MASTERARBEIT | MASTER'S THESIS

Titel | Title

Tourenplanung mit realen Distanzen und Fahrzeiten im Vergleich zu euklidischen Distanzen

verfasst von | submitted by

Ahmet Burak Turgut BSc (WU)

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 915
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Betriebswirtschaft
Betreut von | Supervisor:	Univ.-Prof. Mag. Dr. Karl Franz Dörner Privatdoz.

Kurzfassung

Diese Masterarbeit befasst sich mit der Thematik TSP, präziser ausgedrückt VRP (eng. Vehicle Routing Problem). Hierbei handelt es sich um das Problem der Routenplanung eines Fahrzeugs, das eine große Lücke in der Praxis aufweist. Auch in der Supply Chain Management spielt das VRP eine bedeutsame Rolle. In diesen Zusammenhang zielt die Masterarbeit mit diversen Algorithmen daraufhin ab, komplexe Tourenplanungen umfassend durchzuführen. Dabei werden reale Distanzen und Fahrzeiten mit euklidischer Distanz verglichen. Die realen Daten werden durch die Bing Maps API abgerufen. Vertiefend zu dieser Thematik werden unterschiedliche Algorithmen verwendet, um die Auswirkung der Auswahl von Algorithmen auf die Routenplanung zu eruieren. Folgende Algorithmen werden angewandt: Nearest Insertion, Cheapest Insertion, Saving Algorithmus und genetischer Algorithmus. Für den genetischen Algorithmus wurden zwei verschiedene Verbesserungsansätze entwickelt, wobei das Resultat der Routenplanung mit der Bing Maps API besser abgeschnitten hat. Resümierend hat die Arbeit bewiesen, dass der genetische Algorithmus mit der Verwendung der entwickelten Optimierungsansätzen und in Kombination mit der Bing Maps API übergreifend die bessere Routenplanung generiert.

Inhaltsverzeichnis

Kurzfassung II
Inhaltsverzeichnis III
Abbildungsverzeichnis V
Tabellenverzeichnis VII
Formelverzeichnis VII
1. Einleitung 1
2. Theoretischer Input 3
2.1. TSP 4
2.2. VRP 6
2.3. Nearest Insertion 8
2.4. Cheapest Insertion 10
2.5. Clarke-Wright Saving Algorithmus 13
2.6. Genetischer Algorithmus 16
2.6.1. Initialisierung 18
2.6.2. Fitness 18
2.6.3. Selektion 19
2.6.4. Crossover 22
2.6.5. Mutation 24
3. Bing Maps 27
3.1. Besonderheiten der Bing Maps API in Bezug auf LKW-Routenplanung 27
3.2. Methodischer Ansatz für die Verbindung an Bing Maps API 29
3.3. Einschränkungen und Herausforderungen 30
4. Programm 32
4.1. Nearest Insertion 32
4.2. Cheapest Insertion 33
4.3. Saving Algorithmus 35

4.4. Genetischer Algorithmus 45
4.4.1. Herkömmlicher genetischer Algorithmus 46
4.4.2. Verbesserungsansatz 1 56
4.4.3. Verbesserungsansatz 2 57
4.5. Bing Maps API Verbindung 60
4.6. HTML und JavaScript-Datei für die Visualisierung der Routenplanung 65
4.7. VBA-Code für die Auswertung der HTML-Dateien 67
5. Ergebnisse & Diskussion 69
6. Conclusio und Ausblick 74
7. Literaturverzeichnis 76
8. Appendix 80

Abbildungsverzeichnis

Abbildung 1 - Nearest Insertion für VRP mit Kapazitätsbeschränkung 8
Abbildung 2 - Cheapest Insertion mit Kapazitätsbeschränkung 10
Abbildung 3 - Flussdiagram von genetischem Algorithmus 17
Abbildung 4 - Beispiel für Roulette Wheel Selektion 20
Abbildung 5 - Ablauf einer Tournament-Selektion 21
Abbildung 6 - Ablauf des geordneten Crossover 23
Abbildung 7 - Universell-Crossover 24
Abbildung 8 - Swap-Mutation 25
Abbildung 9 - Invers-Mutation 26
Abbildung 10 - Asynchroner Verbindungsaufbau zu Bing Maps API mit GET-Methode 29
Abbildung 11 - Nearest Insertion: Erzeugung von Tourliste 32
Abbildung 12 - Nearest Insertion: Routenplanungsprozess 33
Abbildung 13 - Cheapest Insertion: Erstellung einer Tour 34
Abbildung 14 - Cheapest Insertion: Bestimmung der Position und des Knotenpunktes 35
Abbildung 15 - Erzeugung von einer Saving Liste 36
Abbildung 16 - Saving Algorithmus: Abruf von SavingsMap 36
Abbildung 17 - Saving Algorithmus: Zuordnung im Fall A 37
Abbildung 18 - Saving Algorithmus: Zuordnung im Fall B – X-Knotenpunkt bekannt 37
Abbildung 19 - Saving Algorithmus: Zuordnung im Fall B – Y-Knotenpunkt bekannt 38
Abbildung 20 - Saving Algorithmus: Zuordnung im Fall C-1 39
Abbildung 21 - Saving Algorithmus: Zuordnung im Fall C-2 40
Abbildung 22 - Saving Algorithmus: Zuordnung im Fall C-3 41
Abbildung 23 - Saving Algorithmus: Zuordnung im Fall C-4 41
Abbildung 24 - Saving Algorithmus: Zuordnung im Fall C-5 42
Abbildung 25 - Saving Algorithmus: Zuordnung im Fall C-6 42
Abbildung 26 - Saving Algorithmus: Zuordnung im Fall C-7 43
Abbildung 27 - Saving Algorithmus: Zuordnung im Fall C-8 43
Abbildung 28 - Saving Algorithmus: Zuordnung im Fall C-9 44
Abbildung 29 - Saving Algorithmus: Löschung von überflüssigen Touren 45
Abbildung 30 - Genetische Algorithmus: Initialisierungsparameter 46
Abbildung 31 - Genetischer Algorithmus: Initialisierung 1 47
Abbildung 32 - Genetischer Algorithmus: Initialisierung 2 48

Abbildung 33 - Genetischer Algorithmus: Fitnessberechnung 48
Abbildung 34 - Genetischer Algorithmus: Entfernungsberechnung 49
Abbildung 35 - Genetische Entfernung - Initialisierung Funktion 49
Abbildung 36 - Genetischer Algorithmus: RouletteSelection 50
Abbildung 37 - Genetischer Algorithmus: TournamentSelection 50
Abbildung 38 - Genetischer Algorithmus: Ordered Crossover 51
Abbildung 39 - Genetischer Algorithmus: Universal Crossover 52
Abbildung 40 - Genetischer Algorithmus: Inverse Mutation 53
Abbildung 41 - Genetischer Algorithmus: Swap Mutation 53
Abbildung 42 - Genetischer Algorithmus: Löschung von Depots und Erzeugung einer String 54
Abbildung 43 - Genetischer Algorithmus: Wieder Erzeugung von Routenplanung 55
Abbildung 44 - Genetischer Algorithmus: Abspeicherung von der Elite-Chromosome 55
Abbildung 45 - Versbesserungsansatz durch Mischung von Funktionen des genetischen Algorithmus 56
Abbildung 46 - Verbesserungsansatz: Schwellwertbestimmung 57
Abbildung 47 - Verbesserungsansatz: Deep-Dive und Einspeisung von Routenplanung 57
Abbildung 48 - Verbesserungsansatz: Deep-Dive Unterprogramm 58
Abbildung 49 - Verbesserungsansatz: Speicherung von der besten Population 59
Abbildung 50 - Bing Maps Key 60
Abbildung 51 - Abwicklung von Entfernungsabfrage 61
Abbildung 52 - Regelwerk für die Kommunikation mit dem API 62
Abbildung 53 - Erstellung von der URL 62
Abbildung 54- Erstellung von Distanzmatrix und Fahrzeitmatrix 63
Abbildung 55 - Erzeugung einer CSV-Datei 64
Abbildung 56 - Überschreibung der JavaScript-File mit den Routenplanungsdaten 65
Abbildung 57 - Erstellung der HTML-Datei 66
Abbildung 58 - VBA Code Teil 1 67
Abbildung 59 - VBA Code Teil 2 68
Abbildung 60 - Ergebnisse der Routenplanungen mit dem herkömmlichen genetischen Algorithmus 70
Abbildung 61 - Ergebnisse der Routenplanungen mit dem Verbesserungsansatz 1 des genetischen Algorithmus 70

Abbildung 62 - Ergebnisse der Routenplanungen mit dem Verbesserungsansatz 2 des genetischen Algorithmus 71
Abbildung 63 - Ergebnisse der Routenplanungen mit den Verbesserungsansätzen 1 und 2 des genetischen Algorithmus 71
Abbildung 64 –Visualisierung der Routenplanung 1 80
Abbildung 65 - Visualisierung der Routenplanung 2 80
Abbildung 66 - Beispielhafter Visualisierung der Routenplanung 3 81
Abbildung 67 - Beispielhafter Visualisierung der Routenplanung 4 81

Tabellenverzeichnis

Tabelle 1 - Berechnungsdauer von TSP mit n Knotenpunkte und asymmetrische Entfernung . 6
Tabelle 2 - Distanzmatrix mit Kapazitätsverbrauch 9
Tabelle 3 - Berechnete Ersparnisse aus Tabelle 2 15

Formelverzeichnis

Formel 1 - Berechnung aller möglichen Rundreisen (asymmetrisch) (Zakir, 2010) 5
Formel 2 - Berechnung aller möglichen Rundreisen (symmetrisch) (Zakir, 2010) 5
Formel 3 - Formel für die Verlängerung der Tour 11
Formel 4 - Ersparnisformel 13
Formel 5 - Fitnessformel 19

1. Einleitung

In der Zeit der Erderwärmung und Ressourcenknappheit wird die Flottenmanagement und Routenplanungen zunehmend ein wichtiger Bestandteil der nachhaltigen Versorgung. Um einen Beitrag für die Zukunft zu leisten, widmet sich diese Masterarbeit der Routenplanungen mittels Bing Maps. Diese Arbeit bestrebt, dass die zukünftige Lösung in Routenplanungen effizienter und nachhaltiger wird. In diesem Sinne wird die Routenplanung von Bing Maps mit Routenplanung durch euklidische Distanz verglichen. Der Unterschied zwischen den realen Distanzen und Fahrzeiten mit der euklidischen Distanz wird verglichen und die unterschiedlichen Merkmale hervorgehoben.

Die Arbeit befasst sich mit einem realen Problem. Dieses Problem umfasst die Glas-Sammelstellen von MA48 in Wien. Aus der Datenbank herausgelesen gibt es 2695 Sammelorte. Die Container sind flächendeckend in Wien positioniert. Aus diesen 2695 Sammelstellen wurde stichprobenmäßig 150 Glas-Sammelstellen ausgesucht. Die 150 Sammelstellen erstrecken sich in alle Stadtteile. Außerdem können diese Container ca. 880 Kg an Glas sammeln. Ein MA48 LKW, der ein Gesamtgewicht von 33.000 Kg hat, kann ca. 16.300 Kg an Glas recyceln. Angesicht dessen kann ein LKW nur 18 Standorte besuchen, wenn der Ausgangspunkt die volle Mengenanzahl der Container ist. Hierbei ergibt sich das Problem, wenn die zu sammelnden Stellen größer als 18 sind, muss eine neue Tour in der Routenplanung eingeplant werden (Stadt-Wien, 2020).

Die Umsetzung von den Routenplanungen findet in der Programmiersprache C#, JavaScript und VBA statt. Die Programmiersprache C# baut die Verbindung zu dem Bing Maps API auf und berechnet mit verschiedenen Algorithmen die Tourenplanungen aus. Diese Algorithmen sind Nearest Insertion, Cheapest Insertion, Saving Algorithmus und genetischer Algorithmus. Dieses API verfügt zahlreiche Funktion für die Routenberechnungen. Dies wird ausführlich im Kapitel Bing Maps präsentiert. Die Programmiersprache JavaScript dient auch für die Verbindung bzw. Abfrage von der Bing Maps API. Das JavaScript-Programm wird im Nachhinein in eine HTML-Datei implementiert, um die Routenplanungen auch visuell darzustellen. Die dritte Programmiersprache VBA wird in Excel programmiert. Dies unterstützt die Auswertung aller ausgeführten Routenplanungen. Dabei werden 1536 Touren bzw. HTML-Dateien ausgewertet und in einem Tabellenblatt je nach Algorithmus und Version zusammengefasst.

Die Analyse des Vergleichs wird mit unterschiedlicher Anzahl an Knotenpunkte durchgeführt. Der Betrachtungsraum von den Knotenpunkten beginnt mit 25 und endet mit 150. Es wird in 25-iger Schritten verglichen. In anderen Worten ist die Anzahl an Knotenpunkte: 25, 50, 75, 100, 125 und 150. Dies dient auch dazu, die Varianz zwischen der einzelnen Betrachtungsräume und Algorithmen zu vergleichen. Die Routenplanung mancher Algorithmen weisen verschiedene Ergebnisse zur Anzahl der Kontenpunkte auf. Je größer es wird, desto mehr differenzieren sich die Ergebnisse der einzelnen Algorithmen. Deshalb zielt die Arbeit einerseits auf die Auswertung von verschiedenen Algorithmen und deren Auswirkung bei verschiedener Anzahl an Knotenpunkte ab. Anderseits wird die Berechnung von realen Distanzen und Fahrzeiten mit euklidischen Distanzen in Vergleich gezogen.

Dies zufolge werden in der Arbeit folgende Fragen beantwortet:

- Inwiefern bietet die Bing Maps Routenplanung im Vergleich zu euklidischer Distanz Vorteile?
- Woran unterschieden sich die einzelnen Algorithmen voneinander?
- Wie viel an CO2 Ausstoß wird durch die bessere Routenplanung gespart?

In dieser Hinsicht wird zu Beginn der Masterarbeit die Literatur von den einzelnen Algorithmen enthüllt. Dabei werden auch die einzelnen Werkzeuge vom genetischen Algorithmus präsentiert, die in der Umsetzung verwendet werden. Nach der Literaturrecherche wird die Bing Maps API vorgestellt. Währenddessen wird die Verbindung zu der API und weitere Instrumente jener vorgestellt. Weiterführend werden die Einschränkungen der API demonstriert. Darauffolgend wird die Umsetzung jeglicher Algorithmen in die jeweiligen Programmiersprachen vorgestellt, infolgedessen werden die Ergebnisse diskutiert. Zum Schluss werden die wichtigsten Erkenntnisse zusammengefasst und ein Ausblick wird für weitere Forschungsziele verschaffen.

2. Theoretischer Input

Die Literaturrecherche zielt darauf, einen umfangreichen Überblick auf die Themen Travel Salesmen Problem (TSP) und Vehicle Routing Problem (VRP) zu schaffen. Zunächst vertieft sich dieses Kapitel ins TSP. Dabei werden die Inhalte ausführlich erklärt. Folgend zu dem TSP findet die Beschreibung der Algorithmen, die für die Routenplanung verwendet wurden, statt. Dabei werden folgende vier Algorithmen herangezogen:

- Nearest Insertion
- Cheapest Insertion
- Saving Algorithmus
- Genetischer Algorithmus

Die obengenannten Algorithmen sind heuristische Verfahren. Ein heuristisches Verfahren ist eine Methodik, die für eine komplexe Fragestellung eine schnelle Lösung sucht. Diese muss nicht optimal sein, aber soll dennoch das Problem beseitigen. Für die akzeptable Lösungsfindung wird der Aufwand im Allgemeinen geringgehalten. Nach Hans Jürgen Zimmerman (1992) kann man heuristische Verfahren in fünf Unterpunkte gliedern:

1) ***Ausschluss potenzieller Lösung:***

   In den heuristischen Verfahren wird nicht das gesamte Lösungsspektrum untersucht, stattdessen wird nur ein Teil des Lösungsraum geforscht, um den Lösungsaufwand zur reduzieren. Währenddessen kann es vorkommen, dass bei den ausgeschlossenen Teilen des Lösungsraums die gesuchte bzw. optimale Lösung liegt. Deshalb garantiert das heuristische Verfahren nicht eine existierende optimale Lösung zu finden (Zimmermann, 1992).

2) ***Nicht-willkürliche Suchprozesse***

   Ein heuristisches Verfahren ist eine Methodik, die für eine komplexe Fragestellung eine schnelle Lösung sucht. Diese muss nicht optimal sein, aber soll dennoch das Problem beseitigen (Zimmermann, 1992).

3) ***Fehlende Lösungsgarantie:***

Die Konvergenz kann bei heuristischen Verfahren nicht gegen eine bestimmte Lösung verglichen und bewiesen werden, weil in den komplexen Problemen in einer überschaubaren Zeit entweder nicht die gleiche Lösung ermittelt wird oder die heuristischen Verfahren nicht dieselben Lösungen liefern (Zimmermann, 1992).

4) Subjektive Stopregelung:

Da die Eigenschaft Konvergenz fehlt und die meisten Algorithmen in einer dauerschleife (Loop) geraten, ist es wichtig eine Stopregel in den Verfahren einzubauen. Die Stopregelung kann je nach Art des Verfahrens, Grundlage des Verfahrens sowie Problemeigenschaft definiert werden. Häufig wird dieser Eigenschaft von den Verfahrensbenutzern bestimmt (Zimmermann, 1992).

5) Steuerungsmöglichkeiten:

Die Effizienz der Heuristik Verfahren ist überwiegend von problemstrukturabhängig. In den meisten Fällen kann keine Vorhersage in Bezug auf die Struktur des Problems und die Lösung dessen durch das heuristische Verfahren getroffen werden. Deshalb haben die Benutzer*innen eine Steuerungsmöglichkeit bei Verwendung der heuristischen Verfahren, um die Lösungsermittlung an die Struktur des Problems anzupassen. Die Steuermöglichkeit hat zwei positive Effekte auf das Nutzen des Verfahrens. Einerseits kann während des Lösungsverfahrens die Problemstruktur angepasst werden. Anderseits steigt oft die Akzeptanz des Verfahrens durch die Steuerung dessen seitens der Benutzer*innen (Zimmermann, 1992).

Zusammenfassend liegt das Hauptaugenmerk dieses Kapitel zunächst auf die Literaturrecherche über das Problem des Handelsreiseden (Travel Salesman Problem). Darauf aufbauend wird die Routenplanung für Fahrzeugflotte präsentiert (Vehicle Routing Problem). Die Masterarbeit widmet sich auf das VRP, wobei auch alle Algorithmen für die Lösung dieses Problem eingesetzt wird. Danach werden die Algorithmen bzw. Heuristiken präsentiert, die sich mit der Lösung des VRP beschäftigen.

2.1. TSP

Das TSP-Problem wird auch als Problem des Handelsreisenden genannt, welches versucht die günstigste Route zu planen. Dabei müssen alle Standorte besucht werden, wobei der Startpunkt

zum Schluss zurückgekehrt wird. Weiters dürfen alle Standorte nur einmal besucht werden. Die Folge der Besuche, die mit dem Startpunkt beginnt und dem Endpunkt aufhört, wird Tour genannt (Applegate, et al., 2011).

Das TSP-Problem ist einer der meistuntersuchten Optimierungsprobleme. Der Grund dahinter ist seiner Schwierigkeit und dem praktischen Interesse zuzuordnen. Es ist wichtig, dass in praktischen Anwendungen dieses Problem für dynamische Umgebungen und größere Anzahl an Standorte sehr effizient laufen soll. Deshalb werden zumeist gering zeitaufwendige Heuristiken angewendet. Wenn auch eine bestimmte Heuristik gute Performance bei vielen Instanzen vorzeigt, muss diese Anwendung nicht robust genug sein, um eine richtige Entscheidung in alle Situationen zu bilden. Demzufolge wird auch in der Masterarbeit verschiedene Instanzen benutzt, um zu überprüfen, ob die verwendeten Heuristiken und Algorithmen auch gesamtheitlich gut auftreten (Gil-Gala, et al., 2022).

In diesem Zusammenhang ist es noch wichtig die NP-schwer zu erwähnen. Das TSP-Problem weist NP-schwer auf. Angesichts dieser Tatsache ist es nicht immer möglich eine optimale Lösung zu finden. Das hängt von der Größe des Problems ab. Je größer das Problem wird, desto zeitaufwändiger ist das Finden der optimalen Lösung. Es ist anzumerken, dass auch für kleine Anzahl an Standorten die Routenplanung komplex ist. Für „n“ Standtorte müssen alle möglichen Kombinationen errechnet werden, damit eine optimale Sequenz gefunden wird. Die darunter stehenden Formeln dienen zur Berechnung aller möglichen Kombinationen (Bochtis & Sørensen, 2009) (Schorpp, 2011).

$$(n-1)!$$

Formel 1 - Berechnung aller möglichen Rundreisen (asymmetrisch) (Zakir, 2010)

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

Formel 2 - Berechnung aller möglichen Rundreisen (symmetrisch) (Zakir, 2010)

Die Formel 1 wird verwendet, wenn zum Beispiel „a“ nach „b“ nicht die gleiche Entfernung hat, wie „b“ nach „a“. Für die symmetrischen Entfernungen wird die Formel 2 verwendet. Das heißt, dass die Entfernungen zwischen „a“ nach „b“ und „b“ nach „a“ identisch sind.

Knoten-punkte	Prozessor Taktfrequenz	Routen-Möglichkeiten	Dauer (s)	Dauer (min)	Dauer (Std)	Tag	Monat	Jahr
1	3200000000	1	0	0	0	0	0	0
2	3200000000	1	0	0	0	0	0	0
4	3200000000	6	0	0	0	0	0	0
8	3200000000	5040	0	0	0	0	0	0
10	3200000000	362880	0	0	0	0	0	0
12	3200000000	39916800	0	0	0	0	0	0
14	3200000000	6227020800	2	0,03	0	0	0	0
16	3200000000	1,30767E+12	409	7	0,1	0	0	0
18	3200000000	3,55687E+14	111153	1853	30,9	1,3	0,04	0
20	3200000000	1,21645E+17	38014094	633568	10559	440	15	1,2

Tabelle 1 - Berechnungsdauer von TSP mit n Knotenpunkte und asymmetrische Entfernung

Die Tabelle 1 visualisiert die Berechnungsdauer von TSP, indem alle möglichen Kombinationen der Routenplanung ausgerechnet werden. Dabei sind die Entfernungen zwischen den beiden Standorten asymmetrisch. Die Taktfrequenz des Prozessors wurde mit 3,2 GHz angegeben, d.h., dass für 20 Standorte eine Berechnungsdauer von einem Jahr gebraucht wird, um eine optimale Lösung zu finden. Deshalb werden Algorithmen verwendet, damit die Zeit überbrückt werden kann. Diese Algorithmen liefern nicht immer eine optimale Lösung. Aufgrund dessen besteht die Notwendigkeit, ein Algorithmus mit verschiedenen Instanzen zu testen. Dabei sollte dieser Algorithmus auch mit verschiedenen Heuristiken verglichen werden, um die Leistungsfähigkeit des Algorithmus zu vergleichen.

2.2. VRP

Das Vehicle-Routing-Problem errechnet die günstigste Tour für eine bestimmte Anzahl an Standorten. Es ist eine ähnliche Vorgehensweise wie TSP. Bei dem VRP darf jeder Standort jeweils von einem Fahrzeug besucht werden. Weiters ist das Depot sowohl der Startort als auch der zuletzt besuchte Ort von der Route. Eine Einschränkung ist, dass die Kapazität des Fahrzeuges nicht während der Fahrt überstiegen werden darf (eng. Capacitated VRP – CVRP auch nur VRP genannt). Falls die Kapazität des Fahrzeuges überschritten ist, muss eine neue Tour in die Routenplanung geplant werden. Eine Erweiterung des TSP ist multiple TSP, indem mehrere Personen involviert sind, um den Kundenportfolio zu befriedigen. Dies ist derselbe

Ansatz, wie VRP. Der Unterschied liegt nur darin, dass bei TSP die Routenplanung für Personen durchgeführt wird und bei VRP für die Fahrzeuge (Bochtis & Sørensen, 2009).

Das VRP führt zu Dantzig und Ramser (1959) zurück, in dem eine effiziente Routenplanung für eine Tanklastwagenflotte durchgeführt wurde. Diese Flotte sollte aus der Raffinerie einzelne Tankstellen beliefern. Allmählich wurde dieser Ansatz von Clark und Wright (1964) neu definiert. Seitdem wird das VRP mit verschiedenen Ansätzen erweitert. Diese Ansätze beruhen auf unterschiedliche reale Probleme, wie öffentliche Verkehrsmittel, Transportation von Industrie Ware über die gesamte Supply Chain, Abfallsammlung oder Straßenbereinigung (Bochtis & Sørensen, 2009).

Das VRP kann neben Kapazitätseinschränkung noch andere Arten an Restriktionen haben. Eine dieser Restriktionen ist die maximale Länge der Route, sie wird als DCVRP (engl. distance constraind VRP) abgekürzt. Ein anderes Vorhaben wäre die Lieferung zu einem bestimmten Zeitfenster (engl. VRP with time windows – VRPTW). Eine weitere Kategorie ist, dass die Kunden einerseits neue Waren bestellen und anderseits die zu einem anderen Zeitpunkt gelieferte Waren wieder zurückschicken können (engl. VRP with pickup and delivering – VRPPD). Eine alternative Problemstellung zum VRPPD ist die VRPB. In der VRPB-Problemstellung müssen zuerst alle Lieferungen verteilt werden, danach können die Retourlieferungen abgeholt werden (engl. VRP with Backhouse – VRPB). Im Übrigen können die Standorte und zukünftige Bedarfe noch unbekannt (dynamische VRP- DVRP) oder wahrscheinlichkeitsbasiert bekannt sein (stochastische VRP – SVRP) (Bochtis & Sørensen, 2009).

In dieser Masterarbeit wird die Routenplanung auf das VRP basieren. Folgende Bedingungen müssen bei der Routenplanung mit VRP erfüllt werden (Kara, et al., 2008):

- Jede Route muss mit dem Depot anfangen und enden
- Jede/r Standort/Kund*in muss jeweils einmal in der Route der Routenplanung vorkommen
- In einer Route darf die Kapazität des Fahrzeuges nicht überstiegen werden
- Zielsetzung ist, dass die Kosten/Entfernung der Routenplanung minimisiert werden.

Resümierend ist es wichtig, die einzelnen Bedingungen bei dem VRP einzuhalten und die Routenplanung strikt nach diesen Bedingungen zu konzipieren. Diese Bedingungen werden in den Algorithmen definiert und bearbeitet. Die nächsten 4 Unterkapiteln stellen diese Algorithmen vor. Die Literaturrecherche der vier Algorithmen beschränkt sich auf das CVRP.

2.3. Nearest Insertion

Die Nearest Insertion Heuristik ist ein Greedy-Algorithmus. Ein Greedy-Algorithmus wählt schrittweise den Folgezustand aus, in dem je nach Zielfunktion die bestmögliche Option zum Zeitpunkt der Auswahl getroffen wird. Diese Algorithmen führen eine schnelle Auswertung durch, wobei sie aber nicht die optimale Lösung liefern. Weiters wird Nearest Insertion als ein Einfüge-Verfahren deklariert. Das Einfüge-Verfahren ergänzt schrittweise die nicht besuchten Standorte in die Tour, bis kein nicht besuchter Standort übriggeblieben ist oder eine Einschränkung (Kapazität, Distanz) nicht mehr zugelassen wird (Zimmermann, 1992).

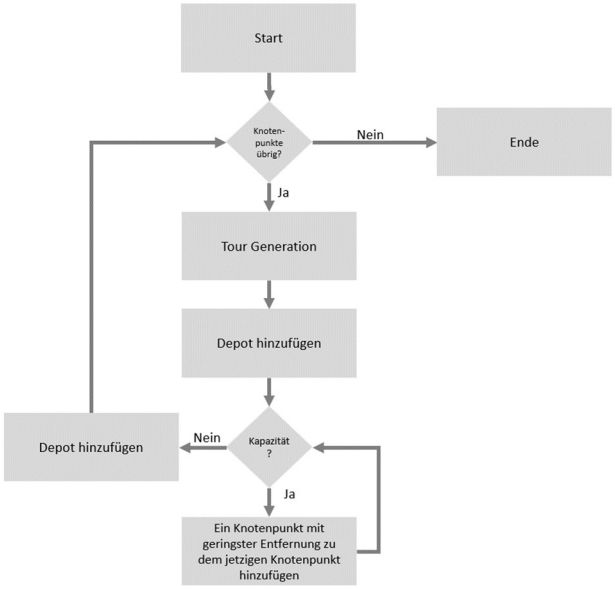


Abbildung 1 - Nearest Insertion für VRP mit Kapazitätsbeschränkung

Die Nearest Insertion Heuristik für die Tourenplanung von LKW mit Kapazitätsbeschränkung wird in der Abbildung 1 dargestellt. Am Anfang wird eine Distanzmatrix erzeugt, hierfür wird die Entfernung jeder Standortpaare berechnet. Danach wird es ermöglicht, mit Nearest Insertion Heuristik die Tourenplanung durchzuführen. Dafür setzt man die Startknoten und Endknoten mit dem Depot fest. Anschließend wird abgefragt, welches der Knotenpunkte, die in der errechneten Distanzmatrix, am nächsten zu dem Startpunkt ist. Falls jener erfolgreich gefunden wird, wird nochmal dieselbe Vorgehensweise für den ermittelten Knotenpunkt errechnet. Hieraus ist zu erschließen, dass der nächste Kontenpunkt zu dem ermittelten Knotenpunkt gesucht wird. Dies wird so lange durchgeführt, bis entweder die Kapazitätsbeschränkung übersteigt oder keine weiteren Knotenpunkte mehr vorhanden sind. Wenn die Kapazitätsbeschränkung übertroffen wurde, wird eine neue Tour geöffnet und dieselben

Schritte erneut durchgeführt. Das wiederholt sich so lange bis kein Knotenpunkt mehr vorhanden ist.

Die Nearest Insertion Heuristik wird anhand eines Beispiels näher erklärt. Dafür wird zuerst eine fünf mal fünf Distanzmatrix erzeugt. Dafür werden pseudo zufällige Zahlen als Entfernungsangabe vergeben. Die Entfernungen werden in der Tabelle 2 dargestellt.

d_{ij}	1	2	3	4	5	6
1	0	2	5	2	1	3
2	2	0	4	5	5	3
3	4	2	0	4	5	3
4	2	4	3	0	1	5
5	1	3	3	4	0	5
6	2	4	5	2	1	0
c_j	0	4	5	2	3	6

Tabelle 2 - Distanzmatrix mit Kapazitätsverbrauch

In der Tabelle 2 handelt sich um eine asymmetrische Distanzmatrix. Weiters ist die benötigte Kapazität c_j für die Fahrt nach Standort j angeführt. Die variable d_{ij} zeigt die Entfernung zwischen zwei Knotenpunkten bzw. Standorten. Weiters wird die Nutzlast mit sieben Tonnen beispielhaft angenommen. Hierbei muss erwähnt werden, dass der Startort sowie die Endposition die Nummer eins sind. Nachdem alle Angaben für den Lösungsvorgang gegeben ist, kann die Nearest Insertion Heuristik angewandt werden. Zu Beginn wird mit Standort eins gestartet. Als nächster Schritt würde der Standort fünf folgen, weil es die kleinste Entfernung hat. Dies wird so lange durchgeführt, bis die Kapazität des Fahrzeuges nicht mehr zulässt.
In diesem Fall wird folgende Tour erzeugt: „1-5-2-1“. Die Kapazität des Fahrzeuges ist hiermit voll ausgelastet und eine neue Tour muss in der Tourenplanung geöffnet werden. Wenn die Nearest-Insertion-Berechnung weitergeführt wird, führt dieses zu den folgenden Resultaten:

- T_1 = {1-5-2-1} - leere Kapazität: 0 Streckenlänge:1+3+2=6
- T_2 = {1-4-3-1} - leere Kapazität: 0 Streckenlänge:2+3+4=9
- T_3 = {1-6-1} - leere Kapazität: 1 Streckenlänge:3+2= 5
- Gesamtstreckenlänge: 20

Zusammenfassend löst die Nearest Insertion Heuristik sehr schnell das Problem, indem es bei jeder Position den nächsten naheliegenden Standort sucht. Von da an muss geklärt werden, ob die Kapazität des Fahrzeuges nicht mit der neuen Standortlieferung überstiegen wird. Die

Schwäche liegt darin, dass zum Schluss tendenziell nur noch die schlechten Verbindungen übrigbleiben und dadurch die Tourenplanung im Gesamtbild betrachtet verschlechtert wird.

2.4. Cheapest Insertion

Die Cheapest-Insertion-Heuristik ist wie die Nearest-Insertion-Heuristik eine Einfüge-Verfahren sowie der Greedy-Algorithmus. Die Cheapest-Insertion-Heuristik zielt daraufhin ab, dass bei der Tourenbildung immer der bestmögliche Standort an der besten Stelle der Tour eingefügt wird. Dafür wird bei jeder Einfüge-Stufe für alle offenen Standorte, die noch nicht zu einer Tour zugeordnet wurden, eine Entfernungsberechnung durchgeführt. Im Zuge dessen wird ermitteln, an welcher Stelle man den bestmöglichen Standort einfügen kann. Dieser Ablauf wird mittels eines Flussdiagramms in der Abbildung 2 veranschaulicht.

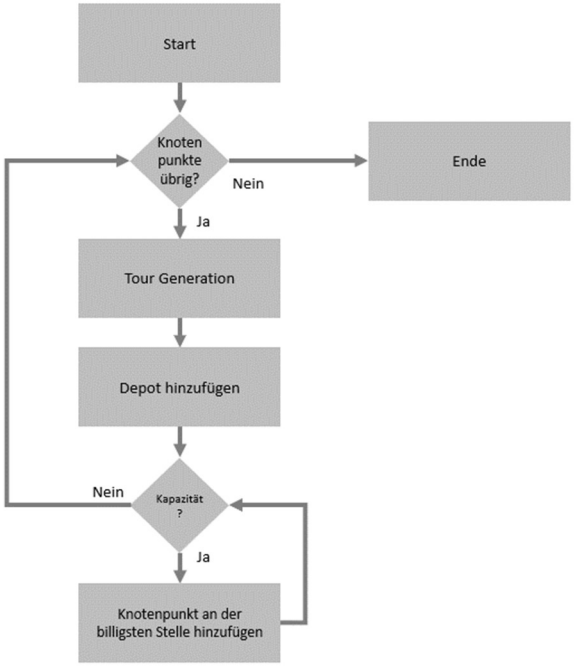


Abbildung 2 - Cheapest Insertion mit Kapazitätsbeschränkung

Da es sich um eine Kapazitätsbeschränkung der Heuristik handelt, wird zuerst nachgefragt, ob Standorte/Knotenpunkte noch offen sind. Wenn dies nicht der Fall ist, kann die Cheapest Insertion hiermit beendet werden. Falls es noch offene Stellen gibt, wird als ersteres eine neue Tour generiert. Nachdem die Tour erstellt wurde, wird in dieser Tour ein Startpunkt und Endpunkt ergänzt. Wie bereits erwähnt, ist der Startort ebenso der letzte besuchte Ort. Danach wird für die erstelle Tour eine Tourenplanung durchgeführt. Dabei wird geachtet, dass bei jedem eingefügten Standort die Kapazität nicht überstiegen wird. Die Berechnung wird für jede Stelle der bereits generierten Tour durchgeführt. Dies beginnt mit jener Stelle, die nach dem

Startort kommt und endet vor dem Endort. Für jede bestehende Stelle in der zu generierende Tour wird für alle Standorte, die noch nicht besucht worden sind, die Entfernung der Tour berechnet. In diesem Zusammenhang wird einerseits die Stelle von der Tour ermittelt, in der eingefügt werden sollte und andererseits werden die bis jetzt noch nicht besuchten Knotenpunkt selektiert. Somit steigt die Anzahl der zu berechnenden Stellen in der Tour. Im Gegensatz dazu sinkt die Anzahl von dem nicht besuchten Standorten, bei denen iteriert wurde. Das Programm wird so lange ausgeführt, bis entweder die Kapazität nicht mehr zulässt oder das Programm keine Standorte mehr findet, die eingefügt werden muss.

Die Funktionsweise der Cheapest Insertion Heuristik wird anhand eines Fallbeispiels näher erläutert. Die Tabelle 2 wird als Distanz-Matrix für die Berechnung von Cheapest Insertion verwendet. Die Abkürzung T_p steht für eine geschlossene Subtour, in dem nur ein Teil der Knotenpunkte aufgestellt werden. Weiters stellt k die nicht besuchten Knotenpunkte dar. Somit wird für jede Kante(i,j) in der Tp eine Verlängerungsberechnung durchgeführt, bei der zwischen i und j das Einfügen von k berechnet wird. Weiters wird die Kapazitätsbeschränkung wie bei Nearest Insertion als sieben Tonnen angenommen. Die verlängerte Tour wird also mit folgender Formel berechnet:

$$d_{ik} + d_{kj} - d_{ij}$$

Formel 3 - Formel für die Verlängerung der Tour

1: Berechnung für den ersten Standort, die mithilfe von Cheapest Insertion eingefügt wird.

T_1 = {1,2,1} => $d_{12}+d_{21}$= 2+2 = 4 C: 3 T_1= {1,3,1] => $d_{13}+d_{31}$= 5+4=9 C=2

T_1 = {1,4,1} => $d_{14}+d_{41}$= 2+2 = 4 C: 5 T_1 = {1,5,1] => $d_{15}+d_{51}$= 1+1=2 C=4

T_1 = {1,6,1} => $d_{16}+d_{61}$= 3+2 = 5 C: 1

In diesem Fall wird der Standort fünf an der Stelle 2 eingefügt.

2: Wegen Kapazitätsbeschränkungen kann nur Standort 2 oder 4 eingefügt werden.

T_1 = {1,2,5,1} => $d_{12}+d_{25}-d_{15}$= 2+5-1 = 6 C:0

T_1 = {1,5,2,1] => $d_{52}+d_{21}-d_{51}$=3+2-1=4 C:0

T_1 = {1,4,5,1} => $d_{14}+d_{45}-d_{15}$= 2+1-1 = 2 C:2

T_1 = {1,5,4,1] => $d_{54}+d_{41}-d_{51}$=4+2-1=5 C:2

T_1 = {1,4,5,1} => Subtour Streckenlänge: 2+1+1=4

Dies wird so lange durchgeführt, bis keine k mehr übrig sind.

Somit kommt man auf das folgende Ergebnis:

- T_1 = {1,4,5,1} => Subtour Streckenlänge: 2+1+1=4
- T_2 = {1,2,1} => Subtour Streckenlänge: 2+2=4
- T_3 = {1,3,1} => Subtour Streckenlänge: 5+4=9
- T_4 = {1,6,1} => Subtour Streckenlänge: 3+2=5
- Gesamtstreckenlänge: 22

Zusammengefasst fällt die Berechnung von Cheapest Insertion schlechter als die Berechnung von Kapitel 2.3 (Nearest Insertion) aus, wenn die Distanzmatrix von Tabelle 2 als Berechnungsbasis herangezogen wird. Die Ursache ist die Kapazitätsbeschränkung, weil hier vier Subtouren erstellt werden müssen, hingegen bei dem Nearest Insertion nur drei. Dennoch heißt es nicht, dass das Cheapest Insertion Verfahren konstant schlechter ausfällt als Nearest Insertion. Deshalb ist es wichtig die Algorithmen untereinander zu vergleichen, um das bestmögliche Ergebnis zu wählen.

Die Studie von Rosenkrantz et. al. (2009) zeigt, dass 50 Knotenpunkte mehrmals zufälligerweise in einem Bereich gestreut werden. Die Tourenplanung wurde mittels vier verschiedener Heuristiken (Nearest Insertion, Cheapest Insertion, Farthest Insertion und Nearest Neighbor) zur jeweiligen Streuung durchgeführt. Der Vergleich stellt dar, dass die Cheapest Insertion im Vergleich zu Farthest Insertion bis sieben Prozent bessere und bis zu 12 Prozent schlechtere Ergebnisse liefert. Dabei ist zu beachten, dass diese Heuristik die am weit entferntesten Standorte zuerst in die Toure einfügt. In dieser Studie wird betont, dass in dem Großteil der Tourenplanung die Farthest Insertion besser abschließt als die anderen drei Algorithmen (Rosenkrantz, et al., 2009).

Es ist wichtig zu untermauern, dass in diesem Abschnitt angeführtes Beispiel nichts über den Cheapest Algorithmus aussagt, sondern ausschließlich zur Demonstration jenes dient. Im Kapitel 5 wird der Cheapest Insertion Algorithmus mit drei anderen Algorithmen verglichen und infolgedessen werden die Ergebnisse präsentiert. Die Bewertung der Cheapest Insertion im Vergleich zu den anderen drei Algorithmen wird ebenso im Kapitel 5 stattfinden.

2.5. Clarke-Wright Saving Algorithmus

Der Clarke-Wright Saving Algorithmus (Kurzform Saving Algorithmus) ist einer von vielen namhaften Heuristiken. Wegen seiner Schnelligkeit, Einfachheit und schnellen Anpassungsfähigkeit lässt sich öfters in der realen Welt zur Anwendung kommen. Es findet auch einen Platz in der CVRP. Dieser Algorithmus errechnet anfänglich die Ersparnisliste. Für die Errechnung der Ersparnisliste wird der Entfernungsparameter benötigt. Die Ersparnisse werden mittels Verknüpfung von zwei verschiedenen Knotenpunkten berechnet. Dafür kommt die Formel 4 zur Anwendung (Çatay & Doyuran, 2011).

$$S_{ij} = c_{1i} + c_{j1} - c_{ij}$$

Formel 4 - Ersparnisformel

In der Formel 4 bildet „S_{ij}“ die Ersparnisse des Knotenpunktpaars „i-j“. Weiters stellt der Buchstabe „c“ die Kosten zwischen den Knotenpunkten „i“ und „j“ dar. Das Depot wird mit „1“ dargestellt. Danach wird die Ersparnisliste absteigend von der größten Ersparnis bis zur kleinsten Ersparnis sortiert. Somit kann die Routenplanung mit der Kapazitätsbeschränkung begonnen werden. In der Literatur kommen zwei Routenplanungsarten für die Saving Algorithmus vor. Einerseits kann die Routenplanung in Serie durchgeführt werden. Das heißt zuerst wird die erste Tour befüllt, bevor man mit der zweiten Tour anfängt. Das Abbruchkriterium ist die Kapazitätsbeschränkung. Bei der parallelen Routenplanung findet die Tourenzuordnung zeitgleich statt. Die Studie von Cordeau et al. (2002) widerlegt, dass die parallele Zuordnung bessere Ergebnisse liefert als die serielle Routenplanung. Der Saving Algorithmus ist auch ein Greedy-Algorithmus, die schrittweise seinen Folgezustand auswählt (Cordeau, et al., 2002).

Bei der Routenplanung mit dem Saving Algorithmus gibt es vier verschiedene Zuordnungsmöglichkeiten. Die Zuordnung findet mithilfe der Ersparnisliste nach der Größe statt. Diese Möglichkeiten werden in der nachstehenden Aufzählungspunkte aufgelistet. Mitunter werden Beispiele erwähnt, die die Zuordnungsmethoden veranschaulichen. Bei diesen Beispielen bildet der Buchstabe „x“ den ersten Knotenpunkt des Knotenpaars und „y“ den zweiten Knotenpunkt des Knotenpaars dar. Das Depot wird als eine „1“ dargestellt. Ein weiterer Anhaltspunkt ist, dass bei jeder Verknüpfungsart überprüft wird, ob die Kapazität der Tour mit dieser Zusammenführung nicht die Kapazitätsbeschränkungen überschreitet. Diese Beschränkung gilt bei der CVRP, welches auch in dieser Arbeit in Vordergrund steht. Bei der Zuordnung gilt:

- **Depot**:

  Bei dem Depot Zuordnung hat bis jetzt keiner der beiden Knotenpunktpaare eine Zuweisung hinsichtlich der Tour bekommen. Somit wird eine neue Tour für die beiden Knotenpaare geöffnet. Als Beispiel wird die Tour mit „1-x-y-1“ geöffnet.

- **x-Knotenpunkt**:

  In der x-Knotenpunkt Zuordnung ist der linke Knotenpunkt „x“ in einer Tour enthalten im Gegensatz zum rechten Knotenpunkt „y“, welcher bisher nicht zugeordnet wurde. Die Depots („1“) dienen als Orientierungspunkt, wie und wo die Knotenpunkte verknüpft werden können. Wenn zum Beispiel die Tour als „1-3-4-x-1 erstellt wurde, wird der Knotenpunkt „y“ neben „x“ platziert und lautet dann „1-3-4-x-y-1“. Falls zum Beispiel „x“ an der Mitte des Tours liegen würde (wie zum Beispiel „1-3-x-4-1“), könnte man dieses Knotenpaar von der Ersparnisliste nicht in die Tour einfügen. Somit wird das nächste Knotenpaar in der Ersparnisliste ausgewählt. Danach wird überprüft welche Art von Zuordnung es haben wird. Weiters ist die Verknüpfung möglich, wenn der Knotenpunkt an der zweiten stelle der Tour auftaucht. Somit kann die Tour wie „1-y-x-3-4-1“ gebildet werden.

- **y-Knotenpunkt:**

  Der Knotenpunkt „y“ kommt in einer Tour vor und der Knotenpunkt „x“ wurde bisher nicht zugeordnet. Hier gelten die gleichen Regelungen wie in der Zuordnung von x-Knotenpunkt. Zum Beispiel befindet sich der Knotenpunkt „y“ an der vorletzten Stelle „1-3-4-y-1“. In diesem Fall ist es möglich, dass der Knotenpunkt „x“ zwischen „y“ und „1“ platziert werden kann. Somit entsteht folgende Tour „1-3-4-y-x-1“.

- **Verknüpfung:**

  Die Verknüpfung kombiniert beide Knotenpunkte in einer Tour. In diesem Fall werden beide Touren in einer großen Tour vereint. Dabei wird auf die Kapazitätsbeschränkung geachtet. Falls es überschritten wird, wird das nächste Ersparnispaar ausgewertet. Als Beispiel kann jenes angeführt werden: die Tour des Knotenpunktes „x“ beträgt „1-x-6-7-2-1“ und die Tour von Knotenpunkt „y“ lautet „1-3-4-y-1“. In diesem Fall wird die Tour „1-3-4-y-x-5-6-2-1“ erstellt.

Die Zuordnungen erfolgen so lang, bis keine offenen Stellen mehr übrigbleiben und alle Knotenpunkte in der Routenplanung vorkommen. Die Funktionsweise des Saving Algorithmus wird anhand eines Fallbeispiels dargelegt. Dafür wird die Distanzmatrix von der Tabelle 2 in eine Ersparnisliste umgewandelt. Die absteigend sortierte Ersparnisstabelle ist in der Tabelle 3 ersichtlich. Anhand dieser Tabelle wird die Routenplanung mit Saving Algorithmus geplant.

Knotenpaar	Ersparnis	Knotenpaar	Ersparnis
3-2	5	2-6	1
3-6	4	3-5	1
3-4	3	6-2	1
4-3	3	4-2	0
6-4	3	5-2	0
6-5	3	2-4	-1
2-3	2	4-6	-1
4-5	2	5-4	-1
5-3	2	2-5	-2
6-3	2	5-6	-2

Tabelle 3 - Berechnete Ersparnisse aus Tabelle 2

Nachdem die Ersparnistabelle erstellt wurde, kann die Routenplanung durchgeführt werden. Bevor die Routenplanung beginnt, ist noch erwähnenswert, dass die Kapazitätsbeschränkung wie bei Nearest Insertion und Cheapest Insertion auch als sieben Tonnen angenommen wird. Die Routenplanung beginnt mit den Knotenpaaren „3-2“. Für zwei Knoten wird eine Kapazität von neun Tonnen benötigt. Das Problem besteht nun darin, dass diese über die Kapazitätsgrenze

von dem festgelegten Wert sieben Tonnen ist. Die nächste Paarung kann auch wegen der Kapazitätsbeschränkung nicht erfüllt werden. Nur durch die Knotenpaarung „3-4“ kann eine Tour erzeugt werden. Daraus ergibt sich die Tour „1-3-4-1“ und erreicht somit die volle Kapazität sieben, womit diese Tour abgeschlossen ist. Wie bereits beschrieben. folgt anschließend die Auswertung bis keine offenen Knotenpunkte zu Verfügung stehen.

Das Ergebnis sieht folgend aus:

- T_1 = {1-3-4-1} => Kapazität:0 Streckenlänge = 5+4+2 = 11
- T_2 = {1-5-2-1} => Kapazität:0 Streckenlänge = 1+3+2 = 6
- T_3 = {1-6-1} = Kapazität:1 Streckenlänge = 3+2 = 5
- Gesamtstreckenlänge: 22

Die Knotenpaare „3-4“ sowie „4-3“ haben den gleichen Ersparniswert, aber eine andere Streckenlänge. Da das Knotenpaar „3-4“ früher in der Ersparnisliste vorkommt, fällt die Gesamtstreckenlänge mit 22 aus. Dieser kleine Unterschied lässt die Gesamtstreckenlänge verglichen mit Nearest Insertion um zwei verlängern. Die restlichen Touren in dem Saving Algorithmus fallen gleich aus wie in der Nearest Insertion. Dieses Beispiel besagt nicht, dass Saving Algorithmus schlechter ist als die Nearest Insertion Algorithmus, sondern demonstriert vielmehr die Berechnung der Heuristik.

Die wissenschaftliche Arbeit von Cordeau et al. (2002) gibt an, dass viele verschiedene Anpassungen im Bereich Sortierung (Nelson, et al., 1985) (Paessens, 1988), Formelerweiterung und Verarbeitung der Datenstruktur (Yellow, 1970) (Gaskell, 1967), und Vereinigung von den Knotenpunkten (Altinkemer & Gavish, 1991) (Wark & Holt, 1994) für die Saving Algorithmus entworfen wurden. In der Recherche von Cordeau et al. (2002) wird betont, dass diese Anpassungen die Einfachheit sowie auch die Schnelligkeit des Saving Algorithmus beeinträchtigen kann (Cordeau, et al., 2002).

2.6. Genetischer Algorithmus

Die genetischen Algorithmen sind eine Erweiterung von Darwins Selektionstheorie und wird für kombinatorische Probleme verwendet. Darwins Theorie besagt, dass je mehr das Individuum sich an ihre Umgebung anpasst, desto höher ist die Überlebenswahrscheinlichkeit im Kampf gegen die Existenz. Die kombinatorischen Probleme haben viel Anspruch an Rechenleistung. Für diese Fälle zeigt der genetische Algorithmus eine gute Leistung in der Berechnungszeit sowie in der Lösungsqualität. Zusätzlich haben die genetischen Algorithmen

bessere Eigenschaften als die restlichen evolutionären Algorithmen. Die Verwendung von der Mutations- und Kreuzungseigenschaft gleichzeitig verglichen mit den anderen evolutionären Algorithmen schafft den genetischen Algorithmus hohe Immunität nicht in den lokalen Optima festzustecken. Weiters ist es geringer anfällig für das Problem der vorzeitigen Konvergenz im Gegensatz zu den anderen evolutionären Algorithmen (Ramadan, 2012).

Das Problem der vorzeitigen Konvergenz ist beim genetischen Algorithmus, dass den hochgestuften Genen die Population des Algorithmus dominieren und somit der Algorithmus in das lokale Optimum gelangt. Die vorzeitige Konvergenz entsteht im Allgemeinen durch den Verlust der Diversität in der Population. Deshalb ist es wichtig in dem genetischen Algorithmus während der Durchläufe, die Diversität der Population zu gewährleisten. Für diese Fälle werden verschiedene Methoden in den nächsten Kapiteln entworfen (Malik & Wadhawa, 2014).

Die Abbildung 3 - Flussdiagram von genetischem Algorithmus zeigt einzelne Prozessschritte des genetischen Algorithmus. Diese Übersicht dient zur Veranschaulichung des Algorithmus. In diesem Zusammenhang wird die einzelne Komponente des Algorithmus visualisiert.

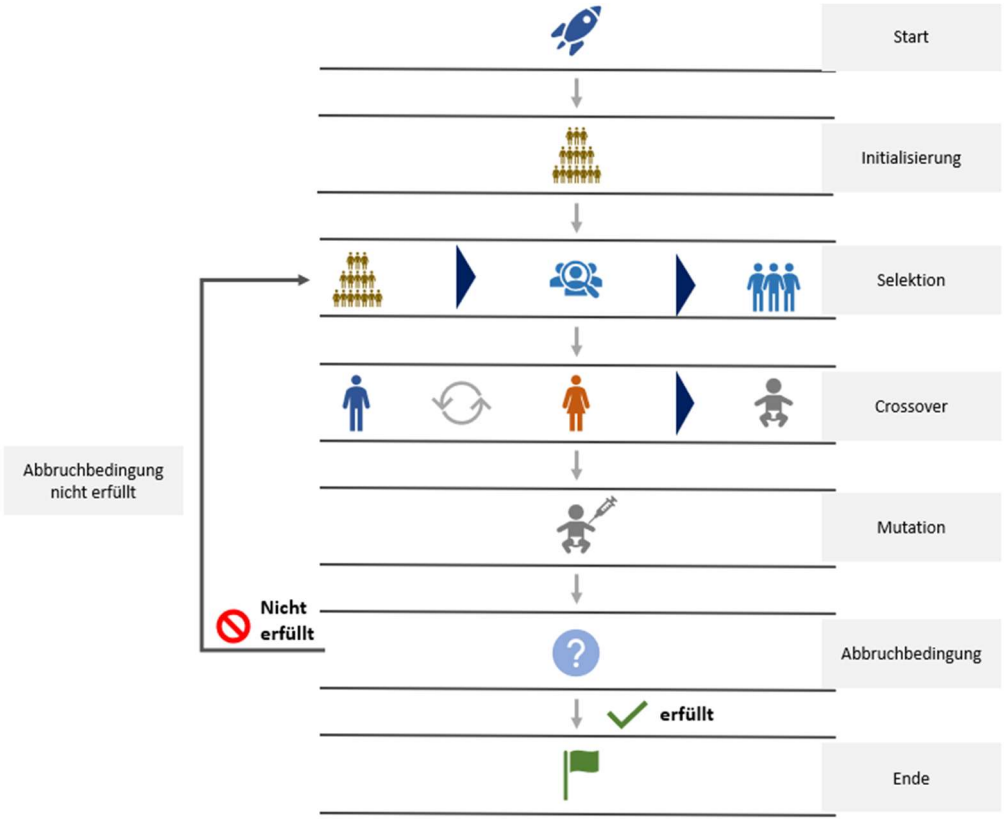


Abbildung 3 - Flussdiagram von genetischem Algorithmus

Das Grundkonstrukt des genetischen Algorithmus basiert auf Initialisierung, Selektion, Crossover, Mutation, Abbruchbedingung und Fitness. Diese Konstrukte werden in den folgenden Unterpunkten erklärt. Weiters ist es erwähnenswert, dass die Touren in der Tourenplanung gleichzeitig (parallel) bearbeitet werden. Damit ist gemeint, dass alle Touren in der Tourenplanung zu einem String umgewandelt werden. Danach werden erst die Operatoren Selektion, Crossover und Mutation angewendet.

2.6.1. Initialisierung

Zu Beginn des genetischen Algorithmus wird eine zufällige Population generiert. Diese Population besteht aus diversen Routenplanungen. Weiters können diese Routenplanungen je nach Einschränkungen aus mehreren Touren bestehen. Die Generierung der Population ist pseudo zufällig. Somit steigt die Wahrscheinlichkeit der gleichen Routenplanung mit steigender Anzahl an Population. Ebenso ist es wichtig zu erwähnen, dass größere Populationen verschiedene Vorteile haben. Einerseits erlauben größere Population eine Menge an verschiedenen Lösungen. Andererseits sind größere Populationen widerstandsfähiger gegen Verlust der Diversität in der Population. Die Diversität der Population sinkt, wenn die generierte Population an die besten Routenplanungen annähert. In diesem Fall wird die Suche auf bessere Individuen (Routenplanung) in der Population eingeschränkt. Folglich wird die Findung von neuen Ergebnissen erschwert. Dadurch wird das Problem der vorzeitigen Konvergenz verursacht. Bei einer großen Population bleibt die Langlebigkeit von der Diversität länger. Das Problem bei großen Populationen ist, dass die Berechnungsdauer steigt. Meistens ist die Evaluierung von der Fitness die Hauptursache für längere Berechnungszeiten. Deshalb ist es sehr wichtig, wie groß die Population ist (Ryan, et al., 2003).

2.6.2. Fitness

Für jedes Individuum in der Population wird der Fitnesswert berechnet. Der Fitnesswert ist ein Maß für die Qualität einer Lösung. Dieser Wert ist äußerst wichtig, denn die Selektion der Touren basiert auf diesen Wert. Je besser die Fitness ist, desto größer ist die Auswahlwahrscheinlichkeit. Nichtdestotrotz können die Individuen mit kleiner Fitness auch ausgewählt werden, weil die genetische Selektion probabilistisch ist. Allerdings liegt nicht immer die beste Fitness nahe der optimalen Lösung. Für die Populationsvielfalt ist es auch wichtig, dass die Fitnesswerte mit niedrigerem Level ausgewählt werden. Somit kann das Problem der vorzeitigen Konvergenz auch verhindert werden (Shukla, et al., 2015) (Zhong, et al., 2005).

Für VRP lautet die Fitnessformel wie folgt:

$$Fitness = \frac{1}{Distanz}$$

Formel 5 - Fitnessformel

Die Fitness Formel kann durch Potenzierung von der Distanz erweitert werden. Dies sorgt für noch mehr Vielfalt in der Population in einer engere Lösungsbereich.

2.6.3. Selektion

Die Selektionseigenschaft befasst sich mit der Auswahl der Elternteile (= sind ausgewählte Tourenplanung von Population in TSP bzw. VRP). Für die Selektion gibt es verschiedene Ansätze. Diese Schwierigkeit variiert durch die Komplexität des Problemansatzes. In diesem Zusammenhang ist die Auswahl der geeigneten Selektionsmethodik ein mühevoller Schritt, weil es einerseits die Performance des genetischen Algorithmus beeinflussen kann und anderseits die Ausführgeschwindigkeit einschränken kann. Weiters wird die Performance einerseits von der Konvergenzrate bestimmt. Andererseits spielt die Anzahl an erzeugten Generationen eine Rolle. Aus den selektierten Elternteilen werden Chromosomen erzeugt. Diese Chromosomen entstehen durch die Elternteile. In VRP sind es die Touren, indem die Reihenfolge an Standorten von einem Elternteil genommen wird. Dafür gibt es das Crossover-Verfahren, die im nächsten Unterpunkt näher erläutert wird. All diese Chromosomen repräsentieren ein mögliches Ergebnis in dem Suchumfeld (Shukla, et al., 2015) (Zhong, et al., 2005)**.**

In dieser Arbeit werden zwei Selektionsarten angewandt. Die erste Art ist die Roulette Wheel Selektion und die zweite ist die Tournament Selektion. Die Roulette Wheel Selektionsart ist eine proportionale Selektionsstrategie, welche am häufigsten verwendet wird. Die Abbildung 4 - Beispiel für Roulette Wheel Selektion zeigt eine mögliche Darstellung für die Roulette Wheel Selektion (Zhong, et al., 2005).

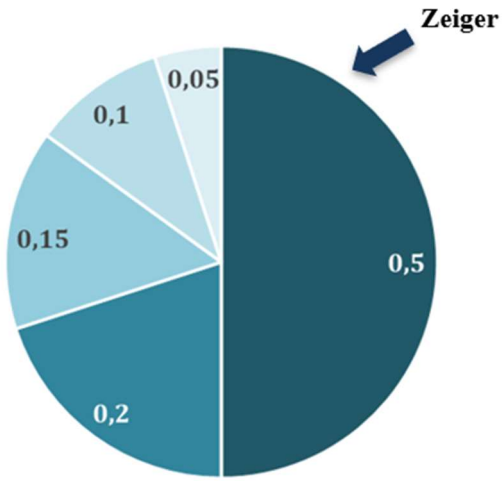


Abbildung 4 - Beispiel für Roulette Wheel Selektion

Das Roulette Rad ist in verschiedene Sektoren geteilt. Die Sektorengröße bildet in dem VRP die Fitnesswerte der einzelnen Tourenplanungen. Das heißt, dass jeder Sektor eine Tourenplanung in der Population des genetischen Algorithmus darstellt. Die Zeigerstellung kann nicht von vorhinein bestimmt werden. Je größere die Fitness ist, desto höher ist die Auswahlwahrscheinlichkeit (Zhong, et al., 2005).

In den folgenden Schritten wird die Roulette-Wheel-Selektion abgehandelt:

1) Zuerst müssen die Fitnesswerte von den einzelnen Tourenplanungen in der Population ermittelt werden. Deshalb wird als ersteres die Fitnessberechnung für die gesamte Population durchgeführt (Zhong, et al., 2005).

2) Danach wird die Fitnesssumme von der gesamten Population gerechnet (Zhong, et al., 2005).

3) In diesem Schritt gibt es verschiedene Ansätze. Für diese wissenschaftliche Arbeit wird zuerst die Fitnesssumme mit einem pseudozufällige Doublewert multipliziert und wird in die Variable F_p gespeichert. Danach wird das Rad gedreht (d.h., dass die Tourenplanung pseudozufällig von der Population ausgewählt wird). Während jeder Drehung wird der Fitnesswert von der Tourenplanung, die durch den Zeiger gekennzeichnet wird, in die Variable F_s summiert. Bei jeder Addierung wird abgefragt, ob die errechnete pseudozufällige Fitnesssumme F_p erreicht wurde. Das Verfahren endet, indem die addierte Fitnesssumme F_s größer gleich pseudozufälligen Fitnesssumme F_p ist (Zhong, et al., 2005).

Die letzte Auswahl des Zeigers wird als ein Elternteil für die Crossover-Funktion nominiert. Also wird diese Selektionsfunktion zweimal abgerufen, damit beide Elternteile für die Crossover-Funktion zur Verfügung stehen.

Die zweite Selektionsart ist die Tournament-Selektion. In dieser Selektionsart fällt die Auswahl durch die Fitnesswerte von den einzelnen Tourenplanungen. Der höchste Fitnesswert in der zufällig generierten Liste wird als ein Elternteil definiert. Bevor die Tournament-Selektion durchgeführt wird, muss in erster Linie die Tournament-Größe bestimmt werden. Um es anders auszudrücken, die pseudo zufällige Auswahl an Teilnehmer*innen für das Wettbewerb untereinander wird definiert. Die Abbildung 5 - Ablauf einer Tournament-Selektion präsentiert eine beispielhafte Prozedur der Tournament-Selektion. Nachfolgend zu der Abbildung werden die einzelnen Schritte der Tournament-Selektion ausführlich erklärt (Zhong, et al., 2005).

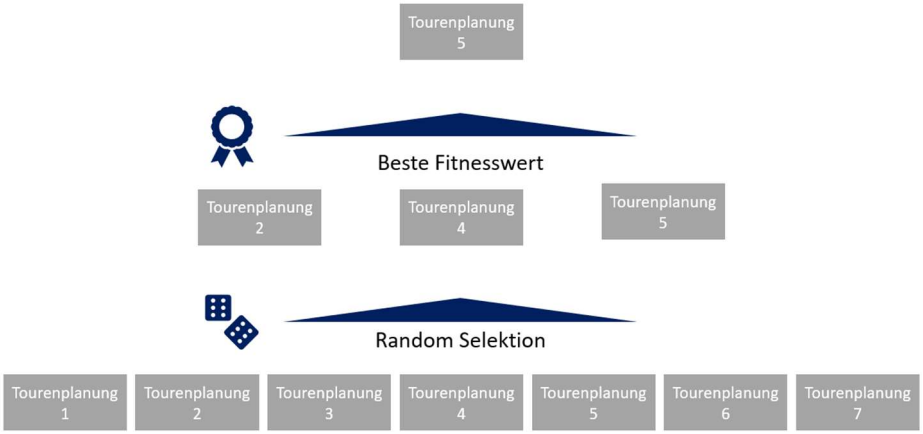


Abbildung 5 - Ablauf einer Tournament-Selektion

Die unten angeführten Aufzählungen bieten Einblicke in den Ablauf der Tournament-Selektion:

1) Am Anfang der Tournament-Selektion wird von der Population eine bestimmte Anzahl an Tourenplanungen für die Turnierphase aufgenommen. Die Anzahl an Tourenplanungen wird beim Start des Turniers bestimmt.

2) In der Abbildung 5 - Ablauf einer Tournament-Selektion wurden drei Tourenplanungen entschieden.

3) Nachdem die pseudo zufällige Auswahl getroffen wurde, wird eine Eruierung zwischen diesen ausgewählten Tourenplanungen erhoben.

4) Zum Schluss wird die Tourenplanung mit dem besten Fitnesswert als Sieger und als einer der Elternteile ernannt.

Wie auch in der Roulette-Wheel-Selektion, wird die Tournament-Selektion ein zweites Mal ausgeführt, um das zweite Elternteil zu bestimmen. Diese beiden Selektionsarten werden auch in der Umsetzungsphase verwendet.

2.6.4. Crossover

Das Crossover zielt im genetischen Algorithmus auf die Erzeugung von neuen Routenplanungen (werden als Kind (child) benannt), indem von den beiden Elternteilen neue Routenplanungen kombiniert werden. Die Elternteile werden durch die Selektion bestimmt. Nach der Bestimmung der Elternteile können „child"-Touren erzeugt werden. In diesem Zusammenhang werden in dieser Arbeit zwei Crossover-Arten für den genetischen Algorithmus verwendet. Die erste Crossover-Funktion ist das geordnete Crossover (engl. ordered crossover, Abk. OX). Das OX ist eine Erweiterung des modifizierten Crossover (engl. modified crossover). Es ist ein einfaches Permutation-Crossover, dass in wenigen Schritten durchgeführt werden kann. In dieser Hinsicht wird zuerst die Abgrenzung von dem Elternteil 1 bestimmt. Dafür werden zwei zufällige Werte generiert. Diese Werte befinden sich in der Abgrenzung von der Tourplangröße. Das heißt, der zufällige Wert wird zwischen null und der Anzahl an Standorten generiert. Diese zwei Werte bestimmen die Abgrenzung von dem ersten Elternteil. Die Standorte in dieser Abgrenzung werden zu der gleichen Position in die „child"-Tour übertragen. Die restlichen Standorte, die noch offengeblieben sind, treten in der gleichen Reihenfolge von dem zweiten Elternteil in die „child" – Tour auf. Dieses Vorgehen wird in der Abbildung 6 - Ablauf des geordneten Crossover präsentiert (Kumar, et al., 2022).

Abbildung 6 - Ablauf des geordneten Crossover

Als zweites Crossover wurde das Universal-Crossover verwendet die durch Misevičius et al. (2018) entwickelt wurde. Das universelle Crossover zeichnet sich für seine Flexibilität aus. Es ist eine ähnliche Crossover-Funktion wie das simulierte binäre Crossover (Kusum & Manoj, 2007) (Misevičius, et al., 2018).

Die Autor*innen Misevičiusm et al. (2018) schrieben in ihrer wissenschaftlichen Arbeit, dass das Universal-Crossover im Gegensatz zu dem simulierten binäre Crossover auf eine Zufallsmaske basiert. In dieser Crossover-Funktion wird zuerst eine Zufallsmaske erzeugt. Diese Maske besteht aus Nullen und Einsern. In anderen Worten ist es eine zufällig erzeugte binäre Folge von Zahlen. Diese Folge an Zahlen hat genau die gleiche Länge wie die beiden Elternteile, die durch die Selektionsverfahren ausgesucht wurden. Wenn in der Maske eine eins steht, wird von dem ersten Elternteil an der jeweiligen Position der Standort ausgewählt und in die Kindtourplanung eingefügt. Wenn eine Null in der Maske steht, dann wird an dieser Stelle von dem zweiten Elternteil der Standort in die Routenplanung eingefügt. Falls der Standort in die Kindroutenplanung eingefügt wurde, nimmt das Universal-Crossover einen zufälligen Standort zwischen den nicht eingefügten Standorten. Das Vorgehen verdeutlicht die Abbildung 7.

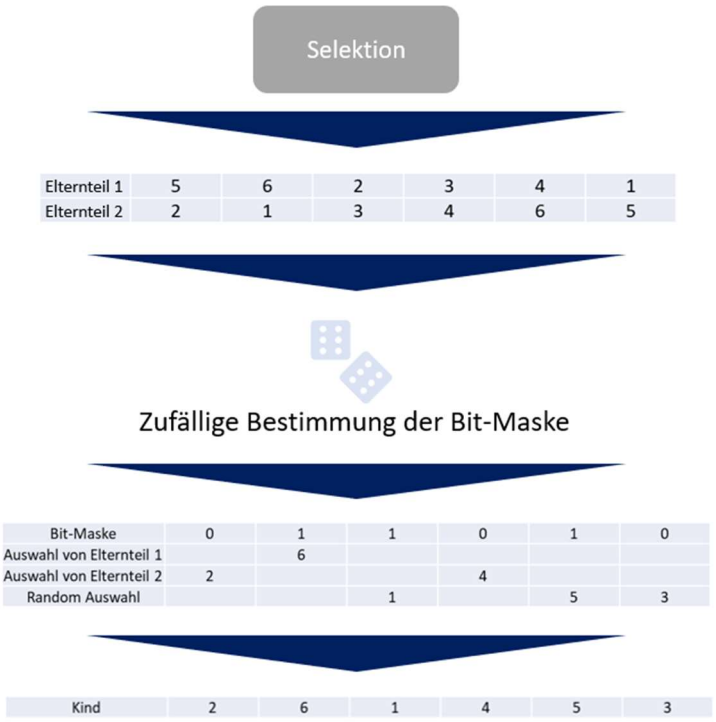


Abbildung 7 - Universell-Crossover

Durch die zwei Crossover-Funktionen werden die Kinder erzeugt. Dies wird so lange durchgeführt, bis die angegebene Population durch die Erzeugung von den Kindern erreicht wird.

2.6.5. Mutation

Die Mutation wirkt gegen das Problem der vorzeitigen Konvergenz, in dem sie in der Kind-Tour die Position von den Standorten vertauscht. Somit wird mit diesem Faktor noch mehr Vielfalt in die Population erzielt. Demzufolge wurde auch für die Mutation zwei Variationen verwendet. Die erste Variation ist die Swap-Mutation und die zweite Variation ist Invers-Mutation. In der Swap-Mutation werden zufällige zwei Standorte ausgewählt und die Positionen miteinander getauscht. Somit liegt der erste ausgewählte Standort in der Position von dem zweiten ausgewählten Standort und vice versa. Für die Veranschaulichung der Swap-Mutation wurde die Abbildung 8 entworfen (Soni & Tapas, 2014).

Abbildung 8 - Swap-Mutation

In der Abbildung 8 wird die Position 2 und 5 zufällig ausgewählt. Demnach werden die Standorte sechs und fünf miteinander getauscht. Anschließend erhielt der Standort fünf die Position zwei und der Standort sechs die Position 5.

Die zweite Mutation ist die Invers-Mutation. Zuerst werden zwei zufällige Positionen in der Kind-Tourenplanung ausgewählt. Die zwei Positionen dienen als Zeiger für die Umkehrung. Nachdem die beiden Positionen bestimmt wurden, wandelt die Invers-Mutation die Standorte zwischen diesen zwei Punkten in die umgekehrte Richtung um (Soni & Tapas, 2014).

Die Abbildung 9 präsentiert die einzelnen Schritte von der Invers-Mutation.

Abbildung 9 - Invers-Mutation

Die Invers-Mutation beginnt wie die Swap-Mutation mit einer zufälligen Auswahl von zwei Positionen in der Kind-Tour. Die Umkehrung findet zwischen diesen zwei Positionen statt. Wie auch in der Abbildung 9 ersichtlich wird, wird der String 6-1-4-5 entgegengesetzt zu 5-4-1-6. Daraus ergibt sich die Kind-Tourenplanung 2-5-4-1-6.

An dieser Stelle ist die Erwähnung der Abbruchbedingung des genetischen Algorithmus relevant. Die Abbruchbedingung beendet die Tourenplanung dann, wenn:

- entweder eine gewisse Menge an Verbesserungen durchgeführt wurde
- oder eine bestimmte Anzahl an Runden keine Verbesserungen in der Routenplanung ergeben hat.

Für diese Arbeit spricht die zweite Methode von hoher Relevanz. Die Generation von dem genetischen Algorithmus läuft so lange ab, bis sich die beste Tourenplanung zum festgelegten Intervall nicht verbessert.

3. Bing Maps

Es gibt verschiedene Kartenmaterialanbieter mit einer API-Funktion. Eine der größten ist Bing Maps. Durch den Abschluss der Partnerschaft mit TomTom (Kartenmaterialanbieter) im Jahr 2019 wurde das Mapping-Portfolio verstärkt (Bing-Maps-Team, 2020). Eine API ist eine programmierbare Schnittstelle, indem verschiedene Abfragen durchgeführt werden. Die Bing Maps API bietet unterschiedliche Funktionen mit der die Routenplanungen errechnet werden. Einerseits kann man die Art des Bewegungsmittels wählen, andererseits ist es möglich die Bewegungsmitteleigenschaften anzugeben. Diese Faktoren bewirken auf die Routenplanung. In dieser Arbeit wurde Bing Maps API für die LKW-Routenplanung eingesetzt. Diese Funktion war nicht bei Google Maps erhältlich. Darüber hinaus ist es noch wichtig zu erwähnen, dass die Bing Maps API anhand von realen geografischen Daten und Straßenverkehrsordnungen eine Routenplanung realisiert.

In diesem Kapitel werden am Anfang die Besonderheiten der Bing Maps API aufgezählt. Die Besonderheiten werden sich nur auf die LKW-Routenplanung einschränken. Darauffolgend wird der Verbindungsaufbau zur Schnittstelle erklärt. In diesem Zusammenhang wird der methodische Ansatz für einen stabilen Verbindungsaufbau demonstriert. Zum Schluss werden die Einschränkungen von der Bing Maps API vorgetragen. Die Einschränkungen basieren auf die tatsächlichen Probleme während der Programmierung bzw. Aufbau der Bing Maps API.

3.1. Besonderheiten der Bing Maps API in Bezug auf LKW-Routenplanung

Die Bing Maps API ermöglicht eine Routenplanung speziell für die LKWs. Dabei können die Eigenschaften wie Größe, Breite, Typ von dem Transportmittel und vieles mehr angegeben werden. Ergänzend ist es wichtig zu erwähnen, dass nicht alle LKWs die gleiche Route fahren dürfen. Die unten angeführte Liste zeigt Exemplare, weswegen die LKWs nicht dieselbe Route wählen dürfen (Bing Maps, 2022).

- Die Größe und das Gewicht der LKWs kann für die Brücken ein Problem darstellen. In diesem Fall wird die Routenplanung für das LKW spezifisch durchgeführt. Somit werden die Brücken mit geringer zugelassener Höhe auf eine alternative Route umgeleitet. Weiters gilt diese Regelung auch für das Gewicht. In diesem Fall plant Bing Maps eine alternative Route.

- Die Tunnels haben meistens eine Einschränkung. Die LKWs dürfen nicht mit gefährlichen und schnell entzündbaren Ladungen durch den Tunnel fahren. Durch Bing Maps wird eine Analyse durchgeführt, damit die Tunnel bei gefährlichen Schadstoffen umfahren werden.

- Weitere Probleme können Steigungen, Geschwindigkeitsbeschränkungen und enge Kurvenfahrten für die LKWs darstellen. In diesem Zusammenhang plant die Bing Maps API alle möglichen Szenarien und vermeidet die Problemstellen, in dem eine Umleitung geplant wird. Für die Geschwindigkeitsbeschränkung sucht die Bing Maps API eine alternative Route, damit die Strecke schneller zurückgelegt werden kann.

Die oben genannten Auswahlkriterien sind nur ein Teil des Portfolios. Es gibt viele weitere Optionen, in denen man festlegen kann, wie die Routenplanung folgen soll. Es beginnt bei der Minimierung von Grenzübergängen bis zur Vermeidung von Autobahnen. Die wird auch für die Routenplanungen von PKWs angeboten.

Dennoch fokussiert sich diese wissenschaftliche Arbeit auf die Eigenschaften der LKWs. Die angegebenen Parameter des LKWs sind Höhe, Breite, Länge, Gewicht, Anzahl an Achsen, maximaler Drehungsradius, maximale Steigung und gefährliches Transportgut. Bei gefährlichem Transportgut wurde nichts deklariert. Ein weiteres Kriterium war die Optimierungsmöglichkeit. Zum einen ist es möglich nach der jetzigen Verkehrsinformation zu optimieren und zum anderen kann die Auswahl auf die Minimierung der Fahrtzeit beruhen, ohne die Verkehrsinformation in Betracht zu ziehen. Die zweite Option wurde auch für diese Arbeit bevorzugt, weil die Verkehrslage sich ändern kann und sie die Auswertung beeinflussen würde. Demzufolge wurde nicht anhand der Verkehrslage optimiert, sondern mithilfe der Fahrzeit.

Die Visualisierung von den geplanten Routen wird mithilfe der Bing Maps API durgeführt. Diesbezüglich wird eine JavaScript-Programmierung und eine HTML-Datei benötigt. Die HTML-Datei dient zur Visualisierung und die JavaScript-Programmierung wird für die Verbindung mit der Bing Maps Schnittstelle benötigt. Durch die Abfrage wird die Karte mit der jeweiligen Tourenplanung in der HTML-Datei angezeigt. Zusammengefasst wird durch die Abfrage die Karte mit der Routenplanung von Bing Maps zur Verfügung gestellt. Dies erleichtert wiederum die Visualisierung der Routenplanungen.

3.2. Methodischer Ansatz für die Verbindung an Bing Maps API

Zwei HTTP-Methoden werden von der Bing Maps API unterstützt. Diese zwei HTTP-Methoden sind GET und POST. Bei der POST-Methode müssen alle Parameter in dem Body der Anfrage mit JSON-Objekt eingebettet werden. Es ist komplexer als bei der GET-Methode. Deshalb wird meistens von dem Entwickler*innen die GET-Methode verwendet. In der GET-Methode muss nur ein URL (Uniform Resource Locator, ist ein Standard für die Adressierung von einer Website im Internet) erzeugt werden. Die Browser und Server haben eine Einschränkung in Bezug auf die Länge des URLs. Herkömmlich sollte die Länge des URLs nicht länger als 2083 Charakter sein. Nichtdestotrotz, wurde die GET-Methode in der Arbeit verwendet, weil die Länge des URLs den maximalen Wert nicht überschreitet (Bing Maps, 2022).

Ebenfalls sind auch zwei verschiedene Varianten für den Verbindungsaufbau vorhanden. Die Verbindung kann synchron aufgebaut werden, demgegenüber ist es auch möglich sie asynchron aufzubauen. Bei der synchronen Abfrage wird auf die Antwort von dem Server gewartet, bis eine Antwort erhalten wird. Das Programm wird so lange angehalten, bis es die Antwort erhält und springt nicht auf den nächsten Befehl. Im Allgemeinen braucht die Routenplanung für die LKWs mehr Rechenarbeit bedingt von der LKW-Eigenschaft. Aus diesem Anlass wird von Bing Maps die asynchrone Variante für lange Strecken empfohlen. Somit ist es möglich, Parallelabfragen durchzuführen und die Gesamtwartezeit zu verringern. Natürlich gibt es auch hier Einschränkungen die im Kapitel 3.3 betont werden (Bing Maps, 2022).

Nichtdestotrotz wurde die asynchrone Variante in dieser wissenschaftlichen Arbeit verwendet. Dazu wurde der asynchrone Verbindungsaufbau zum Server und die Abfrage in der Abbildung 10 aufgezeichnet.

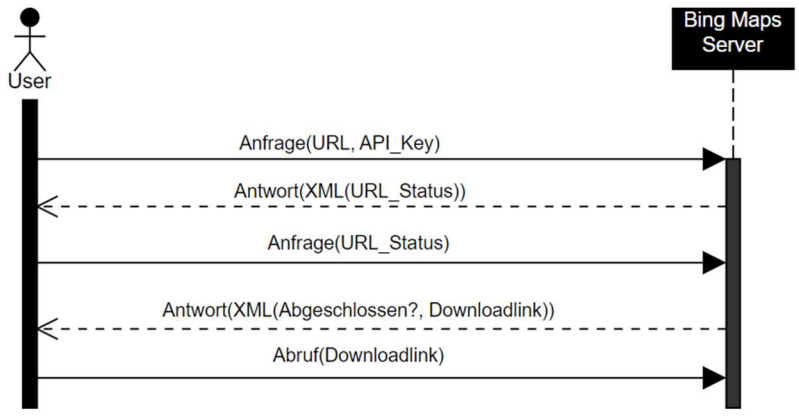


Abbildung 10 - Asynchroner Verbindungsaufbau zu Bing Maps API mit GET-Methode

In der Abbildung 10 wird die GET-Methode verwendet, d.h., dass die Anfrage mit einem URL gesendet wird. Weiters handelt es hier um eine asynchrone Abfrage. Zunächst wird der URL erstellt, welcher folgende Parameter beinhaltet: LKW-Eigenschaften, Koordinate von den jeweiligen Knotenpunkten und API-Key. Mit diesen Parametern wird der Server angefragt. Der Server antwortet mit einem XML-file in dem der „URL_Status“ liegt. Dieser „URL_Status“ dient zur Verfolgungszwecke, damit die Kommunikation mit dem Server stattfinden kann, ob die Routenplanung erfolgreich war. Als nächstes wird dieser „URL_Status“ angefragt, ob die Antwort von dem Server mit der Routenplanung erstellt wurde. Wenn es fertig ist, antwortet der Server mit einem Downloadlink. Dieser Downloadlink ist auch eine XML-Datei, welche verschiedene Informationen zur Routenplanung enthält. Es beginnt mit einer Entfernungsangabe bis hin zur schriftlichen Wegbeschreibung. Wenn der Server die Routenplanung noch nicht zu Ende berechnet hat, wird auch kein Downloadlink erzeugt. Somit müssen die User*innen weiterhin Nachfragen, bis dieser Downloadlink erstellt wird. In Abhängigkeit von dem Downloadlink wird der letzte Schritt ausgeführt, indem der Downloadlink zum Schluss geöffnet wird. Somit wird die asynchrone Verbindung vollendet und die Routenplanung wurde erfolgreich abgeschlossen (Bing Maps, 2022).

Durch die asynchrone Verbindung zu der Bing Maps API wurde in dieser Arbeit eine Distanzmatrix erstellt. Diese Distanzmatrix beinhaltet 150 Knotenpunkte. Zufolge dessen wurde der Server insgesamt 22.500-mal angefragt. Im nächsten Kapitel werden die Herausforderungen, die durch diese An- und Abfragen herausgekommen sind, diskutiert.

3.3. Einschränkungen und Herausforderungen

In diesem Kapitel werden die Einschränkungen von der Bing Maps API präsentiert. Durch diese Einschränkungen entstanden verschiedene Herausforderungen. Diese Herausforderungen werden in diesem Kapitel zusammengefasst.

Die erste Herausforderung war die genaue Angabe der Koordinaten. Bei geringen Abweichungen kam es zu Unterbrechungen von dem Programm. Für die 150 Knotenpunkte, mit dem die Routenplanung durchgeführt wurde, wurde die Koordinaten manuell ausgesucht. Wenn eine große Abweichung von der Straße gab, generierte das Programm eine Fehlermeldung. Um diese Ungenauigkeit zu beseitigen, wurden die Koordinaten mit 11 Kommastellen angegeben. Somit wurde die Präzision von den angegebenen Koordinaten erhöht und die Erstellung von der Distanzmatrix erfolgreich abgeschlossen.

Die nächste Erschwernis ist die Abfragebegrenzung für Basisnutzer*innen. Das Unternehmen bietet für Entwicklungszwecke eine gratis Version an, mit welcher die Abfragen an die Bing Maps API durchgeführt werden können. Natürlich haben Basisnutzer*innen Einschränkungen. Einerseits dürfen jährlich 125.000 Abfragen durchgeführt werden. Anderseits lässt Bing Maps pro Tag maximale 50.000 Abfragen zu. Deshalb wurde das Programm auf die Einschränkung von dem Abfragen angepasst, welche einzeln durchgeführt wurden, wobei die asynchrone Verbindung parallele Abfragen erlaubt. Die Begründung dahinter ist, dass bei parallelen Abfragen das Risiko an Fehleranteil steigt. Darüber hinaus wurde die Distanzmatrix nur einmal erzeugt und für die gesamte Testphase verwendet. Damit ist gemeint, dass die Erzeugung von der Distanzmatrix nur einmal erfolgte und bei der Testphase von den einzelnen Algorithmen nicht bei jedem Anlass von neu generiert wurde, aufgrund der Begrenzung der Abfragen. Weiters mussten auch die Abfragen für die Visualisierung der Routenplanungen konserviert werden. Dem liegt zu Grunde, dass die Routenplanung anhand der Distanzmatrix erstellt wird und bei den Abfragen nicht nach Visualisierung angefordert wird. Deshalb musste auch mit den Abfragen schonend umgegangen werden.

Insofern kann die letzte Einschränkung beziehungsweise Erschwernis diskutiert werden. Folglich zur Begrenzung der Abfragen und Durchführung der einzelnen Abfragen wird die Dauer der Erstellung von der Distanzmatrix beeinflusst. Die durchschnittliche Zeit für eine Abfrage dauerte zirka drei bis vier Sekunden. Für eine Distanzmatrix mit 150 Knotenpunkten werden 22.500 Abfragen benötigt. Die Schlussfolgerung ist, dass rund 21 Stunden nur für die Erstellung der Distanzmatrix gebraucht wird. Natürlich kann die Zeit mit einer Umstellung auf die Parallelabfragemethode deutlich verringert werden. Dafür wird eine Lizenzvereinbarung mit der Bing Maps API benötigt, welche kostenpflichtig sind. In diesem Fall muss der Vertrieb von Bing Maps kontaktiert werden. Dies kann durch die Website von Bing Maps gemacht werden. Bei der Beantragung muss ein Formular ausgefüllt werden, für welche Zwecke die Bing Maps API benötigt wird.

Zusammenfassend wurden die meisten Einschränkungen durch zeitliche Verzögerungen verursacht. Die genaue Koordinatenangabe hat demzufolge auch die serverseitigen Fehlermeldungen erlöscht. Demzufolge konnte eine Distanzmatrix mit 150 Knotenpunkten erzeugt werden. Das ist die Grundlage für die Erstellung der einzelnen Routenplanungen, die mit verschiedenen Algorithmen geplant werden.

4. Programm

In diesem Kapitel werden die wichtigsten Punkte der Programmierung vorgestellt. Dabei werden sowohl die Algorithmen aufgelistet als auch der Verbindungsaufbau zu Bing Maps präsentiert. Darüber hinaus ist noch wichtig zu erwähnen, dass die Verbesserungsansätze demonstriert werden. Ergänzend werden noch die JavaScript-Datei und HTML-Datei vorgeführt. Abschließend wird der VBA-Code präsentiert, der als Grundlage für die Auswertungen dient.

4.1. Nearest Insertion

Der Nearest Insertion Algorithmus wurde in Orientierung an das Kapitel 2.3 aufgebaut. Das Grundgerüst dieses Algorithmus basiert auf eine Funktion, wodurch die gesamte Routenplanung durchgeführt wird.

```
List<Tour> nodeNearestNeighbour = new List<Tour>();
int nodeHasNotBeenVisited = 0;


foreach (Point p in PointList)
{
    if (p.HasBeenVisited == false)
    {
        nodeHasNotBeenVisited++;
    }
}
nodeHasNotBeenVisited--;
```

Abbildung 11 - Nearest Insertion: Erzeugung von Tourliste

Zu Beginn der Nearest Insertion wird eine Liste erzeugt in dem die Tourenplanungen gespeichert werden (siehe Abbildung 11 - Nearest Insertion). Danach wird die Anzahl an Knotenpunkten gezählt und in einer Variable abgespeichert. Diese Variable „nodeHasNotBeenVisited“ zeigt die Anzahl an nicht besuchten Standorten, die für die nächsten Programmierschritte benötigt werden.

```
while (nodeHasNotBeenVisited > 1)
{
    currentCity = 0;
    nodeHasNotBeenVisited--;
    List<int> nodeNearestTour = new List<int>();
    PointList[currentCity].HasBeenVisited = true;
    nodeNearestTour.Add(currentCity + 1);
    double distance = double.MaxValue;
    int nearestNeighbour = -1;
    int tourcapacity = 0;
    for (int j = 0; j < PointList.Count - 1; j++)
    {
        for (int i = 0; i < PointList.Count; i++)
        {
            if ((currentCity != i) && (distanceMatrix[currentCity, i] < distance)
                && PointList[i].HasBeenVisited == false && maxCapacity >= (tourcapacity + PointList[i].Capacity))
            {
                distance = distanceMatrix[currentCity, i];
                nearestNeighbour = i + 1;

            }
        }
        if (PointList[nearestNeighbour - 1].HasBeenVisited == false)
        {
            PointList[nearestNeighbour - 1].HasBeenVisited = true;
            tourcapacity += PointList[nearestNeighbour - 1].Capacity;
            nodeNearestTour.Add(nearestNeighbour);
            currentCity = nearestNeighbour - 1;
            distance = double.MaxValue;
        }
    }
    nodeNearestTour.Add(nodeNearestTour[0]);
    PointList[0].HasBeenVisited = false;
    nodeNearestNeighbour.Add(new Tour(nodeNearestTour, 0, 0, tourcapacity));
    nodeHasNotBeenVisited = 0;
    foreach (Point p in PointList)
    {
        if (p.HasBeenVisited == false)
        {
            nodeHasNotBeenVisited++;
        }
    }
```

Abbildung 12 - Nearest Insertion: Routenplanungsprozess

In der Abbildung 12 wird der Routenplanungsprozess von dem Nearest Insertion Algorithmus veranschaulicht. Die Routenplanung beginnt mit der Erstellung einer Tour. Der Standort wird am Anfang in die Tour eingearbeitet. Danach wird diese Tour so lang befüllt, bis die Kapazität von dem LKW nicht mehr zulässt. Das wird durch die zwei For-Schleifen gesteuert. Die zwei For-Schleifen berechnen immer den nächsten Standort in der Distanmatrix. Nachdem dies ermittelt wurde, wird überprüft, ob die Kapazitätseinschränkung die Einbettung dieses Kontenpunkts in die Tour zulässt. Die For-Schleifen führen die Tourenplanung durch, bis entweder die Kapazität nicht zulässt oder keine Knotenpunkte mehr übriggeblieben sind. Wenn keine Kapazität mehr vorhanden ist, wird die Tour mit dem Knotenpunkt „Depot“ geschlossen und eine neue Tour wird geöffnet im Hinblick, dass noch nicht zugewiesene Standorte existieren. Ansonsten hört die Tourenplanung für die Nearest Insertion in diesem Zeitpunkt auf.

4.2. Cheapest Insertion

Der Cheapest Insertion Algorithmus ist in zwei Funktionen aufgeteilt. Diese Funktionen wurden nach Kapitel 2.4 aufgebaut. In einer der Funktionen wird die Tourenplanung

durchgeführt (siehe Abbildung 13). In der zweiten Funktion werden die Berechnungen für die Tourenplanung errechnet und den billigsten Knotenpunkt wird zu Auswahl an die Tourenplanung-Funktion verschickt (siehe Abbildung 14).

```
while (unvisitedNodes.Count > 0)
{
    visitedNodes.Clear();
    visitedNodes.Add(startingindex);// add starting node
    visitedNodes.Add(startingindex); // add endpoint
    int tourcapacity = 0;

    bool breakFlag = false;
    while (!breakFlag)
    {
        if (unvisitedNodes.Count == 0)
        {
            break;
        }
        int pointToInsert = GetCheapestPoint(distanceMatrix, visitedNodes, unvisitedNodes, out int insertionIndex);
        int addcapacity = PointList[pointToInsert - 1].Capacity;

        if (tourcapacity + addcapacity <= maxCapacity)
        {
            tourcapacity += addcapacity;
            visitedNodes.Insert(insertionIndex + 1, pointToInsert);
            unvisitedNodes.Remove(pointToInsert);

        }
        else
        {
            breakFlag = true;
        }


    }

    dummy.Add(new List<int>(visitedNodes));
    //cheapestInsertiontour.Add(new Tour(dummy, 0, 0, tourcapacity));

}
foreach (List<int> point in dummy)
{
    int dummyCapacity = 0;
    List<int> dummyListToHold = new List<int>();
    dummyListToHold = point;
    for (int i = 0; i < dummyListToHold.Count; i++)
    {
        dummyCapacity += PointList[dummyListToHold[i]-1].Capacity;
    }

    cheapestInsertiontour.Add(new Tour(point, 0, 0, dummyCapacity));
}
```

Abbildung 13 - Cheapest Insertion: Erstellung einer Tour

Die erste Funktion beginnt mit der Erstellung einer Tour. In dieser Tour wird zweimal das Depot eingefügt, um den Anfangspunkt und Endpunkt zu deklarieren. Danach wird eine Schleife so lange durchgeführt, bis die LKW-Kapazität nicht mehr zulässt oder keine Knotenpunkte mehr vorhanden sind. In dieser Schleife wird bei jeder Iteration die billigste Einfügeposition mit einem bestimmten Knotenpunkt ausgerechnet. Das findet mit der Unterfunktion, die in der Abbildung 14 dargestellt ist, statt.

```
public int GetCheapestPoint(double[,] distanceMatrix, List<int> visitedNodes, List<int> unvisitedNodes, out int index)
{
    double CheapestInsertionCost = double.MaxValue;
    int result = 0;

    index = 0;
    //iterate over all visited nodes
    for (int i = 0; i < visitedNodes.Count - 1; i++)
    {
        int before = visitedNodes[i];
        int after = visitedNodes[i + 1];
        double distanceInTour = distanceMatrix[before - 1, after - 1];
        double distanceA = 0;
        double distanceB = 0;
        int pointToInsert = 0;
        //iterate over all not visited nodes
        for (int j = 0; j < unvisitedNodes.Count; j++)
        {
            pointToInsert = unvisitedNodes[j];
            distanceA = distanceMatrix[before - 1, pointToInsert - 1];
            distanceB = distanceMatrix[after - 1, pointToInsert - 1];

            double insertionCost = distanceA + distanceB - distanceInTour;
            if (CheapestInsertionCost == double.MaxValue ||
                insertionCost < CheapestInsertionCost)
            {
                CheapestInsertionCost = insertionCost;
                result = pointToInsert;
                index = i;
            }


        }


    }
    return result;
}
```

Abbildung 14 - Cheapest Insertion: Bestimmung der Position und des Knotenpunktes

Die zweite Funktion bestimmt die Position und den Knotenpunkt, welche in die Tour eingefügt werden sollen. Diese Funktion besteht aus zwei Schleifen. In einer wird nach dem Knotenpunkt gesucht und in der anderen wird nach der Position erfragt. Die zwei Schleifen werden so lange durchgeführt, bis die Iteration in der Position fertig ist und keine ausgelassenen Knotenpunkte mehr bestehen. In der innen Schleife (zweite For-Schleife) wird überprüft, ob die Tour mit dem jetzigen Knotenpunkt j und der Position i die beste Kombination aus den beiden überschlägt. Wenn dieses der Fall ist, wird die beste Kombination mit dem jetzigen Paar überschrieben. Nachdem die Iteration fertig ist, wird das beste Kombinationspaar an die erste Funktion verschickt.

4.3. Saving Algorithmus

Der Saving Algorithmus wurde nach dem Theorieteil Kapitel 2.5 aufgebaut. In diesem Teil gibt es wie bei dem Cheapest Insertion Algorithmus zwei Funktionen. In der Abbildung 15 wird die Erstellung der Saving-Liste abgebildet. Die weiteren Abbildungen beginnend mit Abbildung 16 bis Abbildung 29 sind für die Routenplanung vorgesehen.

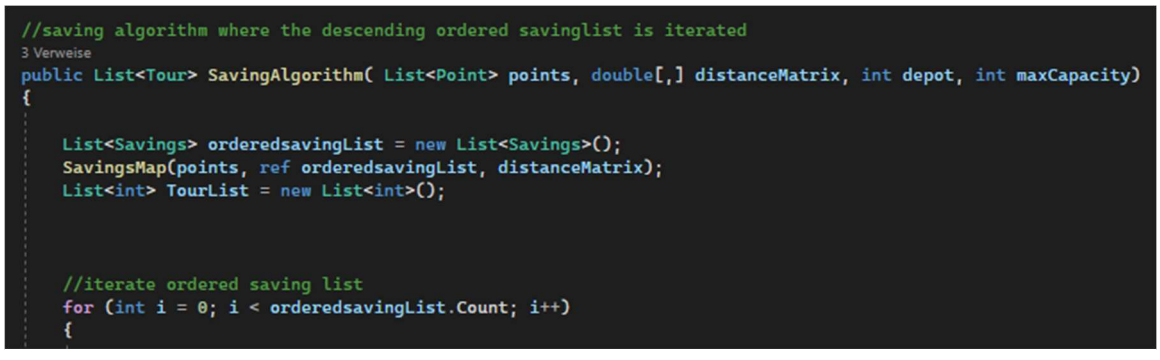
```
//---------------------------------Savingmap-------------------------------
1 Verweis
public void SavingsMap(List<Point> points, ref List<Savings> orderedsavinglist,double[,] distanceMatrixForEuc)
{
    List<Savings> savinglist = new List<Savings>();//new list of class savings

    for (int i = 1; i < points.Count; i++)
    {
        for (int j = 1; j < points.Count; j++)
        {
            bool valueInList = false;
            //query for duplicate values
            foreach (Savings s in savinglist)
            {
                if (s.NodeX == j + 1 & s.NodeY == i + 1)
                {
                    valueInList = true;
                }
            }
            if (i != j)
            {
                //if the values is not included before than include
                if (valueInList == false)
                {
                    double depotToI = distanceMatrixForEuc[0,i];
                    double depotToJ = distanceMatrixForEuc[0,j];
                    double iToj = distanceMatrixForEuc[i,j];
                    double savings = depotToI + depotToJ - iToj;
                    savinglist.Add(new Savings(i + 1, j + 1, savings));
                }
            }
        }


    }
    //order saving list is decending order
    orderedsavinglist = savinglist.OrderByDescending(x => x.SavingsValue).ToList();
}
```

Abbildung 15 - Erzeugung von einer Saving Liste

Die Abbildung 15 berechnet durch die Distanzmatrix die Ersparnisse für jedes einzelne Knotenpunktpaar aus und speichert sie in der Ersparnisliste ab. Diese Vorgehensweise wird mit drei Schleifen gesteuert. Wenn die Erzeugung der Ersparnisliste abgeschlossen wird, findet die Sortierung jener nach der Größe (absteigend) statt. Somit kann die Routenplanung nun mit dieser Liste generiert werden. Die Berechnungsformel für die Ersparnisse wurde in dem Kapitel 2.5 ausführlich erklärt. Diese Funktion heißt „Savingsmap“ und wird gleich am Anfang der Routenplanungsfunktion aufgerufen (siehe Abbildung 16). Danach wird über die erzeugte Ersparnisliste mit einer For-Schleife iteriert.



```
//saving algorithm where the descending ordered savinglist is iterated
3 Verweise
public List<Tour> SavingAlgorithm( List<Point> points, double[,] distanceMatrix, int depot, int maxCapacity)
{

    List<Savings> orderedsavingList = new List<Savings>();
    SavingsMap(points, ref orderedsavingList, distanceMatrix);
    List<int> TourList = new List<int>();



    //iterate ordered saving list
    for (int i = 0; i < orderedsavingList.Count; i++)
    {
```

Abbildung 16 - Saving Algorithmus: Abruf von SavingsMap

Es gib vier Zuordnungsmöglichkeiten die bei der Routenplanung vorkommen (siehe Kapitel 2.5). Die vier Zuordnungsmöglichkeiten wurden in dem Code auf drei Buchstaben

zusammengefasst. Die Zusammenfassung erfolgt bei der Zuordnung von der X-Knotenpunkt und der Y-Knotenpunkt.

```
// a.Either, neither i nor j have already been assigned to a route, in which case a new route is initiated including both i and j.
if (!TourList.Contains(orderedsavingList[i].NodeX) & !TourList.Contains(orderedsavingList[i].NodeY))
{
    TourList.Add(depot);//add depot
    TourList.Add(orderedsavingList[i].NodeX);//add node x
    TourList.Add(orderedsavingList[i].NodeY);//add node y
    TourList.Add(depot);// add depot
}
```

Abbildung 17 - Saving Algorithmus: Zuordnung im Fall A

Die erste Zuordnungsmöglichkeiten ist, wenn weder der X-Knotenpunkt noch der Y-Knotenpunkt in eine Tour verwiesen wurde. In diesem Fall wird eine neue Tour generiert und die beiden Knotenpunkte werden in diese Tour zugeordnet. Dieses Vorgehen wird auch in der Abbildung 17 veranschaulicht.

```
//b. x node is known
else if (TourList.Contains(orderedsavingList[i].NodeX) & !TourList.Contains(orderedsavingList[i].NodeY))
{
    int index = 0;
    int indexPlusone = 0;
    int indexMinusone = 0;
    int dummyCapacity = 0;
    index = TourList.IndexOf(orderedsavingList[i].NodeX);
    indexPlusone = index + 1;
    indexMinusone = index - 1;

    // if after index x follows 0
    if (TourList.ElementAt(indexPlusone) == 1)
    {
        int iterationForCapacity = index;
        //capacity calculation for the tour of x
        while (iterationForCapacity > 0)
        {
            if (TourList.ElementAt(iterationForCapacity) != 1)
            {
                int idOfcapacity = TourList.ElementAt(iterationForCapacity);
                dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
                iterationForCapacity--;
            }
            else { break; }
        }
        //add the new node capacity
        int nodeY = orderedsavingList[i].NodeY;
        dummyCapacity += points.Find(x => x.ID == nodeY).Capacity;
        //query capacity of with insertion is smaller than max capacity
        if (dummyCapacity <= maxCapacity)
        {
            TourList.Insert(index + 1, orderedsavingList[i].NodeY); // insert node y after x
        }

    }
    // if befor index x follows 0
    else if (TourList.ElementAt(indexMinusone) == 1)
    {
        //capacity calculation for the tour of x
        int iterationForCapacity = index;
        while (iterationForCapacity < TourList.Count)
        {
            if (TourList.ElementAt(iterationForCapacity) != 1)
            {
                int idOfcapacity = TourList.ElementAt(iterationForCapacity);
                dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
                iterationForCapacity++;
            }
            else { break; }
        }
        //add the new node capacity
        int nodeY = orderedsavingList[i].NodeY;
        dummyCapacity += points.Find(x => x.ID == nodeY).Capacity;
        //query capacity of with insertion is smaller than max capacity
        if (dummyCapacity <= maxCapacity)
        {
            TourList.Insert(index, orderedsavingList[i].NodeY);//insert node y befor x
        }
    }
```

Abbildung 18 - Saving Algorithmus: Zuordnung im Fall B – X-Knotenpunkt bekannt

Die Abbildung 18 zeigt die Zuordnungsmöglichkeit, wenn der Knotenpunkt x bekannt und der Knotenpunkt y unbekannt ist. Zuerst wird abgefragt, wo das Depot liegt. Das ist deshalb wichtig, um die Verknüpfungsstelle zu anerkennen. Wenn dieses Problem durch die Bestimmung der Position gelöst wurde, kann die Kapazität der Tour überprüft werden. Nach der Kapazitätsüberprüfung ist es möglich den Knotenpunkt in die Tour zu implementieren.

```
//b. y node is known
else if (!TourList.Contains(orderedsavingList[i].NodeX) & TourList.Contains(orderedsavingList[i].NodeY))
{
    int index = 0;
    int indexPlusone = 0;
    int indexMinusone = 0;
    int dummyCapacity = 0;
    index = TourList.IndexOf(orderedsavingList[i].NodeY);
    indexPlusone = index + 1;
    indexMinusone = index - 1;
    // if after index y follows 0
    if (TourList.ElementAt(indexPlusone) == 1)
    {
        int iterationForCapacity = index;
        //capacity calculation for the tour of y
        while (iterationForCapacity > 0)
        {
            if (TourList.ElementAt(iterationForCapacity) != 1)
            {
                int idOfcapacity = TourList.ElementAt(iterationForCapacity);
                dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
                iterationForCapacity--;
            }
            else { break; }
        }
        // add new capacity to capcity
        int nodeX = orderedsavingList[i].NodeX;
        dummyCapacity += points.Find(x => x.ID == nodeX).Capacity;
        //querf if dummy capacity is smaller than maxCapacity
        if (dummyCapacity <= maxCapacity)
        {
            TourList.Insert(index + 1, orderedsavingList[i].NodeX);
        }
    }
    // if befor index y follows 0
    else if (TourList.ElementAt(indexMinusone) == 1)
    {
        int iterationForCapacity = index;
        //capacity calculation for the tour of y
        while (iterationForCapacity < TourList.Count)
        {
            if (TourList.ElementAt(iterationForCapacity) != 1)
            {
                int idOfcapacity = TourList.ElementAt(iterationForCapacity);
                dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
                iterationForCapacity++;
            }
            else { break; }
        }
        // add new capacity to capcity
        int nodeX = orderedsavingList[i].NodeX;
        dummyCapacity += points.Find(x => x.ID == nodeX).Capacity;
        //querf if dummy capacity is smaller than maxCapacity
        if (dummyCapacity <= maxCapacity)
        {
            TourList.Insert(index, orderedsavingList[i].NodeX);
        }
    }
}
```

Abbildung 19 - Saving Algorithmus: Zuordnung im Fall B – Y-Knotenpunkt bekannt

Die Abbildung 19 ist das Gegenstück von der Abbildung 18. Hier ist der Knotenpunkt y bekannt und der Knotenpunkt x unbekannt. Dieselbe Reinfolge wie bei der Abbildung 18 gilt auch hier. Zunächst wird die Position bestimmt, indem der Knotenpunkt x eingesetzt werden soll. Anschließend wird die Kapazität überprüft, ob die Nutzlast von der LKW nicht überschritten wurde. Zum Schluss wird der Knotenpunkt zugeordnet.

```
//c.Or, both i and j have already been included in two different existing routes and
//neither point is interior to its route, in which case the two routes are merged.
else if (TourList.Contains(orderedsavingList[i].NodeX) & TourList.Contains(orderedsavingList[i].NodeY))
{
    int dummyCapacity = 0;
    int nodeX = orderedsavingList[i].NodeX;
    int nodeY = orderedsavingList[i].NodeY;
    int indexX = TourList.IndexOf(orderedsavingList[i].NodeX);
    int indexY = TourList.IndexOf(orderedsavingList[i].NodeY);
    int k = 0;
    bool checkRouteX = false;
    bool checkRouteY = false;
    //check if two values are not in the same route
    while (k < TourList.Count)
    {
        if (TourList[k] == 1)
        {
            k++;
            while (TourList[k] != 1)
            {
                if (TourList[k] == nodeX)
                {
                    checkRouteX = true;

                }
                else if (TourList[k] == nodeY)
                {
                    checkRouteY = true;
                }

                k++;
            }
            if (checkRouteX == true & checkRouteY == true)
            {
                break;
            }
            checkRouteY = false;
            checkRouteX = false;

        }
        k++;
    }
```

Abbildung 20 - Saving Algorithmus: Zuordnung im Fall C-1

Die Abbildung 20 präsentiert den Anfang der letzten Zuordnungsmöglichkeit. In dieser Zuordnung ist die x-Koordinate und die y-Koordinate in zwei verschiedenen Touren vorhanden. Deshalb muss überprüft werden, wie die Verknüpfung von den beiden Touren gelingen kann. Aufgrund dessen wird in der Abbildung 20 kontrolliert, dass die beiden Standorte nicht in der gleichen Tour vorkommen.

```
// if nodes in two different Tours and neither point is interior to its route =>than merge it together
// search after depot and merge it from there

if (checkRouteY == false && checkRouteX == false) // query for two different tours
{
    int indexXMinusOne = indexX - 1;
    int indexYMinusOne = indexY - 1;
    int indexXPlusOne = indexX + 1;
    int indexYPlusOne = indexY + 1;
    //if the x node +1 is depot  and ynode-1 is depot
    if (TourList.ElementAt(indexXPlusOne) == 1 & TourList.ElementAt(indexYMinusOne) == 1)
    {
        //calculate capacity of tour where x is included
        int iterationForCapacity = indexX;
        while (iterationForCapacity > 0)
        {
            if (TourList.ElementAt(iterationForCapacity) != 1)
            {
                int idOfcapacity = TourList.ElementAt(iterationForCapacity);
                dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
                iterationForCapacity--;
            }
            else { break; }
        }
        //calculate capacity of tour where y is inserted and add it to tour where x is inserted
        iterationForCapacity = indexY;
        while (iterationForCapacity < TourList.Count)
        {
            if (TourList.ElementAt(iterationForCapacity) != 1)
            {
                int idOfcapacity = TourList.ElementAt(iterationForCapacity);
                dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
                iterationForCapacity++;
            }
            else { break; }
        }
```

Abbildung 21 - Saving Algorithmus: Zuordnung im Fall C-2

Als nächstes werden die vier Möglichkeiten überprüft, die während der Verknüpfung auftreten können. Die erste Variante wird in der Abbildung 21 vorgeführt. Es wird abgefragt, ob die x-Koordinate der vorletzte Standort in der Tour ist. Weiters wird noch überprüft, ob sich die y-Koordinate in dem zweiten Standort der eigenen Tour befindet. Wenn diese zwei Bedingungen erfüllt sind, wird von den beiden Touren die Kapazität ausgerechnet. Die Kapazitätsberechnung erfolgt durch zwei While-Schleifen. Sie iteriert in den beiden Touren über die Standorte und während dessen werden die Kapazitäten berechnet.

```
//if x inserted tour and y inserted tour capccity smaller than max capacity merge tour
if (dummyCapacity <= maxCapacity)
{
    //check index x smaller than y to know how to implement the values of tour y
    if (indexX < indexY)
    {
        int methodeXplusYminus = indexY;
        while (methodeXplusYminus < TourList.Count) // itereat as long as depot of tour y is reached
        {
            //stop if depot reached
            if (TourList.ElementAt(methodeXplusYminus) != 1)
            {
                TourList.Insert(indexX + 1, TourList.ElementAt(methodeXplusYminus)); // insert the value of y tour in x
                TourList.RemoveAt(methodeXplusYminus + 1);// remove the value of y tour from old tour
                indexX++;
                methodeXplusYminus++;
            }
            else { break; }
        }
    }
    //check index to know how to implement the values of tour y
    else if (indexY < indexX)
    {
        int methodeXplusYminus = indexX;
        while (methodeXplusYminus < TourList.Count) // itereat as long as depot of tour y is reached
        {
            //stop if depot reached
            if (TourList.ElementAt(indexX) != 1)
            {
                TourList.Insert(indexY, TourList.ElementAt(indexX)); // insert the value of y tour in x
                TourList.RemoveAt(indexX + 1);// remove the value of y tour from old tour
                methodeXplusYminus++;
            }
            else { break; }
        }

    }
}
```

Abbildung 22 - Saving Algorithmus: Zuordnung im Fall C-3

Nach dem die Kapazität berechnet wurde, wird abgefragt, ob die Verknüpfung stattfinden kann. Dies Abfrage wird in der Abbildung 22 durchgeführt. Wenn die Kapazität zulässt, gelingt die Verknüpfung der beiden Touren. Dafür führt das Programm eine If-Abfrage durch und bestimmt, ob der Index von x-Knotenpunkt größer oder kleiner ist. Wenn das bestimmt wurde, kann die Zuordnung je nach der Größe des Indexes erfolgen.

```
//if the x node +1 is depot  and ynode+1 is depot
else if (TourList.ElementAt(indexXPlusOne) == 1 & TourList.ElementAt(indexYPlusOne) == 1)
{
    //calculate capacity of tour, where x is inserted
    int iterationForCapacity = indexX;
    while (iterationForCapacity > 0)
    {
        if (TourList.ElementAt(iterationForCapacity) != 1)
        {
            int idOfcapacity = TourList.ElementAt(iterationForCapacity);
            dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
            iterationForCapacity--;
        }
        else { break; }
    }
    //calculate capacity of tour, where y is inserted
    iterationForCapacity = indexY;
    while (iterationForCapacity > 0)
    {
        if (TourList.ElementAt(iterationForCapacity) != 1)
        {
            int idOfcapacity = TourList.ElementAt(iterationForCapacity);
            dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
            iterationForCapacity--;
        }
        else { break; }
    }
```

Abbildung 23 - Saving Algorithmus: Zuordnung im Fall C-4

In der Abbildung 23 wird die zweite Option durchgeführt. In dem abgefragt wird, ob die beiden Knotenpunkte in der vorletzten Stelle in ihrer Routenplanung sind. Danach wird die aktuelle Kapazität von beiden Touren berechnet.

```
//check capacity
if (dummyCapacity <= maxCapacity)
{
    //check index to know how to implement the values of tour y
    if (indexX < indexY)
    {
        int methodeXplusYplus = indexY;
        while (methodeXplusYplus > 0)
        {
            //stop if 0 reached
            //insert at x
            if (TourList.ElementAt(methodeXplusYplus) != 1)
            {
                TourList.Insert(indexX + 1, TourList.ElementAt(methodeXplusYplus)); // insert the value of y tour in x
                TourList.RemoveAt(methodeXplusYplus + 1);// remove the value of y tour from old tour
                indexX++;
            }
            else { break; }
        }
    }
    //check index to know how to implement the values of tour y
    else if (indexX > indexY)
    {
        int methodeXplusYplus = indexY;
        while (methodeXplusYplus > 0)
        {
            //stop if 0 reached
            if (TourList.ElementAt(methodeXplusYplus) != 1)
            {
                TourList.Insert(indexX + 1, TourList.ElementAt(methodeXplusYplus));// insert the value of y tour in x
                TourList.RemoveAt(methodeXplusYplus);// remove the value of y tour from old tour

                methodeXplusYplus--;
            }
            else { break; }
        }
    }

}
```

Abbildung 24 - Saving Algorithmus: Zuordnung im Fall C-5

Falls die Kapazität nicht die Nutzlast von dem LKW übersteigt, können die beiden Touren miteinander verknüpft werden (siehe Abbildung 24). Dabei wird nochmals überprüft, wie die Vereinigung stattfinden soll. Nachdem die Vereinigungsstelle bestimmt wurde, kann die Verknüpfung stattfinden.

```
//if the x node -1 is depot  and ynode-1 is depot
else if (TourList.ElementAt(indexXMinusOne) == 1 & TourList.ElementAt(indexYMinusOne) == 1)
{
    //calculate capacity of tour, where x is inserted
    int iterationForCapacity = indexX;
    while (iterationForCapacity < TourList.Count)
    {
        if (TourList.ElementAt(iterationForCapacity) != 1)
        {
            int idOfcapacity = TourList.ElementAt(iterationForCapacity);
            dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
            iterationForCapacity++;
        }
        else { break; }
    }
    //calculate capacity of tour, where y is inserted
    iterationForCapacity = indexY;
    while (iterationForCapacity < TourList.Count)
    {
        if (TourList.ElementAt(iterationForCapacity) != 1)
        {
            int idOfcapacity = TourList.ElementAt(iterationForCapacity);
            dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
            iterationForCapacity++;
        }
        else { break; }
    }
```

Abbildung 25 - Saving Algorithmus: Zuordnung im Fall C-6

In der Abbildung 25 wird die dritte Option für die Verknüpfung dargestellt. Dabei sind beide Knotenpunkte an der zweiten Stelle in den jeweiligen Touren. Hier wird es wie bei den anderen Optionen zuerst die Kapazitätseinschränkung überprüft.

```
    //check capacity
    if (dummyCapacity <= maxCapacity)
    {
        //check index to know how to implement the values of tour y
        if (indexX < indexY)
        {
            int methodeXminusYminus = indexY;
            while (methodeXminusYminus < TourList.Count)
            {
                //stop if zero reached
                if (TourList.ElementAt(methodeXminusYminus) != 1)
                {
                    TourList.Insert(indexX, TourList.ElementAt(methodeXminusYminus));// insert the value of y tour in x
                    TourList.RemoveAt(methodeXminusYminus + 1);// remove the value of y tour from old tour
                    methodeXminusYminus++;
                }
                else { break; }
            }
        }
        //check index to know how to implement the values of tour y
        else if (indexX > indexY)
        {
            int methodeXminusYminus = indexY;
            while (methodeXminusYminus < TourList.Count)
            {
                //stop if zero reached
                if (TourList.ElementAt(methodeXminusYminus) != 1)
                {
                    TourList.Insert(indexX, TourList.ElementAt(methodeXminusYminus));// insert the value of y tour in x
                    TourList.RemoveAt(methodeXminusYminus);// remove the value of y tour from old tour
                    indexX--;
                }
                else { break; }
            }
        }

    }


}
```

Abbildung 26 - Saving Algorithmus: Zuordnung im Fall C-7

Nach der Kapazitätsüberprüfung wird die Verknüpfungsstelle ausgesucht (s. Abbildung 26). Durch die Bestimmung der Verknüpfungsstelle werden die beiden Touren gematcht.

```
//if the xnode - 1 is depot and ynode + 1 depot
else if (TourList.ElementAt(indexXMinusOne) == 1 & TourList.ElementAt(indexYPlusOne) == 1)
{
    //calculate capacity of tour, where x is inserted
    int iterationForCapacity = indexX;
    while (iterationForCapacity < TourList.Count)
    {
        if (TourList.ElementAt(iterationForCapacity) != 1)
        {
            int idOfcapacity = TourList.ElementAt(iterationForCapacity);
            dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
            iterationForCapacity++;
        }
        else { break; }
    }
    //calculate capacity of tour, where y is inserted
    iterationForCapacity = indexY;
    while (iterationForCapacity > 0)
    {
        if (TourList.ElementAt(iterationForCapacity) != 1)
        {
            int idOfcapacity = TourList.ElementAt(iterationForCapacity);
            dummyCapacity += points.Find(x => x.ID == idOfcapacity).Capacity;
            iterationForCapacity--;
        }
        else { break; }
    }
```

Abbildung 27 - Saving Algorithmus: Zuordnung im Fall C-8

Die letzte Option für die Verknüpfung wird in der Abbildung 27 - demonstriert. In diesem Fall ist der x Knotenpunkt auf der zweiten Stelle in der Tour. Der y Knotenpunkt sollte an der vorletzten Stelle liegen. Wenn diese Option zutrifft, findet eine Kapazitätsüberprüfung statt.

```
            //check capacity
            if (dummyCapacity <= maxCapacity)
            {
                //check index to know how to implement the values of tour y
                if (indexX < indexY)
                {
                    int methodeXminusYplus = indexY;
                    while (methodeXminusYplus < TourList.Count)
                    {
                        //stop if zero reached
                        if (TourList.ElementAt(methodeXminusYplus) != 1)
                        {
                            TourList.Insert(indexX, TourList.ElementAt(methodeXminusYplus));// insert the value of y tour in x
                            TourList.RemoveAt(methodeXminusYplus + 1);// remove the value of y tour from old tour

                        }
                        else { break; }


                    }
                }
                //check index to know how to implement the values of tour y
                else if (indexY < indexX)
                {

                    int methodeXminusYplus = indexY;
                    while (methodeXminusYplus < TourList.Count)
                    {
                        //stop if zero reached
                        if (TourList.ElementAt(methodeXminusYplus) != 1)
                        {
                            TourList.Insert(indexX, TourList.ElementAt(methodeXminusYplus));// insert the value of y tour in x
                            TourList.RemoveAt(methodeXminusYplus);// remove the value of y tour from old tour
                            methodeXminusYplus--;
                            indexX--;

                        }
                        else { break; }
                    }
                }
            }
        }
    }
}
```

Abbildung 28 - Saving Algorithmus: Zuordnung im Fall C-9

Ist die Kapazitätsüberprüfung in Ordnung, werden die Verknüpfungsstellen überprüft und schlussendlich die beiden Touren miteinander verknüpft (s. Abbildung 28).

```
    // delete toures with only depot
    for (int j = 0; j < TourList.Count - 2; j++)
    {
        if (TourList[j] == 1 && TourList[j + 1] == 1 && TourList[j + 2] == 1)
        {

            TourList.RemoveAt(j);
            TourList.RemoveAt(j);

        }
    }
}
List<Tour> savingAlgoTourList = new List<Tour>();
List<int> dummyListForSavingAlgo = new List<int>();
List<List<int>> dummysavingAlgoTourList = new List<List<int>>();
for (int i = 0; i < TourList.Count; i++)
{
    if (TourList[i] == 1)
    {
        dummyListForSavingAlgo.Add(TourList[i]);
        i++;
        for (int j = i; j < TourList.Count; j++)
        {
            if (TourList[j] != 1)
            {
                dummyListForSavingAlgo.Add(TourList[j]);
            }
            else
            {
                dummyListForSavingAlgo.Add(TourList[j]);

                i = j;
                break;
            }
        }
    }
    dummysavingAlgoTourList.Add(new List<int>(dummyListForSavingAlgo));
    dummyListForSavingAlgo.Clear();
}
```

Abbildung 29 - Saving Algorithmus: Löschung von überflüssigen Touren

In zwei Zuordnungsmöglichkeiten (B und C) bleiben leere Touren übrig, weil die Standorte von einer Tour in die andere übertragen werden. Deshalb müssen die übrig gebliebenen Touren gelöscht werden. Dieser Prozess wird in der Abbildung 29 demonstriert. Es wird nach Touren gesucht, in denen zweimal das Depot übriggeblieben ist. Nur unter dieser Voraussetzung werden die Touren gelöscht. Nachdem die Tourenplanung fertig erstellt wurde, wird die gesamte Tourenplanung in die „savingAlgoTourList“- Liste abgespeichert. Dadurch wird die Tourenplanung mit dem Saving Algorithmus abgeschlossen.

4.4. Genetischer Algorithmus

In diesem Abschnitt wird das Programm des genetischen Algorithmus vorgestellt. Dabei wird zuerst die Umsetzung von dem genetischen Algorithmus in der Programmiersprache mit den Basisfunktionen demonstriert. Danach werden die einzelnen Erweiterungen, die durch diese Arbeit geplant, entwickelt, umgesetzt und getestet wurden, präsentiert. Dabei gibt es vier

Versionen bzw. Erweiterungsstufen. Das Programm wird fünfmal von neu gestartet. Hierbei werden alle berechneten Werte gelöscht (Hard-Reset). Weiters werden in dem Programm die Routenplanungsberechnungen 100-mal durchgeführt. In diesen 100 Durchführungen wird immer abgefragt, ob die gefundenen Erkenntnisse beziehungsweise Population besser sind als bis zu diesem Zeitpunkt errechneten Routenplanungen. Diese Abfrage findet erst statt, wenn die Abbruchbedingung erfüllt ist. Die Abbruchbedingung wird in dem Kapitel 4.4.1 näher erklärt. Nachdem die Abbruchbedingung erfolgt, findet eine erneuerte Populationserzeugung statt (Soft-Reset). Bei dem Soft-Reset wird nur die Population von neu generiert. Diese Vorgehensweise wird für Verbesserungsansätze benötigt, die ab dem Kapitel 4.4.2 präsentiert werden.

4.4.1. Herkömmlicher genetischer Algorithmus

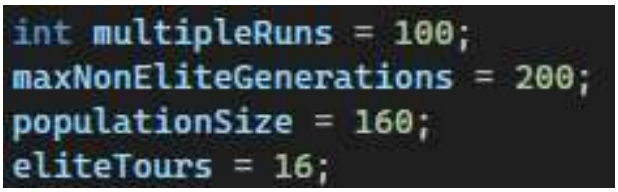


Abbildung 30 - Genetische Algorithmus: Initialisierungsparameter

Die Initialisierungsparameter werden in der Abbildung 30 vorgeführt. Der „populationSize“-Parameter gibt an, wie viele Tourenplanungen im genetischen Algorithmus generiert werden. Die Variable „eliteTours“ gibt an, wie viele Tourenplanungen in die nächste Runde des genetischen Algorithmus übertragen wird. Dabei werden die besten Tourenplanungen aus der Liste genommen. Der genetische Algorithmus wird nach einem bestimmten Durchlauf beendet. Dafür wird auf nicht verbesserte Tourenplanungen geachtet. Sondern, wenn die beste erzeugte Tourenplanung 200 Runden lang sich nicht verbessert, wird der genetische Algorithmus beendet.

```
public void RandomTourgeneration(List<Point> pointList, int maxCapacity, double[,] distanceMatrix)
{

    List<int> listForRandomNotVisited = new List<int>();
    for (int i = 0; i < pointList.Count; i++)
    {
        listForRandomNotVisited.Add(pointList[i].ID);
    }
    listForRandomNotVisited.Remove(1);
    List<List<int>> randomList = new List<List<int>>();
    while (listForRandomNotVisited.Count > 0)
    {
        List<int> dummyRandomList = new List<int>();

        int tourcapacity = 0;

        bool breakFlag = false;
        while (!breakFlag)
        {
            if (listForRandomNotVisited.Count == 0)
            {
                break;
            }
            int pointToInsert = rnd.Next(listForRandomNotVisited.Count);
            pointToInsert = listForRandomNotVisited[pointToInsert];
            pointToInsert = pointList[pointToInsert - 1].ID;
            int addcapacity = pointList[pointToInsert - 1].Capacity;

            if (tourcapacity + addcapacity <= maxCapacity)
            {
                tourcapacity += addcapacity;
                dummyRandomList.Add(pointToInsert);
                listForRandomNotVisited.Remove(pointToInsert);
            }
            else
            {
                breakFlag = true;
            }


        }
        dummyRandomList.Insert(0, 1);
        dummyRandomList.Add(1);
        randomList.Add(new List<int>(dummyRandomList));

    }
```

Abbildung 31 - Genetischer Algorithmus: Initialisierung 1

In der Abbildung 31 wird die Initialisierung des genetischen Algorithmus programmiert. Hier wird vorerst eine neue Liste für die Routenplanung erzeugt. In dieser Routenplanung werden die Touren abgespeichert. Nachdem wird eine zweite Liste für die Standorte erzeugt, die besucht werden müssen. Aus der Liste mit den Standorten werden zufällige Knotenpunkte ausgesucht und in eine Tour abgespeichert. Die ausgesuchten Standorte werden von der Liste „nicht besuchte Standorte“ gelöscht. Diese Vorgehensweise wird so lange durchgeführt, bis die Kapazitätsbeschränkung einer Tour nicht mehr zulässt. Wenn noch „nicht besuchte Standorte“ in der Liste besteht, wird eine neue Tour geöffnet. Dieselbe Vorgehensweise fängt von vorne an, bis keine besuchten Standorte mehr in der Liste vorhanden sind.

```
    List<Tour> randomTourList = new List<Tour>();
    foreach (List<int> point in randomList)
    {
        int dummyCapacity = 0;
        List<int> dummyListToHold = new List<int>();
        dummyListToHold = point;
        for (int i = 0; i < dummyListToHold.Count; i++)
        {
            dummyCapacity += pointList[dummyListToHold[i] - 1].Capacity;
        }

        randomTourList.Add(new Tour(point, 0, 0, dummyCapacity));
    }
    randomTourList = CalculationOfTourDistance(pointList,distanceMatrix,randomTourList);
    TourOfChromosome = randomTourList;
    TotalDistanceOfTour();
    CalculateFitness();

}
```

Abbildung 32 - Genetischer Algorithmus: Initialisierung 2

In der Abbildung 32 wird die erzeugte Liste „randomList“ in einzelne Touren zerlegt und im Routenplanung Liste „randomTourList“ abgespeichert. Danach werden die zwei Unterprogramme aufgerufen, bei denen die Routenplanungsentfernung und der Fitnesswert der Routenplanung ausgerechnet werden.

```
// Calculation of fitness
2 Verweise
public void CalculateFitness()
{
    Fitness = 0;
    Fitness = 1 / (Math.Pow(Distance, 8)) ;
}
```

Abbildung 33 - Genetischer Algorithmus: Fitnessberechnung

Die Abbildung 33 bildet die Berechnung von Fitness aus. Die Beschreibung von Fitness wurde in Kapitel 2.6.2 ausführlich beschrieben. Die Funktion gibt den berechneten Fitnesswert als Objektvariable zurück. Sie berechnet sich aus 1/Distanz^8.

```
10 Verweise
public List<Tour> TourOfChromosome { get; set; }
15 Verweise
public double Distance { get; set; }
23 Verweise
public double Fitness { get; set; }

3 Verweise
public void TotalDistanceOfTour()
{
    Distance = 0;
    for(int i = 0; i < TourOfChromosome.Count; i++)
    {
        Distance+= TourOfChromosome[i].Distance;
    }
}
private static Random rnd=new Random();

2 Verweise
public List<Tour> CalculationOfTourDistance(List<Point> pointList, double[,] distanceMatrix, List<Tour> randomList)
{
    for (int i = 0; i < randomList.Count; i++)
    {
        double dummyCapacity = 0;
        List<int> dummyListToHold = new List<int>();
        dummyListToHold = randomList[i].ListofTours;
        for (int j = 0; j < dummyListToHold.Count-1; j++)
        {
            dummyCapacity += distanceMatrix[dummyListToHold[j] - 1, dummyListToHold[j + 1] - 1];
        }

        randomList[i].Distance = dummyCapacity;

    }


    //add last to first point distance
    return randomList;
}
```

Abbildung 34 - Genetischer Algorithmus: Entfernungsberechnung

Es gibt zwei Entfernungsberechnungsfunktionen. In der ersten Funktion wird die Entfernung von einer Tour ausgerechnet. In der zweiten Funktion wird die Entfernung für die gesamte Tourenplanung ermittelt. Die zwei Funktionen werden in der Abbildung 34 illustriert. Die Funktion „CalculationOfTourDistance“ berechnet für jede einzelne Tour in der Routenplanung die Entfernung aus. Diese Funktion iteriert über die gesamte Tour und berechnet in der Tour die Entfernung zwischen jedem Standort. Schlussendlich wird die gesamte Entfernung von der Tour ermittelt. Damit die gesamte Entfernung für die Routenplanung berechnet wird, dient die Funktion „TotalDisatnceOfTour“. Diese Funktion addiert die Tourenentfernungen in eine Routenplanung. Dabei geht die Funktion schrittweise die Touren durch und addiert die berechnete Tourenentfernungen.

```
1 Verweis
public void Initialization(List<Point> PointList, int populationSize, double[,] distanceMatrix, int maxCapacity)
{
    for (int i = 1; i <= populationSize; i++)
    {
        var chromosome = new Chromosome();
        chromosome.RandomTourgeneration(PointList, maxCapacity, distanceMatrix);
        PopulationList.Add(chromosome);
    }

    //sort population by desending order
    PopulationList = PopulationList.OrderByDescending(chromosome => chromosome.Fitness).ToList();

}
```

Abbildung 35 - Genetische Entfernung - Initialisierung Funktion

In der Abbildung 35 werden Routenplanungen in der Größe der Population generiert. Dabei wird die Funktion in der Abbildung 31 und Abbildung 32 abgerufen. Die generierte Routenplanungen werden in die „PopulationListe“ abgespeichert.

```
public Chromosome RouletteSelection()
{
    //sum fitness of population
    double SumOfFitness = PopulationList.Sum(chromosome => chromosome.Fitness);
    //generate random fitness
    double randomFitness = random.NextDouble() * SumOfFitness;
    double cumlativeProbaility = PopulationList[0].Fitness;

    for (int i = 0; i < PopulationList.Count; i++)
    {
        if (randomFitness < cumlativeProbaility)
        {
            return PopulationList[i];
        }
        cumlativeProbaility += PopulationList[i + 1].Fitness;
    }

    return PopulationList.First();
}
```

Abbildung 36 - Genetischer Algorithmus: RouletteSelection

Die Abbildung 36 und Abbildung 37 zeigen die zwei Selektionsarten, welche im Kapitel 2.6.3 ausführlich beschrieben wurden. Die Abbildung 36 zeigt die Roulette-Selektion. Sie kennzeichnet sich dadurch, dass von der gesamten Population die Fitnesswerte aufsummiert werden. Danach wird dieser Fitnesswert mit einer zufälligen Zahl multipliziert. Eine weitere Variable wird für die Berechnung vom kumulativen Fitnesswert deklariert. Dieser Wert wird durch die zufällige Auswahl von den Routenplanungen aus der Populationsliste errechnet. Die For-Schleife wird so lange durchgeführt, bis die zufällige Fitnesszahl kleiner ist als die kumulative Fitnesszahl.

```
public Chromosome TournamentSelection()
{
    //tournament size
    int pickK = 5;
    //random pick
    int index = random.Next(0, PopulationList.Count);

    int bestIndex = index;

    for (int i = 0; i < pickK; i++)
    {
        //overwrite index with new random value
        index = random.Next(0, PopulationList.Count);
        //if index greater then bestindex then overwrite best index with index
        if (PopulationList[bestIndex].Fitness < PopulationList[index].Fitness)
        {
            bestIndex = index;

        }
    }
    return PopulationList[bestIndex];
}
```

Abbildung 37 - Genetischer Algorithmus: TournamentSelection

Die Abbildung 37 präsentiert die Tournament-Selektion. Für die Tournament-Selektion wurde die Tournament-Größe mit fünf bestimmt. Die Tournament-Größe gibt an, wie viele Routenplanungen aus der Population entnommen werden. Am Anfang der Tournament-Selektion wird die erste zufällig ausgewählte Routenplanung als bester Index definiert. Danach werden mittels der For-Schleife vier weitere Routenplanungen von der Populationsliste abgefragt und mit dem besten Index verglichen. Wenn die neue Routenplanung besser ist als die vorherige, dann wird der beste Index mit dem neuen Index überschrieben.

```
public List<Tour> OrderedCrossover(List<int> parentOne, List<int> parentTwo, List<Point> PointList, double[,] distanceMatrix, int maxCapacity)
{
    int crossoverPointOne = random.Next(0, parentOne.Count - 1);

    int crossoverPointTwo = random.Next(0, parentTwo.Count - 1);
    if (crossoverPointOne > crossoverPointTwo)
    {
        int dummy = crossoverPointTwo;
        crossoverPointTwo = crossoverPointOne;
        crossoverPointOne = dummy;
    }
    //genes to copy
    List<int> genesToCopy = parentOne.GetRange(crossoverPointOne, crossoverPointTwo - crossoverPointOne);
    //genes to fill
    List<int> genesToFill = parentTwo.Except(genesToCopy).ToList();

    // create child tour
    List<int> childTour = new List<int>();
    childTour = genesToFill.Take(crossoverPointOne)
        .Concat(genesToCopy)
        .Concat(genesToFill.Skip(crossoverPointOne))
        .ToList();

    List<Tour> tourList = GeneratingTourwithDepot(PointList, childTour, distanceMatrix, maxCapacity);
    return tourList;
}
```

Abbildung 38 - Genetischer Algorithmus: Ordered Crossover

In den nächsten zwei Funktionen handelt es sich um ein Crossover, das im Kapitel 2.6.4 vorgestellt wurde. Die Abbildung 38 stellt die „OrderedCrossover“ vor. Der Code beginnt mit der Auswahl von zwei Crossover-Punkten. Diese werden zufällig aus der Routenplanung entnommen. Danach wird überprüft, welcher der beiden ausgewählten Indizes größer ist. Nachdem wird die Child-Tour durch die Gene von den beiden Elternpaaren erzeugt. Es werden dabei bestimmte Indizes von dem ersten Elternpaar kopiert und die übrig gebliebenen Standorte von dem zweiten Elternpaar. Als letzter Schritt wird die erzeugte Child-Tour in die Tour-Liste gespeichert. Somit wird das Ordered-Crossover beendet.

In der nächsten Abbildung (Abbildung 39) wird die zweite Crossover-Funktion vorgestellt. Dieses ist das Universal-Crossover. Wie auch in dem Kapitel 2.6.4 angeführt, wurde es nach der wissenschaftlichen Arbeit von Misevičius et al. (2018) nachgebaut. Diese Crossover-Funktion basiert auf die binäre Zahlenfolge. Die binäre Zahlenfolge ist ausschlaggebend für die Entscheidung von welchem Elternpaar die Standorte ausgewählt werden sollen.

```
public List<Tour> UniversalCrossover(List<int> parentOne, List<int> parentTwo, List<Point> PointList, double[,] distanceMatrix, int maxCapacity)
{
    List<int> childTourForUNIX = new List<int>();

    List<int> randomBitMask = new List<int>();
    for (int i = 0; i < PointList.Count; i++)
    {
        randomBitMask.Add(random.Next(0, 2));

    }
    List<int> randomselectlist = new List<int>();
    randomselectlist = Enumerable.Range(2, PointList.Count - 1).ToList();
    for (int i = 0; i < PointList.Count - 1; i++)
    {
        int index = 0;
        int nodeOfparentOne = parentOne[i];
        int nodeOfparentTwo = parentTwo[i];
        //add common genes
        if (parentOne[i] == parentTwo[i] && !childTourForUNIX.Contains(nodeOfparentOne))
        {
            childTourForUNIX.Add(nodeOfparentOne);
            index = randomselectlist.IndexOf(nodeOfparentOne);
            randomselectlist.RemoveAt(index);
        }
        else if (randomBitMask[i] == 1 && !childTourForUNIX.Contains(nodeOfparentOne)) //add gene of parent one if 1
        {
            childTourForUNIX.Add(nodeOfparentOne);
            index = randomselectlist.IndexOf(nodeOfparentOne);
            randomselectlist.RemoveAt(index);
        }
        else if (randomBitMask[i] == 0 && !childTourForUNIX.Contains(nodeOfparentTwo))
        {
            childTourForUNIX.Add(nodeOfparentTwo);
            index = randomselectlist.IndexOf(nodeOfparentTwo);
            randomselectlist.RemoveAt(index);
        }
        else
        {
            int f = 0;
            while (f < 1)
            {
                int indexOfRandomChoose = random.Next(randomselectlist.Count);
                int randomValue = randomselectlist[indexOfRandomChoose];
                if (!childTourForUNIX.Contains(randomValue))
                {
                    childTourForUNIX.Add(randomValue);
                    index = randomselectlist.IndexOf(randomValue);
                    randomselectlist.RemoveAt(index);
                    f++;
                }
            }
        }

    }
```

Abbildung 39 - Genetischer Algorithmus: Universal Crossover

Der Universal-Crossover-Code beginnt mit der Erstellung einer Bit-Maske. Diese Bit-Maske wird zufällig generiert. Darauf wird über die erzeugte Bit-Maske iteriert. Wenn die Position eine 1 aufweist, werden die Gene von dem ersten Elternpaar an der Stelle genommen. Angenommen, dass die Bit-Maske eine 0 aufweist. Dann werden die Gene von dem zweiten Elternpaar genommen. Falls diese Gene, die eingefügt werden sollen, bereits implementiert wurden, wird ein zufällige Gene in der Tour ausgewählt und eingefügt. Dieses wird so lange durchgeführt bis keine offenen Standorte bzw. Gene mehr übrig sind, die noch eingefügt werden sollten. Demzufolge wird die Funktion beendet und die Child-Tour wird mit Hilfe des Universal-Crossovers erstellt.

Als nächster Schritt kommen die Mutation Funktionen. Für die Mutation gibt es ebenso zwei Funktionen, deren Theorie im Kapitel 2.6.5 ausführlich beschrieben wurde. Einerseits wird die Inverse-Mutation im Code veranschaulicht, wobei anderseits die Swap-Mutation präsentiert wird.

```
public List<Tour> inverseMutation(List<int> childTour, List<Point> PointList, double[,] distanceMatrix, int maxCapacity)
{
    int index = 0;
    int indexTwo = 0;
    //avoid same indicies
    while (index == indexTwo)
    {
        index = random.Next(0, childTour.Count - 1);
        indexTwo = random.Next(0, childTour.Count - 1);
    }
    if (indexTwo < index)
    {
        int dummy = index;
        index = indexTwo;
        indexTwo = dummy;
    }
    //reverese tour
    childTour.Reverse(index, indexTwo - index);
    List<Tour> tourList = GeneratingTourwithDepot(PointList, childTour, distanceMatrix, maxCapacity);
    return tourList;
}
```

Abbildung 40 - Genetischer Algorithmus: Inverse Mutation

In dem oben abgebildeten Code (siehe Abbildung 40) wird die Inverse Mutation vorgestellt. Zuerst werden zwei Index-Variablen definiert. Durch die While-Schleife wird garantiert, dass die beiden Indizes nicht gleich groß sind. Dabei werden die Indizes zufällig ausgesucht. Nachdem das Auswahlverfahren beendet wurde, wird abgefragt, welches der beiden Indizes größer ist. Wenn die Positionen bereits bestimmt wurden, werden die Standorte zwischen den beiden Indizes umgekehrt zugeordnet. Zum Schluss wird die Funktion „GeneratingTourwithDepot“ aufgerufen. Die Funktion „GeneratingTourwithDepot“ wird im Nachgang erklärt.

```
public List<Tour> swapMutation(List<int> childTour, List<Point> PointList, double[,] distanceMatrix, int maxCapacity)
{
    for (int i = 0; i < 1; i++)
    {
        int index = 0;
        int indexTwo = 0;
        //avoid same indicies
        while (index == indexTwo)
        {
            index = random.Next(0, childTour.Count - 1);
            indexTwo = random.Next(0, childTour.Count - 1);
        }
        int dummy = childTour[indexTwo];
        childTour[indexTwo] = childTour[index];
        childTour[index] = dummy;
    }

    List<Tour> tourList = GeneratingTourwithDepot(PointList, childTour, distanceMatrix, maxCapacity);

    return tourList;
}
```

Abbildung 41 - Genetischer Algorithmus: Swap Mutation

Die Abbildung 41 illustriert die Swap-Mutation. Zu Beginn wird, wie auch bei der Inverse Mutation zwei Indizes bestimmt. Diese werden wieder durch eine While-Schleife durchgeführt. Nachdem die beiden Indizes bestimmt wurden, tauscht der Code die Plätze von den beiden Standorten. Am Ende des Codes wird die Funktion „GeneratingTourwithDepot“ aufgerufen.

Im Nachfolgenden werden die Funktionen für die Tourenplanungen mit Depot und ohne Depot präsentiert. Bei der Bestimmung von der Child Tour werden die Depots rausgelöscht und ein String aus den Standorten gebildet. Somit kann der genetische Algorithmus parallele Routenplanungen durchführen. Das heißt, dass die Touren in der Routenplanung nicht einzeln konstruiert werden, sondern eine parallele Routenplanung für alle Touren stattfindet. Die Löschung der Depots wird in der Funktion „GeneratingTourwithoutDepot“ verwirklicht. Nachdem die Mutation durchgeführt wird, ruft das Programm die Funktion „GeneratingTourwithDepot“ auf. Diese Funktion teilt wieder die Child Tour in Touren auf und fügt die Depots dazu. Dementsprechend hat eine Routenplanung die Touren nach der jeweiligen Kapazität aufgeteilt.

```
private List<int> GeneratingTourwithoutDepot(Chromosome parent)
{
    List<int> parentWithoutDepo = new List<int>();
    for (int j = 0; j < parent.TourOfChromosome.Count; j++)
    {
        List<List<int>> dummyParentWithoutDepo = new List<List<int>>();
        dummyParentWithoutDepo.Add(parent.TourOfChromosome[j].ListofTours);
        for (int k = 0; k < dummyParentWithoutDepo[0].Count - 1; k++)
        {
            if (dummyParentWithoutDepo[0][k] == 1)
            {
                k++;
                parentWithoutDepo.Add(dummyParentWithoutDepo[0][k]);
            }
            else
            {
                parentWithoutDepo.Add(dummyParentWithoutDepo[0][k]);
            }
        }
    }


    return parentWithoutDepo;
}
```

Abbildung 42 - Genetischer Algorithmus: Löschung von Depots und Erzeugung einer String

In der Abbildung 42 wird die Funktion „GeneratingTourwithoutDepot“ demonstriert. Sie iteriert über alle Touren in der Elternpaartour. Die Iteration über die Listen wird durch zwei For-Schleifen durchgeführt. Die erste For-Schleife geht einzelne Touren in der Routenplanung durch. Bei der zweiten For-Schleife werden die Standorte in der jeweiligen Tour iteriert. Dabei werden die Depots gelöscht und die Standorte als String in eine Liste abgespeichert. Diese Liste wird dann für die nächsten Schritte des genetischen Algorithmus zur Verfügung gestellt.

```
private List<Tour> GeneratingTourwithDepot(List<Point> PointList, List<int> childTour, double[,] distanceMatrix, int maxCapacity)
{
    int tourcapacity = 0;
    List<Tour> tourList = new List<Tour>();
    List<List<int>> tempTour = new List<List<int>>();
    List<int> temp = new List<int>();
    for (int i = 0; i < childTour.Count; i++)
    {
        int pointToInsert = childTour[i];
        int addcapacity = PointList[pointToInsert - 1].Capacity;
        if (tourcapacity + addcapacity <= maxCapacity)
        {
            tourcapacity += addcapacity;
            temp.Add(pointToInsert);
            if (i == childTour.Count - 1)
            {
                temp.Insert(0, 1);
                temp.Add(1);
                tempTour.Add(new List<int>(temp));
            }
        }
        else
        {
            temp.Insert(0, 1);
            temp.Add(1);
            tempTour.Add(new List<int>(temp));
            temp.Clear();
            tourcapacity = 0;
            i--;
        }
    }
    foreach (List<int> point in tempTour)
    {
        int dummyCapacity = 0;
        List<int> dummyListToHold = point;
        for (int i = 0; i < dummyListToHold.Count; i++)
        {
            dummyCapacity += PointList[dummyListToHold[i] - 1].Capacity;
        }

        tourList.Add(new Tour(point, 0, 0, dummyCapacity));
    }
    Chromosome childTourChromosome = new Chromosome();
    tourList = childTourChromosome.CalculationOfTourDistance(PointList, distanceMatrix, tourList);
    return tourList;
}
```

Abbildung 43 - Genetischer Algorithmus: Wieder Erzeugung von Routenplanung

Die Abbildung 43 präsentiert die Umwandlung der Routenplanungen. Hier werden die rausgefilterten Depots wieder in die Tourenplanung implementiert. Dabei wird die erzeugte Childtour mit einer For-Schleife iteriert. Dadurch wird eine neue Tour erzeugt. Die Standorte der Childtour werden in die neue Tour eingefügt, bis die Kapazitätseinschränkung nicht mehr zulässt. Nachdem eine Tour voll ist, wird mit der zweiten Tour angefangen. Das wird so lange wiederholt, bis die Childtour zur Gänze durchspielt wurde.

```
if (PopulationList[0].Fitness > eliteChromosome.Fitness)
{
    eliteChromosome = PopulationList[0];
    nonEliteGenerations = 0;
    if (eliteChromosome.Distance < saveTheBestRound.Distance | z == 1)
    {
        roundSave = PopulationList;
        saveTheBestRound = eliteChromosome;
    }
}
```

Abbildung 44 - Genetischer Algorithmus: Abspeicherung von der Elite-Chromosome

Die obige Abbildung (Abbildung 44) illustriert das Abspeichern der besten Routenplanung und deren Population. Durch zwei IF-Abfragen wird ermittelt, ob die aktuelle Routenplanung besser

ist als die gespeicherte Routenplanung. Wenn das der Fall ist, wird die Variable „eliteChromosome“ mit dem aktuellen Wert überschrieben. Wenn die erzeugte „eliteChromosome“ Variable besser ist als die gespeicherte Variable, dann wird die Liste „roundSave“ mit der aktuellen Populationsliste überschrieben. Das Abspeichern der besten Rundenpopulation wird für den Verbesserungsansatz 2 in Kapitel 4.4.3 benötigt.

4.4.2. Verbesserungsansatz 1

Abbildung 45 - Versbesserungsansatz durch Mischung von Funktionen des genetischen Algorithmus

Der erste Verbesserungsansatz basiert auf die Verwendung von verschiedenen Funktionen für die Routenplanung. Hier werden alle vordefinierten Funktionen von Selektion, Crossover und Mutation für die Erzeugung der Population verwendet. Die Population von dem genetischen Algorithmus wird mit einer Funktionsmischung erzeugt. Da in dieser wissenschaftlichen Arbeit zu jeder Funktion zwei Arten eingeplant ist, werden schlussendlich acht verschiedenen Routenplanungen durchgeführt und in die Population eingespeist. Für jede Art der Erzeugung werden 20 Child-Tours erzeugt und damit wird die Populationsgröße auf 160 Routenplanungen erreicht. Darauffolgend wird von jeder Art der Erzeugung 2 Elitetouren in die nächste Runde übernommen. Somit ist die Gesamtzahl an Elitetouren 16.

Für diese Vorgehensweise sind drei For-Schleifen in der Abbildung 45 vorhanden. Die erste For-Schleife wählt zwei Selektionsarten. Diese sind zum einen die Roulette-Selektion und zum anderen die Tournament-Selektion. Die zweite For-Schleife bestimmt die Crossover-Art. Dabei wird zwischen dem Ordered-Crossover und dem Universal-Crossover unterschieden. Die letzte For-Schleife setzt die Mutationsart fest. Die zwei wesentlichen Mutationsarten für diese Arbeit sind Swap-Mutation und Invers-Mutation. Die erzeugten Routenplanungen werden in jeder Iteration in die Populationsliste gespeichert. Danach lässt das Programm die Standardvorgehensweise von dem genetischen Algorithmus durchlaufen.

4.4.3. Verbesserungsansatz 2

Die zweite Verbesserung widmet sich auf die Erzeugung der Routenplanungen. Dabei werden die Routenplanungen dazu gezwungen, dass sie ab einem gewissen Schwellwert ein „Deep-Search“ durchführen. Wenn dieser Wert erreicht wird, führt das Programm eine Einschränkung bzgl. Routenplanungen durch. Durch diese Einschränkung werden nur noch Routenplanungen erzeugt, die nicht schlechter als die beste Routenplanungsentfernung mal 1,3 abschneiden. Somit wird in einem bestimmten Umfeld detaillierter nach der besten Routenplanung gesucht.

```
if (z == 1) { saveDeepSearchTime = 1.1 * eliteChromosome.Distance; }
```

Abbildung 46 - Verbesserungsansatz: Schwellwertbestimmung

In der ersten Runde wird der Schwellwert für das Deep-Search bestimmt. Danach wird dieses Deep-Search für die kommenden 99 Runden benutzt. Dieses Vorgehen kann in der Abbildung 46 abgelesen werden. Dabei wird die beste Routenplanungsentfernung genommen und mit 1.1 multipliziert. Man muss beachten, dass die erste Runde von dem genetischen Algorithmus nur für die Bestimmung von dem Deep-Search-Wert ist. In dieser Runde nimmt man eine kleinere Populationsgröße an, damit es schneller durchgeführt wird. Somit wird grob eingeschätzt, in welchem Bereich das Programm die beste Routenplanung suchen soll.

```
nextChange += 1;
deepSearch = false;
if (eliteChromosome.Distance <= saveDeepSearchTime)
{
    if (placeTheRouters == true && nextChange > 50)
    {
        for (int randomadd = 1; randomadd < 17; randomadd++)
        {
            PopulationList.Add(roundAI[random.Next(0, roundAI.Count - 1)]);

        }
        placeTheRouters = true;
        nextChange = 0;
    }
    deepSearch = true;
}
nonEliteGenerations++;
```

Abbildung 47 - Verbesserungsansatz: Deep-Dive und Einspeisung von Routenplanung

In der obigen Abbildung wird die Setzung von der Deep-Search-Variable visualisiert, in dem der Schwellwert erreicht wurde (siehe Abbildung 47 - Verbesserungsansatz: Deep-Dive). Die

Setzung von der Deep-Search-Variable wird mit einer IF-Abfrage überprüft. Wenn die beste Routenplanung in dem genetischen Algorithmus einen besseren Entfernungswert hat als der Schwellwert, dann wird ein Deep-Search durchgeführt. Hier lässt sich noch eine IF-Abfrage sehen. Diese verbessert den Populationsgehalt, indem die Routenplanungen aus der früheren erzeugten Population zufälligerweise eingefügt werden. Jede erzeugte Population wird nach einer Logik abgespeichert (siehe Abbildung 49). Diese Methodik lässt es verwirklichen, dass die Population eine Vielfalt von verschiedenen Routenplanungen bekommt. In diesem Zusammenhang kann durch die Einspeisungen eine Varianz in der Population geschafft werden, was wiederum gegen die lokalen Optima und das Problem der vorzeitigen Konvergenz wirkt. Die Einspeisung findet erst dann statt, wenn 50 Runden lang keine Verbesserungen durch den genetischen Algorithmus bei der Routenplanung sichtbar werden. Dabei werden 16 Routenplanungen in die Population eingespeist. Die nächste Einspeisung findet erst wieder nach 50 Runden statt, wenn keine Verbesserung in der Routenplanung vorhanden ist. Wenn 200 Runden lang keine Verbesserung resultiert, findet insgesamt vier Mal eine Einspeisung statt. Im schlimmsten Fall wird durch die Einspeisungen keine Child-Tour erzeugt, was wiederum keine Wirkung auf den genetischen Algorithmus bei der Routenplanung hat.

```
bool flagForAdd = false;


if (deepSearch == true)
{
    double averageOfFitness = 0;

    averageOfFitness = PopulationList.First().Distance * 1.3;

    if (averageOfFitness < child.Distance)// && deepSearch == true)
    {
        flagForAdd = true;
    }

}

if (flagForAdd == false)
{
    genNewPop.Add(child);
}
//genNewPop = genNewPop.Distinct().ToList();

genNewPop = genNewPop
            .GroupBy(x => x.TourOfChromosome)
            .Select(g => g.First())
            .ToList();


i = genNewPop.Count;
```

Abbildung 48 - Verbesserungsansatz: Deep-Dive Unterprogramm

Die letzte Abbildung für den Deep-Dive illustriert das Unterprogramm, in dem die Erzeugung von Child-Tours eine bessere Routenplanungsentfernung liefern soll als der Schwellwert (Abbildung 48). Nach der Erzeugung der Child-Tour, kommt dieser Code in der Abbildung infrage. Dabei wird zunächst überprüft, ob die Deep-Dive Variable gesetzt wurde. Wenn das der Fall ist, wird ermittelt, ob die erzeugte Routenplanungsentfernung schlechter ist als die beste Routenplanung in der Population multipliziert um den Faktor 1,3. Je nachdem wird entschieden, ob diese Routenplanung in die neue Population eingefügt wird oder nicht. Nachdem werden die doppelten Routenplanungen aus der neuen Populationsliste gelöscht. Infolgedessen wird erreicht, dass die neu erzeugten Populationen keine gleichen Routenplanungen enthalten, und eine höhere Varianz in der Population gebildet wird. Folglich wird die Anzahl an erzeugten Populationen gezählt. Somit wird festgestellt, wie viele Child-Tour noch erzeugt werden sollte. Zusammengefasst gibt es zwei Funktionen, womit sich dieses Unterprogramm beschäftigt. Bei einem wird nach den Duplikationen überprüft. Im zweiten Fall wird die Routenplanung auf einem bestimmten Entfernungsbereich eingeschränkt. Das Ziel von dem Ganzen ist es, den genetischen Algorithmus dazu zu zwingen eine bessere Routenplanung und eine detaillierte Recherche in einem abgekapselten Bereich durchzuführen.

```
if (z == 1) { saveDeepSearchTime = 1.1 * eliteChromosome.Distance; }
if (eliteChromosome.Distance <= saveTheBestRound.Distance | z == 1)
{
    for (int addToListCounter = 0; addToListCounter <= roundSave.Count - 1; addToListCounter++)
    {
        roundAI.Add(roundSave[addToListCounter]);
    }
    roundAI = roundAI
            .GroupBy(i => i.TourOfChromosome)
             .Select(g => g.First())
             .ToList();
    if (z == 1 | roundAI.Count < 100)
    {
        roundAI = roundAI.OrderBy(x => x.Distance).ToList();
    }
    else
    {
        roundAI = roundAI.OrderBy(x => x.Distance).Skip(100).Take(160).ToList();
    }
    saveTheBestRound = eliteChromosome;
}
```

Abbildung 49 - Verbesserungsansatz: Speicherung von der besten Population

Die obige Abbildung präsentiert die Speicherung von den Populationserzeugungen des genetischen Algorithmus (Abbildung 49). Dieser Teil des Programms wird erst ausgeführt, sobald die Abbruchbedingung bei der Populationserzeugung erfüllt ist. In diesem Teil des Codes wird geachtet, dass nur bei einer Verbesserung in die „roundAI“ hinzugefügt wird. Dies wird durch eine IF-Abfrage festgelegt. In dieser IF-Abfrage wird überprüft, ob die gespeicherte

Routenplanungsentfernung größer ist als die aktuelle Routenplanungsentfernung. Dabei werden die besten Routenplanungsentfernungen verglichen. Die Variable „eliteChromosome“ und die Populationsliste „roundSave“ werden während der Erstellung der Routenplanungen gespeichert (siehe Abbildung 44). Nach der Bestätigung, dass der aktuelle „eliteChromosome“ besser ist als die alte, wird die Populationsliste „roundSave“ in die Liste „roundAI“ hinzugefügt. Daraufhin wird die „roundAI“-Liste bereinigt, indem es die doppelt abgespeicherten Routenplanungen entfernt. Damit nicht die besten Routenplanungen in der „roundAI“ gespeichert sind, werden die ersten 100 Routenplanungen in der Liste nicht in Betracht gezogen. Somit wirkt man gegen die lokalen Optima und das Problem der vorzeitigen Konvergenz. Es ist noch bemerkenswert, dass die „roundAI“-Liste für die Einspeisung in die Routenplanung dient (siehe Abbildung 47).

4.5. Bing Maps API Verbindung

In diesem Unterkapitel wird der Verbindungsaufbau und die Abrufe von den einzelnen Entfernungen demonstriert. Der Verbindungsaufbau wird zwischen der Bing Maps API und dem User etabliert. Die Vorgehensweise wurde im Kapitel 3 diskutiert. Die Umsetzung der Theorie in die Programmierumgebung wird in dieser Sektion angestrebt.

```
private string _ApiKey = System.Configuration.ConfigurationManager.AppSettings.Get("BingMapsKey");
```

Abbildung 50 - Bing Maps Key

Für den Aufbau der Verbindung wird ein Bing Maps Key benötigt. Der Bing Maps Key wird durch die Eröffnung eines Accounts auf der Bing Maps Webseite zur Verfügung gestellt. Damit die Angaben von der Informationssicherheit gewährleistet werden, muss der API-Schlüssel in eine Variable abgespeichert werden und darf nicht im Hauptprogramm schriftlich beziehungsweise notiert erscheinen. Somit wurde dieser API-Schlüssel in den App-Settings versteckt und von dort aus abgerufen (siehe Abbildung 50). Dieser Abruf wird in die Variabel „_ApiKey“ gespeichert.

Für die Berechnung der Distanzmatrix gibt es drei Unterfunktionen. Die erste Funktion „HttpRequestForBing“ führt die Kommunikation mit dem Server durch. Weiters liefert die erste Funktion die Antwort von dem Server zu der zweiten Funktion. Die zweite Funktion „GetResponse“ bestimmt, in welchem Abstimmungsbereich beziehungsweise welcher Stufe die Kommunikation der Anfrage ist. Somit regelt die zweite Funktion die Kommunikation und ruft für die jeweilige Ebene die erste Funktion auf. Die dritte Funktion „Address“ erstellt die

Internetadresse mit den jeweiligen Standorten und dem LKW-Parameter. Nachdem die Internetadresse erstellt wurde, ruft die „Address“-Funktion die zweite Funktion auf, damit die Kommunikation mit dem Sever gestartet werden kann. Dieses Vorgehen wird in den nächsten Abbildungen detailliert erklärt.

```
6 Verweise
private string HttpRequestForBing(string url, string typeOfRequest)
{
    var httpRequest = (HttpWebRequest)WebRequest.Create(url);
    httpRequest.Accept = "application/xml";
    var httpResponse = (HttpWebResponse)httpRequest.GetResponse();

    string xmlResult;
    using (var streamReader = new StreamReader(httpResponse.GetResponseStream()))
    {
        var result = streamReader.ReadToEnd();
        xmlResult = result;
    }
    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.LoadXml(xmlResult);
    XmlNodeList parentNode = xmlDoc.GetElementsByTagName(typeOfRequest);
    string backUrl = "";
    backUrl = parentNode[0].InnerText;
    Console.WriteLine(parentNode[0].InnerText);
    return backUrl;
}
```

Abbildung 51 - Abwicklung von Entfernungsabfrage

Die Entfernungen zwischen den einzelnen Positionen werden in der Abbildung 51 berechnet. Dafür baut das Programm mit der Schnittstelle eine Verbindung auf und fragt mittels der erzeugten URL (beziehungsweise Internetadresse) die Entfernung zwischen zwei Positionen. Dabei bekommt diese Funktion zwei Variablen zugewiesen. Eine Variable ist die URL und die zweite Variable ist der Typ der Abfrage. Zu den einzelnen Typen wird in der nächsten Abbildung detailliert geschildert. Nichtdestotrotz dient die Abbildung 51 für die Kommunikation mit dem Server. Zuerst wird eine Anfrage mit der URL generiert. Danach wird mittels „GetResponse“ die Abfrage durchgeführt. Die Antwort von dem Server wird in eine Variable gespeichert. Die Antwort wird in XML-Format verschickt. Diese Antwort wird anhand von XML-Reader gelesen. Je nach Typ der Abfrage wird nach dem XML-Tag gesucht und der Inhalt in eine Variable gespeichert. Zum Schluss schickt der Code die Variable wieder zurück zum abgerufenen Ort.

```
private double GetResponse(Uri url,out double timeOfTour)
{

    string urlString = url.ToString();
    string callBackUrl = HttpRequestForBing(urlString, "CallbackUrl");
    string isTrueOrNot = HttpRequestForBing(callBackUrl, "IsCompleted");
    while (isTrueOrNot == "false")
    {
        System.Threading.Thread.Sleep(1000);
        isTrueOrNot = HttpRequestForBing(callBackUrl, "IsCompleted");
    }
    string resultUrl = HttpRequestForBing(callBackUrl, "ResultUrl");
    double distance = double.Parse(HttpRequestForBing(resultUrl, "TravelDistance"), CultureInfo.InvariantCulture);
    timeOfTour=double.Parse(HttpRequestForBing(resultUrl, "TravelDuration"), CultureInfo.InvariantCulture);
    return distance;
}
```

Abbildung 52 - Regelwerk für die Kommunikation mit dem API

In der Abbildung 52 wird das Regelwerk für die Kommunikation mit der Schnittstelle von Bing Maps präsentiert. Dabei gibt es drei Stufen. Die erste Stufe „CallbackUrl“ ist der Aufbau und die Anfrage von der Entfernung zwischen zwei Standorten. Da es sich um eine asymmetrische Verbindung handelt, wie auch in dem Kapitel 3.2 beschrieben, wartet das Programm auf eine Antwort auf die Anfrage. Die Nachfrage auf die Antwort wird durch eine While-Schleife durchgeführt. In dieser While-Schleife wird nach der Antwort mit „IsCompleted“ so lange abgefragt, bis die Antwort abgefangen werden kann. In der Antwort gibt es einen Link, mit dem in die dritte Stufe der Anfrage übergegangen werden kann. In der letzten Stufe wird mit dem Antwortlink die Entfernung abgespeichert. Es wird nicht nur die Entfernung gespeichert, sondern auch die Fahrzeit.

```
1 Verweis
public double Adress(string wayPoint0, string wayPoint1, string optimize, string vehicleHeight, string vehicleWidth,
    string vehicleLength, string vehicleWeight, string vehicleAxles, string vehicleMaxGradient, string vehicleMinTurnRadius, string vehicleHazardousMaterials,out double timeOfTour)
{
   //Uri geocodeRequest = new Uri(string.Format("http://dev.virtualearth.net/REST/v1/Locations?q={0}&key={1}", query, _ApiKey));
   Uri geocodeRequest = new Uri(string.Format("https://dev.virtualearth.net/REST/v1/Routes/TruckAsync?" +
       "wp.0={0}&" +
       "wp.1={1}&" +
       "optmz={2}&" +
       "height={3}&" +
       "width={4}&" +
       "vl={5}&" +
       "weight={6}&" +
       "axles={7}&" +
       "vmg={8}&" +
       "vmtr={9}&" +
       "vhm={10}&" +
        "du=km&key={11}&output=xml", wayPoint0, wayPoint1, optimize, vehicleHeight, vehicleWidth, vehicleLength, vehicleWeight,
        vehicleAxles, vehicleMaxGradient, vehicleMinTurnRadius, vehicleHazardousMaterials, _ApiKey));
   timeOfTour = 0;
   double distance= GetResponse(geocodeRequest,out timeOfTour);
   return distance;
}
```

Abbildung 53 - Erstellung von der URL

Die Abbildung 53 demonstriert die Erstellung der Internetadresse. Dabei werden die Parameter von dem LKW auch angegeben. Nach dem die festgelegten Parameter, Standorte, und der API-Schlüssel verkettet werden, ruft die Funktion „Address“ das Kommunikationsglied „GetResponse“ auf. Auf dieser Weise wird die Kette von Funktionen geschlossen und die Ermittlung von der Entfernung und Fahrzeit durchgeführt.

```
public double[,] DisatanceMatrixForBingApi(List<Point> points, ref double[,] timeMatrix)
{
    double[,] distanceMatrix = new double[points.Count, points.Count];
    double timeOfTour = 0;
    string optimize = "time";
    string vehicleHeight = "3.40";
    string vehicleWidth = "2.55";
    string vehicleLength = "8.5";
    string vehicleWeight = "32000";
    string vehicleAxles = "4";
    string vehicleMaxGradient = "15";
    string vehicleMinTurnRadius = "22";
    string vehicleHazardousMaterials = "None";
    var add = new ApiConnection();
    for (int i = 0; i < points.Count; i++)
    {
        for (int j = 0; j < points.Count; j++)
        {
            string wayPoint0 = "" + points[i].XCoordinate + "," + points[i].YCoordinate;
            string wayPoint1 = "" + points[j].XCoordinate + "," + points[j].YCoordinate;
            distanceMatrix[i, j] = add.Address(wayPoint0, wayPoint1, optimize, vehicleHeight, vehicleWidth,
                vehicleLength, vehicleWeight, vehicleAxles, vehicleMaxGradient, vehicleMinTurnRadius, vehicleHazardousMaterials, out timeOfTour);
            timeMatrix[i, j] = timeOfTour;
        }
    }
    return distanceMatrix;

}
```

Abbildung 54- Erstellung von Distanzmatrix und Fahrzeitmatrix

Die oben abgebildete Abbildung stellt die Erzeugung der Distanz- und Fahrzeitmatrix dar (siehe Abbildung 54). Zudem werden zuerst die LKW-Daten deklariert. Die Höhe des LKWs beträgt 3,4 Meter und die Breite 2,55 Meter. Die Länge des LKW macht 8,50 Meter aus. Das Gesamtgewicht beträgt 32 Tonnen und die Nutzlast davon ist 16,3 Tonnen. Der LKW hat vier Achsen. Es kann bis zu 15 Prozent Steigung hinter sich legen und hat einen Wendekreis von 22 Grad. Nachdem die Variablen definiert wurden, kann die Erzeugung von einer Distanzmatrix zuzüglich eine Fahrzeitmatrix beginnen. Diesbezüglich verwendet das Programm zwei For-Schleifen. Diese zwei For-Schleifen definieren die Position beziehungsweise den Standort, wobei die Koordinaten in zwei verschiedene Variablen gespeichert werden. Folglich wird die „Address"- Funktion abgerufen, die in der Abbildung 53 abgebildet ist. Die Rückgabewerte, die Entfernung und die Fahrtzeit werden in zwei Arrays abgespeichert. Diese Vorgehensweise wird so lange durchgeführt, bis keine Standorte mehr übrig bleiben.

Es ist wichtig zu erwähnen, dass die echten LKW-Daten von der MA48 nicht geforscht werden konnte. Deshalb wurde ein ähnlicher LKW-Typ genommen. Da die LKWs verschiedene Spezifikationen haben, variieren die technischen Daten der LKWs ziemlich stark voneinander. Die technischen Merkmale des LKWs wurden der Firma Hasse Transport GmbH (4-Achs-Kipper) angenommen (Hasse Transport GmbH, o.D.). Dazu passend wurde noch ein Kran der Firma HMF-Ladekran des Modells 1910-K3 ausgesucht (HMF Ladekrane & Hydraulik GmbH, o.D.).

```
public void WriteToFile(double[,] distanceMatrix, List<Point> pointList)
{
    //save at folder C:/..../hw5/bin/debug
    string sCurrentDirectory = AppDomain.CurrentDomain.BaseDirectory;
    string sFile = Path.Combine(sCurrentDirectory, @"distanzmatirx.csv");
    string file = Path.GetFullPath(sFile);

    using (var stream = File.CreateText(file))
    {
        for (int n = 0; n < PointList.Count; n++)
        {
            for (int j = 0; j < PointList.Count; j++)
            {
                stream.Write(distanceMatrix[n, j] + "-");

            }
            stream.WriteLine();
        }

    }


}

0 Verweise
public void WriteToFile2(double[,] timeMatrix, List<Point> pointList)
{
    //save at folder C:/..../hw5/bin/debug
    string sCurrentDirectory = AppDomain.CurrentDomain.BaseDirectory;
    string sFile = Path.Combine(sCurrentDirectory, @"timematirx.csv");
    string file = Path.GetFullPath(sFile);

    using (var stream = File.CreateText(file))
    {
        for (int n = 0; n < PointList.Count; n++)
        {
            for (int j = 0; j < PointList.Count; j++)
            {
                stream.Write(timeMatrix[n, j] + "-");

            }
            stream.WriteLine();
        }

    }


}
```

Abbildung 55 - Erzeugung einer CSV-Datei

Die letzte Abbildung in diesem Abschnitt zeigt die Erzeugung einer CSV-Datei (siehe Abbildung 55). Das ist deshalb essenziell, weil die Erzeugung der Distanz- und Zeitmatrix abgekapselt ermittelt wurde. Die Probleme wurden im Kapitel 3.3 ausführlich erklärt. Zusammengefasst mussten die ermittelten Entfernungen und Zeiten, bedingt von maximaler Anzahl an Anfragen an die API, gespeichert werden. Infolgedessen wurden zwei Funktionen in der Abbildung 55 erzeugt. Die erste Funktion erzeugt eine CSV-Datei für die Distanzmatrix und die zweite Funktion erzeugt eine CSV-Datei für die Zeitmatrix. Diese zwei Matrizen werden in der Routenplanung verwendet. Die Erzeugung von der CSV-Datei funktioniert mit zwei For-Schleifen. Die erste For-Schleife gibt die Zeile der Distanzmatrix an. In der zweiten

For-Schleife werden die einzelnen Spalten abgehandelt. Somit wird mit der Kombination Zeile und Spalte die Entfernung von zwei Standorten in die CSV-Datei abgespeichert.

4.6. HTML und JavaScript-Datei für die Visualisierung der Routenplanung

Der vorletzte Abschnitt des Kapitels „Programm“ präsentiert den Code für die Visualisierung der errechneten Routenplanungen. In diesem Zusammenhang werden zwei Dateien erstellt. Die HTML-Datei dient zur Visualisierung der Routenplanungen, wobei die JavaScript-Datei für die Kommunikation zwischen der API und des HTML-File dient.

```
const string fileName = "imagetTheRoute2.js";
for (int j = 0; j < listOfTours.Count; j++)
{
    var content = File.ReadAllText(fileName);
    List<double> cordinates = new List<double>();
    for (int i = 0; i < listOfTours[j].ListofTours.Count; i++)
    {
        //define Loaction
        string addtoWP01 = Convert.ToString(PointList[listOfTours[j].ListofTours[i] - 1].XCoordinate);
        addtoWP01= addtoWP01.Replace(",", ".");
        string addtoWP02 = Convert.ToString(PointList[listOfTours[j].ListofTours[i] - 1].YCoordinate);
        addtoWP02 = addtoWP02.Replace(",", ".");
        string addAtLast = "var wp" + (i+1) + " = new Microsoft.Maps.Directions.Waypoint({\r\n        location: new Microsoft.Maps.Location("+ addtoWP01 + ","+ addtoWP02 + ")\r\n    });\r\n";
        content = content.Insert(content.LastIndexOf(""), addAtLast);
    }
    for (int i = 0; i < listOfTours[j].ListofTours.Count; i++)
    {
        //set defined Locations
        string addAtLast = "    directionsManager.addWaypoint(wp"+(i+1)+");\r\n";
        content = content.Insert(content.LastIndexOf(""), addAtLast);
    }
    string addAtLast01 = "    directionsManager.calculateDirections();\r\n});\r\n}";
    content = content.Insert(content.LastIndexOf(""), addAtLast01);

    File.WriteAllText("js/"+nameOfFile + j + ".js", content);
}
```

Abbildung 56 - Überschreibung der JavaScript-File mit den Routenplanungsdaten

Die Abbildung 56 illustriert die Erstellung der JavaScript-Datei für die Routenplanungen. Je nach Volumen der Routenplanungen werden genauso viele JavaScript-Dateien erstellt. Das heißt, wenn eine Routenplanung neun Touren beinhaltet, werden neun JavaScript-Dateien erstellt. Ein Template wird für die JavaScript-Datei verwendet, in der die einzelnen Touren abgespeichert werden. Für jede Tour wird der Dateiname angepasst. Die erste For-Schleife gibt die Anzahl an Touren an. Die nächsten zwei For-Schleifen definieren die Standorte, welche besucht werden sollten und tragen sie in die JavaScript-Datei ein. Die zweite For-Schleife dient zur Definition der JavaScript-Variable. In der dritten For-Schleife werden die definierten Variablen gesetzt. Demzufolge werden auch die Routenplanungen in der HTML-File angezeigt.

```
public void OverwriteTheHtmlfile()
{
    string dir = "js";
    List<string> nameOfFile = new DirectoryInfo(dir).GetFiles().Select(o => o.Name).ToList();
    const string fileName = "imagetohtml.html";


    for (int i = 0; i < nameOfFile.Count; i++)
    {
        var content = File.ReadAllText(fileName);
        if (nameOfFile[i] == "imagetohtml.html")
        {

        }
        else
        {
            //Replace all values in the HTML
            content = content.Replace("{" + 0 + "}", Convert.ToString(nameOfFile[i]));
            string dummyreplacer = nameOfFile[i].Replace(".js", "");
            //Write new HTML string to file
            File.WriteAllText("js/"+dummyreplacer + ".html", content);
        }
    }


}
```

Abbildung 57 - Erstellung der HTML-Datei

Die letzte Abbildung in diesem Abschnitt zeigt die Erstellung der HTML-Datei (siehe Abbildung 57). Wie auch in der JavaScript-Datei wird auch hier ein vorgefertigtes Template verwendet, in dem nur die JavaScript-Datei eingebettet wird. Durch die For-Schleife werden die erstellen JavaScript-Dateien in dem Ordner iteriert. Anhand der JavaScript-Dateien werden die Dateinamen der HTML-Dateien definiert. In weiterer Folge wird auch bei jeder Iteration die jeweilige JavaScript-Datei in die HTML-Datei eingebettet.

Zusammengefasst werden bei jeder Iteration die HTML-Dateinamen und die Einbettung der JavaScript-Datei in die HTML-Datei erfolgreich durchgeführt. Die Anzahl der JavaScript-Dateien und HTML-Dateien entspricht genau die Anzahl an Touren in einer Routenplanung. Wenn die HTML-Datei geöffnet wird, läuft im Hintergrund der JavaScript-Code. Dieser JavaScript-Code führt eine Anfrage an die Bing Maps Schnittelle durch, indem der Sever die angeführten Standorte in eine Abbildung wieder zurückschickt. Zudem wird noch eine Wegbeschreibung für die angeführten Standorte dokumentiert. Die Reihenfolge von den Standorten wird eingehalten und in der Visualisierung demzufolge dargestellt.

4.7. VBA-Code für die Auswertung der HTML-Dateien

Der letzte Abschnitt befasst sich mit der Auswertung der einzelnen Tourenplanungen. Da es insgesamt 1536 Touren für die verschiedene Versionen gibt, wurde für die schnelle Auswertung ein VBA-Code zusammengefasst. Der VBA-Code wird in Excel geschrieben. In diesem VBA-Code werden die einzelnen Tourenplanungen geöffnet. Folgend werden die Entfernungen und die Fahrtzeiten rausgelesen. Danach wird ein Screenshot von der Seite aufgenommen, um die Routenplanung zu dokumentieren.

```
Sub tagofwebsite()
Dim selenium As New selenium.WebDriver
Dim arrayOfAllFiles As Variant

    For m = 1 To 4
        Dim arrayOfAllNoddes As Variant
        arrayOfAllNoddes = Array("150", "125", "100", "75", "50", "25")
        For Each Item In arrayOfAllNoddes
            versionOfPrint = "v" & m
            screenShotOf = "ScreenShotsOf" & versionOfPrint & "_" & Item
            calculationOf = "CalculationOf" & versionOfPrint & "_" & Item
            Sheets.Add.Name = screenShotOf
            Sheets.Add.Name = calculationOf

            'k defines the rows in which the values or screenshot will be placed
            k = 0
            arrayOfAllFiles = Array("nearestInsertionBing", "nearestInsertionTimeBing", _
            "nearestInsertionDistanceEuc", "cheapestInsertionTour", "cheapestInsertionTourTime", _
            "cheapestInsertionTourEuc", "savingAlgoTourList", "savingAlgoTourListTime", _
            "savingAlgoTourListEuc", "geneticAlgoBing", "geneticAlgotime", "geneticAlgoEuc")
            'iteration of Algorithms
            For Each Files In arrayOfAllFiles
                k = k + 1
                'iteration over Tours
                If Item = "150" Then
                    iterateTo = 8
                ElseIf Item = "125" Then
                    iterateTo = 6
                ElseIf Item = "100" Then
                    iterateTo = 5
                ElseIf Item = "75" Then
                    iterateTo = 4
                ElseIf Item = "50" Then
                    iterateTo = 2
                ElseIf Item = "25" Then
                    iterateTo = 1
                End If
```

Abbildung 58 - VBA Code Teil 1

In der Abbildung 58 wird der erste Teil von dem VBA-Code präsentiert. In diesem Teil werden einerseits die Variablen deklariert und anderseits gibt es drei verkettete FOR-Schleifen. Die erste FOR-Schleife definiert die Version der Auswertung. Als Nächstes kommt die zweite FOR-Schleife, in dem die Anzahl der Standorte deklariert wird. Die dritte FOR-Schleife wählt den Algorithmus für die Auswertung aus.

```
i = 0 To iterateTo
openTheFile = Files & i & ".html"
selenium.Start "chrome"
selenium.Get "C:\Users\Ahmet Burak Turgut\Documents\laptop\master\Masterarbeit\RoutingWithBing " & versionOfPrint & "\" & Item & "\" & openTheFile
VBA.SendKeys "{F11}"
Application.Wait Now + TimeValue("00:00:06")
selenium.TakeScreenshot().SaveAs ("C:\Users\Ahmet Burak Turgut\Documents\laptop\master\Masterarbeit\bilder\" & Files & "_" & Item & "_" & versionOfPrint & ".png")
' Find an element by CSS selector with data attribute
Dim element As selenium.WebElement
Set element = selenium.FindElementByClass("drTitleRight")
Dim dataTagValue As String
dataTagValue = Replace(element.Text, " km", "")
Set element = selenium.FindElementByClass("drHours")
Dim dataTagTimeHours As String
dataTagTimeHours = element.Text
Set element = selenium.FindElementByClass("drMins")
Dim dataTagTimeMins As String
dataTagTimeMins = element.Text

' Write HTML text to a sheet (assuming Sheet1, adjust as needed)
Dim ws As Worksheet
Set ws = ThisWorkbook.Sheets(calculationOf)
columnsToplaceThevalues = i + 1
lastRow = ws.Cells(ws.Rows.Count, columnsToplaceThevalues).End(xlUp).Row
If lastRow = 1 Then
Else
    lastRow = lastRow + 1
End If
defineThecolumnToPutThevalues1 = Replace(Cells(lastRow, columnsToplaceThevalues).Address, "$", "")
defineThecolumnToPutThevalues2 = Replace(Cells(lastRow + 1, columnsToplaceThevalues).Address, "$", "")

' Write HTML text to the sheet
ws.Range(defineThecolumnToPutThevalues1).Value = dataTagValue * 1
If dataTagTimeHours = "" Then
    ws.Range(defineThecolumnToPutThevalues2).Value = dataTagTimeMins * 1
Else
    ws.Range(defineThecolumnToPutThevalues2).Value = dataTagTimeHours * 60 + dataTagTimeMins * 1
End If


' Close the browser window
selenium.Quit
```

Abbildung 59 - VBA Code Teil 2

Nachdem die Auswertungsart bestimmt wurde, führt das Programm den Code in der Abbildung 59 aus. Zunächst wird die HTML-Datei geöffnet, die durch die Tourenplanung erzeugt wurde. Im Anschluss wird ein Screenshot von dem Bildschirm aufgezeichnet und wird in einem Ordner als PNG-Datei abgespeichert. Danach liest das Programm die Fahrzeit und die Entfernung der HTML-Seite aus. Diese Daten werden anschließend in einem vordefinierten Excelarbeitsblatt abgespeichert. Zum Schluss wird die geöffnete HTML-Datei geschlossen. Diese Vorgehensweise wird, wie bereits oben angeführt,1536-mal durchgeführt.

5. Ergebnisse & Diskussion

Diese Kapitel befasst sich mit den Ergebnissen der einzelnen Algorithmen und deren Verbesserungsansätzen. Wie schon im vorherigen Kapitel präsentiert, gibt es zwei Verbesserungsansätze des genetischen Algorithmus. In diesem Zusammenhang wurden vier Versionen für die Auswertung entwickelt. In diesen Versionen ändert sich ausschließlich des genetischen Algorithmus. Die restlichen Algorithmen werden in der Programmierung beibehalten. Die erste Version gibt die Auswertung der herkömmlichen Version (siehe Kapitel 4.4.1) wieder. Als nächstes wird der Verbesserungsansatz 1 (siehe Kapitel 4.4.2) mit der Versionsnummer zwei präsentiert. Die dritte Version wird mit dem Verbesserungsansatz 2 (siehe Kapitel 4.4.3) ausgewertet. Zum Schluss wird die Auswertung mit beiden Verbesserungsansätzen dargestellt (Version 4). Das heißt, dass beide Verbesserungsansätze im genetischen Algorithmus eingesetzt werden.

Für die Auswertung wurde das Makro, das in dem Kapitel 4.7 erläutert wurde, verwendet. Resümierend öffnet dieses Makro die einzelnen HTML-Dateien und liest die Zeiteinheit sowie die Entfernungseinheit der einzelnen Touren. Zuletzt werden diese rausgelesenen Daten aufsummiert und in den unten präsentierten Abbildungen notiert. Da die Aufnahmen nicht zeitgleich stattfinden, kann in der Zeiteinheit der einzelnen Versionen verschiedene Zeitangaben erfolgen. Die Begründung ist, dass die Routenplanungen in Echtzeit bewertet werden. Die Verkehrslage hat somit eine Einwirkung auf die Auswertung. Deshalb wurde für einzelne Versionen nochmals mit allen Algorithmen eine Auswertung durchgeführt. Somit wurde die Zeitspanne zwischen den einzelnen Auswertungen verkürzt und in Hinblick auf die zeitliche Verzerrung durch die Verkehrslage vermindert. Das heißt, dass das Programm als ersteres die Auswertung durch die Anzahl an Standorten (z.B. 150 Standorte) mit allen Algorithmen durchführt. Dieses Vorgehen wird so lange wiederholt, bis die Anzahl an Standorten vollständig ausgewertet werden. Danach wurde die Auswertung für die nächsten Versionen durchgeführt.

Die Struktur der unten angeführten Abbildungen wurden einheitlich dargestellt. Die erste Zeile zeigt die Anzahl der Standorte und bildet ein Cluster. Dieses Cluster hat jeweils vier Auswertungen. Die erste Spalte von dem Cluster zeigt die Entfernung des jeweiligen Algorithmus an. Das zweite Cluster bildet die Fahrtzeit ab. Als nächstes werden die Abweichungen bzw. Verbesserungen der beiden Parameter (Entfernung, Fahrtzeit) vom Nearest Insertion Algorithmus nach Entfernung berechnet. In einer Spalte wird die Abweichung

nach Entfernung und in der anderen Spalte wird die Abweichung nach Fahrzeit in Prozent dargestellt. In der ersten Spalte ab der dritten Zeile sind die einzelnen Algorithmen notiert. Die grünlich verfärbten Zellen stellen die beste Routenplanung dar nach dem jeweiligen Parameter dar.

Anzahl an Standorte	150				125				100				75				50				25			
Verwendete Algorithmusarten	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*
Nearest insertion nach Entfernung*	482,4	1207	0%	0%	408,9	1043	0%	0%	365	925	0%	0%	234,1	626	0%	0%	136,8	387	0%	0%	102,4	269	0%	0%
Nearest insertion nach Fahrzeit	489,5	1169	-1%	3%	412,7	1004	-1%	4%	343,7	809	6%	13%	239,7	604	-2%	4%	143	400	-5%	-3%	105,6	273	-3%	-1%
Nearest insertion nach euklidischer Abstand	528,1	1263	-9%	-5%	441,3	1066	-8%	-2%	372,8	930	-2%	-1%	259,4	649	-11%	-4%	140,8	403	-3%	-4%	108,8	268	-6%	0%
Cheapest Insertion nach Entfernung	482,8	1244	0%	-3%	411,2	1077	-1%	-3%	325,8	900	11%	3%	232,8	651	1%	-4%	139,4	399	-2%	-3%	108,5	288	-6%	-7%
Cheapest Insertion nach Fahrzeit	523,5	1226	-9%	-2%	410,4	997	0%	4%	343,4	890	6%	4%	238,5	629	-2%	0%	148,9	400	-9%	-3%	104,9	274	-2%	-2%
Cheapest Insertion nach euklidischer Abstand	544	1304	-13%	-8%	441,5	1138	-8%	-9%	371,6	947	-2%	-2%	248,2	648	-6%	-4%	148,1	416	-8%	-7%	105,5	270	-3%	0%
Saving Algo nach Entfernung	438	1144	9%	5%	372,4	971	9%	7%	279,6	779	23%	16%	200,1	562	15%	10%	143,4	401	-5%	-4%	82,7	220	19%	18%
Saving Algo nach Fahrzeit	424	1080	12%	11%	368,4	955	10%	8%	289,9	775	21%	16%	197,5	550	16%	12%	135,7	396	1%	-2%	86,8	218	15%	19%
Saving Algo nach euklidischer Abstand	487,6	1186	-1%	2%	411,5	1024	-1%	2%	326,1	859	11%	7%	225,3	582	4%	7%	139	401	-2%	-4%	85,2	239	17%	11%
Gentic Algo nach Entfernung v1	497	1357	-3%	-12%	390,3	1082	5%	-4%	303,2	814	17%	12%	191	548	18%	12%	120,5	356	12%	8%	80,4	222	21%	17%
Gentic Algo nach Fahrzeit v1	519,4	1262	-8%	-5%	418,4	1061	-2%	-2%	310,6	840	15%	9%	194,4	528	17%	16%	123,2	350	10%	10%	85,6	205	16%	24%
Genetic Algo nach euklidischer Abstand v1	476,5	1223	1%	-1%	385,2	1011	6%	3%	293,3	816	20%	12%	205,7	541	12%	14%	126,6	366	7%	5%	84,1	226	18%	16%

Abbildung 60 - Ergebnisse der Routenplanungen mit dem herkömmlichen genetischen Algorithmus

In der Abbildung 60 wird der genetische Algorithmus mit herkömmlicher Variante präsentiert. In dieser Auswertung wird ersichtlich, dass für komplexere Routenplanungen von 100 Standorten beginnend der Saving Algorithmus besser ausschlägt als die anderen Algorithmen. Dabei fällt die Verbesserung im Gegensatz zu Nearest Insertion zwischen 8 bis 23 Prozent aus. Im Gegensatz dafür schlägt der genetische Algorithmus mit herkömmlicher Variante bis zu 75 Standorten besser aus. Das Verbesserungspotenzial dabei liegt je nach Parameter zwischen 10 bis 24 Prozent. Erstaunlich ist es, dass die Routenplanungen mit der euklidischen Distanz generell schlechter ausfallen als die Routenplanungen mit Bing Maps. Ein weiterer interessanter Aspekt ist, dass die Berechnung nach Fahrzeit für den Saving Algorithmus mit 125 und 150 Standorte eine bessere Entfernung liefert als die Routenplanung nach der Entfernungsparameter.

Anzahl an Standorte	150				125				100				75				50				25			
Verwendete Algorithmusarten	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*
Nearest insertion nach Entfernung*	482,4	1189	0%	0%	408,9	1024	0%	0%	365	924	0%	0%	234,1	638	0%	0%	136,8	399	0%	0%	102,4	279	0%	0%
Nearest insertion nach Fahrzeit	489,5	1178	-1%	1%	412,7	1015	-1%	1%	343,7	816	6%	12%	239,7	614	-2%	4%	143	406	-5%	-2%	105,6	284	-3%	-2%
Nearest insertion nach euklidischer Abstand	528,1	1250	-9%	-5%	441,3	1061	-8%	-4%	372,8	942	-2%	-2%	259,4	667	-11%	-5%	140,8	413	-3%	-4%	108,8	276	-6%	1%
Cheapest Insertion nach Entfernung	482,8	1239	0%	-4%	411,2	1076	-1%	-5%	325,8	903	11%	2%	232,8	659	1%	-3%	139,4	410	-2%	-3%	108,5	288	-6%	-3%
Cheapest Insertion nach Fahrzeit	523,5	1222	-9%	-3%	410,4	998	0%	3%	343,4	898	6%	3%	238,5	642	-2%	-1%	148,9	414	-9%	-4%	104,9	273	-2%	2%
Cheapest Insertion nach euklidischer Abstand	544	1312	-13%	-10%	441,5	1143	-8%	-12%	371,6	972	-2%	-5%	248,2	670	-6%	-5%	148,1	427	-8%	-7%	105,5	275	-3%	1%
Saving Algo nach Entfernung	438	1146	9%	4%	372,4	973	9%	5%	279,6	793	23%	14%	200,1	574	15%	10%	143,4	411	-5%	-3%	82,7	231	19%	17%
Saving Algo nach Fahrzeit	424	1066	12%	10%	368,4	950	10%	7%	289,9	775	21%	16%	197,5	565	16%	11%	135,7	406	1%	-2%	86,8	241	15%	14%
Saving Algo nach euklidischer Abstand	487,6	1190	-1%	0%	411,5	1034	-1%	-1%	326,1	868	11%	6%	225,3	600	4%	6%	139	415	-2%	-4%	85,2	247	17%	11%
Gentic Algo nach Entfernung v2	514,7	1306	-7%	-10%	373,9	1032	9%	-1%	294,4	838	19%	9%	187,2	548	20%	14%	120,3	377	12%	6%	80,4	226	21%	19%
Gentic Algo nach Fahrzeit v2	511,2	1240	-6%	-4%	410,7	1037	0%	-1%	310,7	824	15%	11%	187,8	538	20%	16%	122,5	353	10%	12%	85,6	223	16%	20%
Genetic Algo nach euklidischer Abstand v2	466,1	1221	3%	-3%	369	993	10%	3%	304,3	842	17%	9%	204,5	554	13%	13%	128,2	389	6%	3%	84,6	233	17%	16%

Abbildung 61 - Ergebnisse der Routenplanungen mit dem Verbesserungsansatz 1 des genetischen Algorithmus

Die Abbildung 61 präsentiert den genetischen Algorithmus mit dem Verbesserungsansatz 1. In dieser Abbildung sind die Muster der besten Routenplanungen dieselben wie bei der Abbildung 60. Bei den Routenplanungen zwischen 100 und 150 Standorten schneidet der Saving Algorithmus besser ab. Ähnlich wie in der Abbildung 60 bildet der genetische Algorithmus zwischen 25 und 75 ein besseres Bild dar. Durch den Verbesserungsansatz 1 werden die Routenplanungen des genetischen Algorithmus minimal besser als bei der herkömmlichen Version des genetischen Algorithmus.

Anzahl an Standorte	150				125				100				75				50				25			
Verwendete Algorithmusarten	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*
Nearest insertion nach Entfernung*	482,4	1197	0%	0%	408,9	1024	0%	0%	365	931	0%	0%	234,1	614	0%	0%	136,8	386	0%	0%	102,4	273	0%	0%
Nearest insertion nach Fahrzeit	489,5	1176	-1%	2%	412,7	1008	-1%	2%	343,7	814	6%	13%	239,7	600	-2%	2%	143	391	-5%	-1%	105,6	276	-3%	-1%
Nearest insertion nach euklidischer Abstand	528,1	1264	-9%	-6%	441,3	1062	-8%	-4%	372,8	934	-2%	0%	259,4	641	-11%	-4%	140,8	398	-3%	-3%	108,8	275	-6%	-1%
Cheapest Insertion nach Entfernung	482,8	1249	0%	-4%	411,2	1071	-1%	-5%	325,8	889	11%	5%	232,8	633	1%	-3%	139,4	389	-2%	-1%	108,5	287	-6%	-5%
Cheapest Insertion nach Fahrzeit	523,5	1230	-9%	-3%	410,4	997	0%	3%	343,4	895	6%	4%	238,5	620	-2%	-1%	148,9	389	-9%	-1%	104,9	263	-2%	4%
Cheapest Insertion nach euklidischer Abstand	544	1313	-13%	-10%	441,5	1144	-8%	-12%	371,6	950	-2%	-2%	248,2	636	-6%	-4%	148,1	406	-8%	-5%	105,5	266	-3%	3%
Saving Algo nach Entfernung	438	1144	9%	4%	372,4	970	9%	5%	279,6	772	23%	17%	200,1	553	15%	10%	143,4	396	-5%	-3%	82,7	216	19%	21%
Saving Algo nach Fahrzeit	424	1071	12%	11%	368,4	957	10%	7%	289,9	760	21%	18%	197,5	543	16%	12%	135,7	385	1%	0%	86,8	223	15%	18%
Saving Algo nach euklidischer Abstand	487,6	1189	-1%	1%	411,5	1020	-1%	0%	326,1	854	11%	8%	225,3	580	4%	6%	139	396	-2%	-3%	85,2	234	17%	14%
Gentic Algo nach Entfernung v3	457,8	1262	5%	-5%	363,3	1021	11%	0%	268,4	756	26%	19%	192,5	536	18%	13%	119,1	335	13%	13%	80,4	214	21%	22%
Gentic Algo nach Fahrzeit v3	471,7	1181	2%	1%	386,5	961	5%	6%	290,2	766	20%	18%	192,4	524	18%	15%	124,6	345	9%	11%	85,6	205	16%	25%
Genetic Algo nach euklidischer Abstand v3	429,4	1130	11%	6%	375,7	960	8%	6%	298,5	785	18%	16%	196,8	519	16%	15%	127,5	370	7%	4%	84,1	221	18%	19%

Abbildung 62 - Ergebnisse der Routenplanungen mit dem Verbesserungsansatz 2 des genetischen Algorithmus

In der Abbildung 62 wird der Verbesserungsansatz 2 präsentiert. Allgemein schneidet der genetische Algorithmus in der Routenplanung besser ab. Diese drei Ausreißer zeigen, dass die Tendenz auf dem Saving Algorithmus liegt. Für die Routenplanung mit einer Standortzahl 150 ist der Saving Algorithmus nach Fahrzeit besser als die anderen Algorithmen. Sowohl in Entfernung als auch in Fahrzeit ist dieser die bessere Auswahl. Wenn die Standortzahl auf 125 reduziert wird, präsentiert die Auswertung des Saving Algorithmus nach Fahrzeit eine bessere Option in der Kategorie Fahrzeit mit 957 min. Bei den restlichen Ergebnissen ist entweder der genetische Algorithmus nach Entfernung oder nach Fahrzeit die bessere Auswahlmöglichkeit. Ein Gegenüberstellen der Abbildung 61 und Abbildung 62 zeigt eine eindeutige Verbesserung des genetischen Algorithmus mit der Versionsnummer drei. Somit hat der Verbesserungsansatz zwei eine bessere Auswirkung auf den genetischen Algorithmus. Der Einfluss von den beiden Verbesserungsansätzen, die für den genetischen Algorithmus eingesetzt werden, ist in der nächsten Darstellung abgebildet.

Anzahl an Standorte	150				125				100				75				50				25			
Verwendete Algorithmusarten	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*	Entfernung	Zeit	Δ Entfernung*	Δ Zeit*
Nearest insertion nach Entfernung*	482,4	1176	0%	0%	408,9	1020	0%	0%	365	889	0%	0%	234,1	590	0%	0%	136,8	371	0%	0%	102,4	261	0%	0%
Nearest insertion nach Fahrzeit	489,5	1167	-1%	1%	412,7	986	-1%	3%	343,7	785	6%	12%	239,7	572	-2%	3%	143	379	-5%	-2%	105,6	259	-3%	1%
Nearest insertion nach euklidischer Abstand	528,1	1249	-9%	-6%	441,3	1048	-8%	-3%	372,8	900	-2%	-1%	259,4	612	-11%	-4%	140,8	384	-3%	-4%	108,8	258	-6%	1%
Cheapest Insertion nach Entfernung	482,8	1224	0%	-4%	411,2	1053	-1%	-3%	325,8	854	11%	4%	232,8	608	1%	-3%	139,4	377	-2%	-2%	108,5	270	-6%	-3%
Cheapest Insertion nach Fahrzeit	523,5	1221	-9%	-4%	410,4	974	0%	5%	343,4	862	6%	3%	238,5	601	-2%	-2%	148,9	383	-9%	-3%	104,9	253	-2%	3%
Cheapest Insertion nach euklidischer Abstand	544	1285	-13%	-9%	441,5	1107	-8%	-9%	371,6	906	-2%	-2%	248,2	615	-6%	-4%	148,1	396	-8%	-7%	105,5	253	-3%	3%
Saving Algo nach Entfernung	438	1128	9%	4%	372,4	947	9%	7%	279,6	741	23%	17%	200,1	535	15%	9%	143,4	383	-5%	-3%	82,7	210	19%	20%
Saving Algo nach Fahrzeit	424	1059	12%	10%	368,4	925	10%	9%	289,9	736	21%	17%	197,5	518	16%	12%	135,7	378	1%	-2%	86,8	210	15%	20%
Saving Algo nach euklidischer Abstand	487,6	1175	-1%	0%	411,5	990	-1%	3%	326,1	814	11%	8%	225,3	554	4%	6%	139	388	-2%	-5%	85,2	224	17%	14%
Gentic Algo nach Entfernung v4	402,3	1090	17%	7%	332,7	884	19%	13%	269,5	719	26%	19%	180,9	486	23%	18%	117,1	332	14%	11%	80,4	206	21%	21%
Gentic Algo nach Fahrzeit v4	441	1070	9%	9%	354,8	875	13%	14%	275,3	695	25%	22%	189,8	497	19%	16%	118,2	311	14%	16%	85,6	194	16%	26%
Genetic Algo nach euklidischer Abstand v4	426,4	1088	12%	7%	351,2	897	14%	12%	293,3	752	20%	15%	201,4	523	14%	11%	130,3	365	5%	2%	84,1	211	18%	19%

Abbildung 63 - Ergebnisse der Routenplanungen mit den Verbesserungsansätzen 1 und 2 des genetischen Algorithmus

Die Abbildung 63 stellt die Versionsnummer vier von dem genetischen Algorithmus verglichen mit diversen Algorithmen dar. Für diese Versionsnummer wurden beide Verbesserungsansätze angewandt. Das Resultat hiervon zeigt sich darin, dass beide Verbesserungsansätze hinsichtlich der Verbesserung gegenseitig positiv korrelieren. Das heißt, dass die beiden Verbesserungsansätze eine positive Wirkung auf die Routenplanung haben. Ausgenommen dieser zwei Ausreißer ist die Routenplanung mit genetischem Algorithmus die bessere Variante. Die Verbesserung zu dem Nearest Insertion Algorithmus nach euklidischer Distanz liegt zwischen 17 und 30 Prozent. Im Gegensatz zu dem Saving Algorithmus mit bester Variante (Minimum der drei Entfernungsauswertung wurde für den Saving Algorithmus als Vergleichswert angenommen) stellt der genetische Algorithmus für die gesamte

Standortsammlung eine Verbesserung von durchschnittlich zirka 7 Prozent nach Entfernungsdaten dar. Nichtsdestotrotz bietet der Saving Algorithmus nach Fahrzeit für die Standortanzahl 150 eine bessere Variante an, wenn die Fahrtzeit in Betracht gezogen wird. Dabei entwirft der Saving Algorithmus eine Verbesserung von 11 Minuten, obwohl die Entfernung mit 21,7 Kilometer schlechter ausfällt.

Weiters ist der Fortschritt, in Hinblick auf die restlichen Versionierungen, deutlich ersichtlich, da der genetische Algorithmus bei der Versionierung vier überwiegend bessere Routenplanungen entwickelt. Die Auswertung der Versionsnummer vier gegenüber der Versionsnummer drei des genetischen Algorithmus liegt dem zu Grunde, dass die Routenplanungen mit höheren Standortzahlen eine bessere Tourenplanungen anbieten. In den kleineren Standortzahlen sind Routenplanungen für beide Versionsnummer fast identisch. Wenn der genetische Algorithmus nach Entfernung mit der Versionsnummer vier und mit der Versionsnummer drei verglichen wird, sind ausschließlich für zwei Standortbereiche (100 und 25) keine Verbesserungen hervorgesehen. Für die Standortanzahl 75, 125 und 150 hat der genetische Algorithmus (bester Auswahl von den drei Optionen) mit der Versionsnummer vier gegenüber der Versionsnummer drei zwischen 6 bis 8 Prozent eine verbesserte Routenplanung geliefert, wenn die Ergebnisse nach dem Entfernungsparameter verglichen wurde.

Der Vergleich zwischen euklidischer Entfernung und der Bing Maps Daten (Entfernung, Fahrzeit) wird anhand der Abbildung 63 evaluiert. Wenn der Nearest Insertion Algorithmus betrachtet wird, ist es ersichtlich, dass die Routenplanung mit der Bing Maps API nach Entfernung für jede Standortanzahl besser ausfällt als bei der Routenplanung nach euklidischer Entfernung. Dabei wird der Parameter Entfernung verglichen. Durch die Verwendung der Bing Maps API liegt die Verbesserung für den Nearest Insertion Algorithmus bei bis zu 10 Prozent, wenn die Entfernungen von den Routenplanungen betrachtet werden. Nichtdestotrotz ist es erwähnenswert, dass die Routenplanungen nach Fahrzeit in dem Standortbereich (50) in der Entfernung um zwei Prozent schlechter ausfallen als die euklidische Distanz. Wenn der Cheapest Insertion Algorithmus betrachtet wird, wird eine ähnliche Verbesserung mit Bing Maps ersichtlich. Dabei ist immer einer der Varianten von Bing Maps die bessere Option als die euklidische Distanz. Das wird sowohl bei dem Saving Algorithmus als auch beim genetischen Algorithmus deutlich. Zusammenfassend schlägt die Routenplanung mit Bing Maps für die angeführten Algorithmen erkennbar besser aus als die euklidische Distanz. Dabei ist bei der Routenplanung mit Bing Maps großteils die Routenplanung nach Entfernung die

bessere Auswahl als die Routenplanung nach Fahrzeit, wenn die Entfernungsparameter für die Auswertung bestimmt werden.

Nach Angaben des Autors Andreas Mykitiuk (2023) im Blogpost auf der Seite Webfleet verbraucht ein LKW mit einer Nutzlast von ca. 16 Tonnen 25l Diesel pro Kilometer. Nach Angaben von der Firma Michelin (o.D.) erzeugt 1 Liter Diesel 2,68 kg CO2. Durch die Multiplizierung der beiden Angaben kommt der Wert 67 kg CO2 pro 100 km raus. Ein Fallbeispiel wird mit der Abbildung 63 entwickelt. In dieser Abbildung wurde erwiesen, dass zwischen der euklidischen Distanz und dem genetischen Algorithmus nach Entfernung die Strecke bei 150 Standorte um ca. 125 km reduziert wird. Durch diese Routenplanung wird 31,25 Liter an Sprit gespart. Die Auswertung verdeutlicht, dass die Einsparung von 31,25 Liter Diesel 83,75 kg CO2 weniger ausgestoßen lässt. Wenn davon ausgegangen wird, dass täglich 150 Glas Container entleert werden, werden 252 Arbeitstage im Jahr 2024 Sprit von 7.875 Liter Diesel sowie eine CO2-Einsparung von 21.105 kg gespart. Dieses Fallbeispiel zeigt, dass durch die Bing Maps Routenplanung sehr viel an Sprit als auch an CO2 eingespart wird. Die Effizienz für größere Routenplanungen steigt mit Bing Maps und dem genetischen Algorithmus deutlich. Hier wird nicht nur CO2 bzw. Sprit gespart, sondern auch die Verschleißteile von den LKWs werden nachhaltiger genutzt. Dadurch werden die Verschleißteile später ausgetauscht. Im Gesamten werden durch die Bing Maps API und dem genetischen Algorithmus Ressourcen schonend und nachhaltiger geplant - als die traditionelle Variante namentlich Nearest Insertion Algorithmus nach euklidischer Distanz.

6. Conclusio und Ausblick

Diese Masterarbeit leitet mit theoretischem Input von den angewandten Algorithmen in die Thematik ein. Dadurch wurden fachliche Inhalte über folgende Algorithmen verdeutlicht: Nearest Insertion, Cheapest Insertion, Saving Algorithmus und genetischer Algorithmus.

Anschließend wurde die Bing Maps API vorgestellt. Die Bing Maps API sticht mit der Fähigkeit in der Auswertung der LKW-Daten hervor. Dabei wertet Bing Maps durch die Daten des LKWs die optimale Route aus. Hierbei werden zum Beispiel Eigenschaften wie Höhe, Breite und Gewicht betrachtet und gleichzeitig die nicht befahrbaren Routen werden eliminiert. In diesem Kapitel wurden die Schwierigkeiten bzw. Herausforderungen der Bing Maps API dokumentiert. Hierbei muss betont werden, dass diese Herausforderungen im Rahmen der Basisuser*innen entstanden ist.

Darauffolgend wurde im Kapitel „Programmierung“ mit Hilfe des Theorieteils die verschiedenen Algorithmen in Programmiersprache umgesetzt. Dabei wurde für die Ausführung der Algorithmen die Programmiersprache C# verwendet. Für die Visualisierung der Routenplanungen wurde die Programmiersprache JavaScript verwendet, welche in eine HTML-Datei eingebettet wurde. Die letzte Programmiersprache in dieser Arbeit ist VBA. Sie wurde für die Auswertung der 1536 HTML-Dateien in Anwendung gebracht. Ein wesentlicher Aspekt in diesem Kapitel ist die Entwicklung der Verbesserungsansätze für den genetischen Algorithmus.

Im Kapitel 5 werden die Ergebnisse durch vier Tabellen visualisiert und anschließend diskutiert. In Betracht auf die gestellten Fragen wird in diesem Kapitel nach Antworten gesucht.

Zusammenfassend kann der Schluss gezogen werden, dass die Verwendung der Bing Maps API eine deutliche Verbesserung in der Routenplanung mit sich bringt. Die Schlussfolgerung ist, dass die Verwendung der Bing Maps API für die angewendete Algorithmen die bessere Routenplanung durchführt als bei den Routenplanungen mit euklidischer Distanz. Darüber hinaus ist in den meisten Fällen der genetische Algorithmus mit den Verbesserungsansätzen 1 und 2 die bessere Auswahl unter den Algorithmen mit der Bing Maps Kombination. Mithilfe der Auswertungen konnte ein Fallbeispiel dargestellt werden. Im diesem Fallbeispiel findet ein Vergleich zwischen dem Nearest Insertion Algorithmus nach euklidischer Distanz und dem genetischen Algorithmus nach Entfernung (die Entfernungen wurde mit Bing Maps ermittelt) für eine Standortanzahl von 150 statt. In diesem Vergleich hat sich herausgestellt, dass

zwischen Nearest Insertion nach euklidischer Distanz und genetischer Algorithmus nach Entfernung mit der Versionsnummer 4 ein Entfernungsunterschied von 125 km ergibt. Die Routenplanung mit dem genetischen Algorithmus schließt mit einer kürzeren Entfernung. Dieses Fallbeispiel offenbart, dass pro Tag 31,25 Liter an Diesel gespart wird, wenn die Anzahl an besuchten Glas Container pro Tag 150 Standorte beträgt. Aufs Jahr umgerechnet ergibt dieser Wert 7.875 Liter an Diesel. Infolgedessen werden pro Jahr 21.105 kg CO2 eingespart. Dadurch wird erkennbar, dass die Auswahl der Algorithmen und die Planungsmethode eine enorme Auswirkung auf die Umwelt hat.

Für die weitere Forschung in diesem Bereich kann zunächst durch den genetischen Algorithmus andere Funktionsarten wie Selektion, Crossover und Mutation getestet werden. Dabei könnte die gleiche Methodik, ähnlich wie beim Verbesserungsansatz 1, eingesetzt werden. Der Verbesserungsansatz 1 besteht aus einer Mischung von mehreren Funktionsarten für die Erzeugung der Population.

Weiters könnte ein mögliches Forschungsfeld die Erhöhung der Geschwindigkeit des genetischen Algorithmus in Betracht ziehen. In dieser Arbeit wurden erhebliche Zeiten für die Ausführung des Programms vertrieben. Hierbei wäre auch interessant zu beobachten, ob durch die Änderung genetischer Algorithmus-Funktionen, die zeitliche Effizienz gesteigert werden kann. Dennoch sollten infolgedessen die Routenplanungen nicht schlechter ausfallen. Ein weiterer Ansatz wäre, dass die Berechnungen des Programms nicht mehr in der CPU-Einheit, sondern in der GPU-Einheit stattfinden. Dafür ist die Anpassung des gesamten Programms vonnöten.

Die Abrufe der Bing Maps API waren für Entwickler*innen Version limitiert, weshalb eine Distanzmatrix erzeugt wurde. Anhand dieser statischen Distanzmatrix fand die Durchführung der Routenplanungen statt. An dieser Stelle öffnet sich für die Forschung neue Betrachtungsfenster, unter der Bedingung, dass die Routenplanungen nicht anhand einer statischen Distanzmatrix durchgeführt werden. Stattdessen werden die Fahrdaten (Entfernung, Fahrtzeit) während der Routenplanungen dynamisch abgerufen. Die dynamische Einsetzung von den Fahrdaten könnte mit dem genetischen Algorithmus leicht implementiert werden. Dafür ist jedoch ein Abschluss des Abonnements im Bing Maps Portal erforderlich.

7. Literaturverzeichnis

Altinkemer, K. & Gavish, B., 1991. Parallel Savings Based Heuristics for the Delivery Problem. *Operations Research,* Issue 39, pp. 456-1469.

Applegate, D. L., Bixby, R. E., Chvátal, V. & Cook, W. J., 2011. *The Traveling Salesman Problem : A Computational Study.* https://web-s-ebscohost-com.uaccess.univie.ac.at/ehost/ebookviewer/ebook/bmxlYmtfXzM5MDUxMl9fQU41?sid=f0d7064b-5c43-4067-b827-c902394bfd53@redis&vid=4&format=EB&rid=4 Hrsg. s.l.:Princeton: Princeton University Press.

Bing Maps, 2022. *Calculate a Truck Route.* [Online]
Available at: https://learn.microsoft.com/en-us/bingmaps/rest-services/routes/calculate-a-truck-route
[Zugriff am 29 10 2023].

Bing-Maps-Team, 2020. *Bing Maps.* [Online]
Available at: https://blogs.bing.com/maps/2020-06/bing-maps-is-updating-its-base-map-data-source/
[Zugriff am 28 10 2023].

Bochtis, D. & Sørensen, C., 2009. The vehicle routing problem in field logistics part I. *Biosystems Engineering,* pp. 447-457. https://doi.org/10.1016/j.biosystemseng.2009.09.003.

Çatay, T. & Doyuran, B., 2011. A robust enhancement to the Clarke–Wright savings algorithm, Journal of the Operational Research Society, 62:1, 223-231, DOI: 10.1057/jors.2009.176.

Clarke, G. & Wright, J. V., 1964. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research. 12,* p. 568–581.

Cordeau, J.-F.et al., 2002. A Guide to Vehicle Routing Heuristics. *The Journal of the Operational Research Society 53,* 5(53), pp. 512-522.

Dantzig, G. B. & Ramser, J. H., 1959. The truck dispatching problem. *Management Science, 6(1),* p. 80–91.

Gaskell, T., 1967. Bases for Vehicle Fleet Scheduling.. *OR,* Issue 18, pp. 281-295.

Gil-Gala, F., Đurasević, M., Sierra, M. & Varela, R., 2022. Building Heuristics and Ensembles for the Travel Salesman Problem. In: Ferrández Vicente, J.M., Álvarez-Sánchez, J.R., de la Paz López, F., Adeli, H. (eds) Bio-inspired Systems and Applications: from

Robotics to Ambient Intelligence. *IWINAC 2022. Lecture Notes in Computer Science, vol 13259. Springer, Cham. https://doi.org/10.1007/978-3-031-06527-9_13.*

Hasse Transport GmbH, o.D.. *Hasse Transport.* [Online]
Available at: https://www.hasse-transport.de/kontakt/impressum.html
[Zugriff am 08 01 2024].

HMF Ladekrane & Hydraulik GmbH, o.D.. *HMFcranes.* [Online]
Available at: https://de.hmfcranes.com/produkte/ladekrane/hubkapazitat/15-31-mt/1910k-rcs
[Zugriff am 08 01 2024].

Kara, I., Kara, B. & Kadri, M., 2008. Cumulative Vehicle Routing Problems.. In: T. Caric & H. Gold, Hrsg. *Vehicle Routing Problem.* Vienna, Austria: I-Tech, pp. 97-110.

Kumar, R. et al., 2022. *Analyzing The Performance Of Crossover Operators (OX, OBX, PBX, MPX) To Solve Combinatorial Problems.* Faridabad, India, pp. 817-821, doi: 10.1109/COM-IT-CON54601.2022.9850689, International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON).

Kusum, D. & Manoj, T., 2007. A new crossover operator for real coded genetic algorithms. *Applied Mathematics and Computation,* 1(188), pp. 895-911.

Malik, S. & Wadhawa, S., 2014. Preventing Premature Convergence in Genetic Algorithm Using DGCA and Elitist Technique. *International Journal of Advanced Research in Computer Science and Software Engineering,* 6(4), pp. 410-418.

Mattfeld, D. & Vahrenkamp, R., 2014. Knotenorientierte Rundreisen. In: *Logistiknetzwerke.* Wiesbaden: Springer Gabler, pp. 229-275.

Michelin, o.D.. *Connectedfleet Michelin.* [Online]
Available at: https://connectedfleet.michelin.com/de/blog/so-berechnen-sie-den-co2-abdruck-ihrer-flotte
[Zugriff am 18 01 2024].

Misevičius, A., Kuznecovaitė, D. & Platužienė, J., 2018. Some Further Experiments with Crossover Operators for Genetic Algorithms. *Informatica,* 3(29), pp. 499-516.

Mykitiuk, A., 2023. *Webfleet.* [Online]
Available at: https://www.webfleet.com/de_de/webfleet/blog/so-viel-kraftstoff-verbrauchen-lkw/
[Zugriff am 18 01 2024].

Nelson, M., Nygard, K., Griffin, J. & Shreve, W., 1985. Implementation techniques for the vehicle routing problem. *Computers & Operations Research,* Issue 12, pp. 273-283.

Paessens, H., 1988. The savings algorithm for the vehicle routing problem. *European Journal of Operational Research,* Issue 34, pp. 336-344.

Ramadan, S. Z., 2012. Reducing premature convergence problem in genetic algorithm: application on travel salesman problem. *Comput. Inf. Sci.,* 6(1), pp. 47-57.

Rosenkrantz, D., Stearns, R. & Lewis, P., 2009. An analysis of several heuristics for the traveling salesman problem. In: S. S. Ravi & S. K. Shukla, Hrsg. *Fundamental Problems in Computing.* s.l.:Springer Dordrecht, pp. 45-68.

Ryan, C. et al., 2003. Genetic Programming : 6th European Conference,. *EuroGP ,* Issue Springer Berlin Heidelberg .

Schorpp, S., 2011. *Dynamic Fleet Management for International Truck Transportation : Focusing on Occasional Transportation Tasks.* 1 Hrsg. Wiesbaden: Gabler Verlag / Springer Fachmedien Wiesbaden GmbH, Wiesbaden.

Shukla, A., Pandey, H. M. & Mehrotra, D., 2015. *Comparative review of selection techniques in genetic algorithm.* s.l., International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), Greater Noida, India, pp. 515-519, doi: 10.1109/ABLAZE.2015.7154916..

Soni, N. & Tapas, K., 2014. Study of Various Mutation Operators in Genetic Algorithms. *(IJCSIT) International Journal of Computer Science and Information Technologies,* 3(5), pp. 4519-4521.

Stadt-Wien, 2020. *data.gv.at.* [Online]
Available at: https://www.data.gv.at/katalog/dataset/8d482810-26da-4bbc-8599-6b1fd1ab8743#additional-info
[Zugriff am 22 10 2023].

Wark, P. & Holt, J., 1994. A Repeated Matching Heuristic for the Vehicle Routeing Problem. *Journal of The Operational Research Society,* Issue 45, pp. 1156-1167.

Yellow, P., 1970. A Computational Modification to the Savings Method of Vehicle Scheduling. *Journal of The Operational Research Society ,* Issue 21, pp. 281-283.

Zakir, H. A., 2010. Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator. *International Journal of Biometrics and Bioinformatics (IJBB), 3,* pp. 96 - 105.

Zhong, J., Hu, X., Zhang, J. & Min, G., 2005. *Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms.* s.l., International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06), Vienna, pp. 1115-1121, doi: 10.1109/CIMCA.2005.

Zimmermann, H., 1992. Heuristische Verfahren. In: *Methoden und Modelle des Operations Research.* Wiesbaden: Vieweg+Teubner Verlag, pp. 258-259.

8. Appendix

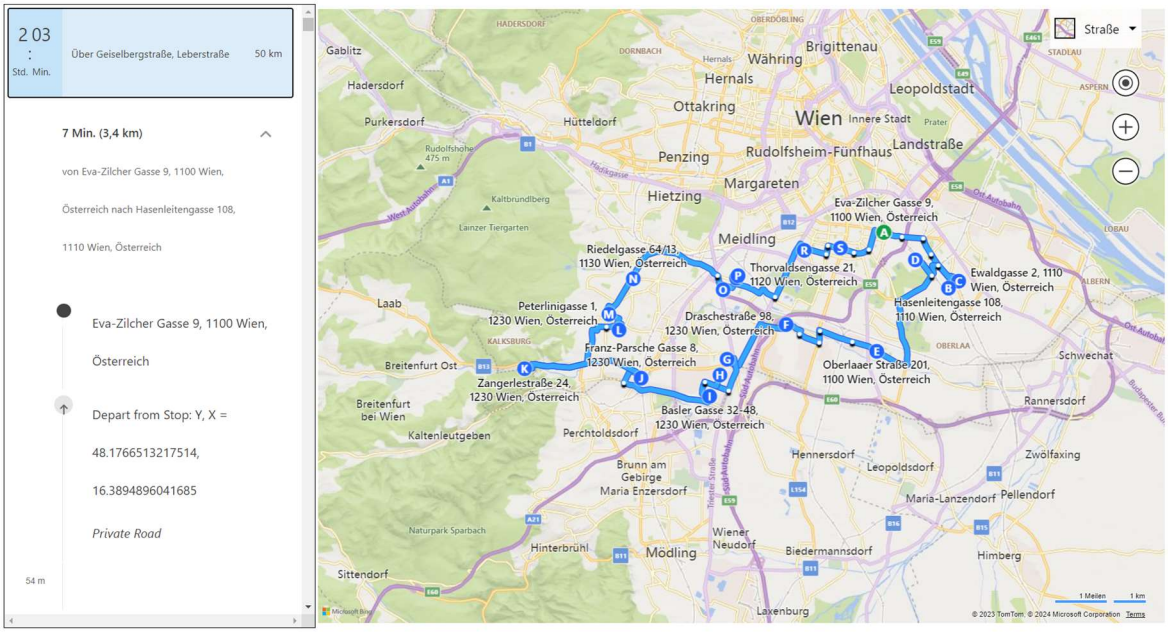


Abbildung 64 –Visualisierung der Routenplanung 1

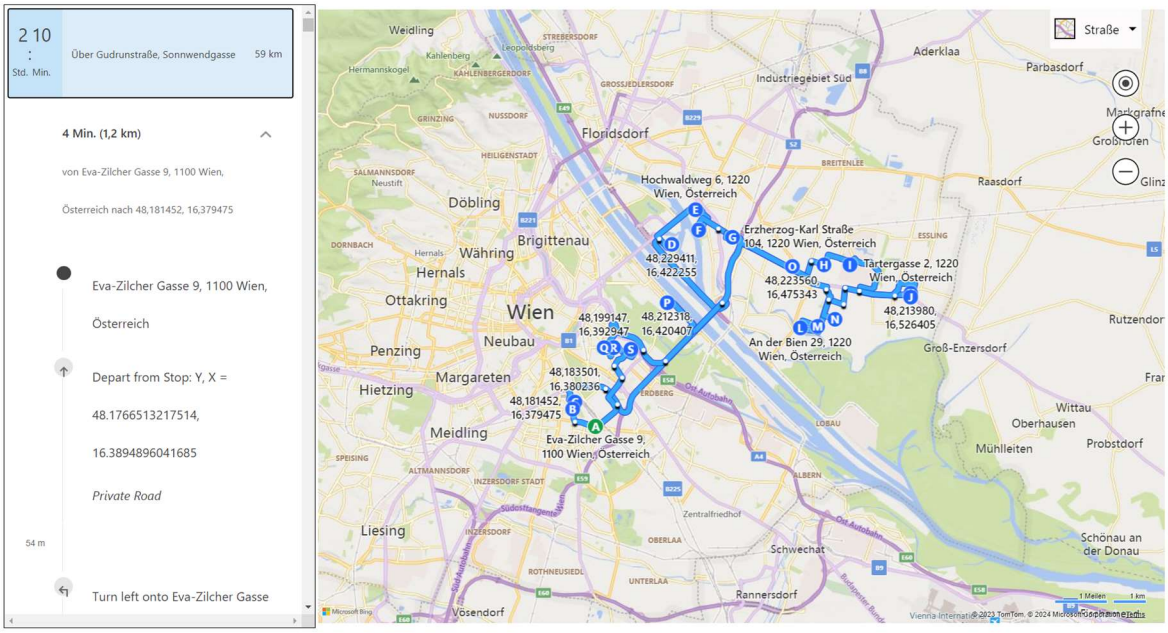


Abbildung 65 - Visualisierung der Routenplanung 2

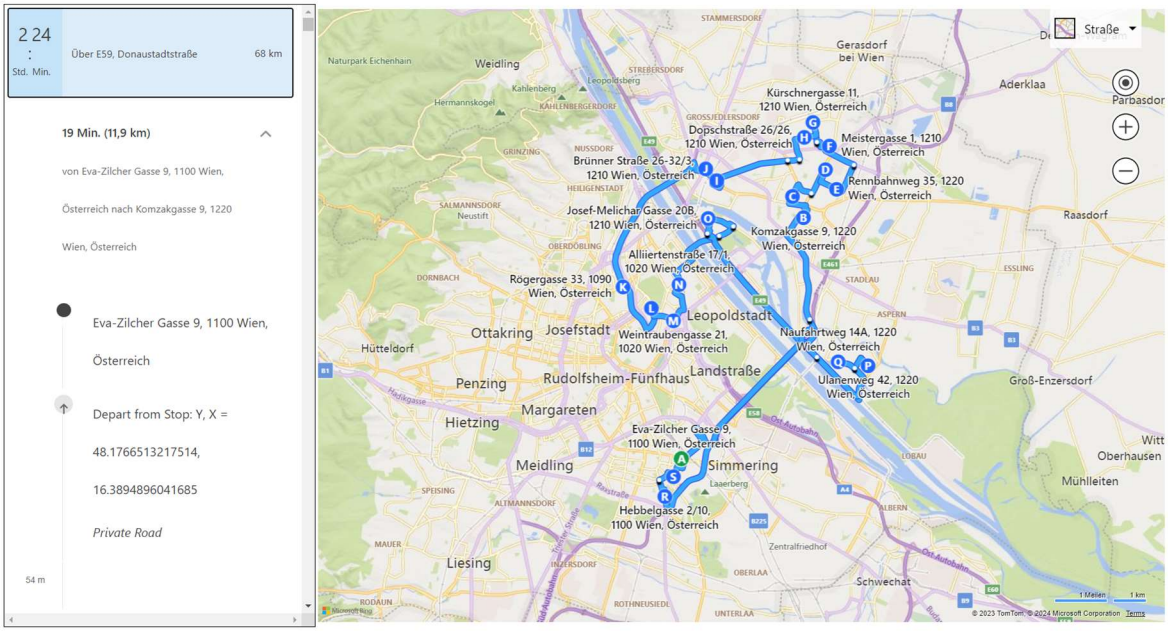


Abbildung 66 - Beispielhafter Visualisierung der Routenplanung 3

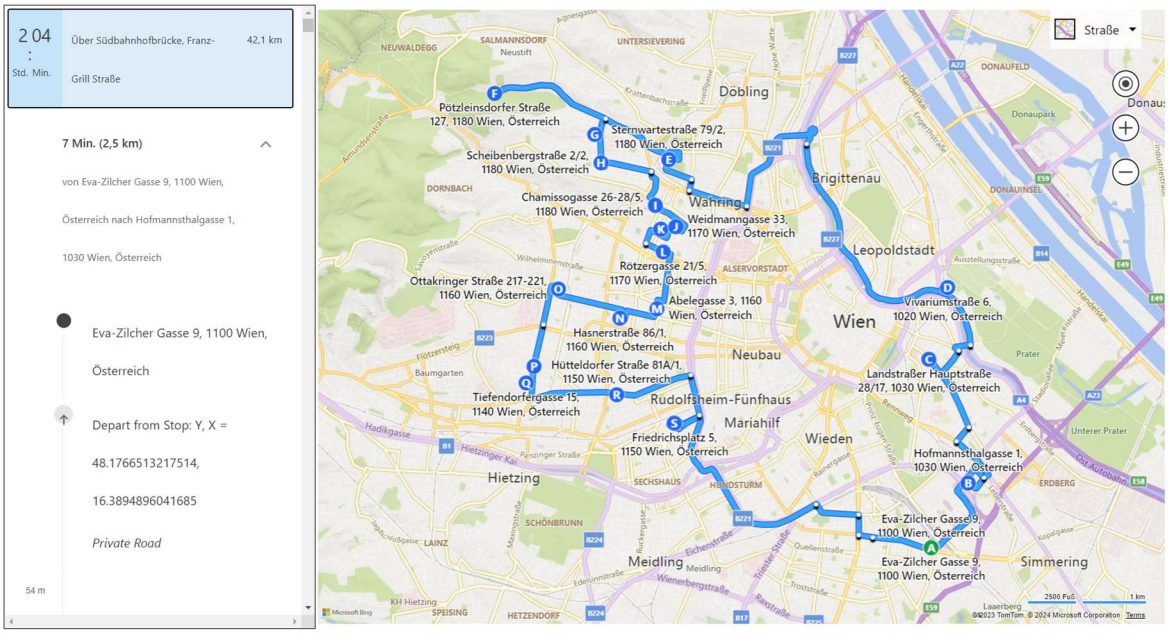


Abbildung 67 - Beispielhafter Visualisierung der Routenplanung 4