

# BWINF 35, Runde 2 - Aufgabe 2

## Rechtsrum in Rechthausen

Robin Schmöcker

### Inhaltsverzeichnis

|          |  |    |
|----------|--|----|
| 1.       | Aufgabenteil 1: Definition Linksabbiegen .....                   | 4  |
| 2.       | Aufgabenteil 2: Veranschaulichung der Definition.....            | 5  |
| 2.1.     | Linksabbiegen als gefährliches Verkehrsmanöver .....             | 5  |
| 2.2.     | Anwendung der Definition auf verschiedene Beispiele .....        | 6  |
| 2.3.     | Anwendung der Definition auf Beispiel der Aufgabenstellung ..... | 7  |
| 3.       | Allgemeine Vorüberlegungen .....                                 | 8  |
| 3.1.     | Rechthausen als gerichteter Graph .....                          | 8  |
| 3.2.     | Beliebig Abbiegen als gerichteter Graph .....                    | 9  |
| 3.3.     | Winkelberechnung.....  | 9  |
| 3.4.     | Rechthausen als gewichteter Graph.....                           | 10 |
| 3.5.     | Umwandlung in gerichteten Graphen.....                           | 10 |
| 3.5.1.   | Umwandlung für Rechtsabbiegen .....                              | 10 |
| 3.5.2.   | Beispiel.....  | 11 |
| 3.5.3.   | Umwandlung für beliebig Abbiegen .....                           | 12 |
| 3.5.4.   | Laufzeit.....  | 12 |
| 3.5.5.   | Speicherverbrauch .....  | 12 |
| 4.       | Aufgabenteil 3: Kürzester Weg (Navi).....                        | 13 |
| 4.1.     | Idee .....   | 13 |
| 4.2.     | Umsetzung.....   | 13 |
| 4.2.1.   | Start und Zielknoten hinzufügen .....                            | 13 |
| 4.2.2.   | Dijkstra Algorithmus.....  | 14 |
| 4.2.3.   | Struktogramm.....  | 15 |
| 4.2.4.   | Laufzeit und Speicherverbrauch.....                              | 15 |
| 4.2.4.1. | Laufzeit.....  | 15 |
| 4.2.4.2. | Speicherverbrauch .....  | 15 |
| 4.2.5.   | Beispiele .....  | 16 |
| 5.       | Aufgabenteil 4: Sind alle Knoten gegenseitig erreichbar .....    | 23 |
| 5.1.     | Vorüberlegungen .....  | 23 |
| 5.1.1.   | – Gibt es nicht offensichtliche Szenarien? .....                 | 23 |
| 5.1.2.   | – Vorbereitung.....  | 24 |

|        |  |    |
|--------|--|----|
| 5.2.   | Lösungsansatz mittels Aufgabenteil 3 – verworfen .....                                 | 24 |
| 5.3.   | Grundidee .....  | 24 |
| 5.3.1. | Verbesserung 1 – Zusammenfassung von Zyklen.....                                       | 24 |
| 5.3.2. | Verbesserung 2 – Zwischenergebnisse der Breitensuche speichern.....                    | 25 |
| 5.3.3. | Verbesserung 3 – Gruppenkombinationen speichern .....                                  | 25 |
| 5.4.   | Zyklen zusammenfassen.....   | 25 |
| 5.5.   | Programmablauf .....   | 28 |
| 5.6.   | Laufzeit.....  | 28 |
| 5.7.   | Speicherverbrauch .....  | 29 |
| 5.8.   | Beispiele .....  | 30 |
| 5.8.1. | Beispiel 1 – Aufgabenstellungsgraph.....   | 30 |
| 5.8.2. | Beispiel 2 – Nicht alle Knoten erreichbar .....  | 31 |
| 6.     | Aufgabenteil 5: Größter Unterschied .....  | 32 |
| 6.1.1. | Vorüberlegung .....  | 32 |
| 6.1.2. | Brute-Force mit Dijkstra.....  | 32 |
| 6.1.3. | Heuristischer Ansatz.....  | 32 |
| 6.1.4. | Erste Idee Abstand 2.....  | 33 |
| 6.1.5. | Nächste Idee- In benachbarten Knoten nach größerem Faktor suchen (Umgesetzte Idee) ... | 34 |
| 6.2.   | Umsetzung.....   | 35 |
| 6.3.   | Programmablauf .....   | 35 |
| 6.4.   | Laufzeit.....  | 36 |
| 6.5.   | Speicherverbrauch .....  | 37 |
| 6.6.   | Beispiele .....  | 37 |
| 7.     | Inhaltliche Erweiterung: Effizientes illegales Abbiegen .....                          | 41 |
| 7.1.   | Vorüberlegung.....   | 41 |
| 7.2.   | Problematik .....  | 41 |
| 7.3.   | Idee und Umsetzung .....   | 41 |
| 7.4.   | Programmablauf .....   | 42 |
| 7.5.   | Laufzeit.....  | 42 |
| 7.6.   | Speicherverbrauch .....  | 42 |
| 7.7.   | Beispiele .....  | 43 |
| 7.8.   | Anwendung .....  | 45 |
| 8.     | Straßengenerator – Aufgabenteil 6 .....  | 47 |
| 8.1.   | Vorüberlegung.....   | 47 |
| 8.2.   | Umsetzung.....   | 47 |
| 8.3.   | Weiterer Nutzen .....  | 48 |

|      |   |     |
|------|---|-----|
| 9.   | Gemessene Laufzeiten und Speicherverbräuche ..... | 49  |
| 9.1. | Grenzen des Programms.....                        | 50  |
| 10.  | Programm-Dokumentation .....                      | 51  |
| 11.  | Quelltextauszüge .....                            | 58  |
| 12.  | Bedienungsanleitung für das GUI .....             | 103 |

## 1. Aufgabenteil 1: Definition Linksabbiegen

Meine Definitionen zum Linksabbiegen.

### 1. Definition des Winkels einer Straße zu einer Kreuzung.

Die Kreuzung sei der Ursprung eines Koordinatensystems. Die positive x-Achse entspreche  $0^\circ$ . Gegen diese Achse wird der Winkel, wie in der Mathematik üblich gegen den Uhrzeigersinn, der von der Kreuzung wegführenden Straße definiert (siehe Abb. 1-3). Der Winkel muss im Intervall  $[0^\circ; 360^\circ)$  liegen.

### 2. Definition Geradeausfahren.

Geradeausfahren wird wie im üblichen Sprachgebrauch definiert. Man befahre eine Kreuzung über eine Straße mit Winkel  $\alpha$  dann muss der Winkel der geradeausführenden Straße  $(\alpha+180^\circ) \bmod 360$  entsprechen. Beispiel siehe Abb. 1:  $d \rightarrow b$ ,  $b \rightarrow d$ .

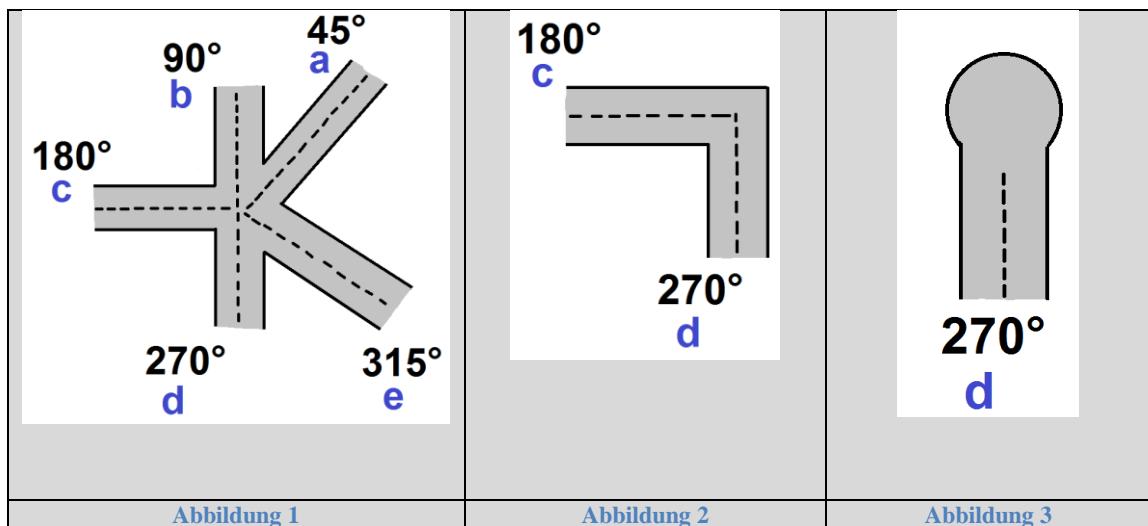
### 3. Definition Rechtsabbiegen.

Rechtsabbiegen wird als das Verlassen einer Straße über die Straße mit dem nächstgrößeren Winkel definiert. Existiert kein größerer Winkel, wird von  $0^\circ$  aufwärts gesucht. Einfache Beispiele siehe Abb. 1  $d \rightarrow e$ ,  $e \rightarrow a$ ,  $a \rightarrow b$ ,  $b \rightarrow c$ ,  $c \rightarrow d$ .

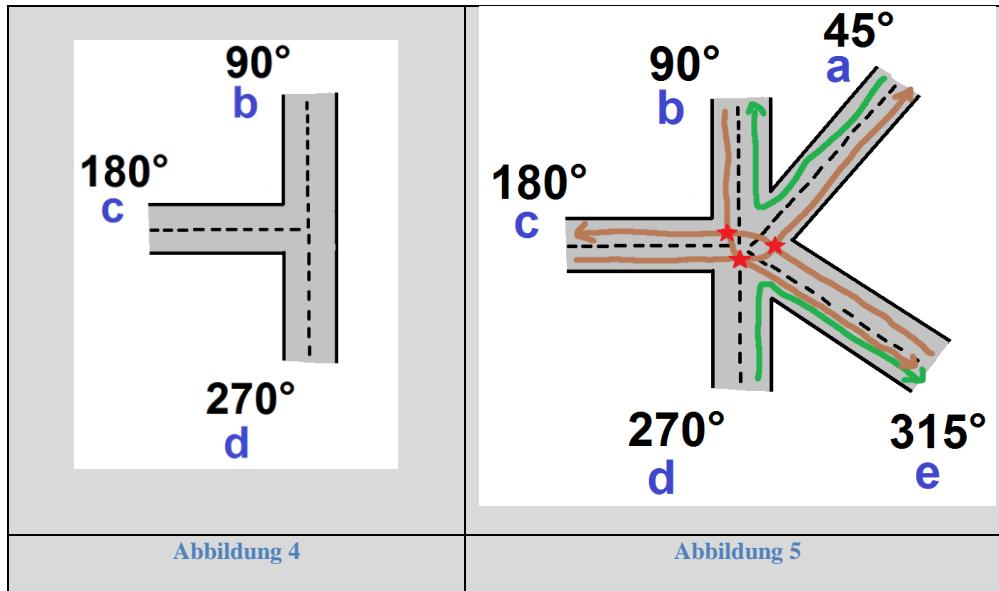
Anmerkung: Damit der nächst größere Winkel eindeutig ist, muss gewährleistet sein, dass keine zwei Straßen mit demselben Winkel auf eine Kreuzung zuführen (siehe auch 3.3 Winkelberechnung).

### 4. Definition Linksabbiegen.

Linksabbiegen ist das Verlassen einer Kreuzung über eine Straße, die weder als Rechtsabbiegen noch als Geradeausfahren zu kategorisieren ist.



## 2. Aufgabenteil 2: Veranschaulichung der Definition



### 2.1. Linksabbiegen als gefährliches Verkehrsmanöver

Beim Abbiegen von einer Straße entsteht eine gefährliche Situation dadurch, dass man die Fahrbahn entgegengesetzt fahrender Autos kreuzt. Meine Definition des Linksabbiegens erhält diese Problematik. In Abbildung 5 sind sicheres Rechtsabbiegen (grüne Pfeile) und einige gefahrenträchtige Linksabbiegefahrwege (braune Pfeile) sowie die möglichen Unfallstellen (rot) eingezeichnet.

Eine Gefahrensituation kann aber auch durch Geradeausfahren entstehen, weil zum Beispiel auf einer Kreuzung, wo sich zwei Straßen im rechten Winkel treffen, sich auch hier die Fahrwege kreuzen würden. Geradeausfahren wurde jedoch vom Bürgermeister explizit erlaubt.

## 2.2. Anwendung der Definition auf verschiedene Beispiele

| Kreuzung mit mehreren Straßen (Abb. 1) |                 |                  |               |  |
|--|-----------------|------------------|---------------|--|
| Annäherung auf Straße                  | Rechts-abbiegen | Geradeaus fahren | Linksabbiegen | Anmerkung  |
| a                                      | b               | -                | c,d,e,a       | Man beachte, dass das Verlassen über die Eingangsstraße bei dieser Kreuzung als Linksabbiegen zu betrachten ist. |
| b                                      | c               | d                | e,a,b         |  |
| c                                      | d               | -                | e,a,b,c       |  |
| d                                      | e               | b                | a,c,d         |  |
| e                                      | a               | -                | b,c,d,e       |  |

| Abknickender Straßenverlauf (Abb. 2) |                 |                  |               |  |
|--------------------------------------|-----------------|------------------|---------------|--|
| Annäherung auf Straße                | Rechts-abbiegen | Geradeaus fahren | Linksabbiegen | Anmerkung  |
| c                                    | d               | -                | c             | In solch einer Kreuzung muss immer die Straße befahren werden auf welcher man nicht gekommen ist, um nicht links abzubiegen. Auch hier wird durch die Definition das Wenden als Linksabbiegen gewertet und ist daher verboten. |
| d                                    | c               | -                | d             | Dem Straßenverlauf von d nach c folgen, gilt laut unserer Definition als Rechtsabbiegen, während im normalen Sprachgebrauch wohl eher von Linksabbiegen die Rede wäre, da das Lenkrad nach links gedreht werden muss.          |

| Sackgasse (Abb. 3)    |                 |                  |               |   |
|-----------------------|-----------------|------------------|---------------|---|
| Annäherung auf Straße | Rechts-abbiegen | Geradeaus fahren | Linksabbiegen | Anmerkung   |
| d                     | d               | -                | -             | Die einzige entartete „Kreuzung“ in der Wenden erlaubt ist, weil es als Rechtsabbiegen gilt |

| Besonderer Fälle (Abb. 4) |                 |                  |               |  |
|---------------------------|-----------------|------------------|---------------|--|
| Annäherung auf Straße     | Rechts-abbiegen | Geradeaus fahren | Linksabbiegen | Anmerkung  |
| b                         | c               | d                | b             |  |
| c                         | d               | -                | b,c           |  |
| d                         | b               | b                | c,d           | Hier tritt der Fall ein, dass das Abbiegen auf eine Straße gleichzeitig Geradeausfahren und Rechtsabbiegen ist, da beide Definitionen auf dieses Abbiegen zutreffen. Für die Definition des Linksabbiegens ist aber relevant, dass mindestens einer der anderen Fälle eintritt. In diesem Beispiel tun dies sogar beide. |

### 2.3. Anwendung der Definition auf Beispiel der Aufgabenstellung

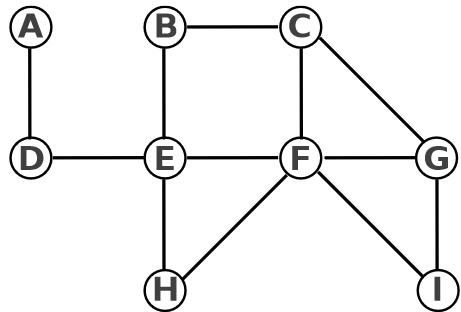


Abbildung 6

Laut Aufgabenstellung sollen die Routen DADEF sowie GCB und FEBC erlaubt sein, weil nicht links abgebogen wird, die Routen DEB und IGF hingegen nicht. Wendet man meine Definition auf die Routen dieses Beispiels an, so sieht man, dass diese Vorgabe an jeder Kreuzung erfüllt ist.

### 3. Allgemeine Vorüberlegungen

#### 3.1. Rechthausen als gerichteter Graph

Zunächst kann Rechthausen als ungerichteter Graph betrachtet werden. Das Eingabeformat zumindest entspricht dieser Vorstellung. Durch das Linksabbiegeverbot können beim Einfahren auf eine Kreuzung nicht immer alle Kanten bzw. Straßen zum Verlassen befahren werden. Daher wird ein Modell benötigt, das diese Situation darstellt. Das Modell wird ein statischer, gerichteter Graph sein mit dem Aufgaben gelöst werden können. Jede Kreuzung/Knoten wird anhand von mehreren Teilknoten repräsentiert, die das Befahren der Kreuzung über die einzelnen Straßen darstellen.

Im Folgenden wird anhand von Abb. 1 demonstriert, wie diese Repräsentation zu betrachten ist.

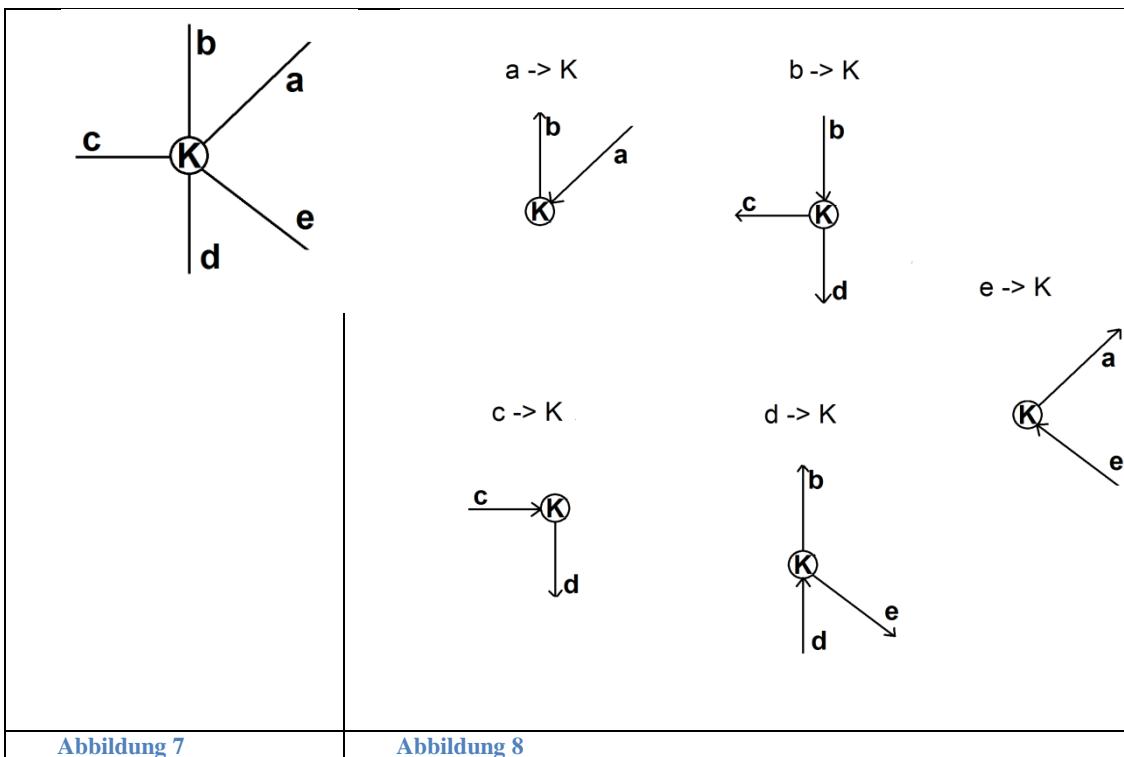


Abb. 7 entspricht Abb. 1 in abstrakter Form und Abb. 8 zeigt die Teilknoten, die das Überfahren der Kreuzung über jede mögliche Straße zeigen. Das Befahren der Kreuzung über Straße „b“ (eingehender Pfeil) erlaubt es nur noch über „c“ und „d“ weiterzufahren (ausgehende Pfeile). Dies entspricht übrigens auch der Tabelle in Abschnitt 2.2.

Um das Modell zu repräsentieren, wird dieses Prinzip nun auf jede Kreuzung des Graphen angewendet. Die eingehenden Kanten der Teilknoten entsprechen dem Befahren der Kreuzung über eine andere Kreuzung, die ausgehenden Kanten dem erlaubten Weiterfahren auf eine andere Kreuzung.

Der Vorteil dieses Modells ist, dass der neuerzeugte Graph nun statisch und gerichtet ist und genau den erlaubten Abbiegungen entspricht. Der Nachteil ist, dass nun jede Kreuzung anhand von mehreren Teilknoten repräsentiert wird und man in den Aufgabenteilen darauf Rücksicht nehmen

muss. Insbesondere muss beim Start von einer Kreuzung jeder Teilknoten als Startpunkt in Betracht gezogen werden.

Es werden folgende Begrifflichkeiten definiert, da später darauf Bezug genommen werden soll.

**Definition Überknoten:** Knoten aus der die Teilknoten hervorgegangen sind. Dies entspricht den Kreuzungen von Rechthausen.

**Definition Unterknoten:** Knoten, die das Befahren eines Überknotens über eine bestimmte Kante repräsentieren.

### 3.2. Beliebig Abbiegen als gerichteter Graph

Da der in 3.1 definierte Graph gerichtet ist, muss das Programm mit gerichteten Graphen arbeiten. Daher bietet es sich an, auch den ungerichteten Graphen aus den Eingabedateien in einen gerichteten Graphen zu konvertieren. Dies geschieht, indem man jede ungerichtete Kante durch zwei gerichtete Kanten ersetzt, die zwischen den beiden Endknoten hin- und zurückgehen. Dadurch wird vermieden, dass ein Algorithmus für gerichtete und ungerichtete Graphen geschrieben werden muss.

### 3.3. Winkelberechnung

Der Winkel wird berechnet gegen die x-Achse mittels der eingebauten Math.arctan2-Funktion, welche den Winkel direkt zurückgibt, was wiederum mit der eingebauten Math.toDegrees-Funktion umgewandelt wird. Der Winkel wird auf zwei Nachkommastellen gerundet. Das hat zwar den Nachteil, dass bei extrem weit entfernten Kreuzungen, die sehr dicht beieinander liegen, die einkommenden Straßen als gleiche Winkel gewertet würden. Eine Berechnung zeigt, dass bei einer Entfernung von 1000 Einheiten und einem Abstand auf der y-Achse von einer Einheit trotzdem noch eine Winkeldifferenz von  $0.06^\circ$  auftritt. Allerdings wären derart dicht nebeneinanderlaufende Straßen de facto miteinander verschmolzen und daher in der Realität eigentlich nicht möglich. Wenn man auch weiter entfernt liegende Straßen mit genauem Winkel abbilden möchte, müsste man nur die Rundungsroutine auf mehr Nachkommastellen anpassen. Der Vorteil dieser Methode liegt darin, dass Rundungsdifferenzen von Doublezahlen so vermieden werden und man gleiche und gegenüberliegende Winkel besser vergleichen kann.

Wie schon in der Definition zum Rechtsabbiegen erwähnt wurde, dürfen keine zwei Straßen mit dem gleichen Winkel auf eine Kreuzung treffen. Der Grund dafür ist, dass beim Rechtsabbiegen sonst möglicherweise mehrere Straßen die Bedingung des nächst größeren Winkels erfüllen und daher nicht mehr eindeutig entschieden werden könnte, auf welche Straßen abgebogen werden muss. Aber die eindeutige Entscheidbarkeit ist in der Aufgabenstellung vorausgesetzt.

### 3.4. Rechthausen als gewichteter Graph

In Teil 3 und 5 der Aufgabenstellung ist gefordert, die Lösung für verschiedene Weglängenmaße zu erstellen. Um die folgenden Algorithmen zu implementieren ist also ein kantengewichteter Graph zu wählen. Die Kapazität der befahrenen Kanten muss der Weglänge entsprechen.

- Ist für eine Route die Fahrstrecke gefordert, wird die Entfernung zweier Kreuzungen den Kanten als Kapazität zugewiesen, wobei hier die Entfernung zweier Knoten mittels Satz des Pythagoras, unter Einbeziehung der Koordinaten der Kreuzungen, gebildet werden kann.
- Ist für eine Route die Anzahl der überfahrenen Kreuzungen erwünscht, kann den Kanten einfach eine konstante Kapazität 1 zugewiesen werden. Dann entspricht die Weglänge der Anzahl der befahrenen Straßen.

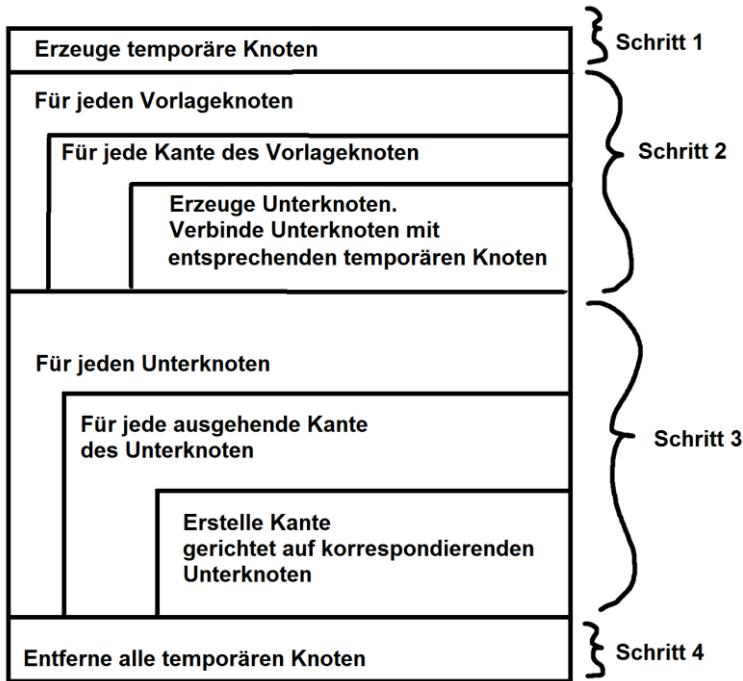
### 3.5. Umwandlung in gerichteten Graphen

#### 3.5.1. Umwandlung für Rechtsabbiegen

Die Umwandlung ist in 4 Schritte zu unterteilen.

1. Es wird ein neuer Graph erzeugt und alle Knoten des Vorlagegraphen werden in diesen kopiert. Die kopierten Knoten werden im Folgenden als „temporäre Knoten“ bezeichnet und werden folgend mit \* gekennzeichnet. Dies entspricht etwa einem Überknoten.
2. Für jede Kante eines Knotens des Vorlagegraphen wird ein weiterer Knoten X im neuen Graph erzeugt. Dies entspricht einem Unterknoten. Der neue Knoten X erhält eine Kante kommend von dem temporären Knoten Y\* der dem Quellknoten der Kante aus dem Vorlagegraph entspricht. Wenn man den Vorlageknoten über diese Kante besucht, hat man durch das Linksabbiegeverbot 1-2 Möglichkeiten die Kreuzung zu verlassen. Die Zielknoten der erlaubten ausgehenden Kanten des Vorlagegraphen haben durch 1. einen entsprechenden temporären Knoten Z\* im neuen Graphen. Der Knoten X erhält nun ausgehende Kanten zu diesen temporären Knoten Z\*. Man kann noch nicht direkt die korrekten Zielunterknoten verbinden, weil diese vielleicht noch gar nicht existieren.
3. Aktuell ist jeder Unterknoten Y noch mit temporären Knoten Z\* verbunden. Daher wird nun für jede ausgehende Kante von Y zum verbundenen temporären Knoten Z\* eine weitere Kante zu dem korrespondierenden Unterknoten Z erstellt. Es ist darauf zu achten, dass derjenige Unterknoten Z gewählt wird, der dem Befahren von Y\* nach Z\* entspricht. Es genügt nur die ausgehenden Kanten zu betrachten, da jede eingehende Kante eines Knoten ausgehende Kante von ihrem Quellknoten ist.
4. Alle temporären Knoten und damit verbundenen Kanten werden aus dem Graphen entfernt.

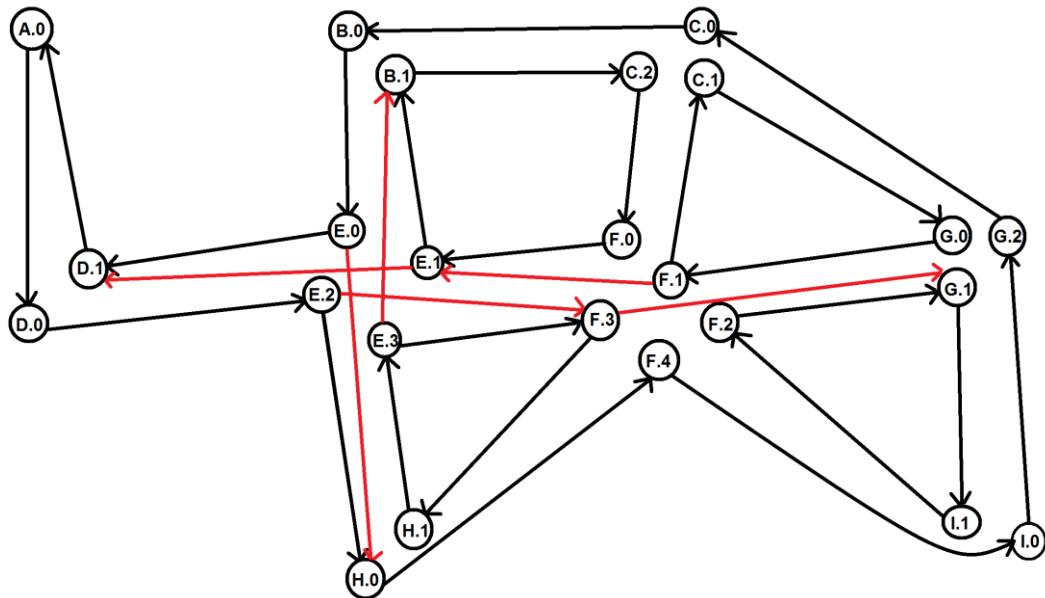
Im Folgenden ist ein Struktogramm des Programmablaufes zu sehen.



## Abbildung 9

### 3.5.2. Beispiel

Anhand des Straßenplans der Aufgabenstellung wird hier das Ergebnis der Umwandlung gezeigt. Man sieht, dass für jede Kreuzung genauso viele Unterknoten angelegt wurden, wie sie eingehende Straßen hat. Rot markiert habe ich diejenigen Kanten, die Geradeausfahnen darstellen. Alle anderen Kanten entsprechen Rechtsabbiegen und sind hier schwarz gefärbt.



## Abbildung 10

### 3.5.3. Umwandlung für beliebig Abbiegen

Siehe Punkt 3.2

### 3.5.4. Laufzeit

Anhand des Struktogramms zur Umwandlung in einen Rechtsabbiegegraphen wird die Laufzeit pro Schritt angegeben. Die Laufzeit wird anhand eines Graphen G mit  $|V|$  Knoten und  $|E|$  Kanten abgeschätzt.  $N = |V| + |E|$

Schritt 1:  $O(N)$  Im Grunde werden hier nur die Knoten des Vorlagegraphen kopiert

Schritt 2:  $O(|V| + 2*|E|)$  Jeder Knoten wird iteriert und letztlich resultiert jede Kante in dem Erzeugen von zwei Unterknoten, da ja jede Kante mit zwei Knoten verbunden ist.  $O(|V| + 2*|E|) \leq O(N + 2*N) = O(N)$

Schritt 3: Jeder Knoten hat vor diesem Schritt maximal zwei ausgehende Kanten. All diese Kanten werden umgerichtet auf die korrespondierenden Unterknoten. Anzahl der Knoten vor diesem Schritt ( $O(|V| + 2*|E|) \leq O(N)$ )

Die Laufzeit dieses Schrittes beträgt demnach:  $O(N*2) = O(N)$ .

Schritt 4:  $O(N)$  Es werden hier einfach die anfangs hinzugefügten Knoten, welche inzwischen Kanten gerichtet auf Unterknoten haben, entfernt

Die Gesamlaufzeit beträgt:  $O(N) + O(N) + O(N) + O(N) = O(4N) = O(N)$

Diese Annahme lässt sich in den gemessenen Laufzeiten für Graphen verschiedener Größen bestätigen (siehe Kapitel 9 – Darstellung: rote Raute).

### 3.5.5. Speicherverbrauch

Durch das Umwandeln in einen Rechtsabbiegegraphen werden pro Kante im Originalgraph, zwei Knoten erzeugt. Jeder dieser neu erzeugten Knoten kann nun maximal zwei eingehende und zwei ausgehende Kanten besitzen. Wenn der Originalgraph  $|V|$  Knoten und  $|E|$  besitzt so ist dessen Speicherverbrauch  $O(|V| + |E|)$  und demnach der Speicherverbrauch des umgewandelten Graphen  $O(2*|E| + 4*(2*|E|) + |V|) = O(|V| + |E|)$ . Der Speicherverbrauch des umgewandelten Graphen sowie des Originalgraphen ist also linear. Diese Annahme lässt sich in den gemessenen Speicherverbräuchen für Graphen verschiedener Größen bestätigen (siehe Kapitel 9 – Darstellung: rote Raute).

## 4. Aufgabenteil 3: Kürzester Weg (Navi)

### 4.1. Idee

Wir verwenden den in 3.4.1 erstellten Graphen, da dieser das Linksabbiegeverbot widerspiegelt. Wie bereits erwähnt muss darauf geachtet werden, dass eine Kreuzung von mehreren Unterknoten repräsentiert wird. Um den schnellsten Weg von einem Start- zu einem Zielknoten zu finden, muss in dem Graphen dafür gesorgt werden, dass über alle möglichen ausgehenden Kanten des Startknotens versucht wird, den kürzesten Weg zum Zielknoten zu finden, falls ein Weg existiert. Von allen möglichen Wegen wird der Kürzeste ausgewählt. Um alle Unterknoten des Startknotens durchzugehen, wird ein zusätzlicher Knoten erstellt, der eine Kante gerichtet auf jeden Startunterknoten besitzt. Die Kanten besitzen alle die Kapazität 0 insbesondere auch dann wenn die Weglänge anhand der Anzahl der besuchten Knoten gemessen wird, um die Länge des Weges nicht zu verfälschen. Für den erstellten Zielknoten werden analog eingehende Kanten von allen Unterknoten des Zielknotens erstellt. Danach kann der Dijkstra Algorithmus auf diesen Graphen angewendet werden, um das Ergebnis zu erhalten.

### 4.2. Umsetzung

Der Fall, dass der Startknoten gleich dem Zielknoten ist, ist trivial. Die Anwendung des folgenden Algorithmus ist für diesen Fall nicht von Nöten.

#### 4.2.1. Start und Zielknoten hinzufügen

Die Grundidee ist es den Originalstartknoten dem Graph wieder hinzuzufügen. Das Problem bei der Suche nach dem kürzesten Weg ist hierbei, dass man den schnellsten Weg zwischen allen Unterknoten des Starts A\* zu allen Unterknoten des Ziels D\* finden müsste. Der kürzeste dieser Wege wäre dann der kürzeste Weg zwischen den Kreuzungen. Die Frage nach dem schnellsten Weg lässt sich aber darauf reduzieren nur einen Weg innerhalb des umgewandelten Graphen zu finden. Man fügt dem Graphen zwei zusätzliche Knoten (A) und (D) hinzu. (A) hat eine Kante gerichtet auf jeden Unterknoten (hier A1 und A2) des Startknotens A\* mit der Kapazität 0, weil es sich hierbei nicht um eine echte Straße handelt, sondern um eine Repräsentation des Starts von allen Startunterknoten gleichzeitig. Der zweite Knoten (D) hat jeweils eine eingehende Kante kommend von allen Unterknoten (hier nur D) des Zielknoten D\* mit der Kapazität 0. Dies entspricht quasi dem Hinzufügen der Originalkreuzung in den Rechtsabbiegegraphen, der die Originalknoten nicht mehr enthält, sondern nur noch die Unterknoten hierzu.

Im Folgenden eine Visualisierung wie die Knoten hinzugefügt werden. Hier ist ein umgewandeltes Rechthausen abgebildet. Es wird der schnellste Weg zwischen A\* und D\* gesucht. Ohne weitere Veränderung müsste man den schnellsten Weg von beiden der A\* Unterknoten zu dem Unterknoten von D\* finden und den Schnellsten auswählen. Mit dem Hinzufügen der beiden Start (A) und Zielknoten (D) muss nur noch ein schnellster Weg zwischen diesen beiden hinzugefügten Knoten gefunden werden.

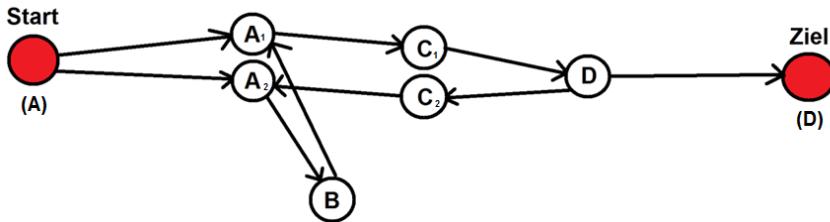


Abbildung 11

#### 4.2.2. Dijkstra Algorithmus

Der schnellste Weg innerhalb eines gerichteten, kantengewichteten Graphen kann mithilfe des Dijksta Algorithmus bestimmt werden.

Dies funktioniert wie folgt:

1. Alle Knoten erhalten nun zusätzlich die Eigenschaften Entfernung zum Startknoten und Vorgängerknoten und ob sie besucht wurden oder nicht. Der hinzugefügte Startknoten erhält eine Distanz von 0, die Distanz der restlichen Knoten des Graphen wird mit  $\infty$  initialisiert.
2. Solange noch unbesuchte Knoten existieren, wird unter diesen der Knoten mit der geringsten Entfernung ausgewählt. Ist der ausgewählte Knoten der Zielknoten oder entspricht dessen Entfernung  $\infty$ , wird der Algorithmus beendet. An dem jeweils ausgewählten Knoten werden folgende Operationen durchgeführt.
  - a. Der Knoten wird als besucht markiert
  - b. Die aktuelle Entfernung plus das Kantengewicht zu allen unbesuchten Nachbarknoten wird berechnet
  - c. Falls der berechnete Wert kleiner ist als die gespeicherte Distanz im Nachbarknoten, wird diese durch die Berechnete ersetzt. Zusätzlich wird dann im Nachbarknoten der aktuelle Knoten als Vorgänger eingetragen.
3. Der Weg kann zurückverfolgt (Backtracking) werden, indem man betrachtet, welchen Vorgänger der Zielknoten hat. Nun schaut man, welchen Vorgänger dieser Knoten hat. Dies wiederholt man solange, bis der Vorgänger Knoten dem Startknoten entspricht. Dies ist dann der schnellste Weg zwischen den beiden Knoten. Falls kein Weg zwischen den Knoten existiert, wurde der Zielknoten auch nicht besucht und dieser hat keinen Vorgänger.

Genauere Informationen zum Dijkstra Algorithmus inklusive einiger Visualisierungen findet man auf der englischen Wikipedia unter  
[https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm).

### 4.2.3. Struktogramm

|   |
|---|
| Wandle Ursprungsgraphen in einen gerichteten Graphen um<br>(siehe Punkt 3.5.1)                              |
| Füge einen Knoten, verbunden mit allen Startunterknoten, hinzu  <br>Kantenkapazität = 0 (siehe Punkt 4.2.2) |
| Füge einen Knoten, verbunden mit allen Zielunterknoten, hinzu  <br>Kantenkapazität = 0 (siehe Punkt 4.2.2)  |
| Wende Dijkstra Algorithmus auf den ergänzten Graphen an   |
| Extrahiere Weg aus gesammelten Informationen des Dijkstra<br>Algorithmus                                    |

### 4.2.4. Laufzeit und Speicherverbrauch

#### 4.2.4.1. Laufzeit

Das Umwandeln des Originalgraphen in den Rechtsabbiegegraph wird hier nicht berücksichtigt, da es beim Laden des Graphen geschieht und ab nun jede Navigation mit diesem Graphen beginnt.

Das Hinzufügen von den zwei extra Knoten wird ebenfalls vernachlässigt, da es sich de facto um eine fast konstante Laufzeit handelt.

Laut der deutschen Wikipedia ist die Zeit Komplexität für den Dijkstra Algorithmus bei Verwendung einer Vorrangwarteschlange etwa  $O(|V| * \log(|V|) + |E|)$  bei einem Graphen mit  $|V|$  Knoten und  $|E|$  Kanten. Das in der Implementierung verwendete Bibliotheksmodul „TreeSet“ setzt die benötigten Operationen (geringstes Element abrufen, Element aktualisieren) optimal um, weshalb die Laufzeit des Dijkstra Algorithmus in diesem Verfahren der obigen Laufzeit entspricht. Die Messwerte lassen diese Schlussfolgerung zumindest zu.

<https://de.wikipedia.org/wiki/Dijkstra-Algorithmus#Zeitkomplexit.C3.A4t>

Das Abrufen des Weges über die Daten des Dijkstra Algorithmus (Backtracking) liegt im Normalfall deutlich unter  $O(|V|)$ . Das schlechteste Szenario wäre, wenn der Weg aus sämtlichen Knoten des Graphen bestünde.

Insgesamt ergibt sich also eine Laufzeit von:

$$O(1) + O(|V| * \log(|V|) + |E|) + O(|V|) = O(|V| * \log(|V|)).$$

Die abgeschätzte Laufzeit lässt sich in den Messreihen zur Anwendung dieses Verfahrens wiederfinden (siehe Kapitel 9 – gelbe Raute).

#### 4.2.4.2. Speicherverbrauch

Da für jeden Knoten des Graphen lediglich zusätzlich seine Informationen bezüglich des Dijkstra Algorithmus gespeichert werden (Entfernung, Vorgänger, Besucht) und auch neben dem Rechtsabbiegegraphen zusätzlich keine signifikanten Informationen gespeichert werden, beträgt der Speicherverbrauch bei einem Graphen mit  $|V|$  Knoten und  $|E|$  Kanten  $O(|V| + |E|) = O(N)$ . Dieser lineare Speicherverbrauch ist ebenfalls wieder in den Messungen dazu nachzuvollziehen (siehe Kapitel 9 – Darstellung: gelbe Raute).

#### 4.2.5. Beispiele

##### Beispiel 1:

Es ist der schnellste Weg von A zu D gesucht. In diesem Beispiel wurde als Weglängenmaß die Knotenanzahl gewählt. Das heißt, allen Kanten wird eine Kapazität von 1 zugewiesen.

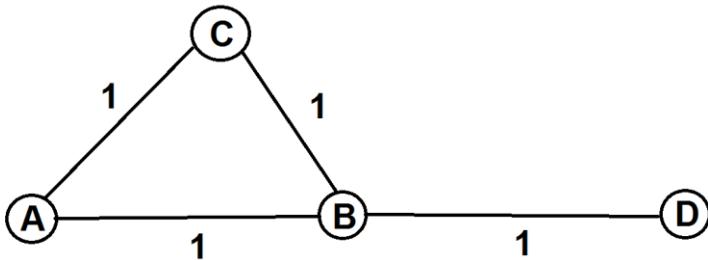


Abbildung 12

Im nächsten Schritt wird der Graph in einen gerichteten Graphen umgewandelt. Es werden auch Start und Zielknoten, verbunden mit den Unterknoten von A und D, hinzugefügt.

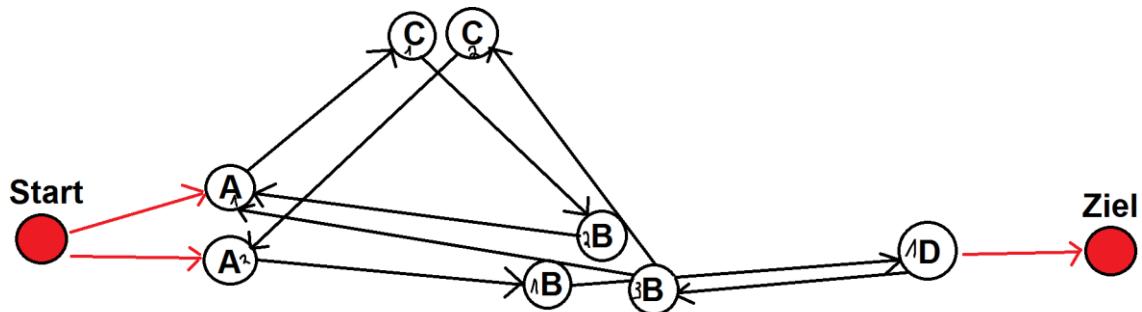


Abbildung 13

Im darauf folgenden Schritt wird der Dijkstra Algorithmus auf den Graphen angewendet mit den hinzugefügten Knoten als Start und Zielknoten. Nachdem keine unbesuchten Knoten mehr existieren, oder der Zielknoten gefunden wurde, wird der Weg abgerufen. Bei korrekter Anwendung wurde der Zielknoten von  $D_1$  aufgerufen,  $D_1$  von  $B_1$  und  $B_1$  von  $A_2$  und  $A_2$  von Start. Der Weg besteht zwar aus Unterknoten, bei der Ausgabe werden aber nur die Überknoten der Unterknoten ausgegeben. Um den Weg in richtiger Reihenfolge zu erhalten, drehen wir die oben beschriebene Reihenfolge um und entfernen die imaginären Start und Zielknoten und erhalten als schnellsten Weg von A zu D:  $A \rightarrow B \rightarrow D$

Im Folgenden werden die einzelnen Schritte des Dijkstra-Algorithmus bei diesem Graphen tabellarisch dargestellt.

| Knoten     | A1       | A2       | C1       | C2       | B1       | B2       | B3       | D1   | Start    | Ziel |
|------------|----------|----------|----------|----------|----------|----------|----------|------|----------|------|
| Vorgänger  | -        | -        | -        | -        | -        | -        | -        | null | -        |      |
| Entfernung | $\infty$ | 0    | $\infty$ |      |
| Besucht    | n        | n        | n        | n        | n        | n        | n        | n    | n        |      |

| Knoten     | A1    | A2    | C1 | C2       | B1 | B2 | B3       | D1   | Start    | Ziel |
|------------|-------|-------|----|----------|----|----|----------|------|----------|------|
| Vorgänger  | Start | Start | A1 | -        | A2 | C1 | -        | null | -        |      |
| Entfernung | 0     | 0     | 1  | $\infty$ | 1  | 2  | $\infty$ | 0    | $\infty$ |      |
| Besucht    | j     | j     | j  | n        | n  | n  | n        | j    | n        |      |

| Knoten     | A1    | A2    | C1       | C2       | B1       | B2       | B3       | D1   | Start    | Ziel |
|------------|-------|-------|----------|----------|----------|----------|----------|------|----------|------|
| Vorgänger  | Start | Start | -        | -        | -        | -        | -        | null | -        |      |
| Entfernung | 0     | 0     | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0    | $\infty$ |      |
| Besucht    | n     | n     | n        | n        | n        | n        | n        | j    | n        |      |

| Knoten     | A1    | A2    | C1 | C2       | B1 | B2 | B3       | D1 | Start | Ziel     |
|------------|-------|-------|----|----------|----|----|----------|----|-------|----------|
| Vorgänger  | Start | Start | A1 | -        | A2 | C1 | -        | B1 | null  | -        |
| Entfernung | 0     | 0     | 1  | $\infty$ | 1  | 2  | $\infty$ | 2  | 0     | $\infty$ |
| Besucht    | j     | j     | j  | n        | j  | n  | n        | j  | n     |          |

| Knoten     | A1    | A2    | C1 | C2       | B1       | B2       | B3       | D1   | Start    | Ziel |
|------------|-------|-------|----|----------|----------|----------|----------|------|----------|------|
| Vorgänger  | Start | Start | A1 | -        | -        | -        | -        | null | -        |      |
| Entfernung | 0     | 0     | 1  | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0    | $\infty$ |      |
| Besucht    | j     | n     | n  | n        | n        | n        | n        | j    | n        |      |

| Knoten     | A1    | A2    | C1 | C2       | B1 | B2 | B3       | D1 | Start | Ziel |
|------------|-------|-------|----|----------|----|----|----------|----|-------|------|
| Vorgänger  | Start | Start | A1 | -        | A2 | C1 | -        | B1 | null  | D1   |
| Entfernung | 0     | 0     | 1  | $\infty$ | 1  | 2  | $\infty$ | 2  | 0     | 2    |
| Besucht    | j     | j     | j  | n        | j  | n  | n        | j  | j     | n    |

| Knoten     | A1    | A2    | C1 | C2       | B1       | B2       | B3       | D1   | Start    | Ziel |
|------------|-------|-------|----|----------|----------|----------|----------|------|----------|------|
| Vorgänger  | Start | Start | A1 | -        | -        | -        | -        | null | -        |      |
| Entfernung | 0     | 0     | 1  | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0    | $\infty$ |      |
| Besucht    | j     | j     | n  | n        | n        | n        | n        | j    | n        |      |

| Knoten     | A1    | A2    | C1 | C2       | B1 | B2 | B3       | D1 | Start | Ziel |
|------------|-------|-------|----|----------|----|----|----------|----|-------|------|
| Vorgänger  | Start | Start | A1 | -        | A2 | C1 | -        | B1 | null  | D1   |
| Entfernung | 0     | 0     | 1  | $\infty$ | 1  | 2  | $\infty$ | 2  | 0     | 2    |
| Besucht    | j     | j     | j  | n        | j  | n  | n        | j  | j     | n    |

**Weg: Start -> A2 -> B1 -> D1 -> Ziel**

**Abbildung 14**

Es ist auch möglich, dass in Rechthausen kein Weg zwischen zwei Knoten existiert. Würde man das oben beschriebene Verfahren trotzdem anwenden, erhält man trotzdem am Ende Daten zu den einzelnen Knoten (siehe 7)). Würde man nun probieren den Weg aus den Daten zu extrahieren, wird der Zielknoten keinen Vorgänger haben, da ja kein Weg existiert. Ist dies der Fall kann identifiziert werden, dass kein Weg zwischen zwei Knoten existiert.

Angenommen Knoten B würde im obigen Graphen nicht existieren, dann existiert kein Weg zwischen Knoten A und D. Der Graph würde so aussehen.



**Abbildung 15**

Im Folgenden befinden sich wieder die Daten zu den einzelnen Knoten während des Dijkstra-Algorithmus zum obigen Graphen. Dieses Beispiel verdeutlicht wie erkannt wird, dass kein Weg zwischen zwei Knoten existiert unabhängig davon, ob es sich um ein umgewandeltes Rechthausen handelt. A wird als Startknoten und D als Zielknoten gewählt.

| 1)         | <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Knoten</th> <th>A</th> <th>C</th> <th>D</th> </tr> </thead> <tbody> <tr> <td>Vorgänger</td> <td>null</td> <td>-</td> <td>-</td> </tr> <tr> <td>Entfernung</td> <td>0</td> <td><math>\infty</math></td> <td><math>\infty</math></td> </tr> <tr> <td>Besucht</td> <td>n</td> <td>n</td> <td>n</td> </tr> </tbody> </table> | Knoten   | A        | C | D | Vorgänger | null | - | - | Entfernung | 0 | $\infty$ | $\infty$ | Besucht | n | n | n | 3) | <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Knoten</th> <th>A</th> <th>C</th> <th>D</th> </tr> </thead> <tbody> <tr> <td>Vorgänger</td> <td>null</td> <td>A</td> <td>-</td> </tr> <tr> <td>Entfernung</td> <td>0</td> <td>1</td> <td><math>\infty</math></td> </tr> <tr> <td>Besucht</td> <td>j</td> <td>j</td> <td>n</td> </tr> </tbody> </table> | Knoten | A | C | D | Vorgänger | null | A | - | Entfernung | 0 | 1 | $\infty$ | Besucht | j | j | n |
|------------|---|----------|----------|---|---|-----------|------|---|---|------------|---|----------|----------|---------|---|---|---|----|---|--------|---|---|---|-----------|------|---|---|------------|---|---|----------|---------|---|---|---|
| Knoten     | A   | C        | D        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Vorgänger  | null  | -        | -        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Entfernung | 0   | $\infty$ | $\infty$ |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Besucht    | n   | n        | n        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Knoten     | A   | C        | D        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Vorgänger  | null  | A        | -        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Entfernung | 0   | 1        | $\infty$ |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Besucht    | j   | j        | n        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| 2)         | <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Knoten</th> <th>A</th> <th>C</th> <th>D</th> </tr> </thead> <tbody> <tr> <td>Vorgänger</td> <td>null</td> <td>A</td> <td>-</td> </tr> <tr> <td>Entfernung</td> <td>0</td> <td>1</td> <td><math>\infty</math></td> </tr> <tr> <td>Besucht</td> <td>j</td> <td>n</td> <td>n</td> </tr> </tbody> </table>                   | Knoten   | A        | C | D | Vorgänger | null | A | - | Entfernung | 0 | 1        | $\infty$ | Besucht | j | n | n | 4) | <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Knoten</th> <th>A</th> <th>C</th> <th>D</th> </tr> </thead> <tbody> <tr> <td>Vorgänger</td> <td>null</td> <td>A</td> <td>-</td> </tr> <tr> <td>Entfernung</td> <td>0</td> <td>1</td> <td><math>\infty</math></td> </tr> <tr> <td>Besucht</td> <td>j</td> <td>j</td> <td>j</td> </tr> </tbody> </table> | Knoten | A | C | D | Vorgänger | null | A | - | Entfernung | 0 | 1 | $\infty$ | Besucht | j | j | j |
| Knoten     | A   | C        | D        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Vorgänger  | null  | A        | -        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Entfernung | 0   | 1        | $\infty$ |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Besucht    | j   | n        | n        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Knoten     | A   | C        | D        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Vorgänger  | null  | A        | -        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Entfernung | 0   | 1        | $\infty$ |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |
| Besucht    | j   | j        | j        |   |   |           |      |   |   |            |   |          |          |         |   |   |   |    |   |        |   |   |   |           |      |   |   |            |   |   |          |         |   |   |   |

Abbildung 16

Nach Schritt 4) wurden alle Knoten des Graphen besucht. Nun würde man probieren, den Weg zurückzuverfolgen. Doch beim Zielknoten D ist kein Vorgänger vorhanden, das heißt, er wurde von keinem anderen Knoten entdeckt. Es kann also hier festgestellt werden, dass kein Weg zwischen Knoten A und D existiert.

Im Anschluss das Ergebnis meines Programms, angewendet auf diese Aufgabe mit und ohne Knoten B.

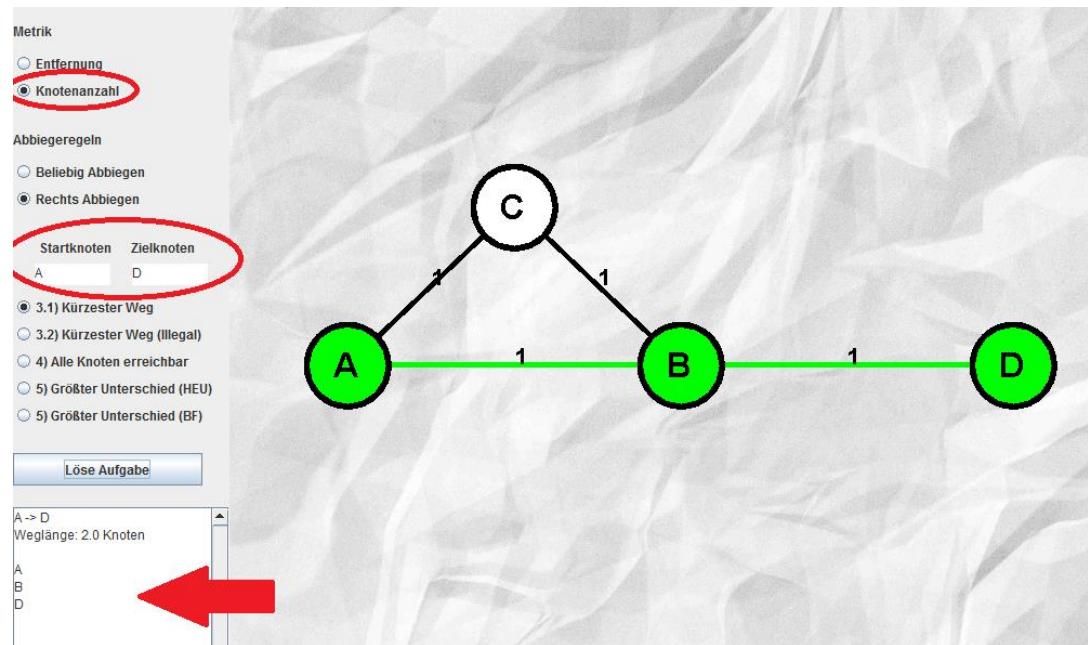


Abbildung 17 (Graph liegt bei)

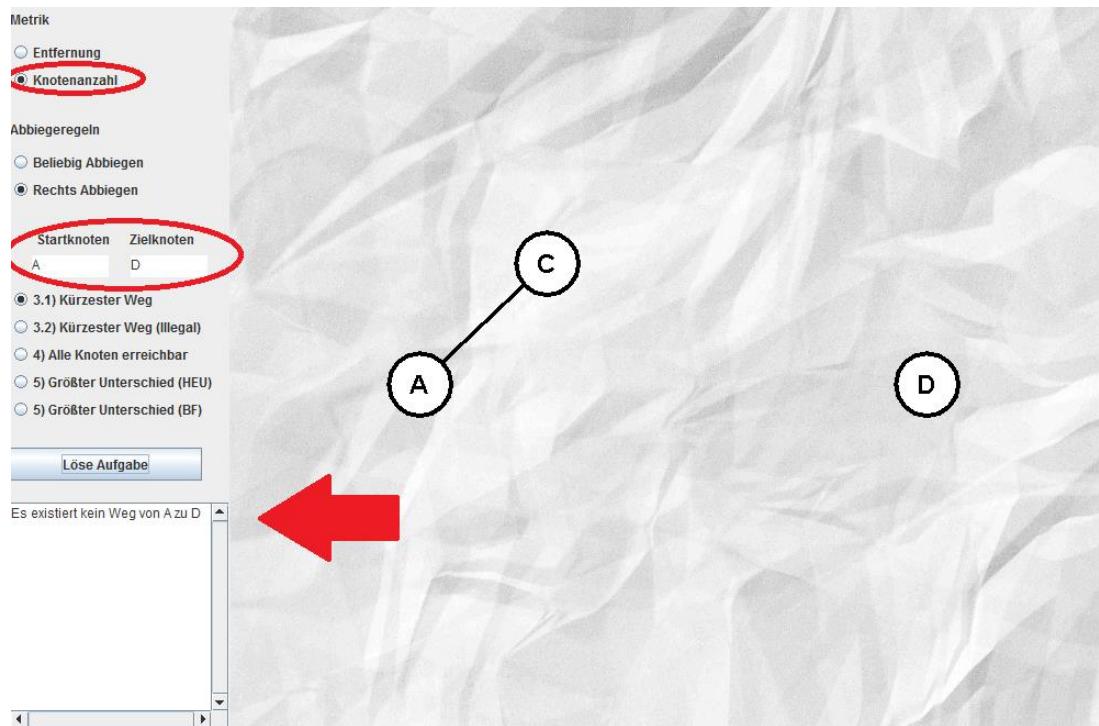


Abbildung 18 (Graph liegt bei)

### Beispiel 2:

Im Folgenden wird der Graph aus der Aufgabenstellung behandelt (Abb. 6). Es wurde mithilfe des Programms eine Tabelle erstellt, die die kürzesten Wege von allen Knoten zu allen Knoten mit den beiden Weglängenmaßen sowie mit- und ohne Linksabbiegen enthält. Die obere Tabelle verwendet die Knotenanzahl als Weglängenmaß, die untere die Entfernung. Existieren zwischen zwei Knoten mehrere Wege mit gleicher Entfernung, wurde nur einer von ihnen in der Tabelle dargestellt. Zusätzlich, um diese Tabellen später zu verwenden, wurde gleichzeitig noch der Quotient der beiden Weglängen gebildet und auf 2 Nachkommastellen gerundet.

Die Tabelle muss so gelesen werden: Weg von S einer Zeile zu T einer Spalte

|                   | A    | B                 | C                 | D               | E               | F           | G             | H         | I             |
|-------------------|------|-------------------|-------------------|-----------------|-----------------|-------------|---------------|-----------|---------------|
| Beliebig abbiegen |      | 3 A-D-E-B         | 4 A-D-E-F-C       | 1 A-D           | 2 A-D-E         | 3 A-D-E-F   | 4 A-D-E-F-G   | 3 A-D-E-H | 4 A-D-E-F-I   |
| Rechts abbiegen   | A    | 6 A-D-E-F-H-E-B   | 7 A-D-E-H-F-I-G-C | 1 A-D           | 2 A-D-E         | 3 A-D-E-F   | 4 A-D-E-F-G   | 3 A-D-E-H | 5 A-D-E-H-F-I |
| Faktor            |      | 2,00              | 1,75              | 1,00            | 1               | 1           | 1             | 1,00      | 1,25          |
| Beliebig abbiegen |      | 3 B-E-D-A         | 1 B-C             | 2 B-E-D         | 1 B-E           | 2 B-C-F     | 2 B-C-G       | 2 B-E-H   | 3 B-C-G-I     |
| Rechts abbiegen   | B    | 3 B-E-D-A         | 1 B-C             | 2 B-E-D         | 1 B-E           | 2 B-C-F     | 5 B-E-H-F-I-G | 2 B-E-H   | 4 B-E-H-F-I   |
| Faktor            | 1,00 |                   | 1,00              | 1,00            | 1               | 1           | 2,5           | 1,00      | 1,33          |
| Beliebig abbiegen |      | 4 C-F-E-D-A       | 1 C-B             |                 | 3 C-F-E-D       | 1 C-F       | 1 C-G         | 2 C-F-H   | 2 C-G-I       |
| Rechts abbiegen   | C    | 4 C-B-E-D-A       | 1 C-B             |                 | 3 C-F-E-D       | 1 C-F       | 1 C-G         | 3 C-B-E-H | 5 C-B-E-H-F-I |
| Faktor            | 1,00 |                   | 1,00              |                 | 1,00            | 1           | 1             | 1,50      | 2,50          |
| Beliebig abbiegen |      | 1 D-A             | 2 D-E-B           | 3 D-E-F-C       |                 | 1 D-E       | 2 D-E-F       | 3 D-E-F-G | 2 D-E-H       |
| Rechts abbiegen   | D    | 1 D-A             | 5 D-E-F-H-E-B     | 6 D-E-H-F-I-G-C |                 | 1 D-E       | 2 D-E-F       | 3 D-E-F-G | 2 D-E-H       |
| Faktor            | 1,00 |                   | 2,50              | 2,00            |                 | 1           | 1             | 1,00      | 1,33          |
| Beliebig abbiegen |      | 2 E-D-A           | 1 E-B             | 2 E-F-C         | 1 E-D           |             | 1 E-F         | 2 E-F-G   | 1 E-H         |
| Rechts abbiegen   | E    | 2 E-D-A           | 1 E-B             | 2 E-B-C         | 1 E-D           |             | 1 E-F         | 2 E-F-G   | 1 E-H         |
| Faktor            | 1,00 |                   | 1,00              | 1,00            | 1,00            |             | 1             | 1,00      | 1,50          |
| Beliebig abbiegen |      | 3 F-E-D-A         | 2 F-C-B           | 1 F-C           | 2 F-E-D         | 1 F-E       |               | 1 F-G     | 1 F-H         |
| Rechts abbiegen   | F    | 3 F-E-D-A         | 2 F-E-B           | 1 F-C           | 2 F-E-D         | 1 F-E       |               | 1 F-G     | 1 F-H         |
| Faktor            | 1,00 |                   | 1,00              | 1,00            | 1,00            | 1           |               | 1         | 1,00          |
| Beliebig abbiegen |      | 4 G-F-E-D-A       | 2 G-C-B           | 1 G-C           | 3 G-F-E-D       | 2 G-F-E     | 1 G-F         |           | 2 G-F-H       |
| Rechts abbiegen   | G    | 4 G-F-E-D-A       | 2 G-C-B           | 1 G-C           | 3 G-F-E-D       | 2 G-F-E     | 1 G-F         |           | 4 G-C-B-E-H   |
| Faktor            | 1,00 |                   | 1,00              | 1,00            | 1,00            | 1           |               | 2,00      | 1,00          |
| Beliebig abbiegen |      | 3 H-E-D-A         | 2 H-E-B           | 2 H-F-C         | 2 H-E-D         | 1 H-E       | 1 H-F         | 2 H-F-G   |               |
| Rechts abbiegen   | H    | 7 H-E-B-C-F-E-D-A | 2 H-E-B           | 3 H-E-B-C       | 6 H-E-B-C-F-E-D | 1 H-E       | 1 H-F         | 3 H-F-I-G |               |
| Faktor            | 2,33 |                   | 1,00              | 1,50            | 3,00            | 1           | 1             | 1,5       |               |
| Beliebig abbiegen |      | 4 I-F-E-D-A       | 3 I-F-E-B         | 2 I-G-C         | 3 I-F-E-D       | 2 I-F-E     | 1 I-F         | 1 I-G     | 2 I-F-H       |
| Rechts abbiegen   | I    | 6 I-G-C-B-E-D-A   | 3 I-G-C-B         | 2 I-G-C         | 5 I-G-C-B-E-D   | 4 I-G-C-B-E | 1 I-F         | 1 I-G     | 5 I-G-C-B-E-H |
| Faktor            | 1,50 |                   | 1,00              | 1,00            | 1,67            | 2           | 1             | 1         | 2,50          |

Abbildung 19

|                   | A    | B                     | C                     | D                   | E                   | F              | G                 | H            | I                 |
|-------------------|------|-----------------------|-----------------------|---------------------|---------------------|----------------|-------------------|--------------|-------------------|
| Beliebig abbiegen |      | 6,00 A-D-E-B          | 8,00 A-D-E-B-C        | 2,00 A-D            | 4,00 A-D-E          | 6,00 A-D-E-F   | 8,00 A-D-E-F-G    | 6,00 A-D-E-H | 8,83 A-D-E-F-I    |
| Rechts abbiegen   | A    | 12,83 A-D-E-F-H-E-B   | 14,83 A-D-E-F-H-E-B-C | 2,00 A-D            | 4,00 A-D-E          | 6,00 A-D-E-F   | 8,00 A-D-E-F-G    | 6,00 A-D-E-H | 10,00 A-D-E-F-G-I |
| Faktor            | 2,14 |                       | 1,85                  | 1,00                |                     | 1,00           | 1,00              | 1,00         | 1,13              |
| Beliebig abbiegen |      | 6,00 B-E-D-A          | 2,00 B-C              | 4,00 B-E-D          | 2,00 B-E            | 4,00 B-C-F     | 4,83 B-C-G        | 4,00 B-E-H   | 6,83 B-E-F-I      |
| Rechts abbiegen   | B    | 6,00 B-E-D-A          | 2,00 B-C              | 4,00 B-E-D          | 2,00 B-E            | 4,00 B-C-F     | 11,66 B-E-H-F-I-G | 4,00 B-E-H   | 9,66 B-E-H-F-I    |
| Faktor            | 1,00 |                       | 1,00                  | 1,00                | 1,00                | 1,00           | 2,41              | 1,00         | 1,41              |
| Beliebig abbiegen |      | 8,00 C-B-E-D-A        | 2,00 C-B              |                     | 6,00 C-B-E-D        | 4,00 C-B-E     | 2,00 C-F          | 2,83 C-G     | 4,83 C-F-H        |
| Rechts abbiegen   | C    | 8,00 C-B-E-D-A        | 2,00 C-B              |                     | 6,00 C-F-E-D        | 4,00 C-F-E     | 2,00 C-F          | 2,83 C-G     | 6,00 C-B-E-H      |
| Faktor            | 1,00 |                       | 1,00                  |                     | 1,00                | 1,00           | 1,00              | 1,24         | 2,41              |
| Beliebig abbiegen |      | 2,00 D-A              | 4,00 D-E-B            | 6,00 D-E-F-C        |                     | 2,00 D-E       | 4,00 D-E-F        | 6,00 D-E-F-G | 4,00 D-E-H        |
| Rechts abbiegen   | D    | 2,00 D-A              | 10,83 D-E-F-H-E-B     | 12,83 D-E-F-H-E-B-C |                     | 2,00 D-E       | 4,00 D-E-F        | 6,00 D-E-F-G | 8,00 D-E-F-G-I    |
| Faktor            | 1,00 |                       | 2,71                  | 2,14                |                     | 1,00           | 1,00              | 1,00         | 1,17              |
| Beliebig abbiegen |      | 4,00 E-D-A            | 2,00 E-B              | 4,00 E-B-C          | 2,00 E-D            |                | 2,00 E-F          | 4,00 E-F-G   | 2,00 E-H          |
| Rechts abbiegen   | E    | 4,00 E-D-A            | 2,00 E-B              | 4,00 E-B-C          | 2,00 E-D            |                | 2,00 E-F          | 4,00 E-F-G   | 2,00 E-H          |
| Faktor            | 1,00 |                       | 1,00                  | 1,00                | 1,00                |                | 1,00              | 1,00         | 1,24              |
| Beliebig abbiegen |      | 6,00 F-E-D-A          | 4,00 F-C-B            | 2,00 F-C            | 4,00 F-E-D          | 2,00 F-E       |                   | 2,00 F-G     | 2,83 F-H          |
| Rechts abbiegen   | F    | 6,00 F-E-D-A          | 4,00 F-E-B            | 2,00 F-C            | 4,00 F-E-D          | 2,00 F-E       |                   | 2,00 F-G     | 2,83 F-I          |
| Faktor            | 1,00 |                       | 1,00                  | 1,00                | 1,00                |                | 1,00              | 1,00         | 1,00              |
| Beliebig abbiegen |      | 8,00 G-F-E-D-A        | 4,83 G-C-B            | 2,83 G-C            | 6,00 G-F-E-D        | 4,00 G-F-E     | 2,00 G-F          |              | 4,83 G-F-H        |
| Rechts abbiegen   | G    | 8,00 G-F-E-D-A        | 4,83 G-C-B            | 2,83 G-C            | 6,00 G-F-E-D        | 4,00 G-F-E     | 2,00 G-F          |              | 8,83 G-C-B-E-H    |
| Faktor            | 1,00 |                       | 1,00                  | 1,00                | 1,00                |                | 1,00              |              | 1,83              |
| Beliebig abbiegen |      | 6,00 H-E-D-A          | 4,00 H-E-B            | 4,83 H-F-C          | 4,00 H-E-D          | 2,00 H-E       | 2,83 H-F          | 4,83 H-F-G   |                   |
| Rechts abbiegen   | H    | 14,00 H-E-B-C-F-E-D-A | 4,00 H-E-B            | 6,00 H-E-B-C        | 12,00 H-E-B-C-F-E-D | 2,00 H-E       | 2,83 H-F          | 6,00 H-E-F-G |                   |
| Faktor            | 2,33 |                       | 1,00                  | 1,24                | 3,00                | 1,00           | 1,00              | 1,24         | 1,00              |
| Beliebig abbiegen |      | 8,83 I-F-E-D-A        | 6,83 I-G-C-B          | 4,83 I-G-C          | 6,83 I-F-E-D        | 4,83 I-F-E     | 2,83 I-F          | 2,00 I-G     | 5,66 I-F-H        |
| Rechts abbiegen   | I    | 12,83 I-G-C-B-E-D-A   | 6,83 I-G-C-B          | 4,83 I-G-C          | 10,83 I-G-C-B-E-D   | 8,83 I-G-C-B-E | 2,83 I-F          | 2,00 I-G     | 10,83 I-G-C-B-E-H |
| Faktor            | 1,45 |                       | 1,00                  | 1,00                | 1,59                | 1,83           | 1,00              | 1,00         | 1,91              |

Abbildung 20

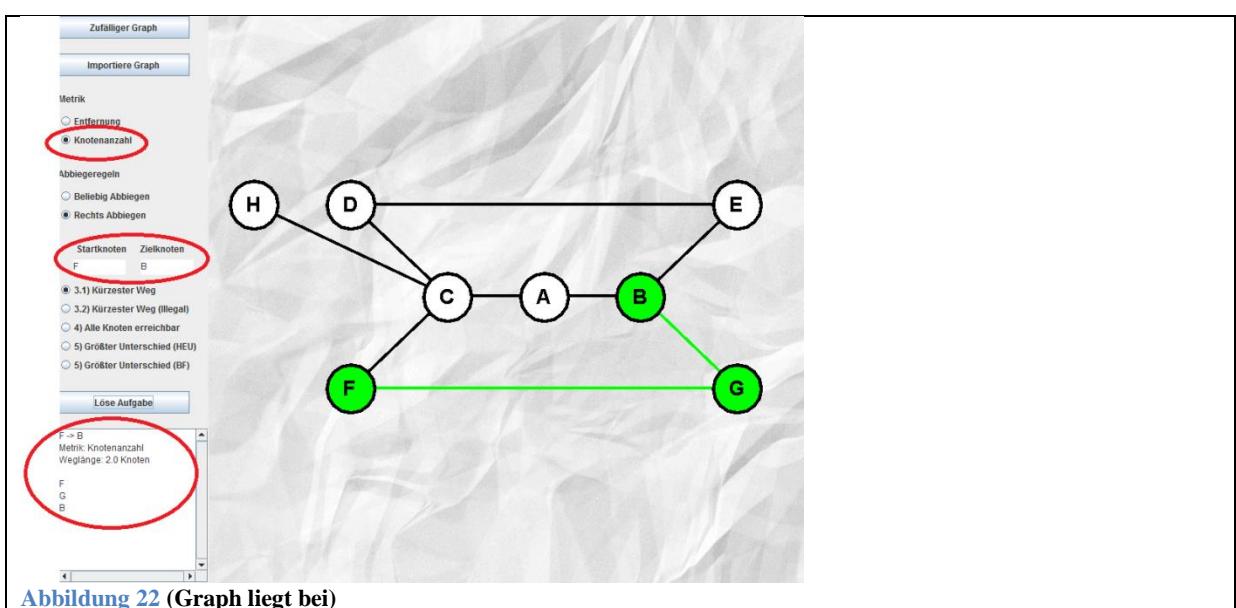
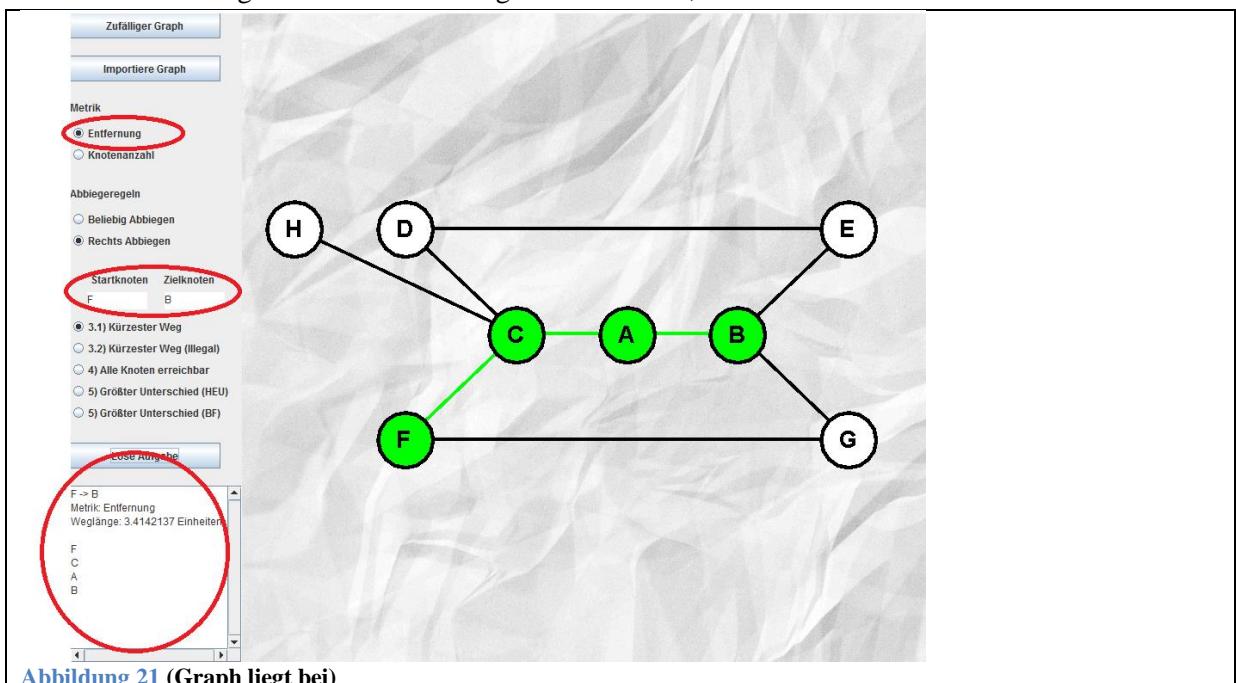
Rot eingefärbten Zellen sind die in der Aufgabenstellung erwähnten Wege A → C und H → A. Demnach ist der kürzeste Weg von A nach C: A → D → E → F → H → E → B → C mit einer Weglänge

von 7 Knoten oder 14,83 Einheiten. Der kürzeste Weg von H nach A ist: H→E→B→C→F→E→D→A mit einer Weglänge von 7 Knoten oder 14 Einheiten.

Die blaue Färbung kann zunächst ignoriert werden, da diese nur für den Aufgabenteil 5 relevant ist.

### Beispiel 3:

Als nächstes wird ein Beispiel behandelt bei welchem durch die verschiedenen Weglängenmaße sich unterschiedliche kürzeste Wege ergeben. Es ist in dem Graph der folgenden Abbildungen der Weg F→B gesucht. In der oberen Abbildung wurde als Weglängenmaß Entfernung gewählt, in der unteren die Knotenanzahl. Hier ist zu sehen, dass das Weglängenmaß in einen unterschiedlichen Weg resultiert. Entfernung: F→C→A→B, Knotenanzahl: F→G→B.



#### Beispiel 4:

Zuletzt ein Beispiel in dem das Programm auf einen etwas größeren Graph angewendet wurde. An diesem Beispiel ist gut zu erkennen, wie sich der kürzeste Weg zwischen zwei Knoten durch das Linksabbiegeverbot ändert. In diesem Beispiel war der kürzeste Weg von Knoten 1/1 zu 51/51 gesucht. Oben befindet sich das Ergebnis, der kürzeste Weg, mit Linksabbiegen und unten das Ergebnis ohne Linksabbiegen.

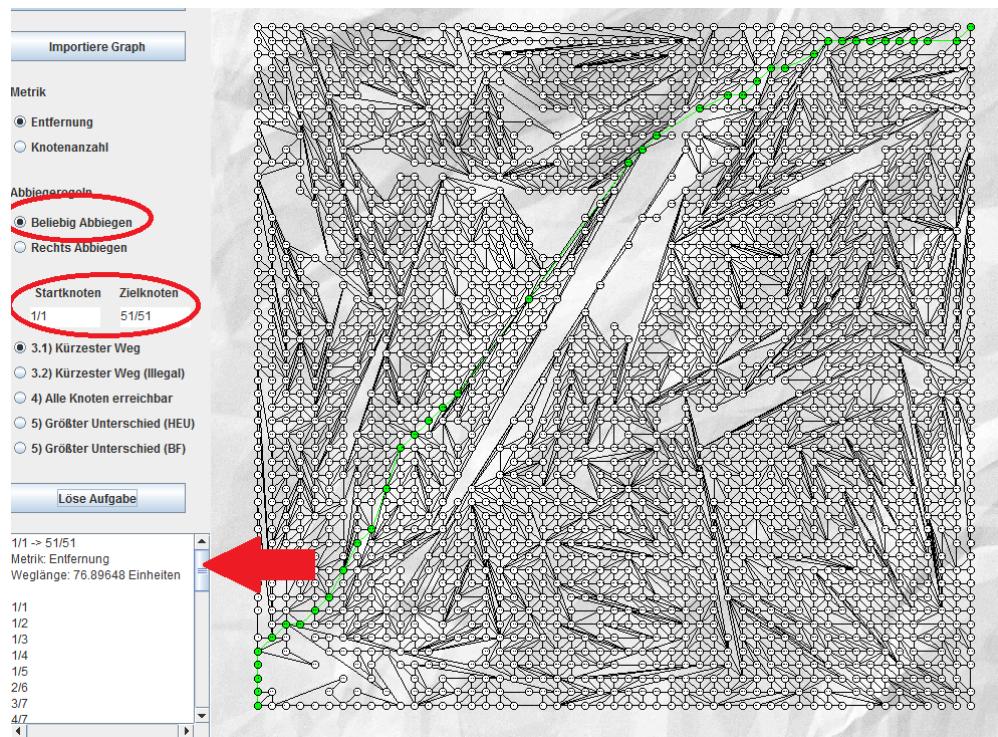


Abbildung 23 (Graph liegt bei)

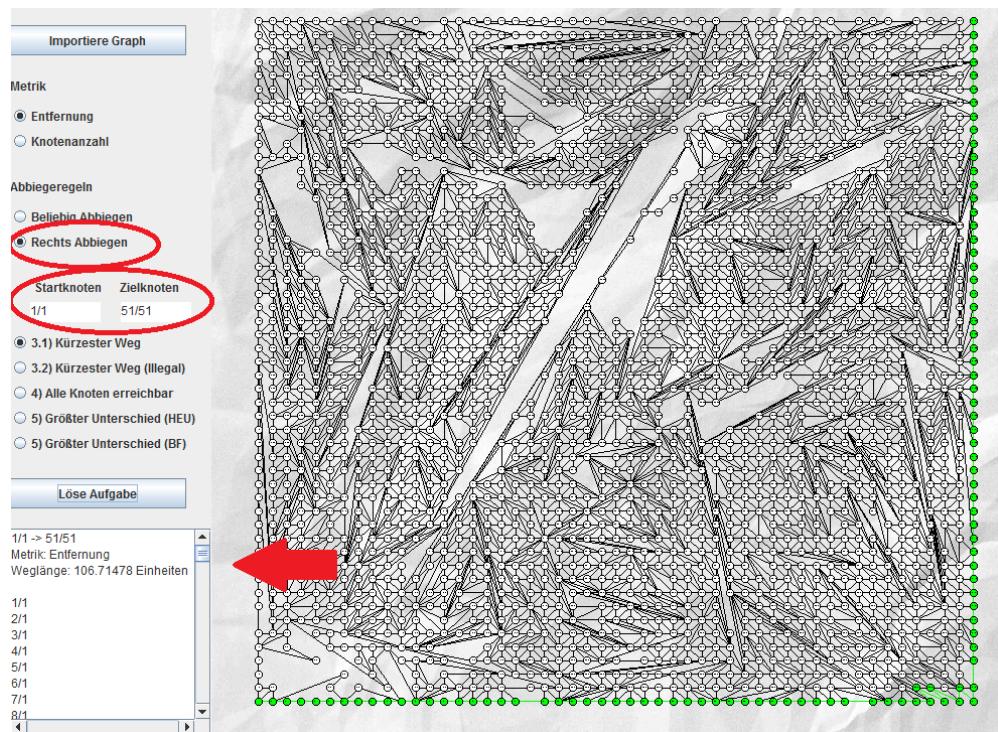


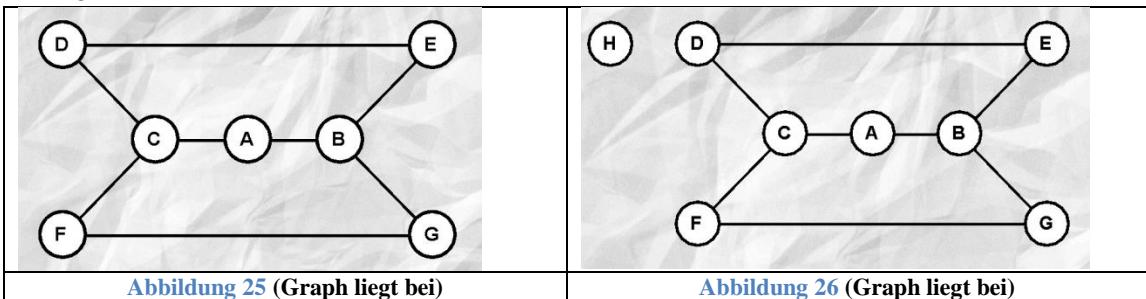
Abbildung 24 (Graph liegt bei)

## 5. Aufgabenteil 4: Sind alle Knoten gegenseitig erreichbar

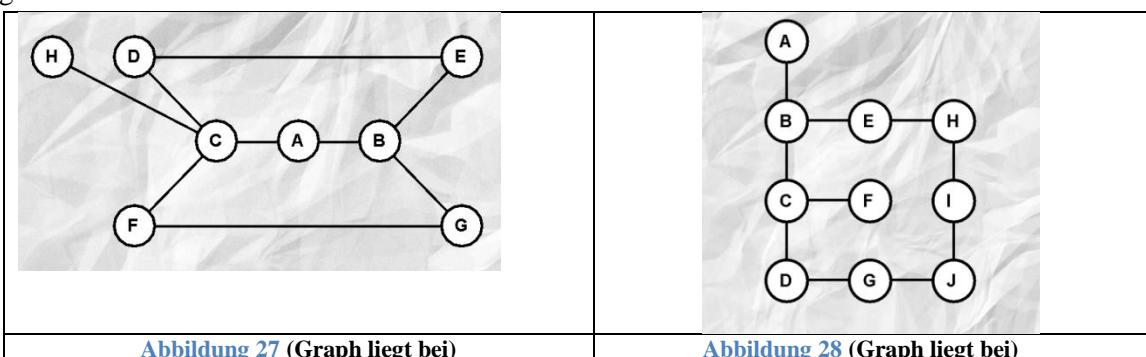
### 5.1. Vorüberlegungen

#### 5.1.1. – Gibt es nicht offensichtliche Szenarien?

Zunächst scheint es so, als ob die Aufgabe lösbar wäre, indem man überprüft, ob der ungerichtete Rechtsabbiegegraph zusammenhängend ist. Folgende Abbildung zeigt zwei Beispiele bei welchem diese Vorgehensweise funktionieren würde.



Die in den Abbildungen gezeigten Graphen können jedoch so angepasst werden, dass die Vermutung nicht mehr stimmt und das Ergebnis anders bestimmt werden muss. Hier, in Abbildung 27, ist Knoten H zwar zusammenhängend mit dem restlichen Graphen, jedoch ist man nicht in der Lage mit einem Linksabbiegeverbot Knoten A von Knoten H zu erreichen. Auch ist es nicht möglich von A aus H zu erreichen.



Die Abbildung 28 zeigt einen Graphen mit einem ebenfalls nicht offensichtlich auftretenden Fall. In diesem Beispiel kann man von Knoten F aus A erreichen jedoch nicht andersrum. Beim Entwickeln eines Verfahrens sollten diese Sonderfälle auch mit einberechnet werden.

### 5.1.2. – Vorbereitung

In diesem Aufgabenteil wird wenn nicht anders erwähnt immer der Rechtsabbiegegraph gemeint sein, der nur die erlaubten Fahrwege enthält (siehe 3.5.1). Die Umwandlung wird auch hier der erste Schritt des Algorithmus sein.

## 5.2. Lösungsansatz mittels Aufgabenteil 3 – verworfen

Zunächst scheint es naheliegend, das Programm insofern zu erweitern, dass man den kürzesten Weg Algorithmus, welcher in Aufgabenteil 3 erarbeitet wurde, auf alle Knotenpaare anwendet, da dieser unter anderem eine Aussage darüber liefert, ob sich zwei Knoten gegenseitig erreichen können. Diese Laufzeit würde aber  $O(N^2) * O(\text{Aufgabenteil3})$  betragen. Es wird aber ein deutlich schnellerer Lösungsansatz verwendet.

## 5.3. Grundidee

Im Grunde wird überprüft, ob von jeder Kreuzung jede andere Kreuzung, ohne Linksabbiegen erreichbar ist. Es wird geschaut, ob von jeder Kreuzung mindestens ein Unterknoten über einen der Unterknoten des Startknotens erreichbar ist. Ist dies nicht der Fall wird abgebrochen und das Knotenpaar zurückgegeben. Der Vorteil gegenüber 5.2 ist, dass hier eine Breitensuche von einem Knoten zu allen anderen Knoten ausgeführt wird, wohingegen in 5.2 eine neue Breitensuche für jeden Zielknoten gestartet wurde. Es ist nicht sinnvoll alle nicht erreichbaren Knotenpaare zurückzugeben, da nicht zusammenhängende Graphen mindestens Knotenzahl-1 Paare enthalten.

### 5.3.1. Verbesserung 1 – Zusammenfassung von Zyklen

Die Idee ist es Knoten von Zyklen zu einem einzigen Knoten zusammenzufassen und danach nur noch mit diesem zu arbeiten. Dies ist möglich, weil sobald man in der Lage ist, einen Knoten eines Zyklus zu erreichen, es garantiert ist, alle anderen Knoten desselben Zyklus ebenfalls zu erreichen.

Der Rechtsabbiegegraph besteht aus mehreren zusammenhängenden Teilgraphen. Für jeden dieser Teilgraphen, der aus mindestens zwei Knoten besteht, gilt:

Da der Knoten ein- und ausgehende Kanten im Teilgraph besitzt (Selbst bei Sackgassen kann die Kreuzung wieder verlassen werden. Siehe Beispiele Aufgabenteil 2), muss mindestens ein Zyklus existieren. Ein Autofahrer in Rechthausen kann jede Kreuzung, egal wie er sie befährt, zu einer anderen Kreuzung wieder verlassen. Da es aber nur endlich viele Kreuzungen gibt, muss irgendwann eine Kreuzung zum zweiten Mal besucht werden, was bedeutet, dass mindestens ein Zyklus im Teilgraph existiert. Genaugenommen liegt jeder Unterknoten, der mit mindestens einem anderen Unterknoten zusammenhängt, in einem Zyklus, da beim Verlassen der Kreuzung und immer Rechtsabbiegen irgendwann wieder zu dem Knoten zurückgekehrt werden kann. Dies kann wie folgt bewiesen werden. Würde man aus dem Rechtsabbiegegraph alle Kanten entfernen, die Geradeausfahren repräsentieren, dann hätte jeder Knoten exakt eine eingehende und ausgehende Kante und da mindestens ein Zyklus existiert, hängt dieser dann nur mit Knoten im selben Zyklus zusammen. Folglich befinden sich dann alle zusammenhängenden Knoten im Rechtsabbiegegraph ohne Geradeauskanten in einem Zyklus. Gilt dies für einen solchen Graphen, dann gilt dies erst recht für den Rechtsabbiegegraphen mit zusätzlichen Geradeauskanten.

Definition: Alle Knoten, die sich gegenseitig erreichen können (größtmöglicher Zyklus), werden zu einem neuen Knoten zusammengefasst. Dieser zusammengefasste Knoten wird jetzt **Knotengruppe** genannt.

Der Graph wird durch das Zusammenfassen zu Knotengruppen verkleinert und kann so schneller bearbeitet werden. Folgend werden nicht nur die zusammengefassten Knoten, sondern alle Knoten, nach diesem Schritt als Knotengruppen bezeichnet.

### 5.3.2. Verbesserung 2 – Zwischenergebnisse der Breitensuche speichern

Es wird zwar für jede Kreuzung überprüft, ob von all ihren Unterknoten mindestens ein Unterknoten aller anderen Kreuzungen erreichbar ist, jedoch kann es vorkommen, dass von einem Unterknoten der Kreuzung noch nicht alle anderen Kreuzungen erreichbar sind, sondern erst von der Gesamtheit der Unterknoten. Deshalb wird von jedem dieser Unterknoten eine Breitensuche ausgeführt. Um dies zu verbessern speichert jede Breitensuche ihre erreichten Knoten. Stößt man dann in einer anderen Breitensuche auf einen Knoten, von welchem aus schon gesucht wurde, können anstelle vom Weitersuchen, die gespeicherten Ergebnisse verwendet werden. Hierbei ist erwähnenswert, dass die Ergebnisse nicht für jeden Unterknoten gespeichert werden, sondern nur für die Knotengruppe, die bei der Breitensuche als normaler Knoten interpretiert wird.

### 5.3.3. Verbesserung 3 – Gruppenkombinationen speichern

Pro Kreuzung wird eine Breitensuche von jeder ihrer Unterknoten, die jeweils Teil einer Knotengruppe sind, gestartet. Es kann also passieren, dass die Unterknoten zweier Kreuzungen zu den gleichen Knotengruppen zusammengefasst wurden, also die Breitensuchen bei denselben Knotengruppen starten würden. Es kann sich demnach gemerkt werden, welche Kombinationen an Knotengruppen es ermöglichen, alle anderen Kreuzungen zu erreichen. Wenn man also erkennt, dass die Kombinationen an Knotengruppen von denen aus Breitensuchen starten würden, bereits abgesucht worden sind, muss dieser Vorgang nicht nochmal wiederholt werden.

## 5.4. Zyklen zusammenfassen

Das Zusammenfassen von allen Zyklen in einem gerichteten Graphen lässt sich in zwei Schritte unterteilen. Es muss zuerst ein Zyklus entdeckt und danach vereinfacht werden.

Ein Zyklus bestehend aus  $n$  Knoten  $V_1, \dots, V_n$  lässt sich zusammenfassen in dem man einen neuen Knoten erstellt oder auf einem dieser Knoten  $V_i$  vereinigt. Ich habe mich hier für das Vereinigen entschieden. Der Knoten  $V_i$  erhält eine Liste aller anderen in ihm vereinigten Knoten. Die Kanten der nun verschwundenen Knoten werden auf den neuen Knoten verschoben, so dass  $V_i$  nun zu jedem Zielknoten  $V_z$  eine ausgehende Kante hat, die irgendeiner der Knoten  $V_1-V_n$  hatte. Selbiges gilt für die eingehenden Kanten.

Im Folgenden ist ein Beispiel zu sehen wie ein Zyklus zusammengefasst werden kann. In diesem Beispiel bilden Knoten B, C und D einen Zyklus. Diese können nun zu einem Knoten vereinfacht werden. Dieser Knoten besitzt dann alle Kanten, verbunden mit allen Kanten der Ursprungsknoten. Also eine Kante kommend von A und gerichtet auf E.

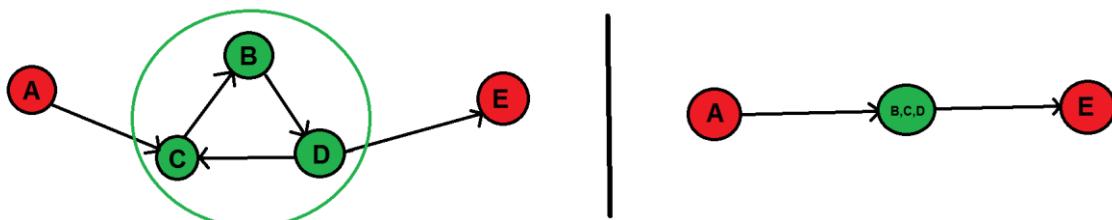


Abbildung 29

Zyklen werden mittels einer Tiefensuche entdeckt. Findet man während der Tiefensuche einen Knoten der bereits in der gleichen Tiefensuche entdeckt wurde, so existiert ein Zyklus. Der Zyklus kann bestimmt werden in dem für jeden Knoten der Tiefensuche gespeichert wird über welchen dieser aufgerufen wurde.

Alle Knoten während der Tiefensuche sind drei Stadien zuzuweisen.

1. Ein Knoten kann bereits entdeckt worden sein, jedoch wurde er noch nicht als nicht zu einem Zyklus gehörend identifiziert. Hat ein Knoten der gerade von der Tiefensuche aufgerufen wurde, einen folgenden Knoten in diesem Stadium, wurde ein Zyklus identifiziert.
2. Auch kann ein Knoten noch gar nicht von der Tiefensuche entdeckt worden sein.
3. Zuletzt kann ein Knoten zu keinem Zyklus gehörend klassifiziert werden. In diese Phase gerät der Knoten, wenn durch die Tiefensuche keiner seiner ausgehenden Knoten einen Weg zurück zum Knoten gefunden hat. Würde ein Knoten einen eindeutig nicht zu einem Zyklus gehörenden Knoten aufrufen, kann dieser einfach ignoriert werden, da von ihm kein Weg zum Knoten, der ihn aufrufen würde, gefunden werden kann.

Nachdem eine Tiefensuche ausgehend von einem Knoten ausgeführt wurde, können immer noch Zyklen im Graphen existieren. Es werden also solange Tiefensuchen gestartet, bis alle Knoten des Graphen (auch die neu Entstandenen durch das Zusammenfassen) als nicht zu einem Zyklus gehörend kategorisiert wurden. Weitere Tiefensuchen sollten aber nur von Knoten starten, die noch zu einem Zyklus gehören könnten also sich nicht in Stadium 3 befinden.

Damit Informationen aus vorherigen Tiefensuchen nicht verschwendet werden, werden alle Knoten gespeichert, die sich in Stadium 3 befinden.

Im Folgenden ein Beispiel, das visualisiert, wie dieses Verfahren funktioniert.

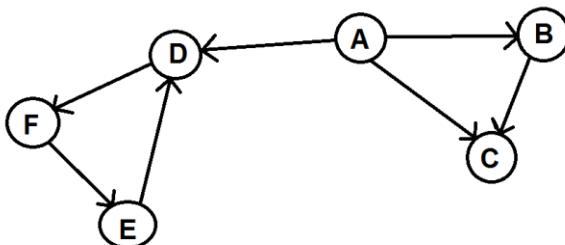


Abbildung 30

In diesem Graphen befindet sich ein Zyklus bestehend aus den Knoten D, F und E.

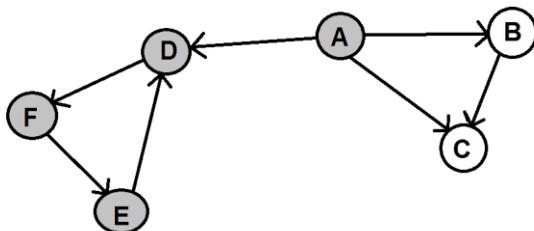


Abbildung 31

Die Tiefensuche beginnend bei A ruft Knoten D auf welcher F aufruft, der E aufruft. Im nächsten Schritt würde Knoten E D aufrufen. Hier grau markiert sind die Knoten, die sich gerade in Stadium 1 befinden. Da Knoten D sich bereits in Stadium 1 befindet, muss ein Zyklus existieren. Nun wird

die Aufrufliste zurückverfolgt. Knoten D wurde von E aufgerufen, welcher von F aufgerufen wurde und F wurde von D aufgerufen. Diese Knoten können nun zu einem Zyklus zusammengefasst werden. Der daraus entstandene Graph sieht wie folgt aus.

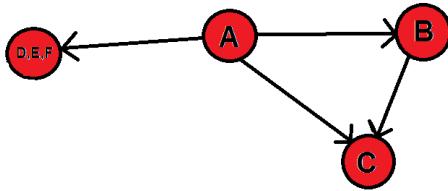


Abbildung 32

In diesem Fall ist ein nicht zyklischer Graph entstanden – aber das weiß der Algorithmus noch nicht. Es könnte immer noch der Fall sein, dass Zyklen im Graphen existieren. In diesem speziellen Fall können keine Informationen aus der vorherigen Tiefensuche übernommen werden, da kein Knoten in Stadium 3 verschoben wurde. Die Tiefensuche beginnt erneut bei einem Knoten. In diesem Falle ist es wieder Knoten A. A beginnt die Suche indem er den neuen Knoten „D, E, F“ aufruft. Da an Knoten „D, E, F“ kein weiterer ausgehender Knoten angrenzt, kann sich kein Folgeknoten im Stadium 1 oder 2 befinden, welches die Voraussetzung für das Weitersuchen an einem Knoten ist. Somit kann „D, E, F“ als nicht zu einem Zyklus gehörend identifiziert werden. Auf der Abbildung werden Knoten in Stadium 3 rot markiert. Knoten A sucht die nächste Kante ab und ruft Knoten B auf, welcher C aufruft. C kann keinen Knoten mehr aufrufen und wird daher in Stadium 3 verschoben. Darauf folgt dann, dass nun Knoten B keinen Knoten mehr aufrufen kann, und wird nun ebenfalls in Stadium 3 verschoben. Die letzte Möglichkeit der Tiefensuche ist nun, dass Knoten A seine letzte Kante untersucht. C ist aber bereits in Stadium 3 und daher ist die Tiefensuche beendet. Da alle Knoten sich nun in Stadium 3 befinden, wurden alle Knoten des Graphen erfolgreich entfernt und der Algorithmus terminiert.

## 5.5. Programmablauf

Im Folgenden ist ein Struktogramm des gesamten Programmablaufs zu sehen

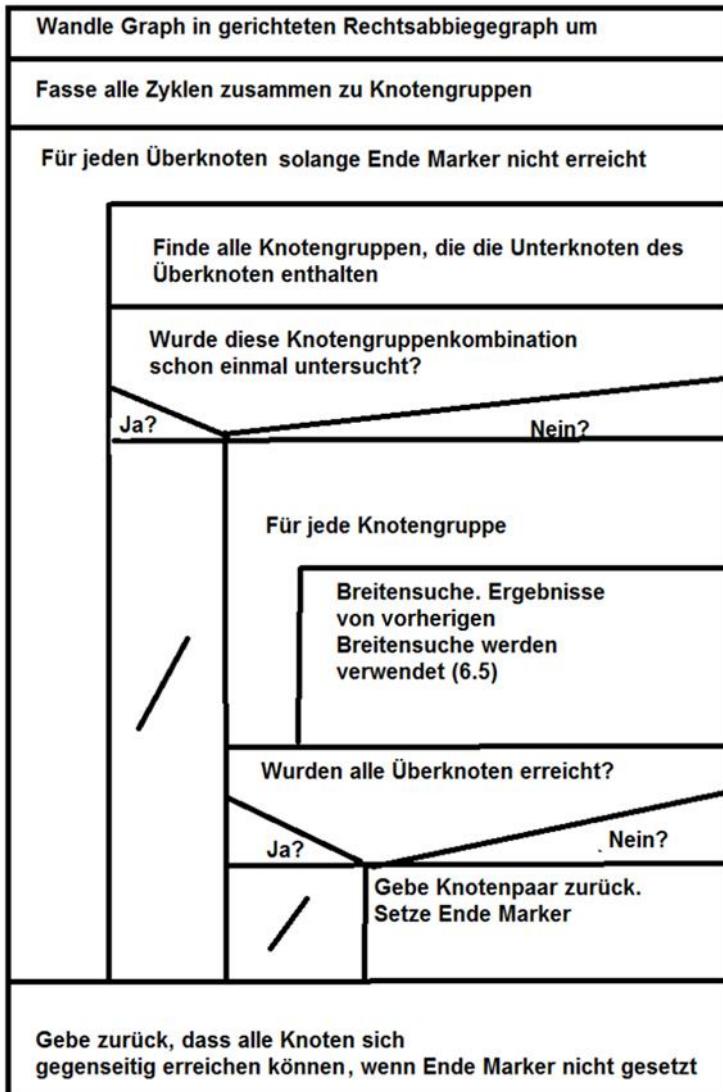


Abbildung 33

## 5.6. Laufzeit

Das Umwandeln in einen Rechtsabbiegegraphen kann zunächst mit einer linearen Laufzeit von  $O(N)$  beschrieben werden (siehe 3.5.4).

Zusammenfassen von Zyklen: Bei einem Graphen  $G$  mit  $|V|$  Knoten und  $|E|$  Kanten wird das Finden eines Zyklus mit  $O(|V|+|E|)$  abgeschätzt, weil schlimmstenfalls alle Knoten und Kanten einmal besucht werden. Beim Zusammenfassen des Zyklus verringert sich die Anzahl der Knoten um (Zykluslänge-1) und die Anzahl der Kanten mindestens um die Zykluslänge. Schlimmstenfalls – falls der Zyklus nur aus zwei Knoten besteht – verringert sich der Graph auch nur um einen Knoten und zwei Kanten. Der schlimmste Fall beinhaltet auch, dass bei der Tiefensuche kein Knoten in Stadium 3 verschoben wird. Somit schätzt ich eine Worst-Case Laufzeit von  $O((|E|+|V|)^2) \leq O(N^2)$ , wenn man  $|V|-1$  mal einen minimalen Zyklus entfernt.

Die Breitensuchen hängen von der Anzahl  $|V_G|$  der übriggebliebenen Knotengruppen und der Anzahl  $|E_G|$  der noch verbliebenen Kanten ab. Die erste Breitensuche muss zumindest alle Knotengruppen des Graphen untersuchen, daher beträgt dessen Laufzeit  $O(|V_G| + |E_G|) = O(N_G)$ . Für jede Breitensuche muss immer ein Knoten weniger in den folgenden Breitensuchen untersucht werden, da nach jeder Breitensuche sich von einer Knotengruppe gemerkt wird welche anderen Knotengruppen diese erreichen kann. Die Laufzeit kann dadurch hier mit  $O(N_G + (N_G - 1) + \dots + 1) = O(N_G^2)$  beschrieben werden.

Die Laufzeit der Algorithmen angewendet auf den Rechtsabbiegegraphen skaliert genauso wie die des Originalgraphen, da der Zuwachs an Knoten beim Rechtsabbiegegraphen mit wachsender Graphengröße linear ist. Die Gesamlaufzeit beträgt also  $O(N + N^2 + N^2) = O(N^2)$ .

Hier bestätigen die Messreihen diese Vermutung (siehe Kapitel 9 – Darstellung: blaue Raute).

## 5.7. Speicherverbrauch

Wir betrachten den Rechtsabbiegegraph  $G_R$  mit  $|V_R|$  Knoten und  $|E_R|$  Kanten zum Originalgraphen  $G_O$  mit  $|V_O|$  Knoten und  $|E_O|$  Kanten.

Zunächst werden Knotengruppenkombinationen gespeichert von denen aus alle Knoten erreichbar sind. Alle Unterknoten eines Überknoten sind auf verschiedene Knotengruppen verteilt. Pro Überknoten ergibt sich also eine Knotengruppenkombination. Im schlechtesten Fall betrüge der Speicherverbrauch hier  $O(|V_O|)$ , wenn alle Überknoten unterschiedliche Knotengruppenkombinationen bilden und all diese auch alle Knoten erreichen können.

Der meiste Speicher dadurch verbraucht, dass nach jeder Breitensuche gespeichert wird, welche anderen Knotengruppen eine Knotengruppe erreichen kann. Im schlechtesten Fall würde jede Knotengruppe jede andere Knotengruppe erreichen, wodurch sich ein Speicherverbrauch von  $O(|V_R|^2)$  ergibt.

Der gesamte Speicherverbrauch beträgt also  $O(|V_O| + |V_R|^2) = O(|V_R|^2)$ .

Hier zeigt sich eine Abweichung von der Vorhersage zu den tatsächlich gemessenen Speicherverbräuchen, da diese augenscheinlich linear und nicht quadratisch sind (siehe Kapitel 9 – Darstellung: blaue Raute). Diese Abweichung ist wahrscheinlich damit zu erklären, dass der angesprochene Worst-Case, dass alle Knotengruppen quasi alle anderen Knotengruppen erreichen können und dies auch speichern, so gut wie nie eintritt. An einigen Beispielen habe ich mir die Anzahl der Knotengruppen anzeigen lassen und dabei festgestellt, dass fast alle nur eine kleine Anzahl an Knotengruppen im Verhältnis zur Graphengröße speichern mussten, wohingegen nur eine Handvoll Knotengruppen die vermutete Worst-Case Bedingung erfüllen.

## 5.8. Beispiele

### 5.8.1. Beispiel 1 – Aufgabenstellungsgraph

Verwendet wird der Rechtsabbiegegraph aus 3.5.2. Der erste Schritt ist das Zusammenfassen von Zyklen. Die Reihenfolge der aufeinanderfolgenden Zusammenfassungen ist nicht definiert, aber das Endergebnis ist immer dasselbe. Zur Visualisierung verwende ich für den ersten Schritt die gleichfarbig markierten Unterknoten.

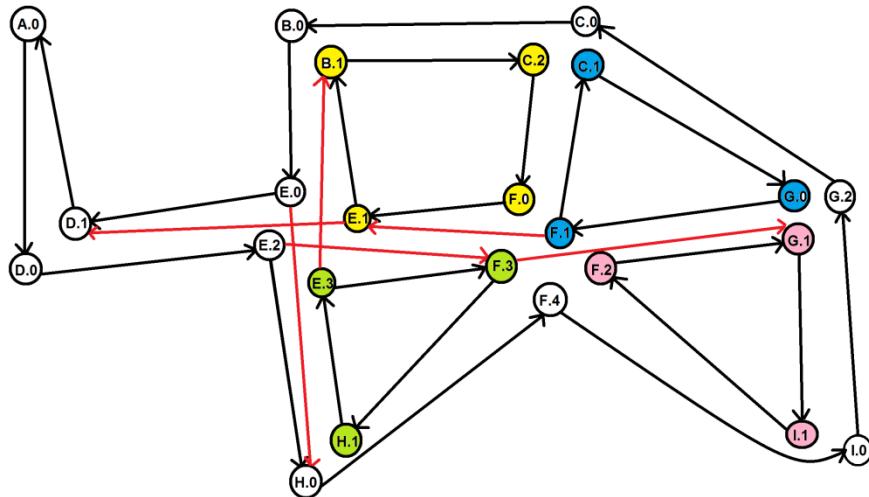


Abbildung 34

Dies führt zu folgendem Graphen, der weiter vereinfacht werden kann, weil sich hier noch ein Zyklus befindet.

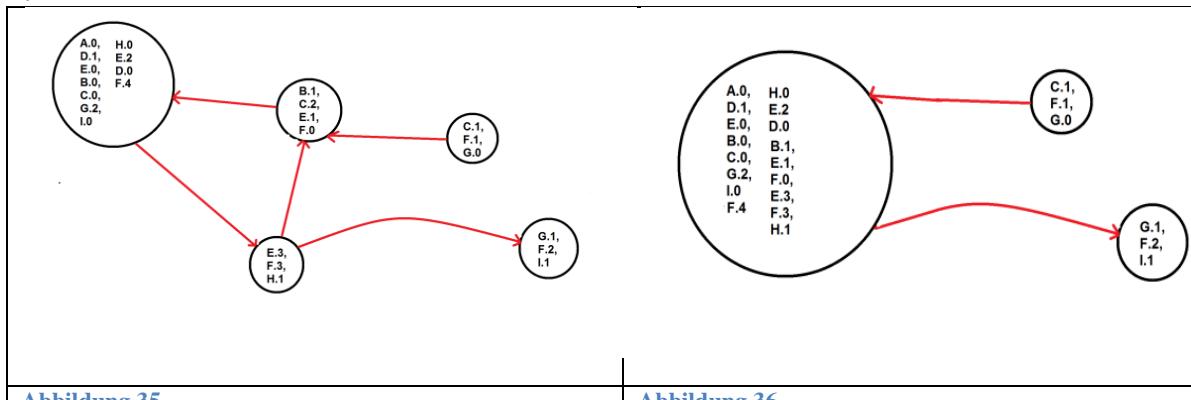


Abbildung 35

Abbildung 36

Die Besonderheit hier ist, dass alle Knoten schon einen Unterknoten in der größten Knotengruppe haben, so dass hier alle Breitensuchen, durch das Finden dieser größten Knotengruppe, automatisch alle Überknoten erreichen.

### 5.8.2. Beispiel 2 – Nicht alle Knoten erreichbar

Es wird der Graph aus 6.1.1 verwendet und in Gruppen zusammengefasst.

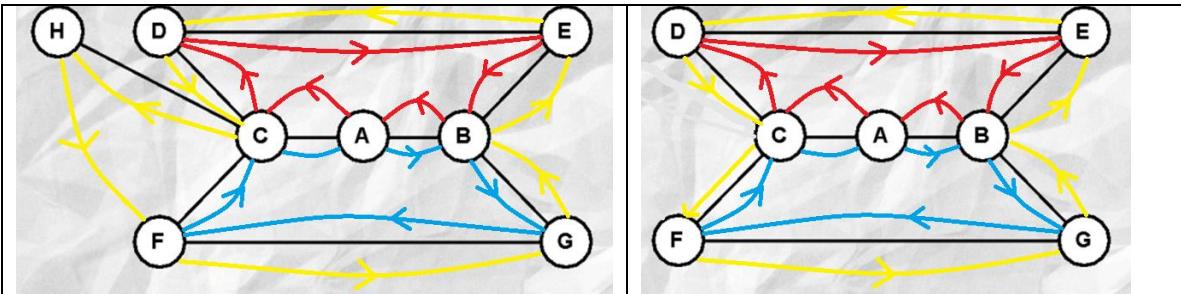
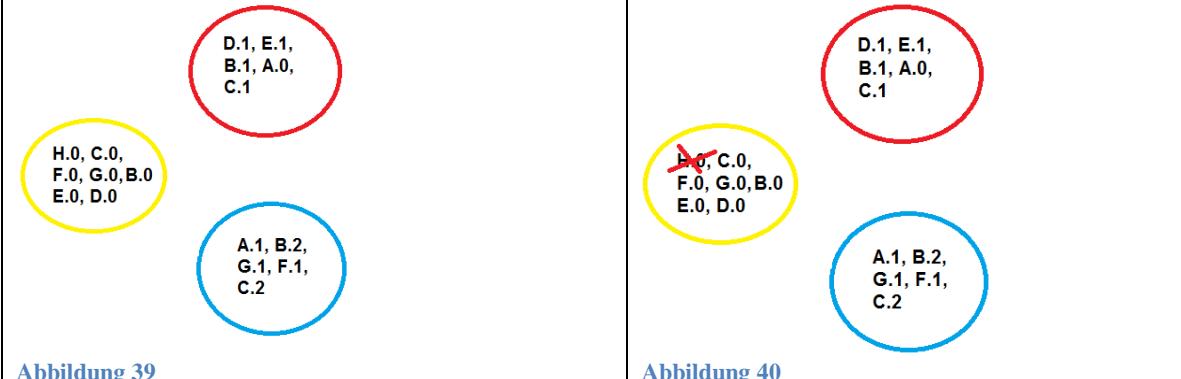


Abbildung 37 (Graph liegt bei)

Abbildung 38 (Graph liegt bei)



Die Gruppen sind die Unterknoten, die von den farbig markierten Pfeilen zusammengefasst werden. Da keine Geradeausstrecken möglich sind, ergeben sich jeweils 3, nicht miteinander verbundene, Knotengruppen.

Im linken Beispiel sieht man, dass die Knoten H und A sich nicht gegenseitig erreichen können. Knoten H ist nur in der gelben Knotengruppe vorhanden, die Unterknoten von A jedoch nur in der roten und der blauen Knotengruppe. Da die Gruppen nicht zusammenhängen, findet die Breitensuche von allen Unterknoten von A keinen Unterknoten von H und umgekehrt. Für alle anderen Knotenkombinationen  $V_i$  und  $V_j$  findet sich eine der Knotengruppen, wo sowohl ein Unterknoten  $V_i$  als auch ein Unterknoten von  $V_j$  zu finden ist.

Im Beispiel auf der rechten Seite können sich alle Knoten gegenseitig erreichen. Die Unterknoten jedes Überknotens sind so in den Knotengruppen verteilt, dass jeder von ihnen mindestens einen anderen Unterknoten jedes Überknoten erreichen kann.

Das Verfahren wurde an allen Beispielgraphen des Forums ausprobiert – das Beispiel der Homepage ist oben abgehandelt.

| Hochgeladen von | Dateiname        | Alle Knoten erreichbar            |
|-----------------|------------------|-----------------------------------|
| Fabian Märkert  | rechthausen3.txt | Nein – Bsp. 8587 → 6854           |
| Fabian Michel   | beispiel1.txt    | Nein – Bsp. P→V                   |
| Fabian Michel   | beispiel2.txt    | Nein – Bsp. B→k                   |
| Fabian Michel   | beispiel3.txt    | Nein – Bsp. au→ct                 |
| Leon Windheuser | U-turn.txt       | Ja                                |
| Robin Schmöcker | Alle             | Ja, weil absichtlich so generiert |

## 6. Aufgabenteil 5: Größter Unterschied

### 6.1.1. Vorüberlegung

Zunächst, unabhängig von später gewählten Verfahren, ist zu beachten, dass bei bestimmten Graphen schon im Vorhinein ausgesagt werden kann, dass der gesuchte Faktor unendlich ist, nämlich dann, wenn ein Knoten einen anderen nicht mehr erreichen kann, obwohl dies ohne das Linksabbiegeverbot möglich war. Solch ein Knotenpaar kann nur in einem zusammenhängenden Graphen auftreten. Daher wird das Straßennetz, um solch ein potentielles Knotenpaar zu finden, in einzelne zusammenhängende Graphen unterteilt. Nun muss lediglich überprüft werden, ob in einem der Teilgraphen mit Linksabbiegeverbot ein Knoten einen anderen nicht mehr erreichen kann. Dies kann mit der Lösung der vorherigen Aufgabe überprüft werden.

Weiterhin ist es auch möglich, dass Knoten nicht in einem zusammenhängenden Graphen liegen. Für Knoten, die sich also weder im Rechtsabbiegegraph noch im Beliebigabbiegegraph erreichen ändert sich also nichts. Der Faktor dieses Falls wird mit 1 definiert wie bei allen anderen Knotenpaaren, die dieselbe Entfernung voneinander im Beliebigabbiegegraph und Rechtsabbiegegraph haben.

Nach längeren Recherchen bin ich auf den Warshall-Floyd Algorithmus gestoßen, der bei geringer Anpassung eine exakte Lösung für das Problem liefern würde, jedoch ist bei der Implementierung das Problem aufgetreten, dass die Laufzeit (Dreifach geschachtelte Schleife) sowie der Speicherverbrauch (Quadratische Matrix) nicht tragbar sind, weshalb eine andere Lösung des Problems gesucht ist. Für kleinere Graphen funktioniert folgende Methode.

### 6.1.2. Brute-Force mit Dijkstra

Für jede Kreuzung wird mittels des Dijkstra Algorithmus der schnellste Weg zu allen anderen Kreuzungen mit und ohne Linksabbiegeverbot gefunden. Danach wird für jeden der gefundenen Knoten der gesuchte Faktor gebildet, von welchen sich der größte gemerkt wird. Dies geschieht für alle Knoten. Es ergibt sich somit eine Laufzeit von  $O(N) * O(\text{Dijkstra}) = O(N^2 \log(N))$ . Nur für kleine Graphen ist dies akzeptabel aber für größere Graphen wird ein anderer Ansatz benötigt. Wie die Laufzeit des Dijkstra Algorithmus zustande kommt, ist in Kapitel 4.2.4.1 beschrieben.

### 6.1.3. Heuristischer Ansatz

Es liegt im Ermessen des Benutzers, an seiner Hardware und an der Größe des Graphen, ob dieser die genaue, aber zeitaufwendige Lösung wählt, oder die heuristische, aber deutlich schnellere Methode, die im Folgenden beschrieben wird.

Betrachtet man eine Reihe von zufällig erstellten Graphen und wählt aus diesen einen zufälligen Knoten, ist es auffällig, dass der Knoten mit dem größten Unterschied sich meist in unmittelbarer Nähe befindet.

Man kann dies visualisieren, indem man jeden Knoten eines Graphen abhängig von seinem Unterschiedsfaktor zu einem Knoten färbt. Im Folgenden befindet sich eine so erstellte Karte. Der grüne Punkt repräsentiert den Knoten von welchem aus die Unterschiede bestimmt werden. Je schwärzer eine Fläche ist, desto geringer der Unterschied, je röter, desto größer der Unterschied. Der gelb eingekreiste Bereich enthält den Ursprungsknoten und Knoten in unmittelbarer Nähe.

Es ist hier zu erkennen, dass je weiter man sich vom Startpunkt entfernt, die Farbe immer dunkler wird, also der Faktor geringer wird je weiter man vom Ursprung entfernt ist. Die Knoten mit dem größten Faktor jedoch befinden sich in der Nähe des Ursprungsknoten.

Diese Erkenntnis könnte sich,- wie folgt, begründen lassen. Der Faktor setzt sich aus dem Teilen der Weglänge mit Linksabbiegeverbot durch die Weglänge ohne Verbot zusammen. Je weiter ein Knoten also vom Ursprungsknoten entfernt ist, gemessen an der Weglänge ohne Verbot, desto größer muss die Weglänge mit Verbot sein um den Faktor um den gleichen Weg zu erhöhen.  
Die Theorie ist also, dass das gesuchte Knotenpärchen dicht bei einander liegt.

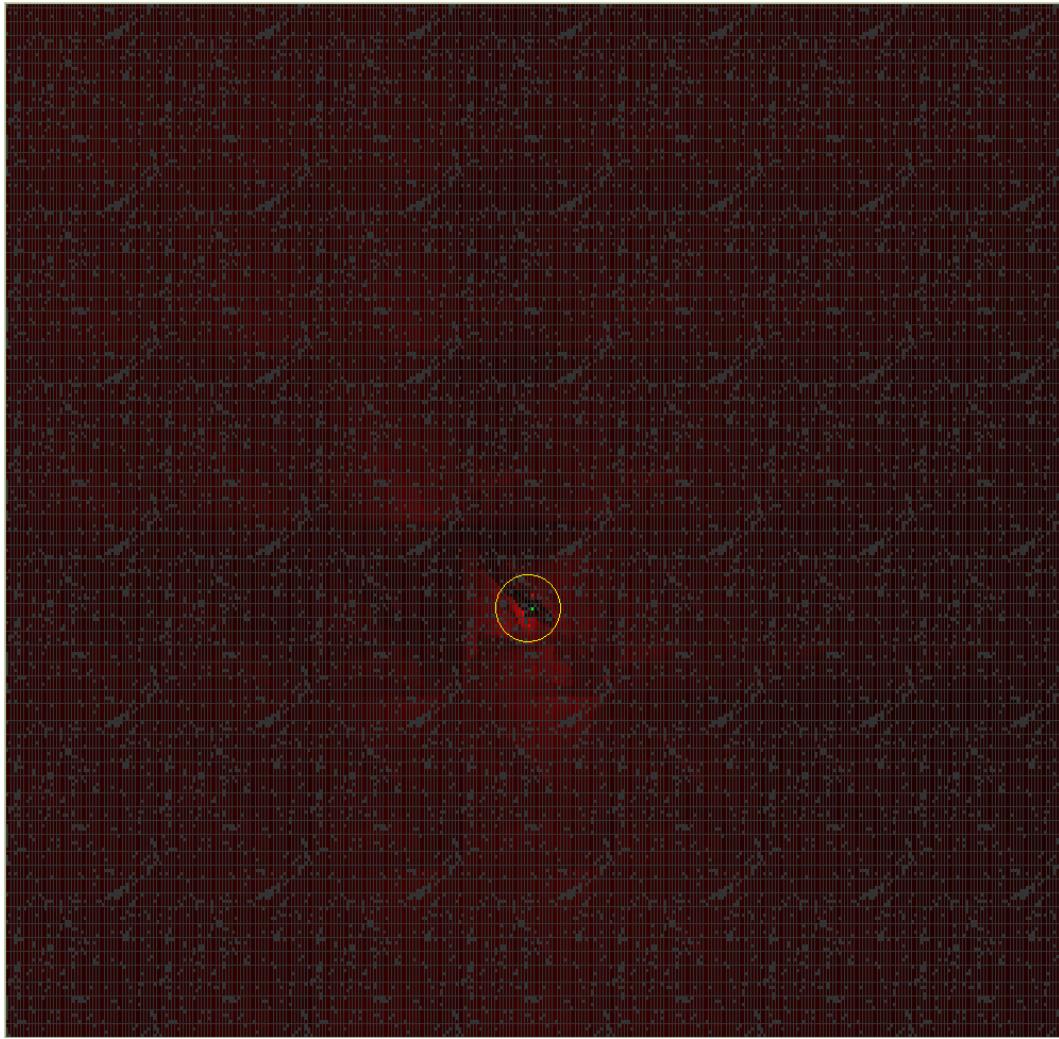


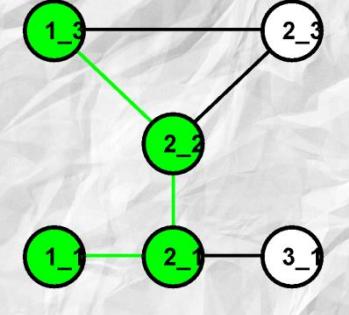
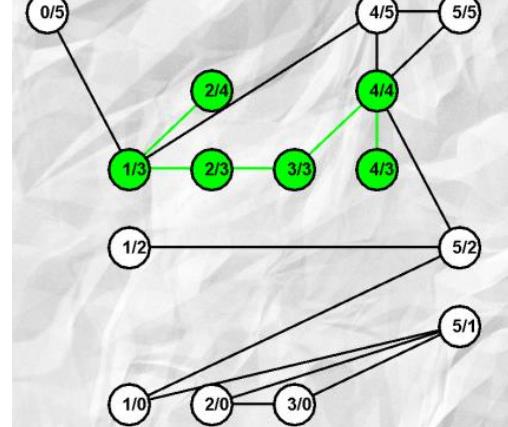
Abbildung 41 – 350x300 Graph

Demnach ist ein Verfahren gesucht, was ausgehend von jedem Knoten in dessen nahen Umkreis nach dem größten Unterschied sucht.

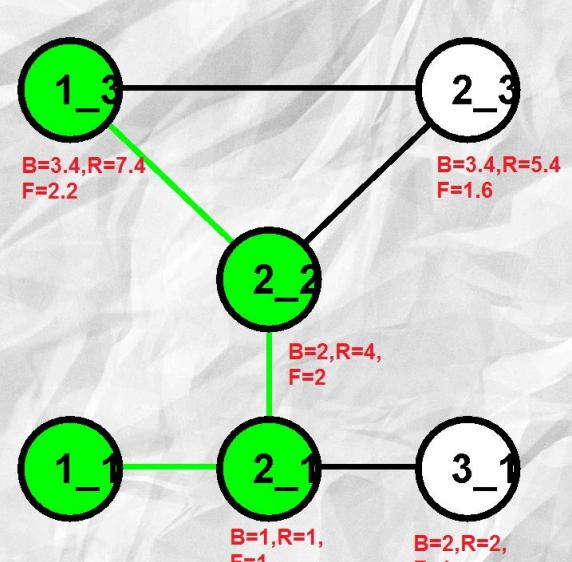
#### 6.1.4. Erste Idee Abstand 2

Die erste Vermutung war es, dass das Knotenpaar mit dem größten Faktor lediglich 2 Knoten voneinander entfernt liegt. Betrachtet man wieder eine Reihe von zufällig erstellten Graphen wird schnell deutlich, dass das gesuchte Knotenpaar zwar häufig nur zwei Knoten entfernt ist, jedoch es auch dazu kommen kann, dass es auch weiter auseinander liegen kann. Im Folgenden sind

Beispiele abgebildet bei denen das Knotenpaar mit dem größten Unterschied mehr als zwei Knoten entfernt liegt (Bei Weglängenmaß Entfernung – aber es gäbe auch Beispiele mit Knotenzahl). Die folgenden beiden Beispiele zeigen solche Graphen, wobei hier der Knotenabstand 3 bzw. 5 Knoten beträgt.

|                                  |  |   |
|----------------------------------|--|---|
| Graph:                           |                             |   |
|                                  | <b>Abbildung 42 (Graph liegt bei)</b>  | <b>Abbildung 43 (Graph liegt bei)</b>   |
| Reihenfolge bei Rechtsabbiegen : | $1\_1 \rightarrow 2\_1 \rightarrow 3\_1 \rightarrow 2\_1 \rightarrow 2\_2 \rightarrow 2\_3 \rightarrow 1\_3$ | $4/3 \rightarrow 4/4 \rightarrow 5/2 \rightarrow 1/2 \rightarrow 1/0 \rightarrow 5/1 \rightarrow 2/0 \rightarrow 3/0 \rightarrow 5/1 \rightarrow 1/0 \rightarrow 5/2 \rightarrow 4/4 \rightarrow 5/5 \rightarrow 4/5 \rightarrow 1/3 \rightarrow 2/4$ |
| Faktor:                          | 2.171573   | 7.634126  |

### 6.1.5. Nächste Idee- In benachbarten Knoten nach größerem Faktor suchen (Umgesetzte Idee)

|   |  |
|---|--|
|  | B=Beliebig Abbiegen<br>R=Rechts Abbiegen<br>F=Faktor<br>-Alle Zahlen sind auf eine Stelle gerundet |
| <b>Abbildung 44 (Graph liegt bei)</b>   |  |

Mittels einer Breitensuche werden von einem Knoten alle seine Nachbarn gefunden, von welchen dann ebenfalls deren Nachbarn gefunden werden, usw. Es wird die Breitensuche an einem Knoten aber nur dann fortgeführt, wenn sein Faktor  $\geq$  dem Faktor des Vorgängers ist.

Im obigen Beispiel startet die Breitensuche bei  $1\_1$ . Im ersten Schritt wird Knoten  $2\_1$  gefunden mit dem Faktor 1. Im darauffolgenden Schritt werden die Knoten  $3\_1$  und  $2\_2$  gefunden, da ihre

Faktoren mit 1 bzw. 2 größer gleich 1 sind. Im letzten Schritt findet Knoten 2\_2 nur Knoten 1\_3, welcher einen größeren Faktor besitzt. Knoten 2\_3 wird von 2\_2 nicht gefunden, da sein Faktor mit  $1.6 < 2$  ist.

## 6.2. Umsetzung

Das Verfahren basiert darauf, dass für alle Knoten des Graphen eine lokal beschränkte Breitensuche ausgeführt wird. Der größte gefundene Faktor aller Breitensuchen wird als Ergebnis zurückgeliefert.

Um den Unterschied eines Knotens zum Startknoten während der Breitensuche effizient zu finden, wird der Dijkstra Algorithmus verwendet. Dieser findet den kürzesten Weg zum gewünschten Knoten im Rechtsabbiegegraph und im Graph ohne Linksabbiegeverbot. Aus den Weglängen kann dann der Faktor gebildet werden.

Im ersten Schritt wird für die direkten Nachbarn des Startknotens der Faktor ermittelt und die zugehörigen Knoten werden in eine Liste weiter zu untersuchender Knoten eingetragen. Der Faktor der direkten Nachbarn ist immer 1, weil vom Startknoten aus jede beliebige Straße befahren werden kann.

In den folgenden Schritten werden für jeden Knoten in der Liste der noch zu untersuchenden Knoten mit demselben Verfahren diejenigen Knoten in diese Liste eingefügt, deren Faktor größer oder gleich dem Faktor des gerade untersuchten Knotens ist. Es wird solange gesucht bis die Liste der noch zu untersuchenden Knoten leer ist.

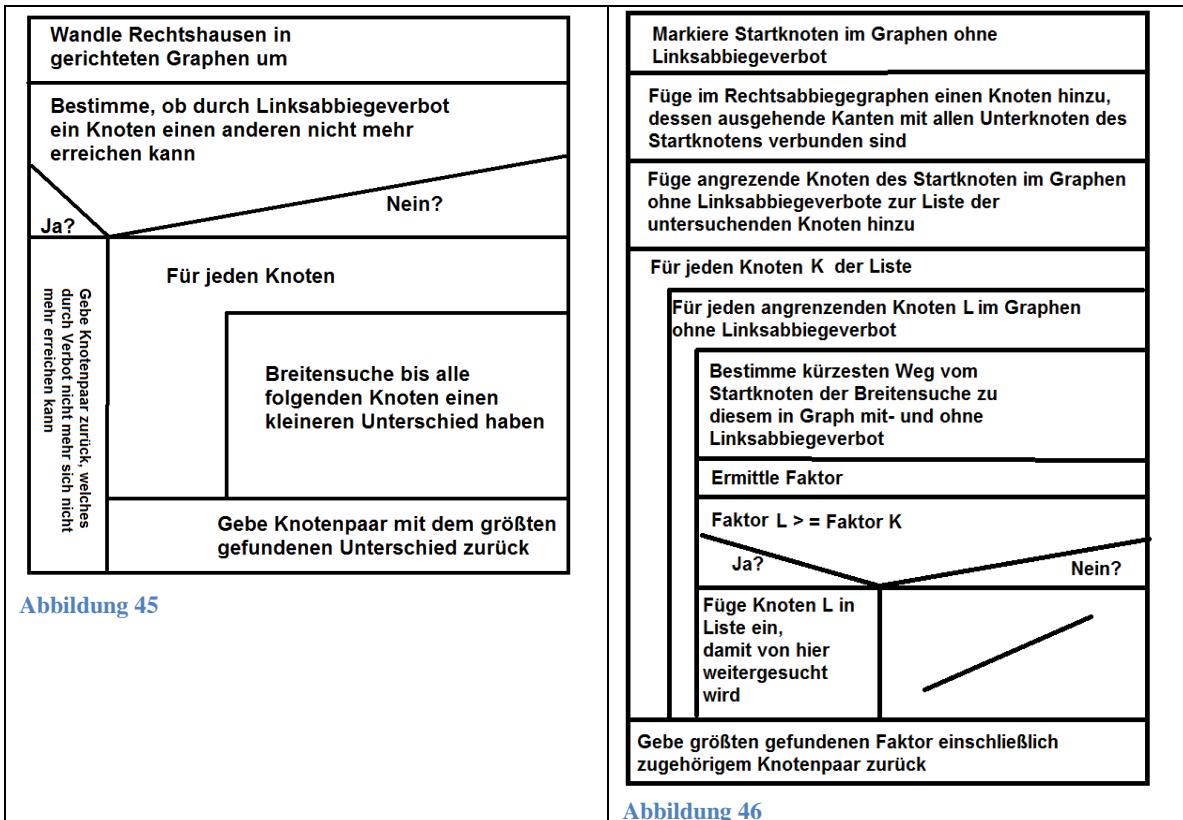
Um dies effizient zu gestalten, werden die bisherigen Ergebnisse der Dijkstra Algorithmen solange nicht verworfen, solange die Liste noch bearbeitet wird, um Daten wiederzuverwenden. Bestenfalls ist es so, dass der gesuchte Knoten schon als Nebenprodukt des Dijkstra Algorithmus gefunden wurde.

Wie auch schon im kürzesten Weg Algorithmus beschrieben wurde, startet die Suche nach dem kürzesten Weg zwischen zwei Knoten im Rechtsabbiegegraphen bei einem neu erstellten Knoten, verbunden mit allen Unterknoten des Startknoten (Siehe 4.2.1).

Hinweis: Auch dieses Verfahren ist leider nicht perfekt, da es auch hier Ausnahmen gibt, die von dem Algorithmus nicht gefunden werden wie eine große Anzahl von zufällig erzeugten Graphen leider ergeben hat. Man könnte das Verfahren erweitern, um auch diese Ausnahmen zu erschlagen, aber da es sich hier um ein heuristisches Verfahren handelt, und die gefundenen Ergebnisse immer recht nah am echten Ergebnis lagen, habe ich mich entschieden keine weiteren Ergänzungen vorzunehmen, da diese die Laufzeit dann wieder verringern würden. In Beispiel 2 wird ein solches Beispiel behandelt.

## 6.3. Programmablauf

Im Folgenden ein Struktogramm des Programmablaufes



## 6.4. Laufzeit

Die Laufzeit wird für einen Graphen G mit  $|V|$  Knoten und  $|E|$  Kanten beschrieben.

Der Optimalfall dieses Verfahrens ist es, wenn bevor der eigentliche Algorithmus startet, identifiziert wird, dass durch das Linksabbiegeverbot mindestens ein Knoten einen anderen Knoten nicht mehr erreichen kann. Die Laufzeit hierfür setzt sich aus dem Unterteilen des Graphen in einzelne zusammenhängende Teilgraphen zusammen und dem Anwenden des Aufgabenteils 4 auf diese Teilgraphen. Für diesen Fall ergibt sich daher eine Laufzeit von  $O(|V|^2)$ .

Andernfalls müssen in die Laufzeit die Breitensuchen ausgehend von jedem Knoten mit eingerechnet werden. Im schlechtesten Fall würde das Abbruchskriterium einer Breitensuche bei keinem Knoten zutreffen. Das heißt der Dijkstra Algorithmus im Rechtsabbiegegraph und im Beliebigabbiegegraph würde erst mit dem Erreichen von allen Knoten terminiert werden. Eine Breitensuche ausgehend von einem Knoten würde dann  $O(2^*|V|*\log(|V|)+|E|) = O(|V|*\log(|V|))$  betragen. Breitensuchen von allen Knoten kann dann mit  $O(|V|^2*\log(|V|))$  beschrieben werden.

Aber im besten Fall kann es dazu kommen, dass das Abbruchskriterium die Suche bei allen Knoten lokal beschränkt. Es müssen zwar trotzdem alle Knoten des Graphen iteriert werden, jedoch würde sich hier eine konstante Laufzeit für die einzelnen Breitensuchen ergeben. Hier würde sich also eine Laufzeit für die Breitensuchen von  $O(|V|^1) = O(|V|)$  ergeben.

Die tatsächliche Laufzeit liegt in dem Bereich zwischen den beiden angegebenen Laufzeiten.

Die Messreihen zeigen, dass sich die tatsächliche Laufzeit schwankend in den angegebenen Bereichen befindet. Genaugenommen scheint das Wachstum der gemessenen Laufzeit etwas mehr als linear zu sein (Siehe Kapitel 9 – Darstellung: grüne Raute).

## 6.5. Speicherverbrauch

Der Speicherverbrauch wird für einen Graphen G mit  $|V|$  Knoten und  $|E|$  Kanten beschrieben.

Der Speicherverbrauch vom ersten Schritt setzt sich aus dem Teilen des Graphen in zusammenhängende Graphen und dem Anwenden des Aufgabenteils 4 zusammen. Beim Teilen kann der Speicherverbrauch mit  $O(|V|+|E|)$  dargestellt werden, da im Grunde ein neuer Graph mit der gleichen Knoten und Kantenanzahl gebildet wird. Der zweite Teil besteht darin Aufgabenteil 4 auf diese einzelnen Graphen anzuwenden. Der Speicherverbrauch beträgt hierfür zusätzlich noch  $O(|V|^2)$  (siehe Kapitel 5). Der Speicher für den ersten Teil ist also  $O(|V|+|E|+|V|^2) = O(|V|^2)$ .

Der restliche Speicherverbrauch ergibt sich durch das Speichern des Knotenpaares mit dem größten Unterschied was gefunden wurde,  $O(1)$  und dem verbrauchten Speicher während einer Breitensuche. Da vor jeder Breitensuche keine Informationen von vorherigen Breitensuchen verwendet werden, ist nur der Speicherverbrauch durch eine Breitensuche relevant. Während einer Breitensuche werden Informationen bezüglich des Dijkstra Algorithmus für alle Knoten des Rechtsabbiegegraph und Beliebigabbiegegraph gespeichert. Im schlechtesten Fall werden in beiden Graphen durch die Breitensuche alle Knoten untersucht. Somit ergäbe sich ein Speicherverbrauch von  $O(2*|V|)$ .

Der gesamte Speicherverbrauch beträgt also  $O(|V|^2+2*|V|) = O(|V|^2)$ .

Hierzu wurde ebenfalls eine Messreihe durchgeführt (siehe Kapitel 9 – Darstellung: grüne Raute). Hier ist zu erkennen, dass die Speicherverbräuche aus den Messreihen deutlich schwanken und keine genauen Aussagen über diese getroffen werden können. Dies ist damit zu begründen, dass der Speicherverbrauch sehr von der Form des Graphen abhängig ist und möglicherweise bei gewissen Konstellationen, Breitensuchen eher lokal beschränkt sind (kleiner Speicherverbrauch) und bei manchen eher nicht (größerer Speicherverbrauch). Wenn man sich nur auf die größten Graphen bezieht, scheint sich ein Speicherverbrauch zwischen linear und quadratisch anzudeuten.

## 6.6. Beispiele

Betrachtet man Beispiele zu dieser Aufgabe muss man grundsätzlich zwischen 2 Fällen unterscheiden. Der erste Fall wäre, dass durch das Linksabbiegeverbot ein Knoten einen anderen nicht mehr erreichen kann, also ein unendlicher Faktor entsteht, wie in der Vorbemerkung schon erwähnt. Ein solcher Graph wurde bereits im letzten Aufgabenteil behandelt (siehe Abb. 27).

Der interessante Fall ist, dass durch das Verbot sich ein Weg um einen bestimmten Faktor erhöht. Im Folgenden ein Beispiel zu diesem Fall.

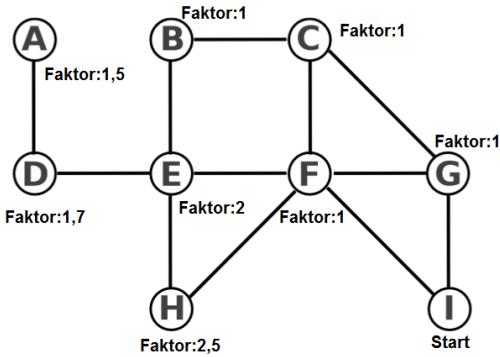
**Beispiel 1:**

Das Verfahren wird am Beispielgraphen der BwInf-Hompage beschrieben.

Als Weglängenmaß wird die Knotenanzahl gewählt. Zunächst lässt mit dem in der Vorüberlegung beschriebenen Verfahren herausfinden, dass kein Knotenpaar existiert, dass einen unendlich großen Unterschiedsfaktor aufweist.

Im Folgenden wird die Breitensuche ausgehend von einem Knoten beschrieben und der so größte gefundene Unterschied bestimmt. Das gesamte Verfahren besteht aber darin diese Suche von allen Knoten auszuführen und von denen den größten gefundenen Faktor zu wählen. Die Breitensuche startet in der Beispieldurchsuche bei Knoten I. Die ersten Knoten bei denen der Faktor gebildet wird, sind die Nachbarknoten von I, also Knoten F und G. Mittels Dijkstra Algorithmus wird von I zu ihnen der kürzeste Weg mit und ohne Linksabbiegen gefunden. Da beide Knoten an I, dem Startknoten, angrenzen, betragen die Weglängen  $I \rightarrow G$  und  $I \rightarrow F$  in beiden Graphen 1. Der Dijkstra Algorithmus stoppt sofort nachdem der jeweils gewünschte Zielknoten gefunden wurde, aber die Werte, (Knoten: Vorgänger, Entfernung, Besucht) werden noch nicht verworfen, da eventuell weitergesucht werden soll. Im folgenden Schritt sind die zu untersuchenden Knoten H, E, und C, weil diese an die gerade gefundenen Knoten F, G angrenzen. Die Dijkstra Algorithmen laufen nun mit den bisher gesammelten Informationen weiter bis auch die jeweils gesuchten Knoten gefunden werden. So ergeben sich die Faktoren der gerade erwähnten Knoten. Bei E tritt erstmalig ein Faktor ungleich 1 auf, da der Weg in Rechtsabbiegegraph länger ist als der im Originalgraphen (Rechtsabbiegegraph:  $I \rightarrow G \rightarrow C \rightarrow B \rightarrow E \rightarrow H$  (5 Knoten), Beliebigabbiegegraph:  $I \rightarrow F \rightarrow H$  (2 Knoten)) Faktor =  $5/2 = 2,5$ . Die Faktoren sind in der Abbildung zu sehen. Da alle Faktoren nicht geringer sind als der des vorherigen Knoten, welche in diesem Falle nur F und G sind, können all diese Knoten im nächsten Schritt weiter untersucht werden. An die gerade untersuchten Knoten grenzen die noch nicht untersuchten Knoten B und D an. Knoten E würde Knoten B nicht in der Liste der als nächstes zu untersuchenden Knoten eintragen, da der Faktor von B < dem von E ist. C jedoch würde B in die Liste eintragen da der Faktor von C, 1, nicht kleiner als der von B, 1, ist. Das B im nächsten Schritt untersucht wird ist aber unerheblich, da an diesem keine weiteren, noch nicht untersuchten Knoten angrenzen. D wird im nächsten Schritt nicht untersucht, da der Faktor von E größer ist als der von D ist. Dies bedeutet, dass Knoten D im nächsten Schritt Knoten A nicht untersuchen würde, und die Breitensuche an dieser Stelle beendet wird. Der während der Suche größte gefundene Faktor beträgt 2,5, nämlich der Faktor  $I \rightarrow H$ .

Wie oben bereits beschrieben, würde eine solche Breitensuche nun an allen Knoten des Graphen ausgeführt werden. Der größte gefundene Faktor aller Suchen wird ausgegeben. In diesem Falle ist der größte Faktor  $H \rightarrow D$ , welcher 3 beträgt. In den Tabellen (Abbildung 19-20) sind die Faktoren aller Knoten zu allen Knoten eingetragen. Das Knotenpaar  $H \rightarrow D$  wurde blau eingerahmt. Der Weg  $H \rightarrow D$  ohne Linksabbiegen ist:  $H \rightarrow E \rightarrow B \rightarrow C \rightarrow F \rightarrow E \rightarrow D$  mit einer Weglänge von 6 Knoten. Der Weg mit Linksabbiegen ist:  $H \rightarrow E \rightarrow D$  mit einer Weglänge von nur 2 Knoten. Der Faktor 3 kommt so zustande.



**Abbildung 47**

Anmerkung:

In diesem kleinen Beispiel wurden nahezu alle Knoten des Graphen untersucht. Der am weitesten entfernte untersuchte Knoten D von I aus, hat einen Abstand 3 Knoten, was fast dem Durchmesser des Graphen entspricht. Bei deutlich größeren Graphen hingegen sucht das Verfahren, gerade durch die Abbruchbedingungen, lediglich lokal begrenzte Bereiche ab.

### Beispiel 2:

In der Umsetzung wurde erwähnt, dass dieses Verfahren nicht zwangsläufig die perfekte Lösung findet. Dies kann dadurch passieren, dass ab einem Knoten nicht weitergesucht wird, jedoch Knoten, die auf diesen folgen doch noch einen größeren Unterschied aufweisen. Im Folgenden ist solch ein Graph abgebildet. In diesem Beispiel werden die kürzesten Wege anhand der Entfernung zweier Knoten gemessen. Den tatsächlich größten Unterschiedsfaktor bilden Knoten 0/0 und 0/2. Bei dem Lösungsverfahren, was verwendet wird, findet Knoten 0/0 als einzigen Nachbarn den Knoten 1/0 mit Faktor 1. Dessen erster Nachbar 1/1 hat keinen kleineren Faktor, weswegen von diesem zunächst weitergesucht werden kann. Dasselbe gilt für den anderen Nachbarn 4/1. Von 4/1 aus findet man wieder 1/1 welcher sich aber bereits in der Liste der weiter zu untersuchenden Knoten befindet. Man sieht, dass alle Nachbarn von 1/1 einen kleineren Faktor besitzen, so dass das Verfahren an diesem Punkt nicht weitersucht. Es ist aber zu erkennen, dass der Knoten 0/2 doch noch einen größeren Faktor gehabt hätte. Der Faktor 3.9 von Knoten 0/0 → 0/2 ist übrigens der größte Faktor ausgehend von jedem Knoten des Graphen. Hieran wird aber auch die Stärke des Verfahrens deutlich. Trotz der Tatsache, dass nicht das exakte Ergebnis bestimmt wurde, ist die Abweichung der Werte des Faktors des heuristischen Verfahrens und der exakten Lösung sehr gering ( $\Delta\text{Faktor} = 3.9 - 3.6 = 0.3$  (<10% Abweichung)).

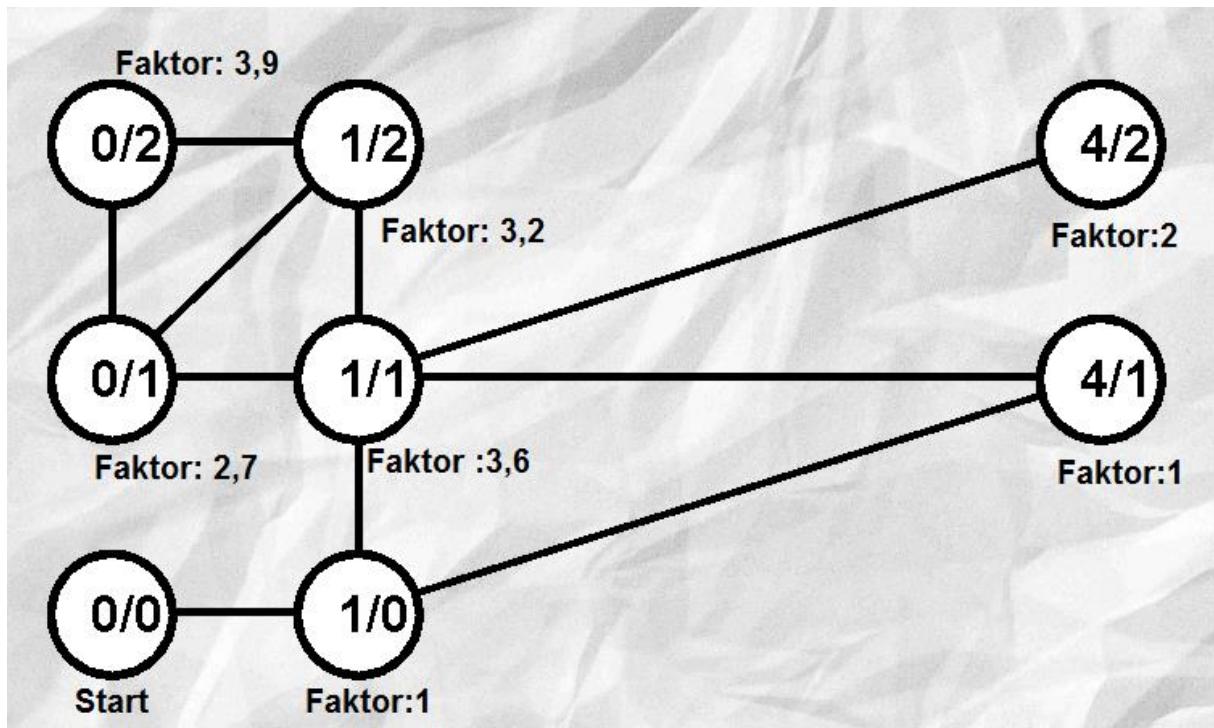


Abbildung 48 (Graph liegt bei)

Das Verfahren wurde an einigen Beispielgraphen des Forums ausprobiert – das Beispiel der Homepage ist oben abgehandelt.

| Hochgeladen von | Dateiname        | Größter Faktor (Entfernung) | Größter Faktor (Knotenanzahl) |
|-----------------|------------------|-----------------------------|-------------------------------|
| Fabian Märkert  | rechthausen3.txt | 27 (7798→7699)              | 27 (7798→7699)                |
| Fabian Michel   | beispiel1.txt    | $\infty$ (W→Q)              | $\infty$ (W→Q)                |
| Fabian Michel   | beispiel2.txt    | $\infty$ (17→d)             | $\infty$ (17→d)               |
| Fabian Michel   | beispiel3.txt    | $\infty$ (gz→bk)            | $\infty$ (gz→bk)              |
| Leon Windheuser | U-turn.txt       | 3 (E→C)                     | 3 (E→C)                       |
| Robin Schmöcker | 00013x00020.txt  | ~6.22 (3/3→2/1)             | 8 (3/3→2/1)                   |
| Robin Schmöcker | 01029x02688.txt  | ~75.21 (39/3→40/4)          | 66.5 (40/3→39/6)              |
| Robin Schmöcker | 05628x14058.txt  | ~154.10 (70/47→69/46)       | 117.5 (61/52→67/54)           |

## 7. Inhaltliche Erweiterung: Effizientes illegales Abbiegen

### 7.1. Vorüberlegung

Durch das neue Gesetz mag die Anzahl an Unfällen in Rechthausen zwar gesunken sein, jedoch erreichen die Bewohner von Rechthausen ihr Ziel deutlich langsamer als zuvor und sind deswegen verärgert. Manche Bewohner nehmen es deshalb sogar in Kauf, mehrere Male illegal links abzubiegen, um deutlich schneller an ihr Ziel zu gelangen. Wenn sie erwischt werden drohen Strafen, so dass sie einen möglichst effizienten Weg wählen wollen. Dafür wird aber ein AddOn für das Navigationssystem benötigt bei dem man angeben kann, wie häufig man maximal illegal Abbiegen möchte.

### 7.2. Problematik

Um dieses AddOn zu ermöglichen, kann nicht das bisher verwendete Shortest-Path Verfahren benutzt werden, da weder der Rechtsabbiegegraph noch der Beliebigabbiegegraph das Problem repräsentieren. Die inhaltliche Erweiterung hier ist weniger ein neues Verfahren, sondern eher eine Idee, das Problem so umzuwandeln ist, dass die bisherigen Verfahren möglichst ohne größere Modifikationen anwendbar sind.

### 7.3. Idee und Umsetzung

Die oben erwähnte Reduzierung lässt sich wie folgt ermöglichen. Man betrachtet zunächst den umgewandelten gerichteten Graphen  $G_0$  und erzeugt von diesem so viele Kopien ( $G_1, G_2, \dots, G_n$ ) wie man illegal links abbiegen möchte. Man stelle sich vor, dass  $G_0$  ein Straßensystem repräsentiert in welchem man noch nicht links abgebogen ist und in dem nur legal gefahren werden kann. Die erste Kopie  $G_1$  stellt das Straßensystem dar in dem man genau einmal links abgebogen ist und in dem nur legal weitergefahrene werden kann, usw. Um das illegale Abbiegen zu modellieren wird für eine Kreuzung folgende Erweiterung vorgenommen. In einem Graphen  $G_i$  besitzt jeder Knoten nur ausgehende Kanten zu den Knoten, die man befahren dürfte, käme man über eine bestimmte Kante. Man kann nun jedem dieser Unterknoten eine Kante zu den Zielknoten geben, die man gerade nicht besuchen dürfte, wenn man den Unterknoten besucht. Nur sind diese Kanten dann nicht auf die Knoten im gleichen Graphen  $G_i$ , sondern auf die in der folgenden Kopie  $G_{i+1}$  gerichtet. Das Befahren einer solchen Kante entspricht dann dem illegalen Linksabbiegen. Nun gilt es nur noch den kürzesten Weg von einem Unterknoten des Startknotens in  $G_0$  zu einem Unterknoten des Zielknotens in irgendeinem Graphen  $G_i$  zu finden genau wie in Aufgabenteil 3. Um dies ebenfalls zu vereinfachen werden wie üblich zwei Knoten hinzugefügt, die als Start und Zielknoten dienen. Der erste Knoten besitzt wieder Kanten gerichtet auf alle Unterknoten des Zielknotens im Originalgraphen  $G_0$ . Der Zielknoten besitzt aber diesmal Kanten kommend von allen Unterknoten des Zielknotens aller Graphen  $G_i$ .

Eine alternative Vorstellung dieser Idee ist sich das Ausgangsstraßensystem als einen Stapel von übereinander liegenden Ebenen zu denken. Eine Etage tiefer gehen ist analog zum Linksabbiegen.

## 7.4. Programmablauf

Im Folgenden ein Struktogramm des Programmablaufs.

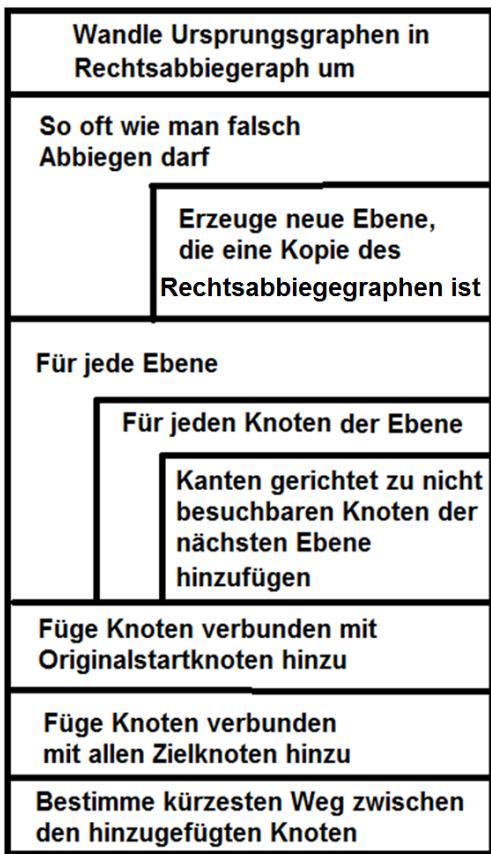


Abbildung 49

## 7.5. Laufzeit

Die Laufzeit des Verfahrens setzt sich einerseits aus dem Kopieren des Graphen, dem Verbinden der Ebenen, dem Hinzufügen der Start und Zielknoten sowie dem Dijkstra Algorithmus zusammen. Es wird ein Graph G mit  $|V|$  Knoten und  $|E|$  Kanten betrachtet.

Das Kopieren des ursprünglichen Graphen beträgt  $O(|V|+x*|V|+x*|E|) \leq O(x*N) \leq O(N)$ . x steht dafür wie oft man falsch Abbiegen darf – dies ist zwar beliebig wählbar, aber konstant für die O Notation.

Um Knoten in verschiedenen Ebenen mit Kanten zu verbinden muss jeder Knoten und jede Kante iteriert werden. Es ergibt sich also eine Laufzeit von  $O(x*N) = O(N)$ .

Das Hinzufügen der extra Knoten als Vorbereitung für den Dijkstra Algorithmus hat eine konstante Laufzeit  $O(1)$ .

Die Laufzeit des Dijkstra Algorithmus beträgt  $O(|V|*\log(|V|)+|E|)$  (siehe Kapitel 4.2.4.1). Der Graph, der nun aus mehreren Kopien des Originalgraphen besteht, hat eine Größe von  $x*N = O(N)$ . Die Laufzeit auf diesen Graphen angewendet beträgt damit  $O(|N|*\log(|N|))$ .

Die Laufzeit des gesamten Verfahrens beträgt also  $O(N+1+N^2) = O(N*\log(N))$ .

Hierfür wurde keine eigene Messreihe mehr erstellt, da sich die Laufzeit hier ebenfalls analog zum Aufgabenteil 3 verhält.

## 7.6. Speicherverbrauch

Es wird ein Rechtsabbiegegraph G mit  $|V|$  Knoten und  $|E|$  Kanten betrachtet.

Der Graph wird so oft kopiert wie man falsch Abbiegen darf. Im Grunde wird der Speicherverbrauch des Ausgangsgraphen mit der Anzahl wie oft man links abbiegen darf  $x$ , multipliziert. Der Speicher beträgt also  $O(x*(|V|+|E|)+|V|+|E|) = O(N)$ .

Zusätzlich muss der gesamte Speicher, der für den Dijkstra Algorithmus verwendet wird nun auf den neu entstandenen Graphen übertragen werden. Der Speicherverbrauch des Dijkstra beträgt  $O(N)$ , da für jeden Knoten seine Informationen bezüglich des Algorithmus gespeichert werden. Hier würde dann der Speicherverbrauch beim neu entstandenen Graphen wieder  $O(x*N) = O(N)$  betragen.

Der gesamte Speicherverbrauch liegt also bei  $O(N+N) = O(N)$  für jedes festgewählte  $x$ .

Hierfür wurde keine eigene Messreihe mehr erstellt, da der Speicherverbrauch hier ebenfalls linear sein muss, was bereits mit der Messung von Aufgabenteil 3 gezeigt wurde.

## 7.7. Beispiele

Es wird folgendes Rechthausen betrachtet. Es ist der kürzeste Weg von Knoten D zu Knoten A gesucht. Es ist erlaubt, einmal links abzubiegen. Als Weglängenmaß wird die Anzahl der besuchten Knoten gewählt. Alle Kanten haben also eine Kapazität von 1. Der Übersichtlichkeit halber wird dies aber nicht eingezeichnet.

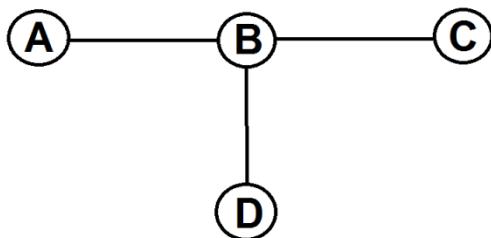


Abbildung 50 (Graph liegt bei)

Im ersten Schritt wird der gerichtete Graph erstellt.

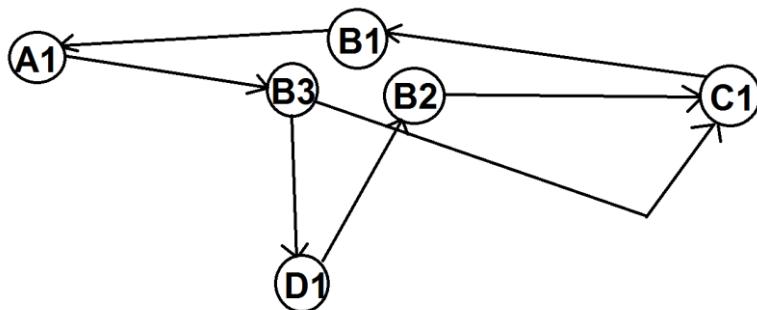


Abbildung 51

Im nächsten Schritt werden so viele Kopien des Graphen erzeugt, wie man falsch Abbiegen darf. In diesem Fall darf man einmal falsch Abbiegen und daher wird auch nur eine Kopie erstellt. Der resultierende Graph sieht wie folgt aus. Der obige Graph repräsentiert also ein Straßensystem in dem nicht links abgebogen wurde. Der untere Graph steht für ein Rechthausen, in welchem bereits einmal links abgebogen wurde. Im unteren Graphen sind die Namen der Knoten zusätzlich um ein \* ergänzt.

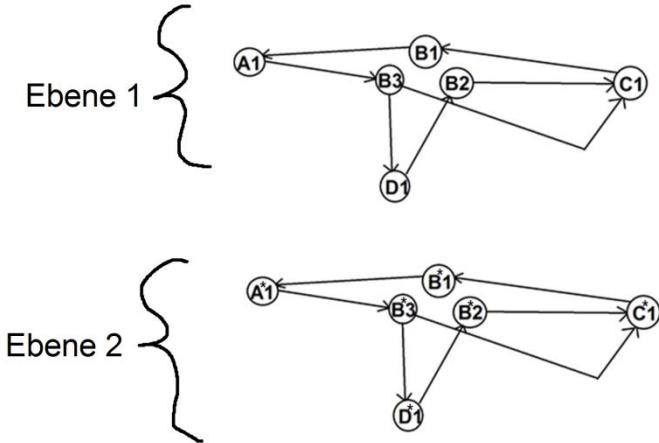


Abbildung 52

Nun wird von jedem Unterknoten eine Kante gerichtet auf die Unterknoten erstellt, zu denen man nicht abbiegen darf. Die Kanten sind aber immer auf die Knoten in der nächsten Kopie gerichtet. Im Folgenden ist der Graph abgebildet nach diesem Schritt. Knoten B3 bedeutet man besucht die Kreuzung B über den Knoten A. In diesem Falle dürfte man zu C und D weiterfahren. Es ist jedoch nicht erlaubt von diesem Knoten aus wieder zu Knoten A zu fahren. Es wird also eine Kante erstellt, die auf den Unterknoten von A in der nächsten Kopie ( $A1^*$ ), der das Besuchen über Knoten B repräsentiert, gerichtet ist. Besucht man Kreuzung B über Kreuzung C, dann ist es verboten Kreuzung D oder C zu befahren. Auch hier ist wieder eine Kante zur nächsten Kopie erstellt worden, gerichtet auf die entsprechenden Unterknoten. Knoten B2 steht für das Besuchen der Kreuzung B über Kreuzung D. Es ist hier nicht erlaubt auf A abzubiegen oder wieder zu D zurückzufahren. Deshalb hat B2 zwei Kanten, gerichtet auf die beiden Unterknoten der nächsten Kopie. Alle rot eingezeichneten Kanten, repräsentieren hier also illegales Abbiegen.

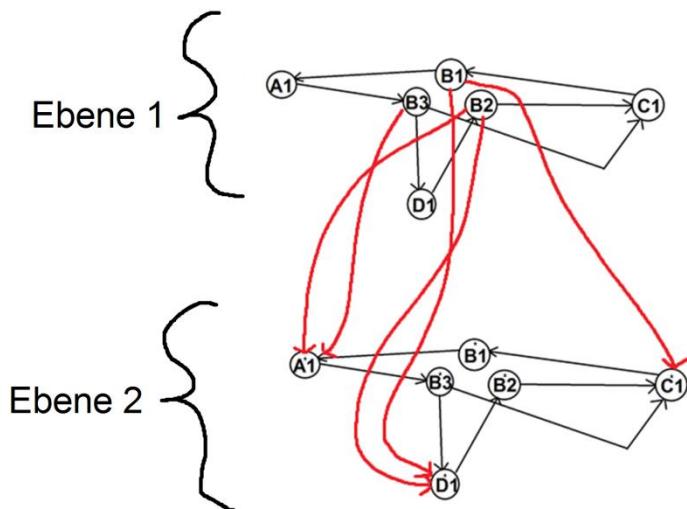


Abbildung 53

Im nächsten Schritt werden zwei Knoten hinzufügt, die als Start und Zielknoten für den Dijkstra Algorithmus dienen. Der Startknoten ist verbunden mit allen Unterknoten von Kreuzung D, die keine Kopie sind. Der hinzugefügte Zielknoten besitzt eine Kante kommend von allen Unterknoten der Zielkreuzung, also von Kreuzung A aller Kopien. In diesem Fall sind  $A1^*$  und  $A1$  mit diesem verbunden.

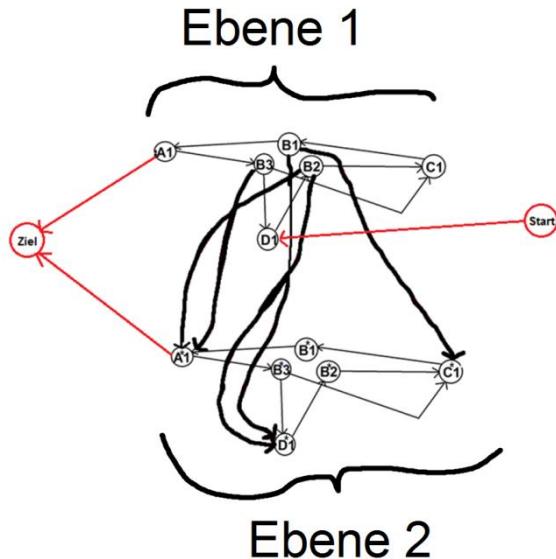


Abbildung 54

Im letzten Schritt und um letzten Endes den kürzesten Weg zu bestimmen, muss lediglich der kürzeste Weg zwischen dem hinzugefügten Start- und Zielknoten bestimmt werden, so dass letztendlich das Problem auf ein schon gelöstes Problem reduziert wurde. Der kürzeste Weg, wenn man den Graphen oben betrachtet, ist es nämlich vom Startknoten aus Knoten D1 dann B2 zu besuchen. Hier macht es dann Sinn falsch abzubiegen zu Knoten A1\* von welchem man dann den Zielknoten erreichen kann. Der schnellste Weg mit einmal falsch abbiegen ist also Start = D1 → B2 → A1\* = Ziel, also D → B → A.

## 7.8. Anwendung

Im Folgenden wird das Ergebnis der Anwendung der inhaltlichen Erweiterung anhand eines Graphen verdeutlicht. Die Bilder zeigen den kürzesten Weg zwischen zwei Knoten (1/1 und 10/10) und die Weglänge gemessen anhand der Knotenzahl. Die grün markierten Knoten sind Knoten die auf dem kürzesten Weg liegen. Kanten, die rot markiert sind, sind die Kanten auf welche das Abbiegen illegal ist. Ab 4 maligen Falschabbiegens entspricht die Weglänge des gefundenen Weges der optimalen Weglänge.

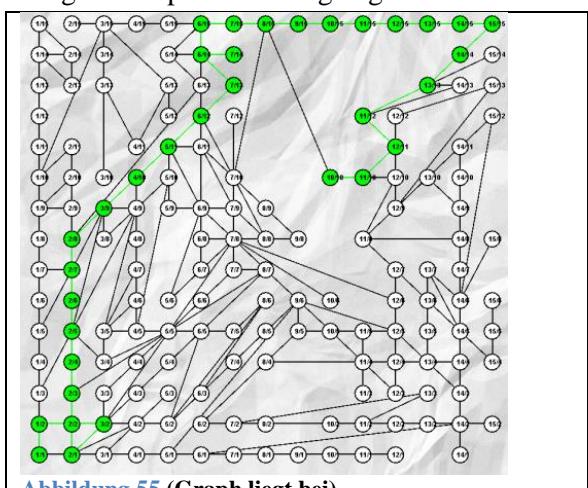
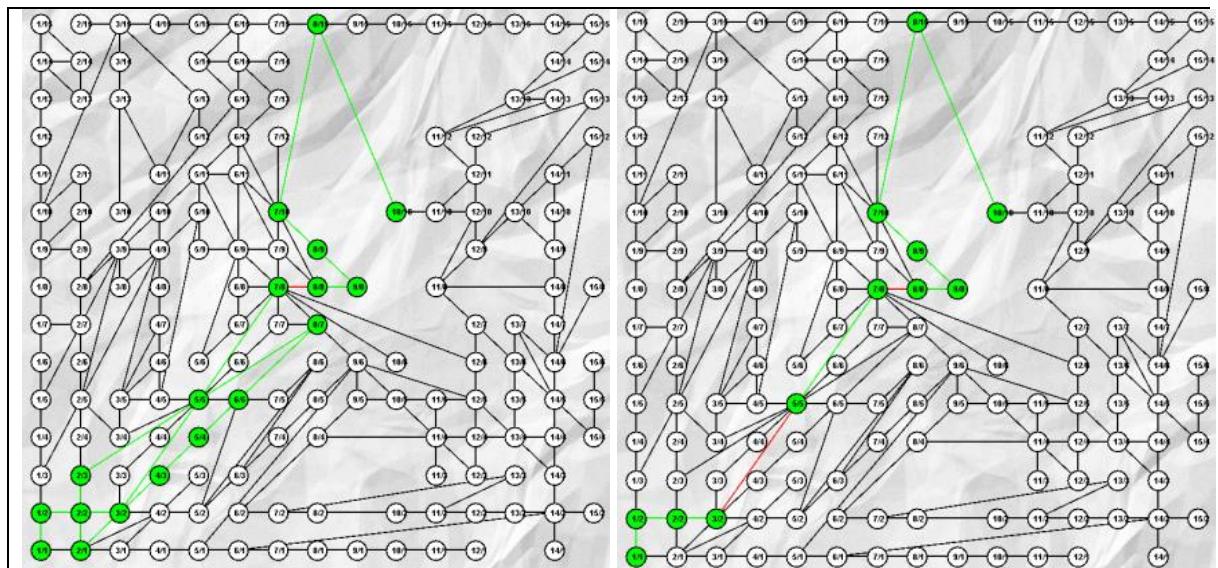


Abbildung 55 (Graph liegt bei)

Weglänge: 35 Knoten | 0x links abgebogen

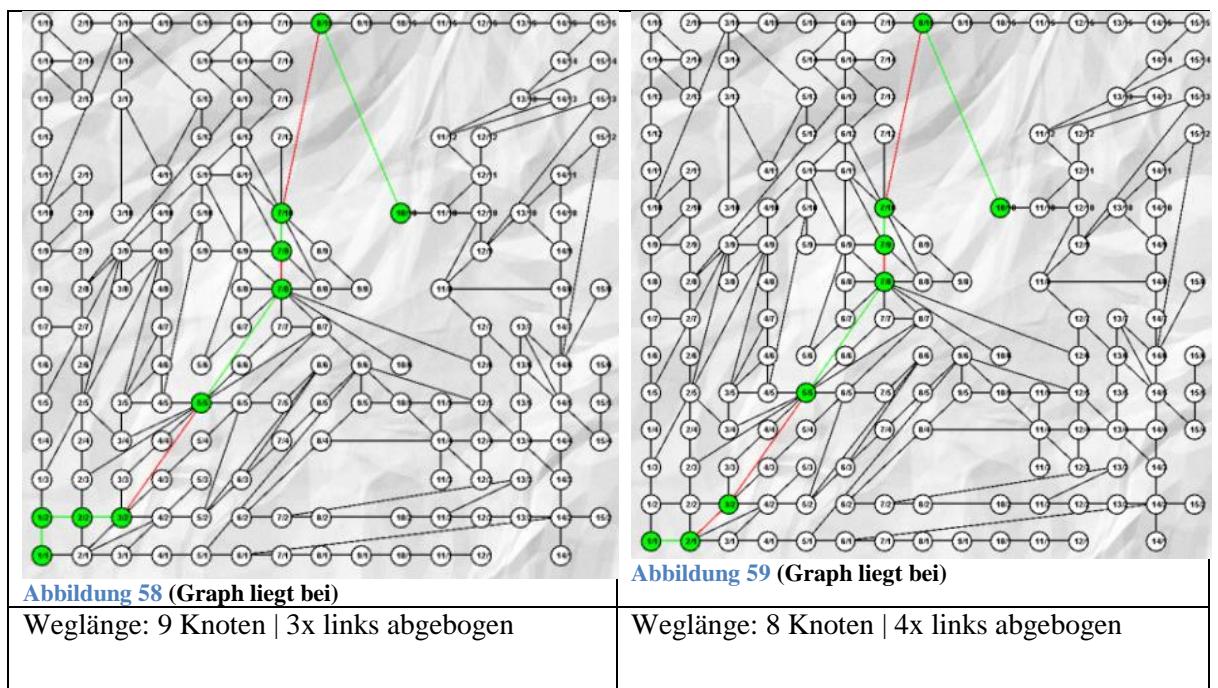


**Abbildung 56 (Graph liegt bei)**

Weglänge: 20 Knoten | 1x links abgebogen

**Abbildung 57 (Graph liegt bei)**

Weglänge: 11 Knoten | 2x links abgebogen



**Abbildung 58 (Graph liegt bei)**

Weglänge: 9 Knoten | 3x links abgebogen

**Abbildung 59 (Graph liegt bei)**

Weglänge: 8 Knoten | 4x links abgebogen

## 8. Straßengenerator – Aufgabenteil 6

### 8.1. Vorüberlegung

Der letzte Aufgabenteil sieht vor, dass man Beispiele auf der BwInf-Hompage behandelt. Da aber nur eine einzige Aufgabe, die aus der Aufgabenstellung, dort zu finden ist, und diese Aufgabe in allen vorherigen Kapiteln als Beispiel herangezogen wurde, impliziert die Aufgabenstellung in meinen Augen das Ausdenken bzw. Generieren von eigenen Testfällen. Es ist also sinnvoll das Programm so zu erweitern, dass es in der Lage ist Straßennetze zu generieren.

Folgende Anforderungen an den Generator habe ich mir gestellt, um realitätsgtreue Straßennetze zu erzeugen.

- Kanten (Straßen) dürfen sich nur bei Kreuzungen überschneiden (keine Brücken oder Tunnel)
- Positionierung der Knoten innerhalb der Menge  $\{(x,y) \mid x \in \mathbb{N}; y \in \mathbb{N}; x \leq X; y \leq Y\}$ . Einerseits ist die Beispielaufgabe auch mit ganzen Zahlen gestellt. Andererseits kann man so exakt bestimmen, ob sich drei Knoten auf einer Geraden befinden
- Alle Kreuzungen müssen sich gegenseitig, auch mit Linksabbiegeverbot, erreichen können, damit die Aufgabenteile 3-5 nicht frühzeitig abbrechen

Folgende Parameter ergaben sich, um eine Variation an Graphen zu erstellen.

- In welchen X und Y Bereich die Knoten verteilt sein dürfen
- Knotenanzahl in dem Bereich (Dichte der Kreuzungen)
- Kantenanzahl (Vernetzungsdichte)
- Seed, um gleiche zufällig generierte Graphen mit gleichen Parametern erzeugen zu können

### 8.2. Umsetzung

Das Erzeugen eines zufälligen Graphen untergliedert sich in 4 Schritte.

1. So viele Knoten wie erwünscht sind, werden zufällig im angegebenen Bereich verteilt
2. Solange die Kantenanzahl des Graphen unter der erwünschten Kantenanzahl liegt, werden zwei zufällige Knoten ausgewählt. Danach wird überprüft, ob auf der Strecke zwischen diesen beiden Knoten weitere Knoten des Graphen liegen. Danach wird jeweils eine Kante zwischen konsekutiven (Reihenfolge wird durch Entfernung zu einem der ursprünglich ausgewählten Knoten bestimmt) Knoten erstellt, falls diese keine bereits existierende Kante schneidet, damit keine Tunnel/Brücken auftreten.
3. Der Graph wird so angepasst, dass alle Knoten sich gegenseitig erreichen können. Zunächst wird die Methode, ob sich alle Knoten erreichen können, so angepasst, dass diese alle Knotenpaare, die sich nicht gegenseitig erreichen können, zurückgibt. Das Verfahren bricht einfach nicht ab, wenn es ein Knotenpaar, was die Bedingung erfüllt, gefunden hat, sondern speichert dieses lediglich in eine Liste, welche am Ende zurückgeliefert wird. Solange also noch nicht gegenseitig erreichbare Knoten im Graphen

- existieren, werden die Knoten entfernt, die am häufigsten in der von der Methode, ob sich alle Knoten gegenseitig erreichen können, zurückgelieferten Liste, vorhanden sind.
4. Optional wird neben dem Laden des erzeugten Graphen in das GUI der Graph ebenfalls in eine Textdatei, entsprechend dem auf der Homepage verwendeten Format, gespeichert.

Im Folgenden ist das Eingabefenster des Straßennetzgenerators abgebildet.

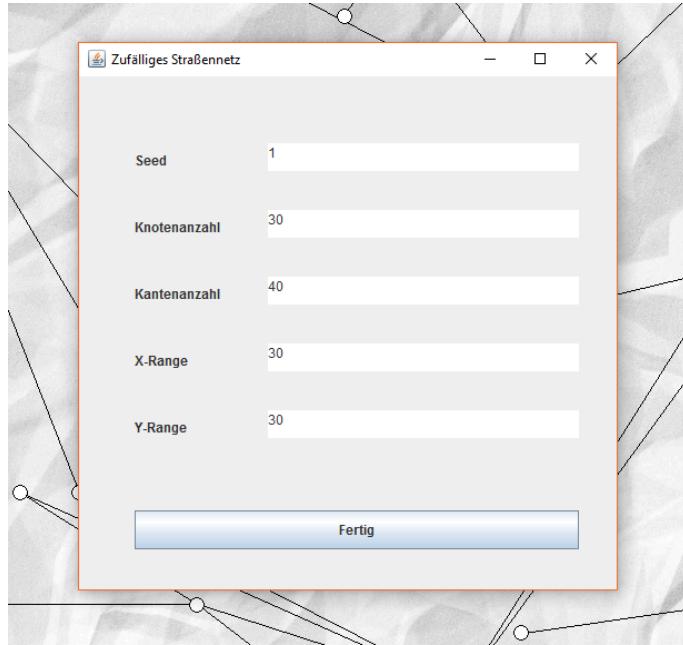


Abbildung 60

### 8.3. Weiterer Nutzen

Die Testdatengenerierung wurde außer zum Überprüfen der Aufgabenstellung auch verwendet, um Laufzeit – Speicherverbrauch Analysen zu erstellen. Weiterhin wurde das Verfahren benutzt um Graphen mit speziellen Eigenschaften zu finden (siehe Abbildung 43, 48).

## 9. Gemessene Laufzeiten und Speicherverbräuche

Im Folgenden sind zwei Grafiken von Messungen des Speicherverbrauchs und der Laufzeit für alle Programmteile anhand von ausgewählten größeren Straßennetzten.

Die Grafiken wurden in Excel erstellt und die Achsen sind logarithmisch skaliert, so dass man visuell etwa erkennen kann, ob es sich um ein lineares quadratisches oder kubisches Verhalten handelt. Weiterhin wurden die Messreihen mit einem konstanten Faktor pro Messreihe multipliziert, zur besseren Visualisierung, da sich bei einem konstanten Faktor die Steigung bzw. der Exponent von N nicht ändert. Orientierungsgraden für jeden der genannten Exponenten wurden auch in die Grafik mit eingezeichnet. Der Vergleich der Vorhersagen in den vorherigen Kapiteln mit den gemessenen Ergebnissen findet sich jeweils in den zugehörigen Kapiteln wieder. Die Graphen wurden zufällig mittels des Straßengenerators, beschrieben in Kapitel 8, generiert.

Die x-Achse gibt die Graphengröße (Kanten+Knoten) an, wohingegen die y-Achse den jeweils benötigten Speicher angibt.

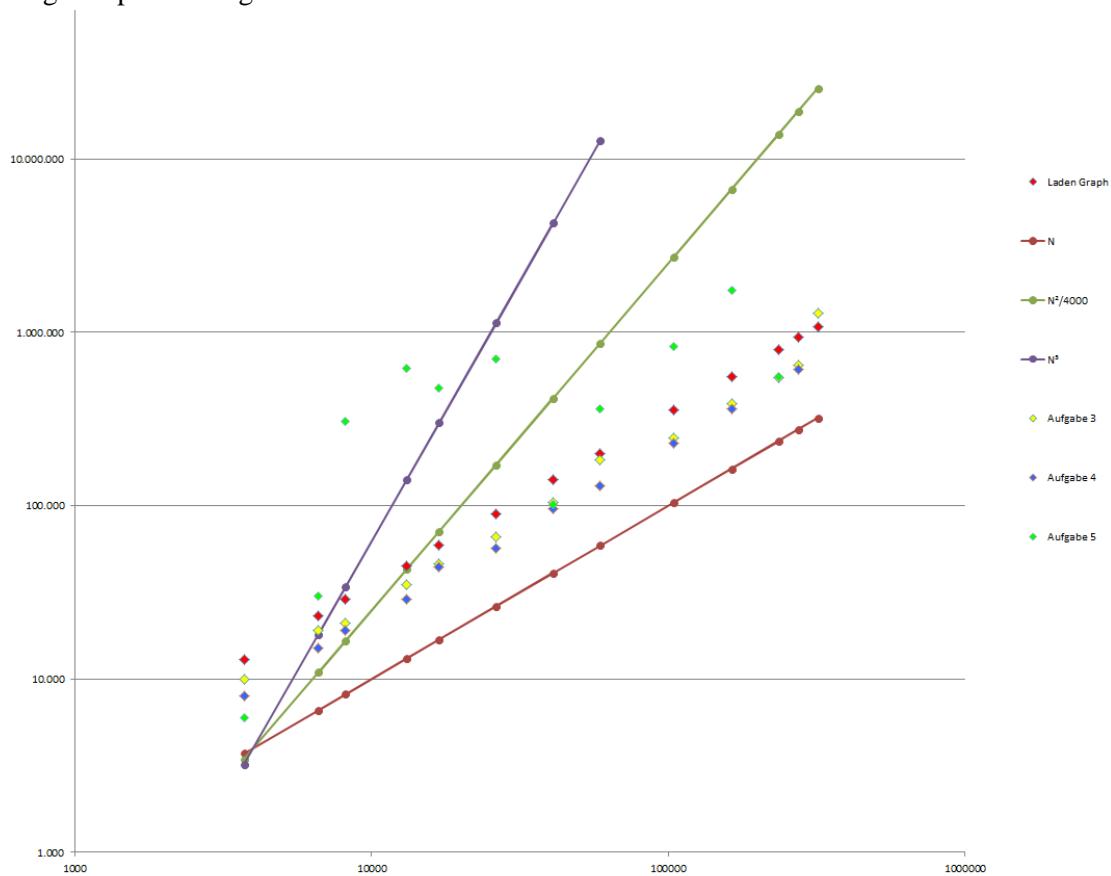


Abbildung 61 – Speicherverbrauch

Die x-Achse gibt die Graphengröße (Kanten+Knoten) an, wohingegen die y-Achse die jeweils benötigte Laufzeit angibt.

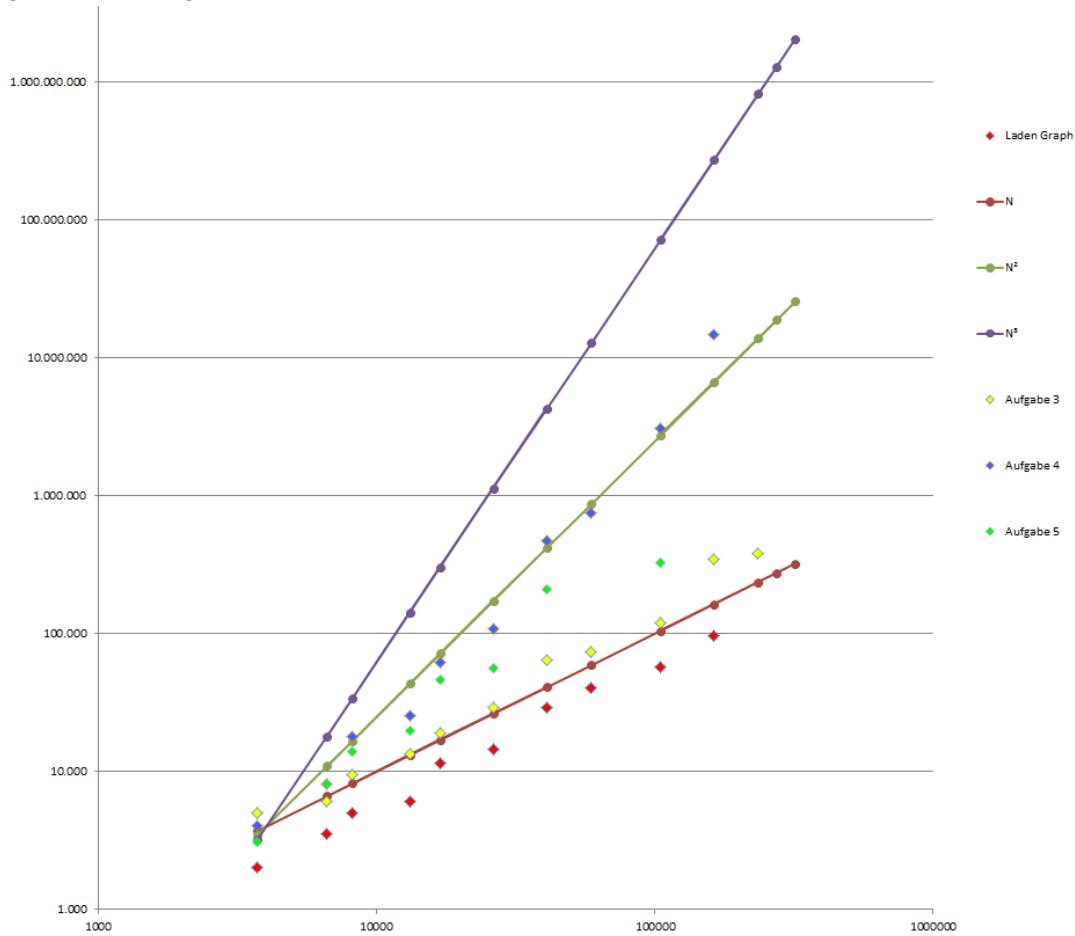


Abbildung 62 – Laufzeit

## 9.1. Grenzen des Programms

Neben dem Sammeln der Messdaten wurden ebenfalls die Grenzen des Programms ausgelotet. Der größte getestete, beigelegte Graph („Grenzen.txt“), an dem die Verfahren erfolgreich getestet wurden, besteht aus 94 642 Knoten und 179 550 Kanten. Die Grenzen des Programms wurden daran gemessen, ob das Programm terminiert.

**Kürzester Weg:** Terminiert innerhalb weniger Sekunden

**Alle Knoten gegenseitig erreichbar:** Dauerte ca. 15-20 Minuten, terminierte jedoch

**Größter Unterschied (Heuristisch):** Das Verfahren dauerte hier länger als 30 Minuten. Trotzdem wurde eine Lösung ermittelt

**Größter Unterschied (BruteForce):** Eine Hochrechnung hat ergeben, dass wahrscheinlich eher Ostern und Weihnachten auf einen Tag gefallen sind bevor das Programm terminiert

**Kürzester Weg mit illegalem Abbiegen:** 2x illegal Abbiegen bei default Einstellungen (8x illegales Abbiegen nach Zuweisung von  $\frac{3}{4}$  des Speichers des Rechners) war möglich. Die Laufzeit betrug bei 2x illegalem Abbiegen knapp 1 Minute (und bei 8x illegalem Abbiegen etwa 4 Minuten).

## 10. Programm-Dokumentation

Alle Methoden und somit auch die Umsetzung aller Aufgaben wurden in Java realisiert.

Das Programm basiert auf den Klassen **DGraph**, **Vertex**, **Edge**, welche zusammen einen gerichteten, gewichteten Graph darstellen. Die Hauptprozedur liest über **Import\_graph** ein Straßennetz ein. Aufgabenteil 3, das Finden eines kürzesten Weges, wird mit der Klasse **Shortest\_path** gelöst. Aufgabenteil 4, die Überprüfung, ob alle Knoten sich gegenseitig erreichen können, wurde mit der Klasse **Vertices\_not\_reachable** umgesetzt. Aufgabenteil 5, wurde mit den Klassen **Biggest\_difference\_heuristic** und **Biggest\_difference\_bruteforce** umgesetzt. Ersteres löst das Problem mit der beschriebenen, heuristischen Methode, letzteres bestimmt die genaue Lösung, was auch beschrieben wurde. Die inhaltliche Erweiterung, das Finden eines kürzesten Weges mit illegalen Abbiegungen wurde mit der Klasse **Shortest\_path\_illegal\_turns** umgesetzt. Im Folgenden befindet sich eine Übersicht der wichtigsten, für die Lösung der Aufgabe relevanten Klassen und deren wichtigsten Methoden und erwähnenswerten Attribute, falls deren Umsetzung nicht trivial ist. Die wichtigsten Quelltext-Schnipsel, verantwortlich für die Umsetzung der beschriebenen Algorithmen, befinden sich kommentiert im Anschluss. Diese Methoden sind mit (\*) gekennzeichnet.

- **DGraph - Klasse**

Die Implementierung der Algorithmen erfordert eine Klasse, welche einen kantengewichteten, gerichteten Graphen repräsentiert. Diese Klasse heißt **DGraph** und ist abstrakt und ihre wichtigsten Funktionen und erwähnenswerten Attribute sind:

**Methoden:**

- **addVertex<sup>(\*)</sup>** – Fügt dem Graphen einen Knoten mit einer Position und einem festgelegten Namen hinzu
- **removeVertex<sup>(\*)</sup>** – Entfernt aus dem Graphen einen Knoten inklusive all seiner Kanten. Weiterhin wird von allen Knoten, die mit ihm verbunden waren, die Kante zu ihm entfernt
- **getVertex** – Liefert das zugehörige Knotenobjekt
- **mergeVertices<sup>(\*)</sup>** – Verschmilzt bestimmte Knoten zu einem einzigen Knoten und gibt dem Knoten die Information welche Knoten zu ihm zusammengefasst wurden
- **addEdge<sup>(\*)</sup>** – Fügt dem Graphen eine gewichtete, gerichtete Kante zwischen zwei Knoten hinzu
- **removeEdge<sup>(\*)</sup>** – Entfernt eine Kante aus dem Graphen, in dem die ausgehende Kante des Quellknoten, sowie die eingehende Kante des Zielknoten entfernt werden
- **save** – Speichert einen Graphen in eine Datei
- **getMergedVertex** – Durch das Verschmelzen von Knoten, sind bestimmte Knoten nicht mehr im Graphen vorhanden. Diese Methode liefert den Knoten zurück, zu welchem ein Knoten zusammengeschmolzen wurde

**Attribute:**

- **vertices** – Dieses Attribut speichert alle Knoten des Graphen in Form eines HashSets. Dies bietet dem Vorteil gegenüber einer normalen Liste, dass mit konstanter Laufzeit überprüft werden kann, ob sich ein Knoten im Graph befindet. Außerdem können bestimmte Elemente schnell aus der Liste entfernt werden
- **vertices\_by\_name** – Hierbei handelt es sich um eine HashMap, die als Keywert einen Knotennamen und als Value das dazugehörige Knotenobjekt speichert. Dies ermöglicht es mit nahezu konstanter Laufzeit auf Knoten über ihren Namen zugreifen zu können

- **edges** – Dieses Attribut speichert alle Kanten in Form eines HashSets, um wie bei **vertices** schnell überprüfen zu können, ob ein bestimmtes Element im Graph enthalten ist und um bestimmte Kanten schnell aus der Liste zu entfernen
- **mergeList** – Mit diesem Attribut kann für einen Knoten direkt festgestellt werden, ob er in einen anderen Knoten verschmolzen wurde und gibt in diesem Fall den Verschmelzungsknoten zurück. Hierzu wird eine HashMap verwendet

- **DGraph\_right\_turn – Klasse**

Die Klasse ist eine Unterklasse der Klasse **DGraph** und stellt ein Rechthausen dar, welches in einen gerichteten Graph umgewandelt wurde.

**Methoden:**

- **convert<sup>(\*)</sup>** – Erstellt aus der Vorlage eines **DGraph\_all\_directions** nach dem in 3.5 beschriebenen Verfahren einen **DGraph\_right\_turn** (Rechtsabbiegegraph).

- **DGraph\_all\_directions – Klasse**

Die Klasse ist eine Unterklasse der Klasse **DGraph** und stellt einen Graphen dar in welchem man beliebig abbiegen darf. Dies entspricht einem ungerichteten Graphen, da zu jeder Kante eine Rückkante gebildet wird.

**Methoden:**

- **createInverseEdges** – Bildet für jede Kante eine Rückkante, so dass dieser Graph als ungerichtet betrachtet werden könnte. Vor Aufruf dieser Methode, stellt dieser Graph nur eine Vorlage zum Umwandeln der **convert** Methode dar

- **Vertex - Klasse**

Die **DGraph** Klasse speichert Knotenobjekte. Diese Objekte besitzen ebenfalls folgende wichtige Funktionen und erwähnenswerte Attribute.

**Methoden:**

- **getName** – Der Name des Knoten wird abgerufen
- **getCoordinates** – Die Koordinaten des Knoten werden abgerufen
- **getIncomingEdges** – Liefert Liste aller eingehenden Kanten
- **getOutgoingEdges** – Liefert Liste aller ausgehenden Kanten
- **getAngle** – Liefert den Winkel mit dem eine Kante auf den Knoten trifft
- **getAllowedEdges<sup>(\*)</sup>** – Gibt zurück welche Kanten befahren werden können, wenn man den Knoten über eine bestimmte Kante befährt
- **getProhibitedEdges<sup>(\*)</sup>** – Gibt zurück welche Kanten man nicht befahren dürfte, würde man den Knoten über eine bestimmte Kanten befahren
- **getHierarchieTop<sup>(\*)</sup>** – Im Rechtsabbiegegraph wird zu einem Unterknoten der Überknoten zurückgegeben. Ist der Knoten selber ein Überknoten, liefert er sich selber zurück. Im Beliebigabbiegegraph wird der Knoten selber zurückgegeben, da hier keine Über- und Unterknoten existieren
- **getHierarchieEnd<sup>(\*)</sup>** – Im Rechtsabbiegegraph werden zu einem Überknoten alle Unterknoten zurückgegeben. Ist der Knoten selber ein Unterknoten, liefert die Methode alle Knoten, die denselben Überknoten besitzen. Im Beliebigabbiegegraph wird der Knoten selber zurückgegeben, da hier keine Über- und Unterknoten existieren

- **getMergedTops** – Liefert alle verschiedenen Überknoten, der verschmolzenen Knoten

**Attribute:**

- **outgoing\_edges** – Speichert alle ausgehenden Kanten des Knoten in Form eines HashSets. Dies bietet den Vorteil, dass nahezu konstant überprüft werden kann, ob eine bestimmte Kante mit dem Knoten verbunden ist. Außerdem ermöglicht ein HashSet ein schnelles Entfernen eines bestimmten Elements
- **incoming\_edges** - Analog
- **outgoing\_vertices** – Speichert alle Zielknoten aller ausgehenden Kanten auch innerhalb einer HashMap. Das Schlüssellement ist ein Zielknoten einer ausgehenden Kante, das Value ist das dazugehörige Kantenobjekt. Dies bietet zusätzlich den Vorteil, dass schnell überprüft werden kann, ob der Knoten mit einem anderen Knoten verbunden ist. Zusätzlich kann, wenn vorhanden, ein Kantenobjekt, verbunden mit zwei Knoten, schnell abgerufen werden. Analog hierzu werden die eingehenden Kanten gespeichert
- **incoming\_vertices** – Analog
- **mergedTops** – Speichert alle verschiedenen Überknoten, der zu diesem Knoten verschmolzenen Unterknoten in Form eines HashSet. Hier wird hauptsächlich aus dem Grund ein HashSet verwendet, da dies doppelte Elemente vermeidet, da ja mehrere Unterknoten den gleichen Überknoten besitzen können
- **angles\_map** – Hierbei handelt es sich um eine TreeMap, die alle Winkel aller Kanten, die auf den Knoten treffen, speichert sowie als Value das dazugehörige Kantenobjekt. Dies ist dazu da, um in der **getAllowedEdges** Methode schnell die Kante mit dem nächst größeren Winkel abrufen zu können. Um auch effizient den Winkel einer Kante abzurufen, wird zusätzlich zu jeder Kante auch in einer HashMap der Winkel gespeichert

• **Vertex\_right\_turn – Klasse**

Die Klasse ist eine Unterklasse der **Vertex** Klasse. Diese Knoten werden als Knotenobjekte im **D\_Graph\_right\_turn** verwendet.

**Methoden:**

- **getParent** – Gibt den Überknoten zurück
- **getRelatives** – Gibt alle Unterknoten, die denselben Überknoten haben wie der Knoten, zurück
- **getChildren** – Gibt alle Unterknoten zurück

• **Edge - Klasse**

Die vorherigen Klassen arbeiten beide mit Kantenobjekten. Die **Edge** Klasse ermöglicht es mit gewichteten Kanten zu arbeiten. Der Konstruktor berechnet aus den Koordinaten der Knoten, die Kapazitäten, sofern als Weglängenmaß Entfernung gewählt ist, sonst wird die Kapazität auf 1 gesetzt. Die wichtigsten Funktionen der Klasse sind Folgende.

**Methoden:**

- **getCapacity** – Die Kapazität einer Kante wird abgerufen
- **setCapacity** – Die Kapazität einer Kante wird gesetzt
- **getSourceVertex** – Gibt den Knoten zurück, aus welchem eine Kante entspringt
- **getTargetVertex** – Gibt den Knoten zurück, auf welchen eine Kante gerichtet ist

- **Vertices\_not\_reachable - Klasse**

Diese Klasse ist dazu da, um zu bestimmen, ob zwei Kreuzungen existieren, die sich gegenseitig nicht erreichen können.

**Methoden:**

- **VerticesWhichCantReachEachOther<sup>(\*)</sup>** – Bestimmt, ob sich mindestens zwei Kreuzungen nicht gegenseitig erreichen können. Ist dies der Fall werden diese Kreuzungen zurückgegeben. Es werden erst alle Zyklen zusammengefasst, danach werden alle Knoten des Graphen iteriert und von ihnen eine Breitensuche gestartet. Findet eine Breitensuche nicht alle anderen Kreuzungen des Graphen, terminiert der Algorithmus. Die Methode gibt je nach Wunsch entweder alle nicht gegenseitig erreichbaren Knotenpaare (Benötigt für Straßengenerator) zurück oder nur eins (Benötigt für Aufgabe 4 und 5)
- **getReachableVerticesFromVertexGroup<sup>(\*)</sup>** – Liefert eine Liste an Überknoten, die von einer Liste an Knotengruppen aus erreichbar sind. Es wird eine Breitensuche von allen Knotengruppen ausgeführt
- **ExistVerticesNotReachableBecauseLeftProhibition<sup>(\*)</sup>** – Liefert, falls vorhanden, ein Knotenpaar was durch das Linksabbiegeverbot sich nicht mehr gegenseitig erreichen kann. Die Methode teilt einen Beliebigabbiegegraph zunächst in zusammenhängende Teilgraphen und überprüft, ob in einem von diesen mit Linksabbiegeverbot mindestens ein Knotenpaar sich nicht gegenseitig erreichen kann

- **Split\_graph – Klasse**

Diese Klasse wird dazu benötigt, um einen Graphen in einzelne zusammenhängende Graphen zu unterteilen. Dies wird für die Methode **ExistVerticesNotReachableBecauseLeftProhibition** gebraucht.

**Methoden:**

- **getVerticesConnectedToVertex<sup>(\*)</sup>** – Liefert alle Knoten, die mit einem Knoten verbunden sind. Dies geschieht mittels Breitensuche
- **splitGraph<sup>(\*)</sup>** – Teilt einen Graph in einzelne zusammenhängende Teilgraphen und gibt diese zurück

- **Remove\_Cycles - Klasse**

Die Aufgabe der Klasse Remove\_Cycles ist es in einer **DGraph** Instanz alle Zyklen zu vereinfachen. Die wichtigsten Funktionen sind im Folgenden zu finden.

**Methoden:**

- **removeAllCycles<sup>(\*)</sup>** – Entfernt alle Zyklen des Graphen
- **removeCycleIteration<sup>(\*)</sup>** – Sucht mittels Tiefensuche im Graphen nach einem Zyklus. Wurde einer gefunden, wird dieser zusammengefasst. Zurückgeliefert wird, ob ein Zyklus vereinfacht wurde und welche Knoten als eindeutig nicht zu einem Zyklus gehörend identifiziert worden sind.

**Attribute:**

Jeder Knoten befindet sich immer in einem Stadium. Es werden hierzu 3 HashSets verwendet, die alle Knoten im jeweiligen Stadium speichern. Dies ermöglicht es, Knoten schnell von einem

in ein anderes Stadium zu verschieben. Zusätzlich kann schnell überprüft werden in welchem Stadium sich ein Knoten befindet und wie viele Knoten sich in einem Stadium. Diese drei HashSets heißen:

- **graySet** (Stadium 1)
- **whiteSet** (Stadium 2)
- **blackSet** (Stadium 3)

- **Shortest\_Path - Klasse**

Diese Klasse bestimmt den kürzesten Weg zwischen zwei Kreuzungen. Die Methoden der Klasse sind folgende.

**Methoden:**

- **Shortest\_path<sup>(\*)</sup>** – Bestimmt den kürzesten Weg zwischen zwei Knoten und gibt diesen zurück indem zunächst imaginäre Knoten über **setupDijkstra** hinzugefügt werden. Danach wird der Weg über die **Dijkstra**-Klasse vollständig bestimmt.
- **setupDijkstra<sup>(\*)</sup>** – Fügt dem Graphen zwei Knoten hinzu, die als Start- und Zielknoten für den Dijkstra Algorithmus verwendet werden. Der Startknoten ist mit allen Knoten verbunden, die die Startkreuzung repräsentieren und der Zielknoten mit allen, die die Zielkreuzung repräsentieren.

- **Dijkstra - Klasse**

Diese Klasse wendet den Dijkstra Algorithmus auf einen gerichteten Graphen an. Die Klasse besteht aus folgenden Methoden.

**Methoden:**

- **applyDijkstra<sup>(\*)</sup>** – Wendet den Dijkstra Algorithmus auf einen gerichteten Graphen an. Es wird ein Start und Zielknoten angegeben. Außerdem wird angegeben, ob der Algorithmus terminiert, wenn der Zielknoten erreicht wurde. Andernfalls wird die Entfernung, bzw. der Weg zu jedem anderen Knoten ermittelt. Es wird eine Liste mit Knoten und deren für den Algorithmus relevanten Eigenschaften zurückgegeben (Entfernung zum Quellknoten, besucht, Vorgänger). Informationen zu den unbesuchten Knoten werden als sortierte Liste (TreeSet) gespeichert, um es zu ermöglichen, dass schnell der nächste unbesuchte Knoten mit der geringsten Entfernung ausgewählt werden kann. Alle Knoten und deren Informationen werden in Form einer HashMap gespeichert, um mit fast konstanter Laufzeit auf die Informationen eines beliebigen Knoten zugreifen zu können
- **getPathFromParentMap<sup>(\*)</sup>** – Extrahiert den Weg zwischen zwei Knoten mithilfe der Ergebnisse des Dijkstra Algorithmus mittels Backtracking

Die **Dijkstra** Klasse benötigt eine Hilfsklasse, die Informationen bezüglich der Knoten speichert. Diese Hilfsklasse nennt sich **dijkstra\_object** und speichert einen Knoten, ob dieser besucht wurde, die Entfernung zum Startknoten und von welchem Knoten dieser aufgerufen wurde. Außerdem lassen sich mit **compareTo** zwei dieser Objekte vergleichen, was benötigt wird um diese in einer sortierten Liste zu speichern.

- **Bigest\_Difference\_heuristic – Klasse**

Die Klasse ist dazu da um, ein Knotenpaar zu bestimmen bei welchem sich die Weglänge um den möglichst größten Faktor erhöht. Die Klasse setzt diese Aufgabe mittels des heuristischen Verfahrens um.

**Methoden:**

- ***getBiggestDifference*<sup>(\*)</sup>** – Diese Methode führt eine Breitensuche von jedem Knoten des Graphen aus und liefert den während den Breitensuchen größten gefundenen Faktor mit dem Knotenpaar zurück
- ***bfsDifference*<sup>(\*)</sup>** – Eine Breitensuche ausgehend von einem Knoten wird ausgeführt. Folgende Knoten mit geringerem Unterschiedsfaktor werden nicht weiter untersucht. Das Knotenpaar mit dem größten gefundenen Faktor während der Breitensuche, wird zusätzlich mit dem Faktor zurückgeliefert
- ***bfsDijkstra*<sup>(\*)</sup>** – Führt den Dijkstra Algorithmus solange aus bis ein gesuchter Knoten gefunden wird. Informationen bezüglich der Knoten können in späteren Aufrufen dieser Methode übernommen werden
- ***getDifferenceObjectFromDijkstra*<sup>(\*)</sup>** – Bildet aus zwei Weglängen, in Form von **dijkstra\_object**'en, das Unterschiedsobjekt vom Typ **difference\_object**, aus welchem wiederrum der in der Aufgabenstellung geforderte Faktor berechnet wird

• **Biggest\_Difference\_bruteforce – Klasse**

Diese Klasse hat dieselbe Aufgabe wie die als letztes beschriebene Klasse. Der Unterschied liegt darin, dass hier das exakte Ergebnis auf Kosten der Laufzeit bestimmt wird.

**Methoden:**

- ***getBiggestDifference*** – Diese Methode, ähnlich wie die aus **Biggest\_difference\_heuristic**, bestimmt den größten Unterschied in dem jeder einzelne Knoten des Graphen untersucht wird. Der größte Unterschied gefunden bei einem Knoten wird als Knotenpaar, verbunden mit dem Faktor zurückgeliefert
- ***getBiggestDifferenceSingleVertex*** – Diese Methode ist das Kernstück dieser Klasse. Von einem Knoten wird mittels Dijkstra Algorithmus der kürzeste Weg von ihm zu allen anderen Knoten im Rechtsabbiegegraph und im Beliebigabbiegegraph bestimmt. Für jeden Knoten wird dann der Unterschiedsfaktor gebildet und der größte gefundene wird zurückgeliefert.
- ***getDifferenceObjectFromDijkstra*** – Bildet aus zwei Weglängen in Form von **dijkstra\_object**'en das Unterschiedsobjekt vom Typ **difference\_object**, aus welchem wiederrum der in der Aufgabenstellung geforderte Faktor berechnet wird

Die beiden, eben genannten Klassen benötigen ebenfalls eine Hilfsklasse, die ein Knotenpaar sowie den Unterschiedsfaktor zwischen den Knoten speichert. Diese Hilfsklasse nennt sich **difference\_object** und besitzt neben den genannten Attributen noch die Möglichkeit über **compareTo** sich mit anderen Unterschiedsobjekten zu vergleichen.

• **Shortest\_Path\_illegal\_turns – Klasse**

Diese Klasse bestimmt den kürzesten Weg zwischen zwei Knoten ausgehend davon, dass man x-mal falsch Abbiegen darf.

**Methoden:**

- ***createLayers***<sup>(\*)</sup> – Kopiert die Anfangskanten und Anfangsknoten des Graphen so oft wie man falsch abbiegen darf
- ***connectLayers***<sup>(\*)</sup> – Verbindet die Ebenen des Graphen, so dass alle Kanten zu nicht befahrbaren Unterknoten auf der folgenden Ebene erzeugt werden
- ***getPath***<sup>(\*)</sup> – Verbindet obige Methoden indem es zuerst die Ebenen erstellt, diese dann verbindet, imaginäre Start- und Zielknoten hinzufügt, um schließlich den kürzesten Weg durch **Shortest\_Path** Klasse zu bestimmen

- **MapGenerator – Klasse**

Diese Klasse ermöglicht es mit vorgegebenen Parametern einen zufälligen Graphen zu erzeugen und in eine Textdatei zu schreiben.

**Methoden:**

- ***generateMap***<sup>(\*)</sup> – Hauptmethode der Klasse, welche alle anderen Methode kombiniert, um einen Graphen zu generieren. Zuerst werden Knoten zufällig verteilt, danach werden Kanten hinzufügt. Daraufhin werden Knoten solange entfernt bis sich alles gegenseitig erreichen kann. Zuletzt wird der Graph in das GUI geladen und bei Wunsch des Benutzers in eine Datei geschrieben
- ***generateVertices***<sup>(\*)</sup> – Fügt einem Graphen eine bestimmte Anzahl an Knoten hinzu, welche in einem vorgegebenen Bereich verteilt sind
- ***addEdges***<sup>(\*)</sup> – Fügt dem Graphen eine bestimmte Anzahl an Kanten hinzu. Die Kanten überschneiden sich nicht
- ***makeAllVerticesReachable***<sup>(\*)</sup> – Bestimmt mittels der **Vertices\_not\_reachable** Klasse alle Knotenpaare, die sich nicht gegenseitig erreichen können. Es werden immer die Knoten entfernt, die am häufigsten in der Liste vorkommen. Dieser Vorgang wird solange ausgeführt bis alle Knoten sich gegenseitig erreichen können

**Import\_graph – Klasse**

Diese Klasse hat die Aufgabe mit ihrer einzigen gleichnamigen Methode ***import\_graph*** den Benutzer eine Datei auswählen zu lassen, aus welcher dann ein Graph eingelesen wird.

## 11. Quelltextauszüge

Im Folgenden befinden sich kommentiert die meisten der oben beschriebenen Methoden. Attribute und simple aber wichtige Methoden wie „Getter“ und „Setter“ sind hier nicht enthalten, da deren Umsetzung uninteressant ist.

Im Code wird sich „[...]“ häufig wiederfinden. Dies ist eine Markierung dafür, dass Quelltext ausgelassen wurde, der nicht relevant für die Lösung der Aufgabe ist (GUI, Statistiken etc.).

Die Quelltextauszüge aus den Klassen befinden sich in folgender Reihenfolge:

1. **DGraph**
2. **DGraph\_right\_turn**
3. **Vertex**
4. **Edge**
5. **Shortest\_Path**
6. **Dijkstra**
7. **Shortest\_Path\_illegal\_turns**
8. **Vertices\_not\_reachable**
9. **Remove\_Cycles**
10. **Split\_graph**
11. **Biggest\_difference\_heuristic**
12. **Biggest\_difference\_bruteforce**
13. **MapGenerator**

## DGraph – Klasse: Datenstruktur für einen gerichteten Graph

```
/*
 * Fügt dem Graphen einen Knoten mit einem Namen, Koordinaten und ob dieser
 * imaginär ist, hinzu. Falls der angegebene Name bereits einem anderem Knoten
 * zugehörig ist, wird eine Fehlermeldung geworfen. Der zugefügte Knoten wird
 * als Objekt zurückgegeben. Ist der Knoten imaginär wird dieser in der
 * Winkelberechnung nicht berücksichtigt.
 */
public Vertex addVertex(Point coordinates, String name, boolean imaginary) throws IllegalArgumentException {
    if (name == null)
        throw new IllegalArgumentException("Name must not be null");
    if (vertices_by_name.containsKey(name))
        throw new IllegalArgumentException("Double identifier");
    Vertex ver = new Vertex(coordinates, name, imaginary);
    vertices.add(ver);
    vertices_by_name.put(name, ver);
    return ver;
}

/*
 * Entfernt einen Knoten aus dem Graphen. Alle Kanten, die mit diesen Knoten verbunden sind,
 * werden ebenfalls entfernt. Wurde der Knoten entfernt, ist sein Name wieder
 * nutzbar.
 */
public void removeVertex(Vertex vertex) {
    // Ausgehende Kanten entfernen
    for (Edge edg : new ArrayList<Edge>(vertex.getOutgoingEdges()))
        removeEdge(edg);
    // Eingehende Kanten entfernen
    for (Edge edg : new ArrayList<Edge>(vertex.getIncomingEdges()))
        removeEdge(edg);
    // Knoten aus Listen entfernen
    vertices.remove(vertex);
    vertices_by_name.remove(vertex.getName());
}
```

```
/*
 * Eine Kante ausgehend von "source_vertex" und gerichtet auf "target_vertex"
 * wird dem Graphen hinzufügt. Falls einer der beiden Knoten nicht im Graphen
 * vorhanden ist oder wenn der Zielknoten dem Quellknoten entspricht, wird ein
 * Fehler ausgegeben. Existiert bereits eine Kante ausgehend vom Quellknoten
 * gerichtet zum Zielknoten, wird die Kante nicht erstellt. Die Methode gibt
 * die hinzugefügte Kante als Objekt zurück.
 */
public Edge addEdge(Vertex source_vertex, Vertex target_vertex) throws IllegalArgumentException {
    if (!vertices.contains(source_vertex) || !vertices.contains(target_vertex))
        throw new IllegalArgumentException("[addEdge] One of the selected vertices does not exist in the graph!");

    if (source_vertex == target_vertex)
        throw new IllegalArgumentException("[addEdge] source vertex equals target vertex");

    if (source_vertex.outgoingVerticesContainsVertex(target_vertex)) {
        return null;
    }

    Edge edge = new Edge(source_vertex, target_vertex);
    edges.add(edge);

    target_vertex.addIncomingEdge(edge);
    source_vertex.addOutgoingEdge(edge);

    return edge;
}

/*
 * Eine Kante wird aus dem Graphen entfernt.
 */
public void removeEdge(Edge edg) {
    // Aus Quellknoten entfernen
    edg.sourceVertex().removeOutgoingEdge(edg);
    // Aus Zielknoten entfernen
    edg.targetVertex().removeIncomingEdge(edg);
    // Aus Kantenliste entfernen
    edges.remove(edg);
}
```

```
/*
 * Verschmilzt eine Liste von Knoten mit einem Zielknoten. Ebenfalls werden
 * alle Kanten, verbunden mit einem der Knoten der Liste, nun mit dem
 * Zielknoten verbunden.
 */
public void mergeVertices(Vertex merge_target, ArrayList<Vertex> vertices) throws IllegalArgumentException {

    for (Vertex to_merge : vertices) {
        // Falls der Knoten mit sich selber verschmolzen werden soll, wird ein
        // Fehler ausgegeben
        if (to_merge == merge_target)
            throw new IllegalArgumentException("Zielknoten kann nicht mit sich selbst gemergt werden.");
        // Alte, eingehende Kanten von Knoten aus Liste auf Mergeknoten umrichten
        for (Edge edg : new ArrayList<Edge>(to_merge.getIncomingEdges())) {
            removeEdge(edg);
            if (edg.sourceVertex() != merge_target)
                addEdge(edg.sourceVertex(), merge_target);
        }
        // Alte, ausgehende Kanten von Knoten der Liste von Mergeknoten kommend,
        // erzeugen
        for (Edge edg : new ArrayList<Edge>(to_merge.getOutgoingEdges())) {
            removeEdge(edg);
            if (merge_target != edg.targetVertex())
                addEdge(merge_target, edg.targetVertex());
        }
        // Überknoten wird zu Liste der gemergten Überknoten des Mergeknotens
        // hinzugefügt
        merge_target.addMergedVertex(to_merge);
        // Falls der zu mergende Knoten selber Produkt einer Verschmelzung war,
        // werden zu ihm gemergten Überknoten in den Mergeknoten übertragen
        merge_target.getMergedTops().addAll(to_merge.getMergedTops());
        to_merge.getMergedTops().clear();
        // Aktualisierung zu welchem Knoten der zu mergende Knoten gemergt wurde
        putSplitList(to_merge, merge_target);
        // Entfernen des zu verschmelzenden Knotens aus dem Graphen
        removeVertex(to_merge);
    }
}
```

```
/*
 * Knoten und seine gemergten Knoten
 * werden nun eingetragen, dass sie
 * zu einem bestimmten Knoten
 * verschmolzen wurden.
 */
private void putSplitList(Vertex vertex, Vertex merge_target) {

    if (inverseMergeList.containsKey(vertex)) {
        for (Vertex ver : inverseMergeList.get(vertex))
            addToMap(ver, merge_target);
        inverseMergeList.remove(vertex);
    }

    addToMap(vertex, merge_target);

}

/*
 * Hilfsmethode für obere Methode.
 * Ein Knoten wird entsprechend
 * in beide Mergelisten eingetragen.
 */
private void addToMap(Vertex element, Vertex map) {
    if (!inverseMergeList.containsKey(map)) {
        HashSet<Vertex> set = new HashSet<Vertex>();
        set.add(element);
        inverseMergeList.put(map, set);
    } else
        inverseMergeList.get(map).add(element);

    mergeList.put(element, map);
}
```

## DGraph\_right\_turn – Klasse: Datenstruktur für Rechtsabbiegegraph

```
/*
 * Wandelt einen Beliebigabbiegegraph in einen Rechtsabbiegegraph um.
 */
public static DGraph_right_turn convert(DGraph_all_directions graph) throws IllegalArgumentException {

    // Überprüfung, ob der umzuandelnde Graph zulässig ist, also keine
    // Rückkanten besitzt
    if (!graph.getEdges().isEmpty()) {
        Edge test_edge = graph.getEdges().iterator().next();
        if (test_edge.sourceVertex().incomingVerticesContainsVertex(test_edge.targetVertex())) {
            throw new IllegalArgumentException(
                "Ein Graph mit Rückkanten kann nicht konvertiert werden wegen Anomaliegefahr");
        }
    }

    DGraph_right_turn new_graph = new DGraph_right_turn();

    // Knoten des umzuandelnden Graph in neuen Graph kopieren
    Stack<Vertex> old_vertices = new Stack<Vertex>();
    for (Vertex ver : graph.getVertices()) {
        old_vertices.push(new_graph.addVertex(ver.getCoordinates(), ver.getName()));
    }

    // Aufteilung jedes Knoten in Unterknoten
    Stack<Vertex> createdVertices = new Stack<Vertex>();
    for (Vertex ver : graph.getVertices()) {
        splitVertex(new_graph, graph, ver, createdVertices);
    }

    // Erzeugte Unterknoten untereinander verbinden
    for (Vertex vertex : createdVertices) {
        for (Edge edge : new ArrayList<Edge>(vertex.getOutgoingEdges())) {
            // Bestimmte passenden Unterknoten
            Vertex new_target = null;
            for (Vertex child : ((Vertex_right_turn) edge.targetVertex()).getChildren()) {
                if (child.incomingVerticesContainsVertex(((Vertex_right_turn) vertex).getParent())) {
                    new_target = child;
                    break;
                }
            }
            // Passe Kante an
            new_graph.addEdge(vertex, new_target);
            new_graph.removeEdge(edge);
        }
    }
}
```

```

// Entferne Überknoten aus Graph
for (Vertex old_ver : old_vertices)
    new_graph.removeVertex(old_ver);

return new_graph;
}

/*
 * Teilt Knoten in Unterknoten auf.
 */
private static void splitVertex(DGraph_right_turn destination_graph, DGraph_all_directions graph, Vertex vertex,
    Stack<Vertex> createdVertices) {

    // Überprüfung, ob Knoten A keine Kanten besitzt
    if (vertex.getOutgoingEdges().size() == 0 && vertex.getIncomingEdges().size() == 0) {
        Vertex_right_turn new_vertex = (Vertex_right_turn) destination_graph.addVertex(vertex.getCoordinates(),
            generateVertexname(vertex.getName(), 0));
        ((Vertex_right_turn) destination_graph.getVertex(vertex.getName())).addChild(new_vertex);
    }

    int i = 0;
    /*
     * Da umzuwendender Graph ungerichtet ist, wird zwischen eingehenden und
     * ausgehenden Kanten nicht differenziert. Es müssen daher ausgehende und
     * eingehende Kanten abgearbeitet werden.
     */
    for (Edge edge : vertex.getOutgoingEdges()) {
        // Erzeuge Unterknoten a für Kante von B->A; erlaubte Wege seien A->C und
        // optional A->D (falls geradeaus)
        Vertex_right_turn new_vertex = (Vertex_right_turn) destination_graph.addVertex(vertex.getCoordinates(),
            generateVertexname(vertex.getName(), i++));
        ((Vertex_right_turn) destination_graph.getVertex(vertex.getName())).addChild(new_vertex);
        // Erzeuge eingehende Kante von B zu Unterknoten a
        destination_graph.addEdge(destination_graph.getVertex(edge.targetVertex().getName()), new_vertex);
        Edge[] path = vertex.getAllowedEdges(edge);
        // Erzeuge ausgehende Kanten von a zu C und optional D
        for (Edge path_edge : path) {
            if (path_edge.sourceVertex() == vertex)
                destination_graph.addEdge(new_vertex, destination_graph.getVertex(path_edge.targetVertex().getName()));
            else
                destination_graph.addEdge(new_vertex, destination_graph.getVertex(path_edge.sourceVertex().getName()));
        }
        createdVertices.push(new_vertex);
    }

    for (Edge edge : vertex.getIncomingEdges()) {
        // Erzeuge Unterknoten a für Kante von B->A; erlaubte Wege seien A->C und
        // optional A->D (falls geradeaus)
    }
}

```

```

        Vertex_right_turn new_vertex = (Vertex_right_turn) destination_graph.addVertex(vertex.getCoordinates(),
            generateVertexname(vertex.getName(), i++));
        ((Vertex_right_turn) destination_graph.getVertex(vertex.getName())).addChild(new_vertex);
        // Erzeuge eingehende Kante von B zu Unterknoten a
        destination_graph.addEdge(destination_graph.getVertex(edge.getSourceVertex().getName()), new_vertex);
        Edge[] path = vertex.getAllowedEdges(edge);
        // Erzeuge ausgehende Kanten von a zu C und optimal D
        for (Edge path_edge : path) {
            if (path_edge.getSourceVertex() == vertex)
                destination_graph.addEdge(new_vertex, destination_graph.getVertex(path_edge.getTargetVertex().getName()));
            else
                destination_graph.addEdge(new_vertex, destination_graph.getVertex(path_edge.getSourceVertex().getName()));
        }
        createdVertices.push(new_vertex);
    }

}

```

## Vertex – Klasse: Datenstruktur für einen Knoten

```

* Gibt den obersten Knoten, der Rangfolge wieder in der sich der Knoten
* befindet. Ist der Knoten Teil eines Rechtsabbiegegraph, wird sein
* Überknoten zurückgegeben. Ist der Knoten Teil eines Beliebigabbiegegraph,
* gibt er sich selber zurück.
*/
public Vertex getHierarchieTop() {
    if (this instanceof Vertex_right_turn && ((Vertex_right_turn) this).getParent() != null)
        return ((Vertex_right_turn) this).getParent();
    return this;
}

```

```
/*
 * Gibt die untersten Knoten, der Rangfolge wieder in der sich der Knoten
 * befindet. Ist der Knoten Teil eines Rechtsabbiegegraph und ein Unterknoten,
 * gibt er alle Knoten, die denselben Überknoten haben wie er selbst. Ist der
 * Knoten Teil eines Rechtsabbiegegraphen und ein Überknoten gibt er seine
 * Unterknoten zurück. Ist der Knoten Teil eines Beliebigabbiegegraph gibt er
 * sich selbst zurück.
 */
public Stack<Vertex> getHierarchieEnd() {
    if (!(this instanceof Vertex_right_turn)) {
        Stack<Vertex> end = new Stack<Vertex>();
        end.push(this);
        return end;
    } else {
        Vertex_right_turn me = (Vertex_right_turn) this;
        if (me.getParent() == null) {
            return me.getChildren();
        } else {
            return me.getRelatives();
        }
    }
}

/*
 * Eine Kante, die diesen Knoten als Quellknoten hat, wird in die zugehörigen
 * Listen eingetragen. Diese Methode wird und darf nur über die addEdge
 * Methode aus der DGraph-Klasse aufgerufen werden.
 */
void addOutgoingEdge(Edge edge) {
    outgoing_edges.add(edge);
    outgoing_vertices.put(edge.targetVertex(), edge);
    addAngle(edge);
}

/*
 * Entfernt eine Kante, die diesen Knoten als Quellknoten hat, aus den
 * zugehörigen Listen. Diese Methode wird darf nur über die Methode removeEdge
 * aus der DGraph-Klasse aufgerufen werden.
 */
void removeOutgoingEdge(Edge edge) {
    outgoing_edges.remove(edge);
    outgoing_vertices.remove(edge.targetVertex());
    removeAngle(edge);
}
```

```
/*
 * Eine Kante, gerichtet auf diesen Knoten ,wird in die zugehörigen Listen
 * gespeichert. Diese Methode wird und darf nur über die addEdge Methode aus
 * der DGraph-Klasse aufgerufen werden.
 */
void addIncomingEdge(Edge edge) {
    incoming_edges.add(edge);
    incoming_vertices.put(edge.sourceVertex(), edge);
    addAngle(edge);
}

/*
 * Eine Kante gerichtet auf diesen Knoten wird aus den zugehörigen Listen
 * entfernt. Diese Methode wird und darf nur über die removeEdge Methode aus
 * der DGraph-Klasse aufgerufen werden.
 */
void removeIncomingEdge(Edge edge) {
    incoming_edges.remove(edge);
    incoming_vertices.remove(edge.sourceVertex());
    removeAngle(edge);
}

/*
 * Entfernt den Winkel einer Kante aus den Winkellisten insofern dieser Knoten
 * nicht imaginär ist.
 */
private void removeAngle(Edge edge) {
    if (this instanceof Vertex_right_turn || edge.targetVertex().isImaginary() || edge.sourceVertex().isImaginary())
        return;
    angles_map.remove(edges_angles.remove(edge));
}

/*
 * Berechnet den Winkel einer Kante und fügt diesen in die Listen ein,
 * insofern der Knoten nicht imaginär ist.
 */
private void addAngle(Edge edge) {
    if (this instanceof Vertex_right_turn || edge.targetVertex().isImaginary() || edge.sourceVertex().isImaginary())
        return;
    // Winkelberechnung
    double angle = getAngle(edge);
    // Eintrag in Listen
    angles_map.put(angle, edge);
    edges_angles.put(edge, angle);
}
```

```
/*
 * Bestimmt den Winkel einer Kante zu diesem Knoten.
 */
private double getAngle(Edge edge) {
    Vertex v1, v2;
    if (edge.sourceVertex() == this) {
        v1 = edge.sourceVertex();
        v2 = edge.targetVertex();
    } else {
        v1 = edge.targetVertex();
        v2 = edge.sourceVertex();
    }
    // Winkelberechnung
    double angle = angleOf(v1.getCoordinates(), v2.getCoordinates());
    return angle;
}

/*
 * Gibt die Kanten zurück, die man befahren darf wenn man den Knoten über eine
 * Kante besucht.
 */
public Edge[] getAllowedEdges(Edge edge) throws IllegalArgumentException {

    // Wenn nur eine Kante existiert, kann auch über diese zurückgefahren werden
    if (angles_map.size() == 1)
        return new Edge[] { edge };
    // Rufe Winkel der Kante ab
    Double angleOfEdge = edges_angles.get(edge);
    // Überprüfung, ob Knoten imaginär ist
    if (angleOfEdge == null)
        throw new IllegalArgumentException(
            "Man kann eine Kreuzung nicht über imaginäre Knoten besuchen oder von einer imaginären Kreuzung kommen!");
    // Kante mit dem nächst größeren Winkel wird ermittelt
    Double next_angle = angles_map.higherKey(angleOfEdge);
    if (next_angle == null)
        next_angle = angles_map.higherKey(new Double(-1));
    // Überprüfung, ob eine gegenüberliegende Kante existiert
    if (angles_map.containsKey((angleOfEdge + 180) % 360)
        && angles_map.get((angleOfEdge + 180) % 360) != angles_map.get(next_angle))
        return new Edge[] { angles_map.get((angleOfEdge + 180) % 360), angles_map.get(next_angle) };
    return new Edge[] { angles_map.get(next_angle) };
}
```

```
/*
 * Gibt alle Kanten zurück, die man nicht befahren darf, wenn man den Knoten
 * über eine Kante besucht.
 */
public Edge[] getProhibitedEdges(Edge visiting_edge) {
    // Wenn nur eine Kante existiert, gibt es keine illegale Kante
    if (angles_map.size() == 1 || angles_map.isEmpty())
        return new Edge[0];
    // Legale Kanten werden abgerufen
    Edge[] nextEdges = getAllowedEdges(visiting_edge);
    Edge[] prohibitedEdges = new Edge[angles_map.size() - nextEdges.length];
    // Ermittle alle Kanten, die nicht legal sind
    int index = 0;
    for (Edge edge : edges_angles.keySet()) {
        if (!(nextEdges[0] == edge || (nextEdges.length > 1 && nextEdges[1] == edge)))
            prohibitedEdges[index++] = edge;
    }
    return prohibitedEdges;
}

/*
 * Gibt den Winkel einer Geraden zurück, die auf den Knoten trifft.
 */
private double angleOf(Point pos1, Point pos2) {
    final double deltaY = (pos2.y - pos1.y);
    final double deltaX = (pos2.x - pos1.x);
    // atan2 = [-pi;+pi] -> [-180°;+180°]
    final double result = Math.toDegrees(Math.atan2(deltaY, deltaX));
    return normalizeAngle(result);
}

/*
 * Winkel auf Intervall [0;360) und auf zwei Nachkommastellen normieren.
 */
private double normalizeAngle(double angle) {
    angle = (double) Math.round(angle * 100) / 100;
    if (angle < 0)
        angle += 360;
    return angle;
}
```

## Edge – Klasse: Gewichtete Kante

```
/*
 * Konstruktor, Festlegung von wo die Kante startet und worauf sie gerichtet
 * ist. Es wird automatisch die Kapazität der Kante anhand der Koordinaten der
 * Knoten berechnet.
 */
public Edge(Vertex source_vertex, Vertex target_vertex) {
    this.source_vertex = source_vertex;
    this.target_vertex = target_vertex;
    if (GUI_Graph.capacity_is_distance)
        capacity = (float) source_vertex.getCoordinates().distance(target_vertex.getCoordinates());
    else
        capacity = 1;
}
```

## Shortest\_path – Klasse: Bestimmung des kürzesten Weges

```
/*
 * Liefert den kürzesten Weg zwischen zwei Kreuzungen
 */
public static ArrayList<Vertex> shortest_path(DGraph graph, Vertex source, Vertex sink) {
    [...]
    // Füge imaginäre Start- und Zielknoten hinzu
    Vertex[] source_and_sink = setupDijkstra(graph, source.getHierarchieEnd(), sink.getHierarchieEnd());
    [...]
    // Wende den Dijkstra Algorithmus auf Graphen an
    HashMap<Vertex, dijkstra_object> parentMap = Dijkstra.applyDijkstra(graph, source_and_sink[0], source_and_sink[1],
        true);
    [...]
    // Der kürzeste Weg wird aus den erhaltenen Daten des Dijkstras extrahiert
    ArrayList<Vertex> path = Dijkstra.getPathFromParentMap(source_and_sink[1], parentMap);
    if (path != null) {
        path.remove(0);
        path.remove(path.size() - 1);
    }
    [...]
    // Originalzustand des Graphen wird wieder hergestellt
    graph.removeVertex(source_and_sink[0]);
    [...]
    graph.removeVertex(source_and_sink[1]);
    [...]
    return path;
}
```

```
/*
 * Fügt einem Graphen zwei imaginäre Knoten hinzu, welche als Start- und
 * Zielknoten für den Dijkstra Algorithmus dienen. Gibt die hinzugefügten
 * Knoten zurück.
 */
public static Vertex[] setupDijkstra(DGraph graph, Stack<Vertex> source, Stack<Vertex> sink) {
    Vertex addedSource = graph.addVertex(source.peek().getCoordinates(), "#Source", true);
    Vertex addedSink = graph.addVertex(sink.peek().getCoordinates(), "Sink", true);

    // Verbindung mit allen Startunterknoten
    for (Vertex vertex : source) {
        if (!graph.getVertices().contains(vertex))
            continue;
        Edge edge = graph.addEdge(addedSource, vertex);
        edge.setCapacity(0);
    }

    // Verbindung mit allen Zielunterknoten
    for (Vertex vertex : sink) {
        if (!graph.getVertices().contains(vertex))
            continue;
        Edge edge = graph.addEdge(vertex, addedSink);
        edge.setCapacity(0);
    }

    return new Vertex[] { addedSource, addedSink };
}
```

## Dijkstra – Klasse: Anwendung des Dijkstra Algorithmus

```
/*
 * Wendet den Dijkstra Algorithmus auf einen Graphen an und gibt die dadurch
 * erhaltenen Informationen zu den einzelnen Knoten zurück. Falls nur der
 * kürzeste Weg zu einem bestimmten Knoten gesucht ist, wird abgebrochen
 * sobald dieser gefunden wird. Andernfalls wird zu jedem Knoten der kürzeste
 * Weg bestimmt.
 */
public static HashMap<Vertex, dijkstra_object> applyDijkstra(DGraph graph, Vertex source, Vertex sink,
    boolean terminateAtSink) {
    dijkstra_object.resetIDS();

    // Allen Knoten wird die Eigenschaft unbesucht und Entfernung unendlich
    // zugewiesen
    TreeSet<dijkstra_object> vertices = new TreeSet<dijkstra_object>();
    HashMap<Vertex, dijkstra_object> objects_by_vertex = new HashMap<Vertex, dijkstra_object>();
    for (Vertex vertex : graph.getVertices()) {
        if (vertex != source) {
            dijkstra_object object = new dijkstra_object(vertex);
            objects_by_vertex.put(vertex, object);
            vertices.add(object);
        }
    }

    // Startknoten wird besucht
    dijkstra_object source_object = new dijkstra_object(source);
    source_object.setDistance(0);
    source_object.setParent(null);
    objects_by_vertex.put(source, source_object);
    vertices.add(source_object);
    // Algorithmus terminiert dann, wenn alle Knoten besucht wurden
    while (!vertices.isEmpty()) {
        // Der unbesuchte Knoten mit der kleinsten Entfernung wird ausgewählt
        dijkstra_object min_distance = vertices.pollLast();
        // Wird der Zielknoten gefunden wird sofort abgebrochen, falls erwünscht
        if (min_distance.isDistanceInfinite() || (min_distance.getVertex() == sink && !terminateAtSink))
            break;

        min_distance.setVisited();
        // Distanz des Knotens zum Quellknoten wird abgerufen
        float min_capacity = min_distance.getDistance();
        Vertex min_distance_vertex = min_distance.getVertex();

        // Anliegende Knoten werden aktualisiert
        for (Edge edge : min_distance_vertex.getOutgoingEdges()) {
            if (objects_by_vertex.get(edge.targetVertex()).isVisited())
                continue;
            objects_by_vertex.get(edge.targetVertex()).setDistance(min_capacity + edge.getCapacity());
            objects_by_vertex.get(edge.targetVertex()).setParent(min_distance);
        }
    }
}
```

```
// Distanz vom Quellknoten zu diesem Überknoten über aktuell iterierte
// Knoten wird berechnet
float distance = min_capacity + edge.getCapacity();
dijkstra_object outgoing_vertex_object = objects_by_vertex.get(edge.targetVertex());
// Aktualisierung, falls Distanz geringer
if (outgoing_vertex_object.isDistanceInfinite() || distance < outgoing_vertex_object.getDistance()) {
    vertices.remove(outgoing_vertex_object);
    outgoing_vertex_object.setDistance(distance);
    outgoing_vertex_object.setParent(min_distance_vertex);
    vertices.add(outgoing_vertex_object);
}
}

return objects_by_vertex;
}

/*
* Liefert via Backtracking den kürzesten Weg von einem Knoten zum
* Quellknoten.
*/
public static ArrayList<Vertex> getPathFromParentMap(Vertex sink, HashMap<Vertex, dijkstra_object> parentMap) {
    //Liste aller Knoten des Weges in Reihenfolge
    ArrayList<Vertex> path = new ArrayList<Vertex>(parentMap.size());
    Vertex parent = sink;
    boolean no_way_found = false;
    //Backtracking
    while (parent != null) {
        dijkstra_object o = parentMap.get(parent);
        if (o.hasNoParent()) {
            no_way_found = true;
            break;
        }
        path.add(parent);
        parent = o.getParent();
    }

    if (no_way_found)
        return null;
    return path;
}
```

## Shortest\_Path\_illegal\_turns – Klasse: Kürzester Weg mit falschem Abbiegen

```
/*
 * Liefert den kürzesten Weg zwischen zwei Knoten mit einer variablen Anzahl
 * an erlaubten falschen Abbiegen.
 */
public static ArrayList<Vertex> getPath(DGraph_all_directions graph_all, DGraph_right_turn graph_right, Vertex source,
    Vertex target, int illegal_turns) {
    [...]
    @SuppressWarnings("unchecked")
    // Speichere alle anfänglichen Unterknoten des Startknoten
    Stack<Vertex> originalSources = (Stack<Vertex>) source.getHierarchieEnd().clone();
    // Erzeuge Ebenen (Kopien des Graphen)
    ArrayList<HashSet<Vertex>> createdLayers = createLayers(graph_right, illegal_turns);
    [...]
    // Verbinde Knoten mit Zielknoten auf nächst tieferer Ebene durch Kanten,
    // die falsches Abbiegen repräsentieren
    connectLayers(graph_all, graph_right, createdLayers);
    [...]
    // Füge imaginären Startknoten und Zielknoten hinzu
    Vertex[] frameVertices = Shortest_Path.setupDijkstra(graph_right, originalSources, target.getHierarchieEnd());
    [...]
    // Dijkstra Algorithmus
    HashMap<Vertex, dijkstra_object> results = Dijkstra.applyDijkstra(graph_right, frameVertices[0], frameVertices[1],
        true);
    [...]
    // Erhalte den Weg aus den Ergebnissen des Dijkstra Algorithmus
    ArrayList<Vertex> path = Dijkstra.getPathFromParentMap(frameVertices[1], results);
    if (path != null) {
        // Falls ein Weg gefunden wurde, werden die temporären Start und
        // Zielknoten wieder entfernt
        path.remove(0);
        path.remove(path.size() - 1);
    }
    return path;
}
```

```
/*
 * Erzeugt eine Kopie aller Kanten und Knoten innerhalb des Graphen so oft wie
 * man falsch Abbiegen darf. Die Namen der Knoten aller Kopien unterscheiden
 * sich durch ein zusätzliches # am Anfang ihres Namens.
 */
private static ArrayList<HashSet<Vertex>> createLayers(DGraph_right_turn graph, int illegal_turns) {
    // Speichern aller Knoten, die in den Ebenen enthalten sind
    ArrayList<HashSet<Vertex>> layer_vertices = new ArrayList<HashSet<Vertex>>(illegal_turns + 1);
    layer_vertices.add(new HashSet<Vertex>(graph.getVertices()));
    StringBuilder prefix_builder = new StringBuilder(illegal_turns);
    ArrayList<Vertex> vertices = new ArrayList<Vertex>(graph.getVertices());
    ArrayList<Edge> edges = new ArrayList<Edge>(graph.getEdges());
    // Für jede Ebene
    for (int i = 0; i < illegal_turns; i++) {
        prefix_builder.append("#");
        String prefix = prefix_builder.toString();
        HashSet<Vertex> layer_set = new HashSet<Vertex>();
        // Füge Knoten der neuen Ebenen hinzu
        for (Vertex vertex : vertices) {
            Vertex addedVertex = graph.addVertex(vertex.getCoordinates(), prefix + vertex.getName());
            Vertex_right_turn parent = (Vertex_right_turn) vertex.getHierarchieTop();
            parent.addChild((Vertex_right_turn) addedVertex);
            layer_set.add(addedVertex);
        }
        // Füge Kanten der neuen Ebene hinzu
        for (Edge edge : edges) {
            graph.addEdge(graph.getVertex(prefix + edge.sourceVertex().getName()),
                          graph.getVertex(prefix + edge.targetVertex().getName()));
        }
        layer_vertices.add(layer_set);
    }
    // Hier existieren jetzt Kopien ohne Verbindungen untereinander
    return layer_vertices;
}
```

```

/*
 * Verbindet die verschiedenen Ebenen des Graphen durch Kanten von Unterknoten
 * zu Unterknoten in der nächsten Ebene, die illegales Abbiegen
 * repräsentieren.
 */
private static void connectLayers(DGraph_all_directions graph_all, DGraph_right_turn graph_right,
    ArrayList<HashSet<Vertex>> layers) {
    // Ist die Liste nur 1 groß, wurden keine Kopien erzeugt und es kann auch
    // nichts verbunden werden
    if (layers.size() < 2)
        return;
    // Bilden des Präfix des tiefsten Ebene - Teilstring davon werden zum
    // schnellen Zugriff benötigt
    StringBuilder builder_last_layer_prefix = new StringBuilder(layers.size());
    for (int i = 0; i < layers.size(); i++)
        builder_last_layer_prefix.append("#");
    String last_layer_prefix = builder_last_layer_prefix.toString();
    // Für jede Ebene mit Nachfolger
    for (int i = layers.size() - 2; i >= 0; i--) {
        // Für jeden Knoten jeder Ebene
        for (Vertex vertex : layers.get(i)) {
            if (vertex.getIncomingEdges().isEmpty())
                continue;
            // Ermitteln zu welchen Überknoten das Abbiegen eigentlich verboten ist
            Vertex parentVertex = graph_all.getVertex(vertex.getHierarchieTop().getName());
            Vertex incomingVertexParent = graph_all
                .getVertex(vertex.getIncomingEdges().iterator().next().sourceVertex().getHierarchieTop().getName());
            Edge incomingEdge = parentVertex.getIncomingEdge(incomingVertexParent);
            if (incomingEdge == null)
                incomingEdge = parentVertex.getOutgoingEdge(incomingVertexParent);
            Edge[] prohibitedEdges = parentVertex.getProhibitedEdges(incomingEdge);

            // Erzeuge Kante zu Unterknoten in nächster Ebene zu denen Abbiegen
            // verboten ist
            for (Edge edge : prohibitedEdges) {
                Vertex target_ver = null;
                if (edge.sourceVertex() == parentVertex)
                    target_ver = edge.targetVertex();
                else
                    target_ver = edge.sourceVertex();

                String prefix = last_layer_prefix.substring(0, i + 2);
                Vertex_right_turn nextLayerVer = null;
                int count = 0;
                // Suche korrekten Unterknoten für diese Kante
                do {
                    nextLayerVer = (Vertex_right_turn) graph_right.getVertex(prefix + target_ver.getName() + "." + (count++));
                    if (nextLayerVer != null && nextLayerVer.getIncomingEdges().iterator().next().sourceVertex()
                        .getHierarchieTop().getName().equals(parentVertex.getName())) {

```

```
        graph_right.addEdge(vertex, nextLayerVer);
        break;
    }
} while (nextLayerVer != null);
}
}
}
```

## Vertices\_not\_reachable – Klasse: Bestimmung, ob alle Knoten sich gegenseitig erreichen können

```
/*
 * Liefert, falls vorhanden, ein Knotenpaar zurück, was durch das
 * Linksabbiegeverbot sich nicht gegenseitig erreichen kann.
 */
public static Vertex[] ExistVerticesNotReachableBecauseLeftProhibition(DGraph_all_directions graph_all) {
    // Unterteilung in zusammenhängende Graphen
    Stack<DGraph_all_directions> splitGraphs = Split_graph.splitGraph(graph_all);
    for (DGraph_all_directions graph : splitGraphs) {
        // Erstelle Rechtsabbiegegraph aus zusammenhängendem Graph
        DGraph_right_turn graph_right = DGraph_right_turn.convert(graph);
        // Prüfung, ob ein Knotenpaar existiert, was im Rechtsabbiegegraph nicht
        // gegenseitig erreichbar ist.
        // In einem zusammenhängendem Graph ohne Linksabbiegeverbot können alle
        // Knoten sich gegenseitig erreichen
        Stack<Vertex[]> non_reachable_right = VerticesWhichCantReachEachOther(graph_right, false);
        if (!non_reachable_right.isEmpty())
            return new Vertex[] { non_reachable_right.peek()[0], non_reachable_right.peek()[1] };

    }
    return null;
}
```

```

/*
 * Gibt alle oder nur ein Beispiel von Knoten zurück, die sich nicht
 * gegenseitig erreichen können.
 */
public static Stack<Vertex[]> VerticesWhichCantReachEachOther(DGraph graph, boolean find_all_pairs) {
    // Speichert alle Knotenpaare, die sich gegenseitig nicht erreichen können
    Stack<Vertex[]> non_reachable_vertices = new Stack<Vertex[]>();
    // Alle Überknoten werden gespeichert
    HashSet<Vertex> hierarchie_tops = new HashSet<Vertex>();
    for (Vertex vertex : graph.getVertices()) {
        if (!hierarchie_tops.contains(vertex.getHierarchieTop()))
            hierarchie_tops.add(vertex.getHierarchieTop());
    }
    // Alle Zyklen des Graphen werden zusammengefasst
    Remove_Cycles.removeAllCycles(graph);
    // Speichert alle Knoten, die erreichbar von einem Knoten sind
    HashMap<Vertex, HashSet<Vertex>> iterated_list = new HashMap<Vertex, HashSet<Vertex>>();

    [...]
    // Liste von Knotengruppen (mehrere Knoten) von denen man alle Kreuzungen
    // erreichen kann
    HashSet<String> set_of_nodes_which_can_reach_all_other_vertices = new HashSet<String>();
    for (Vertex vertex : hierarchie_tops) {
        [...]
        // Durch Entfernen von Zyklen sind bestimmte Knoten verschmolzen.
        // Bestimmen in welcher Knotengruppe sich alle Unterknoten befinden
        Stack<Vertex> relatives_locations = new Stack<Vertex>();
        TreeSet<String> serializedRelatives = new TreeSet<String>();
        for (Vertex relative : vertex.getHierarchieEndi()) {
            Vertex currentLocation;
            if (!graph.getVertices().contains(relative))
                currentLocation = graph.getMergedVertex(relative);
            else
                currentLocation = relative;
            relatives_locations.push(currentLocation);
            serializedRelatives.add(currentLocation.getName());
        }
        // Erstellt Zeichenkette, die repräsentiert in welchen Knotengruppen sich
        // die Unterknoten des Überknoten befinden
        String vertex_distribution = "";
        for (String string : serializedRelatives.descendingSet())
            vertex_distribution += string + "/";

        if (set_of_nodes_which_can_reach_all_other_vertices.contains(vertex_distribution))
            continue;
        // Es werden alle Überknoten ermittelt, die von der aktuellen Knotengruppe
        // aus erreichbar sind
        HashSet<Vertex> reachable = getReachableVerticesFromVertexGroup(graph, relatives_locations, iterated_list);
        // Überprüfung, ob ein Überknoten nicht erreicht wurde
    }
}

```

```
if (reachable.size() < hierarchie_tops.size()) {
    HashSet<Vertex> non_reachable = new HashSet<Vertex>(hierarchie_tops);
    non_reachable.removeAll(reachable);
    // Rückgabe vom nicht gegenseitig erreichbaren Knotenpaar
    if (!find_all_pairs) {
        [...]
        non_reachable_vertices.push(new Vertex[] { vertex, non_reachable.iterator().next() });
        return non_reachable_vertices;
    } else {
        // Speichern aller nicht gegenseitig erreichbaren Knoten
        for (Vertex not_reached : non_reachable)
            non_reachable_vertices.push(new Vertex[] { vertex, not_reached });
    }
} else
    set_of_nodes_which_can_reach_all_other_vertices.add(vertex_distribution);
}

[...]

return non_reachable_vertices;
}
```

```
/*
 * Liefert alle Überknoten, die von einer Knotengruppe erreichbar sind.
 */
private static HashSet<Vertex> getReachableVerticesFromVertexGroup(DGraph graph, Stack<Vertex> source,
    HashMap<Vertex, HashSet<Vertex>> iterated_list) {

    HashSet<Vertex> visited_vertices = new HashSet<Vertex>();
    visited_vertices.addAll(source);
    // Breitensuche ausgehend von jedem Startunterknoten
    for (Vertex vertex : source) {

        if (iterated_list.containsKey(vertex)) {
            visited_vertices.addAll(iterated_list.get(vertex));
            continue;
        }

        HashSet<Vertex> iteration_list = new HashSet<Vertex>();
        iteration_list.add(vertex);
        Stack<Vertex> in_queue = new Stack<Vertex>();
        // Liste wird mit allen Knotengruppen gefüllt erreichbar von Startknoten
        // einer Breitensuche
        HashSet<Vertex> reference_to_source_set = new HashSet<Vertex>();
        iterated_list.put(vertex, reference_to_source_set);
        // Breitensuche
        while (!iteration_list.isEmpty()) {

            for (Vertex next_vertex : iteration_list) {
                for (Edge edge : next_vertex.getOutgoingEdges()) {
                    if (reference_to_source_set.contains(edge.targetVertex()))
                        continue;

                    visited_vertices.add(edge.targetVertex());

                    reference_to_source_set.add(edge.targetVertex());
                    // Falls möglich übernehmen von Informationen aus vorherigen
                    // Breitensuchen
                    if (iterated_list.containsKey(edge.targetVertex())) {
                        reference_to_source_set.addAll(iterated_list.get(edge.targetVertex()));
                        visited_vertices.addAll(iterated_list.get(edge.targetVertex()));
                    } else
                        in_queue.push(edge.targetVertex());
                }
            }
        }

        iteration_list.clear();
        iteration_list.addAll(in_queue);
        in_queue.clear();
    }
}
```

```
    }

}

// Ermitteln aller erreichten Überknoten
HashSet<Vertex> reachable = new HashSet<Vertex>();
for (Vertex vertex : visited_vertices) {
    reachable.add(vertex.getHierarchieTop());
    reachable.addAll(vertex.getMergedTops());
}

return reachable;
}
```

## Remove\_Cycles – Klasse: Entfernen aller Zyklen aus einem Graph

```
/*
 * Entfernt alle Zyklen aus einem Graphen.
 */
static void removeAllCycles(DGraph graph) {
    HashSet<Vertex> whiteSet = new HashSet<Vertex>(graph.getVertices());
    HashSet<Vertex> blackSet = new HashSet<Vertex>();
    while (removeCycleIteration(graph, whiteSet, blackSet));
}
```

```
/*
 * Findet und entfernt einen Zyklus aus einem Graphen. Liefert zurück, ob ein
 * Zyklus entfernt werden konnte. Knoten, die als nicht zu einem Zyklus
 * gehörend identifiziert wurden, werden gespeichert. Knoten, die noch nicht
 * untersucht wurden, werden gespeichert.
 */
private static boolean removeCycleIteration(DGraph graph, HashSet<Vertex> stillWhite, HashSet<Vertex> ardyBlack) {
    // Alle unerforschten Knoten
    HashSet<Vertex> whiteSet = stillWhite;
    // Alle Knoten die gerade erforscht werden
    HashSet<Vertex> graySet = new HashSet<Vertex>();
    // Alle Knoten, die nicht zu einem Zyklus gehören
    HashSet<Vertex> blackSet = ardyBlack;
    // Speichert welcher Knoten von welchem Knoten aufgerufen wurde (Für
    // Backtracking)
    HashMap<Vertex, Vertex> parentMap = new HashMap<Vertex, Vertex>();
    // Der Algorithmus läuft solange bis alle Knoten erforscht wurden
    while (whiteSet.size() > 0) {
        // Ein Knoten aus der unerforschten Liste wird gewählt
        Vertex current = whiteSet.iterator().next();
        // Tiefensuche probiert einen Zyklus zu finden
        Vertex cycle_end = dfs(null, current, whiteSet, graySet, blackSet, parentMap);
        // cycle_end ist null, wenn kein Zyklus gefunden wurde
        if (cycle_end != null) {
            // Alle Knoten des Zyklus werden zum Knoten, der als Ende des Zyklus
            // markiert wurde, zusammengefügt
            removeCycleFromGraphWithMap(graph, cycle_end, parentMap);
            // Alle Knoten, die sich noch in der grauen Liste befinden, werden
            // wieder in die weiße Liste verschoben
            for (Vertex gray : new ArrayList<Vertex>(graySet)) {
                if (graph.getVertices().contains(gray))
                    moveVertex(gray, graySet, whiteSet);
            }
            // Falls ein Zyklus gefunden wurde wird abgebrochen und dies
            // entsprechend zurückgegeben
            return true;
        }
    }
    // Rückgabe, dass kein Zyklus gefunden wurde
    return false;
}
```

```
/*
 * Fasst einen Zyklus mittels Backtracking zusammen.
 */
private static void removeCycleFromGraphWithMap(DGraph graph, Vertex cycle_end, HashMap<Vertex, Vertex> parentMap) {
    // Liste der Knoten, die verschmolzen werden.
    ArrayList<Vertex> merge = new ArrayList<Vertex>();

    // Backtracking
    Vertex child = parentMap.get(cycle_end);
    while (child != cycle_end) {
        merge.add(child);
        child = parentMap.get(child);
    }
    // Liste mit den Zyklusknoten wird als Argument an die Methode zum
    // Verschmelzen dieser geliefert
    graph.mergeVertices(cycle_end, merge); // Der Zyklus wird zusammengefasst
}
```

```
/*
 * Tiefensuche. Alle gefunden Knoten während der Tiefensuche werden
 * gespeichert. Findet die Tiefensuche einen bereits aufgerufenen Knoten, muss
 * ein Zyklus existieren.
 */
private static Vertex dfs(Vertex parent, Vertex current, HashSet<Vertex> whiteSet, HashSet<Vertex> graySet,
    HashSet<Vertex> blackSet, HashMap<Vertex, Vertex> parentMap) {
    // Eintrag in die Backtrackingliste
    parentMap.put(current, parent);
    // Knoten wird in die Liste der zu erforschenden Knoten verschoben
    moveVertex(current, whiteSet, graySet);
    for (Edge edge : current.getOutgoingEdges()) {
        // Die Zielknoten aller ausgehenden Kanten werden untersucht
        Vertex neighbor = edge.targetVertex();
        // Falls der gefundene Knoten sich in der schwarzen Liste befindet, wurde
        // dieser bereits erforscht und ist daher uninteressant
        if (blackSet.contains(neighbor)) {
            continue;
        }
        // Falls sich der gefundene Knoten schon in der Liste der zu erforschenden
        // Knoten befindet (graue Liste), heißt es, dass ein Zyklus existiert
        if (graySet.contains(neighbor)) {
            parentMap.put(neighbor, current);
            return neighbor;
        }
        // Hier befindet sich der angrenzende Knoten in der weißen Liste.
        // Die Tiefensuche wird von diesem aus weiter fortgeführt.
        Vertex cycle_end = dfs(current, neighbor, whiteSet, graySet, blackSet, parentMap);
        if (cycle_end != null)
            return cycle_end;
    }
    // An dieser Stelle gehört der untersuchte Knoten nicht zu einem
    // Zyklus und wird daher in die schwarze Liste verschoben
    moveVertex(current, graySet, blackSet);
    return null;
}

/*
 * Verschiebt einen Knoten aus einer Liste in eine andere Liste. Der Knoten
 * wird aus einer Liste entfernt und einer anderen hinzugefügt.
 */
private static void moveVertex(Vertex vertex, HashSet<Vertex> sourceSet, HashSet<Vertex> destinationSet) {
    sourceSet.remove(vertex);
    destinationSet.add(vertex);
}
```

## Split\_graph – Klasse: Unterteilung in zusammenhängende Teilgraphen

```
/*
 * Teilt einen Graphen in einzelne zusammenhängende Graphen auf.
 */
public static Stack<DGraph_all_directions> splitGraph(DGraph_all_directions graph) {
    Stack<DGraph_all_directions> splitGraphs = new Stack<DGraph_all_directions>();
    HashSet<Vertex> unexplored_vertices = new HashSet<Vertex>(graph.getVertices());
    while (!unexplored_vertices.isEmpty()) {
        Vertex nextVertex = unexplored_vertices.iterator().next();
        // Bestimme Knoten, die mit ausgewähltem Knoten verbunden sind
        HashSet<Vertex> connectedVertices = getVerticesConnectedToVertex(nextVertex);
        unexplored_vertices.removeAll(connectedVertices);
        // Bilde Graph aus zusammenhängenden Knoten
        DGraph_all_directions splitGraph = new DGraph_all_directions();
        for (Vertex vertex : connectedVertices)
            splitGraph.addVertex(vertex.getCoordinates(), vertex.getName(), false);
        for (Vertex vertex : connectedVertices) {
            for (Edge edge : vertex.getOutgoingEdges())
                splitGraph.addEdge(splitGraph.getVertex(edge.sourceVertex().getName()),
                                   splitGraph.getVertex(edge.targetVertex().getName()));
        }
        splitGraphs.push(splitGraph);
    }
    return splitGraphs;
}
```

```
/*
 * Liefert alle Knoten, die mit einem Knoten zusammenhängen. Optional werden
 * nur die Knoten geliefert, die auch erreichbar sind. Dies ist mittels
 * Breitensuche umgesetzt.
 */
private static HashSet<Vertex> getVerticesConnectedToVertex(Vertex source) {
    HashSet<Vertex> visited_vertices = new HashSet<Vertex>();
    HashSet<Vertex> iteration_list = new HashSet<Vertex>();
    visited_vertices.add(source);
    iteration_list.add(source);
    Stack<Vertex> in_queue = new Stack<Vertex>();
    // Breitensuche
    while (!iteration_list.isEmpty()) {
        for (Vertex vertex : iteration_list) {
            for (Edge edge : vertex.getIncomingEdges()) {
                if (visited_vertices.contains(edge.sourceVertex()))
                    continue;
                visited_vertices.add(edge.sourceVertex());
                in_queue.push(edge.sourceVertex());
            }
            for (Edge edge : vertex.getOutgoingEdges()) {
                if (visited_vertices.contains(edge.targetVertex()))
                    continue;
                visited_vertices.add(edge.targetVertex());
                in_queue.push(edge.targetVertex());
            }
        }
        iteration_list.clear();
        iteration_list.addAll(in_queue);
        in_queue.clear();
    }
    return visited_vertices;
}
```

## BIGGEST\_DIFFERENCE\_HEURISTIC – Klasse: Bestimmung des größten Faktors (Heuristisch)

```
/*
 * Liefert heuristisch den größten Unterschied, entstanden durch das
 * Linksabbiegeverbot, und das dazugehörige Knotenpaar.
 */
public static difference_object getBiggestDifference(DGraph_all_directions graph_all, DGraph_right_turn graph_right) {
    [...]
    // Überprüfung, ob Unterschied im Graphen überhaupt existieren kann
    if (graph_all.getVertices().size() < 2) {
        [...]
        difference_object substitute = new difference_object(null, null);
        substitute.setDifference(1);
        return substitute;
    }
    // Überprüfung, ob mindestens zwei Knoten durch das Linksabbiegeverbot nicht
    // mehr erreichbar sind
    Vertex[] infinite_difference = Vertices_not_reachable.ExistVerticesNotReachableBecauseLeftProhibition(graph_all);
    if (infinite_difference != null) {
        [...]
        return new difference_object(infinite_difference[0], infinite_difference[1]);
    }
    graph_all.createInverseEdges();
    [...]
    difference_object biggest_difference = null;
    // Iterierung aller Knoten des Graphen
    for (Vertex vertex : new ArrayList<Vertex>(graph_all.getVertices())) {
        [...]
        // Größter möglicher Unterschied von diesem Knoten zu einem anderen Knoten
        // wird ermittelt
        difference_object biggest_difference_current_vertex = bfsDifference(graph_all, graph_right, vertex);

        // Speicherung des bisher größten gefundenen Unterschieds
        if (biggest_difference == null || biggest_difference.compareTo(biggest_difference_current_vertex) == 1)
            biggest_difference = biggest_difference_current_vertex;
    }
    [...]
    return biggest_difference;
}
```

```

/*
 * Gibt einen möglichen größten Unterschied von einem Knoten zu einem anderen
 * Knoten des Graphen zurück.
 */
private static difference_object bfsDifference(DGraph_all_directions graph_all, DGraph_right_turn graph_right,
Vertex source) {

    dijkstra_object.resetIDS();

    // Speichert alle Knoten und ihren Unterschied zum Quellknoten 'source'
    HashMap<Vertex, difference_object> visited_vertices = new HashMap<Vertex, difference_object>();

    ArrayList<Vertex> iteration_list = new ArrayList<Vertex>();
    HashSet<Vertex> in_queue = new HashSet<Vertex>();

    // Vorbereitung Breitensuche
    iteration_list.add(source);
    difference_object source_object = new difference_object(source, source);
    source_object.setDifference(1);
    visited_vertices.put(source, source_object);

    difference_object biggest_difference = null;

    // Speicherung aller Daten für Dijkstra im Beliebigabbiegegraph
    TreeSet<dijkstra_object> vertex_data_all = new TreeSet<dijkstra_object>();
    HashMap<Vertex, dijkstra_object> vertex_data_map_all = new HashMap<Vertex, dijkstra_object>();
    HashMap<Vertex, dijkstra_object> vertex_data_tops_map_all = new HashMap<Vertex, dijkstra_object>();
    // Speicherung aller Daten für Dijkstra im Rechtsabbiegegraph
    TreeSet<dijkstra_object> vertex_data_right = new TreeSet<dijkstra_object>();
    HashMap<Vertex, dijkstra_object> vertex_data_map_right = new HashMap<Vertex, dijkstra_object>();
    HashMap<Vertex, dijkstra_object> vertex_data_tops_map_right = new HashMap<Vertex, dijkstra_object>();

    // Hinzufügen von extra Knoten, verbunden mit allen Unterknoten des
    // Startknotens, als Vorbereitung für den Dijkstra
    Vertex addedVertexAll = bfsDijkstraSetup(graph_all, vertex_data_map_all, vertex_data_all, source);
    Vertex addedVertexRight = bfsDijkstraSetup(graph_right, vertex_data_map_right, vertex_data_right,
        graph_right.getVertex(DGraph_right_turn.generateVertexname(source.getName(), 0)));

    while (!iteration_list.isEmpty()) {

        for (Vertex vertex : iteration_list) {
            // Der Unterschied vom 'source' Knoten zum aktuellen Knoten wird
            // abgerufen
            difference_object difference_of_current_vertex = visited_vertices.get(vertex);
            for (Edge edge : vertex.getOutgoingEdges()) { // Allen Kanten des
                // aktuellen Knoten
                // iterieren
                if (visited_vertices.containsKey(edge.targetVertex()))
                    continue;
            }
        }
    }
}

```

```

// Zur aktuell betrachteten Kante wird die Weglänge des 'source'
// Knoten zum Zielknoten der Kante im Beliebigabbiegraph wird
// ermittelt
dijkstra_object dijk_all = vertex_data_map_all.get(bfsDijkstra(vertex_data_tops_map_all,
    vertex_data_map_all, vertex_data_all, edge.targetVertex()));
// Zur aktuell betrachteten Kante wird die Weglänge des 'source'
// Knoten zum Zielknoten der Kante im Rechtsabbiegraph wird
// ermittelt
dijkstra_object dijk_right = vertex_data_map_right
    .get(bfsDijkstra(vertex_data_tops_map_right, vertex_data_map_right, vertex_data_right,
        graph_right.getVertex(DGraph_right_turn.generateVertexname(edge.targetVertex().getName(), 0))
        .getHierarchieTop()));
// Faktor aus den Weglängen wird gebildet
difference_object difference_of_neighbor_vertex = getDifferenceObjectFromDijkstra(source, edge.targetVertex(),
    dijk_all, dijk_right);

// Überprüfung, ob der Unterschied des anliegenden Knoten größter oder
// gleich dem des aktuellen Knotens ist
if (difference_of_current_vertex.compareTo(difference_of_neighbor_vertex) != -1) {
    visited_vertices.put(edge.targetVertex(), difference_of_neighbor_vertex);
    // Speicherung des bisher größten gefundenen Unterschieds
    if (biggest_difference == null || biggest_difference.compareTo(difference_of_neighbor_vertex) == 1) {
        biggest_difference = difference_of_neighbor_vertex;
    }
    in_queue.add(edge.targetVertex());
}
}

iteration_list.clear();
iteration_list.addAll(in_queue);
in_queue.clear();
}

// Ursprungszustand des Graphen wiederherstellen
graph_all.removeVertex(addedVertexAll);
graph_right.removeVertex(addedVertexRight);

if (biggest_difference == null)
    return source_object;
return biggest_difference;
}

```

```
/*
 * Bildet aus zwei Weglängen den Unterschiedsfaktor. Ist eines der
 * Eingabedijkstraobjekte gleich null ist dies gleich bedeutend damit, dass
 * der Zielknoten eines Weges, den das Dijkstra Objekt eigentlich
 * repräsentieren sollte, nicht erreicht werden konnte.
 */
private static difference_object getDifferenceObjectFromDijkstra(Vertex source, Vertex sink,
    dijkstra_object dijkstra_object_alldirections, dijkstra_object dijkstra_object_right) {
    difference_object difference = new difference_object(source, sink);
    // Beide Weglängen gleich unendlich = Knoten können sich in keinem Graph
    // erreichen = Faktor 1
    if (dijkstra_object_alldirections == null && dijkstra_object_right == null) {
        difference.setDifference(1);
        return difference;
    }
    // Ist nur eine Weglänge unendlich, ist auch der Faktor unendlich
    if (dijkstra_object_alldirections == null) // Dürfte nicht vorkommen, da
                                                // hier mehr erreicht wird als
                                                // im Rechtsabbiegegraph
        return difference;
    if (dijkstra_object_right == null)
        return difference;
    // Faktor berechnen
    difference.setDifference(dijkstra_object_right.getDistance() / dijkstra_object_alldirections.getDistance());
    return difference;
}
```

```
/*
 * Bereitet Anwendung des Dijkstra Algorithmus vor indem ein Knoten, verbunden
 * mit allen Unterknoten eines Startknotens, mit Kantenkapazitäten von 0
 * erstellt werden. Gibt den hinzugefügten Knoten zurück.
 */
private static Vertex bfsDijkstraSetup(DGraph graph, HashMap<Vertex, dijkstra_object> vertex_data_map,
    TreeSet<dijkstra_object> vertex_data, Vertex source) {
    // Füge Knoten hinzu
    Vertex addedSource = graph.addVertex(new Point(0, 0), "#Source", true);
    // Verbinde neuen Knoten mit Unterknoten des Startknoten
    for (Vertex vertex : source.getHierarchieEnd()) {
        if (!graph.getVertices().contains(vertex))
            continue;
        Edge edge = graph.addEdge(addedSource, vertex);
        edge.setCapacity(0);
    }
    // Einträge in die Listen für Dijkstra Algorithmus
    dijkstra_object source_object = new dijkstra_object(addedSource);
    source_object.setDistance(0);
    source_object.setParent(null);
    vertex_data_map.put(addedSource, source_object);
    vertex_data.add(source_object);
    return addedSource;
}
```

```

/*
 * Anwendung des Dijkstra Algorithmus bis ein bestimmter Knoten gefunden wird.
 * Führt und verwendet Informationen bezüglich aller bisher erreichter Knoten.
 * Eingabeparamter:
 * vertexTops: Überknoten und deren Entfernung zum Ursprungsknoten
 * vertex_data_map: Unterknoten und deren Entfernung zum Ursprungsknoten
 * vertex_data: Geordnete Liste mit Entfernungsobjekten
 * parentTop: Überknoten zu dem Weg gesucht ist
 */
private static Vertex bfsDijkstra(HashMap<Vertex, dijkstra_object> vertexTops,
    HashMap<Vertex, dijkstra_object> vertex_data_map, TreeSet<dijkstra_object> vertex_data, Vertex parentTop) {

    // Wurde bereits ein Unterknoten des gesuchten Überknotens gefunden, ist der
    // kürzeste Weg bereits gefunden
    if (vertexTops.containsKey(parentTop))
        return vertexTops.get(parentTop).getVertex();

    while (!vertex_data.isEmpty()) {
        // Unbesuchter Knoten mit geringster Entfernung wird ausgewählt
        dijkstra_object min_distance = vertex_data.last();
        // Ist Knoten gesuchter Knoten wird Algorithmus terminiert
        if (min_distance.getVertex().getHierarchieTop() == parentTop)
            return min_distance.getVertex();

        // Knoten wird besucht
        vertex_data.remove(min_distance);
        min_distance.setVisited();
        // Falls Knoten der Unterknoten mit der geringsten Entfernung ist, wird
        // seine Entfernung im Überknoten eingetragen
        dijkstra_object hierarchieTop = vertexTops.get(min_distance.getVertex().getHierarchieTop());
        if (hierarchieTop == null || min_distance.compareTo(hierarchieTop) == 1) //=1, wenn neuer Weg kleiner als bisheriger Weg
            vertexTops.put(min_distance.getVertex().getHierarchieTop(), min_distance);

        // Distanz dieses Knotens zum Ursprungsknoten
        float distance_of_vertex = min_distance.getDistance();
        Vertex min_distance_vertex = min_distance.getVertex();
        // Distanz zum Ursprungsknoten wird bei allen anliegenden Knoten
        // aktualisiert
        for (Edge edge : min_distance_vertex.getOutgoingEdges()) {
            // Abrufen der Distanz des anliegenden Knoten
            dijkstra_object nearbyVertexData = vertex_data_map.get(edge.targetVertex());
            // Wurde anliegender Knoten noch nicht entdeckt, wird dieser nun in
            // Listen eingetragen
            if (nearbyVertexData == null) {
                dijkstra_object data = new dijkstra_object(edge.targetVertex());
                vertex_data.add(data);
                vertex_data_map.put(edge.targetVertex(), data);
            }
            // Wurde anliegender Knoten bereits besucht, wird dieser ignoriert
        }
    }
}

```

```
else if (nearbyVertexData.isVisited())
    continue;
// Distanz vom Quellknoten zu anliegenden Knoten wird ermittelt
float distance = distance_of_vertex + edge.getCapacity();
dijkstra_object outgoing_vertex_object = vertex_data_map.get(edge.targetVertex());
// Falls Distanz kleiner als die gespeicherte Distanz des Knoten wird
// aktualisiert
if (outgoing_vertex_object.isDistanceInfinite() || distance < outgoing_vertex_object.getDistance()) {
    vertex_data.remove(outgoing_vertex_object);
    outgoing_vertex_object.setDistance(distance);
    outgoing_vertex_object.setParent(min_distance_vertex);
    vertex_data.add(outgoing_vertex_object);
}
}
return null;
}
```

## **Bigest\_Difference\_Bruteforce – Klasse: Bestimmung des größten Faktors (Brute-Force)**

```
/*
 * Liefert den größten Unterschied, entstanden durch
 * das Linksabbiegeverbot und das
 * dazugehörige Knotenpaar.
 */
public static difference_object getBigestDifference(DGraph_all_directions graph_all, DGraph_right_turn graph_right){

    [...]

    //Überprüfung, ob Unterschied im Graphen überhaupt existieren kann
    if(graph_all.getVertices().size() < 2){
        [...]
        difference_object substitute = new difference_object(null,null);
        substitute.setDifference(1);
        return substitute;
    }

    //Überprüfung, ob mindestens zwei Knoten durch das Linksabbiegeverbot nicht mehr erreichbar sind
    Vertex[] infinite_difference = Vertices_not_reachable.ExistVerticesNotReachableBecauseLeftProhibition(graph_all);
    if(infinite_difference != null){
        [...]
        return new difference_object(infinite_difference[0],infinite_difference[1]);
    }
    graph_all.createInverseEdges();

    int percent = graph_all.getVertices().size()/100;
    if(percent == 0) percent = 1;
    int progress = 0;

    //Iterierung aller Knoten des Graphen
    difference_object biggest_difference = null;
    for(Vertex vertex : graph_all.getVertices()){
        [...]
        //Größter Unterschied von diesem Knoten zu einem anderen Knoten wird ermittelt
        difference_object biggest_difference_current_vertex = getBigestDifferenceSingleVertex(graph_all,graph_right,vertex);
        //Speicherung des bisher größten gefundenen Unterschieds
        if(biggest_difference == null || biggest_difference.compareTo(biggest_difference_current_vertex) == 1)
            biggest_difference = biggest_difference_current_vertex;

    }
    [...]
    return biggest_difference;
}
```

```

/*
 * Sucht den den größten Unterschied,
 * der von einem ausgewählten Knoten
 * zu einem anderen Knoten des
 * Graphen existiert.
 */
private static difference_object getBiggestDifferenceSingleVertex(DGraph_all_directions graph_all, DGraph_right_turn graph_right, Vertex source){
    //Die Entfernung des kürzesten Weg von "source" Knoten zu allen anderen Knoten des Graphen im Beliebigabbiegegraph wird berechnet
    HashMap<Vertex,dijkstra_object> mapAll = Dijkstra.applyDijkstra(graph_all,source,null,false);
    //Die Entfernung des kürzesten Weg von "source" Knoten zu allen anderen Knoten des Graphen im Rechtsabbiegegraph wird berechnet
    HashMap<Vertex,dijkstra_object> mapNoLeft =
        getMapRightProhibition(graph_right,graph_right.getVertex(DGraph_right_turn.generateVertexname(source.getName(),0)));
    difference_object biggest_difference = null;

    //Es wird die Weglänge zu jedem Knoten mit und ohne Linksabbiegeverbot berechnet und dividiert
    for(Vertex vertex : graph_all.getVertices()){
        if(vertex == source) continue;

        //Die Weglänge im Beliebigabbiegegraph wird abgerufen
        dijkstra_object all_directions = mapAll.get(vertex);
        //Für jeden Überknoten existiert eine bestimmte Anzahl an Unterknoten im Rechtsabbiegegraph
        //Es wird die Weglänge des Unterknotens gewählt zu welchem die Weglänge am kürzesten ist
        dijkstra_object no_left = null;
        for(Vertex relative : ((Vertex_right_turn)graph_right.getVertex(DGraph_right_turn.generateVertexname(vertex.getName(),0))).getRelatives()){
            if(no_left == null || mapNoLeft.get(relative).compareTo(no_left) == 1)
                no_left = mapNoLeft.get(relative);
        }
        //Aus beiden Weglängen wird der Unterschied gebildet
        difference_object difference_current_vertex = getDifferenceObjectFromDijkstra(source,vertex,all_directions,no_left);
        //Speicherung des bisher größten gefundenen Unterschieds
        if(biggest_difference == null || biggest_difference.compareTo(difference_current_vertex) == 1)
            biggest_difference = difference_current_vertex;
    }
    return biggest_difference;
}

```

```
/*
 * Bildet aus zwei Weglängen den Unterschiedsfaktor
 */
private static difference_object getDifferenceObjectFromDijkstra(Vertex source, Vertex sink, dijkstra_object dijkstra_object_source,
dijkstra_object dijkstra_object_sink){
    difference_object difference = new difference_object(source,sink);
    //Sind beide Weglängen unendlich so ist der Unterschiedsfaktor 1, weil die Knoten sich nicht erreichen können, selbst wenn man beliebig
abbiegt
    if(dijkstra_object_source.isDistanceInfinite() && dijkstra_object_sink.isDistanceInfinite()){
        difference.setDifference(1);
        return difference;
    }
    //Ist eine der Distanzen Unendlich und die andere nicht, kann das oben erzeugte Objekt zurückgegeben werden, da dieses mit unendlich
Unterschied initialisiert wurde
    if(dijkstra_object_source.isDistanceInfinite())
        return difference;
    if(dijkstra_object_sink.isDistanceInfinite())
        return difference;
    //Quotient wird in Unterschiedsobjekt errechnet
    difference.setDifference(dijkstra_object_sink.getDistance()/dijkstra_object_source.getDistance());
    return difference;
}

/*
 * Gibt eine Liste mit den kürzesten
 * Wegen zu allen Knoten im Rechtsabbiegegraph zurück
 */
private static HashMap<Vertex,dijkstra_object> getMapRightProhibition(DGraph_right_turn graph, Vertex source){
    //Knoten, verbunden mit allen Unterknoten des Startknoten, wird erstellt
    Vertex addedSource = graph.addVertex(new Point(0,0), "#Source");
    for(Vertex vertex : source.getHierarchieEnd()){
        if(!graph.getVertices().contains(vertex)) continue;
        Edge edge = graph.addEdge(addedSource,vertex);
        edge.setCapacity(0);
    }
    //Erhalte über Dijkstra Algorithmus kürzesten Weglängen zu allen Knoten
    HashMap<Vertex,dijkstra_object> result = Dijkstra.applyDijkstra(graph, addedSource, null, false);
    //Stelle Ursprungszustand wieder her
    graph.removeVertex(addedSource);
    return result;
}
```

## MapGenerator – Klasse: Erzeugen von zufällig generierten Straßennetzen

```
/*
 * Erzeugt einen zufälligen Graph mit gegebenen Rahmenbedingungen, welcher in
 * das GUI geladen und optional in eine Datei gepseichert wird.
 */
public static void generateMap(int x_range, int y_range, int vertex_count, int edge_count, long seed) {
    final Random generator = new Random(seed);

    DGraph_all_directions random_graph = new DGraph_all_directions();

    // Generiere zufällig platzierte Knoten
    generateVertices(generator, random_graph, x_range, y_range, vertex_count);

    // Generiere zufällig verbundene Kanten, die
    // sich nicht überschneiden
    addEdges(generator, random_graph, edge_count);

    // Passe Graphen so an, dass alle Knoten sich gegenseitig
    // erreichen können
    makeAllVerticesReachable(random_graph);

    [...]

    // Speichern ,falls erwünscht, in Datei
    try {
        if (JOptionPane.showConfirmDialog(null, "Erzeugten Graphen speichern?") == 0)
            DGraph.save(random_graph, JOptionPane.showInputDialog("Wie soll die Datei mit dem erzeugten Graphen heißen?"));
    } catch (Exception e) {
    }
}
```

```
/*
 * Passt Graph so an, dass sich alle Knoten gegenseitig erreichen können.
 */
private static void makeAllVerticesReachable(DGraph_all_directions graph) {
    Stack<Vertex[]> pairs = null;
    // Solange Knoten existieren, die sich nicht gegenseitig erreichen können
    // werden Knoten entfernt
    do {
        // Bestimme alle Knoten, die sich nicht gegenseitig erreichen können
        pairs = Vertices_not_reachable.VerticesWhichCantReachEachOther(Solver.DGraph_right_turn.convert(graph), true);
        if (pairs.isEmpty())
            continue;
        HashMap<Vertex, Integer> appearances = new HashMap<Vertex, Integer>();
        // Zähle wie oft jeder Knoten in der Liste vorkommt
        for (Vertex[] pair : pairs) {
            for (Vertex vertex : pair) {
                if (!appearances.containsKey(vertex))
                    appearances.put(vertex, 1);
                else
                    appearances.put(vertex, appearances.get(vertex) + 1);
            }
        }

        int highest_value = -1;
        for (Map.Entry<Vertex, Integer> entry : appearances.entrySet()) {
            if (entry.getValue() > highest_value)
                highest_value = entry.getValue();
        }
        // Entferne alle Knoten aus dem Graphen, die am häufigsten in der Liste
        // vorkommen
        for (Map.Entry<Vertex, Integer> entry : appearances.entrySet()) {
            if (entry.getValue() == highest_value)
                graph.removeVertex(graph.getVertex(entry.getKey().getName()));
        }
    } while (!pairs.isEmpty());
}
```

```
/*
 * Erzeuge zufällige, sich nicht überschneidende Kanten.
 */
private static void addEdges(final Random generator, DGraph_all_directions random_graph, int edge_count) {
    ArrayList<String> vertex_names = new ArrayList<String>(random_graph.getVertices().size());
    for (Vertex vertex : random_graph.getVertices())
        vertex_names.add(vertex.getName());
    Collections.sort(vertex_names);
    int iterations = 0;
    while (random_graph.getEdges().size() < edge_count) {
        if (iterations++ >= 1000)
            break;
        // Erhalte zwei zufällige Knoten
        Vertex source = random_graph.getVertex(vertex_names.get((int) (generator.nextFloat() * vertex_names.size())));
        Vertex target = random_graph.getVertex(vertex_names.get((int) (generator.nextFloat() * vertex_names.size())));
        if (target == source)
            continue;
        // Erhalte Liste an Knoten, die Kante zwischen den beiden zufälligen
        // Knoten schneidet
        ArrayList<Vertex> intersection_list = getIntersectingVertices(random_graph, new Edge(source, target));
        // Erzeuge Kanten zwischen allen Knoten in der Liste
        for (int j = 0; j < intersection_list.size() - 1; j++) {
            boolean intersecting_edge = false;
            // Erzeuge Kante, dann wenn sie keine andere Kante des Graphen schneidet
            for (Edge edge : random_graph.getEdges()) {
                if (edge.sourceVertex() == intersection_list.get(j) || edge.targetVertex() == intersection_list.get(j + 1)
                    || edge.sourceVertex() == intersection_list.get(j + 1) || edge.targetVertex() == intersection_list.get(j))
                    continue;
                if (new Line2D.Float(edge.sourceVertex().getCoordinates(), edge.targetVertex().getCoordinates())
                    .intersectsLine(new Line2D.Float(intersection_list.get(j).getCoordinates(),
                        intersection_list.get(j + 1).getCoordinates())))
                    {
                        intersecting_edge = true;
                        break;
                    }
            }
            // Füge Kante hinzu
            if (!intersecting_edge
                && !intersection_list.get(j).incomingVerticesContainsVertex(intersection_list.get(j + 1))) {
                if (random_graph.addEdge(intersection_list.get(j), intersection_list.get(j + 1)) != null)
                    iterations = 0;
            }
        }
    }
}
```

```
/*
 * Liefert eine Liste an Knoten, die auf einer Kante liegen. Da der Generator
 * nur ganzzahlige Koordinaten liefert, muss nur mittels ggT das minimale
 * deltaX und deltaY gebildet werden und dann vom Start mit diesem delta alle
 * Zwischenpunkten abgearbeitet werden.
 */
private static ArrayList<Vertex> getIntersectingVertices(DGraph graph, Edge edge) {
    ArrayList<Vertex> intersection_list = new ArrayList<Vertex>(graph.getVertices().size());
    intersection_list.add(edge.sourceVertex());
    // deltaX und deltaY der Kante bestimmen
    int deltaX = (int) (edge.targetVertex().getCoordinates().getX() - edge.sourceVertex().getCoordinates().getX());
    int deltaY = (int) (edge.targetVertex().getCoordinates().getY() - edge.sourceVertex().getCoordinates().getY());
    // ggT bestimmen und deltaX und deltaY durch ggT teilen
    // falls ggT=0 (deltaX = 0 || deltaY = 0) wird nur die jeweilige Achse
    // betrachtet
    int ggT = ggT(Math.abs(deltaX), Math.abs(deltaY));
    if (ggT > 0) {
        deltaX /= ggT;
        deltaY /= ggT;
    } else {
        if (deltaX == 0) {
            deltaY /= Math.abs(deltaY);
        } else {
            deltaX /= Math.abs(deltaX);
        }
    }
    // Startwerte setzen
    int x = edge.sourceVertex().getCoordinates().x;
    int y = edge.sourceVertex().getCoordinates().y;
    // Alle Zwischenpunkte bis zum Ziel erzeugen
    while (x != edge.targetVertex().getCoordinates().x || y != edge.targetVertex().getCoordinates().y) {
        x += deltaX;
        y += deltaY;
        if (graph.getVertex(x + "/" + y) != null)
            intersection_list.add(graph.getVertex(x + "/" + y));
    }
    return intersection_list;
}
```

```
/*
 * Liefert den größten gemeinsamen Teiler zweier Zahlen (Euklidischer
 * Algorithmus).
 */
private static int ggT(int i, int j) {
    if (i == 0 || j == 0)
        return -1;
    int k = -1;
    while (k != 0) {
        k = i % j;
        i = j;
        j = k;
    }
    return i;
}
```

```
/*
 * Erzeugt zufällig platzierte Knoten.
 */
private static void generateVertices(final Random generator, DGraph_all_directions random_graph, int x_range,
    int y_range, int vertex_count) {
    // Bei weniger als Hälfte der möglichen Knotenplatzierungen werden Knoten
    // zufällig platziert
    if (vertex_count < x_range * y_range / 2) {
        HashSet<String> placedVertices = new HashSet<String>();
        for (int i = 0; i < vertex_count; i++) {
            int x, y;
            String serializedPos;
            do {
                x = (int) (generator.nextFloat() * x_range);
                y = (int) (generator.nextFloat() * y_range);
                serializedPos = x + "/" + y;
            } while (placedVertices.contains(serializedPos));
            placedVertices.add(serializedPos);
            random_graph.addVertex(new Point(x, y), serializedPos, false);
        }
    } else {
        // Bei mehr als Hälfte der möglichen Knotenplatzierungen werden zunächst
        // allen möglichen Platzierungen gesetzt und dann solange Platzierungen
        // entfernt bis die die gewünschte Anzahl erreicht ist
        for (int i = 1; i <= x_range; i++) {
            for (int j = 1; j <= y_range; j++)
                random_graph.addVertex(new Point(i, j), i + "/" + j, false);
        }
        int remove_count = x_range * y_range - vertex_count;
        for (int i = 0; i < remove_count; i++) {
            Vertex vertex = null;
            while (vertex == null)
                vertex = random_graph.getVertex(
                    (int) (generator.nextFloat() * x_range + 1) + "/" + (int) (generator.nextFloat() * y_range + 1));
            random_graph.removeVertex(vertex);
        }
    }
}
```

## 12. Bedienungsanleitung für das GUI

Um das Programm zu bedienen und um das in Aufgabenstellung beschriebene Navigationssystem zu entwerfen, wird eine Benutzeroberfläche benötigt.

Anforderungen an die Benutzeroberfläche:

- Straßen netze aus Dateien laden
- Darstellung eines Straßen netzes als kantengewichteter Graph mit variabler Größe
- Schnelle und wiederholte Anwendung der Aufgabenteile (Navigationssystem)
- Visualisierung der Ergebnisse der Aufgabenteile
- Fehlerhaftes Verhalten des Nutzers größtenteils verhindern (DAU)
- Eigenes Erstellen von Straßen netzen

Die Benutzeroberfläche sollte zwar möglichst selbsterklärend sein, jedoch gebe ich hier für die Anwendung eine kleine Anleitung mit.

Um Graphen mit kleiner und großer Größe darzustellen, erlaubt das Programm ein Heraus- und Hereinzoomen sowie das Bewegen innerhalb des Graphen. Zoomen ist möglich mithilfe des Mausrads. Man kann sich innerhalb des Graphen bewegen in dem man an einer beliebigen Stelle des Graphen mit der Maus zieht. Auf der linken Seite (1-12) befinden sich hauptsächlich Einstellungen, die die Aufgaben betreffen. Auf der rechten Seite (13-23) befinden sich hauptsächlich Funktionen, die die Grafik betreffen.

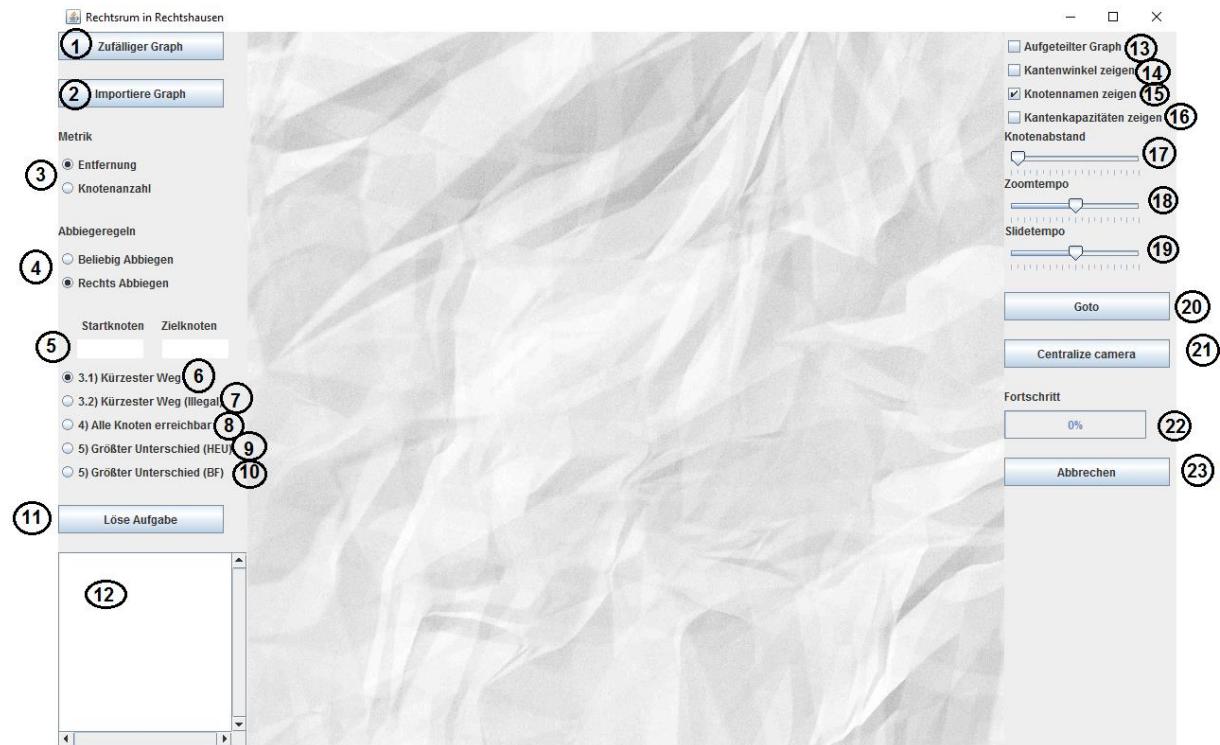


Abbildung 63

1. Funktion zum Erzeugen eines zufälligen Graphen (Siehe Kapitel 8)
2. Laden eines Straßennetzes aus einer Textdatei (Der Dateiselektor startet im Unterordner „graphen“, der sich im gleichen Verzeichnis befinden muss, wie die .jar Datei)
3. Einstellung des Weglängenmaßes
4. Einstellung, ob die Lösungen ausgehend davon bestimmt werden, ob man links abbiegen darf oder nicht
5. Textfelder in dem die Knotennamen zweier Knoten eingetragen werden können von denen der kürzeste Weg bestimmt werden soll (siehe Punkt 6)
6. Auswahl, dass der kürzeste Weg zwischen zwei Knoten bestimmt werden soll (Aufgabenteil 3)
7. Auswahl, dass der kürzeste Weg zwischen zwei Knoten bestimmt werden soll ausgehend davon, dass man x-mal links abbiegen darf, bestimmt werden soll (inhalt. Erweiterung)
8. Auswahl, dass bestimmt werden soll, ob sich alle Knoten gegenseitig erreichen können (Aufgabenteil 4)
9. Auswahl, dass das Knotenpaar bestimmt werden soll für das das Linksabbiegeverbot die Weglänge um den möglichst großen Faktor erhöht. Es wird die heuristische Methode verwendet (Aufgabenteil 5) Verwendung bei größeren Graphen wo Brute-Force zu lange dauert
10. Auswahl, dass das Knotenpaar bestimmt werden soll für das das Linksabbiegeverbot die Weglänge um den möglichst großen Faktor erhöht. Es wird das genaue Ergebnis bestimmt (Aufgabenteil 5). Verwendung bei kleineren Graphen
11. Löst die in den vorherigen Punkten gestellte Aufgabe
12. Ausgabe, was bestimmt worden ist und falls möglich Informationen bezüglich des Status des Programmes, sowie dem Ergebnis der jeweiligen Aufgabe
13. Auswahl, ob der Rechtsabbiegegraph oder Beliebigabbiegegraph angezeigt werden soll
14. Auswahl, ob Knoten mit den Winkeln seiner angrenzenden Kanten beschriftet werden soll
15. Festlegung, ob Knotennamen angezeigt werden sollen
16. Festlegung, ob Kanten mit ihrer Kapazität beschriftet werden sollen
17. Ermöglicht es zu variieren wie weit Knoten auseinander liegen sollen
18. Bestimmung der Sensibilität des Zooms der Kamera
19. Festlegung der Sensibilität der Bewegung innerhalb des Graphen
20. Die Kamera wird auf einen ausgewählten Knoten zentriert
21. Die Kamera wird so eingestellt, dass der gesamte Graph zu sehen ist
22. Anzeige wie weit das Programm fortgeschritten ist, falls es gerade dabei ist eine Aufgabe zu lösen
23. Falls gerade eine Aufgabe ausgeführt wird, kann diese hier abgebrochen werden