W-Seminararbeit
aus dem Abiturjahrgang 2014/16

W-Seminar: Extrema

Seminarleiterin: Claudia Müller

Thema der Arbeit:

Wegfindung - Vergleich verschiedener Algorithmen

Verfasser:

Maximilian Léon Stark

Abgabetermin: 10. November 2015

Erreichte Punktzahl:

Unterschrift der Seminarleiterin:

Inhaltsverzeichnis

1 Risikofaktor Navigationsgerät

„Wenn möglich, bitte wenden" auf der Autobahn. „Jetzt links abbiegen" im Kreisverkehr. Navigationssysteme können Todesfallen oder Verursacher schwerer Unglücke sein. Denn die Menschen vertrauen ihnen oft blind. So zum Beispiel erging es einem 33-jährigen im niedersächsischen Einbeck. Denn als die Polizei an der Unfallstelle eintraf, bot sich ihr ein kurioses Bild: Der Pkw steckte auf einer abwärtsführenden Fußgängertreppe fest. Jegliche Versuche des Fahrers, sein Fahrzeug zu befreien, blieben erfolglos. Bedanken darf sich dieser Mann, ebenso wie eine junge Frau, die aufgrund eines Tippfehlers in einem Ort 850 km entfernt vom gewünschten Ziel eintraf, bei der allzu freundlichen Stimme aus der Mittelkonsole [1]. Darum ist es umso wichtiger, dass „Navis" immer über aktuellste Kartendaten verfügen und auch ausgiebig auf Fehler geprüft werden.

Aber nicht nur detailreiche Straßeninformationen sind für ein gutes Navigationsgerät von Bedeutung, sondern auch die Verarbeitung dieser. Denn die Daten können noch so genau sein; wenn das Gerät keine vernünftigen Wege berechnen kann, ist es genau so unbrauchbar. Dies wird in dieser Arbeit behandelt: der Vergleich verschiedener Methoden zur Wegberechnung in einem Straßen-*Netz* oder ähnlichem. Es werden drei *Algorithmen* vorgestellt und verglichen, um festzustellen, für welche Zwecke welche Methode am zielführendsten ist. Als Werkzeug zur genaueren Untersuchung und zur Erzeugung von Vergleichsstatistiken habe ich ein Programm namens *PathFinder* geschrieben, in dem die *Algorithmen* adaptiert sind. Die Ausführungen in dieser Arbeit sind in gewisser Weise als Bedienungsanleitung des Programms zu sehen, da sich sämtliche Darstellungen und Tabellendaten darauf stützen.

Mit der Mathematik als Leitfach, im Themenbereich von Extremwertproblemen, wird konkret das *Problem des kürzesten Wegs* in Angriff genommen. Der optimale, kürzeste Weg zeichnet sich aber nicht nur durch seine Eigenschaft, am schnellsten von A nach B zu gelangen, aus, sondern auch durch die für die Bestimmung dieses Wegs benötigte Zeit und den Rechenaufwand. Denn ein „Navi", das vier Stunden rechnet, um den besten Weg zu ermitteln, wird sich nicht bei den Konsumenten durchsetzen. Gleiches gilt aber auch für ein Gerät, welches den Fahrer ohne Rechenzeit über Feld- und Waldwege lotst. Es muss eine *Balance zwischen Quantität und Qualität* gefunden werden. Und wo diese Mitte liegt, gilt es nun herauszufinden.

2 Grundlagen und Terminologie

Zunächst werden in diesem Abschnitt die grundlegenden Begriffe der Graphen-Theorie geklärt. Auch Fachbegriffe aus der Implementierung durch die Informatik werden erläutert.

Generell sind *Algorithmen* eine festgelegte Abfolge von Schritten um Daten zu verarbeiten. In der Informatik sind diese einzelnen Schritte Befehle.

Die maximale Laufzeit eines *Algorithmus*, auch genannt *Zeitkomplexität*, wird in der „big-O" -Notation (engl. für „großes O") in Form des *Landau-Symbols O* angegeben. Dabei werden sämtliche kleineren Polynome aufgrund ihres geringeren Wachstums vernachlässigt. So zum Beispiel lässt sich ein *Algorithmus*, der in der Zeit $O(n^2+5n)$ abläuft, auf die *Komplexität* $O(n^2)$ reduzieren. Hierbei stellt n die Anzahl an *Iterationen*, also Schritten dar. Diese Angabe wird als primäres Vergleichskriterium von *Algorithmen* verwendet [7].

Die Graphen-Theorie dient als Basis dieser Arbeit. Zentrale Bedeutung hat der namensgebende *Graph* $G = (V, E)$, alternativ auch *Netz* genannt, welcher aus einer Menge von *Knoten* V (von engl. „Vertex") und einer Menge *Kanten* E (von engl. „Edge") besteht.

Zeichnerisch werden *Knoten* als Punkte oder Kreise dargestellt; *Kanten* als Verbindungslinien zwischen zwei *Knoten*. Jede *Kante* hat einen *Startknoten* und einen *Endknoten*.

Sobald sich keinerlei *Kanten* in der Darstellung kreuzen, wird ein *Graph* als *planar* bezeichnet. Wenn von einer *gerichteten Kante* die Rede ist, lässt sich diese als Pfeil interpretieren, da die Verbindung unidirektional gilt. Ebenso gibt es die *gewichteten Kanten*, denen nicht nur zwei *Knoten* zugeordnet werden, sondern zusätzlich noch ein Gewicht w (von engl. „Weight"), ein Zahlenwert, der als „Reise" -Kosten der Verbindung zwischen den beiden *Knoten* gesehen werden kann.

In der Wegfindung ist ein *Weg* P (von engl. „Path") als geordnete Abfolge von *Knoten* definiert. Da in der Regel jedes *Knoten*paar nur einfach verbunden ist, reicht in der Implementierung diese Annahme aus.

Unter *Backtracking* versteht man in der Wegfindung das rückwärtige Abbarbeiten der Suchergebnisse eines *Pathfinding-Algorithmus* vom *Zielknoten* aus.

Somit erhält man den gewünschten Weg als Ergebnis. Visuell wird der *Graph* durch

einen *Layout-Algorithmus* dargestellt, der allen *Knoten* durch gewisse Berechnungen Positionen zuteilt (vgl. Abschnitt 5).
Um ein *Netz* zu generieren, wird eine Zufallsfunktion verwendet. Hierzu wird ein standardisierter *Pseudozufalls*-Generator verwendet [2]. Dieser generiert kaum oder nur schwer vorhersagbare Abfolgen von Zahlen. Aufgrund der nicht echten Zufälligkeit wird ein sogenanntes *Seed*-System benutzt, eine spezielle Zahl, mit deren Übergabe an den Generator stets die selbe Zahlenfolge erzeugt werden kann.

3 Aufbau und Bedienung des Programms *PathFinder*

Das selbstgeschriebene Programm *PathFinder*, im eigentlichen Fokus stehend, fungiert sowohl als visuelle Möglichkeit der Darstellung von *Graphen*, als auch als Quelle für Vergleichsdaten und Messungen in selbst erzeugten Szenarien. Geschrieben ist die Anwendung in der Programmiersprache *Java* unter Verwendung der *JavaFX*-Standardbibliothek [3] und umfasst über 3000 Zeilen Code in 39 Quelldateien.[1]

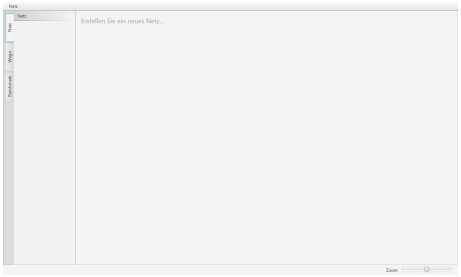

Abb. 1: Die Start- und Hauptansicht von *PathFinder*

Auf den ersten Blick ist die Anwendungsoberfläche in zwei größere Bereiche aufgeteilt. Im linken, kleineren Seitenbereich werden detaillierte Informationen über den *Graphen*, bereits berechnete Wege und die Konfigurationsmöglichkeiten neuer Wege, in mehrere „Tabs" unterteilt, angezeigt. Der große rechte Bereich, zu Beginn der Anwendung nur mit „Erstellen Sie ein neues Netz..."(Abb. 1) beschriftet, dient als Hauptansicht von sowohl des *Graphen*, als auch der Vergleichsstatistiken und Tabellen.
Die Bedienung kann vollständig mit der Maus erfolgen, da sich sämtliche Funktionen visuell intuitiv und minimalistisch präsentieren. Nur vereinzelt führen Tastatureingaben oder „Hotkeys" zu mehr Komfort oder Genauigkeit der Anwendung. So kann beispielsweise das *Relayout* (siehe Abschnitt 5) des *Graphen* per „L" Taste, das *Generieren* (siehe Abschnitt 4) eines neuen *Netzes* unter Benutzung von „N" erfolgen.

[1] siehe Anhang: CD-ROM

4 Konstruktion eines Graphen in *PathFinder*

Im nächsten Schritt wird nun ein *Graph* erzeugt und die Funktionsweise des Generators betrachtet. Durch Klicken auf „Erstellen Sie ein neues Netz...” wird ein Dialog-Fenster geöffnet (Abb. 2), welches verschiedene Generierungs-*Parameter* zur Konfiguration anbietet.

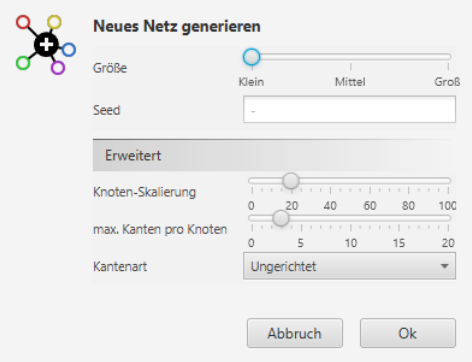

Abb. 2: Dialog zur *Netz*-Generierung

Unterteilt sind diese Einstellungen in zwei Bereiche: *Generell* und *Erweitert*. *Generelle* Optionen sind für den einfachen Gebrauch ausreichend mit einem Regler für die Größe s_g und ein *Seed*-Eingabefeld ausgestattet. Der *Seed* wird verwendet, um die Möglichkeit zu haben, in späteren Tests mit dem gleichen *Graphen* zu arbeiten.

Im *Erweitert*-Bereich lässt sich die Generierung aufs Genaueste einstellen. So können die Anzahl an Maximalknoten n_{max} und die maximale *Kanten*-Anzahl e_{max} pro *Knoten* festgelegt werden. Ebenso kann die Wahl zwischen drei Typen t von *Kanten* getroffen werden: *Ungerichtet*, *Gemischt* und *Gerichtet*, was alle *Kanten* des zu generierenden *Graphen* betrifft. Die Option *Gemischt* bewirkt, dass die Gerichtetheit jeder *Kante* zufallsbedingt ist.

Durch Bestätigen per Klick auf „Ok”wird der Generator mit diesen *Parametern* gestartet und ein *Netz* erzeugt.

Zunächst wird der *Seed* für den Zufallsgenerator gesetzt. Danach wird aus den gegebenen Grenzwerten die tatsächliche Menge von *Knoten* berechnet und in den *Graphen* eingesetzt. Daraufhin wird für jeden *Knoten* eine Anzahl an *Kanten* bestimmt. Durch das „Clampen”, d.h Einzwicken, Eingrenzen, der Start- und Generierungswerte mit

$$e = max\Big(1,\ R\Big(0,\ min\Big(\frac{n}{2} - 1,\ e_{max}\Big)\Big)\Big)$$

$$\begin{pmatrix} max(a,b) \rightarrow \text{Größere der beiden Parameter} \\ min(a,b) \rightarrow \text{Kleinere der beiden Parameter} \end{pmatrix}$$

wird gewährleistet, dass der Generator nicht mehr *Kanten* platzieren kann, als eindeutig möglich ist. Jetzt wird versucht, sämtliche *Knoten* durch zufällige Wahl mit einem anderen *Knoten* zu verbinden, wobei der jeweils gesuchte *Knoten* weder der

Ausgangsknoten selbst, noch ein bereits verbundener *Knoten* sein soll. Sobald eine Kombination gefunden wurde, wird die entsprechende *Kante* mit einem ebenfalls

Alg. 1 *Graph-Generator*

geg.: Zufallsgenerator R, max. Kantengewicht $W_{max} = 30$
ges.: Graph g

```
 1: prozedur GENERIEREGRAPH(seed, s_g, n_max, e_max, t)
 2:     setze Seed von R zu seed
 3:     sei n R(n_max/2, n_max) * s_g   ▷ Zufällige Anzahl im Interval [n_max/2; n_max[
 4:     füge n Knoten zu g hinzu
 5:     für i = 0 → n wiederhole
 6:         sei e max(1, R(0, min(n/2 − 1, e_max)))
 7:         für j = 0 → e wiederhole
 8:             sei index i
 9:             wiederhole
10:                 sei index R(0, n)
11:             solange index gleich i oder Knoten_i mit Knoten_index verbunden
12:             sei e Kante von Knoten_i zu Knoten_index, Gewicht w = R(0, W_max)
13:             wenn t = GEMISCHT oder (t = GERICHTET und R() > R()) dann
14:                                                   ▷ R > R = Zufallstest
15:                 setze e gerichtet
16:             ende wenn
17:             füge e zu g hinzu
18:         ende für
19:     ende für
20: ende prozedur
```

zufallsgenerierten *Gewicht* erstellt. Dann wird auf Basis des *Kanten-Typs* die Gerichtetheit bestimmt und schließlich wird die *Kante* im *Graphen* platziert (Alg. 1)[2].

5 Visuelles Layout von Graphen

In vorangegangen Abschnitten wurde das grundlegende Konzept eines *Graphen* bereits dargestellt. Wenn man sich nun mit der optimalen visuellen Darstellung eines *Graphen* auseinandersetzt, begibt man sich in die Thematik der *Layouts* (von engl. „Anordnung") eines *Graphen*.

Es gibt die verschiedensten Ansätze, zu einer übersichtlichen Visualisierung zu gelangen, darunter die *force-directed algorithms* (von engl. „kraft-gerichtet" oder „kraftbasiert")[4]. Diese simulieren ein einem großen Molekül ähnelndes Konstrukt, in dem verschiedene *Kräfte*, die von *Knoten* und *Kanten* ausgehen, aufeinander wirken. Das

[2] vgl. Anhang: GraphGenerator.java

Ziel solcher Simulationen ist das *mechanische Equilibrium*, die gegenseitige Aufhebung jeglicher wirkenden *Kräfte*.

Vorteile dieser Methode sind die enorme Flexibilität der Simulation und die sehr zufriedenstellenden Resultate in Bezug auf die *Planarität* des visualiserten *Graphen*. Als Nachteil lässt sich die mitunter sehr lange Laufzeit der Berechnung sehen, die benötigt wird, um ein akzeptables Ergebnis zu erhalten; insbesondere bei sehr großen *Netzen*.

Der in *PathFinder* verwendete *Algorithmus* ist der *Fruchtermann-Reingold Algorithmus*[3], der 1991 von Thomas M. J. Fruchterman und Edward M. Reingold an der University of Illinois veröffentlicht wurde [6]. Ihre Methode verfolgt die Prinzipien, dass verbundene *Knoten* nebeneinander liegen und sämtliche *Knoten* trotzdem nicht zu nahe beieinander platziert werden sollten.

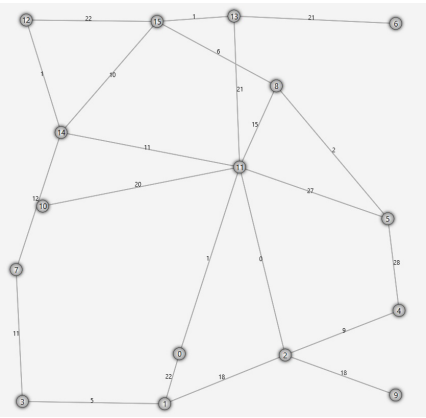

Abb. 3: F.-R. *Algorithmus*

Der Algorithmus versucht außerdem, alle *Knoten* n gleichmäßig innerhalb eines Rahmens zu verteilen, einer als Parameter w und h (von engl. „width" und „height") definierten Maximalfläche. Zunächst wird die Konstante k, die optimale Distanz zwischen *Knoten*, als

$$k = \sqrt{\frac{w \cdot h}{n}}$$

definiert. Sie findet Verwendung in den beiden *Kräften* der Simulation. Die *Kraft* F_a beschreibt die Anziehung zwischen *Knoten*, F_r die Abstoßung dieser voneinander.

$$F_a(x) = \frac{x^2}{k} \qquad F_r(x) = \frac{k^2}{x}$$

Der Ablauf der Simulation lässt sich in drei Schritten zusammenfassen. Zuerst wird F_a, danach F_r, für jeden einzelnen *Knoten* berechnet. Im dritten Schritt werden die Effekte dieser berechneten *Kräfte* umgesetzt (*Disposition*), aber nur in durch die *Temperatur* begrenzter Länge. Die *Temperatur* ist ein Wert, der bei jedem Durchlauf des *Algorithmus* bis auf 0 verringert wird, um die Verschiebungen der *Knoten* immer präziser werden zu lassen. Anfangs wird der Wert beliebig festgelegt. Der vorgeschla-

[3] [5, Kapitel 12.3, S. 386f]

gene und damit auch in *PathFinder* umgesetzte Startwert t_0, dessen Änderung auf der Funktion $t(s)$ abgebildet wird, entspricht

$$t_0 = \frac{1}{10} \cdot w \qquad t(s) = t_0 - \frac{t_0}{s_{max}} \cdot s \qquad s_{max} = s_g \cdot 500$$

Außerdem wird eine Maximalanzahl an Simulationsschritten s_{max} definiert, in deren Abhängigkeit die *Temperatur* verringert wird. Die Maximalgröße wird wie angegeben berechnet, wobei s_g die bei der *Graph*-Erzeugung angegebene *Größe* ist. Somit erhält man in *PathFinder* folgende Gesamtfunktion:

$$t(s) = \frac{w}{10} - \frac{w}{s_g \cdot 5000} \cdot s$$

6 Wegfindungs-Algorithmen

Nachdem jetzt sowohl das Programm *PathFinder*, als auch die darin angewandten Methoden zur Generierung und Visualisierung von *Graphen* erläutert worden sind, wird nun auf die Wegfindung eingegangen.

Generell ist das Ziel des *Pathfindings* je nach Aufgabenstellung den kürzesten, optimalen oder den Weg mit den wenigsten Hindernissen zwischen zwei *Knoten* zu finden; und das so schnell und recheneffizient wie möglich. Nun könnte man ganz pragmatisch an die Umsetzung herangehen und einfach den Weg zwischen jedem im *Graphen* existierenden *Knoten*paar vorberechnen und abspeichern. Somit können in der Situation selbst alle notwendigen Wegdaten bequem und schnell abgerufen werden. Nur hat ein solcher *Algorithmus* eine *Zeitkomplexität* $O(V^2)$, die für größere *Graphen* einfach untragbar ist (vgl. Unterabschnitt 6.2). Außerdem wächst die benötigte Speicherkapazität in gleichem Maße.

Man kann also nicht alles vorberechnen, sondern muss einen Großteil in "real time", also Echtzeit berechnen. In dieser Arbeit wird nur auf die vollständig in Echtzeit ablaufenden Umsetzungen eingegangen, doch *Algorithmen* wie *Kontraktions-Hierarchien*, die im Voraus eine kompaktere und performantere Version des gesamten *Graphen* errechnen, sind, besonders in sehr großen Netzwerken oder Systemen, von Bedeutung, da diese trotz des höheren Rechenaufwands enorm zu einer schnelleren Laufzeit beitragen [8].

Die nun folgenden Untersuchungen der einzelnen *Algorithmen* werden alle auf den

gleichen *Graphen* angewandt. Es wird ein *großer Graph* eingestellt, der *Seed* als 101115 bestimmt und immer der Weg vom *Knoten* Nr. 18 zu Nr. 14 gesucht.

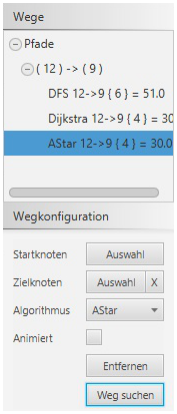

Abb. 4: Weg-Konfiguration

Um in *PathFinder* Wege berechnen zu lassen, muss auf der linken Seite des Programms der „Wege" -Tab ausgewählt sein. (Abb. 4). Im oberen Teilbereich der erscheinenden Bedienungsoberfläche werden bereits erzeugte Wege mit Details über *Start-* und *Zielknoten*, als auch *Gesamtgewicht* und *Algorithmus* angezeigt. Durch Selektion eines Weges wird dessen Verlauf in der Hauptansicht des *Graphen* angezeigt. Außerdem werden durch die Selektion automatisch *Start-* und *Zielknoten* für eine mögliche Suche des gleichen Wegs, nur mit einer anderen Methode festgelegt, und farbig hervorgehoben (grün = Start, rot = Ziel)(vgl. Abb. 5). Denn das Programm erlaubt nur die einmalige Suche von genau identischen Wegen mit dem selben *Algorithmus*.

Im unteren Bereich können *Start-* und *Zielknoten* manuell in der *Graph*anzeige vorgenommen werden. Zudem kann der gewünschte *Algorithmus* ausgewählt werden und es wird eine Schaltfläche „animiert" angeboten. Durch deren Aktivierung wird die Funktionsweise der Suche Schritt für Schritt nachvollziehbar und verzögert dargestellt.

6.1 Gröbste Züge von Intelligenz: Tiefensuche

Die Tiefensuche, oder kurz DFS (von engl. „depth first search"), hat ihren Namen von ihrer Funktionalität. Der Kerngedanke hinter dem *Algorithmus* ist das kontinuierliche „Gehen" in eine Richtung, sprich der *Graph* wird so lange wie möglich in eine Richtung *traversiert* (lat. für „entlang gehen") und erst sobald das Voranschreiten nicht mehr gegeben ist, werden Schritte zurückgegangen und andere Richtungen gewählt.

Die Richtung wird in *PathFinder* durch das *Kantengewicht* bestimmt. Es werden alle anliegenden *Kanten* eines *Knoten* der Größe nach sortiert und die Unbesuchte mit dem geringsten *Gewicht* wird gewählt. Das setzt sich so lange fort, wie es noch unbesuchte *Knoten* im *Netz* gibt, oder das Ziel nicht erreicht wurde. Falls ein Weg gefunden wird, so wird, wie in allen weiteren vorgestellten *Algorithmen*, *Back-*

tracking angewandt[4]. Der Weg wird also *rückwärts* über die *Vorgänger* konstruiert. Die Tiefensuche hat eine *Zeitkomplexität* von $O(V + E)$[5], wobei V der Anzahl an *Knoten*, und E der Anzahl an *Kanten* entspricht. Abb. 5 zeigt das Ergebnis der Tiefensuche im Fallbeispiel des Wegs von *Knoten* 18 zu 14. Es ergeben sich die Gesamtkosten $w_{dfs} = 116$ durch Addition aller $e_{dfs} = 10$ verwendeten *Kantengewichte*. Die gemessene Laufzeit der Wegsuche beträgt $t_{dfs} = 72,61709\mu s$. Hierbei ist zu beachten, dass aufgrund von variierender Prozessorauslastung und Speicherbelegungen starke zeitliche Schwankungen zwischen wiederholten Suchabläufen bestehen. Aus diesem Grund ist die angegebene Laufzeit der Durchschnitt aus 100 aufeinanderfolgenden Abläufen.

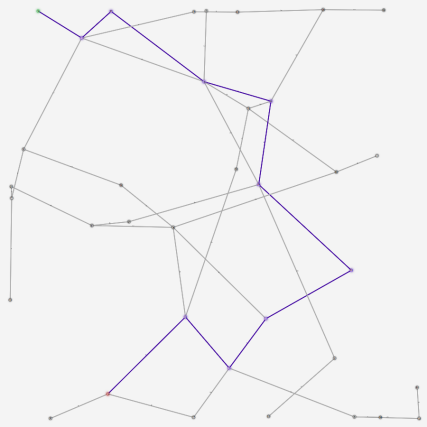

Abb. 5: Ergebnis der Tiefensuche

Im Gegensatz zu den nachfolgend vorgestellten *Algorithmen* ist die Tiefensuche in *PathFinder rekursiv*, statt *iterativ*, implementiert. Das bedeutet, dass die Suche aus ineinander verschachtelten Funktionsaufrufen besteht. Die Such-Funktion ruft sich selbst also mit neuen Parametern, dem aktuell untersuchten *Knoten* und dessen *Vorgänger*, selbst wiederholt auf und setzt so die Suche fort. Diese Verschachtlung wird entweder durch das Erreichen des *Zielknotens* oder der vollständigen Untersuchung des *Graphen* abgebrochen. Das hat mitunter negative Auswirkungen auf die Laufzeit, vereinfacht aber die Programmierung.

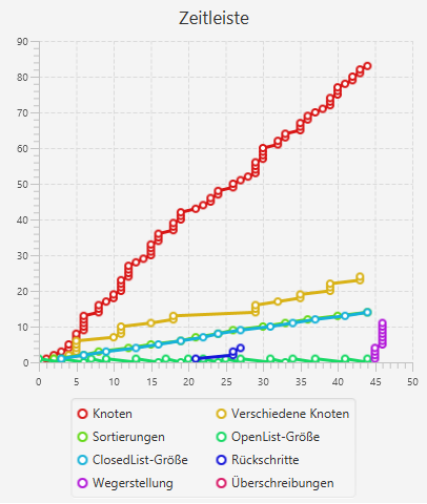

Abb. 6: Verhalten der Tiefensuche

Neben der Gesamtlaufzeit wird in *PathFinder* ebenfalls das Verhalten der *Algorithmen* aufgezeichnet. So kommt das Liniendiagramm in Abb. 6 zustande. Es werden Werte wie die Anzahl an inspizierten *Knoten*, bzw. *verschiedenen Knoten*, und die Menge an *Sortier*-Vorgängen in ein Anzahl-Zeit Diagramm eingetragen. Auf Daten wie die *Open-* und *Closed-List* Größe wird in Un-

[4][9, Kapitel 22.3, S. 457f]
[5][9, Kapitel 22.3, S. 459]

terabschnitt 6.3 eingegangen. Aus der Grafik lässt sich die lineare Funktionsweise der Tiefensuche herleiten, da sämtliche Graphen durch Mittelungsgeraden abgebildet werden können. Dies gibt weiteren Aufschluss auf die geringe *Intelligenz* der Tiefensuche, da sie keine Annäherungen an den *Zielknoten* versucht, sondern den gesamten *Graphen* gleichmäßig absucht.

Um den Codeauszug in Alg. 2[6] möglichst gering zu halten, wird hier die *iterative* Version vorgestellt. Dabei erfolgt die Suche durch eine *explizit* festgelegte Wiederholungsschleife „solange [...] wiederhole" (Alg. 2, Z. 4).

Alg. 2 *Tiefensuche*

geg.: Graph $G = (V,\ E)$
ges.: Weg P_{ab} von n_a nach n_b

```
 1: prozedur TIEFENSUCHE(n_a, n_b)
 2:     markiere alle Knoten in G als unbesucht, außer n_a
 3:     sei n_x aktiver Knoten n_a
 4:     solange nicht alle Knoten besucht sind wiederhole
 5:         wenn n_x ist n_b dann
 6:             erschließe Weg P_ab durch Vorgänger und beende Suche
 7:         ende wenn
 8:         wenn n_x keine unbesuchten Nachbarn hat dann
 9:             sei n_x Vorgänger von n_x
10:         sonst
11:             sortiere unbesuchte Nachbarn von n_x
12:             sei n_next unbesuchter Nachbar mit geringstem Kantengewicht
13:             setze n_x als Vorgänger von n_next
14:             sei n_x n_next
15:         ende wenn
16:     ende solange
17: ende prozedur
```

Durch den „brute-force" -Charakter (engl. für „rohe Gewalt") der Tiefensuche ist ihr Einsatz in Szenarien von großen Graphen und benötigter hoher Rechengeschwindigkeit nicht zu empfehlen, denn das simple Ausprobieren von beinahe allen möglichen Wegen ist in keiner Weise als effizient, geschweige denn „intelligent" zu bezeichnen. Tatsächlich wird der *Algorithmus* hauptsächlich für einen komplett anderen Zweck als die Wegfindung zwischen zwei *Knoten* verwendet. Vielmehr kommt er beim *Traversieren* des gesamten *Graphen* zum Einsatz, um beispielsweise einen gewissen *Knoten* zu suchen oder den *Graphen* in einen *depth-first Baum*, also eine

[6] vgl. Anhang: DFS.java

alternative Variante der Datenrepräsentation, umzuwandeln. In der Datenverarbeitung entspricht ein *Baum* bildlich einem umgedrehten natürlichen Baum mit einer Wurzel und sich immer weiter aufteilenden, einzelnen „Ästen" und stellt somit eine Daten-*Hierarchie* dar.

6.2 Erfunden in 20 Minuten: Der Dijkstra-Algorithmus

„*Eines Morgens war ich mit meiner Freundin in Amsterdam shoppen, und müde setzten wir uns in ein Terassencafé, tranken eine Tasse Kaffee [...] und dann entwarf ich den Algorithmus. Wie gesagt, es war eine 20-Minuten Erfindung.*"[7] (Edsger W. Dijkstra).

Im Zusammenhang mit seiner Präsentation für die Eröffnung des ARMAC[8] 1956, entwarf der niederländische Informatiker Edsger Wybe Dijkstra einen *Algorithmus* zur Ermittlung des kürzesten Wegs von Rotterdam nach Groningen – in einem Café, innerhalb von 20 Minuten. Der *Algorithmus* (Alg. 3)[9] funktioniert wie folgt:

Alg. 3 *Dijkstra-Algorithmus*

geg.: Graph $G = (V,\ E)$
ges.: Weg P_{ab} von n_a nach n_b

```
 1: prozedur DIJKSTRA(n_a, n_b)
 2:     setze die Distanz jedes Knotens auf ∞
 3:     sei Q Liste aller Knoten
 4:     setze Distanz von n_a auf 0
 5:     solange Q nicht leer ist wiederhole
 6:         sei u Knoten mit geringster Distanz aus Q
 7:         wenn u ist n_b dann
 8:             erschließe Weg P_ab durch Vorgänger und beende Suche
 9:         ende wenn
10:         entferne u aus Q
11:         für jeden Nachbarn l von u wiederhole
12:             sei a Distanz_u + Kantengewicht_ul
13:             wenn a < Distanz_l dann
14:                 setze Distanz von l auf a
15:                 setze u als Vorgänger von l
16:             ende wenn
17:         ende für
18:     ende solange
19: ende prozedur
```

[7] [10, (engl.), S. 42f]
[8] Automatische Rechenmaschine Mathematisches Zentrum, Amsterdam
[9] vgl. Anhang: Dijkstra.java

Zunächst wird die *Distanz*, also die Entfernung vom *Startknoten* n_a aus, auf ∞ gesetzt. Nicht dass die Entfernung tatsächlich als unendlich betrachtet wird, sondern dieser „Wert" gilt auch als Zeichen, dass die entsprechenden *Knoten* noch unbesucht sind.

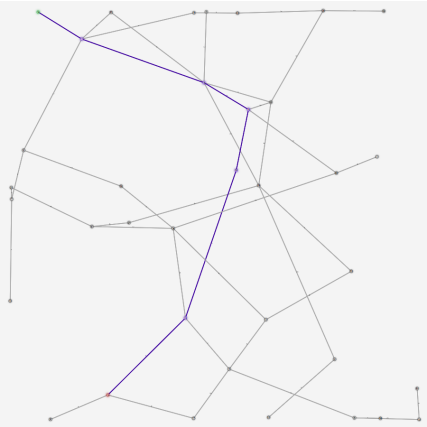

Abb. 7: Ergebnis von Dijkstras *Algorithmus*

Schnellere, aber dadurch komplexere, Umsetzungen des *Algorithmus* verwenden statt einer einfachen Liste Q eine *Priority-Queue* (für engl. „Prioritäts-Warteschlange"), die auch interne Sortieroptimierungen zur Steigerung der Gesamtlaufzeit verwendet. Eine Liste lässt sich als normale Auflistung der einzelnen Elemente ansehen, die keinerlei Sonderberechnungen vornimmt, wodurch die *Zeitkomplexität* $O(V^2)$ beträgt.

Diese Auflistung wird nun Stück für Stück, in sortierter Reihenfolge, abgearbeitet. Für die *Nachbarn* des aktuell *günstigsten Knotens* wird die *Distanz* vom *Startknoten* n_a aus berechnet, und gegebenenfalls ausgebessert, falls ein kürzerer Weg gefunden werden sollte. Sobald der *Zielknoten* zur Behandlung ansteht, wird die Suche beendet, da ausreichend Schritte erfolgt sind. Dijkstras *Algorithmus* unternimmt also, im Gegensatz zur zuvor behandelten Tiefensuche, intelligente Schritte zur Eingrenzung des *Zielknotens*.

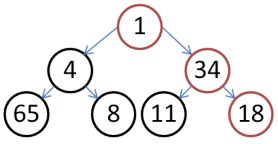

Abb. 8: Problem von *Greedy Algorithmen*

Darum zählt der Dijkstra-*Algorithmus* zur Kategorie der *Greedy-Algorithmen*, die sich durch die Eigenschaft auszeichnen, von ihrem aktuellen Standpunkt aus immer das nächste *lokale Maximum* zu suchen. Abb. 8 veranschaulicht die Problematik von dieser Art *Algorithmus*, denn es wird nicht das globale Maximum von $1 \rightarrow 4 \rightarrow 65 = 70$, sondern nur das jeweils lokale Maximum, mit einem Ergebnis von 53, erzielt.

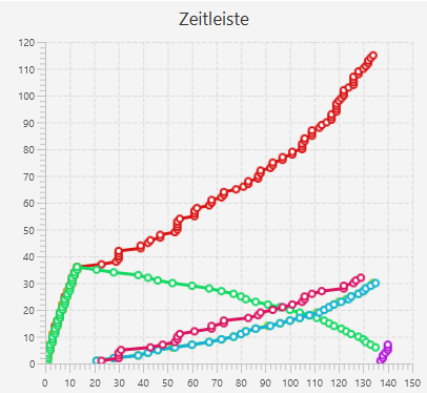

Abb. 9: Verhalten des Dijkstra-*Algorithmus*

Im direkten Vergleich von Abb. 5 und Abb. 7 wird ersichtlich, dass Dijkstras *Algorithmus* zu einem *besseren* Ergebnis gekommen ist. In der *Durchschnitts*-Messung ergeben $t_{dijkstra} = 223,76864\mu s$ deutlich

mehr Zeitaufwand als bei der Tiefensuche. Denn es werden mehr *Knoten*-Abfragen $n_{dijkstra} = 115$ durchgeführt und durch die *Warteschleifenfunktionsweise* von Q werden die bisherigen Suchergebnisse $sort_{dijkstra} = 30$ mal umsortiert (vgl. Alg. 3, Z. 6). Durch die 32 Überschreibungen, die im Fallbeispiel stattfinden, wird ein sehr viel besseres Ergebnis erreicht. Denn so kommt der Dijkstra-Algorithmus in $e_{dijkstra} = 6$ Schritten mit einem *Gesamtgewicht* von $w_{dijkstra} = 78$ zum *Zielknoten*.
Anders als die Tiefensuche findet der *Dijkstra*-Algorithmus tatsächlich Verwendung in der Wegfindung, auch bei Navigationsgeräten. Denn schließlich wurde er dazu entworfen, den kürzesten Weg zwischen zwei Städten zu finden. Allerdings werden heute zahlreich optimierte Versionen verwendet, darunter der A*-*Algorithmus*, der als Nächstes und Letztes untersucht wird.

6.3 Der Alleskönner: Der A*-Algorithmus

Der größte Nachteil des zuvor behandelten Dijkstra-*Algorithmus* ist das Besuchen von zu vielen Knoten, die von Anfang an aus der Suche ausgeschlossen werden könnten. Der A* (gesprochen „A star" oder „A Stern") -*Algorithmus* basiert auf Dijkstras Konzept, doch er erzielt weitaus bessere Ergebnisse durch die Verwendung von *Heuristiken*.
Erstmals veröffentlicht wurde der A*-*Algorithmus* 1968 von Peter Hart, Nils Nilsson und Bertram Raphael, am Stanford Research Institute[10] [11].
Kern des von ihnen vorgestellten *Algorithmus* ist die *Kosten*-Funktion

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

die sich aus einer Funktion der *tatsächlichen Kosten* $g(n)$ und den *heuristischen Kosten* $h(n)$ zusammensetzt.
Heuristik (von altgr. εὑρίσκειν *heuriskein* „entdecken") wird die Methode der Problemlösung trotz unzureichenden Wissens genannt. Genauigkeit und Vollständigkeit der Lösung werden gegen Geschwindigkeit eingetauscht. Denn *heuristische* Ansätze verwenden Schätzungen und Behauptungen, die die „Zukunft", also einen zukünftigen Stand der Simulation oder Problemsituation betreffen, und handelt nach diesen. Peter M. Todd und Gerd Gigerenzer verfassten 1999 ein *Paper* zu *Heuristik* und

[10] Heute bekannt als SRI International

rationalem Denken, „Simple heuristics that make us smart"[12], worin beschrieben wird, wie *heuristische Schätzungen* beim Treffen von Entscheidungen im Gehirn eine Rolle spielen. Im A*-*Algorithmus* wird die *heuristische* Komponente $h(n)$ zur Eingrenzung der Suche zum *Zielknoten* verwendet.

Wie im Kontext von Straßennetzen üblich, wird $h(n)$ in *PathFinder* durch die „Luftlinie" zwischen zwei *Knoten* repräsentiert, also die *pythagoräische Distanz* $d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$ der *Knoten* voneinander.

Alg. 4 *A*-Algorithmus*

geg.: Graph $G = (V,\ E)$
ges.: Weg P_{ab} von n_a nach n_b

```
 1: prozedur AStar(n_a, n_b)
 2:     sei O Open-List
 3:     sei C Closed-List
 4:     setze g von n_a auf 0 und errechne h von n_a
 5:     füge n_a zu O hinzu
 6:     solange O nicht leer ist wiederhole
 7:         sei u Knoten mit geringstem f_u = g_u + h_u aus O
 8:         verschiebe u von O nach C
 9:         wenn u ist n_b dann
10:             erschließe Weg P_ab durch Vorgänger in C und beende Suche
11:         ende wenn
12:         für jeden Nachbarn l von u wiederhole
13:             wenn l in O ist dann
14:                 aktualisiere f_l und Vorgänger, falls günstiger
15:             sonst wenn l nicht in C ist dann
16:                 setze g_l = g_u + Kantengewicht_ul und errechne h_l
17:                 setze u als Vorgänger von l
18:                 füge l zu O hinzu
19:             ende wenn
20:         ende für
21:     ende solange
22: ende prozedur
```

Hier kommen zum ersten Mal die Begriffe *Open-List* und *Closed-List* (vgl. Alg. 4[11], Z. 2f) vor, die auch in den Verhaltensdiagrammen (vgl. Abb. 6, 9, 11) referenziert werden. In der *Open-List* werden alle *Knoten* aufgeführt, die als mögliche nächste Schritte in der Wegsuche gesehen werden. Der jeweils günstigste *Knoten* aus dieser Liste wird für den nächsten Schritt ausgewählt (Z. 7) und in die *Closed-List*, also die engere Auswahl aus *Knoten*, verschoben (Z. 8). In dieser Sammlung fin-

[11] vgl. Anhang: AStar.java

den sich nun alle *Knoten*, die bei der Wegkonstruktion durch *Backtracking* bei Beendung des Suchvorgangs betrachtet werden. Dies ist auch der einzige große Unterschied zum Dijkstra-*Algorithmus*. Der restliche Verlauf des *Algorithmus* ist fast identisch mit Abbruch bei Erreichen des *Zielknotens*, *Überschreiben* bei Entdeckung von Abkürzungen und *nachbar*-basierter *Graph-Erkundung*. Neben der bereits beschriebenen *Heuristik*-Komponente $h(n)$ verwendet A* auch eine berechnete Komponente $g(n)$, die aus der Summe der *Kantengewichte* besteht (Z. 16).

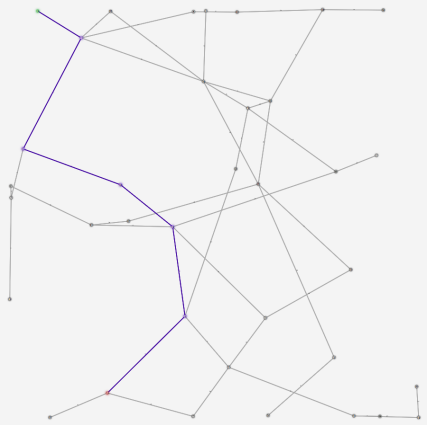

Abb. 10: Ergebnis des A*-*Algorithmus*

Bei Beobachtung des *Algorithmus* stellt man fest, dass zu Beginn keine klare Suchrichtung erkennbar ist, doch schon bald sehr zielstrebig vorgegangen und auf schnellstem Wege ein Ergebnis herbeigeführt wird (vgl. Abb. 11). In nur $t_{astar} = 109,71132\mu s$ findet A* einen Weg mit $w_{astar} = 94$ in $e_{astar} = 6$ Schritten. Er benötigt somit weniger als halb so lange wie der Dijkstra-*Algorithmus*, erzielt aber ein nur geringfügig schlechteres Ergebnis. Am Bedeutendsten ist jedoch die geringe Anzahl an besuchten *Knoten*. Während Dijkstra 115 (nicht verschiedene) *Knoten* besucht, benötigt A* nur 29, um zum Resultat zu gelangen.

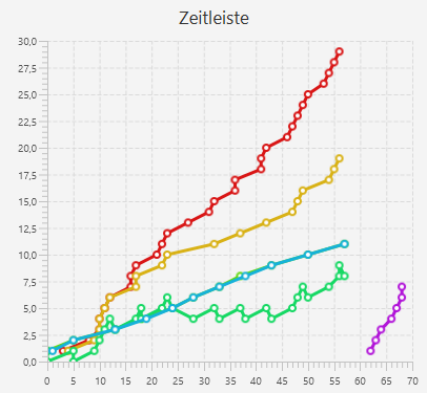

Abb. 11: Verhalten des A*-*Algorithmus*

Aus diesem Grund ist der A*-*Algorithmus* auch der meistverbreiteste der *Pathfinding-Algorithmen*, besonders im Kontext von Computerspielen. Schnelle und akzeptable Lösungen und ein flexibles Konzept machen A* so attraktiv. Zudem gibt es einige optimierte Fassungen, darunter die Verwendung eines *Fast-Stacks* (für engl. „schneller Stapel"), in dem alle Knoten der *Open-List* platziert werden, die den gleichen, und vorallem höchsten Gesamtwert $f(n)$ besitzen, was den Sortierungsprozess der *Open-List* um einige *Knoten* reduziert und so für noch mehr Geschwindigkeit sorgt. Allgemein lässt sich über die *Zeitkomplexität* des A**Algorithmus* keine Aussage treffen, da die-

se von der *Heuristik*-Funktion $h(n)$ abhängt. Doch der „Luftlinien" -Ansatz gilt als der Effektivste. Dafür ist er nur in gewissen Situationen anwendbar. Nämlich wenn die Zeit und Rechenleistung gegeben ist, um den gesamten *Graphen* visualisieren zu lassen.

7 Vergleichsstatistik und Fazit

Die Zahlen sprechen für sich. Spitzenreiter oder Zweitbester in jeder Testkategorie zu sein macht den A*-*Algorithmus* zum klaren Favoriten (Tabelle 1).

Tabelle 1: Die drei Algorithmen im direkten Vergleich

	Laufzeit /μs	Kosten	Kanten	versch.	Knoten	Sortierungen
DFS	**72,61709**	116	10	24	83	14
Dijkstra	223,76864	**78**	**6**	36	115	30
A*	109,71132	94	**6**	**19**	**29**	**11**

Es ist somit nicht verwunderlich, dass der A*-*Algorithmus* am weitesten verbreitet ist. Besonders in der Spiele-Industrie. Doch auch die Tiefensuche und Dijkstras *Algorithmus* finden auch ihre Verwendung. Nämlich in weiter entwickelten *Algorithmen*, wie A* oder den weiteren Vertreter der *-*Algorithmus* Familie, also B*, D*, IDA*. Diese sind aber für besondere Szenarien optimiert, zum Beispiel Wegfindung in einem nicht vollständig bekannten Umfeld, oder wenn sich die Umstände während der Suche verändern sollten.

8 Ausblick auf das weite Feld der Wegfindung

In dieser Arbeit wurden nur die geläufigsten und auch einfachsten *Algorithmen* vorgestellt. Durch wachsende Datenmengen und das Bedürfnis von mehr Rechengeschwindigkeit, wird in diesem Feld der Informatik auch heute noch viel geforscht. *Algorithmen*, die von echten Navigationsgeräten verwendet werden, haben beispielsweise heutzutage auch eine „Alternativrouten" -Funktion oder ähnliches, welche zusätzlichen Rechenaufwand beansprucht. Auch in anderen Branchen, wie der Spieleprogrammierung wird stets nach neuen Ansätzen gesucht. So wurde beispielsweise auf der

GDC[12] 2015 die *JPS+ mit Gouldbounding*[13] vorgestellt, eine Zusammensetzung aus einem optimierten Raster-basierten Such-*Algorithmus* und einer visuellen Methode der Sucheingrenzung.
Man ist also lange noch nicht am Ziel. Die Suche geht weiter...
Nach dem optimalen Weg.

[12]Game Developers Conference in San Francisco

Literatur

[1] Stern.de: Kuriose Navi-Unfälle
http://www.stern.de/digital/technik/navi-missgeschicke–in-100-metern-fahren-sie——in-den-fluss-3087618.html
zul. abgerufen am 27.10.15

[2] Java Zufalls-Funktion
http://docs.oracle.com/javase/7/docs/api/java/util/Random.html
zul. abgerufen am 18.10.15

[3] JavaFX-Homepage
http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm
zul. abgerufen am 14.10.15

[4] Stephen G. Kobourov
„Spring embedders and force directed graph drawing algorithms.", 2012
http://arxiv.org/pdf/1201.3011v1.pdf
zul. abgerufen am 24.10.15

[5] Stephen G. Kobourov
„Force-Directed Drawing Algorithms", 2013
https://cs.brown.edu/ rt/gdhandbook/chapters/force-directed.pdf
zul. abgerufen am 30.10.15

[6] Thomas M. J. Fruchterman und Edward M. Reingold
„Graph drawing by force-directed placement", 1991
ftp://ftp.mathe2.uni-bayreuth.de/axel/papers/reingold:graph_drawing_by_force_directed_placement.pdf
zul. abgerufen am 24.10.15

[7] Donald E. Knuth
„Big Omicron and big Omega and big Theta", S. 18–24, 1976
http://www.phil.uu.nl/datastructuren/09-10/knuth_big_omicron.pdf
zul. abgerufen am 30.10.15

[8] Robert Geisberger
„Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks", 2008
http://algo2.iti.kit.edu/schultes/hwy/contract.pdf
zul. abgerufen am 30.10.15

[9] Thomas H. Cormen
„Introduction to Algorithms", 2. Ausgabe, 2001
http://www.mif.vu.lt/∼valdas/ALGORITMAI/LITERATURA/Cormen/Cormen.pdf
zul. abgerufen am 30.10.15

[10] Thomas J. Misa und Philip L. Frana
„An interview with Edsger W. Dijkstra"
Commun. ACM, 53(8):41–47, 2010
http://dl.acm.org/citation.cfm?doid=1787234.1787249
zul. abgerufen am 01.11.15

[11] P.E. Hart, N.J. Nilsson, und B. Raphael
„A formal basis for the heuristic determination of minimum cost paths"
Systems Science and Cybernetics, IEEE Transactions on, 4(2):100–107, 1968
http://ai.stanford.edu/∼nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf
zul. abgerufen am 03.11.15

[12] Peter M. Todd und Gerd Gigerenzer
„Précis of 'Simple heuristics that make us smart'", S. 727–780, 2000
http://psy2.ucsd.edu/∼mckenzie/ToddGigerenzer2000BBS.pdf
zul. abgerufen am 03.11.15

[13] Steve Rabin
„JPS+: Over 100x Faster than A*", 2015
http://gdcvault.com/play/1022094/JPS-Over-100x-Faster-than
zul. abgerufen am 05.11.15

```java
package de.dakror.wseminar.graph.generate;

import de.dakror.wseminar.Const;
import de.dakror.wseminar.WSeminar;
import de.dakror.wseminar.graph.DefaultGraph;
import de.dakror.wseminar.graph.Edge;
import de.dakror.wseminar.graph.Graph;
import de.dakror.wseminar.graph.WeightedEdge;

/**
 * @author Dakror
 */
public class GraphGenerator<V> {
  /**
   * @param params the bundle of generation parameters
   * @return the generated graph
   */
  @SuppressWarnings("unchecked")
  public Graph<V> generateGraph(Params<String> params) {
    long seed = params.get("seed");
    WSeminar.setSeed(seed);

    Graph<V> graph = new DefaultGraph<>();

    int nodeAmount = params.orElse("nodes", Const.nodeAmount);

    int nodes = (WSeminar.r.nextInt(nodeAmount / 2) + nodeAmount / 2) * (int)
    params.get("size");

    for (int i = 0; i < nodes; i++) {
      try {
        graph.addVertex((V) (Integer) i);
      } catch (Exception e) {
        throw new IllegalStateException("Generics not matching graph type!", e);
      }
    }

    System.out.println("Added " + graph.getVertices().size() + " nodes to the graph.");

    int edgesPlaced = 0;
```

```
    int edge_type = params.get("edge_type");

    for (int i = 0; i < nodes; i++) {
      int edges = Math.max(WSeminar.r.nextInt(Math.min(graph.getVertices().size() / 2 - 1,
      params.orElse("edges", Const.edgeAmount))), 1);

      for (int j = 0; j < edges; j++) {
        int index = i;
        do {
          index = WSeminar.r.nextInt(nodes);
        } while (index == i || graph.areConnected(graph.getVertices().get(i),
        graph.getVertices().get(index)));

        Edge<V> edge = new WeightedEdge<V>(graph.getVertices().get(i),
        graph.getVertices().get(index), WSeminar.r.nextInt(Const.edgesMaxCost));

        if (edge_type == 1 || (edge_type == 2 && WSeminar.r.nextFloat() >
        WSeminar.r.nextFloat())) edge.setDirected(true);

        graph.addEdge(edge);
      }

      edgesPlaced += edges;
    }

    System.out.println("Made " + edgesPlaced + " connections.");

    return graph;
  }
}
```

```java
package de.dakror.wseminar.graph.algorithm;

import static de.dakror.wseminar.util.Benchmark.Type.*;

import java.util.HashMap;
import java.util.List;
import java.util.stream.Collectors;

import de.dakror.wseminar.Const.State;
import de.dakror.wseminar.graph.Edge;
import de.dakror.wseminar.graph.Graph;
import de.dakror.wseminar.graph.Path;
import de.dakror.wseminar.graph.Vertex;
import de.dakror.wseminar.graph.VertexData.PathCommons;
import de.dakror.wseminar.graph.WeightedEdge;
import de.dakror.wseminar.graph.algorithm.base.PathFinder;
import de.dakror.wseminar.util.Visualizer;

/**
 * @author Maximilian Stark | Dakror
 */
public class DFS<V> extends PathFinder<V> {
  HashMap<Vertex<V>, PathCommons<V>> meta;

  public DFS(Graph<Vertex<V>> graph, boolean animate) {
    super(graph, animate);
    meta = new HashMap<>();
  }

  @Override
  public Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to) {
    Visualizer.resetAll(graph, true, false);
    BM.time();

    if (!takeStep(null, from, to)) return null;

    Path<Vertex<V>> p = new Path<Vertex<V>>();
    p.setUserData("DFS" + (animate ? " anim" : "") + " " + from.data() + "->" + to.data());
    Vertex<V> v = to;
```

```
    while (meta.get(v).parent != null) {
      p.add(0, v);
      v = meta.get(v).parent;

      BM.add(PATH_CREATION);
    }

    p.add(0, from);
    BM.add(PATH_CREATION);
    p.calculateCost(graph);

    p.setBenchmark(BM);

    BM.time();
    cleanup();
    Visualizer.resetAll(graph, false, false);
    return p;
  }

  @Override
  protected boolean takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to) {
    PathCommons<V> pc = new PathCommons<>();
    pc.parent = parent;
    meta.put(node, pc);
    Visualizer.setVertexState(node, State.OPENLIST, false);
    BM.add(OPEN_LIST_SIZE);

    if (node.equals(to)) return true;

    List<Edge<Vertex<V>>> edges = graph.getEdgesFrom(node).stream().filter(e -> {
      Vertex<V> v = e.getOtherEnd(node);

      BM.add(v);

      boolean free = meta.get(v) == null;
      Visualizer.setEdgeActive(e, free, false);
      return free;
    }).sorted((a, b) -> Float.compare(a instanceof WeightedEdge ? ((WeightedEdge<Vertex<V>>)
    a).getWeight() : 0,
                                      b instanceof WeightedEdge ? ((WeightedEdge<Vertex<V>>)
                                      b).getWeight() : 0)).collect(Collectors.toList());

    BM.add(SORTS);

    // is target reachable?
    for (Edge<Vertex<V>> e : edges) {
      Vertex<V> oe = e.getOtherEnd(node);
      BM.add(oe);
      if (oe.equals(to)) {
        BM.sub(OPEN_LIST_SIZE);
        BM.add(CLOSED_LIST_SIZE);
        Visualizer.setVertexState(node, State.CLOSEDLIST);
        Visualizer.setEdgePath(e, true);
        return takeStep(node, oe, to);
      }
    }
```

```
    // take next step
    for (Edge<Vertex<V>> e : edges) {
      Vertex<V> oe = e.getOtherEnd(node);
      BM.add(oe);
      Visualizer.tick();

      BM.sub(OPEN_LIST_SIZE);
      BM.add(CLOSED_LIST_SIZE);
      Visualizer.setVertexState(node, State.CLOSEDLIST);
      Visualizer.setEdgePath(e, true);

      for (Edge<Vertex<V>> e1 : edges) {
        if (e1 == e) continue;
        Visualizer.setEdgeActive(e1, false, false);
      }

      if (takeStep(node, oe, to)) return true;

      for (Edge<Vertex<V>> e1 : edges) {
        if (e1 == e) continue;
        Visualizer.setEdgeActive(e1, true, false);
      }

      Visualizer.setEdgePath(e, false);
    }

    BM.add(BACK_TRACKS);
    Visualizer.setVertexState(node, Visualizer.isEnabled() ? State.BACKTRACK : null, false);

    return false;
  }
}
```

```java
package de.dakror.wseminar.graph.algorithm;

import static de.dakror.wseminar.util.Benchmark.Type.*;

import java.util.ArrayList;
import java.util.Collections;

import de.dakror.wseminar.Const.State;
import de.dakror.wseminar.graph.Edge;
import de.dakror.wseminar.graph.Graph;
import de.dakror.wseminar.graph.Path;
import de.dakror.wseminar.graph.Vertex;
import de.dakror.wseminar.graph.VertexData.InfPath;
import de.dakror.wseminar.graph.WeightedEdge;
import de.dakror.wseminar.graph.algorithm.base.PathFinder;
import de.dakror.wseminar.util.Benchmark.Type;
import de.dakror.wseminar.util.Visualizer;

/**
 * @author Maximilian Stark | Dakror
 */
public class Dijkstra<V> extends PathFinder<V> {
  ArrayList<Vertex<V>> list;

  public Dijkstra(Graph<Vertex<V>> graph, boolean animate) {
    super(graph, animate);

    list = new ArrayList<>();
    metaClasses = new Class<?>[] { InfPath.class };
  }

  @SuppressWarnings("unchecked")
  @Override
  public Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to) {
    Visualizer.resetAll(graph, true, false);
    BM.time();

    for (Vertex<V> v : graph.getVertices()) {
      v.add(new InfPath<>());
      if (v.equals(from)) v.get(InfPath.class).d = 0;
      list.add(v);
```

```
    BM.add(v);
    BM.add(OPEN_LIST_SIZE);
    Visualizer.setVertexState(v, State.OPENLIST, true);
  };

  boolean found = false;

  Vertex<V> v = null;

  while (!list.isEmpty()) {
    Collections.sort(list, (a, b) -> {
      int c = Float.compare(a.get(InfPath.class).d, b.get(InfPath.class).d);
      if (c == 0) return a.data().toString().compareTo(b.data().toString());
      return c;
    });
    BM.add(SORTS);
    v = list.remove(0);
    BM.sub(OPEN_LIST_SIZE);
    BM.add(CLOSED_LIST_SIZE);
    Visualizer.setVertexState(v, State.CLOSEDLIST, true);

    if (v.equals(to)) {
      found = true;
      break;
    }

    takeStep(null, v, to);
  }

  if (!found) return null;

  Path<Vertex<V>> p = new Path<Vertex<V>>();
  p.setUserData("Dijkstra" + (animate ? " anim" : "") + " " + from.data() + "->" +
  to.data());

  for (Edge<Vertex<V>> e : graph.getEdges()) {
    Visualizer.setEdgePath(e, false, false, false);
  }

  while (v != null) {
    p.add(0, v);

    if (v.get(InfPath.class).parent != null) {
      Visualizer.setEdgePath(graph.getEdge(v, v.get(InfPath.class).parent), true, true);
    }
    Visualizer.setVertexState(v, State.BACKTRACK, false);
    v = v.get(InfPath.class).parent;

    BM.add(PATH_CREATION);
  }
  p.calculateCost(graph);

  p.setBenchmark(BM);


  BM.time();
  cleanup();
```

```
    Visualizer.resetAll(graph, true, false);

    return p;
  }

  @SuppressWarnings("unchecked")
  @Override
  protected boolean takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to) {
    for (Edge<Vertex<V>> e : graph.getEdges(node)) {
      Vertex<V> oe = e.getOtherEnd(node);
      BM.add(oe);
      if (list.contains(oe)) {
        float alt = node.get(InfPath.class).d + (e instanceof WeightedEdge ?
        ((WeightedEdge<Vertex<V>>) e).getWeight() : 1);
        if (alt < oe.get(InfPath.class).d) {
          oe.get(InfPath.class).d = alt;
          oe.get(InfPath.class).parent = node;
          Visualizer.setEdgePath(e, true, true);
          BM.add(Type.OVERRIDES);
        }
      }
    }

    return false;
  }
}
```

```java
package de.dakror.wseminar.graph.algorithm;

import static de.dakror.wseminar.util.Benchmark.Type.*;

import java.util.ArrayList;
import java.util.TreeSet;

import de.dakror.wseminar.Const.State;
import de.dakror.wseminar.graph.Edge;
import de.dakror.wseminar.graph.Graph;
import de.dakror.wseminar.graph.Path;
import de.dakror.wseminar.graph.Vertex;
import de.dakror.wseminar.graph.VertexData.Heuristics;
import de.dakror.wseminar.graph.VertexData.Position;
import de.dakror.wseminar.graph.WeightedEdge;
import de.dakror.wseminar.graph.algorithm.base.PathFinder;
import de.dakror.wseminar.util.Visualizer;

/**
 * @author Maximilian Stark | Dakror
 */
public class AStar<V> extends PathFinder<V> {
  TreeSet<Vertex<V>> openList;
  ArrayList<Vertex<V>> closedList;

  public AStar(Graph<Vertex<V>> graph, boolean animate) {
    super(graph, animate);
    openList = new TreeSet<>((a, b) -> Float.compare(a.get(Heuristics.class).F(),
    b.get(Heuristics.class).F()));
    closedList = new ArrayList<>();

    metaClasses = new Class<?>[] { Heuristics.class };
  }

  @SuppressWarnings("unchecked")
  @Override
  public Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to) {
    Visualizer.resetAll(graph, true, false);
    BM.time();

    Heuristics<V> h = new Heuristics<>();
```

```
    h.H = distance(from, to);
    from.add(h);

    openList.add(from);
    BM.add(OPEN_LIST_SIZE);

    Vertex<V> last = null;
    while (true) {
      if (openList.size() == 0) return null;

      Vertex<V> v = openList.pollFirst();
      BM.add(SORTS);
      BM.sub(OPEN_LIST_SIZE);

      closedList.add(v);
      BM.add(CLOSED_LIST_SIZE);

      if (v.get(Heuristics.class).parent != null) Visualizer.setEdgePath(graph.getEdge(v,
      v.get(Heuristics.class).parent), true, true);
      Visualizer.setVertexState(v, State.CLOSEDLIST, true);

      if (takeStep(last, v, to)) break;

      last = v;
    }

    Path<Vertex<V>> p = new Path<Vertex<V>>();
    p.setUserData("AStar" + (animate ? " anim" : "") + " " + from.data() + "->" + to.data());
    Vertex<V> v = to;

    for (Edge<Vertex<V>> e : graph.getEdges()) {
      Visualizer.setEdgePath(e, false, true, false);
    }

    while (v != null) {
      p.add(0, v);
      if (v.get(Heuristics.class).parent != null) {
        Visualizer.setEdgePath(graph.getEdge(v, v.get(Heuristics.class).parent), true, true);
      }
      Visualizer.setVertexState(v, State.BACKTRACK, false);
      v = v.get(Heuristics.class).parent;

      BM.add(PATH_CREATION);
    }
    p.calculateCost(graph);

    p.setBenchmark(BM);

    BM.time();
    cleanup();
    Visualizer.resetAll(graph, true, false);

    return p;
  }

  @SuppressWarnings("unchecked")
  @Override
```

```
  protected boolean takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to) {
    if (node.equals(to)) {
      to.add(node.get(Heuristics.class));
      return true;
    }

    float nG = node.get(Heuristics.class).G;
    for (Edge<Vertex<V>> e : graph.getEdgesFrom(node)) {
      Vertex<V> v = e.getOtherEnd(node);
      BM.add(v);
      if (v.get(Heuristics.class) == null) {
        Heuristics<V> h = new Heuristics<>();
        h.G = nG + weight(e);
        h.H = distance(v, to);
        h.parent = node;
        v.add(h);
        openList.add(v);
        Visualizer.setEdgeActive(e, true, true);
        Visualizer.setVertexState(v, State.OPENLIST, false);
        BM.add(OPEN_LIST_SIZE);
      } else if (nG + weight(e) < v.get(Heuristics.class).G) {
        BM.add(OVERRIDES);
        v.get(Heuristics.class).G = nG + weight(e);

        Visualizer.setEdgeActive(graph.getEdge(v.get(Heuristics.class).parent, v), false,
        true);

        v.get(Heuristics.class).parent = node;

        Visualizer.setEdgeActive(e, true, false);
      }
    }

    return false;
  }

  float weight(Edge<Vertex<V>> e) {
    return e instanceof WeightedEdge ? ((WeightedEdge<Vertex<V>>) e).getWeight() : 1;
  }

  float distance(Vertex<V> v, Vertex<V> to) {
    return v.get(Position.class).pos.dst(to.get(Position.class).pos);
  }
}
```

E r k l ä r u n g

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

Ich bin damit einverstanden, dass innerhalb der Schule von Dritten in diese Seminararbeit Einsicht genommen werden kann.

....., den ….....
(Ort) *(Datum)*

.....
(Unterschrift Schüler)