TSP-Solver

Dokumentation

von Patrick Bassner

Karlsfeld, April 2018

Inhaltsverzeichnis

1 Einleitung

Der TSP-Solver ist ein Programm, dass das Problem des Handlungsreisenden für bis zu 11 beliebige Standorte in kurzer Zeit bzw. für bis zu 13 Standorte in akzeptabler Zeit löst. Es wurde im Rahmen eines Programmierwettbewerbs ("Codecompetition") von IT-Talents.de erstellt und wurde dort zur Bewertung eingereicht. Der TSP-Solver führt die Berechnung anhand von den durch die Google Maps Geocoding API erfassbaren Adressen durch, sofern zwischen diesen eine Routenberechnung möglich ist. Hierbei wird als Abstandsmaß nicht die reine Entfernung, sondern der Stromverbrauch eines Elektro-LKW verwendet, der abhängig von der Steigung der Strecke variieren kann.

2 Bedienung und Oberfläche

2.1 Übersicht

Die Weboberfläche des Programms tritt wie in der folgenden Abbildung gezeigt auf.

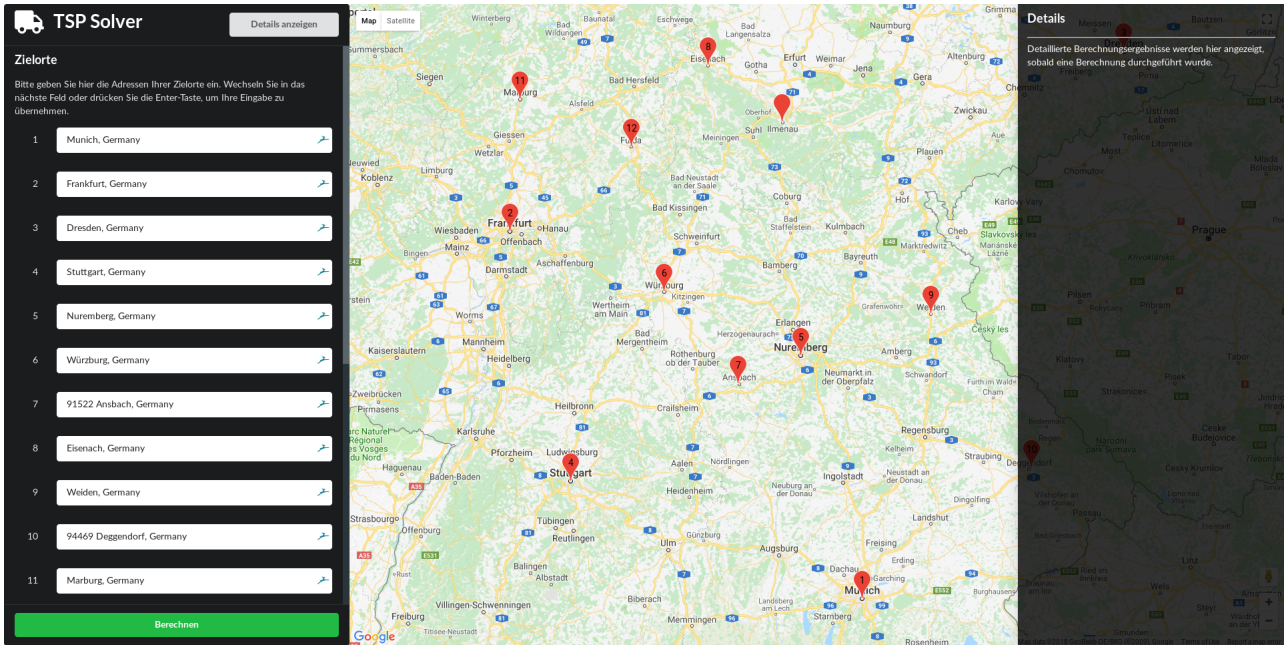

Abb. 1: **Die normale Oberfläche des Programms mit leerer Detailanzeige und ohne Route. Die Addressfelder sind mit Beispielwerten gefüllt.**

2.2 Kartenansicht

In der Mitte der Oberfläche befindet sich die Karte. Diese zeigt alle eingegebenen Standorte mit farbigen, mit der entsprechenden Zahl beschrifteten Markierungen und ggf. auch die berechnete Rundreise zwischen diesen an, welche als hellblauer Pfad eingezeichnet wird. Die Karte skaliert automatisch, um alle Punkte anzuzeigen und den mittleren Viewport vollständig auszufüllen.

2.3 Eingabemaske

Der dunkle Bereich auf der linken Seite enthält alle Steuerungselemente. Sie können hier bis zu 13 Standorte angeben. Beachten Sie, dass sich die Berechnungszeit bei Verwendung von 12 oder 13 Standorten auf bis zu 70 Sekunden erhöhen kann.
Scrollen Sie herunter, um die Energieeinstellungen Ihres Elektro-LKW zu modifizieren. Sie können folgende Einstellungen vornehmen:

- Ob der Zusatzverbrauch bzw. die Einsparung linear interpoliert werden soll (siehe Abschnitt 3.2).
- Den Grundenergieverbrauch Ihres LKW in kWh/km

- Den Steigungsgrenzwert, ab dem der LKW mehr Energie verbraucht, sowie den Mehrverbrauch an sich (in Prozent).
- Den Gefällegrenzwert, ab dem der LKW weniger Energie verbraucht, sowie die Einsparung an sich (in Prozent). Beachten Sie, dass auch hier positive Werte verwendet werden müssen, d.h. für eine Einsparung von 10% bei 3% Gefälle geben Sie bitte positive Werte anstelle von -10% bzw. -3% ein.

Beachten Sie, dass das Ändern einer Adresse auch die Kartenanzeige aktualisiert, sobald das Eingabefeld den Fokus verliert oder Sie `Enter` drücken. Wollen Sie weniger als 11 Adressen in dieser Rundreise verwenden, so lassen Sie die übrigen Felder einfach leer.
Oben neben dem Programmtitel finden Sie einen Button, der die Detaileinblendung aufklappt. Ganz unten finden Sie einen Button, um die Berechnung zu starten. Startet die Berechnung nicht, wenn Sie diesen Button drücken, ist vermutlich eine Ihrer Eingaben fehlerhaft. Das Programm weist sie durch geeignete Animationen auf das fehlerhafte Feld hin. Bitte beachten Sie inbesondere, dass

- mindestens zwei Adressfelder ausgefüllt sein müssen
- kein Adressfeld ausgefüllt ist, dessen Vorgänger nicht ausgefüllt ist (Ausnahme: Feld 1)
- alle Energiewerte ausgefüllt sind

2.4 Detailbereich

Über den Button in der Eingabemaske ganz oben rechts können Sie den Detailbereich einblenden. Dieser enthält einige Zusatzinformationen über die durchgeführte Berechnung, sobald eine solche druchgeführt worden ist.

Insbesondere finden Sie hier die genaue Pfadangabe im Format:

<Adress-Index> @ <Höhe der Adresse über NN> –[<Distanz zwischen den beiden Adressen> / <Stromverbrauch zwischen den beiden Adressen>]–> <Adress-Index der nächsten Adresse> ... usw.

3 Technische Details und Algorithmen

Das Programm ist in Java unter Verwendung von Spring Boot programmiert worden. Für den Aufbau der Oberfläche wurde eine HTML-Seite mit JavaScript-Unterstützung (inkl. jQuery) unter der Verwendung von SemanticUI-Komponenten verwendet. In den folgenden Abschnitten soll nun etwas näher auf die Backendfunktionalität eingegangen werden.

3.1 Verwendung von Google Maps APIs

Der TSP-Solver verwendet verschiedene APIs von Google zum Geocodieren von Adressen, zum Anzeigen der Karte und für die Entfernungs- und Routenberechnung. Die Wahl fiel hier auf die APIs von Google, da sie in bestimmtem Umfang kostenlos zu verwenden sind und gute Bibliotheken für Java und JavaScript bereitstellen, durch die sehr einfach auf die APIs zugegriffen werden kann.

Die verwendeten APIs sind:

- **Google Maps JavaScript API**
  Die JavaScript API wird in der Weboberfläche verwendet. Sie stellt unter anderem die visuelle Karte und vereinfachten Zugang zu anderen Google APIs bereit.
  Die API erlaubt pro Tag und API-Key 25000 Map Renderings, was für die normale Verwendung dieses Programms mehr als ausreichend ist.

- **Google Maps Distance Matrix API**
  Durch diese API können Distanzmatrizen abgefragt werden, indem eine Liste von Startpunkten und Zielen angegeben wird. Das Ergebnis enthält dann die Entfernungen und Reisedauern von jedem Startpunkt zu jedem Zielpunkt. Im Falle dieses Programms sind beide Eingabelisten identisch, um von jedem eingegeben Standort eine Entfernung zu jedem anderen zu erhalten. Leider begrenzt Google die mögliche Anzahl an Ergebniselementen pro Anfrage auf 100, wodurch Anfragen mit 11x11 Standorten nicht möglich sind. Daher teilt der TSP-Solver jede Anfrage in zwei Anfragen mit je 5 und 6 Standorten auf. Hierdurch entsteht keine höhere Kontingentverwendung, da die Kontingente dieser API nicht mit der Anzahl der Anfragen, sondern mit der Anzahl der Ergebniselemente gemessen werden. Es ist zu beachten, dass die Distance Matrix API die Entfernung auf der von Google empfohlenen Route zurückgibt, die nicht in jedem Fall die kürzeste sein muss. Dies wird in Kauf genommen, da die einzelnen Anfragen über die Goolge Maps Directions API deutlich mehr Zeit in Anspruch nehmen würden, da diese nur 50 Anfragen pro Sekunde zulässt; hinzukämen deutlich mehr Netzwerklatenzen, da die kürzeste Route nur übertragen wird, wenn alle Routenalternativen angefragt werden, was zu einer höheren Antwortzeit des Servers führt.
  Die API stellt ein Kontingent von 2500 Ergebniselementen pro Tag und API-Key zur Verfügung, wodurch leider nur 20 Anfragen pro Tag und API-Key mit 11 Standorten durchgeführt werden können, da pro Berechnung 121 Ergebniselemente benötigt werden.

- **Google Maps Geocoding API**
  Die Google Geocoding API stellt einen Dienst für Geocodierungen von Adressen zu Koordinaten bereit. Diese wird benötigt, um in jedem Fall fehlerfrei alle anderen APIs mit korrekten Standortdaten zu bedienen.
  Die API stellt ein Kontingent von 2500 Anfragen pro Tag und API-Key bereit, was ausreichend ist.

- **Google Maps Directions API**
  Diese API stellt exakte Routenbeschreibungen zur Verfügung. Der TSP-Solver benötigt sie, um einen codierten Pfad zu ermitteln, der dann mit Hilfe der Google Maps JavaScript API auf einer Karte eingezeichnet wird. Die API unterstützt bis zu 23 Wegpunkte pro Anfrage, wodurch das gesamte Ergebnis der Rundreiseberechnung mit Hilfe einer einzigen Anfrage in einen Pfad umgewandelt werden kann.
  Die API stellt ein Kontingent von 2500 Anfragen pro Tag und API-Key bereit.

- **Google Maps Elevation API**
  Diese API stellt Höheninformationen über jede Art von Standorten zur Verfügung. Dies wird benötigt, um die Steigung eines Streckenabschnitts zu ermitteln.
  Die API stellt 2500 Anfragen pro Tag und API-Key bereit.

3.2 Berechnung des Energieverbrauchs

In dieser speziellen Version des TSP wird nicht mit reinen Entfernungswerten, sondern mit dem Energieverbrauch eines Elektro-LKW gearbeitet, der diese Strecken abfährt.
Da dieser bei Steigung mehr bzw. bei Gefälle weniger Strom verbrauchen kann, müssen diese Werte gesondert berechnet werden.
Vor jeder Berechnung wird daher mit den folgenden Formeln der Stromverbrauch für jeden einzelnen Streckenabschnitt ermittelt. Ein Streckenabschnitt bedeutet hier die Strecke von einen vom Benutzer spezifizierten Ort zu einem anderen.

Ohne Interpolation. Bei der Berücksichtigung von Steigungen und Gefälle ohne linearer Interpolation wird angenommen, dass ab einem bestimmten Steigungsgrenzwert ein bestimmter Anteil an Zusatzenergie benötigt wird (analog für die Einsparung). Dieser bleibt gleich, egal wie sehr die tatsächliche Steigung den Grenzwert überschreitet.
Der Verbrauch berechnet sich für einen Streckenabschnitt wie folgt:

$$W_A = s_A * P * \begin{cases} (1 + d^+) & \text{für} \quad g >= T^+ \\ (1 - d^-) & \text{für} \quad g <= T^- \\ 1 & \text{sonst} \end{cases} \tag{1}$$

mit

$$W_A = \text{Elektrische Arbeit für Streckenabschnitt A} \tag{2}$$
$$s_A = \text{Strecke des Streckenabschnitts A} \tag{3}$$
$$P = \text{Leistung des LKW (kWh / m)} \tag{4}$$
$$d^+ = \text{Mehrverbrauchsfaktor des LKW} \tag{5}$$
$$T^+ = \text{Mehrverbrauchsgrenzwert des LKW (Steigungsgrenzwert)} \tag{6}$$
$$d^- = \text{Einsparungsfaktor des LKW} \tag{7}$$
$$T^- = \text{Einsparungsgrenzwert des LKW (Steigungsgrenzwert)} \tag{8}$$
$$g_A = \text{Steigung des Streckenabschnitts A} \tag{9}$$

Mit Interpolation. Eine realistischere Annäherung wird mit linearer Interpolation erreicht - dies ist die Standardeinstellung.
Der Verbrauch berechnet sich für einen Streckenabschnitt dann wie folgt:

$$W_A = s_A * P * \begin{cases} (1 + \frac{g_A}{T^+} * d^+) & \text{für} \quad g_A > 0 \wedge T^+ > 0 \\ (1 - \frac{g_A}{T^-} * d^-) & \text{für} \quad g_A < 0 \wedge T^- < 0 \\ 1 & \text{sonst} \end{cases} \tag{10}$$

3.3 Algorithmus zur Rundreisenberechnung

In den nachfolgenden Abschnitten wird vereinfachend häufig vom *kürzesten Weg* oder der *kürzesten Strecke* gesprochen. In diesem Fall ist damit jedoch immer der Weg mit dem geringsten Stromverbrauch gemeint. Dies gilt analog natürlich auch für ähnliche Formulierungen, etwa für den *längsten Pfad*.

3.3.1 Grundfunktionalität

Zur Lösung des TSP wurde zunächst aufgrund der geringen Anzahl an möglichen Wegpunkten ein trivialer Algorithmus implementiert, der in seiner Grundform alle $(n-1)!$ möglichen Rundreisen berechnet und dann eine kürzeste Rundreise auswählt. Dies lässt sich für 11 Punkte in wenigen Sekunden bewerkstelligen, da es lediglich $(11-1)! = 10! = 3628000$ mögliche Rundreisen gibt. Zudem wurde in der Angabe eine exakte Lösung gefordert ("[...]vom Startpunkt aus die kürzeste Route finden, [...]"), was durch Heuristiken nicht erreicht wird. Es handelt sich im Übrigen um ein asymmetrisches TSP, da die Entfernung und auch die Reisedauer zwischen zwei Standorten im realen Straßennetz je nach Reiserichtung variieren kann, z.B. durch Einbahnstraßen. Daher gibt es $(n-1)!$ und nicht $(n-1)!/2$ mögliche Rundreisen.

Es wurde eine simple Rekursion implementiert, die in jedem Aufruf den bisherigen Pfad mit dessen Strecke übergibt, den Pfad dann jeweils einmal für jeden noch nicht abgefahrenen Punkt mit ebendiesem erweitert und sich selbst mit dem neuen Pfad mit aktualisierter Pfadstrecke aufruft. Gibt es keinen solchen Punkt, so wird der Pfad mit dem Startpunkt

erweitert und die Strecke des Pfades entsprechend aktualisiert. Anschließend werden Pfad und Länge zurückgegeben - dies ist die endgültige Strecke des Pfades.

In jedem Rekursionsaufruf wählt die Methode dann unter den von den weiteren rekursiven Aufrufen zurückgelieferten Pfaden einen von der Strecke her kürzesten und gibt diesen zurück. Dies setzt sich im Rekursionsbaum nach oben fort, bis eine kürzeste Rundreise gefunden wurde.

3.3.2 Optimierungen

Früherkennung langer Pfade (Branch-and-Bound-Prinzip) Da dieser o.g. Ablauf für 11 Standorte allerdings ein paar Sekunden benötigt ($8 < s < 12$, je nach Wegpunktanordnung), wurde der Algorithmus optimiert, indem eine Variable für einen globalen kürzesten Pfad angelegt wurde, dessen Strecke zu Beginn der Berechnung maximal lang ist (`Long.MAX_VALUE`). Wird ein kürzerer Pfad gefunden, so wird die Variable mit diesem belegt. Nun wird in jedem Rekursionsaufruf die Strecke des aktuellen (meist noch nicht vollständigen) Pfades mit der Strecke dieses globalen kürzesten Pfades verglichen. Die weitere Berechnung des Pfades und der weiteren Branches, die aus diesem hervorgehen würden, wird abgebrochen, wenn gilt:

$$l_{cur} > l_s$$

mit

$$l_{cur} = \text{Strecke des aktuellen Pfades} \tag{11}$$

$$l_s = \text{Strecke des global kürzesten Pfades} \tag{12}$$

Dadurch können sehr viele Berechnungen, die ohnehin zu einem sehr langen Pfad führen würden, frühzeitig erkannt und unterbrochen werden, was die Berechnungszeit drastisch minimiert und den restlichen Lösungsraum verkleinert. Für den Anwender hat dies den Effekt, dass die Berechnung meist ab 30% deutlich schneller durchgeführt wird, da es wahrscheinlich ist, dass bereits früh ein Pfad gefunden wird, der nah am Optimum liegt. Im schlechtesten Fall müsste der Algorithmus allerdings auch mit dieser Optimierung alle mögliche Rundreisen berechnen. Die Rundreise, die am Ende berechnet wird, ist auch mit dieser Optimierung immer eine kürzeste Rundreise.

Sortierung der Standorte für Nearest-Neighbor-Heuristik Eine weitere Optimierung besteht im zeilenweisen Sortieren der Distanzmatrix. Der Grundalgorithmus verwendet in jedem Rekursionsaufruf eine einfache `for`-Schleife, um jeden noch nicht angefahrenen Standort einmal zu verwenden. Sortiert man diese Standorte nun nach der Entfernung vom aktuellen Punkt, so erhöht sich die Wahrscheinlichkeit in vielen Fällen, dass früher eine Route gefunden wird, die nah am Optimum ist, sodass die zuvor beschriebe Optimierung deutlich mehr Erfolg hat. Hierzu sortiert das Programm allerdings nicht die Matrix selbst um, da die Reihenfolge der Spalten der Matrix nicht verändert werden darf, da sonst das Ergebnis (der Pfad, also eine Liste aus Indices) keinen Sinn mehr ergeben würde. Daher wird eine zweite Matrix geführt, die für jedes Element die Indexreihenfolge der anderen Elemente angibt, die ihm an nächsten sind. Das nachfolgende Beispiel soll

dies veranschaulichen: In der linken Matrix sind die Entfernungen zwischen den Punken A, B, C angegeben, die rechte Matrix zeigt die zweite Matrix, die die Sortierreihenfolge zeigt.

	A	B	C
A	0	30	20
B	31	0	50
C	20	49	0

	0	1	2
A	0	2	1
B	1	0	2
C	2	0	1

Beim Durchlauf der `for`-Schleife wird nun z.B. nicht der Punkt B als erstes verwendet, wenn man sich an Punkt A befindet, sondern zuerst Punkt C, da sich dieser näher an A befindet. Dies sieht man in der Matrix daran, dass der Schleifendurchlauf mit $i = 1$ das Element mit Index 2 aufruft, welches C ist.

Es handelt sich um eine Anwendung der Nearest-Neighbor-Heuristik, da in der allerersten Pfadberechnung, die immer vollständig ist, immer genau ein Standort gewählt wird, der dem aktuellen Standort am nächsten ist. Tatsächlich lassen sich für diese Heuristik Beispiele konstruieren, bei denen das Ergebnis beliebig schlecht ausfällt - allerdings ist die Wahrscheinlichkeit höher, dass die vom Nutzer eingegebene Reihenfolge deutlich schlechtere Ergebnisse liefert. In der Praxis und im realen Anwendungsfall wird die Nearest-Neighbor-Heuristik aber häufig die ersten Schritte der optimalen Rundreise korrekt ermitteln.

Ein Beispiel hierfür ist die Ermittlung einer Rundreise durch folgende Städte, eingegeben in dieser Reihenfolge:

1. München
2. Hamburg
3. Frankfurt
4. Wroclaw
5. Leipzig
6. Kassel
7. Nürnberg
8. Stuttgart
9. Magdeburg
10. Meiningen
11. Hannover

Ohne Nearest-Neighbor-Heuristik dauerte die Ermittlung der idealen Rundreise auf dem Testsystem (Intel i7-5600U, 12GB RAM) im Durchschnitt etwa 3,83 Sekunden und benötigte 176054 vollständige Pfadberechnungen, während die Ermittlung mit der Heuristik im

Durchschnitt etwa 3,1 Sekunden dauerte und nur 98128 vollständige Pfadberechnungen benötigte. Die Optimierung durch Früherkennung langer Pfade war hierbei aktiv, da sonst immer alle Pfade berechnet worden wären und ein Vergleich somit nicht möglich wäre. Multithreading war zum Vergleich nicht aktiv (siehe nächster Abschnitt).

Multithreading Eine weitere sehr effektive Optimierung ist das Multithreading. Da sehr viele Vergleiche durchgeführt werden müssen, führt das Aufteilen der Rechenlast auf alle Kerne bei Mehrkernprozessoren zu einer enormen Leistungssteigerung.

Hierzu teilt das Programm den ersten Rekursionsdurchlauf, also alle Pfaderweiterungen, die vom Startpunkt aus erfolgen, gleichmäßig auf einer der vierfachen Anzahl der verfügbaren Prozessorkerne entsprechenden Threadmenge auf. Dies führt dazu, dass die Anzahl der nötigen vollständigen Pfadberechnungen für die gleiche Eingabe nicht in jedem Durchlauf gleich ist, da die Threads nicht in jedem Durchlauf immer mit gleicher Geschwindigkeit arbeiten, sondern diese Scheduler-bedingt variieren kann. Allerdings führt dieser kleine Nachteil keinesfalls zu einer relevanten Verschlechterung der Rechenzeit. Tatsächlich lässt sich die Rechenzeit auf dem Testsystem bei einer Eingabe von 11 Standorten von den im obigen Versuch angegebenen 3.1s auf 0.30s reduzieren - das sind gerade einmal etwa 10% der vorherigen Zeit. Die Optimierung ist also so effektiv, dass entsprechende Synchronisationen auf die globalen Variablen und die Threadverwaltung nur zu vernachlässigende Zeit in Anspruch nehmen.
Insgesamt wurde durch verschiedene Optimierungen die Rechenzeit für eine Eingabe mit 11 Standorten von 8 bis 12 Sekunden im ursprünglichen Grundalgorithmus auf 0.30 Sekunden reduziert, wodurch das Programm keine für den Nutzer bemerkbaren Wartezeiten verursacht, die nicht auf Latenzen bei der Kommunikation mit Google zurückzuführen sind. Die folgende Tabelle zeigt die Unterschiede der Optimierung zusammengefasst auf.

Ohne Optimierung	mit Erkennung langer Pfade	mit NN-Heuristik	mit Multithreading
8-12s	3.83s	3.1s	0.3s
100%	47.8% - 31.9%	38.8% - 25.8%	3.8% - 2.5%