Leibniz Universität Hannover

Institut für Theoretische Informatik

Visualisierung verschiedener Algorithmen zum Travaling Salesperson Problem

Eine Bachelorarbeit von

Florian Qengaj
Matr. Nr.: 3216880

11. November 2019

Prof. Dr. Heribert Vollmer
Erstprüfer

Dr. Arne Meier
Zweitprüfer

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hannover, 11. November 2019 Unterschrift

Inhaltsverzeichnis

1

Einleitung

> *„Time is really the only capital that any human being has, and the only thing he can't afford to lose."*
>
> – Thomas A. Edison

Damals zu Zeiten Thomas A. Edison so wie auch heute ist Zeit unser wertvollstes Gut. In allen Gebieten der Wirtschaft gilt daher: Zeit ist Geld. Besonders Unternehmen wie Speditionen, Lieferdienste, ambulante Pflegedienste, aber auch Bau- und Handwerksgewerbe sind daher bestrebt in möglichst kurzer Zeit viel zu leisten. Beispielsweise benötigen Unternehmen, die Mitarbeiter im Außendienst haben, Softwareunterstützung zur Disposition und Kontrolle ihrer Mitarbeiter. Dies soll die Wettbewerbsfähigkeit am Markt gewährleisten. Hierzu werden beispielsweise die kostengünstigsten Routen berechnet. Zusätzliche Faktoren wie die Streckenlänge, benötigte Fahrzeit und auch weitere Verkehrsinformationen wie z.B. Baustellen oder Stau sind ebenfalls wichtig, da diese Informationen zu einer Variation der Wegekosten führen können. Mit unserer heutigen, modernen Technik wird die Verbindung zwischen zwei Punkten relativ schnell berechnet und dem Nutzer angezeigt. Noch vor kurzer Zeit haben Nutzer auf ein analoges Medium wie Atlas, Straßenatlas oder Stadtkarten zurückgegriffen. Doch da das sehr mühselig ist, blieb die Frage der Automatisierung stets offen; wie kann eine Optimierung der Routenplanung, unter Einbezug der Kostenfaktoren, erzielen werden?

Diese prägnante Fragestellung ist gleichzeitig auch Kern des Traveling Salesperson Problem. Das Problem des Handelsreisenden ist ein Minimierungsproblem aus dem Bereich der theoretischen Informatik, genauer ist dies ein NP-vollständiges

Entscheidungsproblem. Diese Probleme werden als „besonders schwer“ bezeichnet, da für das exakte Lösen dieser Probleme keine effizienten Lösungsverfahren existieren. Anhand eines Routenplaners lässt sich dies beispielhaft verdeutlichen. Möchte ein Nutzer die kürzeste Route von Hannover nach Berlin finden, müsste der Routenplaner alle möglichen Routen von Hannover bis Berlin ausprobieren und miteinander vergleichen. Schnell wird deutlich, dass das bloße ausprobieren jeder einzelnen Route enorm viel Zeit beanspruchen wird. Solche Algorithmen heißen Brute-Force Algorithmen, denn sie ziehen jede mögliche Lösung in Betracht und finden damit auch immer das Optimum. Auch wenn die Lösung immer Optimal ist, ist es oft nicht praktikabel. Entsprechend kann beispielsweise auch im Kreis gefahren oder eine Straße genommen werden, die entgegen der Richtung des Ziels führt. Das Problem des Handelsreisenden ist also, dass die Lösungsmöglichkeiten exponentiell wachsen und damit die Laufzeit des Algorithmus ebenso. Ein grundlegendes Verständnis für die Optimierung der Entscheidungsprobleme, hier speziell des Handelsreisenden, ist daher essentiell um immer effizientere Lösungsverfahren zu realisieren. In der vorliegenden Arbeit gehe ich daher im weiteren Verlauf zunächst auf die theoretischen Grundlagen ein und komme dann im 3. Kapitel (S. 10) zu den Approximationsalgorithmen mit der minimalen Spannbaum und Christofides Heuristik, die ich näher erläutere, da sie den Kern für die im 4. Kapitel (S. 14) durchgeführte Implementierung darstellen. Schließlich gebe ich im 5. Kapitel (S. 28) ein Fazit meiner Arbeit.

2

Grundlagen

In diesem Kapitel erörtere ich zunächst alle notwendigen Definitionen und Vokabeln, die zum Verständnis dieser Arbeit beitragen sollen. Sowohl für die theoretische Grundlage der verwendeten Algorithmen im Kapitel 3 (S. 10) als auch für die Implementierung, welche im Kapitel 4 (S. 14) behandelt wird.

Im Abschnitt 2.1 werden zunächst die notwendigen Begriffe der Graphentheorie erklärt. Im Anschluss werden, in Abschnitt 2.2, Begriffe definiert, die für die Analyse der Problemstellung des Traveling Salespersons essentiell sind.

2.1 Begriffe der Graphentheorie

Ungerichtete Graph

Ein ungerichteter Graph

$$G = (E, K)$$

besteht aus einer nicht leeren, endlichen Eckenmenge

$$|E| \geq 1$$

und einer Kantenmenge

$$K \subseteq \{\{a, b\} \in E \mid a \neq b\}.$$

Insbesondere schließen diese Graphen die Exisitenz von Schleifen und doppelten Kanten aus. Des Weiteren benötigt, ist der Begriff der Vollständigkeit, um das Traveling Salesperson Problem beschreiben zu können. Ein Graph $G = (E, K)$ ist vollständig genau dann, wenn gilt:

$$\forall v, w \in E \; gilt \; v \neq w \rightarrow \{v, w\} \in K$$

Mit anderen Worten, zwischen zwei beliebigen, aber von einander verschiedenen, Knoten besteht immer eine direkte Verbindung. Jeder Knoten ist daher in genau $|E| - 1$ Kanten enthalten.

Es sei $G = (E, K)$ ein Graph. Eine Folge von Ecken $u_1 \dots u_n \in E$ heißt Weg, oder Pfad in G, wenn gilt:

$$\{u_i, u_{i+1}\} \in K \text{ für } i = 1 \dots n - 1$$

Bemerkung: Da wir bei dieser Arbeit von einen vollständigen, ungerichteten Graphen ausgehen, ist jede folge von Ecken aus E, ohne Gleiche aufeinander folgende Ecken, ein Pfad.

Außerdem möchte ich für einen Pfad p folgendes definieren:

- p heißt *einfach*, wenn gilt, alle Kanten in p paarweise verschieden sind.
- p hießt *eckeneinfach*, wenn gilt, alle Ecken in p paarweise verschieden sind.
- p heißt *geschlossen*, wenn gilt, die erste Ecke in p gleich der letzten ist.
- p heißt *Kreis*, wenn gilt, p ist einfach, geschlossen und eckeneinfach.

Gewichtete Graphen

Sei $G = (E, K, D)$ ein Graph, dann definieren wir D als eine Matrix mit den Gewichten der Kanten:

$$D = \left(d_{i,j}\right)_{\{i,j\} \in K}$$

Spannbaum

Sei $G = (E, K)$ ein Graph. G nennt man Baum, wenn gilt:

- G enthält kein Kreis
- Für alle Knoten $p, q \in E$ existiert ein Weg der p mit q verbindet.

Ein Spannbaum $T = (E, V)$ in $G = (E, K)$ ist ein Teilgraph von G, sodass T ein Baum mit $V \subseteq K$. Der Baum T ist minimal, wenn V mit minimalem Gewicht ist.

Maching

Sei $G = (E, K)$ ein Graph. Für $u \in E$ sei der Grad von u

$$deg(v) := |\{\{a, b\} \in K \mid u \in \{a, b\}\}$$

Ein Matching in G ist eine Kantenteilmenge, wenn für alle $u \in E$ gilt: $deg(u) \geq 1$. Das Matching ist perfekt oder maximal wenn gilt, dass $deg(u) = 1$. Das perfekte Matching mit den geringsten Kantengewichten wird als minimal bezeichnet.

Euklidischer Abstand

Der euklidische Abstand ist anschaulich, der Abstand, den man mithilfe eines Lineals zwischen zwei Punkten zeichnen und messen könnte. Der Abstand zweier Punkte $p = (p_1, \ldots, p_n)$ und $q = (q_1, \ldots, 1_n)$ mit $p_1, \ldots, p_n, q_1, \ldots, q_n \in \mathbb{R}$ wird definiert als:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + \ldots + (p_n - q_n)^2} \tag{2.1}$$

Da ein Quadrat jeder Zahl größer gleich null ist, ist die Summe in der Wurzel auch stets größer gleich null. Dadurch lässt sich folgern, dass die Distanz zweier echt verschiedener Punkte positiv ist. Es gilt $(x - y)^2 = (y - x)^2$ für $x, y \in \mathbb{R}$ daraus folgt eine Symmetrie. Des Weiteren erfüllt die eukidische Distanzfunktion auch die Dreiecksungleichung. Also gilt:

$$d(p, q) = d(q, p) \qquad \text{(Symmetrie)}$$
$$d(p, j) + d(j, q) \geq d(p, q) \qquad \text{(Dreiecksungleichung)}$$

Eulerkreis

Im 18. Jahrhundert stellte Leaonard Euler erstmals das Königsberger Brückenproblem. Dort suchte Euler einen Spazierweg über die Brücken von Königsberg, sodass er wieder zu Hause ankäme. Die Suche stellte sich problematisch heraus, da er über jede der sieben Brücken genau einmal gehen wollte (mehr dazu unter [6] aufrufbar). In Abbildung 2.1 ist eine Skizze der Königsberger Brücken aus dem

18. Jahrhundert zu sehen. Die gelben Rechtecke sollen Brücken darstellen, die zwischen den weißen Landmassen verbinden.

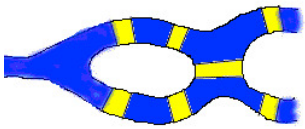

Abbildung 2.1: Skizze von Königsberg aus [6]

Diese Skizze lässt sich in einen Graphen übersetzen, indem man die Landmassen als Knoten und die Brücken als Kante zwischen den Knoten modelliert werden. So erhalten wir den in Abbildung 2.2 skizzierten Graphen.

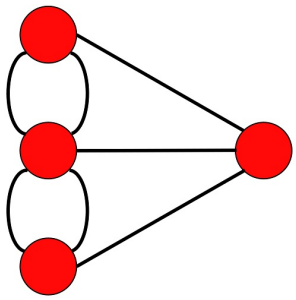

Abbildung 2.2: Könisberg als Graph

Wird die Fragestellung auf dieses Modell überführt, so führt es auf die Suche nach einem geschlossenen, einfachen Weg in dem Graphen.

Es wird definiert, dass in einem Graphen $G = (E, K)$ ein Eulerkreis existiert gdw. wenn für jeden Knoten $u \in E$ gilt, dass der Grad von u gerade ist. Auf den ersten Blick auf Abbildung 2.2 zu sehen, keiner der Knoten hat einen geraden Grad. Somit gibt es keinen Eulerweg und insbesondere auch keinen Eulerkreis. Konkret heißt das: Euler musste in seinem täglichen Spaziergang wenigstens eine Brücke doppelt gehen[1].

[1] Zwei der Brücken wurden im Verlauf des Krieges 1945 zerstört. Es ist seit dem Möglich einen einfachen Weg zu finden, aber immer noch keinen Kreis.

2.2 Travaling Salesperson Problem

Angenommen ein Handelsreisender möchte einige Städte besuchen, um dort geschäftlich tätig zu sein. Er startet in Hannover und möchte Berlin, Kassel, Hamburg, Leipzig und Magdeburg besuchen, bevor er wieder nach Hannover zurück fährt. Da der Handelsreisende nur in seiner Stammtankstelle tankt, fragt er sich, ob er die gesamte Reise, mit einem vollen Tank, schaffen kann. Um einen möglichst kurzen Reiseweg zu haben, muss sich der Handelsreisende nun Gedanken machen, in welcher Reihenfolge er die Städte besuchen wird. Er entscheidet sich für die Strecke Hannover, Kassel, Leipzig, Magdeburg, Berlin, Hamburg, Hannover[2]. Für einen Menschen ist es, bei 6 Reisezielen, noch recht intuitiv eine Lösung zu finden, die von der bestmöglichen Lösung nur wenig abweicht. Doch schon bei 15 Reisezielen gibt es rund 44 Milliarden mögliche Rundreisen und selbst die Intuition des Menschen tut sich schwer eine optimale Lösung zu finden. Daher ist es hier dringend notwendig Lösungsverfahren zu finden, welche, selbst bei 3 oder 4-stelligen Zielpunkten, eine gute in Lösung annehmbarer Zeit finden können.

2.2.1 Das Entscheidungsproblem

Doch geht es bei dem Travaling Salesperson Problem (kurz TSP) nicht nur um Städte. Abstrahiert wird in einem ungerichteten und vollständigen Graphen mit Kantengewichten, nach einem minimalen *Kreis* gesucht[3].

Das TSP gehört zu der Klasse der NP-Vollständigen Probleme. Das bedeutet, es gibt keinen effizienten Algorithmus um das Problem zu entscheiden. Den Beweis dafür, sowie die Definition des Entscheidungsproblems entstammen von Arne Meier und Heribert Vollmer in ihrem Buch „Komplexität von Algorithmen“ ([1] S.101). Das Entscheidungsproblem ist wie folgt definiert:

$$TSP = \left\{ \langle \left(d_{i,j}\right)_{1\leq i,j\leq n}, B\rangle \;\middle|\; \begin{array}{l} n, d_{i,j} \in \mathbb{N} \text{ und es gibt ein} \\ \pi \in S_n \text{ mit } \sum_{i=1}^{n-1} d_{\pi(n),\pi(i+1)} + \\ d_{\pi(n),\pi(1)} \leq B \end{array} \right\} \tag{2.2}$$

[2] Entsprechende Route Mithilfe von Google Maps einsehbar unter https://bit.ly/TSPVisu

[3] Äquivalent dazu ist die Aussage, dass nach einem Hamiltonkreis mit minimalen Kantengewichten gesucht wird.

Die Eingaben sind als Adjazenzmatrix zu verstehen mit den Distanzen als Einträge. Die Schranke B ist beim Gedanken an das eingangs erwähnte Beispiel, die Anzahl der Kilometer die der Handelsreisende mit vollem Tank erreichen kann. Und die Frage ist ob es eine Reise gibt die schaffbar ist.

2.2.2 Das Optimierungsproblem

Da das Entscheidungsproblem zum Handelsreisenden NP-Vollständig ist, gehört auch das entsprechende Optimirungsproblem zur Komplexitätsklasse NPO. Ein Optimierungsproblem ist durch *I*nstanzen, *s*olutions, *m*etrik und *t*arget in einem Quadrupel

$$P = (I, s, m, t)$$

definiert (vgl. [1], S. 128).

Instanzen geben das Format der Eingabe vor.

solution ist eine Funktion, die eine Instanz nimmt und auf die Menge ihrer möglichen Lösungen abbildet. Die Vereinigung aller möglichen Lösungen für alle Instanzen ist die Menge aller überhaupt möglichen Lösungen (wird hier mit S abgekürzt).

metrik: $m : I \times S \to \mathbb{N}$ ist eine die Maßfunktion. $m(x, y)$ ist die Güte der Lösung y mit $y \in s(x)$ und der Instanz $x \in I$.

target: gibt das Ziel des Obtimierungsproblems an. $t \in \{min, max\}$.

Das zu dieser Arbeit wichtige Optimierungsproblem ist das, des minimalen metrischen TSP. Hier wird die Rundreise mit den geringsten Kosten gesucht. Die Instanzen sind die gleichen wie beim TSP, aber ohne die, beim Entscheidungsproblem 2.2 notwendige, Schranke. Metrisch heißt es, weil bei den Instanzen die Symmetrie und das Erfüllen der Dreiecksungleichung vorrausgesetzt wird[4]. Diese Einschränkung lässt, wie wir im Kapitel 3 sehen werden, eine bessere Approximierung zu. Es folgt die Definition des MinMTSP (vgl. [[1]], S.144):

Instanzen: $\left(d_{i,j}\right)_{1 \leq i,j \leq n}$ mit $d_{i,j}, n \in \mathbb{N}$ und $i, j, k \in \{1, \ldots, k\}$ die sowohl die Symmetrie als auch die Dreiecksungleichung erfüllen.

solution: $\pi \in S_n$

[4] Da der euclidische Abstand 2.1 genau das erfüllt wird er hier hauptsächlich genutzt.

metrik $m : I \times S \to \mathbb{N}$ ist eine die Maßfunktion. $m(x, y)$ ist die Güte der Lösung y mit $y \in s(x)$ und der Instanz $x \in I$.

target gibt das Ziel des Optimierungsproblems an. $t \in \{min, max\}$.

2.2.3 Approximationsalgorithmen

Da wir nun Entscheidungs- und Optimierungsprobleme kennengelernt haben, können wir anfangen uns damit zu beschäftigen, die Qualität der Algorithmen miteinander zu vergleichen. Wir definieren einen Approximationsalgorithmus A zu einem Optimierungsproblem $P = (I, s, m, t)$, als ein Algorithmus, mit polynomialer Laufzeit, für welches A eine mögliche Lösung zu jeder Instanz zurückgibt.(vgl. [[1]], S.134). Nun möchten wir einen Indikator, der zeigt, wie gut A das Ziel von P erfüllt. Sei $x \in I$, $y \in s^*(x)$ eine optimalen Lösungen zu x und $z = A(x)$ die von A produzierte Lösung, dann ist die *Performanz(-rate)* mit der A das Ziel t_P erfüllt wie folgt definiert:

$$R_A(x, y) = \begin{cases} \dfrac{m(z)}{m(y)}, & \text{wenn } t_P = min \\ \dfrac{m(y)}{m(z)}, & \text{wenn } t_P = max \end{cases}$$

Insbesondere hießt das $R(x, y) = 1$ gdw. $y \in s^*(x)$.

3

Verwendete Approximationsalgorithmen

Ich werde mich für die Approximationsalgorithmen des MinMTSP beschränken. Das MinMTSP wurde in 2.2.2 bereits veranschaulicht. Dieses Optimierungsproblem wird durch die Symmetrie und Dreiecksungleichung der induzierten Metrik, besser zu approximieren sein. Speziell handelt es sich hierbei um c-Approximationsalgorithmen. Diese Algorithmen garantieren eine konstante Güte, die im schlechtesten Fall c mal schlechter ist als eine optimale Lösung. Zunächst werden wir in 3.1 die einfachere Methode kennenlernen, die uns schon eine 2-Approximierung des MinMTSP bietet. Anschließend werden wir die Idee dieser Methode aufgreifen und in 3.2 einen 1.5-Approximationsalgorithmus aufzeigen.

3.1 Minimum Spanning Tree-Heuristik

Die *M*inimum *S*panning *T*ree-Heuristik (abgekürzt MST-Heuristik, bzw. von [1] auch TreeTSP gennant) ist ein heuristisches Verfahren, das darauf basiert, dass ein minimaler Spannbaum die Kostengünstigsten Kanten enthält, um alle Knoten miteinander zu verbinden. Um eine Rundreise zu bekommen muss in diesem Spannbaum noch ein Pfad gefunden werden, der idealerweise keine Kanten doppelt enthält. Daher wird ein Eulerkreis in dem Spannbaum gesucht. Um einen Eulerkreis zu konstruieren wird die Kantenmenge verdoppelt, sodass jede Kante genau zwei Mal vorhanden ist. Der Grad jedes Knotens ist grade und nach der Definition aus 2.1 existiert ein Eulerkreis. Den Eulerkreis finden wir mithilfe einer Tiefensuche, wobei jedes mehrfache vorkommen der Knoten gestrichen wird. Die Rendite ist eine Rundreise, welche maximal doppelt so schlecht ist wie die optimale

Rundreise.

Das TreeTSP ist ein 2-Approximierungsalgorithmus. Das folgt direkt aus dem zweiten Schritt des Algorithmus und der Dreiecksungleichung die durch die Metrik induziert wird. Beim Überspringen, bzw. Streichen der doppelten Knoten, muss durch die Dreiecksungleichung angenommen werden, dass der Weg nicht größer werden kann [1]. Diesen Engpass versucht der Algorithmus von Chrisofides zu lösen.

Dieser soeben beschriebene Ablauf, entspricht dem Algorithmus 3.1., darauffolgend, dessen Laufzeitanalyse in Tabelle 3.1.

Algorithmus 3.1: TreeTSP

Eingabe: Graph $G = (V, E, w)$ als Instanz von MinMTSP

1: Berechne einen minimalen Spannbaum $T = (V, E_T)$ in G.
2: $M = \left(V, E'_T\right)$ als Multigraph in dem jede Kante E_T genau zweimal vorkommt.
3: Starte von einem Zufälligen Knoten eine Tiefensuche in M, behalte dabei nur das erste Vorkommen jedes Knotens mit $W = (v_1, v_2, \ldots, v_1)$.

Ausgabe: Rundreise $(v_1, v_2, \ldots, v_1)$

Schritt	Laufzeitkomplexität
1	$O(n^2)$
2	$O(n)$
3	$O(n)$

Tabelle 3.1: Laufzeitanalyse des Algorithmus 3.1 mit $n = |V|$

Die Laufzeit der Prozedur ergibt sich direkt aus dem Schritt mit dem schnellst wachsenden Polynom; das Erstellen des minimalen Spannbaums. Hier wird Prims Algorithmus verwendet mit einer Adjazenzmatrix. Dadurch muss immer auf alle Knoten zugegriffen werden. Es resultiert eine Laufzeit von $O\left(n^2\right)$. Die Laufzeit lässt sich verbessern indem statt einer Adjazenzmatrix eine Prioritätswarteschlange genutzt wird. Nach [3] fällt der Algorithmus dann unter die Laufzeitkomplexitätsklasse $O\left(|K| \times \log\left(|K|\right)\right)$

[1] Für einen formellen Beweis verweise ich auf [1] S. 147 f.

3.2 Algorithmus von Christofides

Lange Zeit wurde geglaubt, dass es keinen Algorithmus mit polynomialer Laufzeit geben könne, der besser als ein 2-Approximationsalgorithmus ist. Bis Nicolas Christofides 1976 den Algorithmus 3.2 vorstellte (vgl. [4]). Dieses als Christofides Algorithmus bekannt gewordenes Verfahren ist für das MinMTSP eine 1,5-Approximierung. Für den formellen Beweis verweise ich wieder auf [1] (S. 140).

Grundsächlich unterscheidet sich Christofides Algorithmus nur in einem Schritt zum TreeTSP (Algorithmus 3.1). In dem dritten Schritt wird für die im Schritt zuvor ausgewählten Knoten, ein perfektes Matching gefunden. Die Anzahl der Knoten mit ungeradem Grad muss gerade sein. Das folgt daraus, dass die Summe der Grade für jeden Knoten gerade ist, denn wir zählen jede Kante doppelt. Da die Anzahl der Knoten mit ungeradem Grad gerade ist, gibt es immer ein perfektes Matching. Nachdem die Kantenmenge des Matchings, der Kantenmenge des Spannbaum hinzugefügt wurde, hat jeder Knoten einen geraden Grad, da diese Knoten mit genau einer weiteren Kante verbunden wurden. Nach 2.1 existiert also ein Eulerkreis. Wie auch zuvor konstruieren wir den Eulerkreis, indem wir eine Tiefensuche durchführen und nur das erste Vorkommen der Knoten behalten. Der entstehende Pfad ist unsere Rundreise.

Algorithmus 3.2: Algorithmus von Christofides

Eingabe: Graph $G = (V, E, w)$ als Instanz von MinMTSP

1: Berechne einen Minimalen Spannbaum $T = (V, E_T)$ in G.
2: Ermittle alle Knoten C mit ungradem Grad in T, dann ist $G' = (C, (C \times C))$ der vollständige Graph mit der Knotenmenge C.
3: Finde ein perfektes Matching H mit minimalem Gewicht in G'.
4: $M = \left(V, E'_T \cup H\right)$ als Multigraph indem jede Kante $E_T \cup H$ genau zweimal vorkommt.
5: Starte von einem Zufälligen Knoten eine Tiefensuche in M, behalte dabei nur das erste Vorkommen jedes Knotens mit $W = (v_1, v_2, \ldots, v_1)$.

Ausgabe: Rundreise $(v_1, v_2, \ldots, v_1)$

Die Tabelle 3.2 zeigt die Laufzeitanalyse der einzelnen Schritte. Besondere Aufmerksamkeit verdient der dritte Schritt. Hier beziehe ich mich auf Edmonds Blossom Algorithmus.[2] Der Algorithmus ist mit Laufzeit mit $O\left(mn^2\right)$ abzuschätzen

[2]Mittlerweile gibt es Algorithmen, die auf dem Blossom Algorithmus aufbauen und eine bessere Laufzeit schaffen. Eines davon ist der sogenannte Blossom V Algorithmus von [[5]].

Schritt	Laufzeitkomplexität
1	$O(n^2)$
2	$O(n)$
3	$O(mn^2)$
4	$O(1)$
5	$O(n)$

Tabelle 3.2: Laufzeitanalyse des Algorithmus 3.1, mit $n = |E|$ und $m = |K|$

([2] S. 11 ff.). Es ist eindeutig zu sehen, dass dieser Teil der zeitkritische Schritt in dem Algorithmus von Christofides ist und daher in Zeit $O\left(mn^2\right)$ läuft.

4

Die Implementierung

In diesem Kapitel geht es um die Umsetzung der, in den letzten Kapitel beschriebenen, Approximationsalgorithmen in ausführbarem Programmcode. Das Ziel dabei ist nicht einzig die Umsetzung, sondern ebenfalls das Vermitteln eines intuitiven Verständnisses für die Algorithmen mithilfe von Visualisierungen. Für weitere Informationen zur konkreten Umsetzung der Approximationsalgorithmen verweise ich auf den Anhang A. In diesem Kapitel werde ich hingegen meine Überlegungen im Verlauf von verschiedenen Entscheidungen im Implementierungsprozess festhalten.

Hierzu widme ich mich in 4.1 (S. 14) meinem Entscheidungsprozess für die Programmiersprache, wo ich auch näher erläutern werde wieso ich mich für Java entschieden habe. Anschließend werde ich in 4.2 (S. 16) auf die resultierende Struktur des Programmes eingehen und dazu, unter anderem, ein UML-Klassendiagramm visualisieren und beschreiben. Alle Informationen fließen dann in 4.3 (S. 20) zusammen, sodass ein ganzheitliches Verständnis des Programmes geschaffen wird.

4.1 Java als Programmiersprache

In diesem Teil der Arbeit befasse ich mich mit den Anforderungen, die ich an die Programmiersprache gestellt habe. Im Rahmen der vorliegenden Arbeit wird eine Veranschaulichung von diversen abstrakten Algorithmen erzeugt. Dafür werden wir zunächst die Anforderungen des Projekts im Allgemeinen begutachten.

Zu Anfang ist es wichtig umfassende Bibliotheken der Graphischen Benutzeroberflächen[1] nutzen zu können. Diese werden die Oberfläche darstellen, aber auch

[1] Auch Graphical User Interface bzw. GUI genannt.

die Benutzeraktionen einfangen. Java bietet für diesen Zweck eine sehr ausgereifte Bibliothek, die sich im Verlauf der Entwicklung nützlich gemacht hat. Auch wenn Java mit der JavaFX Bibliothek über viele Steuerelemente der GUI verfügt, so ist es ebenso unkompliziert eigene Steuerelemente zu implementieren. Diese individuelle Anpassung ist insbesondere für die vorliegende Arbeit wichtig geworden.

Doch viele Programmiersprachen bieten heutzutage solche Möglichkeiten und Bibliotheken. Eine weitere Stärke von Java als Programmiersprache ist, dass es auf allen möglichen Geräten ausführbar ist. Der Code wird für die Java virtuel machine (kurz JVM) übersetzt und nicht für den einen Rechner oder Rechnertypen. Ein Javaprogramm läuft also nicht direkt auf dem Gerät sondern in einer virutellen Maschine, die wie ein „Dolmetscher“ funktioniert. Das bedeutet, dass diese JVM Befehle von dem Programm nimmt und sie in, für den Rechner verständliche, Signale übersetzt. In umgekehrter Richtung werden auch die Signale des Rechners von der JVM übersetzt und dem Programm übergeben. Java ist für alle möglichen Systeme verfügbar. Darunter fallen unter anderem auch Windows, Linux und MacOS. Dies ist wichtig, weil die resultierende Software, als Lehranwendung, auf möglichst vielen Geräten verfügbar sein soll. Je weniger Voraussetzungen ein Rechner erfüllen muss, um das Programm ausführen zu können, desto besser ist es, da dann mehr Menschen diese auch nutzen können.

Doch selbst die Unabhängigkeit der Plattform ist nicht einzigartig für Java. Diesen Vorteil machen sich immer mehr und mehr Programmiersprachen zu Nutze. Javascript oder Python, um nur wenige zu nennen, können ebenfalls plattformübergreifend programmiert werden. Der wohl größte Vorteil von Java ist ein Gleichgewicht aus Simplizität der Programmierung und dennoch hoher Kontrolle für Entwickler. Zum Beispiel prüft die Programmiersprache „C“ zur Laufzeit weder Typ- noch Speichersicherheit. Das ist für diese Programmiersprache sowohl Fluch als auch Segen zugleich. Einerseits Produziert die Sprache Ressourcen effizientere Programme, da kein Overhead entsteht, doch macht es sie für die Programmierung umständlicher. Java auf der anderen Seite lässt diese kritischen Operationen nicht zu. Das macht Java zur praktikablen und zweckdienlichen Programmiersprache für plattformübergreifende Programme. Ich habe mich daher bereits in der Vorbereitungsphase für Java als Programmiersprache entschieden.

4.2 Programm Design

In diesem Abschnitt werde ich auf die Programmstruktur eingehen. Hauptsächlich wird das Model-View-Controller Designpattern als Vorlage genutzt. Das bedeutet, dass das Programm in drei größtenteils voneinander unabhängige Strukturen unterteilt wird.

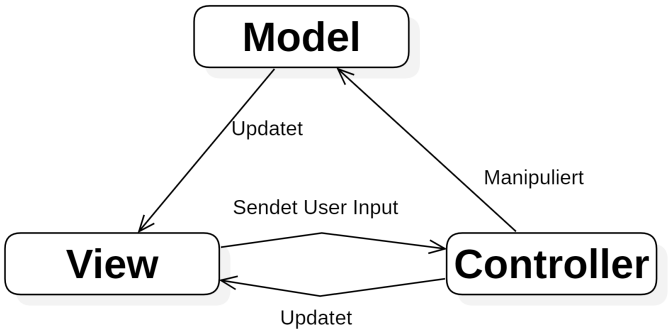

Abbildung 4.1: Das Model-View-Controller Designpattern

Die Abbildung 4.1 stellt schematisch dar, wie die drei Strukturen sich untereinander beeinflussen. Die *View* kümmert sich um die Interaktionen mit dem Nutzer. Insbesondere bedeutet das, die Eingaben abzufangen und Ausgaben darzustellen. Die View gibt die Nutzereingaben an den Controller, nimmt gegebenenfalls neue Daten aus dem Modell und lässt sie dem Nutzer darstellen. Der *Controller* ist das Rechenzentrum in diesem Pattern. Es implementiert alle Funktionen die gebraucht werden, um den Funktionsumfang zu erreichen. Die Ergebnisse können direkt oder indirekt in dem Modell festgehalten werden. Das Bedeutet, entweder werden die Rohdaten gespeichert oder führen zu einem Zustandswechsel im Model. Das *Model* lässt sich als eine zentrale Speicherschnittstelle vorstellen, ähnlich einer Datenbank. Alle problemrelevanten Daten und Zustände werden in der Regel im Model festgehalten. Das Model kann ein Update in der View hervorrufen.

Dieses Strukturmuster wurde entwickelt, um einen flexiblen Programmentwurf zu ermöglichen. Damit sind spätere Änderungen sowie Erweiterungen mühelos umsetzbar. Hierzu das folgende Beispiel: wir nehmen an, dass nicht mehr die euklidische Distanz als Abstand zwischen zwei Punkten genutzt werden soll. Es soll eine beliebige andere Metrik genutzt werden. Dann ist es recht einfach die kritische

Stelle im Code umzuschreiben. Der Controller implementiert die Logik der Software und damit auch wie die Abstände ermittelt werden. Wir schreiben die genutzte Abstandsfunktion um und das Modell bekommt diese anders berechneten Werte. Das Model kümmert es nicht weiter mit welchen Werten gerechnet wird, weil es lediglich die Abstände speichert. Für die View sind die konkreten Werte irrelevant, denn sie muss bloß wissen wie sie die Werte auf dem Bildschirm darstellen kann. Für die vorliegende Arbeit bedeutet das, dass der Funktionsumfang jederzeit mit wenig Aufwand erweitert oder bestehende Funktionen geändert werden können, um weitere Probleme zu lösen und darzustellen.

4.2.1 Verwendete Bibliotheken

Um einen möglichst hohen Kontrollgrad zu bekommen habe ich mich entscheiden nur wenige Fremdbibliotheken zu nutzen. Hauptsächlich wurden die Standardbibliotheken aus dem Java Development Kid der Version 12 von Oracle genutzt. Außerdem wurde das zugehörige Open Souce „JavaFX“ Framwork in der Version 12 benutzt. Das JavaFX Framework wurde für alle Benutzeroberflächen und Steuerelemente verwendet. Es war auch sehr hilfreich um die Nutzerereignisse an den Controller weiter zu geben und zu verarbeiten. Für das perfekte Matching in Christofides Algorithmus habe ich die Unterstützung der „JGraphT“ Bibliothek von Barek Naveh zur Hand genommen.[2]

4.2.2 Struktur des Programmes

In dem Folgenden werde ich die Struktur des Programmes beschreiben. Dazu wird das UML-Klassendiagramm auf Seite 19 als Referenz genommen.

Mit Blick auf die in 4.2 abgebildeten Pakete, wird schnell deutlich, dass sie nach Model-View-Controller benannt und unterteilt wurden. In dem Controllerpaket finden sich die Klassen, die zur Steuerung und Berechnung der Software nötig sind. Dabei beschränkt sich die Klasse *MainStageController* darauf die Benutzeraktionen aus der View zu bekommen und den Programmablauf zu koordinieren. Die statische Klasse *Util*, im gleichnamigen Paket, hingegen implementiert alle Funktionen zum Lösen der TSP Instanzen.

Das Paket Model modelliert die, zum Lösen des TSP, nötigen Datenstrukturen.

[2]Die Homepage der Bibliothek ist unter „https://jgrapht.org“ zu erreichen.

Primär werden dazu die Klassen *Graph* und *City* genutzt. Die *GraphState*-Klasse in diesem Paket hält die Zustandsveränderungen eines Graphen fest.

Die *MapView*-Klasse des *View* Pakets legt fest, wie die Graphen dargestellt und visualisiert werden sollen. Alle weiteren Steuerelemente des Programmes sind in der Datei *MainStage.fxml* festgehalten und von JavaFX geladen.

Vergleichen wir nun das in Abbildung 4.1 dargestellte Schema des Model-View-Controller mit dem UML-Klassendiagramm in Abbildung 4.2, wird deutlich, dass sie sich viele strukturelle Merkmale teilen. So ist die *Graph*-Klasse, die Klasse, die an dem die problemrelevanten Daten zentral zur Verfügung gestellt werden. Der Graph modelliert Punkte über die *City*-Klasse. Sie wird als Wrapper benutzt, um auf die Koordinaten zuzugreifen. Kanten zwischen den City-Objekten sind in der Adjazenzmatrix festgehalten. Außerdem hat jeder Graph eine Liste von *GraphState*s, die Zustandsübergänge darstellen sollen. Jede Modifizierung soll hier festgehalten werden, um für eine spätere Darstellung bereit zu stehen. Wie diese Darstellung aussieht, liegt an der Implementierung der *MapView*. Die Klassen *MapView* und *MainStageController* greifen dazu auf dieselbe Instanz eines *Graph*en zu. Die Klasse des View-Paketes wird, im Gegensatz zur Controller-Klasse, aus dem Modell ausschließlich lesen. Die *MapView* liest die Koordinaten aus der *City*-Klasse und rechnet sie in Koordinaten um, die auf dem Bildschirm darstellbar sind. Die *MapView* selbst ist passiv und reagiert nur auf Updateanweisungen des Controllers. Diese Anweisung wird beim Klick auf einem der Navigationsleisteknöpfe gefeuert. Die Berechnungen auf dem Graphen übernimmt aber die *Util*-Klasse aus dem gleichnamigen Packet im Controllerpaket. In Abbildung 4.2 ist in dem Controllerpaket, neben den bereits erwähnten Klassen, eine weitere Klasse zu sehen: die *DrawLoop*-Klasse. Sie implementiert ein *Runnable*[3]. Diese Klasse sollte insbesondere für die Implementierung eines Play-Buttons dienen. Es würde in regelmäßigen Abständen ein Update der View anstoßen[4].

Neben den Gemeinsamkeiten ist der wohl größte Unterschied das Verhältnis von View zu Model. In der Implementierung löst wie schon erwähnt das Model kein Update auf der View aus. Diese Updates können nur von dem Controller kommen.

[3]Runnable ist eine abstrackte Klasse aus der Java Standardbibliothek. Sie wird hauptsächlich benutzt, um Aufgaben auf Threads zu verlagern. Daher sieht man diese Klasse oft in Kombination mit der *Thread*-Klasse, die ebenfalls aus Javas Standardbibliothek stammt. Tatsächlich kann ein Thread mit einem Runnable als Parameter erstellt werden. Für weitere Informationen dazu möchte ich auf die Documentation der Standard Java Bibliotheken hinweisen.

[4]Warum dieses Feature nicht voll funktionsfähig ist und welche Probleme sich ergeben haben wird im Kapitel 5 näher beschrieben.

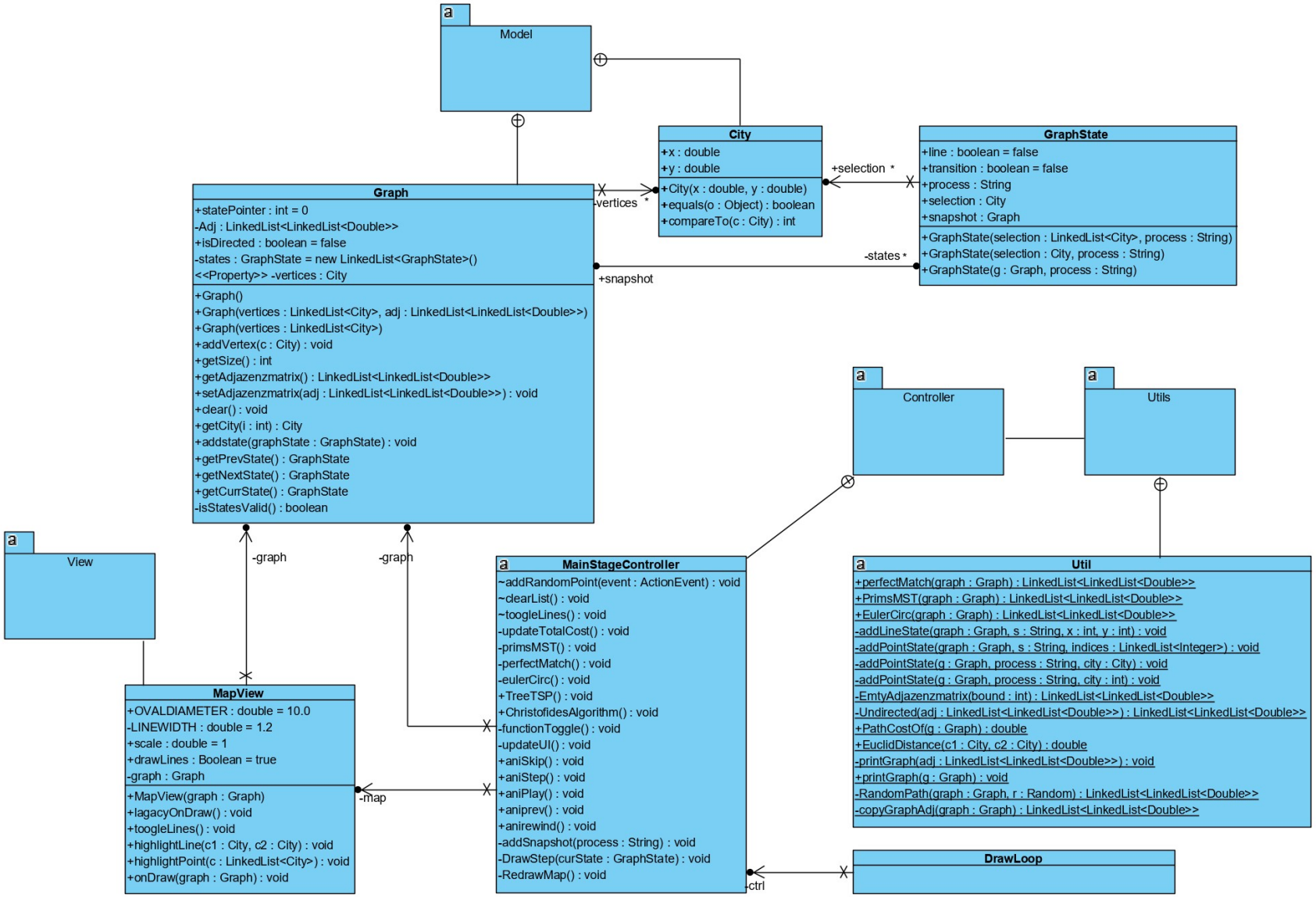

Abbildung 4.2: UML Klassendiagramm der Packete Model, View und Controller

4.3 Verwendung des Programmes

4.3.1 Anwendungsanleitung

Beim Start des Programmes öffnet sich die Hauptfensteransicht (siehe Abbildung 4.3).

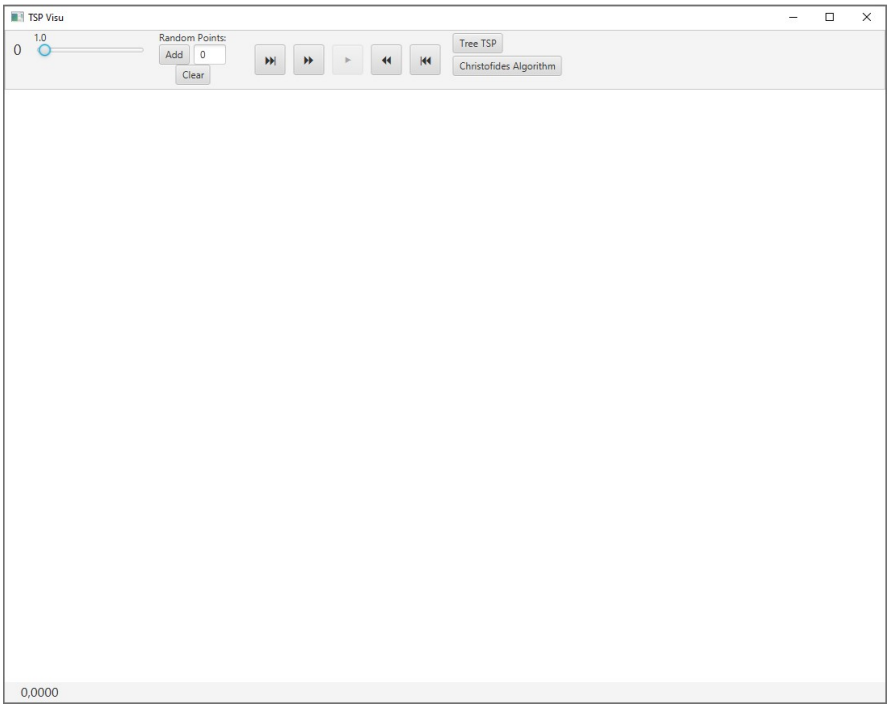

Abbildung 4.3: Initiales Programmfenster

Alle Steuerelemente sind in der oberen grauen Leiste platziert (siehe Abbildung 4.4).

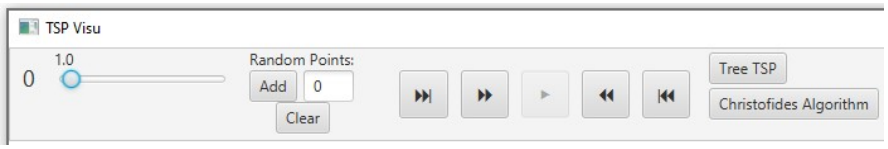

Abbildung 4.4: Initiales Controll Panel

Beginnend links bei dem Knotenzähler, wird jedes der Elemente im Folgenden beschrieben. Für ein etwas anschaulicheres Beispiel verweise ich auf Abschnitt 4.3.2 auf Seite 24.

Das in Abbildung 4.5 hervorgehobenen Element, ist der Knotenzähler. Dieser Zähler gibt, an wie viele Knoten in dem Graphen zu sehen sind.

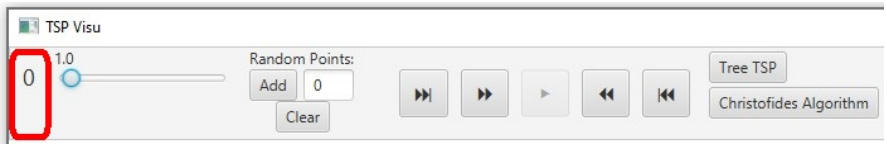

Abbildung 4.5: Knotenzähler

Rechts neben dem Knotenzähler befindet sich der Schieberegeler (siehe Abbildung 4.6). Dieser Regler gibt vor wie groß der Graph auf der Oberfläche darzustellen ist. Essentiell lässt sich dieses Feature wie einen Zoom vorstellen. Zum Vergrößern wird der Schieberegler mit der Maus auf die linke- und zum verkleinern auf die rechte Seite gezogen. Dabei ist von einer Halbierung bis zu einer Verzehnfachung der Anzeigegröße alles möglich. Das Label drüber zeigt den momentan verwendeten Zoomstatus an.

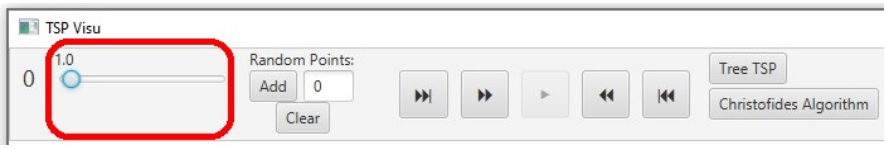

Abbildung 4.6: Schieberegeler zur Vergrößerung/Verkleinerung der Anzeige

Als nächstes sehen wir, in Abbildung 4.7, Elemente zum Hinzufügen von Punkten in dem Graphen, bzw. zum Löschen des Graphen. Drückt man auf den “Add„-Button wird die, in der Textbox stehende Anzahl von Knoten dem Graphen hinzugefügt. Jeder dieser Knoten wird an einer zufälligen Stelle platziert. Alternativ zum Add-Button kann die Eingabe auch mit der Enter-Taste auf der Tastatur bestätigt werden. Mit dem Betätigen der Taste „Clear“ kann der momentane Graph gelöscht werden. Dieser Knopf entsperrt außerdem auch wieder die Oberfläche, falls sie zuvor gesperrt wurde. Mehr zum Sperrmechanismus folgt im weiteren Verlauf.

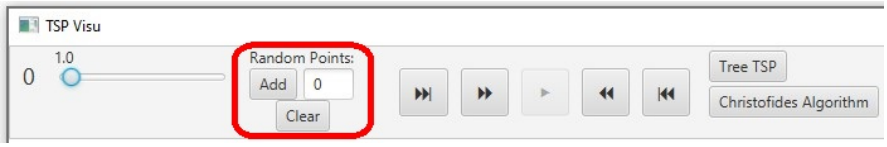

Abbildung 4.7: Elemente um Knoten im Graphen Hinzuzufügen bzw. den Graphen zu Löschen

Rechts als nächstes kommen die Animationssteuerelemente (siehe Abbildung 4.8). Mit diesen Knöpfen lassen sich die Schritte in der Animation steuern. Zunächst haben wir den Button zum Vorspulen, wodurch zum nächsten Abschnitt der Animation vorgespult werden kann. Dann kommt der Button um einen einzelnen Schritt in der Animation weiter zu gehen. In der Mitte der Komponente ist der ausgegraute Playbutton. Dieser Button ist ausgegraut, weil dadurch eventuell einige unbeabsichtigte Nebeneffekte entstehen können. Beim Drücken dieses Knopfes kann ein reibungsloser Programmablauf nicht garantiert werden. Neben diesem Knopf zu finden, sind die beiden Knöpfe um Schritte in der Animation zurück zu spulen. Der linke der beiden Knöpfe spult einen einzelnen Schritt zurück und der rechte spult zurück bis zum Start des vorherigen Abschnitts.

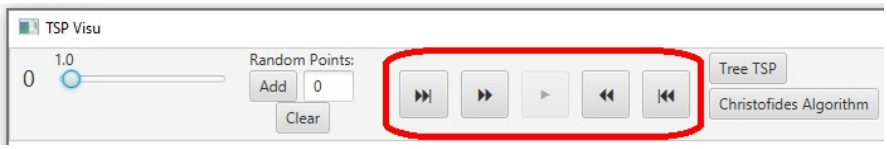

Abbildung 4.8: Die Animationssteuerelemente

Zuletzt haben wir in dem Controllpanel, die in Abbildung 4.9 hervorgehobenen Knöpfe. Diese Knöpfe implementieren die Approximationsalgorithmen zum Lösen des MinMTSP auf dem Graphen. Die entsprechenden Animationen werden hinzugefügt und können über die Animationssteuerelementen zugegriffen werden. Durch das Drücken auf einem dieser Knöpfe wird das User Interface teilweise gesperrt. Das bedeutet, es ist nicht mehr möglich Aktionen auszuführen, die den Graphen manipulieren. Die einzige Ausnahme ist das Löschen des Graphen.

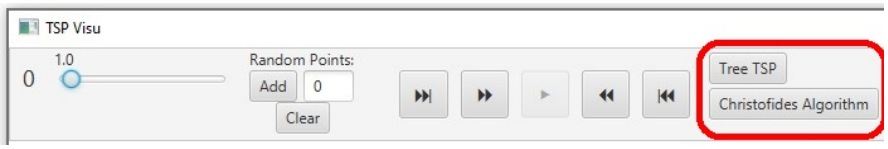

Abbildung 4.9: Knöpfe der Approximationsalgorithmen

Durch Drücken auf die Taste “F5„ zeigt sich ein, in 4.10 abgebildetes, verstecktes Menü. Dieses Menü ist zu Testzwecken entwickelt und es wird davon abgeraten die Funktionen einzeln zu verwenden, da sie zu unerwünschtem Verhalten führen können. Der Knopf „Random Point“ fügt einen einzelnen Punkt an einer zufälligen stelle an dem Graphen hinzu. Das funktioniert auch wenn der Graph gesperrt ist und kann zu nicht erwünschten Verhalten, Fehlern und Abstürzen führen. Der Knopf „Hide Lines“ verhindert das Zeichnen der Kanten zwischen den Punkten auf dem Graphen. Speziell, wenn der Graph viele Knoten und damit noch mehr Kanten hat, ist dieser Knopf enorm hilfreich. Dieser Knopf verhindert auch das Zeichnen der Animationskanten. Bei den letzten drei Knöpfen wird die Funktion ausgeführt, mit der sie benannt wurden.

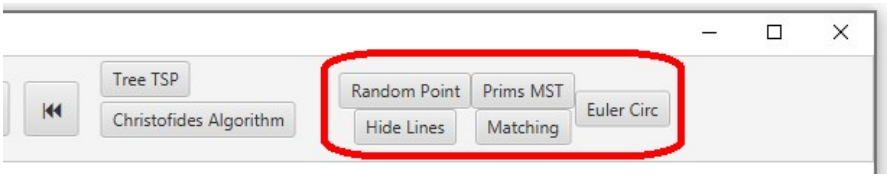

Abbildung 4.10: Verstecktes Entwickler Menu

Als Nächstes möchte ich über den Körper der Benutzeroberfläche, die Graphdarstellungskomponente, reden. Nach dem Starten des Programmes ist es das weiße Rechteck im Zentrum des Programmes, siehe Abbildung 4.11. Auf dieser Fläche wird der Graph und alle Animationen auf dem Graphen gezeichnet. Anhand eines Klicks an einer Stelle auf der Fläche wird dem Graphen an dieser Stelle ein Punkt hinzugefügt. Diese Funktion kann ebenfalls, bei Bedarf, gesperrt werden.

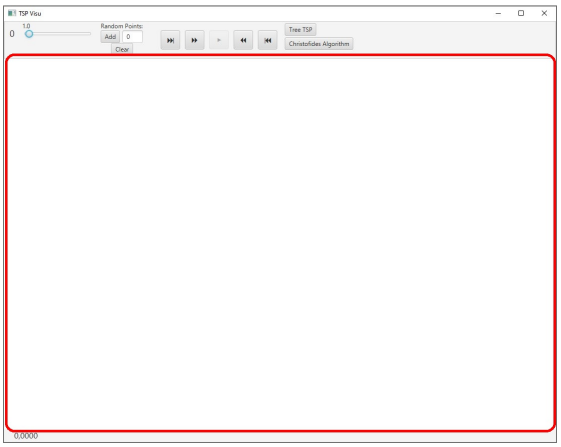

Abbildung 4.11: die Graphdarstelungskomponente

Im Fenster, in der Ecke links unten, befindet sich das letzte Benutzoberflächenelement. Dieses Element wird in Abbildung 4.12 noch einmal veranschaulicht. Dieses Element zeigt die Gesamtkosten des finalen Graphen. Wurde noch keines der Algorithmus aus der in Abbildung 4.9 gestartet, zeigt dieses Element die Summe aller Kantengewichte an. Die Gesamtkosten entsprechen in diesem Fall der Summe der Gewichte aller im Graphen enthaltenen Kanten.

Abbildung 4.12: Gesamtkosten des Graphens Label

4.3.2 Use Case

Kommen wir nun zum Verwendungsbeispiel. Als Beispiel wird hier gezeigt, wie man in dem Programm einen Graphen mit 15 Punkten erstellt. Zehn der Punkte werden zufällig generiert und fünf werden manuell platziert. Jeweils an den Ecken einer und den letzten in der Mitte der Graphenfläche. Wurden die Knoten hinzugefügt, möchten wir die Darstellungsgröße verdoppeln. Als nächstes werden wir eine Rundreise mithilfe des TreeTSP Approximationsalgorithmus errechnen. Jedoch wollen wir nur sehen wie der Spannbaum generiert wird, nicht aber wie der Eulerkreis gefunden wird. Nachdem wir die Rundreise gesehen haben möchten wir

den Graphen löschen und das Programm schließen. Als Voraussetzung nehme ich an, dass das Programm schon gestartet ist.

Wir beginnen damit, das weiße Textfeld auf der Kontrollleiste anzuklicken und schreiben eine numerische zehn hinein, so wie in Abbildung 4.13 zu sehen ist. Wir drücken die „Enter“-Taste auf der Tastatur.

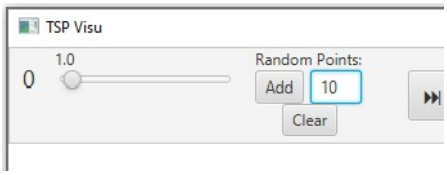

Abbildung 4.13: Hinzufügen von 10 zufällig verteilten Punkten

In dem Zentrum des Programmes sind zehn zufällig verteilte Punkte erschienen und der Knotenzähler zeigt „10„ an. Jetzt klicken wir zunächst die vier Ecken, dann die Mitte, des weißen Rechtecks im Zentrum des Programmes an. Der Knotenzähler zeigt eine „15“ an. Die Gesamtkosten sind auch größer Null geworden. Als nächstes werden wir die Anzeigegröße verdoppeln. Dazu halten wir den Schieberegler geklickt und ziehen ihn solange nach rechts, bis auf dem Label darüber „2.0“ steht. Alternativ zum Drücken und Halten, kann man den runden Knopf des Schiebereglers anklicken und dann mit den Pfeiltasten der Tastatur in die entsprechende Richtung ziehen. Abbildung 4.14 zeigt wie es nun aussehen kann.

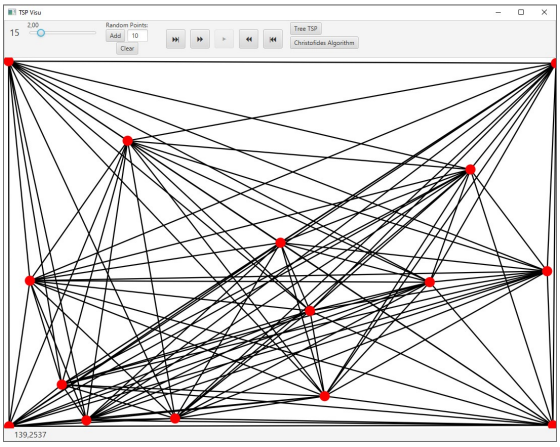

Abbildung 4.14: 15 Punkte wurden hinzugefügt und die Anzeigegröße verdoppelt

Wie bereits eingangs erwähnt, möchten wir den TreeTSP verwenden, um eine

Rundreise in dem Graphen zu finden. Dazu Klicken wir in dem Bereich der Approximationsalgorithmen auf den Button „TreeTSP“. Zu beobachten ist zunächst, dass die Knöpfe der Approximationsalgorithmen und der „Add“ Knopf ausgegraut sind. Klickt man auf die Graphenfläche, passiert nichts. Die Gesamtkosten in der unteren linken Ecke haben sich signifikant verringert. Das sind schon die Kosten der resultierenden Rundreise. Da wir uns die Konstruktion des Spannbaums ansehen wollten, werden wir diesen Schritt für Schritt anzeigen lassen. Dazu drücken wir den Doppelpfeil ohne den waagerechten Strich dahinter, das ist der zweite Button von links von den Animationssteuerelementen. Schon beim ersten Mal Klicken, stellt man fest, dass ein Text über den Knöpfen erscheint. Dieser Text gibt Auskunft darüber, wo wir uns im Prozess befinden. Beim Durchklicken werden manche Punkte erst gelb, dann grün markiert. Und die Kanten zwischen den grünen Knoten werden blau markiert. Das ist die Konstruktion des Spannbaums. Dabei bedeutet gelb, dass dieser Knoten bereits entdeckt wurde und grün das dieser der Menge der Knoten aus dem Spannbaum hinzugefügt wurde. Blaue Kanten sind ebenfalls Kanten, die dem minimalen Spannbaum hinzugefügt wurden sind. Das könnte in etwa so aussehen, wie in Abbildung 4.15

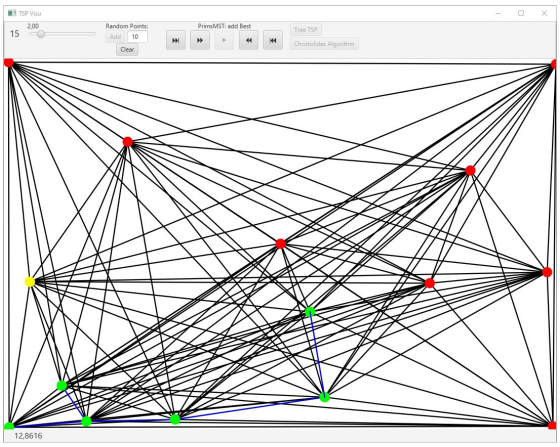

Abbildung 4.15: Hervorhebung bei der Konstruktion eines minimalen Spannbaums

Wir könnten uns jeden Schritt, der zur dem minimalen Spannbaum führt, einzeln anzeigen lassen, indem wir den selben Knopf wiederholt betätigen. Wenn wir dies tun, werden nach einer Weile die bunten Markierungen im Graphen verschwinden und die Knoten sind wieder rot bzw. die Kanten schwarz. Wenn das passiert wird dies auch im Prozesstext über den Animationssteuerelementen, durch einen

veränderten Prozesstitel, wiedergegeben. Abbildung 4.16 gibt den beschrieben Zustand wieder.

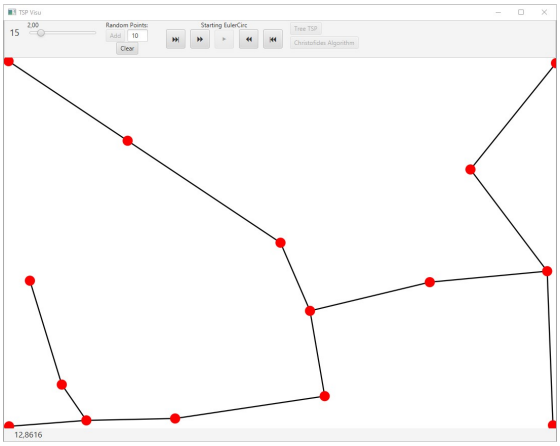

Abbildung 4.16: Programmzustand nach dem ersten Teil des Algorithmus

Wenn wir wollten, könnten wir uns die Berechnungsschritte noch einmal anschauen. Aber wir entscheiden uns dafür, an das Ende der Berechnungen vorzuspulen und die resultierende Rundreise zu sehen. Mit dem Betätigen des linken Knopfs des Animationssteuerelements, gelangen wir direkt ans Ende. Wir sehen nun die entstandene Rundreise aus dem initialen Graphen. Die Summe über die Kantengewichte ist unten links zu lesen, sie hat sich während der einzelnen Schritte nicht geändert. Auch an diesem Zeitpunkt hätten wir die Chance nochmal an den Anfang zu spulen und uns jeden der Schritte wiederholt darstellen zu lassen. Aber wir Drücken den „Clear“-Knopf und löschen den Graphen. Der Prozesstext über den Animationssteuerelementen gibt „Cleared“ aus. Die zuvor ausgegrauten Knöpfe sind wieder verfügbar. Wir sind im Anfangszustand und könnten wieder von vorne anfangen.

5

Evaluation

In diesem finalen Kapitel werden die wichtigsten Ergebnisse dieser Arbeit rekapituliert und zu den Ergebnissen Stellung genommen.

Essentiell für die vorliegende Arbeit ist es zu verstehen wie die Algorithmen ablaufen und dieses Verständnis dann zu verbildlichen. Doch um zu verstehen was diese Algorithmen bewirken, mussten zunächst einige der Grundlagen erörtert werden. Darunter ist auch die formelle Definition des Traveling Salesperson Problems, inklusive des entsprechenden Optimierungsproblems gefallen. Beim Behandeln der Approximationsalgorithmen ist aufgefallen, dass das Unterproblem des minimum TSP, nämlich das metrische minimum TSP, durch die Metrik besser zu approximieren ist. In diesem Zusammenhang wurden die zwei entscheidenden Algorithmen, die Tree-Heuristik und der Algorithmus von Christofides, dieser Arbeit untersucht. Es verblieb noch zu zeigen, wie die Algorithmen sich in eine Software übersetzt haben lassen. Unter anderem wurde dazu ein Verwendungsbeispiel genannt.

Das Ergebnis ist eine in Java programmierte Software, die das Errechnen von Rundreisen ermöglicht. Graphen können auf verschiedene Weisen erstellt werden. Die Software ermöglicht es die Schritte der Berechnungen einzeln, aber auch grob darzustellen. Eine Voraussetzung für diesen hohen Grad der Kontrolle ist es gewesen die Algorithmen und Strukturen selbst zu implementieren.

Doch, wie in der Softwareentwicklung häufig der Fall, ist auch diese Software nicht frei von unbeabsichtigtem Verhalten. Das wichtigste, nicht funktionstüchtige, Feature ist das automatische Abspielen der Animationen. Auch wenn die Funktion im Quellcode präsent ist und weitestgehend funktioniert, sind noch einige Fehlfunktionen zu beobachten. Unter Umständen entstehen unbeabsichtigte Neben-

läufigkeiten, die das Programm unbenutzbar machen, bis ein Neustart vollzogen wurde. Daher habe ich mich entschieden den entsprechenden Knopf zu dieser Programmfunktion zu deaktivieren. Infolgedessen müssen Benutzer die Schritte manuell durchlaufen und können nicht auf die Bequemlichkeit eines automatischen Ablaufs zurückgreifen. Diese fehlende Funktion wirft zugleich Licht auf einen weiteren Makel. Alle Berechnungen werden in dem selben Prozess und sogar in demselben Prozessfaden durchgeführt. Dieses Design bringt einige Nachteile mit sich. Etwa bei großen Berechnungen, wenn eine Rundreise in einem großen Graphen berechnet wird, kann das Programm für Benutzereingaben blockieren. Grade für Nutzer mit weniger leistungsstarken Rechnern, kann dies schnell Frustration ausrufen.

Doch werfen wir einen Blick auf die andere Seite der Medaille. Denn will man nicht grade eine Rundreise in einem Graphen mit mehreren hunderten Knoten berechnen, geschieht dies schnell. In dem Kapitel 3 (S. 10) wurde gezeigt, dass der TreeTSP Algorthmus in Zeit $O\left(n^2\right)$ liegt. Ein vergleichsweise schnell laufender Algorithmus. Darüber hinaus ist die Kontrolle, über das Anzeigen der Hervorhebungen, sehr gut. Wenn einmal etwas berechnet wurde, ist das Programm extrem ressourcensparend. Auch schwächere Computer sollten das Zeichnen der Hervorhebungen handhaben können.

Von Vorteil ist auch, dass die Struktur der Software erlaubt Erweiterungen einzubauen. Speziell kann eine Auslagerung der rechenintensiven Abschnitte in Threads unkompliziert umgesetzt werden. Selbstverständlich gilt das auch für die Approximationsalgorithmen. Auch wenn die Auswahl in der Hinsicht karg wirkt, können neue Probleme unkompliziert eingearbeitet werden.

Abschließend ist festzuhalten, dass die Algorithmen abstrakt und teilweise nur sehr schwer vorstellbar sind. Selbst wenn sie konzeptionell nachvollziehbar sind. Durch eine entsprechende Visualisierung wird das Verständnis auf ein intuitives Level angehoben, was die kognitive Verarbeitung deutlich vereinfacht.

Quellenverzeichnis

Literatur

[1] H. Vollmer A. Meier. *Komplexität von Algorithmen*. lehmanns media, 2015. Kap. Ein Bausatz von NP-vollständigen Problemen (siehe S. 7–12).

[2] S. Vare A. Shoemaker. *Edmonds' Blossom Algorithm*. Stanford University, CME 323, 6. Juni 2016. URL: https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/shoemaker_vare.pdf (besucht am 11. 11. 2019) (siehe S. 13).

[3] T. Prasad B. Hasan. *lerning algorithm eBook*. riptutorial.com. Kap. 48, S. 247. URL: https://riptutorial.com/de/ebook/algorithm (besucht am 11. 11. 2019) (siehe S. 11).

[4] N. Christofides. *Worst-case analysis of a new heuristic for the travelling salesman problem*. Management Sience Research Report No. 388. Carnegie Mellon University (CMU), Feb. 1976. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a025602.pdf (besucht am 11. 11. 2019) (siehe S. 12).

[5] V. Kolmogorov. „Blossom V: a new implementation of a minimum cost perfect matching algorithm". *Math. Prog. Comp.* 1 (1 2009). Hrsg. von J. Eckstein, S. 43–67. URL: https://doi.org/10.1007/s12532-009-0002-8 (siehe S. 12).

Online-Quellen

[6] Dipl. Ing. Dr. M. Predota. *Königsberger Brückenproblem*. URL: http://finanz.math.tu-graz.ac.at/~predota/old/history/resultate/bruecken.html (besucht am 11. 11. 2019) (siehe S. 5, 6).

Abbildungsverzeichnis

Anhang A

Implementierung der Approximationsalgorithmen

Hier werden die Implementierungen der einzelnen Schritte für die Approximierungsalgorithmen behandelt. Diese Funktionen sind in der statischen Klasse *Util* in dem Gleichnamigen Paket, welches sich im Controllerpackage befindet. Bei der Umsetzung mussten einige Besonderheiten beachtetet und Entscheidungen getroffen werden, die ich hier unter anderem an Codebeispielen veranschauliche.

A.1 Prims minimum spanning tree Algorithmus

Beginnen wir direkt bei der Implementierung von Prims Algorithmus zur Konstruktion eines minimalen Spannbaums aus einem beliebigen Graphen. Die Methode wird mit dem folgenden Funktionsrumpf definiert.

```
93   public static LinkedList<LinkedList<Double>> PrimsMST(Graph graph){
```

Zunächst werden einie Hilfsvariablen declariert. Da wir an einer zufälligen Stelle die Kontruktion beginnen, wird ein Zufälliger Knoten der Liste der besuchten Knoten hinzugefügt. Die Liste der „indexOfVisited“ enthält diese bereits Besuchten Knoten. Es wird aufgrund der folgenden Zeile solange iteriert, bis alle Knoten besucht wurden.

```
103         while(indexOfVisited.size() < graph.getSize()){
104             LinkedList<Integer> listOfBest = new LinkedList<>(),
105                                 verticesToDisplay = new LinkedList<>();
```

Der Spannbaum wird erweitert, indem von allen Kanten in denen ein entdeckter und ein unenteckter Knoten enthalten ist, die Kante mit dem geringsten gewicht hinzugefügt wird. Der Knoten der über diese Kante verbunden ist, wird somit auch entdeckt. Dieser Schritt wurde wie Folgt übersetzt.

```
107             for (int i = 0; i < indexOfVisited.size(); i++) {
108
109                 // nehme eine Spalte aus der Adjazenzmatrix
110                 int currNodeIndex = indexOfVisited.get(i);
111                 LinkedList<Double> weightsFromCurr = graph.
        getAdjazenzmatrix().get(currNodeIndex);
112
113                 int nextBest = -1;
114                 // für jeden Besuchten Knoten finde einen neuen mit dem Kleinsten
        gewicht
115                 for(int j = 0; j < weightsFromCurr.size(); j++){
116                   ...
```

Es werden in jedem Schritt für alle entdeckten Knoten alle Verbindungen verglichen und der die Kante mit dem geringsten Kosten wird als Kandidat für die nächste Kante zwischengespeichert. Zu jedem Knoten werden die Kanten aufsteigend ihres Gewichts sortiert[1]. Jeweils die Kante mit dem geringsten Gewicht, wird der Liste „listOfBest“ hinzugefügt. Dies ist die Liste der Kandidaten für die Kante mit dem geringsten Gewicht. Von den Kandidaten wird die Kante mit den geringsten Kosten dem Spannbaum hinzugefügt werde.

```
107             MST.get(indexOfVisited.get(indexOfBest)).set(listOfBest.get(
        indexOfBest), weightOfBest);
108
109             indexOfVisited.add(listOfBest.get(indexOfBest));
```

Da alle weiteren Algorithmen einen ungerichteten Graphen erwarten, der Spannbaum allerdings offensichtlich gerichtet ist, entnehmen wir die Richtung aus den Kanten. Der minimale Spannbaum kann nun Zurückgegeben werde.

A.2 Eulerkreis durch Tiefensuche

Die Erzeugung des Euler Kreises geschieht durch einen Tiefendurchlauf des zusammenhängenden Graphen. Der Algorithmus merkt sich dabei die Reihenfolge der

[1] Dafür wird ein insertion Sortiertalgorithmus benutzt.

Besuchten Knoten. Die Kanteneinträge in der Rückgabeadjazenzmatrix ergeben sich aus der Reihenfolge in der die Knoten Besucht wurden.

Die Funktion wird auf dem Eingabegraphen ausgeführt und gibt einen geschlossenen Eulerweg in Form einer Adjazenzmatrix zurück. Dies ist auch an dem folgenden Funktionsrumpf ausgedrückt.

```
164     public static LinkedList<LinkedList<Double>> EulerCirc(Graph graph){
```

Für eine Tiefensuche, benötigen wir eine Queue. Hier benutzen wir eine doppeltverkettete Liste. Da wir nur Elemente an den Anfang stellen werden, könnte auch eine einfach verkettete Listen verwenden werden. Außerdem wird über besuchte Knoten Buch geführt, weil die Existenz von Kreisen in den Graphen nicht ausgeschlossen ist. Die Liste „visited“ enthält diese besuchten Knoten. Wie schon bei der Kontruktion des minimalen Spannbaums mit Prims Algorithmus, beginnen wir die Tiefensuche an einer zufälligen Stelle.

```
175         // Start from a random Point.
176         int start = (int)(Math.random()*graph.getSize());
177 
178         queue.add(start);
```

Der Algorithmus nimmt das oberste Element der Queue und versucht die mit diesem verbundenen Knoten der Queue hinzuzufügen. Offensichtlich muss dafür eine Kante zwischen den Knoten existieren, aber der Zielknoten darf auch nicht bereits besucht sein. Kanten werden in der Adjazenzmatrix mit einem Gewicht größer Null modelliert. Um die Existenz von Kanten zu prüfen, wird nach Einträgen größer Null prüfen. Dieser Ablauf wird von dem folgende Codeausschnitt umgesetzt.

```
198             // look at the ADJ for curr vertex
199             LinkedList<Double> ConnWeights = adj.get(curr);
200 
201             LinkedList<Integer> ConnCityIndex = new LinkedList<>();
202             // create a orderd List of connected Vert
203             for (int i = 0; i < ConnWeights.size(); i++) {
```

Ich habe mich entscheiden, die neu entdeckten Knoten, vor dem Einfügen in die Queue, zu sortieren. Das oberste Element ist der Knoten, der über über das geringste Gewicht mit dem aktuellen Knoten verbunden ist. Da der Graph Kreise enthalten kann, kann es zur Situationen kommen, dass Knoten öfter entdeckt werden. Wiederholt entdeckte Knoten werden aus ihrer Position in der Queue entfernt und neu an den Anfang der Liste einsortiert.

```
205                 double newweight = ConnWeights.get(i);
206
207                 if (newweight != 0 && !visited.contains(i)){
208                     boolean added = false;
209                     // insertion sort.
210                     for (int j = 0; j < ConnCityIndex.size(); j++) {
211                       if (newweight < ConnWeights.get(ConnCityIndex.get(j)))
      {
212                             ConnCityIndex.add(j, i);
213                             added = true;
214                             break;
215                         }
216                     }
217                     if (!added){
218                         ConnCityIndex.add(i);
219                     }
220
221                     //rediscover node
222                     queue.remove((Integer)i);
223                 }
```

Die Reihenfolge der Knoten aus der „visited“ Liste ergeben den Eulerkreis. Die Einträge in der Adjazenzmatrix werden in jedem Schritt, bevor ein Knoten aus der Queue genommen wird, eingetragen. Die Adjazenzmatrix „EulerCirc“ wird dann das Resultat zurückgegeben.

A.3 Kolmogorov Blossom V Algorithmus zum perfekten Matching

Der Blossom V Algorithmus von Kolomogorov wurde in der vorliegenden Arbeit mithilfe der „JGraphT“ Bibliothek von Barek Naveh [2] implementiert. Dieser Algorithmus findet ein Matching mit minimalen Kosten für einen generellen Graphen. Wenn es ein perfektes Matching gibt, findet dieser Algorithmus auch diesen mit minimalen Kosten. Da das Matching in der vorliegenden Arbeit nur bei Kanten mit ungeradem Grad ausgeführt werden soll, ist dies in der Funktion eingearbeitet.

Auch diese Methode nimmt einen beliebigen Graphen als Eingabe und gibt die Vereinigung der Kanten des Graphen mit dem des perfekten Matchings wieder.

[2] Die Homepage der Bibliothek ist unter „https://jgrapht.org“ zu erreichen.

Der Methodenrumpf sieht folgendermaßen aus:

```
31     public static LinkedList<LinkedList<Double>> perfectMatch(Graph graph){
```

Wie eingangs schon angedeutet, werden zunächst die Knoten mit ungeradem Grad gesucht und in der Liste „vertexWithOddDeg“ festgehalten. Der folgende Code zeigt die Befehlsfolge, die für jeden Knoten die Anzahl der Kanten zählt. Da jede Kante ein Gewicht echt größer Null hat, wird für jeden Knoten gezählt, wie viele Einträge diese Bedingung in der Adjazenzmatrix erfüllen. Diese Anzahl ist somit der Grad des Knotens.

```
38         for (int i = 0; i < perfectMatchAdj.size(); i++) {
39             int deg = 0;
40             for (int j = 0; j < perfectMatchAdj.get(i).size(); j++) {
41                 if (perfectMatchAdj.get(i).get(j) != 0) deg++;
42             }
43 
44             if (deg%2 != 0) {
45                 vertexWithOddDeg.add(i);
46                 matchingGraph.addVertex(graph.getCity(i));
47             }
48         }
```

Da der Blossom Algorithmus wenig zum Verständnis, der in der vorliegenden Arbeit behandelten Approximationen beiträgt und im Vergleich zu den anderen Algorithmen viel aufwändiger ist, habe ich die Unterstützung von Fremdbibliotheken wahrgenommen. Wie eingangs erwähnt wird die „JGraphT“ Bibliothek von Barek Naveh [3] benutzt um das perfekte Matching zu finden. Der Algorithmus von Kolomogorov nimmt in der Bibliothek von Barek Naveh einen seiner selbst implementierten Graphenklassen. Daher übersetzt die Funktion zunächst die Graphenklasse, die zum zweck der vorliegenden Arbeit geschaffen wurde, in eine Graphenklasse von herrn Naveh. In diesem Fall wurde ein Objekt der Klasse „DefaultUndirectedWeightedGraph“ benutzt.

org.jgrapht.Graph<City, DefaultWeightedEdge> matchingGraph = new DefaultUndirectedWeightedGraph<>(DefaultWeightedEdge.class); Anschließend werden die Kanten mit folgendem Code dem matching Graphen hinzugefügt.

```
55 for (int i = 0; i<vertexWithOddDeg.size(); i++){
56             for(int j = i+1; j<vertexWithOddDeg.size(); j++){
57                 if (i ==j) continue;
```

[3] Die Homepage der Bibliothek ist unter „https://jgrapht.org“ zu erreichen.

```
58                City s = graph.getCity(vertexWithOddDeg.get(i));
59                City t = graph.getCity(vertexWithOddDeg.get(j));
60                double dist = EuclidDistance(s,t);
61
62                DefaultWeightedEdge e = matchingGraph.addEdge(s,t);
63                matchingGraph.setEdgeWeight(e, dist);
64            }
65        }
```

Nachdem die Kanten hinzugefügt wurden, kann der Blossom V Algorithmus ausgeführt werden Anschließend werden die Kantenmengen der Graphen vereinigt. Der folgende Codeabschnitt enthält die Anweisungen dafür.

```
31        MatchingAlgorithm pm = new KolmogorovWeightedPerfectMatching(
      matchingGraph);
32        for (Object o: pm.getMatching()) {
33            if(o instanceof DefaultWeightedEdge){
34                DefaultWeightedEdge e = (DefaultWeightedEdge) o;
35                City s = matchingGraph.getEdgeSource(e);
36                City t = matchingGraph.getEdgeTarget(e);
37                double dist = matchingGraph.getEdgeWeight(e);
38
39                int sIndex = graph.getVertices().indexOf(s);
40                int tIndex = graph.getVertices().indexOf(t);
41
42                addLineState(graph, "Perfekt Matching. Kanten vereinigen",
      sIndex, tIndex, StateColor.BLUE);
43                perfectMatchAdj.get(sIndex).set(tIndex, dist);
44                perfectMatchAdj.get(tIndex).set(sIndex, dist);
45            }
46        }
```

Es Wird die Adjazenzmatrix „perfectMatchAdj“ zurückgegeben, die sowohl die Kanten des eingegeben Graphen, als auch die Kanten des Matchings enthält.