Bachelorarbeit

Visualisierung verschiedener Heuristiken für TSP

Niels Janson

Institut für Theoretische Informatik
Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier

10. Juni 2020

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 10. Juni 2020

_____

Niels Janson

Inhaltsverzeichnis

1 Einleitung

Was ist der kürzeste Weg, um einen Supermarkt zu betreten, alle Artikel der Einkaufsliste einzusammeln, zur Kasse zu gehen und ihn wieder zu verlassen?

Wahrscheinlich haben die Meisten beim regelmäßigen Einkaufen eine ungefähre Vorstellung, welche intuitive Route sie möglichst effizient durch den Supermarkt leitet ohne sich weitere Gedanken zu machen, ob es eventuell noch etwas besser wäre nicht jetzt, sondern erst auf dem Weg zur Kasse in dieses oder jenes Regal zu gehen.

Hinter dieser, im ersten Moment nicht sehr anspruchsvoll und relevant wirkenden Überlegung verbirgt sich das Rundreiseproblem, auch Problem des Handlungsreisenden genannt, aus der theoretischen Informatik. Tatsächlich ist es für den Menschen sehr schnell unmöglich, die optimale Route zu finden. Doch auch unter Zuhilfenahme leistungsfähiger Computer würde die Berechnungsdauer bei einer zu langen Einkaufliste, mehr Zeit benötigen, als der eigentliche Einkauf. Würden zum Beispiel gerade einmal 20 Produkte auf der Einkaufsliste stehen, so resultierte dies schon in über 2 Trillionen möglichen Routen durch den Markt.

Auch fernab des Supermarktes findet dieses Problem viel Beachtung. Beispielsweise in automatisierten Logistikzentren, bei denen es wichtig ist, dass Roboter ihre Pfade möglichst optimal planen, bei Ridesharing Dienstleistern um Kunden schneller befördern zu können oder auch bei der Entwicklung integrierter Schaltkreise, bei der Pfade von Leiterbahnen mit tausenden Knoten zu optimieren sind, um kompakte, schnelle und energiesparende Schaltungen zu realisieren.

Spätestes bei Letzteren wird offensichtlich, dass die Berechnung des einen kürzesten Pfades jeglichen Berechnungskapazitäten sprengen würde. Doch wie auch beim Planen des Einkaufs ist es für gewöhnlich gar nicht so entscheidend, ob der wirklich kürzeste Pfad ermittelt wird. Vielmehr reicht es eine einigermaßen gute Route zur Hand zu haben, bei der man beispielsweise nicht nach dem ersten Produkt aus der Gemüseabteilung zur Tiefkühlabteilung geht, danach wieder zurück zum Gemüse und so weiter. Für solche Ansätze gibt es heuristische Näherungsverfahren, die entweder eine Route ermitteln, die um einen gewissen Faktor vom Optimum abweicht, oder bestehende Pfade weiter optimieren.

In dieser Arbeit soll dabei das Augenmerk auf drei Verfahren der Eröffnungsheuristiken gelegt werden. Als simpelste die Nearest-Neighbour-Heuristik, sowie weiter die Nearest-Insertion-Heuristik und zuletzt die Greedy-Heuristik. Dabei soll ein schon bestehendes Programm zur Visualisierung um diese Verfahren erweitert werden, um praktisch sehen und verstehen zu können, wie diese arbeiten und eventuell zu erkennen welche Vor- und Nachteile sich ergeben. Weiter sollen noch die Möglichkeiten geschaffen werden, bestehende Problemstellungen einlesen und die Ergebnisse der Verfahren speichern zu können.

2 Grundlagen

In diesem Kapitel werden die nötigen Definitionen und Grundlagen der in dieser Arbeit verwendeten Begriffe erläutert.

2.1 Distanzen

Die Distanz oder auch der Abstand zwischen zwei Punkten a und b in einem n-dimensionalen Raum kann, je nach Anforderung, unterschiedlich definiert werden.

Die *euklidische Distanz* ist die intuitivste Form der Distanz, sie beschreibt im Grunde die Luftlinie zwischen zwei Punkten. Sie ist wie folgt definiert:

$$d(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

Die *Manhatten Distanz* ergibt sich, wenn Bewegungen nur parallel zu den Achsen des Koordinatensystems möglich sind. Definiert ist sie als

$$d(a,b) = \sum_{i=1}^{n}|a_i - b_i|$$

Bei der *Chebyshev Distanz* sind Bewegungen ebenfalls nur parallel zu den Achsen möglich, es wird aber nur das Maximum in die jeweiligen Richtungen als Distanz betrachtet.

$$d(a,b) = \max(|a_1 - b_1|, ..., |a_n - b_n|)$$

2.2 Graphen

Graphen dienen häufig der Repräsentation zusammenhängender Strukturen. Ein Graph G besteht aus einer nichtleeren endlichen Knotenmenge V und einer Kantenmenge E

und ist definiert als

$$G = (V, E) \text{ mit } E \subseteq \{\{a, b\} \mid a, b \in V, a \neq b\}$$

Die Anzahl der Knoten wird im Folgenden als n bezeichnet.

Ein solcher Graph ist *schleifenfrei*, da $a \neq b$. Weiter ist E selbst eine Menge, somit kann es keine zwei gleichen Kanten geben. Ein solcher Graph wird *einfach* genannt. Außerdem gilt für jede Kante $\{a, b\} = \{b, a\}$, sie ist also *ungerichtet*.

Eine wichtige Eigenschaft von Knoten ist ihr *Grad*. Dieser bezeichnet die Anzahl der mit einem Knoten a aus der Knotenmenge verbundenen weiteren Knoten und ist wie folgt definiert:

$$\deg(a) = |\{b \mid \{a, b\} \in E\}|$$

Für das Problem des Handlungsreisenden betrachten wir nur vollständige Graphen. Ein Graph heißt genau dann *vollständig*, wenn für alle Knoten $u, v \in V$ gilt: $u \neq v \Rightarrow \{u, v\} \in E$. Jeder Knoten ist also mit jedem anderen durch eine Kante verbunden.

Daraus folgt weiter, dass die Anzahl der Kanten in einem vollständigen Graphen $\binom{n}{2}$ beträgt und für jeden Knoten $v \in V$ gilt $\deg(v) = n - 1$.

Repräsentiert werden kann ein Graph mit den Knoten $V = \{v_1, ..., v_n\}$ auch als $(n \times n)$-Matrix. Diese wird *Adjazenzmatrix* von G genannt und ist definiert als

$$a_{i,j} := \begin{cases} 1, & \text{falls } \{v_i, v_j\} \in E \\ 0, & \text{sonst} \end{cases}$$

Den Kanten eines Graphen können zusätzlich auch noch Gewichte zugeordnet werden. Hierzu wird die Definition des Graphen um eine $(n \times n)$-Matrix D wie folgt erweitert:

$$G = (V, E, D) \text{ mit } D = (d_{i,j})$$

wobei $d_{i,j}$ das Gewicht der Kante $a_{i,j}$ in der Adjazenzmatrix repräsentiert. In dieser Arbeit wird D dafür genutzt den Abstand zweier Knoten darzustellen, somit sind die Einträge von $D \in \mathbb{Q}^+$.

Ein *Weg* ist eine Folge $u_1, ..., u_s$ von Knoten mit $\{u_i, u_{i+1}\} \in K$ für $i = 1, ..., s - 1$. Sind alle Knoten des Weges voneinander verschieden, so wird dieser auch *Pfad* genannt. Sind der erste und letzte Knoten eines Weges gleich und sind alle Knoten, bis auf den ersten und letzten und die Kanten des Weges paarweise verschieden, so spricht man

von einem *Kreis* oder auch *Rundweg*. Da der Start- und Endknoten eines Rundweges doppelt im Weg vorkommen, kann ein Rundweg kein Pfad sein. Haben die Kanten des Weges ein Gewicht, so ist die Länge des Weges definiert als die Summe der Gewichte eben dieser Kanten.

2.3 Das Problem des Handlungsreisenden

Das Problem des Handlungsreisenden besteht darin, einen möglichst kurzen Rundweg durch eine bestimmte Anzahl an Städten zu finden. Hierbei soll jede Stadt genau einmal besucht werden und am Ende des Weges will der Handlungsreisende wieder am Anfang angekommen sein. Die Anzahl der möglichen Rundwege ergibt sich daraus, auf wie viele Möglichkeiten man die Stände anordnen kann. Im Falle von n Städten sind das $n!$ verschiedene Rundwege.

Formal lässt sich das Problem des Handlungsreisenden, englisch Traveling Salesman Problem, kurz *TSP*, wie folgt definieren:

$$TSP = \left\{ \langle (d_{i,j})_{1 \leq i,j \leq n}, B \rangle \;\middle|\; \begin{array}{l} n \in \mathbb{N}, d_{i,j} \in \mathbb{Q}^+ \text{ und es gibt ein } \pi \in S_n \\ \text{mit } \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)} \leq B \end{array} \right\}$$

Die Eingabe $(d_{i,j})$ ist dabei die Gewichtsmatrix D, wobei die darin enthaltenen Gewichte den Abstand der Knoten zueinander repräsentieren. B ist eine Gesamtlänge, die es, sofern möglich, zu erreichen oder zu unterschreiten gilt. Der Weg selbst wird durch π, einen Eintrag der symmetrischen Gruppe S_n, dargestellt, wobei die Einträge in π den durchnummerierten Knoten entsprechen. Zu erwähnen ist noch, dass die Interpretation der Einträge von π somit keinen Rundweg darstellt, da der Start- beziehungsweise Endknoten nicht zweimal vorkommen können. Für das TSP selbst wird dies mit der Addition des Kantengewichts eben dieser Kante zu der Summe berücksichtigt.

2.4 Komplexität des TSP

Wie aus der formalen Definition hervorgeht ist die Lösung eines TSPs mit n Knoten, sofern sie existiert, ein Eintrag aus der symmetrischen Gruppe S_n. Ungünstigerweise hat eine solche Gruppe aber $n!$ Einträge, es ist also offensichtlich, dass das Überprüfen aller Einträge bei steigendem n sehr schnell keine Option darstellt. Tatsächlich ist aktuell kein Algorithmus bekannt, der dieses Problem in vertretbarem zeitlichem Aufwand

löst, wahrscheinlich gibt es auch keinen. Was hingegen aber möglich ist, ist mit einem gegebenen Rundweg zu ermitteln, ob dieser kürzer als B ist. Daraus folgt, dass das TSP NP-vollständig ist. Für genauere Ausführungen hierzu sei auf "Ist P = NP? Einführung in die Theorie der NP-Vollständigkeit" von Steffen Reith und Heribert Vollmer [4, Seite 12] verwiesen.

2.5 Heuristiken

Wie erwähnt sind keine Verfahren bekannt um den kürzesten Weg für das Problem des Handlungsreisenden mit vertretbarem Aufwand zu finden. Somit sind für die Praxis andere Verfahren notwendig um zumindest möglichst kurze Routen zu finden, man behilft sich mit Heuristiken. Diese lassen sich in zwei Gruppen aufteilen:

Eröffnungsheuristiken erzeugen aus einem gegeben Graphen einen Rundweg. Die Länge dieses Weges kann, je nach Heuristik, beliebig stark vom kürzesten Rundweg abweichen, oder auch nur um einen gewissen Faktor.

Darauf aufbauend gibt es verbessernde Heuristiken, die versuchen gegebene Rundwege weiter zu verkürzen. Auch hier kann die Güte der Ergebnisse beliebig stark vom kürzesten Weg abweichen.

3 Verwendete Heuristiken

Diese Arbeit befasst sich mit Eröffnungsheuristiken. Zu Grunde liegen also beliebige ungerichtete vollständige Graphen. Zu jeder Heuristik wird zusätzlich auf interessante Aspekte der praktischen Umsetzung eingegangen, die aus der abstrakten Beschreibung nicht sofort folgen. Als Datentypen finden hierbei 2-dimensionale Arrays Verwendung, durch die Gewichtsmatrizen D eines gewichteten Graphen repräsentiert werden. Die Einträge in diesem Array ergeben sich aus den in Kapitel 2.1 beschriebenen Distanzfunktionen. Sie sind also als Distanz oder Abstand zwischen den Knoten zu verstehen. Weiter werden auch Listen, vornehmlich als Darstellung von Pfaden und Rundwegen, genutzt, und 1-dimensionale Arrays um sonstige Eigenschaften der Knoten und Kanten zu speichern. Es kann davon ausgegangen werden, dass die jeweiligen Schreib- und Lesezugriffe auf diese Strukturen mit aktuellen CPUs in konstanter Laufzeit geschehen.

Anmerkung: Diese Annahme gilt nur solange die jeweils maximale Anzahl der Elemente des Graphen innerhalb der Wortbreite der CPU liegt. Eine der Heuristiken arbeitet mit allen Kanten, die im Graphen enthalten sind, bei n Knoten sind das also $\binom{n}{2} = \frac{n(n-1)}{2}$. Ausgehend von einem Graphen mit 2^{64} Kanten, sollte die Laufzeitbetrachtung der Umsetzungen bis zu einem Graphen mit 6.074.000.999 Knoten zutreffen.

3.1 Nearest-Neighbour-Heuristik

Die Nearest-Neighbour-Heuristik ist einer der einfachsten Ansätze einen Rundweg in einem Graphen zu ermitteln.

Der Gedanke hinter dieser Heuristik ist, vom aktuellen Knoten aus immer zum nächstgelegenen zu gehen.

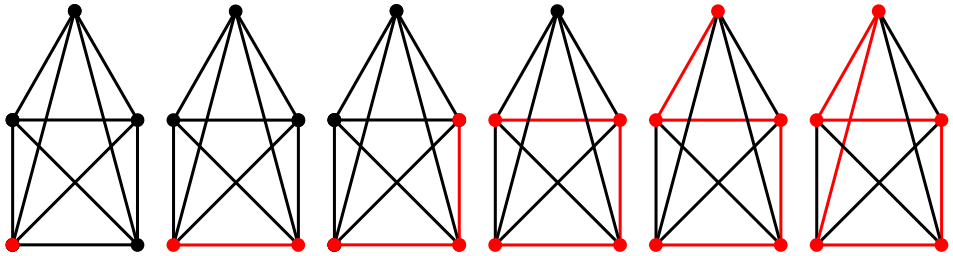
Der abstrakte Algorithmus dazu umfasst drei Schritte:

1. Wähle einen (beliebigen) Knoten im Graphen als Startknoten aus und füge ihn zum Pfad hinzu.

2. Finde einen Knoten, der dem zuletzt ausgewählten am nächsten und noch nicht im Pfad enthalten ist. Füge ihn zum Pfad hinzu. Wiederhole dies solange, bis alle Knoten im Pfad enthalten sind.

3. Füge den Startknoten hinzu um einen Rundweg zu erzeugen.

Aufgrund der "Kurzsichtigkeit" des Verfahrens kann der resultierende Rundweg wesentlich länger sein als der kürzeste. Dies ist auch in Abbildung 3.1 zu erkennen, wobei 3.1a Schritt 1 darstellt, 3.1b - 3.1e Schritt 2 und 3.1f Schritt 3.



(a) (b) (c) (d) (e) (f)

Abbildung 3.1: Beispiel Nearest-Neighbour-Heuristik

Der Algorithmus terminiert in $\mathcal{O}(n^2)$ [1, Seite 1].

3.1.1 Umsetzung

Im Folgenden wird die Implementierung der Nearest-Neighbour-Heuristik erläutert.

Sie basiert auf einer Kopie der Gewichtsmatrix des Graphen als 2-dimensionales Array. Anfangs wird ein beliebiger Knoten A gewählt, in einer Liste, die den Pfad repräsentiert gespeichert und die komplette A-te Spalte der Matrix mit Null beschrieben.

Nun wird in Zeile A nach dem Minimum gesucht, wobei Einträge mit dem Wert Null ignoriert werden. Dies geschieht einerseits, weil der Graph schleifenfrei ist und somit auf der Hauptdiagonialen ausschließlich Nullen stehen. Andererseits wird diese Eigenschaft aber auch dazu verwendet, Knoten in der Matrix für weitere Schritte zu ignorieren. Beim Suchen nach dem Minimum in einer Zeile kann, da anfangs ja die A-te Spalte mit Null überschrieben wurde, die Stelle A in der Zeile nie die Stelle des Minimums sein.

Angenommen also wir starten in Zeile A, in der das Minimum m in Spalte B vorkommt, so könnte diese wie folgt aussehen:

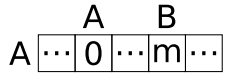


Abbildung 3.2: Nearest Neighbour A nach B

Knoten B ist A somit am nächsten, er wird also zur Pfadliste hinzugefügt und daraufhin die gesamte Spalte B der Matrix mit Null beschrieben. Nun wird in Zeile B nach dem Minimum gesucht. Ist in Zeile B das Minimum m nun an der Stelle C so könnte diese wie in Abbildung 3.3 aussehen:

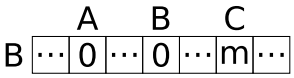


Abbildung 3.3: Nearest Neighbour B nach C

Der Knoten C wird zum Pfad hinzugefügt und die Spalte C in der Matrix wieder mit Nullen beschrieben. Dieser Vorgang wiederholt sich insgesamt $n - 1$ Mal.

Letztendlich wird der Pfad noch geschlossen, indem der Ausgangsknoten zum Pfad hinzugefügt wird.

Laufzeit

Die Laufzeitbetrachtung der Umsetzung dieser Heuristik ist recht einfach. Das Suchen eines Minimums in einer Zeile in der Matrix liegt offenkundig in $\mathcal{O}(n)$, das Schreiben einer Spalte ebenso. Diese Schritte werden $n - 1$ Mal ausgeführt, die Komplexität der Implementierung liegt also in $\mathcal{O}(n^2)$.

3.2 Nearest-Insertion-Heuristik

Die Idee der Nearest-Insertion-Heuristik ist inkrementell einen Rundweg mit einem möglichst nah gelegenen freien Knoten zu erweitern. Dieser wird dann so in den Rundweg eingefügt, dass er möglichst wenig an Länge zunimmt.

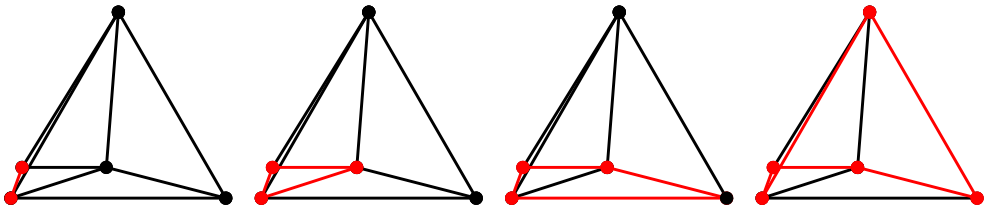
Genauer sieht das Verfahren wie folgt aus:

1. Wähle die kürzeste Kante im Graphen und füge dessen Knoten zum Pfad hinzu.
2. Suche einen Knoten außerhalb des Pfades mit der geringsten Entfernung zu den Knoten in dem Pfad.

3. Finde das Knotenpaar im Weg, bei dem beim Einfügen des Knotens der Weg minimal länger wird. Füge den neuen Knoten zwischen diesen beiden Knoten in den Rundweg ein.

4. Wiederhole die Schritte 2 und 3 bis alle Knoten im Rundweg sind.

Abbildung 3.4 zeigt einen exemplarischen Ablauf dieser Heuristik. Es is zu erkennen, dass auch diese Heuristik nicht zwingend den kürzesten Rundweg findet. Abbildung 3.4a entspricht dabei Schritt 1, 3.4b - 3.4d für sich jeweils den Schritten 2 und 3.



(a) (b) (c) (d)

Abbildung 3.4: Beispiel Nearest-Insertion-Heuristik

Der Rundweg, den dieser Algorithmus erzeugt, ist maximal zweimal so lang, wie der optimale, sofern die Knoten in einem metrischen Raum liegen. Der Algorithmus terminiert in $\mathcal{O}(n^2)$.[5, Seite 573]

3.2.1 Umsetzung

Wie die Nearest-Insertion-Heuristik in quadratischer Laufzeit zu implementieren ist, ist nicht so offensichtlich, wie zum Beispiel bei der Nearest-Neighbour-Heuristik. Aus Schritt 4 geht hervor, dass Schritt 2 und 3 $n-2$ Mal durchlaufen werden. Um also eine quadratische Gesamtlaufzeit zu erreichen, müssen diese in linearer Zeit abzuarbeiten sein.

Ermitteln des nächstgelegenen Knotens außerhalb des Rundwegs

Der naive Ansatz wäre, alle Kanten zwischen den Knoten innerhalb und außerhalb des Rundweges nach der kürzesten zu durchsuchen. Dieses Vorgehen würde aber einen quadratischen Aufwand mit sich bringen und das für jeden Knoten, der hinzugefügt werden würde. Somit würde diese Heuristik in $\mathcal{O}(n^3)$ terminieren.

Um also den nächstgelegenen Knoten zu allen sich im Pfad befindlichen Knoten zu finden, bedarf es ein paar Hilfsstrukturen und einer iterativen Vorgehensweise. Zum

einen wird in einer Liste `unusedNodes` festgehalten, welche Knoten noch nicht im Pfad enthalten sind. Zum anderen wird ein Array `nearestUsedNode` mit n Elementen genutzt um jedem Knoten den nächstgelegenen Pfadknoten zuzuweisen. Dabei repräsentieren die Indizes von `nearestUsedNode` die durchnummerierten Knoten. In den Einträgen von `nearestUsedNode` steht dann der Pfadknoten, der dem jeweiligen Indexknoten am nächsten ist.

Nachdem also der erste Knoten A als einer der Knoten der kürzesten Kante festgelegt ist, wird `unusedNodes` mit den restlichen Knoten B_1 bis B_{n-1} initialisiert. `nearestUsedNode` wird an den Stellen B_1 bis B_{n-1} mit A beschrieben, da dies der bisher einzige ausgewählte und somit nächste Knoten ist.

Veranschaulicht wird dies in Abbildung 3.5. Die grauen Pfeile zeigen dabei vom unausgewählten Knoten auf den nächstgelegenen ausgewählten. Die Werte in `nearestUsedNode` an den Stellen der Knoten innerhalb des Pfades sind irrelevant, und werden mit * dargestellt.

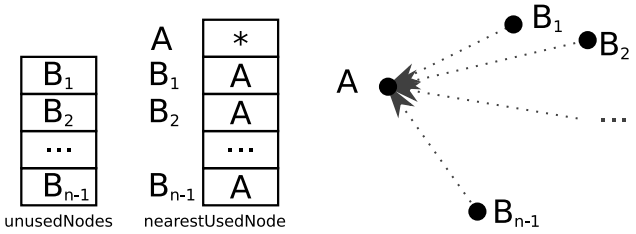


Abbildung 3.5: Nearest Insertion initialisiert

Mit dieser Struktur kann nun in linearer Laufzeit der Knoten mit der geringsten Distanz zu allen Konten im Rundweg gefunden werden. Hierfür werden die Knotenpaare aus den Einträgen in `unusedNodes` mit den entsprechenden Einträgen aus `nearestUsedNode` gebildet und die Distanz dieses Paares aus der Gewichtsmatrix ermittelt. Aus dem Minimum dieser Distanzen geht somit der nächstgelegene Knoten hervor.

Angenommen, der Knoten B_1 hat die geringste Distanz zu A, so ist dieser der, der daraufhin zum Pfad hinzugefügt wird. Er wird separat gespeichert und aus `unusedNodes` gelöscht. Der Eintrag an Stelle B_1 in `nearestUsedNode` verliert somit seine Bedeutung. Die Datenstrukturen sähen nun wie in Abbildung 3.6 aus.

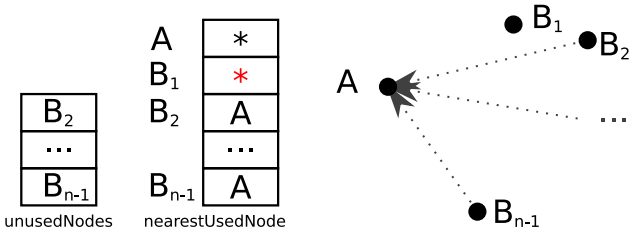


Abbildung 3.6: Nearest Insertion Knoten auswählen

Daraufhin wird für alle verbleibenden Knoten in `unusedNodes` nach dem gleichen Verfahren wie vorher ermittelt, ob deren Distanz zum neuen Knoten geringer ist, als die zum bisherigen Eintrag. Falls dem so ist, wird der neue Knoten an der entsprechenden Stelle in `nearestUsedNode` geschrieben. Wäre also die Distanz von B_1 nach B_2 geringer als von A nach B_2, so würden sich die Strukturen wie folgt ändern:

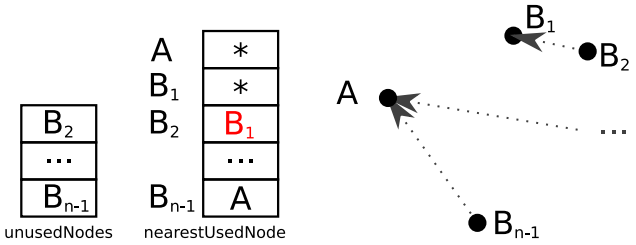


Abbildung 3.7: Nearest Insertion aktualisiert

Somit ist nach dem Auswählen eines neuen Knotens zum Einfügen in den Rundweg wieder für jeden Knoten außerhalb bekannt, welcher im Rundweg in kürzester Distanz zum ihnen liegt. Die Zugriffe auf die Strukturen benötigen eine konstante Laufzeit, bei maximal n Elementen ist also von linearer Laufzeit auszugehen.

Einfügen des Knotens in den Rundweg

Ist der Knoten mit dem geringsten Abstand zu einem Knoten im Rundweg gefunden, so muss dieser noch eingefügt werden. Hierfür wird für jedes Knotenpaar im Rundweg geprüft, um welchen Betrag sich die Länge des Rundwegs ändern würde, wenn der neue Knoten zwischen diesen beiden eingefügt werden würde. Es wird angenommen, der neue Knoten wäre A und das zu prüfende Knotenpaar wäre (B_i, C_i), wobei i von 1 bis zur aktuellen Anzahl der Knoten im Rundweg geht. Beim Einfügen dieses Knotens würde sich der Rundweg also um $d_{A,B} + d_{A,C} - d_{B,C}$ verlängern. Zwischen das Knotenpaar, bei dem dieser Betrag minimal ist, wird der neue Knoten eingefügt.

Abbildung 3.8 zeigt dieses Vorgehen. Die Kante, die das jeweilige Knotenpaar verbindet, wird dabei rot, die von den bisherigen Knoten zum neuen grün dargestellt.

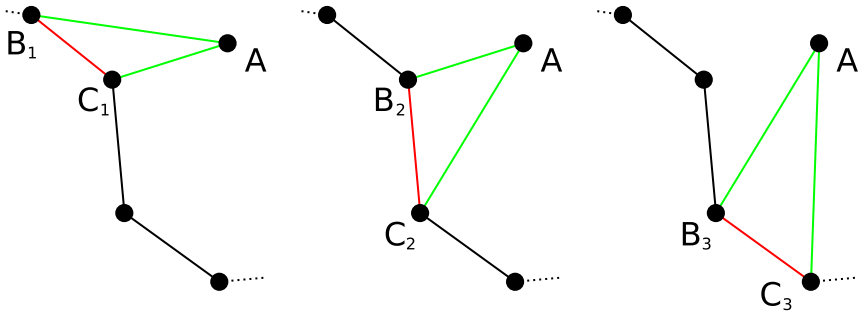


Abbildung 3.8: Nearest Insertion: Kante ersetzen

Laufzeit

Wie Eingangs erwähnt muss das Auswählen und Einfügen eines neuen Knotens ungefähr $n-1$ Mal wiederholt werden. Es wurde gezeigt, dass dies jeweils in linearer Laufzeit zu n möglich ist, somit ist insgesamt eine quadratische Laufzeit erreicht worden.

3.3 Greedy-Heuristik

Die Idee hinter dieser Heuristik ist immer möglichst kurze Kanten für den finalen Rundweg auszuwählen. Hierfür werden die Kanten der Länge nach sortiert und nach und nach einer Menge hinzugefügt. Zuvor wird noch überprüft, ob der Grad der Knoten dieser Kanten zwei beträgt, oder sich durch Hinzufügen der Kante ein Kreis mit weniger als n Knoten in der Kantenmenge bilden würde. In diesem Fall wird die Kante nicht hinzugefügt. Der genauere Ablauf sieht folgendermaßen aus:

1. Sortiere alle Kanten der Länge nach.
2. Wähle die kürzeste Kante aus, durch dessen Hinzufügen kein Rundweg oder ein Knoten mit Grad größer zwei entsteht.
3. Wiederhole Schritt 2 solange, bis n Kanten in dem Pfad enthalten sind.

Abbildung 3.9 zeigt einen exemplarischen Ablauf dieser Heuristik. Auch hier ist zu erkennen, dass der erzeugte Rundweg nicht optimal ist.

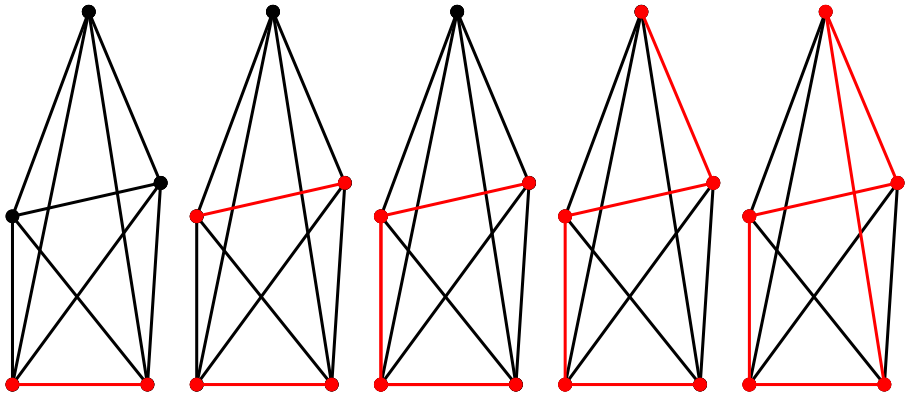


Abbildung 3.9: Beispiel Greedy-Heuristik

Diese Heuristik arbeitet in $\mathcal{O}(n^2 log(n))$.

3.3.1 Umsetzung

In dieser Heuristik wird der Rundweg nicht basierend auf Knoten, sondern auf Kanten ermittelt. Diese Kanten werden schrittweise einer Menge hinzugefügt, die im Folgenden als Pfadmenge bezeichnet wird. Auch bei der Greedy-Heuristik sind einige Kniffe erforderlich, um die erwähnte Laufzeit zu erreichen. Es werden zwar nur n Kanten in die Pfadmenge eingefügt, es müssen aber im Worst Case für alle Kanten die Einfügekriterien überprüft werden. Da somit bis zu $\frac{n(n-1)}{2}$ Kanten zu prüfen sind, muss diese Prüfung in $\mathcal{O}(log(n))$ liegen. Tatsächlich ist dies sogar in konstanter Laufzeit möglich, wie im Folgenden gezeigt wird.

Ermitteln von Kreisen in konstanter Laufzeit

Sind zwei Knoten, jeweils mit einem Grad von 1, durch eine oder mehrere Kanten verbunden, so bilden diese Kanten ein Wegsegment. Knoten, die nur in einer Kante in der Pfadmenge vorkommen, werden Endknoten genannt.

Anfangs wird ein einfaches Array `othernode` angelegt, dass so viele Elemente hat, wie der Graph Knoten. Die Indizes dieses Arrays repräsentieren wieder die durchnummerierten Knoten. An den Stellen der Endknoten sind in `othernode` die jeweils durch das Wegsegment verbundenen anderen Endkonten gespeichert. Im einfachsten Fall ist dieses Wegsegment lediglich eine einzelne Kante. Verbindet diese Kante zum Beispiel die Knoten 1 und 3, so steht an der Stelle 1 im Array die 3 und an der Stelle 3 die 1.

Alle anderen Einträge sind -1, das bedeutet, dass der Knoten kein Endknoten eines Wegsegments ist.

Zuerst wird nun geprüft, ob das Hinzufügen einer neuen Kante einen Knotengrad von mehr als zwei erzeugen würde. Das führt dazu, dass bei der Prüfung, ob ein Kreis entstehen würde, sichergestellt ist, dass die Knoten der potentiellen Kanten nur Endknoten sein können oder noch in keiner Kante der Pfadmenge vorkommen.

Dadurch ergeben sich für eine neue Kante, die die Knoten A und B verbindet, vier mögliche Fälle:

Die neue Kante grenzt an kein bisheriges Wegsegment: Dieser Fall wird dadurch erkannt, dass im Array sowohl an den Stellen A, als auch B eine -1 steht. Ist dies der Fall, so müssen lediglich neue Einträge im Array angelegt werden. An der Stelle A wird also B eingetragen und andersherum. Die Verweise, die sich somit ergeben, werden in Abbildung 3.10 als graue Pfeile dargestellt.

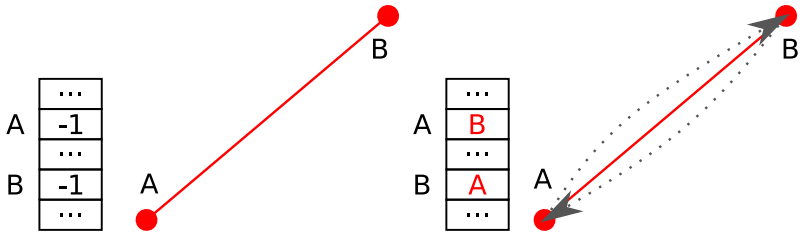


Abbildung 3.10: Neue Kante hinzufügen

Die neue Kante grenzt an ein bisheriges Wegsegment: Dieser Fall wird dadurch erkannt, dass entweder an Stelle A oder B ein Eintrag ungleich -1 steht. Das Wegsegment wird also um die neue Kante erweitert. Angenommen, das Segment führt von Knoten B über beliebig viele weitere zum Knoten C. Dann wird an die Stelle A in `othernode` der neu zugeordnete Endknoten C geschrieben und anders herum. An Stelle B wird -1 eingetragen, weil dieser Knoten nun kein Endkonten mehr ist.

In Abbildung 3.11 wird die neue Kante rot und das bisherige Wegsegment schwarz dargestellt. Die grauen Pfeile stellen die Zuordnungen der Endknoten der Wegsegmente in `othernode` dar.

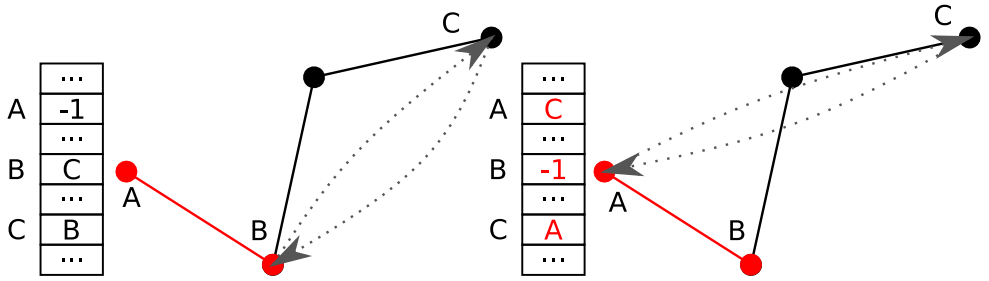


Abbildung 3.11: Wegsegment erweitern

Die neue Kante grenzt an zwei bisherige Wegsegmente: Erkannt wird dieser Fall dadurch, dass sowohl an der Stelle A als auch B Einträge ungleich -1 stehen und an Stelle A nicht B steht. Die neue Kante verbindet also zwei unterschiedliche Wegsegmente miteinander. Diese Segmente führen im Beispiel von Knoten C zu A und von B zu D. An den Stellen A und B wird nun -1 eingetragen, da diese Knoten keine Endknoten mehr sind. An der Stelle C wird D eingetragen und anders herum.

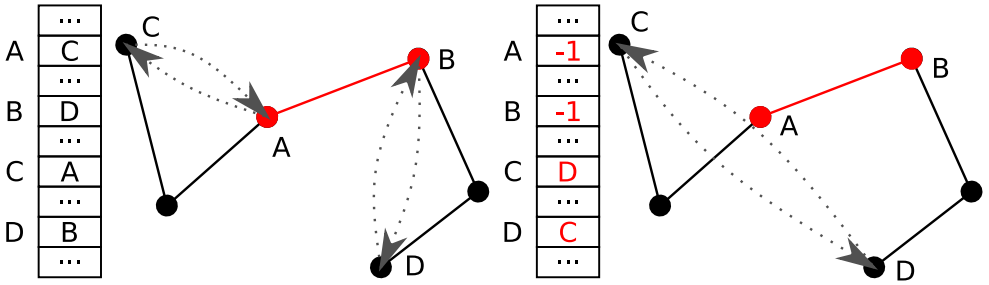


Abbildung 3.12: Wegsegmente verbinden

Die neue Kante schließt ein Wegsegment: Wenn B an der Stelle A steht wird erkannt, dass die neue Kante ein Wegsegment schließen würde. Somit darf diese Kante, sofern es nicht die letzte ist, nicht übernommen werden. Abbildung 3.13 verdeutlicht diese Situation noch einmal, im Wegsegment zwischen A und B liegen noch beliebig viele Knoten C_1 bis C_x.

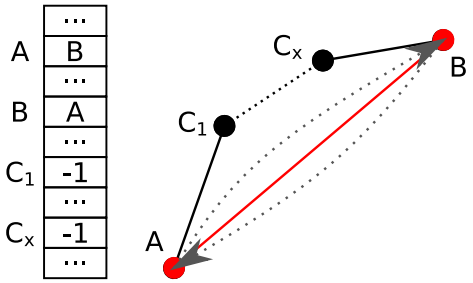


Abbildung 3.13: Wegsegment schließen

Laufzeit

Da die Prüfung der Kriterien in konstanter Laufzeit geschieht, und für maximal $\binom{n}{2}$ Kanten geschehen muss, würde eine quadratische Laufzeit resultieren. Der kritische Teil dieser Heuristik ist allerdings Schritt 1: Sortiere alle Kanten der Länge nach. Wie erwähnt gibt es bei n Knoten $\binom{n}{2}$ Kanten, für die Laufzeitbetrachtung kann man also von n^2 Kanten ausgehen. Da vergleichende Sortieralgorithmen im Worst Case in $nlog(n)$ terminieren ([6, Folie 95]), tun sie dies bei n^2 Elementen also in $n^2log(n)$.

4 Implementierung

Diese Arbeit basiert auf dem Programm TSPvisu, dem Ergebnis der Arbeit "Visualisierung verschiedener Algorithmen zum Travaling Salesperson Problem" von Florian Quengaj.[2]

Programmiersprache und Architekturmuster sind also vorgegeben. Java als Programmiersprache bietet unter anderem den signifikanten Vorteil, dass die Anwendung weitgehend plattformunabhängig ist. [2, Seite 15] Außerdem ist Java weit verbreitet und entsprechend bekannt, wodurch das Programm auch später für dritte Personen leicht zu ändern und zu erweitern ist, wie unter anderem diese Arbeit exemplarisch zeigt. Dazu trägt auch das Designpattern Model-View-Controller maßgeblich bei. Durch die Einteilung des Programms in drei überwiegend voneinander unabhängige Strukturen war es einfach möglich, neue Heuristiken oder Anpassungen bei der Anzeige hinzuzufügen oder umzusetzen. Die Strukturierung und der Aufbau des Programms ergeben sich ausführlich aus der bereits erwähnten Arbeit von Florian Quengaj und werden an dieser Stelle nicht weiter dargestellt. Die vorgenommenen relevanten Änderungen und Erweiterungen sind im Anhang aufgeführt.

4.1 Nutzung

Im Folgenden wird kurz auf die Bedienung des Programms eingegangen, da die Anordnung der Schaltflächen teilweise geändert wurde und neue hinzugekommen sind.

Zum Programmstart erscheint das Hauptfenster der Anwendung (Abbildung 4.1). Im oberen Teil des Fensters befindet sich der Bereich für die Steuerelemente (Abbildung 4.2), darunter der Anzeigebereich für den Graphen und unten eine Leiste für Informationen und Darstellungsoptionen (Abbildung 4.3).

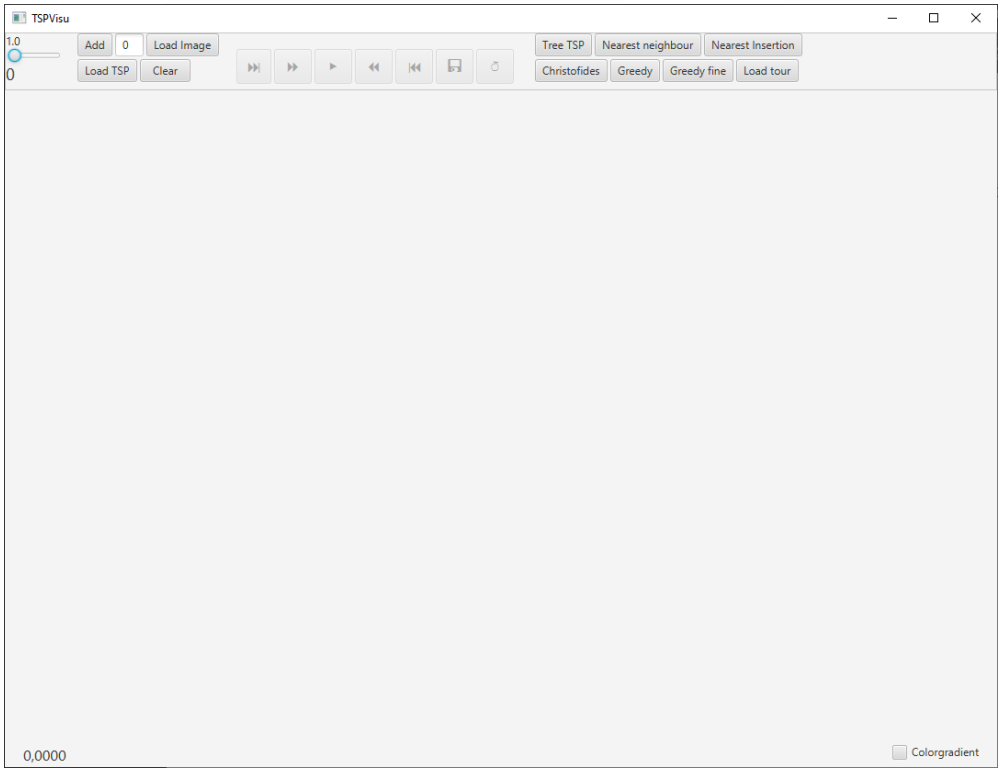


Abbildung 4.1: TSPvisu Hauptfenster

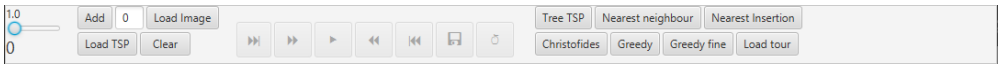


Abbildung 4.2: TSPvisu Steuerelemente

Abbildung 4.3: TSPvisu Fußleiste

4.1.1 Steuerelemente

Im ersten Bereich (Abbildung 4.4) der Steuerelemente befindet sich oben ein Slider mit dazugehöriger Anzeige für dessen Wert. Hiermit lassen sich die Größe der Knoten und die Dicke der Kanten im Anzeigebereich ändern. Darunter ist ein Zähler, der die aktuelle Anzahl an Knoten im Graphen darstellt.

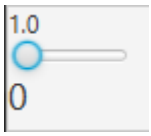


Abbildung 4.4:
Slider & Zähler

Daneben (Abbildung 4.5) befinden sich die Kontrollelemente für die Erstellung des Graphen. Mit "Add" und dem dazugehörigen Eingabebereich lassen sich die eingegebene Anzahl an zufälligen Knoten zum Graphen hinzufügen. "Load Image" lädt ein Hintergrundbild, mehr dazu in Kapitel 4.2.4. "Load TSP" dient zum Einladen von Graphen, siehe Kapitel 4.2.1. Mit "Clear" wird der Graph gelöscht.

Abbildung 4.5:
Graphenerstellung

Daneben (Abbildung 4.6) befinden sich die Animationssteuerelemente für die Schritte der jeweiligen Heuristiken. Von links nach rechts haben diese die folgenden Funktionen:

- In großen Schritten vorgehen
- In kleinen Schritten vorgehen
- Animation automatisch durchlaufen (Nicht implementiert)
- Einen kleinen Schritt zurückgehen
- Einen großen Schritt zurückgehen
- Speichern, siehe Kapitel 4.2.2
- Ursprünglichen Graphen wiederherstellen, siehe Kapitel 4.2.5

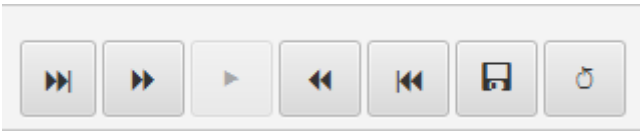


Abbildung 4.6: Animationssteuerelemte

Ganz rechts befinden sich die Elemente zum Ausführen der Heuristiken, siehe Abbildung 4.7. Durch Betätigen dieser werden, abgesehen von der letzten Schaltfläche, die jeweiligen Heuristiken auf den Graphen angewandt. “Tree TSP“ und “Christofides“ waren schon aus der vorhergegangen Arbeit implementiert, “Greed“ und “Greedy fine“ wenden die gleiche Heuristik an, stellen diese nur unterschiedlich detailliert dar. “Load Tour“ öffnet einen Auswahldialog um gespeicherte Rundwegen zu laden, siehe Kapitel 4.2.3.

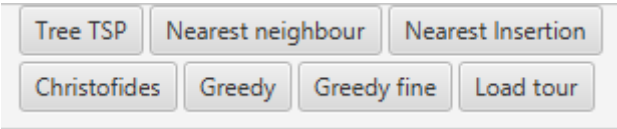


Abbildung 4.7: Heuristiken

4.2 Funktionale Erweiterungen

Das Programm sollte im Zuge dieser Arbeit um einige Funktionen erweitert werden. Diese umfassen die Ein- und Ausgabe von Graphen und Rundwegen, Erweiterungen in Bezug auf die Benutzerfreundlichkeit und mehr Optionen bei der Darstellung und Visualisierung.

Dateiformat

Für die Ein- und Ausgabe der Graphen und Rundwege werden *.tsp und *.tour Dateien verwendet, wie sie die Universität Heidelberg im Zuge der TSPLIB definiert hat. [3] Durch eine möglichst umfangreiche Unterstützung dieses Formats ergibt sich die Möglichkeit, andersartige Probleme zu lösen. Der ursprüngliche Funktionsumfang beschränkte sich auf 2-dimensionale euklidische TSPs. Als Distanzen zwischen den Knoten können aber nun auch andere als die 2-dimensionale euklidischen Distanz verwendet werden. Weiter kann sogar die Darstellung der Graphen komplett unabhängig von den Gewichtsmatrizen sein. Die aus dem TSPLIB Format übernommenen Optionen sind die folgenden:

- EUC_2D: Die Knoten sind in einem 2-dimensionalen Raum gegeben, es wird die euklidische Distanz zwischen ihnen berechnet.
- EUC_3D: Die Knoten sind in einem 3-dimensionalen Raum gegeben, es wird die euklidische Distanz zwischen ihnen berechnet. Sind keine separaten Daten

zur Anzeige der Knoten vorgegeben, werden nur deren x- und y-Koordinaten zur Darstellung verwendet.

- MAX_2D: Die Knoten sind in einem 2-dimensionalen Raum gegeben, es wird die Chebyshev Distanz zwischen ihnen berechnet. Dies ist in der Farbgradientendarstellung (siehe Kaptitel 4.2.6) in Abbildung 4.8 zu erkennen. Alle grünen Kanten haben in der x- bzw. y-Richtung die gleiche maximale Länge.

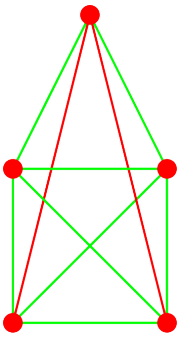


Abbildung 4.8: MAX_2D

- MAX_3D: Die Knoten sind in einem 3-dimensionalen Raum gegeben, es wird die Chebyshev Distanz zwischen ihnen berechnet. Sind keine separaten Daten zur Anzeige der Knoten vorgegeben, werden nur deren x- und y-Koordinaten zur Darstellung verwendet.
- MAN_2D: Die Knoten sind in einem 2-dimensionalen Raum gegeben, es wird die Manhatten-Distanz zwischen ihnen berechnet. In der Farbgradientendarstellung in Abbildung 4.9 ist dies an den grünen Kanten zu erkennen. Die Kanten des "Daches" legen exakt die halbe Höhe und Breite zurück, wie die anderen grünen Kanten hoch bzw. breit sind.

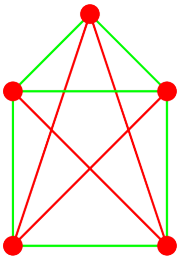


Abbildung 4.9: MAN_2D

- MAN_3D: Die Knoten sind in einem 3-dimensionalen Raum gegeben, es wird die Manhatten-Distanz zwischen ihnen berechnet. Sind keine separaten Daten zur Anzeige der Knoten vorgegeben, werden nur deren x- und y-Koordinaten zur Darstellung verwendet.
- CEIL_2D: Die Knoten sind in einem 2-dimensionalen Raum gegeben, es wird die aufgerundete euklidische Distanz zwischen ihnen berechnet.
- EXPLICIT: Die darzustellenden Knoten werden in einem anderen Bereich abgelegt. Es wird eine explizite Gewichtsmatrix für das TSP angegeben. Somit lassen sich noch gänzlich andere Distanzbeziehungen modellieren.

4.2.1 Graphen Einladen

TSPvisu sollte im Rahmen dieser Arbeit noch um die Möglichkeit erweitert werden, Graphen aus Dateien einlesen zu können. Dies hat zusätzlich den positiven Nebeneffekt,

die Funktionsweise der Heuristiken besser testen zu können. Beispielhaft ist ein solcher eingeladener Graph in Abbildung 4.12a zu sehen. Er stellt 52 Orte in Berlin dar.

4.2.2 Rundwege speichern

Weiter sollte auch die Option geschaffen werden, ermittelte Rundwege speichern zu können. Dies ist ebenfalls mit dem Dateiformat der TSPLIB möglich. Nachdem eine Heuristik auf den Graphen angewendet wurde, wird die Schaltfläche zum speichern auswählbar, siehe Abbildung 4.10. Beim betätigen wird ein Auswahldialog für den Speicherort und -namen aufgerufen. Der Rundweg wird daraufhin als *.tour Datei gespeichert.

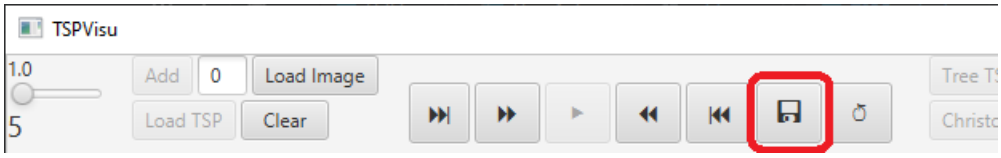


Abbildung 4.10: Schaltfläche: Rundweg speichern

4.2.3 Rundwege laden

Es besteht ebenfalls die Möglichkeit, Rundwege einzuladen. Die Schaltfläche dafür befindet sich im Bereich der Heuristiken, da hierdurch ebenfalls eine Tour auf dem Graphen erstellt wird, siehe Abbildung 4.11.

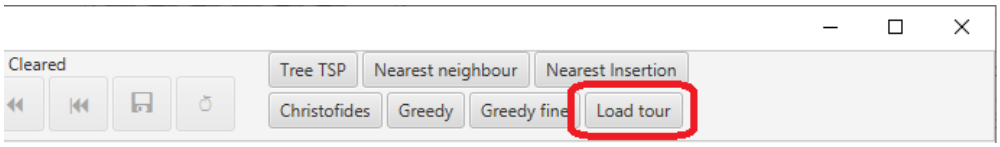
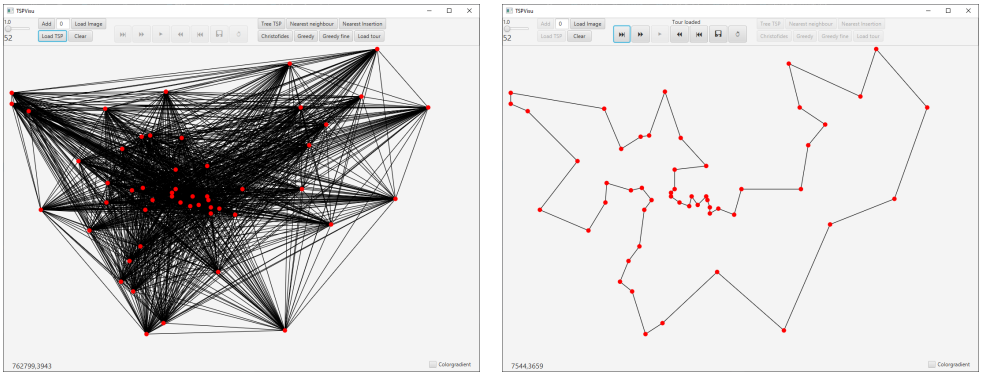


Abbildung 4.11: Schaltfläche: Rundweg einladen

Dadurch ist es ebenfalls möglich optimale Touren, wie sie unter anderem in der TSPLIB zur Verfügung stehen, einzuladen und darzustellen, um so zum Beispiel im Vergleich mit den anderen Heuristiken deren Güte bewerten zu können. Eine solche optimale Tour ist in Abbildung 4.12b zu sehen.



(a) Vollständiger Graph nach dem Einladen (b) Optimale Tour

Abbildung 4.12: Berlin52 aus der TSPLIB in TSPvisu

4.2.4 Einladen von Hintergrundbildern

Zu besseren Verdeutlichung der TSPs als solche kann es hilfreich sein ein Hintergrundbild einzuladen, beispielsweise von einer Landkarte. So können einfach beliebige Punkte, also im Beispiel einzelne Ort auf der Karte, angeklickt werden. Dadurch wird der Rundweg für den Nutzer greifbarer. Ein solcher Rundweg vor einem Hintergrundbild wird in Abbildung 4.13 gezeigt.

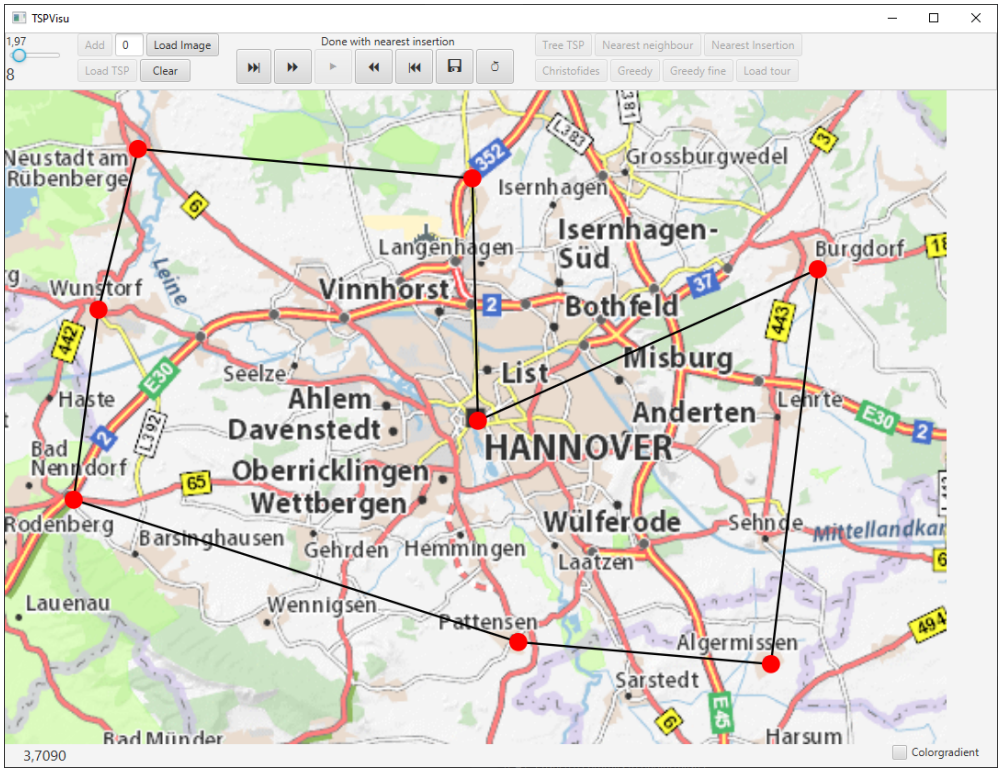


Abbildung 4.13: Bild eingeladen

4.2.5 Rundwegermittlung rückgängig machen

Bei der Nutzung des Programms hat sich gezeigt, dass es sehr praktisch ist, nach dem Durchlaufen einer Heuristik den Originalzustand des Graphen wieder herzustellen, um zum Beispiel auf dem gleichen Graphen eine andere Heuristik anzuwenden. Hierfür wurde eine weitere Schaltfläche hinzugefügt (Abbildung 4.14). Diese stellt wieder den initialen vollständige Graphen her. Außerdem werden die Schaltflächen zur Ausführung der Heuristiken wieder aktiviert und die Animationssteuerelemente und Schaltflächen zum speichern und rückgängig machen deaktiviert.

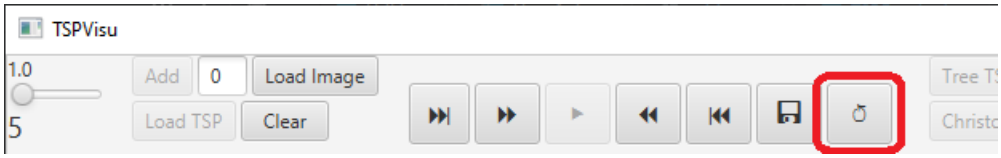


Abbildung 4.14: Schaltfläche: Rückgängig machen

4.2.6 Darstellung der Kantengewichte als Farbgradient

Durch die Möglichkeit Dateien einzuladen, in denen Kantengewichte von der 2-dimensionalen euklidische Distanz abweichen können, wurde die Notwendigkeit geschaffen, diese entsprechend darzustellen. Hierzu wurde eine Checkbox hinzugefügt, um zwischen der normalen Darstellung des Graphen mit schwarzen Kanten und einer Darstellung des Gewichts der Kanten als Farbgradient zu wechseln. Hierbei wird das geringste Kantengewicht grün dargestellt, das höchste rot. Die Bereiche dazwischen als Farbübergang über gelb, im HSV-Farbraum. (Abbildung 4.15)

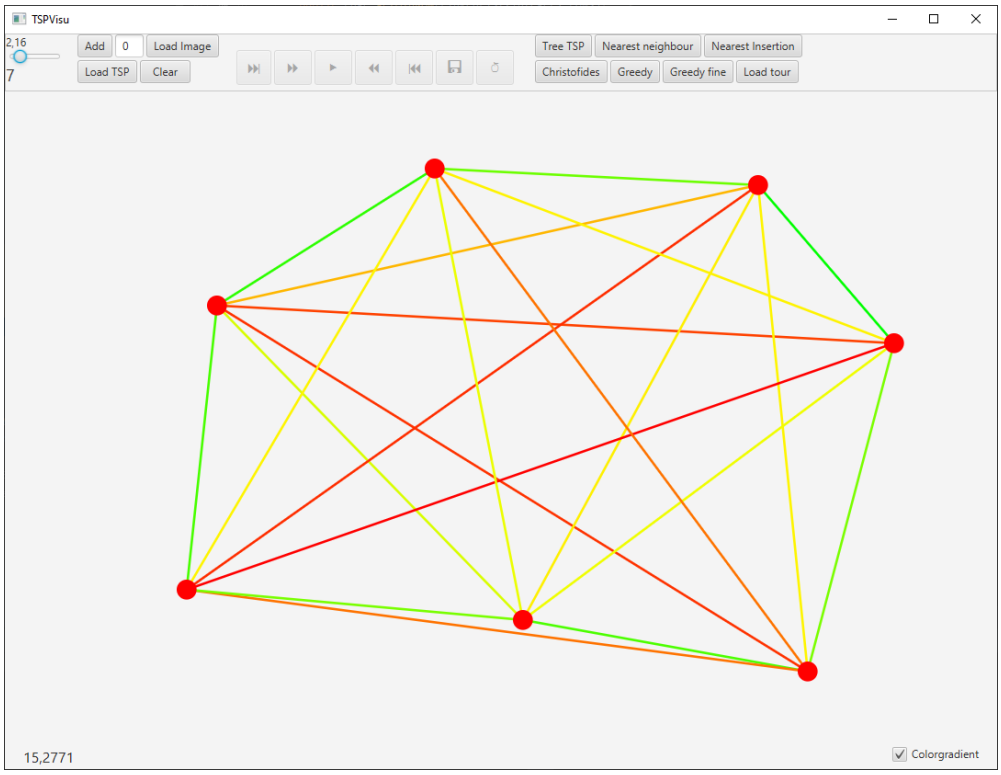


Abbildung 4.15: Farbgradient

5 Zusammenfassung und Ausblick

Im Zuge dieser Arbeit wurde das bestehende Programm TSPvisu sowohl um neue Heuristiken, als auch um weitere Funktionen erweitert. Hierfür wurde Eingangs auf die grundlegenden Begrifflichkeiten eingegangen um die verwendeten Verfahren kompakt beschreiben zu können. Die Heuristiken wurden daraufhin abstrakt erläutert und ihre Abläufe teilweise grafisch dargestellt. Zusätzlich wurde detaillierter auf zeitkritische Bereiche der Implementierung eingegangen. Besonderes Augenmerk wurde dabei darauf gelegt, wie spezielle Suchen und Prüfungen während der Ermittlung der Rundwege schneller durchgeführt werden können. Weiter wurde kurz auf die grundlegenden Funktionen des Programms eingegangen, tiefgehender auf die, die in dieser Arbeit hinzugefügt wurden.

Die umzusetzenden Heuristiken wurden erfolgreich in die bestehen Programmstruktur integriert und sichergestellt, dass die Laufzeit dieser den theoretischen Vorgaben entsprechen. Dank der Modularität des gegebenen Programmaufbaus war dies möglich, ohne große Bereiche der Struktur anpassen zu müssen. Durch die Unterstützung des Dateiformats des TSPLIB Projekts ergab sich außerdem die Möglichkeit, andere Knotendistanzen als die 2-dimensionale euklidische Distanz zu verwenden. Hieraus resultierte die Notwendigkeit, diese entsprechend darzustellen. Die Darstellung der Kanten als Farbgradient ist dem sehr dienlich, doch auch bei der euklidischen Distanz sorgt sie mitunter für eine bessere Übersicht über den Graphen. Weiter stellte sich bei der praktischen Nutzung heraus, dass das Zurücksetzten des Graphen in seinen Anfangszustand vorteilhaft ist, um verschiedenen Heuristiken besser vergleichen zu können. Letztlich ist durch die Möglichkeit Hintergrundbilder einzuladen die Option geschaffen worden, das Problem des Handlungsreisenden anhand einfacher grafischer Beispiele leicht verständlich zu machen.

Wie diese Arbeit exemplarisch bewiesen hat, sind weitere Heuristiken einfach zu integrieren. Es wäre durchaus denkbar, andere Eröffnungsheuristiken wie zum Beispiel die Farthest-Insertion-, oder die Random-Insertion-Heuristik zu übernehmen, interessanter jedoch wären wohl verbessernde Heuristiken, wie die k-Opt-Heuristik auf gefundene Routen anzuwenden. Denkbar wäre ebenfalls ein grafischer Dateiexport aller

Einzelschritte einer Heuristik, zum Beispiel für Animationen in Präsentationen.

Literaturverzeichnis

[1] Christian Nilsson. Heuristic algorithms for travelling salesman problem. *Linköping University*, 2003.

[2] Florian Qengaj. Visualisierung verschiedener algorithmen zum travaling salesperson problem, 2019.

[3] Gerhard Reinelt. Tsplib95, 1995. URL `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95`.

[4] Steffen Reith and Heribert Vollmer. Ist P = NP? Einführung in die theorie der NP-vollständigkeit. Technical Report 269, Universitat Würzburg, 2001. URL `https://www.thi.uni-hannover.de/fileadmin/thi/publikationen/re-vo01.pdf`.

[5] Daniel Rosenkrantz, Richard Stearns, and Philip II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6:563–581, 09 1977. doi: 10.1137/0206041. URL `https://www.researchgate.net/publication/220616869_An_Analysis_of_Several_Heuristics_for_the_Traveling_Salesman_Problem`.

[6] G. Zachmann. Algorithmen und datenstrukturen, 2010. URL `https://cgvr.cs.uni-bremen.de/teaching/info2_10/folien/10_sortieren_3.pdf`. Zugegriffen am: 30.05.2020.

A Die Heuristiken in Java

A.1 Nearest-Neighbour-Heuristik

Der erste Teil des Codes stellt den ersten Schritt der Nearest-Neighbour-Heuristik aus Kaptiel 3.1, bzw. der Umsetzung aus 3.1.1 dar. Es werden die nötigen Datenstrukturen angelegt, unter anderem eine Kopie der Gewichtsmatrix (Z. 370), und ein beliebiger Startknoten ausgewählt (Z. 372). Wie bei der Umsetzung erwähnt wird die Spalte des Startknotens in der Matrix mit 0 überschrieben (Z. 373).

```
369     public static LinkedList<LinkedList<Double>>
            NearestNeighbour(Graph graph) {
370         Graph.AdjMatrix adj = graph.getAdjMatrix();
371         LinkedList<Graph.Edge> path = new LinkedList<>();
372         int currentNode = (int) (Math.random() * graph.
                getSize());
373         adj.setCol(currentNode, 0.0);
```

Danach wird $n-1$ Mal der nächstgelegene Knoten zum zuletzt ausgewählten ermittelt (Z. 375), zum Pfad hinzugefügt (Z. 376) und abermals die dazugehörige Spalte in der Gewichtsmatrix auf 0 gesetzt (Z. 379).

```
374         for (int i = 0; graph.getSize() - 1 > i; i++) {
375             Graph.Edge nextEdge = adj.getMinRow(
                    currentNode);
376             path.add(nextEdge);
377             currentNode = nextEdge.node2;
378             addLineState(graph, "Add shortest edge",
                    nextEdge.node1, nextEdge.node2, StateColor.
                    BLUE);
379             adj.setCol(currentNode, 0.0);
380         }
```

Letztlich wird der Pfad mit einer Kante vom letzten zum ersten Knoten geschlossen (Z. 381) und die nötigen Rückgabeparameter erstellt.

```
381             path.add(graph.getAdjMatrix().getEdge(currentNode,
                    path.get(0).node1));
382             addLineState(graph, "close the route", currentNode
                    , path.get(0).node1, StateColor.BLUE);
383             LinkedList<LinkedList<Double>> NN = path2adj(path)
                    ;
384             graph.addstate(new GraphState(new Graph(graph.
                    getVertices(), NN), "Done with nearest
                    Neighbour", StateColor.GREEN));
385             return NN;
386         }
```

A.2 Nearest-Insertion-Heuristik

Für die Nearest-Insertion-Heuristik werden auch zuerst alle Datenstrukturen angelegt und initialisiert, insbesondere `unusedNodes` und `nearestUsedNode`, wie in der Umsetzung in Kapitel 3.2.1 erläutert.

```
173         public static LinkedList<LinkedList<Double>>
                NearestInsertion(Graph graph) {
174             int n = graph.getSize();
175             Graph.AdjMatrix distances = graph.getAdjMatrix();
176             Graph.AdjMatrix adj = graph.getAdjMatrix();
177             LinkedList<Graph.Edge> path = new LinkedList<>();
178             ArrayList<Integer> unusedNodes = new ArrayList<>(n
                    );
179             ArrayList<Integer> nearestUsedNode = new ArrayList
                    <>(n);
180
181             for (int i=0;i<n;i++)
182                 unusedNodes.add(i);
```

Danach wird nach der kürzesten Kante als erster Teil des initialen Rundweges gesucht (Z. 185) und der erste Knoten dieser Kante allen anderen Knoten in `nearestUsedNode` als

nächstgelegener Pfadknoten zugewiesen (Z. 190, 191). Daraufhin wird der zweite Knoten der ersten Kante hinzugefügt und der diesen beiden nächstgelegene Knoten gefunden um den initialen Rundweg zu vervollständigen. Die notwendigen Datenstrukturen, um die nächstgelegenen Knoten in linearer Laufzeit zu ermitteln, werden währenddessen entsprechend aktualisiert.

```
193             int secondNode = firstEdge.node2;
194             unusedNodes.remove((Integer) secondNode);
195             nearestUsedNode.remove((Integer) secondNode);
196             //update nearestUsedNode structure
197             for (int unusedNode:unusedNodes){
198                 double oldMinDist = distances.get(unusedNode,
                        nearestUsedNode.get(unusedNode));
199                 double newDist = distances.get(secondNode,
                        unusedNode);
200                 if (oldMinDist > newDist){
201                     nearestUsedNode.set(unusedNode, secondNode
                            );
202                 }
203             }
204
205             path.add(firstEdge);
206             addLineState(graph, "NI: Start with shortest edge"
                    , firstNode, secondNode, StateColor.BLUE);
207             //second edge
208             //get third node in the old inefficient way
209             int thirdNode = 0;
210             double minDist = Double.MAX_VALUE;
211             for (int unusedNode:unusedNodes){
212                 double dist = distances.get(unusedNode,
                        nearestUsedNode.get(unusedNode));
213                 if (dist<minDist) {
214                     minDist = dist;
215                     thirdNode = unusedNode;
216                 }
217             }
```

```
218             unusedNodes.remove((Integer) thirdNode);
219             //update nearestUsedNode structure
220             for (int unusedNode:unusedNodes){
221                 double oldMinDist = distances.get(unusedNode,
                        nearestUsedNode.get(unusedNode));
222                 double newDist = distances.get(thirdNode,
                        unusedNode);
223                 if (oldMinDist > newDist){
224                     nearestUsedNode.set(unusedNode, thirdNode)
                            ;
225                 }
226             }
227             //display first path
228             addPointState(graph, "Find nearest node",
                    thirdNode, StateColor.GREEN);
229             path.add(new Graph.Edge(secondNode, thirdNode,
                    distances.get(secondNode, thirdNode)));
230             addLineState(graph, "Add edge to new node",
                    secondNode, thirdNode, StateColor.BLUE);
231             //third edge
232             Graph.Edge thirdEdge = new Graph.Edge(firstNode,
                    thirdNode, distances.get(firstNode, thirdNode))
                    ;
```

Danach wird, solange der Rundweg nicht vollständig ist, der nächstgelegene Knoten außerhalb des Pfades zu denen innerhalb gesucht (Z. 237 - 245), dieser aus `unusedNodes` entfernt (Z. 246) und wieder die Einträge in `nearestUsedNode` aktualisiert (Z. 248 - 254). Daraufhin wird nach der Kante gesucht, zwischen deren Knoten der neue Knoten eingefügt werden soll (Z. 256 - 264).

```
236             while (path.size() < n) {
237                 int newNode = 0;
238                 minDist = Double.MAX_VALUE;
239                 for (int unusedNode:unusedNodes){
240                     double dist = distances.get(unusedNode,
                            nearestUsedNode.get(unusedNode));
241                     if (dist<minDist) {
```

```
242                     minDist = dist;
243                     newNode = unusedNode;
244                 }
245             }
246             unusedNodes.remove((Integer) newNode);
247             //update nearestUsedNode structure
248             for (int unusedNode:unusedNodes){
249                 double oldMinDist = distances.get(
                        unusedNode, nearestUsedNode.get(
                        unusedNode));
250                 double newDist = distances.get(newNode,
                        unusedNode);
251                 if (oldMinDist > newDist){
252                     nearestUsedNode.set(unusedNode,
                            newNode);
253                 }
254             }
255
256             Graph.Edge minExtraDistEdge = new Graph.Edge()
                    ;
257             double minExtraDist = Double.MAX_VALUE;
258             for (Graph.Edge e : path) {
259                 double extraDist = distances.get(e.node1,
                        newNode) + distances.get(e.node2,
                        newNode) - e.distance;
260                 if (extraDist < minExtraDist) {
261                     minExtraDist = extraDist;
262                     minExtraDistEdge = e;
263                 }
264             }
265             addPointState(graph, "Find␣nearest␣outer␣node"
                    , newNode, StateColor.YELLOW);
266             path.remove(minExtraDistEdge);
267             addLineState(graph, "remove␣edge␣to␣be␣
                replaced", minExtraDistEdge.node1,
                minExtraDistEdge.node2, StateColor.RED);
```

```
268                 path.add(distances.getEdge(minExtraDistEdge.
                        node1, newNode));
269                 addLineState(graph, "add first edge to new
                        node", minExtraDistEdge.node1, newNode,
                        StateColor.BLUE);
270                 path.add(distances.getEdge(newNode,
                        minExtraDistEdge.node2));
271                 addLineState(graph, "add second edge to new
                        node", newNode, minExtraDistEdge.node2,
                        StateColor.BLUE);
272
273             }
```

Letztendlich wird der ermittelte Pfad zurückgegeben.

```
274             LinkedList<LinkedList<Double>> NI = path2adj(path)
                    ;
275             graph.addstate(new GraphState(new Graph(graph.
                    getVertices(), NI), "Done with nearest
                    insertion", StateColor.GREEN));
276             return NI;
277         }
```

A.3 Greedy-Heuristik

Zu Anfang in der Greedy-Heuristik werden erst wieder die nötigen Strukturen initialisiert, wie in Kapitel 3.3.1 beschrieben, allen voran `otherNode`. In Z. 105 werden danach alle Kanten des Graphen sortiert und die kürzeste als erste ausgewählt (Z. 107 - 108). Die Strukturen um Knotengrade und Kreise zu ermitteln werden entsprechend angepasst (Z. 109 - 112).

```
94          public static LinkedList<LinkedList<Double>> Greedy(
                Graph graph, boolean fine) {
95
96              int n = graph.getSize();
97              int nEdges = n * (n - 1) / 2;
98              Graph.Path path = new Graph.Path();
```

```
99          int[] degrees = new int[n];
100         int[] otherNode = new int[n];
101         int unused = -1;
102         Arrays.fill(otherNode, unused);
103
104         LinkedList<Graph.Edge> edges = graph.edgeList();
105         edges.sort(Comparator.comparingDouble(o -> o.
                distance));
106
107         Graph.Edge firstEdge = edges.pop();
108         path.add(firstEdge);
109         degrees[firstEdge.node1]++;
110         degrees[firstEdge.node2]++;
111         otherNode[firstEdge.node1] = firstEdge.node2;
112         otherNode[firstEdge.node2] = firstEdge.node1;
```

Daraufhin wird in einer Schleife die nächst kürzeste Kante ermittelt (Z. 118) und überprüft, ob die Grade der Knoten dieser Kante 2 betragen (Z. 121), oder diese Kante einen Kreis schließt (Z.122). Wenn die Kante einen Kreis schließt, aber auch die letzte fehlende Kante zum Rundweg darstellt, wird die Schleife verlassen, der Rundweg ist komplett (Z. 125 - 129). Wird ansonsten der Kreis geschlossen, oder sind Knotengrade zu groß, wird die Kante verworfen (Z. 132 - 136). Ist dies alles nicht der Fall, so werden die Knotengrade aktualisiert und mittels einer Fallunterscheidung auch `otherNode` (Z. 140 - 158).

```
117         for (int i = 0; i < nEdges - 1; i++) {
118             Graph.Edge e = edges.pop();
119             if (fine)
120                 addLineState(graph, "Check␣shortest␣edge",
                        e.node1, e.node2, StateColor.YELLOW);
121             boolean degreesTooHigh = degrees[e.node1] > 1
                    || degrees[e.node2] > 1;
122             boolean circleClosed = otherNode[e.node1] == e
                    .node2;
123
124             //if the path has enough edges and a circle
                    was created we are done
```

```
125             if (path.size() == n - 1 && circleClosed) {
126                 path.add(e);
127                 addLineState(graph, "Close route", e.node1
                        , e.node2, StateColor.BLUE);
128                 break;
129             }
130
131             //skip current edge if the degrees of one node
                    would get to high or a circle would be
                    created
132             if (degreesTooHigh || circleClosed) {
133                 if (fine)
134                     addLineState(graph, "Discard Edge", e.
                            node1, e.node2, StateColor.RED);
135                 continue;
136             }
137             path.add(e);
138             addLineState(graph, "Accept edge", e.node1, e.
                    node2, StateColor.BLUE);
139
140             degrees[e.node1]++;
141             degrees[e.node2]++;
142             if (otherNode[e.node1] != unused && otherNode[
                    e.node2] == unused) {  //extend
143                 otherNode[otherNode[e.node1]] = e.node2;
144                 otherNode[e.node2] = otherNode[e.node1];
145                 otherNode[e.node1] = unused;
146             } else if (otherNode[e.node2] != unused &&
                    otherNode[e.node1] == unused) {   //extend
147                 otherNode[otherNode[e.node2]] = e.node1;
148                 otherNode[e.node1] = otherNode[e.node2];
149                 otherNode[e.node2] = unused;
150             } else if (otherNode[e.node1] == unused &&
                    otherNode[e.node2] == unused) {    //new
151                 otherNode[e.node1] = e.node2;
152                 otherNode[e.node2] = e.node1;
```

```
153                 } else {    //connect
154                     otherNode[otherNode[e.node1]] = otherNode[
                            e.node2];
155                     otherNode[otherNode[e.node2]] = otherNode[
                            e.node1];
156                     otherNode[e.node1] = unused;
157                     otherNode[e.node2] = unused;
158                 }
159             }
```

Letztlich wird der ermittelte Pfad wieder zurückgegeben.

```
162             LinkedList<LinkedList<Double>> GR = path2adj(path)
                    ;
163             graph.addstate(new GraphState(new Graph(graph.
                    getVertices(), GR), "Done with Greedy",
                    StateColor.GREEN));
164             return GR;
165         }
```

B Die Erweiterungen in Java

B.1 Einladen von Hintergrundbildern

Nach einem Auswahldialog wird das Hintergrundbild eingeladen (Z. 575) und die Dimensionen des Bildes und des Anzeigebereichs ermittelt (Z. 576 - 580). Daraus wird berechnet, wie das Bild skaliert werden muss, um möglichst groß dargestellt werden zu können (Z. 579 - 587). Die Anzeige wird über CSS Eigenschaften der Javakomponenten umgesetzt (Z. 589 - 590).

```
557     public void loadBackground() throws
            FileNotFoundException {
558         //...
575         Image image = new Image(new FileInputStream(String
                .valueOf(file)));
576         double imgW = image.getWidth();
577         double imgH = image.getHeight();
578
579         double W = map.getWidth();
580         double H  = map.getHeight();
581         double scale;
582         if (imgW/imgH > W/H)
583             scale = W/imgW;
584         else
585             scale = H/imgH;
586         imgW *= scale;
587         imgH *= scale;
588
589         String style = "-fx-background-image: url(" + file.
                toURI() + "); -fx-background-repeat: stretch; -fx
                -background-size: " + imgW + ", " + imgH;
```

```
590             this.BaseLayout.getChildren().get(2).setStyle(
                    style);
591
592         }
```

B.2 Darstellung der Kantengewichte als Farbgradient

Um die Kanten als Farbgradienten darstellen zu können wurde die Funktion `onDraw(...)` modifiziert. Es werden in Z. 146 - 147 das minimale und maximale Kantengewicht ermittelt und daraus mit dem Gewicht `w` der jeweiligen Kante eine Farbe aus dem HSV-Farbraum ermittelt (Z. 168 - 169). Soll die Kante nicht farbig dargestellt werden, wird sie schwarz angezeigt (Z. 171).

```
145
146             double maxWeight = graph.getAdjMatrix().
                    getMaxWeight();
147             double minWeight = graph.getAdjMatrix().
                    getMinWeight();
148             //...
166                                 Color c;
167                                 if (useGradient)
168                                     //Wenn Kanten als
                                            Farbgradient
                                            dargestellt werden
                                            sollen. Geringes
                                            Gewicht -> Gruen, hohes
                                             Gewicht -> Rot
169                                     c = javafx.scene.paint.
                                            Color.hsb(-((w-
                                            minWeight) / (maxWeight
                                             - minWeight))*120+120,
                                             1,1 );
170                                 else
171                                     c = Color.BLACK;
172                                 gc.setStroke(c);
```

```
173                                 gc.strokeLine(c1x + (diameter
                                        / 2), c1y + (diameter / 2),
                                         c2x + (diameter / 2), c2y
                                        + (diameter / 2));
```