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

„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 850km 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. 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 Vergleiche der selbigen angestellt, um festzustellen, für welche Zwecke welche Methode am zielführendsten ist. Als Werkzeug zur genaueren Untersuchung und für Vergleichsstatistiken habe ich zusätzlich ein Programm namens *PathFinder* geschrieben, in dem die *Algorithmen* adaptiert sind. Diese Ausführungen 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

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 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 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.

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 *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.[1]

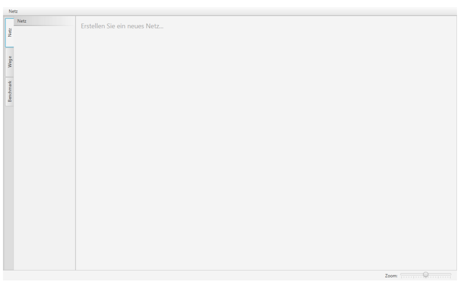

Abb. 1: Die Start- und Hauptansicht von *PathFinder*

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 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 *Ge-*

[1] siehe Anhang: CD-ROM

nerieren (siehe Abschnitt 4) eines neuen *Netzes* unter Benutzung von „N” erfolgen.

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 Genererierungs-*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

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

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

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

```
prozedur generiereGraph(seed, s_g, n_max, e_max, t)
    setze Seed von R zu seed
    sei n R(n_max/2, n_max) * s_g    ▷ Zufällige Anzahl im Interval [n_max/2; n_max[
    füge n Knoten zu g hinzu
    für i = 0 → n wiederhole
        sei e max(1, R(0, min(n/2 − 1, e_max)))
        für j = 0 → e wiederhole
            sei index i
            wiederhole
                sei index R(0, n)
            solange index gleich i oder Knoten_i mit Knoten_index verbunden
            sei e Kante von Knoten_i zu Knoten_index, Gewicht w = R(0, W_max)
            wenn t = Gemischt oder (t = Gerichtet und R() > R()) dann
                                                        ▷ R > R = Zufallstest
                setze e gerichtet
            ende wenn
            füge e zu g hinzu
        ende für
    ende für
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*.

[2] vgl. Anhang: GraphGenerator.java

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 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.

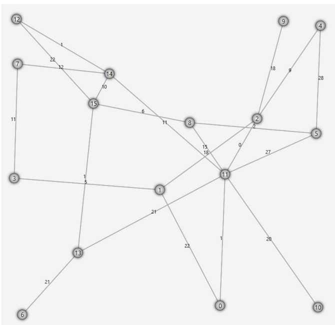
Abb. 3: F.-R. Algorithmus

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.

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

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

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

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 vorgeschlagene 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 hindernissärmsten 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 in einer *Distanz-Matrix* 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 gesam-

ten *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 und der Seed ist 7646137120994539520. Es wird immer der Weg vom *Knoten* Nr. 4 zu Nr. 12 gesucht.

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 nämlich das kontinuierliche „Gehen" in eine Richtung, sprich der *Graph* wird so lange wie möglich in eine Richtung *traversiert* (von lat. „entlang gehen") und erst sobald das Voranschreiten nicht mehr gegeben ist, werden Schritte zurückgegangen und ande-

Alg. 2 *Tiefensuche*

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

prozedur TIEFENSUCHE(n_a, n_b)
    markiere alle *Knoten* in G als *unbesucht*, außer n_a
    sei n_x aktiver *Knoten* n_a
    solange nicht alle *Knoten besucht* sind **wiederhole**
        wenn n_x ist n_b **dann**
            erschließe Weg P_{ab} durch *Vorgänger* und beende Suche
        ende wenn
        wenn n_x keine unbesuchten Nachbarn hat **dann**
            sei n_x *Vorgänger* von n_x
        sonst
            sortiere unbesuchte Nachbarn von n_x
            sei n_{next} unbesuchter Nachbar mit geringstem *Kantengewicht*
            setze n_x als *Vorgänger* von n_{next}
            sei n_x n_{next}
        ende wenn
    ende solange
ende prozedur

re 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 (Alg. 2)[4]. Falls ein Weg gefunden wird, so wird, wie in allen weiteren vorgestellten *Algorithmen*, *Backtracking* angewandt[5].

[4] vgl. Anhang: DFS.java
[5] [9, Kapitel 22.3, S. 457ff]

6.2 Heuristik als Mittel zum Ziel: Der Dijkstra-Algorithmus

Alg. 3 *Dijkstra-Algorithmus*

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

-

6.3 Der Allstar: 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

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

import de.dakror.wseminar.Const;

 * @author Dakror
public class GraphGenerator<V> {
   * @param params the bundle of generation parameters
  @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.*;

 * @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) {

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

    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);

    cleanup();
    Visualizer.resetAll(graph, false, false);

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

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)