Aufgabe 3: Abbiegen?

Teilnahme-Id: 9693

Bearbeiter/-in dieser Aufgabe:
Nick Djerfi

20. April 2020

Inhaltsverzeichnis

1 Lösungsidee

Das Problem kann man als Graphen sehen, auf dem man den kürzesten Pfad berechnen muss. Der Algorithmus benutzt für die Berechnung zwei Pathfinding Algorithmen. Einmal wird der allgemein kürzeste Pfad berechnet und dannach wird der optimale Pfad mit den wenigsten Kurven berechnet.

1.1 Kürzester Pfad

Damit man weiß, wie lang ein berechneter Pfad maximal sein kann wird zuerst der Dijkstra Algorithmus verwendet, bei dem das Gewicht einer Kante durch ihre Länge gegeben ist. Dies gibt den allgemein kürzesten Pfad aus, von dem man die maximale Länge mit der gegebenen Verlängerung berechnen kann.

1.2 Optimaler Pfad

Wenn man nun die Minimierung der Kurvenanzahl als Ziel setzt, stößt man mit dem Dijkstra Algorithmus auf ein Problem. Das Gewicht einer Kante ist nun abhängig vom zuvor besuchten Knoten. (siehe Abb. 1) Dies macht den zuvor statischen Graphen zu einem dynamischen, auf dem der Dijkstra Algorithmus nicht den optimalen Pfad findet.

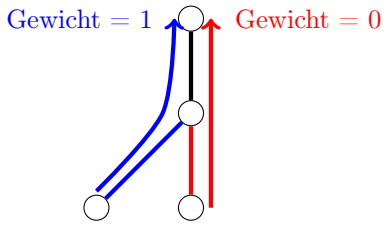


Abbildung 1: Dynamisches Kantengewicht

1.2.1 Problem mit üblichen Pathfinding Algorithmen

Mit üblichen Pathfinding Algorithmen, wie Dijkstra, BFS, Bellman-Ford, usw., kommt man aufgrund der Dynamik des Graphen auf falsche Lösungen. Das Problem liegt darin, dass jeder Knoten nur einmal besucht wird. Dadurch kann es vorkommen, dass der Algorithmus einen Pfad wählt, der mit der nächsten Kante eine weitere Kurve hat und somit um eine Kurve zum falschen Ergebnis kommt. Das folgende Beispiel erläutert dies anhand des Dijkstra Algorithmus.

In Abbildung 2 kommt das Problem am rot markierten Knoten auf. Zuerst wird dieser vom Knoten darüber links auf eine Distanz von 1 gesetzt. Daraufhin wird der nächste Knoten links besucht, aber der Vorgänger des markierten Knoten nicht erneuert, da ebenfalls eine Distanz von 1 berechnet wird. Letzenendlich wird am Zielknoten eine Distanz von 2 berechnet, da der Vorgänger des markierten Knoten zur Berechnung des Winkels herangezogen wird.

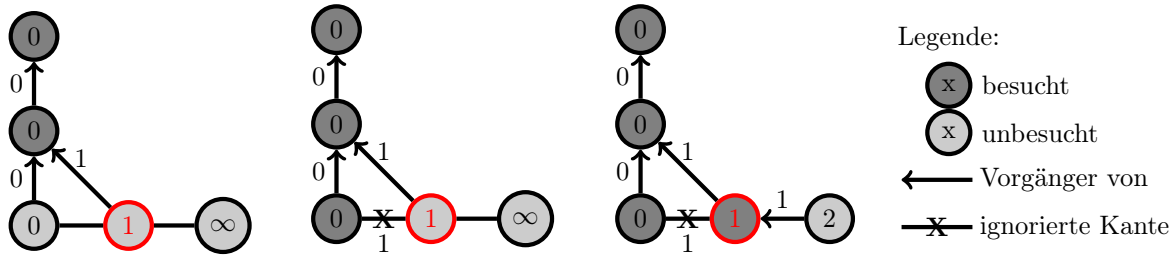


Abbildung 2: Dijkstra falsches Ergebnis

In Abbildung 3 hingegen kommt der Dijkstra Algorithmus auf das richtige Ergebnis. Hier wird, durch Verschieben des Zielknotens, der gewählte Pfad zum richtigen.

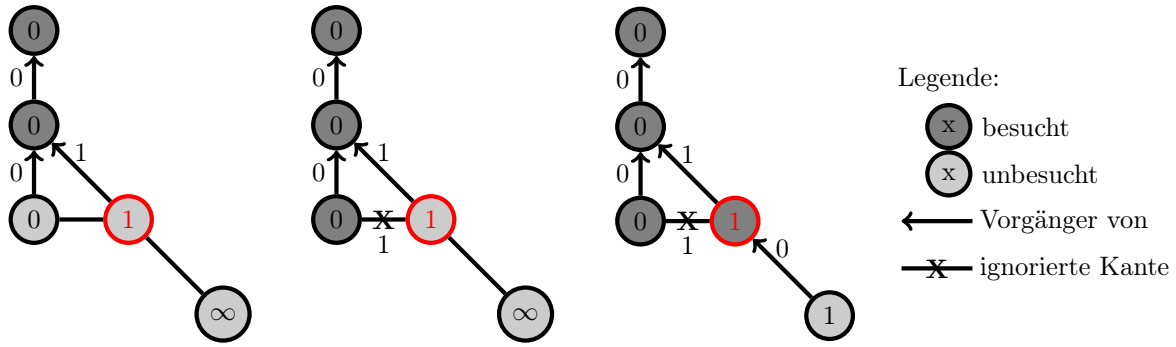


Abbildung 3: Dijkstra richtiges Ergebnis

1.2.2 Kurvenbestimmung

Ob eine Kante eine Kurve bildet kann berechnet werden, indem man die zuvor begangene Kante hinzunimmt, um drei Punkte zu erhalten.

$$\vec{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix}; \vec{b} = \begin{bmatrix} x_b \\ y_b \end{bmatrix}; \vec{c} = \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

Nun nimmt man einen dieser Punkte als Zentrum und bildet beide Vektoren zwischen den Punkten und dem Zentrum.

$$\overrightarrow{AB} = \vec{b} - \vec{a} = \begin{bmatrix} x_b - x_a \\ y_b - y_a \end{bmatrix}; \overrightarrow{AC} = \vec{c} - \vec{a} = \begin{bmatrix} x_c - x_a \\ y_c - y_a \end{bmatrix}$$

Das Kreuzprodukt der beiden Vektoren ergibt 0 wenn die Punkte kollinear sind, also keine Kurve bilden.

$$\overrightarrow{AB} \times \overrightarrow{AC} = \vec{0}$$

$$(x_b - x_a)(y_c - y_a) - (y_b - y_a)(x_c - x_a) = 0$$

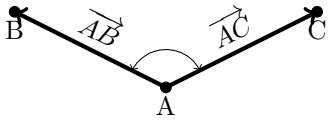


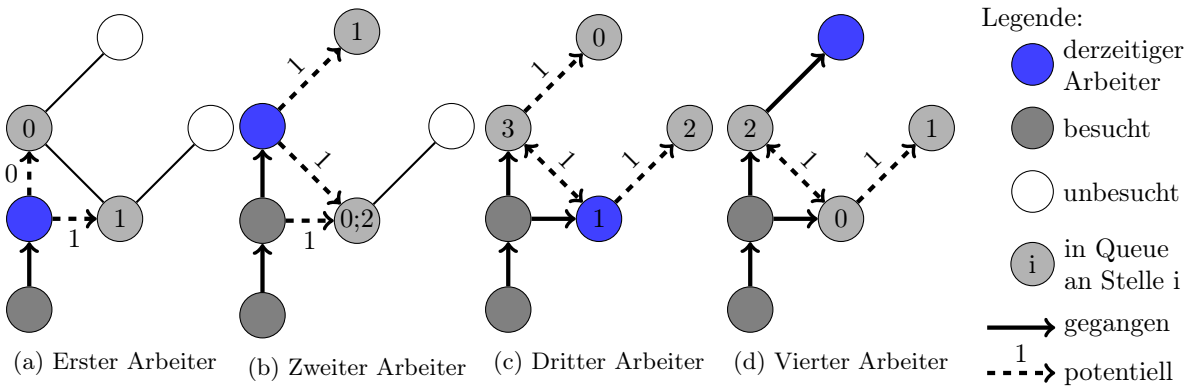
Abbildung 4: Kurvenbestimmung

1.3 Pathfinding Algorithmus

Um das zuvor genannte Problem zu beheben, braucht man einen Algorithmus, der Knoten mehrmals besuchen kann. Dazu dient ein modifizierter Dijkstra Algorithmus, der nicht die Knoten abspeichert, sondern die Pfade. In einer Warteschlange wird dann jeder Knoten zusammen mit dem bisherigen Pfad als "Arbeiter" abgearbeitet. Damit am Ende der Arbeiter mit dem optimalen Pfad zuerst das Ziel erreicht ist die Warteschlange eine Prioritätswarteschlange, sortiert nach der Gesamtkurvenanzahl der Arbeiter. Somit wird immer der Arbeiter mit den wenigsten Kurven ausgesucht. Dies gibt die optimale Lösung, da der Graph keine negativen Kantengewichte enthält und somit die Kurvenanzahl nie weniger als die des ersten Arbeiters in der Prioritätswarteschlange werden kann.

In Abbildung 5 entspringen dem ersten Arbeiter zwei neue Arbeiter. Einer geht zum Knoten darüber und hat eine Gesamtkurvenanzahl von 0. Der andere geht zum Knoten rechts und hat eine Gesamtkurvenanzahl von 1. Damit ist der obere Arbeiter in der Prioritätswarteschlange auf Index 0 und der rechte auf Index 1. Wenn der nächste Arbeiter von der Warteschlange genommen wird, entspringen dem nun ein Arbeiter beim Knoten darüber rechts und ein Arbeiter an der selben Stelle wie der vorige. Man sieht hier, dass der Algorithmus Knoten öfters besuchen kann, da ein Knoten unter zwei verschiedenen Pfaden in der Prioritätswarteschlange liegt. Da nun alle Knoten die gleiche Gesamtkurvenanzahl haben wird

einfach der erste aus der Warteschlange genommen und wieder neue Arbeiter generiert. Wenn nun der oberste Knoten der Zielknoten ist, wird mit dem vierten Arbeiter die Suche abgebrochen und der Pfad des Arbeiters zurückgegeben.



(a) Erster Arbeiter (b) Zweiter Arbeiter (c) Dritter Arbeiter (d) Vierter Arbeiter

Abbildung 5: Modifizierter Dijkstra

1.3.1 Einhaltung der Verlängerung

Damit jeder Arbeiter zu jedem Zeitpunkt die maximale Verlängerung einhält, muss man zu jedem Knoten die Distanz zum Start- bzw. Zielknoten abspeichern. Diese Informationen werden vom Dijkstra Algorithmus automatisch berechnet. Bei dieser Dijkstra Implementation wird für jeden Knoten der Vorgänger und die Distanz zum Startknoten abgespeichert. So erhält man eine Lookup-Table, die diese Informationen für jeden Knoten enthält. Da die Distanz zum Startknoten gespeichert wird, wird der modifizierte Dijkstra vom Zielknoten aus begonnen. Addiert man die bereits gegangene Strecke eines Arbeiters mit der Mindestdistanz zwischen seiner Position und dem Startknoten, erhält man die Mindestdistanz des Arbeiters und kann diese mit der maximalen Verlängerung vergleichen. Ist die Mindestdistanz größer als die maximal erlaubte Distanz wird der Arbeiter verworfen.

1.4 Additionen

Der Algorithmus lässt sich durch die Abstraktion mittels Arbeitern leicht verallgemeinern.

1.4.1 K-kürzeste Pfade

Dass der Algorithmus mehrere Pfadmäglichkeiten gleichzeitig berechnet, macht es möglich, ihn allgemein für das Problem, K-kürzeste Pfade zu finden, zu benutzen. So kann man beim Ausführen des Programms einen weiteren Parameter angeben, der angibt wie viele Pfade generiert werden sollen. Im Algorithmus wird dann nicht nach dem ersten fertigen Arbeiter abgebrochen, sondern erst nachdem die angegebene Anzahl an Arbeitern das Ziel erreicht hat.

1.4.2 Kürzester optimaler Pfad

Wenn man das Sortierungskriterium der Prioritätswarteschlange erweitert, kann man mit dem Algorithmus aus mehreren optimale Pfaden auch den kürzesten optimalen Pfad finden. Da jeder Arbeiter zur Gesamtkurvenanzahl auch die Gesamtdistanz speichert muss nur der Vergleichsoperator so angepasst werden, dass bei gleicher Gesamtkurvenanzahl die Gesamtdistanz beider Arbeiter verglichen wird.

2 Umsetzung

Die Implementation des Algorithmus besteht aus zwei Methoden, die den Dijkstra und den Algorithmus implementieren.

2.1 Übersetzen von Punkten zu Indizes

Um die Laufzeit und Leserlichkeit zu verbessern wird jeder Punkt im Straßennetz in eine Liste gespeichert. In den Implementationen beider Pathfinding Algorithmen werden nur noch die Indizes der Punkte verwendet. Erst bei der Ausgabe werden die Indizes wieder in Punkte übersetzt und ausgegeben. Dies hat den Vorteil, dass Listen für schon besuchte Knoten und Vorgängern als einfache Arrays realisiert werden können und nicht über Hashmaps oder ähnlichem implementiert werden müssen.

2.2 Datentypen

Der Graph wird als Adjazenzliste gespeichert, da dies die Implementation von Pathfinding Algorithmen sehr vereinfacht. Die Lookup-Table zur Übersetzung von Punkten ist ein einfaches Array an Punkten, in dem der Index eines Punktes diesen repräsentiert.

2.2.1 Worker Struct

Das Worker Struct speichert einen Arbeiter ab. Dieses speichert die derzeitige Position, die Gesamtdistanz, die Gesamtkurvenanzahl und den gegangenen Pfad des Arbeiters. Weiterhin definiert es für die Prioritätswarteschlange im Algorithmus Vergleichsoperatoren, die Arbeiter erst nach Gesamtkurvenanzahl und dann nach Gesamtdistanz sortieren.

2.2.2 Predecessor Struct

Das Predecessor Struct speichert lediglich den Vorgänger und die Distanz zum Startknoten eines Knotens für den Dijkstra Algorithmus.

2.3 Dijkstra Implementation

Die Dijkstra Implementation befindet sich in der Methode `findShortestPath` und ist, abgesehen von ein paar Ungewohnheiten durch C++, wie gewohnt über eine Prioritätswarteschlange implementiert. Diese Prioritätswarteschlange wird solange durchlaufen, bis diese leer ist.

```
while(!queue.empty())
{
  // Zu bearbeitender Knoten wird aus der Queue genommen
  uint32_t u = queue.top().second;
  queue.pop();

  [ ... ]
}
```

Dadurch, dass die Prioritätswarteschlange der STL den Vergleichsoperator des Objekts benutzt, speichert die Warteschlange Distanz-Knoten Paare, die dann nach Distanz sortiert werden.

```
// Distanz-Knoten Paar
typedef std::pair<float, uint32_t> Distance;

// Priority Queue sortiert nach Distanz (std::pair vergleicht zuerst ersten Wert)
std::priority_queue<Distance, std::vector<Distance>, std::greater<Distance>> queue;
```

Außerdem kann man die Werte in der STL Prioritätswarteschlange nicht ändern. Deswegen werden die neuen Distanz-Knoten Paare einfach in die Warteschlange eingefügt, ohne die alten zu löschen. Dies hat nur eine minimale Auswirkung auf die Laufzeit, da schon besuchte Knoten übersprungen werden und dies durch die Implementierung als Array in konstanter Zeit möglich ist.

```
// Speichert fuer alle Knoten, ob diese schon besucht wurden
std::vector<bool> visited(lookup.size(), false);

  [ ... ]

// Schon besuchte Knoten ueberspringen
if (visited[u])
  continue;

// Knoten als besucht markieren
visited[u] = true;
```

```
  [ ... ]

// Nachbarknoten in die Queue pushen, da es in C++ keinen leichten
// Weg gibt Elemente einer std::priority_queue zu aendern
queue.push(std::make_pair(distance, v));
```

Am Ende wird ein Array an Predecessor Structs zurückgegeben, dass für jeden Knoten den Vorgänger und die Distanz zum Startknoten abspeichert. Dieses Array wird für jeden Nachbarn, der in die Prioritätswarteschlange eingefügt wird, erneuert.

```
// Alle benachbarten Knoten durchlaufen
for (auto v : adjList[u])
{
  // Schon besuchte ueberspringen
  if (visited[v])
    continue;

  // Neue Distanz berechnen
  float distance = predecessor[u].distance + getDist(u, v);

  // Mit alter Distanz vergleichen
  if (distance < predecessor[v].distance)
  {
    // Vorgaenger und Distanz updaten
    predecessor[v].predecessor = u;
    predecessor[v].distance = distance;

    // Nachbarknoten in die Queue pushen, da es in C++ keinen leichten
    // Weg gibt Elemente einer std::priority_queue zu aendern
    queue.push(std::make_pair(distance, v));
  }
}
```

2.4 Pathfinding Algorithmus Implementation

Der Pathfinding Algorithmus befindet sich in der `findFewestTurnPaths` Methode. Diese nimmt das zuvor berechnete Array an Predecessor Structs und die Anzahl zu berechnende Pfade an und gibt ein Array an Arbeiter Structs zurück, das die gewünschte Anzahl an Pfaden speichert. Zuerst wird eine Prioritätswarteschlange an Arbeitern mit dem ersten Arbeiter am Zielknoten initialisiert und die maximale Länge eines Pfades berechnet.

```
// Priority Queue an Arbeitern (werden nach Kurvenanzahl und Gesamtdistanz sortiert)
std::priority_queue<Worker, std::vector<Worker>, std::greater<Worker>> queue;

// Anfangsarbeiter in die Queue pushen
queue.push(Worker(end));

// Maximaldistanz berechnen (Kuerzester Pfad * Verlaengerungsfaktor)
float maxDistance = predecessor[end].distance * maxPercentage;
```

Dann wird, wie auch in der Dijkstra Implementation, die Queue solange durchlaufen, bis diese leer ist.

```
while (!queue.empty())
{
  // Den naechsten Arbeiter aus der Queue nehmen
  Worker worker = queue.top();
  queue.pop();

  [ ... ]
}
```

Daraufhin wird überprüft, ob der Pfad des Arbeiters die Maximaldistanz einhält. Dazu wird die derzeitige Strecke mit der minimalen Distanz zum Startknoten addiert und mit der zuvor berechneten Maximaldistanz verglichen.

```
// Arbeiter mit zu langem Weg ueberspringen
if (worker.distance + predecessor[worker.position].distance > maxDistance)
  continue;
```

Dannach wird die Position des Arbeiters zu seinem Pfad hinzugefügt und überprüft, ob der Arbeiter am Ziel - beziehungsweise am Anfang - ist. Ist der Arbeiter an seinem Ziel wird er zu der Liste an gefundenen Pfaden hinzugefügt und die Suche abggebrochen, wenn genug Pfade gefunden wurden.

```
// Derzeitige Position zum Pfad hinzufuegen
worker.path.push_back(worker.position);

// Ist der Arbeiter am "Ziel"?
if (worker.position == start)
{
  // Arbeiter/Pfad abspeichern
  paths.push_back(worker);

  // Wenn die gewuenschte Anzahl an Pfaden erreicht wurde abbrechen
  if (paths.size() >= pathCap)
    break;

  // Zum naechsten Arbeiter gehen
  continue;
}
```

Zuletzt werden alle Nachbarknoten des Arbeiters durchlaufen und neue Arbeiter mit den Positionen der Nachbarknoten in die Warteschlange eingefügt. Dazu wird zuerst überprüft, ob der Arbeiter den Nachbarknoten schon mal besucht hat um unnötige Pfade zu eliminieren. Daraufhin wird die neue Gesamtdistanz des Arbeiters berechnet und die neue Kurvenanzahl berechnet, wenn mindestens drei Knoten vorhanden sind. Zuletzt wird ein neuer Arbeiter mit den neuen Daten in die Warteschlange eingefügt.

```
// Alle Nachbarknoten durchgehen
for (uint32_t v : adjList[worker.position])
{
  // Keine schon besuchten Knoten besuchen
  if (std::find(worker.path.begin(), worker.path.end(), v) != worker.path.end())
    continue;

  // Neue Distanz berechnen
  float dist = worker.distance + getDist(worker.position, v);

  // Neue Kurvenanzahl berechnen
  int turns = worker.turns;
  if (worker.path.size() > 1) // Sind 3 Punkte verfuegbar?
    turns += isCurve(worker.position, *(worker.path.end() - 2), v);

  // Neuen Arbeiter in die Queue pushen
  queue.push(Worker(v, dist, turns, worker.path));
}
```

2.5 Übersetzten des Straßennetzes in die Adjazenzliste

Es werden nach und nach die Straßen eingelesen und in die Adjazenzliste und die Lookup-Table eingefügt. Dazu wird für beide Punkte überprüft, ob diese bereits in der Lookup-Table sind. Wenn ja, wird nur der Index an dem sich die Punkte befinden abgespeichert. Wenn nicht, werden sie in die Lookup-Table eingefügt und ebenfalls der Index gespeichert.

```
int nodeA;
Point p = {x1, y1};

// Ist der Knoten schon in der Lookup Table vorhanden?
auto it = std::find(lookup.begin(), lookup.end(), p);
if (it == lookup.end())
{
  // Falls nicht, dann Punkt und Index speichern
  nodeA = lookup.size();
  lookup.push_back(p);
}
else
  // Andernfalls nur den Index des Knoten auslesen
  nodeA = std::distance(lookup.begin(), it);
```

Beide Indizes werden darauf in die Adjazenzliste eingefügt.

```
// Kante in der Adjazenzliste speichern
adjList[nodeA].insert(nodeB);
adjList[nodeB].insert(nodeA);
```

2.6 Laufzeitanalyse

Der Unterschied beider Pathfinding Algorithmen liegt darin, dass der normale Dijkstra Algorithmus auf der Basis von Knoten arbeitet. Der modifizierte Algorithmus dahingegen arbeitet auf der Basis von Pfaden. Generell kann man dann die Anzahl an Knoten $|V|$ durch die Anzahl an möglichen Pfaden $|P|$ ersetzen. Die Anzahl der möglichen Pfade zwischen zwei Punkten sind auf einem dichten Graphen $(|V|-2)!$, oder einfach $|V|!$. Die Laufzeit des Dijkstra Algorithmus kommt stark auf die benutzten Datentypen und den Graphen an. Generell kommt die Laufzeit auf zwei Variablen T_{dk}, die Laufzeit der *decrease-key* Operation des benutzen Datentyps, und T_{em}, die Laufzeit der *extract-minimum* Operation, an.

$$O(|E| * T_{dk} + |V| * T_{em})$$

Für den modifizierten Dijkstra Algorithmus wird hier $|V|$ durch $|V|!$ ersetzt.

$$O(|E| * T_{dk} + |V|! * T_{em})$$

2.6.1 Beobachtungen

Die Straßenkarten aus den Beispielen haben keine überkreuzenden Kanten, weswegen man davon ausgehen kann, dass es sich bei den Straßennetzen um planare Graphen handelt. Ein planarer Graph lässt sich, wie es der Name verrät, in einer Ebene darstellen, ohne dass sich Kanten schneiden. Ebenfalls besitzten die Straßennetze keine Schleifen oder Mehrfachkanten, weswegen die Kantenanzahl durch den eulerschen Polyedersatz beschränkt werden kann.

$$|E| \leq 3|V| - 6$$
$$|E| \in O(|V|)$$

Also gehören planare Graphen zu der Klasse der dünnen Graphen, was die Laufzeit beider Pathfinding Algorithmen verbessert.

$$\text{Normaler Dijkstra: } O(|V| * T_{dk} + |V| * T_{em})$$
$$\text{Modifizierter Dijkstra: } O(|V| * T_{dk} + |V|! * T_{em})$$

Ein weiterer Aspekt planarer Graphen ist, dass der durchschnittliche Knotengrad eines planaren Graphen aufgrund des eulerschen Polyedersatzes immer < 6 ist. Dadurch kann man sagen, dass für große Knotenanzahlen die Anzahl möglicher Pfade wesentlich kleiner ist als $|V|!$. $|V|!$ gilt nämlich nur für komplette Graphen, wo jeder Knoten mit jedem anderen verbunden ist, also $(|V|-1)$ Kanten hat. Ist $|V| \gg 6$ dann ist $|P| \ll |V|!$. Eine Suche nach einer genauen Formel für die Anzahl der Pfade auf Basis des durchschnittlichen Knotengrades ergab keine Ergebnisse.

2.6.2 Eine Formel für das asymptotische Verhalten von $|P|$

Um eine obere Grenze für die Anzahl von Pfaden zu bekommen, kann man alle Kombinationen an Kanten zusammenaddieren. Anhand der Abbildung 6 mit drei Knoten sieht man, dass das Addieren aller Kantenkombinationen von 1 bis $(|V|-1)$ eine obere Grenze für $|P|$ gibt. Hier kann man nämlich entweder zwei Knoten mittels einer Kante, oder drei Knoten mit zwei Kanten verbinden. Gäbe es noch einen vierten Knoten könnte man diese mit drei Kanten verbinden. Die Anzahl an Möglichkeiten, wie man k Kanten auf $|E|$ Kanten verteilen kann, gibt der Binomialkoeffizient. Also ergibt sich die Summe an Binomialkoeffizienten $|E|$ über k von $k=1$ bis $k=|V|-1$ als Obergrenze für $|P|$.

$$|P| = O\left(\sum_{k=1}^{|V|-1} \binom{|E|}{k}\right)$$

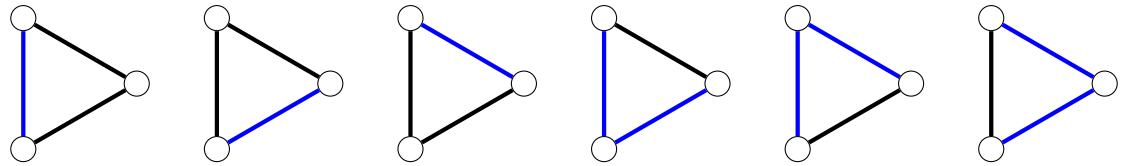


Abbildung 6: $|P| = 6$ bei $|V| = 3$ mit $|E| = 3$ und $O(|P|) = 6$

Nun kann man den Fakt ausnutzen, dass sich die Kantenanzahl auf dünnen Graphen asymptotisch wie die Knotenanzahlen verhält, also $|E| = O(|V|)$ gilt.

$$|P| = O\Big(\sum_{k=1}^{|V|-1} \binom{|V|}{k}\Big)$$

Mithilfe des binomischen Lehrsatzes kann man diese Gleichung lösen.

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

Um die Formel so umzustellen, dass nur noch die Binomialkoeffizienten summiert werden, muss für x und y eine 1 eingesetzt werden.

$$2^n = \sum_{k=0}^{n} \binom{n}{k} * 1^{n-k} * 1^k = \sum_{k=0}^{n} \binom{n}{k}$$

In der Formel für $O(|P|)$ wird nur von 1 bis $|V| - 1$ iteriert, also muss das Ergebnis -2 gerechnet werden, da $\binom{n}{0} = 1$ und $\binom{n}{n} = 1$ ist.

$$2^n = \sum_{k=0}^{n} \binom{n}{k}$$

$$2^n = \sum_{k=1}^{n-1} \binom{n}{k} + \binom{n}{0} + \binom{n}{n}$$

$$2^n = \sum_{k=1}^{n-1} \binom{n}{k} + 2$$

$$2^n - 2 = \sum_{k=1}^{n-1} \binom{n}{k}$$

Ersetzt man nun n durch die Knotenanzahl erhält man das Ergebnis.

$$2^{|V|} - 2 = \sum_{k=1}^{|V|-1} \binom{|V|}{k}$$

$$O(|P|) = 2^{|V|} - 2 = 2^{|V|}$$

Die Laufzeit des modifizierten Dijkstra sieht nun schon besser aus.

$$O(|V| * T_{dk} + 2^{|V|} * T_{em})$$

2.6.3 Implementationsdetails

Die Laufzeit der *pop* und *push* Operationen der STL Prioritätswarteschlange haben beide eine Laufzeit von $O(log(n))$, während die *top* Operation eine konstante Laufzeit $O(1)$ hat. Die *extract-minimum* Operation besteht aus dem Abrufen eines Arbeiters mit *top* und dem darauffolgendem Löschen mit *pop*. Daraus ergibt sich für T_{em} eine Laufzeit von $O(log(n))$. Die *decrease-key* Operation besteht aus einer *push* Operation mit dem neuen Wert, da die Werte in der Warteschlange nur schwer verändert werden konnen. Somit ist die Laufzeit von $T_{dk} = O(log(n))$.

2.6.4 Zusammentragen der Ergebnisse

Wenn nun alle Ergebnisse zusammengetragen werden, kommt man für den normalen Dijkstra Algorithmus auf eine Laufzeit von:

$$O(|E| * T_{dk} + |V| * T_{em})$$
$$O(|V|log(|V|) + |V|log(|V|))$$
$$O((|V| + |V|)log(|V|))$$
$$O(2|V|log(|V|))$$
$$O(|V|log(|V|))$$

Für den modifizierten Dijkstra muss man beachten, dass T_{dk} und T_{em} sich auf die Warteschlange beziehen, die alle Pfade speichert. Deswegen muss hier $|P|$ anstatt $|V|$ verwendet werden.

$$O(|E| * T_{dk} + |P| * T_{em})$$
$$O(|V|log(|P|) + |P|log(|P|))$$
$$O(|V|log(2^{|V|}) + 2^{|V|}log(2^{|V|}))$$
$$O((|V| + 2^{|V|})log(2^{|V|}))$$
$$O(2^{|V|}log(2^{|V|}))$$

Die Laufzeit des gesamten Programms setzt sich aus beiden Laufzeit zusammen. Beide Laufzeit werden miteinander addiert, da jeder Algorithmus einmal nacheinander benutzt wird.

$$O(|V|log(|V|) + 2^{|V|}log(2^{|V|}))$$
$$O(2^{|V|}log(2^{|V|}))$$

Asymptotisch überwiegt der modifizierte Dijkstra, weswegen die Laufzeit des normalen Dijkstra verworfen werden kann.

2.7 Real-world performance

In der Praxis ist der Algorithmus wesentlich schneller, als es die Laufzeitanalyse vermuten lässt. In den folgenden Tabellen sind die Laufzeiten des Programms unter verschiedenen Bedingungen aufgelistet. Damit die Ausgabe der 100000 Pfade auf der Konsole die Zeiterfassung nicht beeinflusst, wurde die Ausgabe hierfür deaktiviert.

In Tabelle 1 sieht man die Laufzeit in Millisekunden, wenn man die Beispiele mit verschiedenen Pfadanzahlen berechnen lässt. Das Längenlimit wurde "deaktiviert" indem die Verlängerung auf 100000% gesetzt wurde. Die Zeiten bei *abbiegen0* sind konstant, da der Algorithmus dort nur maximal 22 Pfade findet und somit der Rechenaufwand nicht steigt.

Der benutzte Befehl ist:

```
$ time ./abbiegen ${file} 100000 -c ${count} -n
```

Zeit t [ms]		Beispiel [Nr.]			
		0	1	2	3
# Pfade	1	5	12	6	7
	10	5	27	8	18
	100	5	80	15	43
	1000	5	266	50	130
	10000	5	1067	554	997
	100000	5	7575	4710	3689

Tabelle 1: Zeiten (versch. Pfadanzahlen, ohne Längenlimit) [Intel i7-6700 @ 3,40 GHz]

In Tabelle 2 sind die Laufzeiten in Millisekunden im Bezug auf verschiedene Verlängerungen angegeben. Hierzu wurde für jedes Beispiel ein Pfad berechnet und jeweils die maximale Verlängerung verändert.

Der benutzte Befehl ist:

```
$ time ./abbiegen ${file} ${length} -n
```

Zeit t [ms]		Beispiel [Nr.]			
		0	1	2	3
Verläng. [%]	10	5	6	6	6
	15	5	6	6	6
	20	5	11	6	7
	30	5	17	6	8
	50	5	29	7	8
	100	5	32	8	10

Tabelle 2: Zeiten (versch. Verlängerungen, ein Pfad) [Intel i7-6700 @ 3,40 GHz]

2.8 Bedienung

Das Programm lässt sich über die Konsole starten und kann einige Kommandozeilenargumente annehmen. Die Ausgabe wird ebenfalls auf der Konsole angezeigt.

2.8.1 Argumente

Zum Ausführen des Programm muss immer die Eingabedatei und die Verlängerung angegeben werden. Wenn nicht anders angegeben ist der erste Parameter die Eingabedatei und der zweite Parameter die Verlängerung.

```
$ ./abbiegen <Eingabedatei> <Verlaengerung>
$ ./abbiegen ../Beispiele/abbiegen0.txt 30
```

Um zum Beispiel die Anzahl der Pfade zu deklarieren, die Ausgabe zu deaktivieren oder die Eingabedatei explizit zu deklarieren gibt es weitere Parameter, die man steuern kann.

kurz	lang	Syntax	Beschreibung
`-i`	`--input`	`-i <Eingabedatei>`	Spezifiziert die Eingabedatei explizit
`-p`	`--percentage`	`-p <Verlaengerung>`	Spezifiziert die Verlängerung explizit
`-c`	`--count`	`-c <Pfadanzahl>`	Spezifiziert die Pfadanzahl
`-d`	`--debug`	`-d`	Gibt zusätzliche Informationen aus
`-n`	`--no-print`	`-n`	Deaktiviert die Ausgabe

2.8.2 Ausgabe

Die Ausgabe zeigt zuerst den optimalen Pfad mit weiteren Eigenschaften an. Wenn mehrere Pfade generiert werden, werden die weiteren Pfade darunter ausgegeben.

```
$ ./abbiegen ../Beispiele/abbiegen2.txt 30 -c 5

Pfad mit den wenigsten Kurven:
(0,0) (1,0) (3,1) (4,1) (5,1) (6,1) (7,1) (8,2) (9,3) (9,2) (9,1) (9,0)
Kurven (bester Pfad): 4
Länge (bester Pfad) : 13.0645
Länge (kürest. Pfad): 10.8863
Verlängerung        : 1.20008 < 1.3

Alternative Pfade:
(0,0) (1,0) (2,0) (4,1) (5,1) (6,1) (7,1) (8,2) (9,3) (9,2) (9,1) (9,0)
Kurven : 4
Länge  : 13.0645
Faktor : 1.20008 < 1.3

(0,0) (1,0) (2,0) (3,0) (4,1) (5,1) (6,1) (7,1) (8,2) (9,3) (9,2) (9,1) (9,0)
Kurven : 4
Länge  : 13.2426
Faktor : 1.21644 < 1.3

(0,0) (1,0) (3,1) (4,1) (5,1) (6,1) (7,1) (8,2) (9,1) (9,0)
Kurven : 5
Länge  : 11.0645
Faktor : 1.01636 < 1.3
```

```
(0,0) (1,0) (2,0) (4,1) (5,1) (6,1) (7,1) (8,2) (9,1) (9,0)
Kurven : 5
Länge  : 11.0645
Faktor : 1.01636 < 1.3
```

3 Beispiele

Im folgenden werden zuerst die Ergebnisse des Programms bei allen Beispieldateien aufgelistet, daraufhin wird anhand des ersten Beispiels die Funktionsweise anschaulich erläutert.

3.1 Ergebnisse

In Tabelle 3 sind die Ergebnisse des Programms bei allen Beispieldateien mit den Verlängerungen 10%, 15%, 20% und 30% aufgelistet. Dabei sind Zeilen, bei denen der Algorithmus auf das gleiche Ergebnis kommt, verbunden um Platz zu sparen. Die Tabelle zeigt für jede Beispieldatei und jede Verlängerung die Anzahl an Kurven des optimalen Pfades, die Länge des optimalen Pfades, die Länge des kürzesten Pfades, den Verlängerungsfaktor zwischen dem optimalen und dem kürzesten Pfad, und den optimalen Pfad.

Datei	Verl.	Kurven	Länge	min. Länge	Faktor	Pfad
abbiegen0	10%	3	5,828	5,828	1,000	(0, 0) (0, 1) (1, 1) (2, 2) (3, 3) (4, 3)
	15%	2	6,414		1,101	(0, 0) (0, 1) (0, 2) (1, 3) (2, 3) (3, 3) (4, 3)
	20%					
	30%	1	7,000		1,201	(0, 0) (0, 1) (0, 2) (0, 3) (1, 3) (2, 3) (3, 3) (4, 3)
abbiegen1	10%	6	17,301	17,122	1,010	(0, 0) (1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (7, 2) (9, 3) (10, 2) (10, 1) (11, 1) (12, 1) (13, 1) (14, 1) (14, 0)
	15%	5	19,122		1,117	(0, 0) (1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (7, 2) (9, 3) (11, 4) (11, 3) (12, 3) (13, 3) (14, 3) (14, 2) (14, 1) (14, 0)
	20%					
	30%					
abbiegen2	10%	5	11,064	10,886	1,016	(0, 0) (1, 0) (3, 1) (4, 1) (5, 1) (6, 1) (7, 1) (8, 2) (9, 1) (9, 0)
	15%					
	20%					
	30%	4	13,064		1,200	(0, 0) (1, 0) (3, 1) (4, 1) (5, 1) (6, 1) (7, 1) (8, 2) (9, 3) (9, 2) (9, 1) (9, 0)
abbiegen3	10%	4	17,886	17,122	1,045	(0, 0) (1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (7, 2) (9, 3) (10, 3) (11, 3) (12, 3) (13, 3) (14, 3) (14, 2) (14, 1) (14, 0)
	15%					
	20%					
	30%					

Tabelle 3: Ergebnisse

3.2 Beispiel an `abbiegen0.txt`

Im folgenden wird die Funktionsweise des Programms an dem Beispielstraßennetz erläutert. Hier wird darauf eingegangen, wie die Rückgabe des normalen Dijkstra Algorithmus benutzt wird, wie der modifizierte Dijkstra den optimalen Pfad findet und wie dieser alternative Pfade findet.

3.2.1 Die Vorgängerkarte des Dijkstra Algorithmus

Am Ende des Dijkstra Algorithmus wird eine Vorgängerkarte zurückgegeben. Diese speichert für jeden Knoten den Vorgänger und die Distanz zum Startknoten ab. In Abbildung 7 ist die Vorgängerkarte des Beispiels abgebildet. Daran kann man erkennen, dass der kürzeste Pfad eine Länge von 5,8 hat und die unteren Knoten entlangläuft.

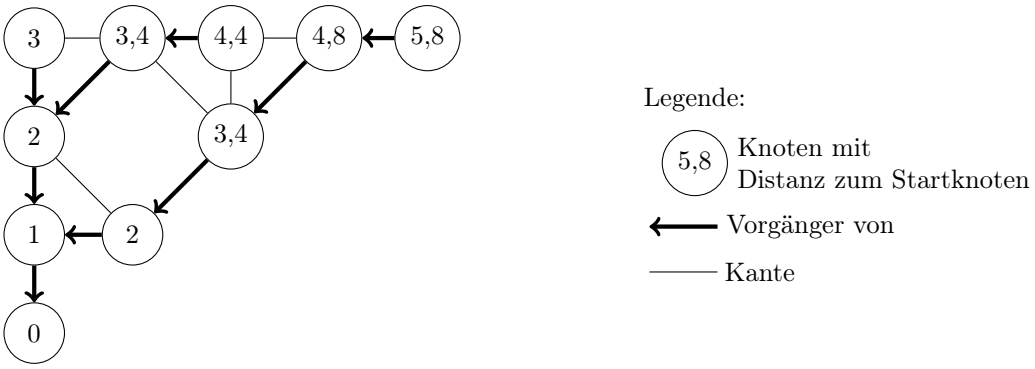
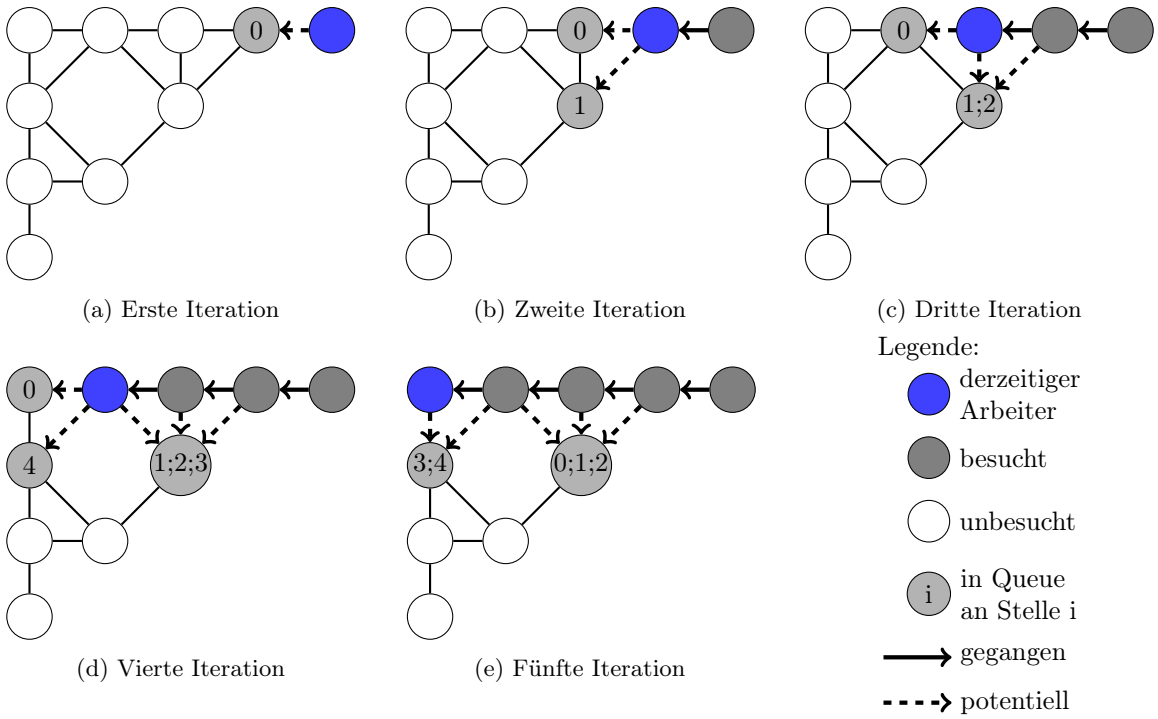


Abbildung 7: Vorgängerkarte

3.2.2 Den optimalen Pfad finden

Da der Dijkstra Algorithmus für jeden Knoten die Minimaldistanz zum Startknoten berechnet hat, wird der modifizierte Dijkstra Algorithmus vom Zielknoten aus gestartet. Dieser läuft zuerst die obere Kante entlang, da jede Kante dort keine Kurve bildet und somit ein Gewicht von 0 hat. In Abbildung 8 sind die ersten fünf Iterationen des Algorithmus dargestellt. Hier wird immer der erste Arbeiter aus der Prioritätswarteschlange genommen, da dieser die beste Kurvenanzahl und den kürzesten Pfad hat. Hieran kann man auch die Verlängerung gut erklären. Jeder Arbeiter speichert nämlich die Länge seines bisherigen Pfades. Dies kann man zusammen mit der Mindestdistanz zum Startknoten verwenden, um unnötige Arbeiter direkt zu verwerfen. Beträgt die Verlängerung 15% ergibt sich aus der Länge 5,8 des kürzesten Pfades eine Maximaldistanz von 6,7. Addiert man in der fünften Iteration die bisherige Distanz 4 des Arbeiters mit der Minimaldistanz 3 zum Startknoten an seiner Position, erhält man die Mindestlänge, die der Arbeiter noch erreichen kann. Ist diese länger als die Maximaldistanz von 6,7 wird der Arbeiter direkt verworfen. Hier beträgt diese Mindestlänge 7 und somit würde der Arbeiter verworfen werden.

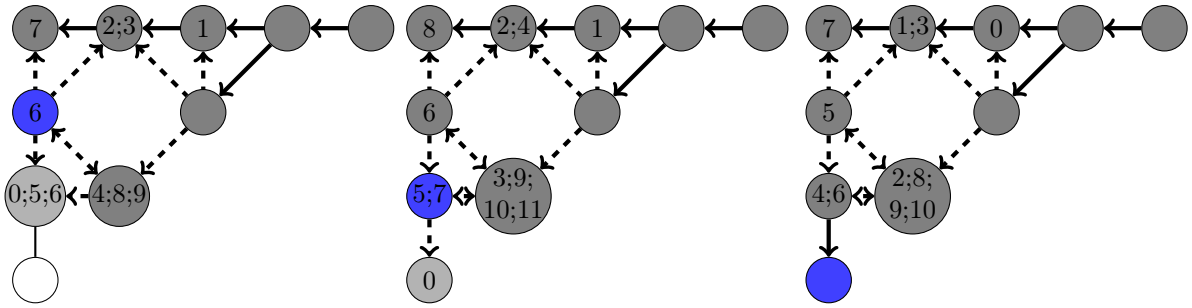


(a) Erste Iteration (b) Zweite Iteration (c) Dritte Iteration

(d) Vierte Iteration (e) Fünfte Iteration

Abbildung 8: Entlang der oberen Kante laufen

In den folgenden Iterationen 6 bis 13 werden nach und nach alle Arbeiter mit einer Kurvenanzahl von 1 abgearbeitet, bis der optimale Pfad gefunden wird. Hier werden die Arbeiter ebenfalls nach Distanz sortiert, also werden zuerst die Arbeiter mit einem kürzeren bisherigen Pfad abgearbeitet. In den letzten drei Iterationen geht der Arbeiter links oben mit dem optimalen Pfad zum Ziel. (siehe Abb. 9)



(a) 11. Iteration (b) 12. Iteration (c) 13. Iteration

Abbildung 9: Zum Ziel laufen

3.2.3 Mehrere optimale Pfade finden

Wenn mehrere Pfade gefragt sind, stoppt der Algorithmus nicht, nachdem der erste Arbeiter am Ziel ist. Hier werden so viele Iterationen durchloffen, bis die gewünschte Anzahl an Arbeitern am Ziel ist. Eine Liste dieser Arbeiter wird dann zurückgegeben.

4 Quellcode

4.1 Struct Definitionen (pathfinder.hpp)

```
// Speichert einen Punkt für die Lookup Table
// und Kurven-/Distanzberechnung
struct Point
{
  int x;
  int y;

  Point(int x, int y);
  Point();

  bool operator==(const Point &b) const;
};

// Speichert von einem Knoten den Vorgaenger und die Gesamtdistanz
// fuer die Dijkstra Implementation
struct Predecessor
{
  uint32_t predecessor; // Vorgaenger
  float distance;       // Gesamtdistanz

  Predecessor(uint32_t predecessor, float distance);
  Predecessor();
};

// Stellt einen "Arbeiter" in der "BFS" Implementation
struct Worker
{
  uint32_t position;          // Derzeitige Position
  float distance;             // Gesamtdistanz
  int turns;                  // Kurvenanzahl
  std::vector<uint32_t> path; // Gegangener Pfad

  Worker(uint32_t start);
  Worker(uint32_t pos, float dist, int turns, std::vector<uint32_t> path);
  Worker();

  // Sortiert zuerst nach Kurvenanzahl, dann nach Distanz
  bool operator>(const Worker &rhs) const;
};
```

4.2 Globale Variablen (pathfinder.hpp)

```cpp
// Stellt eine Lookup Table dar, die
// allen Punkten im Graph eine Nummer zuordnet
extern std::vector<Point> lookup;

// Speichert den Graphen in einer Adjazenzliste ab
extern std::map<uint32_t, std::set<uint32_t>> adjList;

// Nummern des Start- und Endknoten
extern uint32_t start, end;

// Maximale Verlaengerung in Prozent (z.B. 130% = 1,30)
extern float maxPercentage;
```

4.3 Worker Vergleichsoperator (pathfinder.cpp)

```cpp
// Sortiert zuerst nach Kurvenanzahl, dann nach Distanz
bool Worker::operator>(const Worker &rhs) const
{
  if (turns == rhs.turns)
      return distance > rhs.distance;
  return turns > rhs.turns;
}
```

4.4 Helfermethoden (pathfinder.cpp)

```cpp
// Berechnet die Distanz zwischen zwei Punkten
static inline float getDist(uint32_t a, uint32_t b)
{
  //Punkte aus der Lookup-Table holen
  auto pA = lookup[a];
  auto pB = lookup[b];

  if (adjList[a].find(b) == adjList[a].end()) // Keine Kante
    return std::numeric_limits<float>::infinity();

  return std::sqrt(std::pow(pA.x - pB.x, 2) + std::pow(pA.y - pB.y, 2));
}

// Testet ob drei Punkte im Pfad kollinear sind
// c: Derzeitige Position/Punkt
// a, b: nachfolgender bzw. voriger Punkt
static inline int isCurve(uint32_t c, uint32_t a, uint32_t b)
{
  if (a == b)
    return 1;

  //Punkte aus der Lookup-Table holen
  auto pA = lookup[a];
  auto pB = lookup[b];
  auto pC = lookup[c];

  return (pC.x - pA.x) * (pB.y - pA.y) - (pC.y - pA.y) * (pB.x - pA.x) != 0;
}
```

4.5 `findShortestPath` Methode (pathfinder.cpp)

```cpp
// Berechnet für alle Knoten den kuerzesten Pfad zum Startknoten
std::vector<Predecessor> findShortestPath()
{
  // Speichert die Gesamtdistanz von einem Knoten
  typedef std::pair<float, uint32_t> Distance;

  // Priority queue zur Dijkstra Implementation, sortiert nach Distanz eines Knoten
  std::priority_queue<Distance, std::vector<Distance>, std::greater<Distance>> queue;

  // Speichert fuer alle Knoten, ob diese schon besucht wurden
  std::vector<bool> visited(lookup.size(), false);

  // Speichert fuer alle Knoten den Vorgänger und die Gesamtdistanz
  std::vector<Predecessor> predecessor(lookup.size());

  // Startknoten und Queue initialisieren
```

```
  queue.push(std::make_pair(0, start));
  predecessor[start].distance = 0;

  // Bis die Queue leer ist
  while(!queue.empty())
  {
    // Zu bearbeitender Knoten wird aus der Queue genommen
    uint32_t u = queue.top().second;
    queue.pop();

    // Schon besuchte Knoten ueberspringen
    if (visited[u])
      continue;

    // Knoten als besucht markieren
    visited[u] = true;

    // Alle benachbarten Knoten durchlaufen
    for (auto v : adjList[u])
    {
      // Schon besuchte ueberspringen
      if (visited[v])
        continue;

      // Neue Distanz berechnen
      float distance = predecessor[u].distance + getDist(u, v);

      // Mit alter Distanz vergleichen
      if (distance < predecessor[v].distance)
      {
        // Vorgaenger und Distanz updaten
        predecessor[v].predecessor = u;
        predecessor[v].distance = distance;

        // Nachbarknoten in die Queue pushen, da es in C++ keinen leichten
        // Weg gibt Elemente einer std::priority_queue zu aendern
        queue.push(std::make_pair(distance, v));
      }
    }
  }

  // Vorgaengerliste zurückgeben
  return predecessor;
}
```

4.6 `findFewestTurnPaths` Methode (pathfinder.cpp)

```
// Berechnet die angegebene Anzahl an Pfaden mit den wenigsten Kurven
std::vector<Worker> findFewestTurnPaths(
  std::vector<Predecessor> &predecessor,
  size_t pathCap
)
{
  // Priority Queue an Arbeitern (werden nach Kurvenanzahl und Gesamtdistanz sortiert)
  std::priority_queue<Worker, std::vector<Worker>, std::greater<Worker>> queue;

    // Anfangsarbeiter in die Queue pushen
  queue.push(Worker(end));

  // Speichert die gefundenen Pfade ab
  std::vector<Worker> paths;

  // Maximaldistanz berechnen (Kuerzester Pfad * Verlaengerungsfaktor)
  float maxDistance = predecessor[end].distance * maxPercentage;

  // Solange die Queue nicht leer ist
  while (!queue.empty())
  {
    // Den nächsten Arbeiter aus der Queue nehmen
    Worker worker = queue.top();
    queue.pop();

    // Arbeiter mit zu langem Weg ueberspringen
```

```
27      if (worker.distance + predecessor[worker.position].distance > maxDistance)
          continue;
29
        // Derzeitige Position zum Pfad hinzufuegen
31      worker.path.push_back(worker.position);

33      // Ist der Arbeiter am "Ziel"?
        if (worker.position == start)
35      {
          // Arbeiter/Pfad abspeichern
37        paths.push_back(worker);

39        // Wenn die gewuenschte Anzahl an Pfaden erreicht wurde abbrechen
          if (paths.size() >= pathCap)
41          break;

43        // Zum naechsten Arbeiter gehen
          continue;
45      }

47      // Alle Nachbarknoten durchgehen
        for (uint32_t v : adjList[worker.position])
49      {
          // Keine schon besuchten Knoten besuchen
51        if (std::find(worker.path.begin(), worker.path.end(), v) != worker.path.end())
            continue;
53
          // Neue Distanz berechnen
55        float dist = worker.distance + getDist(worker.position, v);

57        // Neue Kurvenanzahl berechnen
          int turns = worker.turns;
59        if (worker.path.size() > 1) // Sind 3 Punkte verfuegbar?
            turns += isCurve(worker.position, *(worker.path.end() - 2), v);
61
          // Neuen Arbeiter in die Queue pushen
63        queue.push(Worker(v, dist, turns, worker.path));
        }
65    }

67    //Pfade zurueckgeben
      return paths;
69  }
```

4.7 Ausführen des Algorithmus (main.cpp)

Hier ist `pathCap` der eingelesene Parameter, welcher angibt, wieviel Pfade generiert werden sollen. Dieser ist, wenn nicht explizit gesetzt, auf 1 gesetzt.

```
1   if (!flags.percentage)
    {
3     if (argc <= optind)
        printUsage(argv);
5
      // Eingegebene Verlaengerung in Multiplikationsfaktor umrechnen
7     maxPercentage = std::stof(argv[optind++]) / 100.f + 1;
    }
9
    // Datei parsen
11  parseFile(filePath);

13  // Vorgaengerkarte berechnen
    auto predecessor = findShortestPath();
15
    // Die gewuenschte Anzahl an Pfaden berechnen
17  auto paths = findFewestTurnPaths(predecessor, pathCap);
```