FernUniversität in Hagen

Fakultät für Mathematik und Informatik

Masterarbeit

Analyse heuristischer Algorithmen zur Lösung des Orientierungsproblems

Erstellt von: Mahmoud Machaka
Studiengang: M.Sc. Praktische Informatik
E-Mail: mahmoud.machaka@studium.fernuni-hagen.de

Erstprüfer: Dr. Sebastian Küpper
Zweitprüferin: Dr. Daniela Keller
eingereicht am: 28.02.2022

Abstract

Beim Orientierungsproblem handelt es sich um ein NP-schweres Optimierungsproblem, für welches bislang kein deterministischer Algorithmus gefunden werden konnte, der eine exakte Lösung in Polynomialzeit lösen kann. Das Orientierungsproblem beschäftigt sich mit der Suche einer Tour innerhalb eines gewichteten Graphen, welche den durch den Besuch von Knoten erzielten Gesamtprofit maximiert, ohne dabei eine definierte Kostenobergrenze zu überschreiten. In dieser Abschlussarbeit wird die Komplexität des Orientierungsproblems untersucht und die Anwendung heuristischer Algorithmen zur Lösung des Problems motiviert. Weiterhin werden die heuristischen Algorithmen von Santini (2019) und Kobeaga et al. (2018) zur Lösung des Orientierungsproblems vorgestellt und analysiert. Bei der Analyse der Algorithmen werden Unterschiede in der Performanz beider Algorithmen festgestellt, die einerseits auf ihre Konfiguration und die Größe der zugrundeliegenden Graphen zurückgeführt werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

1 Einleitung

Die Tourenplanung ist ein intensiv untersuchtes Forschungsgebiet, welches sich mit der Entwicklung von Verfahren zur Tourenoptimierung beschäftigt. Die Thematik gewinnt insbesondere durch den wachsenden E-Commerce-Bereich und den damit verbundenen logistischen Herausforderungen an Relevanz. Ziel ist es hier, Waren oder Güter in möglichst kurzer Zeit oder besonders ressourcensparend auszuliefern, indem entsprechende Routen zu den Kunden kalkuliert werden. Neben den Positionen der Kunden können hier auch Faktoren wie beispielsweise die Anzahl und Kapazitäten der Lieferfahrzeuge oder Geschäftszeiten, in denen Kunden beliefert werden dürfen, eine Rolle spielen. Laut dem Bundesverband für E-Commerce und Versandhandel ist der Bereich des Onlinehandels Mitte 2021 um 23,2% gegenüber dem Vorjahr gewachsen, was nicht zuletzt auf die Corona-Pandemie zurückzuführen ist (vgl. BEVH 2021). Um den damit verbundenen logistischen Herausforderungen zu begegnen, werden Tourenplanungsverfahren eingesetzt, mit denen Touren berechnet werden können, die für die jeweiligen Anwendungsfälle optimiert sind. Neben der Logistik gibt es auch andere Anwendungsgebiete, wie z. B. die Reiseplanung in der Tourismus-Branche oder die Entwicklung effizienter Navigationssysteme, welche die Forschungen in diesem Gebiet weiter antreiben. Aus dem Tourenplanungsproblem haben sich seit seiner Einführung durch Dantzig/Ramser (1959) diverse problemabhängige Spezialisierungen entwickelt, zu denen auch das in dieser Abschlussarbeit betrachtete Orientierungsproblem zählt.

1.1 Problemstellung

Als Spezialisierung der Tourenplanung handelt es sich beim Orientierungsproblem um ein NP-schweres Optimierungsproblem, bei dem den Kunden jeweils ein Profitwert zugeordnet wird. Diese Profite werden eingesammelt, indem die entsprechenden Kunden besucht bzw. in eine Tour aufgenommen werden. Beim Orientierungsproblem wird eine Tour gesucht, bei der der erzielte Gesamtprofit maximiert wird, ohne dabei eine zuvor definierte Kostenobergrenze (z. B. Fahrzeit oder Wegstrecke) zu überschreiten. Die Tour muss zudem an definierten Punkten starten und enden. Im Kontext der Logistik kann es sich hierbei beispielsweise um Lagerhäuser handeln, in welchen die Lieferfahrzeuge beladen werden und nach der Auslieferung der Waren zurückkehren müssen. Da die Kosten der Tour einen gewissen Schwellenwert nicht überschreiten dürfen, müssen nicht alle Kunden zwangsweise besucht werden (vgl. Vansteenwegen/Gunawan 2019: 1-3).

Das Orientierungsproblem setzt sich demnach aus dem sogenannten Rucksackproblem und dem Problem des Handlungsreisenden zusammen, da die zu besuchenden Kunden ausgewählt und eine Reihenfolge bestimmt werden muss, in welcher diese Kunden besucht werden sollen. Aufgrund seiner Komplexität wurde für das Orientierungsproblem bisher kein deterministischer Algorithmus gefunden, mit dem die exakte bzw. beste Lösung in Polynomialzeit gefunden werden kann. Stattdessen werden heuristische Algorithmen herangezogen, welche diese Lösung in praktikabler Zeit approximieren können. Diese Algorithmen können sich unter anderem in ihrer Laufzeit und Konsistenz sowie in der Qualität der von ihnen erzeugten Lösungen unterscheiden (vgl. Golden et al. 1987: 307 f.).

1.2 Ziel der Arbeit

In dieser Abschlussarbeit wird das Orientierungsproblem hinsichtlich seiner Komplexität untersucht, um die Relevanz heuristischer Algorithmen herauszuarbeiten. Weiterhin werden der „Adaptive Large Neighbourhood Search“ Algorithmus von Santini (2019) und der evolutionäre

Algorithmus von Kobeaga et al. (2018) vorgestellt, implementiert und analysiert. Zur Einordnung beider heuristischer Algorithmen bezüglich ihrer Performanz werden außerdem weitere Algorithmen aus der Literatur herangezogen. Bei der Analyse soll unter anderem untersucht werden, inwiefern sich die Parametereinstellungen der Algorithmen und die Form der Graphen, auf welche die Algorithmen angewendet werden, auf die Qualität der erzeugten Lösungen auswirken.

1.3 Aufbau der Arbeit

Nach der Einleitung werden in dieser Abschlussarbeit zunächst einige Grundlagen der Graphentheorie und Tourenplanung sowie ein kleiner Auszug aus der Komplexitätstheorie vorgestellt, die für ein besseres Verständnis der Abschlussarbeit notwendig sind. Anschließend wird in Kapitel 3 das Orientierungsproblem präsentiert und formal definiert. Außerdem werden hier Variationen bzw. weitere Ausprägungen des Problems benannt und mit diversen Anwendungsfällen verknüpft. Das Kapitel schließt mit einer Betrachtung der Komplexität des Problems, aus welchem die Relevanz für heuristische Algorithmen abgeleitet wird. In Kapitel 4 werden der „Adaptive Large Neighbourhood Search“ Algorithmus von Santini (2019) und der evolutionäre Algorithmus von Kobeaga et al. (2018) vorgestellt. Im darauffolgenden Kapitel werden wesentliche Aspekte der Softwareentwicklung bzw. das Vorgehen bei der Implementierung beider Algorithmen in dieser Abschlussarbeit erläutert. Die Implementierung der Algorithmen dient der Durchführung von Benchmarking-Tests, mit welchen in Kapitel 6 beide Algorithmen hinsichtlich ihrer Performanz analysiert werden. Die Abschlussarbeit schließt in Kapitel 7 mit einem Ausblick und einer Zusammenfassung der Erkenntnisse.

2 Grundlagen

In diesem Kapitel werden grundlegende Begriffe und Konzepte der Graphentheorie und Tourenplanung erläutert, die dem besseren Verständnis des Orientierungsproblems dienen. Um das Orientierungsproblem hinsichtlich seiner Komplexität einordnen zu können, werden in Abschnitt 2.2 außerdem einige Grundlagen der Komplexitätstheorie vorgestellt.

2.1 Graphentheorie und Tourenplanung

Mithilfe graphentheoretischer Konzepte können verschiedene praktische Probleme abstrahiert und modelliert werden, um die Analyse und Lösung dieser Probleme zu vereinfachen. Die Graphentheorie lässt sich auf das sogenannte „Königsberger Brückenproblem“ und den Überlegungen des Mathematikers Leonhard Euler im Jahr 1736 zurückführen. Die Stadt Königsberg, welche seit ihrem Wiederaufbau 1946 Kaliningrad heißt, wird durch den Fluss Pregel durchlaufen und historisch in vier Gebiete unterteilt, welche seiner Zeit über sieben Brücken miteinander verbunden waren. Das Brückenproblem beschäftigte sich mit der Frage, ob eine zusammenhängende Tour existiert, in welcher jede dieser sieben Brücken genau einmal überquert wird. Leonhard Euler abstrahierte das Problem, indem er jedes Ufer als Knoten und jede Brücke als eine Kante darstellte, wie in Abbildung 1 dargestellt wird. Informationen, wie beispielsweise die tatsächliche Form der Ufer oder die Abstände der Brücken, sind in der Abstraktion nicht enthalten, da sie keinen Beitrag zur Lösung des Problems leisten. Aufbauend auf dieser graphentheoretischen Darstellung zeigte Euler, dass es für Königsberg keine entsprechende Tour geben kann. In den folgenden Abschnitten werden die für diese Abschlussarbeit notwendigen graphentheoretischen Begriffe definiert und elementare Konzepte erklärt (vgl. Krumke/Noltemeier 2012: 3–4).

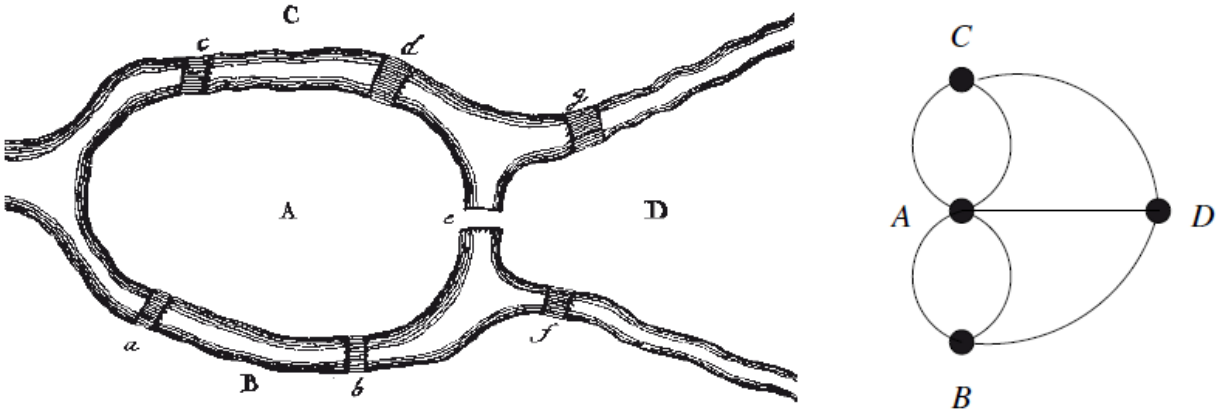


Abbildung 1: Grundriss der Stadt Königsberg und der abgeleitete Graph (Krumke/Noltemeier 2012: 4)

2.1.1 Gerichtete und ungerichtete Graphen

Ein Graph $G = (V, E)$ besteht aus einer nicht-leeren Knotenmenge V (*engl. vertices*) und einer Kantenmenge E (*engl. edges*). Bezüglich der Notation werden Knoten in dieser Arbeit als $v_i \in V$ dargestellt, wobei i ein beliebiger Name oder ein Index sein kann. Eine Kante $e \in E$ beschreibt die Beziehung zweier Knoten $v_i, v_j \in V$ und wird in dieser Arbeit als (v_i, v_j) dargestellt.

Definition 1: Definition eines Graphen (vgl. Briskorn 2019: 17-19)

Bei gerichteten Graphen bezeichnet der Knoten v_i der Kante (v_i, v_j) den Anfangsknoten und v_j entsprechend den Endknoten der Kante. In diesem Fall kann die Kante nur von v_i nach v_j traversiert werden. Bei ungerichteten Graphen sind beide Knoten einer Kante hingegen austauschbar, sodass eine Kante in beide Richtungen durchlaufen werden kann. Ein ungerichteter Graph kann immer in einen gerichteten Graphen transformiert werden, indem für jede Kante (v_i, v_j) eine entsprechende Kante (v_j, v_i) eingefügt wird. Beim Orientierungsproblem wird initial ein vollständiger gerichteter Graph angenommen. Das bedeutet, dass zwischen jedem Knotenpaar v_i, v_j mit $i \neq j$ und $i, j \in \{0, \dots, |V| - 1\}$ eine Kante existiert (vgl. Briskorn 2019: 17-19).

2.1.2 Gewichtete Knoten und Kanten

Die Knoten und Kanten eines Graphen G können mithilfe von Gewichtsfunktionen c und s mit numerischen Werten versehen werden. Die Funktion $c = (c_{i,j})_{i,j=1}^{n}$ beschreibt hierbei die Gewichte aller Kanten, während die Funktion $s = (s_i)_{i=1}^{n}$ die Gewichte aller Knoten des Graphen darstellt. Die Definition des Graphen G wird um diese Funktionen erweitert und entsprechend als Quadrupel $G = (V, E, c, s)$ dargestellt.

Definition 2: Gewichtete Graphen (vgl. Briskorn 2019: 22-23)

Die Verwendung von Knoten- und Kantengewichten kann der Modellierung weiterer Anwendungsfälle dienen. Wenn beispielsweise Kunden eines Lieferdienstes als Knoten in einem Graphen dargestellt werden, so kann jeder Knoten mit einem Wert versehen werden, der die erwarteten Einnahmen anzeigt. Im selben Anwendungsfall können die Kanten zwischen den Knoten als Wegstrecken interpretiert werden. Die Kantengewichte können hier beispielsweise die Länge der Strecke, Dauer der Fahrt oder den Treibstoffverbrauch beschreiben (vgl. Briskorn 2019: 22-23).

Beim Orientierungsproblem werden sowohl Knoten als auch Kanten mit entsprechenden Gewichten versehen. Ein Knotengewicht stellt den *Profit* dar, der durch den Besuch des Knotens eingenommen werden kann. Die Kantengewichte beschreiben jeweils die *Reisekosten* einer Kante bzw. die Kosten zwischen einem Knotenpaar (v_i, v_j). In dieser Arbeit werden als Reisekosten die euklidischen Distanzen zwischen zwei Knoten in einem zweidimensionalen Raum verwendet. Für alle Knoten $u, v, w \in V$ gilt außerdem hinsichtlich ihrer Distanzen die Dreiecksungleichung $c_{u,w} \leq c_{u,v} + c_{v,w}$. Das bedeutet, dass der Weg vom Knoten u nach w über den Knoten v nicht kürzer als der direkte Weg vom Knoten u zum Knoten w sein darf. Vollständige Graphen mit gewichteten Kanten, welche diese Bedingung erfüllen, werden auch metrische Graphen genannt (vgl. Krumke/Noltemeier 2012: 135).

2.1.3 Besondere Strukturen

In einem gerichteten Graphen können Wege von einem Knoten v_i zu einem Knoten v_j durch eine Kantenfolge mit $n,\ n \geq 1$, Kanten beschrieben werden. Die erste Kante eines Weges von v_i zu einem Knoten v_j muss dabei den Knoten v_i als Startknoten besitzen, während die letzte Kante mit dem Knoten v_j enden muss. Weiterhin dürfen nur Kanten in der Kantenfolge verwendet werden, die auch in der Kantenmenge E des Graphen enthalten sind. Schließlich muss – ausgenommen von der ersten Kante – jede Kante mit dem Endknoten der

vorangegangenen Kante beginnen. Als Notation einer Kantenfolge werden in dieser Arbeit die Anfangsknoten jeder Kante und der Endknoten des Weges aufgeführt, sodass sich für die Kanten $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ die Kantenfolge $(v_1, v_2, v_3, \dots, v_n)$ ergibt.

Definition 3: Wege im Graphen (vgl. Briskorn 2019: 23)

Innerhalb eines Weges dürfen Knoten mehrmals enthalten sein. Für das Orientierungsproblem ist diese Eigenschaft jedoch nicht erwünscht, da der Profit eines Knotens nur einmal eingesammelt bzw. ein Kunde nur ein einziges Mal besucht werden darf. Die einzigen Ausnahmen bilden hier die Start- und Endknoten v_i und v_j des Weges, die in diesem Kontext gleich oder verschieden sein dürfen.

Wege, bei denen die Start- und Endknoten voneinander verschieden sind und bei denen innerhalb des Weges kein Knoten mehrmals besucht wird, werden *Pfade* genannt. Diese Wege werden hingegen als *Kreise* bezeichnet, wenn die Start- und Endknoten v_i und v_j identisch sind.

Definition 4: Pfade und Kreise im Graphen (vgl. Briskorn 2019: 24)

Das Orientierungsproblem akzeptiert als Lösungen sowohl Pfade als auch Kreise. Sogenannte hamiltonsche Pfade und Kreise stellen weitere besondere Strukturen dar, welche relevant für den Crossover-Operator des evolutionären Algorithmus (s. Abschnitt 4.3.5.2) sind. Bei diesen Wegen bzw. Kreisen müssen alle Knoten $v \in V$ des Graphen genau einmal besucht werden. Für hamiltonsche Kreise gilt auch hier die Ausnahme, dass der Weg beim Startknoten endet und damit der Startknoten zweimal besucht werden darf (vgl. Krumke/Noltemeier 2012: 50).

2.2 Komplexitätstheorie

Zur Einordnung des Orientierungsproblems hinsichtlich seiner Komplexität und zur Klassifizierung von Algorithmen in Bezug auf ihre Laufzeit wird in diesem Abschnitt ein kleiner Auszug aus der Komplexitätstheorie präsentiert. Tiefergehende Erläuterungen und Definitionen können der Literatur entnommen werden (vgl. Vossen/Witt (2016) oder Krumke/Noltemeier (2012)).

2.2.1 Die $\mathcal{O}$-Notation

Bei der Betrachtung eines Algorithmus oder einer Funktion $f(n)$ mit Hinblick auf ihre Laufzeit-Komplexität wird untersucht, wie sich die Laufzeit mit wachsender Eingabegröße verhält. Die Eingabegröße n hängt vom betrachteten Problem ab und kann beispielsweise die Anzahl zu sortierender Elemente einer Menge oder die Anzahl der Knoten eines betrachteten Graphen beschreiben. In der Regel genügt es, das Wachstum mit einer einfachen Schrankenfunktion nach oben hin abzuschätzen. Da potentiell unendlich große Eingabemengen ($n \to \infty$) betrachtet werden, ist es hierbei wichtiger zu unterscheiden, ob die Laufzeit z. B. linear (n) oder quadratisch (n^2) wächst, statt zwischen den quadratischen Laufzeiten $1{,}5n^2$ und $2n^2$ zu differenzieren. Diese obere Schrankenfunktion wird mit der sogenannten $\mathcal{O}$- Notation beschrieben, welche im Folgenden definiert wird:

Seien $f, g: \mathbb{N}_0 \to \mathbb{R}_+$ zwei Funktionen. Die Funktion f ist *von der Ordnung* g, falls es eine Konstante $c > 0$ und eine natürliche Zahl n_0 gibt, so dass gilt:

$$f(n) \leq c \cdot g(n) \ \text{für alle} \ n \geq n_0$$

Mit $\mathcal{O}(g)$ wird die Menge der Funktionen bezeichnet, die von der Ordnung g sind:

$$\mathcal{O}(g) = \{f: \mathbb{N}_0 \to \mathbb{R}_+ | \ es\ gibt\ c > 0\ und\ n_0 \in \mathbb{N}_0\ mit\ f(n) \leq c \cdot g(n) \ \text{für alle} \ n \geq n_0\}$$

Definition 5: Die $\mathcal{O}$-Notation (vgl. Vossen/Witt 2016: 383 f.)

Zur Klassifizierung des Wachstums bzw. zur Bestimmung der Ordnung einer Funktion wird mithilfe der $\mathcal{O}$- Notation $f \in \mathcal{O}(g)$ also ausgesagt, dass „alle Funktionswerte von f ab einer festen Stelle n_0 kleiner als die entsprechenden Funktionswerte von g (abgesehen von einer Konstanten c)“ sind (Vossen/Witt 2016: 384). So kann mithilfe der $\mathcal{O}$-Notation schnell eine Aussage über die obere Grenze der Laufzeit eines Algorithmus oder einer Funktion $f(n)$ für wachsende n getroffen werden (vgl. Vossen/Witt 2016: 383-386).

Anhand folgender Beispiele soll dieser Sachverhalt veranschaulicht werden. Für den Druck einer Spielfigur benötigt ein 3D-Drucker im Schnitt zwei Stunden. Zur Anfertigung einer weiteren Spielfigur werden weitere zwei Stunden benötigt. Werden Wartungsarbeiten oder der regelmäßige Wechsel des Drucker-Filaments vernachlässigt, so kann gut eingeschätzt werden, wie viel Zeit der Drucker zur Erzeugung von n Spielfiguren benötigt. Der Prozess hat in diesem Fall ein lineares Wachstum. Mit der $\mathcal{O}$-Notation wird dies durch $\mathcal{O}(n)$ ausgedrückt. Für eine quadratische Zeitkomplexität $\mathcal{O}(n^2)$ kann beispielsweise ein unabhängiger Unterhändler angenommen werden, der in bilateralen Gesprächen zwischen mehreren Parteien verhandeln muss. Der Unterhändler nimmt die Aussagen der ersten Partei entgegen und vermittelt diese jeweils in Einzelgesprächen an die anderen Parteien. Das Vorgehen muss auch für alle anderen Parteien wiederholt werden. Kommt nun eine weitere Partei an den Verhandlungstisch, so steigt der Aufwand für den Unterhändler quadratisch, da er die Aussage bzw. das Angebot der neuen Partei an alle anderen Parteien bilateral vermitteln und auch ihre Aussagen an die neue Partei weitergeben muss. Tabelle 1 gibt eine Übersicht über wichtige Wachstumsklassen und ihre Darstellungen in der $\mathcal{O}$-Notation.

Wachstum	Ordnung
konstant	$\mathcal{O}(1)$
logarithmisch	$\mathcal{O}(\log(n))$
linear	$\mathcal{O}(n)$
n-log-n	$\mathcal{O}(n \cdot \log(n))$
polynomiell	$\mathcal{O}(n^k), k \geq 2$
exponentiell	$\mathcal{O}(d^n), d > 1$
faktoriell	$\mathcal{O}(n!)$

Tabelle 1: Wichtige Wachstumsklassen (Vossen/Witt 2016: 389, modifiziert)

2.2.2 Die Klassen P und NP

Mit den Problemklassen P und NP können Entscheidungsprobleme klassifiziert werden. Für Entscheidungsprobleme der Klasse P gilt, dass sie von deterministischen Turingautomaten in polynomieller Zeit lösbar sind. Das bedeutet, dass ein deterministischer Algorithmus zur Lösung des Problems existiert, der hinsichtlich seiner Komplexität durch $\mathcal{O}(n^k), k \geq 2$ nach oben beschränkt wird. Der Klasse NP werden hingegen alle Entscheidungsprobleme zugeordnet, die von nicht-deterministischen Turingautomaten in polynomieller Zeit lösbar sind. Es lässt sich zeigen, dass Lösungen der Probleme der Klasse NP in polynomieller Zeit durch einen deterministischen Turingautomaten verifiziert werden können (vgl. Garey/Johnson 1979: 27–32).

Der Fokus auf die Polynomialzeit ist wichtig, da Algorithmen mit dieser oberen Wachstumsschranke $\mathcal{O}(n^k), k \geq 2$ entsprechende Probleme praktisch bzw. in zumutbarer Zeit lösen können, wenn k entsprechend klein ausfällt. Algorithmen mit stärkerem Wachstum (z. B. exponentielles Wachstum) kommen in der Regel wesentlich schneller an die Grenzen ihrer Praktikabilität für wachsende n (vgl. Vossen/Witt 2016: 385).

Da jedes Problem, welches in polynomieller Zeit von einem deterministischen Turingautomaten lösbar ist, auch von einem nicht-deterministischen Turingautomaten in polynomieller Zeit gelöst werden kann, gilt, dass die Klasse P eine Untermenge der Klasse NP ist, also P ⊆ NP. Der Umkehrschluss, dass auch NP ⊆ P und damit P = NP gilt, ist ein bis heute ungelöstes Millennium-Problem. Aus dem Fall P = NP ließe sich schließen, dass es für jedes Problem der Klasse NP auch einen deterministischen Algorithmus geben muss, der die Lösung in Polynomialzeit lösen kann. Diese Folgerung – so erfreulich sie auch klingen mag – hätte gravierende Folgen für einige Anwendungsbereiche, wie z. B. die der Kryptographie. Zur Sicherung der Integrität und Vertraulichkeit der Daten werden die Komplexitäten mathematischer Probleme ausgenutzt. So basiert das asymmetrische Kryptoverfahren „RSA“ (nach Rivest, Shamir und Adleman) auf dem Problem der Primfaktorzerlegung, für welches bis heute kein effizienter deterministischer Algorithmus gefunden wurde (vgl. Krumke/Noltemeier 2012: 23-27).

Mit den Komplexitätsklassen P und NP wurden in diesem Abschnitt bisher Entscheidungsprobleme klassifiziert. Für diese Probleme können Entscheidungsverfahren formuliert werden, die basierend auf einer Eingabe eine Entscheidung bzw. Aussage treffen können. Eine Lösung für jedes dieser Probleme kann in polynomieller Zeit deterministisch verifiziert werden. Als Beispiel sei an dieser Stelle das NP-vollständige Problem des Handlungsreisenden (engl. *travelling salesman problem (TSP)*) in seiner Entscheidungsvariante angeführt. Hier stellt sich die Frage, ob für einen gegebenen Graphen G eine Tour existiert, welche alle Knoten genau einmal besucht und maximal die Länge L besitzt. Zwar wurde bisher kein deterministischer Algorithmus zur Lösung des Problems in Polynomialzeit gefunden, jedoch kann eine präsentierte Lösung schnell überprüft werden. Es müsste nämlich lediglich geprüft werden, ob die Lösung bzw. die Tour alle Knoten des Graphen genau einmal enthält und kürzer als ein Wert L ist (vgl. Garey/Johnson 1979: 18-23).

Neben den Entscheidungsproblemen existieren unter anderem auch Optimierungsprobleme, zu denen das Orientierungsproblem zählt und die per Definition nicht den Klassen P oder NP zugeordnet werden können. Bei einem Optimierungsproblem wird unter der Verwendung einer Ziel- bzw. Fitnessfunktion die beste Lösung aus einer Lösungsmenge gesucht. Generell wird ein Problem X als NP-schweres Problem bezeichnet, wenn alle Probleme der Klasse NP auf X in Polynomialzeit reduziert werden können. Im Falle, dass ein deterministischer Algorithmus zur

Lösung dieses Problems in Polynomialzeit gefunden würde, ließen sich also auch alle anderen Probleme der Klasse NP mit einem deterministischen Turingautomaten in Polynomialzeit lösen, nachdem sie polynomiell auf X reduziert wurden (vgl. Vossen/Witt 2016: 394). Wenn es sich beim NP-schweren Problem X außerdem um ein Entscheidungsproblem handelt bzw. wenn es in der Menge NP liegt, so wird es als NP-vollständiges Problem bezeichnet (vgl. Garey/Johnson 1979: 34-38).

Beim Orientierungsproblem handelt es sich um ein NP-schweres Optimierungsproblem, für welches noch kein deterministischer Algorithmus zur Prüfung der Korrektheit einer Lösung in Polynomialzeit gefunden wurde (s. Abschnitt 3.5). Beim Orientierungsproblem wird nämlich eine Tour gesucht, die den erzielten Gesamtprofit maximiert und dabei eine Kostenobergrenze nicht überschreitet. Wird hier eine Lösung präsentiert, so müsste sie zur Verifikation mit allen anderen potentiellen Lösungen verglichen werden oder es müsste zunächst die exakte Lösung bestimmt und der präsentierten Lösung gegenübergestellt werden. Analog zum Verhältnis von NP zu NP-vollständig lassen sich NP-vollständige Probleme auf NP-schwere Probleme reduzieren (s. Abschnitt 3.5). NP-schwere Probleme sind also mindestens so komplex wie NP-vollständige Probleme. Abbildung 2 illustriert die Zusammenhänge der Komplexitätsmengen für die Fälle $P = NP$ und $P \neq NP$ (vgl. Garey/Johnson 1979: 32-38).

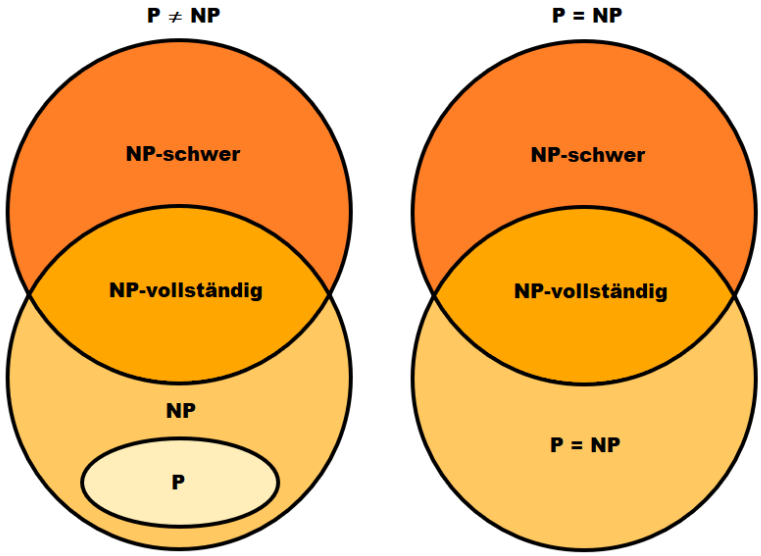


Abbildung 2: Übersicht der Komplexitätsklassen (Baeldung 2021b, modifiziert)

3 Orientierungsproblem

Neben der Definition des Orientierungsproblems und seiner Einordnung zu ähnlichen Optimierungsproblemen werden in diesem Kapitel auch einige Varianten des Problems und praktische Anwendungsfälle vorgestellt. Das Kapitel schließt mit der Betrachtung der Komplexität des Problems ab und beleuchtet dabei, warum in der Praxis heuristische Algorithmen zur Lösung des Problems verwendet werden.

3.1 Einführung

Das Orientierungsproblem, welches bereits von Tsiligirides (1984) und Golden et al. (1987) behandelt wurde und auch als Tourenplanungsproblem mit Profiten bezeichnet wird, ist auf den Orientierungslauf zurückzuführen. Bei dieser sportlichen Aktivität, welche oftmals in dichten Wäldern abgehalten wird, werden neben einem Start- und Zielpunkt diverse weitere Kontrollpunkte festgelegt. Letztere werden mit einem variierenden Punktewert versehen. Beim Orientierungslauf müssen sich die Teilnehmer in einer begrenzten Zeit vom Start- zum Zielpunkt begeben und dabei möglichst viele Punkte durch den Besuch der Kontrollpunkte einsammeln. Die Person, welche es rechtzeitig zum Zielpunkt schafft und dabei die meisten Punkte einsammeln konnte, gewinnt das Spiel (vgl. Golden et al. 1987: 307).

3.2 Definition

In diesem Abschnitt wird eine graphentheoretische Modellierung des Orientierungsproblems vorgenommen und das Problem formal beschrieben. Gegeben sei eine Knotenmenge $V = \{v_1, \dots, v_N\}$, in welcher jeder Knoten $v_i \in V$ mit einem nicht-negativen Profitwert $p_{v_i} \in \mathbb{R}^+$ versehen wird. Sowohl der Startknoten v_1 als auch der Endknoten v_N erhalten den Profitwert $p_{v_1} = p_{v_N} = 0$. Die Start- und Endknoten dürfen auch identisch sein bzw. von einem Knoten repräsentiert werden. Das Ziel des Orientierungsproblems ist die Berechnung einer Tour, die durch eine Untermenge der Knotenmenge V traversiert und unter Berücksichtigung einer Kostenobergrenze T_{max} den erzielten Gesamtprofit maximiert. Beim Besuch profitbehafteter Knoten werden ihre Profite auf diesen erzielten Gesamtprofit addiert, wobei ein Knoten nur einmal besucht werden darf. Die Traversierungskosten für jedes Knotenpaar (v_i, v_j) werden mit $t_{v_{ij}}$ beschrieben und werden zu den Kosten der Tour gezählt, wenn der Knoten v_j nach dem Knoten v_i besucht wird (vgl. Vansteenwegen/Gunawan 2019: 16).

Darauf aufbauend kann das Orientierungsproblem auch als ganzzahliges lineares Optimierungsproblem mit einer Entscheidungsvariablen $x_{v_{ij}}$ modelliert werden. Wird ein Knoten v_j nach einem Knoten v_i in einer Tour besucht, so ist $x_{v_{ij}} = 1$. Andernfalls wird der Wert $x_{v_{ij}}$ auf 0 gesetzt (vgl. Vansteenwegen/Gunawan 2019: 17).

Weiterhin werden von Vansteenwegen und Gunawan (2019: 17) folgende Rahmenbedingungen definiert:

1. Das Ziel des Orientierungsproblems wird durch folgende Zielfunktion beschrieben:

$$Maximiere \sum_{i=2}^{N-1} \sum_{j=2}^{N} p_{v_i} \cdot x_{v_{ij}} \qquad (3.2.1)$$

   Demnach wird der Profit des Knotens v_i genau dann berücksichtigt, wenn dieser Knoten eine ausgehende Kante besitzt bzw. wenn ein Knoten v_j existiert, sodass $x_{v_{ij}} = 1$ gilt.

Das Ziel des Orientierungsproblems ist es, den erzielten Gesamtprofit durch die Wahl zu besuchender Knoten zu maximieren. Zu beachten ist, dass gemäß der Zielfunktion der Profit eines Knotens mehrfach bzw. für jeden seiner ausgehenden Kanten berücksichtigt werden kann. Die Einschränkung, dass jeder Knoten nur eine ausgehende Kante besitzen darf, wird in der Rahmenbedingung 3.2.3 vorgenommen.

2. Die zu berechnende Tour muss im Startknoten beginnen und im Endkonten münden:

$$\sum_{j=2}^{N} x_{v_{1j}} = \sum_{i=1}^{N-1} x_{v_{iN}} = 1 \qquad (3.2.2)$$

Gemäß dieser Rahmenbedingung muss der Startknoten eine ausgehende und der Endknoten eine eingehende Kante besitzen. Diese Bedingung kann sowohl von Graphen erfüllt werden, deren Start- und Endknoten verschieden sind, als auch von Graphen mit identischen Start- und Endknoten.

3. Die zu berechnende Tour muss zusammenhängend sein und jeder Knoten darf höchstens einmal besucht werden:

$$\sum_{i=1}^{N-1} x_{v_{ik}} = \sum_{j=2}^{N} x_{v_{kj}} \leq 1; \quad \forall k = 2, \dots, (N-1) \qquad (3.2.3)$$

4. Die Gesamtkosten der Tour dürfen die Kostenobergrenze T_{max} nicht übersteigen:

$$\sum_{i=1}^{N-1} \sum_{j=2}^{N} t_{v_{ij}} \cdot x_{v_{ij}} \leq T_{max} \qquad (3.2.4)$$

5. Die Tour darf keine Subtouren enthalten, d.h. alle besuchten Knoten müssen durch eine zusammenhängende Tour besucht werden:

$$\sum_{v_i \in S} \sum_{v_j \in S} x_{v_{ij}} \leq |S| - 1; \quad \forall S \subset V \setminus \{v_1, v_N\} \qquad (3.2.5)$$

Die Bedingung 3.2.5 fordert demnach, dass die Anzahl der Kanten aller Knotenuntermengen S, welche die Start- und Endknoten nicht enthalten, jeweils kleiner als die Anzahl der in ihnen enthaltenen Knoten sein muss.

3.3 Einordnung des Problems

Das Orientierungsproblem setzt sich aus dem Rucksackproblem (engl. *knapsack problem*) und dem Problem des Handlungsreisenden (engl. *travelling salesman problem*) zusammen, die in diesem Abschnitt vorgestellt werden (vgl. Vansteenwegen/Gunawan 2019: 17 f.).

Das Rucksackproblem ist ein NP-schweres Problem der kombinatorischen Optimierung, welches erstmals von Mathews (1896) beschrieben und in den letzten Jahrzenten intensiv erforscht wurde. Beim klassischen Rucksackproblem wird ein Rucksack mit einer begrenzten Kapazität

bzw. Traglast angenommen. Aus einer Menge von Objekten, die sowohl mit einem Nutzwert als auch mit einem Gewichtswert versehen sind, sollen Objekte so ausgewählt und in den Rucksack eingepackt werden, dass der erzielte Nutzwert maximiert wird, ohne die Kapazität des Rucksacks zu überschreiten (vgl. Caccetta/Kulanoot 2001: 5547). Formal lässt sich das Problem als ganzzahliges lineares Optimierungsproblem wie folgt beschreiben:

$$x_i = \begin{cases} 1, wenn\ Objekt\ i\ gewählt\ wurde \\ 0, sonst \end{cases}, \qquad i \in \{1, \dots, n\} \tag{3.3.1}$$

$$Zielfunktion: \ \ Maximiere \sum_{i=1}^{n} p_i \cdot x_i \tag{3.3.2}$$

$$\sum_{i=1}^{n} w_i \cdot x_i \leq c \tag{3.3.3}$$

Dabei beschreibt die Variable p_i den Nutzwert eines Objekts und w_i sein Gewicht. Die Kapazität des Rucksacks wird durch die Variable c ausgedrückt (vgl. Caccetta/Kulanoot 2001: 5548).

Das Rucksackproblem kann beispielsweise in der Logistik beim Beladen von Transportfahrzeugen oder bei der Begrenzung von Lagervorräten angewendet werden. Neben der klassischen Variante des Problems gibt es auch Variationen, die sich in der Anzahl verfügbarer Rucksäcke unterscheiden und bei denen einzelne Objekte mehrfach hinzugefügt werden dürfen (vgl. Caccetta/Kulanoot 2001: 5547).

Beim Problem des Handlungsreisenden handelt es sich um eine Spezialisierung der Tourenplanung. Bei diesem NP-schweren kombinatorischen Optimierungsproblem werden n Reiseziele angenommen, die alle von einem Händler in einer Tour besucht werden sollen. Das Problem besteht darin, die kürzeste Tour zu finden, in der alle Reiseziele besucht werden und die wieder im Ausgangspunkt endet. Nach Dantzig et al. (1954) kann das Problem als ganzzahliges lineares Optimierungsproblem wie folgt beschrieben werden:

$$x_{ij} = \begin{cases} 1, wenn\ Knoten\ j\ nach\ Knoten\ i\ besucht\ wird \\ 0, sonst \end{cases}, \qquad i, j \in \{1, \dots, n\} \tag{3.3.4}$$

$$Zielfunktion: Minimiere \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} \cdot x_{ij} \tag{3.3.5}$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \qquad j \in \{1, \dots, n\} \tag{3.3.6}$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, \qquad i \in \{1, \dots, n\} \tag{3.3.7}$$

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1, \forall Q \subset \{1, \dots, \mathrm{n}\}, |\mathrm{Q}| \geq 2 \tag{3.3.8}$$

Die Variable x_{ij} stellt die Entscheidungsvariable dar und die Variable c_{ij} beschreibt die Kosten bzw. Distanz zwischen zwei Knoten. Gemäß den Bedingungen (3.3.6) und (3.3.7) muss jeder Knoten genau eine eingehende und genau eine ausgehende Kante besitzen. Weiterhin werden Subtouren durch die Bedingung (3.3.8) ausgeschlossen (vgl. Dantzig et al. 1954: 1).

Das Problem des Handlungsreisenden bildet die Grundlage für die diverse Forschungen an allgemeinen Methoden zur Lösung diskreter Optimierungsprobleme. Praktische Anwendungsfälle des Problems finden sich meist im Logistik- bzw. Transportbereich. Ein Anwendungsfall mit historischem Hintergrund ist die Planung effizienter Schulbus-Routen. Der Mathematiker Merrill M. Flood (1956) nahm diesen Anwendungsfall als Anlass, erste Forschungen am Problem des Handlungsreisenden vorzunehmen.

3.4 Praktische Anwendung und Ausprägungen des Orientierungsproblems

In diesem Abschnitt wird die Relevanz des Orientierungsproblems anhand praktischer Anwendungsszenarien betrachtet. Dabei werden auch einige Varianten des Orientierungsproblems benannt, um die Vielfältigkeit des Problems darzustellen.

3.4.1 Logistik

Das Orientierungsproblem kann zur Optimierung diverser logistischer Prozesse herangezogen werden. Generell ist das der Fall, wenn aus einer Menge zu bedienender Kunden eine Auswahl getroffen und eine entsprechende Route bestimmt werden soll. Die Kunden dürfen dabei in der Regel nur einmal besucht bzw. bedient werden, wobei einige Ausprägungen des Orientierungsproblems von dieser Rahmenbedingung abweichen. Ein konkretes Anwendungsbeispiel ist die Planung von Auslieferungen, um möglichst ressourcensparend und effizient Waren auszuliefern. Wenn nun mehrere Lieferfahrzeuge zur Verfügung stehen, müssen die Kunden und Routen entsprechend so ausgewählt werden, dass ein Kunde nicht von zwei Lieferfahrzeugen besucht wird und die Routen für die gesamte Fahrzeugflotte optimiert sind. Dieses Szenario lässt sich mit dem *Team Orienteering Problem (TOP)* modellieren. Für den Fall, dass die maximale Kapazität der Fahrzeuge relevant für die Planung ist, kann das *Capacitated Team Orienteering Problem (CTOP)* angewendet werden. Die weitere Spezialisierung *CTOP with incomplete service (CTOP-IS)* kommt dann zum Einsatz, wenn ein Kunde nicht vollständig bedient werden muss und dadurch weitere profitable Kunden bedient werden können. Als Anwendungsbeispiel können hierfür Fahrer eines Lieferunternehmens angenommen werden, die teilnehmende Restaurants bedienen, indem sie Mahlzeiten abholen und an die Kunden der Restaurants ausliefern. Die Fahrer müssen nicht alle Lieferungen eines Restaurants übernehmen und können auf dem Lieferweg weitere Restaurants bedienen bzw. weitere Mahlzeiten abholen. Weiterhin lässt sich das *CTOP* in das *Split Delivery CTOP (SDCTOP)* spezialisieren, wenn Kunden von mehreren Fahrzeugen bedient werden dürfen. Diese Variante könnte bei Lieferungen durch einen Möbelhändler zur Anwendung kommen, wenn sperrige und handliche Artikel in unterschiedlichen Fahrzeugen transportiert werden sollen (vgl. Vansteenwegen/Gunawan 2019: 83-84).

Beim Komplettladungsverkehr werden die Waren eines Kunden abgeholt und an ein gewünschtes Ziel befördert. Dabei wird das Lieferfahrzeug ausschließlich mit den Waren eines Kunden beladen. Da der Profit erst dann erzielt wird, wenn die Waren vom Start- zum Zielort befördert wurden bzw. wenn eine Kante im Graphen traversiert wurde, werden hier die Kunden als Kanten im Graphen repräsentiert. Dieses Anwendungsszenario lässt sich mit dem

Orienteering Arc Routing Problem (OARP) bzw. *Team OARP (TOARP)* modellieren, falls mehrere Lieferfahrzeuge zur Verfügung stehen (vgl. Vansteenwegen/Gunawan 2019: 84).

Ein weiteres Anwendungsgebiet in der Logistik ist die Verwaltung von Lagerbeständen. Zur Minimierung von Lagerkosten ist es für Unternehmen vorteilhaft, wenn Waren nur bei Bedarf bzw. wenn nur unmittelbar benötigte Mengen von einem Zulieferer bezogen werden. Für den Zulieferer bedeutet dies aber, dass er das Unternehmen mehrmals mit entsprechend kleineren Lieferungen anfahren muss. Um Lieferkosten zu minimieren bevorzugen Zulieferer stattdessen wenige Lieferfahrten mit entsprechend hohem Liefervolumen. Um diese entgegengesetzten Ziele adäquat adressieren zu können, werden in der Praxis die Bestände eines Unternehmens vom Zulieferer selbst verwaltet. Dieser muss gewährleisten, dass die Lagerbestände ausreichend aufgestockt werden und sowohl seine als auch die Kosten des Kunden minimiert werden. Dieses *Inventory Routing Problem (IRP)* beschäftigt sich demnach mit der Planung der Lieferungen. Wenn die Bedarfe eines Kunden eine Kontinuität über einen längeren Zeitraum aufweisen, kann das *Cyclic IRP (CIRP)* zur Modellierung des Anwendungsfalls verwendet werden. Im Gegensatz zum *IRP* werden hier unendliche Planungszeiträume mit wiederkehrenden Bedarfen angenommen. Für den Fall, dass nur ein Lieferfahrzeug zur Verfügung steht, kommt das *Single-Vehicle CIRP (SV-CIRP)* zum Einsatz. Diese Spezialisierung ähnelt dem *Inventory Orienteering Problem (IOP)* mit dem Unterschied, dass beim *SV-CIRP* ein Zyklus nicht limitiert wird (vgl. Vansteenwegen/Gunawan 2019: 85).

Wenn Kunden gruppiert werden und die Profite einer Gruppe nur dann eingesammelt werden können, wenn alle Kunden der Gruppe bedient wurden, wird das *Clustered Orienteering Problem (COP)* zur Modellierung verwendet. Das *Set Orienteering Problem (SetOP)* wird hingegen verwendet, wenn nur ein Kunde einer Gruppe bedient werden muss, um die Profite aller Kunden der Gruppe einnehmen zu können. Das kann beispielsweise dann sinnvoll sein, wenn das Routing innerhalb einer Gruppe von den Kunden intern günstiger realisiert werden kann als von einem externen Zulieferer. In diesem Fall wird nur ein Kunde mit den Waren der gesamten Gruppe beliefert. Der Kunde leitet die Waren dann unabhängig vom Zulieferer an seine Partner weiter (vgl. Vansteenwegen/Gunawan 2019: 85).

Der letzte von Vansteenwegen und Gunawan (2019: 85) beschriebene Anwendungsfall im Bereich der Logistik bezieht sich auf den Kontext humanitärer Hilfe. Hier wird ein Bergungshubschrauber angenommen, der mit der Suche nach Überlebenden und der Versorgung gefundener Personen beauftragt wird. Das Problem besteht hier darin, dass während der Versorgung gefundener Personen die Suche nach weiteren Überlebenden pausiert werden muss. Der Profit ist hier in Relation zu den Kosten zu betrachten. Je größer die Not einer gefundenen Person, desto eher lohnt es sich zu landen und die Person zu versorgen. Andernfalls könnte es sinnvoller sein, diese Person nicht zu versorgen und nach weiteren Überlebenden zu suchen, die in größerer Not schweben. Dieses Szenario lässt sich als *Orienteering Problem with variable Profits (OPVP)* modellieren.

3.4.2 Tourismus

Das Orientierungsproblem findet auch im Tourismusbereich Einzug. Ein Reiseveranstalter, der für seine Kunden planen muss, welche Sehenswürdigkeiten sie an einem Ort in welcher Reihenfolge besuchen sollten, ist eine Beispiel für solch eine Anwendung. Abhängig von den Präferenzen der Kunden würde der Reiseveranstalter in diesem Fall die Sehenswürdigkeiten mit Profiten versehen. Wenn der Besuch der Sehenswürdigkeiten auf mehrere Tage verteilt werden

soll, kann auch hier das *Team Orienteering Problem* zur Modellierung verwendet werden. Für den Fall, dass die Sehenswürdigkeiten nur zu bestimmten Zeiten besucht werden dürfen (z. B. wegen der Öffnungszeiten) kann das *Orienteering Problem with Time Windows (OP-TW)* bzw. *TOP-TW* angewendet werden. Jedes dieser Probleme kann weiter vertieft werden, wenn z. B. einige Sehenswürdigkeiten zwingend besucht werden müssen oder wenn einige Sehenswürdigkeiten nur an bestimmten Tagen besucht werden können. Auch die Profite können über die Tage variieren, wenn z. B. das aktuelle Wetter den Profit beeinflusst (vgl. Vansteenwegen/Gunawan 2019: 88).

Eine dieser Vertiefungen ist das *Time Dependent TOP-TW (TD-TOP-TW)*, welche beispielsweise dann angewendet werden kann, wenn mehrere Transportmöglichkeiten zur Verfügung stehen, welche die Reisezeit zu den Sehenswürdigkeiten beeinflussen. So muss gegebenenfalls für die Fahrt mit öffentlichen Verkehrsmitteln mehr Reisezeit eingeplant werden, als wenn ein privater Reisebus verwendet würde. Ähnlich zum Beispiel mit dem Komplettladungsverkehr aus dem vorangegangenen Abschnitt kann die Strecke zwischen zwei Sehenswürdigkeiten selbst profitbehaftet sein. Das kann z. B. dann zutreffen, wenn der Weg zum nächsten Museum durch eine berühmte Einkaufspassage oder über eine Brücke mit historischer Relevanz führt. Da die Stecken neben ihren Kosten nun auch mit einem Profit versehen sind, wird das Problem als *Arc Orienteering Problem (AOP)* bezeichnet. Eine weitere Ausprägung des Orientierungsproblems ist das *Orienteering Problem with hotel selection (OPHS)*. Wenn sich z. B. eine Reisegruppe vorgenommen hat, Sehenswürdigkeiten in einem größeren Gebiet über mehrere Tage verteilt zu besuchen, so kann es vorteilhaft sein, die Unterkunft regelmäßig zu wechseln, sodass die neue Unterkunft in der Nähe möglichst vieler unbesuchter Sehenswürdigkeiten liegt. Demnach sind die Unterkünfte, welche als Start- und Endknoten interpretiert werden können, bei dieser Variante des Orientierungsproblem veränderlich (vgl. Vansteenwegen/Gunawan 2019: 88).

3.5 Komplexität

Bei der Anwendung des Orientierungsproblems werden vollständige Graphen $G = (V, E)$ angenommen. Das bedeutet, dass zwischen jedem Knotenpaar v_i und v_j aus V eine Kante (v_i, v_j) existiert. Bei Graphen mit n Knoten existieren also $|E| = \frac{n \cdot (n-1)}{2} = \binom{n}{2}$ Kanten. Ziel des Orientierungsproblems ist es, aus dieser Kantenmenge eine Tour bzw. Kantenfolge zu bilden, sodass der durch den Besuch der Knoten erzielte Gesamtprofit maximiert wird, ohne dabei eine vorab definierte Kostenobergrenze T_{max} zu überschreiten. Die Tour muss dabei von einem bestimmten Startknoten ausgehen und in einem definierten Endknoten münden und darf keine Knoten mehrmals besuchen (vgl. Witt 2013: 15).

Die Anzahl möglicher Lösungen bestimmt sich daher durch die Anzahl aller Pfade zwischen zwei Knoten in einem vollständigen Graphen. Der kürzeste Pfad verbindet die Start- und Endknoten direkt mit einer einzigen Kante und hat daher die Länge 1. Für Pfade der Länge 2 stehen $n - 2$ weitere Möglichkeiten zur Verfügung. Bei längeren Pfaden müssen nun auch die unterschiedlichen Permutationen der Knotenkombinationen berücksichtigt werden. So gibt es beispielsweise bei Graphen mit vier Knoten zwei zu betrachtende Pfade der Länge 3, nämlich $(v_{Start}, v_1, v_2, v_{Ende})$ und $(v_{Start}, v_2, v_1, v_{Ende})$.

Sei k mit $0 \leq k \leq n - 2$ die Anzahl der Knoten, die neben dem Start- und Endknoten besucht werden sollen, so können $\binom{n-2}{k}$ Knotenkombinationen ausgewählt und in beliebiger Reihenfolge angeordnet werden. Die Anzahl der Pfade der Länge $k + 1$, in denen neben dem Start- und

Endknoten k weitere Knoten besucht werden, bestimmt sich also durch $\binom{n-2}{k} \cdot k!$. Zur Bestimmung der Anzahl aller Pfade, werden die Pfade für alle möglichen k aufsummiert, sodass sich insgesamt

$$|L| = \sum_{k=0}^{n-2} \binom{n-2}{k} \cdot k!, \qquad k \in \{0, \dots, n-2\}$$

unterschiedliche Pfade zwischen zwei Knoten im vollständigen Graphen ergeben (vgl. Witt 2013: 5, 15).

Wie Tabelle 2 zeigt, käme ein naiver Brute-Force Ansatz, bei dem alle Pfade zur Lösung des Orientierungsproblems berechnet werden, mit wachsender Knotenanzahl schnell an seine Grenzen. Unter der Annahme, dass ein moderner Rechner 100 Millionen Pfade pro Sekunde berechnen kann, könnte solch ein Rechner mit diesem Ansatz die exakte Lösung für einen Graphen mit 10 Knoten innerhalb 1ms berechnen. Für Graphen mit 20 Knoten wäre der Rechner schon fünfeinhalb Jahre beschäftigt. Graphen mit 40 Knoten würden den Rechner 450 Quadrilliarden (10^{27}) Jahre Rechenzeit kosten. Dieser Brute-Force Ansatz hat also eine faktorielle Zeitkomplexität, die mit der $\mathcal{O}$-Notation durch $\mathcal{O}(n!)$ ausgedrückt wird und damit für praktische Anwendungen nicht sinnvoll eingesetzt werden kann.

Knotenanzahl des Graphen	Anzahl möglicher Pfade	Rechenzeit bei $10^8 \frac{Pfade}{Sekunde}$
4	5	$50\ ns$
5	16	$160\ ns$
6	65	$650\ ns$
10	109.601	$\sim 1\ ms$
15	$\sim 1{,}69 \cdot 10^{10}$	$\sim 3\ min$
20	$\sim 1{,}74 \cdot 10^{16}$	$\sim 5{,}5\ Jahre$
40	$\sim 1{,}42 \cdot 10^{45}$	$\sim 4{,}5 \cdot 10^{29}\ Jahre$

Tabelle 2: Anzahl möglicher Pfade zwischen zwei Knoten in vollständigen Graphen

Nach Garey und Johnson (1979: 113) ist ein Problem NP-schwer, wenn es mindestens so schwer ist, wie alle NP-vollständigen Probleme. Im Falle, dass ein Algorithmus gefunden würde, der dieses NP-schwere Problem deterministisch in Polynomialzeit lösen könnte, gäbe es dann für alle NP-vollständigen Probleme auch jeweils einen deterministischen Algorithmus, der die Lösung in Polynomialzeit lösen könnte nachdem die Probleme polynomiell auf das NP-schwere Problem reduziert wurden. Wie bereits in Abschnitt 3.3 erläutert, setzt sich das Orientierungsproblem aus dem Rucksackproblem und dem Problem des Handlungsreisenden (*TSP*) zusammen, die sich laut dem Informatiker Richard Karp (1972: 93 ff.) der Klasse NP-vollständig zuordnen lassen.

Um zu zeigen, dass das Orientierungsproblem NP-schwer ist, wird zunächst gezeigt, dass es einen Algorithmus gibt, der die exakte Lösung deterministisch in Polynomialzeit bestimmen kann. Weiterhin wird das Problem des Handlungsreisenden herangezogen, in welchem die Frage lautet, ob eine Tour im Graphen existiert, die alle Knoten des Graphen besucht und kürzer als T ist. Diese Variante der TSP ist ein NP-vollständiges Entscheidungsproblem, da die Lösung in Polynomialzeit verifiziert werden kann und alle Probleme der Klasse NP auf das Problem in Polynomialzeit reduziert werden können (vgl. Vossen/Witt 2016: 407 f.). Das NP-vollständige Problem des Handlungsreisenden lässt sich nun auf das Orientierungsproblem reduzieren, indem

jeder Knoten mit einem Profitwert von 1 versehen und zwei beliebige Start- und Endknoten gewählt werden und $T_{max} = T$ gesetzt wird. Nun könnte der effiziente Algorithmus zur Lösung des Orientierungsproblems auf dieses Modell angewendet werden, um auch diese Variante des TSP in Polynomialzeit lösen zu können. Wenn nämlich der Algorithmus eine Tour als Lösung liefert, dessen erzielter Gesamtprofit mit der Anzahl der Knoten übereinstimmt, dann kann die Frage des Entscheidungsproblems bejaht werden. Falls solch eine Tour durch den Algorithmus nicht gefunden werden kann, würde die Frage des Entscheidungsproblems verneint werden können (vgl. Golden et al. 1987: 307 f.).

Weil bisher kein Algorithmus gefunden wurde, der das Orientierungsproblem deterministisch in Polynomialzeit lösen kann, werden in der Praxis heuristische Verfahren angewendet, die sich der exakten Lösung in verhältnismäßig zumutbarer Laufzeit annähern. Die heuristischen Verfahren können sich dabei unter anderem in ihrer Laufzeit und in der Güte der von ihnen erzeugten Lösungen unterscheiden.

4 Heuristische Algorithmen zur Lösung des Orientierungsproblems

In diesem Kapitel werden zwei heuristische Algorithmen zur Lösung des Orientierungsproblems vorgestellt. Da beide Algorithmen das gleiche Optimierungsproblem betrachten, überschneiden sich stellenweise auch ihre Eingabeparameter. Die Gemeinsamkeiten werden daher im Abschnitt 4.1 erläutert. Anschließend werden sowohl der *Adaptive Large Neighbourhood Search* Algorithmus von Santini (2019) als auch der evolutionäre Algorithmus von Kobeaga et al. (2018) vorgestellt.

4.1 Gemeinsame Eingabeparameter und Funktionen

Beide Algorithmen betrachten einen metrischen gerichteten Graphen G, der sich aus einer Knotenmenge V und einer initial leeren Kantenmenge E zusammensetzt. Die Knotenmenge besteht aus mindestens zwei Knoten, wobei ein Knoten den Startpunkt und ein weiterer Knoten den Endpunkt der zu berechnenden Tour darstellt. Während die Start- und Endknoten die gleichen Koordinaten besitzen dürfen, müssen die Koordinaten der übrigen Knoten verschieden sein, um Kreise innerhalb einer Tour zu vermeiden. Neben ihren Koordinaten besitzen die Knoten einen Profitwert. Der erzielte Profit einer Tour berechnet sich aus der Summe der Profite aller besuchten Knoten. Zu beachten ist, dass sowohl der Start- als auch der Endknoten keinen Profitwert besitzen, während die Profite der übrigen Knoten größer als Null sein müssen. In dieser Arbeit wird außerdem ein zweidimensionaler Raum angenommen, sodass die Koordinaten der Knoten lediglich aus einer x- und y-Koordinate bestehen.

Ein weiterer gemeinsamer Parameter ist die Kostenobergrenze T_{max}. Wie in Abschnitt 3.2 beschrieben, soll für den Graphen G eine Tour gefunden werden, welche den Gesamtprofit maximiert und dabei diese Kostenobergrenze nicht überschreitet. Als Kosten werden in dieser Arbeit die euklidischen Distanzen zwischen den Knoten verwendet.

Um die Güte einer Lösung messen und Lösungen miteinander vergleichen zu können, wird außerdem eine Ziel- bzw. Fitnessfunktion $f(\cdot)$ benötigt. Die Fitnessfunktion wird in dieser Arbeit durch den Quotienten aus dem erzielten Gesamtprofit zur Summe der Profite aller Knoten beschrieben, also:

$$f = \frac{\sum Profite\ besuchter\ Knoten}{\sum Profite\ aller\ Knoten}$$

4.2 Adaptive Large Neighbourhood Search Algorithmus

Der von Santini (2019) vorgestellte *Adaptive Large Neighbourhood Search* Algorithmus (ALNS) zur heuristischen Lösung des Orientierungsproblems ist eine Abwandlung des ALNS Algorithmus zur Lösung des Pickup-And-Delivery Problems mit Zeitfenstern, welches erstmals von Ropke und Pisinger (2006) vorgestellt wurde (vgl. Santini 2019: 6). Beim Pickup-And-Delivery Problem müssen Routen für mehrere Transportfahrzeuge gefunden werden, sodass Waren möglichst effizient von definierten Standorten abgeholt und zu definierten Zielorten geliefert werden können (vgl. Ropke/Pisinger 2006: 455).

Die grundlegende Idee des ALNS Algorithmus besteht darin, eine initial erzeugte Lösung iterativ zu verändern, indem Knoten aus der Tour entfernt und neue Knoten hinzugefügt werden. Das Entfernen und Hinzufügen von Knoten wird durch sogenannte Zerstör- und Reparaturmethoden realisiert. Dabei muss gewährleistet sein, dass die in einer Iteration erzeugte Lösung zulässig ist

bzw. die Kostenobergrenze T_{max} nicht überschreitet und weitere in Abschnitt 3.2 beschriebene Rahmenbedingungen des Orientierungsproblems erfüllt (vgl. Santini 2019: 6).

Wie im Pseudocode des ALNS Algorithmus in Abbildung 3 zu erkennen ist, werden neben der Instanz des Orientierungsproblems acht Eingabeparameter erwartet. Der Parameter M beschreibt die Anzahl der durchzuführenden Iterationen. Parameter x_0 enthält eine initial erzeugte Lösung. Die Vorgehensweise zur Erzeugung dieser initialen Lösung wird in Abschnitt 4.2.1 detailliert beschrieben. Weiterhin wird je eine Liste von Zerstör- und Reparaturmethoden erwartet, aus denen in jeder Iteration entsprechend eine Methode per gewichtetem Zufall ausgewählt wird. Die Gewichte bestimmen sich anhand der Bewertungen der Methoden λ_D bzw. λ_R. Darüber hinaus besteht optional die Möglichkeit, eine lokale Suchmethode L zur Optimierung der Lösung anzuwenden. Schließlich wird noch eine Ziel- bzw. Fitnessfunktion $f(\cdot)$ erwartet, mit welcher die Güte der Lösungen bewertet werden kann und die einen Vergleich verschiedener Lösungen ermöglicht vgl. Santini 2019: 6, 8).

Bei der Anwendung des ALNS Algorithmus müssen tatsächlich aber nur die Anzahl der Iterationen M, ein Aggressivitätsfaktor α (s. Abschnitt 4.2.3) und ein sogenannter Decay-Faktor h (s. Abschnitt 4.2.8) zur Konfiguration des Algorithmus vom Benutzer eingegeben werden, da die übrigen Parameter feste Bestandteile des Programms sind oder automatisch ermittelt werden.

Der ALNS Algorithmus beginnt mit einer Initialisierungsphase, welche sich über die ersten fünf Zeilen des Pseudocodes erstreckt. Hier wird zunächst eine initiale Lösung x_0 (s. Abschnitt 4.2.1) erzeugt und den Variablen x und x^* zugewiesen. Dabei stellt x^* die global-beste Lösung dar, während x die aktuelle Lösung repräsentiert. Weiterhin werden in dieser Phase sowohl die Iterationsvariable i als auch die Bewertungen λ_D und λ_R mit dem Wert 1 initialisiert (vgl. Santini 2019: 6, 8).

Die Zeilen 6 bis 19 des Pseudocodes beschreiben den Hauptteil des ALNS Algorithmus, in welchem iterativ neue Lösungen durch das Entfernen und Hinzufügen von Knoten aus der aktuellen Lösung erzeugt werden. Zunächst werden per gewichtetem Zufall je eine Zerstör- und eine Reparaturmethode ausgewählt. In den Abschnitten 4.2.3 und 4.2.4 werden die von Santini vorgesehenen Methoden erläutert. Es können jedoch auch andere Heuristiken integriert werden, was den ALNS Algorithmus in dieser Hinsicht erweiterbar macht. Auf die Lösung x wird zunächst die gewählte Zerstörmethode angewendet. Das Ergebnis wird anschließend der gewählten Reparaturmethode zugeführt und das Resultat in einer Variablen x' gespeichert. Um sicherzustellen, dass diese Lösung zulässig ist bzw. die Kostenobergrenze T_{max} nicht überschreitet, muss für unzulässige Lösungen noch die in Abschnitt 4.2.5 beschriebene Methode zur Wiederherstellung der Zulässigkeit angewendet werden (vgl. Santini 2019: 6, 8).

Im Anschluss kann optional eine lokale Suchmethode zur Verbesserung der Lösung x' angewendet werden. Zweck dieser Methode ist es, für die besuchten Knoten der Lösung ein besseres Routing zu finden. Damit sollen die Kosten der aktuellen Tour gesenkt werden, um das Hinzufügen weiterer Knoten in der nächsten Iteration zu ermöglichen. In Abschnitt 4.2.6 wird näher auf diese Methode eingegangen (vgl. Santini 2019: 6, 8).

Wenn die neu erzeugte Lösung das Akzeptanzkriterium erfüllt, welches im Abschnitt 4.2.7 vorgestellt wird, so wird sie als neue aktuelle Lösung x übernommen. Wenn diese Lösung sogar die global-beste Lösung x^* übertrifft, so wird Letztere in Zeile 15 des Pseudocodes durch die neue Lösung ersetzt (vgl. Santini 2019: 8).

Die in einer Iteration verwendeten Zerstör- und Reparaturmethoden werden abhängig von der Güte der erzeugten Lösung bewertet. Mit dem Bewertungssystem, auf welches in Abschnitt 4.2.8 eingegangen wird, soll sichergestellt werden, dass Methoden, die gute Lösungen erzeugt haben, in den nächsten Iterationen mit einer höheren Wahrscheinlichkeit gewählt werden. Nachdem M Iterationen abgeschlossen wurden, wird die bis dahin gefundene global-beste Lösung x^* als Ergebnis zurückgegeben. In dieser Arbeit wird außerdem optional ermöglicht, den Algorithmus dann zu terminieren, wenn M Iterationen abgeschlossen oder ein einstellbares Zeitlimit erreicht wurde. Die zeitliche Begrenzung kann eine bessere Vergleichbarkeit der Algorithmen unterstützen und ermöglicht auch das Testen größerer Graphen in Benchmarking-Tests in vertretbarer Zeit (vgl. Santini 2019: 8).

```
// Instanz des Orientierungsproblems
EINGABE: Graph G mit G = (V,E,c,s), |V| = n ≥ 2, E = ø
EINGABE: Start- und Endknoten des Graphen v0 und v1
EINGABE: Kostenvektor c = (c_ij)_{i,j=1}^n, c_ij ≥ 0
EINGABE: Profitvektor s = (s_i)_{i=1}^n, s_i ≥ 0
EINGABE: Kostenobergrenze T_max

// Parameter des ALNS Algorithmus
EINGABE: Anzahl der Iterationen M
EINGABE: Initiale Lösung x0
EINGABE: Liste von Zerstörmethoden D
EINGABE: Liste von Reparaturmethoden R
EINGABE: Bewertungen der Zerstörmethoden λD, ∀D ∈ D
EINGABE: Bewertungen der Reparaturmethoden λR, ∀R ∈ R
EINGABE: Lokale Suchmethode L
EINGABE: Ziel- bzw. Fitnessfunktion f(·)

AUSGABE: Heuristische Lösung x*

   1     x = x0
   2     x* = x0
   3     i = 1
   4     λD = 1, ∀D ∈ D
   5     λR = 1, ∀R ∈ R
   6     WHILE i ≤ M DO
   7        Wähle D ∈ D mit Wahrscheinlichkeit proportional zu λD
   8        Wähle R ∈ R mit Wahrscheinlichkeit proportional zu λR
   9        Berechne x‘ = R(D(x))
   10       Wenn neue global-beste Lösung gefunden wurde: x‘ = L(x‘)
   11       IF x‘ erfüllt Akzeptanzkriterium THEN
   12             x = x‘
   13       END IF
   14       IF f(x) > f(x*) THEN
   15             x* = x
   16       END IF
   17       Aktualisiere Bewertungen λD und λR
   18       i = i+1
   19    END WHILE
   20    RETURN x*
```

Abbildung 3: Pseudocode des ALNS-Algorithmus (Santini 2019: 8, modifiziert)

4.2.1 Erzeugung einer initialen Lösung

Zu Beginn des ALNS Algorithmus wird eine initiale Lösung bzw. Tour erzeugt, die im späteren Verlauf durch Anwendung von Zerstör- und Reparaturmethoden iterativ verändert wird. Zur Erzeugung dieser Lösung werden zunächst der Start- und Endknoten des Graphen ($v_{Start},\ v_{Ende} \in V$) zur Tour hinzugefügt, indem die Kante (v_{Start}, v_{Ende}) in die leere Tour aufgenommen wird.

Anschließend werden die übrigen unbesuchten Knoten gemischt bzw. in randomisierter Reihenfolge gezogen. Für jeden dieser Knoten wird die Stelle in der Tour gesucht, bei der durch das Einfügen des Knotens die geringsten Mehrkosten entstehen. Der Knoten wird jedoch nur dann in die Tour aufgenommen, wenn die dadurch entstehende Tour nicht die Kostenobergrenze T_{max} überschreitet. Falls das Einfügen des Knotens zur Verletzung der Kostenobergrenze führt, wird er ignoriert und der nächste Knoten in der gemischten Knotenmenge gezogen (vgl. Santini 2019: 9).

Für den besonderen Fall, dass die Kostenobergrenze kleiner als die Distanz zwischen v_{Start} und v_{Ende} ist, wird der ALNS Algorithmus abgebrochen, da keine Tour vom Start- zum Endknoten gebildet werden kann, ohne die Kostenobergrenze zu überschreiten.

4.2.2 Clustering

Ein wichtiges Konzept, auf denen sich zwei der in den nächsten Abschnitten vorgestellten Zerstör- und Reparaturmethoden stützen (s. Abschnitte 4.2.3.3 und 4.2.4.4), ist das sogenannte *Clustering* von Graphen bzw. das Gruppieren nahegelegener Knoten. Das Orientierungsproblem beschäftigt sich mit der Suche nach einem Pfad oder Kreis, bei welcher die Knoten eines Graphen so besucht werden, dass der Profit maximiert wird und dabei eine bestimmte Kostenobergrenze T_{max} nicht überschritten wird. Beim ALNS Algorithmus werden hierfür aus einer Lösung iterativ Knoten entfernt und anschließend Knoten in die Tour wieder eingefügt. Dabei gibt es verschiedene Methoden bzw. Heuristiken nach denen diese Knoten ausgewählt werden können, zu denen auch die Wahl von Clustern zählt. Ein Cluster C beschreibt hierbei eine Teilmenge der Knotenmenge V des Graphen G. Ein Graph kann in mehrere disjunkte Cluster $C_1, \dots, C_m \subset V$ mit $C_i \cap C_j = \emptyset,\ i \neq j,\ \forall i, j$ unterteilt werden. Beim ALNS Algorithmus von Santini (2019) wird dabei ein potentiell unvollständiges Clustering angenommen, bei dem nicht alle Knoten einem Cluster zugewiesen werden müssen (vgl. Santini 2019: 4-6).

Die Verwendung von Clustern entstammt folgender Überlegung: Wenn in eine Tour ein Knoten v_i aufgenommen werden soll, der relativ weit entfernt ist, so kann es vorteilhaft sein, auch seine örtlich naheliegenden Nachbarknoten in die Tour aufzunehmen. Der Grund hierfür ist, dass zur Aufnahme des Knotens v_i in die Tour bereits relativ hohe Mehrkosten in Kauf genommen wurden. Die Aufnahme der örtlichen Nachbarn ist demgegenüber relativ günstig. Analog gilt die Überlegung auch für das Entfernen eines Knotens. Hier kann es von Vorteil sein, wenn gleich das gesamte Cluster aus der Tour entfernt wird, um auch die hohen Reisekosten zum Cluster selbst einzusparen (vgl. Santini 2019: 4 f.).

Abbildung 4 veranschaulicht diesen Sachverhalt. Ausgehend von der auf der linken Seite der Abbildung dargestellten Tour soll nun mindestens ein weiterer Knoten hinzugefügt werden. Zur Auswahl stehen fünf Knoten, wobei vier Knoten einem Cluster zugeordnet werden können. Unter der Annahme, dass die Profite der zur Auswahl stehenden Knoten gleich sind oder sich nicht wesentlich unterscheiden, kann es vorteilhafter sein, den Knoten eines Clusters und seine

örtlichen Nachbarn zu besuchen. Im Vergleich zu den Kanten, die zum Cluster hinführen bzw. vom Cluster zurückführen, sind die Kanten innerhalb des Clusters verhältnismäßig kurz. Da ohnehin ein Knoten hinzugefügt werden muss und damit hohe Mehrkosten nicht vermieden werden können, ist es also vorteilhafter, auch die Nachbarn des Knotens für vergleichsweise geringe Mehrkosten mit in die Tour aufzunehmen.

Dieser Sachverhalt lässt sich auch auf praktische Anwendungsfälle übertragen. Wenn beispielsweise eine Reise von Bonn nach Berlin geplant ist, um fünf Sehenswürdigkeiten zu besuchen und der Reiseveranstalter kundgibt, dass leider vier dieser Sehenswürdigkeiten wegen geplanten Baumaßnahmen oder Gesundheitsverordnungen nicht besucht werden können, wäre es sinnvoller, die Reise nach Berlin gänzlich abzusagen bzw. zu vertagen. Der Besuch einer Sehenswürdigkeit würde in diesem Fall die Reisekosten nach Berlin nicht rechtfertigen, weshalb das gesamte Cluster „Berlin“ aus der Reise entfernt werden würde.

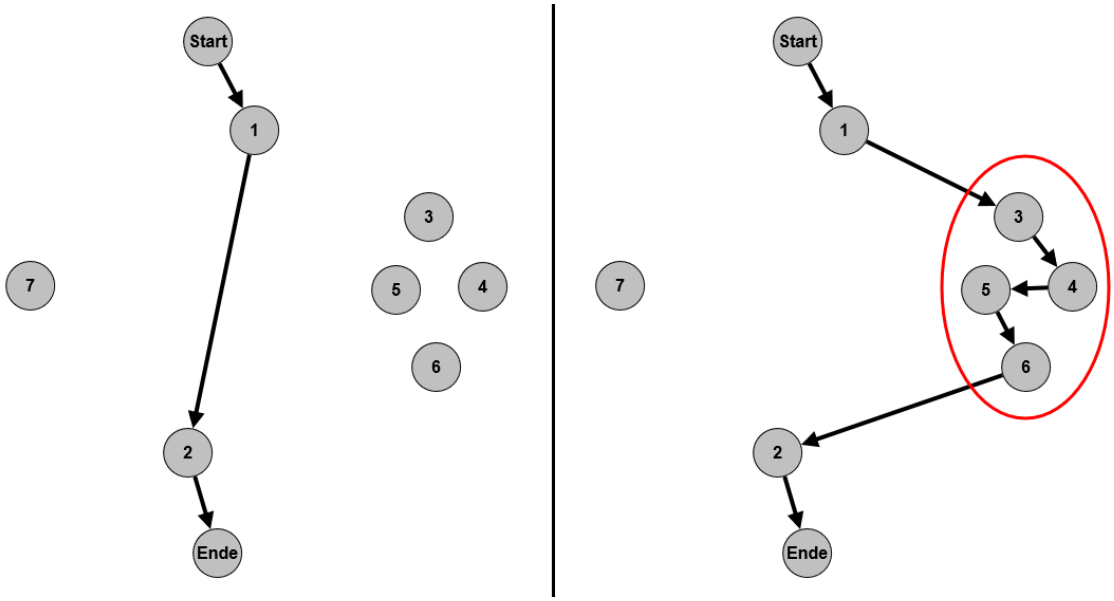


Abbildung 4: Hinzufügen eines Clusters

Für das Clustering eines Graphen wird der von Santini (2019) abgeänderte Dbscan-Algorithmus verwendet, welcher von Ester et al. (1996) vorgestellt wurde. Der Name „Dbscan“ steht für „Density-based spatial clustering of applications with noise“. Im Gegensatz zu anderen Clustering-Algorithmen benötigt dieser Algorithmus nicht die Anzahl zu erzeugender Cluster als Eingabeparameter. Stattdessen wird ein Radius $r \in \mathbb{R}^+$ und eine Zahl $N \in \mathbb{N}^+$ erwartet. Der Radius beschreibt die maximale Distanz zweier Knoten zueinander, um noch als Nachbarn zählen zu können. Die Zahl N wird zur Bestimmung sogenannter Kernknoten benötigt. Ein Knoten wird dann als Kernknoten bezeichnet, wenn er zu $N - 1$ weiteren Knoten benachbart ist. Die Nachbarn eines Kernknotens, die selber auch Kernknoten sein dürfen, werden dann als erreichbare Knoten bezeichnet. Ein Cluster wird gebildet, in dem ausgehend von einem Kernknoten alle seine erreichbaren Knoten hinzugefügt werden und dann rekursiv von allen weiteren Kernknoten im Cluster deren erreichbare Nachbarn ins Cluster aufgenommen werden. Knoten, die weder Kernknoten noch erreichbare Knoten sind, werden als Außenseiter bezeichnet (vgl. Santini 2019: 5 f.).

Der Dbscan-Algorithmus wurde von Santini (2019) so angepasst, dass sowohl der Radius r als auch die Nachbarsanzahl für Kernknoten N automatisch bestimmt werden können. Hierzu werden

die jeweiligen Abstände aller Knoten zu den nächstliegenden Knoten im Graphen betrachtet. Der Radius r wird dann auf den größten dieser Abstandswerte gesetzt, sodass jeder Knoten mindestens einen Nachbarn besitzt und damit eine Chance hat, zu einem Cluster zugeordnet zu werden. Zur Bestimmung der Zahl N werden für alle Knoten die Anzahl ihrer Nachbarn betrachtet. Die Zahl N wird auf die am häufigsten auftretende Nachbarsanzahl gesetzt (vgl. Santini 2019: 5 f.).

4.2.3 Zerstörmethoden

Durch die Anwendung einer Zerstörmethode auf eine Lösung bzw. Tour werden eine Menge besuchter Knoten aus dieser Lösung entfernt, indem die entsprechenden Kanten aus der Kantenmenge gelöscht werden. Ausgenommen von dieser Methode sind der Start- und Endknoten der Tour, die in jedem Fall erhalten bleiben.

Die Anzahl der zu entfernenden Knoten wird durch den Aggressivitätsfaktor $\alpha \in \mathbb{R}: 0 < \alpha < 1$ beeinflusst, welche der Zerstörmethode übergeben werden muss. Wird eine Zerstörmethode auf eine Tour angewendet, die neben den Start- und Endknoten noch k weitere Knoten besucht, so werden maximal $\alpha \cdot k$ Knoten entfernt. Wenn der Faktor α entsprechend klein gewählt wird bzw. gegen 0 geht, werden in einer Iteration keine Knoten aus der Tour entfernt. Dies hätte zur Folge, dass durch die darauffolgenden Reparaturmethoden keine neuen Knoten hinzugefügt werden können und folglich in den Iterationen des ALNS Algorithmus keine Veränderungen der Lösung erzielt werden können. Wird der Faktor α jedoch zu groß gewählt, so werden in jeder Iteration ausgenommen vom Start- und Endknoten alle besuchten Knoten entfernt. Die Konsequenz wäre, dass in jeder Iteration eine neue Lösung erzeugt werden würde, die keine nicht-zufälligen Gemeinsamkeiten zu den Lösungen vorangegangener Iterationen besitzt (vgl. Santini 2019: 9).

Zu beachten ist, dass eine Zerstörmethode keine zulässige Lösung hinsichtlich der Kostenobergrenze erzeugen muss. Da hier aber nur zulässige Lösungen der Zerstörmethode zugeführt werden, können keine unzulässigen Lösungen entstehen. Dies liegt daran, dass durch die Anwendung einer Zerstörmethode die Kosten der Tour nicht steigen können, da keine Knoten hinzugefügt werden und die Dreiecksungleichung (s. Abschnitt 2.1.2) gilt (vgl. Santini 2019: 9).

Im Folgenden werden die von Santini (2019) verwendeten Zerstörmethoden vorgestellt.

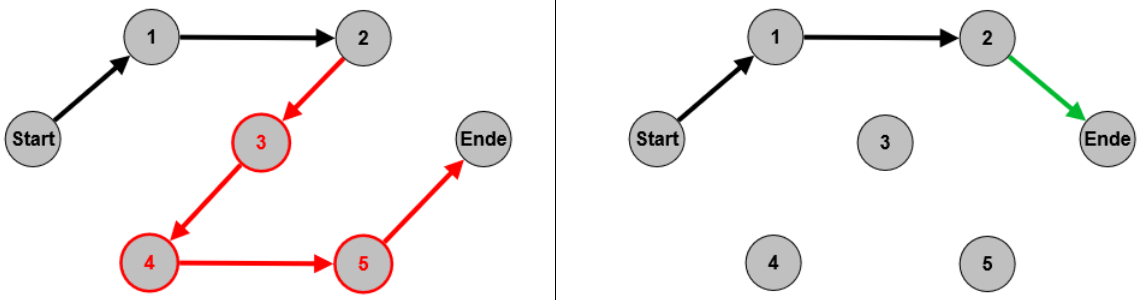
4.2.3.1 Random Remove

Mit der *Random Remove* Methode werden $\alpha \cdot k$ zufällig ausgewählte Knoten aus einer Tour $(v_{Start}, v_1, \dots\ v_k, v_{Ende})$ entfernt. Die Knoten v_{Start} und v_{Ende} bleiben der Tour jedoch erhalten, da sie nicht entfernt werden dürfen. Wird ein Knoten aus der Tour entfernt, so werden sein Vorgänger und sein Nachfolger durch eine neue Kante verbunden. Aufgrund der geltenden Dreiecksungleichung (s. Abschnitt 2.1.2) kann diese neu erzeugte Tour nicht länger als die vorherige Tour sein (vgl. Santini 2019: 9).

4.2.3.2 Random Sequence Remove

Eine Eigenschaft der *Random Remove* Methode ist, dass zu entfernende Knoten gleichmäßig im Graphen verteilt sind, da sie zufällig ausgewählt werden. Dies kann dazu führen, dass die direkten Nachbarn eines entfernten Knotens von der Zerstörmethode unberührt bleiben bzw. nicht aus der Tour entfernt werden. Wenn im Rahmen einer Reparaturmethode vorgesehen wird, einen dieser entfernten Knoten wieder hinzuzufügen, so ist die Wahrscheinlichkeit entsprechend hoch, dass er wieder in seine ursprüngliche Stelle in der Tour eingefügt wird (vgl. Santini 2019: 9 f.).

Um diesen Effekt zu vermeiden, werden mit der *Random Sequence Remove* Methode die Knoten entfernt, die mit einer Kantenfolge direkt verbunden sind bzw. eine Sequenz bilden. Dazu wird aus einer Tour $(v_{Start}, v_1, \dots\ v_k, v_{Ende})$ zunächst ein Knoten v_i gewählt. Anschließend wird dieser Knoten und $\alpha \cdot k - 1$ seiner Nachfolger aus der Tour entfernt. Für den Fall, dass nicht genügend Nachfolger entfernt werden können, weil der Knoten v_i entsprechend weit hinten in der Tour positioniert ist bzw. weil die Anzahl der zu entfernenden Knoten zu groß gewählt wurde, können auch seine Vorgänger entfernt werden. Auch hier ist zu beachten, dass v_{Start} und v_{Ende} nicht aus der Tour entfernt werden dürfen (vgl. Santini 2019: 9 f.).



Annahmen: Knoten v_4 zufällig ausgewählt, $\alpha * k = 3$

Abbildung 5: Veranschaulichung der *Random Sequence Remove* Methode

Beim Beispiel, welches in Abbildung 5 dargestellt ist, wird die *Random Sequence Remove* Methode auf die Tour $(v_{Start}, v_1, v_2, v_3, v_4, v_5, v_{ende})$ angewendet. Unter der Annahme, dass drei Knoten entfernt werden sollen und dass der Knoten v_4 zufällig als erster Knoten ausgewählt wurde, werden durch die Methode die Knoten v_3, v_4 und v_5 entfernt. Der Knoten v_3 wird anstelle von v_{ende} entfernt, da der Endknoten nicht aus der Tour entfernt werden darf. Nachdem die Kanten, welche mit den zu entfernenden Knoten verbunden sind, gelöscht wurden, wird der Knoten v_2 mit dem Endknoten v_{ende} durch eine neue Kante verbunden, wodurch die Tour $(v_{Start}, v_1, v_2, v_{ende})$ entsteht.

4.2.3.3 Random Cluster Remove

Zwar werden mit der *Random Sequence Remove* Methode Knoten entfernt, die in einer Tour unmittelbar aufeinander folgen, jedoch müssen sie nicht zwangsweise örtlich bzw. im Koordinatenraum nahe beieinander liegen. Dies kann zu dem Effekt führen, dass scheinbar zufällig ausgewählte Knoten an verschiedenen Stellen im Graphen entfernt werden. Mit der *Random Cluster Remove* Methode werden stattdessen naheliegende bzw. benachbarte Knoten entfernt.

Bei der *Random Cluster Remove* Methode wird zunächst, wie in Abschnitt 4.2.2 beschrieben, ein Clustering auf den Graphen G angewendet. Da es sich um ein potentiell unvollständiges Clustering handelt, kann es Knoten geben, die keinem Cluster zugeordnet werden und deswegen als Außenseiter bezeichnet werden. Die übrigen Knoten der Knotenmenge V' werden den Clustern $C_1, \dots, C_m \subset V'$ zugeordnet. Aus der Menge von Clustern wird nun ein Cluster C_l zufällig gewählt, aus welchem entsprechend Knoten entfernt werden sollen. Wenn es maximal $\alpha \cdot k$ Knoten im Cluster gibt, die in der Tour enthalten sind, werden mit dieser Methode alle Knoten dieses Clusters aus der Tour entfernt. Sollte das ausgewählte Cluster jedoch mehr als $\alpha \cdot k$

Knoten enthalten, die auch in der Tour enthalten sind, so werden aus dem Cluster $\alpha \cdot k$ Knoten zufällig ausgewählt und entfernt (vgl. Santini 2019: 10).

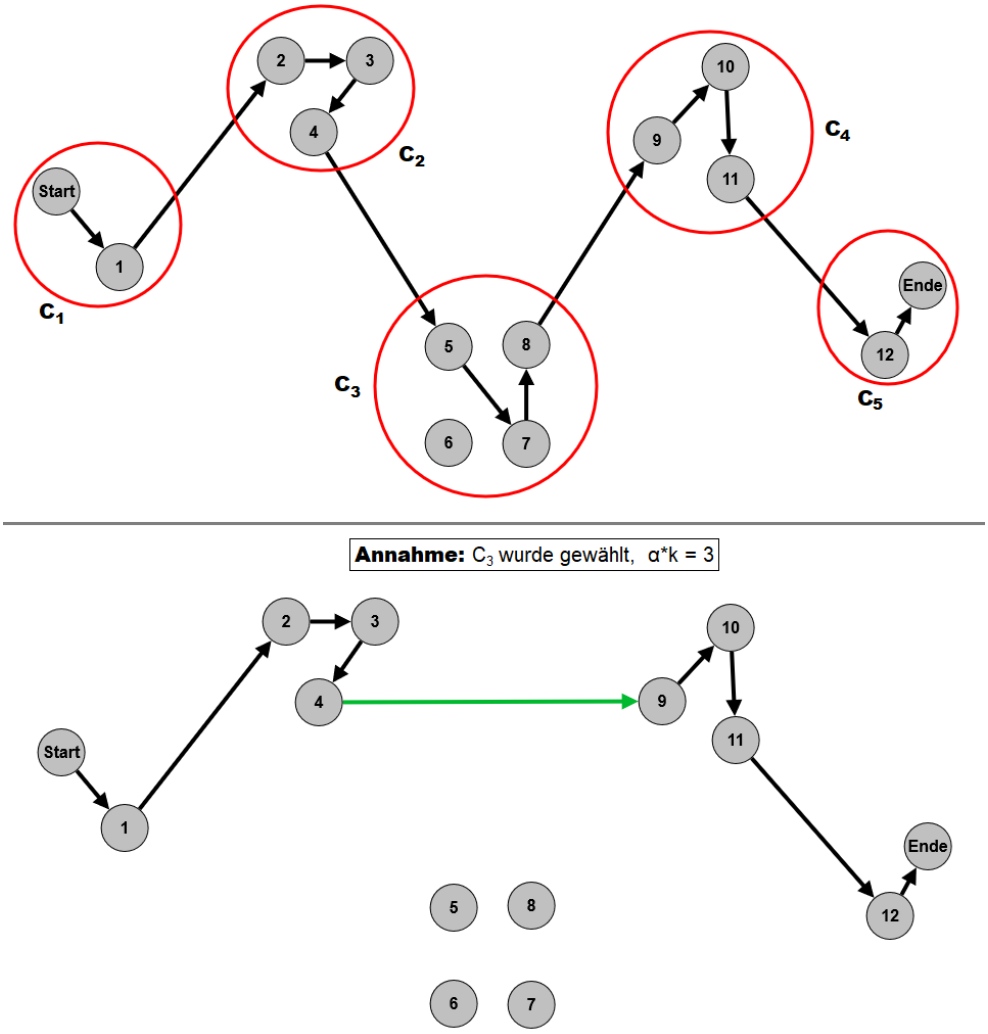


Abbildung 6: Beispielhafte Anwendung der *Random Cluster Remove* Methode

Abbildung 6 zeigt exemplarisch, wie die *Random Cluster Remove* Methode angewendet werden kann. In diesem Beispiel wird angenommen, dass Faktor $\alpha = 0.3$ gewählt wurde. Da neben dem Start- und Endknoten 11 weitere Knoten besucht werden, ergibt sich, dass $\lfloor 0.3 \cdot 11 \rfloor = 3$ Knoten entfernt werden sollen. Wie in der Abbildung dargestellt wird, lässt sich der Graph in fünf Cluster einteilen, wobei hier das Cluster C_3 ausgewählt wurde. Dieses Cluster enthält nur drei Knoten, die auch in der Tour enthalten sind, weshalb alle diese Knoten aus der Lösung entfernt werden können. Im Falle, dass der Knoten v_6 auch in der Tour enthalten wäre, müssten drei der vier Knoten des Clusters zufällig ausgewählt werden. In diesem Beispiel wurde das Cluster C_3 vollständig aus der Tour entfernt und die Tour durch eine neue Kante (v_4, v_9) zusammengesetzt.

4.2.4 Reparaturmethoden

Die Anwendung einer Reparaturmethode auf eine Lösung bezweckt die Erhöhung des erzielten Gesamtprofits durch das Hinzufügen weiterer Knoten in eine Tour. Zwar darf eine Lösung während der Ausführung einer Reparaturmethode die Kostenobergrenze T_{max} vorübergehend überschreiten, jedoch muss nach Abschluss der Methode gewährleistet werden, dass die

Rahmenbedingungen des Orientierungsproblems von der Lösung eingehalten werden. Dafür werden die hier veränderten und potentiell unzulässigen Lösungen der im Abschnitt 4.2.5 beschriebenen Methode zugeführt, in welcher die ungünstigsten Knoten aus der Tour entfernt werden, um die Kosten auf ein zulässiges Maß zu senken (vgl. Santini 2019: 10).

Im Folgenden werden die von Santini (2019) vorgestellten Reparaturmethoden erläutert.

4.2.4.1 Random Repair

Bei der *Random Repair* Methode werden zunächst alle unbesuchten Knoten aufgelistet. Mithilfe einer zufällig erzeugten Zahl $x \in \mathbb{R}: 0 \leq x \leq 1$ wird anschließend bestimmt, welcher Anteil der unbesuchten Knoten der aktuellen Tour hinzugefügt werden sollen. Die hinzuzufügenden Knoten werden zufällig aus der Liste unbesuchter Knoten gewählt (vgl. Santini 2019: 10 f.).

Alle hinzuzufügenden Knoten werden in die Tour an die jeweils günstigste Stelle eingefügt bzw. an den Stellen, an welchen die geringsten Mehrkosten entstehen. Beim Einfügen wird die Kostenobergrenze vorerst nicht berücksichtigt, was potentiell zu einer Verletzung der Rahmenbedingung führen kann. Daher wird nach Abschluss der Reparaturmethode die in Abschnitt 4.2.5 beschriebene Methode auf die hier erzeugte Lösung angewendet (vgl. Santini 2019: 10 f.).

4.2.4.2 Prize Repair

Analog zur *Random Repair* Methode wird auch beim *Prize Repair* zunächst der Anteil hinzuzufügender Knoten mithilfe einer Zufallszahl $x \in \mathbb{R}: 0 \leq x \leq 1$ bestimmt. Statt nun jedoch Knoten zufällig aus der Liste unbesuchter Knoten zu wählen, werden hier die profitabelsten Knoten in die jeweils günstigste Stelle der Tour eingefügt. Da auch hier während der Prozedur die Kostenobergrenze T_{max} nicht berücksichtigt wird, muss im Anschluss die in Abschnitt 4.2.5 beschriebene Methode angewendet werden (vgl. Santini 2019: 11).

4.2.4.3 Greedy Repair

Der *Greedy Repair* Algorithmus betrachtet alle Kombinationen unbesuchter Knoten $v_i \in V'$ zu allen Kanten der Tour $e \in E$, in welchen der Knoten potentiell eingefügt werden kann, und entscheidet iterativ, welche Kombination die bestmögliche Wahl zum aktuellen Zeitpunkt darstellt. Mit der Wahl einer Kombination (v_i, e) und der damit neu entstehenden Tour darf jedoch die Kostenobergrenze T_{max} nicht überschritten werden. Wie in Abbildung 7 illustriert wird, kann durch die ständige Wahl der aktuell besten Kombination das Gesamtproblem jedoch nicht optimal gelöst werden. Zur Vereinfachung werden in diesem Beispiel für alle unbesuchten Knoten gleiche Profitwerte angenommen. Der Knoten v_1 wird vom *Greedy Repair* Algorithmus in der ersten Iteration bevorzugt, da er die geringsten Mehrkosten verursacht. Die restlichen Knoten könnten in diesem Beispiel dann nicht mehr besucht werden, da sonst die Kostenobergrenze überschritten würde. Die optimale Lösung hätte hingegen vorausschauend zunächst den Knoten v_2 mit etwas höheren Mehrkosten als v_1 zur Tour hinzugefügt. Da die Knoten v_3 und v_4 relativ nahe zum Knoten v_2 stehen, können diese anschließend mit relativ geringen Mehrkosten noch in die Tour aufgenommen werden (vgl. Santini 2019: 10).

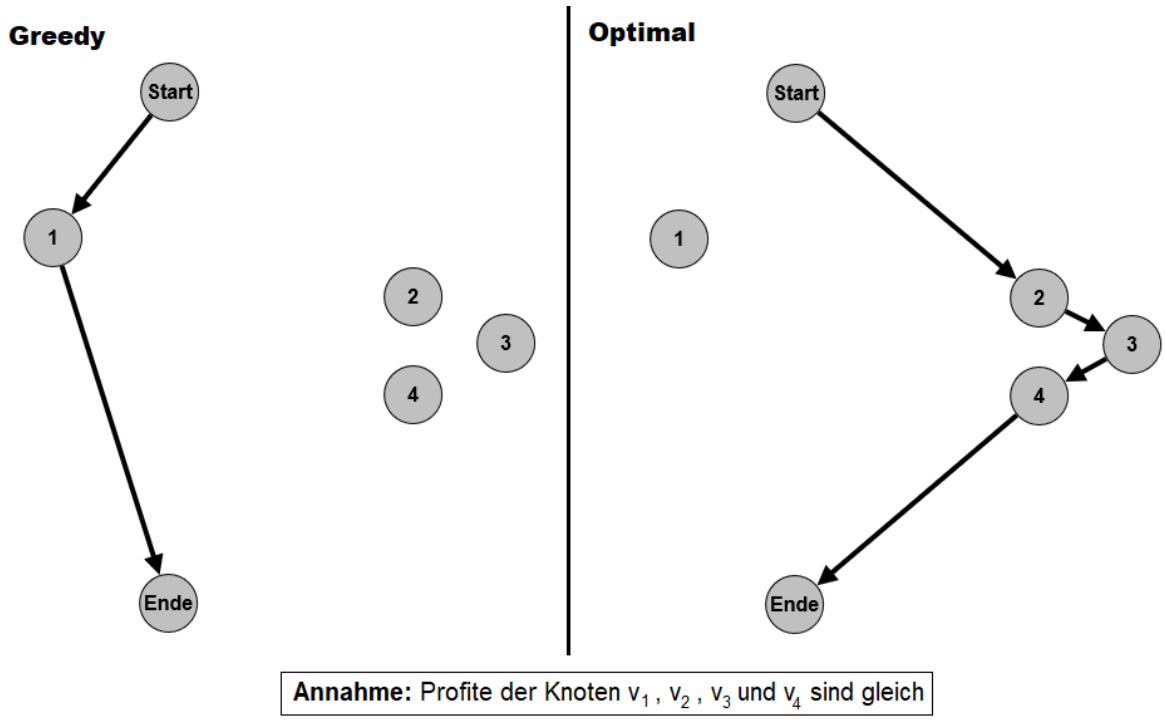


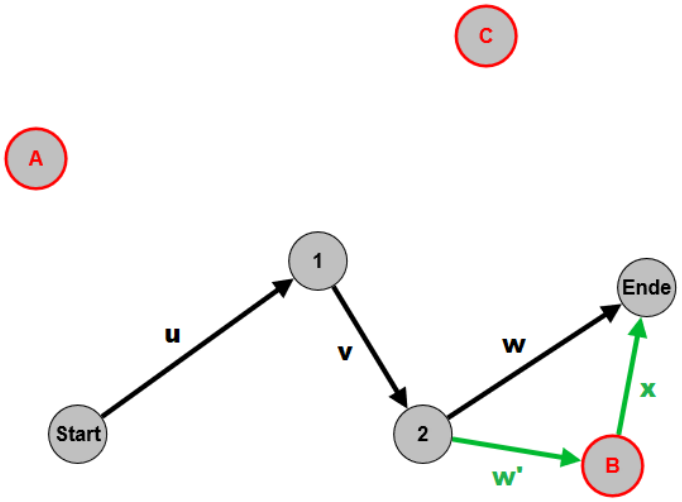
Abbildung 7: Nachteil des Greedy Repair Algorithmus

Zu Beginn des *Greedy Repair* Algorithmus werden alle möglichen Kombinationen (v_i, e) aufgelistet. Für jede Kombination wird außerdem das Verhältnis der durch den Besuch des Knotens v_i entstehenden Mehrkosten zum erzielten Profit berechnet. Für das Einfügen des Knotens v_i in die Kante (v_m, v_n) berechnet sich das Verhältnis r also durch

$$r(v_i, (v_m, v_n)) = \frac{Distanz(v_m, v_i) \; + \; Distanz(v_i, v_n) - Distanz(v_m, v_n)}{Profit(v_i)}$$

Aus der Liste wird nun die Kombination mit dem geringsten Verhältnis ausgewählt, wobei diese Kombination nicht zur Überschreitung der Kostenobergrenze T_{max} führen darf. Nachdem der Knoten v_i in die Kante e eingefügt wurde, muss die Liste für die nächste Iteration aktualisiert werden. Da sich jedoch nur die zur Kante e adjazenten Werte durch das Einfügen des Knotens verändern, reicht es aus, lediglich diese Werte statt der gesamten Liste zu aktualisieren (vgl. Santini 2019: 10).

Abbildung 8 zeigt beispielhaft eine Iteration des *Greedy Repair* Algorithmus. Im Beispiel wird von einer Tour $(v_{Start}, v_1, v_2, v_{ende})$ ausgegangen. Die Knoten v_A, v_B und v_C sind aktuell nicht in der Tour enthalten und stehen daher zu Auswahl. Daher wird für jeden dieser Knoten das Verhältnis r zu allen Kanten der Tour berechnet und in einer Liste gespeichert. Zur Vereinfachung des Beispiels werden die drei zu Auswahl stehenden Knoten mit dem gleichen Profitwert versehen und die Kostenobergrenze entsprechend so gewählt, dass sie durch das Einfügen aller Knoten nicht überschritten werden kann. Unter diesen Annahmen wird das geringste Verhältnis r vom Knoten v_B und der Kante (v_2, v_{ende}) erbracht, da diese Kombination die geringsten Mehrkosten verursacht. Diese Kante wird daher durch die zwei neuen Kanten (v_2, v_B) und (v_B, v_{ende}) ersetzt und die adjazenten Einträge der Liste für die nächste Iteration aktualisiert, wie in Abbildung 8 erkennbar ist.



1. Iteration (vorher)	1. Iteration (nachher)
r(A, u) = 4	r(A, u) = 4
r(A, v) = 7	r(A, v) = 7
r(A, w) = 13	r(A, w') = 15
	r(A, x) = 17
r(B, u) = 11	
r(B, v) = 6	r(C, u) = 17
r(B, w) = 2	r(C, v) = 15
	r(C, w') = 15
r(C, u) = 17	r(C, x) = 14
r(C, v) = 15	
r(C, w) = 13	

Abbildung 8: Beispielhafte Anwendung der *Greedy Repair* Methode

4.2.4.4 Random Cluster Repair

Die *Random Cluster Repair* Methode zielt analog zum *Random Cluster Remove* Verfahren auf die örtliche Nähe hinzuzufügender Knoten ab. In dieser Methode wird zunächst ein potentiell unvollständiges Clustering auf einen Graphen ausgeführt. Von den entstandenen Clustern wird nun im Zufallsverfahren ein Cluster ausgewählt, dessen Knoten alle zur Tour hinzugefügt werden, auch wenn die Kostenobergrenze T_{max} dabei überschritten wird. Knoten des Clusters, die bereits vor Ausführung der Methode in der Tour enthalten waren, werden dabei ignoriert, um sie nicht mehrfach in die Tour aufzunehmen. Nach Abschluss der Methode wird die Tour dem in Abschnitt 4.2.5 beschriebenen Verfahren zugeführt, um sicherzustellen, dass die Kostenobergrenze T_{max} nicht überschritten wird (vgl. Santini 2019: 11).

4.2.5 Wiederherstellung der Zulässigkeit einer Lösung

Abgesehen von der *Greedy Repair* Methode kann durch die Anwendung jeder Reparaturmethode eine Lösung erzeugt werden, deren Kosten die Kostenobergrenze T_{max} überschreiten. Um die Zulässigkeit einer ungültigen Lösung wiederherzustellen, werden aus ihr so lange Knoten entfernt, bis ihre Kosten nicht mehr T_{max} überschreiten. Ähnlich zum *Greedy Repair* Verfahren werden alle Knoten der Tour betrachtet und nach dem Verhältnis ihrer Kosten und ihrem Profit bewertet. Durch das Entfernen des Knotens v_i aus der Tour $(v_{Start}, v_1, \dots, v_h, v_i, v_j, \dots, v_{Ende})$

werden die beiden Kanten (v_h, v_i) und (v_i, v_j) durch die Kante (v_h, v_j) ersetzt und der Profit des Knotens v_i vom erzielten Gesamtprofit subtrahiert. Außer für den Fall, dass der Knoten v_i auf der Kante (v_h, v_j) liegt, würde das Entfernen des Knotens immer zu einer echten Kostenreduktion führen. Das Verhältnis s der Kostenreduktion zum verlorenen Profit berechnet sich wie folgt:

$$s(v_i) = \frac{Distanz(v_h, v_i) \; + \; Distanz(v_i, v_j) - Distanz(v_h, v_j)}{Profit(v_i)}$$

Es werden sukzessive die Knoten mit den größten Verhältnissen s aus der Tour entfernt, bis die Gesamtkosten der Tour die Kostenobergrenze T_{max} nicht mehr überschreiten (vgl. Santini 2019: 12).

4.2.6 Lokale Suchmethode

Wie bereits im Abschnitt 4.2 beschrieben und in Zeile 10 des Pseudocode in Abbildung 3 dargestellt, kann eine Lösung optional mit einer lokalen Suchmethode optimiert werden. Zwar kann diese Methode nach jeder Iteration auf die aktuelle Lösung angewendet werden, jedoch schlägt Santini (2019: 8) wegen ihrem erhöhten Rechenaufwand vor, sie nur dann anzuwenden, wenn eine neue global-beste Lösung x^* gefunden wurde.

Eine mögliche Umsetzung einer Suchmethode ist die sogenannte *Fill*-Methode, bei der einer Tour mithilfe des *Greedy Repair* Verfahrens solange Knoten hinzugefügt werden, bis die Kostenobergrenze T_{max} erreicht wird. Diese Methode bietet sich jedoch nur dann an, wenn die letzte auf die Lösung angewandte Reparaturmethode von der *Greedy Repair* Methode verschieden ist. Eine weitere Möglichkeit ist die Anwendung und heuristische Lösung des Problems des Handlungsreisenden (*TSP*). Anschließend wird das *Fill*-Verfahren eingesetzt, um weitere Knoten in die Tour aufzunehmen, weshalb das Verfahren den Namen *TSPFill* trägt (vgl. Santini 2019: 12).

Die letzte von Santini (2019) beschriebene und in dieser Arbeit umgesetzte Suchmethode ist das sogenannte *2OptFill*-Verfahren. Hierbei wird die zunächst die von Croes (1958) beschriebene *2-opt*-Methode zur Lösung des TSP angewendet, indem Kreuzungen in der Tour aufgelöst werden. Anschließend wird auch hier die Fill-Methode eingesetzt.

Die *2-opt*-Methode nimmt eine Tour $(v_{Start}, \dots, v_i, \dots, v_x, \dots, v_{Ende})$ entgegen und unterteilt die Tour in die drei Abschnitte $(v_{Start}, \dots, v_{i-1})$, $(v_i, \dots, v_x)$ und $(v_{x+1}, \dots, v_{Ende})$. Anschließend wird der zweite Abschnitt bzw. die zweite Kantenfolge umgekehrt, die Tour wieder zusammengesetzt und geprüft, ob eine Kostenreduktion erzielt wurde. Falls eine kürzere Tour gefunden wurde, wird sie als neue Tour übernommen. Die *2-opt*-Methode wird dann zurückgesetzt und auf die neue Tour angewendet. Die Methode terminiert, wenn alle möglichen Kombinationen aus v_i und v_x geprüft wurden und dabei keine Kostenreduktion mehr erzielt werden konnte (vgl. Croes 1958: 793 ff.).

In Abbildung 9 wird die Anwendung der *2-opt*-Methode beispielhaft illustriert. Die Tour $(v_{Start}, v_1, v_5, v_4, v_3, v_2, v_6, v_{Ende})$ wurde an den Knoten v_5 und v_2 in die Unterabschnitte $(v_{Start}, v_1), (v_5, v_4, v_3, v_2)$ und (v_6, v_{Ende}) unterteilt. Die untere Tour in der Abbildung ist die kürzere, kreuzungsfreie Tour dargestellt, die nach der Umkehrung des zweiten Abschnitts entsteht.

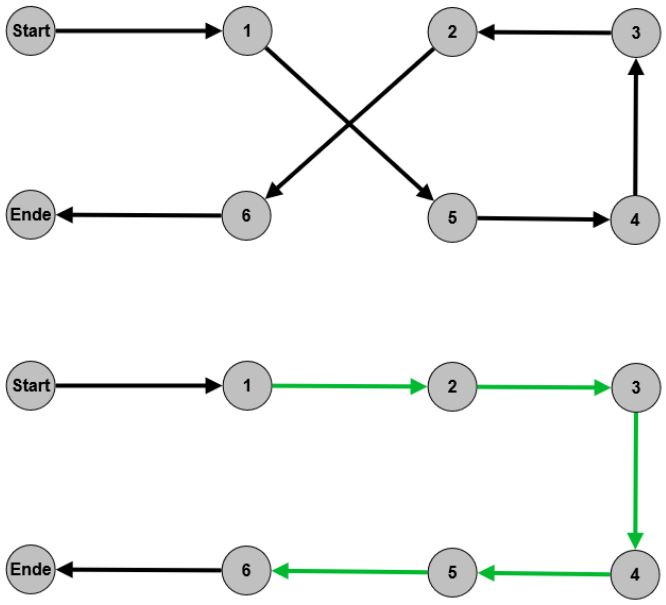


Abbildung 9: Beispielhafte Anwendung der 2-opt-Methode

4.2.7 Akzeptanzkriterium

Die Lösungen, die im ALNS Algorithmus in jeder Iteration durch Anwendung einer Zerstör- und Reparaturmethode entstehen, werden anhand ihres erzielten Profits bewertet. Eine Lösung x' wird nur dann in die nächste Iteration übernommen, wenn sie besser als die global-beste Lösung x^* ist oder wenn sie zwar schlechter als die Lösung x^* ist und folgende Bedingung erfüllt:

$$\frac{f(x^*) - f(x')}{f(x^*)} < T$$

Der Wert T beschreibt einen Grenzwert, der sich mit jeder Iteration des ALNS Algorithmus verringert. In dieser Arbeit wird der Grenzwert mit einem Maximalwert $T_0 = 1$ initialisiert und iterativ um $\frac{T_0}{M}$ bzw. $\frac{1}{Iterationen}$ reduziert, sodass sich eine lineare Veränderung des Grenzwertes gegen 0 ergibt. Laut dieser Bedingung kann eine Lösung x', die schlechter als x^* ist, nur dann akzeptiert werden, wenn die Differenz zur Lösung x^* klein genug ist. Je weiter fortgeschritten der ALNS Algorithmus ist bzw. je mehr Iterationen bereits durchlaufen wurden, desto geringer darf diese Differenz sein, damit x' noch akzeptiert werden kann (vgl. Santini 2019: 8 f.).

4.2.8 Bewertung der Zerstör- und Reparaturmethoden

Abhängig von der Anordnung der Knoten eines Graphen und der Ausprägung der aktuellen Tour können einige Zerstör- oder Reparaturmethoden bessere Lösungen liefern als andere. Aus diesem Grund werden die in einer Iteration angewendeten Methoden entsprechend der Qualität der von ihnen erzeugten Lösung bewertet, wenn diese Lösung das Akzeptanzkriterium erfüllt. Die Bewertungen beeinflussen die Wahrscheinlichkeit, mit der eine Methode in den nächsten Iterationen wiedergewählt werden kann (vgl. Santini 2019: 9).

Wie in den Zeilen 4 und 5 des Pseudocodes in Abbildung 3 beschrieben wird, erhalten alle Methoden $\mathbb{F} \in \mathcal{D} \cup \mathcal{R}$ als Bewertung initial den Wert 1. Die Aktualisierung einer Bewertung $\lambda_{\mathbb{F}}$ berechnet sich wie folgt:

$$\lambda_{\mathbb{F}} = h \cdot \lambda_{\mathbb{F}} + (1 - h) \cdot \varphi$$

Der sogenannte Decay-Faktor $h \in \mathbb{R}: 0 \leq h \leq 1$, der vom Benutzer gewählt wird, beeinflusst hierbei, wie stark die vorangegangenen Bewertungen berücksichtigt werden. Wird der Decay-Faktor auf 1 gesetzt, führt dies dazu, dass die vorangegangenen Bewertungen vollständig übernommen werden und die Bewertung der aktuellen Iteration verworfen wird. In diesem Extremfall würden sich die Bewertungen nach ihrer Initialisierung nicht mehr verändern. Mit einem Decay-Faktor von 0 vergisst der Algorithmus hingegen alle vorherigen Bewertungen und übernimmt vollständig die Bewertung der aktuellen Iteration (vgl. Santini 2019: 9).

Die Bewertung der in der aktuellen Iteration verwendeten Zerstör- und Reparaturmethoden wird durch den Wert $\varphi \in \mathbb{R}$ beschrieben. Wurde in einer Iteration eine neue global-beste Lösung gefunden, so werden die verwendeten Methoden mit der höchsten Bewertung ω_1 belohnt. Wenn stattdessen eine Lösung gefunden wurde, die besser als die vorangegangen Lösung ist, so wird sie mit ω_2 bewertet. Falls auch dies nicht zutrifft und die Lösung dennoch akzeptiert wurde (s. Abschnitt 4.2.7), so wird sie mit ω_3 bewertet. Die Verhältnisse der Bewertungen werden in dieser Arbeit durch $\omega_1 \geq \omega_2 \geq \omega_3 \geq 0$ beschrieben und die Werte konkret auf $\omega_1 = 15$, $\omega_2 = 5$ und $\omega_3 = 3$ festgelegt. Die Methoden werden nicht bewertet, wenn die von ihnen erzeugte Lösung nicht akzeptiert wurde (vgl. Santini 2019: 9).

4.3 Evolutionärer Algorithmus

Der von Kobeaga et al. (2018) vorgestellte heuristische Algorithmus zur Lösung des Orientierungsproblems lehnt sich an natürlichen Prozessen bzw. genetischen Operationen an. Die grundlegende Idee des evolutionären Algorithmus besteht darin, gute Lösungen miteinander zu kreuzen und zu mutieren, um ihre markanten Merkmale zu vereinen und neue Lösungen zu erzeugen. Die durch das Kreuzen entstandene Lösung soll die besten Eigenschaften ihrer Eltern-Lösungen übernehmen. Mithilfe der Mutation wird anschließend versucht, neue vorteilhafte Charakteristika zu finden und zu integrieren.

Der evolutionäre Algorithmus, dessen Pseudocode in Abbildung 10 aufgeführt ist, nimmt neben der Instanz des Orientierungsproblems vier weitere Parameter zur Konfiguration entgegen. Die Populationsgröße $npop$ beschreibt die Anzahl der zu beobachtenden Lösungen. In der ersten Zeile des Pseudocodes wird eine initiale Population aufgebaut, indem $npop$ Lösungen zufällig erzeugt werden. Wie in Abschnitt 4.3.1 beschrieben, beeinflusst der zweite Parameter p hierbei die Wahrscheinlichkeit, mit der ein Knoten in eine initiale Lösung aufgenommen wird. Der Hauptteil des Algorithmus, der sich über die Zeilen 6 bis 20 des Pseudocodes erstreckt, gliedert sich in zwei Blöcke, wobei in einer Iteration nur einer der beiden Blöcke ausgeführt wird. Während im ersten Block genetische Methoden auf die Population angewendet werden, enthält der zweite Block Methoden zur Optimierung der Population. Der Parameter $d2d$ definiert hierbei, wie häufig der zweite Block ausgeführt wird. Je größer $d2d$ gewählt wird, desto seltener wird der Optimierungsblock durchlaufen. Der letzte zu wählende Parameter $ncand$ beschreibt die Größe der Kandidatenliste, aus der zwei Eltern-Lösungen für den Kreuzungsoperator gewählt werden können (s. Abschnitt 4.3.5.1). Da die Kandidaten eine Untermenge der gesamten Population sind, darf der Wert nicht größer als die Populationsgröße $npop$ sein (vgl. Kobeaga et al. 2018: 2 f.).

Wie bereits beschrieben beginnt der evolutionäre Algorithmus mit der Erzeugung einer Population bestehend aus $ncand$ zufälligen Lösungen (s. Abschnitt 4.3.1). Um die Kosten der Lösungen zu reduzieren, werden sie mithilfe einer lokalen Suchmethode optimiert (s. Abschnitt 4.3.2). Trotz dieser Optimierung ist es möglich, dass die Lösungen die Kostenobergrenze T_{max} überschreiten, weshalb mithilfe des *Drop*-Operators solange Knoten aus den Lösungen entfernt werden, bis sie die Rahmenbedingung nicht mehr verletzen (s. Abschnitt 4.3.3). Mithilfe des *Add*-Operators wird anschließend versucht, den Touren, welche die Kostenobergrenze noch nicht erreicht haben, weitere Knoten hinzuzufügen (s. Abschnitt 4.3.4) (vgl. Kobeaga et al. 2018: 3).

Nach der Initialisierungsphase wird in der While-Schleife abhängig von der Iteration und der Wahl des Parameters $d2d$ entweder der Block mit genetischen Methoden (Zeile 9 bis 14) oder der Block mit Optimierungsmethoden (Zeile 16 bis 18) ausgeführt. In Letzterem werden bis auf die Erzeugung einer initialen Population alle Methoden aus der Initialisierungsphase wiederholt, um die veränderte Population zu optimieren (vgl. Kobeaga et al. 2018: 3).

Im Block mit den genetischen Methoden werden zunächst aus $ncand$ zufällig gewählten Kandidaten zwei sogenannte Eltern-Lösungen ausgewählt, wie in Abschnitt 4.3.5.1 beschrieben wird. Die gewählten Eltern-Lösungen werden mithilfe der in Abschnitt 4.3.5.2 erläuterten *Crossover*-Methode gekreuzt, sodass eine Kind-Lösung entsteht, welche wesentliche Merkmale der Eltern-Lösungen vereint. Nach der Mutation der Kind-Lösung (s. Abschnitt 4.3.5.3) wird abschließend im genetischen Block geprüft, ob die neu erzeugte Lösung besser als die schlechteste Lösung in der Population ist. In diesem Fall wird sie durch die neue Kind-Lösung ersetzt; andernfalls wird die erzeugte Lösung verworfen. Beim Block mit den genetischen Methoden ist zu beachten, dass die hier entstehenden Lösungen die Kostenobergrenze T_{max} überschreiten dürfen, da der evolutionäre Algorithmus immer mit der Ausführung des Optimierungsblocks endet, in welchem die Zulässigkeit der Lösungen wiederhergestellt wird (vgl. Kobeaga et al. 2018: 3).

Die While-Schleife wird solange wiederholt, bis sich ein gewisser Anteil der Population hinsichtlich ihrer Güte der besten Lösung angleicht (s. Abschnitt 4.3.6) und der Optimierungsblock zuletzt ausgeführt wurde. In diesem Fall wird die beste Lösung der Population zurückgegeben und der Algorithmus terminiert. Für eine bessere Vergleichbarkeit der Algorithmen kann auch eine zeitliche Obergrenze für die Laufzeit festgelegt werden. In diesem Fall würde der Algorithmus dann terminieren, wenn entweder das Stopp-Kriterium erfüllt ist oder das Zeitlimit erreicht wurde. In jedem Fall wird der Optimierungsblock als letzter Block ausgeführt, um die Gültigkeit der Lösungen zu gewährleisten (vgl. Kobeaga et al. 2018: 3).

```
// Instanz des Orientierungsproblems
EINGABE: Graph G mit G = (V,E,c,s), |V| = n ≥ 2, E = ø
EINGABE: Start- und Endknoten des Graphen v_0 und v_1
EINGABE: Kostenvektor c = (c_{i,j})_{i,j=1}^n, c_{i,j} ≥ 0
EINGABE: Profitvektor s = (s_i)_{i=1}^n, s_i ≥ 0
EINGABE: Kostenobergrenze T_max

// Parameter des evolutionären Algorithmus
EINGABE: Populationsgröße npop ∈ ℕ, npop ≥ 1
EINGABE: Wahrscheinlichkeit p ∈ ℝ, 0 ≥ p ≥ 1
EINGABE: Optimierungshäufigkeit d2d ∈ ℕ, d2d ≥ 2
EINGABE: Anzahl der Kandidaten ncand ∈ ℕ,ncand ≥ 1, ncand ≤ npop

AUSGABE: Heuristische Lösung G'

   1     Erzeuge initiale Population
   2     Optimiere Lösungen der Population
   3     Drop-Operator
   4     Add-Operator
   5     i = 0
   6     WHILE NOT (Stopp-Kriterium erfüllt AND mod(i,d2d) = 0) DO
   7        i = i + 1
   8        IF mod(i,d2d) ≠ 0 THEN
   9              Wähle zwei Eltern-Lösungen
   10             Kreuze Eltern-Lösungen
   11             Mutiere Kind-Lösung
   12             IF Kind-Lösung ist besser als schlechteste Lösung in
         Population THEN
   13                   Ersetze schlechteste Lösung in Population durch
         Kind-Lösung
   14             END IF
   15       ELSE
   16             Optimiere Lösungen der Population
   17             Drop-Operator
   18             Add-Operator
   19       END IF
   20    END WHILE
   21    RETURN G' = Beste Lösung in der Population
```

Abbildung 10: Pseudocode des evolutionären Algorithmus (Kobeaga et al. 2018: 3, modifiziert)

4.3.1 Erzeugung der initialen Population

Die im Algorithmus betrachtete Population setzt sich aus $npop$ Lösungen zusammen, die zu Beginn des Algorithmus zufällig erzeugt werden. Dabei wird einer Tour zunächst der Start- und Endknoten hinzugefügt. Die übrigen unbesuchten Knoten werden dann gemischt und nacheinander mit einer Wahrscheinlichkeit p in eine zufällige Stelle der Tour integriert. Bei einer Knotenmenge V mit $|V| = n$ werden im Schnitt also $n \cdot p$ Knoten in eine initiale Lösung aufgenommen. Die so erzeugten Lösungen dürfen die Kostenobergrenze T_{max} überschreiten (vgl. Kobeaga et al. 2018: 4).

4.3.2 Optimierung der Population

Wenn die Touren zweier Lösungen x_1 und x_2 die gleiche Knotenmenge $V' \subseteq V$ besuchen, müssen sie nicht zwangsweise die gleichen Kosten aufweisen. Neben der Wahl der zu besuchenden Knoten hat nämlich auch die Reihenfolge, in der die Knoten besucht werden, maßgeblichen Einfluss auf die Reisekosten, wie Abbildung 11 illustriert. Die Optimierung einer Route senkt die Reisekosten und kann die Aufnahme weiterer Knoten in die Tour im späteren Verlauf des Algorithmus ermöglichen. Außerdem kann sich die Optimierung einer Route auch auf ihre Gültigkeit auswirken. So ist es möglich, dass die Kosten der Lösung x_1 unter T_{max} liegen, während die Lösung x_2 diese Rahmenbedingung verletzt, obwohl beide Lösungen die gleichen Knoten besuchen (vgl. Kobeaga et al. 2018: 8 f.).

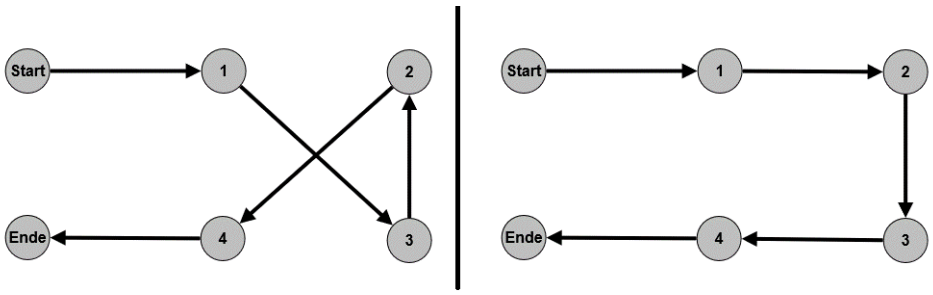


Abbildung 11: Gegenüberstellung einer nicht-optimierten und optimierten Route

Um die Lösungen zu optimieren wird die für den ALNS Algorithmus verwendete und im Abschnitt 4.2.6 beschriebene 2-opt-Methode von Croes (1958) als lokale Suchmethode verwendet.

4.3.3 Drop-Operator

Trotz der im Abschnitt 4.3.2 beschriebenen Optimierung der Lösungen kann es möglich sein, dass die Lösungen der Population die Kostenobergrenze T_{max} überschreiten. Daher werden mit dem *Drop*-Operator solange die ungünstigsten Knoten aus der Tour entfernt, bis die Kosten der Tour unter die Kostenobergrenze fallen (vgl. Kobeaga et al. 2018: 9).

Der *Drop*-Operator funktioniert analog zu der im Abschnitt 4.2.5 beschriebenen Methode zur Wiederherstellung der Zulässigkeit einer Tour. Jeder Knoten erhält eine Bewertung, die von seinem Profit und seinen verursachten Mehrkosten abhängt. Es werden sukzessive die Knoten mit der schlechtesten Bewertung entfernt. Der *Drop*-Operator terminiert, sobald die Kosten einer Tour kleiner oder gleich T_{max} sind oder wenn keine weiteren Knoten mehr entfernt werden können. Die Start- und Endknoten der Tour dürfen nicht entfernt werden und bleiben daher von diesem Operator unberührt (vgl. Kobeaga et al. 2018: 9).

4.3.4 Add-Operator

Um den Profit einer Tour zu erhöhen, werden nach der Optimierung der Routen und Anwendung des *Drop*-Operators mithilfe des *Add*-Operators weitere Knoten in die Tour aufgenommen, wenn dadurch die Kostenobergrenze nicht überschritten wird (vgl. Kobeaga et al. 2018: 9-11).

Die Wahl des nächsten hinzuzufügenden Knotens v_i richtet sich nach seinem Mehrwert, der sich wiederum über den Quotienten seines Profits s_i zu seinen verursachten Mehrkosten berechnet, also:

$$mehrwert(v_i) = \frac{s_i}{mehrkosten(v_i)}$$

Um die Mehrkosten zu berechnen, die durch das Hinzufügen des Knotens v_i in die Tour entstehen, werden zunächst die drei nächstgelegenen Knoten v_k, v_l und v_m gesucht, die bereits in der Tour enthalten sind. Zur Ermittlung der Mehrkosten treffen Kobeaga et al. (2018: 10 f.) folgende Fallunterscheidung, die in Abbildung 12 veranschaulicht wird:

1. Maximal zwei der drei nächstgelegenen Knoten sind durch eine Kante in der Tour verbunden. In diesem Fall beschreiben die Mehrkosten jene Kosten, die durch das Einsetzen des Knotens v_i in diese Kante entstehen. Für die Kante (v_k, v_l) gilt somit:

$$mehrkosten(v_i) = Distanz(v_k, v_i) + Distanz(v_i, v_l) - Distanz(v_k, v_l)$$

2. Ausgehend von drei Knotenpaaren, die aus den drei nächstgelegenen Knoten v_k, v_l und v_m gebildet werden, sind mindestens zwei der Knotenpaare jeweils durch eine Kante direkt verbunden. In der Regel dürfen nur zwei der Knotenpaare direkt verbunden sein, da andernfalls ein Kreis in der Tour entstehen würde. Da ein profitbehafteter Knoten jedoch nicht mehrmals besucht werden darf, wäre dies nicht zulässig. Nur für den Spezialfall, dass es sich bei einem der drei nächstgelegenen Knoten um den Start- und Endknoten der Tour handelt, der folglich zweimal in der gesamten Tour enthalten sein darf, kann es hier drei Kanten geben. In beiden Fällen wird die Kante ausgewählt, bei der das Einsetzen des Knotens v_i die geringsten Mehrkosten verursacht.

3. Keiner der drei nächstgelegenen Knoten ist direkt durch eine Kante in der Tour verbunden. In diesem Fall wird geprüft, in welchen der maximal sechs Kanten, die mit den drei nächstgelegenen Knoten verbunden sind, der Knoten v_i mit den geringsten Mehrkosten eingesetzt werden kann. Wenn sich der Start- und/oder Endknoten unter den nächstgelegenen Knoten befinden, reduziert sich die Anzahl der zu prüfenden Kanten.

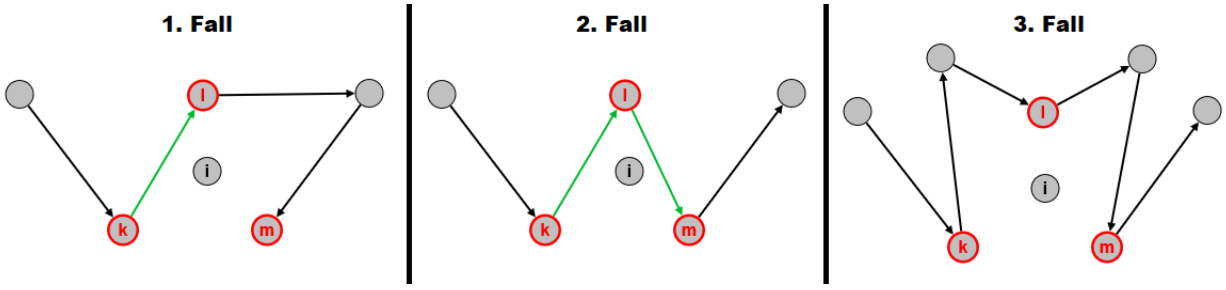


Abbildung 12: Fallunterscheidung zur Berechnung der Mehrkosten für den *Add*-Operator

Nach der Berechnung der Mehrkosten für alle unbesuchten Knoten v_i werden die Mehrwerte wie eingangs beschrieben bestimmt. Dabei ist zu beachten, dass der Mehrwert eines Knotens auf 0 gesetzt wird, wenn durch seine Aufnahme in die Tour die Kostenobergrenze T_{max} überschritten würde. Die Knoten werden anschließend gemäß ihrem Mehrwert absteigend sortiert und sukzessive in die Tour aufgenommen. Der *Add*-Operator terminiert, wenn keine unbesuchten Knoten mehr existieren oder wenn kein Knoten mehr hinzugefügt werden kann, ohne dabei die Kostenobergrenze zu überschreiten (vgl. Kobeaga et al. 2018: 10 f.).

4.3.5 Genetische Methoden

Mithilfe der genetischen Methoden *Select*, *Crossover* und *Mutate* sollen ausgehend von der bestehenden Population neue Lösungen erzeugt werden. Wie im Pseudocode in Abbildung 10 dargestellt, werden die Methoden bei $k \in \mathbb{N}^+$ Iterationen insgesamt $k \cdot (d2d - 1)$ mal ausgeführt.

4.3.5.1 Select: Wahl zweier Eltern-Lösungen

Ziel dieser Methode ist die Auswahl zweier Lösungen aus der Population, um sie anschließend miteinander zu kreuzen und dadurch eine neue Lösung zu erzeugen. Als Kandidaten werden hierfür zunächst $ncand$ Lösungen zufällig aus der aktuellen Population gewählt. Anschließend wird der Fitnesswert der schlechtesten Lösung $f_{min} = \min(f(x_i))$ bestimmt. Der Fitnesswert wird durch die Fitnessfunktion $f(\cdot)$ bestimmt und berechnet sich, wie in Abschnitt 4.1 erläutert, aus dem erzielen Profit einer Lösung dividiert durch die Summe der Profite aller Knoten des Graphen. Jeder Kandidat x_i wird anschließend mit einer Bewertung $r_i = f(x_i) - f_{min} + 1$ versehen. Mit dieser Bewertung wird für jeden Kandidaten schließlich die Wahrscheinlichkeit $p_i = \frac{r_i}{\sum r_i}$ berechnet, mit der dieser als Eltern-Lösung ausgewählt werden kann. Demnach werden die beiden Eltern-Lösungen per gewichtetem Zufall aus der Kandidatenliste gezogen, wobei bessere Lösungen eine entsprechend größere Chance auf eine Auswahl haben (vgl. Kobeaga et al. 2018: 5).

4.3.5.2 Crossover: Kreuzung der Eltern-Lösungen

Mit der *Crossover* Methode wird durch Kreuzung zweier Eltern-Lösungen eine neue Kind-Lösung erzeugt. Bei der Kreuzung handelt es sich um die Übernahme wesentlicher Merkmale der Eltern, um möglichst viele Informationen in der neuen Lösung zu vereinen. Diese Methode baut auf der Annahme auf, dass durch die Kreuzung von Lösungen mit hoher Güte eine Kind-Lösung erzeugt wird, die ebenfalls eine hohe Güte aufweist (vgl. Kobeaga et al. 2018: 5).

Zu den wesentlichen Merkmalen zählen sowohl die von den Eltern-Lösungen besuchten Knoten als auch ihre verwendeten Pfade. So soll die Kind-Lösung in jedem Fall die von beiden Eltern besuchten Knoten enthalten. Knoten, die hingegen nur von einer Eltern-Lösung besucht werden, sollen mit einer gewissen Wahrscheinlichkeit in der neuen Lösung aufgenommen werden. Nicht berücksichtigt werden hingegen Knoten, die in keiner der beiden Eltern-Lösungen besucht werden. Weiterhin sollen die von der neuen Lösung zu besuchenden Knoten mithilfe der von den Eltern-Lösungen verwendeten Kanten in der Tour aufgenommen werden (vgl. Kobeaga et al. 2018: 5).

Der von Whitley et al. (1989) vorgestellte *Crossover* Algorithmus ist für Hamiltonkreise (s. Abschnitt 2.1.3) konzipiert und wurde von Kobeaga et al. (2018) so erweitert, dass er auch auf einfache Kreise angewendet werden kann, die nicht alle Knoten der Knotenmenge V enthalten müssen. Da in dieser Arbeit die Start- und Endknoten nicht identisch sein müssen, wurde der Algorithmus so modifiziert, dass er nun auch auf Pfade angewendet werden kann.

Die *Crossover* Methode beginnt zunächst mit der Erzeugung einer sogenannten *Edge Map*, welche die wesentlichen Informationen der in Abbildung 13 gezeigten Eltern-Graphen zusammenfasst. Die zu den Eltern-Graphen zugehörige *Edge Map* wird in Tabelle 3 dargestellt. Die von beiden Lösungen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ besuchten Knoten v werden als **gemeinsame Knoten** (in der Abbildung rot dargestellt) bezeichnet. Jeder dieser Knoten v erhält einen Eintrag in der *Edge Map*. Bei den sogenannten **verbundenen Knoten** u handelt es sich auch um *gemeinsame Knoten*, zu denen ein Pfad vom Knoten v besteht, der keine weiteren

gemeinsamen Knoten enthält. *Verbundene Knoten* sind also *gemeinsame Knoten*, die mit dem Knoten v über eine Kante direkt oder ausschließlich über nicht-gemeinsame Knoten verbunden sind. Der **Grad** eines *gemeinsamen Knotens* definiert sich über die Anzahl der mit ihm *verbundenen Knoten*. Bei den **zwischenliegenden Pfaden** handelt es sich um die Pfade zwischen den Knoten v und u, welche außer den beiden Knoten keine weiteren *gemeinsamen Knoten* enthalten. Während der Erzeugung der *Edge Map* ist zu beachten, dass die gerichteten Kanten vorübergehend als ungerichtete Kanten interpretiert werden und daher in beide Richtungen traversiert werden dürfen (vgl. Kobeaga et al. 2018: 5).

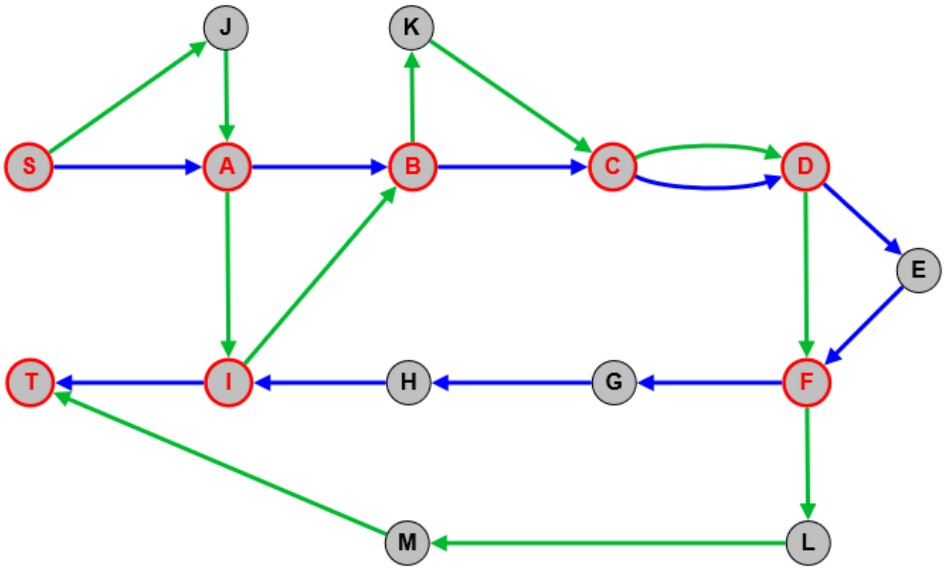


Abbildung 13: Beispiel zweier zu kreuzender Eltern-Lösungen

Gemeinsamer Knoten	Verbundene Knoten	Grad	Zwischenliegende Pfade
S	*A*	1	*(S, A), (S, J, A)*
A	*S* *B* *I*	3	*(A, S), (A, J, S)* *(A, B)* *(A, I)*
B	*A* *I* *C*	3	*(B, A)* *(B, I)* *(B, C), (B, K, C)*
C	*B* *D*	2	*(C, B), (C, K, B)* *(C, D)*
D	*C* *F*	2	*(D, C)* *(D, E, F), (D, F)*
F	*D* *I* *T*	3	*(F, E, D), (F, D)* *(F, H, G, I)* *(F, L, M, T)*
I	*A* *B* *F* *T*	4	*(I, A)* *(I, B)* *(I, H, G, F)* *(I, T)*
T	*I* *F*	2	*(T, I)* *(T, M, L, F)*

Tabelle 3: Edge Map der Beispielgraphen

Nach der Erzeugung der *Edge Map* für beide Eltern-Lösungen wird der Algorithmus mit dem Startknoten des Graphen, der gleichzeitig ein *gemeinsamer Knoten* ist, als aktueller Knoten initialisiert. Von den mit ihm verbundenen *gemeinsamen Knoten* wird nun der Knoten mit dem geringsten Grad als Folgeknoten ausgewählt und ein entsprechender Pfad zufällig ausgewählt, falls mehrere Pfade zu diesem Knoten existieren. Auch für den Fall, dass mehrere *verbundene Knoten* den gleichen Grad besitzen, wird die Auswahl per Zufallsprinzip getroffen. Sollte im Eintrag des aktuellen Knotens kein *verbundener Knoten* existieren, der nicht bereits besucht wurde, wird zufällig einer der anderen unbesuchten *gemeinsamen Knoten* als Folgeknoten bestimmt und mit einer neuen Kante verbunden (vgl. Kobeaga et al. 2018: 5-8).

Nachdem nun ein Folgeknoten und ein entsprechender Pfad gewählt wurden, wird der aktuelle Knoten aus der *Edge Map* entfernt, indem sowohl sein eigener Eintrag als auch seine Vorkommen als *verbundener Knoten* in anderen Einträgen gelöscht werden. Der Folgeknoten wird nun als aktueller Knoten bezeichnet und die Prozedur wiederholt. Die *Crossover* Methode terminiert, wenn die *Edge Map* leer ist bzw. keine unbesuchten *gemeinsamen Knoten* mehr existieren (vgl. Kobeaga et al. 2018: 5-8).

Für ein besseres Verständnis der *Crossover* Methode werden im Folgenden zwei Iterationen anhand der in Abbildung 13 gezeigten Eltern-Graphen beschrieben, deren initiale *Edge Map* in Tabelle 3 dargestellt wird. Ein mögliches Ergebnis der *Crossover* Methode wird in Abbildung 14 illustriert.

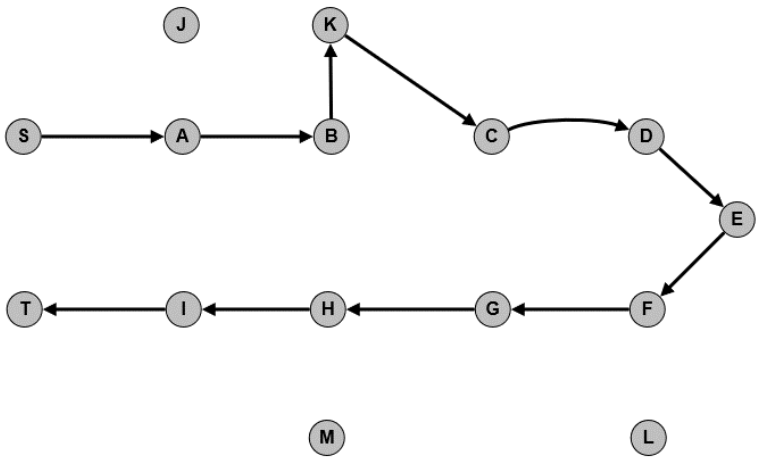


Abbildung 14: Mögliches Ergebnis der *Crossover* Methode

Beginnend vom Startknoten S kann als Folgeknoten nur der Knoten A gewählt werden. Da es zwei Pfade zwischen den Knoten S und A gibt, wird ein Pfad zufällig ausgewählt. In diesem Beispiel wird nun der Pfad (S, A) gewählt. Der Knoten S wurde nun abgearbeitet, weshalb sein Eintrag aus der *Edge Map* entfernt werden muss. Da er als *verbundener Knoten* im Eintrag des Knotens A vertreten ist, wird er auch hier entfernt und der Grad des Knotens A auf 2 reduziert. Der Knoten A wird nun zum aktuellen Knoten ernannt. Als Folgeknoten stehen nun die Knoten B und I zur Auswahl. Da der Knoten B einen geringeren Grad besitzt, muss er als Folgeknoten ausgewählt werden. Zum Knoten B existiert auch nur der Pfad (A, B) vom Knoten A, der über keine weiteren *gemeinsamen Knoten* geht. Daher wird dieser Pfad ausgewählt, alle Vorkommen des Knotens A aus der *Edge Map* entfernt und der Knoten B als Folgeknoten ernannt. Mit diesem

Vorgehen werden nun die restlichen *gemeinsamen Knoten* in die neue Tour aufgenommen. Die *Crossover* Methode terminiert, sobald alle acht *gemeinsamen Knoten* in der Tour enthalten sind.

Wird der *Crossover* Algorithmus von Kobeaga et al. (2018) auf einen Pfad anstelle eines einfachen Kreises angewendet, so kann es durch eine ungünstige Wahl der Folgeknoten dazu kommen, dass der Endknoten in die Tour aufgenommen wird, bevor die restlichen *gemeinsamen Knoten* hinzugefügt werden. Wenn im Beispiel in Abbildung 14 der Knoten T anstelle des Knotens I als Folgeknoten von F gewählt worden wäre, so könnte der Knoten I nicht mehr besucht werden, da sich sonst der Endknoten des Graphen ändern würde. Um dieses Problem zu umgehen, wird in dieser Arbeit der Endknoten von der *Crossover* Methode ausgenommen und nach Abschluss der Methode mit dem letzten hinzugefügten *gemeinsamen Knoten* durch eine Kante verbunden.

4.3.5.3 Mutate: Mutation der Kind-Lösung

Mit der *Mutate* Methode soll die zuvor erzeugte Kind-Lösung verändert werden, um das Entdecken neuer Merkmale zu ermöglichen. Die Mutation kann sowohl positive als auch negative Auswirkungen auf die Qualität einer Lösung haben.

Zur Mutation der Kind-Lösung wird zunächst ein zufälliger Knoten v aus dem Graphen bzw. der Knotenmenge V gewählt, der weder der Start- noch der Endknoten sein darf. Anschließend wird geprüft, ob dieser Knoten in der Tour der Kind-Lösung enthalten ist. In diesem Fall wird dieser Knoten aus der Tour entfernt und die zu ihm adjazenten Knoten, also sein Vorgänger und Nachfolger, miteinander verbunden. Sollte der Knoten v nicht in der Tour enthalten sein, wird er analog zum *Add*-Operator (s. Abschnitt 4.3.4) an die für ihn günstigste Stelle der Tour hinzugefügt (vgl. Kobeaga et al. 2018: 8).

4.3.6 Stopp-Kriterium

Das Stopp-Kriterium ist erfüllt, wenn ein gewisser Anteil der Population die gleiche Güte bzw. den gleichen Fitnesswert hat, wie die beste Lösung der Population. In dieser Arbeit wurde der dafür benötigte Anteil auf 25% gesetzt. Zu diesem Anteil werden auch Lösungen gezählt, deren Güte um bis zu $1{,}5\%$ von der Güte der besten Lösung abweichen, da insbesondere bei größeren Graphen identische Fitnesswerte nur nach unverhältnismäßig langer Zeit erzielt werden können (vgl. Kobeaga et al. 2018: 12).

Das Stopp-Kriterium wird am Ende des Optimierungsblocks bzw. nach der Ausführung des *Add*-Operators geprüft. Das bedeutet, dass der Algorithmus nur dann terminieren kann, wenn der Optimierungsblock als letzter Block ausgeführt wurde. Dies ist deswegen notwendig, da im Block mit den genetischen Methoden unzulässige Lösungen erzeugt werden können, die im Optimierungsblock in zulässige Lösungen umgewandelt werden. Wie eingangs im Abschnitt 4.3 beschrieben, kann neben dem Stopp-Kriterium auch ein Zeitlimit festgelegt werden, um eine bessere Vergleichbarkeit der Algorithmen zu ermöglichen. Diese zeitliche Überprüfung folgt in diesem Fall direkt nach der Überprüfung des Stopp-Kriteriums. Da der Optimierungsblock nur einmal alle $d2d$ Iterationen ausgeführt wird und der Algorithmus nur nach dessen Ausführung terminieren darf, kann die zeitliche Obergrenze nur als Richtwert verstanden werden, der vom Algorithmus für finalisierende Operationen überschritten werden darf (vgl. Kobeaga et al. 2018: 12).

5 Softwareentwicklung

In diesem Kapitel wird auf die Vorgehensweise bei der Implementierung der Algorithmen und auf die dabei verwendeten Programmiermethoden und -techniken eingegangen. Außerdem wird der generelle Workflow bei der Nutzung der Anwendungen beschrieben und der Ablauf bei der Entwicklung der Benutzerschnittstellen vorgestellt. Neben der Anwendung zur Lösung des Orientierungsproblems auf einzelne Graphen wurde auch eine Anwendung zur Durchführung von Benchmarking-Tests für größere Graphen entwickelt, welche auf dem beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) gefunden werden können.

5.1 Modellierung der Graphen

Zur Implementierung der Algorithmen von Kobeaga et al. und Santini ist es notwendig, eine entsprechende Modellierung der Graphen vorzunehmen. Von der Modellierung hängen unter anderem die Performanz der Algorithmen und der Umgang mit den Graphen während der Programmierung ab.

Wie in Abschnitt 3.2 beschrieben lässt sich das Orientierungsproblem auf gewichtete Graphen $G = (V, E, c, s)$ anwenden, bei denen die Knoten $v \in V,\ |V| \geq 2$ mit einem Profitwert und die Kanten $e \in E$ mit einem Kostengewicht versehen sind. Die Start- und Endknoten des Graphen besitzen einen Profitwert von 0 und ihre Koordinaten dürfen identisch sein. Profitbehaftete Knoten, die kontextabhängig auch Kunden genannt werden können, dürfen nicht mehrfach besucht werden. Als Lösung des Problems sind demnach nur Pfade oder einfache Kreise zulässig (s. Abschnitt 2.1.3).

Die Knoten-Objekte werden in dieser Arbeit mit ihrer x- und y-Koordinate und ihrem Profitwert parametrisiert. Eine Kante besteht wiederum aus einem Start- und Endknoten, während sich sein Kantengewicht sich aus der euklidischen Distanz zwischen diesen beiden Knoten bestimmt.

Ein Graph besteht aus einer nicht-leeren Knotenmenge und einer potentiell leeren Kantenmenge. Zur Implementierung des Graphen müssen für diese Mengen jeweils eine geeignete Datenstruktur gewählt werden. Die Wahl der Datenstruktur beeinflusst einerseits den Umgang mit den Daten während der Programmierung. So können Sortierungen mit Datenstrukturen ohne Indizierung (z. B. *HashMap*) nicht oder nur schwer vorgenommen werden. Andererseits wirkt sich die Wahl der Datenstruktur auch auf die Performanz der Anwendungen aus, da sich elementare Operationen, wie z. B. das Hinzufügen oder Entfernen von Elementen, in ihrer Komplexität wesentlich unterscheiden können.

In dieser Arbeit wird sowohl für die Knoten- als auch für die Kantenmenge die Datenstruktur *ArrayList* verwendet, welche Objekte in einer dynamischen Liste intern verwaltet. Wie Tabelle 4 zeigt, haben elementare Operationen, wie der Zugriff auf ein Objekt, eine konstante Zeitkomplexität. Die Operationen *add* und *remove* weisen hingegen eine lineare Zeitkomplexität auf. Da im Verlauf der Algorithmen häufig auf Elemente der Mengen zugegriffen wird und das Hinzufügen und Entfernen von Elementen vergleichsweise selten vorkommt, wurde die *ArrayList* der doppelt-verketteten Liste *LinkedList* vorgezogen. Andere Datenstrukturen, wie z. B. *HashMaps* wurden in dieser Arbeit nicht gewählt, da ihnen entweder elementare Eigenschaften (z. B. Indizierung) fehlen oder weil sie umständlicher in ihrer Handhabung sind (vgl. Oracle 2022).

Elementare Operation	Zeitkomplexität
$add(\ Object\ o\ )$	$\mathcal{O}(n)$
$remove(\ Object\ o\ )$	$\mathcal{O}(n)$
$get(\ int\ index\ )$	$\mathcal{O}(1)$
$set(\ int\ index, Object\ o\ )$	$\mathcal{O}(1)$
$next()$	$\mathcal{O}(1)$

Tabelle 4: Zeitkomplexität elementarer Operationen der ArrayList (vgl. Oracle 2022, modifiziert)

Zu beachten ist, dass die ersten beiden Einträge der Knotenliste für die Start- und Endknoten reserviert sind. Falls beide Knoten identisch sein sollten, müssen entsprechend zwei Knoten mit gleichen Koordinaten an den Anfang der Liste gesetzt werden. Für die Kantenliste gilt, dass die erste Kante mit dem Startknoten des Graphen beginnen und die letzte Kante mit dem Endknoten des Graphen enden muss. Weiterhin gilt, dass der Endknoten einer Kante gleichzeitig als Startknoten der Folgekante verwendet werden muss, um eine zusammenhängende Tour zu modellieren. So wird die Tour $(v_{Start}, v_1, v_2, v_{Ende})$ durch die Kanten (v_{Start}, v_1), (v_1, v_2) und (v_2, v_{Ende}) in dieser Reihenfolge in der Kantenmenge dargestellt.

Da das Einlesen von Graphen aus Dateien ermöglicht werden soll, muss für das Parsen der Dateien auch eine entsprechende Formatierung festgelegt werden. Wie in Abbildung 15 dargestellt, werden die Knoten eines Graphen zeilenweise definiert. Jede Zeile enthält die x- und y-Koordinate sowie den Profitwert des Knotens, welche jeweils durch einen Tabstopp getrennt werden müssen. Die Start- und Endknoten müssen dabei in den ersten beiden Zeilen der Graph-Datei definiert werden und jeweils einen Profitwert von 0 besitzen. Kanten dürfen an dieser Stelle nicht mit angegeben werden, da das Orientierungsproblem und damit auch die Anwendungen lediglich die Knotenmenge der vollständigen Graphen und keine vordefinierten Touren erwarten.

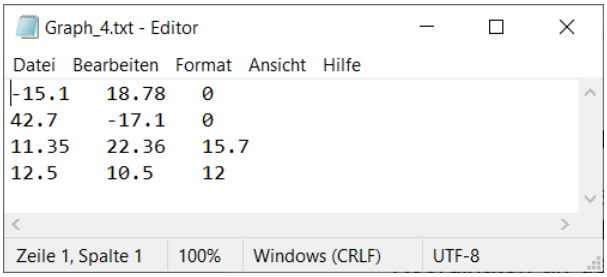


Abbildung 15: Graph-Datei eines Beispielgraphen mit vier Knoten

5.2 Programmierung der Algorithmen

In dieser Arbeit werden zur Unterstützung des Entwicklungsprozesses und zur Qualitätssicherung moderne Programmiertechniken und -methoden eingesetzt, die in diesem Abschnitt vorgestellt werden.

Beide Anwendungen werden mit der objektorientierten Programmiersprache Java SE 8 entwickelt, wobei als Entwicklungsumgebung Eclipse zum Einsatz kommt. Durch das Angebot diverser Funktionalitäten erleichtern Entwicklungsumgebungen wie Eclipse die Entwicklung von Anwendungen ungemein. Zu diesen Funktionalitäten zählen beispielsweise das Debugging, das visuelle Hervorheben von Codestellen sowie eine Autovervollständigung einzelner Codeabschnitte. Auch die Möglichkeit des automatisierten Code-Refactorings unterstützen hier den Entwicklungsprozess und die Lesbarkeit des Codes. Hierdurch können nämlich nachträgliche Änderungen am bestehenden Code auch dann einfach umgesetzt werden, wenn bereits große

Teile des Codes fertiggestellt wurden. Die Alternative hierzu wären manuelle Änderungen am Quellcode, die aufgrund des Codeumfangs zeitaufwändig und fehleranfällig sein können.

Zur Vereinfachung der Projektverwaltung und der Integration von Plugins und weiterer Java Bibliotheken wird das Projektmanagementwerkzeug Maven verwendet. Mit Maven kann der Build-Prozess konfiguriert und automatisiert werden. Die Konfiguration wird in den sogenannten POM-Dateien (*Project Object Model*) im XML-Format vorgenommen. Bei multi-modularen Projekten, zu dem das Projekt dieser Arbeit zählt, kann Maven jeweils in den einzelnen Modulen oder zentral im gemeinsamen Parent-Modul konfiguriert werden (vgl. Szczukocki 2020).

In Abbildung 16 wird ein Auszug einer POM-Datei dargestellt, in der die Abhängigkeiten zu den verwendeten Bibliotheken commons-math3, commons-lang3 sowie der JUnit Bibliothek in den entsprechenden Versionen definiert werden. Diese Bibliotheken werden zu Beginn des Build-Prozesses heruntergeladen, im entsprechenden Modul des Projekts eingebunden und können alsdann verwendet werden. Über die sogenannten *Goals* können die einzelnen Aufgaben angegeben werden, die von Maven im Build-Prozess zu erledigen sind. Für dieses Projekt werden die Ziele „*clean javadoc:javadoc install*“ verwendet. Mit „*clean“* werden im Build-Prozess bestehende Ressourcen und Class-Dateien aus vorangegangen Build-Prozessen gelöscht. Das Ziel „*javadoc:javadoc*“ gibt die Anweisung zur Ausführung des JavaDoc-Plugins an, welches im nächsten Absatz vorgestellt wird. Die Anweisung zur Kompilierung, Ausführung etwaiger Tests und zur Paketierung wird abschließend durch das Ziel „*install*“ gegeben (vgl. The Apache Software Foundation 2021).

```
<dependencies>
	<dependency>
		<groupId>org.apache.commons</groupId>
		<artifactId>commons-math3</artifactId>
		<version>3.6.1</version>
	</dependency>

	<dependency>
		<groupId>org.apache.commons</groupId>
		<artifactId>commons-lang3</artifactId>
		<version>3.12.0</version>
	</dependency>

	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.12</version>
		<scope>test</scope>
	</dependency>
</dependencies>
```

Abbildung 16: Auszug einer POM-Datei des Projekts

Die Dokumentation des Quelltextes wird in dieser Arbeit mit dem eben genannten JavaDoc-Plugin realisiert, welches auch mit Maven eingebunden und konfiguriert werden kann. Das JavaDoc-Plugin verwendet die im Code enthaltenen Kommentare, die entsprechend formatiert sein müssen, und erzeugt daraus eine Dokumentation im HTML-Format, welches auf dem beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) hinterlegt ist. In Abbildung 17 wird als Beispiel die JavaDoc-Kommentierung des ALNS-Konstruktors im

Quellcode gezeigt. Diese Beschreibungen bzw. Kommentare finden sich dann in der in Abbildung 18 gezeigten Dokumentation wieder.

```
/**
 * Konstruktor: Erzeugt eine ALNS-Instanz.
 *
 * @param g           Graph, für den eine Lösung gefunden werden soll
 * @param Tmax        Kostenobergrenze
 * @param interations Anzahl durchzuführender Iterationen
 * @param alpha       Aggressivitätsfaktor
 * @param h           Decay-Faktor
 * @param maxTime     Gibt eine maximale Laufzeit in Minuten an, falls der Wert
 *                    über 0 gewählt wird.
 */
public ALNS(Graph g, double Tmax, int interations, double alpha, double h, int maxTime) {
    this.Tmax = Tmax;
    this.alpha = alpha;
    this.h = h;
    this.iterations = interations;
    this.g = new Graph(g);
    this.maxTime = maxTime * 60000;
    this.cancelled = false;
}
```

Abbildung 17: JavaDoc-Kommentierung des ALNS-Konstruktors im Quellcode

Constructor Detail

ALNS

```
public ALNS(Graph g,
            double Tmax,
            int interations,
            double alpha,
            double h,
            int maxTime)
```

Konstruktor: Erzeugt eine ALNS-Instanz.

Parameters:

g - Graph, für den eine Lösung gefunden werden soll

Tmax - Kostenobergrenze

interations - Anzahl durchzuführender Iterationen

alpha - Aggressivitätsfaktor

h - Decay-Faktor

maxTime - Gibt eine maximale Laufzeit in Minuten an, falls der Wert über 0 gewählt wird.

Abbildung 18: Auszug der JavaDoc-Dokumentation des ALNS-Konstruktors

Ein weiteres Werkzeug, welches in dieser Arbeit zum Einsatz kommt, ist das JUnit-Framework in der Version 4.12, mit welchem das Verhalten einzelner Code-Abschnitte getestet werden kann.

JUnit wurde ursprünglich von Erich Gamma und Kent Beck entwickelt und ermöglicht das Erzeugen und Ausführen von Testfällen, mit dem das erwartete Verhalten des Programmes mit dem tatsächlichen Verhalten automatisiert und wiederholt verglichen werden kann. Auf das Testen des Programmes wird weiter in Abschnitt 5.3 eingegangen (vgl. JUnit Community 2021).

Zur automatisierten Prüfung der Testabdeckung wird das *JaCoCo*-Plugin verwendet. *JaCoCo* zeigt an, welche Codeabschnitte, wie z. B. Funktionen, nicht oder nur teilweise von JUnit-Tests abgedeckt werden. Wenn eine Funktion beispielsweise durch mehrere IF-ELSE Anweisungen verzweigt wird und die JUnit-Tests für diese Funktion nicht alle Verzweigungen erreichen, so wird dies dem Entwickler angezeigt (vgl. Baeldung 2021a).

Neben der Entwicklung von JUnit hat Herr Gamma mit der Entwicklung sogenannter Entwurfsmuster einen weiteren Beitrag zu dieser Arbeit geleistet. Im Buch „Design Patterns - Elements of Reusable Software“ von Gamma et al., welches 1994 erstmalig erschien, werden wiederkehrende Programmierprobleme -und Lösungen beschrieben. Bei den beschriebenen Entwurfsmustern handelt es sich zwar nicht um Lösungen, die ohne Weiteres kopiert und im Quelltext übernommen werden können, jedoch werden Kernaspekte der Lösungen so beschrieben, dass sie sich in das aktuelle Softwareprojekt entsprechend übertragen lassen (vgl. Gamma et al. 1995). Der Einsatz der Entwurfsmuster wird in Abschnitt 5.4 näher erläutert.

Um die Entwicklung der graphischen Benutzeroberfläche (GUI) zu unterstützen, wird zunächst das *Balsamiq Wireframes* Werkzeug verwendet, mit dem die GUI vorab gestaltet bzw. skizziert werden kann. Die tatsächliche Entwicklung bzw. Implementierung der graphischen Benutzeroberflächen wird mit dem Eclipse Plugin *WindowBuilder* umgesetzt. Der Einsatz beider Werkzeuge und die Gestaltung der GUI werden im Abschnitt 5.5 detaillierter vorgestellt.

5.3 Unit-Tests

Wie im vorangegangenen Abschnitt bereits erwähnt, wird zum Testen der Funktionalität einzelner Komponenten das Testframework JUnit 4.12 in Kombination mit dem *JaCoCo*-Plugin verwendet. Letzteres zeigt an, welche Teile des Quellcodes durch die Tests abgedeckt werden. Der Quellcode mitsamt aller JUnit-Tests ist auf dem beigefügten Datenträger und im Github-Repository (vgl. Machaka 2022) hinterlegt.

Mithilfe der Tests sollen Fehler in der Implementierung identifiziert werden, die auf dem ersten Blick nur schwer zu erkennen sind und die sich auch durch nachträgliche Veränderungen an der Implementierung einschleichen können. Falls es in den Maven Goals nicht explizit unterbunden wird, werden die Tests regelmäßig und automatisiert bei jedem Build-Prozess ausgeführt. Schlägt ein Test fehl, so wird dies dem Entwickler mit einem roten Balken und entsprechenden Verweisen zur Fehlerstelle visuell angezeigt und der Build-Prozess abgebrochen.

Ein Testfall definiert sich aus einer Eingabe (z. B. Aufruf einer Methode) und einer erwarteten Ausgabe. Weicht das tatsächliche Ergebnis von der erwarteten Ausgabe ab, so wird dies als fehlgeschlagener Test gedeutet. Zwar können mit den Testfällen auch größere Einheiten (z. B. Klassen oder Packages) getestet werden, jedoch bietet es sich wegen ihrer Komplexität und ihres Umfangs eher an, sich auf das Testen der Methoden zu fokussieren (vgl. Langr et al. 2015: 13 ff.).

Eine Herausforderung bei der Prüfung der Algorithmen in dieser Arbeit ist die Tatsache, dass stellenweise Zufallsverfahren zum Einsatz kommen, wenn beispielsweise der nächste zu löschende Knoten selektiert oder wenn aus mehreren Operationen eine Methode per gewichtetem Zufall bestimmt werden soll. Wenn jedoch die Zufallswahl abstrahiert wird und das Ergebnis anhand übergeordneter Messwerte geprüft werden kann, so können Fehler auch auf diesem Wege identifiziert werden.

Ein Beispiel für solch eine Abstraktion wird in Abbildung 19 dargestellt. Hierbei handelt es sich um einen Auszug aus den JUnit-Tests für die in Abschnitt 4.2.4.1 beschriebene *Random Repair* Methode. Unabhängig davon, welche Knoten nun tatsächlich von der *Random Repair* Methode zur Tour hinzugefügt werden, muss es sich beim Ergebnis um eine gültige Tour handeln. Das bedeutet, dass die Start- und Endknoten in der Tour an den richtigen Stellen enthalten sein müssen, die Tour unterbrechungsfrei ist und dass kein Knoten mehrfach in der Lösung besucht werden darf. Durch geschickte Wahl der Testobjekte und der Kostenobergrenze können auch Aussagen über die Anzahl hinzuzufügender Knoten getroffen und damit die Methode weiter getestet werden.

```
Graph g1 = new Graph(nodeList1, edgeList1);
Graph result1 = randomRepair.repair(g1, 100);

assertTrue("Der Graph ist zulässig",
        result1.tourHasValidDepots() && result1.tourIsUninterrupted() && result1.tourHasNoDuplicates());

assertTrue("Es wurden bis zu 7 Knoten hinzugefügt", result1.getUnvisitedNodes().size() <= 7);
assertTrue("Kostenobergrenze wurde nicht überschritten", result1.getCostOfTour() <= 100);


for (Edge e : result1.getE()) {
    assertTrue("Knoten wurden an beste Position eingefügt",
            e.getStartNode().getX() < e.getEndNode().getX());
}
```

Abbildung 19: Auszug eines JUnit-Tests für die *Random Repair* Methode

Eine Besonderheit der Tests ist die Tatsache, dass es zu Falschmeldungen kommen kann. So muss ein grüner Balken in JUnit nicht zwangsweise bedeuten, dass die Implementierung fehlerfrei ist und umgekehrt bedeutet ein roter Balken nicht unbedingt, dass die Implementierung fehlerhaft ist. Es besteht beispielsweise die Möglichkeit, dass der Test selbst fehlerhaft ist und daher einen Fehler anzeigt, obwohl die Implementierung korrekt ist. Aus Sicht der Qualitätssicherung ist es jedoch besser, fälschlicherweise einen Fehler angezeigt zu bekommen, da dann die Ursache bzw. die vermeintliche Fehlerquelle intensiver untersucht werden kann. Wenn aber sowohl die Implementierung als auch der Test fehlerhaft sind und ein Fehler deswegen nicht angezeigt wird, ist die Wahrscheinlichkeit groß, dass der Fehler unentdeckt bleibt (vgl. Matt 2021).

Dieses Problem wird in dieser Arbeit einerseits dadurch adressiert, dass die zu testenden Einheiten möglichst klein gehalten werden. Dies reduziert die Komplexität der Testfälle und damit auch die Fehlerrate. Andererseits werden während der Implementierung absichtlich Fehler manuell eingebaut, um zu prüfen, ob sie von den Testfällen detektiert werden. Das sogenannte *Mutation-Testing*, welches bei der professionellen Softwareentwicklung zum Einsatz kommt, baut auf diesen Ansatz auf und automatisiert den Prozess (vgl. Matt 2021).

5.4 Softwarearchitektur

Die Softwarearchitektur beschreibt die Abhängigkeiten einzelner Komponenten in der Anwendung zueinander und hat u. a. Einfluss auf die Performanz, Erweiterbarkeit und Lesbarkeit der Software. Wie bereits in Abschnitt 5.1 beschrieben, kommen in dieser Arbeit einige der von Gamma et al. (1995) beschriebenen Entwurfsmuster (engl. s*oftware patterns*) zur Strukturierung der Software zum Einsatz. Mithilfe dieser Patterns werden wiederkehrende Probleme in Implementierungen und die entsprechenden Lösungsansätze beschrieben.

Das erste in dieser Arbeit implementierte Muster ist das sogenannte *Factory Method*-Pattern. Mit diesem Pattern wird die Erzeugung der Klassen *ALNS* und *EvolutionaryAlgorithm*, welche das Interface *IHeuristicAlgorithm* implementieren, in einer *Factory*-Instanz zentralisiert. Wird eine Instanz eines dieser Objekte von einer Klasse benötigt, so wendet sie sich an die *Factory* und fordert die Algorithmus-Instanz mithilfe der *Factory*-Methode an.

Der Rückgabetyp der *Factory*-Methode ist das Interface, welches von beiden Algorithmus-Klassen implementiert wird. Da die anfordernde Klasse nun nicht mehr den konkreten Klassennamen *„ALNS“* kennen muss, führt dies zu einer Entkopplung der anfordernde Klasse und der *ALNS*-Klasse (vgl. Gamma et al. 1995: 121-125). Neben der Entkopplung trägt das *Factory Method*-Pattern auch zur besseren Lesbarkeit bei, da nun die komplexen Objekte nicht mehr an verschiedenen Stellen des Codes instanziiert werden müssen. Außerdem können mit diesem Muster neue heuristische Algorithmen einfacher hinzugefügt werden, da sie nun nur noch an einer Stelle eingebunden werden müssen.

In Kombination mit dem *Factory Method*-Pattern wird in dieser Arbeit auch das *Singleton*-Pattern umgesetzt. Mit diesem Pattern soll sichergestellt werden, dass von der *Factory*-Klasse nur eine Instanz erzeugt werden kann, da die Erzeugung mehrerer Instanzen keinen Mehrwert bringen und ein zentraler Zugriffspunkt zu diesen wichtigen Klassen eine bessere Kontrolle bzw. Steuerung erlauben. Zur Erläuterung haben Gamma et al. (1995: 144-146) hierfür das Beispiel angeführt, dass es zwar sinnvoll sein kann, mehrere Drucker zu haben, aber mehrere Spooler zur Verwaltung der Druckaufträge zu Problemen führen können. Das *Singleton*-Pattern wird in Kombination mit dem *Factory Method*-Pattern umgesetzt, indem die Sichtbarkeit des Konstruktors der *Factory*-Klasse auf *privat* gesetzt wird und zur Erzeugung einer Instanz eine entsprechende öffentliche Methode *getInstance()* bereitgestellt wird. Diese Methode prüft, ob bereits eine Instanz der *Factory*-Klasse existiert und gibt diese zurück. Andernfalls ruft die Methode den privaten Konstruktor auf und speichert die Instanz für künftige Anfragen in einer statischen Variablen ab.

Die Kombination des *Factory Method*-Patterns und des *Singleton*-Patterns werden im UML Diagramm in Abbildung 20 dargestellt.

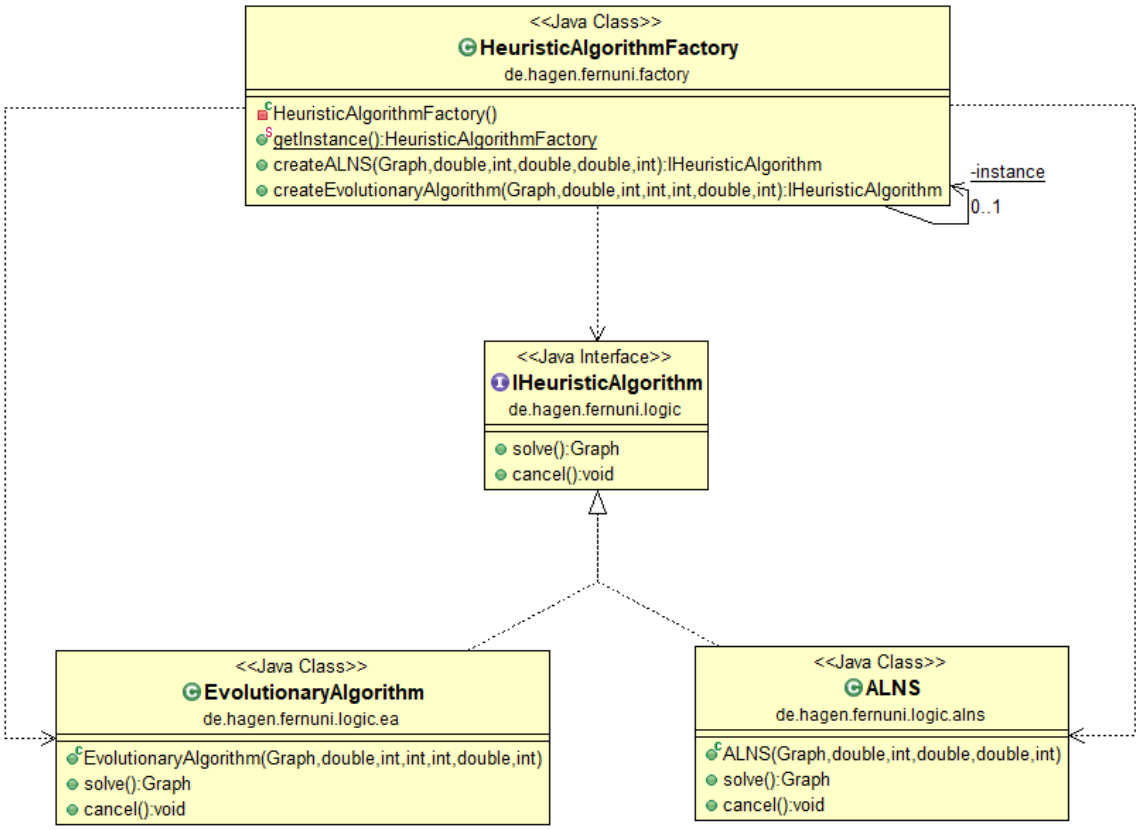


Abbildung 20: UML-Diagramm des *Factory Method-Patterns* und des *Singleton*-Patterns

Das letzte Muster, welches in dieser Arbeit zur Anwendung kommt, ist das *Strategy*-Pattern, dessen vereinfachtes UML-Diagramm in Abbildung 21 dargestellt wird. Mit dem *Strategy*-Pattern wird die Wahl eines Algorithmus zur Laufzeit ermöglicht. Dies ist notwendig, da die Wahl beispielweise von Benutzereingaben abhängen kann, die auch während der Laufzeit des Programms getätigt werden können (vgl. Gamma et al. 1995: 349-353).

Mit dem Strategy-Pattern werden Teile der Algorithmen-Klassen in die *Context*-Klasse ausgelagert. Konkret wird in der *Context*-Klasse die Methode *execute()* implementiert, in welcher die vom *IHeuristicAlgorithm*-Interface deklarierte Methode *solve()* zur Ausführung gebracht wird. Da beide Algorithmen-Klassen *solve()* implementieren müssen, kann zur Laufzeit die Wahl eines konkreten Algorithmus getroffen werden. Die Wahl wird bei der Instanziierung der Kontext-Klasse getroffen, indem die entsprechende Algorithmus-Klasse als Parameter übergeben wird.

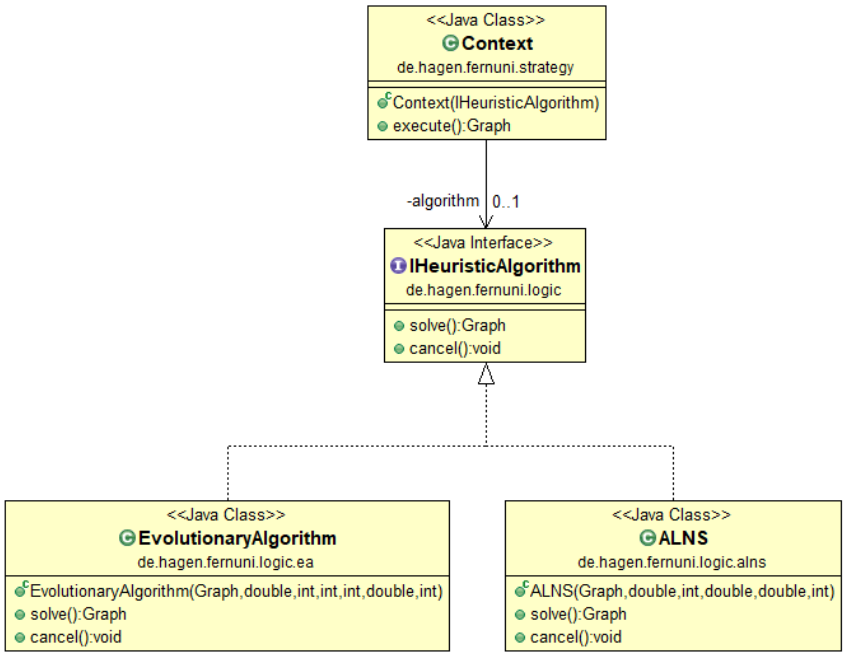


Abbildung 21: UML-Diagramm des *Strategy*-Patterns

5.5 Entwicklung der Benutzerschnittstelle

Zur Skizzierung der graphischen Benutzeroberfläche wird in dieser Arbeit *Balsamiq Wireframes* verwendet. Mit diesem Werkzeug können insbesondere am Anfang des Entwicklungsprozesses die Auswirkungen unterschiedlicher Design-Entscheidungen besser abgeschätzt und der generelle Workflow der Anwendungen modelliert werden, ohne ständig aufwändige Änderungen an der Implementierung der GUI vornehmen zu müssen. Die Abbildung 22 zeigt die Skizzen zweier Fenster der GUI, die mit dem Tool erstellt wurden. Das Balsamiq Wireframes Projekt ist im beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) enthalten.

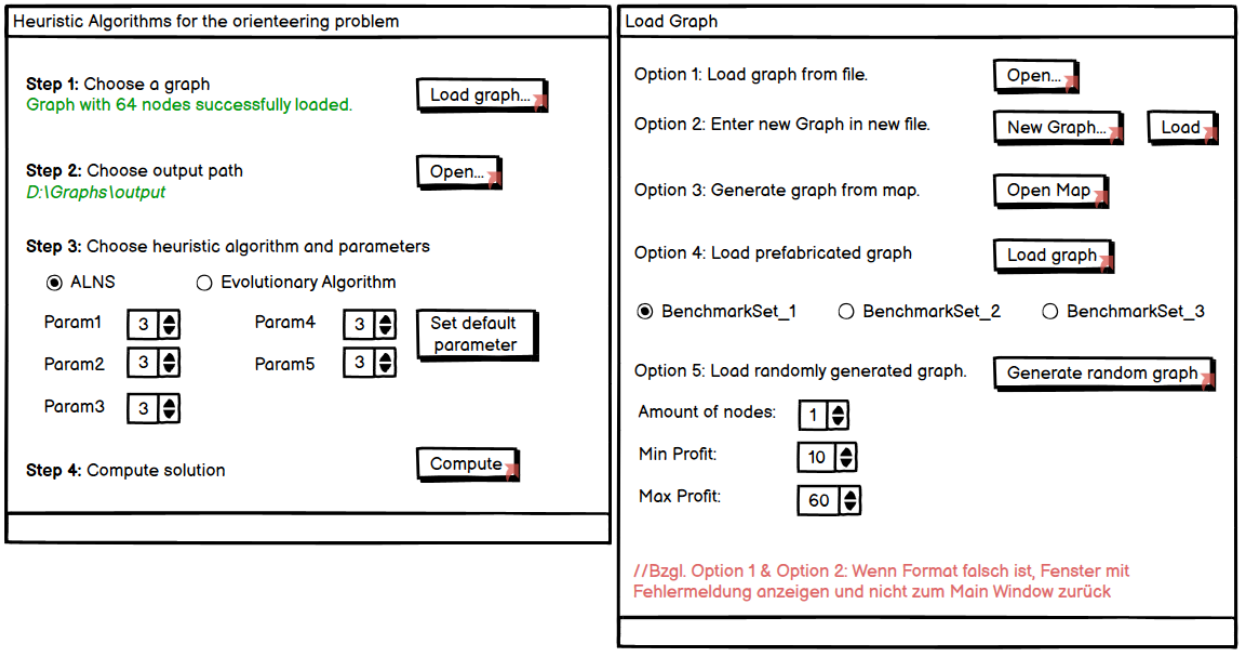


Abbildung 22: Skizzierung der zu entwickelnden GUI mit Balsamiq Wireframes

Die tatsächliche Entwicklung bzw. Implementierung der graphischen Benutzeroberflächen wird mit dem Eclipse-Plugin *WindowBuilder* umgesetzt. Mit dem Plugin kann die GUI graphisch gestaltet werden, indem Elemente, wie z. B. Textfelder, Buttons oder Checkboxen, per Drag-and-Drop hinzugefügt werden. Die Eigenschaften einzelner Elemente können in einer entsprechenden Tabelle bzw. Übersicht schnell und einfach angepasst werden. Das Plugin generiert für die vorgenommenen Änderungen entsprechende Codeabschnitte, die ohne das Plugin manuell programmiert werden müssten. Abbildung 23 zeigt den Codeumfang, der zur Darstellung einer einfachen Textzeile benötigt wird und von *WindowBuilder* automatisiert erzeugt wird. Tiefergehende Eigenschaften, wie z. B. die Formatierung von Textfeldern oder Mausklick-Events, müssen weiterhin manuell programmiert werden, da sie vom Plugin nicht generiert werden können (vgl. Eclipse Foundation o. D.).

```
JLabel lblStep_1 = new JLabel("Schritt 1: W\u00E4hle einen Graphen");
lblStep_1.setBounds(0, 2, 330, 25);
lblStep_1.setFont(new Font("Tahoma", Font.BOLD, 20));
panelLoadGraph.add(lblStep_1);
panelLoadGraph.add(btnLoadGraph);
```

Abbildung 23: Code zur Darstellung eines Textes in der GUI

Zur Gegenüberstellung der in Abbildung 22 gezeigten Skizzen, wird in Abbildung 23 die finale Benutzeroberfläche der entsprechenden Fenster dargestellt.

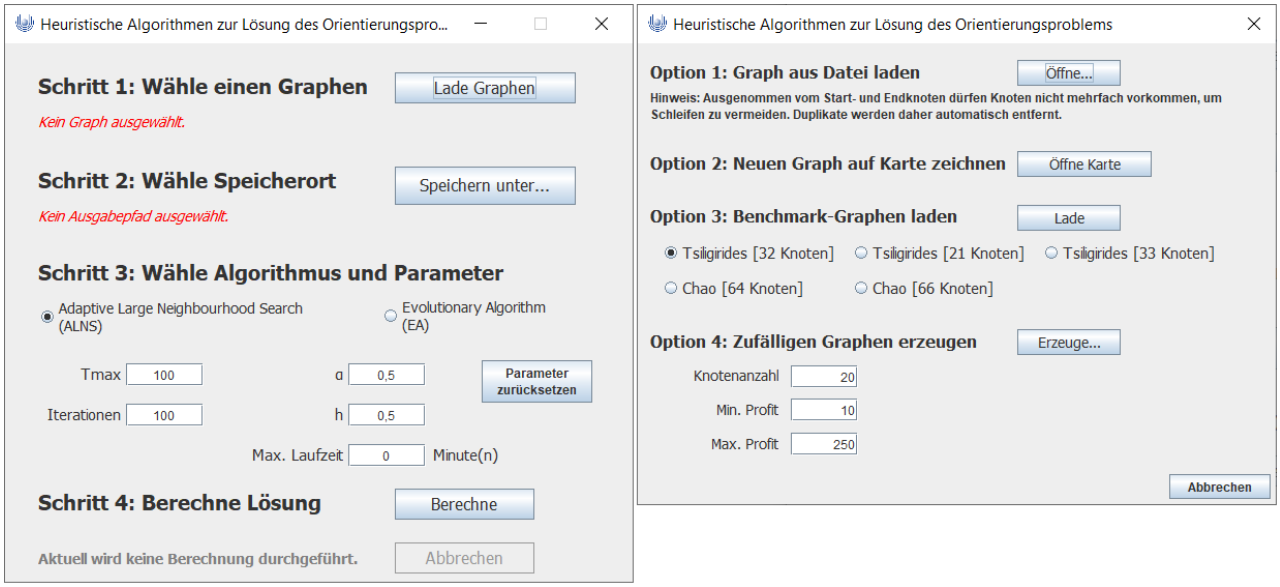


Abbildung 24: Finales Design des Hauptfensters und des "Lade Graphen"-Dialogs

Mit der graphischen Benutzeroberfläche wird auch der generelle Workflow der Anwendungen festgelegt. Bei der Gestaltung ist sowohl auf die einfache Nutzbarkeit und Verständlichkeit zu achten. Der Nutzer sollte jederzeit verstehen, wo er sich aktuell in der Anwendung befindet und was seine nächste Aufgabe ist. Häufige Verzweigungen in der Anwendung und sperrige Beschreibungen verwirren daher in der Regel den Nutzer, anstatt ihn zu unterstützten. Die Anwendung bzw. sein Workflow sollte idealerweise für den Nutzer selbsterklärend sein. Für die Hauptanwendungen dieser Arbeit wird auf der linken Seite in Abbildung 24 das Hauptfenster dargestellt, in welchem der Workflow dargestellt wird und welches den Nutzer schrittweise zur Anwendung eines heuristischen Algorithmus anleitet.

6 Analyse der heuristischen Algorithmen

In diesem Kapitel werden sowohl der *Adaptive Large Neighbourhood Search* Algorithmus von Santini (2019) als auch der evolutionäre Algorithmus von Kobeaga et al. (2018) anhand diverser Benchmarking-Tests analysiert. Dabei wird zunächst untersucht, wie sich die Parametereinstellungen auf die Qualität der erzeugten Lösungen auswirken. Die Parametereinstellungen, welche in den Tests im Schnitt die besten Lösungen erzeugt haben, werden dann für weitere Untersuchungen verwendet. Außerdem wird geprüft, inwiefern die Form der zugrunde liegenden Graphen die Performanz der Algorithmen beeinflussen. So ist es beispielsweise möglich, dass Graphen mit stärker gruppierten Knoten vom ALNS Algorithmus besser behandelt werden können, da hier auch Cluster-basierte Heuristiken zur Anwendung kommen. Anschließend werden beide heuristische Algorithmen mit weiteren Algorithmen aus der Literatur in Benchmarking-Tests verglichen. Für eine bessere Einordnung der Algorithmen werden als Basis- bzw. Referenzwerte die Qualität rein zufällig erzeugter Lösungen herangezogen.

Im folgenden Abschnitt werden zunächst die verwendeten Benchmarking-Instanzen vorgestellt. In Abschnitt 6.2 werden weitere Algorithmen zur Lösung des Orientierungsproblems aus der Literatur kurz vorgestellt bzw. benannt, mit denen sowohl der ALNS Algorithmus als auch der evolutionäre Algorithmus verglichen werden sollen. Auch das Vorgehen zur Erzeugung zufälliger Lösungen wird in diesem Abschnitt erläutert. In den darauffolgenden Abschnitten werden beide Algorithmen anhand unterschiedlicher Benchmarking-Tests hinsichtlich der eingangs beschriebenen Aspekte untersucht.

6.1 Benchmark-Instanzen

Für den Vergleich des ALNS Algorithmus und des evolutionären Algorithmus mit weiteren Algorithmen aus der Literatur werden in Abschnitt 6.5 die Benchmark-Instanzen von Tsiligirides und Chao verwendet. Beide Algorithmen werden außerdem auch anhand größerer Graphen untersucht, die in dieser Abschlussarbeit zufällig erzeugt wurden. Alle Benchmark-Graphen sind auf dem beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) hinterlegt.

Die von Tsiligirides (1984) erzeugten Benchmark-Graphen, welche in Abbildung 25 dargestellt sind, werden in der Literatur gerne herangezogen, um Algorithmen zur Lösung des Orientierungsproblems auf ihre Performanz zu prüfen. Die Benchmark-Instanzen setzen sich aus den Graphen und folgenden Kostenobergrenzen zusammen:

- Kostenobergrenzen für Graph $G1$ (32 Knoten):
  5, 10, 15, 20, 25, 30, 35, 40, 46, 50, 55, 60, 65, 70, 73, 75, 80, 85
- Kostenobergrenzen für Graph $G2$ (21 Knoten):
  15, 20, 23, 25, 27, 30, 32, 35, 38, 40, 45
- Kostenobergrenzen für Graph $G3$ (33 Knoten):
  15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110

Beim Vergleich der Ergebnisse von Tsiligirides mit den zugrundeliegenden Benchmark-Instanzen haben Chao et al. (1996: 484) festgestellt, dass der Graph $G1$ einen Fehler enthält. Der 30. Knoten müsste die Koordinaten (4.90, 14.90) statt (4.90, 18.90) besitzen, da sonst Tsiligirides' Ergebnisse nicht nachgestellt werden können. Aus diesem Grund wird zusätzlich der korrigierte Graph $G4$ eingeführt, welcher mit den gleichen Kostenobergrenzen wie $G1$ getestet wird.

Node I	Problem 1			Problem 2			Problem 3		
	$X(I)$	$Y(I)$	$S(I)$	$X(I)$	$Y(I)$	$S(I)$	$X(I)$	$Y(I)$	$S(I)$
1	10.50	14.40	0	4.60	7.10	0	19.10	24.30	0
2	18.00	15.90	10	5.70	11.40	20	12.60	24.90	20
3	18.30	13.30	10	4.40	12.30	20	14.40	28.00	20
4	16.50	9.30	10	2.80	14.30	30	16.90	28.10	20
5	15.40	11.00	10	3.20	10.30	15	20.70	28.20	20
6	14.90	13.20	5	3.50	9.80	15	12.50	26.60	20
7	16.30	13.30	5	4.40	8.40	10	21.80	27.30	20
8	16.40	17.80	5	7.80	11.00	20	12.50	22.60	20
9	15.00	17.90	5	8.80	9.80	20	22.50	17.00	30
10	16.10	19.60	10	7.70	8.20	20	19.90	15.00	30
11	15.70	20.60	10	6.30	7.90	15	14.90	15.10	30
12	13.20	20.10	10	5.40	8.20	10	11.50	18.60	30
13	14.30	15.30	5	5.80	6.80	10	12.40	29.80	30
14	14.00	5.10	10	6.70	5.80	25	17.80	28.10	30
15	11.40	6.70	15	13.80	13.10	40	9.10	29.80	40
16	8.30	5.00	15	14.10	14.20	40	10.00	32.60	40
17	7.90	9.80	10	11.20	13.60	30	13.90	33.10	40
18	11.40	12.00	5	9.70	16.40	30	19.95	10.30	40
19	11.20	17.60	5	9.50	18.80	50	15.20	8.00	40
20	10.10	18.70	5	4.70	16.80	30	14.70	31.20	50
21	11.70	20.30	10	5.0	5.60	0	7.40	36.50	50
22	10.20	22.10	10				21.00	25.50	50
23	9.70	23.80	10				18.00	25.30	10
24	10.10	26.40	15				19.50	24.70	10
25	7.40	24.00	15				21.40	21.80	10
26	8.20	19.90	15				16.00	21.40	10
27	8.70	17.70	10				18.65	26.20	10
28	8.90	13.60	10				17.90	28.90	10
29	5.60	11.10	10				14.30	19.90	20
30	4.90	18.90	10				17.00	19.00	20
31	7.30	18.80	10				10.80	21.00	20
32	11.20	14.10	0				15.70	23.70	10
33							18.20	24.00	0

Abbildung 25: Tsiligirides' Benchmark-Graphen (Tsiligirides 1984: 800)

Weiterhin stellen Chao et al. (1996: 488) zwei Benchmark-Graphen mit 66 Knoten bzw. 64 Knoten vor, welche in Abbildung 26 dargestellt sind. Die Knoten des Graphen mit 66 Knoten ($G5$) sind quadratisch angeordnet. Die Start- und Endknoten liegen im Zentrum des Quadrats verhältnismäßig nahe beieinander. Die übrigen Knoten haben einen Abstand zu ihren adjazenten Knoten von 2 oder $2\sqrt{2}$ Längeneinheiten. Die Knoten mit der größten Distanz zum Zentrum besitzen höhere Profitwerte als näher gelegene Knoten. Für die Benchmark-Instanzen werden für den Graphen $G5$ Kostenobergrenzen beginnend von 5 bis 130 verwendet, die um jeweils 5 Einheiten inkrementiert werden, sodass sich 26 Instanzen ergeben.

Der zweite von Chao et al. (1996: 488) vorgestellte Graph ($G6$) setzt sich aus 64 rautenförmig angeordneten Knoten zusammen. Die Start- und Endknoten sind auf der oberen bzw. unteren Spitze der Raute platziert. Alle Knoten haben hier eine Distanz von 2 oder $2\sqrt{2}$ Längeneinheiten zu ihren adjazenten Knoten. Für diesen Graphen werden Kostenobergrenzen von 15 bis 80 gewählt, wobei sie auch hier jeweils um 5 Einheiten erhöht werden.

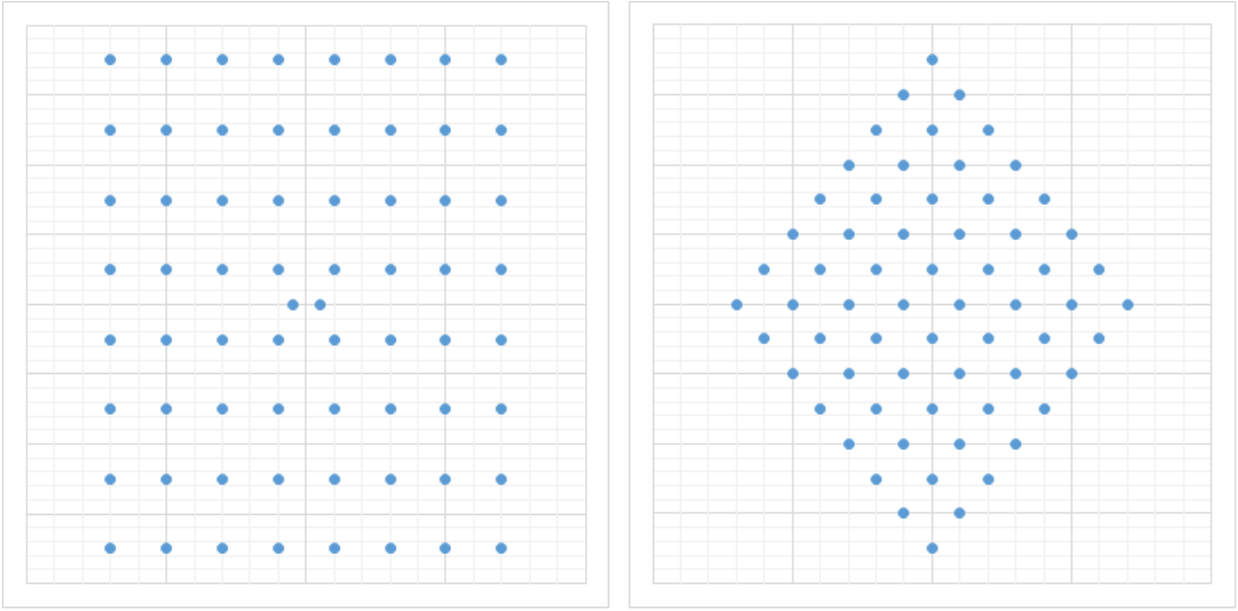


Abbildung 26: Chao et al. Benchmark-Graphen: G5 (links), G6 (rechts)

Für die Benchmarking-Tests in Abschnitt 6.3, mit denen die Auswirkungen unterschiedlicher Parameterkombinationen auf die Qualität der erzeugten Lösungen untersucht werden sollen, werden acht zufällig erzeugte Graphen verwendet. Der kleinste Graph setzt sich aus 100 Knoten zusammen und jeder weitere Graph wächst jeweils um weitere 100 Knoten an. Statt die Knoten rein zufällig anzuordnen werden sie clusterweise hinzugefügt, wie in Abbildung 27 für den Graphen $G10$ mit 400 Knoten dargestellt wird. Für das Benchmarking werden den Graphen folgende Kostenobergrenzen zugeordnet:

Graph	Kostenobergrenze
$G7$ (100 Knoten)	400
$G8$ (200 Knoten)	900
$G9$ (300 Knoten)	1400
$G10$ (400 Knoten)	1900
$G11$ (500 Knoten)	2200
$G12$ (600 Knoten)	2800
$G13$ (700 Knoten)	3600
$G14$ (800 Knoten)	4400

Tabelle 5: Kostenobergrenzen für Graphen $G7$-$G14$

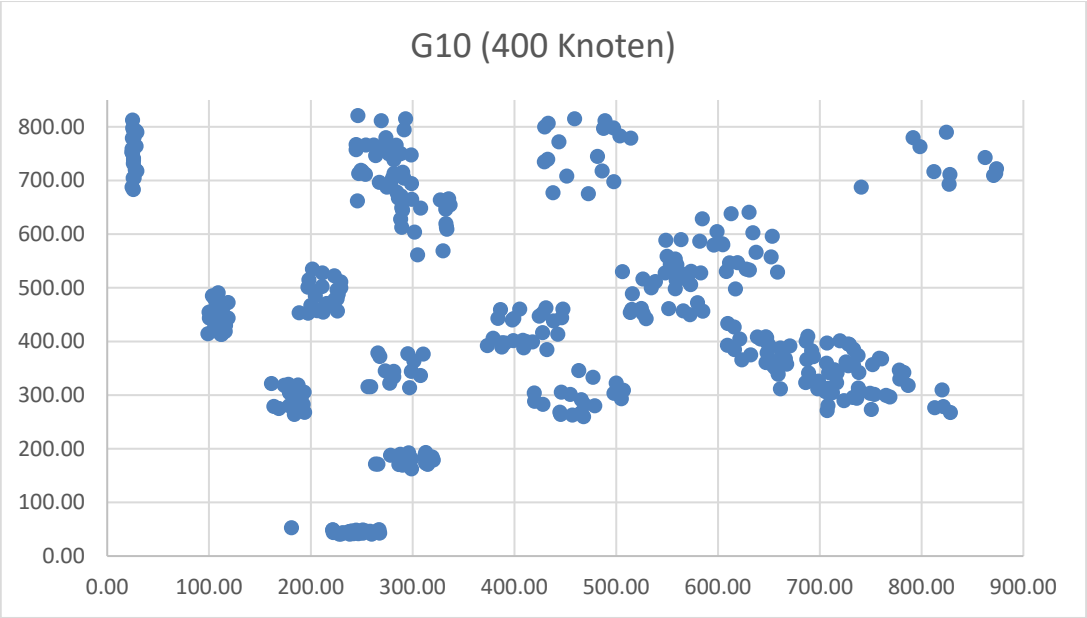


Abbildung 27: Benchmark-Graph $G10$ mit 400 Knoten

Für die Benchmarking-Tests in Abschnitt 6.4, mit welchen geprüft werden soll, inwiefern sich die Form der Graphen auf die Performanz der Algorithmen auswirken, werden drei Graphen mit jeweils 200 Knoten erzeugt. Die Graphen unterscheiden sich in der Anzahl und Größe ihrer Cluster, wie in Abbildung 28 illustriert wird. Graph $G15$ besitzt ein schwaches Clustering, während sich Graph $G16$ aus drei großen Clustern zusammensetzt. Der Graph $G17$ besitzt hingegen ein stärkeres Clustering. Die Knoten dieses Graphen sind in 10 Cluster angeordnet. Für den Graphen $G15$ wird die Kostenobergrenze auf 1600 festgelegt. Für die Graphen $G16$ und $G17$ wird eine Kostenobergrenze von 1400 verwendet.

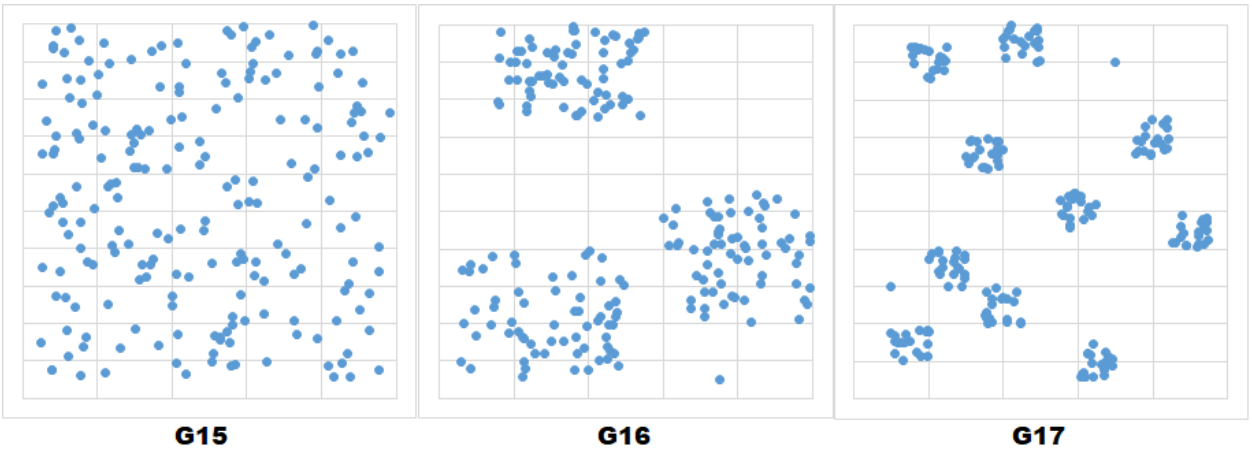


Abbildung 28: Graphen $G15$-$G17$

6.2 Weitere Algorithmen zur Lösung des Orientierungsproblems

Um die in dieser Arbeit vorgestellten Algorithmen besser einordnen zu können, werden sie in einigen Benchmarking-Tests mit weiteren Algorithmen aus der Literatur verglichen. In diesem Abschnitt werden die Kernaspekte dieser Algorithmen kurz vorgestellt oder ihre wesentlichen Konzepte benannt.

Die ersten vier Algorithmen wurden von Tsiligirides (1984) vorgestellt. Dabei handelt es sich um einen stochastischen und einen deterministischen Algorithmus, sowie um die Kombination dieser Algorithmen mit einem Tourenoptimierungsverfahren.

Der deterministische Algorithmus basiert auf dem von Wren und Holliday (1972) vorgestellten Tourenplanungsverfahren, in welchem der geographische Raum in Sektoren unterteilt wird. Die Sektoren werden dabei durch zwei konzentrische Kreise und einem Bogen unterteilt, wie in Abbildung 29 dargestellt wird. Die Touren werden so erzeugt, dass sie innerhalb der Sektoren (gestrichelter Bereich in der Abbildung) liegen, um die Reisekosten gering zu halten. Durch Veränderung der Radii und durch Rotation des Bogens erzeugt Tsiligirides 48 unterschiedliche Sektoren, in welchen die Lösungen erzeugt werden. Die beste gefundene Lösung wird als Ergebnis der Methode zurückgegeben (vgl. Tsiligirides 1984: 799 f.).

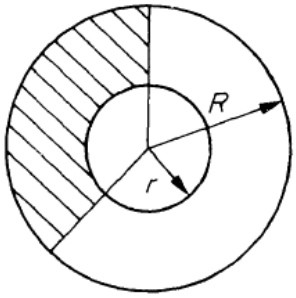


Abbildung 29: Sektoren im deterministischen Algorithmus von Tsiligirides (Tsiligirides, 1984)

Beim stochastischen Algorithmus wird das Monte-Carlo-Verfahren angewendet, um eine große Anzahl von Lösungen zu erzeugen. Dabei wird jedem Knoten v_i ein Attraktivitätswert $A_i = (\frac{s_i}{c_{i,last}})^4$ zugewiesen, wobei s_i für den Profitwert des Knotens steht und $c_{i,last}$ die Distanz des Knotens zum letzten Knoten in der Tour beschreibt. Die vier Knoten mit den höchsten Attraktivitätswerten werden durch $P_i = \frac{A_i}{\sum_{t=1}^{4} A_t}, i = \{1,2,3,4\}$ normalisiert. Ein Knoten v_i wird mit der Wahrscheinlichkeit P_i ausgewählt und in die Tour eingefügt. Dieses Vorgehen wird so lange wiederholt, bis kein Knoten mehr eingefügt werden kann, ohne die Kostenobergrenze T_{max} zu überschreiten. Mit diesem Verfahren werden 3.000 Lösungen erzeugt und die beste Lösung davon ausgewählt (vgl. Tsiligirides 1984: 798 f.).

Tsiligirides stellt außerdem ein Tourenoptimierungsverfahren vor. Bei diesem Verfahren wird zunächst eine 2-opt Heuristik angewendet, um die Reisekosten der aktuellen Tour zu senken. Anschließend werden weitere Knoten zur Tour hinzugefügt, um den erzielten Profit zu erhöhen. Schließlich wird versucht, durch das Entfernen besuchter Knoten und das Hinzufügen unbesuchter Knoten den Gesamtprofit weiter zu erhöhen. Da das Tourenoptimierungsverfahren eine initial erzeugte Lösung benötigt, werden jeweils die Lösungen des stochastischen und des deterministischen Verfahrens verwendet (vgl. Tsiligirides 1984: 801 f.).

Der nächste Algorithmus von Golden et al. (1987) setzt sich aus drei Schritten zusammen. Zunächst werden sukzessive Knoten in die Tour aufgenommen, die einen hohen Profit besitzen und geringe Mehrkosten verursachen. Anschließend wird ein 2-opt-Verfahren zur Kostensenkung eingesetzt, welches besuchte Knoten durch unbesuchte Knoten ersetzt. Am Ende des zweiten Schrittes werden dann die Knoten eingesetzt, welche die geringsten Mehrkosten verursachen, ohne dabei die Kostenobergrenze zu überschreiten. Der dritte Schritt führt den sogenannten

Schwerpunkt einer Lösung L als neue Metrik ein. Die Koordinaten des Schwerpunktes berechnen sich durch

$$x_S = \frac{\sum s(v_i) \cdot x(v_i)}{\sum s(v_i)} \quad und \quad y_S = \frac{\sum s(v_i) \cdot y(v_i)}{\sum s(v_i)}, \forall v_i \in L\,.$$

Auch hier beschreibt $s(v_i)$ den Profitwert des Knotens v_i und $x(v_i)$ bzw. $y(v_i)$ stellen die Koordinaten des Knotens dar. Für alle Knoten des Graphen wird nun das Verhältnis ihres Profits zur Distanz zum Schwerpunkt berechnet. Es werden iterativ die Knoten mit dem größten Profit-Distanz-Verhältnis in eine neue Tour aufgenommen, bis die Kostenobergrenze erreicht wird. Die Schritte 2 und 3 werden anschließend so oft wiederholt, bis sich die neu erzeugte Lösung nicht mehr von der vorangegangenen Lösung unterscheidet (vgl. Golden et al. 1987: 308 f.).

Beim Algorithmus von Chao et al. (1996) wird eine Ellipse auf den Graphen projiziert, wobei die Start- und Endknoten zwei Scheitelpunkte der Ellipse darstellen und die lange Achse der Ellipse auf T_{max} gesetzt wird. Zur Erzeugung von k initialen Lösungen wird ein gieriger Algorithmus auf die Knoten innerhalb der Ellipse angewendet, der lediglich die potentiellen Mehrkosten der Knoten berücksichtigt und dabei ihre Profite ignoriert. Eine Lösung $L_i, 1 \leq i \leq k$ wird erzeugt, indem zunächst der Knoten v_i, der die i-größte Distanz zum Start- und Endknoten besitzt, in die Tour aufgenommen wird, sodass sich der Pfad $(v_{start}, v_i, v_{Ende})$ ergibt. Anschließend werden mit dem gierigen Algorithmus weitere Knoten zur Lösung L_i hinzugefügt. Es werden so lange neue Lösungen erzeugt, bis alle Knoten innerhalb der Ellipse mindestens einer Lösung L_i angehören. Anschließend wird ein 2-opt-Verfahren auf die Lösungsmenge angesetzt. Dazu werden aus der Lösung mit dem höchsten Gesamtprofit schrittweise besuchte Knoten entfernt und durch noch nicht besuchte Knoten ersetzt, die in mindestens einer anderen initialen Lösung enthalten sind (vgl. Chao et al. 1996: 479-482).

Aufgrund ihres Umfangs werden im Folgenden weitere Algorithmen und ihre wesentlichen Konzepte lediglich benannt. Die entsprechenden wissenschaftlichen Ausarbeitungen, in welchen die Algorithmen detailliert beschrieben werden, sind im Literaturverzeichnis referenziert:

- Wang et al. (1995): Einsatz eines künstlichen neuronalen Netzwerkes (Hopfield Netzwerk)
- Mladenović/Hansen (1997): Anwendung einer „Variable Neighboorhood Search“ Methode
- Vansteenwegen et al. (2009): Anwendung einer gesteuerten lokalen Suchmethode
- Sevkli/Sevilgen (2010): Einsatz eines Partikelschwarmverfahrens

Als Referenzwerte werden für die Benchmark-Instanzen außerdem über einen begrenzten Zeitraum hinweg zufällige Lösungen erzeugt und jeweils die beste gefundene Lösung gespeichert.

6.3 Auswirkungen der Parametereinstellungen

In diesem Abschnitt werden die Auswirkungen unterschiedlicher Parametereinstellungen auf die Qualität der erzeugten Lösungen untersucht. Hierfür werden sowohl der ALNS Algorithmus als auch der evolutionäre Algorithmus mit unterschiedlichen Parameterkombinationen auf Graphen mit 100 bis 800 Knoten ($G7$-$G14$) angewendet (s. Abschnitt 6.1). Aufgrund der Menge der Benchmarking-Tests wird die Laufzeit eines Algorithmus auf 10 Minuten beschränkt. Jeder Test wird außerdem fünf Mal wiederholt, um Ausreißer entsprechend abfangen zu können.

Dieses Benchmarking wird auf einem System mit folgenden Spezifikationen durchgeführt:

Systemeigenschaft	Ausprägung
Prozessor	Intel Core i9-9960X @ 3.10 GHz, 16 Kerne, 22MB L3 Cache
Arbeitsspeicher	64GB DDR4 @ 2666 MHz
Betriebssystem	Ubuntu 20.04.3 LTS

Tabelle 6: Spezifikationen des Benchmarking-Systems

Die Java-Laufzeitumgebung bzw. Java Virtual Machine, in der die Tests laufen, wird mit folgenden Optionen konfiguriert:

JVM Option	Beschreibung
-Xms4G	Legt die initiale Heap-Größe auf 4GB fest.
-Xmx32G	Legt die maximale Heap-Größe auf 32GB fest.
-XX:+UseG1GC	Wählt den „G1“ Garbage Collector aus.
-XX:+HeapDumpOnOutOfMemoryError	Protokolliert Abstürze, die im Zusammenhang mit Arbeitsspeicher-Überlastungen stehen.
-XX:+UseGCOverheadLimit	Beschränkt die Zeit der Aufräumarbeiten des Garbage Collectors, bevor eine Fehlermeldung ausgelöst wird.

Tabelle 7: Konfiguration der Java Virtual Machine (vgl. Oracle 2021)

Für den ALNS Algorithmus werden 100 Parameterkombinationen untersucht, die aus dem kartesischen Produkt folgender Parametermengen bestehen:

$$M = \{1000, 2000, 3000, 4000, 5000\}$$
$$\alpha = \{0.1, 0.25, 0.5, 0.75, 0.9\}$$
$$h = \{0.2, 0.4, 0.6, 0.8\}$$

Analog werden die Parameterkombinationen für den evolutionären Algorithmus aus dem kartesischen Produkt folgender Parametermengen gebildet:

$$d2d = \{5, 10, 20, 50\}$$
$$npop = \{10, 20, 50, 100\}$$
$$ncand = \{5, 7, 10, 20\}$$
$$p = \{0.01, 0.05, 0.1, 0.25, 0.5, 0.75\}$$

Mit den Beschränkungen $d2d < npop$ und $ncand < npop$ ergeben sich so 216 Parameterkombinationen für den evolutionären Algorithmus. Weiterhin werden auch folgende Konfigurationen zur Untersuchung herangezogen, die von den Autoren empfohlen wurden:

- ALNS Algorithmus: $M = 5000;\ \alpha = 0.2026;\ h = 0.4314$
- Evolutionärer Algorithmus: $d2d = 50;\ npop = 100;\ ncand = 10;\ p = 0.01$

Die Parametereinstellungen des ALNS Algorithmus konnten abhängig von der Benchmark-Instanz und ausgehend von den Lösungen mit den niedrigsten Gesamtprofiten die erzielte Qualität der Lösungen um rund 10-40% und im Schnitt um ca. 20% verbessern. In Tabelle 8 werden die Ergebnisse des ALNS-Benchmarkings zusammengefasst. Die dazugehörigen Primärdaten und weitere Sichten auf die Daten sind auf dem beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) hinterlegt.

Zusammenfassung - Adaptive Large Neighbourhood Search Algorithmus					
Iteration	Durchschn. Profit (%)	a	Durchschn. Profit (%)	h	Durchschn. Profit (%)
1000	97.358	0.1	95.862	0.2	100.000
2000	98.499	0.25	98.689	0.4	98.474
3000	99.155	0.5	99.671	0.6	98.095
4000	99.165	0.75	99.414	0.8	96.083
5000	99.853	0.9	98.890		

Tabelle 8: Aggregierte Benchmarking-Ergebnisse des ALNS Algorithmus

Der ALNS Algorithmus erzeugte im Schnitt bessere Lösungen mit steigender Iterationsanzahl. In jeder Iteration werden durch das Entfernen und Hinzufügen von Knoten neue Lösungen erzeugt und mit den bisher gefundenen Lösungen verglichen. Je höher die Anzahl der Iterationen gewählt wird, desto höher ist die Wahrscheinlichkeit, eine neue potentiell bessere Lösung zu finden.

Hinsichtlich des Aggressivitätsfaktors α mit dem bestimmt wird, wie viele Knoten in einer Iteration durch eine Zerstörmethode von der aktuellen Lösung zu entfernen sind, wurden die besten Ergebnisse mit $\alpha = 0.5$ und $\alpha = 0.75$ erzielt. Je höher dieser Wert gewählt wird, desto mehr Knoten werden in einer Iteration aus einer aktuellen Lösung entfernt. Das bedeutet gleichzeitig, dass die Gemeinsamkeiten mit vorangegangen Lösungen schneller verworfen werden. Für $\alpha = 0.1$ bedeutet dies hingegen, dass in jeder Iteration lediglich rund 10% der von einer Lösung besuchten Knoten entfernt werden. Dies führt in den folgenden Iterationen dazu, dass ein entsprechend kleiner Spielraum zur Identifikation neuer Lösungen zur Verfügung steht bzw. dass die neuen Lösungen den vorangegangenen Lösungen stärker ähneln. Als Konsequenz ist die Qualität der gefundenen Lösungen in diesem Fall in höherem Maße von der initial erzeugten Lösung abhängig, als für größer gewählte Aggressivitätsfaktoren.

Mit Bezug auf den Decay-Faktor h wurden im Schnitt die besten Lösungen mit $h = 0.2$ erzielt. Wie in Abschnitt 4.2.8 erläutert, beeinflusst der Decay-Faktor die Wahrscheinlichkeit der Wahl einer Zerstör- bzw. Reparaturmethode in den nächsten Iterationen und hängt von der Qualität der von ihnen erzeugten Lösungen ab. Je besser die von einer Zerstör- und Reparaturmethode erzeugte Lösung ist, desto höher ist auch die Wahrscheinlichkeit, dass diese Methoden in den nächsten Iterationen wiedergewählt werden. Die Wahl des Decay-Faktors $h = 0.2$ bedeutet hierbei, dass bei der Bewertung in jeder Iteration die vorangegangenen Bewertungen lediglich zu 20% berücksichtigt werden. Entsprechend hängt die Bewertung der Methoden zu 80% von ihrer Performanz in der aktuellen Iteration ab.

Beim evolutionären Algorithmus lagen die Gesamtprofite der besten Lösungen 25% bis 35% und im Schnitt 30,5% höher als die Gesamtprofite der schlechtesten Lösungen. In Tabelle 9 werden die Ergebnisse des Benchmarkings für den evolutionären Algorithmus zusammengeführt. Auch hier sind die dazugehörigen Primärdaten und weitere Sichten auf die Daten auf dem beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) enthalten.

Zusammenfassung - Evolutionärer Algorithmus							
d2d	Durchschn. Profit (%)	npop	Durchschn. Profit (%)	ncand	Durchschn. Profit (%)	p	Durchschn. Profit (%)
5	95.754	10	82.315	5	96.926	0.01	99.646
10	98.017	20	92.379	7	96.814	0.05	98.434
20	99.969	50	98.080	10	98.450	0.1	98.417
50	99.915	100	99.835	20	100.000	0.25	98.452
						0.5	97.374
						0.75	94.134

Tabelle 9: Aggregierte Benchmarking-Ergebnisse des evolutionären Algorithmus

Wie in Kapitel 4.3 beschrieben wird, gliedert sich der evolutionäre Algorithmus nach der Initialisierungsphase in einen Block mit optimierenden Methoden und einen Block mit genetischen Methoden. In einer Iteration wird jeweils nur einer der beiden Blöcke ausgeführt werden. Der Parameter $d2d$ bestimmt hierbei, wie häufig der Optimierungsblock ausgeführt wird. Beim Benchmarking wurden im Schnitt die besten Ergebnisse erzielt, wenn $d2d = 20$ oder $d2d = 50$ gewählt wurde. In diesen Fällen wird der Optimierungsblock jede zwanzigste bzw. fünfzigste Iteration ausgeführt. Eine mögliche Erklärung für diese Beobachtung kann die Tatsache sein, dass die Methoden des Optimierungsblocks im Vergleich zu den genetischen Methoden relativ zeitaufwändig bzw. rechenintensiv sind. Je häufiger der Block ausgeführt wird, desto seltener können demnach die genetischen Methoden innerhalb des 10-minütigen Zeitlimits ausgeführt werden bzw. desto weniger Iterationen können insgesamt durchlaufen werden.

Hinsichtlich der Parameter $npop$ und $ncand$, mit denen die Populationsgröße bzw. die Größe der Kandidatenliste zur Auswahl zweier Eltern-Lösungen bestimmt wird, haben die Parameterkombinationen mit $npop = 100$ und $ncand = 20$ im Schnitt die besten Ergebnisse erzielt. Zwar steigt mit der Populationsgröße auch die benötigte Rechenzeit, da initial mehr Lösungen erzeugt werden müssen, jedoch wächst gleichzeitig auch der Pool aus denen Lösungen zur Erzeugung neuer Touren gewählt werden können. Das erhöht wiederum die Chance, dass Lösungen mit vorteilhaften Charakteristiken in der Population enthalten sind. Mit $ncand = 20$ werden aus der Population zwanzig Lösungen zufällig ausgewählt, aus denen dann mithilfe des Select-Operators zwei Eltern-Lösungen nach gewichtetem Zufall zur Kreuzung ausgewählt werden. Weil die Kandidaten zunächst zufällig ausgewählt werden, haben auch vergleichsweise schlechte Lösungen mit wenigen vorteilhaften Charakteristiken eine Chance, ausgewählt zu werden.

Bezüglich des Parameters p wurden im Schnitt die besten Lösungen mit $p = 0.01$ erzeugt. Diese Einstellung führt dazu, dass bei der Erzeugung der Population die Knoten jeweils mit einer Wahrscheinlichkeit von 1% in eine initiale Lösung aufgenommen und an eine zufällige Stelle in der Tour eingesetzt werden. Diese geringe Wahrscheinlichkeit hat auch zur Folge, dass in der Initialisierungsphase mehr Knoten durch den Add-Operator hinzugefügt werden können, als wenn bereits große Anteile zufällig ausgewählter Knoten in die Lösungen aufgenommen wurden. Mit dem Add-Operator werden die Knoten nicht zufällig ausgewählt, sondern unter Berücksichtigung ihrer Profite und den von ihnen verursachten Mehrkosten.

Die Parameter $Iterationen = 5000;\ a = 0.5;\ h = 0.2$ für den ALNS Algorithmus und die Parameter $d2d = 20;\ npop = 100;\ ncand = 20;\ p = 0.01$ für den evolutionären Algorithmus haben im Schnitt die besten Ergebnisse in den Benchmarking-Tests erzielt und nähern sich den von den Autoren vorgeschlagenen Parametereinstellungen, mit denen vergleichbare Ergebnisse erzielt werden konnten. In den folgenden Abschnitten werden die in diesem Benchmarking identifizierten Parameterkonfigurationen für weitere Tests verwendet.

Wie Abbildung 30 zeigt, konnten mit dem evolutionären Algorithmus stets bessere Lösungen gefunden werden als mit dem ALNS Algorithmus. Dabei unterlagen die besten Lösungen des ALNS-Benchmarkings den besten Lösungen, die mit dem evolutionären Algorithmus gefunden wurden, um höchstens 2,5%. Lediglich bei der Benchmark-Instanz mit 100 Knoten und $T_{max} = 400$ weisen die besten Lösungen beider Algorithmen den gleichen Gesamtprofit auf.

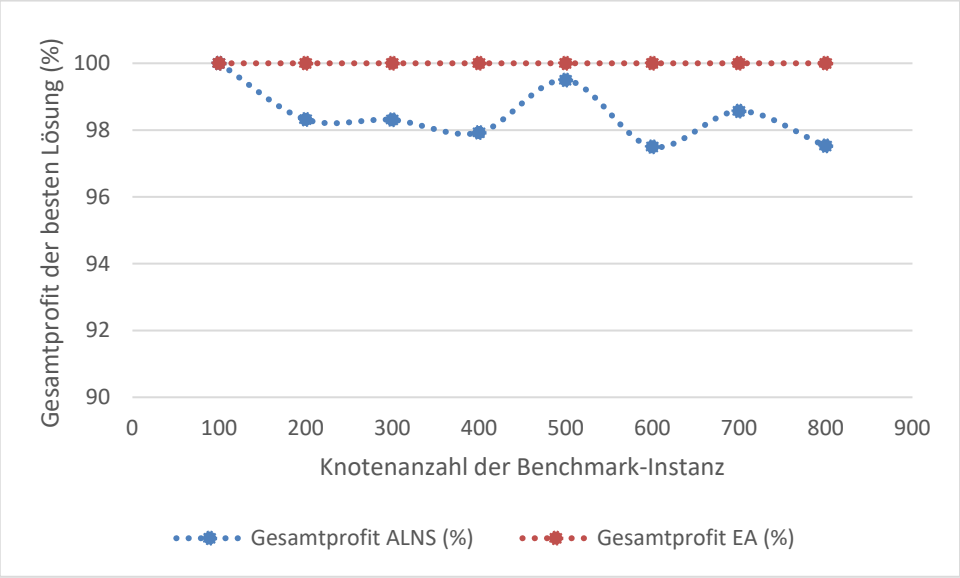


Abbildung 30: Vergleich der Performanz beider Algorithmen

6.4 Einfluss durch die Form der Graphen

In diesem Abschnitt wird untersucht, inwiefern sich die Form der Graphen auf die Performanz beider Algorithmen auswirkt. Hierzu werden drei Graphen mit einer Größe von jeweils 200 Knoten erzeugt. Wie in Abbildung 28 dargestellt wird, unterscheiden sich die Graphen dabei in ihrem Clustering-Faktor. Die beiden Algorithmen werden auf die drei Graphen jeweils fünf Mal mit den in Abschnitt 6.3 ermittelten Parametereinstellungen angewendet, also:

- ALNS Algorithmus: $Iterationen = 5000;\ a = 0.5;\ h = 0.2$
- Evolutionärer Algorithmus: $d2d = 20;\ npop = 100;\ ncand = 20;\ p = 0.01$

Die Ergebnisse des Benchmarkings werden in der Tabelle 10 zusammengefasst. Die dazugehörigen Primärdaten sind im beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) hinterlegt. In diesem Benchmarking wurden die besten Lösungen vom evolutionären Algorithmus erzeugt. Zudem hat der Algorithmus auch im Schnitt bessere Ergebnisse geliefert als der ALNS Algorithmus, wobei in diesem Benchmarking die Differenz mit rund 3% relativ gering ausfiel. Je stärker das Clustering der Graphen desto geringer fiel die Differenz aus. Eine mögliche Erklärung hierfür ist, dass der ALNS Algorithmus unter anderem Cluster-basierte Zerstör- und Reparaturmethoden zur Erzeugung neuer Lösungen verwendet, die

bei Graphen mit stärkerem Clustering bessere Ergebnisse erzielen können. Weil die Unterschiede in der Performanz jedoch nicht wesentlich sind, kann der Einfluss durch die Form eines Graphen vernachlässigt werden.

	Max. Profit	%	Durchschn. Profit	%
ALNS, Kein Clustering (G15)	10746.89	97.24	10638.544	97.96
EA, Kein Clustering (G15)	11051.79	100	10860.574	100
ALNS, Mittleres Clustering (G16)	11371.44	98.13	11255.96	98.61
EA, Mittleres Clustering (G16)	11588.29	100	11415.024	100
ALNS, Starkes Clustering (G17)	13614.21	99.64	13441.734	99.06
EA, Starkes Clustering (G17)	13663.43	100	13568.826	100

Tabelle 10: Performanz der Algorithmen für Graphen $G15$-$G17$

6.5 Gegenüberstellung mit weiteren Algorithmen aus der Literatur

Um die Performanz beider Algorithmen zu untersuchen, werden sie anhand der in Abschnitt 6.1 vorgestellten Benchmark-Instanzen $G1$-$G6$ von Tsiligirides (1984) und Chao (1996) analysiert. Die Ergebnisse beider Algorithmen werden außerdem den Ergebnissen zehn weiterer Algorithmen gegenübergestellt, die in Abschnitt 6.5 aufgeführt sind und den jeweiligen Literaturen entnommen werden können. Als Referenzwerte werden für alle Benchmarking-Instanzen jeweils eine Minute lang Lösungen nach einem Zufallsverfahren erzeugt, von denen die beste Lösung ausgewählt wird. Das Benchmarking-System hat für die hier betrachteten Benchmark-Instanzen im Schnitt jeweils rund 23,5 Millionen Lösungen zufällig erzeugt. Für eine bessere Übersicht der Ergebnisse werden den Algorithmen folgende Abkürzungen vergeben:

- **R**: Referenzlösung, die durch zufällige Erzeugung von Lösungen gefunden wurde
- **T1**: Tsiligirides‘ deterministischer Algorithmus
- **T2**: Tsiligirides‘ stochastischer Algorithmus
- **T3**: Tsiligirides‘ Tourenoptimierungsverfahren mit deterministischer Eingangslösung
- **T4**: Tsiligirides‘ Tourenoptimierungsverfahren mit stochastischer Eingangslösung
- **G**: Golden et al. (1987)
- **C**: Chao et al. (1996)
- **W**: Wang et al. (1995)
- **M**: Mladenović und Hansen (1997)
- **V**: Vansteenwegen et al. (2009)
- **S**: Sevkli und Sevilgen (2010)
- **ALNS**: Adaptive Large Neighbourhood Search Algorithmus von Santini (2019)
- **EA**: Evolutionärer Algorithmus von Kobeaga et al. (2018)

Die Ergebnisse der Benchmarking-Tests werden in den Tabellen 11-13 zusammengefasst. Die Primärdaten für die Algorithmen **R**, **ALNS** und **EA** sind auf dem beiliegenden Datenträger und im Github-Repository (vgl. Machaka 2022) hinterlegt.

Bei den Benchmark-Instanzen $G1$-$G4$ von Tsiligirides (1984) hat der Adaptive Large Neighbourhood Search Algorithmus von Santini (2019) im Benchmarking für alle Kostenobergrenzen zwar keine bessere Lösung finden können, jedoch sind die von dem Algorithmus erzeugten Lösungen hinsichtlich ihrer Qualität gleichauf mit den bisher besten gefundenen Lösungen. Die einzige Ausnahme findet sich in der Benchmark-Instanz $G1$ mit

$T_{max} = 75$. Wie Chao et al. (1996: 484) bereits festgestellt haben, beziehen sich Tsiligirides' Ergebnisse hier auf ein fehlerhaftes Datenset, weshalb seine Lösungen an dieser Stelle nicht nachgestellt werden können. Im korrigierten Datenset $G4$ wurden mit dem ALNS Algorithmus durchweg die besten Ergebnisse erzielt, wie der Tabelle 11 entnommen werden kann. Der evolutionäre Algorithmus von Kobeaga et al. (2018) hat ähnliche Ergebnisse liefern können. In rund 12% der Benchmarking-Instanzen $G1$ bis $G4$ hat dieser Algorithmus jedoch schlechtere Lösungen als seine Konkurrenten erzeugt.

Tmax	R	T1	T2	T3	T4	G	C	W	ALNS	EA
5	10					10	10	10	10	10
10	15					15	15	15	15	15
15	45					45	45	45	45	45
20	65	40	65	65	65	65	65	65	65	65
25	90	65	90	90	90	90	90	90	90	90
30	110	80	110	110	110	110	110	110	110	110
35	115	105	135	135	135	125	135	135	135	135
40	125	105	150	150	150	140	155	155	155	155
46	140	130	175	175	175	165	175	175	175	175
50	150	140	190	190	190	180	190	190	190	190
55	155	160	205	200	205	200	205	205	205	200
60	160	175	220	220	220	205	225	225	225	225
65	175	200	240	240	240	220	240	240	240	240
70	170	200	255	260	260	240	260	260	260	260
73	185	205	260	265	265	255	265	265	265	265
75	190	210	270	275	275	260	270	270	270	270
80	185	220	275	280	280	275	280	280	280	280
85	200	235	280	285	285	285	285	285	285	285

G1

Tmax	R	C	W	ALNS	EA
5	10	10	10	10	10
10	15	15	15	15	15
15	45	45	45	45	45
20	65	65	65	65	65
25	90	90	90	90	90
30	105	110	110	110	110
35	115	135	130	135	135
40	130	155	155	155	150
46	140	175	175	175	175
50	145	190	190	190	190
55	150	205	205	205	200
60	160	220	220	225	225
65	165	240	240	240	240
70	175	260	260	260	260
73	175	265	265	265	265
75	180	275	275	275	275
80	185	280	280	280	280
85	195	285	285	285	285

G4

Tabelle 11: Erzielte Profite für Benchmark-Instanzen G1 und G4

Tmax	R	T1	T2	T3	T4	G	C	W	ALNS	EA
15	120	100	120	120	120	120	120	120	120	120
20	190	130	190	200	200	200	200	200	200	200
23	205	130	205	210	210	210	210	205	210	200
25	210	145	230	230	230	230	230	230	230	230
27	230	160	230	230	230	230	230	230	230	230
30	255	190	250	265	260	260	265	265	265	265
32	265	195	275	300	300	260	300	300	300	300
35	290	200	315	320	320	300	320	320	320	320
38	325	210	355	355	355	355	360	360	360	360
40	345	240	395	385	395	380	395	395	395	395
45	370	270	430	450	450	450	450	450	450	450

G2

Tmax	R	T1	T2	T3	T4	G	C	W	ALNS	EA
15	170	70	100	100	100	170	170	170	170	170
20	190	120	140	140	140	200	200	200	200	200
25	250	140	190	190	190	250	260	250	260	260
30	290	180	240	240	240	320	320	320	320	320
35	350	220	290	280	290	380	390	390	390	390
40	390	240	330	340	330	420	430	420	430	420
45	380	280	370	370	370	450	470	470	470	440
50	420	310	410	420	420	500	520	520	520	520
55	430	340	450	440	460	520	550	550	550	550
60	450	350	500	500	500	580	580	580	580	580
65	460	410	530	530	530	600	610	610	610	610
70	480	460	560	560	560	640	640	640	640	640
75	490	490	590	600	590	650	670	670	670	670
80	520	510	640	640	640	690	710	700	710	700
85	530	520	670	670	670	720	740	740	740	740
90	530	580	690	700	700	770	770	770	770	770
95	570	610	720	740	730	790	790	790	790	790
100	560	640	760	770	760	800	800	800	800	800
105	580	660	770	790	790	800	800	800	800	800
110	600	680	790	800	800	800	800	800	800	800

G3

Tabelle 12: Erzielte Profite für Benchmark-Instanzen G2 und G3

Auch in den Benchmark-Instanzen $G5$ und $G6$ von Chao et al. (1996) schnitt der evolutionäre Algorithmus schlechter ab als die anderen Algorithmen. Nur in 19 von 40 Benchmark-Instanzen hat dieser Algorithmus Lösungen erzeugen können, die so gut wie die bisher gefundenen besten Lösungen sind. Ein Grund für das inkonsistente Verhalten des evolutionären Algorithmus in diesen Benchmarking-Tests kann mit einer verfrühten Erfüllung der Abbruchbedingung zusammenhängen. Verglichen mit den Benchmarking-Tests aus Abschnitten 6.3 und 6.4 werden hier verhältnismäßig kleine Graphen behandelt. Mit dem evolutionären Algorithmus wird eine Population mit $npop$ Lösungen erzeugt und das Routing regelmäßig optimiert. Das kann dazu führen, dass sich die Lösungen innerhalb der Population hinsichtlich ihrer Gesamtprofite angleichen, bevor bessere Lösungen gefunden werden können. Die Konsequenz wäre in diesem Fall ein vorzeitiger Abbruch des Algorithmus. Durch entsprechende Veränderung des Abbruchkriteriums konnten in diesem Benchmarking auch hier konsistent die bisher gefundenen, besten Lösungen nachgestellt werden. Diese Änderungen würden bei größeren Graphen jedoch zu erheblich längeren Laufzeiten führen, weshalb beim evolutionären Algorithmus das Abbruchkriterium entsprechend der Größe der Graphen angepasst werden sollte. Um eine Vergleichbarkeit mit dem im Abschnitt 6.3 getesteten evolutionären Algorithmus beizubehalten, werden die Ergebnisse mit verändertem Abbruchkriterium nicht in den Ergebnissen dieses Benchmarking-Tests aufgenommen.

G5

Tmax	R	C	M	V	S	ALNS	EA
5	10	10	10	10	10	10	10
10	40	40	40	40	40	40	40
15	120	120	120	120	120	120	120
20	200	195	205	175	205	205	205
25	275	290	290	290	290	290	280
30	330	400	400	400	400	400	400
35	370	460	465	465	465	465	465
40	400	575	575	575	575	575	535
45	435	650	650	640	650	650	620
50	460	730	730	710	730	730	710
55	505	825	825	825	825	825	795
60	525	915	915	905	915	915	855
65	525	980	980	935	980	980	920
70	555	1070	1070	1070	1070	1070	960
75	585	1140	1140	1140	1140	1140	1140
80	595	1215	1215	1195	1215	1215	1215
85	630	1270	1270	1265	1270	1270	1240
90	635	1340	1340	1300	1340	1340	1285
95	665	1380	1395	1385	1395	1395	1395
100	680	1435	1465	1445	1465	1465	1465
105	720	1510	1520	1505	1520	1520	1520
110	710	1550	1560	1560	1560	1560	1550
115	755	1595	1595	1580	1595	1595	1595
120	780	1635	1635	1635	1635	1635	1625
125	805	1655	1670	1665	1665	1670	1660
130	790	1680	1680	1680	1680	1680	1680

G6

Tmax	R	C	M	V	S	ALNS	EA
15	96	96	96	96	96	96	96
20	258	294	294	294	294	294	294
25	330	390	390	390	390	390	390
30	354	474	474	474	474	468	468
35	426	570	576	552	576	576	564
40	456	714	714	702	714	714	714
45	480	816	816	780	816	816	804
50	504	900	900	888	900	900	900
55	552	984	984	972	984	984	966
60	552	1044	1062	1062	1062	1062	1026
65	594	1116	1116	1110	1116	1116	1092
70	612	1176	1188	1188	1188	1188	1158
75	624	1224	1236	1236	1236	1236	1236
80	660	1272	1272	1260	1284	1284	1260

Tabelle 13: Erzielte Profite für Benchmark-Instanzen G5 und G6

Eine weitere Beobachtung ist, dass die bisher gefundenen besten Lösungen aus der Literatur zwar nachgestellt, aber weder durch den ALNS Algorithmus noch den evolutionären Algorithmus übertroffen werden konnten. Alle Algorithmen haben außerdem im Vergleich zu den per Zufall erzeugten Referenzwerten im Schnitt wesentlich bessere Ergebnisse und stellenweise sogar Ergebnisse mit 220% höherem Gesamtprofit erzeugen können.

6.6 Zusammenfassung der Ergebnisse

In dieser Abschlussarbeit wurde zunächst untersucht, inwiefern sich die Parametereinstellungen des evolutionären Algorithmus von Kobeaga et al. (2018) und des Adaptive Large Neighbourhood Search Algorithmus von Santini (2019) auf die Qualität der von ihnen erzeugten Lösungen auswirken. Dabei wurde festgestellt, dass abhängig von der Wahl der Parameterkombinationen die Qualität der Lösungen um bis zu 40% verbessert werden konnte. Die in dieser Arbeit gefundenen Parameterkombinationen, welche im Schnitt die besten Ergebnisse lieferten, gleichen den vorgeschlagenen Einstellungen der Autoren bzw. weichen nur stellenweise davon ab.

Außerdem wurde untersucht, inwiefern das Clustering eines Graphen sich auf die Performanz der Algorithmen bzw. auf die Qualität erzeugter Lösungen auswirkt. Der ALNS Algorithmus konnte bei Graphen mit stärkerem Clustering bessere Ergebnisse erzielen, als bei Graphen ohne Clustering. Eine Erklärung für dieses Verhalten kann der Umstand sein, dass der ALNS Algorithmus zwei Cluster-basierte Reparatur- und Zerstörmethoden besitzt, die bei Graphen mit mehreren Clustern besser abschneiden. In beiden bisher erwähnten Benchmarking-Tests unterlagen die vom ALNS Algorithmus erzeugten Lösungen den Lösungen, die durch den evolutionären Algorithmus gefunden wurden. Hierbei war die Differenz hinsichtlich der Qualität erzeugter Lösungen mit bis zu 3% jedoch verhältnismäßig gering.

Beim dritten Benchmarking-Test, in welchem beide Algorithmen gegen weitere Algorithmen aus der Literatur antraten, konnte der ALNS Algorithmus bis auf wenige Ausnahmen die besten Lösungen, die bisher gefunden wurden, nachstellen. Der evolutionäre Algorithmus schnitt in diesem Benchmarking inkonsistenter ab und konnte nur in rund der Hälfte der Fälle die besten, bisher gefundenen Lösungen erzeugen. Eine Erklärung für dieses Verhalten kann die relativ geringe Knotenanzahl der Benchmark-Instanzen sein, die zu einer frühzeitigen Erfüllung des Abbruchkriteriums führen können. Der evolutionäre Algorithmus wurde in diesem Benchmarking so konfiguriert, dass er terminiert, sobald ein Viertel der Lösungen seiner Population die gleichen Gesamtprofite (mit einer Abweichung von 1,5%) aufweisen wie die beste Lösung der Population. Diese Abweichung wurde eingeführt, um die Laufzeit des evolutionären Algorithmus für größere Graphen zu reduzieren. Ohne diese Abweichung konnte auch der evolutionäre Algorithmus konsistent die besten, bisher gefundenen Lösungen erzeugen. Sowohl die Algorithmen von Kobeaga et al. und Santini als auch die anderen Algorithmen aus der Literatur, die für das Benchmarking herangezogen wurde, konnten wesentlich bessere Ergebnisse liefern, als es mit einem rein zufälligen Verfahren möglich war.

7 Fazit und Ausblick

Das Orientierungsproblem ist ein NP-schweres Optimierungsproblem, welches sowohl das NP-vollständige Rucksackproblem als auch das Problem des Handlungsreisenden umfasst. Für das Orientierungsproblem wurde bislang kein deterministischer Algorithmus gefunden, mit welchem aus der faktoriell wachsenden Lösungsmenge die exakte Lösung des Problems in Polynomialzeit bestimmt werden kann. Die Entwicklung bzw. Entdeckung eines solchen Algorithmus hätte tiefgehende Folgen für diverse Anwendungsfelder, wie z. B. der Kryptographie, und würde implizieren, dass die Problemklasse P identisch zur Problemklasse NP ist. Solch eine Entdeckung würde also bedeuten, dass es auch für andere, bislang als zu komplex erachtete Probleme deterministische Algorithmen geben müsste, mit denen die Probleme in Polynomialzeit gelöst werden können.

In der Praxis werden aufgrund der Komplexität der Probleme und der bislang vergeblichen Suche nach solchen Algorithmen zur Lösung solcher Probleme heuristische Verfahren herangezogen, mit denen die exakten Lösungen approximiert werden können. Diese Algorithmen können sich unter anderem in ihrer Laufzeit und in der Qualität ihrer erzeugten Lösungen unterschieden. Als heuristische Algorithmen wurden in dieser Abschlussarbeit sowohl der Adaptive Large Neighbourhood Search Algorithmus von Santini (2019) als auch der evolutionäre Algorithmus von Kobeaga et al. (2018) vorgestellt, implementiert und analysiert.

Die Qualität der von diesen Algorithmen erzeugten Lösung hängt insbesondere von ihrer Konfiguration bzw. von den gewählten Parametern ab. So konnte die Parameterwahl die Qualität der Lösungen um bis zu 40% verbessern. Die Form der zugrundeliegenden Graphen bzw. die Stärke der Clusterbildung hatte im Vergleich dazu wesentlich geringere Auswirkungen auf die Performanz der Algorithmen. Beide Algorithmen wurden außerdem anhand von sechs Benchmark-Instanzen mit zehn weiteren Algorithmen aus der Literatur verglichen. Bei diesem Benchmarking hat der ALNS Algorithmus von Santini (2019) konsistent die besten Ergebnisse liefern können. Die vom evolutionären Algorithmus erzeugten Lösungen unterlagen in knapp der Hälfte der Fälle den Lösungen anderer Algorithmen. Ein Grund für dieses Verhalten kann die verhältnismäßig geringe Knotenanzahl der Benchmark-Instanzen sein, durch welches der evolutionäre Algorithmus oftmals vorzeitig beendet wird. Bei größeren Benchmark-Instanzen schnitt der evolutionäre Algorithmus hingegen besser als der ALNS Algorithmus ab und konnte Lösungen mit durchschnittlich 3% höheren Gesamtprofiten erzeugen.

Weil die Entdeckung eines deterministischen Algorithmus zur exakten Lösung des Orientierungsproblems in Polynomialzeit nicht wahrscheinlich scheint, muss auch in Zukunft an pragmatischen Herangehensweisen geforscht werden, um geeignete Lösungen für diverse Anwendungsfälle im Bereich der Tourenplanung effizient finden zu können.

8 Literaturverzeichnis

Baeldung 2021a — Baeldung (2021a): Intro to JaCoCo, Baeldung, [online] https://www.baeldung.com/jacoco [abgerufen am 17.02.2022].

Baeldung 2021b — Baeldung (2021b): P, NP, NP-Complete and NP-Hard Problems in Computer Science, Baeldung, [online] https://www.baeldung.com/cs/p-np-np-complete-np-hard [abgerufen am 17.02.2022].

BEVH 2021 — BEVH (2021): Onlinehandel mit Waren wächst im ersten Halbjahr 2021 um 23,2 Prozent, Bundesverband E-Commerce und Versandhandel Deutschland e.V. (bevh), [online] https://www.bevh.org/presse/pressemitteilungen/details/onlinehandel-mit-waren-waechst-im-ersten-halbjahr-2021-um-232-prozent.html [abgerufen am 17.02.2022].

Briskorn 2019 — Briskorn, Dirk (2019): *Operations Research: Eine (möglichst) natürlichsprachige und detaillierte Einführung in Modelle und Verfahren*, 1. Aufl. 2020, Berlin, Deutschland: Springer Gabler.

Caccetta/Kulanoot 2001 — Caccetta, Louis/Araya Kulanoot (2001): Computational aspects of hard Knapsack problems, in: *Nonlinear Analysis*, Bd. 47, Nr. 8, S. 5547–5558, [online] doi:10.1016/s0362-546x(01)00658-7.

Chao et al. 1996 — Chao, I-Ming/Bruce L. Golden/Edward A. Wasil (1996): A fast and effective heuristic for the orienteering problem, in: *European Journal of Operational Research*, Bd. 88, Nr. 3, S. 475–489, [online] doi:10.1016/0377-2217(95)00035-6.

Croes 1958 — Croes, G. A. (1958): A Method for Solving Traveling-Salesman Problems, in: *Operations Research*, Bd. 6, Nr. 6, S. 791–812, [online] doi:10.1287/opre.6.6.791.

Dantzig/Ramser, 1959 — Dantzig, G. B./J. H. Ramser (1959): The Truck Dispatching Problem, in: *Management Science*, Bd. 6, Nr. 1, S. 80–91, [online] doi:10.1287/mnsc.6.1.80.

Dantzig et al. 1954 — Dantzig, G. B./R. Fulkerson/S. Johnson (1954): Solution of a Large-Scale Traveling-Salesman Problem, in: *Journal of the Operations Research Society of America*, Bd. 2, Nr. 4, S. 393–410, [online] doi:10.1287/opre.2.4.393.

Eclipse Foundation o. D. — Eclipse Foundation (o. D.): WindowBuilder, Eclipse, [online] https://www.eclipse.org/windowbuilder/ [abgerufen am 17.02.2022].

Ester et al. 1996 — Ester, Martin/Hans-Peter Kriegel/Jörg Sander/Xiaowei Xu (1996): A density-based algorithm for discovering clusters in large spatial databases with noise, in: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, S. 226–231, [online] doi:10.5555/3001460.3001507.

Fischetti et al. 1998 — Fischetti, Matteo/Juan José Salazar González/Paolo Toth (1998): Solving the Orienteering Problem through Branch-and-Cut, in: *INFORMS Journal on Computing*, Bd. 10, Nr. 2, S. 133–148, [online] doi:10.1287/ijoc.10.2.133.

Flood 1956 — Flood, Merrill M. (1956): The Traveling-Salesman Problem, in: *Operations Research*, Bd. 4, Nr. 1, S. 61–75, [online] doi:10.1287/opre.4.1.61.

Gamma et al. 1995 — Gamma, Erich/Ralph E. Johnson/Richard Helm/John Vlissides (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, USA: Addison-Wesley.

Garey/Johnson 1979	Garey, Michael R./David S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York City, USA: W. H. Freeman and Company.
Golden et al. 1987	Golden, Bruce L./Larry Levy/Rakesh Vohra (1987): The orienteering problem, in: *Naval Research Logistics*, Bd. 34, Nr. 3, S. 307–318, [online] doi:10.1002/1520-6750(198706)34:3.
JUnit Community 2021	JUnit Community (2021): JUnit – Frequently Asked Questions, JUnit, [online] https://junit.org/junit4/faq.html [abgerufen am 17.02.2022].
Karp 1972	Karp, Richard M. (1972): Reducibility among Combinatorial Problems, in: *Complexity of Computer Computations*, S. 85–103, [online] doi:10.1007/978-1-4684-2001-2_9.
Kobeaga et al. 2018	Kobeaga, Gorka/María Merino/Jose A. Lozano (2018): An efficient evolutionary algorithm for the orienteering problem, in: *Computers & Operations Research*, Bd. 90, S. 42–59, [online] doi:10.1016/j.cor.2017.09.003.
Krumke/Noltemeier 2012	Krumke, Sven Oliver/Hartmut Noltemeier (2012): *Graphentheoretische Konzepte und Algorithmen: Mit 90 Aufgaben und Online-Service (Leitfaden der Informatik)*, 3. Aufl., Wiesbaden, Deutschland: Vieweg+Teubner Verlag.
Langr et al. 2015	Langr, Jeff/Andy Hunt/Dave Thomas (2015): *Pragmatic Unit Testing in Java 8 With Junit*, Sebastopol, USA: O'Reilly.
Machaka 2022	Machaka, Mahmoud (2022): GitHub - mmachaka/orienteering-problem, GitHub, [online] https://github.com/mmachaka/orienteering-problem [abgerufen am 17.02.2022].
Mathews 1896	Mathews, G. B. (1896): On the Partition of Numbers, in: *Proceedings of the London Mathematical Society*, Bd. s1-28, Nr. 1, S. 486–490, [online] doi:10.1112/plms/s1-28.1.486.
Matt 2021	Matt (2021): False positive & false negative in software testing, Testfully, [online] https://testfully.io/blog/false-positive-false-negative/ [abgerufen am 17.02.2022].
Mladenović/ Hansen 1997	Mladenović, Nenad/Pierre Hansen (1997): Variable neighborhood search, in: *Computers & Operations Research*, Bd. 24, Nr. 11, S. 1097–1100, [online] doi:10.1016/s0305-0548(97)00031-2.
Oracle 2021	Oracle (2021): Java Platform, Standard Edition Tools Reference, Oracle, [online] https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html [abgerufen am 17.02.2022].
Oracle 2022	Oracle (2022): ArrayList (Java Platform SE 8), Oracle, [online] https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html [abgerufen am 17.02.2022].
Ropke/Pisinger 2006	Ropke, Stefan/David Pisinger (2006): An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows, in: *Transportation Science*, Bd. 40, Nr. 4, S. 455–472, [online] doi:10.1287/trsc.1050.0135.
Santini 2019	Santini, Alberto (2019): An adaptive large neighbourhood search algorithm for the orienteering problem, in: *Expert Systems with Applications*, Bd. 123, S. 154–167, [online] doi:10.1016/j.eswa.2018.12.050.
Sevkli/Sevilgen 2010	Sevkli, Zulal/F. Erdogan Sevilgen (2010): Discrete Particle Swarm Optimization for the Orienteering Problem, in: *IEEE Congress on Evolutionary Computation*, S. 1–8, [online] doi:10.1109/cec.2010.5586532.

Szczukocki 2020	Szczukocki, Denis (2020): Multi-Module Project with Maven, Baeldung, [online] https://www.baeldung.com/maven-multi-module [abgerufen am 17.02.2022].
The Apache Software Foundation 2021	The Apache Software Foundation (2021): Maven – Introduction to the Build Lifecycle, Maven, [online] https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html [abgerufen am 17.02.2022].
Tsiligirides 1984	Tsiligirides, Theodore A. (1984): Heuristic Methods Applied to Orienteering, in: *Journal of the Operational Research Society*, Bd. 35, Nr. 9, S. 797–809, [online] doi:10.1057/jors.1984.162.
Vansteenwegen/ Gunawan 2019	Vansteenwegen, Pieter/Aldy Gunawan (2019): *Orienteering Problems: Models and Algorithms for Vehicle Routing Problems with Profits*, 1. Aufl., Cham, Schweiz: Springer.
Vansteenwegen et al. 2009	Vansteenwegen, Pieter/Wouter Souffriau/Greet Vanden Berghe/Dirk Van Oudheusden (2009): A guided local search metaheuristic for the team orienteering problem, in: *European Journal of Operational Research*, Bd. 196, Nr. 1, S. 118–127, [online] doi:10.1016/j.ejor.2008.02.037.
Vossen/Witt, 2016	Vossen, Gottfried/Kurt-Ulrich Witt (2016): *Grundkurs Theoretische Informatik: Eine anwendungsbezogene Einführung - Für Studierende in allen Informatik-Studiengängen*, 6. Aufl., Wiesbaden, Deutschland: Springer Vieweg.
Wang et al. 1995	Wang, Qiwen/Xiaoyun Sun/Bruce L. Golden/Jiyou Jia (1995): Using artificial neural networks to solve the orienteering problem, in: *Annals of Operations Research*, Bd. 61, Nr. 1, S. 111–120, [online] doi:10.1007/bf02098284.
Whitley et al. 1989	Whitley, L. Darrell/Timothy Starkweather/D'Ann Fuquay (1989): Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator, in: *Proceedings of the 3rd International Conference on Genetic Algorithms*, S. 133–140, [online] doi:10.5555/645512.657238.
Witt 2013	Witt, Kurt-Ulrich (2013): *Elementare Kombinatorik für die Informatik: Abzählungen, Differenzengleichungen, diskretes Differenzieren und Integrieren*, Wiesbaden, Deutschland: Springer Vieweg.
Wren/Holliday 1972	Wren, Anthony/Alan Holliday (1972): Computer Scheduling of Vehicles from One or More Depots to a Number of Delivery Points, in: *Operational Research Quarterly (1970–1977)*, Bd. 23, Nr. 3, S. 333, [online] doi:10.2307/3007888.