Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

1 Vorwort

Die nachfolgende Arbeit befasst sich mit der Thematik des 'Traveling Salesman Problem' (Abk. 'TSP', zu Deutsch: 'Das Problem des Handlungsreisenden').

Da es sich bei dem Begriff 'Traveling Salesman Problem' um eine, auch zu Deutsch, etablierte Fachterminologie handelt, wird in der Arbeit ausschließlich die englische Bezeichnung und deren Abkürzung verwendet.

Inhaltlich beschäftigt sich die Arbeit sowohl mit den theoretischen als auch mit den praktischen Aspekten des Traveling Salesman Problem.

Der theoretische Teil behandelt dabei Punkte wie: Definiton, Historie und zugrundeliegende Mathematik des TSP. Die Theorie vermittelt ein wichtiges Grundverständnis, auf welchem die praktische Umsetzung aufbaut.

Durch die Theorie gestützt behandelt der praktische Teil der Ausarbeitung eine vielzahl an Lösungsansätzen, welche sich über die Jahrzehnte etabliert haben. Verschiedene Algorithmen und Heuristiken sind in Form von Programmcode bereitgestellt und werden anhand von Beispielen vermittelt.

Der Programmcode kann in öffentlichen Github Repositories eingesehen werden. Verlinkungen befinden sich im Anhang, Kapitel 'Programmcode'.

2 Das Traveling Salesman Problem – Einführung

„Business leads the traveling salesman here and there, and there is not a good tour for all occurring cases; but through an expedient choice and division of the tour so much time can be won that we feel compelled to give guidelines about this. Everyone should use as much of the advice as he thinks useful for his application. We believe we can ensure as much that it will not be possible to plan the tours through Germany in consideration of the distances and the traveling back and fourth, which deserves the traveler's special attention, with more economy. The main thing to remember is always to visit as many localities as possible without having to touch them twice.“ (Der Handlungsreisende – [...] – von einem alten Commis Voyageur, 1832. Aus dem Deutschen Original, Übersetzt ins Englische von Linda Cook) (David L. Applegate, 2006; Heribert Rau, 2010)

2.1 Problemstellung

Wie der Auszug aus dem Text 'Der Handlungsreisende [...]' bereits beschreibt, wenn auch ein wenig antiquiert formuliert, handelt es sich bei dem TSP um ein Optimierungsproblem. Es zielt darauf ab, die kürzeste mögliche Route zu finden, die es einem Handlungsreisenden erlaubt, eine gegebene Anzahl von Städten genau einmal zu besuchen und zum Ausgangspunkt zurückzukehren.

Die Formulierung lässt Raum für eine differenziertere Beschreibung. Eine mögliche Formulierung, welche das Traveling Salesman Problem aus einer wissenschaftlichen Perspektive beleuchtet, wird im nachfolgenden Kapitel erarbeitet.

2.2 Definition und Formulierung durch die Historie

Die Problematik des Traveling Salesman in der Moderne (Kapitel 2.2.7) zu definieren und auch zu formulieren, ist denkbar einfach. Der Grund für die Einfachheit des TSP sind seine klar definierten und unkomplizierten Grundprinzipien. Um diese Grundprinzipien nachzuvollziehen, werden im Folgenden zunächst die Formulierungen und Definitionen dargelegt, welche das TSP über die Jahre zu dem Optimierungproblem geformt haben, das wir heute kennen.

2.2.1 1832

Eine mögliche Formulierung, wenn auch nicht direkt für das TSP, wurde bereits um 1832 niedergeschrieben. Der Auszug aus dem Text, 'Der Handlungsreisende [...]' (Kapitel 2) hat einen offensichtlich praktischen Charakter, galt er als direkter Ratgeber für Handlungsreisende seiner Zeit.

Ob die Handlungsreisenden des 19ten Jahrhunderts tatsächlich mit der gleichen Problemstellung, zu dem uns bekanntem TSP, konfrontiert waren lässt sich hier jedoch in Frage stellen. Dennoch ist eine offensichtliche Analogy zur moderneren Formulierung des TSP (Kapitel 2.2.7) nicht zu übersehen. (Wikipedia, 2024m; Wikipedia, 2024n)

2.2.2 1857

Neben den praktischen Ratgebern für Handlungsreisende um 1830 gibt es eine weitere Formulierung, die von vielen Quellen als potenzieller Vorläufer des TSP angesehen wird.

Das 'Icosian Game' von William Rowan Hamilton aus dem 19. Jahrhundert ist ein mathematisches Brettspiel. Das Ziel des Spiels besteht darin, einen Hamiltonkreis entlang der Kanten eines Dodekaeders zu finden. Dabei muss jeder Knoten genau einmal besucht werden, und der Endpunkt des Pfads muss identisch mit dem Startpunkt sein.

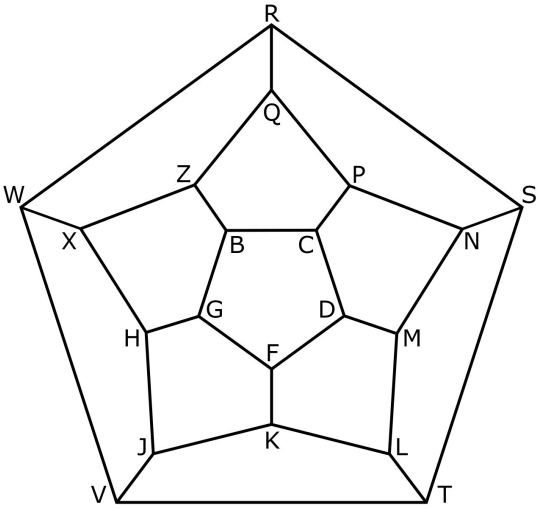


Abbildung 1: Icosian Game – Zeichnung – (Wikimedia, 2022)

- Ein **Hamiltonkreis** ist ein geschlossener Pfad in einem Graphen, der jeden Knoten genau einmal enthält. Der Endpunkt des Pfads muss identisch mit dem Startpunkt sein.
- Ein **Dodekaeder** ist ein geometrischer Körper mit zwölf flachen Seiten, die alle die Form von regelmäßigen Fünfecken haben.

Hamiltons Brettspiel stellt in vielerlei Hinsicht einen wichtigen Schritt zur heutigen Formulierung (Kapitel 2.2.7) des Traveling Salesman Problem dar. Auch wenn das Icosian Game nicht direkt ein Optimierungsproblem definiert, so handelt es sich doch um eine klar formulierte mathematische Problemstellung. Insbesondere der Hamiltonkreis ist ein wichtiges mathematisches und graphtheoretisches Konzept welches für das moderne TSP, aber auch für die Graphentheorie als ganzes von Bedeutung ist. (Wikipedia, 2023a; Wikipedia, 2024d; Wikipedia, 2024e; Wikipedia, 2024g; Wikipedia, 2024m; Wikipedia, 2024n; David L. Applegate, 2006)

2.2.3 1930

Wann das TSP erstmals als mathematisches Optimierungsproblem definiert wurde ist unklar. Eine der ersten, möglicherweise die erste Erwähnung des TSP als mathematisches Optimierungsproblem lässt sich auf den Mathematiker Karl Menger zurückführen. (Gregory Gutin; Abraham P. Punnen, 2007; David L. Applegate, 2006)

Dieser formulierte im Jahr 1930, während eines mathematischen Kolloquiums in Wien folgenden Satz:

> *„Wir bezeichnen als Botenproblem [...] die Aufgabe, für endlich viele Punkte, deren paarweise Abstände bekannt sind, den kürzesten die Punkte verbindenden Weg zu finden.“* (Wikipedia, 2024m)

2.2.4 1930 bis 1954

„Bald [...] wurde die heute übliche Bezeichnung ‘Traveling Salesman Problem’ durch Hassler Whitney von der Princeton University eingeführt.“ (Wikipedia, 2024m)

„[...] Perhaps the first report using this name was published in 1949. However [...] a systematic study of the TSP as a combinatorial optimization problem began with the work of Dantzig [...].“ (Gregory Gutin; Abraham P. Punnen, 2007)

2.2.5 1954

„Seit den 1950er Jahren gewann das Traveling Salesman Problem sowohl in Europa als auch in den USA an Popularität. Herausragende Beiträge leisteten George Dantzig, Delbert Ray Fulkerson und Selmer M. Johnson. [Am Institut der RAND Corporation in Santa Monica, 1954, formulierten sie als erste das Problem als ganzzahliges lineares Programm und entwickelten ein Schnittebenenverfahren zu dessen Lösung.] Sie berechneten eine Tour für ein konkretes Rundreiseproblem (eine sogenannte Probleminstanz) mit 49 Städten und bewiesen, dass es keine kürzere Tour gibt.“ (Wikipedia, 2024m; Donald Davendra, 2010; Gregory Gutin; Abraham P. Punnen, 2007; David L. Applegate, 2006)

2.2.6 1972

„Richard M. Karp bewies [...] 1972 die NP-Vollständigkeit [(zu Englisch: NP-Complete)] des Hamiltonkreisproblems, aus der sich leicht die NP-Äquivalenz des TSP ableiten lässt. Damit lieferte er eine theoretische Begründung für die schwere Lösbarkeit dieses Problems in der Praxis.“ (Wikipedia, 2024m)

- **NP-Equivalent:**

  „Formal ist ein Suchproblem NP-Äquivalent, wenn es [NP] und [NP-Hard] ist. Dies ist genau dann der Fall, wenn das zugehörige Entscheidungsproblem NP-Vollständig ist.“

  (Wikipedia, 2023b; Wikipedia, 2023c)

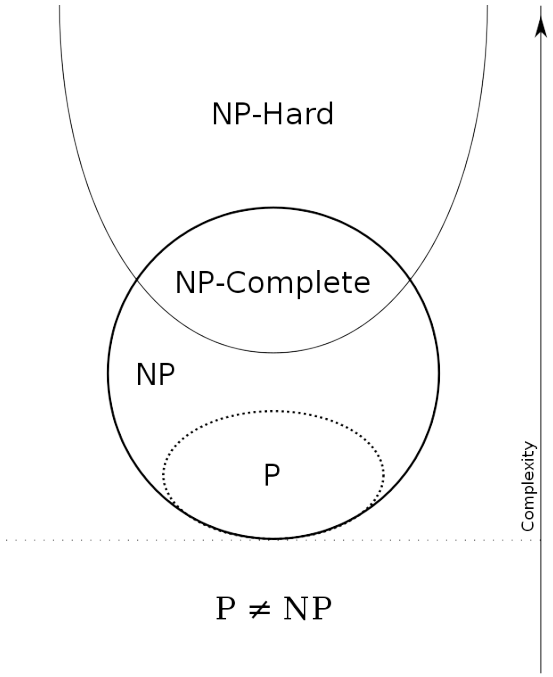


Abbildung 2: P; NP; NP-Hard; NP-Complete – (Wikipedia, 2023e)

- **P:**

  In der Komplexitätstheorie ist P die Komplexitätsklasse, die alle Entscheidungsprobleme enthält, die in Polynomialzeit für deterministische Algorithmen (Turingmaschinen) lösbar sind. Diese Problemklasse wird allgemein als Klasse der praktisch lösbaren Probleme betrachtet.

  (Ingrid Gerdes, 2004; Volker Turau, 1996; Wikipedia, 2022; Wikipedia, 2024l)

- **NP:**

  In der Komplexitätstheorie ist NP die Komplexitätsklasse, die alle Entscheidungsprobleme enthält, die in Polynomialzeit für nicht-deterministische Algorithmen lösbar sind.

  (Ingrid Gerdes, 2004; Volker Turau, 1996; Wikipedia, 2024i)

- **NP-Hard:**

  In der Komplexitätstheorie ist NP-Hard die Komplexitätsklasse, die alle Entscheidungsprobleme enthält, die mindestens in Polynomialzeit für nicht-deterministische Algorithmen lösbar sind, jedoch auch deutlich komplexer sein können.

  (Ingrid Gerdes, 2004; Volker Turau, 1996; Wikipedia, 2023d; Wikipedia, 2024k)

- **NP-Complete:**

  In der Komplexitätstheorie ist NP-Complete die Komplexitätsklasse, die alle Entscheidungsprobleme enthält, die NP-Hard sind aber noch in NP liegen. Dies bedeutet umgangssprachlich, dass es sich vermutlich nicht effizient lösen lässt da es zu den schwierigsten Problemen der Klasse NP gehört.

  (Ingrid Gerdes, 2004; Volker Turau, 1996; Wikipedia, 2023e; Wikipedia, 2024j)

2.2.7 Moderne Formulierung

Das Traveling Salesman Problem (Zu Deutsch: 'Das Problem des Handlungsreisenden') ist ein graphtheoretisches, kombinatorisches, Optimierungsproblem der Komplexitätsklasse NP-Complete.

Gegeben ist ein kantenbewerteter, zusammenhängender Graph mit n Knoten. Gesucht wird ein Hamiltonkreis, also eine Rundreise (auch als Tour bezeichnet), die von einem Startknoten aus jeden Knoten genau einmal besucht. Der resultierende Zyklus ist geschlossen, da der Endknoten mit dem Startknoten identisch ist.

Unter allen geschlossenen Rundreisen, auf denen alle Knoten des Graphen liegen, wird ein solcher mit minimaler Länge gesucht.
(Prof. Dr. Guido Walz, 2017; Berthold Vöcking, 2008; Volker Turau, 1996)

Die wichtigsten Aspekte, die die **Grundstruktur** der Problematik definieren, können wie folgt zusammengefasst werden:

1. Es existiert genau ein Reisender.
2. Die Tour kann als Hamilton-Kreis dargestellt werden.
3. Die Tour ist von minimaler Länge.

Auf der folgenden Seite befindet sich nun eine ungelöste TSP-Instanz sowie die Optimallösung der besagten Instanz. Das gesammelte Hintergrundwissen über Formulierungen und Definitionen kann bewusst auf die Graphen angewendet werden, um sowohl die Struktur als auch die Aspekte hinter der Optimallösung nachzuvollziehen.

Aspekte wie beispielsweise der Hamilton-Kreis lassen sich nun direkt erkennen.

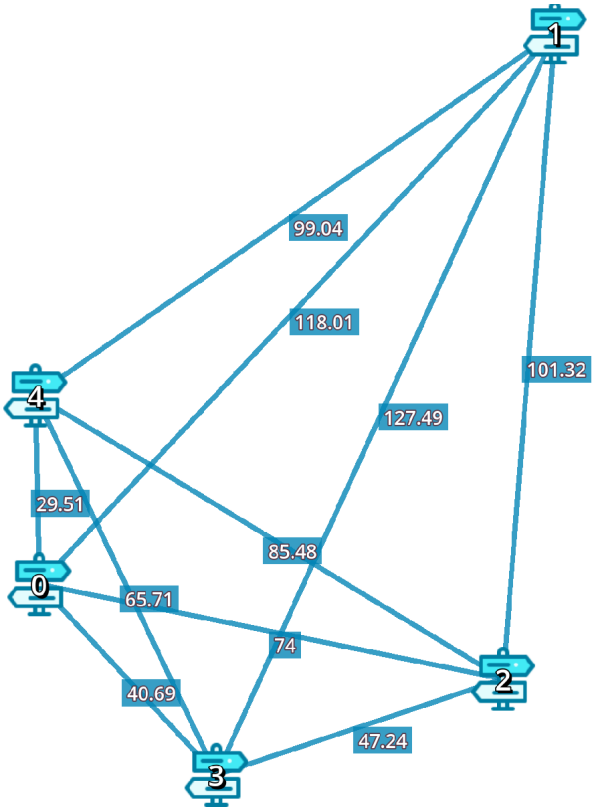


Abbildung 3: TSP Beispiel Ungelöst – (Selbsterstellt)

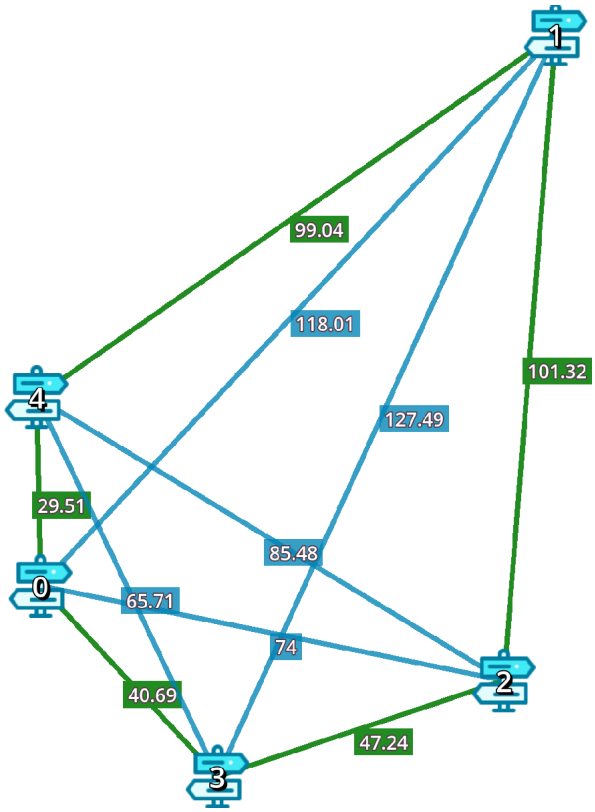


Abbildung 4: TSP Beispiel Optimale Rundreise – (Selbsterstellt)

2.3 Variationen des TSP

Zusätzlich zu der modernen Formulierung können weitere Variationen und Abstraktionen auf Basis des TSP abgeleitet werden. Das TSP selbst lässt sich also durch bestimmte Annahmen weiter definieren und kategoresieren. Dabei kann zwischen zwei Arten von Variationen unterschieden werden.

Zunächst werden einige Variationen (Aspekte) aufgeführt, welche das TSP weiter definieren ohne die zuvor genannte Grundstruktur (Kapitel 2.2.7) zu verändern:

2.3.1 Aspekt der Vollständigkeit

(Wikipedia, 2024m; Wikipedia, 2021)

- Complete TSP: Ein Vollständiges TSP verbindet jeden Knoten v_i mit allen Knoten der Menge $|V|$ mit Ausnahme von v_i selbst. Jeder Knoten v_i kann also von jedem anderem Knoten der Menge erreicht werden.
- Incomplete TSP: Ein Unvollständiges TSP verbindet *nicht* jeden Knoten v_i mit allen anderen Knoten der Menge $|V|$. Jeder Knoten v_i kann von mindestens einem Knoten v_j erreicht werden, jedoch nicht zwangsweise von jedem Knoten der Menge.

2.3.2 Aspekt der Symmetrie

(Gregory Gutin; Abraham P. Punnen, 2007)

- Symmetric TSP (sTSP): Ein symmetrisches TSP differenziert *nicht* zwischen hin- und rückweg, daher gilt folgende Katenbewertung: $e_{ij} = e_{ji}$
- Asymmetric TSP (aTSP): Ein asymmetrisches TSP differenziert zwischen hin- und rückweg, daher gilt folgende Katenbewertung: $e_{ij} \neq e_{ji}$. Umgangssprachlich formuliert, der Weg von Knoten A zu Knoten B ist länger (oder kürzer) als der Weg von Knoten B zu Knoten A.

2.3.3 Aspekt der Metrik

(Wikipedia, 2024m)

- Metric TSP: Ein TSP wird als metrisch bezeichnet wenn es Symmetrtisch ist und zudem die Dreiecksgleichung erfüllt. Die Dreiecksgleichung besagt das die direkte Verbindung e von v_i nach v_j nie länger ist als die Verbindung von v_i nach v_j über einen dritten Knoten v_k. Das bedeutet: $e_{ij} \leq e_{ik} + e_{kj}$
- None-Metric TSP: Ein Nicht-Metrisches TSP erfüllt die Dreiecksgleichung nicht.

Neben den getroffenen Annahmen gibt es auch Variationen, welche die Grundstruktur (Kapitel 2.1.7) des TSP verändern. Diese "speziellen" Formen des TSP finden sich oft in Problemstellungen wieder wo das eigentliche TSP lediglich als Teilproblem auftritt.

2.3.4 m-Salesman(s)-TSP

- Auch genannt: mTSP / TSP With multiple Travelers.

  Bei einem TSP mit m Reisenden starten die jeweiligen Touren von einem oder verschiedenen Startknoten aus. Die Schwierigkeit besteht nun darin sowohl alle individuellen Touren zu minimieren als auch die länge aller Touren (als Gesamt-Tour) minimmal zu halten. Dabei sollen Kollisionen unter den individuellen Touren vermieden werden. In der Praxis kommen zusätzliche Einschränkungen in Form von Anforderungen der Reisenden hinzu. In einem Logistik Unternehmen beispielsweise wollen alle Reisenden circa die selbe (Arbeits-) Zeit unterwegs sein.

  Diese Variation ist ein Teilproblem des 'Vehicle Routing Problem'.

  (Donald Davendra, 2010; Gregory Gutin; Abraham P. Punnen, 2007)

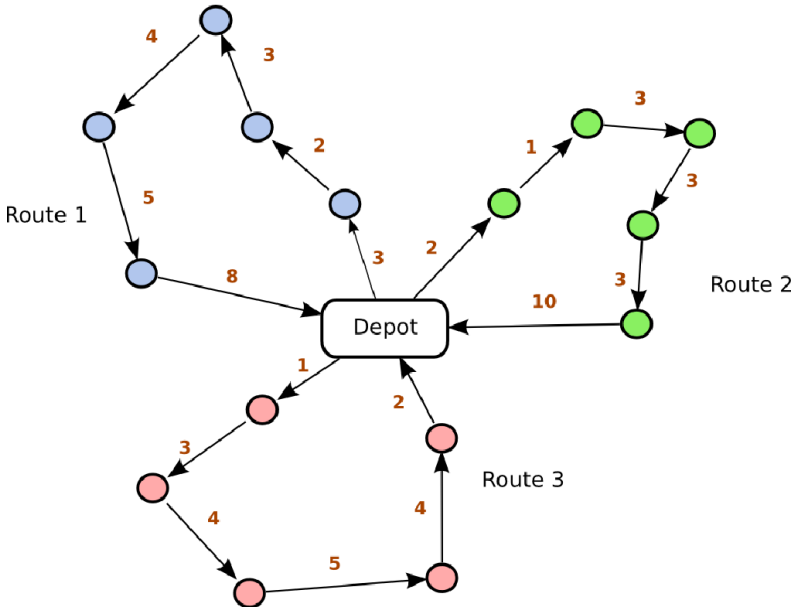


Abbildung 5: Visuelles, Optimiertes Beispiel – mTSP – (Chieng, 2016)

2.3.5 TSPTW

- Auch genannt: TSP With Time Windows.

  Ein TSP mit Zeitfenstern erweitert das TSP um eine zeitliche Komponente. Jeder Kunde hat neben der Entfernung zu anderen Städten auch ein Zeitfenster, das angibt innerhalb welcher Zeit der Besuch stattfinden muss. Diese Zeitfenster können beispielsweise die Geschäftszeiten eines Ladens oder die Verfügbarkeitszeiten eines Kunden darstellen. Das Ziel besteht nun darin, eine Route zu finden, die nicht nur die kürzeste Gesamtdistanz aufweist, sondern auch alle Zeitfenster der Kunden berücksichtigt und keine Zeitfenster verletzt.

  Diese Variation lässt sich wohl am ehesten auf die Problemstellung der Handlungsreisenden um 1830 (Kapitel 2.1.1) anwenden.

  (Gregory Gutin; Abraham P. Punnen, 2007; Yvan Dumas; Jacques Desrosiers, 1995)

2.3.6 MAX-TSP

- Auch genannt: TSP With maximum Tour.

  Das MAX-TSP ist fast komplett identisch dem klassischen TSP gegenüber. Jedoch wird in dieser Variation eine maximale Tour anstatt einer minimalen gesucht.

  (Gregory Gutin; Abraham P. Punnen, 2007)

2.3.7 TSPM

- Auch genannt: TSP With multiple visits.

  Wie der Name bereits beschreibt erlaubt das TSPM dem Reisenden einen Knoten mehrmals zu durchlaufen.

  (Gregory Gutin; Abraham P. Punnen, 2007)

Abschliessend sei zu der Thematik 'Variationen des TSP' noch folgendes gesagt:
Ein TSP kann durch genau eine oder beliebig viele Variationen geprägt sein und kann somit sehr flexibel auf verschiede Problemstellungen angepasst werden.

2.4 Anwendungsfälle des TSP

Neben dem klassischen Anwendungsfall der minimalen Rundreise eines Handlungsreisenden ...

- **Vertriebsmitarbeiter:** Auch heute gibt es noch Handelsvertreter, welche direkt zum Kunden fahren, um Produkte wie Lebensmittel oder Haushaltsgeräte (beispielsweise Staubsauger) zu verkaufen. Dieses Modell wird oft von Unternehmen genutzt, die auf den persönlichen Kontakt mit ihren Kunden setzen, sei es aufgrund der Art der Produkte, regionalen Gegebenheiten oder aus strategischen Gründen. Obwohl der traditionelle 'Tür-zu-Tür' Vertrieb weniger verbreitet ist als früher, gibt es dennoch Branchen die weiterhin auf persönliche Verkaufsgespräche setzen. (Optessa, [o. D.]; Matai u. a., 2010)

... findet sich die Struktur des TSP auch in anderen Optimierungsproblemen wieder.
Die folgenden Anwendungsfälle sind nur einige der vielen Bereiche in denen das TSP, eine Variation des TSP oder das TSP als Teilproblem vorkommt. Das Traveling Salesman Problem ist also auch in unserer Zeit noch immer weit verbreitet und von Bedeutung für verschiedene Anwendungsbereiche. Von der Informatik, den Wirtschaftswissenschaften über die Chemie und der Biologie findet sich das TSP fast überall wieder.
(Wikipedia, 2024m; Optessa, [o. D.])

2.4.1 Tourismusplanung

Ähnlich der klassichen Rundreise kann das TSP im Tourismussektor verwendet werden, um effiziente Reiserouten für Touristen zu planen. Dabei kann die Tour durch mehrere Städte oder Sehenwürdigkeiten einer bestimmten Region laufen. Eine effiziente Tour trägt dazu bei die Reisekosten und den Zeitaufwand zu minimieren. (Matai u. a., 2010)

2.4.2 Fertigung von Platinen

Das TSP kann auch in der Platinenfertigung, insbesondere bei der Bohrung der Platine, angewendet werden, um so die effizienteste Route für den Bohrarm zu bestimmen. (Matai u. a., 2010; David L. Applegate, 2006)

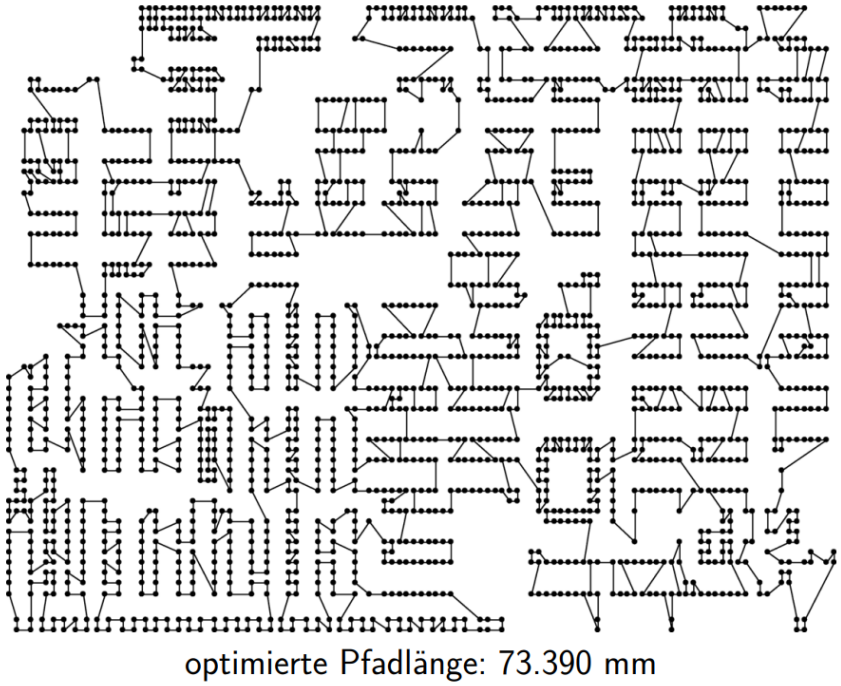


Abbildung 6: Optimale Pfadlänge beim Bohren einer Platine – (Skript zum WPM Praktische Optimierung an der WHS Bocholt, SoSe 2024)

2.4.3 Logistik und Routenplanung

In der Logistik wird das TSP verwendet, um effiziente Routen für Lieferfahrzeuge zu planen. Dabei wird oft das m-Traveler(s)-TSP angewendet. Ziel ist es, die Transportkosten zu minimieren und die Lieferzeiten für m Lieferfahrzeuge zu optimieren.
(David L. Applegate, 2006; Matai u. a., 2010; Optessa, [o. D.])

2.4.4 Optimierung von Produktionsprozessen

In der Fertigungsindustrie wird das TSP verwendet, um die Reihenfolge der Bearbeitung von Teilen oder Produkten zu optimieren. So wird die Durchlaufzeit minimiert und die Produktionskosten werden gesenkt. (Optessa, [o. D.]; Matai u. a., 2010)

2.4.5 Telekommunikationsnetzwerke

Beim Aufbau von Telekommunikationsnetzwerken müssen Techniker verschiedene Standorte besuchen, um Ausrüstung zu installieren oder zu warten. Das TSP wird verwendet,

um die effizienteste Route für diese Besuche zu planen, um die Zeit und die Kosten zu minimieren. (Matai u. a., 2010; Optessa, [o. D.])

2.4.6 Routing von Drohnen

Bei der Auslieferung von Paketen durch Drohnen wird das TSP verwendet, um die optimale Route für die Drohnen zu bestimmen, um die Lieferzeit zu minimieren und die Effizienz der Lieferungen zu maximieren. (Optessa, [o. D.])

2.4.7 Stromnetzoptimierung

Bei der Installation neuer Stromnetze müssen Leitungen zwischen den verschiedenen Knotenpunkten verlegt werden. Hier kann das TSPM helfen Stromnetze zwischen Knoten effektiv zu verlegen, wobei eine Leitung denselben Knoten mehrmals durchlaufen kann. Dies führt dazu das Backup-Leitungen existieren, sollte eine der Leitungen defekt sein. (Matai u. a., 2010; Optessa, [o. D.])

3 Das Traveling Salesman Problem – Lösungsansätze

In den vorherigen Kapiteln wurde das TSP bereits deutlich formuliert, in seinen verschiedenen Formen betrachtet und auch in konkreten Anwendungsfällen aufgeführt.

Basierend auf den angesprochenen Variation definieren wir einige Annahmen welche für folgende Lösungsansätze gelten sollen. Der Graph, welcher das TSP beschreibt ist ...

- Vollständig.
- Symmetrisch.
- Metrisch.

Was die Annahmen bedeuten wurde bereits in 'Variation des TSP' (Kapitel 2.3) erläutert. Des weiteren gelten selbstverständlich auch alle Annahmen und Definition, welche vom klassischen TSP, so wie wir es formuliert haben, aufgestellt werden.

In Ausarbeitungen zum TSP wird immer wieder auf die TSPLIB verwiesen.

„[Die TSPLIB ist] eine Sammlung verschieden schwerer standardisierter Testinstanzen, womit viele Forschergruppen ihre Resultate vergleichen konnten.“ „Gerhard Reinelt stellte 1991 die TSPLIB bereit [...] “ (David L. Applegate, 2006)

Während die TSPLIB eine großartige Ressource für forgeschritte wissenschaftliche Ausarbeitungen ist, ist sie für den Umfang dieser Ausarbeitung schlicht zu umfangreich. Stattdessen werden sich selbst erstellete Probleminstanzen angeschaut, welche in Komplexität und Schwierigkeit einfacher zu steuern sind.

3.1 Definition des Suchraumes

n	**10**	**100**	**1000**	**10000**
n^2	10^2	10^4	10^6	10^8
n^3	10^3	10^6	10^9	10^{12}
2^n	1024	10^{30}	10^{301}	10^{3010}
$n!$	10^6	10^{157}	10^{2567}	10^{35659}
n^n	10^{10}	10^{200}	10^{3000}	10^{40000}

Tabelle 1: Wachstum von Suchräumen – (Selbsterstellt)

Zum Vergleich (*Beide Werte wurden lediglich aus einer einfachen Suchanfrage gezogen*):

- Auf der Erde gibt es 10^{21} oder 1,4 Trilliarden Liter Wasser.
- Im Universum gibt es 10^{84} Atome.

Mit einem Blick auf die Tabelle zum Wachstum von Suchräumen wird zunächst der Suchraum des TSP bestimmt. Bekannt ist die Komplexitätsklasse des TSP, NP-Complete. Da es für Entscheidungsprobleme der Klass NP-Complete keine bekannten Methoden gibt mit welchen eine Lösung in polynomieller Zeit gefunden werden kann, können n^2 und n^3 *nicht* den Suchraum des TSP definieren.

Aus den exponentiellen Suchräumen 2^n, $n!$ und n^n definiert $n!$ den Suchraum des TSP. Dies erschließt sich, da das Traveling Salesman Problem auch als Permutationsproblem definiert wird. Ein Permutationsproblem ist eine Art von mathematischem Problem, bei dem es darum geht, die verschiedenen Möglichkeiten zu ermitteln, wie eine Reihe von Objekten (beim TSP eine Reihe Knoten) angeordnet werden kann. (Ingrid Gerdes, 2004)

Damit lässt sich das TSP grundsätzlich dem Suchraum von $n!$ zuordnen. Zu sagen der Suchraum des TSP wäre $n!$ ist jedoch nicht ganz korrekt. Im Kontext des Traveling Salesman Problem bezieht sich $n!$ auf die Anzahl der möglichen Routen, wenn jeder Knoten als möglicher Startpunkt betrachtet wird. Das heißt es werden alle Permutationen der Knoten berücksichtigt. Tatsächlich spielt der Startpunkt bei einem Rundreiseproblem aber keine Rolle, da man am Ende zum Ausgangspunkt zurückkehrt. Daher kann man einen beliebigen Knoten als Startpunkt festlegen und nur die Permutationen der verbleibenden $n-1$ Knoten betrachten. (Ingrid Gerdes, 2004)

Der Suchraum des aTSP definiert sich als $(n-1)!$ anstelle von $n!$.

Der definierte Suchraum gilt jedoch nur für das aTSP, da das TSP für alle Beispiele als symmetrisches TSP festgelegt wurde kann der Suchraum weiter definiert werden als $(n-1)!/2$. Die $/2$ kommt daher das Hin- und Rückweg zwischen den Knoten in einem sTSP die gleiche Kantenbewertung habe. (Ingrid Gerdes, 2004)

Der Suchraum des sTSP definiert sich als $(n-1)!/2$ anstelle von $(n-1)!$.

3.2 Brute-Force-Algorithmen

Brute-Force-Algorithmen zählen zu den exakten Algorithmen zur Lösung von Optimierungsproblemen. Grundsätzlich unterscheiden wir zwischen zwei Ansätzen. Exakte Algorithmen, welche wir am Beispiel des naiven Brute-Force-Algorithmus durcharbeiten und

Approximative Algorithmen bzw. Heuristiken welche im einem späterem Kapitel nocheinmal genauer definiert werden.

Exakte Algorithmen definieren sich über die Fähigkeit die besagte exakte Lösung, also die Optimallösung zu finden.

3.2.1 Was sind Brute-Force-Algorithmen?

Unter den Brute-Force-Algorithmen gruppieren sich alle Algorithmen welche einer allgemeinen Problemlösungsstrategie folgen. Diese Strategie basiert auf dem Prinzip der vollständigen Suche. Brute-Force-Algorithmen generieren alle Lösungskombinationen (Permutationen) für ein Problem, durchsuchen diese systematisch ohne Rücksicht auf die Struktur des Problems oder auf bereits besuchte Bereiche des Lösungsraums und wählen die beste Lösung aus. (Berthold Vöcking, 2008)

Die Brute-Force-Methode (auch als Holzhammer-Methode bekannt) ist ein einfacher und exakter Ansatz zur Lösung von Problemen.

3.2.2 Der naive Brute-Force-Algorithmus

Die einfachste und primitivste implementierung der Brute-Force-Methode ist der naive Brute-Force-Algorithmus (oft auch nur "Brute-Force-Algorithmus" genannt). Der naive Ansatz bedient sich keiner weiteren Ansätze oder Optimierungen um die Laufzeit zu verbessern. Der Brute-Force-Algorithmus kann wie folgt definiert und implementiert werden: (Berthold Vöcking, 2008)

1. Generiere alle Lösungskombinationen (Permutationen) und speichere diese in eine Liste. Hierzu haben Programmiersprachen in der Regel vordefinierte Funktionen oder Bibliotheken.

2. Der erste Eintrag der Liste wird als vorläufige Lösung zugewiesen.

3. Alle weiteren Einträge der Liste werden systematisch durchlaufen, ist ein Eintrag besser als die zugewiesene vorläufige Lösung ersetzt besagter Eintrag die vorläufige Lösung.

4. Sind alle Permutationen durchlaufen wird die aktuelle/letzte vorläufige Lösung zur Optimallösung.

3.2.3 PseudoCode

```
1 function bruteForce(problem):
2     bestSolution = null
3     for each possibleSolution in problem.allPossibleSolutions():
4         if problem.isSolution(possibleSolution):
5             if bestSolution == null or problem.isBetter(
    possibleSolution, bestSolution):
6                 bestSolution = possibleSolution
7     return bestSolution
```

Code 1: Brute Force Pseudocode – (Selbsterstellt)

3.2.4 Brute-Force zur Lösung des TSP

Im folgenden wird der Brute-Force-Algorithmus auf das TSP angewendet und in folge dessen analysiert. Betrachtet wird die selbe TSP Instanz die bereits als Beispiel in Kapitel 2.1.7 gezeigt wurde.

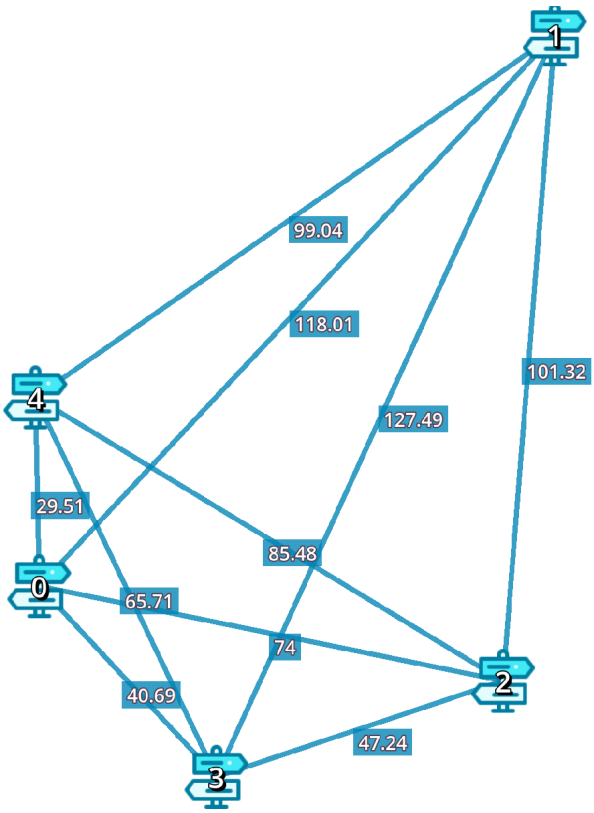


Abbildung 7: TSP Instanz – (Selbsterstellt)

Betrachtet wird zunächst was bereits gegeben ist und was noch gesucht wird.

Gegeben ist:

- Die Menge der Knoten $|V|$ mit $|V| = \{v_0, v_1, v_2, v_3, v_4\}$
- Die Anzahl der Knoten n mit $n = 5$, die sich aus der Menge der Knoten ergibt.
- Die Distanzmetrik $e_{ij} = e(i, j), \quad \forall 1 \leq i, j \leq n, \quad i \neq j : e_{ij} > 0$
- Die Distanzmatrix D mit $D = [e(i, j)]_{1 \leq i,j \leq n}$

e_{ij}	0	1	2	3	4
0	0	118.01	74	40.69	29.51
1	118.01	0	101.32	127.49	99.04
2	74	101.32	0	47.24	85.48
3	40.69	127.49	47.24	0	65.71
4	29.51	99.04	85.48	65.71	0

Tabelle 2: Distanzmatrix für $n = 5$ – (Selbsterstellt)

Gesucht wird:

- Eine Permutation $\pi : \{v_1, v_2, ..., v_n\} \rightarrow \{\pi(v_1), \pi(v_2), ..., \pi(v_n)\}$,
- So dass die Pfadlänge minimiert wird.

Schritt 1:

Generiere alle Permutationen und speichere diese in eine Liste.

```
1    from itertools import permutations
2    [...]
3        for perm in permutations(range(num_cities)):
4        [...]
```

Code 2: Brute Force Python Snippet, Schritt 1 – (Selbsterstellt)

Die Permutation werdem im Snippet nicht in eine seperate Liste geschrieben, sondern der Rückgabewert der Methode direkt in den Loop übergeben. In Python kann der import 'permutations' von 'itertools' genutzt werden, um alle Permutationen einer initialen Liste zu bekommen.

Es fällt auf das hier von allen Permutationen, also von dem Suchraum $n!$ die Rede ist. Wir durchsuchen also einen größeren Suchraum als für die definierte Instanz nötig wäre. Bei größeren Probleminstanzen müsste an dieser Stelle optimiert werden, um somit die Berechnung von unnötigen Permutationen zu vermeiden. Für eine Probleminstanz der Größe $n = 5$ kann dies jedoch im Sinne der Laufzeit vernachlässigt werden.

Schritt 2,3,4:

Der erste Eintrag der List wird als vorläufige Lösung zugewiesen.

Alle weiteren Einträge der Liste werden systematisch durchlaufen, ist ein Eintrag besser als die zugewiesene vorläufige Lösung ersetzt besagter Eintrag die vorläufige Lösung.

Sind alle Permutationen durchlaufen wird die aktuelle/letzte vorläufige Lösung zur Optimallösung.

```
1    [...]
2    def measure_path_length(p, distances):
3        length = 0
4        for i in range(len(p)):
5            length += distances[p[i-1]][p[i]]
6        return length
7
```

```
8     def brute_force(maximalNoOfCities):
9         num_cities = maximalNoOfCities
10        shortest_tour_length = float('inf')
11        shortest_tour = None
12
13        for perm in permutations(range(num_cities)):
14            tour_length = measure_path_length(perm, distances)
15            if tour_length < shortest_tour_length:
16                shortest_tour_length =tour_length
17                shortest_tour = perm
18
19        shortest_tour_cities = [cities[i] for i in shortest_tour]
20
21        print(f"Shortest tour found by brute force: {
    shortest_tour_cities} with length: {shortest_tour_length}")
22     [...]
```

Code 3: Brute Force Python Snippet, Schritt 2,3,4 – (Selbsterstellt)

In Zeile 10 des Snippets wird die länge der Tour initial auf Unendlich gesetzt, dadurch wird der erste Eintrag der Permutationsliste automatisch als vorläufige Optimallösung angenommen. Alle weiteren Einträge werden systematisch durchlaufen wie in der definition des Algorithmus beschrieben. Hat der Loop alle Einträge der Liste durchlaufen ist der Algorithmus beendet und die aktuelle/letzte vorläufige Lösung wird zur Optimallösung.

Finale Lösung mit Brute-Force:

Der vollständige Programmcode kann im Anhang eingesehen werden.

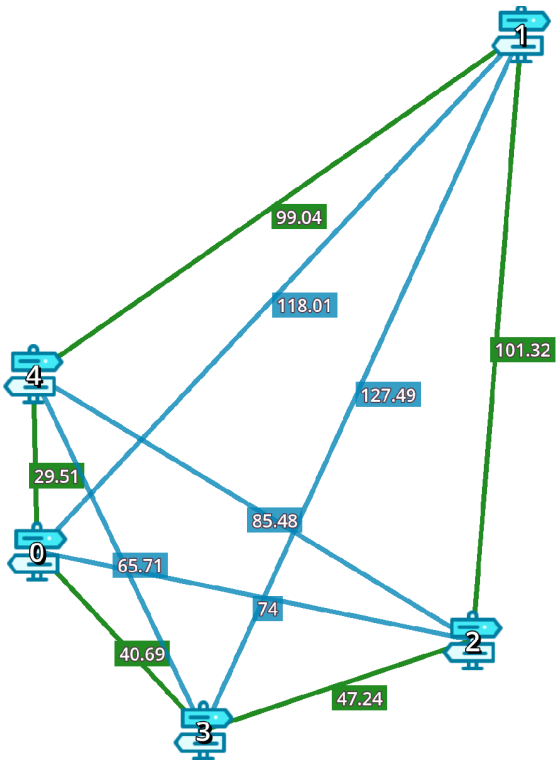


Abbildung 8: TSP Instanz Optimallösung mit Brute-Force – (Selbsterstellt)

3.2.5 Laufzeit und Speicherkomplexität

Lädt man nun die Distanzmatrix als Textdatei in das Programm und lässt es durchlaufen bekommt man folgende Informationen:

n	Länge	Tour	Rechenzeit [sec]
5	317.8	$[v_0, v_4, v_1, v_2, v_3]$	0.000215

Tabelle 3: Brute-Force Output – (Selbsterstellt)

Zusätzlich zu der spezifischen Laufzeit aus der Ausgabe, kann aus dem Code selbst die Laufzeitkomplexität ermittelt werden.

Die Laufzeitkomplexität ist hauptsächlich durch die permutations() Funktion bestimmt, die alle Permutationen der Knoten erzeugt. Da es n Städte gibt, gibt es $n!$ Permutationen, und jede Permutation wird einmal durchlaufen und bewertet. Daher ist die Laufzeitkomplexität dieses Algorithmus $O(n!)$.(Wikipedia, 2024n)

Die Speicherkomplexität ist hauptsächlich durch den Speicher bestimmt, der benötigt wird, um die Distanzen zwischen den Knoten und die aktuell beste Tour zu speichern. Die Distanzmatrix hat eine Größe von $n \times n$ und die beste Tour hat eine Größe von n. Daher ist die Speicherkomplexität dieses Algorithmus $O(n^2)$.

3.2.6 Vorteile, Nachteile und Grenzen

Vorteile:

- Der Algorithmus ist simpel und einfach zu implementieren.
- Der Algorithmus ist exakt und hat somit eine Garantie der Optimallösung.
- Der Algorithmus hängt nicht in lokalen Optima fest.

Nachteile:

- Der Algorithmus hat eine exponentielle Laufzeitkomplexität.
- Der Algorithmus ist statisch und unflexibel.
- Der Algorithmus kann trotz linearer Speicherkomplexität große Speicheranforderungen für große n-Werte haben.

Grenzen und Extremfälle:
Extremfälle für Brute-Force-Algorithmen sind sehr große Eingaben bzw. Probleminstanzen, welche eine große Anzahl potenzieller Lösungen haben. Für solche Fälle kann der Algorithmus vergleichsweise lange laufen was das bestimmen der exakte Lösung unpraktisch werden lässt.

Die Laufzeit wurde bereits für eine kleine Instanz mit $n = 5$ berechnet. In folgender Tabelle werden nun Laufzeiten für $[n = 6, n = 7, \ldots, n = 13]$ aufgelistet. An der Tabelle lässt sich die exponentiell ansteigende Laufzeit gut visualisieren. Läuft der Algorithmus für $n = 5$ noch unter einer Sekunde, braucht er für $n = 13$ bereits 2.8 Stunden. Rechnet man die Laufzeitwerte hoch für beispielsweise $n = 16$ läuft der Algorithmus bereits zwischen 5 und 7 Tagen (Abhängig von der Hardware).

Dabei sind Werte wie $n = 16$ noch klein. Viele Anwendungen arbeiten mit Werten für n die im Bereich von mehreren Hundert bis hin zu Tausend oder sogar noch höher liegen.

n	Rechenzeit [sec]
5	0.000215
6	0.000881
7	0.007182
8	0.045710
9	0.452157
10	4.882488
11	62.188478
12	778.518702
13	10304.870181

Tabelle 4: Brute-Force Laufzeiten – (Selbsterstellt)

3.2.7 Warum nicht Brute-Force?

Obwohl Brute-Force-Algorithmen einfach zu verstehen und zu implementieren sind, wurde bereits aufgezeigt, dass diese für umfangreiche Probleminstanzen ungeeignet sind. Da das TSP in realen Anwendungen in der Regel aber mit großen Eingabedatenmengen arbeitet, werden meist effizientere Algorithmen bevorzugt. (Berthold Vöcking, 2008)

3.3 Approximative Algorithmen (Heuristiken)

Wie im vorherigen Kapitel bereits aufgezeigt ist der Brute-Force Ansatz für große Probleminstanzen ungeeignet. Damit schließt sich der Bogen zu den Approximativen Algorithmen, bzw. den Heuristiken. Zunächst soll jedoch geklärt werden was Approximative Algorithmen und Heuristiken sind. Denn Approximative Algorithmen und Heuristiken sind zwar verwandte, jedoch nicht identische Konzepte. (Volker Turau, 1996)

Approximative Algorithmen (auch Approximationsalgorithmen genannt) sind spezifische Algorithmen die dazu entworfen sind, schnelle Lösungen für Optimierungsprobleme zu finden, die nicht notwendigerweise optimal sind, aber eine gewisse Garantie über die Güte der Lösung bieten. Diese Garantien können beispielsweise in Form von Leistungsverhältnissen (approximationsfaktoren) ausgedrückt werden die angeben, wie nah die Lösung am Optimum liegt. Approximative Algorithmen liefern oft Lösungen, die in polynomieller Zeit ausgeführt werden können und für bestimmte Problemstellungen ausreichend genau sind. (Volker Turau, 1996)

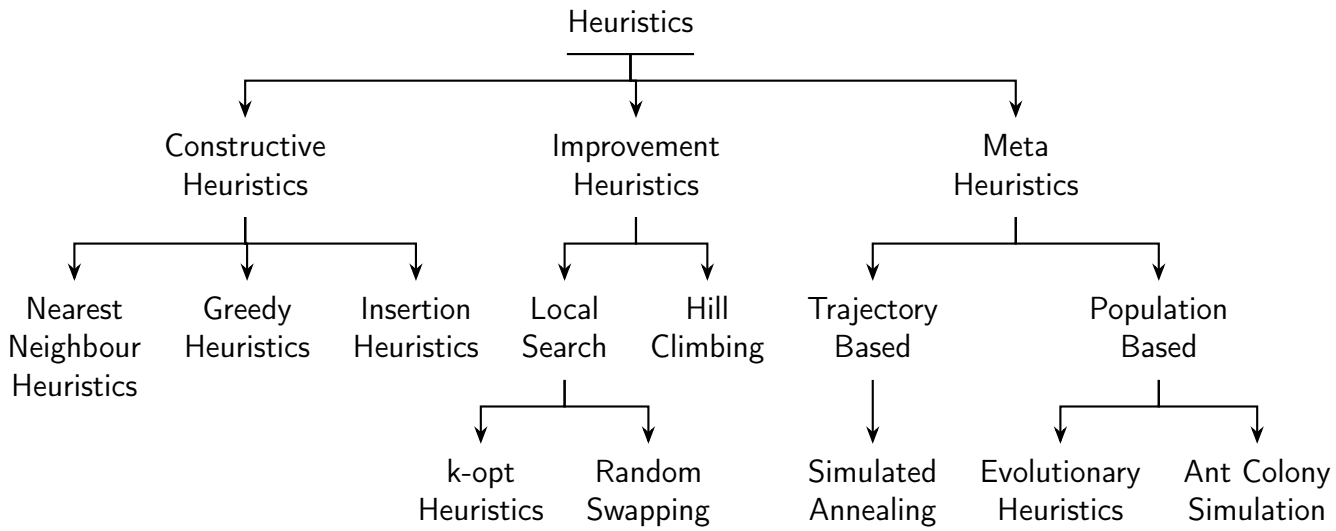
Heuristiken sind dagegen allgemeinere Suchstrategien oder -techniken, die verwendet werden, um schnell Lösungen für Optimierungsprobleme zu finden, ohne eine explizite Garantie für die Qualität der Lösung zu bieten. Heuristiken basieren oft auf Regeln, Erfahrung oder Intuition, um Kandidatlösungen zu generieren und zu verbessern. Im Gegensatz zu exakten Methoden können Heuristiken schnell sein und auch für Probleme eingesetzt werden, die in ihrer vollen Komplexität schwer zu lösen sind. Sie bieten jedoch keine Garantien für die Qualität der gefundenen Lösung, und ihre Leistung kann je nach Problemstellung und Implementierung variieren. (Volker Turau, 1996)

Der Übergang zwischen Algorithmus und Heuristik ist fließend: Eine Heuristik ist eine Technik, aus unvollständigen Eingangsdaten zu möglichst sinnvollen Ergebnissen zu

gelangen. Viele heuristische Vorgehensweisen sind selbst exakt definiert und damit Algorithmen. Bei manchen ist jedoch nicht in jedem Schritt genau festgelegt, wie vorzugehen ist. Das bedeutet der Anwender muss "günstig raten". Sie können nicht (vollständig) als Algorithmus formuliert werden.

Mit anderen Worten, alle Approximationsalgorithmen sind Heuristiken, aber nicht alle Heuristiken sind Approximationsalgorithmen. Approximationsalgorithmen sind eine spezielle Art von Heuristiken, die eine bestimmte Garantie bezüglich der Güte der Lösung bieten. (Volker Turau, 1996)

Die Heuristischen Methoden lassen sich grob in Gruppen kategoresieren. Dabei ist auch hier der Übergang fließend und einige Algorithmen implementieren mehrere Heuritische Aspekte oder lassen sich mehreren Gruppen zuordnen (Grafik in Latex selbst erstellt).



3.4 Constructive-Heuristiken

Konstruktionsheuristiken

- starten mit einer leeren Lösung.
- bauen die Lösung schrittweise auf.
- terminieren, wenn die Lösung vollständig ist.

Constructive-Heuristiken beziehen sich auf Methoden oder Ansätze, die verwendet werden, um Lösungen für komplexe Probleme zu finden. Diese Heuristiken bauen schrittweise eine Lösung auf, indem sie eine Folge von Entscheidungen treffen, die jeweils eine Teillösung erweitern. Im Allgemeinen folgen sie keiner exakten Methode zur Problemlösung, sondern nutzen intuitive, erfahrungsbasierte Regeln, um praktikable und oft nahe an optimalen Lösungen zu finden. (Wikipedia, 2024m)

3.5 Greedy-Heuristiken

Greedy-Heuristiken sind Vorgehensweisen, die bei jedem Schritt die lokal beste Wahl treffen, in der Hoffnung, dass diese lokalen Entscheidungen zu einer global optimalen Lösung führen. (Matai u. a., 2010)

3.5.1 Der Greedy Algorithmus

1. **Initialisierung:** Erstelle eine leere Tour und zwei leere Listen, vertex_visited1 und vertex_visited2, um die besuchten Knoten zu verfolgen.

2. **Kantenwahl:** Durchlaufen werden alle Kanten in der Distanzmatrix. Für jede Kante, dessen zwei Knoten je einen $Grad <= 2$ haben (d.h. kein Knoten der Kante ist in vertex_visited2 oder allgemein Formuliert: Der Grad ist die Anzahl an Kanten die von einem Knoten ausgehen.), wird überprüft, ob das Hinzufügen dieser Kante zur Tour einen ungültigen Zyklus erzeugt. Wenn dies nicht der Fall ist und die Kante das kleinste Gewicht aller geprüften Kanten hat, wird diese Kante als minimum_edge makiert und das zugehörige Knotenpaar als minimum_edge_pair.

3. **Kantenaktualisierung:** Wenn eine gültige Kante gefunden wurde, wird die Distanzmatrix aktualisiert, indem das Gewicht der minimum_edge auf Unendlich gesetzt wird um anzuzeigen, dass diese Kante bereits besucht wurde. Anschließend wird das minimum_edge_pair zur Tour hinzugefügt.

4. **Knotenaktualisierung:** Die Listen vertex_visited1 und vertex_visited2 werden mit den Knoten von minimum_edge_pair aktualisiert. Wenn ein Knoten noch nicht in vertex_visited1 ist, wird dieser hinzugefügt. Wenn er bereits in vertex_visited1 ist, wird er in vertex_visited2 hinzugefügt.

5. **Rekursion und Ausstiegsbedingung:** Wiederholt werden die Schritte 2 bis 4 bis keine gültige Kante mehr gefunden wird.

(Matai u. a., 2010)

Nochmal hervorgehoben, eine Kante ist ungültig wenn sie mindestens eine der folgenden Bedingungen erfüllt.

- Die Kante bereits besucht, also der Tour hinzugefügt wurde.
- Einer der Knoten der die Kante definiert hat einen $Grad > 2$.
- Das einfügen der Kante erzeugt einen ungültigen Zyklus, ein Zyklus in welchem nur eine Teilmenge aller Knoten enthalten sind.

3.5.2 PseudoCode

```
function greedy(problem):
    solution = []
    while problem is not solved:
        bestOption = chooseBestOption(problem)
        if bestOption is valid:
            apply bestOption to problem
            add bestOption to solution
    return solution
```

Code 4: Greedy Algorithmus Pseudocode – (Selbsterstellt)

3.5.3 Greedy Ansatz zur Lösung des TSP

Im folgenden wird der Greedy-Algorithmus auf das TSP angewendet und in folge dessen analysiert. Betrachtet wird die selbe TSP Instanz die bereits als Beispiel in Kapitel 2.1.7 und für die Lösung mit Brute-Force (Kapitel 3.2.4) genutzt wurde.

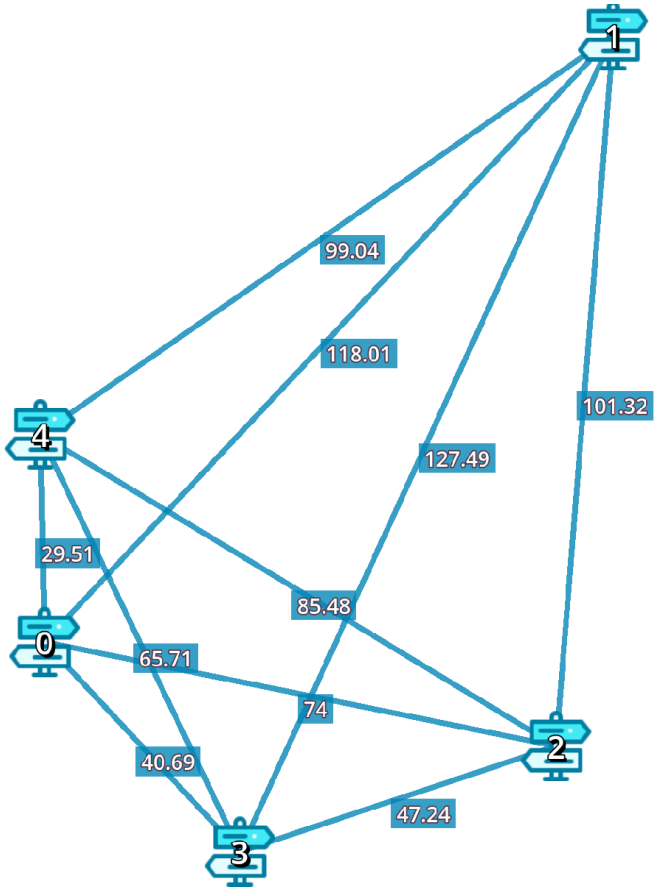


Abbildung 9: TSP Instanz – (Selbsterstellt)

Betrachtet wird zunächst was bereits gegeben ist und was noch gesucht wird. Gegeben und Gesucht sind dabei die selben Punkte wie bereits beim Brute-Force Ansatz (Kapitel 3.2.4) .

Schritt 1:

Erstelle eine leere Tour und zwei leere Listen, vertex_visited1 und vertex_visited2, um die besuchten Knoten zu verfolgen.

```
1 func _recursive_greedy_heuristic_variation(edge_matrix:Array,
    vertex_visited1:Array,vertex_visited2:Array,tour:Array) -> Array:
2 [...]
```

Code 5: Greedy Algorithmus gdscript Snippet, Schritt 1 – (Selbsterstellt)

Das es sich um eine Rekursive implementierung handelt werden die Listen initial als leer in die Funktion übergeben.

Schritt 2:

Durchlaufen werden alle Kanten in der Distanzmatrix. Für jede Kante, dessen Knoten einen Grad kleiner 2 haben (d.h. kein Knoten der Kante ist in vertex_visited2), wird überprüft, ob das Hinzufügen dieser Kante zur Tour einen ungültigen Zyklus erzeugt. Wenn dies nicht der Fall ist und die Kante das kleinste Gewicht aller geprüften Kanten hat, wird diese Kante als minimum_edge makiert und das zugehörige Knotenpaar als minimum_edge_pair.

```
    [...]
  var minimum_edge:float = INF
  var minimum_edge_pair:Vector2 = Vector2.INF

  var valid_edge_found:bool = false
  for i in range(len(edge_matrix)):
    if not vertex_visited2.has(float(i)):
      for j in range(len(edge_matrix)):
        if not vertex_visited2.has(float(j)):
          if i != j:
            if edge_matrix[i][j] < minimum_edge:
              var tour_copy:Array = tour.duplicate()
              tour_copy.append(Vector2(i,j))
              if _creates_invalid_cycle(tour_copy) == false:
                minimum_edge = edge_matrix[i][j]
                minimum_edge_pair = Vector2(i,j)
                valid_edge_found = true
    [...]
```

Code 6: Greedy Algorithmus gdscript Snippet, Schritt 2 – (Selbsterstellt)

Schritt 3:

Wenn eine gültige Kante gefunden wurde, wird die edge_matrix aktualisiert, indem das Gewicht der minimum_edge auf INF gesetzt wird um anzuzeigen, dass diese Kante bereits besucht wurde. Anschließend wird das minimum_edge_pair zur Tour hinzugefügt.

```
    [...]
  if valid_edge_found:
    edge_matrix[minimum_edge_pair.x][minimum_edge_pair.y] = INF
    edge_matrix[minimum_edge_pair.y][minimum_edge_pair.x] = INF
    tour.append(minimum_edge_pair)
        [...]
```

Code 7: Greedy Algorithmus gdscript Snippet, Schritt 3 – (Selbsterstellt)

Schritt 4:

Die Listen vertex_visited1 und vertex_visited2 werden mit den Knoten von minimum_edge_pair aktualisiert. Wenn ein Knoten noch nicht in vertex_visited1 ist, wird dieser hinzugefügt. Wenn er bereits in vertex_visited1 ist, wird er in vertex_visited2 hinzugefügt.

```
    [...]
  if valid_edge_found:
        [...]
    if not vertex_visited1.has(minimum_edge_pair.x):
      vertex_visited1.append(minimum_edge_pair.x)
```

```
6     elif not vertex_visited2.has(minimum_edge_pair.x): vertex_visited2.
    append(minimum_edge_pair.x)
7
8     if not vertex_visited1.has(minimum_edge_pair.y):
9       vertex_visited1.append(minimum_edge_pair.y)
10    elif not vertex_visited2.has(minimum_edge_pair.y): vertex_visited2.
    append(minimum_edge_pair.y)
11        [...]
```

Code 8: Greedy Algorithmus gdscript Snippet, Schritt 4 – (Selbsterstellt)

Schritt 5:

Wiederholt werden die Schritte 2 bis 4 bis keine gültige Kante mehr gefunden wird.

```
1     [...]
2   if valid_edge_found:
3         [...]
4                         _recursive_greedy_heuristic_variation(
    edge_matrix,vertex_visited1,vertex_visited2,tour)
5
6   return tour
```

Code 9: Greedy Algorithmus gdscript Snippet, Schritt 5 – (Selbsterstellt)

Wenn eine gültige Kante gefunden wurde ruft die Funktion sich selbst auf, andernfalls wird das IF-Statement ignoriert und die Tour zurückgegeben.

Aufbau der Tour durch die Rekursion:

Da der Programmcode und auch die Beschreibung auf den ersten Blick ein wenig abstrakt und schwer nachvollziehbar wirken, wird der Algorithmus an einigen Bildern nochmal erklärt. Es werden dabei bewusst auf Listen und Programmcode spezifische Formulierungen verzichtet. Der Algorithmus wird manuell und in logischen Schritten durchgegangen.

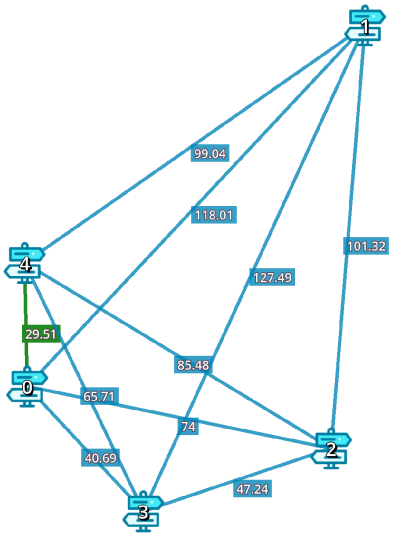


Abbildung 10: Greedy-Algorithmus Schritt 1 – (Selbsterstellt)

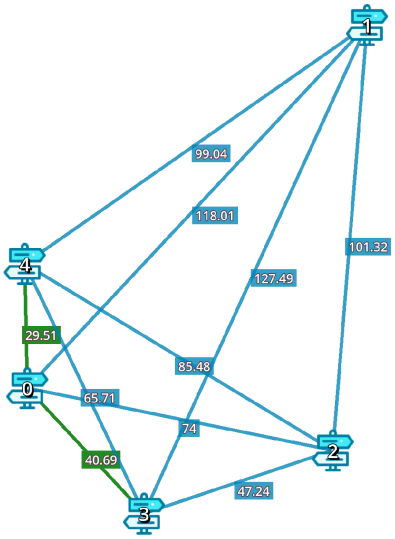


Abbildung 11: Greedy-Algorithmus Schritt 2 – (Selbsterstellt)

In Abbildung 10 sehen wir den ersten Schritt des Greedy-Algorithmus. Dieser hat aus allen noch nicht besuchten Kanten die kürzeste gewählt, $e(0,4)$. Das gleiche passiert in Abbildung 11, aus allen unbesuchten Kanten (dies schließt die eben besuchte Kante $e(0,4)$ aus) wird nun erneut die kürzeste Kante gewählt, $e(0,3)$.

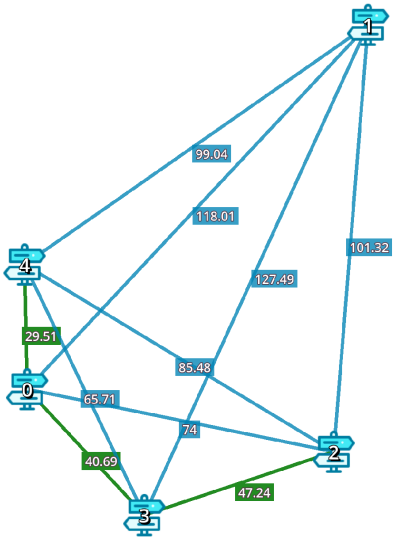


Abbildung 12: Greedy-Algorithmus Schritt 3 – (Selbsterstellt)

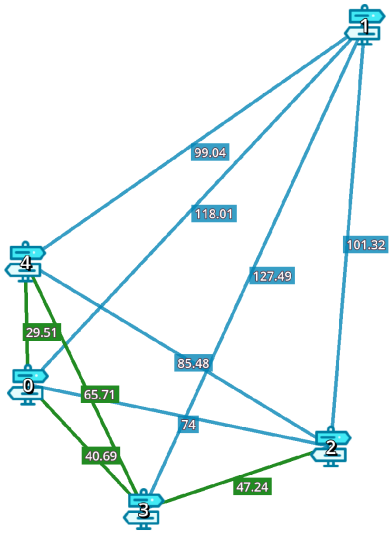


Abbildung 13: Greedy-Algorithmus Schritt 4 – (Selbsterstellt)

Es wird erneut die kürzeste unbesuchte Kante gewählt, $e(2,3)$.

In Abbildung 13 kommt es nun erstmals zur Verletzung einer Bedingung zur Gültigkeit der Kante. Die kürzeste Kante nach $e(2,3)$ wäre $e(3,4)$, jedoch verletzt diese gleich zwei Bedingungen. Wir können visuell erkennen das ein ungültiger Zyklus entsteht. Zusätzlich gehen von Knoten v_3 nun 3 Kanten aus, jedoch darf in einem geschlossen Zyklus jeder

Knoten einen maximalen Grad von 2 besitzen. Damit wird die Kante ungültig und wir schauen auf die nächsten Kanten.

Die nächsten Kanten welche sowohl unbesucht als auch minimal sind wären $e(1,3)$ und $e(2,4)$, jeweils zu sehen in Abbildung 14 und 15. Die Kante $e(1,3)$ erzeugt erneut einen $Grad > 2$ für v_3 während $e(2,4)$ zwar nicht den Grad der verbunden Knoten über 2 setzt, jedoch einen ungültigen Zyklus erzeugt.

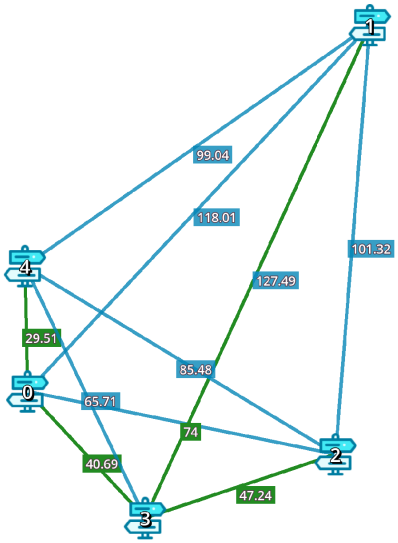


Abbildung 14: Greedy-Algorithmus Schritt 5 – (Selbsterstellt)

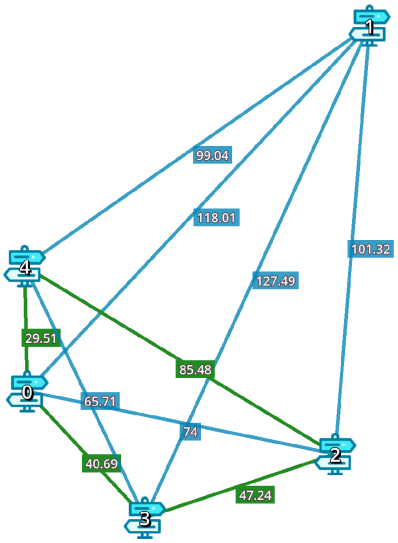


Abbildung 15: Greedy-Algorithmus Schritt 6 – (Selbsterstellt)

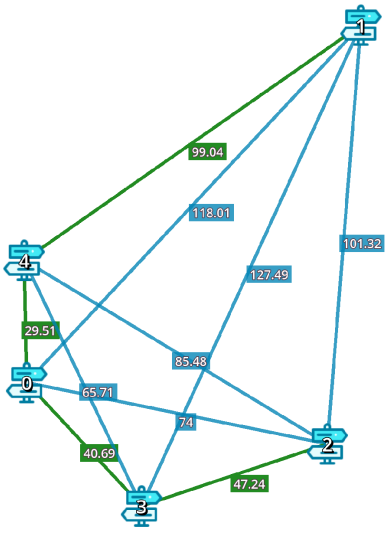


Abbildung 16: Greedy-Algorithmus Schritt 7 – (Selbsterstellt)

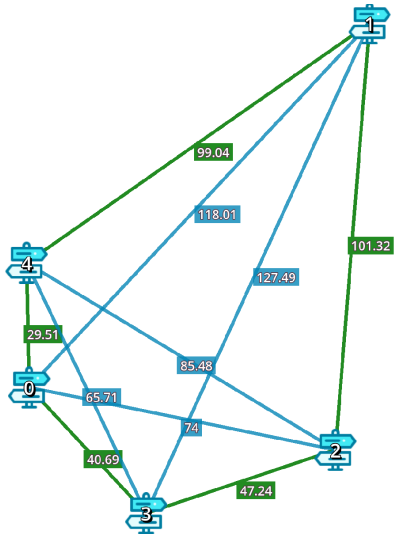


Abbildung 17: Greedy-Algorithmus Schritt 8,9,10 – (Selbsterstellt)

Die nächste gültige, unbesuchte und minimale Kante wäre $e(1,4)$, welche ohne Probleme zur Tour hinzugefügt werden kann.

In Abbildung 17 sind die nachfolgenden 3 Schritte direkt zusammengefasst. Sowohl $e(0,1)$ als auch $e(1,3)$ sind erneut ungültig, wodurch $e(1,2)$ die letzte gültige Kante im Graphen ist und den Garphen vervollständigt.

Weitere Durchläufe der Rekursion würden keine gültigen Kanten finden und der Algorithmus ist beendet. In unserem Fall ist zudem die gefunden Lösung des Greedy Algorithmus auch die Optimallösung.

Der vollständige Programmcode zur Lösung auf Github eingesehen werden.

3.5.4 Laufzeit und Speicherkomplexität

Der gegebene Code implementiert eine Variation des Greedy-Algorithmus zur Lösung des Traveling Salesman Problem. Der Algorithmus sucht nach der kürzesten Kante, die keinen ungültigen Zyklus erzeugt, und fügt sie dem Tour hinzu. Dieser Prozess wird wiederholt, bis alle Kanten besucht wurden.

Die Laufzeitkomplexität dieses Algorithmus ist $O(n^4)$.

Der äußere Loop läuft n Mal, wobei n die Anzahl der Knoten ist. Innerhalb des äußeren Loop gibt es einen inneren Loop, der ebenfalls n Mal läuft. Dies führt zu einer Quadratkomplexität von n^2. Innerhalb des inneren Loop gibt es eine Funktion _creates_invalid_cycle(), die im schlimmsten Fall eine Komplexität von n^2 hat, da sie einen For-Loop und einen While-Loop enthält, die beide bis zu n Mal laufen können. Daher ergibt die Kombination dieser Faktoren eine Gesamtlaufzeitkomplexität von n^4.

Die Speicherkomplexität dieses Algorithmus ist $O(n^2 + n)$.

Es gibt drei Arrays im Funktionsaufruf, die jeweils bis zu n Elemente speichern können. Darüber hinaus gibt es in der Funktion _creates_invalid_cycle() ein Array, das bis zu n Elemente speichern kann. Daraus ergibt sich die Speicherkomplexität $O(n)$. Zählt man die Komplexität mit der Speicherkomplexität der Distanzmatrix ($O(n^2)$) zusammen erhält man die Gesamtspeicherkomplexität $O(n^2 + n)$.

3.5.5 Vorteile, Nachteile und Grenzen

(Donald Davendra, 2010; Ingrid Gerdes, 2004; Wikipedia, 2024c)

Vorteile:

- Der Algorithmus ist einfach zu implementieren.
- Der Algorithmus hat eine polynomielle Laufzeitkomplexität.
- Der Algorithmus ist flexibler als der Brute-Force Ansatz.
- Der Algorithmus hält die Lösung normalerweise innerhalb von 15% bis 20% der Held-Karp Lower-Bound. (Matai u. a., 2010)

Nachteile:

- Der Algorithmus liefert keine Garantie auf die Optimallösung.
- Der Algorithmus hat eine höhere Speicherkomplexität als der Brute-Force Ansatz.

- Der Algorithmus erziehlt mit höherer Wahrscheinlichkeit bessere Ergebnisse bei Problemen, welche die "Greedy-Choice"-Eigenschaft besitzen.

Grenzen und Extremfälle: Grenzen für Greedy-Algorithmen sind in der Regel Probleme, bei denen die "Greedy-Choice"-Eigenschaft nicht gilt. In solchen Fällen kann der Algorithmus eine suboptimale Lösung liefern.

Die "Greedy-Choice"-Eigenschaft ist eine Schlüsseleigenschaft von Greedy-Algorithmen, die besagt, dass eine globale Optimallösung durch die Auswahl einer optimalen Wahl in jeder Entscheidungsstufe erreicht werden kann.

Im Kontext des Traveling Salesman Problems (TSP) ist es jedoch nicht immer möglich, im Voraus zu bestimmen, ob eine bestimmte Instanz des Problems die "Greedy-Choice"-Eigenschaft hat.

Es gibt jedoch bestimmte Fälle, in denen der Greedy-Algorithmus für das TSP die optimale Lösung liefern kann. Ein solcher Fall ist beispielsweise, wenn die Distanzen zwischen den Knoten die Dreiecksungleichung erfüllen (d.h. das TSP ist metrisch). In solchen Fällen kann man sagen, dass die TSP-Instanz die "Greedy-Choice"-Eigenschaft hat. (Prof. Dr. S. Albers, [o. D.])

Es sei nochmal angemerkt das die "Greedy-Choice"-Eigenschaft keine Lösung garantiert. Genauso kann der Greedy-Algorithmus auch in einem TSP ohne "Greedy-Choice"-Eigenschaft eine Optimallösung finden.

3.6 Heuristiken überprüfen mit Lower-Bounds

Um die Effizienz von Heuristiken beim Traveling Salesman Problem zu bestimmen, kann ein Minimum Spanning Tree (MST) als untere Schranke (Lower-Bound) verwendet werden. Dies hilft zu verstehen, wie nah eine gefundene Lösung an der optimalen Lösung liegt.

Ein MST eines Graphen ist ein Baum, der alle Knoten verbindet, ohne Zyklen zu bilden, und die geringstmögliche Gesamtkantenlänge hat.

Da ein MST eine Verbindung zwischen allen Knoten mit der geringstmöglichen Gesamtkantenlänge darstellt, kann die Gesamtlänge des MST als eine untere Schranke für die TSP-Gesamtlänge betrachtet werden. Dies liegt daran, dass der optimale TSP-Zyklus alle Städte verbindet und in der Regel etwas mehr als die MST-Länge betragen wird, weil er zusätzlich eine Rückkehr zur Ausgangsstadt erfordert.

Schrittweise Durchführung:

1. **MST erstellen:**

   Bestimme das MST für den Graphen der Knoten mit den gegebenen Kantenlängen.

2. **MST-Länge berechnen:**

   Summiere die Längen aller Kanten im MST, um die Gesamtlänge zu erhalten.

3. **MST-Länge und TSP-Lange vergleichen:**

   Vergleiche die Länge des MST mit der Länge der durch die Heuristik gefundenen TSP-Lösung.

4. **Lower-Bound richtig auslesen:**

   Die MST-Länge dient als untere Schranke für die TSP-Lösung. Das bedeutet, dass die optimale TSP-Lösung mindestens so groß sein muss wie die MST-Länge. Wenn die Länge der heuristischen TSP-Lösung nur geringfügig größer als die MST-Länge ist, ist die Heuristik effizient. Wenn sie wesentlich größer ist, könnte die Heuristik ineffizient sein.

Durch die Verwendung der MST-Länge als untere Schranke kann also beurteilt werden, wie gut eine Heuristik im Vergleich zur optimalen Lösung abschneidet.
(Donald Davendra, 2010; David L. Applegate, 2006; Berthold Vöcking, 2008; Volker Turau, 1996)

Den Zusammenhang zwischen MST und TSP kann man auch an einigen Bildern erkennen welche beide Lösungen gegenüberstellen. Dabei ist links die Optimallösung welche mit dem naiven Brute-Force-Algorithmus generiert wurde und rechts die Lösung des MST mit dem Prim Algorithmus.

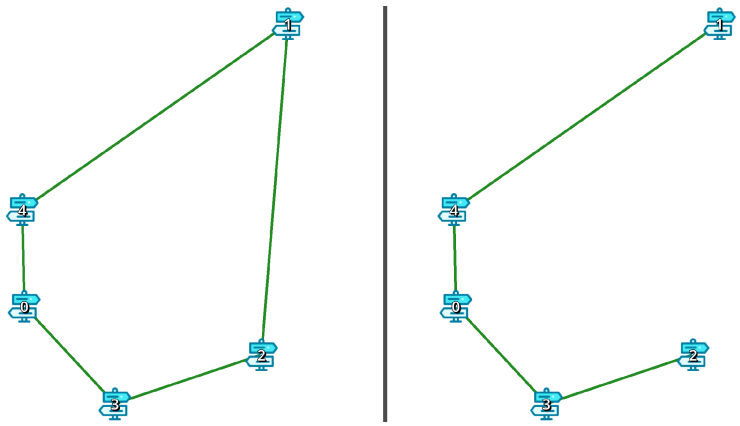


Abbildung 18: TSP MST Vergleich Nr. 1; 4 von 5 identische Kanten – (Selbsterstellt)

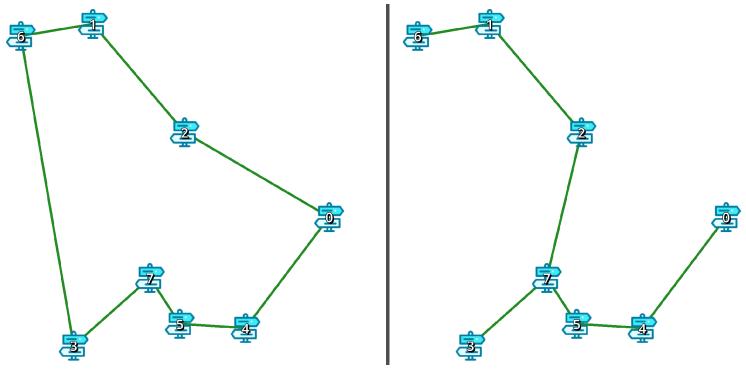


Abbildung 19: TSP MST Vergleich Nr. 2; 6 von 8 identische Kanten – (Selbsterstellt)

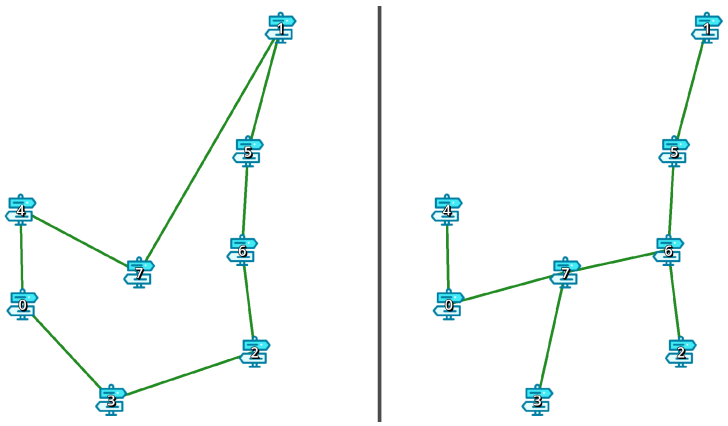


Abbildung 20: TSP MST Vergleich Nr. 3, 4 von 8 identische Kanten – (Selbsterstellt)

3.7 Meta-Heuristiken

Metaheuristiken können sowohl Constructive als auch Improvement Strategien nutzen. Sie haben einen globalen Suchraum und können optional schlechtere Ergebnisse akzeptieren, um ein globales Optimum zu finden.

Metaheuristiken definieren sich dadurch, eine oder mehrere Heuristiken zu finden, zu generieren, zu verändern oder auszuwählen, die eine hinreichend gute Lösung für ein Optimierungsproblem bieten können, insbesondere bei unvollständigen oder unvollkommenen Informationen oder begrenzter Rechenleistung. (Wikipedia, 2024m)

3.7.1 Christofides-Algorithmus

Ein 1976 unabhängig von Nicos Christofides und Anatoli I. Serdjukow beschriebener Approximationsalgorithmus ergab eine Rundreise, die maximal um die Hälfte länger ist als die optimale Tour.

Der Christofides-Algorithmus ist eine Metaheuristik zur Lösung des Traveling Salesman Problem in metrischen Graphen, bzw. metrischen TSP-Instanzen. Er bietet eine garantierte Annäherung an die optimale Lösung und nutzt den Minimum Spanning Tree als Ausgangspunkt.

1. Erzeuge den Minimum Spanning Tree.
2. Finde alle Knoten im MST, die eine ungerade Anzahl von Verbindungen (einen ungeraden Grad) haben.
3. Führe ein minimales perfektes Matching auf den ungeraden Knoten durch. Das bedeutet, dass Paare dieser ungeraden Knoten so verbunden werden, dass die Gesamtlänge der Verbindungen minimal ist und jeder Knoten genau einmal verbunden wird.
4. Kombiniere die Kanten des MST mit den Kanten des minimalen perfekten Matchings, um einen Eulergraphen zu erhalten. Dieser Graph enthält alle Knoten und hat durch das Matching jetzt eine gerade Anzahl von Verbindungen für jeden Knoten.
5. Finde einen Eulerkreis in diesem Eulergraphen, der jede Kante genau einmal durchläuft. Erzeuge daraus einen Hamiltonkreis, indem doppelt besuchte Knoten entfernt werden, während die Reihenfolge der Knoten beibehalten wird.

Der Christofides-Algorithmus bietet eine garantierte Annäherung von höchstens dem 1,5-fachen der optimalen Lösung für das metrische TSP. Das bedeutet, dass die Gesamtlänge der vom Algorithmus gefundenen TSP-Tour höchstens 50% länger ist als die optimale Lösung.

Erklärung der Garantie:

Die Gesamtlänge des MST ist ein Lower-Bound für die optimale TSP-Lösung, da ein TSP-Zyklus alle Knoten verbinden muss und die geringste Verbindung aller Knoten mindestens so lang ist wie der MST. Das minimale perfekte Matching auf den ungeraden Knoten hat eine Gesamtlänge, die ebenfalls durch die MST-Länge begrenzt ist. Da es sich nur auf ungerade Knoten beschränkt, ist die Gesamtlänge des Matchings nicht mehr als die Hälfte der MST-Länge. Die Länge des Eulergraphen, also die Kombination von MST und dem minimalem perfekten Matching, ist somit höchstens 1,5-mal die MST-Länge. Da der Eulerkreis zu einem Hamiltonkreis verkürzt wird, indem doppelte Besuche entfernt werden, bleibt die Länge der endgültigen TSP-Tour ebenfalls höchstens 1,5-mal die Länge des MST, was die garantierte Annäherung des Christofides-Algorithmus an die optimale Lösung erklärt.

(Wikipedia, 2024b; Wikipedia, 2024m; Wikipedia, 2024n; Donald Davendra, 2010)

4 Freiwillige Lektüre: Weitere Einblicke

4.1 Improvement-Heuristiken

Verbesserungsheuristiken (Wikipedia, 2024m)

- starten mit einer gültigen Lösung.
- verändern in mehreren Schritte Teile der Lösung, um sie zu verbessern.
- terminieren, wenn keine Verbesserung mehr möglich ist

4.1.1 k-opt

Das k-opt Verfahren ist eine Improvement-Heuristik zur Verbesserung einer gegebenen Route im TSP durch den Austausch von k Kanten. Ein spezieller Fall davon ist das 2-opt Verfahren, bei dem zwei Kanten aus der Rundreise entfernt und anschließend eingefügt werden. Dies kann zu einer kürzeren Gesamtroute führen.

Die Wahl der zu tauschenden Kanten kann auf verschiedene Weisen erfolgen. Eine Möglichkeit ist die zufällige Auswahl (Random Neighbor). Eine andere Möglichkeit ist die sequentielle Durchsicht aller möglichen Kantenpaare und die Auswahl des ersten Paares, das zu einer Verbesserung führt (Next Improvement). Eine dritte Möglichkeit ist die Durchsicht aller möglichen Kantenpaare und die Auswahl des besten (Best Improvement). Die Wahl der Strategie kann einen großen Einfluss auf die Effizienz des Algorithmus haben.

k-opt Verfahren sind relativ einfach zu implementieren und können oft erhebliche Verbesserungen gegenüber der ursprünglichen Route erzielen. Es ist jedoch wichtig zu beachten, dass sie keine Garantie für die Findung der optimalen Lösung bieten oder überhaupt eine Verbesserung zu erzielen. In vielen Fällen können sie jedoch erhebliche Verbesserungen gegenüber der ursprünglichen Route erzielen.

Ein Nachteil des k-opt Verfahrens ist, dass es oft in lokalen Optima stecken bleibt. Dies liegt daran, dass es nur kleine, lokale Änderungen an der aktuellen Lösung vornimmt und daher möglicherweise nicht in der Lage ist eine deutlich bessere Lösung zu finden, welche eine größere Änderung erfordern würde. Darüber hinaus kann die Wahl von k einen großen Einfluss auf die Effizienz des Algorithmus haben. Ein zu großes k kann den Algorithmus verlangsamen. Ein zu kleines k kann dazu führen, dass der Algorithmus in lokalen Optima stecken bleibt. (Donald Davendra, 2010; Sophia Heimann; Hung P. Hoang; Stefan Hougardy, 2024; Wikipedia, 2024a)

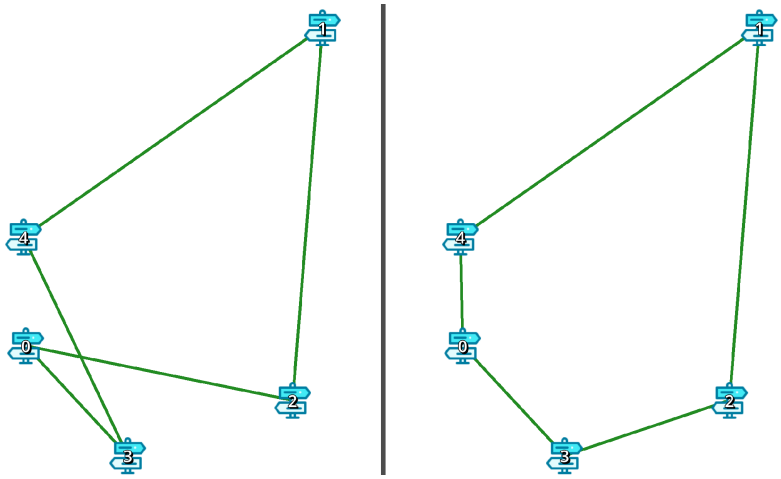


Abbildung 21: TSP Optimallösung durch 2-opt – (Selbsterstellt)

4.1.2 Simulated-Annealing

Simulated Annealing ist eine metaheuristische Methode zur Lösung von Optimierungsproblemen, die vom Prozess des Abkühlens von Metallen inspiriert ist. Im Kontext des TSP funktioniert Simulated Annealing wie folgt:

Zunächst wird eine zufällige Lösung generiert, die als Ausgangspunkt dient. Dann wird eine Nachbarlösung durch eine kleine Änderung der aktuellen Lösung erzeugt. Wenn die Nachbarlösung besser ist als die aktuelle Lösung, wird sie akzeptiert. Andernfalls wird sie mit einer gewissen Wahrscheinlichkeit basierend auf der Temperatur des Systems akzeptiert. Die Temperatur wird im Laufe der Zeit reduziert, wodurch die Akzeptanzwahrscheinlichkeit von schlechteren Lösungen sinkt. Dadurch hat das System die Möglichkeit, aus lokalen Optima zu entkommen und bessere Lösungen zu finden.

Ein wichtiger Eckpunkt von Simulated Annealing ist die Wahl der Abkühlungsfunktion, die bestimmt, wie schnell die Temperatur reduziert wird. Ein zu schnelles Abkühlen kann dazu führen, dass das Verfahren in lokalen Optima stecken bleibt, während ein zu langsames Abkühlen zu einer längeren Laufzeit führen kann.

Vorteile von Simulated Annealing sind seine Fähigkeit, globale Optima zu finden, und seine Anpassungsfähigkeit an verschiedene Problemstellungen. Es ist auch relativ einfach zu implementieren. Nachteile sind die Notwendigkeit, mehrere Parameter wie die Anfangstemperatur und die Abkühlungsfunktion richtig einzustellen, sowie die potenziell lange Laufzeit, insbesondere für komplexe Probleme.

Im Kontext des TSP kann Simulated Annealing als leistungsstarke Heuristik betrachtet werden, die in der Lage ist, qualitativ hochwertige Lösungen zu finden, insbesondere wenn andere Methoden wie exakte Algorithmen zu rechenintensiv sind. Es kann helfen, lokale Minima zu umgehen und die Suche im Lösungsraum zu diversifizieren, um möglicherweise bessere Routen zu entdecken. (Donald Davendra, 2010; Ingrid Gerdes, 2004; Berthold Vöcking, 2008)

Aus zeitlichen Gründen konnten sowohl Simmulated Annealing als auch k-opt nicht ausführlicher behandelt werden, bei Interesse finden sich aber zu beiden Themen Programmcode-Implementierungen (Python) im Anhang und auf Github.

4.2 Weitere Algorithmen und Heuristiken

In nachfolgenden Kapiteln werden einige nennenswerte exakte Algorithmen und Heuristiken aufgeführt. Da es sich hier lediglich um einen Zusatz zu den bereits ausführlich(er) behandelten Lösungsansätzen handelt, werden die nachfolgenden Ansätze nur äußerst grob beschrieben. Sie werden weder mathematisch noch durch Grafiken gestützt, können bei bestehendem Interesse jedoch durch Eigenrecherche weiter vertieft werden, da es sich hier um durchaus bekannte Verfahren handelt.

4.2.1 Branch and Bound / Cut

Branch and Bound ist ein exakter Algorithmus zur Lösung des TSP. Er durchsucht den gegebenen Suchraum systematisch indem er ihn in Teilräume unterteilt und diese nach Teil-Lösungen durchsucht. Der Cut-Algorithmus optimiert Branch and Bound, indem er unnötige Teilräume abschneidet. Branch and Bound / Cut garantiert die Optimallösung und ist effizient für kleine Probleminstanzen des TSP. Ähnlich dem Naive-Brute-Force-Algorithmus ist er jedoch ungeeignet für große Probleminstanzen. Der Hauptunterschied zwischen Branch and Bound und einem naiven Brute-Force-Ansatz liegt in der Effizienz und im Suchraummanagement.
(Gregory Gutin; Abraham P. Punnen, 2007; Egon Balas; Paolo Toth, 1983)

4.2.2 Dynamic Programming

Dynamische Programmierung ist eine generelle Methode zur Lösung von Optimierungsproblemen, einschließlich des TSP. Es optimiert Teilprobleme und speichert deren Lösungen, um daraus eine Gesamtlösung zu bilden. Der Ansatz garantiert die Optimallösung und ist effizient für kleine bis mittelgroße Probleminstanzen des TSP. Auch der Ansatz der Dynamischen Programmierung ist für große Probleminstanzen ungeeignet.
(Berthold Vöcking, 2008)

4.2.3 Held-Karp-Algorithmus

Der Held-Karp-Algorithmus ist eine Variante der Dynamischen Programmierung, welcher die Optimallösung des TSP liefert. Der Hauptunterschied zur Dynamischen Programmierung liegt in der Art und Weise, wie Teilprobleme identifiziert und gelöst werden. Während bei "normaler" Dynamischer Programmierung Teilprobleme unabhängig voneinander gelöst werden, werden beim Held-Karp-Algorithmus Teilprobleme (Teilrouten) schrittweise aufgebaut und in größere Routen integriert, wobei bereits berechnete Teilrouten wiederverwendet werden. Dadurch wird die Anzahl der erforderlichen Berechnungen erheblich reduziert und die Gesamtlösung des TSP effizienter gefunden.
(Wikipedia, 2024f)

4.2.4 Nearest-Neighbour-Algorithmus

Der Nearest-Neighbour-Algorithmus ist eine einfache, aber effektive Methode zur Lösung des Travelling Salesman Problem. Er beginnt an einem zufälligen Punkt und wählt dann immer den nächstgelegenen noch nicht besuchten Punkt als nächsten Schritt. Dies wird wiederholt, bis alle Punkte besucht wurden und schließlich wird zum Ausgangspunkt zurückgekehrt. Obwohl dieser Algorithmus nicht immer die kürzeste Route liefert, ist er doch eine gute Näherungslösung, insbesondere wenn die Distanzen zwischen den Punkten ähnlich sind. Es ist jedoch zu beachten, dass die Qualität der Lösung stark von der Wahl des Startpunkts abhängen kann.
(Wikipedia, 2024h)

4.2.5 Random Swapping

Der Random Swapping Algorithmus ist eine einfache heuristische Improvement Methode, bei der zufällige Vertauschungen von Städten in einer gelösten, nicht optimalen Tour durchgeführt werden. Sollte durch den Tausch eine effizientere Lösung gefunden werden,

wird diese Verwendet. Random Swapping ist schnell und einfach zu implementieren, jedoch sind seine Lösungen oft suboptimal und kann durch die Zufallskomponente keine Verbesserung garantieren.
Random Swapping findet sich unter anderem als Teilschritt in der k-opt Methode wieder. (Donald Davendra, 2010; Sophia Heimann; Hung P. Hoang; Stefan Hougardy, 2024; Wikipedia, 2024a)

4.2.6 Lin-Kerninghan-Algorithmus

Der Lin-Kernighan-Algorithmus ist eine heuristische Improvement Methode zur Lösung des TSP. Er basiert auf Local Search und verbessert die Tour schrittweise. Lin-Kernighan verwendet verschiedene Techniken wie Kantenverschiebung und -tausch, um eine bessere Lösung zu finden. Der Algorithmus findet im Durchschnitt "gute" bis optimumsnahme Lösungen, liefert dennoch keine Garantie für die Optimallösung und kann in einigen Fällen in einem lokalem Optimum festhängen.
(Donald Davendra, 2010)

4.2.7 Ant Colony Simulation

Die Ameisenkolonie-Simulation ist eine metaheuristische Methode. Wie der Name andeutet ist die Methode durch das Verhalten von Ameisenkolonien inspiriert. Ameisen folgen einem stochastischen Prozess. Sie verwenden Pheromonspuren, um Pfade zu markieren und zu aktualisieren. Diese Methode ist robust und kann "gute" Lösungen finden, erfordert jedoch auch eine lange Laufzeit.
(Donald Davendra, 2010; Ingrid Gerdes, 2004)

5 Schlusswort

Im Nachfolgendem werden die wichtigsten Punkte der Ausarbeitung zum Traveling Salesman Problem nocheinmal aufgegriffen und final zusammengetragen.

Das 'Traveling Salesman Problem' ist ein klassisches Optimierungsproblem der Graphentheorie und ist der Komplexitätsklasse NP-Complete zugeordnet. Das bedeutet das es bei großen Probleminstanzen des TSPs nicht möglich ist eine Optimale Lösung mit tragbarer Laufzeit zu finden.

Gegeben ist ein kantenbewerteter, zusammenhängender Graph mit n Knoten. Gesucht wird ein Hamiltonkreis, also eine Rundreise (auch als Tour bezeichnet), die von einem Startknoten aus jeden Knoten genau einmal besucht. Der resultierende Zyklus ist geschlossen, da der Start- und Endknoten identisch sind.

Unter allen geschlossenen Rundreisen, auf denen alle Knoten des Graphen liegen, wird ein solcher mit minimaler Länge gesucht.

Das klassische TSP definiert sich über die Anzahl der Reisenden, genau ein Reisender und die Rundreise selbst, welche sich als Hamiltonkreis minimaler länge darstellen lässt. Neben dem klassischem TSP gibt es viele Variationen des TSPs. Daher findet man die Grundstruktur, Variation des TSP oder das TSP als Teilproblem in vielen Optimierungsproblemen wieder.

Um eine “optimale” Tour zu bestimmmen gibt es verschiedene Lösungsalgorithmen für das TSP, darunter exakte und approximative Methoden. Exakte Algorithmen wie der Naive-Brute-Force-Algorithm können bei kleinen Probleminstanzen eingesetzt werden, um optimale Lösungen zu finden, während sich approximative Algorithmen wie Nearest-Neighbor und Greedy oder der Christofides-Algorithmus für größere Probleminstanzen anbieten, um diese in akzeptabler Zeit “gut genug” lösen zu können.

Da die meisten praktischen Anwendungen mit großen Probleminstanzen des TSP arbeiten, ist es Standard approximative Algorithmen zu nutzen. Um diese Algorithmen zu verbessern werden Improvement Heuristiken eingesetzt, welche eine bereits existierende Tour verbessern sollen. Beispielhaft für Improvement-Heuristiken sind die Verfahren, Random Swapping, k-opt und Simulated Annealing.

Heuristiken bieten oft gute Lösungen, haben jedoch auch ihre eigenen Vor- und Nachteile in Bezug auf Geschwindigkeit und Genauigkeit. Insgesamt bietet das Studium des TSPs ein tiefes Verständnis für die Komplexität von Optimierungsproblemen und die Vielfalt der Algorithmen und Heuristiken, die entwickelt wurden, um diese anzugehen. Es ist ein faszinierendes Forschungsgebiet, das weiterhin sowohl theoretisch als auch praktisch relevant ist.

„[Das Traveling Salesman Problem] dient als eine Art Spielwiese zur Entwicklung von Optimierungsverfahren [im Bereich der Graphentheorie.]“ (Wikipedia, 2024m)

6 Literaturverzeichnis

Literatur

BERTHOLD VÖCKING, 2008. Taschenbuch der Algorithmen. In: Springer Verlag, S. 323–331, 345–361, 413–433. ‘eXamen.press’. ISBN 978-3-540-76393-2.

CHIENG, Hock Hung, 2016. A genetic simplified swarm algorithm for optimizing n-cities open loop travelling salesman problem. In: Auch verfügbar unter: https://api.semanticscholar.org/CorpusID:125530920.

DAVID L. APPLEGATE, 2006. *The Traveling Salesman Problem* [online]. Leseprobe. Princton University Press [besucht am 2024-05-04]. ISBN 0-691-12993-2. Abger. unter: https://books.google.de/books?hl=de&lr=&id=vhsJbqomDuIC&oi=fnd&pg=PP11&dq=traveling+salesman+problem&ots=YLAHTCLVe2&sig=d9fKluL_WE5Y7Q5-VG0qVyDKId0#v=onepage&q&f=false.

DONALD DAVENDRA, 2010. *Traveling Salesman Problem: Theory and Applications* [online]. Leseprobe. InTech [besucht am 2024-05-04]. ISBN 978-953-307-426-9. Abger. unter: https://books.google.de/books?hl=de&lr=&id=gKWdDwAAQBAJ&oi=fnd&pg=PA1&dq=traveling+salesman+problem&ots=aa9w-84fL3&sig=iAAfePysOVCBaSn_JKqCcHf5OyI#v=onepage&q&f=false.

EGON BALAS; PAOLO TOTH, 1983. BRANCH AND BOUND METHODS FOR THE TRAVELING SALESMAN PROBLEM. In: [Online] [besucht am 2024-05-04]. Abger. unter: https://apps.dtic.mil/sti/tr/pdf/ADA126957.pdf.

GREGORY GUTIN; ABRAHAM P. PUNNEN, 2007. *The Traveling Salesman Problem and Its Variations* [online]. Leseprobe. Springer Verlag [besucht am 2024-05-04]. ISBN 1-4020-0664-0. Abger. unter: https://books.google.de/books?hl=de&lr=&id=JBK_BAAAQBAJ&oi=fnd&pg=PR3&dq=traveling+salesman+problem&ots=kZqhIvSX5G&sig=SdMKBmdukwqG8qm7shVufDzvCKk#v=onepage&q&f=false.

HERIBERT RAU, 2010. *Freuden Und Leiden Eines Commis Voyageur* [online]. German Edition. Kessinger Publishing [besucht am 2024-05-04]. ISBN 1167696018. Abger. unter: https://www.amazon.com/Freuden-Leiden-Commis-Voyageur-German/dp/1167696018.

INGRID GERDES, 2004. Evolutionäre Algorithmen. In: Vieweg Verlag, Bd. 1, S. 5–12, 23–31, 227–230. ISBN 3-528-05570-7.

MATAI, Rajesh; SINGH, Surya; MITTAL, M.L., 2010. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. In: ISBN 978-953-307-426-9.

OPTESSA, [o. D.]. The Traveling Salesman Problem in Manufacturing. *Optessa.com* [online] [besucht am 2024-06-08]. Abger. unter: https://www.optessa.com/the-traveling-salesman-problem/.

PROF. DR. GUIDO WALZ, 2017. Travelling Salesman Problem. *Spektrum.de* [online] [besucht am 2024-05-04]. Abger. unter: https://www.spektrum.de/lexikon/mathematik/travelling-salesman-problem/10428#:~:text=auch%20TSP%2C%20Problem%20des%20Handelsreisenden,Ende%20wieder%20zum%20Ausgangsort%20zur%C3%BCckkehren.

PROF. DR. S. ALBERS, [o. D.]. Algorithmentheorie 10 - Greedy Verfahren. *ac.informatik.uni-freiburg.de* [online] [besucht am 2024-06-08]. Abger. unter: https://ac.informatik.uni-freiburg.de/lak_teaching/ws05_06/algotheo/Misc/Slides/10_Greedy_Verfahren.pdf.

SOPHIA HEIMANN; HUNG P. HOANG; STEFAN HOUGARDY, 2024. The k-Opt algorithm for the Traveling Salesman Problem has exponential running time for k >= 5. Auch verfügbar unter: https://www.or.uni-bonn.de/~hougardy/paper/HeimannHoangHougardy2024_arXiv_2402.07061.pdf.

VOLKER TURAU, 1996. Algorithmische Graphentheorie. In: Addison-Wesley Publishing, S. 69–77, 281–318. ISBN 3-89319-938-1.

WIKIMEDIA, 2022. File:Icosian grid small with labels2.svg. *Wikimedia* [online] [besucht am 2024-05-04]. Abger. unter: https://commons.wikimedia.org/wiki/File:Icosian_grid_small_with_labels2.svg.

WIKIPEDIA, 2021. Vollständiger Graph. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/Vollst%C3%A4ndiger_Graph.

WIKIPEDIA, 2022. P (Komplexitätsklasse). *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/P_(Komplexit%C3%A4tsklasse).

WIKIPEDIA, 2023a. Icosian Game. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/Icosian_Game.

WIKIPEDIA, 2023b. NP-Äquivalenz. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/NP-%C3%84quivalenz.

WIKIPEDIA, 2023c. NP-equivalent. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://en.wikipedia.org/wiki/NP-equivalent.

WIKIPEDIA, 2023d. NP-Schwere. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/NP-Schwere.

WIKIPEDIA, 2023e. NP-Vollständigkeit. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/NP-Vollst%C3%A4ndigkeit.

WIKIPEDIA, 2024a. 2-opt. *Wikipedia* [online] [besucht am 2024-06-08]. Abger. unter: https://en.wikipedia.org/wiki/2-opt.

WIKIPEDIA, 2024b. Christofides algorithm. *Wikipedia* [online] [besucht am 2024-06-08]. Abger. unter: https://en.wikipedia.org/wiki/Christofides_algorithm.

WIKIPEDIA, 2024c. Greedy algorithm. *Wikipedia* [online] [besucht am 2024-06-08]. Abger. unter: https://en.wikipedia.org/wiki/Greedy_algorithm.

WIKIPEDIA, 2024d. Hamiltonian path problem. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://en.wikipedia.org/wiki/Hamiltonian_path_problem.

WIKIPEDIA, 2024e. Hamiltonkreisproblem. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/Hamiltonkreisproblem.

WIKIPEDIA, 2024f. Held–Karp algorithm. *Wikipedia* [online] [besucht am 2024-06-08]. Abger. unter: https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm.

WIKIPEDIA, 2024g. Icosian game. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://en.wikipedia.org/wiki/Icosian_game.

WIKIPEDIA, 2024h. Nearest neighbour algorithm. *Wikipedia* [online] [besucht am 2024-06-08]. Abger. unter: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm.

WIKIPEDIA, 2024i. NP (Komplexitätsklasse). *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/NP_(Komplexit%C3%A4tsklasse).

WIKIPEDIA, 2024j. NP-completeness. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://en.wikipedia.org/wiki/NP-completeness.

WIKIPEDIA, 2024k. NP-hardness. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://en.wikipedia.org/wiki/NP-hardness.

WIKIPEDIA, 2024l. P (complexity). *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://en.wikipedia.org/wiki/P_(complexity).

WIKIPEDIA, 2024m. Problem des Handlungsreisenden. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden#Geschichte.

WIKIPEDIA, 2024n. Travelling salesman problem. *Wikipedia* [online] [besucht am 2024-05-04]. Abger. unter: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

YVAN DUMAS; JACQUES DESROSIERS, 1995. An Optimal Algorithm for the Traveling Salesman Problem with Time Windows. In: [Online], Bd. 43, S. 367–371 [besucht am 2024-05-04]. Nr. 2. Abger. unter: https://pubsonline.informs.org/doi/epdf/10.1287/opre.43.2.367.

7 Anhang

7.1 Legende: Mathematisch Ausdrücke

Bezeichnung	Bedeutung
G	Graph
n	Anzahl der Knoten im Graph
m	Anzahl der Reisenden
$\|V\|$	Menge der Knoten im Graph
v	Ein beliebiger Knoten im Graph
v_i	Ein Knoten im Graph
e	Eine beliebige Kante im Graph
$e(i, j)$	Eine Kante die zwei Knoten verbindet
e_{ij}	Wert der Kante $e(i, j)$
$\dots$	$\dots$

Tabelle 5: Legende: Mathematisch Ausdrücke – (Selbsterstellt)

7.2 Programmcode

https://github.com/Cuppixx/TravelingSalesmanProblem_ClassProject_PythonImplementation
https://github.com/Cuppixx/TravelingSalesmanProblem_ClassProject

8 Änderungshistorie

- Rechtschreibfehler und Grammatik verbessert.
- Kapitel 'Warum nicht Brute-Force?' sprachlich angepasst damit der Begriff große Instanzen verständlicher ist.
- Der Begriff Instanzen wurde an passenden stellen zu Probleminstanzen geändert.
- In der Liste der Vorteile des Greedy-Algorithmus wurde die Aussage zur Laufzeitkomplexität von linear auf polynomielle geändert.

9 Erklärung

Ich erkläre, dass ich alle Quellen, die ich zur Erstellung dieser Arbeit verwendet habe, im Quellenverzeichnis aufgeführt habe und dass ich alle Stellen, an denen Informationen (Texte, Bilder, Tabelle, Quellcode) aus diesen Quellen in meine Arbeit eingeflossen sind, als Zitate mit Quellenangabe kenntlich gemacht habe. Mir ist bewusst, dass ein Verstoß gegen diese Regeln zum Ausschluss aus dem Seminar führt.

David Borgert