



Max-Born-Gymnasium

**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/des Seminarleiters:

Inhaltsverzeichnis

1	Risikofaktor Navigationsgerät	3
2	Grundlagen und Terminologie	4
3	Aufbau und Bedienung des Programms <i>PathFinder</i>	5
4	Konstruktion eines Graphen in <i>PathFinder</i>	6
5	Visuelles Layout von Graphen	7
6	Wegfindungs-Algorithmen	9
6.1	Größte Züge von Intelligenz: Tiefensuche	10
6.2	Erfunden in 20 Minuten: Der Dijkstra-Algorithmus	13
6.3	Der Alleskönner: Der A*-Algorithmus	16
7	Vergleichsstatistik und Fazit	18
8	Schluss	19

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. Darum geht es in dieser Arbeit, nämlich die verschiedenen 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, aber genauso wenig 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* $O(n^2)$ kürzen. Hierbei stellt n die Anzahl an *Iterationen*, also Schritten dar. Diese Angabe ist wird als primäres Vergleichskriterium von Laufzeiten 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 Kosten der Beziehung 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 *Knotenpaar* nur einfach verbunden ist, reicht in der Implementierung diese Annahme aus.

Unter *Backtracking* versteht man in der Wegfindung das rückwärtige Abarbeiten 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

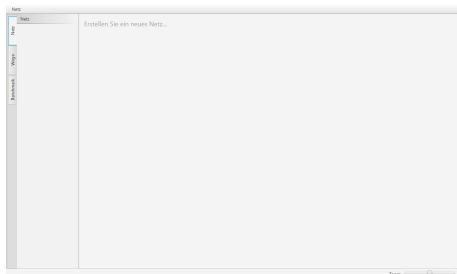


Abb. 1: Die Start- und Hauptansicht von *PathFinder*

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 Anwen-

dung 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.

¹siehe 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 durch

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

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

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  $n \sim R(n_{max}/2, n_{max}) * s_g$   $\triangleright$  Zufällige Anzahl im Intervall  $[n_{max}/2; n_{max}]$ 
4:   füge  $n$  Knoten zu  $g$  hinzu
5:   für  $i = 0 \rightarrow n$  wiederhole
6:     sei  $e \sim \max(1, R(0, \min(n/2 - 1, e_{max})))$ 
7:     für  $j = 0 \rightarrow e$  wiederhole
8:       sei  $index \sim i$ 
9:       wiederhole
10:        sei  $index \sim R(0, n)$ 
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  $t = \text{GEMISCHT}$  oder  $(t = \text{GERICHTET}$  und  $R() > R())$  dann
14:           $\triangleright R > R = \text{Zufallstest}$ 
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 vorangegangenen 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

²vgl. 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 visualisierten *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

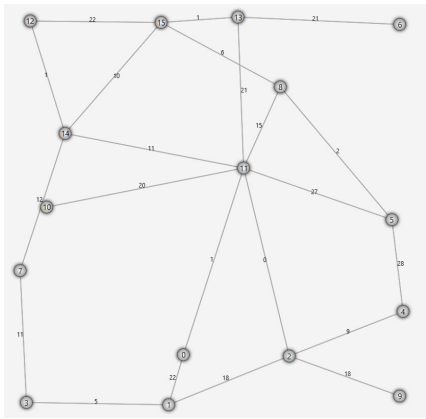


Abb. 3: F.-R. *Algorithmus*

verbundene *Knoten* nebeneinander liegen und sämtliche *Knoten* trotzdem nicht zu nahe beieinander platziert werden sollten.

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 Schritte 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. In der Anwendung wird diese Maximalgröße 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

Nun folgt der eigentliche Hauptteil der Arbeit. 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* den kürzesten, optimalen oder hindernisärmsten Weg zwischen zwei *Knoten* zu finden, je nach Aufgabenstellung; 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 Anwendung selbst alle notwendigen Wegdaten bequem und schnell abgerufen werden. Nur hat ein solcher *Algorithmus* eine *Zeitkomplexität* $O(n^2)$, die für größere *Graphen* einfach untragbar ist. 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. Die Einstellungen sind alle auf den Standard-Werten

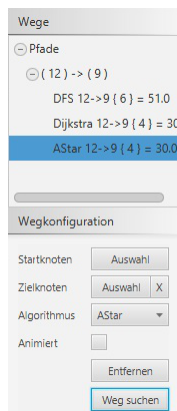


Abb. 4: Weg-Konfiguration

und der Seed ist 7646137120994539520. Es wird immer der Weg vom *Knoten* Nr. 12 zu Nr. 9 gesucht. Zur besseren Übersicht wird einmal die „Relayout“-Funktion in *PathFinder* genutzt.

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. Au-

ßerdem 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 Größte 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*, *Backtracking* angewandt⁴.

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

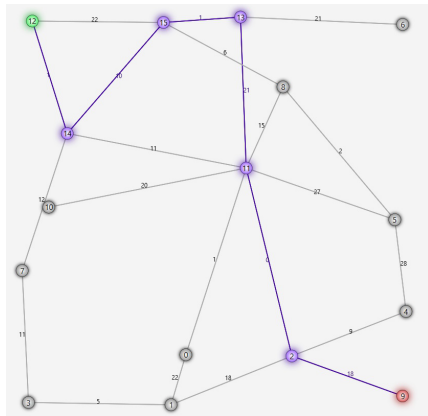


Abb. 5: Ergebnis der Tiefensuche

Tiefensuche im Fallbeispiel des *Wegs* von *Knoten* 12 zu 9. Es ergeben sich die Gesamtkosten $w_{dfs} = 51$ durch Addition aller $e_{dfs} = 6$ verwendeten *Kantengewichte*. Die gemessene Laufzeit der Wegsuche beträgt $t_{dfs} = 312,9389\mu s$. Hierbei ist zu beachten, dass aufgrund von variierender Prozessorauslastung und Speicherbelegungen starke zeitliche Schwankungen zwischen wiederholten Suchabläufen bestehen. Aus die-

sem Grund ist die angegebene Laufzeit der Durchschnitt aus 100 aufeinanderfolgenden Abläufen.

Im Gegensatz zu den nachfolgend vorgestellten *Algorithmen* ist die Tiefensuche in *PathFinder* *rekursiv*, statt *iterativ*, implementiert. Das bedeutet, dass die Suche aus

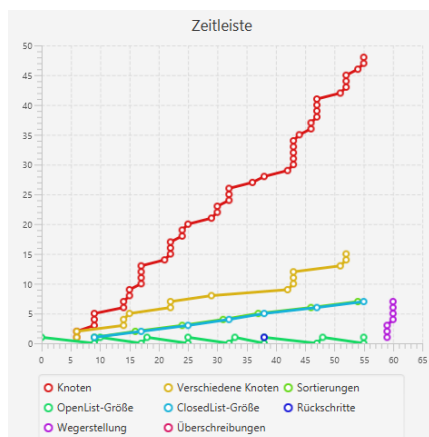


Abb. 6: Verhalten der Tiefensuche

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.

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 Num-

⁴[9, Kapitel 22.3, S. 457f]

⁵[9, Kapitel 22.3, S. 459]

mer von *Sortier*-Vorgängen in ein Anzahl-Zeit Diagramm eingetragen. Daraus 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. 2⁶ 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 alternative Variante der Datenrepräsentation, umzuwandeln. In

⁶vgl. Anhang: DFS.java

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 Terrassencafé, 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 ARMAC⁸ 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. 3⁹) 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  $\infty$ 
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  $a = \text{Distanz}_u + \text{Kantengewicht}_{ul}$ 
13:      wenn  $a < \text{Distanz}_l$  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]

⁸Automatische Rechenmaschine Mathematisches Zentrum, Amsterdam

⁹vgl. 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.

-

6.3 Der Alleskönner: Der A*-Algorithmus

Alg. 4 *A*-Algorithmus*

geg.: Graph $G = (V, E)$

ges.: Weg P_{ab} von n_a nach n_b

-

7 Vergleichsstatistik und Fazit

8 Schluss

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 and 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

Erklärung

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)