Jena and Nils Liebelt. Heuristische algorithmen am beispiel des a*-algorithmus / 8-puzzle. `http://www.gm.fh-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/E_gelb/ALA-HeuristischeAlgorithmen-Jena-Liebelt.pdf`. Zugriffsdatum: 29.09.2016.

[Kön10] R. König. *Simulation und Visualisierung der Dynamik räumlicher Prozesse: Wechselwirkungen zwischen baulichen Strukturen und sozialräumlicher Organisation städtischer Gesellschaften*. Computersimulationen in den Sozialwissenschaften. VS Verlag für Sozialwissenschaften, 2010.

[Kos] Sven Kosub. Einführung in die informatik 2 – bäume & graphen –. `http://algo.uni-konstanz.de/lehre/ss09/ei/baeume-und-graphen-www.pdf`. Zugriffsdatum: 29.09.2016.

[Kum08] E. Kumar. *Artificial Intelligence*. I.K. International Publishing House Pvt. Limited, 2008.

[Len02] M. Lenzen. *Natürliche und künstliche Intelligenz: Einführung in die Kognitionswissenschaft*. Campus Einführungen. Campus Verlag, 2002.

[MT14] D. Mallett and D. Tammet. *Die Poesie der Primzahlen*. Carl Hanser Verlag GmbH & Company KG, 2014.

[Mue] Technische Universitaet Muenchen. Suchstrategien. `http://wind.in.tum.de/seminare/web/WS0001/vortrag01.html`. Zugriffsdatum: 29.09.2016.

[Nap] Universitas Napocensis. Suchverfahren-informierte suche. `https://math.ubbcluj.ro/~csacarea/wordpress/wp-content/uploads/V3KI.pdf`. Zugriffsdatum: 29.09.2016.

[Obe07] W. Oberstenfeld. *T S P - Traveling Salesman Problem: Das Lösungsverfahren*. GRIN Verlag, 2007.

[Pan] Sven Eric Panitz. Grundlagen der kuenstlichen intelligenz. `http://www.cs.hs-rm.de/~panitz/ki/skript.pdf`. Zugriffsdatum: 29.09.2016.

[Pea84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley, 1984.

[PEHR68] N. J. Nilsson P. E. Hart and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[PH72] B. Raphael P.E. Hart, N.J. Nilsson. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter 37*, pages 28–29, 1972.

[PXJW] Kai Rothaus Prof. Xiaoyi Jiang and Steffen Wachenfeld. Küenstliche intelligenz. `http://cvpr.uni-muenster.de/teaching/ws05/kiWS05/script/KI-Kap04-2.pdf`. Zugriffsdatum: 29.09.2016.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence, a modern approach*. Prentice Hall, 1995.

[RN04] Stuart J. Russell and Peter Norvig. *Künstliche Intelligenz. Ein moderner Ansatz*. Pearson Studium, 2., überarb. a. edition, 2004.

[SD07] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[SH06] J. Smed and H. Hakonen. *Algorithms and Networking for Computer Games*. Wiley, 2006.

[uDDSa] Prof. Dr. Manfred Schmidt-Schauß und Dr. David Sabel. Einfuehrung in die methoden der kuenstlichen intelligenz ss 2016. `http://www.ki.informatik.uni-frankfurt.de/lehre/SS2016/KI/skript/skript-KI.pdf`. Zugriffsdatum: 29.09.2016.

[uDDSb] Prof. Dr. Manfred Schmidt-Schauß und Dr. David Sabel. Einfuehrung in die methoden der kuenstlichen intelligenz ws 2012/13. `http://www.ki.informatik.uni-frankfurt.de/lehre/WS2012/KI/skript/skript11Feb13.pdf`. Zugriffsdatum: 29.09.2016.

[uWFR] Melanie Herzog und Wolfgang Ferdinand Riedl. Kürzeste wege - wie kommt man am schnellsten von münchen nach stuttgart. `http://www.doc.ma.edu.tum.de/Hall/KW.pdf`. Zugriffsdatum:29.09.2016.

[Wal] Heiko Waldschmidt. Vergleich von pathfinding-algorithmen. `https://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/PLM/Dokumente/Master_Bachelor_Diplom/masterarbeit.pdf`. Zugriffsdatum: 29.09.2016.

[Wel] Max Welling. A-starsearch. `https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf`. Zugriffsdatum: 29.09.2016.

[WS04] Dieter Werner and Uwe Schneider. *Taschenbuch der Informatik*. Carl Hanser Verlag GmbH & Co. KG; Auflage: 5, 2004.