Lukas Kesch
lukas.kesch@gmail.com
Teilnahme-ID: 53757

Aufgabe 3: Abbiegen?

38. Bundeswettbewerb für Informatik - Runde 2 (20.04.2020)

1. Aufgabenstellung

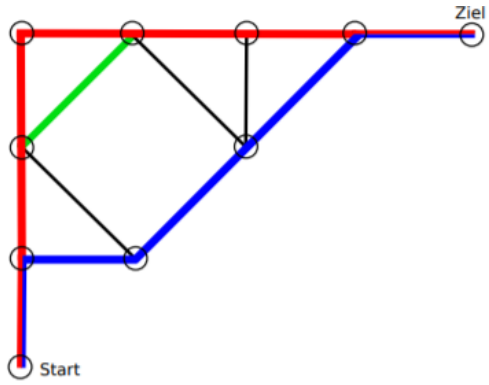
Informationen in diesem Abschnitt werden aus dem offiziellen Aufgabenblatt entnommen (vgl. Aufgabenblatt o. J.) oder aus den Beispielen gefolgert (vgl. Material zu den Aufgaben der 2. Runde o. J.).

1.1 Situation

Fahrradfahrer, die von A nach B gelangen wollen, möchten möglichst wenig abbiegen müssen. Sie nehmen dabei gerne, je nach Wetterlage, einen unterschiedlich hohen (prozentualen) Weglängenaufschlag gegenüber dem kürzesten Weg in Kauf, sofern dieser die Anzahl der Abbiegevorgänge reduziert.

1.2 Beispielkarte

Im folgenden ist eine Beispielkarte zu sehen. In dieser wurden nachträglich Wege gezeichnet.



In der Karte sind ein Start- sowie Zielpunkt markiert. Eine Kreuzung wird durch einen Kreis symbolisiert. Eine Straße verbindet genau zwei Kreuzungen. Wird eine Kreuzung mit einem anderen Winkel verlassen, als diese betreten wird, so findet eine Abbiegung statt. Somit muss in dem blauen Weg dreimal und im roten Weg lediglich einmal abgebogen werden. Der kürzeste Weg zwischen den Start- und Endpunkt ist der, der die geringste Distanz besitzt.

1.3 Getroffene Annahmen

Die folgenden vier Annahmen wurden nach Einsicht in die offizielle Aufgabenstellung und die angegebenen Beispiele getroffen.

1. Jede Straße ist geradlinig, da lediglich ihre Endpunkte angegeben werden.
2. Kreuzungen, an denen abgebogen werden kann, existieren nur an den Endpunkten einer jeden Straße.
3. Einbahnstraßen gibt es nicht. Eine Straße wird für den Radfahrer als in beide Richtungen befahrbar angesehen.
4. Zwischen zwei Knoten gibt es lediglich eine oder keine Straße.

1.4 Lösung

Das Programm soll eine fiktive Stadtkarte einlesen, in der ein Start- sowie ein Endpunkt markiert sind. Der Fahrradfahrer soll nun den prozentualen Weglängenaufschlag angeben, den er bereit ist, gegenüber dem kürzesten Weg in Kauf zu nehmen, um die Anzahl der Abbiegevorgänge zu vermindern. Daraufhin soll das Programm ihm einen passenden Weg vorschlagen.

2. Lösungsidee

2.1 Karte als Graph interpretieren

Die Stadtkarte beinhaltet die Start- und Endkoordinaten einer jeden Straße. Alle Straßen sind geradlinig und können somit als Strecke in einem ebenen kartesischen Koordinatensystem angesehen werden. Die Wegstrecken verlaufen derart, dass sich die Straßen nicht im Inneren der Strecke kreuzen, sondern nur am Start- und Endpunkt in eine Kreuzung münden. Aufgrund dessen sind alle Kreuzungen aus den gegebenen Strecken eindeutig bestimmt.

Die Karte wird nun als Graph gedeutet. Jede Kreuzung repräsentiert einen Knoten in dem Graphen und jede Straße zwischen zwei unterschiedlichen benachbarten Kreuzungen eine gewichtete Kante. Das Kantengewicht wird durch die Länge der Straße bestimmt. Diese entspricht dem euklidischen Betrag des Verbindungsvektors: $d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$. Damit sind die Kantengewichte ausschließlich positiv.

2.2 Ermitteln des kürzesten Weges

Zur Ermittlung der maximal zulässigen Weglänge, die der Benutzer bereit ist, zu akzeptieren, muss zuerst die Länge des kürzesten Weges gefunden werden. Denn der Benutzer gibt lediglich den prozentualen Weglängenaufschlag auf die kürzeste Route an. Hierfür wird der Dijkstra Algorithmus verwendet (vgl. Cormen et al. 2009: S. 658 und Dijkstra's algorithm 2020). Dieser ist der Standardalgorithmus für *single-source shortest path* Probleme. Er liefert immer die optimale Lösung, sofern der Graph keine Kanten mit negativen Gewichten enthält, und hat bei entsprechender Implementierung der Datenstrukturen das bestmögliche Laufzeitverhalten. Da lediglich der kürzeste Weg von dem Start- zu dem Endknoten von Relevanz ist, wird ein Spezialfall (single-source single-destination) des Dijkstra Algorithmus angewandt.

Der Dijkstra Algorithmus geht so vor, dass er von einem Startknoten aus alle Nachbarknoten ermittelt und deren Distanz zum Startknoten speichert, in Folgeschritten auch deren Vorgänger. Dieser ist im ersten Schritt der Startknoten. Nun geht er zu dem Knoten, der dem Startknoten am nächsten ist. Von da aus sucht er alle Nachbarn dieses Punktes und setzt ihre Entfernung auf die zurückgelegte Strecke zu diesem Punkt, solange diese kleiner als die bereits gespeicherte Entfernung ist. Dabei wird auch der Vorgänger abgespeichert. Dieser ist in diesem Fall der Knoten, der dem Startknoten am nächsten ist. Jetzt geht er erneut zu einem Knoten, der dem Startknoten am nächsten ist. Dabei gilt, dass bereits aufgesuchte Knoten wegfallen und somit stets neue Knoten dem Startknoten am nächsten sind. Dieser Vorgang wird solange wiederholt, bis der dem Startpunkt nächste Knoten der Endknoten ist.

2.3 Ermittlung eines Wegvorschlags

2.3.1 Maximallänge

Die Maximallänge, die der vorgeschlagene Weg haben darf, wird aus der Länge des kürzesten Wegs und dem angegebenen Prozentsatz p errechnet. $L_{max} = L_{min} \times (1 + p / 100)$

2.3.2 Abbiegevorgang

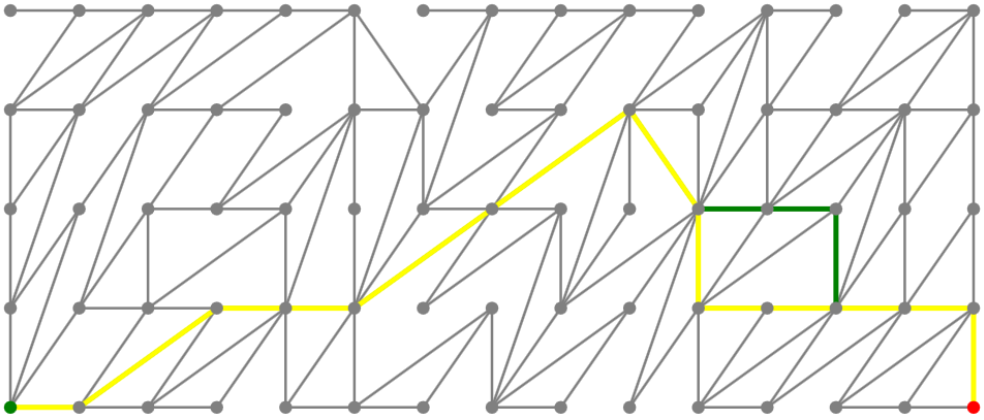
Das Programm muss in der Lage sein, einen Abbiegevorgang zu erkennen, denn es soll deren Anzahl reduzieren können. Hierfür wird jede Kante des Graphen erneut als Vektor betrachtet. Dieser kann dann in seine X-Komponente und in seine Y-Komponente zerlegt werden. Nun kann mit der inversen Funktion des Tangens ($arctan(x)$) der Winkel des Vektors zur X-Achse bestimmt werden. An einem Knoten wird dann der Winkel der Kante, die zu diesem führte, mit dem Winkel der Kante verglichen werden, mit der der Knoten wieder verlassen wird. Bei diesem Vergleich wird mit einem kleinen Epsilon gearbeitet, um Rundungsungenauigkeiten zu umgehen. Zwei Winkel werden als identisch angesehen, wenn gilt $| Winkel1 - Winkel2 | < Epsilon$. Sollten die Winkel nicht übereinstimmen, so findet bei dem Übergehen der einen in die andere Kante ein Abbiegevorgang statt.

Theoretisch funktioniert die Winkelberechnung in einem Fall nicht. Wenn ein Vektor mit ± 90 Grad zur X-Achse ausgerichtet ist, gibt es eine Division durch Null, da $\Delta x = 0$ und $\alpha = arctan(\Delta y / \Delta x)$. Dieser Fall wird aber in der Implementierung abgefangen.

2.3.3 Referenzweg

Für die Berechnung eines Wegvorschlages wird zusätzlich zu der minimal möglichen Weglänge auch die geringst mögliche Anzahl an Abbiegungen in einem kürzesten Weg benötigt.

Der Dijkstra-Algorithmus liefert im ersten Teil zwar den kürzesten Weg zwischen Start und Ziel, aber nicht unbedingt den kürzesten, der zusätzlich auch die geringste Anzahl an Abbiegevorgängen besitzt, da es mehrere gleich lange Wege geben kann mit unterschiedlicher Anzahl an Abbiegevorgängen. Dies ist an folgendem Beispiel zu erkennen:



Der Weg, den der Dijkstra-Algorithmus ermittelt (grün und gelb), ist genauso kurz wie der andere Weg (gelb). Dieser weist aber eine Abbiegung weniger auf.

Deshalb wird nach dem Dijkstra-Algorithmus der Wegfinde-Algorithmus mit $10^{-12}\%$ Aufschlag auf die Länge eines kürzesten Wegs gestartet, um den kürzesten Weg mit der geringsten Anzahl an Abbiegungen zu ermitteln. Der kleine Aufschlag hift wieder, Rundungsproblematiken zu umgehen. Dieser Weg wird dann als Referenzweg für den eigentlichen zweiten Teil verwendet.

2.3.3 Eigentliche Berechnung

Bei der Berechnung des optimalen Weges werden alle theoretisch möglichen Wege in Betracht gezogen. Dies garantiert eine optimale Lösung für die Aufgabe.

Umgesetzt wird dies mit einem rekursiv operierenden Algorithmus. Dieser beginnt an dem Startknoten und durchläuft alle möglichen Wege. Er geht von jedem Knoten aus zu jedem Benachbarten und von da aus erneut zu jedem weiteren benachbarten Knoten und so weiter.

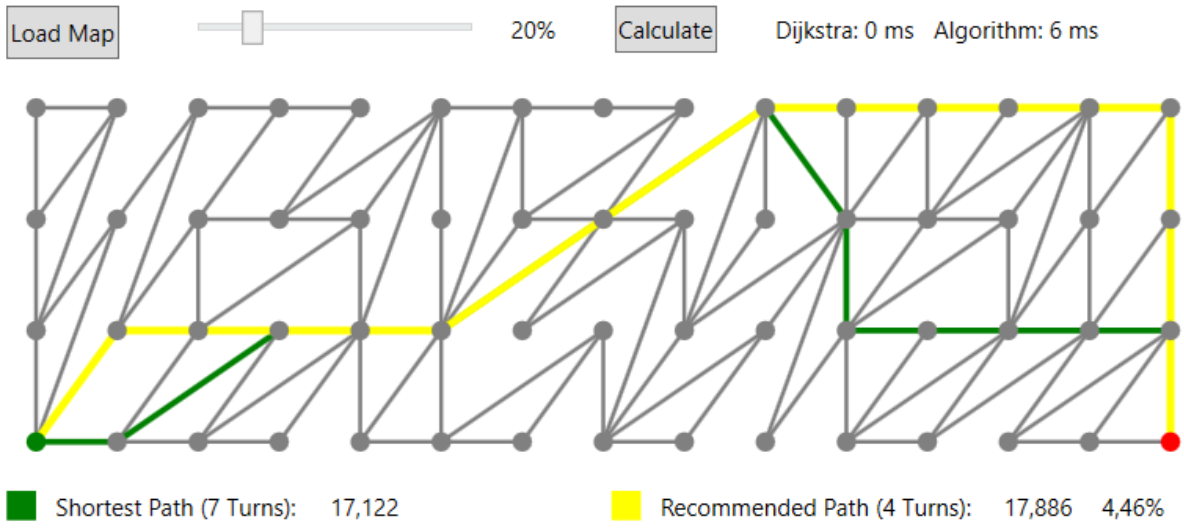
Damit der Algorithmus sich nicht in einer Endlosschleife befindet, wird das erneute Aufsuchen eines bereits besuchten Knotens untersagt. Dies ist legitim, da ohne diese Einschränkung die Anzahl der Abbiegevorgänge und die Weglänge bei einem unnötigen Rundweg größer sind, als bei einem direkteren Weg. Zusätzlich werden alle Routen, die länger als vom Benutzer gewünscht sind oder zu einer höheren Anzahl an Abbiegungen als der Referenzweg führen, frühzeitig verworfen.

Der Algorithmus verschärft die letzten zwei Kriterien zusätzlich während der Laufzeit. Wird ein Weg, dessen Länge die maximale Weglänge nicht überschreitet, gefunden, der zugleich eine geringere Anzahl an Abbiegevorgängen aufweist, so wird diese Anzahl als neue Obergrenze festgelegt. Dieser Weg wird als vorerst bester Weg zwischengespeichert, wird aber überschrieben, wenn ein noch besserer Weg gefunden wird.

Sollten sich ein aktueller Weg mit dem im Zwischenspeicher in der Anzahl an Abbiegevorgängen gleichen, so wird deren Weglänge verglichen. Der kürzere der beiden Wege wird dann als besser betrachtet und zwischengespeichert.

3. Umsetzung

3.1 Benutzeroberfläche



Die Benutzeroberfläche ist so gestaltet, dass bei dem Start des Programms nur der Button *Load Map* sichtbar ist. Wird dieser gedrückt, so öffnet sich ein Windows File Explorer. Dieser startet automatisch in dem Ordner der Straßenkarten. Wenn nun eine Datei ausgewählt wird, so wird diese in das Programm geladen. Der Benutzer erhält dann die Karte mit bereits eingezeichnetem kürzestem Weg (grün) mit den Informationen zu diesem. Jetzt sind auch der Slider, über den der prozentuale Wegaufschlag vorgegeben werden kann, und der Button *Calculate* ersichtlich. Wird dieser betätigt, wird dem Benutzer der ermittelte Weg mit weiteren Informationen ausgegeben. Die Informationen der Wege beinhalten die Länge der jeweilige Wege und die Anzahl an Abbiegevorgängen. Für den vorgeschlagenen Weg wird zusätzlich angegeben, um wie viel Prozent er von dem kürzesten abweicht. Dem Benutzer wird zudem noch die benötigte Rechenzeit für den Dijkstra- und Wegfindealgorithmus angezeigt.

3.2 Allgemeiner Programmablauf

Das Programm beginnt mit dem Einlesen der Beispieldatei. Im folgenden werden die Datenstrukturen gefüllt. Mehr hierzu im Abschnitt *Einlesen der Karte*.

Jetzt wird der Dijkstra-Algorithmus verwendet, um die Länge der kürzesten Strecke vom Start- zum Endknoten zu bestimmen. Genauere Details sind in dem Abschnitt *Dijkstra: Ermittlung des kürzesten Weges* zu finden.

Danach wird die Anzahl der Abbiegungen in dem Weg, den der Dijkstra Algorithmus ermittelt hat, berechnet, um dann den Wegfindealgorithmus (mit $10^{-12}\%$) starten zu können. Eine Erläuterung dieses Algorithmus befindet sich in dem Abschnitt *Ermittlung eines Wegvorschlags*. Dieser wird eingesetzt, da es möglich ist, dass es mehrere kürzeste Wege

gibt, diese sich aber in der Anzahl der Abbiegungen unterscheiden. Nun wird dem Benutzer der kürzeste Weg (grün) mit der geringsten Anzahl an Abbiegungen ausgegeben.

Gibt der Benutzer sich mit diesem Weg nicht zufrieden und startet die Berechnung mit einem beliebigen Prozentsatz (zwischen 0 und 100), so wird der Wegfindealgorithmus mit diesem Wert aufgerufen und schlägt dem Benutzer einen neuen Weg (gelb) vor.

3.3 Datenstrukturen

3.3.1 Verwenden von Structures statt Klassen

Für die Buchhaltung werden Strukturen und keine Klassen verwendet. Diese sind laufzeittechnisch schneller, da sie einen geringeren Overhead und im Gegensatz zu Klassen, die per Definition vom Reference-Type sind, einen Value-Type besitzen. Die Strukturen werden zum Zusammenfassen mehrerer Variablen zu einem neuen Typ genutzt.

3.3.2 Vertex

Diese Struktur stellt einen Knoten in dem Graphen da. Sie speichert folgende Variablen:

- Die X- und Y-Koordinate des Knotens jeweils als Double (Gleitkommazahl)
- Eine Zustandsvariable als Boolean. Diese gibt an, ob der Knoten bereits besucht wurde.
- Eine Indexnummer als Integer (Ganze Zahl). Diese gibt an, als wievieltes ein Vertex-Objekt erstellt werde.
- Eine Liste mit allen Indexnummern der Knoten, die mit dem aktuellen direkt verbunden sind.

3.3.3 ArrayVertices / ListVertices

Diese Datenstruktur speichert alle Vertex-Objekte (Kanten) während der Erstellung derselben in einer Liste variabler Länge. Sind alle Vertex-Objekte erstellt, so wird die Liste in ein Array fester Länge umgewandelt.

3.3.4 VertexInfo

Diese Struktur speichert zusätzliche Informationen zu jedem Knoten, die für den Dijkstra-Algorithmus notwendig sind. Es wird die Distanz zum Startpunkt (Datentyp: double), die Elementnummer des vorherig besuchten Knotens (Datentyp: int) und eine Zustandsvariable (Datentyp: bool), die angibt, ob der Punkt bereits besucht wurde, gespeichert.

3.3.5 RecursionVertexInfo

Diese Struktur fasst die zu übergebenden Variablen der Recursion-Methode (Wegfinde-Algorithmus) in ein Objekt zusammen. Dieses speichert die Anzahl an Abbiegungen (Datentyp: int), die bereits zurückgelegte Distanz (Datentyp: double), den

Winkel (Datentyp: double), mit dem der vorige Knoten relativ zur X-Achse betreten wurde, und eine Liste mit der Reihenfolge der besuchten Knotennummern (Datentyp: List<int>)

3.3.6 Edge

Diese Struktur repräsentiert eine Kante in dem Graphen. Ein Edge-Objekt speichert lediglich die Anfangs- und Endkoordinate der Linie als Gleitkommazahl (Datentyp: double) ab. Der Konstruktor nimmt zwei Vertex-Objekte entgegen.

3.3.7 ArrayEdges / ListEdges

Diese Datenstruktur speichert alle Edge-Objekte während der Erstellung in einer Liste variabler Länge. Sind alle Edge-Objekte erstellt, so wird die Liste in ein Array fester Länge umgewandelt.

3.4 Einlesen der Karte

Die Karte beziehungsweise Datei wird als String eingelesen. Dieser wird, nach Zeilenumbrüchen getrennt, in einem Array abgespeichert.

Zuerst wird dieses Array durchgegangen, um alle Kanten (Straßen) zu ermitteln und jede als Edge-Objekt in der Datenstruktur *ListEdges* abzuspeichern. Gleichzeitig wird jeder noch nicht vorhandene Knoten (Kreuzung) als Vertex-Objekt der *ListVertices* hinzugefügt.

Anschließend wird durch das *ArrayEdges* iteriert, um für jeden Knoten seine Nachbarn zu bestimmen. Diese werden mit ihrem Index in der Liste der Nachbarn des Knotens gespeichert.

3.5 Sonderfall Winkelberechnung

Wie bereits in dem Kapitel *Lösungsidee* Abschnitt *Abbiegevorgang* erwähnt gibt es einen Sonderfall. Dieser tritt ein, wenn ein Vektor mit $\pm 90°$ gegen die X-Achse ausgerichtet ist.

Wenn eine Gleitkommazahl in C# durch eine weitere Gleitkommazahl, die den Wert 0 besitzt, dividiert wird, ist das resultierende Ergebnis *PositiveInfinity* (Zähler > 0) oder *NegativeInfinity* (Zähler < 0) (vgl. Double.PositiveInfinity Field (System) o. J. und Double.NegativeInfinity Field (System) o. J.).

Die Arkustangensfunktion in dem .Net Framework interpretiert diese zwei Fälle richtig und gibt 1.5707963267949 $(\approx \frac{1}{2}\pi)$ für *PositiveInfinity* und -1.5707963267949 $(\approx \frac{-1}{2}\pi)$ für *NegativeInfinity* zurück. Allgemein besitzt sie folgenden Wertebereich: $[\frac{-1}{2}\pi; \frac{1}{2}\pi]$ (vgl. Math.Atan(Double) Method (System) o. J.).

Dieser ist im Sinne der Aufgabe jedoch zu groß, denn $\pm 90°$ sind gleichbedeutend. Deshalb wird der Wertebereich auf $(\frac{-1}{2}\pi; \frac{1}{2}\pi]$ eingeschränkt, damit jeder Winkel eindeutig ist. Wird

dieser nicht beschränkt, so kann es zu “falschen” Abbiegevorgängen auf einer zur X-Achse orthogonalen Teilweg führen, die über mehrere Kreuzungen verläuft.

3.6 Dijkstra: Ermittlung des kürzesten Weges

Der Dijkstra-Algorithmus (vgl. Cormen et al. 2009: S. 658 und Dijkstra’s algorithm 2020) wird so, wie im Kapitel *Lösungsidee,* Abschnitt *Ermitteln des kürzesten Weges* beschrieben, umgesetzt.

Zu Beginn wird die Distanz jedes Knotens zu dem Startknoten auf double.MaxValue gesetzt, außer die des Startknotens selbst. Daraufhin beginnt die Ausführung einer do-while-Schleife. Diese setzt den eigentlichen Algorithmus um. Als erstes werden alle Nachbarn des aktuellen Punktes ermittelt und eventuell deren Distanz und Vorgänger verbessert. Dann wird der dem Startknoten am nächsten befindliche Knoten ermittelt und aufgesucht. Wenn der Knoten dem Endknoten gleicht, so wird der Algorithmus beendet und der kürzeste Weg wird dann in einem Array gespeichert. Zusätzlich wird die Länge des Weges gespeichert.

3.7 Ermittlung eines Wegvorschlags

Der Wegvorschlags-Algorithmus wird, wie in dem Kapitel *Lösungsidee* in dem Abschnitt *Ermittlung eines Wegvorschlags* beschrieben, implementiert.

Es handelt sich um ein Backtracking-Verfahren mit doppelter Abbruchbedingung (Weglängenbegrenzung und Begrenzung der Anzahl der Abbiegevorgänge). Das Backtracking ist vollständig, so dass sichergestellt wird, dass die unter den gemachten Vorgaben optimale Lösung gefunden wird.

Es wird damit begonnen, die maximal mögliche Weglänge zu ermitteln. Hierfür wird die Länge des kürzesten Weges und der angegebene Prozentsatz benötigt.

Nun wird die Anzahl der Abbiegungen des kürzesten Weges berechnet und diese als Obergrenze abgespeichert. Hierfür wird das Array, das den kürzesten Weg speichert, durchlaufen und an jedem Knoten wird überprüft, ob dieser mit einem anderen Winkel verlassen wird, als dieser betreten wurde.

Daraufhin beginnt der eigentliche Algorithmus. Dieser wird für jeden Nachbarn des Startknotens gestartet. Es muss immer der zu besuchende Knoten, der aktuelle Knoten und ein *RecursionVertexInfo*-Objekt übergeben werden.

Am Anfang jeder Iteration holt dieser sich aus dem *RecursionVertexInfo*-Objekt die aktuellen Daten. Dazu gehört die Anzahl an bereits stattgefundenen Abbiegungen, die zurückgelegte Distanz, der Winkel, mit dem der frühere Knoten betreten wurde, und eine Liste aller Vorgängerknoten. Nun wird die Distanz zwischen dem aktuellen und dem zu besuchenden Knoten berechnet und zu der bereits zurückgelegten Distanz addiert. Wenn diese nun größer als die maximale Weglänge sein sollte, so bricht die Methode ab und gelangt somit eine

Rekursionsebene nach oben. Auch wenn der berechnete Winkel zwischen dem aktuellen und dem zu besuchenden Knoten nicht mit dem Winkel übereinstimmt, mit dem der aktuelle Knoten betreten wurde, so wird die Methode verlassen, falls damit die Anzahl an Abbiegungen zu hoch ist. Wird die Methode nicht verlassen, so werden die erhobenen Daten in dem RecursionVertexInfo-Objekt gespeichert und der aufgesuchte Punkt der Liste mit den besuchten Punkten hinzugefügt. Ist der Punkt der Endpunkt und der Weg zu diesem besser als ein gespeicherter Weg, so wird derselbe gespeichert. Nun werden alle unbesuchten Nachbarn des Knotens ermittelt und für jeden ruft die Methode sich selbst auf. Am Ende wird der aktuelle Punkt aus der Liste der besuchten Punkte gelöscht. Dies ist nötig, da die Liste ein Reference-Type ist und somit für jede Rekursionsebene nur die Zeiger und nicht die Inhalte kopiert werden. Für die restlichen Datentypen ist dies nicht notwendig, da diese alle Value-Types sind und somit automatisch für jede Rekursionsebene eine Kopie erstellt wird.

Sind alle Wegkombinationen geprüft, so wird der zwischengespeicherte Weg in ein Weg-Array umgewandelt, welches alle Knoten als Vertex-Objekt speichert. Schließlich werden die Linien zwischen den Knoten in der Karte eingezeichnet und so dem Nutzer der Wegvorschlag ausgegeben.

4. Laufzeitanalyse

Die Anzahl an Kanten (Edges) wird im folgenden mit dem Buchstaben E dargestellt. Die Anzahl an Knoten (Vertices) repräsentiert der Buchstabe V.

Die O-Notation grenzt eine Funktion asymptotisch nach oben ein (obere Schranke). Die Ω-Notation schränkt eine Funktion asymptotisch nach unten ein (untere Schranke). Die Θ-Notation grenzt eine Funktion asymptotisch von oben als auch von unten ein (scharfe Schranke) (vgl. Cormen et al. 2009: S. 47).

4.1 Füllen der Datenstrukturen

Für das Füllen von *ArrayEdges* und *ArrayVertices* werden für jede Kante alle bis dahin vorhandenen Knoten untersucht. Somit ergibt sich eine Komplexität von $\Theta(E \times V)$. Anschließend, um alle Nachbarn eines Knotens zu bestimmen, wird erneut das *ArrayVertices* für jede Kante durchlaufen. Dabei gleicht die Komplexität der des ersten Schrittes. Somit ergibt sich eine Komplexität von $\Theta(2 \times E \times V) = \Theta(E \times V)$.

Es gilt, dass ein verbundener Graph mindestens $V - 1$ Kanten braucht. Damit kann eine untere Schranke allein abhängig von der Anzahl der Knoten aufgestellt werden. Es ergibt sich: $\Omega((V - 1) \times V) = \Omega(V^2)$.

Es gilt außerdem, dass es maximal $V \times (V - 1)$ Kanten geben kann unter der, aufgrund der Aufgabenstellung erfüllten, Voraussetzung, dass je nur eine Kante zwischen zwei Knoten existiert. Es ergibt sich eine obere Schranke von $O(V \times (V - 1) \times V) = O(V^3)$.

Dieser Fall ist jedoch sehr unwahrscheinlich. Wird ein genügend großer Graph als Karte angesehen, so müsste es viele Brücken und Tunnels geben, um das Schneiden zweier Straßen zu verhindern. Dieses ist nach den einführenden Erläuterungen zur Aufgabe untersagt, weil sonst eine neue (eigentlich nicht vorhandene) Kreuzung entstehen würde. Somit ist davon auszugehen, dass dieser Fall zwar theoretisch möglich ist, praktisch aber nicht auftritt. Daraus folgt, dass die Anzahl der Kanten, die aus einem Knoten herausgehen, limitiert und somit unabhängig von der Anzahl der Knoten ist. Die Anzahl der Knoten ist somit proportional zu der Anzahl der Kanten. Die Beispielkarten bestätigen dies. In ihnen hat ein Knoten durchschnittlich maximal vier anliegende Kanten (die jeweils zwei Knoten verbinden), also kann mit $E = k \times V$ gerechnet werden, wobei k der Proportionalitätsfaktor ist. Daraus ergibt sich $O(k \times V \times V) = O(V^2)$. Es fällt auf, dass dies der unteren Schranke gleicht. Somit ergibt sich $\Theta(V^2)$.

4.2 Dijkstra

Der Dijkstra-Algorithmus besitzt bei einer bestmöglichen Implementierung eine Laufzeitklasse von $\Theta(V \times log(V))$. Dies setzt eine PriorityQueue voraus, in der die noch nicht besuchten Knoten gespeichert werden. Allerdings ist im Rahmen der Aufgabe eine einfache Liste völlig ausreichend, da die Graphen relativ klein sind. Mit dieser beträgt die Komplexität $\Theta(V^2)$. Da der Dijkstra-Algorithmus beim Erreichen des Endpunktes abgebrochen wird, ändert sich seine Komplexität zu $O(V^2)$.

Damit beträgt die Gesamtkomplexität normalerweise $\Theta(V^2) + O(V^2) = \Theta(V^2)$. Im theoretisch möglichen Schlimmstfall beträgt sie $O(V^3) + O(V^2) = O(V^3)$.

4.3 Ermittlung eines Wegvorschlags

4.3.1 Problematik

Der Wegfinde-Algorithmus operiert rekursiv. Für solche Algorithmen ist eine Komplexitätsbetrachtung oftmals sehr aufwändig. Zudem ist es unmöglich, die Einsparung der frühzeitigen Abbruchbedingungen (Distanz zu hoch oder Anzahl der Abbiegungen zu hoch) analytisch zu ermitteln. Diese können lediglich statistisch erhoben werden.

4.3.2 Schlimmstfall

Der schlimmste Fall stellt ein Graph da, in welchem jeder Knoten direkt mit allen anderen Knoten verbunden ist. Werden alle möglichen Wege abgelaufen, ohne Knoten mehrfach zu besuchen, so beträgt die Komplexität des Wegfinde-Algorithmus ohne Abbruchbedingungen $\Theta((V-2)!)$. Wie aber bereits in einem vorherigen Abschnitt beschrieben ist dieser Fall sehr unwahrscheinlich. Zusätzlich operiert der Algorithmus mit zwei Abbruchbedingung, die das Wachstum einschränken.

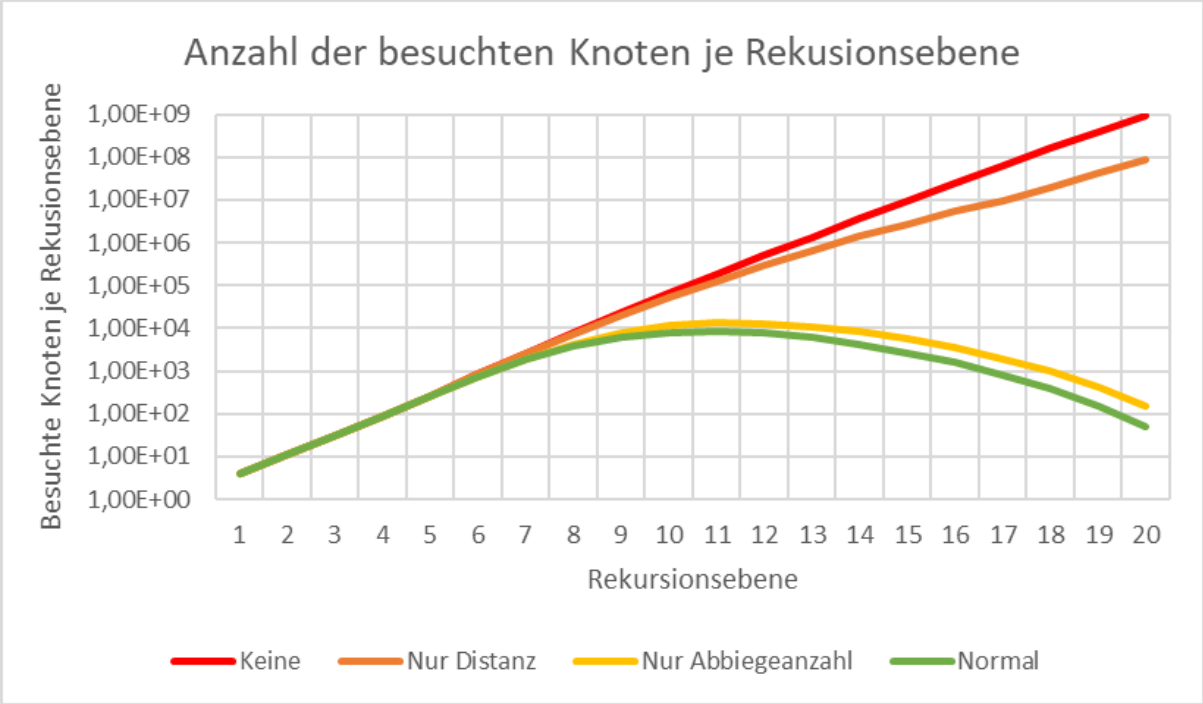
4.3.3 Normalfall

Im Normalfall ist die Anzahl der Kanten proportional zur Anzahl der Knoten. Daraus ergibt sich ein exponentieller Zuwachs der Rechenschritte in dem Wegfinde-Algorithmus für jeden weiteren Knoten. Als obere Schranke kann grob $O(k^V)$ angegeben werden. Die Konstante k gibt an, wie viele Kanten pro Knoten existieren.

Während des Durchgehens eines Graphens steigt die Anzahl der möglichen Wege jedoch oftmals langsamer als die obere Schranke. Dies liegt an den zwei Abbruchbedingungen, die das Wachstum “deckeln”. Zudem wird das Aufsuchen bereits besuchter Knoten untersagt.

4.3.4 Beurteilung der Abbruchbedingungen

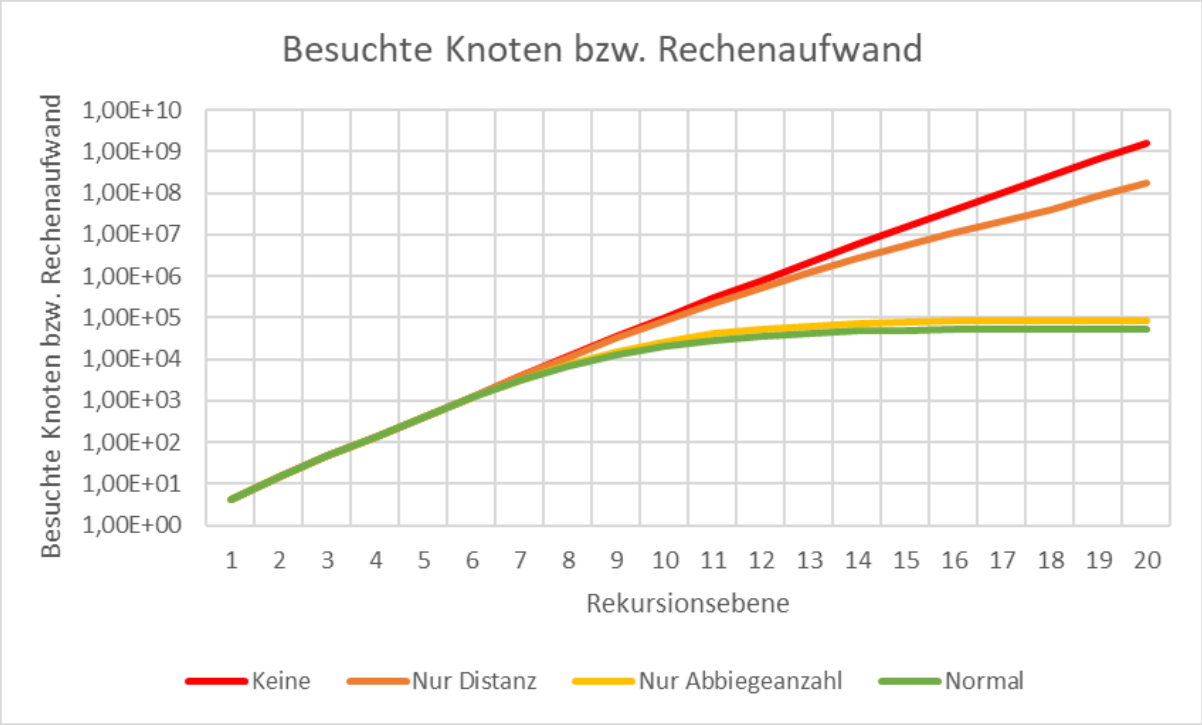
Für die folgende Beurteilung wurden die Beispiele 1 bis einschließlich 3 untersucht. In jeder Rekursionsebene des Wegfinde-Algorithmus wird gezählt, wie viele Knoten in dieser bearbeitet werden. Damit lässt sich die Effizienz der Abbruchbedingungen ermitteln, indem das Programm mit beiden, jeweils einer und keiner Abbruchbedingung gestartet wird. Für alle vier Fälle werden die drei Beispiele mit folgenden Prozentsätzen durchlaufen: 0; 0,2; 0,4; 0,6; 0,8; 1,0. Diese 15 Messwerte pro Fall werden dann zu jeweils einem Durchschnitt zusammengefasst. Im folgenden sind diese in einem Diagramm abgebildet. Aus laufzeittechnischen Gründen wurde dem Algorithmus im Rahmen der Datenerhebung untersagt, tiefer als in die 20. Rekursionsebenen zu gehen.



Bis zur siebten Rekursionsebene gleichen sich die Kurven. In den folgenden ist zu erkennen, dass die Kurve der Abbruchbedingung-Abbiegeanzahl nach Erreichen eines Höhepunktes in der ca. elften Rekursionsebene fällt. Die Kurve der Abbruchbedingung-Distanz steigt kontinuierlich. Daraus folgt, dass die Abbruchbedingung-Abbiegeanzahl sehr effektiv ist. Die Abbruchbedingung-Distanz ist hingegen wenig bis moderat effektiv, spart aber dennoch zusätzliche Rechenzeit.

Aus den Messwerten kann ein weiteres Diagramm erstellt werden. Dieses gibt die Gesamtzahl der besuchten Knoten bis hin zu einer jeden Rekursionsebene an. Mathematisch betrachtet entsprechen die neuen Graphen dem Integral der alten und geben somit den Flächeninhalt unter den Kurven an.

Das Bearbeiten eines Knotens im Programm ist ein linearer Prozess und ist unabhängig von der Rekursionsebene. Deshalb kann das folgende Diagramm auch als eines angesehen werden, welches den Rechenaufwand abbildet.

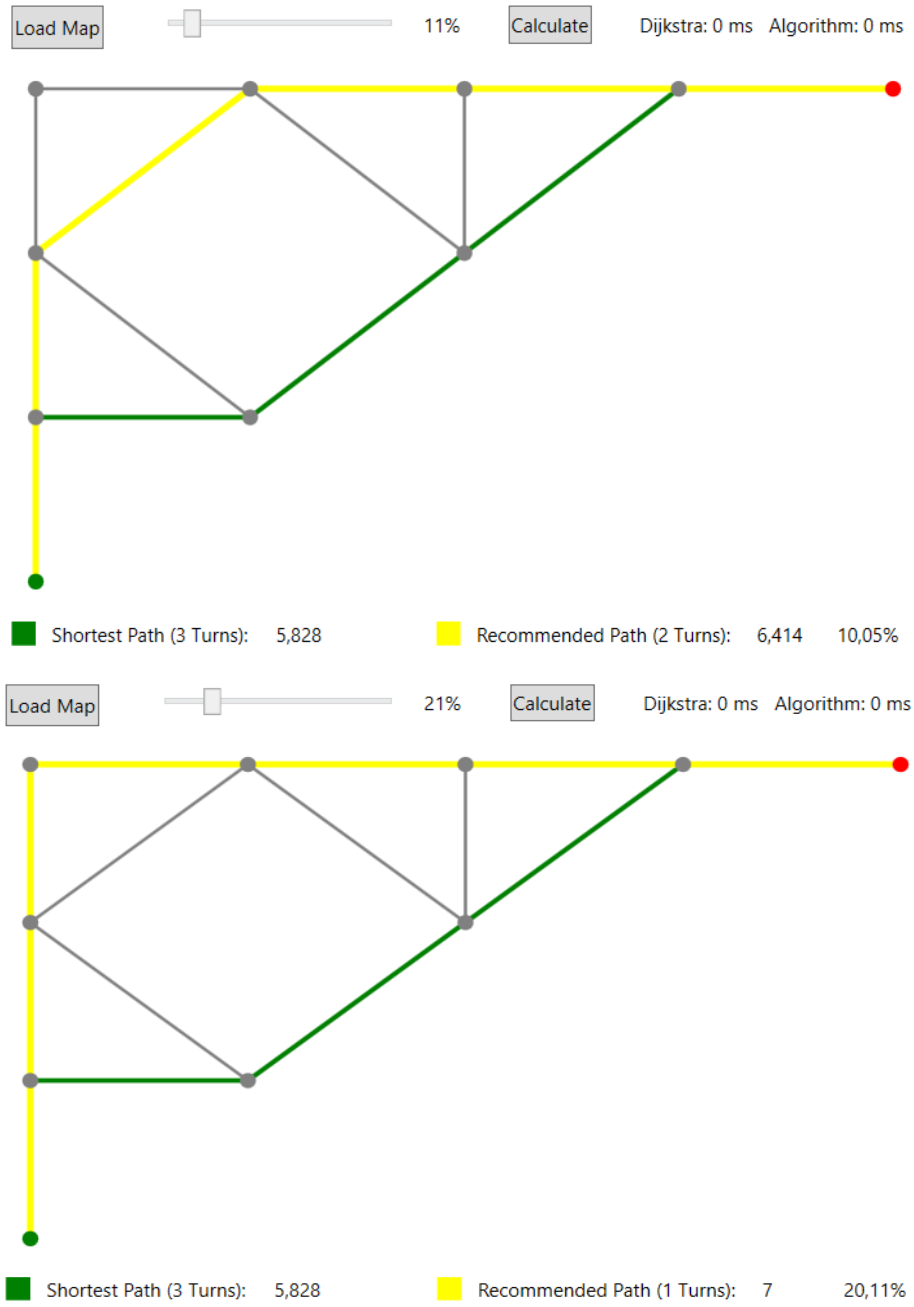


5. Beispiele

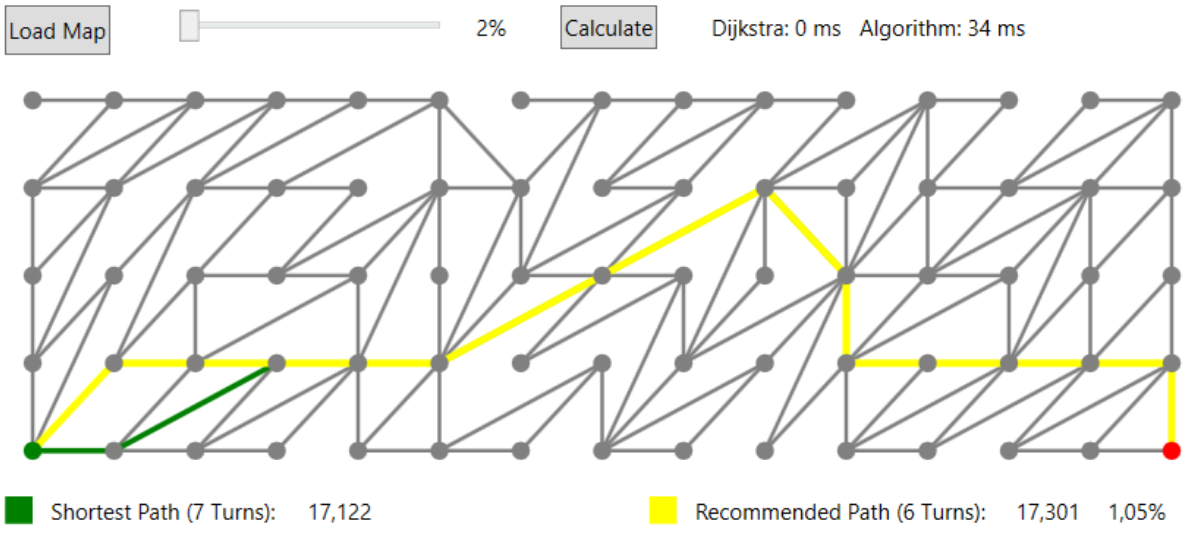
5.1 Gegebene Beispiele

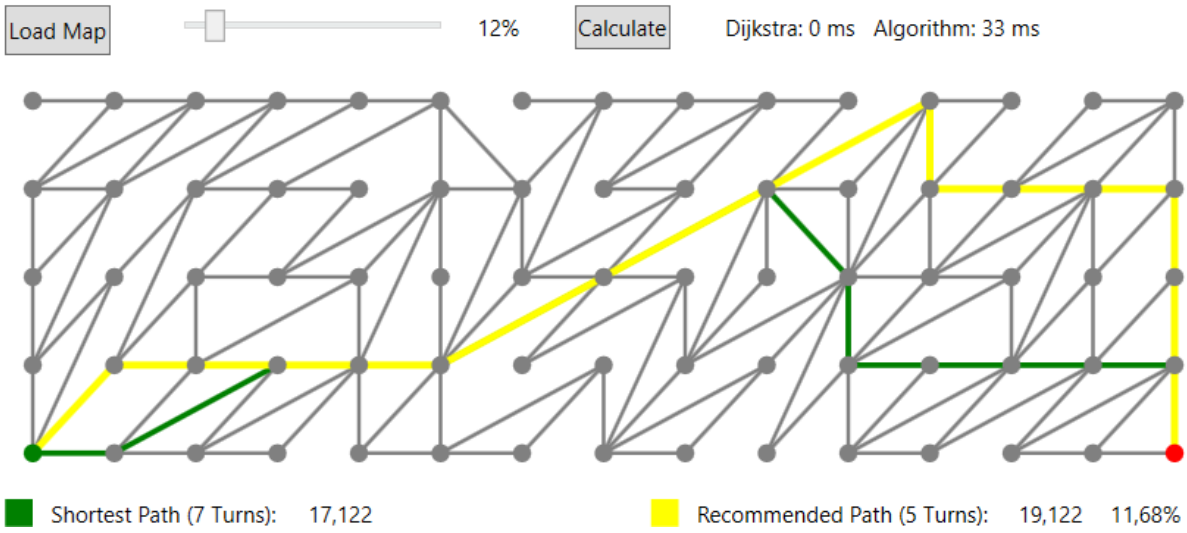
Für die folgenden Beispiele sind jeweils die kürzesten Wege für alle möglichen Abbiegezahlen, die geringer als die Anzahl an Abbiegungen im kürzesten Weg sind, angegeben.

5.1.1 Beispiel 0

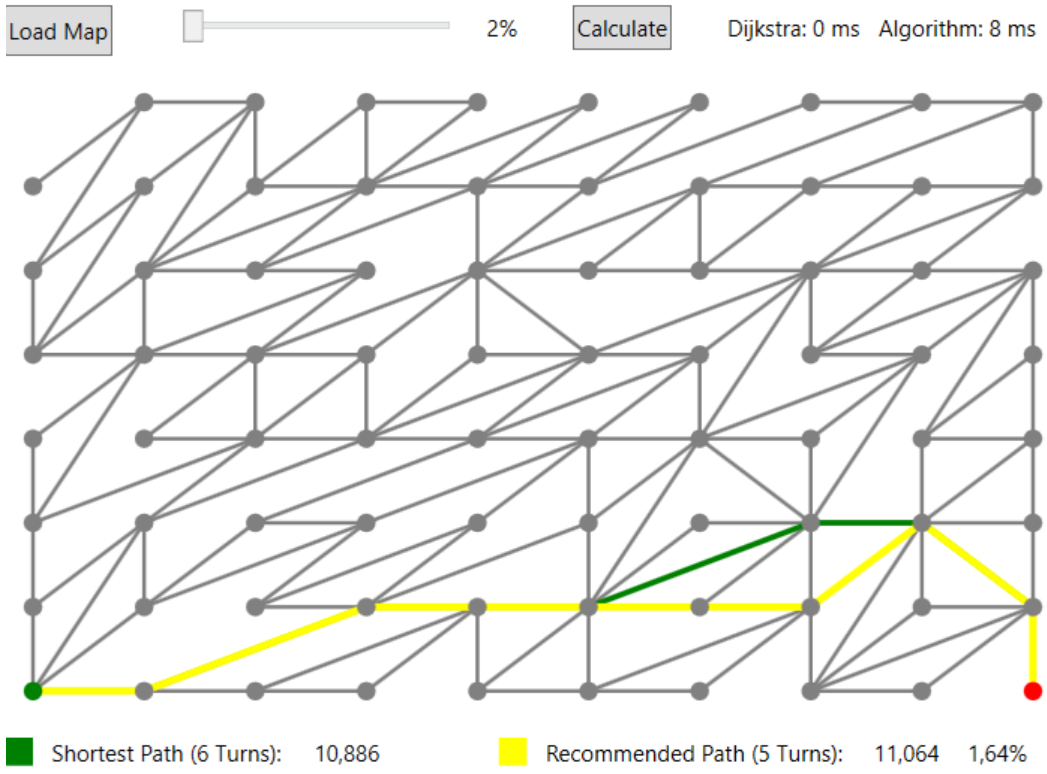


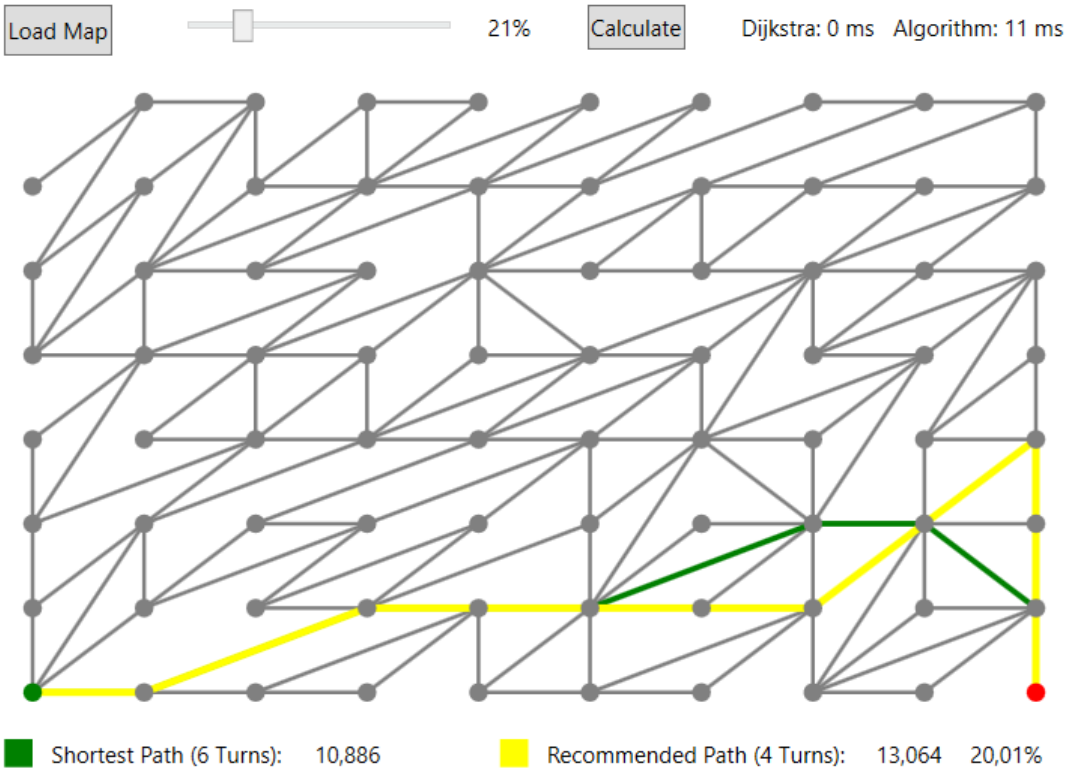
5.1.2 Beispiel 1

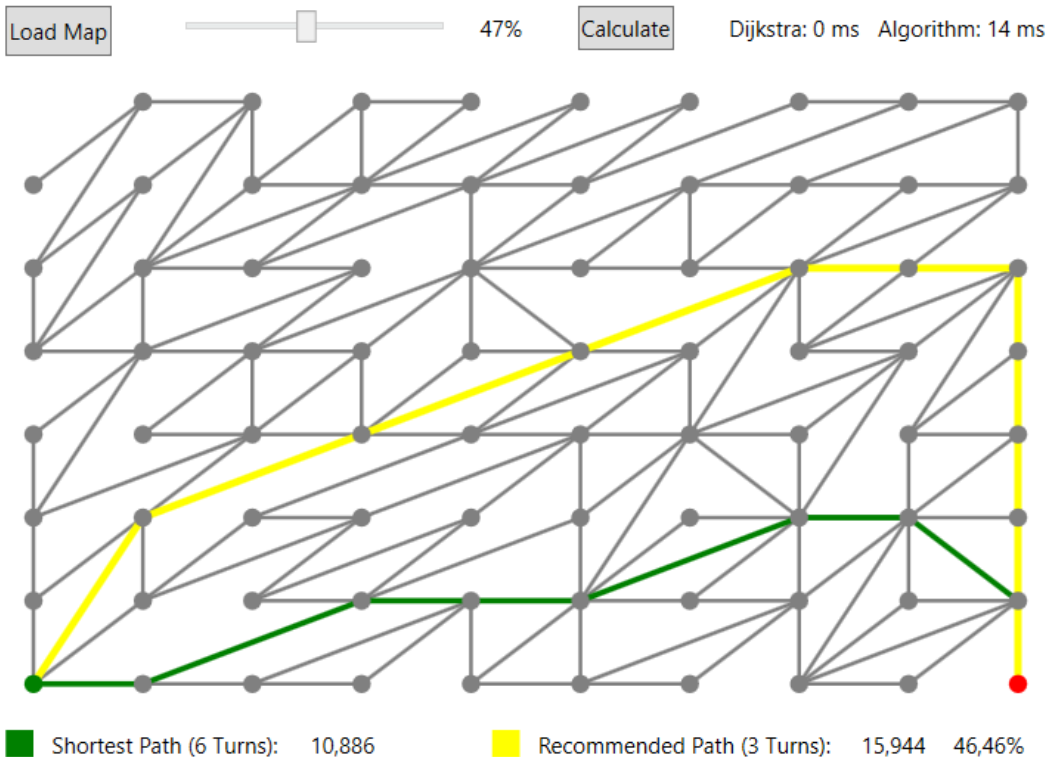




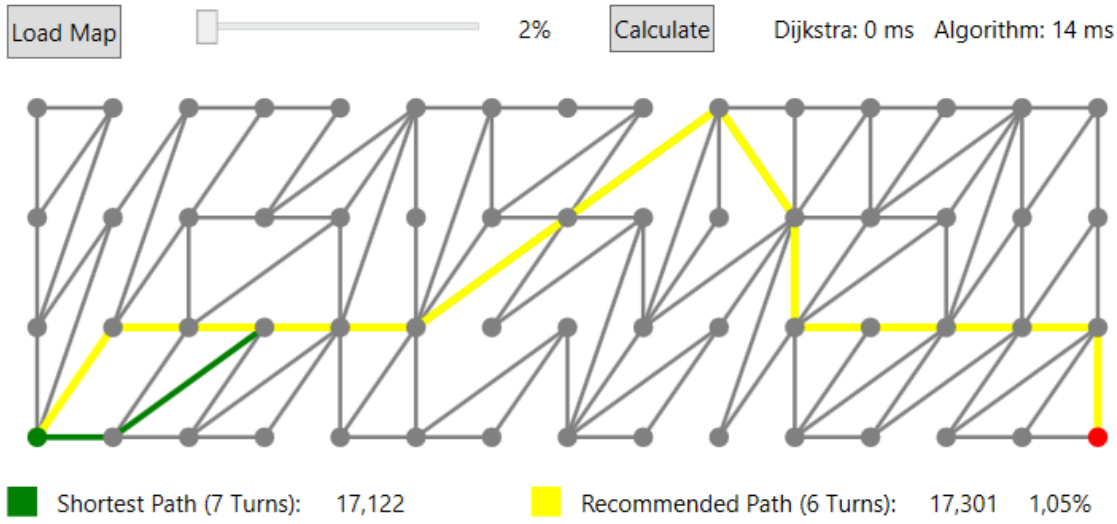
5.1.3 Beispiel 2

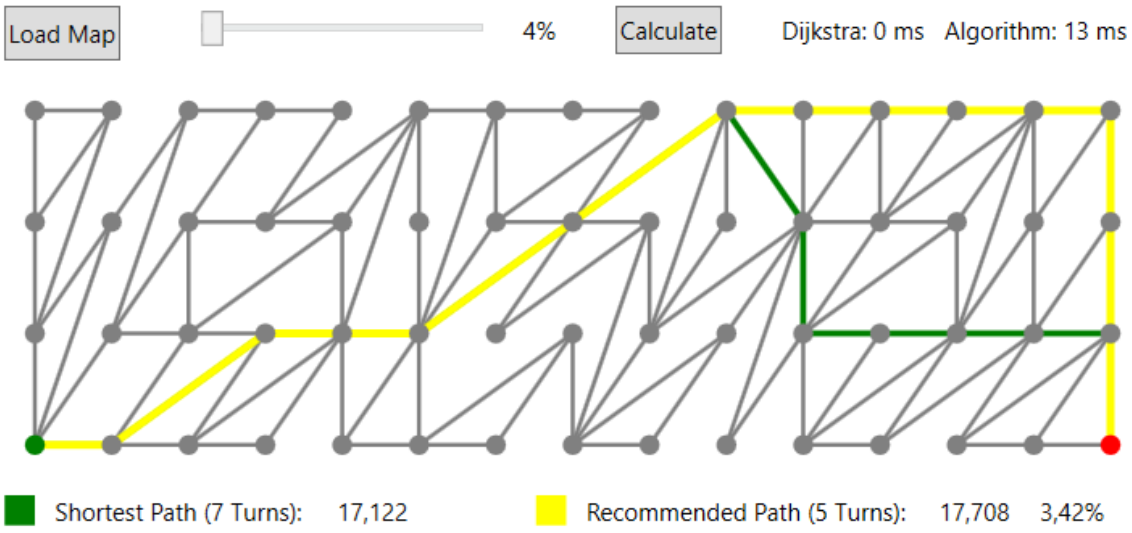


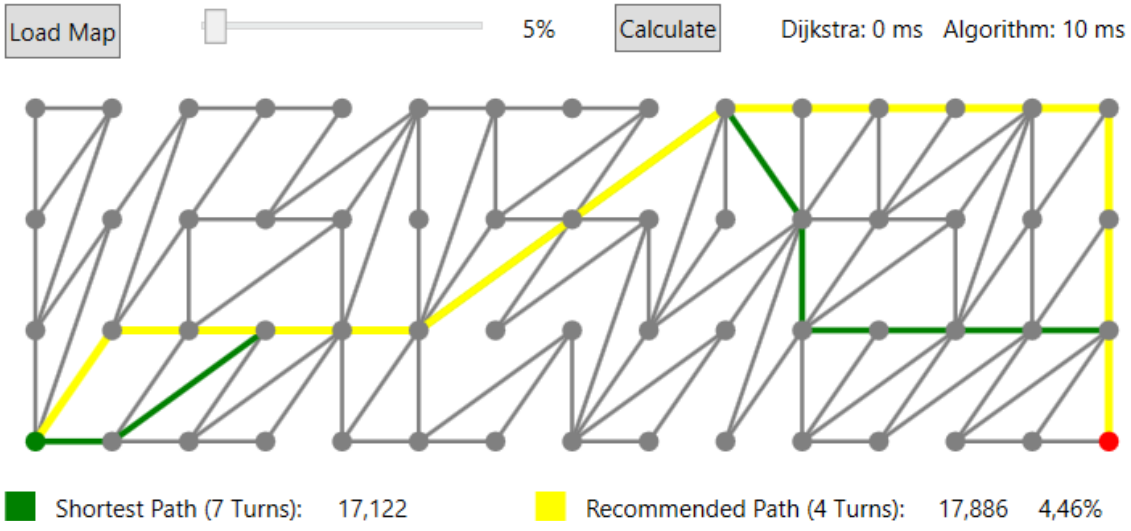



Load Map
47%
Calculate
Dijkstra: 0 ms
Algorithm: 14 ms
Shortest Path (6 Turns): 10,886
Recommended Path (3 Turns): 15,944 46,46%

5.1.4 Beispiel 3







5.1.5 Auswertung

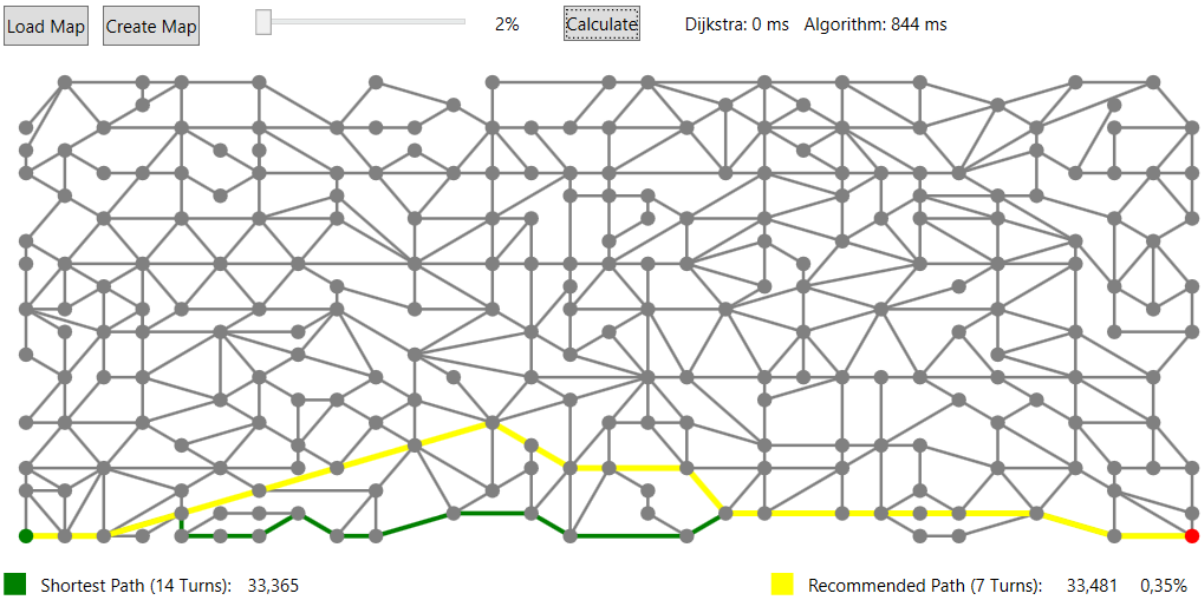
Es stellt sich heraus, dass die Beispiele laufzeittechnisch kein Problem darstellen.

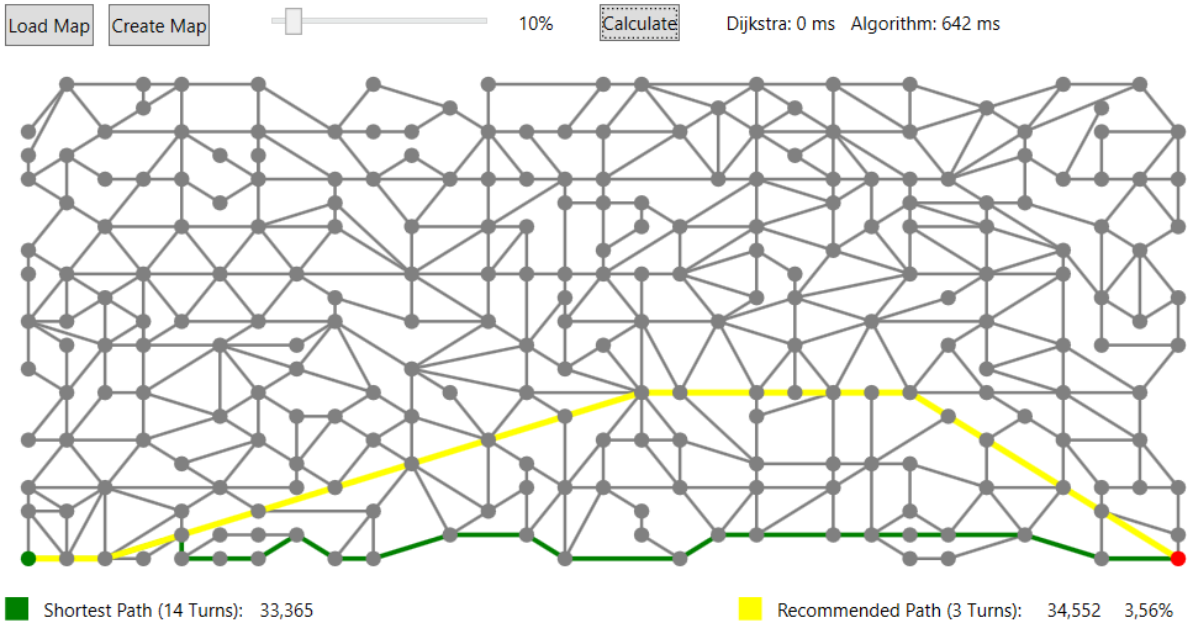
Der Dijkstra-Algorithmus benötigt in jedem dargestellten Fall weniger als eine Millisekunde.

Die durchschnittliche Laufzeit des Wegfinde-Algorithmus beträgt 14 Millisekunden mit einer Standardabweichung von 12 ms.

5.2 Eigene Beispiele

5.2.1 Beispiel 4





Es stellt sich heraus, dass selbst deutlich größere Karten von dem Programm gut bearbeitet werden können. Für beide Fälle liegt die Laufzeit unter einer Sekunde

6. Erweiterungen

6.1 Erstellen und Bearbeiten von Karten

Für den Fall, dass ein Radfahrer seine Straßenkarte in analoger Form besitzt, ist es ihm möglich eine digitale Straßenkarte zu erstellen.

Zuerst muss die Größe der Karte festgelegt werden. Daraufhin kann der Nutzer beliebig viele Straßen zeichnen (und auch wieder zu löschen), sowie einen Start- und Endpunkt setzen. Ist er mit dem Erstellen der Karte fertig, so kann die Karte im Format der Beispieldateien abgespeichert werden.

Diese Erweiterung wurde benutzt, um die in dem Kapitel *Beispiele* Abschnitt *Eigene Beispiele* ersichtliche Karte zu erstellen.

7. Literaturverzeichnis

Aufgabenblatt (o. J.): in: *BwInf*, [online] https://bwinf.de/fileadmin/bundeswettbewerb/38/aufgaben382.pdf [28.12.2019].

Cormen, Thomas / Leiserson / Rivest / Stein (2009): *Introduction to Algorithms*, 3. Aufl., Cambridge, Massachusetts: The MIT Press.

Dijkstra's algorithm (2020): in: *Wikipedia*, [online] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm [08.04.2020].

Double.NegativeInfinity Field (System) (o. J.): in: *Microsoft Docs*, [online] https://docs.microsoft.com/en-us/dotnet/api/system.double.negativeinfinity?view=netframework-4.8 [09.04.2020].

Double.PositiveInfinity Field (System) (o. J.): in: *Microsoft Docs*, [online] https://docs.microsoft.com/en-us/dotnet/api/system.double.positiveinfinity?view=netframework-4.8 [09.04.2020].

Material zu den Aufgaben der 2. Runde (o. J.): in: *BwInf*, [online] https://bwinf.de/bundeswettbewerb/38/2/material/ [28.12.2019].

Math.Atan(Double) Method (System) (o. J.): in: *Microsoft Docs*, [online] https://docs.microsoft.com/en-us/dotnet/api/system.math.atan?view=netframework-4.8 [09.04.2020].

8. Anhang

Die beiliegenden Dokumente befinden sich in folgender Reihenfolge:

1. Messreihe: Beide Abbruchbedingungen
2. Messreihe: Nur Abbiegeanzahl als Abbruchbedingung
3. Messreihe: Nur Distanz als Abbruchbedinung
4. Messreihe: Keine Abbruchbedinung
5. Quellcode: Datenstrukturen
6. Quellcode: Datenverarbeitung
7. Quellcode: Dijkstra
8. Quellcode: Rekursion

Besuchte Knoten je Rekursionsebene - Beide Abbruchbedingungen

Beispiel	*p*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Beispiel1.txt	0	4	11	34	94	265	716	1897	5130	13637	24445	33146	32079	23164	10677	3590	1036	98	0	0	0
Beispiel1.txt	20	4	11	34	94	265	716	1897	3844	6948	11149	13778	14588	13806	11312	7197	3708	1481	342	50	2
Beispiel1.txt	40	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11734	8749	5544	3040	1390	457	95
Beispiel1.txt	60	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1680	716	243
Beispiel1.txt	80	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1687	744	273
Beispiel1.txt	100	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1687	744	273
Beispiel2.txt	0	4	11	33	106	372	1351	4323	9238	8122	2488	417	0	0	0	0	0	0	0	0	0
Beispiel2.txt	20	4	11	33	106	372	1361	4781	12393	19569	16431	10106	2117	273	0	0	0	0	0	0	0
Beispiel2.txt	40	4	11	33	106	372	1361	3811	8317	12108	12428	8710	5290	1862	548	56	0	0	0	0	0
Beispiel2.txt	60	4	11	33	106	372	907	1352	1422	1750	1750	1416	772	313	98	50	22	0	0	0	0
Beispiel2.txt	80	4	11	33	106	372	907	1352	1422	1750	1460	1127	793	375	147	65	41	8	0	0	0
Beispiel2.txt	100	4	11	33	106	372	907	1352	1422	1750	1460	1112	744	381	170	99	46	8	0	0	0
Beispiel3.txt	0	4	11	28	65	166	429	1080	2676	6346	11097	14395	13674	9606	3974	746	112	0	0	0	0
Beispiel3.txt	20	4	11	28	65	166	429	794	1215	1824	2389	2614	2003	1312	796	490	245	82	1	0	0
Beispiel3.txt	40	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
Beispiel3.txt	60	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
Beispiel3.txt	80	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
Beispiel3.txt	100	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
	Avg	4	11	32	88	268	759	1856	3740	6031	7652	8315	7537	6114	4248	2696	1584	820	387	153	49

Besuchte Knoten je Rekursionsebene - Anzahl der Abbiegungen als Abbruchbedingung

Beispiel	*p*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Beispiel1.txt	0	4	11	34	94	265	716	1897	5130	13637	24477	33809	37189	34948	28668	19995	12246	6606	3217	1330	486
Beispiel1.txt	20	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1687	744	273
Beispiel1.txt	40	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1687	744	273
Beispiel1.txt	60	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1687	744	273
Beispiel1.txt	80	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1687	744	273
Beispiel1.txt	100	4	11	34	94	265	716	1897	3844	6948	11149	13778	14594	13921	11738	8771	5674	3229	1687	744	273
Beispiel2.txt	0	4	11	33	106	372	1361	4792	15869	37716	56807	62853	56454	43368	29081	17252	9639	5048	2569	1165	419
Beispiel2.txt	20	4	11	33	106	372	1361	4792	12852	23351	32214	34386	30910	24058	15970	9494	5669	3019	1438	719	267
Beispiel2.txt	40	4	11	33	106	372	1361	3811	8329	12282	13475	11745	8960	5555	3125	1624	841	321	107	17	4
Beispiel2.txt	60	4	11	33	106	372	907	1352	1422	1750	1759	1626	1244	786	470	217	133	29	12	0	0
Beispiel2.txt	80	4	11	33	106	372	907	1352	1422	1750	1460	1127	842	425	205	124	47	8	0	0	0
Beispiel2.txt	100	4	11	33	106	372	907	1352	1422	1750	1460	1112	744	381	173	116	47	8	0	0	0
Beispiel3.txt	0	4	11	28	65	166	429	1080	2676	6346	11097	14502	14700	12163	8713	6209	4406	2631	1454	626	242
Beispiel3.txt	20	4	11	28	65	166	429	794	1215	1824	2389	2614	2003	1312	797	510	285	167	103	44	38
Beispiel3.txt	40	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
Beispiel3.txt	60	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
Beispiel3.txt	80	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
Beispiel3.txt	100	4	11	28	65	166	429	794	1215	1741	2010	1935	1308	819	447	292	184	91	45	12	0
	Avg	4	11	32	88	268	760	1882	4134	7895	11607	13356	12847	10882	8204	5587	3468	1908	973	426	157

Besuchte Knoten je Rekursionsebene - Distanz als Abbruchbedingung

Beispiel	*p*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
Beispiel1.txt	0	4	11	34	94	265	716	1897	5130	13637	35201	87018	169348	245528	163563	71506	18461	382	0
Beispiel1.txt	20	4	11	34	94	265	716	1897	5130	13637	35233	89395	226555	562530	1268491	1928300	2323628	1885795	498545
Beispiel1.txt	40	4	11	34	94	265	716	1897	5130	13637	35233	89395	226715	582069	1520433	3979921	9353163	18817981	23260008
Beispiel1.txt	60	4	11	34	94	265	716	1897	5130	13637	35233	89395	226715	582069	1522342	4054504	10915529	28972964	72542943
Beispiel1.txt	80	4	11	34	94	265	716	1897	5130	13637	35233	89395	226715	582069	1522342	4054520	10921856	29428028	78340837
Beispiel1.txt	100	4	11	34	94	265	716	1897	5130	13637	35233	89395	226715	582069	1522342	4054520	10921856	29428400	78384136
Beispiel2.txt	0	4	11	33	106	372	1351	4323	9238	9530	3228	476	0	0	0	0	0	0	0
Beispiel2.txt	20	4	11	33	106	372	1361	4781	15337	39202	51991	45620	9017	1026	0	0	0	0	0
Beispiel2.txt	40	4	11	33	106	372	1361	4792	15857	49139	135146	246790	319671	151399	57536	2318	0	0	0
Beispiel2.txt	60	4	11	33	106	372	1361	4792	15869	49698	148639	406962	922055	1563354	1252007	844395	138991	16366	225
Beispiel2.txt	80	4	11	33	106	372	1361	4792	15869	49707	149140	431017	1181521	2921346	5291376	7152724	7017947	2458787	815513
Beispiel2.txt	100	4	11	33	106	372	1361	4792	15869	49707	149140	431299	1208560	3281333	8427899	17539182	30079827	31292510	23675423
Beispiel3.txt	0	4	11	28	65	166	429	1080	2676	6346	14634	33474	65447	99786	74541	32934	8375	97	0
Beispiel3.txt	20	4	11	28	65	166	429	1080	2676	6346	14634	33996	81363	198283	460052	780524	1062988	997801	312106
Beispiel3.txt	40	4	11	28	65	166	429	1080	2676	6346	14634	33996	81371	202123	518823	1347407	3257073	6910675	9733784
Beispiel3.txt	60	4	11	28	65	166	429	1080	2676	6346	14634	33996	81371	202123	519044	1359492	3573048	9201452	22395686
Beispiel3.txt	80	4	11	28	65	166	429	1080	2676	6346	14634	33996	81371	202123	519044	1359492	3573627	9265350	23315893
Beispiel3.txt	100	4	11	28	65	166	429	1080	2676	6346	14634	33996	81371	202123	519044	1359492	3573627	9265364	23319747
	Avg	4	11	32	88	268	835	2563	7493	20382	52025	127756	300882	675631	1397716	2773402	5374444	9885664	19810825

19	20
0	0
123452	2060
24312167	17808772
146088191	247522744
202992199	489196867
204267405	516537966
0	0
0	0
0	0
0	0
25439	222
13055599	1716773
0	0
88353	1770
11237950	9070785
45885448	79406076
56135812	125716035
56276681	129157743
42249372	89785434

Besuchte Knoten je Rekursionsebene - ohne Abbruchbedingung

Beispiel	*p*	*Zeit(ms)*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
Beispiel1.txt	0	99731	4	11	34	94	265	716	1897	5130	13637	35233	89395	226715	582069	1522342	4054520	10921856	29428400
Beispiel2.txt	0	285016	4	11	33	106	372	1361	4792	15869	49707	149140	431299	1208695	3301841	8826222	23135754	59523016	150361785
Beispiel3.txt	0	17308	4	11	28	65	166	429	1080	2676	6346	14634	33996	81371	202123	519044	1359492	3573627	9265364
		Avg	4	11	32	88	268	835	2590	7892	23230	66336	184897	505594	1362011	3622536	9516589	24672833	63018516

18	19	20
78384136	204268203	516757773
373063986	909574216	2180602348
23319747	56276711	129173908
158255956	390039710	942178010

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace BwInf38Runde2Aufgabe3Neu
8  {
9      struct Vertex
10     {
11         //Object inspecific
12         private static int Index;
13         public static void ResetIndex()
14         {
15             Index = 0;
16         }
17
18         //Object specific variables
19         public bool Visited;
20         public int ElementNumber;
21         public double X, Y;
22         public List<int> NeighboorsIndices;
23
24         //Constructor
25         public Vertex(double _X, double _Y)
26         {
27             X = _X;
28             Y = _Y;
29             ElementNumber = Index++;
30             NeighboorsIndices = new List<int>();
31             Visited = false;
32         }
33         public Vertex(double _X, double _Y, int _ElementNumber)
34         {
35             X = _X;
36             Y = _Y;
37             ElementNumber = _ElementNumber;
38             NeighboorsIndices = new List<int>();
39             Visited = false;
40         }
41
42         //Override comparison operators
43         public static bool operator ==(Vertex P1, Vertex P2)
44         {
45             return P1.Equals(P2);
46         }
47         public static bool operator !=(Vertex P1, Vertex P2)
48         {
49             return !P1.Equals(P2);
50         }
51     }
52     struct VertexInfo
53     {
```

```
54          public double DistanceToStartpoint;
55          public int IndexOfPriorPoint;
56          public bool Visited;
57      }
58
59      struct Edge
60      {
61          //Object specific variables
62          public double X1, X2;
63          public double Y1, Y2;
64
65          //Constructor
66          public Edge(Vertex point1, Vertex point2)
67          {
68              X1 = point1.X;
69              Y1 = point1.Y;
70
71              X2 = point2.X;
72              Y2 = point2.Y;
73          }
74      }
75      struct RecursionVertexInfo
76      {
77          //public int PointIndex;
78          public int Turns;
79          public double Distance;
80          public double Angle;
81          public List<int> ListOfPriorPoints;
82
83          public RecursionVertexInfo(RecursionVertexInfo recursionPointInfo)
84          {
85              Turns = recursionPointInfo.Turns;
86              Distance = recursionPointInfo.Distance;
87              Angle = recursionPointInfo.Angle;
88
89              ListOfPriorPoints = recursionPointInfo.ListOfPriorPoints.ToList
                  ();
90          }
91      }
92  }
93
```

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.IO;
 4  using System.Text.RegularExpressions;
 5  using System.Windows;
 6
 7  namespace BwInf38Runde2Aufgabe3Neu
 8  {
 9      class Data
10      {
11          //Datastructures
12          public static Vertex[] ArrayVertices;
13          public static Edge[] ArrayEdges;
14          private static List<Vertex> ListVertices = new List<Vertex>();
15          private static List<Edge> ListEdges = new List<Edge>();
16
17          public static bool NewParameters = false;
18          public static string FileName;
19          public static bool ReadDataFromFile()
20          {
21              try
22              {
23                  //Reset
24                  NewParameters = true;
25                  ListEdges = new List<Edge>();
26                  ListVertices = new List<Vertex>();
27                  ArrayEdges = null;
28                  ArrayVertices = null;
29                  Vertex.ResetIndex();
30
31                  //FileContent
32                  string FileString = string.Empty;
33
34                  //Open File Explorer
35                  Microsoft.Win32.OpenFileDialog Dlg = new
                      Microsoft.Win32.OpenFileDialog()
36                  {
37                      Filter = "\"Abbiegen\"-Datei (*.txt)|*.txt|Alle Dateien
                    (*.*)|*.*",
38                      FilterIndex = 0
39                  };
40                  string SamplePath = AppDomain.CurrentDomain.BaseDirectory +
                      "Material";
41                  Dlg.InitialDirectory = SamplePath;
42
43                  //If Dialog is closed
44                  if (Dlg.ShowDialog() != true)
45                  {
46                      throw new ArgumentNullException();
47                  }
48
49                  //FileName = Dlg.FileName;
50                  FileName = Dlg.SafeFileName;
```

```
51
52                  //Set Streams
53                  FileStream File = new FileStream(Dlg.FileName,
                      FileMode.Open, FileAccess.Read);
54                  StreamReader Reader = new StreamReader(File);
55
56                  //Read File
57                  try
58                  {
59                      FileString = Reader.ReadToEnd();
60                  }
61                  catch
62                  {
63                      MessageBox.Show("Invalid File");
64                      throw new ArgumentNullException();
65                  }
66                  finally
67                  {
68                      File.Close();
69                      Reader.Close();
70                  }
71
72                  //Data Processing
73                  string[] InputArray = Regex.Split(FileString,
                      Environment.NewLine);
74                  FillStartAndEndPoint(InputArray);
75                  FillLists(InputArray);
76                  ArrayVertices = ListVertices.ToArray();
77                  ArrayEdges = ListEdges.ToArray();
78                  AddNeighboorPoints();
79                  return true;
80              }
81              catch
82              {
83                  //throw;
84                  ArrayVertices = null;
85                  ArrayEdges = null;
86                  return false;
87              }
88          }
89          private static void FillLists(string[] InputArray)
90          {
91              //Fill all lines in List
92              Vertex vertex1, vertex2;
93              int Length = int.Parse(InputArray[0]);
94              char[] CharTrim = { ' ', '(', ')' };
95              string[] Row;
96              string[] Column1;
97              string[] Column2;
98              for (int i = 0; i < Length; i++)
99              {
100                 Row = InputArray[i + 3].Split(' ');
101                 Row[0] = Row[0].Trim(CharTrim);
```

```
102                 Column1 = Row[0].Split(',');
103                 Row[1] = Row[1].Trim(CharTrim);
104                 Column2 = Row[1].Split(',');
105 
106                 vertex1 = new Vertex(double.Parse(Column1[0]), double.Parse
                      (Column1[1]), -1);
107                 vertex2 = new Vertex(double.Parse(Column2[0]), double.Parse
                      (Column2[1]), -1);
108 
109                 ListEdges.Add(new Edge(vertex1, vertex2));
110 
111                 bool CheckVertex1 = false;
112                 bool CheckVertex2 = false;
113                 foreach (Vertex vertex in ListVertices)
114                 {
115                     if (vertex.X == vertex1.X && vertex.Y == vertex1.Y)
116                     {
117                         CheckVertex1 = true;
118                     }
119                     if (vertex.X == vertex2.X && vertex.Y == vertex2.Y)
120                     {
121                         CheckVertex2 = true;
122                     }
123                 }
124                 if (!CheckVertex1)
125                 {
126                     vertex1.ElementNumber = ListVertices.Count;
127                     ListVertices.Add(vertex1);
128                 }
129                 if (!CheckVertex2)
130                 {
131                     vertex2.ElementNumber = ListVertices.Count;
132                     ListVertices.Add(vertex2);
133                 }
134             }
135         }
136         private static void FillStartAndEndPoint(string[] InputArray)
137         {
138             //Add start- and endpoint to list
139             char[] CharTrim = { ' ', '(', ')' };
140 
141             InputArray[1] = InputArray[1].Trim(CharTrim);
142             string[] Column = InputArray[1].Split(',');
143             ListVertices.Add(new Vertex(double.Parse(Column[0]),
                  double.Parse(Column[1]), 0));
144 
145             InputArray[2] = InputArray[2].Trim(CharTrim);
146             Column = InputArray[2].Split(',');
147             ListVertices.Add(new Vertex(double.Parse(Column[0]),
                  double.Parse(Column[1]), 1));
148         }
149 
150         private static void AddNeighboorPoints()
```

```
151         {
152             //Adds all neighboor indices to all points
153             Edge line;
154             Vertex point;
155             int Index1 = -1;
156             int Index2 = -1;
157             for (int i = 0; i < ArrayEdges.Length; i++)
158             {
159                 line = ArrayEdges[i];
160 
161                 for (int j = 0; j < ArrayVertices.Length; j++)
162                 {
163                     point = ArrayVertices[j];
164                     if (line.X1 == point.X && line.Y1 == point.Y)
165                     {
166                         Index1 = j;
167                     }
168                     else if (line.X2 == point.X && line.Y2 == point.Y)
169                     {
170                         Index2 = j;
171                     }
172                 }
173                 ArrayVertices[Index1].NeighboorsIndices.Add(Index2);
174                 ArrayVertices[Index2].NeighboorsIndices.Add(Index1);
175             }
176         }
177         public static void FindBoundaries(ref double _MinX, ref double
              _MaxX, ref double _MinY, ref double _MaxY)
178         {
179             //Find the boundaries for the coordinate system
180             Vertex point;
181             double MinX = double.MaxValue;
182             double MaxX = double.MinValue;
183             double MinY = double.MaxValue;
184             double MaxY = double.MinValue;
185 
186             for (int i = 0; i < ArrayVertices.Length; i++)
187             {
188                 point = ArrayVertices[i];
189 
190                 if (point.X < MinX)
191                 {
192                     MinX = point.X;
193                 }
194                 else if (point.X > MaxX)
195                 {
196                     MaxX = point.X;
197                 }
198                 if (point.Y < MinY)
199                 {
200                     MinY = point.Y;
201                 }
202                 else if (point.Y > MaxY)
```

```
203                 {
204                     MaxY = point.Y;
205                 }
206             }
207 
208             _MinX = MinX;
209             _MaxX = MaxX;
210             _MinY = MinY;
211             _MaxY = MaxY;
212         }
213         public static double CalculateLength(Vertex P1, Vertex P2)
214         {
215             //Calculates the length between to points
216             double dx = P1.X - P2.X;
217             double dy = P1.Y - P2.Y;
218 
219             return Math.Sqrt(dx * dx + dy * dy);
220         }
221         public static double CalculateAngle(Vertex P1, Vertex P2)
222         {
223             //Calculates the angle between the line that connects the two
                points and the x-axis
224             double dx = P1.X - P2.X;
225             double dy = P1.Y - P2.Y;
226 
227             double angle = Math.Atan(dy / dx) * 360 / (2 * Math.PI);
228             if (angle == -90)
229             {
230                 angle = 90;
231             }
232 
233             return angle;
234         }
235         public static int CalculateMaxTurns()
236         {
237             //Calculates the amount of turns in the shortes path
238             int Turns = 0;
239             double angle1, angle2;
240             for (int i = 0; i < Dijkstra.ShortestPath.Length - 2; i++)
241             {
242                 angle1 = CalculateAngle(Dijkstra.ShortestPath[i],
                    Dijkstra.ShortestPath[i + 1]);
243                 angle2 = CalculateAngle(Dijkstra.ShortestPath[i + 1],
                    Dijkstra.ShortestPath[i + 2]);
244 
245                 if (angle1 != angle2)
246                 {
247                     Turns++;
248                 }
249             }
250             return Turns;
251         }
252     }
```

```
253  }
254
```

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6  using System.Windows;
 7
 8  namespace BwInf38Runde2Aufgabe3Neu
 9  {
10      class Dijkstra
11      {
12          //Datastructures
13          private static double MinPathLength;
14          private static VertexInfo[] ArrayPointInfo = null;
15          public static Vertex[] ShortestPath = null;
16
17          public static void FindShortestPathLength()
18          {
19              //Preparation
20              Prepare();
21              Vertex CurrentPoint = Data.ArrayVertices[0];
22              Vertex StartPoint = Data.ArrayVertices[0];
23              Vertex EndPoint = Data.ArrayVertices[1];
24
25              //Dijkstra
26              do
27              {
28                  FindNeighboorsOfCurrentPointAndUpdateDistance(CurrentPoint);
29                  CurrentPoint = FindNewShortestPointToStartpoint();
30                  ArrayPointInfo[CurrentPoint.ElementNumber].Visited =  true;
31              }
32              while (CurrentPoint != EndPoint);
33
34              //Set lowest path length
35              MinPathLength = ArrayPointInfo[1].DistanceToStartpoint;
36
37              //Create ShortestPath
38              int IndexNewPoint;
39              List<Vertex> ListShortestPath = new List<Vertex>();
40              ListShortestPath.Add(EndPoint);
41              CurrentPoint = EndPoint;
42              do
43              {
44                  IndexNewPoint = ArrayPointInfo                                  ↵
                      [CurrentPoint.ElementNumber].IndexOfPriorPoint;
45                  CurrentPoint = Data.ArrayVertices[IndexNewPoint];
46                  ListShortestPath.Add(CurrentPoint);
47
48              }
49              while (CurrentPoint != StartPoint);
50              ShortestPath = ListShortestPath.ToArray();
51
52          }
```

```
53          private static void Prepare()
54          {
55              //Set Structure
56              ArrayPointInfo = new VertexInfo[Data.ArrayVertices.Length];
57
58              //Starting point
59              ArrayPointInfo[0].IndexOfPriorPoint = -1;
60              ArrayPointInfo[0].DistanceToStartpoint = 0;
61              ArrayPointInfo[0].Visited = true;
62
63              //Other points
64              for (int i = 1; i < ArrayPointInfo.Length; i++)
65              {
66                  ArrayPointInfo[i].DistanceToStartpoint = double.MaxValue;
67                  ArrayPointInfo[i].IndexOfPriorPoint = -1;
68                  ArrayPointInfo[i].Visited = false;
69              }
70          }
71          private static void FindNeighboorsOfCurrentPointAndUpdateDistance
              (Vertex CurrentPoint)
72          {
73              //Checks and may updates neighboor points
74              int IndexNeighboorPoint;
75              VertexInfo CurrentPointInfo = ArrayPointInfo
                  [CurrentPoint.ElementNumber];
76              for (int i = 0; i < CurrentPoint.NeighboorsIndices.Count; i++)
77              {
78                  IndexNeighboorPoint = CurrentPoint.NeighboorsIndices[i];
79
80                  double DistanceOld = ArrayPointInfo
                      [IndexNeighboorPoint].DistanceToStartpoint;
81                  double DistanceNew = CurrentPointInfo.DistanceToStartpoint +
                       Data.CalculateLength(CurrentPoint, Data.ArrayVertices
                      [IndexNeighboorPoint]);
82                  if (DistanceOld > DistanceNew)
83                  {
84                      ArrayPointInfo[IndexNeighboorPoint].DistanceToStartpoint
                       = DistanceNew;
85                      ArrayPointInfo[IndexNeighboorPoint].IndexOfPriorPoint =
                      CurrentPoint.ElementNumber;
86                  }
87              }
88          }
89          private static Vertex FindNewShortestPointToStartpoint()
90          {
91              //Finds the next shortest point to the starting point
92              int IndexShortestPointToStartpoint = -1;
93              double ShortestDistance = double.MaxValue;
94              for (int i = 0; i < ArrayPointInfo.Length; i++)
95              {
96                  if (!ArrayPointInfo[i].Visited && ShortestDistance >=
                      ArrayPointInfo[i].DistanceToStartpoint)
97                  {
```

```
 98                      ShortestDistance = ArrayPointInfo
                        [i].DistanceToStartpoint;
 99                      IndexShortestPointToStartpoint = i;
100                  }
101              }
102              return Data.ArrayVertices[IndexShortestPointToStartpoint];
103          }
104          public static double GetMinPathLength()
105          {
106              return MinPathLength;
107          }
108      }
109
110  }
111
```

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace BwInf38Runde2Aufgabe3Neu
5  {
6      class Recursion
7      {
8          private static double Epsilon = Math.Pow(10.0, -12.0);
9
10         private static int MaxTurns;
11         private static double MaxPathLength;
12         private static double BestPathLength;
13
14         private static Vertex StartPoint;
15         private static Vertex EndPoint;
16         private static RecursionVertexInfo RecommendedPathInfo;
17
18         public static Vertex[] FindRecommendedPath(double Percent)
19         {
20             //Set boundaries
21             MaxPathLength = Dijkstra.GetMinPathLength() * (1 + Percent /
                 100);
22             BestPathLength = MaxPathLength * (1 + Epsilon);
23             MaxTurns = Data.CalculateMaxTurns();
24             StartPoint = Data.ArrayVertices[0];
25             EndPoint = Data.ArrayVertices[1];
26
27             //Prepare datastructure for recursion
28             Vertex NeighboorPoint;
29             RecursionVertexInfo PriorPointInfo = new RecursionVertexInfo();
30             PriorPointInfo.ListOfPriorPoints = new List<int>();
31             PriorPointInfo.ListOfPriorPoints.Add(0);
32             Data.ArrayVertices[0].Visited = true;
33
34             //Start RecursionMethod
35             for (int i = 0; i < StartPoint.NeighboorsIndices.Count; i++)
36             {
37                 NeighboorPoint = Data.ArrayVertices
                     [StartPoint.NeighboorsIndices[i]];
38                 PriorPointInfo.Angle = Data.CalculateAngle(StartPoint,
                     NeighboorPoint);
39                 RecursionMethod(PriorPointInfo, StartPoint, NeighboorPoint);
40             }
41
42             //Create recommended path
43             List<Vertex> ListRecommendedPath = new List<Vertex>();
44             for (int i = 0; i < RecommendedPathInfo.ListOfPriorPoints.Count;
                   i++)
45             {
46                 ListRecommendedPath.Add(Data.ArrayVertices
                     [RecommendedPathInfo.ListOfPriorPoints[i]]);
47             }
48             return ListRecommendedPath.ToArray();
```

```
49          }
50          private static void RecursionMethod(RecursionVertexInfo
              PriorPointInfo, Vertex PriorPoint, Vertex CurrentPoint)
51          {
52              //Get data from PriorPoint
53              int Turns = PriorPointInfo.Turns;
54              double Distance = PriorPointInfo.Distance;
55              double AngleOld = PriorPointInfo.Angle;
56
57              //Check new distance
58              Distance += Data.CalculateLength(PriorPoint, CurrentPoint);
59              if (Distance > MaxPathLength)
60              {
61                  return;
62              }
63
64              //Check angle and may adjust turns
65              double AngleNew = Data.CalculateAngle(PriorPoint, CurrentPoint);
66              if (Math.Abs(AngleNew - AngleOld) > Epsilon && ++Turns >
                  MaxTurns)
67              {
68                  return;
69              }
70
71              //Set current point
72              PriorPointInfo.ListOfPriorPoints.Add
                  (CurrentPoint.ElementNumber);
73              PriorPointInfo.Distance = Distance;
74              PriorPointInfo.Turns = Turns;
75              PriorPointInfo.Angle = AngleNew;
76
77              //May update recommended path
78              if (Distance < MaxPathLength * (1 + Epsilon))
79              {
80                  if ((CurrentPoint == EndPoint) && (Turns < MaxTurns ||
                      (Turns == MaxTurns && Distance < BestPathLength)))
81                  {
82                      MaxTurns = Turns;
83                      BestPathLength = Distance;
84                      RecommendedPathInfo = new RecursionVertexInfo
                        (PriorPointInfo);
85                  }
86              }
87
88
89              //Mark current point as visited
90              Data.ArrayVertices[CurrentPoint.ElementNumber].Visited = true;
91
92              //Calls itself if unvisited neighboors exist
93              for (int i = 0; i < CurrentPoint.NeighboorsIndices.Count; i++)
94              {
95                  int IndexNeighboor = CurrentPoint.NeighboorsIndices[i];
96                  Vertex Neighboor = Data.ArrayVertices[IndexNeighboor];
```

```
97                  if (!Neighboor.Visited)
98                  {
99                      RecursionMethod(PriorPointInfo, CurrentPoint,
                        Neighboor);
100                 }
101             }
102
103             //Reset referance changes
104             PriorPointInfo.ListOfPriorPoints.Remove
                  (CurrentPoint.ElementNumber);
105             Data.ArrayVertices[CurrentPoint.ElementNumber].Visited = false;
106         }
107
108         public static int GetNumberOfTurns()
109         {
110             return MaxTurns;
111         }
112         public static double GetPathLength()
113         {
114             return Math.Round(BestPathLength * 1000) / 1000;
115         }
116     }
117 }
118
```