Technische Universität München

Fakultät für Mathematik
Lehrstuhl für Angewandte Geometrie und Diskrete Mathematik

Lösung und graphische Darstellung des Traveling Salesman Problems in einer Webapplikation

Bachelorarbeit von Sebastian Lotz

Themensteller:	Prof. Dr. Peter Gritzmann
Betreuer:	Dipl.-Math. Oec. Melanie Herzog, M.Sc. Wolfgang Ferdinand Riedl
Abgabedatum:	15. Juli 2014

Hiermit erkläre ich, dass ich diese Arbeit selbstständig angefertigt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Juli 2014

Sebastian Lotz

Abstract

Since its first description in the year 1930, the Traveling Salesman Problem is one of the most prominent examples of combinatorial optimization. It asks for the shortest Hamiltonian cycle on a complete, symmetric graph.
In the first part of this thesis, basic methods for solving this problem are presented. These include the two heuristics nearest-neighbor and multiple-fragment. Representing exact algorithms, a Lagrangian relaxation method based on 1-trees is examined in connection with a branch-and-bound algorithm.
These methods are part of the web application, which was developed during the creation of this thesis and is described in the second part. It is meant to depict the presented algorithms in order to give students an understanding of combinatorial optimization methods. The application therefore uses state-of-the-art web-based technologies such as HTML 5 and jQuery.

Zusammenfassung

Seit seiner ersten Beschreibung im Jahr 1930 ist das Traveling Salesman Problem eines der prominentesten Beispiele ungelöster Probleme der kombinatorischen Optimierung. Es fragt auf einem vollständigen, symmetrischen Graphen nach einem kürzesten Hamiltonkreis.
Im ersten Teil der vorliegenden Arbeit werden grundlegende Verfahren zur Lösung des Problems vorgestellt. Dazu zählen die beiden Heuristiken Nearest-Neighbor und Multiple-Fragment. Als Vertreter exakter Lösungsverfahren wird eine auf Eins-Bäumen basierende Lagrange-Relaxierung in Verbindung mit dem Branch-and-Bound-Algorithmus betrachtet.
Diese Verfahren sind Teil der in der zweiten Hälfte beschriebenen Webapplikation, die im Zuge der Arbeit erstellt wurde. Sie dient der anschaulichen Darstellung der Algorithmen mit dem Ziel, Schülern und Studenten Methoden der kombinatorische Optimierung näherzubringen. Die Applikation bedient sich dabei neuester Webtechnologien wie HTML 5 und jQuery.

Inhaltsverzeichnis

1 Das Traveling Salesman Problem 1
1.1 Entstehung . . . 1
1.2 Definition . . . 2
1.3 Zielsetzung . . . 5

2 Lösung des Traveling Salesman Problems 7
2.1 Heuristische Lösungsverfahren . . . 7
2.1.1 Nearest-Neighbor-Heuristik . . . 7
2.1.2 Multiple-Fragment-Heuristik . . . 13
2.2 Exakte Lösungsverfahren . . . 14
2.2.1 Lagrange-Relaxierung . . . 15
2.2.2 Branch-and-Bound . . . 19
2.3 Weitere Verfahren . . . 21

3 Darstellung in einer Webapplikation 25
3.1 Applikationsstruktur . . . 25
3.2 Front-End . . . 25
3.2.1 Allgemeiner Aufbau . . . 25
3.2.2 Aufbau der einzelnen Tabs . . . 26
3.2.3 Technische Umsetzung . . . 31
3.3 Back-End . . . 34
3.3.1 CherryPy-Server . . . 34
3.3.2 Karten . . . 37
3.3.3 Metriken . . . 38
3.3.4 Berechnung der Lösung . . . 38
3.4 Erweiterungsmöglichkeiten . . . 38

4 Ausblick 43

Literatur 45

Kapitel 1

Das Traveling Salesman Problem

1.1 Entstehung

Das *Traveling Salesman Problem* oder *Problem des Handlungsreisenden*, wie es auf deutsch heißt, beschäftigt sich mit der Frage, wie eine Rundtour durch eine gegebene Menge Städte geplant werden muss (ohne eine Stadt doppelt zu besuchen), damit der insgesamt zurückgelegte Weg möglichst kurz ist.
Dieses Problem taucht in der Literatur zum ersten Mal im Jahr 1831 in einem Buch von Voigt [Voi31] auf. Es trägt den Titel *Der Handlungsreisende wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein. Von einem alten Voyageur.* Entscheidend ist darin ein Ausschnitt auf den letzten Seiten:

> [...] Durch geeignete Auswahl und Planung der Tour kann man oft so viel Zeit sparen, daß wir einige Vorschläge zu machen haben. [...] Der wichtigste Aspekt ist, so viele Orte wie möglich zu erreichen, ohne einen Ort zweimal zu besuchen. [...]

Obgleich diese beiden Sätze die Fragestellung des Traveling Salesman Problems inhaltlich sehr gut treffen, wird der Einzug des Problems in die Mathematik auf einen Beitrag Karl Mengers im Rahmen des Mathematischen Kolloquiums in Wien 1930 zurückgeführt [Men32]:

> Wir bezeichnen als Botenproblem (weil diese Frage in der Praxis von jedem Postboten, übrigens auch von vielen Reisenden zu lösen ist) die Aufgabe, für endlich viele Punkte, deren paarweise Abstände bekannt sind, den kürzesten die Punkte verbindenden Weg zu finden.

Natürlich stimmt auch diese Formulierung nicht exakt mit dem Problem des Handlungsreisenden überein, da hier nur nach dem kürzesten Weg, nicht aber nach der kürzesten Rundtour gefragt ist.
Trotzdem gelangte es von Wien aus 1931/1932 zu Hassler Whitney [Law+85], der es unter dem heutigen Namen 1934 an der Universität Princeton vorstellte. [Flo56]
Ein weiterer Meilenstein war die Einordnung des Traveling Salesman Problems in die

Komplexitätsklasse der NP-schweren Probleme [Kar72]. Damit war bewiesen, dass die Lösung des Problem weit schwieriger ist, als zunächst angenommen.
Doch auch nach dieser Erkenntnis wurden unablässig neue Lösungsmethoden entwickelt - bis heute.

1.2 Definition

Zunächst ist es notwendig, grundlegende Begriffe zu klären, die im Zusammenhang mit der mathematischen Modellierung des Traveling Salesman Problems unabdingbar sind. Sie orientieren sich an [Die10].

Definition 1.1 (Graph)
Ein *Graph* ist ein Paar $G = (V, E)$ Mengen V und $E \subseteq \{\{v, w\} : v, w \in V\}$. Elemente von V heißen *Knoten*, Elemente von E nennt man *Kanten* des Graphen G.
Ein Graph heißt *vollständig*, falls $E = \{\{v, w\} : v, w \in V\}$.

Definition 1.2 (Gewichteter Graph)
Ein Graph $G = (V, E, c)$ mit einer Gewichtsfunktion $c : E \to \mathbb{R}$ heißt *gewichteter Graph.*

Man kann nun Städte mithilfe von Knoten und direkte Strecken zwischen zwei Städten durch Kanten beschreiben. Die Kantengewichte sind die jeweiligen Distanzen zweier Städte zueinander. Diese werden der Übersichtlichkeit halber im weiteren Verlauf mit $c_{vw} := c(\{v, w\})$ abgekürzt.

Definition 1.3 (Inzidenz, Grad)
Ein Knoten v und eine Kante e sind *inzident*, wenn $v \in e$. Die Anzahl der mit v inzidenten Kanten wird *Grad* $d_G(v)$ des Knotens v bezüglich G genannt.

Definition 1.4 (Weg)
Ein *Weg* ist ein nichtleerer Graph $P = (V', E')$ mit $V' = \{v_0, v_1, \ldots, v_k\} \subseteq V$ und $E' = \{\{v_0, v_1\}, \{v_1, v_2\}, ..., \{v_{k-1}, v_k\}\} \subseteq E$, wobei die v_i, $i \in \{1, \ldots, k\}$ paarweise verschieden sind.

Eine Tour kann in der Graphentheorie mithilfe eines *Kreises* modelliert werden:

Definition 1.5 (Kreis)
Ist $P = (V', E')$ ein Weg mit $E' = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-2}, v_{k-1}\}\}$ und $k \geq 3$, so nennt man $C = (V', E' \cup \{v_{k-1}, v_0\})$ *Kreis.*
Enthält ein Kreis alle Knoten des Graphen, nennt man ihn *Hamilton-Kreis*

Es ist sehr intuitiv, anzunehmen, dass der direkte Weg zwischen zwei Städten automatisch kürzer ist, als ein „Umweg“ über eine dritte Stadt. In der Graphentheorie heißt diese Eigenschaft *Dreiecksungleichung.*

Definition 1.6 (Dreiecksungleichung)
Seien $u, v, w \in V$. Dann gilt

$$c_{uv} \leq c_{uw} + c_{vw},\ c_{uw} \leq c_{uv} + c_{vw} \text{ und } c_{vw} \leq c_{uv} + c_{uw}.$$

Nun kann das Traveling Salesman Problem definiert werden.

Definition 1.7 (Traveling Salesman Problem)
Das Problem, auf einem Graphen einen kürzesten Hamilton-Kreis zu finden, wird als *Traveling Salesman Problem* bezeichnet. Es heißt *metrisches Traveling Salesman Problem*, falls für jedes Knotentripel des Graphen die Dreiecksungleichung erfüllt ist.

In der vorliegenden Arbeit soll ausschließlich das metrische Traveling Salesman Problem auf einem ungerichteten (aber nicht zwangsläufig vollständigen) Graphen betrachtet werden.
An dieser Stelle sei bemerkt, dass folgender Satz gilt:

Satz 1.8
Sei $G = (V, E, c)$ ein vollständiger Graph, der die Dreiecksungleichung erfüllt.
Dann gibt es einen kürzesten Weg durch alle Knoten, der keinen Knoten doppelt enthält.

Beweis. Angenommen der kürzeste Weg $P = (V', E')$ enthalte einen Knoten doppelt, dann gibt es Knoten $v_1, v_2, v_3, v_4, v_5 \in V'$ und Kanten $\{v_1, v_2\}, \{v_2, v_3\}, \{v_4, v_2\}, \{v_2, v_5\} \in E'$, wie in Abbildung 1.1 dargestellt.
Die Kanten $\{v_4, v_2\}$ und $\{v_2, v_5\}$ können dann durch die Kante $\{v_4, v_5\}$ ersetzt werden. Der neue Weg ist aufgrund der Dreiecksungleichung höchstens so lang wie der ursprüngliche und damit ebenfalls ein kürzester Weg durch alle Knoten. □

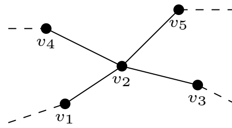


Abbildung 1.1: Ausschnitt einer kürzesten Tour, die den Knoten v_2 doppelt besucht.

Das Traveling Salesman Problem, wie es oben definiert wurde, kann auch als lineares Optimierungsproblem dargestellt werden:

Definition 1.9 (Lineares Optimierungsproblem)
Sei $G = (V, E, c)$, o.B.d.A. $V = \{1, \ldots, n\}$. Dann wird das Problem

$$\min \sum_{i \in V} \sum_{j > i} c_{ij} x_{ij} \tag{1.1}$$

unter den Nebenbedingungen

$$\sum_{j<i} x_{ji} + \sum_{j>i} x_{ij} = 2 \qquad \forall\ i \in V \tag{1.2a}$$

$$\sum_{i \in S} \sum_{\substack{j \in S: \\ j>i}} x_{ij} \leq |S| - 1 \qquad \forall\ S \subset V, S \neq \emptyset \tag{1.2b}$$

$$x_{ij} \in \{0, 1\} \qquad i, j \in V,\ j > i \tag{1.2c}$$

als das zum (symmetrischen) Traveling Salesman Problem gehörige *lineare Optimierungsproblem* bezeichnet.

Zum besseren Verständnis der Definition sollen die Gleichungen näher erläutert werden. Die Funktion (1.1) minimiert die Länge der benutzten Kanten unter den folgenden Voraussetzungen:

- Jeder Knoten ist inzident zu genau zwei Kanten. (1.2a)
- Auf keiner (echten) Knotenteilmenge existiert eine Tour. Daraus folgt, dass in einer Tour auf der gesamten Knotenmenge keine Subtouren enthalten sind. (1.2b)
- Die Kante $\{i, j\}$ ist entweder in der Lösung enthalten ($\Rightarrow x_{ij} = 1$) oder nicht ($\Rightarrow x_{ij} = 0$). (1.2c)

Es soll schließlich ein Beispielgraph $G = (V, E, c)$ auf fünf Knoten eingeführt werden, auf den die in den folgenden Abschnitten beschriebenen Algorithmen exemplarisch angewandt werden. V sei dazu die Knotenmenge $\{A, B, C, D, E\}$ und E die Kantenmenge $\{\{v, w\} \in V \times V : v \neq w\}$. Die Kantengewichte seien die in Abbildung 1.2 angegebenen.

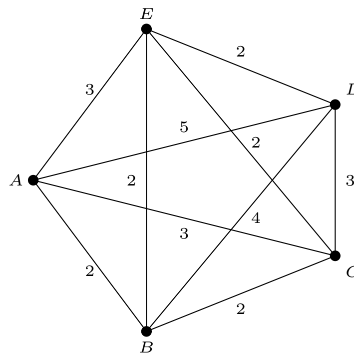


Abbildung 1.2: Vollständiger, ungerichteter Beispielgraph

1.3 Zielsetzung

Ziel der vorliegenden Arbeit ist zunächst die Präsentation grundlegender Lösungsverfahren des Traveling Salesman Problems und anschließend ihre Darstellung in einer Webapplikation. Zu diesem Zweck ist sie in zwei Teile unterteilt. Da sowohl Nearest-Neighbor- als auch Multiple-Fragment-Heuristik sehr anschauliche Beispiele für heuristische Lösungsalgorithmen darstellen, sollen beide zunächst unter mathematischen Gesichtspunkten vorgestellt werden. Als Vertreter der exakten Lösungsverfahren wurde eine Lagrange-Relaxierung gewählt, die Eins-Bäume als Lösungsmenge benutzt. Ihr Grundprinzip, sowie das der Branch-and-Bound-Methode lassen sich ebenfalls sehr gut darstellen. Natürlich stellen diese Verfahren keinesfalls den heutigen Stand der Forschung dar, sondern sind vielmehr erste erfolgreiche Annäherungsversuche an das Traveling Salesman Problem. In einem kurzen Ausblick zu weiteren Verfahren werden schließlich modernere Lösungsmethoden vorgestellt, zu denen auch sogenannte Verbesserungsverfahren zählen.

Im zweiten Teil wird eine Webapplikation präsentiert, die im Zuge dieser Arbeit erstellt wurde. Ihre Berechnungsalgorithmen beruhen in modifizierter Form auf einer Java-Implementierung aus dem Jahr 2004, die aber aufgrund fehlender Unterstützung heutiger Browser praktisch nicht mehr verwendbar ist. Die erstellte Applikation hingegen befindet sich nun auf dem Stand aktueller Webtechnologien wie HTML5 und jQuery. Ihr Aufbau, sowie ihre Funktionsweise sind in Kapitel 3 erklärt. Dabei wird insbesondere auch auf ihre Erweiterbarkeit eingegangen, um das Implementieren zusätzlicher Algorithmen zu vereinfachen.

Kapitel 2

Lösung des Traveling Salesman Problems

2.1 Heuristische Lösungsverfahren

Eine Klasse der Lösungsalgorithmen ist die Klasse der sogenannten *Heuristiken*. Sie garantieren nicht, dass eine optimale Lösung gefunden wird, sondern versuchen diese mithilfe von Entscheidungsregeln möglichst „klug“ anzunähern. Im Falle des Traveling Salesman Problems beschreibt ihre Ausgabe einen Hamilton-Kreis, der mindestens die Länge der optimalen Lösung hat.
Im folgenden Abschnitt sollen zwei dieser Heuristiken näher vorgestellt werden.

2.1.1 Nearest-Neighbor-Heuristik

Eine sehr intuitive Herangehensweise an das Problem ist der sogenannte *Nearest-Neighbor-Algorithmus*:
Zunächst wird ein Knoten (beispielsweise v_1) ausgewählt. Anschließend wird in jedem Iterationsschritt der Knoten v_i ausgewählt, der dem zuletzt ausgewählten Knoten v_{i-1} nächstgelegen und noch nicht im bereits gefundenen Weg enthalten ist. Die Kante $\{v_{i-1}, v_i\}$ wird dann in den Weg eingefügt. Der Algorithmus ist beendet, sobald alle Knoten besucht wurden. Dieses Vorgehen ist formal in Algorithmus 1 beschrieben.

Algorithmus 1 : Nearest-Neighbor-Heuristik

Eingabe : $G = (V, E, c)$
Ausgabe : Kreis V_C, der alle $v \in V$ besucht

```
1 z_1 ← v ∈ V beliebig
2 V' ← V \ {v_1}
3 i ← 2
4 while V' ≠ ∅ do
5     z_i ← argmin_{v∈V'} c({v_{i−1}, v})
6     V' ← V' \ {v_i}
7     i ← i + 1
8 end
9 V_C ← (v_1, ..., v_n, v_1)
```

Man kann sich leicht vorstellen, dass die Kanten ohne die letzte einen relativ kurzen Weg beschreiben, während die letzte Kante (fast) beliebige Länge haben kann und die gefundene Lösung deshalb häufig alles andere als optimal ist.
Zu bemerken ist aber, dass diese letzte Kante tatsächlich in ihrer Länge nach oben beschränkt ist, da sie die direkte Verbindung zwischen erstem und letztem Knoten darstellt, die gemäß der Dreiecksungleichung zwangsläufig kürzer ist als der „Umweg“ über die restlichen Knoten. Sie kann also nicht länger als der Rest des Weges sein.
Trotzdem ist die Lösung des Nearest-Neighbor-Verfahrens beliebig schlecht, wie folgender Satz zeigt.

Satz 2.1 ([RSI77])
Sei $\mathrm{NN}(I)$ *die Länge einer von der Nearest-Neighbor-Heuristik gefundenen Lösung und* $\mathrm{OPT}(I)$ *die optimale Lösung des Problems* I*, dann gilt: Für alle* $r > 1$ *gibt es eine Konfiguration* I *des Traveling Salesman Problems auf* n *Knoten, die unter Voraussetzung der Dreiecksungleichung die Ungleichung*

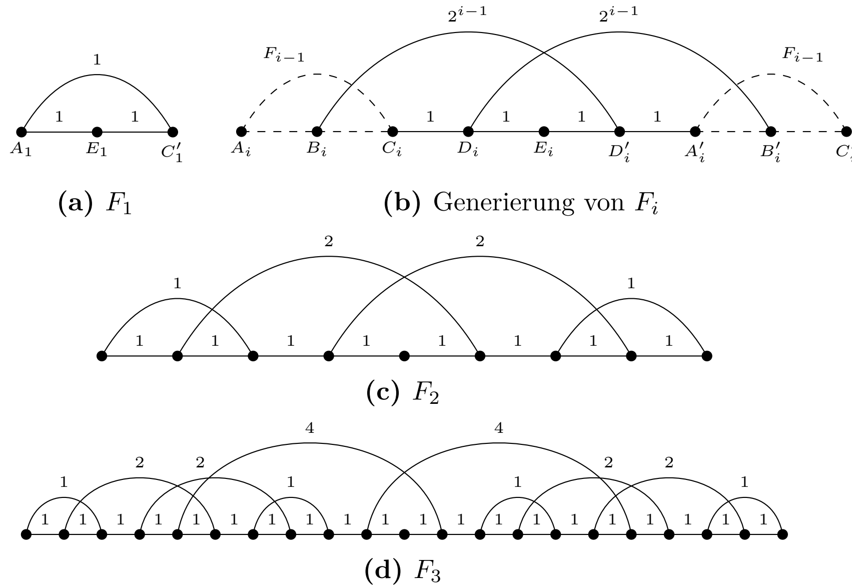
$$\mathrm{NN}(I) \geq r \cdot \mathrm{OPT}(I)$$

erfüllt, wenn n *nur ausreichend groß ist.*

Beweis. Für den Beweis dieses Satzes wird ein geeigneter Graph F_i konstruiert, der folgendermaßen entsteht: Der vollständige Graph F_1 enthält drei Knoten A_1, E_1 und C_1', die paarweise den Abstand 1 haben, wie in Abbildung 2.1a zu sehen ist. $F_i = (V_i, E_i, c_i)$ wird anschließend rekursiv mithilfe folgender Vorschrift generiert:
F_i besteht aus zwei Kopien von F_{i-1}, die über zusätzliche Kanten miteinander verbunden sind. Zum besseren Verständnis werde der linke Knoten A_{i-1} der ersten Kopie mit A_i bezeichnet und der der zweiten Kopie mit A_i'. Analog dazu wird der mittlere Knoten E_{i-1} mit B_i bzw. B_i' und der rechte Knoten C_{i-1} mit C_i bzw. C_i' benannt. Anschließend werden drei neue Knoten D_i, E_i und D_i' hinzugefügt. Mithilfe zusätzlicher Kanten entsteht wieder ein vollständiger Graph. Alle bisherigen Kantengewichte bleiben unverändert. Die Kanten $\{C_i, D_i\}$, $\{D_i, E_i\}$, $\{E_i, D_i'\}$, $\{D_i', A_i'\}$ erhalten das Gewicht 1, $\{B_i, D_i'\}$ und $\{D_i, B_i'\}$ das Gewicht 2^{i-1}. Alle weiteren Kantengewichte sind durch die Länge des kürzesten Weges zwischen den Endknoten der jeweiligen Kante gegeben.
Die Rekursionsvorschrift ist in Abbildung 2.1b grafisch dargestellt, ebenso F_2 und F_3 (Abbildung 2.1c bzw. 2.1d).

Behauptung 1: *Die Anzahl der Knoten im Graphen* F_i *beträgt* $3(2^i - 1)$.
Die Behauptung lässt sich mittels Induktion beweisen. Sei dazu $n(F_i)$ die Anzahl Knoten des Graphen F_i.
Induktionsanfang:

$$n(F_1) = 3$$



(a) F_1 **(b)** Generierung von F_i

(c) F_2

(d) F_3

Abbildung 2.1: Sukzessive Konstruktion des Graphen F_i

Induktionsschritt:

$$\begin{aligned} n(F_{i+1}) &= 2n(F_i) + 3 \\ &= 2(3(2^i - 1)) + 3 \\ &= 3 \cdot 2^{i+1} - 3 \\ &= 3(2^{i+1} - 1) \end{aligned}$$

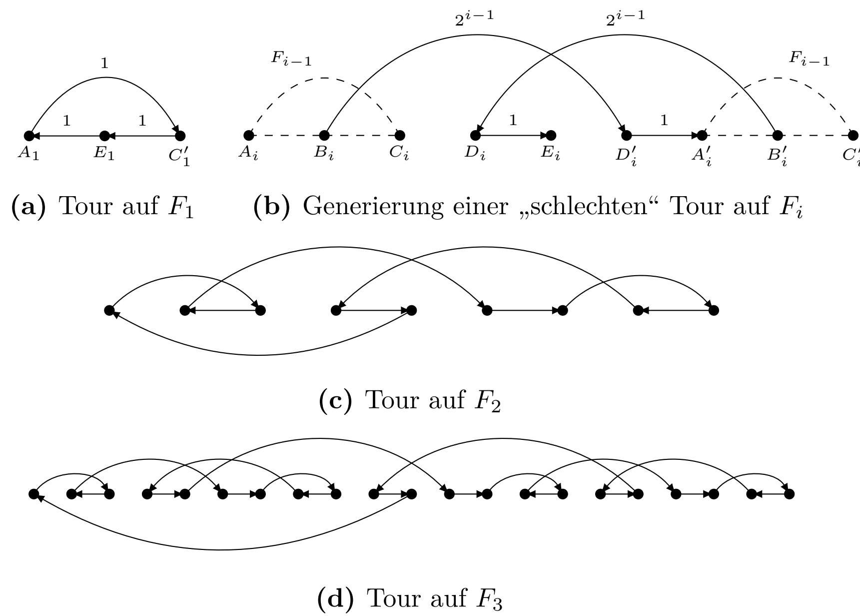
Behauptung 2: $\mathrm{OPT}(I) \leq 6(2^i) - 8$
Durch Ablaufen des Graphen von links nach rechts und anschließender Rückkehr zum Startpunkt auf direktem Weg erhält man eine Tour, die aufgrund der Dreiecksungleichung höchstens von der Länge

$$2(n(F_i) - 1) = 2(3(2^i - 1) - 1) = 6(2^i) - 8$$

sein kann, woraus die Behauptung folgt.

Nun wird eine Tour T_i konstruiert, die ein mögliches Ergebnis der Nearest-Neighbor-Heuristik darstellt. Auch diese Tour wird rekursiv mithilfe der in Abbildung 2.2a und 2.2b abgebildeten Vorschrift generiert. In F_1 startet der Algorithmus im Knoten A_1 und kehrt über die Knoten C'_1 und E_1 wieder zu diesem zurück. Um T_i zu erzeugen, wird zunächst die Tour T_{i-1} in der ersten Kopie von F_{i-1} bis einschließlich der vorletzten

Kante durchlaufen. Statt nun zum Startknoten A_i zurückzukehren, wird über die Kanten $\{B_i, D_i'\}$ und $\{D_i, A_i'\}$ der linke Knoten der zweiten Kopie von F_{i-1} erreicht. Auch diese wird analog zu T_{i-1} durchlaufen ohne zu A_i' zurückzukehren. Von B_i' aus führt der generierte Weg schließlich über D_i und E_i zum Startknoten A_i zurück. Offensichtlich gilt also für jeden Teilgraphen $F_k,\ k < i$, dass die erstellte Tour am linken Knoten beginnt und nach Besuch aller Knoten von F_k in der Mitte endet.



(a) Tour auf F_1 **(b)** Generierung einer „schlechten" Tour auf F_i

(c) Tour auf F_2

(d) Tour auf F_3

Abbildung 2.2: Worst-Case-Verhalten der Nearest-Neighbor-Heuristik auf F_i

Behauptung 3: *Die Länge einer Kante vom mittleren Knoten des Graphen F_i zu einem der beiden äußeren Knoten beträgt $2^i - 1$.*
Um die Länge einer solchen Kante zu beweisen, soll zunächst allgemein die Distanz $z(F_i) = 2^{i+1} - 3$ der beiden äußeren Knoten A_i und C_i' eines Graphen F_i zueinander gezeigt werden:
Induktionsanfang:

$$z(F_1) = 1$$
$$z(F_2) = 5$$

Induktionsschritt:

$$\begin{aligned} z(F_{i+1}) &= \underbrace{z(F_{i-1}) + 2}_{A_i \to B_i} + \underbrace{2^i + 1}_{B_i \to A'_i} + \underbrace{z(F_i)}_{A'_i \to C'_i} \\ &= 2^i - 1 + 2^i + 1 + 2^{i+1} - 3 \\ &= 2^{(i+1)+1} - 3 \end{aligned}$$

Aus der ermittelten Distanz folgt nun für den Graphen F_i die Länge des Weges von A_i nach C_i mit $z(F_{i-1}) = 2^i - 3$. Addiert man nun die Distanz zwischen den Knoten C_i und E_i hinzu, erhält man mit $z(F_{i-1})+2 = 2^i-1$ die Länge des Weges von A_i nach E_i.

Behauptung 4: *Der konstruierte Weg ist ein mögliches Ergebnis der Nearest-Neighbor-Heuristik.*
Dass es sich bei der konstruierten Tour tatsächlich um eine gültige Lösung des Verfahrens handelt, kann man sich leicht klarmachen, wenn man davon ausgeht, dass wegen der Gültigkeit der Dreiecksungleichung eingezeichnete Kanten bevorzugt oder zumindest gleichberechtigt zu den nicht eingezeichneten Kanten ausgewählt werden.
Die Tour T_1 in F_1 ist offensichtlich ein mögliches Ergebnis. Für jeden Rekursionsschritt gilt als Induktionsvoraussetzung, dass die Wege auf den beiden enthaltenen F_{i-1} mögliche Lösungen der Nearest-Neighbor-Heuristik darstellen, wenn man die letzte gefundene Kante (zurück zum Startpunkt) entfernt. Es bleibt also zu zeigen, dass die Verbindung dieser beiden Wege ebenfalls Teil einer solchen Lösung ist.
Ist der Algorithmus bei B_i angekommen, hat er zwei Möglichkeiten fortzufahren. Er kann entweder die Kante $\{B_i, D_i\}$ oder die Kante $\{B_i, D'_i\}$ benutzen, alle weiteren verbleibenden Kanten von B_i aus sind offensichtlich länger und kommen deshalb für den Algorithmus nicht in Betracht.
Aufgrund von Behauptung 3 gilt: $c_{B_iD_i} = c_{B_iC_i} + c_{C_iD_i} = 2^{i-1} - 1 + 1 = 2^{i-1}$. Die Kante $\{B_i, D_i\}$ ist also genauso lang wie die Kante $\{B_i, D'_i\}$ und der Algorithmus kann sich an dieser Stelle tatsächlich für letztere entscheiden.
Analog dazu ist die Distanz zwischen B'_i und D_i mit 2^{i-1} kleiner als die Distanz zwischen B'_i und E_i mit $2^{i-1} + 1$. Von B'_i aus muss der Algorithmus also die Kante $\{B'_i, D_i\}$ in seine Lösung aufnehmen. Anschließend bleiben nur die Knoten E_i und A_i übrig, die er in dieser Reihenfolge besucht.

Behauptung 5: *Der konstruierte Weg hat die Länge $(i + 2)2^i - 3$.*
Die Länge des generierten Weges vom linken bis zum mittleren Knoten, $l(F_i) = (i + 1)2^i - 2$, kann ebenfalls mittels Induktion nachgewiesen werden.
Induktionsanfang:

$$l(F_1) = 2$$

Induktionsschritt:

$$\begin{aligned} l(F_{i+1}) &= 2l(F_i) + 2 \cdot 2^i + 2 \\ &= 2((i+1)2^i - 2) + 2^{i+1} + 2 \\ &= (i+1)2^{i+1} + 2^{i+1} - 2 \\ &= ((i+1)+1)2^{i+1} - 2 \end{aligned}$$

Aus den beiden vorangegangenen Behauptungen 3 und 5 folgt

$$NN(I) = l(F_i) + z(F_i) = (i+1)2^i - 2 + 2^i - 1 = (i+2)2^i - 3.$$

Daraus folgt für das Verhältnis zwischen gefundener und optimaler Tour

$$\frac{\mathrm{NN}(I)}{\mathrm{OPT}(I)} \geq \frac{(i+2)2^i - 3}{6(2^i) - 8}.$$

Die rechte Seite ist für $i \geq 1$ streng monoton steigend, sie kann also jeden Wert $r > 1$ überschreiten. □

Dieser Beweis zeigt, dass die Nearest-Neighbor-Heuristik das *metrische* Traveling Salesman Problem unter entsprechenden Umständen beliebig schlecht löst. Es gibt aber auch Graphen in der euklidschen Ebene, die zu solch schlechten Lösungen führen können. Hurkens und Woeginger [HW04] konnten eine Instanz I des *euklidschen* Traveling Salesman Problems auf n Knoten finden, für die gilt:

$$\frac{\mathrm{NN}(I)}{\mathrm{OPT}(I)} \geq (\sqrt{3} - \frac{3}{2})(\log n - 2)$$

Abbildung 2.3 zeigt schließlich eine Rundtour, die durch Anwendung der Nearest-Neighbor-Heuristik auf den Beispielgraphen entstanden sein könnte.

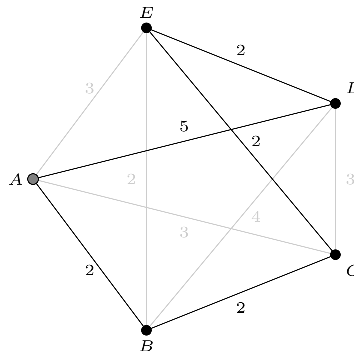


Abbildung 2.3: Mögliche Lösung der Nearest-Neighbor-Heuristik mit Startknoten A und Länge 13

2.1.2 Multiple-Fragment-Heuristik

Statt lokal von einem Knoten aus eine möglichst optimale Kante zu suchen, ist es ebenso möglich, das Problem „globaler“ zu betrachten und in jedem Schritt aus allen verfügbaren Kanten die kürzeste zu wählen. Dazu bietet es sich an, alle Kanten zunächst ihrer Länge nach aufsteigend zu sortieren und anschließend in dieser Reihenfolge in den Lösungsgraphen einzufügen. Um nun tatsächlich eine Tour zu erhalten, muss die „neue“ Kante in jedem Schritt jedoch zwei Bedingungen erfüllen. Nach ihrem Einfügen darf der Graph weder eine (echte) Subtour noch Knoten von Grad größer als 2 enthalten. Diese sogenannte *Multiple-Fragment-Heuristik* ist in Algorithmus 2 dargestellt.
In der Literatur ist das Verfahren häufig unter dem Namen *Greedy-Algorithmus* zu finden. Um Verwechslungen mit dem Oberbegriff für Greedy-Verfahren, zu denen die Nearest-Neighbor-Heuristik beispielsweise auch zählt, zu vermeiden, wird hier der von Bentley geprägte Name gebraucht [Ben92].
Das Verfahren ähnelt grundsätzlich dem Kruskal-Algorithmus, der einen kürzesten Spannbaum sucht und deshalb an die eingefügten Kanten nur die Bedingung stellt, keine Kreise zu erzeugen.
Ebenso wie die Nearest-Neighbor-Heuristik kann auch die Multiple-Fragment-Heuristik zu beliebig schlechten Ergebnissen führen, wie der folgende Satz von Frieze [Fri79] zeigt:

Satz 2.2
Sei $\mathrm{MF}(I)$ *die Länge einer möglichen Lösung der Multiple-Fragment-Heuristik zu einer Instanz* I *des Traveling Salesman Problems und* $\mathrm{OPT}(I)$ *die Länge seiner optimalen Lösung. Dann gibt es einen Graphen* G *auf* $n = m^m$ *Knoten, sodass für* $m \geq 3$ *gilt:*

$$\frac{\mathrm{MF}(I)}{\mathrm{OPT}(I)} \geq \frac{\log_2 n}{3 \log_2 \log_2 n}$$

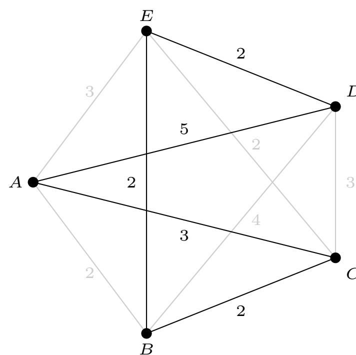


Abbildung 2.4: Eine mögliche Lösung der Multiple-Fragment-Heuristik mit Länge 14

Algorithmus 2 : Multiple-Fragment-Heuristik

Eingabe : $G = (V, E, c)$
Ausgabe : Kreis (V, E_C, c), der alle $v \in V$ besucht

```
1  z_1 ← argmin_{e∈E} c(e)
2  E' ← E \ {z_1}
3  i ← 2
4  n ← |V|
5  while i ≤ n do
6      z' ← argmin_{e∈E'} c(e)
7      G' ← (V, {z_1, ..., z_{i−1}, z'}, c)
8      {x, y} ← z'
9      if d_{G'}(x) < 3 and d_{G'}(y) < 3 and not G' enthält Subtour then
10         z_i ← z'
11         i ← i + 1
12     end
13     E' ← E' \ {z'}
14 end
15 E_C ← {z_1, ..., z_n}
```

2.2 Exakte Lösungsverfahren

Heuristische Verfahren liefern mitunter gute, teilweise sogar optimale Lösungen, dafür aber keinen Beweis für die Optimalität der gefundenen Lösung. Sie dienen deshalb häufig nur als obere Schranke für andere, *exakte Lösungsverfahren*. Letztere nähern sich sukzessive einer optimalen Lösung an.
Im Folgenden sollen zwei anschauliche dieser Verfahren vorgestellt werden.
Um sie beschreiben zu können, müssen zunächst aber noch einige Begriffe definiert werden.

Definition 2.3 (Baum)
Sei $T = (V, E)$ mit $|E| = |V| - 1$ ein Graph, der keinen Kreis enthält. Dann heißt T *Baum*.

Definition 2.4 (Eins-Baum)
Sei $G = (V, E)$ ein Graph mit einer Kantenmenge E, die alle Knoten verbindet und genau einen Kreis enthält. Dann heißt G *Eins-Baum*.

Definition 2.5 (Minimaler Spannbaum)
Sei $G = (V, E, c)$ ein Graph und $T = (V, E', c)$ ein Baum mit $E' \subseteq E$. Dann heißt T *minimaler Spannbaum* von G, falls gilt

$$\sum_{e \in E'} c(e) = \min_{F \subseteq E} \sum_{e \in F} c(e).$$

2.2.1 Lagrange-Relaxierung

Um das Traveling Salesman Problem zu lösen, bedient man sich häufig einer Technik, die die Menge der Lösungen zunächst erweitert, indem man Nebenbedingungen verändert oder entfernt. Sie heißt *Relaxierung* oder *Relaxation*. Eine Lösung des ursprünglichen Problems wird anschließend durch Einbringung der weggefallenen Nebenbedingungen in die Zielfunktion versucht zu erreichen.
Die hier betrachtete Form einer *Lagrange-Relaxierung* benutzt als zulässige Menge die Menge der Eins-Bäume.
Offensichtlich ist auch eine Rundtour auf allen Knoten (und damit auch die gesuchte Lösung) in dieser Menge enthalten. Die Erzeugung eines *minimalen* Eins-Baums erfolgt in zwei Schritten:

- Entferne einen Knoten v^* aus dem Graphen und finde auf dem entstandenen Graphen einen minimalen Spannbaum. Zu diesem Zweck eignen sich gängige Algorithmen, wie z.B. Kruskal oder Prim.
- Finde die zwei kürzesten Kanten zwischen dem entfernten Knoten v^* und dem Restgraphen und füge sie dem Spannbaum hinzu.

Der gefundene minimale Spannbaum auf der Knotenmenge $V \setminus v^*$ ist per Definition höchstens so lang wie eine optimale Tour, aus der die beiden zu v^* inzidenten Kanten entfernt wurden. Da aber auch die im zweiten Schritt hinzugefügten Kanten höchstens die Länge der zwei an v^* anliegenden Kanten der optimalen Tour haben können, ist der gesamte Eins-Baum ebenfalls höchstens so lang wie eine optimale Lösung des Traveling Salesman Problems. Mit $v^* = 1$ erhält man folgende Definition [Law+85]:

Definition 2.6 (Relaxiertes lineares Optimierungsproblem)
Sei $G = (V, E, c)$, o.B.d.A. $V = \{1, \ldots, n\}$. Das relaxierte Problem von Definition 1.9 hat dann folgende Form:

$$\min \sum_{i \in V} \sum_{j > i} c_{ij} x_{ij} \tag{2.1}$$

unter den Nebenbedingungen

$$\sum_{i \in S} \sum_{\substack{j \in V' \setminus S: \\ j > i}} x_{ij} + \sum_{i \in V' \setminus S} \sum_{\substack{j \in S: \\ j > i}} x_{ij} \geq 1 \qquad \forall\, S \subset V' = V \setminus \{1\},\; S \neq \emptyset \tag{2.2a}$$

$$\sum_{i \in V} \sum_{j > i} x_{ij} = n \tag{2.2b}$$

$$\sum_{j \in V} x_{1j} = 2 \tag{2.2c}$$

$$x_{ij} \in \{0, 1\} \qquad i, j \in V, j > i \tag{2.2d}$$

Die erste Nebenbedingung fordert, dass alle Knoten in $V \setminus 1$ über mindestens einen Weg miteinander verbunden sind. Darüber hinaus soll eine Lösung genau n Kanten enthalten (2.2b), von denen zwei inzident zu Knoten 1 sein müssen (2.2c). Gemäß Nebenbedingung (2.2d) kann eine Kante entweder in der Lösung enthalten sein ($x_{ij} = 1$) oder nicht ($x_{ij} = 0$).
Das Ausmaß dieser Erweiterung der Lösungsmenge macht folgende Rechnung deutlich: Die Anzahl aller Spannbäume auf der Knotenmenge $V \setminus \{v^*\}$ beträgt nach der Formel von Cayley $(n-1)^{n-3}$ [Cay89]. Der Knoten v^* kann nun auf $\binom{n-1}{2}$ verschiedene Weisen mit zwei unterschiedlichen Knoten des Spannbaums verbunden werden. Die Anzahl möglicher Eins-Bäume auf einem ungerichteten, vollständigen Graphen ist demnach

$$\begin{aligned}(n-1)^{n-3} \cdot \binom{n-1}{2} &= (n-1)^{n-3} \cdot \frac{(n-1)!}{2!\,(n-3)!} \\ &= (n-1)^{n-3} \cdot \frac{(n-1)(n-2)}{2} \\ &= \frac{1}{2}(n-2)(n-1)^{n-2}\end{aligned}$$

Es gibt dagegen aber nur $\frac{1}{2}(n-1)!$ (ungerichtete) Rundtouren. Für einen Graphen auf beispielsweise 20 Knoten ist gemäß des errechneten Verhältnisses

$$\frac{\frac{1}{2}(n-1)!}{\frac{1}{2}(n-2)(n-1)^{n-2}} = \frac{(n-3)!}{(n-1)^{n-3}}$$

also nur etwa jeder 15-millionste Eins-Baum eine Tour.

Ein so definiertes Verfahren führt nur in seltenen Fällen zu einem befriedigenden Ergebnis. Um eine tatsächliche Lösung des Traveling Salesman Problems zu erzielen, wird nun die Zielfunktion so erweitert, dass sie Abweichungen von Nebenbedingung (1.2a), also von einer Tour, minimiert. Dies geschieht mithilfe eines Lagrange-Multiplikators $\lambda \in \mathbb{R}^n$. Die neue Zielfunktion ist nun mit $X(I) := \{x : x \text{ erfüllt (2.2a) - (2.2d)}\}$ als Menge aller Inzidenzvektoren zulässiger Kantenmengen

$$L(\lambda) = \min_{x \in X(I)} \left\{ \sum_{i \in V} \sum_{j > i} c_{ij} x_{ij} + \sum_{i \in V} \lambda_i \left(\sum_{j < i} x_{ji} + \sum_{j > i} x_{ij} - 2 \right) \right\} \tag{2.3}$$

$$= \min_{x \in X(I)} \left\{ \sum_{i \in V} \sum_{j > i} (c_{ij} + \lambda_i + \lambda_j) x_{ij} \right\} - 2 \sum_{i \in V} \lambda_i \tag{2.4}$$

Da $L(\lambda) \leq OPT(I)$, wie oben bereits bemerkt, versucht man die Lagrange-Funktion zu maximieren um eine Rundtour zu erhalten.

$$L(\tilde{\lambda}) = \max_{\lambda} \{L(\lambda)\} \tag{2.5}$$

Dies erreicht man mithilfe eines *Subgradientenverfahrens*. Das hier dargestellte von Held und Karp [HK71] ist eines der ersten, die für das Traveling Salesman Problem entwickelt wurden.
In jedem Schritt wird zunächst $L(\lambda)$ berechnet. Die Kantenmenge des resultierenden Eins-Baums sei mit $H(\lambda)$ bezeichnet. Dann ist der durch die Komponenten

$$\delta_i^k := (d_{H(\lambda^k)}(i) - 2)_i \ \forall i \in V$$

gegebene Vektor ein Subgradient von $L(\lambda)$ an der Stelle λ^k.
Sei U eine bereits gefundene obere Schranke des Problems und $0 < \alpha^k \leq 2$. Dann ist die Schrittweite t^k des Verfahrens definiert durch

$$t^k = \alpha^k \frac{U - L(\lambda^k)}{\sum_{i \in V} (\delta_i^k)^2}.$$

Durch Addition von λ^k und dem mit der Schrittweite t^k skalierten Subgradienten erhält man nun den Lagrange-Multiplikator λ^{k+1} des nächsten Schritts:

$$\lambda_i^{k+1} = \lambda_i^k + t^k \delta_i^k \quad \forall i \in V$$

In einer näheren Untersuchung des Verfahrens konnten Held, Wolfe und Crowder [HWC74] folgende hinreichende Bedingung für seine Konvergenz zeigen:

Satz 2.7 (Konvergenz des Subgradientenverfahrens)
Das beschriebene Subgradientenverfahren konvergiert gegen eine Lösung, falls

$$\sum_{k=1}^{\infty} t^k = \infty \text{ und } \lim_{k \to \infty} t^k = 0$$

erfüllt sind. Dies ist der Fall, wenn U die Länge einer optimalen Lösung ist und α so gewählt ist, dass $\alpha^0 = 2$ und $\lim\limits_{k \to \infty} \alpha^k = 0$.

Das Verfahren ist in Algorithmus 3 dargestellt, wo die schrittweise Reduzierung von α über einen konstanten Vorfaktor $0 < \omega < 1$ erreicht wird. $H(\lambda^k)$ bezeichnet wiederum

den im k-ten Schritt des Verfahrens gefundenen Eins-Baum.

Algorithmus 3 : Subgradientenverfahren

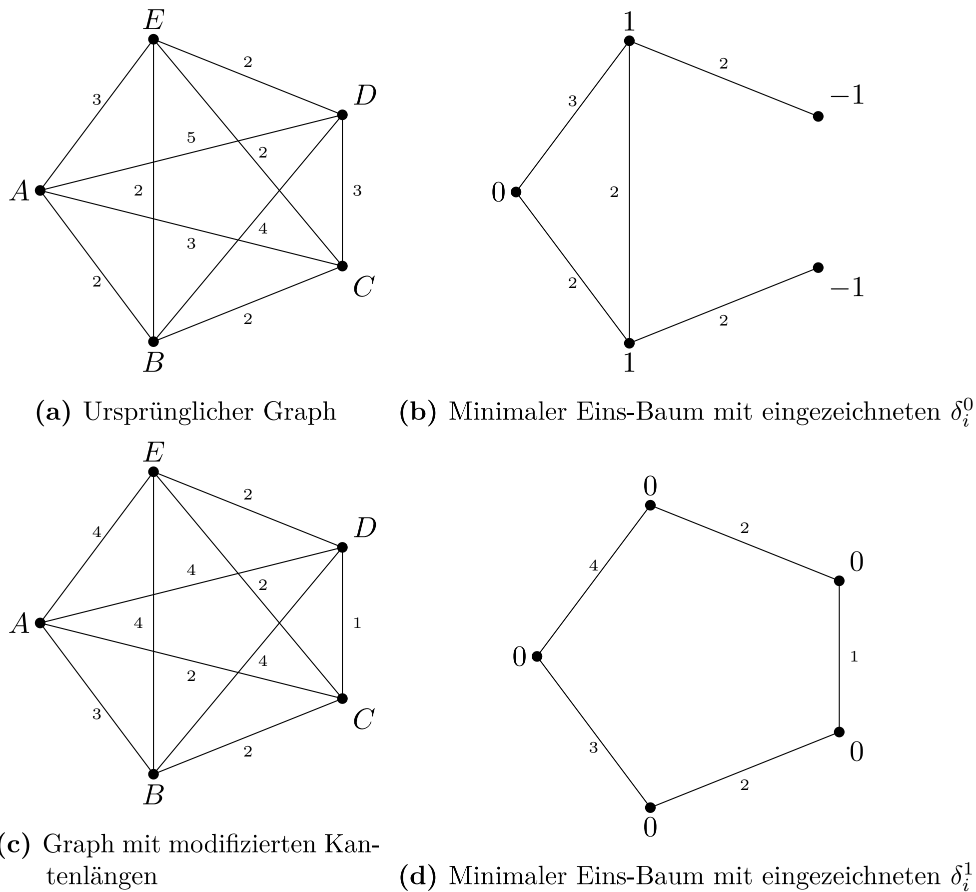
Eingabe : $G = (V, E, c)$, maximale Anzahl Iterationen I, obere Schranke U, Nullvektor λ^0, $0 < \alpha \leq 2$, $0 < \omega < 1$
Ausgabe : Eins-Baum (V, T, c), der alle $v \in V$ besucht

1 $T^0 \leftarrow H(\lambda^0)$
2 $k \leftarrow 0$
3 **while** $k < I$ ***and not*** $d_{T^k}(i) = 2\ \forall i \in V$ ***and not*** $\sum_{e \in T^k} c(e) \geq u$ **do**
4 $\quad t^k = \alpha(u - \sum_{e \in T^k} c(e)) / \sum_{i \in V} (d_{T^k}(i) - 2)^2$
5 $\quad \lambda_i^{k+1} = \lambda_i^k + t^k (d_{T^k}(i) - 2) \quad \forall i \in V$
6 $\quad T^{k+1} \leftarrow H(\lambda^{k+1})$
7 $\quad \alpha \leftarrow \omega \cdot \alpha$
8 $\quad k \leftarrow k + 1$
9 **end**
10 $T \leftarrow T^k$

Für die Implementierung des Verfahrens im Rahmen der vorliegenden Arbeit wurde für die Berechnung des Lagrange-Multiplikators die folgende Regel verwendet, wie sie ursprünglich von Volgenant und Jonker [VJ82] stammt:

$$\lambda_i^{k+1} = \begin{cases} \lambda_i^k & \text{falls } d_{T^k}(i) = 2 \\ \lambda_i^k + t^k \Big(0.6 \cdot \delta_i^k + 0.4 \cdot \delta_i^{k-1}\Big) & \text{sonst} \end{cases}$$

Trotz seiner garantierten Konvergenz ist das Subgradientenverfahren nicht dazu geeignet, Lösungen auf großen Knotenmengen zu finden, da es häufig nur sehr langsam konvergiert und deshalb zu hohen Rechenzeiten führen kann. Es gibt daher verschiedene Abbruchkriterien, wie beispielsweise eine vorab festgelegte Differenz zwischen oberer Schranke und Zwischenergebnis des Verfahrens, die nur einmalig unterschritten werden darf. Die hier dargestellte Version des Verfahrens bricht die Berechnung nach einer bestimmten Anzahl Iterationen ab.
In Abbildung 2.5 ist das Subgradientenverfahren aus Algorithmus 3 mit Schrittweite $t^k = 1\ \forall k$ dargestellt. In Abschnitt 2.2.1, die einen minimalen Eins-Baum darstellt, gilt $\lambda^0 = (0, 1, -1, -1, 1)$ und damit $L(\lambda^0) = 11$. Für die Lösung in Abbildung 2.5d nehmen λ^k und $L(\lambda^k)$ die Werte $\lambda^1 = (0, 0, 0, 0, 0)$ und $L(\lambda^1) = 12$ an. Dies ist die optimale Lösung.



(a) Ursprünglicher Graph **(b)** Minimaler Eins-Baum mit eingezeichneten δ_i^0

(c) Graph mit modifizierten Kantenlängen **(d)** Minimaler Eins-Baum mit eingezeichneten δ_i^1

Abbildung 2.5: Subgradientenverfahren aus Algorithmus 3 mit Schrittweite $t^k = 1$ in jedem Schritt k

2.2.2 Branch-and-Bound

Um die Knotenmenge zu reduzieren und so die Wahrscheinlichkeit einer schnellen Konvergenz des Subgradientenverfahrens zu erhöhen, eignet sich das sogenannte *Branch-and-Bound*-Verfahren. Es besteht grundsätzlich aus drei Schritten, die mehrmals hintereinander ausgeführt werden:

- Teile das Problem in mehrere disjunkte Subprobleme, indem für jedes Subproblem eine zusätzliche Nebenbedingung hinzugefügt wird. (*Branching*)
- Versuche eines der Subprobleme zu lösen.
- Ist die Lösung nicht besser als eine bereits gefundene, bilde zum aktuellen Problem keine neuen Subprobleme. Suche stattdessen ein Problem, das noch ungelöste

Subprobleme enthält und wiederhole den zweiten Schritt. (*Bounding*)

Wie die Bezeichnung „Branching“ bereits verrät, entsteht durch das Unterteilen von Problemen in Subprobleme ein Entscheidungsbaum. Dieser Baum hat die essentielle Eigenschaft, dass die Lösungen des Subgradientenverfahrens vom Startknoten in Richtung der Blätter monoton steigend sind. Diese Tatsache begründet den Bounding-Schritt. Ist eine aktuelle Lösung also länger als eine gefundene obere Schranke, kann keiner der folgenden Knoten diese obere Schranke unterbieten, weil diese ausschließlich längere Lösungen als die aktuelle enthalten.
Obere Schranken können zu Beginn des gesamten Verfahrens relativ schnell mithilfe der in Abschnitt 2.1 vorgestellten Heuristiken gefunden werden. Man bezeichnet sie deshalb häufig auch als *Eröffnungsverfahren*.
Für das Branching existieren zahlreiche Entscheidungsregeln. Sie funktionieren nach dem folgenden Prinzip: In jedem Branch-and-Bound-Schritt k wird die Menge der ausgeschlossenen Kanten E_k und/oder die der erzwungenen Kanten I_k erweitert. Alle Kanten, die weder in E_k noch in I_k enthalten sind, nennt man *frei*. Die in 2.2 definierten Nebenbedingungen werden um

$$x_{ij} = \begin{cases} 0 & (i,j) \in E_k \\ 1 & (i,j) \in I_k \end{cases} \tag{2.6}$$

erweitert.
Eine gängige Branching-Regel für das Traveling Salesman Problem ist die folgende:

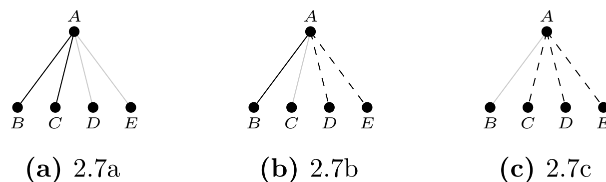
Definition 2.8 (Entscheidungsregel nach Volgenant und Jonker [VJ82])
Sei i ein Knoten, dessen Grad im aktuellen Eins-Baum größer als 2 ist und seien $\{i, j_1\}$ und $\{i, j_2\}$ zwei freie Kanten in demselben Eins-Baum. Dann werden folgende drei Subprobleme gebildet:

$$E_{k1} = E_k \cup \{\{i,j\} : j \notin \{j_1, j_2\}\} \quad I_{k1} = I_k \cup \{\{i,j_1\},\{i,j_2\}\} \tag{2.7a}$$
$$E_{k2} = E_k \cup \{\{i,j_2\}\} \quad I_{k2} = I_k \cup \{\{i,j_1\}\} \tag{2.7b}$$
$$E_{k3} = E_k \cup \{\{i,j_1\}\} \quad I_{k3} = I_k \tag{2.7c}$$

Ist der Knoten i bereits inzident zu einer Kante in I_k, wird das Problem $k1$ nicht erstellt.

Man wählt in einem Eins-Baum also einen Knoten aus, an dem mindestens drei Kanten anliegen. Das erste Subproblem erhält man, indem zwei dieser Kanten für die folgenden Lösungen erzwungen werden und alle restlichen für eine mögliche Lösungen nicht mehr in Betracht kommen. Waren im ersten Subproblem die beiden Kanten $\{i, j_1\}$ und $\{i, j_2\}$ verpflichtend, wird zur Bildung des zweiten Problems die Kante $\{i, j_1\}$ erzwungen und $\{i, j_2\}$ verboten. Das Problem $k1$ ist damit offensichtlich disjunkt zum Problem $k2$, denn die Kante $\{i, j_2\}$ ist im ersten vorhanden, im zweiten jedoch nicht. Verbietet man

im dritten Subproblem nun $\{i, j_2\}$, erhält man ein Problem, das zu den anderen beiden wiederum disjunkt ist, denn diese Kante war in beiden anderen vorgeschrieben. Doch das ist nicht der einzige Vorteil des Verfahrens. Auch die letzte Lösung, die aufgrund des Knotens von Grad größer als 2 nicht optimal gewesen sein kann, ist nicht mehr in der Menge der Subprobleme enthalten. Seien $\{i, j_1\}$, $\{i, j_2\}$ und $\{i, j_3\}$ drei der zu diesem Knoten inzidenten Kanten. Dann ist $\{i, j_3\}$ nicht im ersten Subproblem, $\{i, j_2\}$ nicht im zweiten und $\{i, j_1\}$ nicht im dritten Subproblem enthalten. Eine solche Lösung wurde also erfolgreich ausgeschlossen.
In Abbildung 2.6 ist exemplarisch das Ergebnis der Branching-Regel (mit $i = A$) auf einem Ausschnitt eines Eins-Baums dargestellt. Schwarze Kanten stellen erzwungene Kanten dar, graue symbolisieren ausgeschlossene und unterbrochene sind freie Kanten.



(a) 2.7a **(b)** 2.7b **(c)** 2.7c

Abbildung 2.6: Entscheidungsregel nach Volgenant und Jonker

2.3 Weitere Verfahren

Die hier vorgestellten Verfahren stellen natürlich nur einen Bruchteil der heute existierenden Algorithmen zur Lösung des Traveling Salesman Problems dar. Sie sind gewissermaßen ein Grundstock auf dem die Entwicklung effizienterer Verfahren aufbauen konnte. Nichtsdestotrotz zeigt sich in ihnen ein Grundprinzip, das sich bis heute bewährt hat: Die Länge einer optimalen Tour wird mithilfe von Heuristiken von oben und unter Zuhilfename von exakten Lösungsverfahren von unten beschränkt.
Zwei wichtige Heuristiken dürfen nicht ungenannt bleiben:
Die erste der beiden heißt *Minimal-Spanning-Tree-Heuristik*. Sie sucht auf einem Graphen zunächst einen minimalen Spannbaum, verdoppelt anschließend alle darin enthaltenen Kanten und läuft die dadurch entstandene Tour ab, indem sie jede Kante exakt einmal benutzt. Eine solche Tour, in der darüber hinaus Knoten beliebig häufig besucht werden dürfen, nennt man *Euler-Tour*. Statt auf dieser Tour nun Knoten doppelt zu besuchen, erstellt sie jeweils eine Abkürzung zwischen dem letzten „neu" besuchten Knoten und dem nächsten Knoten, der noch nicht Teil der entstehenden Rundtour ist.
Die andere ist die *Christofides-Heuristik*, die ebenfalls einen minimalen Spannbaum sucht und anschließend auf allen Knoten ungeraden Grades ein minimales *Matching* ermittelt, also je zwei dieser Knoten paarweise durch eine Kante verbindet, sodass das Gesamtgewicht dieser Kanten minimal ist. Das minimale Matching wird dann

in den minimalen Spannbaum eingefügt und die entstandene Euler-Tour wie im Fall der Minimal-Spanning-Tree-Heuristik durch das Benutzen von Abkürzungen in eine Rundtour umgewandelt.
Das Bemerkenswerte dieser beiden Verfahren ist der Grad der Optimalität, den sie garantieren können:

Satz 2.9 (Güte der Minimal-Spanning-Tree-Heuristik [Law+85])
Sei $\mathrm{MST}(I)$ *die Länge einer Lösung der Minimal-Spanning-Tree-Heuristik und* $\mathrm{OPT}(I)$ *die Länge einer optimalen Lösung. Dann gilt für jede Instanz* I *des Traveling Salesman Problems, die die Dreiecksungleichung erfüllt:*

$$\frac{\mathrm{MST}(I)}{\mathrm{OPT}(I)} \leq 2$$

Diese Schranke lässt sich mithilfe des Algorithmus selbst einfach beweisen. Ein minimaler Spannbaum ist offensichtlich höchstens so lang wie eine kürzeste Rundtour, da er alle Knoten auf optimale Weise durch $n-1$ Kanten verbindet. Der „gedoppelte“ Spannbaum hat also ein Gesamtkantengewicht von höchstens $2\,\mathrm{OPT}(I)$. Durch die anschließende Ersetzung einiger Kanten durch Abkürzungen kann die Länge der Tour aufgrund der Dreiecksungleichung nicht länger werden.

Satz 2.10 (Güte der Christofides-Heuristik [CN78])
Sei $\mathrm{C}(I)$ *die Länge einer Lösung der Christofides-Heuristik und* $\mathrm{OPT}(I)$ *die Länge einer optimalen Lösung. Dann gilt für jede Instanz* I *des Traveling Salesman Problems, die die Dreiecksungleichung erfüllt:*

$$\frac{\mathrm{C}(I)}{\mathrm{OPT}(I)} < \frac{3}{2}$$

Weitere bekannte Heuristiken sind die sogenannten *Insertion-Heuristiken*. Dort wird in jedem Schritt ein bisher nicht enthaltener Knoten zur aktuellen Subtour hinzugefügt. Die Auswahl dieses Knotens richtet sich je nach Version *Nearest*, *Farthest*, *Cheapest* oder *Random* nach ihrer Entfernung zu den Knoten der aktuellen Tour, den „Kosten“ ihrer Aufnahme in die Tour oder wird dem Zufall überlassen.
Ein Beispiel einer *Metaheuristik*, also einer Heuristik, die nicht auf ein bestimmtes, sondern viele verschiedene Probleme angewendet werden kann, ist die *Ant-System-Heuristik* mit ihren diversen Weiterentwicklungen [DBT00]. Sie orientiert sich an Ameisen-Kolonien, die auf der Suche nach Futter ausschwärmen und auf ihrem Weg einen Duftstoff hinterlassen. Die Ameise mit dem kürzesten Weg erreicht den Ausgangsort als erste und hat ihren doppelt zurückgelegten Weg auch doppelt markiert. Alle weiteren Ameisen lassen sich nun von diesem Duftstoff leiten. Auf das Traveling Salesman Problem übertragen funktioniert dieser Algorithmus wie folgt: Zunächst werden verschiedene Touren auf dem Graphen erstellt. Dabei erhalten alle Kanten kurzer

Touren eine gute Wertung, Kanten längerer Touren eine entsprechend schlechtere. Bei der Generierung neuer Touren werden nun Kanten mit höherer Wertung bevorzugt. Um den Einfluss „alter“, wenig optimierter Touren auf die Erstellung neuer Touren zu verringern, werden Wertungen weit zurückliegender Schritte langsam „vergessen“.

Statt mit neuen Heuristiken zu versuchen, bessere Lösungen zu finden, bieten sogenannte *Verbesserungsverfahren* die Möglichkeit, bestehende Lösungen nachträglich zu optimieren. Eines der ersten solchen Verfahren ist die *k-Opt-Heuristik*. Sie entfernt $k \geq 2$ Kanten aus einer gefundenen Tour und fügt die gleiche Anzahl Kanten nachträglich so wieder ein, dass das Kantengewicht der neuen Kanten insgesamt minimal ist. Sie entfernt, angewandt auf einen Graphen in der euklidschen Ebene, beispielsweise Kreuzungen von Kanten, die nicht Teil einer optimalen Lösung sein können. In ihrer ursprünglichen Form rät Lin in [Lin65] zum 3-Opt-Verfahren, um mit wenig Zeitaufwand möglichst gute Ergebnisse zu erzielen. Dieser Algorithmus wurde von ihm und Kernighan zur *Lin-Kernighan-Heuristik* erweitert, die k nicht starr festlegt, sondern in jedem Iterationsschritt anpasst [LK73]. In der Version *Chained-Lin-Kernighan-Heuristik* wird abwechselnd die Lin-Kernighan-Heuristik zum Auffinden einer (lokal) optimalen Lösung und typischerweise ein zufälliger 4-Opt-Schritt ausgeführt. Auf diese Weise wird versucht, eine global optimale Lösung zu finden. Ein solches Verfahren, das lokale Optima wieder verlässt und nach besseren Lösungen sucht, gehört zur Klasse der *Simulated Annealing*-Verfahren [Coo11].

Als Vertreter weiterer exakter Lösungsverfahren sei die Klasse der *Schnittebenenverfahren* genannt [DFJ54]. Im allgemeinen Fall des Schnittebenenverfahrens wird zunächst der Lösungsraum der ganzzahligen Lösungen durch eine sogenannte *LP-Relaxierung* erweitert, indem alle Ganzzahligkeitsbedingungen aus dem linearen Optimierungsproblem entfernt werden. Auf dieser Menge wird nun eine Lösung gesucht, die im Optimalfall als Lösung des originalen Problems zulässig ist. Ist das nicht der Fall, wird unter den entfernten Nebenbedingungen eine gesucht, die durch die aktuelle Lösung verletzt ist. Diese wird wieder in das Optimierungsproblem aufgenommen und nach einer neuen Lösung gesucht. Diese Schritte werden nun solange wiederholt, bis alle ursprünglichen Nebenbedingungen erfüllt sind. Ist die Lösung schließlich nicht ganzzahlig, kann auf eine Branch-and-Bound- bzw. hier *Branch-and-Cut*-Methode zurückgegriffen werden. Die Nebenbedingungen bezeichnet man in diesem Zusammenhang mit dem Begriff der *Schnittebene*. Er verdeutlicht eine bildliche Vorstellung, die hinter dem Verfahren steht: Hat das Optimierungsproblem n Variablen, dann kann die Menge der zulässigen Lösungen des relaxierten Problems durch einen n-dimensionalen Polyeder dargestellt werden. Alle ganzzahligen Punkte dieses Polyeders stellen nun ganzzahlige Lösungen dar. Mithilfe der Nebenbedingungen werden Teile des Polyeders „herausgeschnitten“ und so der Lösungsraum wieder verkleinert.
Für das Traveling Salesman Problem gibt es verschiedene Relaxierungen, die für ein

solches Schnittebenenverfahren zur Verfügung stehen. Eine davon ist die *degree LP relaxation*, bei der Graphen in der Lösungsmenge enthalten sind, die Subtouren enthalten. Diese werden im Schnittebenenverfahren sukzessive entfernt [DFJ54]. Eine weitere Art der Nebenbedingungen, sind die in [GP79] beschriebenen *Comb Inequalities*, die Anfang der 70er-Jahre dem Schnittebenenverfahren zu einer neuen Hochzeit verholfen haben [Coo11].
Die Liste der entwickelten Verfahren lässt sich beinahe endlos fortsetzen. Die hier beschriebenen sind also nur eine Auswahl der bekannteren und häufig eingesetzten.

Kapitel 3

Darstellung in einer Webapplikation

Das Traveling Salesman Problem ist ein sehr anschauliches mathematisches Problem, da es sich mit der ganz alltäglichen Frage einer optimalen Tourenplanung beschäftigt. Viele Lösungsverfahren lassen sich darüber hinaus sehr gut illustrieren und vermitteln so in kurzer Zeit ihre Funktionsweise. Aus diesen Gründen ist das Problem sehr gut dazu geeignet, vor allem Schülern und Studenten im Rahmen einer Webapplikation grundlegende Konzepte der kombinatorischen Optimierung näherzubringen. Gerade die spielerische Komponente vermittelt dem Benutzer die Schwierigkeit derartiger Probleme, da ihm häufig bereits das Finden einer optimale Tour auf beispielsweise nur 30 Knoten bereits eine schwierig zu lösende Herausforderung bietet.

3.1 Applikationsstruktur

Die gesamte Anwendung ist in zwei Teile geteilt. Der *Client*, typischerweise ein Browser, ist für die Anzeige des Programms zuständig. Er nimmt darüber hinaus Eingaben eines Benutzers entgegen und verarbeitet diese. Demgegenüber steht ein *Server*, der Anfragen des Clients entgegennimmt und intern verschiedene Programme ausführt. In den meisten Fällen wird anschließend eine Antwort des jeweiligen Programms zurück an den Client geleitet. Die Webapplikation wird in diesem Zusammenhang auch als *Front-End* bezeichnet, während die Serveranwendung das *Back-End* darstellt. Dieses *Client-Server-Modell* bietet die Möglichkeit, aufwändige Rechenoperationen auf dem Server durchzuführen und die Anwendung somit auch für portable Endgeräte nutzbar zu machen, die in ihrer Rechen- und Speicherleistung meistens sehr eingeschränkt sind.

3.2 Front-End

3.2.1 Allgemeiner Aufbau

Um im Webangebot der Technischen Universität München einen hohen Grad an Konsistenz zu erhalten, orientiert sich das Layout der Applikation am Interdisziplinären Projekt von Richard Stotz [Sto13], in dem der Bellman-Ford-Algorithmus implementiert wurde. Auch auf technischer Ebene wurden einige Konzepte übernommen um

die verschiedenen Projekte durch eine einheitliche Vorgehensweise leichter wartbar zu machen.
Die Kopfzeile der Website zeigt auf der linken Seite das Bild einer Landkarte von Deutschland, auf der eine kürzeste Rundtour durch alle 107 verfügbaren Städte zu sehen ist. Daneben befinden sich untereinander die Schriftzüge „Traveling Salesman Problem“ und in kleinerer Schriftart „Das Problem des Handlungsreisenden“, sowie das Emblem der Technischen Universität München. Darunter ist eine Navigationszeile angebracht, die eine Tabstruktur aufweist. Die darin enthaltenen Reiter sind von links nach rechts mit den Titeln „Einführung“, „Spiel erstellen“, „Spiel spielen“, „Optimale Tour“, „Beschreibung des Algorithmus“, „Berechnungsschritte“ und „Weiteres“ beschriftet. Je nach Auswahl des Tabs öffnet sich unterhalb der Navigation eine Box, die eines von zwei möglichen, grundlegenden Layouts aufweist. In den meisten Tabs ist sie in zwei Teile geteilt: Im linken Teil werden Inhalte wie Graphen und wichtige Informationen dargestellt, während der rechte Navigations- und Hinweiszwecken dient. Alternativ wird die gesamte Breite zur Darstellung vorrangig textuellen Inhalts genutzt.
Die Anordnung der Tabs ist bewusst gewählt. Der Aktionsablauf des Nutzers verläuft idealerweise linear in der dargestellten Reihenfolge. Nach einer kurzen Heranführung an die Problematik erstellt der Nutzer im zweiten Tab ein Spiel durch Angabe der notwendigen Parameter und versucht anschließend im folgenden Tab eine optimale Rundtour zu finden. Zur Überprüfung seiner eigenen Lösung wechselt er nun in den nächsten Tab, wo er die optimale Tour angezeigt bekommt. Bevor er sich die Berechnungsschritte ansieht, die zum optimalen Ergebnis geführt haben, erhält er im nächsten Tab zunächst die Möglichkeit, sich über die Funktionsweise der Algorithmen näher zu informieren. Im letzten Tab bekommt der Nutzer weitere Informationen zu verschiedenen Themen rund um das Traveling Salesman Problem.
Gleichzeitig wird der Anwender jedoch nicht zu diesem starren Vorgehen gezwungen. Er kann sich (soweit dies inhaltlich sinnvoll ist) frei zwischen den Tabs hin- und herbewegen und sich zum Beispiel nur informieren, ohne das Spiel selbst zu spielen. Andersherum ist es aber genauso möglich, mehrmals ausschließlich Touren zu erstellen und sich die dazugehörigen Lösungen anzeigen zu lassen, ohne sich die Berechnungsschritte anzusehen.
Die Fußzeile bietet schließlich allgemeine Nutzungshinweise des Internetangebots und die Möglichkeit, zwischen den Sprachen Deutsch und Englisch zu wechseln. Neben der Beschreibung der dargestellten Arbeit befinden sich dort auch Links zu Rechtshinweisen, Impressum und ein Mailing-Link zum Systemadministrator.

3.2.2 Aufbau der einzelnen Tabs

Einführung: Im ersten Tab soll der Benutzer an das Thema „Traveling Salesman Problem“ herangeführt werden. Dazu wird das Problem auf allgemeinverständlichem

Niveau beschrieben und so seine Komplexität aufgezeigt. Der Benutzer kann anschließend mittels der ihm dort angebotenen Buttons in den *Spiel erstellen*-Tab wechseln oder sich direkt weiterführende Informationen im *Beschreibung des Algorithmus*-Tab ansehen.

Abbildung 3.1: Kopfzeile der Applikation

Abbildung 3.2: Menü der Applikation beim Start

Spiel erstellen: Hier kann der Benutzer ein neues Spiel konfigurieren oder ein bereits gespieltes Spiel wiederholen.
Zur Konfiguration stehen ihm drei modifizierbare Parameter zur Verfügung: Verschiedene Karten, die Anzahl der Städte auf der Karte und die verwendete Metrik. Mit Auswahl der Karte werden gleichzeitig auch die Auswahlmöglichkeiten der anderen beiden Größen beschränkt. Auf der Karte „Afrika“ beispielsweise sind bis zu 95 Städte und ausschließlich die Euklidsche Metrik verfügbar.
Zum besseren Verständnis der auswählbaren Metriken hält die Infobox auf der rechten Seite zusätzliche Informationen zu ihren Berechnungsvorschriften und entsprechende Bilder zur Veranschaulichung bereit.

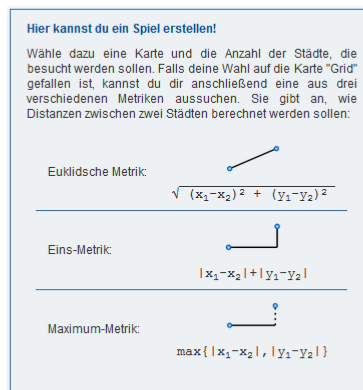


Abbildung 3.3: Beschreibung der verfügbaren Metriken

Spiel spielen: In diesem Tab versucht der Benutzer eine kürzeste Rundtour durch die ihm angezeigten Städte zu zeichnen. Dazu wählt er per einfachem Linksklick eine

Stadt aus, bewegt den Zeiger zur nächsten gewünschten Stadt und klickt ein weiteres Mal. Zwischen beiden Städten wird so eine Kante eingezeichnet und ausgehend von der zuletzt gewählten Stadt befindet sich eine neue Kante in Vorbereitung. Ein Rechts- oder Doppelklick beendet das Zeichnen der aktuellen, unfertigen Kante. Ist keine Stadt ausgewählt und befindet sich der Zeiger über einer bestehenden Kante, kann diese ebenfalls durch Rechts- oder Doppelklick wieder entfernt werden. Die farbliche Füllung der Städte gibt Auskunft darüber, ob an der jeweiligen Stadt bereits genau zwei Kanten anliegen oder nicht. Es ist darüber hinaus nicht möglich, mehr als zwei Kanten an eine Stadt anzulegen. Eine entsprechende Anzeige auf der rechten Seite informiert den Benutzer jederzeit über seine aktuelle Tourlänge.
Hier wird ihm darüber hinaus auch der GameCode angezeigt, den er dazu nutzen kann, dieselbe Spielkonfiguration zu einem späteren Zeitpunkt erneut zu laden.
Bei Bewegung des Zeigers auf eine Stadt wird ihr Name eingeblendet, sofern dies von der Konfiguration vorgesehen ist.
Sobald die eingezeichnete Tour eine Rundtour darstellt, kann auf der rechten Seite ein Button gedrückt werden, der den *Optimale Tour*-Tab öffnet. Ist dies nicht der Fall, kann dieser Tab ausschließlich über die Tableiste geöffnet werden. Vor dem Wechsel wird in diesem Fall auf die unvollständige Tour aufmerksam gemacht mit dem Hinweis, dass nach Verlassen des Tabs im laufenden Spiel nicht mehr in diesen zurückgewechselt werden kann. Der Benutzer soll also nicht die Möglichkeit erhalten, seine Tour nach Bekanntwerden der Lösung zu ändern.
Vor Tabwechsel wird darüber hinaus geprüft, ob auf dem Server die entsprechende Lösung bereit liegt. Ist ihre Berechnung noch nicht abgeschlossen oder wurde sie nach Überschreiten eines bestimmten Zeitfensters abgebrochen, wird der Benutzer über den aktuellen Stand informiert.

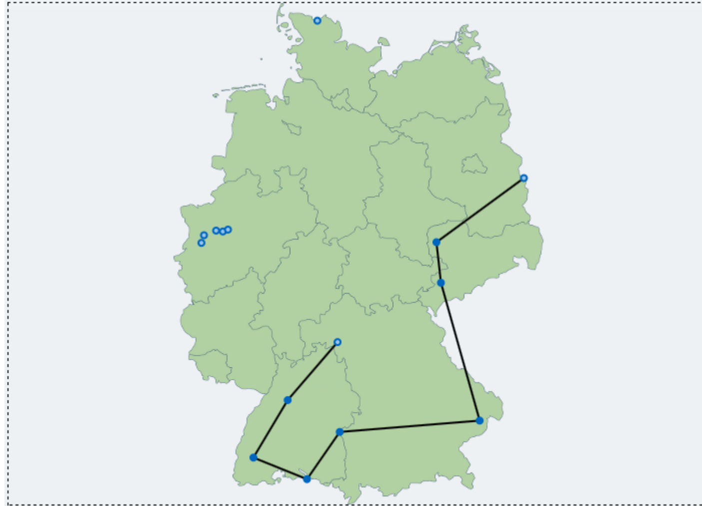


Abbildung 3.4: Ein laufendes Spiel mit angefangener Route

Optimale Tour: Bei der ersten Öffnung des Tabs im laufenden Spiel wird der Benutzer mit der Bewertung seiner erstellten Tour begrüßt. Hier bekommt er neben der Länge seiner eigenen Tour auch die Länge einer optimalen Lösung angezeigt.
Im Tab selbst sieht der Benutzer nun die optimale Tour. Ein Klick auf den „Deine Tour einblenden“-Button auf der rechten Seite legt den zuvor vom Benutzer erstellten Graphen farblich abgehoben unter die Lösung, sodass dieser nun beide Touren vergleichen kann. Auch hier hat der Benutzer Zugriff auf den GameCode und bekommt außerdem die zum Finden der Lösung benötigte Rechenzeit in Zehntelsekunden angezeigt.

Beschreibung des Algorithmus: Zur Vorbereitung auf den folgenden Tab wird dem Benutzer hier die Funktionsweise der verwendeten Algorithmen dargelegt. Ziel dieses Tabs ist das Verständnis der grundlegenden Idee jedes Lösungsverfahrens, ohne durch zu viele Details zu verwirren. Um die Verfahren zu beschreiben, werden zunächst die Begriff *Kante*, *Knoten*, *Graph*, sowie *Heuristik* und *Exaktes Lösungsverfahren* geklärt.

Berechnungsschritte: In diesem Tab wird der Ablauf des gesamten Lösungsalgorithmus am konkreten Beispiel des laufenden Spiels illustriert. Ähnlich dem *Optimale Tour*-Tab befindet sich auf der rechten Seite wieder eine Navigationsbox, aus der sich der linksseitige Graph steuern lässt. In dieser sind alle Berechnungsschritte aufgelistet, die während der Ausführung des Algorithmus zu einer neuen oberen bzw. unteren Schranke geführt haben. Die nebenstehende Zahl gibt die gefundene Schranke an. Mit Klick auf einen der Berechnungsschritte wird der Graph angepasst und die entsprechende Struktur eingeblendet. Die Lagrange-Relaxierung ist hier ein Sonderfall. Der dort angegebene Wert beschreibt nicht die tatsächliche, sondern die gewichtete

Länge des Eins-Baums. Diese resultiert aus der unterschiedlichen Kantengewichtung je Berechnungsschritt, die sich maßgeblich aus dem Lagrange-Multiplikator ergibt. In diesem Zusammenhang sind auch die verschiedenen Größen der „Städte“ zu verstehen. Je höher der Lagrange-Multiplikator des Knotens durch die Anzahl inzidenter Kanten im Algorithmus, desto größer auch der Radius des entsprechenden Knotens. Aus Konsistenzgründen sind auch hier nur die Knoten ausgefüllt, an denen genau zwei Kanten anliegen. Dies ist für den Anwender ein zusätzliches Indiz, an welchen Stellen der Algorithmus im aktuellen Schritt noch Optimierungsbedarf hat. Um das Ergebnis des Nearest-Neighbor-Verfahrens schneller nachvollziehen zu können, ist bei Auswahl des korrespondierenden Berechnungsschritts der Startknoten des Algorithmus rot dargestellt.
Für eine bessere Benutzerfreundlichkeit kann der Anwender die Berechnungsschritte alternativ mit den Navigationsbuttons im oberen Teil der Box steuern ohne seinen Blick vom Graphen nehmen zu müssen. Um aktuellen Schritt und fertige Lösung optisch schnell vergleichen zu können, bietet die Oberfläche auch hier die Möglichkeit, sich den Lösungsgraphen farblich verschieden anzeigen zu lassen.

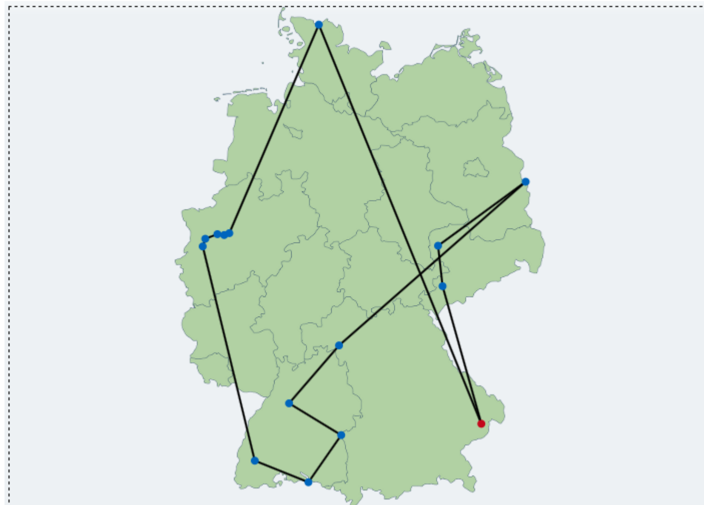


Abbildung 3.5: Das Ergebnis des Nearest-Neighbor-Verfahrens mit rotem Startknoten

Abbildung 3.6: Gefundene Lösungen mit Angabe ihrer (gewichteten) Länge

Weiteres: Im letzten Tab erhält der Nutzer zusätzliche Informationen zu den Themen „Geschichte des Traveling Salesman Problems“, „Die Applikation“ und „Mathestudium an der TUM“.

3.2.3 Technische Umsetzung

Der initiale Zugriff des Benutzers auf die Webapplikation erfolgt in der **Traveling-Salesman-Problem.html**, die als einzige HTML-Datei das Grundgerüst des Front-Ends bildet. Sie bestimmt die Struktur der Internetseite und importiert alle erforderlichen JavaScript-Methoden und CSS-Dateien.

Eine aktuelle jQuery-JavaScript-Bibliothek wird in Form der **jquery-1.10.2.js** eingebunden. Sie ermöglicht auf sehr einfache Weise Manipulationen am Document Object Model (DOM). Das in **jquery-ui-1.10.4.custom.min.js**, sowie **jquery-ui-1.10.4.custom.min.css** definierte jQuery User Interface versieht bestimmte HTML Elemente mit einem neuen Layout. Insbesondere wird darin die Optik der Buttons und Tabs generiert. Das User Interface wurde zuvor im sogenannten ”ThemeRoller” des jQuery-UI-Projekts konfiguriert[1]. Nachträgliche Änderungen können sehr einfach über den im Kommentar der `jquery-ui-1.10.4.custom.min.css` angegebenen Pfad vorgenommen werden. Alle drei vorangehenden Dateien sind im Rahmen der ”MIT-Lizenz” verfügbar, eine Kopie befindet sich im Ordner `tum-theme2`.

Das kaskadierende Stylesheet **style.css** beschreibt alle weiteren Website-spezifischen Layoutregeln.

In der **config.js**-Datei sind zusätzliche Parameter wie die URL und der Port hinterlegt, unter der der Server zu erreichen ist. Auch die Anzahl Städte, die maximal für ein Spiel ausgewählt werden kann, wird hier festgelegt. Darüber hinaus kann mithilfe

[1]http://jqueryui.com/themeroller/

der Variable `language` die Standardsprache gesetzt werden. Derzeit kann zwischen Deutsch (`de`) und Englisch (`en`) gewählt werden.

Die **siteAnimation.js**-Datei ist Dreh- und Angelpunkt aller durch JavaScript indizierten Manipulationen an der Oberfläche. Zunächst wird die Methode `initializeSiteLayout` aufgerufen, die Elemente wie Tabs, Buttons und Menüs in jQuery-UI-Objekte umwandelt und so die Anpassung ihres Layouts anstößt. Auch werden hier viele der Texte in die Oberfläche eingefügt. Dies wurde in JavaScript umgesetzt, da dort der direkte Zugriff auf den Inhalt der durch das jQuery-UI manipulierten HTML-Tags leichter fällt. Das JSON-Objekt `nameMappings` enthält zu diesem Zweck sämtliche Texte in den Sprachen Deutsch und Englisch und kann sehr einfach durch weitere Sprachen ergänzt werden. Für längere Texte und einige andere Elemente wird die Anzeige der gewählten Sprache über die CSS-Selektoren `.languageDE` bzw. `.languageEN` gesteuert. Anschließend werden mithilfe einer entsprechenden Methode aus der `loadUtilities.js` die verfügbaren Spielparameter in den *Spiel erstellen*-Tab geladen. Zuletzt wird das Verhalten bei Tabwechseln festgelegt. Im `activate`-Handler der Tabs wird die jeweils erforderliche der "GraphDrawer"-Klassen instanziert, die weiter unten näher beschrieben sind, und diese als `data`-Attribute im HTML-Element des Tabs abgelegt. Dabei wird ihre `run`-Methode ausgeführt. Im `beforeActivate`-Handler wird die aktuelle "GraphDrawer"-Instanz des Quelltabs gestoppt, da sie für den Zieltab keine Funktion hat und dort definierte Verhaltensweisen sogar "überlagern" könnte.
Klicks auf den *Spiel starten*-Button werden ebenfalls in der `siteAnimation.js` abgefangen. Dazu werden die bestehenden Graphen des potenziell ausgeführten vorherigen Spiels mitsamt seiner geladenen Lösung gelöscht und je nach Auswahl ein neues Spiel generiert und geladen. Alternativ wird die Lösung zu einem bereits gespielten Spiel aus dem Back-End geholt und in das `data`-Attribute des `body`-Tags geschrieben. Die beiden dazugehörigen Lade-Methoden werden im weiteren Verlauf dieses Abschnitts vorgestellt.

Die Klasse `CanvasDrawer` ist in der **canvasDrawer.js** beschrieben. Sie enthält einen Grundstock an Funktionen, die im Zusammenhang mit allen Tabs benötigt werden, die das HTML 5-Element `canvas` nutzen. Im Wesentlichen fügt sie bei Aufruf das Hintergrundbild ins Canvas ein und zeichnet bei entsprechenden Nutzeraktionen die ihm übergebenen Graphen neu. Beginnt der Benutzer mit der Erstellung einer Kante, so wird der gesamte Graph im Takt von zwanzig Millisekunden neu gezeichnet um beim Anwender den Eindruck zu erzeugen, er „ziehe“ die Kante von einem Knoten zum nächsten.
Die Anzeige der Städtenamen beim Platzieren der Maus über einem Knoten wurde ebenfalls hier umgesetzt, da diese für jeden Tab identisch erfolgen soll. Bei Aufruf und Verlassen des aktiven Tabs können über (CSS-)Selektoren festgelegte HTML-Blöcke als Dialog geladen werden. Der `tabIntroDialog` wird einmalig beim Öffnen eines Tabs geladen um eine kurze Einführung in den neuen Tab zu bieten. Der `tabChangeWarningDialog`

wird hingegen nur dann beim Tabwechsel angezeigt, wenn er erforderlich ist. Im Fall des *Spiel spielen*-Tabs beispielsweise wird ein solcher Warndialog geladen um darauf hinzuweisen, dass nach Verlassen des Tabs dieser im laufenden Spiel nicht erneut geöffnet werden kann.

Eine vom `CanvasDrawer` erbende Klasse ist der `GraphDrawer` (zu finden in der **graphDrawer.js**), der im Tab *Spiel spielen* alle notwendigen Methoden zur Interaktion des Anwenders mit der Oberfläche bereitstellt. Dazu gehören alle Mouse-Handler, die das Positionieren der Maus über einer Stadt erkennen und das Zeichnen einer Kante initialisieren oder eine Kante im Graphen festschreiben. Auch das Löschen von Kanten wird in dieser Klasse verarbeitet. Eine Überprüfung der Grade aller Knoten ermöglicht das Aktivieren des *Lösung anzeigen*-Buttons, sowie gegebenenfalls den Hinweis bei Verlassen des Tabs, dass die eingezeichneten Kanten keine Tour beschreiben.

In der **solutionGraphDrawer.js** wird eine weitere Klasse definiert, die vom `CanvasDrawer` erbt. Sie wertet bei Eintritt in den *Optimale Tour*-Tab die vom Anwender eingezeichnete Tour aus, indem sie ihre Länge mit der Länge der optimalen Tour vergleicht. Ein Dialog informiert den Spieler über seinen Erfolg. Über eine entsprechende Methode kann der Superklasse mitgeteilt werden, dass die Kanten eines zweiten Graphen, in diesem Fall des vom Benutzer erstellten, unter den Lösungsgraph gelegt werden sollen.

Für den *Berechnungsschritte*-Tab wird die Klasse `CalculationGraphDrawer` aus der **calculationGraphDrawer.js** instanziert, die zunächst alle Berechnungsschritte in die Navigationsbox schreibt und dort auch ihre Indizes hinterlegt. Der erste Schritt wird initial über die Methode `updateEdgesFromSolution` geladen, die als Parameter den Index des gewünschten Berechnungsschritts akzeptiert. Sie verändert die Knotenradien und überschreibt die Kanten im Graph-Objekt.

Alle Methoden, die physisch auf das Canvas zeichnen, sind in der **canvasUtilities.js** statisch in der gleichnamigen Klasse definiert. Dies sind zum einen `drawNode`, die einen Knoten zeichnet und zum anderen `drawEdge`, die je nach Metrik in verschiedener Weise Kanten in das Canvas einfügt.

Die Schnittstelle zwischen `CanvasDrawer` und den Zeichenbefehlen der `CanvasUtilities` stellt gewissermaßen die **canvasElements.js**-Datei dar. Hier werden alle Elemente, die auf dem Canvas dargestellt werden, auf JavaScript-Objektebene definiert. Diese sind zunächst der Graph als Ganzes (`Graph`) und darunter seine Knoten (`Node`) und Kanten (`Edge`).
Die Klasse `Node` nimmt die "physischen" Koordinaten entgegen und berechnet mithilfe der hinterlegten Randkoordinaten der Karte und dem Abstand des Kartenbilds vom Rand der Client-Anwendung die Position des Knotens auf dem Canvas. Zu beachten

ist bei der Umrechnung, dass im Fall der Verwendung von Kugelkoordinaten je nach Projektion der Karte unterschiedliche Transformationsvorschriften zu verwenden sind. In der aktuellen Version sind zwei verschiedene Projektionen implementiert. Um die Umrechnungsformeln anzugeben sei die geographische Breite durch ϕ und die geographische Länge durch λ im Bogenmaß gegeben.

- Plattkarte/Rektangularprojektion:
  $x = \lambda$
  $y = \phi$
- Mercator-Projektion:
  $x = \lambda$
  $y = \operatorname{arcsinh}(\tan\phi) = \ln\left(\tan(\phi) + \sqrt{\tan^2(\phi) + 1}\right)$

Alle Methoden, die Daten aus dem Back-End laden, sind in der **loadUtilites.js** zusammengefasst. Diese sind `checkCalculationFinished`, die den Status der aktuellen Berechnung abfragt, sowie `loadSolution`, die anschließend eine Lösung aus dem Back-End anfordert und in ein `solution`-Objekt speichert. Darüber hinaus werden mithilfe der Methode `loadAvailableSettings` mögliche Konfigurationsparameter geladen und im `mapConfig`-Objekt abgelegt. Als letzte verfügbare Methode lädt `loadNewGame` die Spielparameter eines neuen Spiels aus dem Back-End und speichert sie in ein `gameConfig`-Objekt.

Die **metrics.js** enthält die Implementierung aller auswählbarer Metriken. Jede dieser Metriken muss in der Methode `registerMetrics` registriert werden und selbst in einer separaten Klasse definiert sein. Diese muss die drei in Tabelle 3.1 dargestellten Methoden zur Verfügung stellen. Um die korrekte Funktion der Metriken zu garantieren, müssen sie mit den im Back-End implementierten Metriken konsistent sein. Insbesondere ihr Registrierungsname und die `calculateDistance`-Methode dürfen sich nicht von dieser Implementierung unterscheiden.

3.3 Back-End

3.3.1 CherryPy-Server

Die Kommunikation zwischen Front- und Back-End-Anwendungen übernimmt ein CherryPy-Server. Der auf der Programmiersprache Python basierende Server ist mit knapp 2,3 MB sehr schlank und wegen seiner Beschränkung auf die nötigsten Funktionen schnell einzurichten. Die Serveranwendung selbst wird mit Ausführung des Skripts `Tsp_CherryPy.py` gestartet und ist anschließend auf dem in der `Tsp_Config.conf`-Konfigurationsdatei deklarierten Pfad `backend_path` verfügbar. Sie hält drei Methoden

bereit, die vom Front-End per `GET` erreichbar sind.
Beim Laden der Website wird über die URL

```
<backend_path>/tsp/availableSettings
```

eine erste Serveranfrage eingereicht, die nach den derzeit verfügbaren Landkarten fragt. Der CherryPy-Server führt daraufhin die parameterlose Anwendung `AvailableSettings.jar` aus, die für jede Karte die drei Eigenschaften

- Kartenname
- Anzahl der verfügbaren Städte
- Verwendbare Metriken

ausgibt. Verschiedene Karten sind durch ein Rautenzeichen (`#`) getrennt. Eine exemplarische Rückgabe ist

```
Afrika|95|Euklidsche Metrik#Italien|107|Euklidsche
Metrik#Deutschland|107|Euklidsche Metrik#Grid|100|all#
```

Die Anwendung `TspApp.jar` bildet das Kernstück der gesamten Applikation. Sie nimmt die vom Endanwender festgelegten Parameter Kartenname, Anzahl der Städte und eine ausgewählte Metrik entgegen, liefert erste für das Spiel notwendige Informationen an den CherryPy-Server und stößt die interne Berechnung der kürzesten Rundtour an. Dazu verarbeitet der CherryPy die `GET`-Anfrage

```
<backend_path>/tsp/initiateCalculation/<Kartenname>/<Anzahl
Städte>/<Metrik>
```

und wandelt sie in den Kommandozeilenbefehl

```
java -jar TspApp.jar <Kartenname> <Anzahl Städte> <Metrik>
```

um.
Die Antwort ist nun abhängig davon, ob die von der Anwendung generierte Spielkonfiguration bereits früher schon einmal gelöst und anschließend gespeichert wurde. Sie besteht aus den in Tabelle 3.2 dargestellten Komponenten, von denen nur die ersten beiden in jedem Fall, also unabhängig von der Existenz früherer Lösungen zurückgegeben werden. Auch hier werden die Informationen durch ein Rautezeichen (`#`) getrennt. Die Ergebnisse beider Anfragen werden jeweils von CherryPy aus der Kommandozeile ausgelesen und direkt an das Front-End weitergeleitet.
Dem GameCode kommt in der gesamten Applikation eine besondere Bedeutung zu. Er setzt sich aus der Kartenkennung, der Anzahl Städte und dem Code der Metrik zusammen. Anschließend wird die Ziffer `0` und ein Hash angehängt. Dieser entsteht

durch das Aneinanderreihen der (dreistelligen) Stadtindizes in aufsteigender Reihenfolge und der anschließenden Berechnung des MD5-Hashwerts. Die Verwendung des stets 32-stelligen Hashs hat zur Folge, dass selbst Spiele mit über 100 Städten einen GameCode bekommen, der gegebenenfalls sogar manuell ohne übermäßigen Zeitaufwand eingegeben werden kann. In erster Linie ermöglicht aber die (Pseudo-)Eindeutigkeit der Zuordnung der ausgewählten Städte zum GameCode das Wiederverwenden von vorhandenen Lösungen bei neu generierten Spielen.
Der CherryPy führt intern eine Liste aller in der aktuellen Server-Sitzung angestoßenen Berechnungen, die jedem GameCode einen Status zuordnet. Dieser ist zu Beginn `0`. Sobald die laufende Berechnung über die offene Verbindung ein `FINISHED` sendet, wird dieser Status auf `1` gesetzt. Läuft die Berechnung nach einer bestimmten Zeit (derzeit 60 Sekunden) immernoch, wird die Berechnung abgebrochen und ihr Status auf `2` gesetzt.
Über die Anfrage

`<backend_path>/tsp/checkCalculationFinished/<GameCode>`

kann dieser Status abgefragt werden.
Anschließend kann mihilfe des GameCodes über die `GET`-Anfrage

`<backend_path>/tsp/fetchSolutionFile/<GameCode>`

auch die Lösung vom CherryPy aus dem Back-End geholt und an den Client übertragen werden. Dies geschieht in Form einer entsprechenden `<GameCode>.txt`-Datei, die mit Ende der Berechnung in den Ordner `\solutions\` geschrieben wurde. Dabei kann es sich entweder um eine unmittelbar zuvor erstellte oder im Zuge einer früheren Berechnung bereits gespeicherte Lösung handeln. Sie enthält die folgenden Attribute:

- Kartenname
- Metrik
- Verwendete Längeneinheit
- Anzeige der Städtenamen (`true`/`false`)
- Randkoordinaten der Karte
- Kugelkoordinaten (`true`/`false`)
- Verwendete Städte
- Benötigte Rechenzeit in Zehntelsekunden
- Optimale Tourlänge

- Optimale Tour
- Berechnungsschritte in der Form <Art der Schranke>|<Algorithmus>|<Länge des Graphen>|<Knotenradien>|<Kanten>

3.3.2 Karten

Alle auswählbaren Karten liegen in einer bestimmten Datenstruktur im Back-End und werden zu Beginn jeder Anwendungsausführung von der Klasse MapManager neu ausgelesen, der anschließend für jede Karte eine Map-Instanz erstellt. Neben dem Eintrag des (kleingeschriebenen) Kartennamens in der Datei \maps\maps.txt müssen zur erfolgreichen Konfiguration drei weitere Dateien im Ordner \maps\<Kartenname>\ vorliegen.
In der config.txt-Datei werden grundlegende Konfigurationsparameter festgelegt, die in Tabelle 3.3 aufgeführt sind. Die Städte der Karte werden in der separaten Datei towns.txt konfiguriert, die zusammen mit der config.txt zum Zeitpunkt der Instanzierung der Klasse Map ausgelesen und gespeichert werden. Dort findet sich für jede Stadt ein Eintrag der Form

<Name>|<Vertikale Koordinate>|<Horizontale Koordinate>|<Ist bevorzugt>

Die konkrete Ausprägung der Koordinaten ist dabei abhängig von der Art der Karte. Landkarten verwenden Kugelkoordinaten zur Angabe von Breiten- bzw. Längengrad, während ebene Karten wie das implementierte Gitter Grid einfache y-x-Koordinaten benötigen. Zudem müssen sie dezimal vorliegen, Nachkommastellen beschreiben also im Fall der Kugelkoordinaten keine Minuten, sondern Zehntelgrad.
Die letzte Komponente enthält schließlich die boolschen Werte 0 oder 1 und bewirkt die bevorzugte Auswahl der jeweiligen Stadt bei Initialisierung des Spiels. Die bevorzugten Städte sind dabei so gewählt, dass sie zu einer guten Streuung über die Karte führen. Kurze Touren in einem kleinen Bereich der Karte werden so vermieden.
Die Grafik selbst muss im Front-End unter der Adresse /maps/<Kartenname>.png gespeichert sein und darf eine Größe von 700x500 Pixeln nicht überschreiten. Bei der Auswahl der passenden Grafik für Landkarten muss außerdem darauf geachtet werden, dass die verwendete Projektion bekannt ist. Der Grund dafür ist die clientseitige Umrechnung der Kugelkoordinaten in Ebenenkoordinaten (siehe Abschnitt 3.2.3). Die bestehenden Landkarten benutzen Bilddateien der Wikimedia Commons-Datenbank, deren Lizenzbestimmungen eine derartige Verwendung erlaubt.
Aufgrund der geringen Anzahl erforderlicher Anpassungen ist die Integration neuer Karten in die Applikation also sehr einfach und kann schnell durchgeführt werden.

3.3.3 Metriken

Zur Berechnung der Strecke zwischen zwei Städten können drei verschiedene Metriken verwendet werden, deren Eigenschaften in Tabelle 3.4 beschrieben sind.
Auf Serverseite sind die Metriken in den Klassen `EuclideanMetrics`, `MaximumMetrics`, sowie `OneMetrics` implementiert und werden vom `MetricsManager` verwaltet. Dieser stellt verschiedene Methoden für den Zugriff auf die Metriken zur Verfügung.
Das Hinzufügen neuer Metriken im Back-End ist ebenfalls sehr einfach möglich. Die neue Klasse `<Name>Metrics` muss von der `Metrics`-Klasse erben und eine konkrete Implementierung der `getDistance`-Methode enthalten. Abschließend wird sie manuell im `MetricsManager` registriert. Die notwendigen Schritte im Front-End sind in Abschnitt 3.2.3 und Tabelle 3.1 beschrieben.

3.3.4 Berechnung der Lösung

Sind alle erforderlichen Parameter gesetzt und ist die Lösung nicht bereits bekannt, startet die Anwendung `Tsp_App` die `Calculation`-Routine. Diese stößt nacheinander verschiedene Algorithmen an, die alle in der Klasse `TSPProcedures` definiert sind.
Zur Bestimmung einer oberen Schranke führt die Methode `calculateUpperBound` zunächst den Nearest-Neighbor- (`nNeighbour`) und anschließend den Multiple-Fragment-Algorithmus (`greedy`) aus. Beide sind in den vorangehenden Kapiteln in Abschnitt 2.1.1 bzw. Abschnitt 2.1.2 näher beschrieben.
Die Berechnung einer möglichst optimalen unteren Schranke ist dagegen aufwändiger. Sie beginnt mit dem Aufruf der Lagrange Relaxation (`lagrangeRelaxation`), die in Abschnitt 2.2.1 dargestellt wurde. Wird auf diese Weise keine optimale Rundtour gefunden, geht das Programm in den Branch-and-Bound-Algorithmus (`branchAndBound`) über. Dabei generiert es mithilfe der Branchingregel von Volgenant und Jonker aus Abschnitt 2.2.2 einen Branching-Baum, der nach dem Prinzip der Tiefensuche durchwandert wird.

3.4 Erweiterungsmöglichkeiten

Die Applikation ist bewusst so gestaltet, dass sie auf einfache Weise erweitert werden kann. Das Hinzufügen von Karten und Metriken wurde in den vorangegangen Abschnitten bereits auf Server- und Client-Seite erläutert. Man kann sich in der Applikation auch Stadtkarten vorstellen, in denen der Benutzer nach optimalen Lösungen sucht. Interessant wäre diesbezüglich beispielsweise das Anwenden der Eins-Metrik auf dem Stadtplan Manhattans.
Da im Back-End alle Algorithmen zentral aus der `Calculation`-Klasse ausgeführt werden, können dort ohne großen Aufwand weitere Verfahren „eingehängt“ werden, deren Darstellung sich für die Applikation eignen würde.

Auch das Front-End ist leicht erweiterbar. Zusätzliche Algorithmen bedürfen zur Darstellung fast ausschließlich Änderungen im `CalculationGraphDrawer`.

`calculateDistance` - Berechnung der Kantenlänge	
Eingabe:	Quellkoordinaten
	Zielkoordinaten
Ausgabe:	Berechnete Kantenlänge

`drawEdge` - Zeichnen einer Kante	
Eingabe:	Kontext des `canvas`
	Layout-Objekt der Kante
	Quellkoordinaten
	Zielkoordinaten

`contains` - Lage übergebener Koordinaten zu einer Kante	
Eingabe:	x-Koordinate
	y-Koordinate
	Quellkoordinaten
	Zielkoordinaten
	Layout-Objekt der Kante
Ausgabe:	Ist Koordinate im Bereich der Kante? (`true`/`false`)

Tabelle 3.1: Die drei bereitgestellten Methoden einer Metrik-Klasse

<table>
<tr><th>Datentyp</th><th>Erklärung</th></tr>
<tr><td>Boolean</td><td>Flag, das die Existenz einer früheren Lösung indiziert</td></tr>
<tr><td>String</td><td>GameCode, der sich eindeutig einer Spielkonfiguration zuordnen lässt</td></tr>
<tr><td>String</td><td>Verwendete Längeneinheit: km oder pixel</td></tr>
<tr><td>Boolean</td><td>Flag, das die Anzeige der Städtenamen auf der Spieloberfläche steuert</td></tr>
<tr><td>String</td><td>Randkoordinaten der Karte in Grad oder Pixeln in der Form <"Nördliche" Grenze>|<"Südliche" Grenze>|<"Westliche" Grenze>|<"Östliche" Grenze></td></tr>
<tr><td>Boolean</td><td>Flag, das die Art der Koordinaten beschreibt: true für Kugelkoordinaten in Grad, false für Ebenenkoordinaten in Pixeln</td></tr>
<tr><td>String</td><td>Ausgewählte Städte</td></tr>
<tr><td>String</td><td>STOP: Steuerbefehl für den CherryPy</td></tr>
</table>

Tabelle 3.2: Komponenten der Serverantwort

Parameter	Wert
MapName	Name der Karte
MapCode	Schlüssel der Karte, der unter anderem zur Generierung des GameCodes verwendet wird
ForceMetrics	*(optional)* Einzig erlaubte Metrik
ShowCityNames	Flag, das die Anzeige der Städtenamen auf der Spieloberfläche steuert: true oder false
LengthUnit	Verwendete Längeneinheit: km oder pixel
Projection	Verwendete Kartenprojektion: mercator oder rectangular
PhysicalBorderN	"Nördliche" Randkoordinate der Karte in Grad oder Pixeln
PhysicalBorderS	"Südliche" Randkoordinate der Karte in Grad oder Pixeln
PhysicalBorderW	"Westliche" Randkoordinate der Karte in Grad oder Pixeln
PhysicalBorderE	"Östliche" Randkoordinate der Karte in Grad oder Pixeln

Tabelle 3.3: Parameter der config.txt

Metrik	Rechenvorschrift	Verwendung
Euklidsche Metrik	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$	In allen Karten
Maximum-Norm	$\max\{\lvert x_1 - x_2\rvert, \lvert y_1 - y_2\rvert\}$	In der Karte Grid
1-Metrik	$\lvert x_1 - x_2\rvert + \lvert y_1 - y_2\rvert$	In der Karte Grid

Tabelle 3.4: Die verschiedenen Metriken

Kapitel 4

Ausblick

Nach mathematischer Betrachtung der verschiedenen Lösungsalgorithmen wurde in den vorangehenden Kapiteln versucht, die Grundlage für eine Webapplikation zu legen, die einfache Lösungsalgorithmen anschaulich darstellt und für zukünftige Projekte leicht erweiterbar ist.
Mithilfe neuerer Technologien konnte dieses Ziel umgesetzt werden. Sie ist jetzt dafür ausgelegt, sich dynamisch zu verändern und sukzessive weiter zu wachsen. Unter den in Abschnitt 2.3 dargestellten Algorithmen befinden sich auch solche, die für eine Darstellung im Browser geeignet sind, wie beispielsweise das k-Opt-Verfahren. Es ist aber durchaus auch denkbar, kompliziertere Algorithmen darzustellen, beispielsweise indem auch die Oberfläche verändert wird.
Die Applikation kann so gewissermaßen dem Weg der Forschung rund um das Traveling Salesman Problem nachgehen, dessen „Fangemeinde“ unaufhörlich versucht, bessere und vor allem schnellere Lösungen zu finden und die Anzahl effizienter Algorithmen dadurch zu erweitern.

Literatur

[Ben92] J. L. Bentley. „Fast Algorithms for Geometric Traveling Salesman Problems“. In: *ORSA Journal of Computing* 4-4 (1992), S. 387–411.

[Cay89] A. Cayley. „A theorem on trees“. In: *The quaterly journal of pure and applied mathematics* 23 (1889), S. 376–378.

[CN78] G. Cornuéjols und G. L. Nemhauser. „Tight bounds for Christofides’ traveling salesman heuristic“. In: *Mathematical Programming* 14 (1978), S. 116–121.

[Coo11] W. J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation.* Princeton University Press, 2011.

[DBT00] M. Dorigo, E. Bonabeau und G. Theraulaz. „Ant algorithms and stigmery“. In: *Future Generation Computer Systems* 16 (2000), S. 851–871.

[DFJ54] G. Dantzig, D. R. Fulkerson und S. M. Johnson. „Solution of a large-scale traveling-salesman problem“. In: *Operations Research* 2 (1954), S. 393–410.

[Die10] R. Diestel. *Graphentheorie, 4. Auflage.* Heidelberg: Springer, 2010.

[Flo56] M. M. Flood. „The Traveling-Salesman Problem“. In: *Operations Research* 4.1 (1956), S. 61–75.

[Fri79] A. M. Frieze. „Worst-case analysis of algorithms for Travelling Salesman Problems“. In: *Methods of Operations Research* 32 (1979), S. 93–112.

[GP79] M. Grötschel und M. W. Padberg. „On the symmetric travelling salesman problem I: inequalities“. In: *Mathemtical Programming* 16 (1979), S. 265–280.

[HK71] M. Held und R. M. Karp. „The Traveling-Salesman Problem and minimum spanning trees: Part II“. In: *Mathematical Programming* 1 (1971), S. 6–25.

[HW04] C. A. J. Hurkens und G. J. Woeginger. „On the nearest neighbor rule for the traveling salesman problem“. In: *Operations Research Letters* 32 (2004), S. 1–4.

[HWC74] M. Held, P. Wolfe und H. P. Crowder. „Validation of subgradient optimization“. In: *Mathematical Programming* 6 (1974), S. 62–88.

[Kar72] R. M. Karp. „Reducibility among combinatorial problems“. In: *Complexity of Computer Computations* (1972), S. 85–103.

[Law+85] E. L. Lawler, J. Lenstra, A. H. G. Rinnooy Kan und D. B. Shmoys. *The Traveling Salesman Problem*. Chichester: Wiley, 1985.

[Lin65] S. Lin. „Computer solutions for the travelling salesman problem". In: *The Bell system technical journal* 44 (1965), S. 2245–2269.

[LK73] S. Lin und B. W. Kernighan. „An effective heuristic algorithm for the travelling-salesman problem". In: *Operations Research* 21 (1973), S. 498–516.

[Men32] K. Menger. „Das Botenproblem". In: *Ergebnisse eines mathematischen Kolloquiums 2*. Leipzig: Teubner, 1932, S. 11–12.

[RSI77] D. J. Rosenkrantz, R. E. Stearns und P. M. L. II. „An analysis of several heuristics for the traveling salesman problem". In: *SIAM Journal on Computing* 6 (1977), S. 563–581.

[Sto13] R. Stotz. *Der Bellman-Ford-Algorithmus*. `http://www-m9.ma.tum.de/material/spp/Bellman-Ford-Algorithmus.html`. 2013.

[VJ82] T. Volgenant und R. Jonker. „A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation". In: *European Journal of Operational Research* 9 (1982), S. 83–89.

[Voi31] B. F. Voigt. *Der Handlungsreisende wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein. Von einem alten Commis-Voyageur.* Illmenau, 1831.