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 Einleitung **3**

2 Grundlagen und Terminologie **4**

3 Aufbau und Bedienung des Programms *PathFinder* **5**

4 Konstruktion eines Graphen in *PathFinder* **6**

5 Visuelles Layout von Graphen **8**

6 Wegfindungs-Algorithmen **9**

6.1 Gröbste Züge von Intelligenz: Tiefensuche 10

6.2 Heuristik als Mittel zum Ziel: Der Dijkstra-Algorithmus 12

6.3 Der Allstar: Der A*-Algorithmus 14

7 Vergleichsstatistik und Fazit **16**

8 Schluss **18**

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 ihnen 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 850km entfernt vom gewünschten Ziel eintraf, bei der allzu freundlichen Stimme aus der Mittelkonsole.[1] Darum ist es umso wichtiger, dass Navi's immer über aktuellste Kartendaten verfügen und auch ausgiebig auf Fehler geprüft werden.

2 Grundlagen und Terminologie

Das Ziel dieser Arbeit ist die Darstellung und der Vergleich verschiedener *Wegfindungs-Algorithmen* in der Anwendung an verschiedenen generierten *Graphen*.
In diesem Abschnitt werden 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 Graphen-Theorie dient als Basis dieser Ausführungen. 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 *Knoten*-Paar nur einfach verbunden ist, reicht in der Implementierung diese Annahme aus.
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 *Pseudozufall*-Generator verwendet [2]. Dieser generiert kaum oder nur schwer vorhersagbare Abfolgen von Zahlen und genügt für unsere Zwecke. 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.

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 mehreren „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 Features 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.

4 Konstruktion eines Graphen in *PathFinder*

Im nächsten Schritt werden wir nun einen Graphen erzeugen lassen und die Funktionsweise des Generators betrachten. Durch Klicken auf „Erstellen Sie ein neues Netz...” wird ein Dialog-Fenster geöffnet (Abb. 2), welches verschiedene Generierungs-*Parameter* zur Konfiguration anbietet.

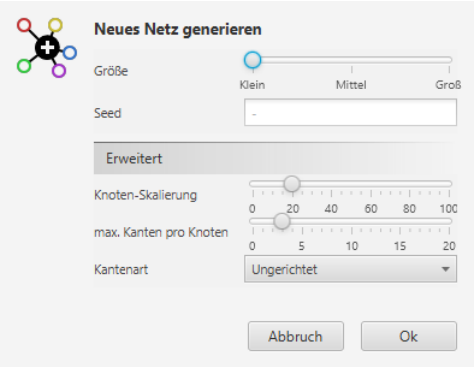

Abb. 2: Dialog zur *Netz*-Generierung

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 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

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

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

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.

Alg. 1 *Graph-Generator*

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

```
 1: prozedur GENERIEREGRAPH(seed, s, n_max, e_max, t)
 2:     setze Seed von R zu seed
 3:     sei n R(n_max/2, n_max) * s     ▷ 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
```

Sobald eine Kombination gefunden wurde, wird die entsprechende *Kante* mit einem ebenfalls zufallsgenerierten *Gewicht* erstellt. Dann wird auf Basis des *Kanten-Typs* die Gerichtetheit bestimmt und schließlich wird die *Kante* im *Graphen* platziert (Alg. 1)[1].

[1] vgl. Anhang: GraphGenerator.java

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 „kraft-basiert")[4]. Diese simulieren ein einem großen Molekül ähnelndes Konstrukt, in dem verschiedene *Kräfte*, die von *Knoten* und *Kanten* ausgehen, aufeinander wirken. Das 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*.

Bei dem in *PathFinder* verwendeten Algorithmus handelt es sich um den *Fruchtermann-Reingold* Algorithmus, welcher 1991 von Thomas M. J. Fruchterman und Edward M. Reingold an der University of Illinois veröffentlicht wurde.[5]

6 Wegfindungs-Algorithmen

6.1 Gröbste Züge von Intelligenz: Tiefensuche

-

6.2 Heuristik als Mittel zum Ziel: Der Dijkstra-Algorithmus

-

6.3 Der Allstar: Der A*-Algorithmus

-

7 Vergleichsstatistik und Fazit

-

8 Schluss

Literatur

[1] vgl. 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] Thomas M. J. Fruchterman and 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

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)