
Duale Hochschule Baden-Württemberg
Mannheim

# Projektarbeit I

**Variation, Analyse und Verbesserung eines Algorithmus zur heuristischen Lösung des Travelling Salesman Problems**

## Studiengang Wirtschaftsinformatik

Studienrichtung Software Engineering


| | |
|---|---|
| Verfasser/in: | Benno Grimm |
| Matrikelnummer: | 5331201 |
| Firma: | SAP SE |
| Abteilung: | Transportation Management |
| Kurs: | WWI18SEA |
| Studiengangsleiter: | Prof. Dr. Julian Reichwald |
| Wissenschaftlicher Betreuer: | Prof. Dr. Julian Reichwald<br>julian.reichwald@dhbw-mannheim.de<br>+49 (0)621 4105 - 1395 |
| Firmenbetreuer: | Peter Wadewitz<br>peter.wadewitz@sap.com<br>+49 6227 7-63730 |
| Bearbeitungszeitraum: | 13.05.2019 – 01.09.2019 |

# Kurzfassung

| | |
|---|---|
| Titel | Variation, Analyse und Verbesserung eines Algorithmus zur heuristischen Lösung des Travelling Salesman Problems |
| Verfasser/in: | Benno Grimm |
| Kurs: | WWI18SEA |
| Ausbildungsstätte: | SAP SE |

Hier können Sie die Kurzfassung der Arbeit schreiben.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Quelltextverzeichnis

# Algorithmenverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **DHBW** | Duale Hochschule Baden-Württemberg |
| **TSP** | Travelling Salesman Problem |
| **LE** | Längeneinheiten |
| **Alg.** | Algorithmus |

# 1 Einleitung

Das TSP ist... Duale Hochschule Baden-Württemberg (DHBW) DHBW

# 2 Das Travelling Salesman Problem

## 2.1 Beschreibung des TSP

Das TSP beschreibt ein die Herausforderung eines Reisenden, der vor Aufgabe steht eine Route durch eine beliebige Anzahl von Städten so zu planen, dass alle Städte genau einmal besucht werden und die Gesamtdistanz der Route möglichst gering ist. Geprägt wurde dieses Problem durch die Handlungsreisenden, welche schon seit langer Zeit vor eben dieser Aufgabe stehen – möglichst viele Kunden mit möglichst wenig Aufwand (hier gleichzusetzten mit Strecke) zu erreichen. Literatur, welche sich mit eben diesem Problem beschäftigt findet sich im deutschsprächigen Raum bereits in der ersten Hälfte des 19. Jahrhunderts.[1] Warum das TSP als ein Problem bezeichnet wird ist recht einfach zu erklären. Mit steigender Anzahl an Städten, die es zu besuchen gilt, steigt die Menge der möglichen Routen exponentiell an. TODO: Dazu später mehr Studien haben gezeigt, dass Menschen in der Lage sind eine Route durch eine geringe Anzahl an optimal zu planen. Bei etwas weniger als zehn zu planenden Städten liegt die durchschnittliche Gesamtdistanz der vom Mensch geplanten Strecken weniger als 1% über der optimalen Route.[2] Steigt die Anzahl der Städte jedoch weiter an, wird es schwieriger für den Menschen gute Ergebnisse zu produzieren. Beginnt man nun Algorithmen zu entwickeln, um dieses Problem computergestützt du lösen, wird schnell auf die Grenze des Computers erreicht. Zwar gibt es mit fünf Städten noch 120 mögliche Routen, bei zehn Städten werden es bereits 3 628 800.

Um einen Algorithmus zu entwickeln, der fähig ist aus dieser Vielzahl von Möglichkeiten die optimale, oder wenigstens eine gute Route, auszuwählen muss das Problem zuerst mathematisch beschrieben werden.
Das TSP lässt sich am besten als ein Problem der Graphentheorie darstellen. Die Städte und Wege zwischen ihnen werden hier als ein Graph $G$, einer Menge von Knoten $K$ und Kanten $E$ dargestellt und wie folgt definiert:[3]

$$G = (K, E) \text{ mit } K, E \neq \emptyset$$

$$E \subseteq K \times K$$

Da die Knoten des Graph Städte repräsentieren kann jedem Knoten ein $x$- und $y$-Wert zugeordnet werden, die Koordinaten und damit ihre Position darstellen. Ein

[1] Applegate et al. 2006, siehe Abb. 1.1, S. 3.
[2] Macgregor und Ormerod 1996, S. 530.
[3] Domschke et al. 2015, S. 71ff.

Knoten wird durch $k_i$ repräsentiert. Eine Kante $E$ verbindet zwei Knoten in der Form $e_a = (k_b, k_c)$ mit $k_b \neq k_c$ und $a, b, c \in \mathbb{N}$ (mit $0 \notin \mathbb{N}$). Damit gilt $K = \{k_1, k_2, ..., k_n\}$ mit $n > 2, n \in \mathbb{N}$. Die Entfernung zwischen zwei Knoten, also die Länge einer Kante, kann als Abbildung

$$\omega : E \to \mathbb{R}$$

dargestellt werden mit

$$\omega(e_a) = \sqrt{|x_{k_b} - x_{k_c}|^2 + |y_{k_b} - y_{k_c}|^2}$$

was hier gleichbedeutend mit $\omega(k_b, k_c) = \sqrt{|x_{k_b} - x_{k_c}|^2 + |y_{k_b} - y_{k_c}|^2}$ ist.

Ein Lösungsverfahren versucht als einen Pfad $P$ über alle Knoten eines Graph so zu finden, dass gilt

$$\forall k \in P, k_a \neq k_b, a, b \in \mathbb{N}, a \neq b, \text{ sodass für } n \text{ Knoten gilt } \sum_{i=2}^{n} \omega(k_{i-1}, k_i)$$

ist möglichst gering. Ein Pfad beschreibt damit eine Abfolge von Knoten. In dieser Arbeit wird ein Pfad durch

$$P = p_1, \cdots, p_m \text{ mit } m \in \mathbb{N}, m \geq 2, \forall p \in K$$

dargestellt. Die in dieser Arbeit entwickelten Algorithmen versuchen das TSP mit einem bereits gesetzten Ausgangsknoten zu lösen, was impliziert, dass auf einem Pfad $P$ $p_1 = k_1$ ist.

## 2.2 Exakte Lösungsverfahren und die Optimale Route

Wie schon angedeutet bedarf die Ermittlung der optimalen Lösung eines enormen Rechenaufwands für einen Computer dar. Bei $n$ Knoten und einem bereits gesetzten $p_1$ bestehen $n-1$ Optionen für $p_2$, $n-2$ für $p_3$ und so weiter. Die Gesamtanzahl aller möglichen Routen berechnet sich also aus

$$\prod_{i=1}^{n-1} i = (n-1)!$$

Für einen Algorithmus, der alle diese Lösungen mit einer Brute-Force-Methode berechnet, um die beste unter ihnen zu finden, bedeutet das eine Zeitkomplexität von $O((n-1)!)$ und somit einen extrem schnellen Anstieg der Rechenzeit bei steigern Eingabemenge.[4] Es existieren zwar effizientere Algorithmen, wie der Help-Karp-Algorithmus[5]

[4] Gurski et al. 2010, S. 18.
[5] Hutchinson et al. 2016, S. 14.

mit einer Laufzeit von $O(2^n n^2)$, allerdings skalieren auch diese exponentiell.
Das Ermitteln der optimalen Route ist also in der Praxis, wo teilweise mehr als 100 Knoten in einer Route verplant werden sollen, nicht wirklich denkbar, jedenfalls solange es keine relevanten Durchbrüche in Computertechnik oder Algorithmik gibt.

## 2.3 Heuristische Lösungsverfahren

Aufgrund der hohen Ressourcen, die das Ermitteln der optimalen Route kostet, werden in der Praxis häufig Heuristiken verwendet. Im Allgemeinen bezeichnet eine Heuristik ein Vorgehen zur Entscheidungsfindung, bei dem nicht alle gegebenen Informationen berücksichtigt werden. Ziel ist es also mit begrenzten Ressourcen, wie Zeit, Speicherplatz, etc., eine gute Entscheidung zu treffen.[6] Dies findet nicht nur in der Mathematik und in Computerwissenschaften Anwendung. Beispielsweise werden auch in der Medizin Entscheidungen, die schnell getroffenen werden müssen und über Leben und Tod eines Patienten entscheiden können, ohne Betrachtung aller existierenden Informationen getroffen.[7]
Eine Heuristik zur Lösung des TSP versucht, anders als ein Algorithmus zur Berechnung der optimalen Lösung, nur eine gute Route in kurzer Zeit zu berechnen. „Gute Lösung“ definiert sich dabei an der Abweichung vom Optimum.

Vertreter solcher Heuristiken sind beispielsweise der Nearest-Neighbor-Algorithmus und der Greedy-Algorithmus. Diese sind in der Lage innerhalb weniger Sekunden hunderte und sogar tausende Knoten in einer Route zu verplanen. Der Nachteil solcher Heuristiken ist allerdings, dass die Abweichung vom Optimum sehr hoch sein kann und mit steigender Anzahl von Knoten weiter zunimmt.[8]

[6] Gigerenzer und Todd 1999, S. 14f.
[7] Gigerenzer und Todd 1999, S. 3.
[8] Johnson et al. 2001, S. 22.

# 3 Verwendete Lösungsverfahren

## 3.1 Insert-First-Verfahren

### 3.1.1 Funktionsweise

Das Insert-First-Verfahren ist ein heuristischer Lösungsansatz des TSPs, bei dem das Betrachten der Knoten zum Aufbau eines Graphen in zufälliger Reihenfolge, bzw. in der Reihenfolge ihrer Erzeugung geschieht. Dabei wird zu einem Zeitpunkt genau ein Knoten betrachtet und an der für ihn bestmöglichen Stelle in den bereits bestehenden Graphen eingefügt.

**Algorithmus 1** Insert-First-Algorithmus

**Require:** Graph $G$, Pfad $P$
**Require:** $G = k_1, k_2, \ldots, k_n$, $n > 2$
1: $p_1 \leftarrow k_1$ ▷ Setzen der ersten beiden Knoten
2: $p_2 \leftarrow k_2$
3: **for** $a \leftarrow 3$, $a \leq n$, $a \leftarrow a + 1$ **do**
4: $j_S \leftarrow -1$ ▷ Index der geringsten Distanz
5: $d_S \leftarrow -1$ ▷ Geringste Distanz
6: **for** $b \leftarrow 1$, $b \leq m$, $b \leftarrow b + 1$ **do**
7: $d_C \leftarrow$ MERGEAT($P$, $b$, $k_a$) DISTANCE ▷ Gesamtdistanz, wenn $k_a$ am Index $b$ in $P$ eingefügt werden würde (siehe Alg. 7 auf Seite 34)
8: **if** $j_S = -1$ **or** $d_C < d_S$ **then**
9: $d_S \leftarrow d_C$ ▷ Kürzeste Distanz wird übernommen
10: $j_S \leftarrow b$ ▷ Und ihr Index
11: **end if**
12: **end for**
13: $P \leftarrow$ MERGEAT($P$, $j_S$, $k_a$) ▷ $k_a$ wird am Index $j_S$ in $P$ eingefügt
14: **end for**
15: **return** new Graph($P$) ▷ Graph mit Pfad $P$ wird zurückgegeben

Als Eingabe erhält der Algorithmus einen Graphen mit einer ungeordneten Liste von $n$ Knoten $k_1, k_2, \ldots, k_n$ mit $n > 2$. Jeder Algorithmus erzeugt einen Pfad $P$, in dem die Knoten des Graphs $G$ eingefügt und angeordnet werden. Nach vorheriger Definition kann jede gefüllte Position $p_k$ im Pfad $P$ mit einem Knoten $k_l$ des Graphs $G$

gleichgesetzt werden. Es gilt also $\forall p \in G$. Wenn beispielsweise:

$$G = k_1, k_2, k_3, k_4$$

und

$$P = p_1, p_2, p_3$$

mit

$$p_1 = k_1, p_2 = k_4, p_3 = k_2$$

dann

$$P = k_1, k_4, k_2$$

Nun wird $k_1$, der erste Knoten aus der übergebenen Liste, in den Pfad des Graphs an erster Stelle, $p_1$, eingefügt. Dies geschieht so oder ähnlich bei allen Verfahren, um einen statischen Ausgangspunkt zu gewährleisten und somit vergleichbare Ergebnisse zu erzielen. Anschließend wird noch der zweite Knoten, $k_2$ angehängt.

Das Vorgehen für das Einfügen der restlichen Knoten lässt sich wie folgt beschreiben: Sei $G$ ein Graph mit einer ungeordneten Menge von $n$ Knoten $k_1, k_2, \ldots, k_n$ und bereits teilweise befülltem Pfad $k_1, \ldots, k_m$ mit $m \geq 2$ und $m < n$. Die Knoten, die noch eingefügt werden müssen, werden in der Reihenfolge ihres Auftretens in der übergebenen Liste in den Graphen eingefügt, womit der als nächstes einzufügende Knoten immer $k_i$ mit $i = m + 1$ ist.
Um die beste Stelle zu ermitteln, in die $k_i$ eingefügt werden soll, wird für jeden möglichen Index, also jede mögliche Stelle, die Gesamtdistanz des entstehenden Graphen berechnet.

Das niedrigste Ergebnis dieser Möglichkeiten wird zusammen mit dem dazugehörigen Index $j$ vermerkt. Nachdem die niedrigste Distanz für $k_i$ errechnet wurde kann anhand des Index' der Knoten an der bestmöglichen Stelle in den Graphen eingefügt werden. Einfügen bedeutet hier, dass alle Knoten, deren Index gleich oder höher $j$ ist um einen Platz nach hinten verschoben werden. Nachdem alle Knoten auf diese Weise verschoben wurden, kann $k_i$ an der Stelle $j$ eingefügt werden, ohne, dass andere Knoten verloren gehen. Beispielhaft sähe das mit den vorher festgelegten Bezeichnungen wie folgt aus:

$$P = k_1, k_2, k_4, k_3 \text{ und } k_{i=5}$$

Durch das ermitteln der Gesamtdistanzen in Abhängigkeit zu den möglichen Einfügestellen wird bekannt, dass $k_{i=5}$ mit dem Index $j = 4$, also zwischen $k_4$ und $k_3$ bestmöglich eingefügt werden kann. Durch das Einfügen nach Algorithmus (Alg.) 7 auf Seite 34 entsteht der Pfad

$$P = k_1, k_2, k_4, k_5, k_3$$

für den Graphen.

### 3.1.2 Zeitkomplexität

Analysiert man den Insert-First-Algorithmus nach seiner Zeitkomplexität lassen sich, in Verbindung mit seinen Ergebnissen, die im Abschnitt 3.1.3 diskutiert werden, Aussagen über seine Effizienz treffen.
Die äußere Iteration beginnend in Zeile 3 des Algorithmus sei hier $I_a$ und die innere beginnend in Zeile 6 $I_b$. $I_a$ wird, bedingt durch die Abbruchbedingung $a \leq n$ und das Inkrementieren von $a$ um 1 nach jeder Iteration $n-2$ mal durchlaufen. Die Laufzeit von $I_b$ ist abhängig von $m$, der Menge der Elemente des Pfads $P$. Da in jedem Iterationsdurchlauf ein Element in $P$ eingefügt wird berechnet sich $m$ in einem Iterationsschritt mit $m = a - 1$. Für eine Funktion $f(n)$, welche die Komplexität des Algorithmus in der Form $f : \mathbb{N} \to \mathbb{N}$ abbildet, lässt sich also sagen

$$f(n) = \sum_{a=3}^{n} m_a$$

$$f(n) = \sum_{a=3}^{n} a - 1$$

$$f(n) = \frac{(n-1) \cdot (n-2)}{2} - 1$$

Womit sich ergibt, dass der Algorithmus eine quadratische Laufzeit der Form

$$f(n) = O(n^2)$$

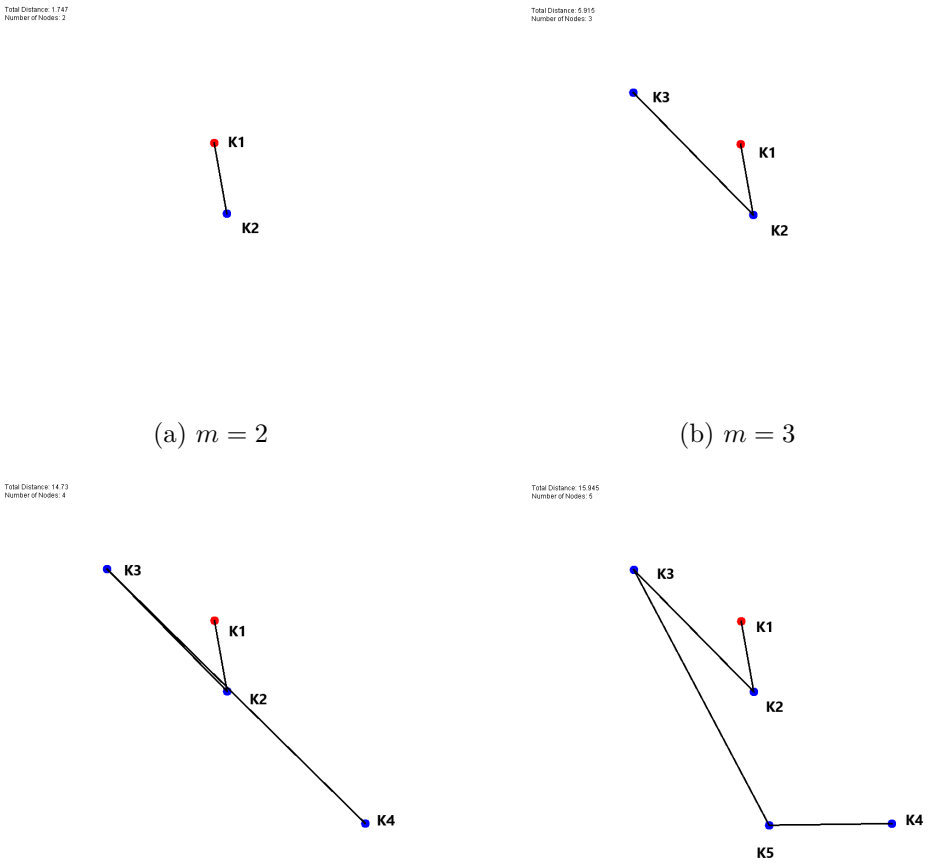
hat. Daraus lässt sich ablesen, dass die Laufzeit des Algorithmus zwar nicht exponentiell zur Eingabemenge, aber wenigstens noch quadratisch steigt. Besser wäre hier ein logarithmisches Verhalten, da Algorithmen dieser Art besser für große Eingabemengen Skalieren.[9]

### 3.1.3 Ergebnisse und Schwächen

Bevor einige durch den Algorithmus generierte Beispiele betrachtet werden, wird hier das Szenario dieser und aller folgender Beispiele, es sei denn ist anderes angegeben, beschrieben. Alle gezeigten Knoten befinden sich auf einem zweidimensionalem Fläche mit den Maßen zehn mal zehn Längeneinheiten (LE). Folglich kann jedem Knoten eine X- und Y-Koordinate zwischen jeweils null und zehn zugeordnet werden. Dementsprechend bewegen sich auch die Gesamtdistanzen er gezeigten Graphen in dieser Größenordnung. Weiterhin werden aus Gründen der Übersichtlichkeit für den Großteil der folgenden Beispiele nur Graphen mit fünf Knoten betrachtet.

[9] Gurski et al. 2010, S. 9ff.

Bei dem Einsatz des oben beschriebenen Algorithmus kommt es zu Ergebnissen, die in ihrer Qualität nah an die optimale Lösung herankommen, teilweise aber auch weit von ihr abweichen können.




(a) $m = 2$ (b) $m = 3$

(c) $m = 4$ (d) $m = 5$

Abbildung 3.1: Insert-First führt zu schlechtem Ergebnis

Auf 3.1c lässt sich erkennen, dass das Einfügen des vierten Knoten $k_4$ nicht optimal geschieht. Besser für $m = 4$ wäre hier der Pfad

$$P = k_1, k_3, k_2, k_4$$

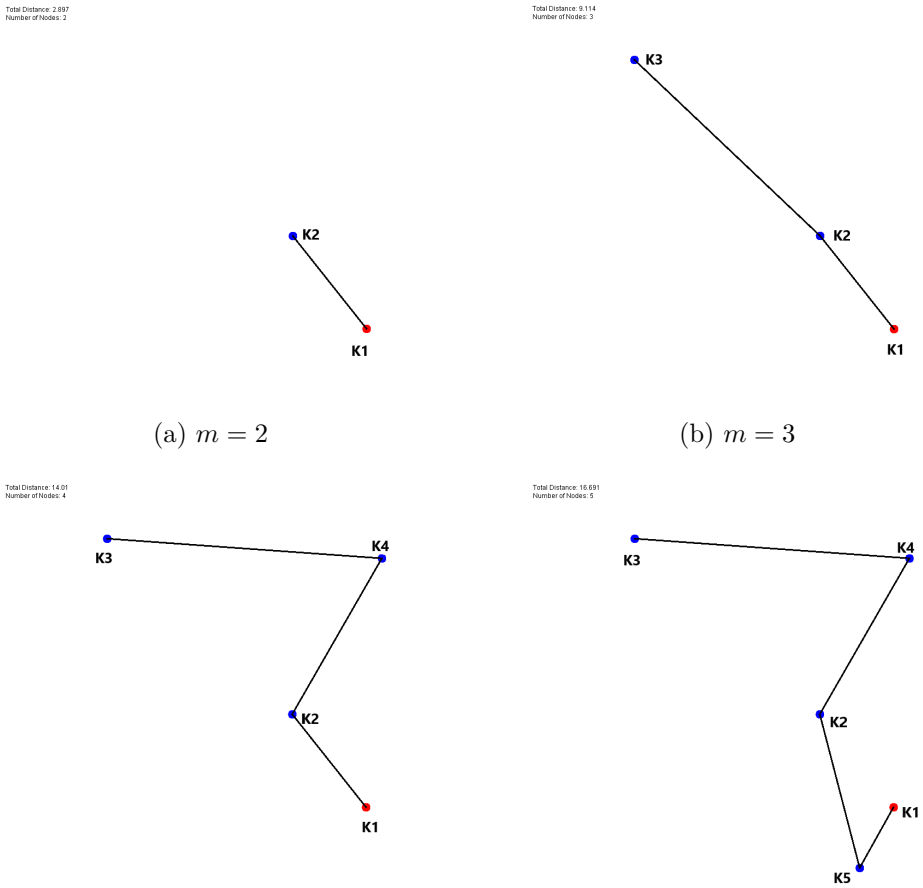
Dieser wird allerdings nicht durch das Insert-First-Verfahren gebildet, da dies eine Änderung des bereits erzeugten Graphen in 3.1b erfordern würde. Dies ist jedoch nicht möglich, da $k_4$ nur zwischen bereits im Pfad des Graphen vorhandenen Knoten eingefügt werden kann, sodass der schlussendlich generierte Graph eine Gesamtlänge

von 15,945 LE hat. Hier lässt sich auch das grundlegende Problem des Algorithmus erkennen: Das Erstellen einer Route ohne vorherige Betrachtung der Gesamtheit der Knoten. Einzelne Teilschritte des Graphen können gut erzeugt werden, wie beispielsweise im Schritt von 3.1a auf der vorherigen Seite zu 3.1b auf der vorherigen Seite. Andere hingegen, wie vorher erwähnt, nicht. Grund hierfür ist die alleinige Betrachtung des Knotens $k_i$. Spezifischer bedeutet das, dass das frühe Einfügen von Knoten in den Pfad eines Graphen später zu Komplikationen führen kann, da es objektiv besser gewesen wäre einen anderen Knoten früher einzufügen. Am konkreten Beispiel führt die generierte Reihenfolge von

$$P = k_1, k_2, k_3$$

in 3.1b auf der vorherigen Seite dazu, dass $k_4$ nur unter einen vergleichsweise großen Gesamtdistanzzuwachs in den Graphen eingefügt werden kann.

Konträr zu diesem schlechten Beispiel ist der Insert-First-Algorithmus auch in der Lage gute bis optimale Ergebnisse zu generieren.




(a) $m = 2$ (b) $m = 3$

(c) $m = 4$ (d) $m = 5$

Abbildung 3.2: Insert-First führt zu guten Ergebnis

Am Beispiel in 3.2 lässt sich erkennen, wie der Insert-First-Algorithmus einen optimalen Pfad mit den gegebenen Knoten generiert. Gerade im Schritt von 3.2c zu 3.2d ist ein funktionierendes und korrektes Einfügen des Knotens in den Graphen zu sehen, bei dem der Anstieg der Gesamtdistanz der Route sehr gering gehalten wird. Hier wird der aktuelle Knoten mit geringem Zuwachs der schlussendlichen Gesamtdistanz in den Graphen eingefügt, sodass die Gesamtdistanz zum Ende bei 16,691 LE liegt. Die scheint zwar höher als das vorherige schlechte beispiel, das liegt aber an den Positionen der einzelnen Knoten.

Für die Bewertung des Algorithmus müssen also beide Seiten betrachtet werden. Zwar ist Insert-First in der Lage eine gute oder auch optimale Route zu erstellen, allerdings beeinflusst die Reihenfolge der Betrachtung der Knoten stark die Qualität des Endergebnisses.

## 3.2 Insert-Furthest-Verfahren

### 3.2.1 Funktionsweise

Aufbauend auf den Erkenntnissen des Insert-First-Verfahrens können experimentell einige Verbesserungsideen abgeleitet und ihre Auswirkungen auf das Erzeugen eines Graphen betrachtet werden. Beim Insert-First-Verfahren wurde festgestellt, dass eine große Schwäche des Algorithmus die Reihenfolge der Betrachtung der Knoten sein kann. Ein möglicher Ansatz, dieser in 3.1.3 auf Seite 7 beschriebenen Schwäche entgegenzuwirken, ist die Einführung eines Kriteriums zur Betrachtung der Knoten. Eine mögliche Umsetzung eines solchen Kriteriums ist das Insert-Furthest-Verfahren. Hier wird der als nächstes einzufügende Knoten ($k_i$) durch seine Distanz zum Vorgänger ($k_{i-1}$) bestimmt.

Ähnlich dem Insert-First-Verfahren wird auch hier ein Graph mit einer Liste von Knoten und einem zu Beginn leerem Pfad erzeugt. Auch hier wird wieder der erste Knoten der Liste $k_1$ als initialer Knoten $p_1$ des Pfades $P$ gesetzt. Der nächste zu betrachtende Knoten ist nun aber nicht $k_2$, sondern wird durch die Distanz zu $k_1$ bestimmt. Ausgewählt wird der Knoten, der am weitesten von $k_1$, bzw. allgemein am weistesten von $k_{i-1}$, entfernt ist und nicht bereits Teil des Pfads ist. Dieser Knoten wird nun auf die gleiche Weise wie die Knoten beim Insert-First-Verfahren in den Pfad des Graphen eingefügt; die Stelle mit der geringsten Distanzerhöhung für den Graphen wird gesucht und $k_i$ an dieser Stelle nach dem im Alg. 7 auf Seite 34 beschriebenen Verfahren eingefügt.
Der Gedanke hinter der dieser Veränderung ist der Versuch Knoten mit größerer Voraussicht als im Insert-First-Verfahren in den Pfad des Graphen einzufügen. Ziel ist es mit den ersten paar Knoten einen Pfad zu generieren, der einen großen Teil der Fläche überspannt, auf der sich Knoten befinden. Das kann insofern zu einem besserem Ergebnis führen, dass die ersten Knoten zwar unter einer, relativ zur schlussendlichen Gesamtlänge des Graphen, hohen Distanzerhöhung eingefügt werden, die nachfolgenden Knoten aber durch geringe Umwege des bestehenden Pfads in den Graphen eingebunden werden können. Auf diese Weise sollen Komplikationen beim Einfügen der letzten Knoten verhindert werden und so suboptimale Graphen wie in 3.1 auf Seite 8 umgangen werden.

**Algorithmus 2** Insert-Furthest-Algorithmus

**Require:** Graph $G$, Pfad $P$
**Require:** $G = k_1, k_2, \ldots, k_n$, $n > 2$
1: $p_1 \leftarrow k_1$ ▷ Setzen des ersten Knoten
2: $i \leftarrow -1$
3: $d_i \leftarrow -1$
4: **for** $a \leftarrow 1$, $a \leq n$, $a \leftarrow a + 1$ **do**
5: $j_F \leftarrow -1$
6: $d_F \leftarrow -1$
7: **for** $b \leftarrow 2$, $b \leq n$, $b \leftarrow b + 1$ **do** ▷ Finde $k_b$ mit der höchsten Distanz zu $p_m$
8: $d_C \leftarrow$ DISTANCE($k_b$, $p_m$) ▷ Distanz zwischen $k_b$ und letztem Knoten $p_m$
9: **if** $k_b \notin P$ **and** ($j_F = -1$ (or) $d_C > d_F$) **then**
10: $d_F \leftarrow d_C$
11: $j_F \leftarrow b$
12: **end if**
13: **end for**
14: $i_S \leftarrow -1$
15: $d_S \leftarrow -1$
16: **for** $b \leftarrow 2$, $b < m$, $b \leftarrow b + 1$ **do**
17: $d_C \leftarrow$ MERGEAT($P$, $b$, $k_{j_F}$) DISTANCE
18: **if** $i_S = -1$ **or** $d_C < d_S$ **then**
19: $i_S \leftarrow b$
20: $d_S \leftarrow d_C$
21: **end if**
22: **end for**
23: $P \leftarrow$ MERGEAT($P$, $k_{j_F}$, $i_S$) ▷ Siehe Alg. 7 auf Seite 34
24: **end for**
25: **return** new Graph($P$)

### 3.2.2 Zeitkomplexität

Der Insert-Furthest-Algorithmus fügt verglichen mit dem Insert-First-Algorithmus ein Auswahlkriterium hinzu.
Dieses Kriterium drückt sich im Pseudocode im Alg. 2 durch eine zusätzliche Schleife in Zeile vier aus. Um nun die Zeitkomplexität zu ermitteln, reicht es die Laufzeit dieser Schleife, $n^2$, auf die in 3.1.2 auf Seite 7 berechnete zu addieren, wodurch sich

$$f(n) = \frac{n^2 + n}{2} + n^2 - 1$$

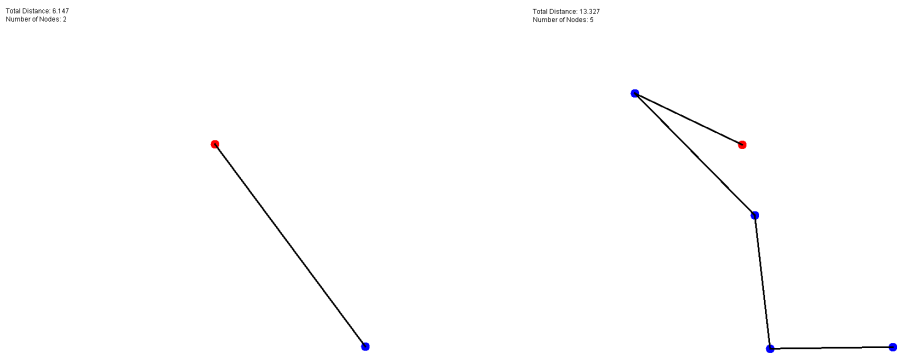
und dadurch auch hier eine Komplexität von

$$f(n) = O(n^2)$$

ergibt. Damit skaliert dieser Algorithmus bei sich verändernder Eingabe in etwa genauso wie Insert-First.

### 3.2.3 Ergebnis und Schwächen

Testet man des Insert-Furthest-Verfahren anhand der Knoten des Beispiels 3.1 auf Seite 8 wird sichtbar, dass der Algorithmus tatsächlich in der Lage ist einen besseren Pfad zu generieren als das Insert-First-Verfahren.



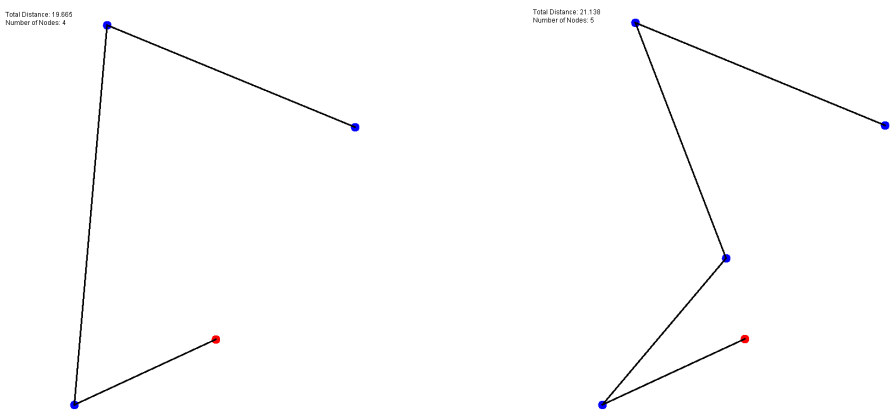

(a) $m = 2$ (b) $m = 5$

Abbildung 3.3: Insert-First führt zu schlechtem Ergebnis

In 3.4a auf der nächsten Seite ist zu erkennen, dass, anstatt wie in 3.1a auf Seite 8 $k_2$ als erster Knoten, $k_4$ eingefügt wird. Dieses Verhalten ist nach der in 3.2.1 auf Seite 11 definierten Funktionsweise zu erwarten, da $k_4$ der Knoten mit der größten Distanz zu $k_1$ ist und daher als erstes in den Pfad des Graphen eingefügt wird. Nach der Ausführung aller Schritte des Algorithmus erhält man den in 3.4b auf der nächsten Seite zu sehenden Graphen. Dieser hat eine Gesamtdistanz von 13,327 LE und ist somit im Vergleich mit dem in 3.1 auf Seite 8 durch das Insert-First-Verfahren erzeugten Graph 2,618 LE oder 16,41% kürzer.

Ebenso wie das Insert-First-Verfahren kann das Insert-Furthest-Verfahren auch Graphen generieren die in ihrer Gesamtdistanz vom Optimum abweichen. Am folgenden

Beispiel wird deutlich, dass auch dieser Algorithmus von ähnlichen Schwächen betroffen ist.




(a) $m = 4$ (b) $m = 5$

Abbildung 3.4: Insert-First führt zu schlechtem Ergebnis

Die der Abbildung 3.4 vorhergehenden Schritte $m = 1$ bis $m = 3$ sind im Anhang zu finden.

Auch hier lässt sich das Problem der Reihenfolge beobachten, welches in 3.1.3 auf Seite 7 beschrieben wurde. Durch das Einfügen des letzten Knotens $k_4$ in den Pfad entsteht ein suboptimaler Graph. In diesem konkreten Beispiel ist das mit dem menschlichen Auge jedoch nicht sofort ersichtlich. Vergleicht man aber die Distanzen zwischen den Knoten $k_4$ und $k_1$ (6,129 LE) mit den zwischen $k_1$ und $k_2$ (5,019 LE) wird schnell ersichtlich, dass eine Verminderung der Distanz durch das Umlegen der Knoten erreicht werden kann. Verursacht wird diese Abweichung vom Optimum dadurch, dass $k_4$ als letztes in den Graphen eingefügt wird, da der Knoten sich, relativ zu den restlichen Knoten, in der Mitte der Fläche befindet und somit aufgrund seiner geringeren Entfernung vom Algorithmus als betrachtet wird. Die optimale Route

$$P = k_1, k_3, k_4, k_2, k_1$$

würde es jedoch erfordern, dass $k_4$ früher betrachtet und in den Pfad eingefügt wird. Der durch den Algorithmus erzeugte Graph hat eine Gesamtlänge von 21,138 LE, während durch das Umlegen zum optimalen Graph eine Länge von 20,028 LE, also Reduktion der Distanz um 1,11 LE oder 5,251% erreicht werden kann.

Das Insert-Furthest-Verfahren verhält sich in einigen Fällen, wie in 3.3 auf der vorherigen Seite gezeigt, besser als das Insert-First-Verfahren, weist aber immer noch

eindeutige Schwächen, gerade im Bezug auf die Reihenfolge der Betrachtung der Knoten. Das Beispiel in 3.4 auf der vorherigen Seite zeigt deutlich, wie auch hier die Positionierung der Knoten Einfluss auf das Endergebnis hat.

## 3.3 Insert-Closest-Verfahren

### 3.3.1 Funktionsweise

Aufbauend auf den durch das Insert-First- und Insert-Furthest-Verfahren gewonnen Erkenntnissen ist es mögliche weitere Variationen der Heuristik zu entwickeln und deren Ergebnisse zu betrachten. Da sowohl in 3.1 auf Seite 5 als auch in 3.2 auf Seite 11 festgestellt wurde, dass ein Grund für suboptimal erstellte Routen die Reihenfolge der Betrachtung der Knoten ist, wird für das Insert-Closest-Verfahren eine weitere anderes Kriterium für eben diese Reihenfolge festgelegt. Wie der Name des Verfahrens schon suggeriert, geschieht hier die Auswahl der Knoten wieder nach ihrer Distanz.

Das Insert-Closest-Verfahren bezeichnet im Grundprinzip die Umkehrung des Insert-Furthest-Prinzips. Anstatt des am weitesten entfernten Knotens wird hier der dem aktuellen Knoten nächste betrachtet.
Zu Beginn beschreibt sich der Algorithmus identisch zum Insert-Furthest-Verfahren. Auch hier wird ein neuer Graph mit einem leerem Pfad `path` und einer Liste von Knoten $k_1, \ldots, k_n$ der Länge $n$ erzeugt. Auch hier wird der erste Knoten $k_1$ als erster Knoten des Pfades festgelegt. Der als nächstes einzufügende Knoten wird wie beim Insert-Furthest-Verfahren durch seine Distanz zum vorherigen bestimmt. Für das Insert-Closest-Verfahren wird der dem Vorherigen Knoten nächste Knoten, unter der Bedingung, dass dieser nicht bereits Teil des Graph ist, in den Pfad eingefügt. Um die beste Stelle zum Einfügen des Knotens zu ermitteln wird das gleiche Verfahren wie bei den beiden vorherigen Algorithmen angewandt; die Stelle, die den geringsten Anstieg für die Gesamtdistanz des Pfads wird ausgewählt. Auch das Vorgehen beim Einfügen orientiert sich hier an den vorherigen Algorithmen. Der aktuelle Knoten $k_i$ wird nach dem in 7 auf Seite 34 dargestelltem Prinzip an dem vorher festgelegten Index $j$ in den Graph eingefügt.
Der beschriebene Algorithmus befindet sich als Pseudocode in etwas ausführlicher Version unter Alg. 2 auf Seite 12.

### 3.3.2 Zeitkomplexität

Da es sich beim Insert-Closest-Algorithmus im eine Abwandlung des Insert-Furthest handelt, ist die Struktur beider Algorithmen gleich – nur eine Bedingung ändert sich.

**Algorithmus 3** Insert-Closest-Algorithmus

**Require:** Graph $G$, Pfad $P$
**Require:** $G = k_1, k_2, \ldots, k_n$, $n > 2$
1: $p_1 \leftarrow k_1$ ▷ Setzen des ersten Knoten
2: $i \leftarrow -1$
3: $d_i \leftarrow -1$
4: **for** $a \leftarrow 1$, $a \leq n$, $a \leftarrow a + 1$ **do**
5: $j_S \leftarrow -1$
6: $d_S \leftarrow -1$
7: **for** $b \leftarrow 2$, $b \leq n$, $b \leftarrow b + 1$ **do** ▷ Finde $k_b$ mit der geringsten Distanz zu $p_m$
8: $d_C \leftarrow$ DISTANCE($k_b$, $p_m$) ▷ Distanz zwischen $k_b$ und letztem Knoten $p_m$
9: **if** $k_b \notin P$ **and** ($j_S = -1$ (or) $d_C > d_S$) **then**
10: $d_S \leftarrow d_C$
11: $j_S \leftarrow b$
12: **end if**
13: **end for**
14: $i_S \leftarrow -1$
15: $d_S \leftarrow -1$
16: **for** $b \leftarrow 2$, $b < n$, $b \leftarrow b + 1$ **do**
17: $d_C \leftarrow$ MERGEAT($P$, $b$, $k_{j_F}$) DISTANCE
18: **if** $i_S = -1$ **or** $d_C < d_S$ **then**
19: $i_S \leftarrow b$
20: $d_S \leftarrow d_C$
21: **end if**
22: **end for**
23: $P \leftarrow$ MERGEAT($P$, $k_{j_S}$, $i_S$) ▷ Siehe Alg. 7 auf Seite 34
24: **end for**
25: **return** new Graph($P$)


(a) $m = 4$ (b) $m = 5$

Abbildung 3.5: Der Insert-Closest Algorithmus kommt zu einem schlechten Ergebnis

Daher kann auch für diesen Algorithmus eine Zeitkomplexität von

$$f(n) = O(n^2)$$

angenommen werden, womit auch dieser Algorithmus wie Insert-First skaliert.

### 3.3.3 Ergebnis und Schwächen

Auch die Ergebnisse dieses Algorithmus gestalten sich sehr divers. Wie bei den beiden anderen Verfahren entstehen hier Graphen, die nahe an die Optimale Route heranreichen oder ihr teilweise auch entsprechen, aber auch solche, die weit von der optimalen Lösung abweichen. Übergibt man dem Algorithmus beispielsweise die gleichen Knoten wie in Abbildung 3.1 auf Seite 8, einem Szenario, bei dem der Insert-First Algorithmus einen suboptimalen Graph generiert, kommt der Insert-Closest Algorithmus, so wie der Insert-Furthest Algorithmus auch, auf die optimale Route.
So wie die anderen Algorithmen stößt aber auch der Insert-Closest Algorithmus bei bestimmten Konstellationen von Knoten an seine Grenzen.

Wie in Abbildung 3.5 zu sehen ist, plant der Algorithmus einen Pfad, der deutlich erkennbar nicht optimal ist. Die einzelnen Schritte, die zur Generierung des Graphen führen finden sich im Anhang in der Abbildung A.1 auf Seite 33. Der in 3.5b gezeigt Pfad hat eine Gesamtdistanz von 15,336 LE. Durch das Ändern der Reihenfolge der Knoten zu

$$P = k_1, k_4, k_5, k_2, k_3$$

könnte eine Verringerung der Distanz im Vergleich zum vorherigen Resultat um 3,208 LE, bzw. 20,918% erreicht werden. Auch hier wird der suboptimal geplante Pfad durch die Reihenfolge der Betrachtung der Knoten verursacht. Im konkreten Beispiel werden die ersten vier Knoten $k_1, k_2, k_4$ und $k_5$ zu einem für sich optimalen Pfad zusammengefügt. $k_3$ wird aufgrund seiner hohen Distanz zu den übrigen Knoten als letztes in den Pfad eingefügt. Im gezeigten Beispiel kommt es also genau zu der Umkehrung des Problems, welches beim Insert-Furthest Algorithmus besteht; dort werden Knoten aufgrund ihrer zu niedrigen Distanz teilweise zu spät eingefügt. Diese Problem hat hier zur Folge, dass $k_3$ als letzter Knoten des Pfads nach dem von ihm am weitesten entfernten Knoten eingefügt wird, was zu einem hohen Zuwachs der Gesamtdistanz führt.

## 3.4 Zusammenfassung der Schwächen und Verbesserungsvorschläge

Die Ergebnisse, welche die Algorithmen erzeugen, können zum jetzigen Zeitpunkt nur begrenzt miteinander verglichen werden, da noch zu wenige Beispiele vorliegen. Allerdings lassen sich durch die wenigen Beispiele die vorliegen bereits einige Beobachtungen treffen, anhand deren die erzeugten Graphen nachträglich noch verbessert werden können. Betrachtet man nun diese Graphen, die durch alle drei Algorithmen erzeugt werden, so fallen bei allen schnell Schwächen auf, die ihr Ergebnis beeinträchtigen. Anhand der vorherigen Beispiele wurde deutlich, dass die Algorithmen zwar in der Lage sind gute Ergebnisse zu erzeugen, gleichzeitig aber auch auf sich allein gestellt nicht sehr zuverlässig sind. Dies trifft sowohl auf das Insert-First Verfahren, als auch auf die anderen vorgestellten Algorithmen zu.
Bezüglich des Verhältnis' von Komplexität und Nutzen lässt sich keine genaue Aussage treffen, welcher Algorithmus effizienter arbeitet. Alle drei Algorithmen weisen eine Komplexität von $O(n) = n^2$ und erzeugen gleichzeitig Ergebnisse, die in ihrer Qualität von optimal bis erkennbar verbesserbar reichen.
Weiterhin ist in Abbildung 3.5 auf der vorherigen Seite ist ein Phänomen zu beobachten, welches gerade bei Anwendung der beschriebenen Algorithmen mit mehr Knoten häufig auftritt. Das Entstehen von Überkreuzungen von Kanten zwischen zwei Knoten (siehe hierzu Abbildung A.1 auf Seite 34). Gerade bei solchen Überkreuzungen besteht immer die Möglichkeit den Graph so umzulegen, dass die Gesamtdistanz sinkt. Daher ist es sinnvoll einen Algorithmus zu entwickeln, der versucht Überkreuzungen aufzulösen.
Weiterhin fallen bei manchen Graphen Konstellationen von Knoten und bestehenden Kanten auf, bei denen durch ein einfaches Umlegen der Kanten, bzw. der Knotenreihenfolge, signifikante Verringerungen in der Distanz erzielt werden können. Beispiels-

weise erkennt man in der Veränderung von Abbildung A.-2h auf Seite 34 zu A.-3i auf Seite 34 ein solches Umlegen und die einhergehende Distanzverminderung. Auch hier kann ein Algorithmus, der gezielt nach solchen Konstellationen sucht und diese aufhebt, Abhilfe schaffen.

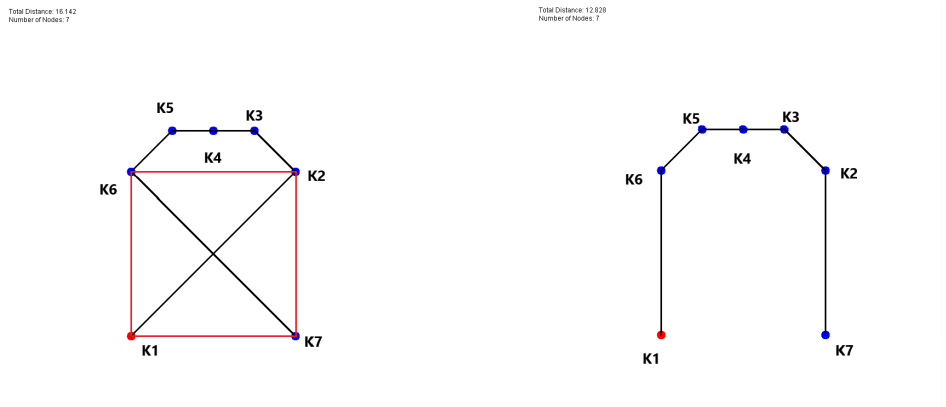
Nachdem diese Algorithmen im nächsten Kapitel implementiert werden folgt eine Analyse der Ergebnisse verschiedener Kombinationen der vorgestellten Algorithmen, basierend auf mehreren Testläufen mit zufällig erzeugten Knoten. Dadurch soll es möglich sein die Algorithmen untereinander und in Verbindung miteinander zu vergleichen.

# 4 Verbesserung eines bestehenden Pfads

## 4.1 Entfernen von Überschneidungen

### 4.1.1 Beschreibung des Problems

Wie in Abschnitt 3.4 auf Seite 18 angedeutet kommt es kommt es bei generierten Graphen zu Überkreuzungen von Teilrouten zwischen jeweils zwei Knoten. Betrachtet man solche Überkreuzungen im Detail fallen einige Gemeinsamkeiten zwischen ihnen auf. So ist es beispielsweise immer möglich eine Überkreuzung durch das Verändern der Reihenfolge der Knoten im Pfad aufzulösen und so eine Verringerung in der Gesamtdistanz zu erreichen.




(a) Graph mit einer Überkreuzung (b) Aufgelöste Überkreuzung

Abbildung 4.1: Graph mit und ohne Überkreuzung (Das rote Rechteck in Abbildung a) dient späteren Illustrationszwecken)

Anhand dieses Beispiels wird nun das Entstehen, Erkennen und Auflösen von Überkreuzungen erläutert.

Eine Überkreuzung repräsentiert das Auftreten eines Schnittpunkts von zwei Kanten eines Graphen in einem für den Graphen relevanten Bereich. Ein Schnittpunkt von zwei Kanten bedeutet hier, dass keine der vier Knoten gleich sein dürfen. Besteht eine Überkreuzung also aus den Knotenpaaren $A$ und $B$ mit $A = k_{A_1}, k_{A_2}$ und $B = k_{B_1}, k_{B_2}$,

dann muss gelten $k_{A_1} \neq k_{A_2} \neq k_{B_1} \neq k_{B_2}$. Ist diese Bedingung nicht erfüllt, kann es keine Überkreuzung geben. Sind nur drei der vier benötigten Knoten einzigartig kann es nicht zu einer Überkreuzung kommen, da dies einen zusammenhängenden Streckenabschnitt der Form $k_1, k_2, k_3$ darstellen würde.

### 4.1.2 Erkennnen von Überkreuzungen

Um Überkreuzungen erkennen zu können ist die Bedingung „in einem für den Graphen relevanten Bereich“ wichtig. Betrachtet man zwei zufällig ausgewählte Kanten als unendliche Linien, stellt man sie also in der zweidimensionalen Ebene als lineare Funktion, mit einer Steigung und einem Schnittpunkt mit der Ordinate, dar, dann schneiden sich alle diese Funktionen an irgendeinem Punkt, es sei denn sie sind parallel zueinander. Um zu überprüfen, ob eine Überkreuzung im für die Generierung eines Graphen relevanten Bereich ist, wird zuerst der Schnittpunkt der beiden Kanten berechnet. Dazu wird aus zwei Knoten einer Kante eine lineare Funktion der Form $f(x) = mx + n$ simuliert, wobei $m = \frac{\Delta y}{\Delta x}$ und $n = y - mx$. Sind die Knoten der beiden Kanten nun $A_1, A_2$ und $B_1, B_2$, dann ergibt sich für die Berechnung der Schnittstelle:

$$x_S = \frac{(y_{B_1} - \frac{y_{B_2} - y_{B_1}}{x_{B_2} - x_{B_1}} \cdot x_{B_1}) - (y_{A_1} - \frac{y_{A_2} - y_{A_1}}{x_{A_2} - x_{A_1}} \cdot x_{A_1})}{(\frac{y_{A_2} - y_{A_1}}{x_{A_2} - x_{A_1}}) - (\frac{y_{B_2} - y_{B_1}}{x_{B_2} - x_{B_1}})} \tag{4.1}$$

$$y_S = \frac{y_{A_2} - y_{A_1}}{x_{A_2} - x_{A_1}} \cdot x_S + (y_{A_1} - \frac{y_{A_2} - y_{A_1}}{x_{A_2} - x_{A_1}} \cdot x_{A_1}) \tag{4.2}$$

Mit $x_S$ und $y_S$ lässt sich der Punkt $S(x_S|y_S)$ konstruieren.

Mit Hilfe des Punkts $S$ gilt es nun zu überprüfen, ob sich dieser im relevanten Bereich befindet. Um dies zu bestimmen wird um die Knoten beider Kanten jeweils ein Recht simuliert, wie es beispielhaft in Abbildung 4.1a auf der vorherigen Seite eingezeichnet ist. Eine Überkreuzung ist genau dann für den Algorithmus relevant, wenn sie in den Rechtecken beider Kanten liegt. Um dies zu überprüfen wird folgender Algorithmus angewandt:

**Algorithmus 4** Erkennen von Überkreuzungen

**Require:** Knoten $A_1, A_2$, Punkt $P$

```
1: score ← 0
2: if x_{A_1} > x_{A_2} then
3:     if x_P > x_{A_2} and x_P < x_{A_1} then
4:         score ← score +1
5:     end if
6: else
7:     if x_P < x_{A_2} and x_P > x_{A_1} then
8:         score ← score +1
9:     end if
10: end if
11: if y_{A_1} > y_{A_2} then
12:     if y_P < y_{A_1} and y_P > y_{A_2} then
13:         score ← score +1
14:     end if
15: else
16:     if y_P > y_{A_1} and y_P < y_{A_2} then
17:         score ← score +1
18:     end if
19: end if
20: return (score == 2)  ▷ score == 2 gibt wahr zurück und signalisiert, dass P im
    Rechteck von A_1 und A_2 ist
```

An dieser Stelle sei angemerkt, dass das Problem der Überkreuzungserkennung auch mit Hilfe von Vektorenskalierung lösbar ist. Dieses Verfahren wird jedoch in dieser Arbeit nicht diskutiert.

### 4.1.3 Auflösen von Überkreuzungen

Um einen Algorithmus zur Auflösung von Überkreuzungen entwickeln zu können, ist es wichtig die Knoten um eine Überkreuzung herum vor und nach deren Auflösung zu betrachten. Dazu kann als Beispiel wieder Abbildung 4.1 auf Seite 20 dienen. Reihenfolge der Knoten in Abbildung 4.1a auf Seite 20 ist

$$P_{alt} = k_1, k_2, k_3, k_4, k_5, k_6, k_7$$

Die Reihenfolge der Knoten nach dem Auflösen in Abbildung 4.1b auf Seite 20 ist

$$P_{neu} = k_1, k_6, k_5, k_4, k_3, k_2, k_7$$

In diesem Beispiel seien die betroffenen Kanten $A$ und $B$ mit den Knoten $A_1 = k_1$, $A_2 = k_2$ und $B_1 = k_6$, $B_2 = k_7$. Die hier interessanten Knoten sind $k_2$ bis $k_6$, da sich deren Reihenfolge umkehrt. Daraus kann gefolgert werden, dass zum Auflösen einer Überkreuzung das Umkehren der betroffenen Knoten in der Mitte reicht. Diese betroffenen Knoten bestimmen sich durch die Kanten $A$ und $B$ – der erste umzukehrende Knoten ist immer $A_2$, während der letzte $B_1$ ist. Dies ist allerdings nur unter der Bedingung wahr, dass $A$ im Graph vor $B$ ist. Ist dies nicht der Fall kehren sich die Rollen der Knoten um und $B_1$ ist der erste, während $A_2$ der letzte umzukehrende Knoten ist. Ein Algorithmus, der auf einem Pfad mit Überkreuzung und bekannten $A$ und $B$ eben dieses Tauschen ausführt, findet sich im Anhang unter Algorithmus 8 auf Seite 35.

Ein vollständiger Algorithmus, der die aufgeführten Methodiken anwenden soll, um Überkreuzungen zu erkennen und aufzulösen, müssen also alle Kanten gegeneinander geprüft werden. Eine simple Umsetzung davon, die alle beschriebenen Methodiken mit einschließt kann sein

**Algorithmus 5** Erkennen und Auflösen von Überkreuzungen auf einem Pfad

**Require:** Pfad $P$
**Require:** $P = p_1, p_2 \cdots, p_n$, $n \geq 4$, $\forall p \in G$
1: **for** $a \leftarrow 2$, $a \leq n$, $a \leftarrow a + 1$ **do**
2: **for** $b \leftarrow b$, $b \leq n$, $b \leftarrow b + 1$ **do**
3: **if not** ($p_a \neq p_b$ **and** $p_a \neq p_{b-1}$ **and** $p_{a-1} \neq p_b$ **and** $p_{a-1} \neq p_{b-1}$ **then**
4: CONTINUE
5: **end if**
6: $x_S \leftarrow$ nach 4.1 auf Seite 21 ▷ $p_a$ und $p_{a-1}$ entsprechen $A_1$ und $A_2$
7: $y_S \leftarrow$ nach 4.2 auf Seite 21 ▷ $p_b$ und $p_{b-1}$ entsprechen $B_1$ und $B_2$
8: **if** CHECK$(p_a, p_{a-1}, S(x_S|y_S))$ **and** CHECK$(p_b, p_{b-1}, S(x_S|y_S))$ **then** ▷ CHECK repräsentiert dabei Alg. 4 auf der vorherigen Seite
9: $P \leftarrow$ RESOLVE$(P, p_a, p_{b-1})$ ▷ Auflösen der Überkreuzung mit Alg. 8 auf Seite 35
10: **end if**
11: **end for**
12: **end for**

Wird das Entfernen von Überkreuzungen nach diesem Prinzip implementiert, ergeben sich einige Randbedingungen, die es Wert sind erwähnt zu werden. Aufgrund der geschachtelten Iterationen über die Kanten des Graphs lässt sich eine Zeitkomplexität von

$$f(n) = O(n^2)$$

ermitteln, womit der Algorithmus im Rahmen der polynomialen Zeitkomplexitätsklasse liegt. Dies bedeutet, dass die Laufzeit des Algorithmus proportional zum Quadrat

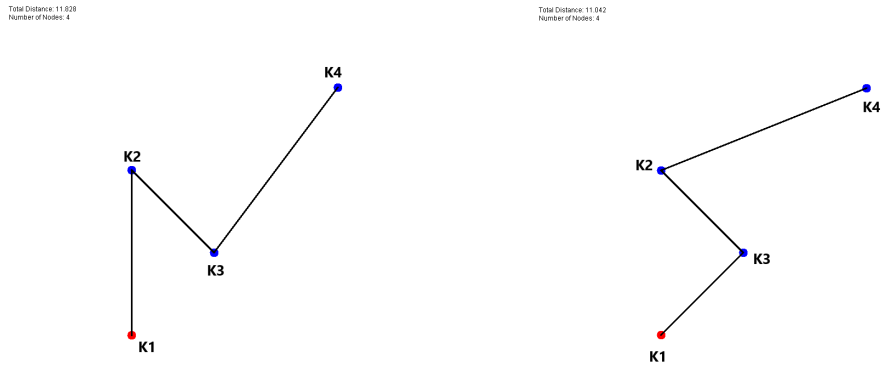
seiner Eingabemenge wächst. Als Eingabemenge können hier Knoten bzw. Kanten eines Graphen behandelt werden, wobei $n$ die Menge der Knoten repräsentiert.
Ein Algorithmus, der Überkreuzungen aus einem Graph entfernt, arbeitet also mit einer ähnlichen Laufzeit wie die Heuristiken, die den Graph vorher erzeugen.

Weiterhin ist es möglich, dass durch das Auflösen einer Überkreuzung eine weitere, neue entsteht. Falls dies so geschieht, dass in den restlichen Iterationen über die Kanten diese Überkreuzung nicht mehr erkannt wird, beispielsweise, wenn dies in der letzten Iteration passiert, dann wird die Überkreuzung nicht vom Algorithmus aufgelöst. Eine Möglichkeit dies zu umgehen ist durch das rekursive Aufrufen des Algorithmus, damit mehrmals auf Überkreuzungen überprüft wird. Allerdings besteht hier Bedarf eine solche Implementierung genauer zu untersuchen, was in dieser Arbeit nicht behandelt wird.

## 4.2 Nachbesserung eines Pfads

### 4.2.1 Darstellung des Grundproblems

Neben den im vorherigen Abschnitt beschriebenen Überkreuzungen kommt es bei den generierten Graphen auch zu solchen, die durch ein einfaches Umlegen der Route verbessert werden können.




(a) Verbesserungsfähiger Graph (b) Verbesserter Graph

Abbildung 4.2: Graph vor und nach der Nachbesserung

Das Beispiel in 4.2 zeigt einen Graphen mit einer initialen Knotenreihenfolge

$$P_{alt} = k_1, k_2, k_3, k_4$$

Durch das Umlegen zu

$$P_{neu} = k_1, k_3, k_2, k_4$$

erfolgt eine Verringerung der Gesamtdistanz. Betrachtet man die hier betroffenen Kanten $E_1 = (k_1, k_2)$, $E_2 = (k_1, k_3)$ und $E_3 = (k_2, k_3)$ kann die Distanzverringerung anhand ihrer einzelnen Distanzen und ihres Auftretens in den beiden Graphen erklärt werden. Da gilt

$$E_1, E_3 \in P_{alt} \text{ und } E_2, E_3 \in P_{neu}$$

kann eine Distanzverringerung mit

$$\omega(E_1) + \omega(E_3) < \omega(E_2) + \omega(E_3)$$

erklärt werden. Im konkreten Beispiel in 4.2 auf der vorherigen Seite drückt sich das durch eine Verringerung der Gesamtdistanz um 0,786LE aus.

### 4.2.2 Algorithmus zur Nachbesserung

Ein Algorithmus, der Nachbesserung auf genau diese Art vornehmen soll, kann wie folgt vorgehen. Betrachte beginnend mit dem zweiten Knoten in einem Pfad jeden Knoten. Für jeden zu betrachteten Knoten wird jede Kante, an dessen Ende der aktuelle Knoten nicht steht, betrachtet. Es gilt zu überprüfen, ob es für die Gesamtdistanz besser ist, wenn der aktuelle Knoten in die aktuelle Kante eingefügt wird. Anders ausgedrückt: $p_i$ mit $i \geq 2, i \in \mathbb{N}$ sei ein Knoten in einem vollständigen Pfad mit $n$ Knoten und $n-1$ Kanten. Zusammen mit $p_i$ wird auch immer eine Kante $e_j = (p_{k-1}, p_k)$ betrachtet, sodass gilt $p_i \notin e_j$. Um nun zu überprüfen, ob es möglich ist eine Verbesserung des Graphs vorzunehmen wird überprüft ob

$$\omega(p_{i-1}, p_i) + \omega(p_i, p_{i+1}) + \omega(e_j) > \omega(p_{i-1}, p_{i+1}) + \omega(p_{k-1}, p_i) + \omega(p_i, p_k)$$

Ist dies der Fall wird die Reihenfolge der Knoten entsprechend geändert und $p_i$ zwischen $p_{i-1}$ und $p_{i+1}$ entfernt und zwischen $p_{k-1}$ und $p_k$ eingefügt. Wie genau das Einfügen in die Kante funktioniert kann in Algorithmus 11 auf Seite 35 nachvollzogen werden.

Eine vollständige Implementierung eines Algorithmus, der Verbesserung an einem bestehenden Graph vornimmt kann wie in Algorithmus 6 auf der nächsten Seite aussehen. Damit beschreibt sich die Zeitkomplexität dieses Algorithmus mit

$$f(n) = O(n^2)$$

Womit er, wie der Algorithmus 5 auf Seite 23 Entfernen von Überkreuzungen, quadratisch zur Eingabemenge skaliert.

**Algorithmus 6** Nachbesserung eines Pfads

**Require:** Pfad $P$
**Require:** $P = p_1, p_2, \cdots, p_n, n > 3, n \in \mathbb{N}$
1: **for** $a \leftarrow 2$, $a \leq n-1$, $a \leftarrow a+1$ **do**
2: **for** $b \leftarrow 3$, $b \leq n$, $b \leftarrow b+1$ **do**
3: **if** $a = b$ **or** $a = b-1$ **then**
4: CONTINUE
5: **end if**
6: **if** $\omega(p_{a-1}, p_a) + \omega(p_a, p_{a+1}) + \omega(p_{b-1}, p_b) > \omega(p_{a-1}, p_{a+1}) + \omega(p_{b-1}, p_a) + \omega(p_a, p_b)$ **then**
7: $P \leftarrow$ RESOLVE$(P, (p_{b-1}, p_b), p_a)$
8: **end if**
9: **end for**
10: **end for**
11: **return** $P$

## 4.3 Ergebnisse und Bewertung der verschiedenen Algorithmen

Nachdem nun drei Algorithmen zur Generierung einer Route und zwei zu deren Nachbesserung vorgestellt wurden, gilt es nun Erkenntnisse über die Ergebnisse dieser Algorithmen im Vergleich zu den anderen zu gewinnen. Dafür wurde eine Testszenario angelegt, in dem für verschiedene Kombinationen dieser Algorithmen mehrmals getestet werden. Jeder der drei Algorithmen zur Routengenerierung wird für sich, mit Auflösen von Überkreuzungen, mit Nachbesserung und mit Auflösen von Überkreuzungen und Nachbesserung getestet, wodurch es zwölf verschiedene Kombinationen gibt. Diese Kombinationen werden 1 000 mal mit je $5, 10, 15, \cdots, 100$ Knoten getestet und die Ergebnisse (Gesamtdistanz des finalen Graphs) vermerkt, wodurch 19 000 Datensätze entstehen.
Tabelle 4.1 auf der nächsten Seite zeigt die möglichen Kombinationen der vorgestellten Algorithmen, wie häufig die Kombination die beste Lösung (im Vergleich mit den anderen Kombinationen) generiert hat und die relative Häufigkeit der Kombination an den besten Ergebnissen. Eindeutig kann hier erkannt werden, dass – unabhängig von der Strategie – Routen, die nachgebessert wurden nach Algorithmus 6, den größten Teil der besten Lösung beanspruchen. Insgesamt erzielte eine Kombination, die nachträglich den Nachbesserungs-Algorithmus anwendet 13 083 von 19 000 (68,86%) besten Ergebnissen. Den größten Teil macht dabei die Kombination, die den Insert-First-Algorithmus verwendet aus. Danach folgen ebenfalls nur mit Nachbesserung der Insert-Furthest-Algorithmus mit 4 228 und der Insert-Closest-Algorithmus mit 3648 besten Ergebnissen. Insgesamt verwenden 11,69% der besten Lösungen nur einen Algorith-

| **Strategie** | **Überkreuzungen** | **Nachbesserung** | **# Beste** | **% Beste** |
|---|---|---|---|---|
| First | Nein | Nein | 1 024 | 5,39% |
| First | Nein | Ja | 5 207 | 27,41% |
| First | Ja | Nein | 51 | 0,27% |
| First | Ja | Ja | 1 237 | 6,51% |
| Furthest | Nein | Nein | 481 | 2,53% |
| Furthest | Nein | Ja | 4 228 | 22,25% |
| Furthest | Ja | Nein | 44 | 0,23% |
| Furthest | Ja | Ja | 1 449 | 7,63% |
| Closest | Nein | Nein | 717 | 3,77% |
| Closest | Nein | Ja | 3 648 | 19,20% |
| Closest | Ja | Nein | 41 | 0,22% |
| Closest | Ja | Ja | 873 | 4,59% |

Tabelle 4.1: Beste Testergebnisse nach verwendeten Algorithmen

mus zu Generierung des Graphen, 0,72% lösen zusätzlich Überkreuzungen auf, 68,86% verwenden den Algorithmus 6 auf der vorherigen Seite zur Nachbesserung und 18,73% beide dieser Verfahren. Von den 19 000 besten Ergebnissen entfallen 7 519 (39,57%) an den eine Kombination mit dem Insert-First-Algorithmus, 5 279 (27,78%) an dne Insert-Closest-Algorithmus und 6 202 (32,64%) an den Insert-Furthest-Algorithmus.

Als statistisch beste Kombination kann also der Insert-First-Algorithmus in Verbindung mit der in 4.2 auf Seite 24 beschriebenen Nachbesserung angesehen werden.

# 5 Zusammenfassung und Ausblick

# Literaturverzeichnis


Applegate, David L. et al. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton: Princeton University Press. ISBN: 978-0-691-12993-8.

Domschke, Wolfgang et al. (2015). „Graphentheorie“. In: *Einführung in Operations Research*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 71–86. ISBN: 978-3-662-48216-2. DOI: 10.1007/978-3-662-48216-2_3. URL: https://doi.org/10.1007/978-3-662-48216-2_3.

Gigerenzer, Gerd und Peter M. Todd (1999). *Simple heuristics that make us smart*. Evolution and cognition. New York und Oxford: Oxford University Press. ISBN: 0-19-512156-2.

Gurski, Frank et al. (2010). *Exakte Algorithmen für schwere Graphenprobleme*. Berlin und Heidelberg: Springer. ISBN: 978-3-642-04499-1. DOI: 10.1007/978-3-642-04500-4.

Hutchinson, Charles et al. (Dez. 2016). *CMU Traveling Salesman Problem*. URL: https://www.math.cmu.edu/~af1p/Teaching/OR2/Projects/P58/OR2_Paper.pdf.

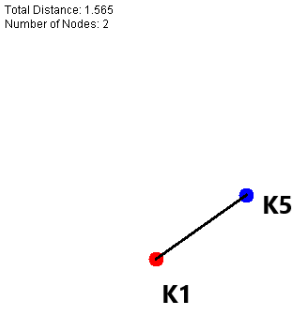
Johnson, David et al. (Dez. 2001). „Experimental Analysis of Heuristics for the ATSP“. In: 12. DOI: 10.1007/0-306-48213-4_9.

Macgregor, James und Thomas Ormerod (Juni 1996). „Human performance on the traveling salesman problem“. In: *Perception & psychophysics* 58, S. 527–539. DOI: 10.3758/BF03213088.

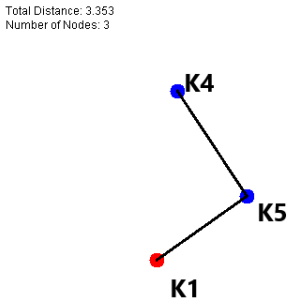
# A Anhang

## A.1 Bilder

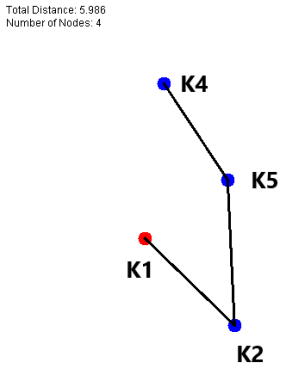

(a) $m = 2$


Total Distance: 1.565
Number of Nodes: 2
K5
K1

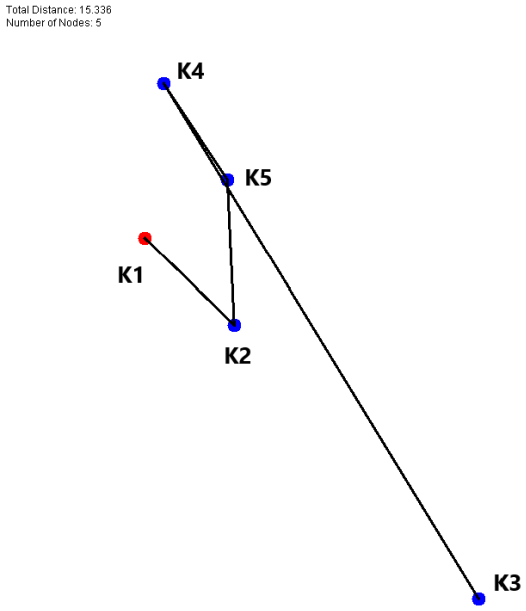
(b) $m = 3$


Total Distance: 3.353
Number of Nodes: 3
K4
K5
K1

(c) $m = 4$

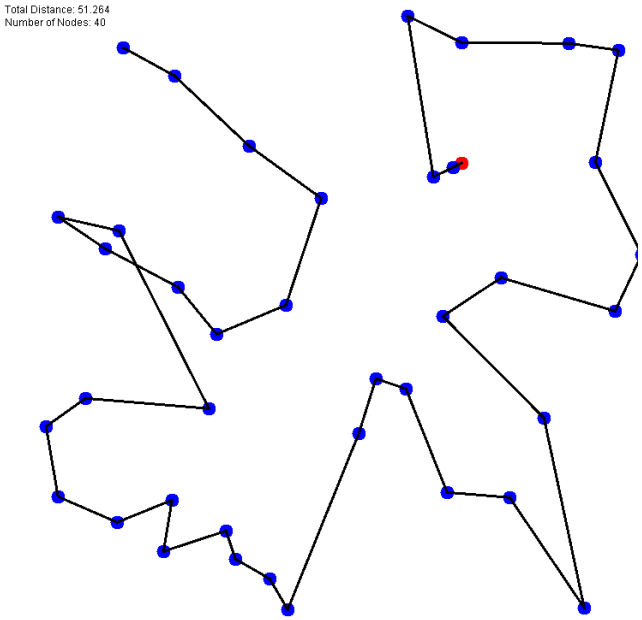



(d) $m = 5$

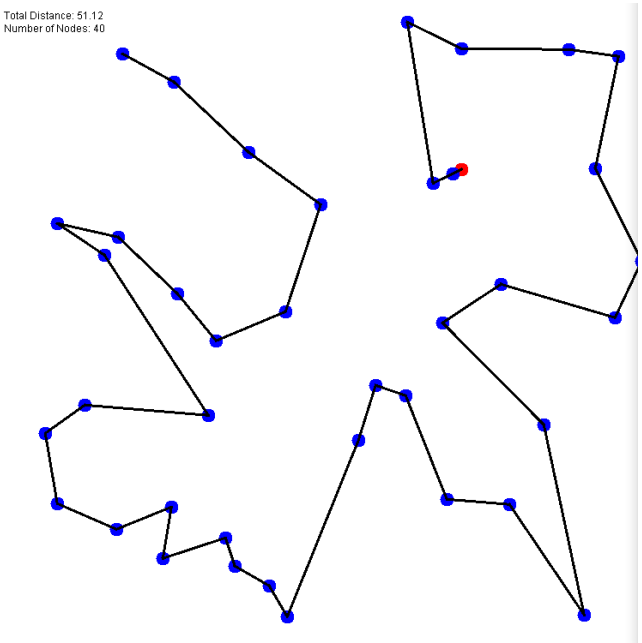



(e) $m = 5$

Abbildung A.-3: Viele Bilder

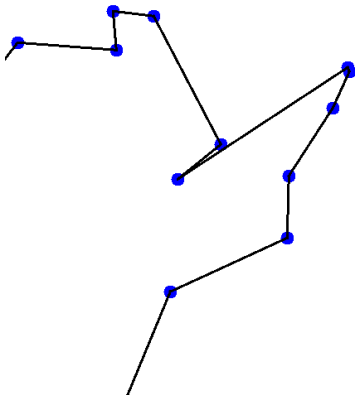



(f) Pfad mit einem Crossover

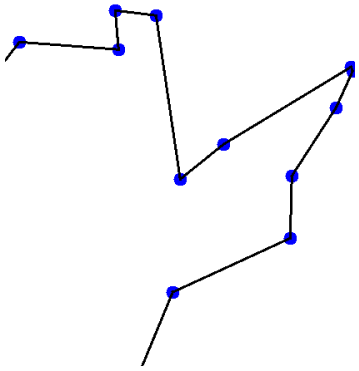



(g) Pfad mit aufgelöstem Crossover

Abbildung A.-3: Pfad aus 40 Knoten mit und ohne Crossover



(h) Teilpfad vor Nachbesserung



(i) Teilpfad nach Nachbesserung

Abbildung A.-3: Beispiel für eine Nachbesserung, Teilpfad

## A.2 Algorithmen

**Algorithmus 7** Einfügen eines neuen Knoten in einen Pfad

**Require:** Pfad $P = p_1, \ldots, p_n$ mit $\forall p \in G$ ▷ Jedes p ist Knoten in Graph $G$
**Require:** Knoten $K^*$, Index $i$, $i \leq n + 1$ ▷ neuer Knoten $K^*$, einzufügen an Index $i$

1: **for** $a \leftarrow n$, $a \geq i$, $a \leftarrow a + 1$ **do**
2: $p_{a+1} \leftarrow p_a$
3: **end for**
4: $p_i \leftarrow K^*$
5: **return** $P$

**Algorithmus 8** Tauschen von Knoten auf einem Graph zwischen zwei eingegebenen Knoten

**Require:** Graph $G$, Knotenpaare $A_1, A_2$ und $B_1, B_2$
**Require:** $G = k_1, k_2, \cdots, k_n$, $n > 4$
1: $i_{A_2} \leftarrow$ INDEX$(A_2)$
2: $i_{B_1} \leftarrow$ INDEX$(B_1)$
3: **if** $i_{A_2} > i_{B_1}$ **then**
4: SWAP$(i_{A_2}, i_{B_1})$ ▷ SWAP weißt beiden Parametern den Wert des anderen zu
5: **end if**
6: **while** $i_{A_2} < i_{B_1}$ **do**
7: SWAP$(k_{i_{A_2}}, k_{i_{B_1}})$
8: **end while**

**Algorithmus 9** Berechnung der Distanz zwischen zwei Knoten

**Require:** Knoten $A$, Knoten $B$
1: $d \leftarrow \sqrt{|x_A - x_B|^2 + |y_A - y_B|^2}$
2: **return** $d$

**Algorithmus 10** Berechnung der Gesamtdistanz eines Pfads

**Require:** Pfad $P$
**Require:** $P = p_1, \cdots, p_n$, $n \geq 2$
1: sum $\leftarrow 0$
2: **for** $a \leftarrow 2$, $a \leq n$, $a \leftarrow a + 1$ **do**
3: sum $\leftarrow sum +$ DISTANCE$(p_{a-1}, p_a)$
4: **end for**
5: **return** sum

**Algorithmus 11** Einfügen eines Knotens in eine Kante

**Require:** Pfad $P$, Kante $e_j$ und Knoten $p_i$
**Require:** $P = p_1, p_2, \cdots, p_n, e_j = (p_{k-1}, p_k), n > 3, k > 1n, i, j, k \in \mathbb{N}$
1: **if** i < k **then**
2: **for** $a \leftarrow i + 1, a < k, a \leftarrow a + 1$ **do**
3: SWAP$(p_i, p_a)$
4: **end for**
5: **else**
6: **for** $a \leftarrow i, a > k - 1, a \leftarrow a - 1$ **do**
7: SWAP$(p_i, p_a)$
8: **end for**
9: **end if**
10: **return** $P$

# Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema: *Variation, Analyse und Verbesserung eines Algorithmus zur heuristischen Lösung des Travelling Salesman Problems* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Benno Grimm