Diplomarbeit

Tourenplanung mit Zeitfenstern für mehrere Fahrzeuge

Philipp Kindermann

Betreuer:
Dr. Joachim Spoerhase
Prof. Dr. Alexander Wolff

Lehrstuhl für Informatik I
Effiziente Algorithmen und wissensbasierte Systeme
Institut für Informatik
Julius-Maximilians-Universität Würzburg

10. Januar 2012

Inhaltsverzeichnis

1. Einleitung **4**
1.1. Problemdefinition . . . 4
1.2. Bisheriger Stand der Forschung . . . 5
1.3. Zielstellung . . . 6
1.4. Durchführung . . . 7

2. Vorverarbeitung **8**
2.1. Filtern und anpassen . . . 8
2.1.1. Fahrzeuge . . . 8
2.1.2. Aufträge . . . 8
2.2. Sortierung der Aufträge . . . 10

3. Savings++-Heuristik **11**
3.1. Klassische Vorgehensweise . . . 11
3.2. Erweiterung des Algorithmus . . . 12
3.3. Initialisierung . . . 13
3.4. Savingsberechnung . . . 13
3.5. Optimierungsschritt . . . 15
3.6. Touren speichern und zerstören . . . 15
3.7. Laufzeit und Speicherbedarf . . . 16

4. Insertion-Heuristik **19**
4.1. Klassische Vorgehensweise . . . 19
4.2. Erweiterung des Algorithmus . . . 20
4.3. Einfügeschritt . . . 20
4.3.1. Beste Einfügeposition . . . 22
4.3.2. Bester Auftrag . . . 22
4.4. Laufzeit und Speicherbedarf . . . 23

5. Nachbearbeitung **25**
5.1. Schwellenakzeptanz . . . 25
5.2. Ruin-Schritt . . . 27
5.3. Recreate-Schritt . . . 28
5.4. Verbesserung des Algorithmus . . . 29

6. Optimale Lösung **33**
6.1. Klassische Formulierung . . . 33

6.2. Erweiterung 35
6.2.1. Allgemein 35
6.2.2. Fahrzeuge 37
6.2.3. Aufträge 38
6.2.4. Touren 39
6.3. Eliminierung von Subtouren 41

7. Experimente **44**
7.1. Beispielinstanzen 44
7.2. Auswertung 58

Anhang **65**

A. Liste der Restriktionen **66**
A.1. Fahrzeuge 66
A.2. Personal 67
A.3. Aufträge 67
A.4. Touren 68

B. Ganzzahliges lineares Programm **70**

C. Abbildungsverzeichnis **74**

D. Liste der Algorithmen **76**

E. Literaturverzeichnis **77**

F. Erklärung der selbständigen Bearbeitung **79**

1. Einleitung

Das Problem der Tourenplanung für mehrere Fahrzeuge ist als das *Vehicle Routing Problem* (VRP) bekannt und wurde 1959 von Dantzig & Ramser [DR59] eingeführt. Dabei soll eine Menge von Aufträgen für eine Menge von Kunden zu optimalen Touren (im Sinne von Kosten und Wartezeiten) für Fahrzeuge zusammengestellt werden. Eine Tour besteht dabei aus einer Teilmenge der gegebenen Aufträge und einer dazugehörigen Reihenfolge. Dieses Problem wurde in der Vergangenheit bereits ausgiebig behandelt. Als Verallgemeinerung des *Traveling Salesman Problems* (TSP) ist auch das VRP NP-schwer. Daher wurden mehrere Heuristiken entwickelt, die mit geringem Rechenaufwand und kurzer Laufzeit gute, aber nicht notwendigerweise optimale Lösungen für das Problem berechnen. Für dieses Problem existieren diverse Anwendungsgebiete, zum einen im Logistikbereich, zum Anderen aber auch bei allen Unternehmen, die ihre Kunden beliefern (z.B. Getränkelieferanten, Möbelindustrie oder Zeitungsverlage). Eine bekannte Variante des VRP ist das m-TSP, bei dem genau m Routen konstruiert werden sollen, um alle gegebenen Punkte zu besuchen. Im Gegensatz zu diesem ist beim VRP die Anzahl der Touren nicht fest vorgegeben, sondern nur durch die Anzahl der verfügbaren Fahrzeuge nach oben beschränkt und soll minimiert werden.

1.1. Problemdefinition

Beim VRP ist eine Menge von *Kunden* gegeben, die von einem *Depot* aus von *Fahrzeugen* und dazugehörigem *Personal* beliefert werden müssen, indem alle gegebenen *Aufträge* ausgeführt werden, die sich jeweils auf einen Kunden beziehen. Dabei soll ein *Tourenplan* konstruiert werden, der aus mehreren *Touren* besteht. Eine Tour entspricht dabei einer Folge von Aufträgen und einem Fahrzeug, das beim Depot startet, nacheinander die zu den Aufträgen gehörigen Kunden anfährt und wieder beim Depot endet. Das Ziel ist es, einen kostenminimalen Tourenplan zu konstruieren, der alle Aufträge ausführt und dabei verschiedene gegebene Nebenbedingungen erfüllt. Beim kapazitätsbeschränkten VRP (capacity-constrained VRP, kurz CVRP) wird für jeden Auftrag eine Menge von Waren und für jedes Fahrzeug eine Kapazität angegeben. Dabei darf für keine Tour die Summe der Warenmengen aller belieferten Aufträge die Kapazität des dazugehörigen Fahrzeugs überschreiten. Beim VRP mit Zeitfenstern (VRP with time indow constraints, kurz VRPTW) ist für jeden Auftrag ein Zeitintervall gegeben, innerhalb dessen der Auftrag ausgeführt, also das Fahrzeug beim dazugehörigen Kunden ankommen muss. Beim distanzbeschränkten VRP (distance-constrained VRP, kurz DVRP) sind für jeden Auftrag zusätzlich Stoppzeiten angegeben, die der Entladezeit beim Kunden entsprechen. Dabei

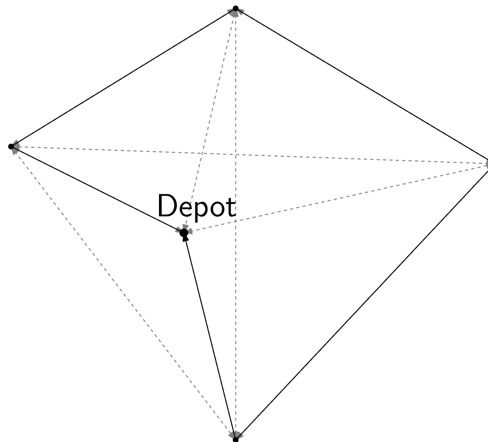


Abb. 1.1.: Lösung eines VRP: Knotenüberdeckung der Kunden

darf die Länge einer Tour, die sich aus Fahr- und Stoppzeiten zusammensetzt, eine vorher festgelegte Obergrenze nicht überschreiten. Im Folgenden werden diese verschiedenen Typen des VRPs kombiniert und es werden weitere Nebenbedingungen hinzugefügt, die beachtet werden müssen. Diese sind im Anhang A nachzulesen.

Das VRP lässt sich durch vollständige gerichtete, gewichtete Graphen modellieren: Sei $G = (V, E)$ ein Graph, wobei $V = \{0, 1, \ldots, n\}$ die Menge der Knoten ist, die das Depot als Knoten 0 und die Kunden als Knoten $1, \ldots, n$ repräsentieren. E ist die Menge der Kanten, also der Wege zwischen jeweils zwei Kunden bzw. zwischen einem Kunden und dem Depot. Durch die Kanten $(i, j) \quad (i \neq j)$ wird eine Distanzmatrix $D = (d_{ij})$ definiert. Aus den Distanzen lassen sich die Kosten berechnen, die durch das Befahren einer Strecke durch ein Fahrzeug entstehen. Gesucht ist nun eine kostenminimale Überdeckung der Knoten mit Kreisen, bei der jeder Knoten (Kunde) genau so viele eingehende Kanten besitzt, wie Aufträge für ihn existieren, wobei jeder Kreis den Knoten 0 beinhalten muss (siehe Abb. 1.1). Im Spezialfall mit genau einem Auftrag pro Kunden besitzt in der Lösung jeder Knoten $1, \ldots, n$ genau eine eingehende und eine ausgehende Kante. Dadurch erhält man eine Menge von Kreisen, wobei jeder Kreis durch den Knoten 0 und damit durch das Depot verläuft und als Tour interpretiert werden kann, die im Depot startet, die auf dem Kreis liegenden Kunden anfährt, die dazugehörigen Aufträge ausführt und wieder im Depot endet. Die Menge dieser Touren entspricht dann dem gesuchten Tourenplan.

1.2. Bisheriger Stand der Forschung

Die im Sinne von Güte und Laufzeit besten Heuristiken zur Lösung des VRPTW wurden von Solomon [Sol87] gegenübergestellt. Dabei verglich er folgende Algorithmen miteinander: die Savings-Heuristik von Clarke und Wright [CW64], eine Nearest-Neighbor Heuristik, die Insertion-Heuristik (auch I1 genannt) und eine Sweep-Heuristik von Gillet und Miller [GM74]. Die besten Ergebnisse erzielte er dabei mit der Insertion-Heuristik.

Der populärste Algorithmus ist allerdings der Savings-Algorithmus. Von Paessens [Pae88] wurden Verbesserungen der Zielfunktion genauer betrachtet sowie eine prozedurale Implementierung angegeben.

Bräysy & Gendreau [BG05] setzten sich mit verschiedenen Möglichkeiten der lokalen Suche und Nachbearbeitung auseinander. Dabei gaben sie zuerst einen Überblick über verschiedene Swap-Verfahren, bei denen einzelne Kunden oder auch ein Folge von Kunden entweder innerhalb der Tour, oder auch zwischen verschiedenen Touren getauscht werden. Anschließend wurden verschiedene Heuristiken miteinander verglichen. Dabei schnitten das Ruin & Recreate Verfahren von Schrimpf et al. [SSSWD00], eine Greedy-Such-Heuristik von Prosser und Shaw [PS96] und eine deterministische, alternierende k-OPT Heuristik von Cordone und Wolfer-Calvo [CC01] am besten ab.

Laporte [Lap92] beschäftigte sich intensiv mit exakten Lösungsverfahren. Dabei stellte er mehrere Lösungswege vor, die hauptsächlich auf ganzzahliger linearer Programmierung basieren. Laporte et al. [LMN86] benutzten dafür einen Branch-and-Bound Algorithmus, der auf dem Spezialfall m-TSP des VRP beruht. Christofides et al. [CT01] bauten ihren Algorithmus sogar auf dem Spezialfall k-degree center tree des m-TSP auf, während Balinski und Quandt [BQ64] eine Mengen Partitionierungs Formulierung des VRP entwickelten. Laporte et al. [LMN85] arbeiteten eine 2-Index Fluss Formulierung für Kapazitäts- und Distanzschranken aus, Fisher und Jaikumar [FJ81] benutzten eine 3-Index Fluss Formulierung für das VRP mit Kapazitätsbeschränkung und Zeitfenstern, aber ohne Stoppzeiten.

1.3. Zielstellung

In der Literatur wird lediglich das Standardproblem der Tourenplanung behandelt, bei dem meist eine unbegrenzte, selten eine begrenzte Zahl von identischen Fahrzeugen mit begrenzter Kapazität zur Verfügung steht. Die Problemstellung umfasst allerdings viele weitere Restriktionen, so dass der Insertion- und der Savings-Algorithmus erweitert und angepasst werden mussten. So ist nur eine feste Menge an Fahrzeugen mit unterschiedlichen Kapazitäten und Restriktionen vorhanden. Außerdem soll eine große Anzahl an Einstellungsmöglichkeiten berücksichtigt werden, die direkten Einfluss auf die Lösung des Problems haben (siehe Anhang A).

Im weiteren Verlauf soll ein lokales Suchverfahren auf Basis von Ruin & Recreate entwickelt werden, das mit zusätzlichem Zeitaufwand die Qualität der ermittelten Touren weiter verbessert. Von Interesse ist ebenfalls die Entwicklung einer exakten Lösung der Probleme durch ganzzahlige lineare Programmierung.

Aufgrund der guten Ergebnisse entschied ich mich dafür, die Insertion-Heuristik anzupassen und zu implementieren. Zur Vergleichbarkeit habe ich ebenfalls den Savings-Algorithmus an das erweiterte Problem angepasst und implementiert. Für die Nachbearbeitung fiel meine Wahl auf das Ruin & Recreate Verfahren nach Schrimpf et al. Als Basis für ein ganzzahliges lineares Programm diente die 3-Index Fluss Formulierung von Fisher und Jaikumar, weil diese bereits Zeitfenster in Betracht zog.

1.4. Durchführung

Die Diplomarbeit entstand im Rahmen einer Zusammenarbeit des Lehrstuhls für Informatik I (vertreten von Prof. Alexander Wolff) und der PASS Logistic Solutions AG (vertreten von Dr. Dirk Schäfer). Herr Schäfer hat an diesem Lehrstuhl unter der Leitung von Prof. Hartmut Noltemeier promoviert und stellte den Kontakt zu Herrn Wolff her, um Studenten zur Neuentwicklung eines Tourenalgorithmus zu finden. Aus dieser Zusammenarbeit resultierte ein Praktikum (Februar bis Mai 2011), welches zur Erstellung der Diplomarbeit verlängert wurde (Juni bis September 2011). Diese Arbeit beschreibt die für PASS entwickelten Algorithmen und deren Resultate.

PASS Logistic Solutions ist eine Untergruppe der PASS Consulting Group, die IT-Produkte, -Lösungen und -Services in diversen Bereichen entwickelt. Das Unternehmen entstand 1981 aus der Diplomarbeit von Gerhard Rienecker. Zu den Aufgabengebieten der PASS Logistic Solutions gehören Tourenplanung und -optimierung (*Plantour*), Flottenmanagement und Controlling (*Trackmanager*) und Auftragsmanagement (*Tracklive*). Im Rahmen der Zusammenarbeit sollte der Tourenplanungs und -optimierungs Algorithmus für Plantour neu geschrieben und dabei verbessert werden. Dabei sind Kunden-, Zeit- und Fahrzeugrestriktionen zu berücksichtigen.

2. Vorverarbeitung

Bevor der eigentliche Algorithmus startet, werden zunächst die Daten vorverarbeitet. Dieser Schritt ist sowohl für Savings als auch für Insertion gleich. Dabei werden ungültige Fahrzeuge und Aufträge entfernt, Aufträge werden angepasst und sortiert (siehe Alg. 1).

Algorithmus 1: preprocess(List<Vehicle> vehicles, List<Order> orders)

```
1 preprocessVehicles(vehicles)
2 preprocessOrder(orders)
```

2.1. Filtern und anpassen

2.1.1. Fahrzeuge

Je nach Einstellungen werden alle Speditions- und alle passiven Fahrzeuge aus der Liste der verfügbaren Fahrzeuge eliminiert, damit diese später nicht zur Tourenplanung verwendet werden (siehe Alg. 2)

Algorithmus 2: preprocessVehicles(List<Vehicle> vehicles)

```
  // siehe Abschnitt 2.1.1
1 Entferne Speditions-Fahrzeuge aus vehicles
2 Entferne passive Fahrzeuge aus vehicles
```

2.1.2. Aufträge

Je nach Einstellungen werden passive Aufträge und Aufträge mit zu geringen Mengen aus der Liste der zu verplanenden Aufträge entfernt. Sind Menge, Gewicht oder Volumen höher als der angegebene Maximalwert, so werden die Mengen des Auftrags je nach Einstellung auf den Maximalwert reduziert, der Auftrag wird auf mehrere Aufträge aufgeteilt oder der Auftrag bleibt unverändert. Außerdem werden für den Auftrag Zeitfenster gesetzt, falls er von sich aus noch keine besitzt. Verfügt der Kunde für das Lieferdatum über feste Öffnungszeiten, so werden diese im Auftrag gespeichert. Wenn der Kunde am Liefertag nicht geöffnet hat, kann der Auftrag nicht ausgeführt werden und wird aus der Liste der zu verplanenden Aufträge gestrichen. Sollten beim Kunden ebenfalls keine Zeitfenster angegeben sein, so werden die Default Zeiten für diese Auftrag verwendet (siehe Alg. 3).

Algorithmus 3: preprocessOrders(List<Order> orders)

```
1  foreach Auftrag ∈ orders do
2      Konvertiere Transporteinheiten in Menge/Volumen/Gewicht
3      if Auftrag ist passiv then
4          Entferne Auftrag aus orders
5          continue
6      if Auftragsmenge/-volumen/-gewicht < Mindestmenge/-volumen/-gewicht then
7          Entferne Auftrag aus orders
8          continue
       // siehe Anhang A: Zeitfenster beachten
9      if Auftrag hat keine Time Slots then
10         if Kunde hat Time Slots then
11             Verwende Time Slots des Kunden
12         else
13             Verwende Default Time Slots
       // siehe Anhang A: Mengen verändern
14     if Mengen ab Obergrenze reduzieren then
15         Reduziere Mengen im gleichen Verhältnis bis Obergrenze
16     else if Mengen ab Obergrenze splitten then
17         Erstelle neue Aufträge mit Obergrenze der Mengen
   // siehe Abschnitt 2.2:
18 Sortiere orders nach Datum und frühest möglicher Ankunftszeit
```

2.2. Sortierung der Aufträge

Damit durch neu angelegte Touren keine Überschneidung mit einer bereits festgelegten Tour für dasselbe Fahrzeug entstehen kann, werden alle Aufträge nach dem frühestmöglichen Anfangszeitpunkt sortiert. So lässt sich für jedes Fahrzeug der früheste Startzeitpunkt ermitteln, indem man die Rückkehrzeit der zuletzt verplanten Tour des Fahrzeugs betrachtet. Weil als erster Auftrag immer der Auftrag mit dem frühestmöglichen Anfangszeitpunkt gewählt wird, können spätere Touren nicht vorher anfangen. So muss man beim Einfügen neuer Aufträge nicht darauf achten, dass sich die Endzeit der Tour mit der Anfangszeit einer bereits verplanten Tour überschneidet.

3. Savings++-Heuristik

Der Savings-Algorithmus ist die bekannteste Heuristik zur Lösung des VRP und wurde von mir als einer von zwei Algorithmen implementiert und an die neuen Restriktionen angepasst. Im Folgenden werden zuerst die klassische Vorgehensweise der Savings-Heuristik vorgestellt und daraufhin die Änderungen beschrieben, die ich zur Anpassung an die zusätzlichen Forderungen vorgenommen habe.

3.1. Klassische Vorgehensweise

Die Savings-Heuristik (ohne Beachtung der Zeitfenster) wurde von Clarke & Wright [CW64] entwickelt. Dabei wird zunächst für jeden Auftrag eine eigene Tour angelegt, die vom Depot zum Kunden und wieder zurück führt (siehe Abb. 3.1). Daraufhin werden jeweils die kompatiblen Touren, die die größte Kostenersparnis (Saving) liefern, miteinander verbunden, indem man sie aneinanderhängt (siehe Abb. 3.2). Sei C_0 das Depot und $C_1, \ldots, C_n$ die Kunden. Dann ist das Saving für zwei Touren $t_i = (C_{i_0}, \ldots, C_{i_m}), t_j = (C_{j_0}, \ldots, C_{j_k}), \quad 1 \leq i_1, \ldots, i_{m-1}, j_1, \ldots, j_{k-1} \leq n, \quad i_0 = i_m = j_0 = j_k = 0$ definiert als die Funktion

$$\text{sav}_{ij} = d_{i_{m-1},0} + d_{0,j_1} - \mu d_{i_{m-1},j_1}, \quad \mu \geq 0$$

wobei $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ der kürzesten Distanz zwischen zwei Kunden/Depots entspricht. Für $\mu = 1$ entspricht das Saving so der Distanz, die man sich einspart, wenn man die Tour t_j an die Tour t_i hängt, so dass man eine neue Tour $t_{i'} = \left(C_{i_0}, \ldots, C_{i_{m-1}}, C_{j_1}, \ldots, C_{j_k}\right)$ erhält.

Von Solomon [Sol87] wurde außerdem eine Methode vorgestellt, pro Kunde ein Zeitfenster zu beachten. Dafür wird für jeden Kunden $C_i, \quad 1 \leq i \leq n$ das Zeitfenster als $[e_i, l_i] \in \mathbb{R} \times \mathbb{R}$, die Wartezeit als $w_i \in \mathbb{R}$ und die Ankunftszeit als $b_i \in \mathbb{R}$ definiert. Hängt man nun zwei Touren wie beschrieben aneinander, so ist die neue Ankunftszeit des ersten

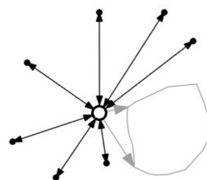


Abb. 3.1.: Initialisierung der Touren

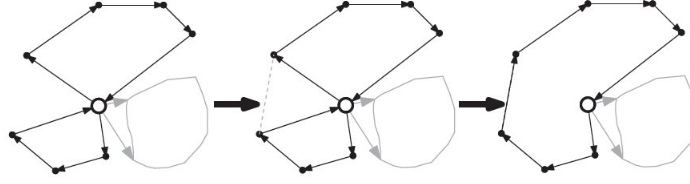


Abb. 3.2.: Verbindung von zwei Touren

Kunden j_1 der zweiten Tour dann

$$b_{j_1}^{\text{new}} = b_{i_m} + w_{i_m} + t_{i_m,j_1}$$

wobei $t : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ der Fahrzeit zwischen zwei Kunden/Depots entspricht. So lässt sich der „Push Forward" definieren, der für alle darauffolgenden Aufträge die Zeit angibt, um die die Ankunftszeit beim Kunden nach hinten verschoben wird:

$$PF_{j_1} = b_{j_1}^{\text{new}} - b_{j_1}$$

$$PF_{j_{r+1}} = \max\{0, PF_{j_r} - w_{j_{r+1}}\},\ \ 1 \leq r \leq m-1$$

Laut Solomon ist eine notwendige und hinreichende Eigenschaft für die Zeitkompatibilität:

$$e_{j_r} \leq b_{j_1}^{\text{new}} \leq l_{j_1}$$

$$e_{j_r} \leq b_{j_r} + PF_{j_r} \leq l_{j_r},\ \ 2 \leq r \leq m$$

3.2. Erweiterung des Algorithmus

In der Literatur wird meistens von einer unbegrenzten Menge an gleichen Fahrzeugen ausgegangen. Weil im Allgemeinen aber nur eine begrenzte Zahl von unterschiedlichen Fahrzeugen zur Verfügung steht, muss der Savings-Algorithmus angepasst werden. Aufgrund der unterschiedlichen Restriktionen (manche Kunden können etwa nur von einem bestimmten Fahrzeug beliefert werden) und der unterschiedlichen Fahrzeugkapazitäten ist es nicht möglich, die Berechnung in einem Schritt durchzuführen. Also wird der Savings++-Algorithmus zunächst nur für ein Fahrzeug ausgeführt (siehe Abschnitt 3.3). Daraufhin werden die entstandenen Touren mit einer Qualitätsfunktion bewertet und nur die beste Tour (siehe Abschnitt 3.6) für das Fahrzeug gespeichert. Für die restlichen Touren versucht man dann jeweils, ein anderes geeignetes Fahrzeug zu finden. Ist dies für eine Tour nicht möglich, wird sie aufgelöst und anschließend der komplette Algorithmus für die verbleibenden Aufträge und ein anderes Fahrzeug von vorne gestartet (siehe Abschnitt 3.6). Dabei muss darauf geachtet werden, dass das Fahrzeug für eine Tour alle Anforderungen der Aufträge erfüllt.

Weil auch Planungen für mehrere Tage durchführbar sein müssen, wird der Algorithmus für jedes Datum, an dem Aufträge vorliegen, separat ausgeführt (siehe Alg. 4). Bei einer Mehrtagesplanung mit Übernachtung müssen allerdings alle Aufträge gemeinsam

verarbeitet werden. Weil keine Daten über Hotels vorliegen, wird dabei von einer Übernachtung beim Kunden ausgegangen. Übernachtungen dürfen allerdings die maximale Wartezeit nicht überschreiten.

Algorithmus 4: Savings++.runAlgorithm(List<Order> orders, List<Vehicle> vehicles)

```
   // siehe Abschnitt 2
1  preprocess()
2  foreach Datum, für das Aufträge vorliegen do
3      dayOrders = Alle Aufträge aus orders von diesem Datum
4      dayVehicles = Kopie von vehicles, um sich zu merken, welche Fahrzeuge an
       diesem Tag bereits verwendet wurden
5      while dayOrders ≠ ∅ do
6          dayTours = initialize(dayOrders, dayVehicles)
           // siehe Abschnitt 3.4
7          Berechne Saving für alle Paare von Touren aus dayTours
8          optimize(dayTours)
9          saveTours(dayTours, dayOrders, dayVehicles)
```

3.3. Initialisierung

Bei der Initialisierung wird zunächst für den frühesten Auftrag das verfügbare Fahrzeug gesucht, bei dem für eine triviale Tour „Depot – Kunde – Depot“ die geringsten Kosten anfallen. Anschließend wird für alle zu diesem Fahrzeug kompatiblen Aufträge jeweils eine weitere triviale Tour angelegt. Gibt es kein kompatibles Fahrzeug für den frühesten Auftrag, so wird dieser nicht verplant und die Suche beginnt mit dem nächsten von vorne. Der Startzeitpunkt der Touren muss dabei aus dem allgemein frühestmöglichen Startzeitpunkt, der Verfügbarkeit des Fahrzeug sowie den Hofzeiten von Fahrzeug, Kunde und Depot berechnet werden.

3.4. Savingsberechnung

Zunächst werden alle Touren paarweise auf ihre Kompatibilität überprüft. Dabei gibt es vier Eigenschaften, die kontrolliert werden müssen. Dafür muss zunächst überprüft werden, ob die Kapazität des Fahrzeugs für die addierten Mengen der beiden Touren ausreichen. Anschließend müssen die Zeitfenster überprüft werden. Dafür wird die Vorgehensweise von Solomon (siehe Abschnitt 3.1) gewählt. Allerdings ist die PushForward-Funktion anzupassen:

Sei s_i die Haltezeit bei Kunde i und a_i die Einfädelzeit nach Kunde i . Hängt man nun zwei Touren wie beschrieben aneinander, so ist die neue Ankunftszeit des ersten Kunden

Algorithmus 5: Savings++.initialize(List<Order> dayOrders, List<Vehicle> dayVehicles)

```
   // siehe Abschnitt 3.3
1  Suche billigstes Fahrzeug aus dayVehicles für den ersten Auftrag aus dayOrders, das
   alle Restriktionen erfüllt
2  Entferne den ersten Auftrag aus dayOrders
3  if Es existiert ein kompatibles Fahrzeug then
4  |  tour = Eine triviale Tour (Depot, Kunde, Depot) mit diesem Fahrzeug
5  |  Speichere tour und je eine weitere triviale Tour (Depot, Kunde, Depot) mit
   |  gleichem Fahrzeug für alle kompatiblen Aufträge aus dayOrders in dayTours
6  |  Entferne alle verplanten Aufträge aus dayOrders
7  |  return dayTours
8  else
9  |_ return initialize()
```

j_1 der zweiten Tour dann

$$b_{j_1}^{\text{new}} = b_{i_m} + w_{i_m} + s_{i_m} + a_{i_m} + t_{i_m,j_1}$$

Weil nun zusätzlich Wartezeiten erlaubt sind, muss die notwendige und hinreichende Eigenschaft für Zeitkompatibilität ebenfalls adaptiert werden:

$$e_{j_r} - \text{maximumWaitingTime} \leq b_{j_1}^{\text{new}} \leq l_{j_1}$$

$$e_{j_r} - \text{maximumWaitingTime} \leq b_{j_r} + PF_{j_r} \leq l_{j_r}, \;\; 2 \leq r \leq m$$

Weil unser Algorithmus nicht die Gesamtdistanz, sondern die Gesamtkosten minimieren soll, wird die Savingsfunktion ebenfalls angeglichen:

$$\text{sav}_{ij} = c_{i_{m-1},0} + c_{0,j_1} - c_{i_{m-1},j_1} - c'_{w_{j_1}} + \text{costPerTour(vehicle)} + \text{fixedCost(depot)}$$

Dabei berechnet $c\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ die Kosten, die entstehen, wenn man mit dem ausgewählten Fahrzeug die Strecke zwischen den beiden Kunden/Depots befährt und die sich aus den Kosten für das Fahrzeug pro Strecke und Zeit, sowie den Kosten für das benötigte Personal pro Zeit zusammensetzen. $c'\colon \mathbb{R} \to \mathbb{R}$ berechnet die Kosten, die für die zusätzliche Wartezeit entstehen. Weil durch die Verkettung eine Tour gespart wird, erhöht sich die Ersparnis zusätzlich um die feste Kosten, die für das Fahrzeug und das Depot pro Tour anfallen.

Möchte man nicht die Kosten, sondern nur die Anzahl der Touren minimieren, so kann man auf den errechneten Savings-Wert noch eine beliebig große Konstante aufaddieren. So werden auch Touren miteinander verknüpft, obwohl dadurch die Gesamtkosten steigen.

3.5. Optimierungsschritt

Nachdem alle Savings berechnet worden sind, werden nun so lange die beiden Touren, bei denen eine Verkettung die größte Ersparnis bringt, miteinander verbunden, bis es kein Tourenpaar mehr gibt, bei dem eine Verkettung die Kosten verringern würde. Hat man zwei Touren

$$t_i = \left(C_{i_0}, \ldots, C_{i_m}\right), t_j = \left(\left(C_{j_0}, \ldots, C_{j_k}\right),\right.$$
$$1 \leq i_1, \ldots, i_{m-1}, j_1, \ldots, j_{k-1} \leq n, \quad i_0 = i_m = j_0 = j_k = 0$$

miteinander verkettet, so dass man eine neue Tour

$$t_{i'} = \left(C_{i'_0}, \ldots, C_{i'_{m+n-1}}\right), \quad i'_l = \begin{cases} 0 & \text{falls } l = 0 \text{ oder } l = m+n-1 \\ i_l & \text{falls } 1 \leq l \leq m-1 \\ j_{l-m+1} & \text{falls } m \leq l \leq m+n-2 \end{cases}$$

erhält, setzt man $t_i = t'_i$ (siehe Alg. 6). Weil wir nun nicht alle Savings wieder von vorne berechnen wollen, entfernen wir zunächst alle Savings aus der Liste, bei denen eine der beiden Touren t_j ist. Daraufhin muss man nur noch die Savings neu berechnen, bei denen eine der beiden Touren t_i ist. Dafür müssen wieder die beiden Touren auf Kompatibilität getestet werden (siehe Abschnitt 3.4). Sollte diese nicht mehr gewährleistet sein, so kann man den dazugehörigen Savingseintrag ebenfalls aus der Liste entfernen.

Algorithmus 6: Savings++.optimize(List<Tour> dayTours)

1 **while** Es existiert ein Touren Paar mit positivem Saving **do**
2 Sei (t_i, t_j) das Tourenpaar mit dem größten Saving, wobei $t_i =$ (Depot, Kunden(t_i), Depot), $t_2 =$ (Depot, Kunden(t_j), Depot)
`// siehe Abschnitt 3.5`
3 Hänge t_j an $t_i \Rightarrow t_i =$ (Depot, Kunden(t_i), Kunden(t_j), Depot)
4 Entferne Tour2 aus dayTours
5 Entferne alle Savings für t_j
6 Berechne alle Savings für t_i neu

3.6. Touren speichern und zerstören

Wenn es kein positives Saving mehr gibt, lassen sich die Touren durch die Savings++-Heuristik nicht mehr verbessern. Weil man nun aber mehrere Touren für ein einziges Fahrzeug berechnet hat, kann man von den erzeugten Touren nur eine einzige speichern. Dafür bewerten wir die Touren nach ihrer Gesamtersparnis, indem wir die Gesamtkosten der Tour mit den Kosten vergleichen, die entstehen würden, wenn man jeden Kunden der Tour einzeln anfahren würde. So erhalten wir für eine Tour t eine Qualitätsfunktion

$$\text{qual}(t) = \frac{cost(t}{\sum\limits_{Auftrag \in t} cost(trivialTour(Auftrag))}.$$

Die qualitativ beste Tour (mit $\min \{qual(t) \mid t \text{ Tour}\}$) wird für das Fahrzeug gespeichert, mit dem der Savings++-Algorithmus durchgeführt wurde. Nun muss man nach Planungsverfahren unterscheiden: Wurde Einfacheinsatz gewählt, so darf dieses Fahrzeug am momentanen Tag keine weiteren Touren fahren. Wurde Mehrfacheinsatz gewählt, so muss man die Verfügbarkeit des Fahrzeugs aktualisieren. Dafür speichert man das Ende der Tour im Fahrzeug. Die nächste Tour darf somit erst ab diesem Zeitpunkt und der Hofzeit starten.

Speichert man pro Durchlauf nur eine einzige Tour, so benötigt der Algorithmus eine sehr hohe Laufzeit (bei einem Testfall von 509 Aufträgen ca. 3 Minuten). Dies lässt sich auf ca. 10 Sekunden verringern, indem man versucht, die restlichen erzeugten Touren ebenfalls zu speichern. Dafür werden die Touren ihrer Qualität nach sortiert und ein anderes Fahrzeug für sie ausgewählt, falls ein kompatibles verfügbar ist. Dafür wird der Start der Tour so verschoben, dass das neue Fahrzeug die Tour zu diesem Zeitpunkt beginnen kann. So erhält man eine Verschiebung der Ankunftszeit des ersten Auftrags der Tour. Ob die komplette Tour verschoben werden kann, ohne dass die Einhaltung der Zeitfenster verletzt wird, kann in gleicher Art und Weise wie bei der Überprüfung von zwei Touren auf Kompatibilität durch PushForwards getestet werden (siehe Abschnitt 3.4). Von allen Fahrzeugen, bei denen dies möglich ist, wird nun das ausgewählt, bei dem die Gesamtkosten für die Tour am Geringsten sind. Damit man allerdings keine zu kurzen Touren erhält, werden neue Fahrzeuge nur ausgewählt, wenn eine festgelegte Kapazitätenauslastung des Fahrzeugs erfüllt wird (diese ist standardmäßig auf 90% festgelegt). So kann es nicht passieren, dass eine Tour, die für ein sehr kleines Fahrzeug berechnet wurde, dann für ein sehr großes Fahrzeug gespeichert wird.

Im Allgemeinen kann nicht für alle Touren sofort ein anderes Fahrzeug gefunden werden. Also werden alle Touren, die nicht gespeichert werden können, wieder zerstört, so dass sie für den nächsten Durchlauf des Algorithmus wieder zur Verfügung stehen (siehe Alg. 7).

3.7. Laufzeit und Speicherbedarf

Sei n die Anzahl der gegebenen Aufträge. Weil nur Kunden betrachtet werden, für die ein Auftrag vorliegt, gibt es maximal n Kunden. Sei k die Anzahl der gegebenen Fahrzeuge. Im Allgemeinen wird von $k < n$ ausgegangen. Da eine Distanzmatrix vorliegt, in der für jedes Paar von Kunden die Distanz gespeichert ist, wird bereits für die Eingabe $O(n^2 + k)$ Speicher benötigt.
In der Vorverarbeitung werden alle Fahrzeuge und Aufträge genau einmal betrachtet. Dabei werden die Aufträge nach ihrem frühesten Startzeitpunkt sortiert. Die Laufzeit beträgt also $O(k + n \log n)$.
Die nächsten Schritte werden nun für jeden Tag, an dem Aufträge vorliegen, so lange durchgeführt, bis alle Touren verplant wurden oder keine mehr verplant werden kann. Im schlechtesten Fall liegen alle Aufträge auf dem gleichen Tag, also muss nur dieser Fall betrachtet werden. Bei der Initialisierung wird ebenfalls jeder Auftrag genau einmal betrachtet. Dabei wird jeder Auftrag mit genau einem oder mit allen Fahrzeugen auf

Algorithmus 7: Savings++.saveTours(List<Tour> dayTours, List<Order> dayOrders, List<Vehicle> dayVehicles)

```
   // siehe Abschnitt 3.6
1  Sortiere Touren aus dayTours nach Verhältnis cost(Tour) / Σ_{Auftrag∈Tour} cost(trivialTour(Auftrag))
2  Speichere Tour mit geringstem Verhältnis
3  if Einfacheinsatz then
4  |  Entferne das Fahrzeug der Tour aus dayVehicles
5  else Mehrfacheinsatz
6  |  Aktualisiere Verfügbarkeit des Fahrzeugs
7  foreach Restliche Touren aus dayTours do
8  |  Suche billigstes kompatibles Fahrzeug
9  |  if Es existiert ein kompatibles Fahrzeug then
10 |  |  Speichere Tour mit neuem Fahrzeug if Einfacheinsatz then
11 |  |  |  Entferne das Fahrzeug der Tour aus dayVehicles
12 |  |  else Mehrfacheinsatz
13 |  |  |  Aktualisiere Verfügbarkeit des Fahrzeugs
14 |  else
15 |  |  Füge alle Aufträge der Tour wieder zu dayOrders hinzu
```

Kompatibilität überprüft.Die Laufzeit hierfür beträgt also $O(k \cdot n)$. Dabei werden $O(n)$ Touren erstellt und gespeichert.
Nun muss für jedes Paar von Touren das Saving berechnet und sortiert werden. Dies benötigt $O(n^2 \log n)$ Zeit und $O(n^2)$ Speicher.
Im Optimierungsschritt werden als Nächstes Touren miteinander verschmolzen. Dabei wird in jedem Iterationsschritt die Gesamtzahl der Touren um 1 verringert. Die innere Schleife wird also maximal $O(n)$ mal aufgerufen. Dabei werden alle Savings der beiden Touren aus der Liste entfernt und daraufhin für die neue Tour die Savings-Werte neu berechnet. Für jede Tour existieren $O(n)$ Savings-Werte. Weil die neuen Werte wieder in die Sortierung aufgenommen werden müssen, benötigt man für jeden Schleifendurchlauf eine Laufzeit von $O(n \log n)$, insgesamt also $O(n^2 \log n)$.
Beim Speichern und Zerstören werden zunächst alle Touren nach ihrer Qualität sortiert, was $O(n \log n)$ Zeit und $O(n)$ Speicher benötigt. Anschließend wird jede der übrigen Touren betrachtet und mit allen verfügbaren Fahrzeugen auf Kompatibilität verglichen. Im schlechtesten Fall wurden keine Touren miteinander verschmolzen, so dass $O(n)$ Touren mit $O(k)$ Fahrzeugen verglichen. Die Laufzeit beträgt also $O(n \log n + k \cdot n)$.
Nachdem möglichst viele Touren gespeichert wurden, beginnt die Schleife von Neuem. Im Falle eines Mehrfacheinsatzes wird die Schleife im schlechtesten Fall $O(n)$ mal durchlaufen. Weil aber immer mindestens eine Tour gespeichert wird, wird die Schleife im Fall eines Einfacheinsatzes nur $O(k)$ mal ausgeführt.

Die Gesamtlaufzeit des Algorithmus beträgt also bei Einfacheinsatz

$$O\left(k + n\log n + k \cdot \left(k \cdot n + n^2 \log n + n^2 \log n + n \log n + k \cdot n\right)\right) = O\left(k^2 n + kn^2 \log n\right)$$

und bei Mehrfacheinsatz

$$O\left(k + n\log n + n \cdot \left(k \cdot n + n^2 \log n + n^2 \log n + n \log n + k \cdot n\right)\right) = O\left(k^2 n + n^3 \log n\right).$$

Wegen $k < n$ lässt sich die Gesamtlaufzeit verallgemeinern zu

$$O\left(n^3 \log n\right).$$

Der Speicherbedarf beträgt dabei

$$O\left(n^2 + k + n + n^2 + n\right) = O\left(n^2 + k\right) \underset{k<n}{=} O\left(n^2\right).$$

4. Insertion-Heuristik

4.1. Klassische Vorgehensweise

Der Insertion-Algorithmus wurde von Solomon [Sol87] vorgestellt. Dabei wird zunächst eine triviale Tour mit einem Auftrag initialisiert, die dann durch weitere Aufträge sequentiell vergrößert wird, indem für jeden bisher unverplanten Auftrag zunächst die beste Einfügeposition gesucht und dann der Auftrag eingefügt wird, bei dem die Kostenersparnis gegenüber einer direkten Anfahrt vom Depot am höchsten ist (siehe Abb. 4.1).

Sei wieder C_0 das Depot, $C_1, \ldots, C_n$ die Kunden und $t = (C_{i_0}, \ldots, C_{i_m})\,, 1 \leq i_1, \ldots, i_{m-1} \leq n, \quad i_0 = i_m = 0$ die aktuelle Tour. Um die beste Einfügeposition zu bestimmen, wird für jeden noch verfügbaren Kunden $C_u, \quad 1 \leq u \leq n$ die beste Einfügeposition $p^*(u) \in \{0, \ldots, m-1\}$ gesucht, indem die Funktion $c_1(i_p, u, i_{p+1})$ berechnet wird, die die zusätzlichen Kosten für die Tour bestimmt, wenn der Kunde C_u zwischen den Kunden C_{i_p} und $C_{i_{p+1}}$ eingefügt wird. Für die beste Einfügeposition $p^*(u)$ gilt dann:

$$c_1\left(i_{p^*(u)}, u, i_{p^*(u)+1}\right) = \min\left[c_1(i_p, u, i_{p+1})\right], \quad p = 0, \ldots, m-1$$

Um die Zeitkompatibilität zu testen, genügt das bereits in Abschnitt 3 vorgestellte Verfahren der Push Forwards.

Um den besten Kunden $C_{u^*}, 1 \leq u^* \leq n$ zu bestimmen, wird die Funktion $c_2\left(i_{p^*(u)}, u, i_{p^*(u)+1}\right)$ berechnet, die die Ersparnis ermittelt, die entsteht, wenn man den Kunden C_u an seiner

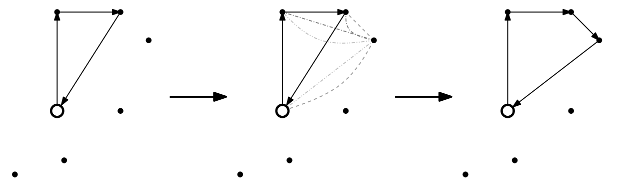


Abb. 4.1.: Einfügeschritt der Insertion-Heuristik

besten Einfügeposition in die Tour einfügt, anstatt ihn direkt vom Depot aus anzufahren. Für den besten Kunden C_{u^*} gilt dann:

$$c_2\left(i_{p^*(u^*)}, u^*, i_{p^*(u^*)+1}\right) = \max\left[c_2\left(i_{p^*(u)}, u, i_{p^*(u)+1}\right)\right], \quad C_u \text{ verfügbar und zulässig}$$

Ein Kunde C_u ist genau dann zulässig, wenn eine beste Einfügeposition $p^*(u)$ existiert, die keine Restriktionen verletzt. Der Kunde C_{u^*} wird dann zwischen den Kunden $C_{i_{p^*(u^*)}}$ und $C_{i_{p^*(u^*)+1}}$ in die Tour t eingefügt. Wenn es keinen Kunden mit zulässiger Einfügeposition mehr gibt, wird eine neue Tour angelegt, es sei denn, es wurden bereits alle Aufträge verplant.

Für die beiden Funktionen $c_1 : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ und $c_2 : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ wurden von Solomon mehrere Möglichkeiten vorgestellt. Zur besseren Vergleichbarkeit zum Savings++-Algorithmus wurde folgende Methode gewählt:

$$c_{11}\left(i_p, u, i_{p+1}\right) = d_{i_p,u} + d_{u,i_{p+1}} - \mu d_{i_p,i_{p+1}}, \quad \mu \geq 0,$$

$$c_{12}\left(i_p, u, i_{p+1}\right) = b_{i_{p+1},u} - b_{i_{p+1}},$$

wobei $b_{i_{p+1},u}$ die Zeit ist, zu der das Fahrzeug beim Kunden $C_{i_{p+1}}$ ankommt, nachdem der Kunde C_u in die Tour eingefügt worden ist. $c_1 1 : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ beschreibt also die zusätzliche Distanz und $c_1 2 : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ die zusätzliche Dauer, die entsteht, wenn der Kunde C_u in die Tour eingefügt wird. c_1 und c_2 werden dann wie folgt berechnet:

$$c_1\left(i_p, u, i_{p+1}\right) = \alpha_1 c_{11}\left(i_p, u, i_{p+1}\right) + \alpha_2 c_{12}\left(i_p, u, i_{p+1}\right), \quad \alpha_1 + \alpha_2 = 1,$$

$$c_2\left(i_p, u, i_{p+1}\right) = \lambda d_{0,u} - c_1\left(i_p, u, i_{p+1}\right), \quad \lambda \geq 0.$$

Dadurch wird der Gewinn maximiert, der durch das Einfügen eines Kunden in eine bereits vorhandene Tour gegenüber einer direkten Anfahrt vom Depot entsteht.

4.2. Erweiterung des Algorithmus

Die Anpassung an die unterschiedlichen Fahrzeuge ist bei der Insertion-Heuristik deutlich einfacher. Weil nicht mehrere Touren parallel, sondern nur einzelne hintereinander aufgebaut werden, kann man bei jedem Initialisierungsschritt ein anderes Fahrzeug auswählen. Dabei wird in jedem Schritt der erste Auftrag aus der Sortierung entnommen und einem Fahrzeug zugewiesen. Deshalb können nur Fahrzeuge ausgewählt werden, die den Auftrag innerhalb des gegebenen Zeitfensters ausführen können. Von diesen Fahrzeugen wird das genommen, bei dem die geringsten Kosten entstehen (siehe Alg. 9).

4.3. Einfügeschritt

Für jeden noch nicht verplanten Auftrag wird zunächst überprüft, ob er überhaupt eingefügt werden kann. Dabei müssen alle Restriktionen des Fahrzeugs und des Auftrags

Algorithmus 8: Insertion.runAlgorithm(List<Order> orders, List<Vehicle> vehicles)

```
   // siehe Abschnitt 2
 1 preprocess()
 2 foreach Datum, für das Aufträge vorliegen do
 3 |  dayOrders = Alle Aufträge aus orders von diesem Datum
 4 |  dayVehicles = Kopie von vehicles, um sich zu merken, welche Fahrzeuge an
 |    diesem Tag bereits verwendet wurden
 5 |  dayTours = Leere Liste für Touren
 6 |  while dayOrders ≠ ∅ do
 7 |  |  tour = initialize(dayOrders, dayVehicles)
 8 |  |  optimize(tour, dayOrders)
 9 |  |  Speichere tour
10 |  |  if Einfacheinsatz then
11 |  |  |  Entferne das Fahrzeug der Tour aus dayVehicles
12 |  |  else Mehrfacheinsatz
13 |  |  |  Aktualisiere Verfügbarkeit des Fahrzeugs
```

Algorithmus 9: Insertion.initialize(List<Order> dayOrders, List<Vehicle> dayVehicles)

```
   // siehe Abschnitt 4.2
 1 Suche billigstes Fahrzeug aus dayVehicles für den ersten Auftrag aus dayOrders, das
   alle Restriktionen erfüllt
 2 Entferne den ersten Auftrag aus dayOrders
 3 if Es existiert ein kompatibles Fahrzeug then
 4 |  return Eine triviale Tour „Depot - Kunde - Depot“ mit diesem Fahrzeug
 5 else
 6 |  return initialize()
```

erfüllt sein (siehe Anhang A). Anschließend wird für alle kompatiblen Aufträge zunächst die Position in der Tour gesucht, bei der die zusätzlichen Kosten für die Tour nach Einfügen des Kunden am geringsten sind. Daraufhin wird für jeden Auftrag und seine beste Einfügeposition die Kosten für das Einfügen in die Tour mit den Kosten verglichen, die anfallen, wenn man den Kunden direkt vom Depot aus anfährt. Der Auftrag, bei dem die so berechnete Ersparnis am höchsten ist, wird dann in die Tour eingefügt. Dies wird so lange wiederholt, bis die Tour nicht mehr vergrößert werden kann, weil für alle restlichen Aufträge entweder Restriktionen verletzt werden, oder eine direkte Anfahrt vom Depot aus billiger ist.

4.3.1. Beste Einfügeposition

Weil die Gesamtkosten der Tour minimiert werden sollen, müssen die Funktionen etwas angepasst werden. In der klassischen Vorgehensweise entspricht die Kostenfunktion c_1 für das Einfügen eines Auftrags nur einer Gewichtung der dadurch zusätzlich entstehenden Distanz und Dauer, also nicht den tatsächlich entstehenden Kosten.

Grundsätzlich sollen möglichst viele Zweimannaufträge zusammen verplant werden, während es sehr schlecht ist, wenige Zweimannaufträge in einer langen Tour unterzubringen. Um dies zu realisieren, wird die Funktion $c_{11}(i, u, j)$ angepasst. Will man einen Zweimannauftrag in eine Tour einfügen, die nur aus Einmannaufträgen besteht, so werden zusätzlich die Kosten für die gesamte Tour berechnet, wenn man einen Beifahrer benötigt. So wird es mit der Länge der Tour immer unwahrscheinlicher, dass das Einfügen eines Zweimannauftrags eine Gesamtersparnis bringt.

Im Gegenzug ist es für Zweimanntouren günstiger, weitere Zweimannaufträge hinzuzufügen, weil für diese die Kosten $c(d_{0,u})$ für eine Anfahrt vom Depot und somit die Ersparnis $c_2(i, u, j)$ größer sind.

4.3.2. Bester Auftrag

Aus allen Aufträgen wird nun der Auftrag u mit Einfügeposition (i, j) gewählt, bei dem die Kostenersparnis am größten ist, wenn man dafür u nicht direkt vom Depot aus anfahren muss. Dafür wird der Auftrag mit dem größten (positiven) $c_2(i, u, j)$ ausgewählt:

$$c_2(i, u, j) = \lambda \cdot (c(d_{0,u}) + costPerTour(v)) - c_1(i, u, j), \quad \lambda \geq 0$$

Hier entspricht v dem Fahrzeug, das der Tour zugeteilt ist, der Default Wert für λ ist 1. Die Kosten für $d_{0,u}$ setzen sich zusammen aus den Kosten für Fahrzeug, Fahrer und (im Falle einer Zweimannbesatzung) Beifahrer.

Solange es noch einen Auftrag mit positivem $c_2(i, u, j)$ gibt, werden die Schritte in den Abschnitten 4.3.1 bis 4.3.2 wiederholt, danach wird die Tour abgeschlossen und die nächste angefangen (siehe Alg. 10).

Algorithmus 10: Insertion.optimize(Tour tour, List<Order> dayOrders)

```
1 while dayOrders ≠ ∅ do
2     foreach Auftrag ∈ dayOrders do
          // siehe Abschnitt 4.3.1
3         Berechne beste zulässige Einfügeposition (sofern existent)
          // siehe Abschnitt 4.3.2
4         Berechne Ersparnisfunktion c2 für Einfügepositon
5     if Es existiert ein Auftrag mit zulässiger Einfügeposition then
6         Füge den Auftrag mit größter Ersparnis in die Tour ein
7         Entferne den Auftrag aus dayOrders
8     else
9         break
```

4.4. Laufzeit und Speicherbedarf

Sei n die Anzahl der gegebenen Aufträge. Weil nur Kunden betrachtet werden, für die ein Auftrag vorliegt, gibt es maximal n Kunden. Sei k die Anzahl der gegebenen Fahrzeuge. Im Allgemeinen wird von $k < n$ ausgegangen. Da eine Distanzmatrix vorliegt, in der für jedes Paar von Kunden die Distanz gespeichert ist, wird bereits für die Eingabe $O(n^2 + k)$ Speicher benötigt.

In der Vorverarbeitung werden alle Fahrzeuge und Aufträge genau einmal betrachtet. Dabei werden die Aufträge nach ihrem frühesten Startzeitpunkt sortiert. Die Laufzeit beträgt also $O(k + n \log n)$.

Die nächsten Schritte werden nun für jeden Tag, an dem Aufträge vorliegen, so lange durchgeführt, bis alle Touren verplant sind oder keine mehr verplant werden kann. Im schlechtesten Fall liegen alle Aufträge auf dem gleichen Tag, also muss nur dieser Fall betrachtet werden. Bei der Initialisierung werden so lange Aufträge betrachtet, bis einer gefunden wurde, für den eine Tour erstellt werden kann. Im schlechtesten Fall müssen dabei $O(n)$ Aufträge mit allen Fahrzeugen auf Kompatibilität überprüft werden. Die Laufzeit hierfür beträgt also $O(k \cdot n)$. Dabei wird eine Tour gespeichert. In der Praxis werden dabei allerdings sehr wenige Aufträge betrachtet, weil im Allgemeinen fast alle Aufträge mit mindestens einem Fahrzeug kompatibel sind.

Nun muss für jeden verbleibenden Auftrag die beste Einfügeposition berechnet werden. Im schlechtesten Fall besteht die Tour aus $O(n)$ Aufträgen und es sind noch $O(n)$ Aufträge unverplant, so dass die Laufzeit dafür $O(n^2)$ beträgt. Für jeden unverplanten Auftrag wird dabei die beste Einfügeposition gespeichert, also liegt der Speicherbedarf in $O(n)$. Wurde die Anzahl an Tourposten allerdings durch die Eingabe begrenzt, so kann eine Tour nur aus $O(1)$ Aufträgen bestehen. Somit schrumpft die Laufzeit auf $O(n)$.

Anschließend wird für jeden Auftrag und seine beste Einfügeposition die Ersparnis berechnet, die entsteht, wenn man den dazugehörigen Kunden in die Tour einfügt, anstatt ihn direkt vom Depot aus anzufahren. Diese Berechnung kostet für jeden Auftrag konstante Zeit, so dass insgesamt $O(n)$ Zeit benötigt wird. Weil nur die höchste Ersparnis

gemerkt werden muss, wird nur $O(1)$ Speicher benötigt. Die Tour wird anschließend um den Auftrag mit der höchsten Ersparnis erweitert.
Diese beiden Schitte werden so lange wiederholt, bis entweder die maximale Anzahl an Tourposten erreicht wird ($O(1)$ mal) oder bis die Tour nicht mehr erweitert werden kann. Berechnet man nun die neuen besten Einfügepositionen, so kann die bereits vorher berechnete beste Einfügeposition wiederverwendet werden. Durch das Einfügen von einem einzelnen Auftrag kamen nur zwei neue mögliche Einfügepositionen hinzu: zwischen dem neuen Auftrag und seinem Vorgänger in der Tour oder zwischen dem neuen Auftrag und seinem Nachfolger. Allerdings existiert die Einfügeposition zwischen dem Vorgänger und dem Nachfolger des neuen Auftrags nicht mehr. Sollte also die alte beste Einfügeposition zwischen diesen beiden Knoten liegen, müssen wieder alle möglichen Einfügepositionen betrachtet werden.

Die Worst-Case Laufzeit des Algorithmus beträgt somit

$$O\left(k + n\log n + k \cdot n + n \cdot \left(n^2 + n\right)\right) = O\left(k + n\log n + k \cdot n + n^3\right) = O\left(k \cdot n + n^3\right).$$

Wegen $k < n$ lässt sich die Gesamtlaufzeit verallgemeinern zu $O\left(n^3\right)$. Der Speicherbedarf beträgt dabei

$$O\left(n^2 + k + n + n\right) = O\left(n^2 + k\right) \underset{k<n}{=} O\left(n^2\right).$$

Der zusätzliche Speicherbedarf, der durch den Algorithmus entsteht, beträgt allerdings nur $O(n)$.

Wie man in Abschnitt 7.2 sieht, ist die praktische Laufzeit des Insertion-Algorithmus trotz gleicher asymptotischer Worst-Case Laufzeit wie der Savings++-Algorithmus deutlich geringer. Durch Rückwärtsanalyse (siehe Seidel [Sei91]) lässt sich für randomisierte Instanzen leicht zeigen, dass der Aufwand pro Auftrag trotzdem erwartet linear ist. Im ersten Schleifendurchlauf besteht die Tour nur aus einem Auftrag, also gibt es nur zwei mögliche Einfügepositionen: zwischen dem Depot und dem Auftrag oder zwischen dem Auftrag und dem Depot. Also müssen nur $O(1)$ Vergleiche durchgeführt werden. Sei nun m die Anzahl der Aufträge in der momentan betrachteten Tour im m-ten Schleifendurchlauf. Dann gab es vor dem Einfügen des letzten Auftrags $m-1$ Aufträge in der Tour und somit m mögliche Einfügepositionen. Somit ist die Wahrscheinlichkeit, dass die alte beste Einfügeposition eines Auftrags die Einfügeposition ist, die nicht mehr existiert, gleich $1/m$. In diesem Fall müssen $m+1$ Einfügepositionen betrachtet werden. Somit ist die erwartete Anzahl der Vergleiche pro Auftrag

$$O\left(\frac{1}{m} \cdot (m+1) + \frac{m-1}{m} \cdot 2\right) = O\left(\frac{m}{m} + \frac{1}{m} + 2 \cdot \left(\frac{m}{m} - \frac{1}{m}\right)\right) = O\left(3 - \frac{1}{m}\right) = O(1).$$

Die (erwartete) Gesamtlaufzeit des Algorithmus beträgt also

$$O\left(k + n\log n + k \cdot n + n \cdot (n + n)\right) = O\left(k + n\log n + k \cdot n + n^2\right) = O\left(k \cdot n + n^2\right).$$

Wegen $k < n$ lässt sich die (erwartete) Gesamtlaufzeit verallgemeinern zu $O\left(n^2\right)$.

5. Nachbearbeitung

Als Nachbearbeitung habe ich die Ruin & Recreate Methode gewählt, die 2000 von Schrimpf et al. [SSSWD00] vorgestellt worden war. Dieses Verfahren geht in zwei Schritten vor: Zunächst wird eine zufällige Anzahl von Kunden, die bezüglich einer Metrik nah beieinander liegen, aus den Touren entfernt. Im zweiten Schritt werden die entfernten Kunden in zufälliger Reihenfolge wieder in die vorhandenen Touren eingefügt. Diese beiden Schritte werden wiederholt, bis eine Abbruchbedingung erfüllt ist (im Allgemeinen eine maximale Anzahl an Durchläufen). Dabei wird durch die Methode der *Schwellenakzeptanz* entschieden, ob der nächste Durchlauf mit der neuen oder der alten Lösung weiterarbeitet. Das Ergebnis entspricht dann der besten Lösung über alle Durchläufe.

Eine Beispieliteration ist in Abb. 5.1 dargestellt. Oben links sieht man die Ausgangslösung, bestehend aus vier Touren. Aus diesen Touren werden nun durch einen Kreis um einen Kunden alle Tourposten ausgewählt, die aus ihren Touren entfernt werden sollen. Unten rechts sieht man dann, wie die Touren nach dem Recreate Schritt aussehen könnten.

5.1. Schwellenakzeptanz

Eine bekannte Methode der Nachbearbeitung ist die lokale Suche. Dabei wird ausgehend von einer Startlösung das Ergebnis schrittweise modifiziert, solange dies eine Verbesserung der Zielfunktion bringt. Bei vielen komplexen Problemen gelangt man allerdigns im Allgemeinen nicht zum optimalen Ergebnis, indem man von einer Startlösung aus versucht, das Resultat zu verbessern, ohne den Zwischenschritt über schlechtere Lösungen zu gehen. Die Wahrscheinlichkeit wäre so sehr hoch, dass man in einem lokalen Minimum (bzw. Maximum) stecken bleibt, so dass man das globale Minimum nicht mehr erreichen kann. Dieses Problem illustriert Abbildung 5.2.

Um aus solchen lokalen Minima zu entkommen, wurde 1990 von Dueck und Scheuer [DS90] die Schwellenakzeptanz Methode als Alternative zur simulierten Abkühlung entwickelt, die unabhängig voneinander 1983 von Kirkpatrick et al. [KGV83] und 1985 von Černý [Č85] beschrieben worden war. Durch Schwellenakzeptanz oder simulierte Abkühlung ist es den Algorithmen möglich, auch schlechtere Lösungen zu akzeptieren, um von diesen aus möglicherweise eine noch bessere Gesamtlösung zu erhalten. Während die simulierte Abkühlung dabei nach jeder Iteration über eine sinkende Wahrscheinlichkeit entscheidet, ob die schlechtere Lösung akzeptiert wird, berechnet die Schwellenakzeptanz nach jedem Schritt einen feste Schwelle T. Eine neue Lösung wird genau dann akzeptiert, wenn sie um maximal T schlechter ist als die alte Lösung. Dafür muss eine Qualitätsfunktion vorliegen, die die Güte einer Lösung beschreibt. In unserem Fall ist dies die

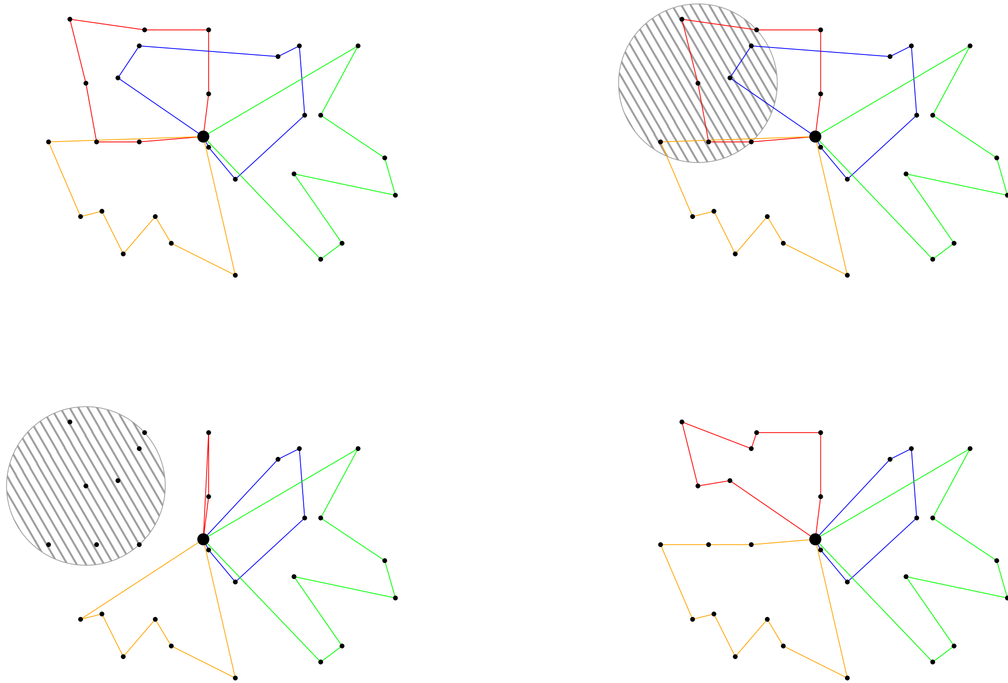


Abb. 5.1.: Eine Ruin & Recreate Iteration

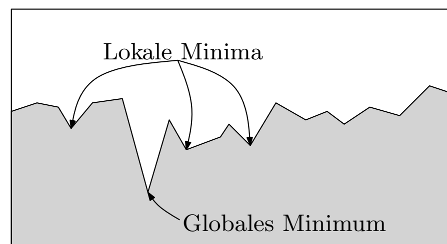


Abb. 5.2.: Lokale Minima

Kostenfunktion, die die Gesamtkosten aller Touren aufsummiert.

Schrimpf et al. benutzen dabei eine exponentielle Abkühlung als Schwellwertfunktion. Als anfänglicher Schwellwert T_0 wurde die Hälfte der Standardabweichung von der Zielfunktion während eines Ruin & Recreate Durchlaufs gewählt, bei dem $T = \infty$ ist. Für die tatsächliche Nachbearbeitung wird dann in jedem Schritt der sinkende Schwellwert durch folgende Funktion berechnet:

$$T = (T_0/2)^{x/\alpha}$$

Dabei wurde die Halbwertszeit α auf 0.1 gesetzt. Die Variable x wurde während des Optimierungsdurchlaufs von 0 bis 1 erhöht. Um dies zu erreichen, wähle ich x = Momentaner Durchlauf / Anzahl Durchläufe. Der Aufbau des Nachbearbeitungsalgorithmus ist in Alg. 11 dargestellt.

Algorithmus 11: ruinRecreate(List<Tour> tours, int max_runs)

```
1  T_0 = Anfänglicher Schwellwert
2  C_old = tours                  // Konfiguration vor Ruin & Recreate Schritt
3  C_ref = C_old                             // Bisher beste Konfiguration
4  for run = 1 to max_runs do
5      x = run / max_runs
6      T = T_0 · exp(− ln 2 · x/α)
7      B = ruin(tours, run)
8      recreate(tours, B)
9      C_new = tours              // Konfiguration nach Ruin & Recreate Schritt
10     if cost(C_new) < cost(C_ref) then
11         C_ref = C_new
12     if cost(C_new) - cost(C_old) < T then
13         C_old = C_new
14     else
15         tours = C_old
16 return C_ref
```

5.2. Ruin-Schritt

Im Ruin-Schritt wird eine zufällige Anzahl an Kunden ausgewählt, die daraufhin aus ihren Touren entfernt werden und in einem „Rucksack“ B gespeichert werden. Dabei gibt es verschiedene Möglichkeiten, diese Kunden auszuwählen. Beim *Radial Ruin* wird ein zufälliger Kunde c der N Kunden ausgewählt. Daraufhin wird eine Zufallszahl $A \leq F \cdot N$ ermittelt, wobei F eine Zahl zwischen 0 und 1 ist. Nun werden c und die $A - 1$ bezüglich einer Metrik nähesten Kunden aus ihren Touren gelöscht und in B gespeichert. Beim *Random Ruin* wird zuerst eine Zufallszahl $A \leq F \cdot N$ ausgewählt. Dann werden A zufällige Kunden aus ihren Touren entfernt und in B gespeichert. Beim *Sequential Ruin* werden

$A \leq F \cdot N$ aufeinander folgende Kunden eines zufälligen Laufs durch die Touren entfernt und in B gespeichert.

Bei der Wahl der Metrik für Radial Ruin gibt es ebenfalls mehrere Möglichkeiten. Die naheliegendste Methode ist die *space deletion*, auf der das Beispiel in Abb. 5.1 basiert. Dabei wird zufällig ein Kunde c und die $A-1$ bezüglich Distanz nähesten Kunden ausgewählt. Diese werden daraufhin entfernt. Alternativ könnte man die Kunden auch durch die Ankunftszeit des beliefernden Fahrzeugs auswählen, was *time deletion* genannt wird. Eine weitere Methode wäre *weight deletion* oder *volume deletion*, wobei die Kunden anhand der zu beliefernden Menge ausgewählt werden.

Für meinen Algorithmus habe ich eine alternierende Folge von Radial Ruin mit space deletion und $F = 0.3$ sowie Random Ruin mit $F = 0.5$ gewählt, siehe Alg. 12.

Algorithmus 12: ruin(List<Tour> tours, int run)

```
1  N = Anzahl Kunden in Touren
2  B = Liste der entfernten Aufträge
3  if run (mod 2) == 1 then Radial Ruin
4      F = 0.3
5      A =random(F * N)
6      c = Zufälliger Kunde
7      Entferne c aus seiner Tour
8      Speichere den zu c gehörigen Auftrag in B
9      for i = 1 to A − 1 do
10         c' = Verplanter Kunde mit minimaler Distanz zu c
11         Entferne c' aus seiner Tour
12         Speichere den zu c' gehörigen Auftrag in B
13 else Random Ruin
14     F = 0.5
15     A =random(F * N)
16     for int i = 1 to A do
17         c = Zufälliger Kunde
18         Entferne c aus seiner Tour
19         Speichere den zu c gehörigen Auftrag in B
20 return B
```

5.3. Recreate-Schritt

Nach dem Ruin-Schritt müssen die entfernten Kunden B nach und nach wieder in Touren eingefügt werden. Eine naheliegende Methode dafür wäre, für jeden Kunden $c \in B$ die Tour auszuwählen, bei der durch das Einfügen die geringsten Kosten entstehen. Diese Methode ist bereits durch den Insertion-Algorithmus bekannt und kann fast genauso

übernommen werden. Der Unterschied hier ist nur, dass wir nicht für eine gegebene Tour den besten einzufügenden Auftrag, sondern für einen gegebenen Auftrag die beste Tour zum Einfügen suchen. Wir entnehmen also in zufälliger Reihenfolge alle Kunden aus c aus B und berechnen für jede Tour und jede kompatible Einfügeposition der Tour die Kosten, die durch das Einfügen von c entstehen. Durch die zahlreichen Restriktionen (wie z.B. Zeitfenster) ist es allerdings möglich, dass keine mögliche Einfügeposition existiert. In diesem Fall muss für c eine neue Tour angelegt werden. Die Vorgehensweise wird in Alg. 13 beschrieben.

Algorithmus 13: recreate(List<Tour> tours, List<Order> B)

```
1 while B ≠ ∅ do
2     Entferne zufälligen Auftrag c aus B
3     foreach tour ∈ tours do
          // siehe Abschnitt 4.3.1
4         Berechne beste zulässige Einfügeposition (sofern existent)
          // siehe Abschnitt 4.3.2
5         Berechne Ersparnisfunktion c2 für Einfügepositon
6     if Es existiert eine Tour mit zulässiger Einfügeposition then
7         Füge c in die Tour mit größter Ersparnis c2 ein
8     else
9         Füge tours eine triviale Tour „Depot - Kunde - Depot“ mit dem billigsten
          verfügbaren Fahrzeug hinzu
```

5.4. Verbesserung des Algorithmus

Um den Algorithmus zu testen, habe ich wieder die von PASS gegebene Testinstanz mit 513 Aufträgen verwendet. Bei 500 Iterationen betrug die Gesamtlaufzeit ca. 8 Minuten. Allerdings wurden dadurch die Gesamtkosten der Touren nur um durchschnittlich 0.3% gesenkt, weshalb ich versucht habe, durch Änderungen der Parameter bessere Ergebnisse zu erzielen.

Die größte Änderung dabei ist, den Ruin & Recreate Algorithmus nicht einmal mit einer großen Anzahl an Iterationen, sondern mehrfach mit weniger Iterationen durchzuführen. So kann der Algorithmus mit der gleichen Laufzeit zum Beispiel 10 mal mit nur 50 Iterationen gestartet werden. Dabei wird nach jedem Neustart die aktuelle Konfiguration auf die bisher beste ermittelte Lösung und der Schwellwert auf seinen Anfangswert gesetzt. Dass dies sinnvoll sein kann, zeigt Abb. 5.3.

In Abb. 5.4 sieht man, wie sich die Kostenreduzierung durch Ruin & Recreate abhängig von der Anzahl der Durchläufe und der Anzahl der inneren Iterationen verhält, wobei das Produkt immer ≈ 500 ergibt. Dabei habe ich über jeweils 10 Nachbearbeitungen das beste, das durchschnittliche und das schlechteste Ergebnis dargestellt. Hier fällt auf, dass man

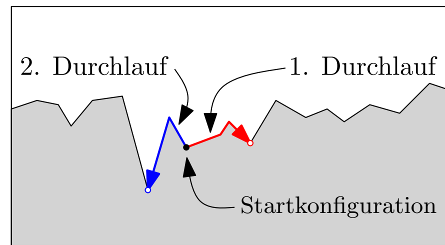


Abb. 5.3.: Zwei Durchläufe mit Ruin & Recreate

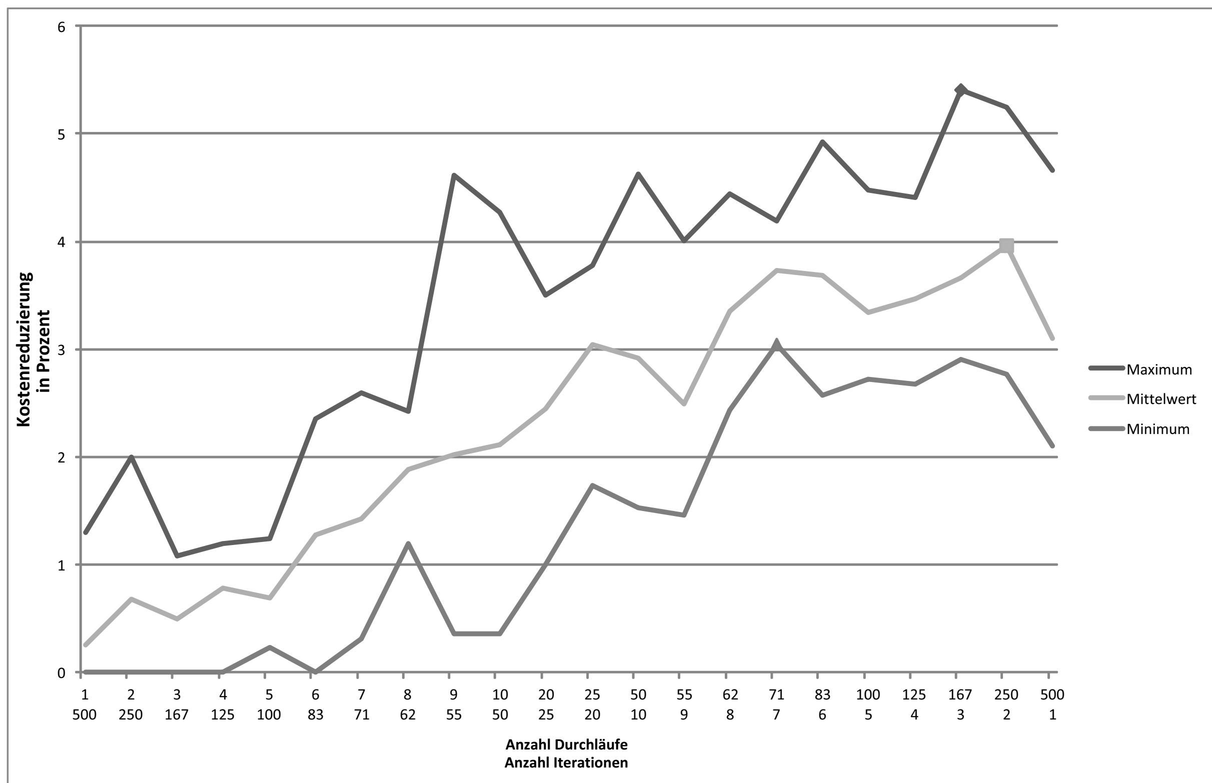


Abb. 5.4.: Verhältnis Anzahl Durchläufe zu Anzahl Iterationen

mit vielen kleinen Durchläufen deutlich bessere Ergebnisse erzielt, als mit einem großen Durchlauf. Das beste Ergebnis (dargestellt als Karo) wurde dabei mit 167 Durchläufen von 3 Iterationen mit 5.41% erzielt, der höchste Minimalwert (dargestellt als Dreieck) wurde bei 71 Durchläufen von 7 Iterationen mit 3.04% erreicht und das beste durchschnittliche Ergebnis (dargestellt als Quadrat) erlangte ich mit 250 Durchläufen von 2 Iterationen mit 3.96%. Für eben dieses Verhältnis erhielt ich auch den höchsten Median, das zweithöchste Maximum und das dritthöchste Minimum. Daher wählte ich als Standardwerte jeweils 2 Iterationen.

Interessant ist ebenfalls, wie sich die Anzahl der Durchläufe auf die Güte der Nachbearbeitung auswirkt. Hierzu habe ich jeweils 25 mal Ruin & Recreate mit unterschiedlicher Anzahl an Durchläufen gestartet. Die Anzahl der Iterationen habe ich fest auf 2 gesetzt, die Anzahl der Durchläufe waren Vielfache von 100, angefangen von 100 bis zu 600 Durch-

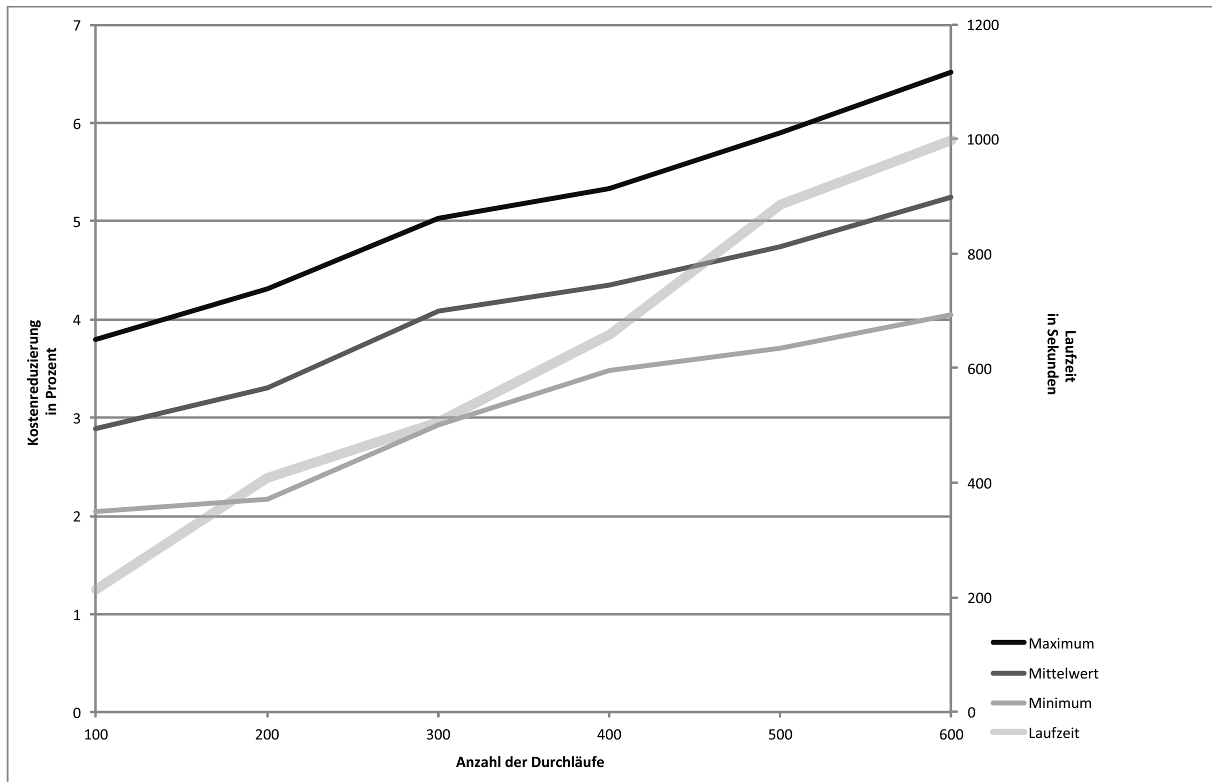


Abb. 5.5.: Anzahl Durchläufe

läufen (siehe Abb. 5.5). Dabei fällt auf, dass die Kostenreduzierung stetig mit der Anzahl der Durchläufe steigt. Ebenfalls erkennt man, dass die Laufzeit des Algorithmus linear abhängig von der Anzahl der Durchläufe ist.

Schrimpf et al. haben für den anfänglichen Schwellwert die Hälfte der Standardabweichung während eines Zufallslaufs durch zulässige Lösungen gewählt. Offen blieb allerdings, wie gut diese Wahl ist. Aus diesem Grund habe ich mehrere Möglichkeiten ausprobiert und miteinander verglichen. Weil die Auswirkung des anfänglichen Schwellwerts bei nur 2 Iterationen sehr gering ist, wählte ich für diesen Test 25 Durchläufe mit jeweils 20 Iterationen (siehe Abb. 5.6). Für jede Wahl des anfänglichen Schwellwerts habe ich dabei wieder 10 mal die Nachbearbeitung durchführen lassen und den Mittelwert gebildet. Dabei fällt auf, dass es zumindest für diesen Datenfall bessere Möglichkeiten gibt, den anfänglichen Schwellwert zu berechnen. Das Ergebnis wurde deutlich besser, wenn die Wahl des anfänglichen Schwellwerts nicht von einem Zufallslauf, sondern von den anfänglichen Kosten abhängt.

Um herauszufinden, welcher Bruchteil der Anfangskosten am sinnvollsten ist, habe ich dafür ebenfalls einen Test durchgeführt. Mit den gleichen Parametern habe ich als Anfangswert jeweils einen Bruchteil der Gesamtkosten der Ausgangslösung gewählt und das Maximum, den Mittelwert und das Minimum der Kostenreduzierung dargestellt. Zum Vergleich habe ich ebenfalls die Quadratwurzel der Ausgangslösung eingefügt (siehe Abb. 5.7). Dabei stellte sich als bester Wert für den anfänglichen Schwellwert 1/10 der Gesamtkos-

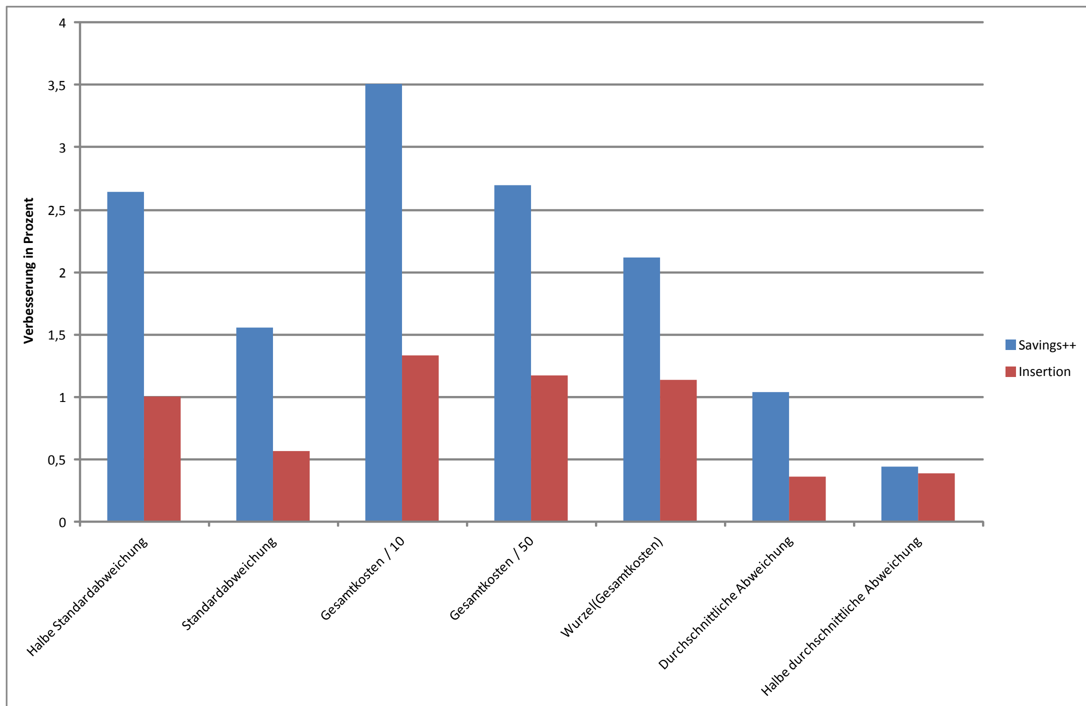


Abb. 5.6.: Anfänglicher Schwellwert

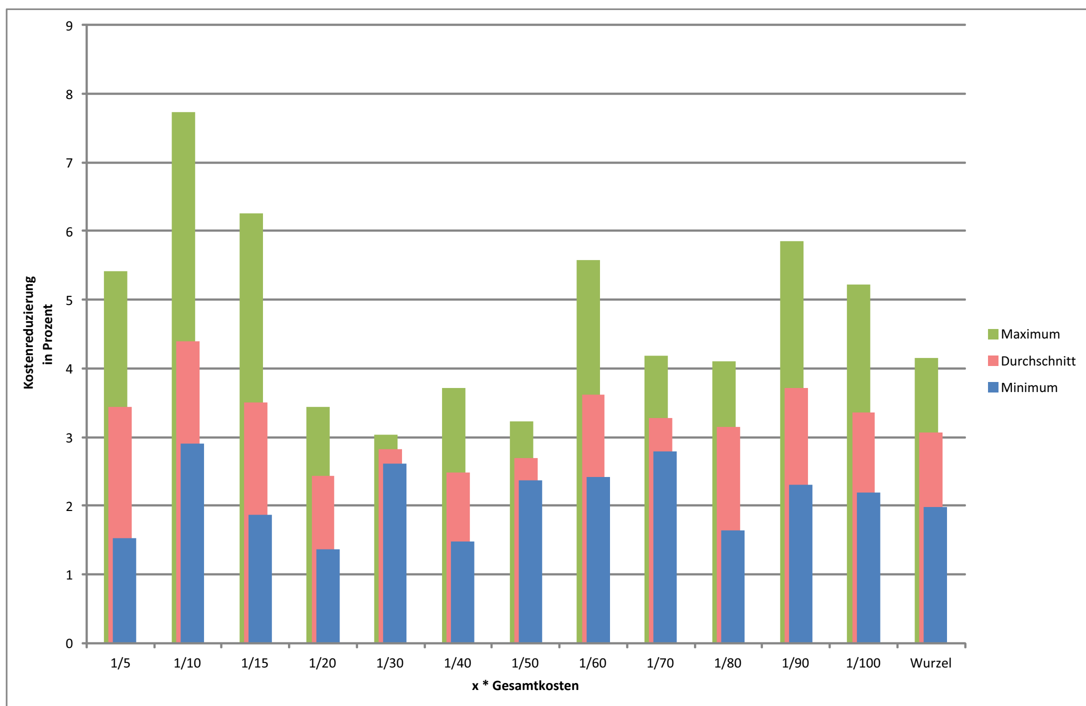


Abb. 5.7.: Anfänglicher Schwellwert in Abhängigkeit von Gesamtkosten

ten der Ausgangslösung heraus. Allerdings wurden diese Werte nur auf einem Datenfall getestet, so dass dies im Allgemeinfall nicht die beste Lösung sein muss.

6. Optimale Lösung

Von Fisher und Jaikumar [FJ81] wurde 1981 eine Drei-Index Fluss Formulierung für das VRPTW ohne Warte- und Stoppzeiten entwickelt. Dabei wird jedem Kunden ein Index $i = 1, \ldots, \#\text{Aufträge}$, jedem Fahrzeug ein Index $k = 1, \ldots, \#\text{Fahrzeuge}$ und dem Depot der Index 0 zugewiesen. Eine binäre Variable $x_{ijk} (i \neq j)$ gibt dann an, ob die Strecke von Kunde/Depot i zu Kunde/Depot j von Fahrzeug k gefahren wird (1 für ja, 0 für nein). Der Hauptunterschied zu den meisten anderen Formulierungen liegt darin, dass die Variablen ebenfalls das benutzte Fahrzeug repräsentieren, während sonst im Allgemeinen nur eine Binärvariable x_{ij} für die Strecke von i nach j existiert. Im Folgenden möchte ich die klassische Formulierung des Ganzzahligen Linearen Programms vorstellen und dann die Anpassungen, die für die zusätzlichen Restriktionen von Nöten sind, beschreiben.

6.1. Klassische Formulierung

Zunächst müssen einige Variablen definiert werden. Kunden und Depots bilden die Menge V der *Knoten*. Eine *Kante* ist der Weg zwischen zwei Punkten. Die Menge aller Kanten wird wird mit E bezeichnet. Es gibt zwei Typen von binären und einen Typ von ganzzahligen Variablen. Die binäre Variable $x_{ijk} (i \neq j)$ ist genau dann 1, wenn in der durch die Variablen beschriebenen zulässigen Lösung die Kante (i, j) durch das Fahrzeug k befahren wird. Die binäre Variable y_{ik} ist genau dann 1, wenn in der optimalen Lösung das Fahrzeug k den Knoten i anfährt. Die Variable b_i entspricht der Ankunftszeit beim Kunden i. Außerdem müssen mehrere Konstanten definiert werden. Die Kosten einer Kante (i, j) beschreibt die Konstante c_{ij}, die Fahrzeit von Knoten i zu j entspricht der Konstanten t_{ij}. Das Zeitfenster für einen Knoten i wird definiert als $[e_i, l_i]$, die Ankunftszeit als b_i. Sei D_k die Kapazität des Fahrzeugs k, d_i die auszuliefernde Menge für Kunde i und T eine sehr große Zahl. Außerdem sei $n = \#\text{Kunden}$ und $m = \#\text{Fahrzeuge}$.

Nun kann das Problem definiert werden. Die Zielfunktion ist dabei die Summe der Kosten aller Kanten, die von einem Fahrzeug befahren werden:

$$\text{minimiere} \sum_{k=1}^{m} \sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} c_{ij} x_{ijk} \tag{0}$$

Um sicherzustellen, dass die Kapazitäten der Fahrzeuge nicht überschritten werden, wird für jedes Fahrzeug die Summe der auszuliefernden Menge d_i für alle belieferten

Kunden ($y_{ik} = 1$) mit der Fahrzeugkapazität D_k verglichen:

$$\sum_{i=0}^{n} d_i y_{ik} \leq D_k \quad (k = 1, \ldots, m) \tag{1}$$

Weil jede Tour im Depot startet, muss jedes Fahrzeug einmal dort gewesen sein:

$$\sum_{k=1}^{m} y_{0k} = m \tag{2}$$

Jeder Kunde muss genau einmal beliefert werden, es muss also jeder Kunde von genau einem Fahrzeug angefahren werden:

$$\sum_{k=1}^{m} y_{ik} = 1 \quad (i = 1, \ldots, n) \tag{3}$$

Wenn ein Fahrzeug k die Kante (i, j) benutzt, muss es folglich sowohl den Knoten i, als auch den Knoten j anfahren. Zusätzlich darf es zu jedem Knoten nur eine eingehende und eine ausgehende Kante geben, ein Fahrzeug darf also nicht mehrfach zu dem gleichen Kunden fahren:

$$\sum_{i=0}^{n} x_{ijk} = y_{jk} \quad (j = 0, \ldots, n;\ k = 1, \ldots, m) \tag{4}$$

$$\sum_{j=0}^{n} x_{ijk} = y_{ik} \quad (i = 0, \ldots, n;\ k = 1, \ldots, m) \tag{5}$$

Jede Tour besteht aus einem einfachen Kreis mit η Knoten und η Kanten, der das Depot 0 enthält. Entfernen wir also den Knoten 0 aus der Tour, so besteht sie nur noch aus einem einfachen Weg von $\eta - 1$ Knoten und $\eta - 2$ Kanten. Um sicherzustellen, dass keine Kreise existieren, die nicht durch das Depot gehen, dürfen für jede Knotenteilmenge $S \subset V \backslash 0$ nur maximal $|S| - 1$ Kanten existieren:

$$\sum_{i,j \in S} x_{ijk} \leq |S| - 1 \quad (S \subset V \backslash 0;\ |S| \geq 2;\ k = 1, \ldots, m) \tag{6}$$

Angenommen, ein Fahrzeug k fährt den Knoten i zum Zeitpunkt b_i an und danach die Kante (i, j) entlang zum Knoten j. Dann muss das Fahrzeug zum Zeitpunkt $b_j = b_i + t_{ij}$ beim Kunden j ankommen. Dabei sei angemerkt, dass

$$(1 - x_{ijk})T = \begin{cases} T & x_{ijk} = 0 \\ 0 & x_{ijk} = 1 \end{cases},$$

der Constraint also nur „greift“, wenn $x_{ijk} = 1$.

$$b_j \begin{cases} \geq b_i + t_{ij} - (1 - x_{ijk})T \\ \leq b_i + t_{ij} + (1 - x_{ijk})T \end{cases} \quad (i, j = 0, \ldots, n;\ k = 1, \ldots, m) \tag{7}$$

Ein Fahrzeug darf nur innerhalb des angegebenen Zeitfensters bei einem Knoten ankommen:

$$e_i \leq b_i \leq l_i \quad (i = 1, \ldots, n) \tag{8}$$

Die Variablen x_{ijk} und y_{ik} sind Binärvariablen, können also nur die Werte 1 („ja") und 0 („nein") annehmen:

$$x_{ijk} \in \{0, 1\} \quad (i, j = 0, \ldots, n;\ k = 1, \ldots, m) \tag{9}$$

$$y_{ik} \in \{0, 1\} \quad (i = 1, \ldots, n;\ k = 1, \ldots, m) \tag{10}$$

6.2. Erweiterung

Das gerade vorgestellte Verfahren funktioniert nur bei einfachen Problemen, in denen die einzigen Restriktionen Zeitfenster und Kapazitäten sind. Es gilt nun, die Formulierung so anzupassen, dass alle von PASS gegebenen Restriktionen erfüllt werden. Dabei gehen wir davon aus, dass die eingegebenen Daten bereits vorverarbeitet wurden (siehe Abschnitt 2), also keine passiven Aufträge oder Fahrzeuge mehr existieren und die Zeitfenster der Kunden bereits richtig berechnet wurden. Ich werde nun nacheinander auf die zusätzlichen Restriktionen eingehen und beschreiben, wie man diese in die Formulierung einbauen kann.

6.2.1. Allgemein

- Fahrzeugspezifische Kosten:
  Die Kosten für den Weg von einem Knoten i zu einem Knoten j sind nicht nur von der Distanz und Konstanten, sondern auch von dem Fahrzeug abhängig, das diesen Weg fährt. Es genügt also nicht, für jede Kante (i, j) eine Kostenkonstante c_{ij} zu haben, wir müssen für die Kosten ebenfalls einen dritten Index k einfügen, der sich auf das benutzte Fahrzeug bezieht. Somit bestimmt die Konstante c_{ijk} die Kosten, die entstehen, wenn das Fahrzeug k die Kante (i, j) fährt. Die Zielfunktion ändert sich somit zu

$$\text{minimiere} \sum_{k=1}^{m} \sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} c_{ijk} x_{ijk} \tag{0'}$$

- Wartezeiten:
  Es ist erlaubt, dass ein Fahrzeug vor dem angegebenen Zeitfenster bei einem Kunden i ankommt. Dadurch entsteht beim Kunden eine Wartezeit w_{ik}. Diese erhöht auch die Kosten der Tour. Weil die zusätzlichen Kosten linear zur Wartezeit sind, gibt es für jedes Fahrzeug k eine Konstante c_k^w, so dass $c_k^w \cdot w_{ik}$ den Kosten entspricht, die durch eine Wartezeit w_{ik} beim Kunden i entstehen. Dementsprechend muss die

Zielfunktion um Wartezeiten erweitert werden:

$$\text{minimiere} \sum_{k=1}^{m}\sum_{i=0}^{n}\left(c_k^w \cdot w_{jk} + \sum_{\substack{j=0\\j\neq i}}^{n} c_{ijk}x_{ijk}\right) \tag{0''}$$

Ebenfalls müssen die Nebenbedingungen angepasst werden, die sich auf die Ankunftszeit beziehen:

$$b_j \begin{cases} \geq b_i + w_{ik} + t_{ij} - (1 - x_{ijk})T \\ \leq b_i + w_{ik} + t_{ij} + (1 - x_{ijk})T \end{cases} \quad (i, j = 0, \ldots, n;\ k = 1, \ldots, m) \tag{7'}$$

$$e_i \leq b_i + \sum_{k=1}^{m} w_{ik} \leq l_i \quad (i = 1, \ldots, n) \tag{8'}$$

Die Wartezeiten dürfen nicht negativ sein, also muss eine zusätzliche Nebenbedingung eingefügt werden:

$$w_{ik} \geq 0 \quad (i = 0, \ldots, n;\ k = 1, \ldots, m) \tag{11}$$

- Zeitfenster beachten:
  Mit dieser Option kann die Beachtung der Zeitfenster ausgeschalten werden. Dies ist eine Konstante atw, wobei

  $$atw = \begin{cases} 1 & \text{falls Zeitfenster beachtet werden müssen} \\ 0 & \text{falls nicht} \end{cases}.$$

  Um diese Option in die Formeln einzubinden, muss man einfach atw auf die Nebenbedingungen aufmultiplizieren, die die Beachtung der Zeitfenster sicherstellt:

  $$atw \cdot e_i \leq atw \cdot b_i + \sum_{k=1}^{m} w_{ik} \leq atw \cdot l_i \quad (i = 1, \ldots, n) \tag{8''}$$

- Anzahl Touren minimieren:
  Bei der klassischen Formulierung wird für jedes gegebene Fahrzeug eine Tour gestartet. Weil wir die Anzahl der Fahrzeuge minimieren wollen, werden auch weniger als m Touren erlaubt. Also muss zunächst eine Nebenbedingung angepasst werden:

  $$\sum_{k=1}^{m} y_{0k} \leq m \tag{2'}$$

  Außerdem muss die Zielfunktion so erweitert werden, dass Touren-Festkosten berücksichtigt werden. Durch die gegebenen Daten sind für jedes Fahrzeug die festen Kosten pro Tour bekannt. Diese werden als Konstanten ctp_k übergeben. Diese Kosten sind auf die Zielfunktion aufzuaddieren:

  $$\text{minimiere} \sum_{k=1}^{m}\left(y_{0k} \cdot cpt_k + \sum_{i=0}^{n}\left(c_k^w \cdot w_{jk} + \sum_{\substack{j=0\\j\neq i}}^{n} c_{ijk}x_{ijk}\right)\right) \tag{0'''}$$

6.2.2. Fahrzeuge

Folgende zusätzliche Restriktionen an die Fahrzeuge sind zu beachten:

- Maximalmenge, -gewicht, -volumen:
  Für jedes Fahrzeug ist eine Obergrenze der drei Kapazitäten statt einer angegeben. Die Kapazitätskonstanten müssen also erweitert werden: Für ein Fahrzeug k sind die Kapazitäten gegeben durch D_k^a (Menge), D_k^w (Gewicht) und D_k^v. Für einen Kunden i werden dementsprechend die benötigten Kapazitäten definiert durch d_i^a, d_i^w und d_i^v. Die Nebenbedingung (1) für Kapazitäten muss daher auch in drei Nebenbedingungen aufgespalten werden:

$$(\sum_{i=0}^{n} d_i^a y_{ik} \leq D_k^a \quad (k = 1, \ldots, m) \tag{1a}$$

$$\sum_{i=0}^{n} d_i^w y_{ik} \leq D_k^w \quad (k = 1, \ldots, m) \tag{1b}$$

$$\sum_{i=0}^{n} d_i^v y_{ik} \leq D_k^v \quad (k = 1, \ldots, m) \tag{1c}$$

- Fahrzeuggruppe:
  Es gibt verschiedene Gruppen von Fahrzeugen, wobei manche Aufträge nur von einer bestimmten Fahrzeuggruppe ausgeführt werden können. Optional können diese Gruppen auch hierarchisch angeordnet sein. Die Fahrzeuggruppenhierarchie hat zur Folge, dass zum Beispiel alle Umschlagstellen mit Fahrzeuggruppe 3 auch von den Fahrzeuggruppen 1 und 2 angefahren werden dürfen. Die Beachtung der Fahrzeuggruppen wird durch die Konstante vh präsentiert, wobei $vh = 1$, wenn Fahrzeuggruppen beachtet werden sollen, und $vh = 0$, wenn Fahrzeuggruppen nicht beachtet werden sollen. Die Hierarchiemethode wird durch eine Konstante vgh präsentiert, die wir definieren als

$$vgh = \begin{cases} 0 & \text{Keine Hierarchie} \\ 1 & \text{Aufsteigend} \\ 2 & \text{Absteigend} \end{cases} .$$

  Außerdem wird für jedes Fahrzeug k eine Fahrzeuggruppe $vg_k \geq 1$ und für jeden Kunden i eine benötigte Fahrzeuggruppe $cg_i \geq 0$ gespeichert, wobei $cg_i = 0$ bedeutet, dass jede Fahrzeuggruppe diesen Kunden anfahren darf. Nun muss sichergestellt werden, dass ein Kunde i nur von einem Fahrzeug k angefahren werden darf, wenn die Fahrzeuggruppen kompatibel sind:

$$y_{ik} \cdot vh \cdot (1 - vgh) \cdot (2 - vgh) \cdot cg_i = y_{ik} \cdot vh \cdot (1 - vgh) \cdot (2 - vgh) \cdot cg_i \cdot vg_k \tag{12a}$$

$$(i = 1, \ldots, n;\ k = 1, \ldots, m)$$

$$y_{ik} \cdot vh \cdot vgh \cdot (2 - vgh) \cdot cg_i \geq y_{ik} \cdot vh \cdot vgh \cdot (2 - vgh) \cdot cg_i \cdot vg_k \quad (i = 1, \ldots, n;\ k = 1, \ldots, m) \tag{12b}$$

$$y_{ik} \cdot vh \cdot vgh \cdot (1 - vhg) \cdot cg_i \leq y_{ik} \cdot vh \cdot vgh \cdot (1 - vhg) \cdot cg_i \cdot vg_k \quad (i = 1, \ldots, n;\ k = 1, \ldots, m) \tag{12c}$$

Dabei sei angemerkt, dass alle drei Formeln immer erfüllt sind, wenn $y_{ik} = 0$, $vh = 0$ oder $cg_i = 0$ ist. Die erste Formel kann nur bei $vgh = 0$, die zweite nur bei $vgh = 1$ und die dritte nur bei $vgh = 2$ nicht erfüllt sein.

- Zusätzliche Fahrzeugeigenschaften und Personal-Restriktionen:
  Es können für die Fahrzeuge zusätzliche Eigenschaften angegeben werden, z.B. eine Hebebühne. In den Aufträgen wird dann deklariert, welche Eigenschaften das Fahrzeug erfüllen muss, das diesen Auftrag ausführt. Ebenso können für das Personal Eigenschaften angegeben werden, die erfüllt sein müssen, z.B. ein Waffenschein. Weil jedem Fahrzeug festes Personal zugewiesen ist, lassen sich die Eigenschaften des Personals zu den Eigenschaften des Fahrzeugs hinzufügen. Sei nun o die Anzahl der Eigenschaften. Dann definieren wir für jeden Kunden i eine binäre Konstante vrc_{il} $(l = 1, \ldots, o)$, wobei

$$vrc_{il} = \begin{cases} 1 & \text{falls das Fahrzeug die Eigenschaft } l \text{ besitzen muss} \\ 0 & \text{falls nicht} \end{cases}.$$

  Analog definieren wir für jedes Fahrzeug k eine binäre Konstante vrv_{kl} $(l = 1, \ldots, o)$, wobei

$$vrv_{kl} = \begin{cases} 1 & \text{falls das Fahrzeug die Eigenschaft } l \text{ besitzt} \\ 0 & \text{falls nicht} \end{cases}.$$

  Nun muss sichergestellt werden, dass jedes Fahrzeug, das einen Auftrag beliefert, alle zusätzlichen Eigenschaften besitzt. Dazu formulieren wir folgende Nebenbedingung:

$$y_{ik} \cdot vrc_{il} = y_{ik} \cdot vrc_{il} \cdot vrv_{kl} \quad (i = 1, \ldots, n; k = 1, \ldots, m; l = 1, \ldots, o) \tag{12}$$

  So wird sichergestellt, dass wenn ein Fahrzeug einen Kunden anfährt und dieser eine gewisse Eigenschaft des Fahrzeugs oder des dazugehörigen Personals fordert, diese auch erfüllt sein muss.

6.2.3. Aufträge

Im Folgenden werden die zusätzlichen Eigenschaften aufgezählt, die jeder Auftrag besitzt und die vom Algorithmus berücksichtigt werden müssen.

- Fahrzeugnummer:
  Man kann Aufträgen über die Fahrzeugnummer feste Fahrzeuge zuweisen. Diese dürfen dann nur von diesem Fahrzeug ausgeführt werden. Dafür wird für jeden Auftrag i eine Konstante vn_i gespeichert, wobei

$$vn_i = \begin{cases} 0 & \text{falls keine Fahrzeugnummer existiert} \\ k & \text{falls Fahrzeug } k \text{ der Fahrzeugnummer entspricht} \end{cases}.$$

Nun muss sicher gestellt werden, dass für $vn_i > 0$ y_{ik} genau dann 1 ist, wenn $k = vn_i$:

$$y_{ik} \cdot vn_i \cdot k = y_{ik} \cdot vn_i \cdot vn_i \tag{13}$$

Diese Nebenbedingung ist nicht erfüllt, wenn $y_{ik} = 1$ für $vn_i > 0$ und $k \neq vn_i$. Wegen Nebenbedingung (3) muss $y_{ik} = 1$ aber für ein k erfüllt sein, wodurch $y_{i,vn_i} = 1$ sein muss.

- Standzeit:
  Für jeden Auftrag wird eine feste Zeit eingeplant, die zum Entladen der Ware nötig ist. Wir definieren also für jeden Kunden i eine Standzeit s_i. Diese muss bei der Berechnung der Ankunftszeit beim nächsten Kunden mit einbezogen werden. Also muss die Nebenbedingung für Ankunftszeiten angepasst werden:

$$b_j \begin{cases} \geq b_i + w_{ik} + s_i + t_{ij} - (1 - x_{ijk})T \\ \leq b_i + w_{ik} + s_i + t_{ij} + (1 - x_{ijk})T \end{cases} \quad (i, j = 0, \ldots, n;\ k = 1, \ldots, m) \tag{7''}$$

6.2.4. Touren

Im Folgenden werden die Eigenschaften aufgezählt, die jede Tour besitzen muss, die vom Algorithmus gebildet wird.

- Maximale Wartezeit:
  Damit die Wartezeiten nicht Überhand nehmen, kann eine maximale Wartezeit pro Tour eingestellt werden. Dies ist eine Konstante mwt. Für jedes Fahrzeug k muss nun die Summe aller Wartezeiten $\leq mwt$ sein:

$$\sum_{i=1}^{n} w_{ik} \leq mwt \quad (k = 1, \ldots, m) \tag{14}$$

- Maximale Anzahl Tourposten:
  Es kann eine maximale Anzahl an Aufträgen eingestellt werden, die pro Tour verplant werden dürfen. Dies ist eine Konstante ct, wobei für $ct = 0$ die maximale Anzahl Tourposten unbegrenzt ist. Für jedes Fahrzeug k muss nun die Anzahl der belieferten Kunden $\leq ct$ sein:

$$ct \cdot \sum_{i=1}^{n} y_{ik} \leq ct \cdot ct \quad (k = 1, \ldots, m) \tag{15}$$

- Einfädelzeit:
  Die Einfädelzeit gibt an, wie lange ein Fahrzeug braucht, um vom Kunden wieder in den Verkehr zu finden. So werden alle Strecken fix um diese Zeit verlängert. Die ist eine Konstante at. Diese muss bei der Berechnung der Ankunftszeit beim nächsten Kunden mit einbezogen werden. Also muss die Nebenbedingung für Ankunftszeiten

angepasst werden:

$$b_j \begin{cases} \geq b_i + w_{ik} + s_i + at + t_{ij} - (1 - x_{ijk})T \\ \leq b_i + w_{ik} + s_i + at + t_{ij} + (1 - x_{ijk})T \end{cases} \qquad (i,j = 0,\dots,n;\ k = 1,\dots,m) \tag{7'''}$$

- Maximale Tourdauer:
  Ebenso kann die Gesamtdauer der Tour begrenzt werden. Die maximale Tourdauer darf dabei vom Algorithmus nicht überschritten werden. Dies ist eine Konstante mtl, wobei für $mtl = 0$ die maximale Tourdauer unbegrenzt ist. Die Tourdauer setzt sich dabei zusammen aus Fahrzeiten, Wartezeiten, Standzeiten und Einfädelzeiten. Damit kein Fahrzeug länger als mtl unterwegs ist, muss eine weitere Nebenbedingung eingefügt werden:

$$mtl \cdot \sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} x_{ijk} t_{ij} + \sum_{i=1}^{n} (w_{ik} + y_{ik} \cdot (s_i + at)) \leq mtl \cdot mtl \quad (k = 1,\dots,m) \tag{16}$$

- Verkehrsflussfaktor:
  Hiermit kann Einfluss auf die Dauer der Strecken genommen werden. Es wird ein Prozentsatz angegeben, der bestimmt, wie schnell das Fahrzeug vorwärts kommt. Dies ist eine Konstante tf. Diese Konstante muss auf alle Fahrzeiten aufmultipliziert werden, wodurch sich zwei Nebenbedingungen ändern:

$$b_j \begin{cases} \geq b_i + w_{ik} + s_i + at + tf \cdot t_{ij} - (1 - x_{ijk})T \\ \leq b_i + w_{ik} + s_i + at + tf \cdot t_{ij} + (1 - x_{ijk})T \end{cases} \qquad (i,j = 0,\dots,n;\ k = 1,\dots,m) \tag{7''''}$$

$$mtl \cdot \left(\sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} x_{ijk} \cdot tf \cdot t_{ij} + \sum_{i=1}^{n} (w_{ik} + y_{ik} \cdot (s_i + at)) \right) \leq mtl \cdot mtl \quad (k = 1,\dots,m) \tag{17'}$$

- Abweichung Öffnung und Schließung:
  Hierdurch kann eine Zeit bestimmt werden, die ein Fahrzeug außerhalb der Zeitfenster beim Kunden eintreffen darf. Dies sind zwei Konstanten: de entspricht der Abweichung von der Öffnungszeit, dl der Abweichung von der Schließzeit. de muss dabei von der Öffnungszeit abgezogen und dl zu der Schließzeit addiert werden. Dadurch ändert sich eine Nebenbedingung:

$$atw \cdot e_i - de \leq atw \cdot b_i + \sum_{k=1}^{m} w_{ik} \leq atw \cdot l_i + dl \quad (i = 1,\dots,n) \tag{8'''}$$

- Entladung in der Öffnungszeit:
  Es kann zusätzlich eingestellt werden, dass ein Fahrzeug nicht nur innerhalb der gegebenen Zeitfenster beim Kunden eintreffen, sondern auch wieder bei ihm abfahren

muss. Dies entspricht einer Konstanten dis, wobei

$$dis = \begin{cases} 1 & \text{falls innerhalb der Öffnungszeit beim Kunden abgefahren werden muss} \\ 0 & \text{falls nicht} \end{cases}$$

Also muss im Fall $dis = 1$ die Abfahrtszeit ebenfalls innerhalb der Zeitfenster liegen. Die Nebenbedingung (8)$^{(3)}$ muss dafür in zwei Nebenbedingungen aufgespalten werden:

$$atw \cdot e_i - de \leq atw \cdot b_i + \sum_{k=1}^{m} w_{ik} \quad (i = 1, \ldots, n) \tag{8a'''}$$

$$atw \cdot b_i + \sum_{k=1}^{m} w_{ik} + dis \cdot s_i \leq atw \cdot l_i + dl \quad (i = 1, \ldots, n) \tag{8b'''}$$

- Frühester Tourstart:
  Hierdurch wird die frühestmögliche Startzeit für alle Touren festgelegt. Eine Tour darf nicht vor dieser Uhrzeit im Depot starten. Dies ist eine Konstante ets. Der Tourstart ts_k für ein Fahrzeug k muss dabei zunächst aus Ankunftszeit beim ersten Kunden und Fahrzeit zu diesem Kunden berechnen. Dafür muss eine neue Nebenbedingung eingefügt werden:

$$ts_k \begin{cases} \geq b_j - tf \cdot t_{0j} - (1 - x_{0jk}) \cdot T \\ \leq b_j - tf \cdot t_{0j} + (1 - x_{0jk}) \cdot T \end{cases} \quad (k = 1, \ldots, m;\ j = 1, \ldots, n) \tag{18a}$$

  Ebenso muss eine Nebenbedingung für den frühesten Tourstart eingebaut werden:

$$ets \leq ts_k \quad (k = 1, \ldots, m) \tag{18b}$$

- Abfahrt innerhalb:
  Hierdurch wird festgelegt, innerhalb welcher Zeit ab dem frühesten Tourstart die Fahrzeuge im Depot starten müssen. Dies ist eine Konstante di. Zur Beachtung muss eine Nebenbedingung erweitert werden:

$$ets \leq ts_k \leq ets + di \quad (k = 1, \ldots, m) \tag{18b'}$$

6.3. Eliminierung von Subtouren

Zur Eliminierung von Subtouren existiert folgende Nebenbedigung:

$$\sum_{i,j \in S} x_{ijk} \leq |S| - 1 \quad (S \subset V \backslash 0;\ |S| \geq 2;\ k = 1, \ldots, m) \tag{6}$$

Für n Kunden und m Fahrzeuge entspricht das aber $O(m \cdot 2^n)$ Nebenbedigungen. Selbst wenn das Hinzufügen jeder Nebenbedingung nur eine Nanosekunde benötigte, würden bei

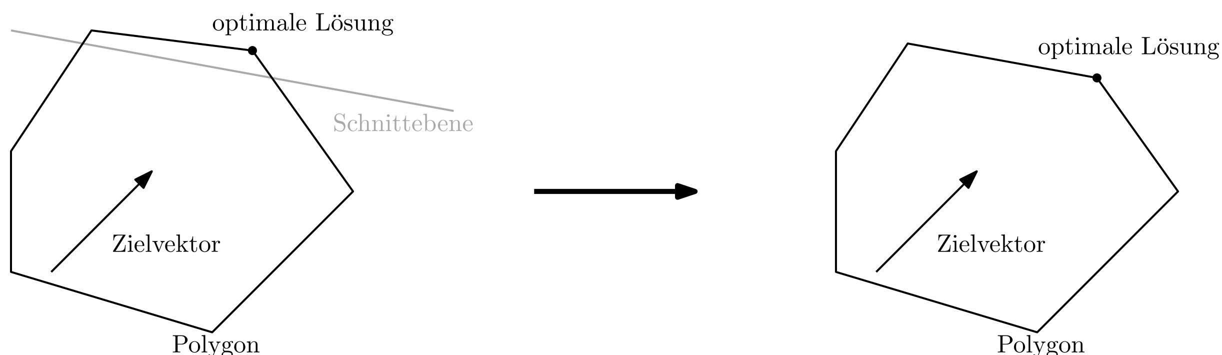


Abb. 6.1.: Einfügen einer Schnittebene in bereits existierende Lösung

100 Kunden bereits dafür mehr als $4,34 \cdot 10^13$ Jahre benötigt werden (zum Vergleich: die Erde existiert seit circa $4,6 \cdot 10^9$ Jahren). Es ist also im Allgemeinen nicht möglich, diese Nebenbedingungen in das Programm einzufügen.

Dieses Problem wird durch das Schnittebenenverfahren gelöst. Dieses ist ursprünglich eine Methode, ganzzahlige lineare Programme mit Hilfe des Simplex-Verfahrens zu lösen, lässt sich aber auch auf diesen Fall anwenden. Die Idee dabei ist wie folgt: Zunächst wird das lineare Programm ohne Eliminierung von Subtouren gestartet. Die Menge der zulässigen Lösungen entspricht dabei einem Polytop, die optimale Lösung einer Ecke des Polytops. Wurde nun die optimale Lösung gefunden, so wird sie auf Subtouren überprüft. Eine Subtour ist dabei ein einfacher Kreis $\langle C_{i_0}, C_{i_1}, \ldots, C_{i_l}, C_{i_0} \rangle \quad (i_0, \ldots, i_l \in \{1, \ldots, n\}$, der nicht durch das Depot verläuft. Existiert in der Lösung für ein Fahrzeug k ein solche Kreis, so ist offensichtlich $x_{i_0 i_1 k} + x_{i_1 i_2 k} + \cdots + x_{i_{l-1} i_l k} + x_{i_l i_0 k} = l$, also $\sum_{i,j \in S} x_{ijk} = l + 1 = |S| > |S| - 1$ für $S = \{i_0, \ldots, i_l\}$. Somit ist die Nebenbedingung (6) nicht erfüllt. Nun wird für eben diese Subtour dem bereits bestehenden Programm eine Nebenbedingung hinzugefügt und die Berechnung weitergeführt. Damit die neue optimale Lösung nicht den gleichen Kreis mit einem anderen Fahrzeug enthält, wird die Nebenbedingung für jedes Fahrzeug erstellt:

$$\sum_{i,j \in S} x_{ijk} = l + 1 = |S| > |S| - 1 \quad (S = \{i_0, \ldots, i_l\};\ k = 1, \ldots, m$$

In Abbildung 6.1 wird illustriert, wie das Einfügen einer Schnittebene in einem zweidimensionalen linearen Programm aussieht.

Um Bereits im Vorhinein viele Subtouren auschließen zu können, werden m Nebenbedingungen eingeführt, die sicherstellen, dass jedes Fahrzeug, das einen Kunden beliefert, das Depot anfahren muss. So können Kreise außerhalb des Depots nur noch auftreten, wenn wenn für das gleiche Fahrzeug bereits eine Tour durch das Depot existiert. Dadurch wird die Anzahl der einzufügenden Schnittebenen deutlich reduziert.

$$y_{0k} \geq \frac{1}{n} \cdot \sum_{i=1}^{n} y_{ik} \quad (k = 1, \ldots, m) \tag{6b}$$

Eine vollständige Beschreibung des ganzzahligen linearen Programms ist im Anhang B nachzulesen.

7. Experimente

Um die Qualität meiner Algorithmen bewerten zu können, habe ich die Ergebnisse zum einen auf bekannten Beispielinstanzen von Solomon und zum anderen auf echten Datenfällen mit dem bereits bestehenden Algorithmus von PASS verglichen.

7.1. Beispielinstanzen

Von Solomon [Sol05] wurden diverse Benchmark-Instanzen für das euklidische VRPTW entwickelt, mit denen sich die Güte der zahlreichen Heuristiken gut bewerten lässt. Er erstellte dafür sechs verschiedene Problemmengen, die aus jeweils 100 Kunden bestehen. In der Problemmenge R sind die Koordinaten der Kunden dabei zufällig generiert, die Problemmenge C besteht aus Clustern (Bündel) von Kunden. Die dritte Problemmenge RC ist dabei eine Mischung aus beiden Varianten. Diese drei Mengen sind jeweils noch einmal in zwei Untermengen 1 und 2 untergliedert, wobei die erste Untermenge sehr enge Zeitfenster und geringe Fahrzeugkapazität besitzt, während diese in der zweiten Untermenge großzügiger gewählt sind. Von Solomon werden ebenfalls die besten bekannten Lösungen von Heuristiken sowie die bisher gefundenen optimalen Lösungen präsentiert. Spoorendonk [Spo07] hat die Liste der optimalen Lösungen erweitert, so dass nun für fast alle Probleme eine optimale Lösung bekannt ist. Die Abbildungen 7.1 – 7.6 zeigen jeweils eine sehr gute Lösung für eine ausgewählte Instanz jeder Problemmenge.

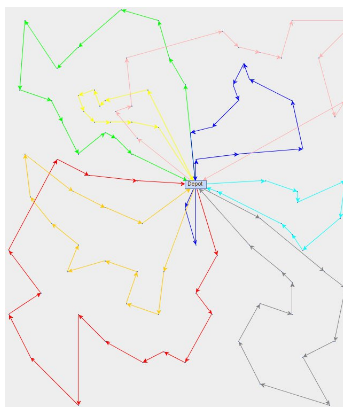


Abb. 7.1.: R104

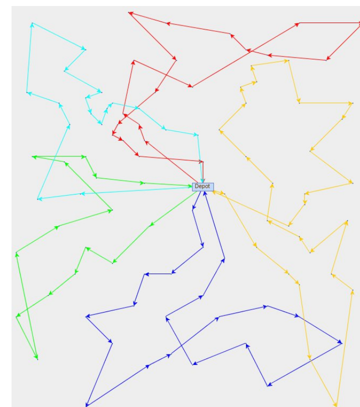


Abb. 7.2.: R206

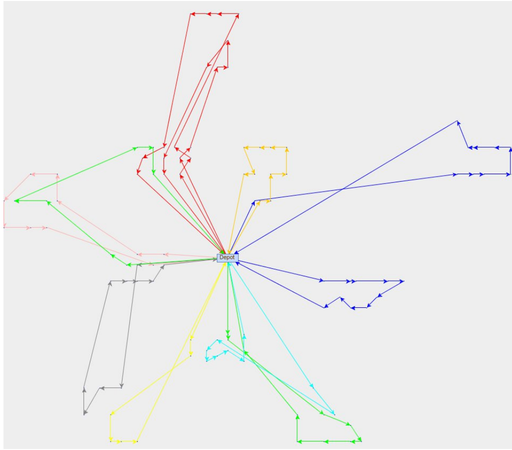


Abb. 7.3.: C109

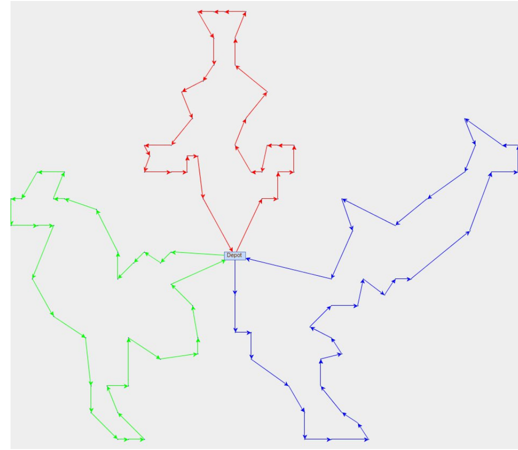


Abb. 7.4.: C206

Abb. 7.5.: RC107

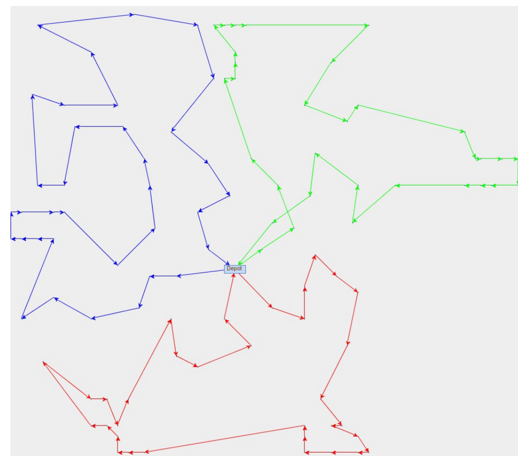
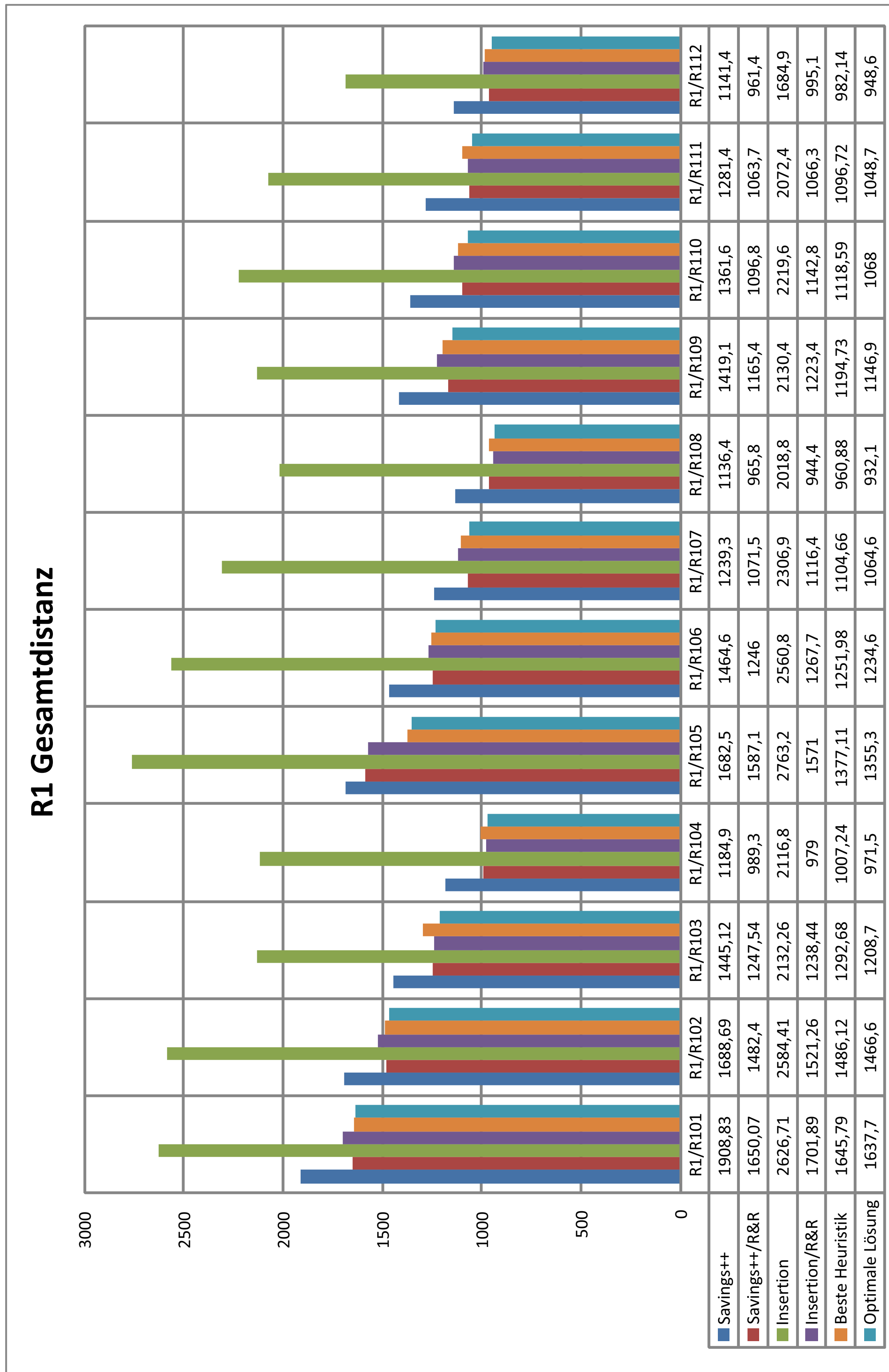


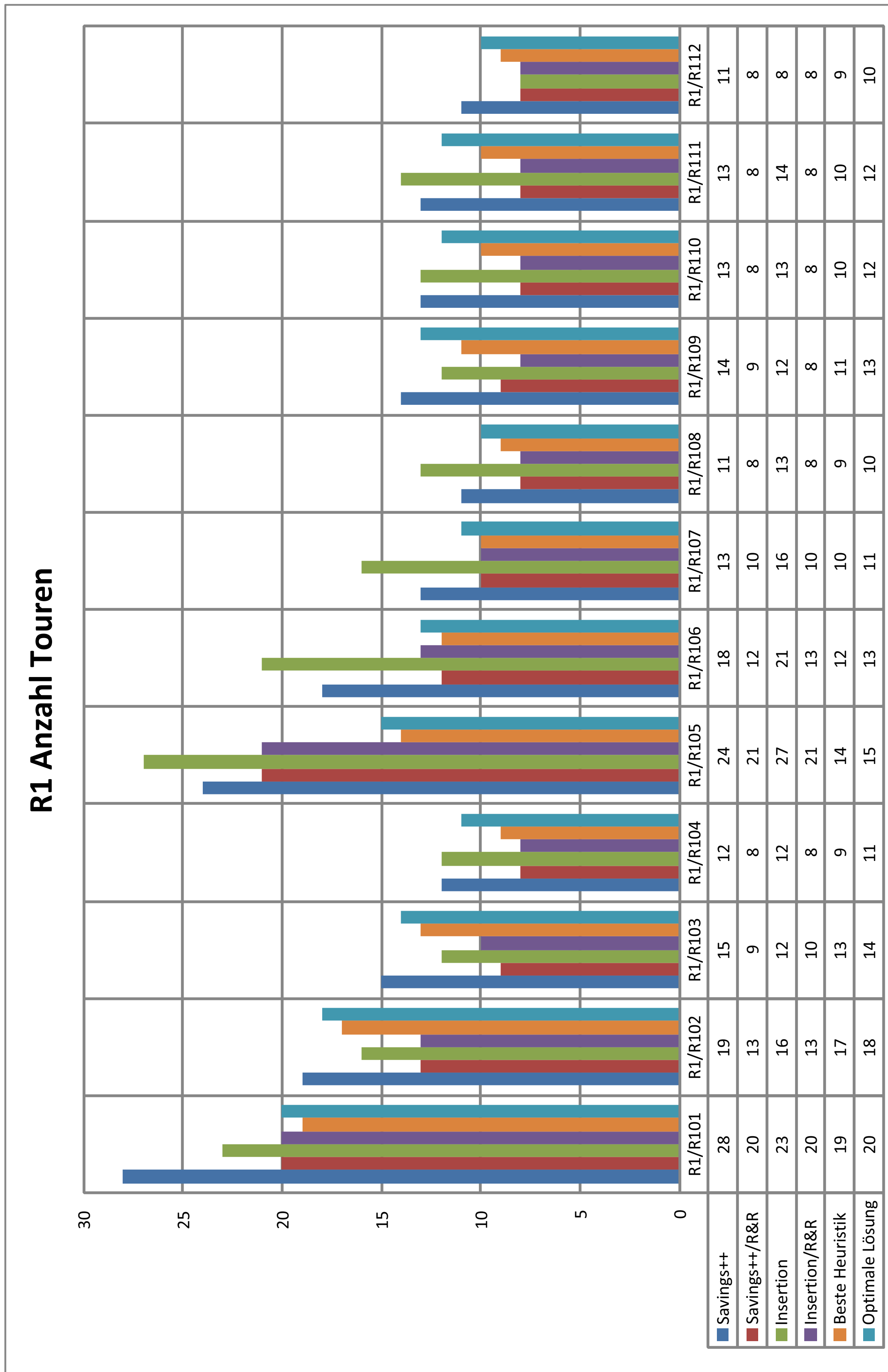
Abb. 7.6.: RC204

Im Folgenden werden die Ergebnisse meiner Algorithmen (mit und ohne Nachbearbeitung) mit den bekannten Lösungen verglichen (siehe Abb. 7.7 – 7.18). Auffällig ist, dass der Insertion-Algorithmus bezüglich Gesamtdistanz deutlich schlechter abschneidet als der Savings++-Algorithmus, während dieser im Allgemeinen mehr Touren benötigt, vor allem bei den Untermengen 2. Bei den meisten Instanzen ist die Lösung mit Nachbearbeitung allerdings schon sehr nahe an der optimalen Lösung. Die Gesamtdistanz ist in manchen Fällen sogar geringer als die beste bekannte heuristische Lösung. Die Werte der besten Heuristiken beziehen allerdings keine Nachbearbeitung mit ein.



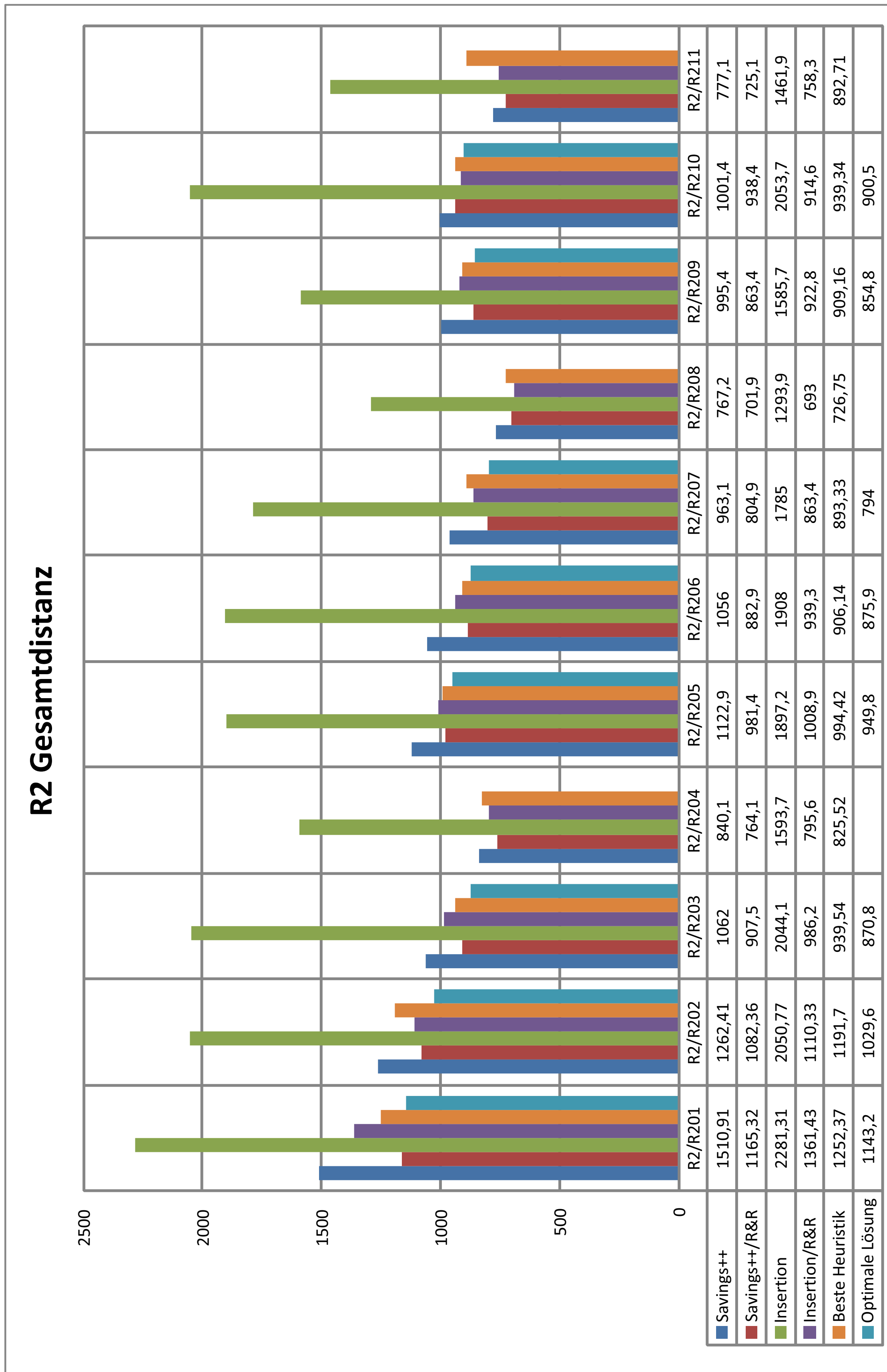
	R1/R101	R1/R102	R1/R103	R1/R104	R1/R105	R1/R106	R1/R107	R1/R108	R1/R109	R1/R110	R1/R111	R1/R112
Savings++	1908,83	1688,69	1445,12	1184,9	1682,5	1464,6	1239,3	1136,4	1419,1	1361,6	1281,4	1141,4
Savings++/R&R	1650,07	1482,4	1247,54	989,3	1587,1	1246	1071,5	965,8	1165,4	1096,8	1063,7	961,4
Insertion	2626,71	2584,41	2132,26	2116,8	2763,2	2560,8	2306,9	2018,8	2130,4	2219,6	2072,4	1684,9
Insertion/R&R	1701,89	1521,26	1238,44	979	1571	1267,7	1116,4	944,4	1223,4	1142,8	1066,3	995,1
Beste Heuristik	1645,79	1486,12	1292,68	1007,24	1377,11	1251,98	1104,66	960,88	1194,73	1118,59	1096,72	982,14
Optimale Lösung	1637,7	1466,6	1208,7	971,5	1355,3	1234,6	1064,6	932,1	1146,9	1068	1048,7	948,6

Abb. 7.7.: Gesamtdistanz R1



	R1/R101	R1/R102	R1/R103	R1/R104	R1/R105	R1/R106	R1/R107	R1/R108	R1/R109	R1/R110	R1/R111	R1/R112
Savings++	28	19	15	12	24	18	13	11	14	13	13	11
Savings++/R&R	20	13	9	8	21	12	10	8	9	8	8	8
Insertion	23	16	12	12	27	21	16	13	12	13	14	8
Insertion/R&R	20	13	10	8	21	13	10	8	8	8	8	8
Beste Heuristik	19	17	13	9	14	12	10	9	11	10	10	9
Optimale Lösung	20	18	14	11	15	13	11	10	13	12	12	10

Abb. 7.8.: Anzahl Touren R1



	R2/R201	R2/R202	R2/R203	R2/R204	R2/R205	R2/R206	R2/R207	R2/R208	R2/R209	R2/R210	R2/R211
Savings++	1510,91	1262,41	1062	840,1	1122,9	1056	963,1	767,2	995,4	1001,4	777,1
Savings++/R&R	1165,32	1082,36	907,5	764,1	981,4	882,9	804,9	701,9	863,4	938,4	725,1
Insertion	2281,31	2050,77	2044,1	1593,7	1897,2	1908	1785	1293,9	1585,7	2053,7	1461,9
Insertion/R&R	1361,43	1110,33	986,2	795,6	1008,9	939,3	863,4	693	922,8	914,6	758,3
Beste Heuristik	1252,37	1191,7	939,54	825,52	994,42	906,14	893,33	726,75	909,16	939,34	892,71
Optimale Lösung	1143,2	1029,6	870,8		949,8	875,9	794		854,8	900,5	

Abb. 7.9.: Gesamtdistanz R2

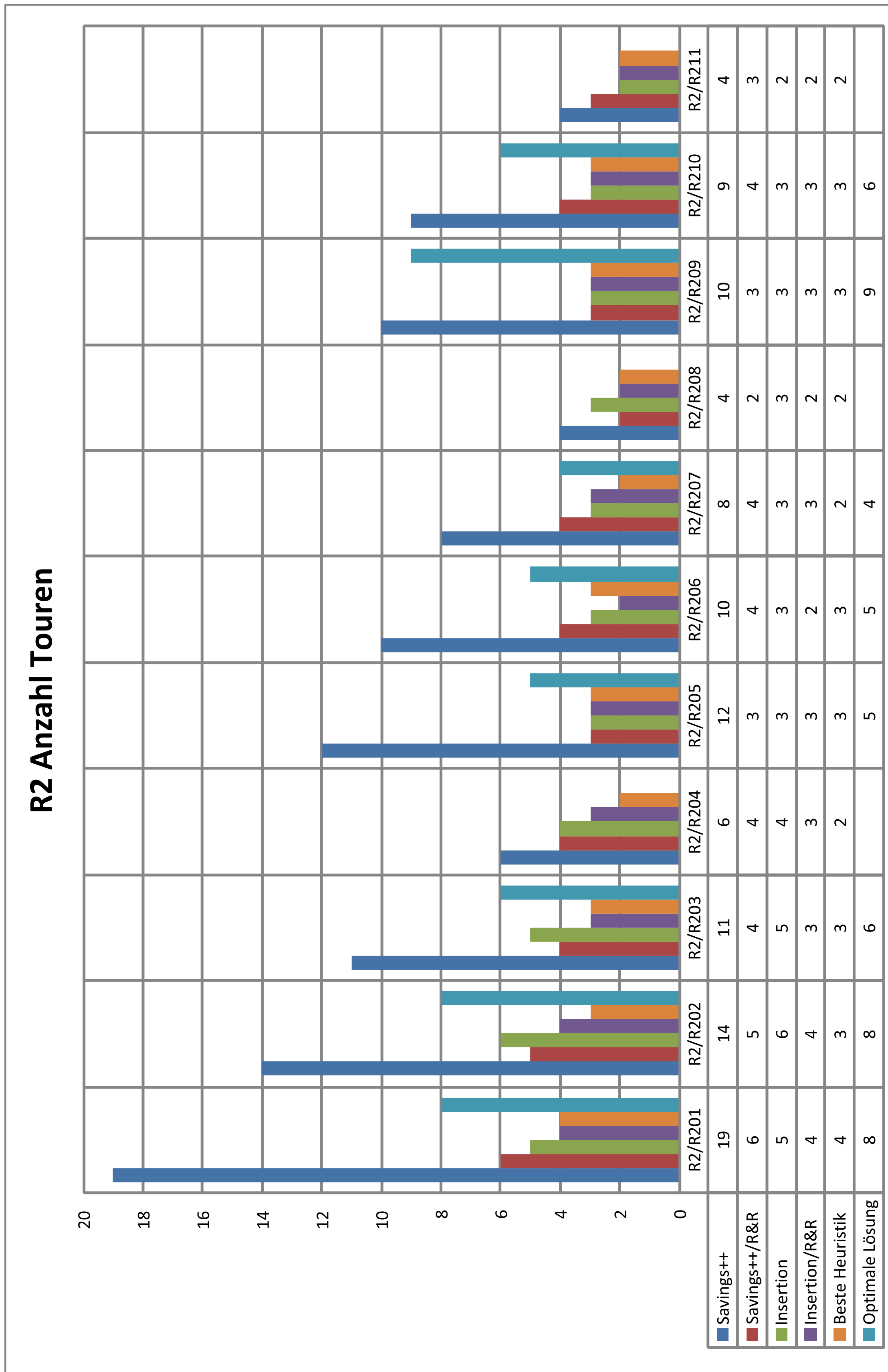
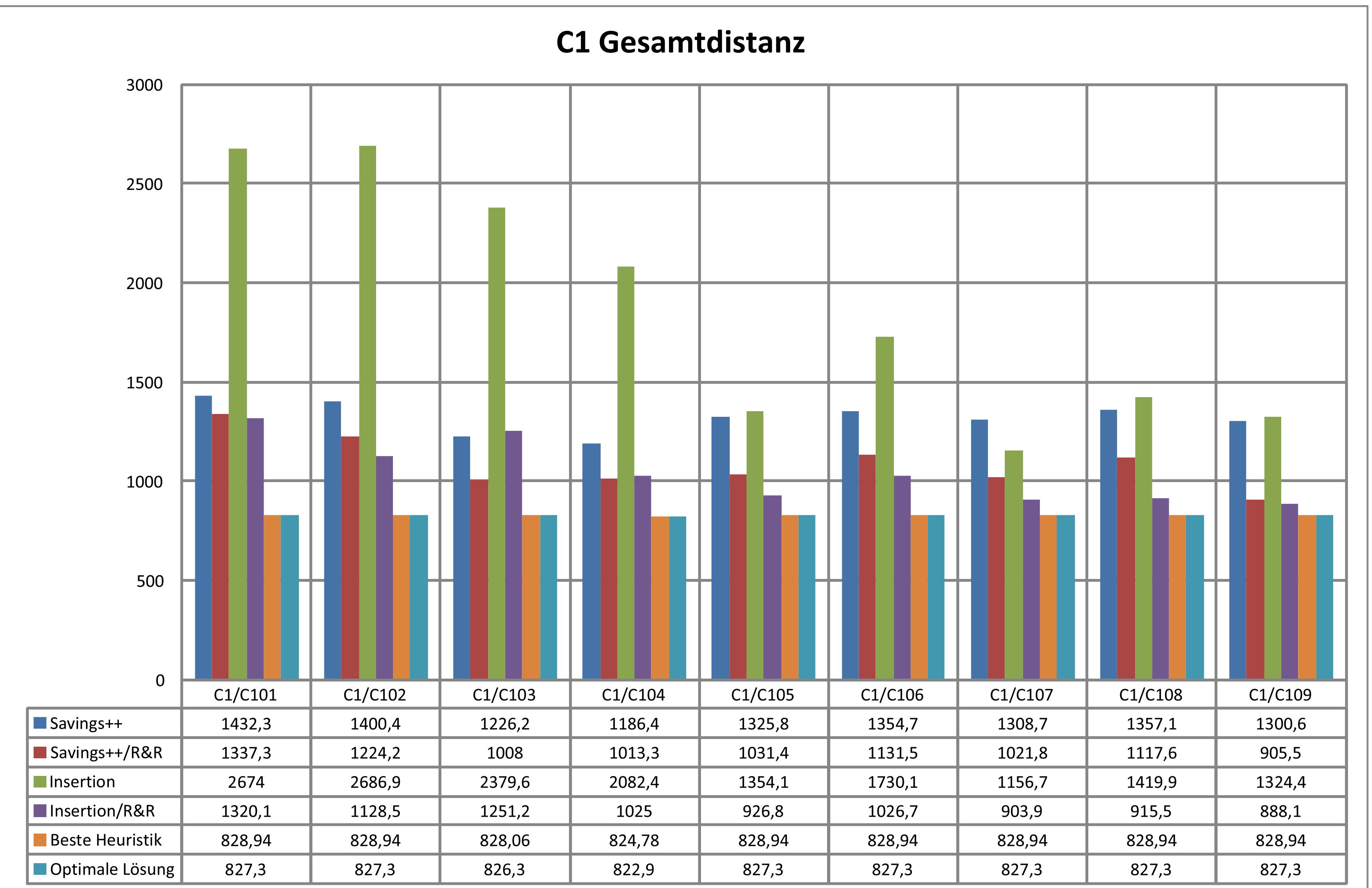


Abb. 7.10.: Anzahl Touren R2



	C1/C101	C1/C102	C1/C103	C1/C104	C1/C105	C1/C106	C1/C107	C1/C108	C1/C109
Savings++	1432,3	1400,4	1226,2	1186,4	1325,8	1354,7	1308,7	1357,1	1300,6
Savings++/R&R	1337,3	1224,2	1008	1013,3	1031,4	1131,5	1021,8	1117,6	905,5
Insertion	2674	2686,9	2379,6	2082,4	1354,1	1730,1	1156,7	1419,9	1324,4
Insertion/R&R	1320,1	1128,5	1251,2	1025	926,8	1026,7	903,9	915,5	888,1
Beste Heuristik	828,94	828,94	828,06	824,78	828,94	828,94	828,94	828,94	828,94
Optimale Lösung	827,3	827,3	826,3	822,9	827,3	827,3	827,3	827,3	827,3

Abb. 7.11.: Gesamtdistanz C1

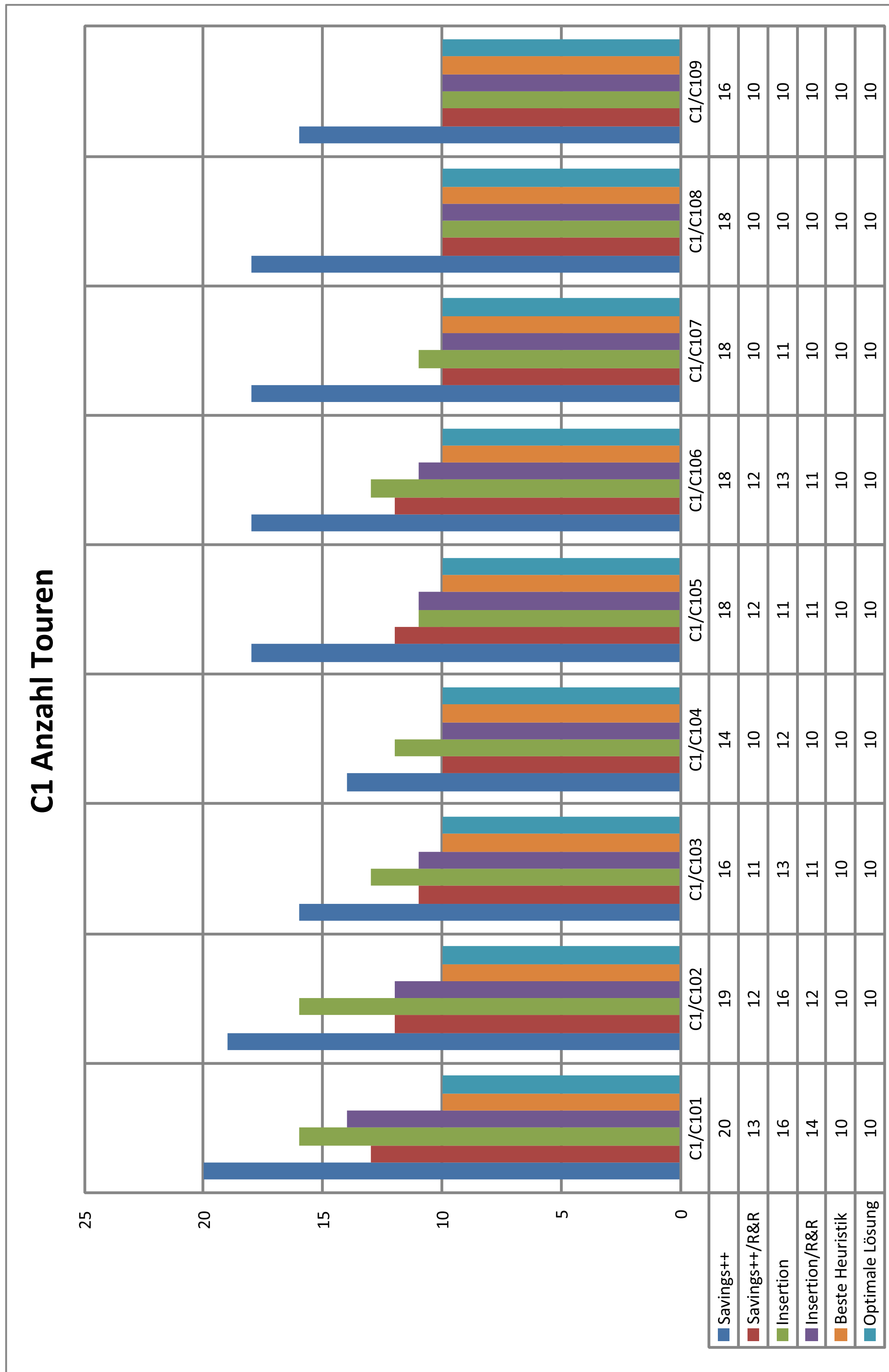
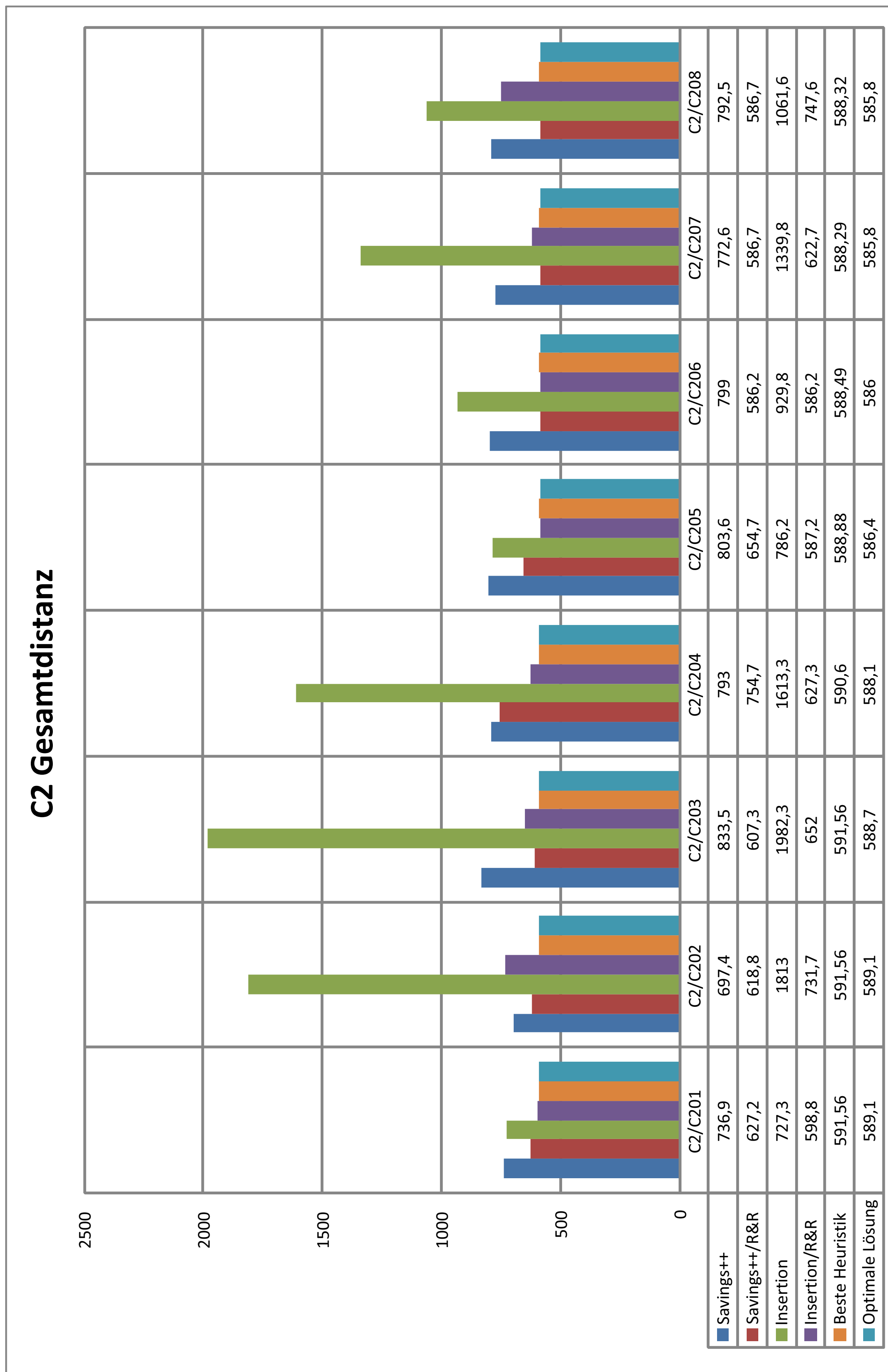


Abb. 7.12.: Anzahl Touren C1



	C2/C201	C2/C202	C2/C203	C2/C204	C2/C205	C2/C206	C2/C207	C2/C208
Savings++	736,9	697,4	833,5	793	803,6	799	772,6	792,5
Savings++/R&R	627,2	618,8	607,3	754,7	654,7	586,2	586,7	586,7
Insertion	727,3	1813	1982,3	1613,3	786,2	929,8	1339,8	1061,6
Insertion/R&R	598,8	731,7	652	627,3	587,2	586,2	622,7	747,6
Beste Heuristik	591,56	591,56	591,56	590,6	588,88	588,49	588,29	588,32
Optimale Lösung	589,1	589,1	588,7	588,1	586,4	586	585,8	585,8

Abb. 7.13.: Gesamtdistanz C2

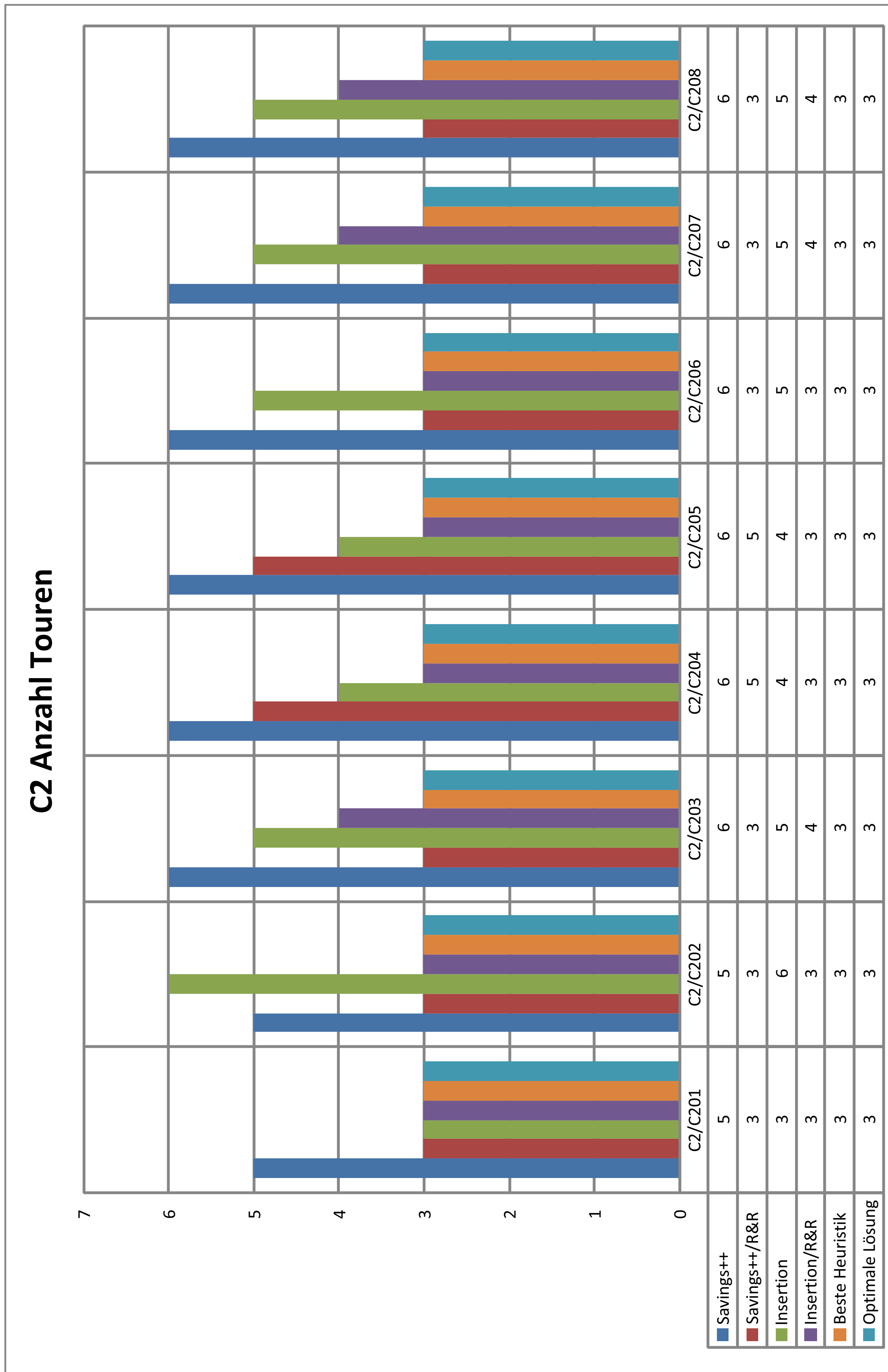
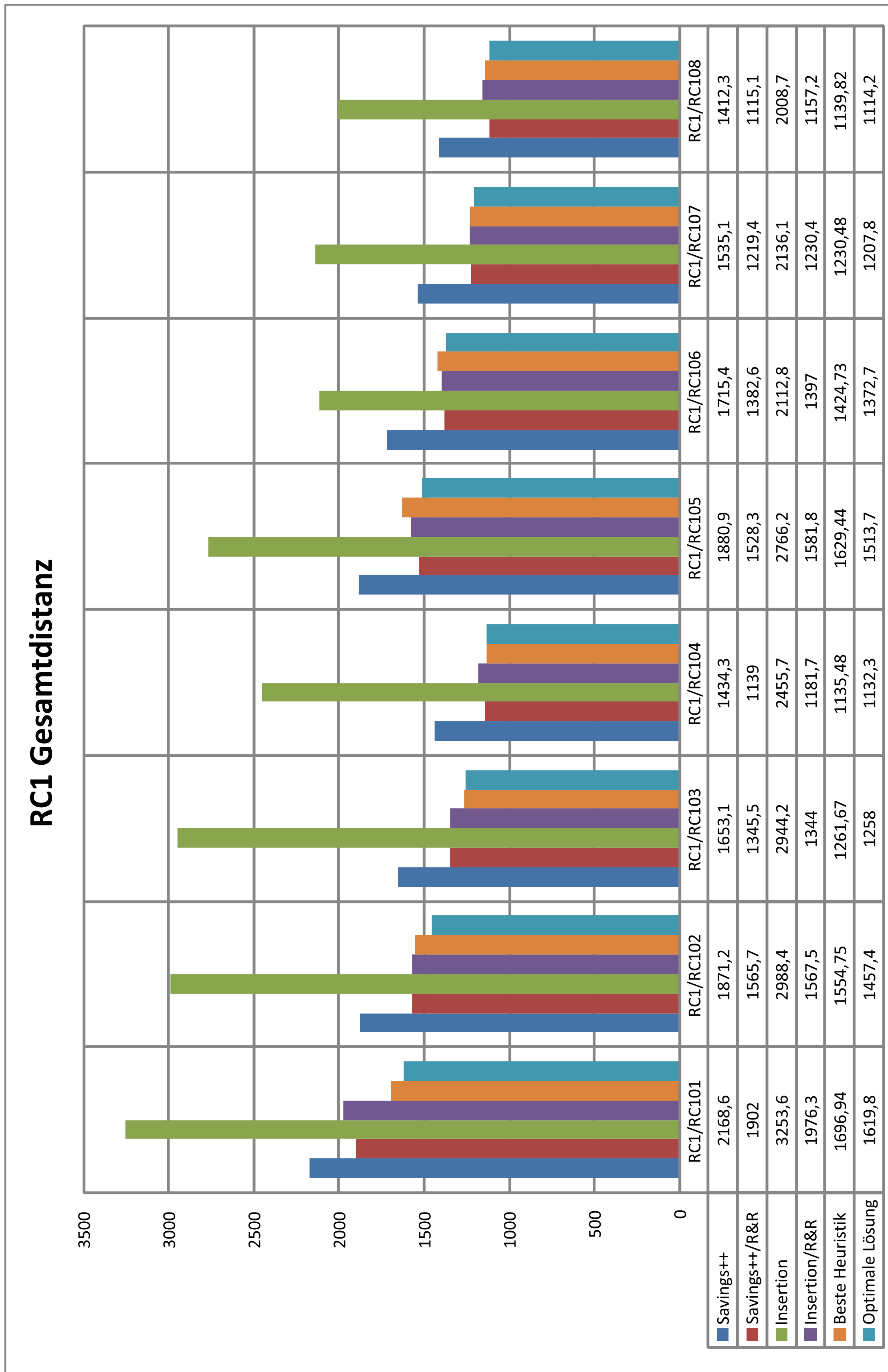


Abb. 7.14.: Anzahl Touren C2



	RC1/RC101	RC1/RC102	RC1/RC103	RC1/RC104	RC1/RC105	RC1/RC106	RC1/RC107	RC1/RC108
Savings++	2168,6	1871,2	1653,1	1434,3	1880,9	1715,4	1535,1	1412,3
Savings++/R&R	1902	1565,7	1345,5	1139	1528,3	1382,6	1219,4	1115,1
Insertion	3253,6	2988,4	2944,2	2455,7	2766,2	2112,8	2136,1	2008,7
Insertion/R&R	1976,3	1567,5	1344	1181,7	1581,8	1397	1230,4	1157,2
Beste Heuristik	1696,94	1554,75	1261,67	1135,48	1629,44	1424,73	1230,48	1139,82
Optimale Lösung	1619,8	1457,4	1258	1132,3	1513,7	1372,7	1207,8	1114,2

Abb. 7.15.: Gesamtdistanz RC1

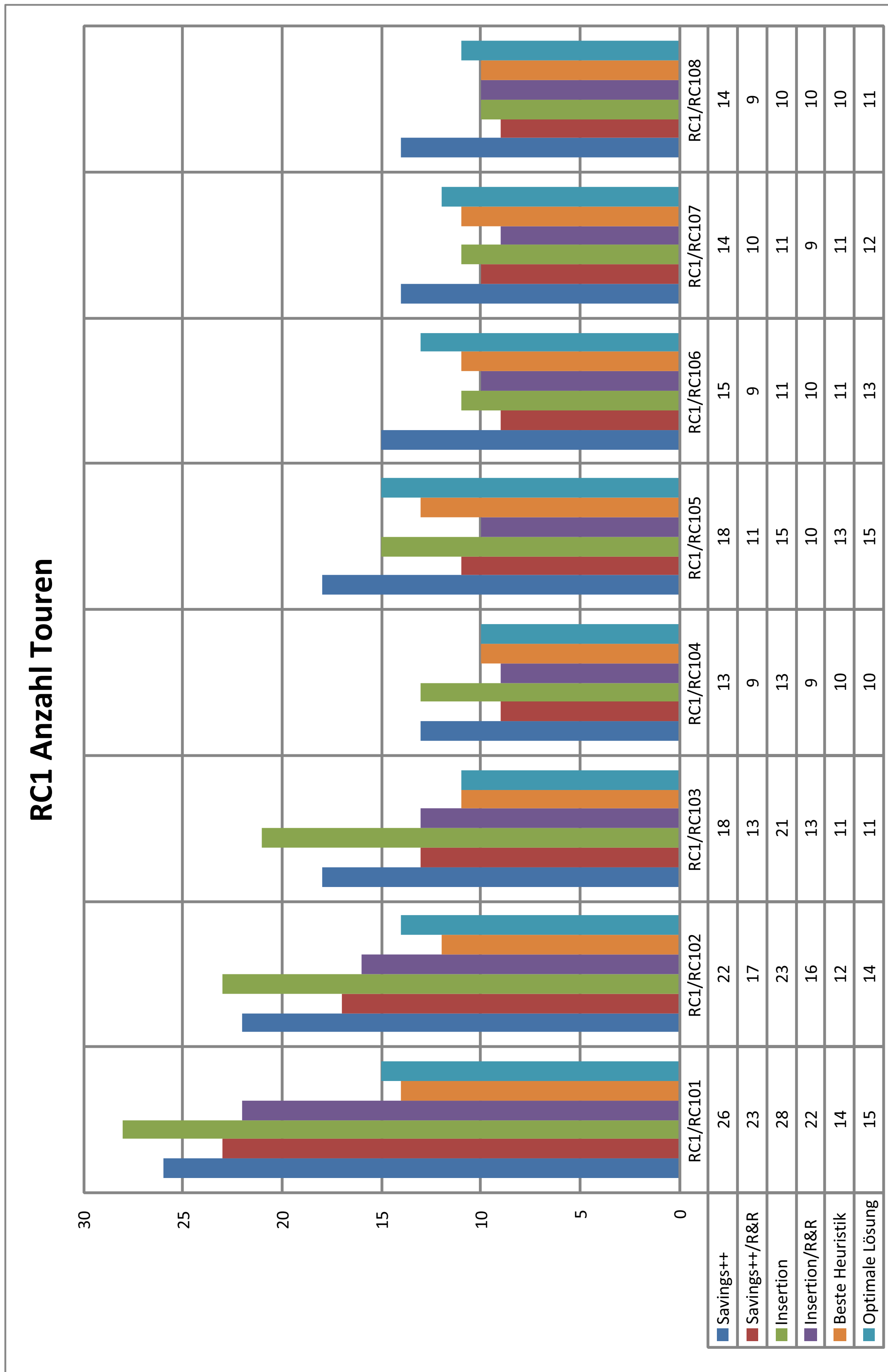


Abb. 7.16.: Anzahl Touren RC1

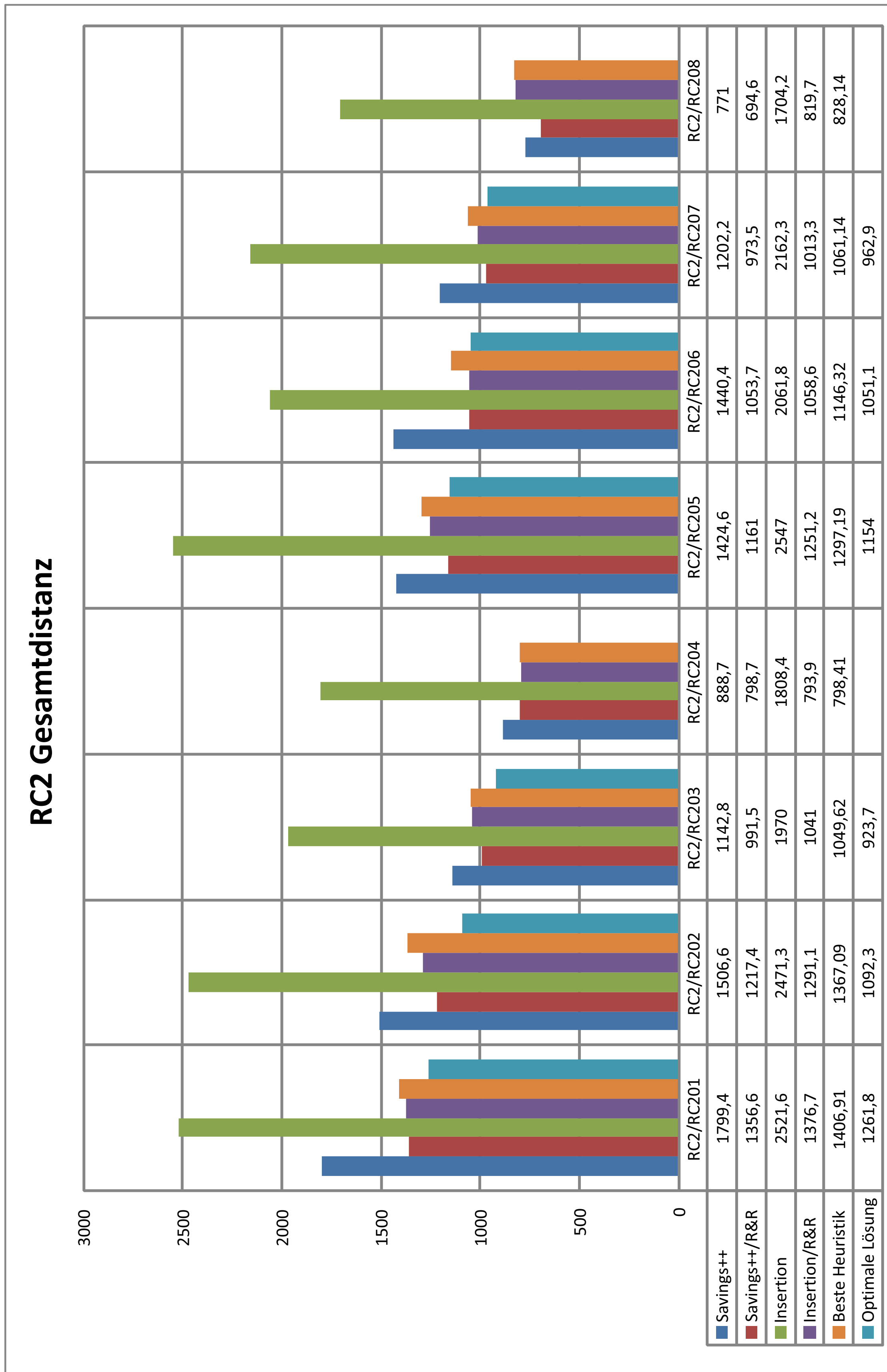


Abb. 7.17.: Gesamtdistanz RC2

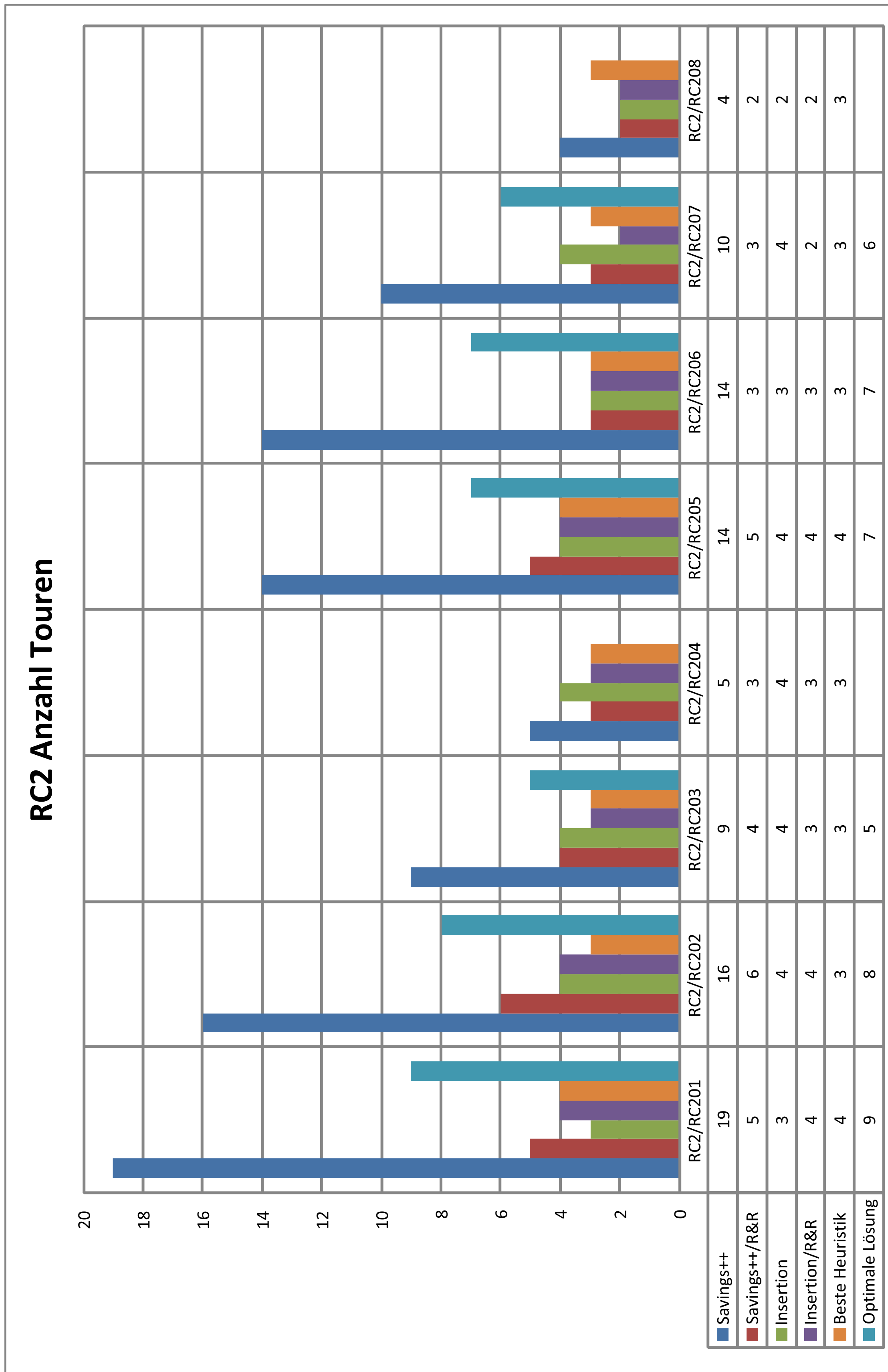


Abb. 7.18.: Anzahl Touren RC2

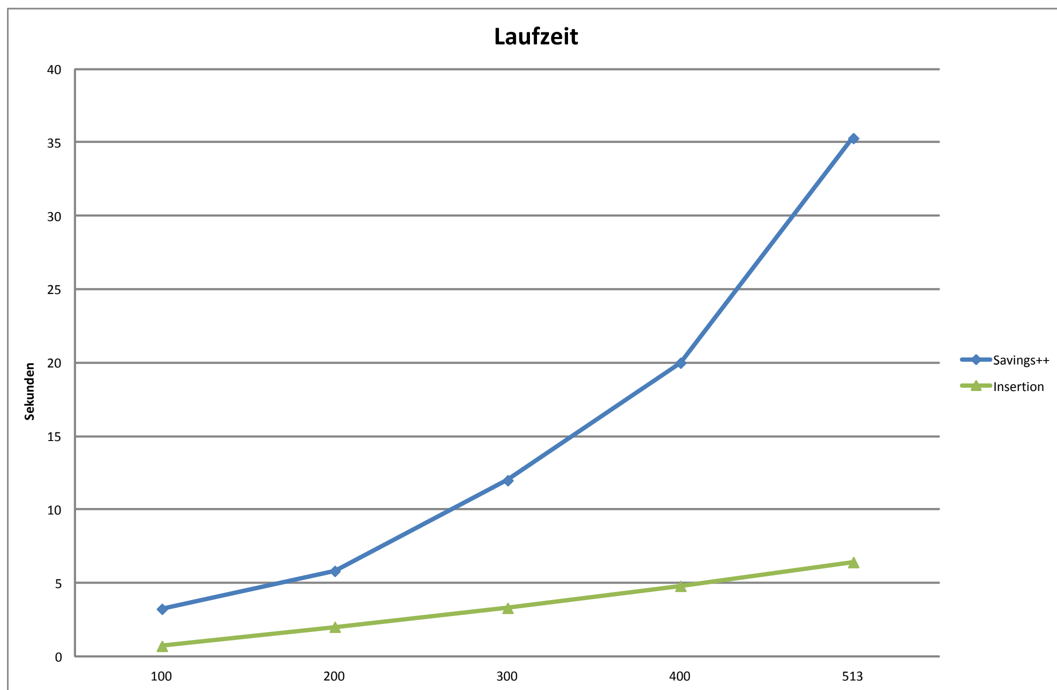


Abb. 7.19.: Vergleich Laufzeit

7.2. Auswertung

Im Folgenden werden die beiden von mir entwickelten Algorithmen mit dem bereits existierenden Algorithmus von PASS verglichen. Hierfür wurden aus reellen Testdaten mit 513 Aufträgen je fünf zufällige Teilinstanzen mit 100, 200, 300 und 400 Aufträgen gebildet und mit den Algorithmen optimiert. Zusätzlich wurde ein Durchlauf für die gesamten Daten durchgeführt.

Für den Algorithmus von PASS liegen leider keine Laufzeitdaten vor. Vergleicht man allerdings die Zeiten von Savings++ und Insertion, so erkennt man, dass die Laufzeitkurve des Savings++-Algorithmus deutlich schneller steigt (siehe Abb. 7.19). Dies liegt daran, dass beim Savings++ aufgrund der unterschiedlichen Fahrzeuge mehrere Durchläufe gebraucht werden, während der Insertion-Algorithmus damit ohne große Anpassung umgehen kann. So beträgt die Gesamtlaufzeit des Savings++-Algorithmus $O\left(n^3 \log n\right)$ und die des Insertion-Algorithmus nur $O\left(n^2\right)$ (erwartet).

Bei der Anzahl der Touren erkennt man bereits Qualitätsunterschiede zwischen den Algorithmen. Meine beiden Algorithmen haben ca. ein Drittel weniger Touren benötigt, um alle Aufträge auszuführen, als der PASS-Algorithmus benötigt hat (siehe Abb. 7.20). Auffällig ist auch, dass der Savings++-Algorithmus (bei Mehrfacheinsatz) bevorzugt gleiche Fahrzeuge mehrfach verwendet, während Insertion und PASS zunächst jedem Fahrzeug eine Tour zuweisen, bevor sie ein Fahrzeug ein zweites Mal verwenden (siehe Abb. 7.21).

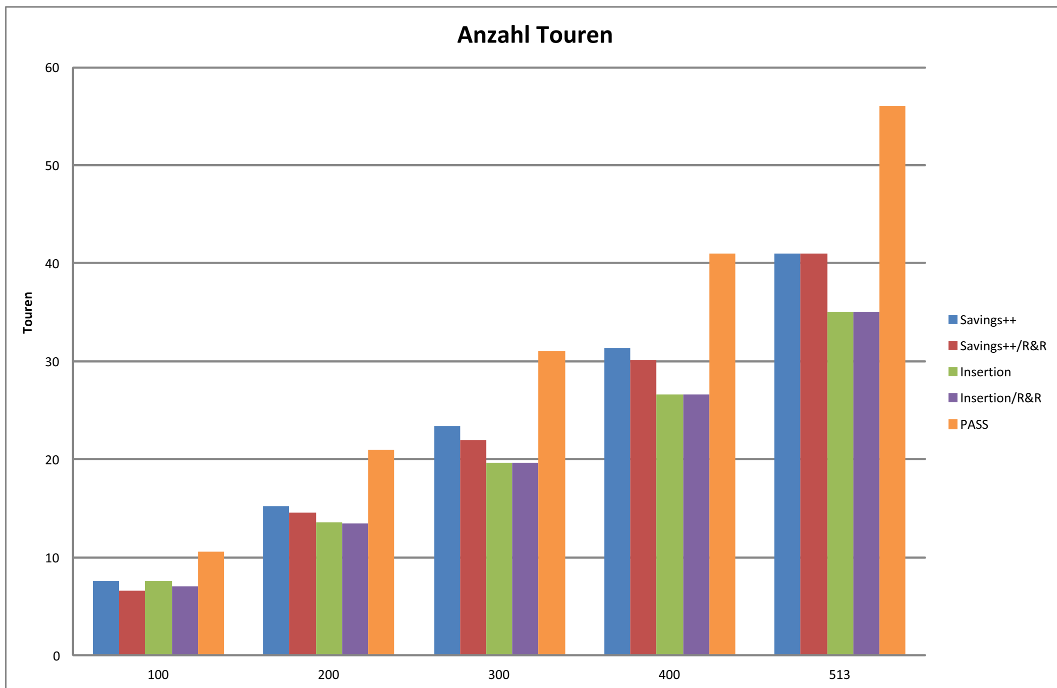


Abb. 7.20.: Vergleich Anzahl Touren

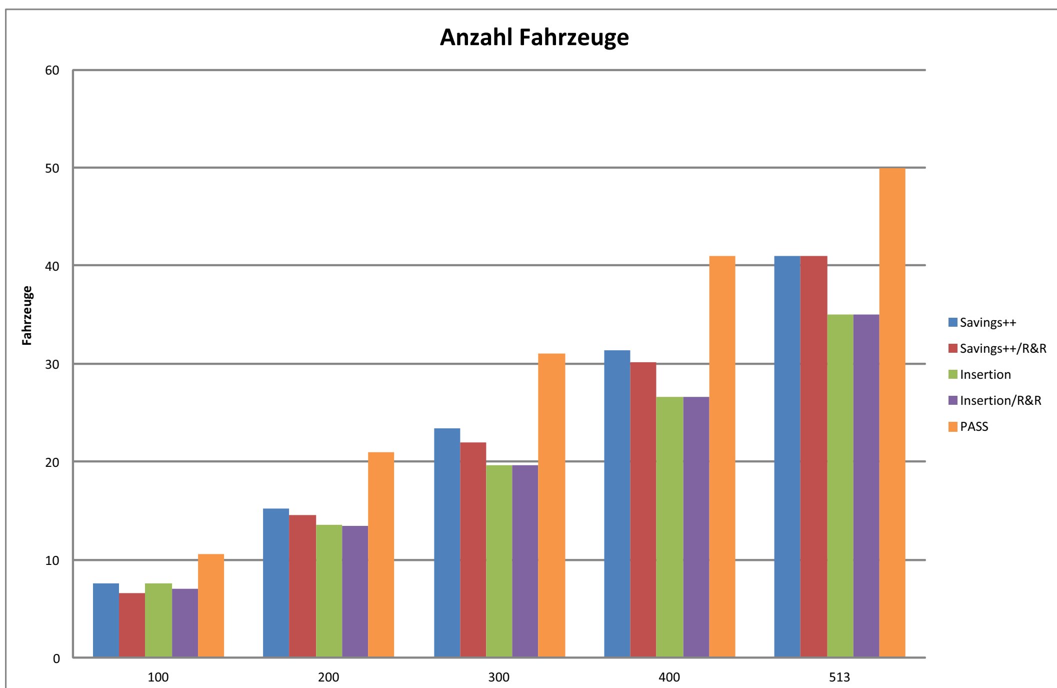


Abb. 7.21.: Vergleich Anzahl Fahrzeuge

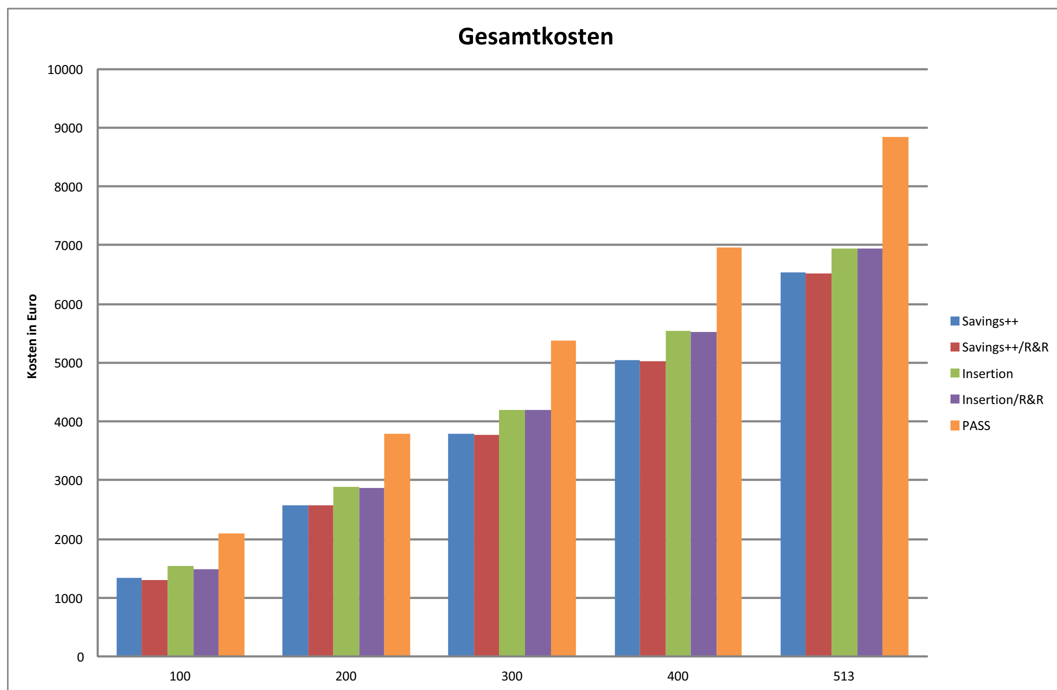


Abb. 7.22.: Vergleich Gesamtkosten

Bei den entstehenden Kosten übertreffen meine beiden Algorithmen den PASS-Algorithmus deutlich. Auffällig dabei ist, dass der Savings++-Algorithmus ebenfalls um einiges besser ist als der Insertion-Algorithmus (siehe Abb. 7.22). So konnte Savings++ die Kosten des PASS-Algorithmus auf der Gesamtinstanz der Testdaten fast halbieren, der Insertion-Algorithmus konnte die Kosten dabei nur um ca. 1/4 reduzieren. Betrachtet man zusätzlich die benötigte Gesamtdistanz (Abb. 7.23) und gesamte Tourdauer (Abb. 7.24), so fällt auf, dass vor allem die Dauer der Touren viel geringer ausfällt. Dies lässt sich dadurch begründen, dass durch die niedrigere Anzahl an Touren weniger Zeit im Depot zum Beladen benötigt wird.

Abschließend lässt sich noch eine grafische Ausgabe der drei Algorithmen betrachten. Dabei erkennt man, dass sich die Touren des Savings++- und des PASS-Algorithmus stärker ähneln als die des Insertion-Algorithmus. Die liegt daran, dass auch PASS mit einer modifizierten Savings-Variante arbeitet, während der Insertion mit seiner sequentiellen Vorgehensweise – im Gegensatz zur parallelen – ganz andere Touren aufbaut (siehe Abb. 7.25 – 7.29).

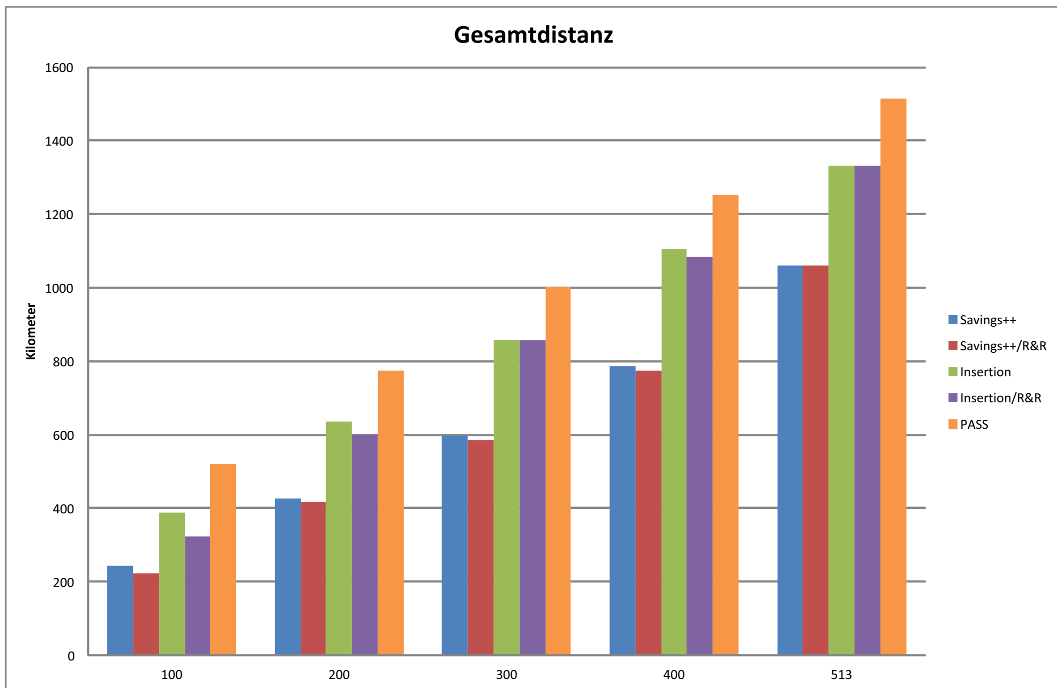


Abb. 7.23.: Vergleich Gesamtdistanz

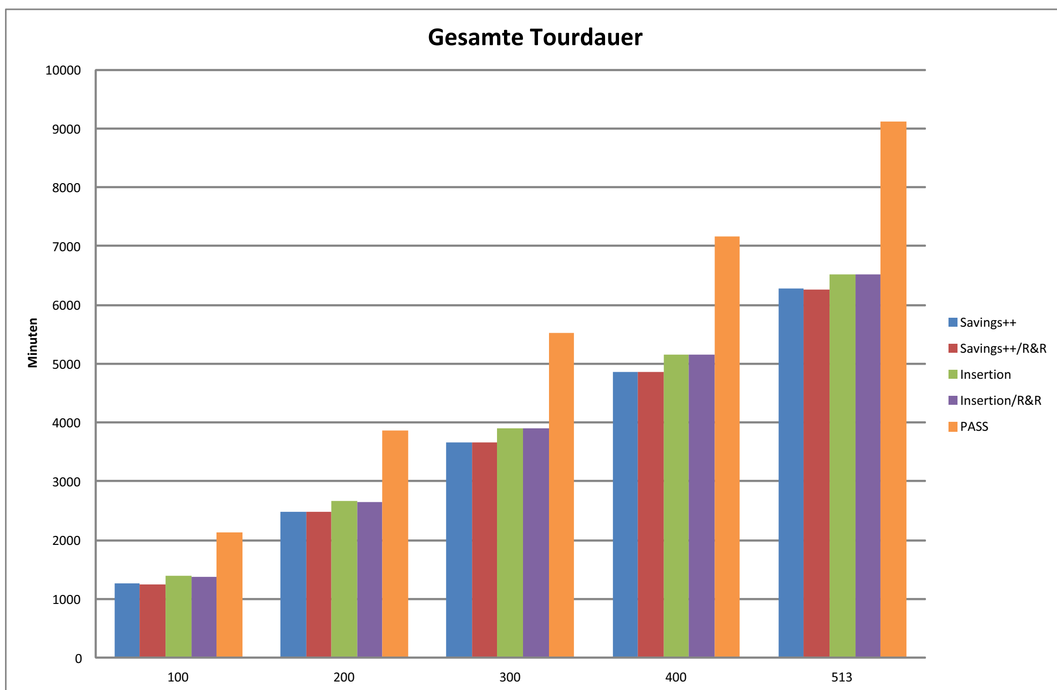


Abb. 7.24.: Vergleich gesamte Tourdauer

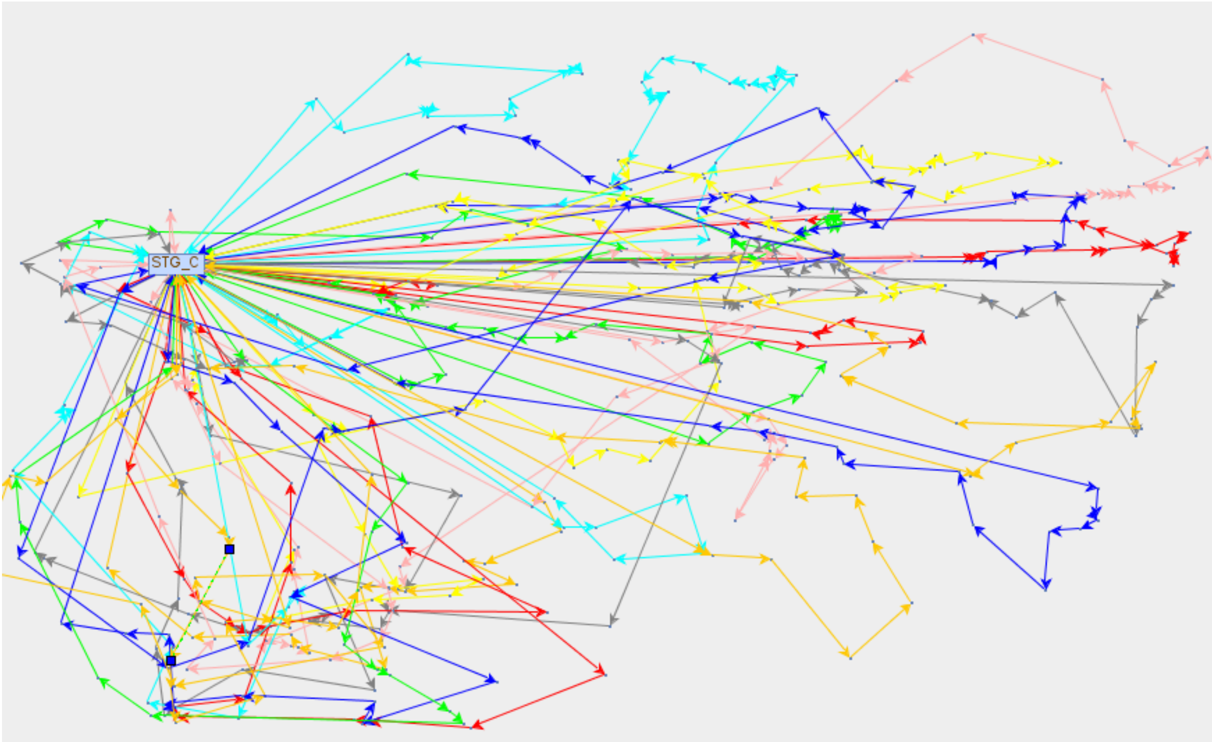


Abb. 7.25.: Ergebnis Savings++

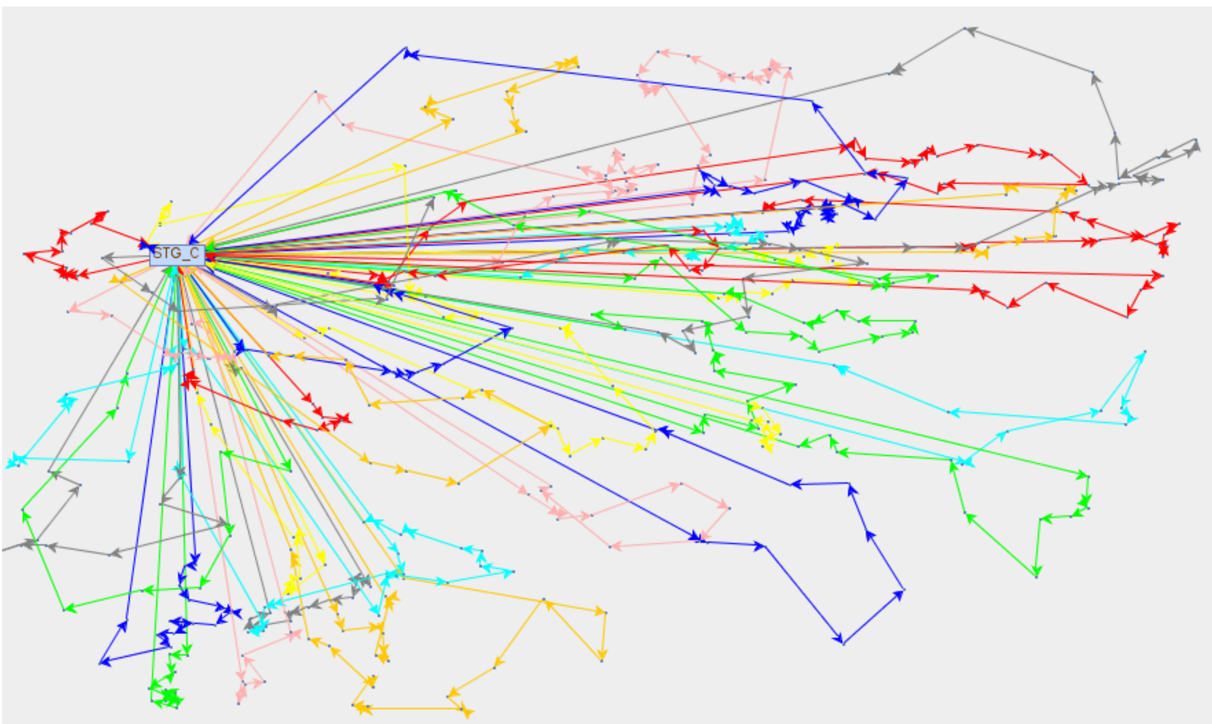


Abb. 7.26.: Ergebnis Savings++ mit Nachbearbeitung

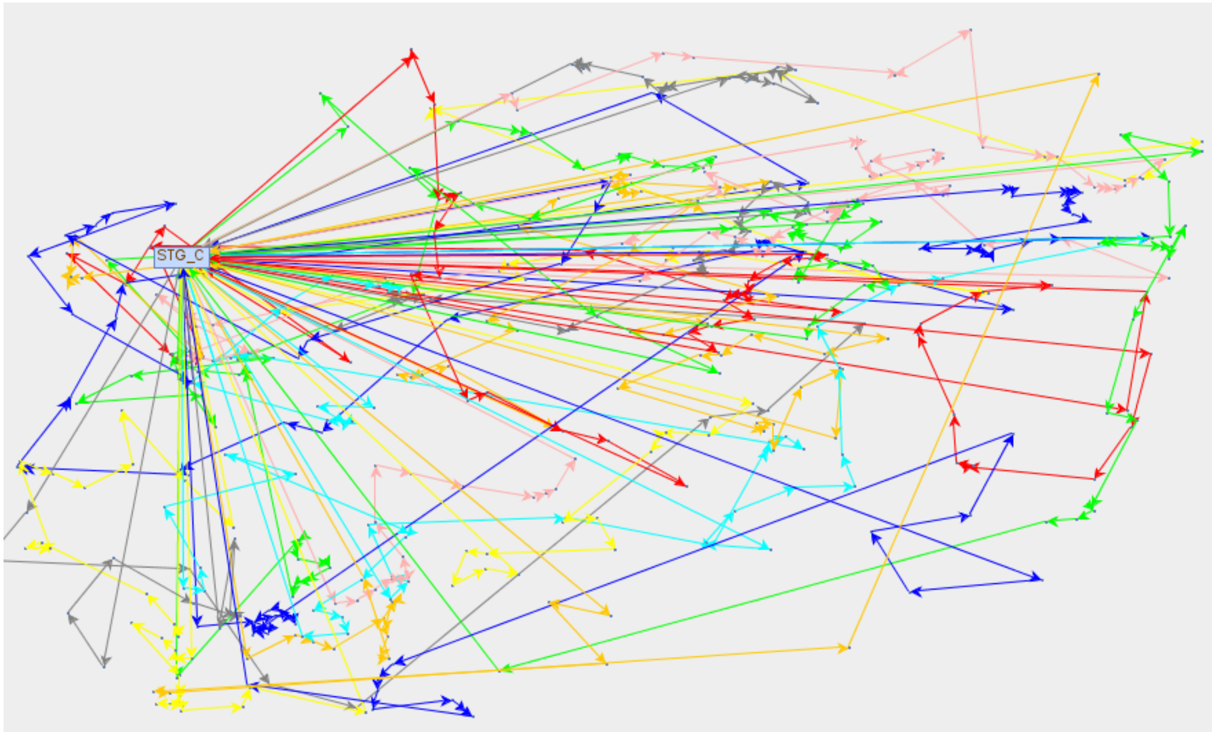


Abb. 7.27.: Ergebnis Insertion

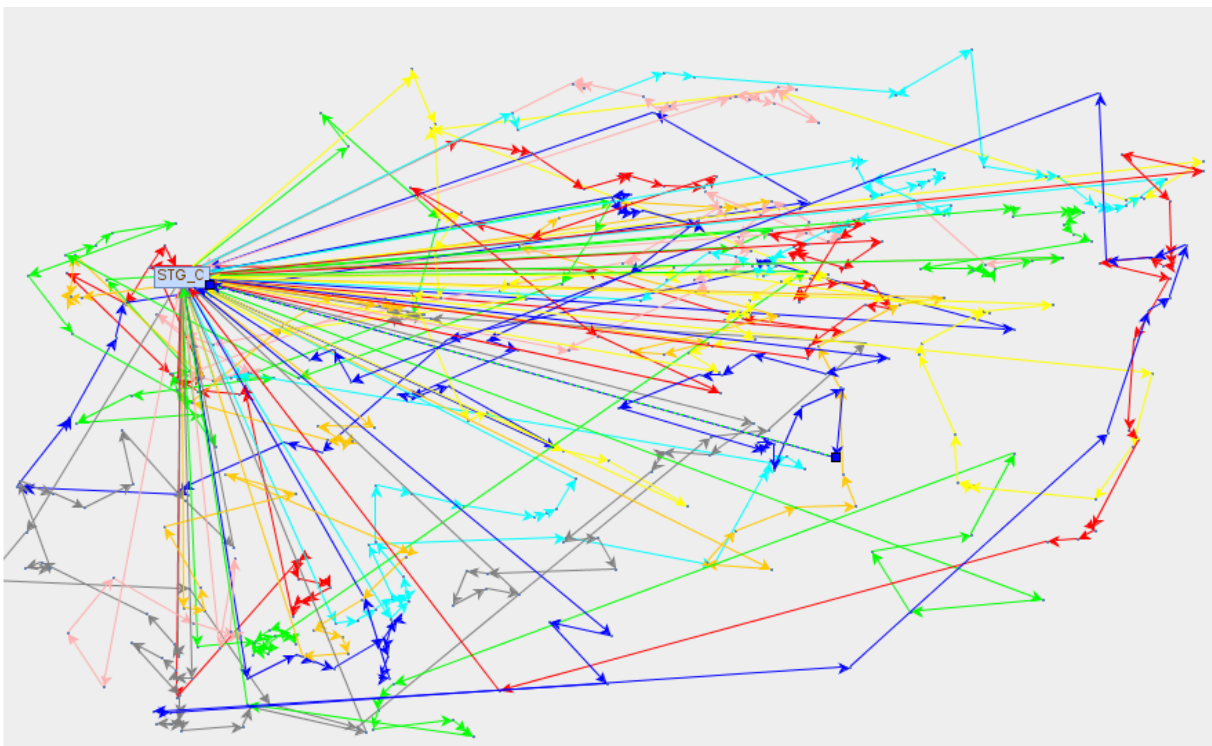


Abb. 7.28.: Ergebnis Insertion mit Nachbearbeitung

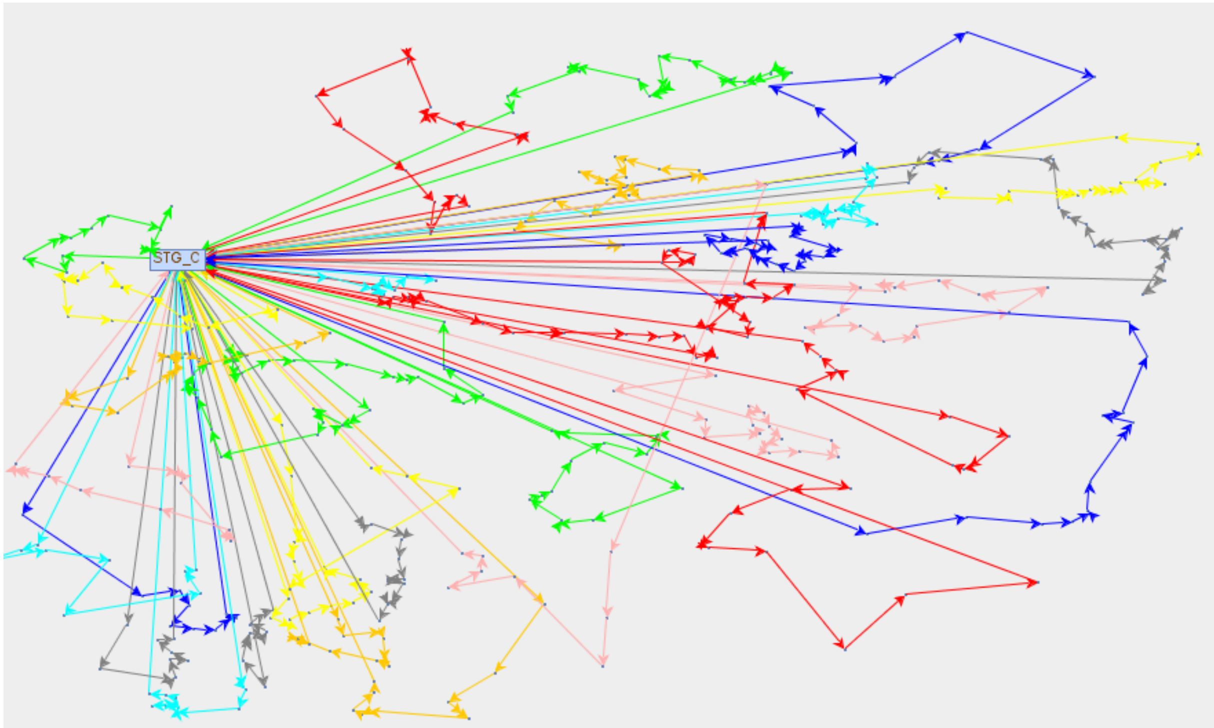


Abb. 7.29.: Ergebnis PASS

Anhang

A. Liste der Restriktionen

Von PASS wurde eine Liste von Restriktionen erstellt, die alle ermittelten Touren meiner Algorithmen erfüllen müssen. Diese werden in den kommenden Abschnitten aufgezählt und kurz beschrieben.

A.1. Fahrzeuge

Im Folgenden werden die Eigenschaften aufgezählt, die jedes Fahrzeug besitzt und die vom Algorithmus berücksichtigt werden müssen.

- Maximalmenge, -gewicht, -volumen:
  Für jedes Fahrzeug ist eine Obergrenze der drei Kapazitäten angegeben. Diese dürfen nicht überschritten werden.
- Hänger:
  Fahrzeugen können in Plantour zusätzlich Hänger zugewiesen werden, die jeweils wieder über zusätzliche Kapazitäten verfügen, allerdings auch die Kosten der Tour erhöhen.
- Speditions- / Passive Fahrzeuge ignorieren:
  Es können einzelne Fahrzeuge als Speditions-Fahrzeuge oder als passive Fahrzeuge markiert werden, die dann optional vom Algorithmus nicht zur Tourenplanung verwendet werden dürfen.
- Fahrzeuggruppe:
  Es gibt verschiedene Gruppen von Fahrzeugen, wobei manche Aufträge nur von einer bestimmten Fahrzeuggruppe ausgeführt werden können. Optional können diese Gruppen auch hierarchisch angeordnet sein. Die Fahrzeuggruppenhierarchie hat zur Folge, dass zum Beispiel alle Umschlagstellen mit Fahrzeuggruppe 3 auch von den Fahrzeuggruppen 1 und 2 angefahren werden dürfen.
- Zusätzliche Eigenschaften:
  Es können für die Fahrzeuge zusätzliche Eigenschaften angegeben werden, z.B. eine Hebebühne. In den Aufträgen wird dann deklariert, welche Eigenschaften das Fahrzeug erfüllen muss, das diesen Auftrag ausführt.
- Kalender berücksichtigen:
  Es ist ein Kalender vorhanden, der die Verfügbarkeit der Fahrzeuge enthält. Hier muss darauf geachtet werden, dass Fahrzeuge nur an verfügbaren Tagen verplant werden.

- Vorgabe Hofzeit:
  Jedes Fahrzeug benötigt eine gewisse Zeit im Depot, bevor es mit seiner ersten Tour beginnen kann. Die Hofzeit kann ebenso beim Kunden und beim Depot eingestellt werden, diese Zeiten sind aufzuaddieren und vor dem Start jeder Tour einzuplanen.

A.2. Personal

Im Folgenden werden die Eigenschaften aufgezählt, die das Personal besitzt und die vom Algorithmus berücksichtigt werden müssen.

- Kalender berücksichtigen:
  Für das Personal ist ebenfalls ein Kalender vorhanden, der die Verfügbarkeit der angestellten enthält. Angestellte dürfen nur an verfügbaren Tagen arbeiten.
- Persönliche Restriktionen:
  Für manche Aufträge sind nur Fahrer mit bestimmten Eigenschaften zulässig, z.B. kann ein Waffenschein nötig sein. Diese müssen bei der Verplanung der Aufträge beachtet werden.

A.3. Aufträge

Im Folgenden werden die Eigenschaften aufgezählt, die jeder Auftrag besitzt und die vom Algorithmus berücksichtigt werden müssen.

- Zeitfenster beachten:
  Es können mehrere Zeitfenster für die Aufträge angegeben werden. Wird ein Auftrag ausgeführt, so muss das Fahrzeug innerhalb eines dieser Zeitfenster beim Kunden ankommen.
- Fahrzeugnummer:
  Man kann Aufträgen über die Fahrzeugnummer festen Fahrzeuge zuweisen. Diese dürfen dann nur von diesem Fahrzeug ausgeführt werden.
- Fahrzeuggruppe, Fahrzeugeigenschaften:
  Siehe Abschnitt A.1.
- Fahrzeug Besatzung Für manche Aufträge ist eine Zweimannbesatzung notwendig. Da sich durch die zusätzlichen Personalkosten die Kosten der gesamten Tour erhöhen, wenn man zwei Besatzungsmitglieder braucht, muss der Algorithmus sinnvoll möglichst viele Zweimannaufträge zu einer Tour zusammenfassen.
- Personal Restriktionen:
  Siehe Abschnitt A.2.
- Vorgabe Öffnungszeit Für Aufträge und Kunden können Zeitfenster angegeben werden, innerhalb deren Dauer das Fahrzeug beim Kunden eintreffen muss. Dabei gilt: Hat der Auftrag keine Zeitfenster, so werden die Öffnungszeiten des Kunden für

den Liefertag genommen. Hat dieser ebenfalls keine Zeitfenster, so werden Default Werte gesetzt.

- Fixdatum beachten:
  Für Aufträge kann ein festes Datum angegeben werden, an dem sie ausgeführt werden müssen. Optional kann diese Funktion auch deaktiviert werden.
- Passive Aufträge ignorieren:
  Aufträge, die als passiv gekennzeichnet sind, werden je nach Einstellung vom Algorithmus ignoriert oder trotzdem verplant.
- Mengen verändern:
  Für Aufträge, deren Menge/Gewicht/Volumen einen Grenzwert überschreitet, können verschiedene Vorgehensweisen gewählt werden:
  - Mengen nicht verändern:
    Der Auftrag bleibt in seiner Form bestehen und wird im Zweifelsfall nicht verplant.
  - Mengen auf Obergrenze reduzieren:
    Es werden Transporteinheiten entfernt, bis die Obergrenze erreicht ist.
  - Mengen ab Obergrenze splitten:
    Aus dem Auftrag werden mehrere kleinere Aufträge erstellt, deren Mengen unter dem Grenzwert liegen.
- Standzeit Für jeden Auftrag wird eine feste Zeit eingeplant, die zum Entladen der Ware nötig ist. Diese kann auch beim Kunden und abhängig von der Menge der Waren angegeben werden.

A.4. Touren

Im Folgenden werden die Eigenschaften aufgezählt, die jede Tour besitzen muss, die vom Algorithmus gebildet wird.

- Mindestmenge, -gewicht, -volumen:
  Es kann eingestellt werden, dass nur Aufträge ab einer Mindestkapazität verplant werden dürfen. Aufträge mit darunter liegenden Mengen werden nicht verplant.
- Maximale Wartezeit:
  Die Gesamtkosten einer Tour können geringer sein, wenn man einen Auftrag in einer Tour hat, bei dem aufgrund der Zeitfenster Wartezeiten entstehen. Damit diese nicht überhandnehmen, kann eine maximale Wartezeit pro Tour eingestellt werden.
- Maximale Anzahl Tourposten:
  Es kann eine maximale Anzahl an Aufträgen eingestellt werden, die pro Tour verplant werden dürfen.
- Maximale Tourdauer:
  Ebenso kann die Gesamtdauer der Tour begrenzt werden. Die maximale Tourdauer

darf dabei vom Algorithmus nicht überschritten werden.

- Planungsvarianten:
  Für die Art der Planung werden fünf verschiedene Varianten unterschieden:
  - Tagesplanung Einfacheinsatz:
    Alle Touren müssen auf dem selben Tag liegen, es darf kein Fahrzeug mehr als eine Tour fahren.
  - Tagesplanung Mehrfacheinsatz:
    Alle Touren müssen auf dem selben Tag liegen, jedes Fahrzeug darf beliebig viele Touren fahren.
  - Mehrtagesplanung ohne Übernachtung:
    Die Touren dürfen auf unterschiedlichen Tagen liegen, allerdings muss jedes Fahrzeug am Abend in das Depot zurückkehren.
  - Mehrtagsplanung mit Übernachtung, Einfacheinsatz:
    Die Touren dürfen auf unterschiedlichen Tagen liegen und die Fahrer dürfen in Hotels übernachten, wenn die Kosten für eine Unterkunft billiger sind, als das Zurückfahren zum Depot und das erneute Hinfahren am nächsten Tag. Jedes Fahrzeug darf nur eine einzige Tour fahren.
  - Mehrtagsplanung mit Übernachtung, Mehrfacheinsatz:
    Die Touren dürfen auf unterschiedlichen Tagen liegen und die Fahrer dürfen in Hotels übernachten, wenn die Kosten für eine Unterkunft billiger sind, als das Zurückfahren zum Depot und das erneute Hinfahren am nächsten Tag. Jedes Fahrzeug darf beliebig viele Touren fahren.
- Verkehrsflussfaktor:
  Hiermit kann Einfluss auf die Dauer der Strecken genommen werden. Es wird ein Prozentsatz angegeben, der bestimmt, wie schnell das Fahrzeug vorwärts kommt.
- Einfädelzeit:
  Die Einfädelzeit gibt an, wie lange ein Fahrzeug braucht, um vom Kunden wieder in den Verkehr zu finden. So werden alle Strecken fix um diese Zeit verlängert.
- Abweichung Öffnung und Schließung:
  Hierdurch kann eine Zeit bestimmt werden, die ein Fahrzeug außerhalb der Zeitfenster beim Kunden eintreffen darf.
- Entladung in der Öffnungszeit:
  Es kann zusätzlich eingestellt werden, dass ein Fahrzeug nicht nur innerhalb der gegebenen Zeitfenster beim Kunden eintreffen, sondern auch wieder bei ihm abfahren muss.
- Frühester Tourstart:
  Hierdurch wird die frühest mögliche Startzeit für alle Touren festgelegt. Eine Tour darfnicht vor dieser Uhrzeit im Depot starten, auch wenn es die Hofzeiten erlauben würden.

B. Ganzzahliges lineares Programm

Im Folgenden wird das resultierende ganzzahlige lineare Programm im Ganzen definiert.

Es gibt zwei Typen von binären und drei Typen von ganzzahligen Variablen. Die binäre Variable $x_{ijk}(i \neq j)$ ist genau dann 1, wenn in der optimalen Lösung die Kante (i,j) durch das Fahrzeug k befahren wird. Die binäre Variable y_{ik} ist genau dann 1, wenn in der optimalen Lösung das Fahrzeug k den Knoten i anfährt. Die Variable b_i entspricht der Ankunftszeit beim Kunden i, w_{ik} der Wartezeit, die beim Kunden i bei der Ankunft von Fahrzeug k entsteht. ts_k entspricht dem Tourstart von Fahrzeug k.

Außerdem müssen mehrere Konstanten definiert werden:

Konstante	Beschreibung
n	Anzahl der Kunden
m	Anzahl der Fahrzeuge
T	Sehr große Zahl
c_{ij}	Kosten, die entstehen, wenn ein Fahrzeug k die Kante (i,j) befährt
t_{ij}	Fahrzeit von Knoten i zu j
c_k^w	Kosten, die durch Wartezeit für ein Fahrzeug k enstehen
cpt_k	Feste Kosten pro Tour für das Fahrzeug k
$[e_i, l_i]$	Zeitfenster für einen Knoten i
D_k^a, D_k^v, D_k^w	Kapazitäten (Menge, Volumen, Gewicht) des Fahrzeugs k
d_i^a, d_i^v, d_i^w	Auszuliefernde Mengen für Kunde i
vg_k	Fahrzeuggruppe des Fahrzeugs k
vrv_{il}	Fahrzeugeigenschaft $l \in \{1, \dots, o\}$ des Fahrzeugs k
cg_i	Benötigte Fahrzeuggruppe des Kunden i
vrv_{il}	Benötigte Fahrzeugeigenschaft $l \in \{1, \dots, o\}$ des Kunden i
vn_i	Benötigte Fahrzeugnummer des Kunden i
s_i	Standzeit beim Kunden i
at	Einfädelzeit nach jedem Kunden
tf	Verkehrsflussfaktor
mtl	Maximale Tourdauer
mwt	Maximale Wartezeit
mwt	Maximale Anzahl Tourposten
de	Tolerierte Abweichung von der Öffnungszeit
dl	Tolerierte Abweichung von der Schließzeit
ets	Frühest möglicher Tourstart

Ebenfalls gibt es folgende binäre Konstanten, die jeweils einer Option von Plantour entsprechen:

$$atw = \begin{cases} 1 & \text{falls Zeitfenster beachtet werden müssen} \\ 0 & \text{falls nicht} \end{cases}$$

$$vgh = \begin{cases} 0 & \text{Keine Fahrzeughierarchie} \\ 1 & \text{Aufsteigende Fahrzeughierarchie} \\ 2 & \text{Absteigende Fahrzeughierarchie} \end{cases}$$

$$dis = \begin{cases} 1 & \text{falls innerhalb der Öffnungszeit beim Kunden abgefahren werden muss} \\ 0 & \text{falls nicht} \end{cases}$$

Mit Hilfe dieser Variablen und Konstanten wird nun das ganzzahlige lineare Programm definiert:

$$\text{(0) minimiere} \sum_{k=1}^{m} \left(y_{0k} \cdot cpt_k + \sum_{i=0}^{n} \left(c_k^w \cdot w_{jk} + \sum_{\substack{j=0 \\ j \neq i}}^{n} c_{ijk} x_{ijk} \right) \right)$$

unter den Nebenbedingungen

$$(1a) \qquad \sum_{i=0}^{n} d_i^a y_{ik} \leq D_k^a \quad (k = 1, \ldots, m)$$

$$(1b) \qquad \sum_{i=0}^{n} d_i^w y_{ik} \leq D_k^w \quad (k = 1, \ldots, m)$$

$$(1c) \qquad \sum_{i=0}^{n} d_i^v y_{ik} \leq D_k^v \quad (k = 1, \ldots, m)$$

$$(2a) \qquad \sum_{k=1}^{m} y_{0k} \leq m$$

$$(2b) \qquad y_{0k} \geq \frac{1}{n} \cdot \sum_{i=1}^{n} y_{ik} \quad (k = 1, \ldots, m)$$

$$(3) \qquad \sum_{k=1}^{m} y_{ik} = 1 \quad (i = 1, \ldots, n)$$

$$(4) \qquad \sum_{i=0}^{n} x_{ijk} = y_{jk} \quad (j = 0, \ldots, n; \; k = 1, \ldots, m)$$

(5) $$\sum_{j=0}^{n} x_{ijk} = y_{ik}$$
$$(i = 0, \ldots, n;\ k = 1, \ldots, m)$$

(6a) $$\sum_{i,j \in S} x_{ijk} \leq |S| - 1$$
$$(S \subset V \backslash 0;\ |S| \geq 2;\ (k = 1, \ldots, m)$$
(werden erst nach und nach eingefügt)

(6b) $$y_{0k} \geq \tfrac{1}{n} \cdot \sum_{i=1}^{n} y_{ik}$$
$$(k = 1, \ldots, m)$$

(7.1) $$b_j \geq b_i + w_{ik} + s_i + at + tf \cdot t_{ij} - (1 - x_{ijk})T$$
$$(i, j = 0, \ldots, n;\ k = 1, \ldots, m)$$

(7.2) $$b_j \geq b_i + w_{ik} + s_i + at + tf \cdot t_{ij} + (1 - x_{ijk})T$$
$$(i, j = 0, \ldots, n;\ k = 1, \ldots, m)$$

(8a) $$atw \cdot e_i - de \leq atw \cdot b_i + \sum_{k=1}^{m} w_{ik}$$
$$(i = 1, \ldots, n)$$

(8b) $$atw \cdot b_i + \sum_{k=1}^{m} w_{ik} + dis \cdot s_i \leq atw \cdot l_i + dl$$
$$(i = 1, \ldots, n)$$

(12a) $$y_{ik} \cdot vh \cdot (1 - vhg) \cdot (2 - vgh) \cdot cg_i = y_{ik} \cdot vh \cdot (1 - vhg) \cdot (2 - vgh) \cdot cg_i \cdot vg_k$$
$$(i = 1, \ldots, n;\ k = 1, \ldots, m)$$

(12b) $$y_{ik} \cdot vh \cdot vgh \cdot (2 - vgh) \cdot cg_i \geq y_{ik} \cdot vh \cdot vgh \cdot (2 - vgh) \cdot cg_i \cdot vg_k$$
$$(i = 1, \ldots, n;\ k = 1, \ldots, m)$$

(12c) $$y_{ik} \cdot vh \cdot vgh \cdot (1 - vhg) \cdot cg_i \leq y_{ik} \cdot vh \cdot vgh \cdot (1 - vhg) \cdot cg_i \cdot vg_k$$
$$(i = 1, \ldots, n;\ k = 1, \ldots, m)$$

(13) $$y_{ik} \cdot vrc_{il} = y_{ik} \cdot vrc_{il} \cdot vrv_{kl}$$
$$(i = 1, \ldots, n; k = 1, \ldots, m; l = 1, \ldots, o)$$

(14) $$y_{ik} \cdot vn_i \cdot k = y_{ik} \cdot vn_i \cdot vn_i$$

(15) $$\sum_{i=1}^{n} w_{ik} \leq mwt$$
$$(k = 1, \ldots, m)$$

(16)
$$ct \cdot \sum_{i=1}^{n} y_{ik} \quad \leq \quad ct \cdot ct \qquad (k = 1, \ldots, m)$$

(17)
$$mtl \cdot \sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} x_{ijk} \cdot tf \cdot t_{ij} + mtl \cdot \sum_{i=1}^{n} (w_{ik} + y_{ik} \cdot (s_i + at)) \quad \leq \quad mtl \cdot mtl \qquad (k = 1, \ldots, m)$$

(18a.1)
$$ts_k \quad \geq \quad b_j - tf \cdot t_{0j} - (1 - x_{0jk}) \cdot T \qquad (k = 1, \ldots, m; \; j = 1, \ldots, n)$$

(18a.2)
$$ts_k \quad \leq \quad b_j - tf \cdot t_{0j} + (1 - x_{0jk}) \cdot T \qquad (k = 1, \ldots, m; \; j = 1, \ldots, n)$$

(18b.1)
$$ets \quad \leq \quad ts_k \qquad (k = 1, \ldots, m)$$

(18b.2)
$$ts_k \quad \leq \quad ets + di \qquad (k = 1, \ldots, m)$$

(9)
$$x_{ijk} \quad \in \quad \{0, 1\} \qquad (i, j = 0, \ldots, n; \; k = 1, \ldots, m)$$

(10)
$$y_{ik} \quad \in \quad \{0, 1\} \qquad (i = 1, \ldots, n; \; k = 1, \ldots, m)$$

(11)
$$w_{ik} \quad \geq \quad 0 \qquad (i = 0, \ldots, n; \; k = 1, \ldots, m)$$

C. Abbildungsverzeichnis

1.1. Lösung eines VRP: Knotenüberdeckung der Kunden 5

3.1. Initialisierung der Touren 11
3.2. Verbindung von zwei Touren 12

4.1. Einfügeschritt der Insertion-Heuristik 19

5.1. Eine Ruin & Recreate Iteration 26
5.2. Lokale Minima 26
5.3. Zwei Durchläufe mit Ruin & Recreate 30
5.4. Verhältnis Anzahl Durchläufe zu Anzahl Iterationen 30
5.5. Anzahl Durchläufe 31
5.6. Anfänglicher Schwellwert 32
5.7. Anfänglicher Schwellwert in Abhängigkeit von Gesamtkosten 32

6.1. Einfügen einer Schnittebene in bereits existierende Lösung 42

7.1. R104 44
7.2. R206 44
7.3. C109 45
7.4. C206 45
7.5. RC107 45
7.6. RC204 45
7.7. Gesamtdistanz R1 46
7.8. Anzahl Touren R1 47
7.9. Gesamtdistanz R2 48
7.10. Anzahl Touren R2 49
7.11. Gesamtdistanz C1 50
7.12. Anzahl Touren C1 51
7.13. Gesamtdistanz C2 52
7.14. Anzahl Touren C2 53
7.15. Gesamtdistanz RC1 54
7.16. Anzahl Touren RC1 55
7.17. Gesamtdistanz RC2 56
7.18. Anzahl Touren RC2 57
7.19. Vergleich Laufzeit 58
7.20. Vergleich Anzahl Touren 59
7.21. Vergleich Anzahl Fahrzeuge 59

7.22. Vergleich Gesamtkosten 60
7.23. Vergleich Gesamtdistanz 61
7.24. Vergleich gesamte Tourdauer 61
7.25. Ergebnis Savings++ 62
7.26. Ergebnis Savings++ mit Nachbearbeitung 62
7.27. Ergebnis Insertion 63
7.28. Ergebnis Insertion mit Nachbearbeitung 63
7.29. Ergebnis PASS 64

D. Liste der Algorithmen

1. preprocess(List<Vehicle> vehicles, List<Order> orders) 8
2. preprocessVehicles(List<Vehicle> vehicles) 8
3. preprocessOrders(List<Order> orders) 9

4. Savings++.runAlgorithm(List<Order> orders, List<Vehicle> vehicles) . . . 13
5. Savings++.initialize(List<Order> dayOrders, List<Vehicle> dayVehicles) . 14
6. Savings++.optimize(List<Tour> dayTours) 15
7. Savings++.saveTours(List<Tour> dayTours, List<Order> dayOrders, List<Vehicle> dayVehicles) 17

8. Insertion.runAlgorithm(List<Order> orders, List<Vehicle> vehicles) 21
9. Insertion.initialize(List<Order> dayOrders, List<Vehicle> dayVehicles) . . 21
10. Insertion.optimize(Tour tour, List<Order> dayOrders) 23

11. ruinRecreate(List<Tour> tours, int max_runs) 27
12. ruin(List<Tour> tours, int run) 28
13. recreate(List<Tour> tours, List<Order> B) 29

E. Literaturverzeichnis

[BG05] O. Bräysy und M. Gendreau: Vehicle Routing Problem with Time Windows, Part I: Route Construction and Local Search Algorithms. *Transportation Science*, 39:104–118, 2005. http://portal.acm.org/citation.cfm?id=1247226.1247233. [S. 6]

[BQ64] M. L. Balinski und R. E. Quandt: On an Integer Program for a Delivery Problem. *Operations Research*, 12:300–304, 1964. [S. 6]

[CC01] R. Cordone und R. W. Calvo: A Heuristic for the Vehicle Routing Problem with Time Windows. *Journal of Heuristics*, 7:107–129, March 2001. http://dl.acm.org/citation.cfm?id=594948.595055. [S. 6]

[CT01] Mingozzi A. Christofides, N. und P. Toth: Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical Programming.*, 20:255–282, 2001. [S. 6]

[CW64] G. Clarke und J. W. Wright: Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 12(4):568–581, 1964. http://dx.doi.org/10.1287/opre.12.4.568. [S. 5, 11]

[DR59] G. B. Dantzig und J. H. Ramser: The Truck Dispatching Problem. *Management Science*, 6(1):80–91, 1959. http://mansci.journal.informs.org/cgi/content/abstract/6/1/80. [S. 4]

[DS90] G. Dueck und T. Scheuer: Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90:161–175, 1990. [S. 25]

[FJ81] M.L. Fisher und R. Jaikumar: A generalized assignment heuristic for vehicle routing. *Networks*, 11:109–124, 1981. [S. 6, 33]

[GM74] B. E. Gillett und L. R. Miller: A Heuristic Algorithm for the Vehicle-Dispatch Problem. *Operations Research*, 22:340–349, 1974. [S. 5]

[KGV83] S. Kirkpatrick, C. D. Gelatt und M. P. Vecchi: Optimization by Simulated Annealing. *Science*, 220:671–680, 1983. [S. 25]

[Lap92] G. Laporte: The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345–358, 1992. http://ideas.repec.org/a/eee/ejores/v59y1992i3p345-358.html. [S. 6]

[LMN85] G. Laporte, H. Mercure und Y. Nobert: Optimal routing under capacity and distance restrictions. *Operations Research*, 33:1050–1073, 1985. [S. 6]

[LMN86] G. Laporte, H. Mercure und Y. Nobert: An exact algorithm for the asymmetrical capacitated vehicle routing problem. *Networks*, 16:33–46, 1986. [S. 6]

[Pae88] H. Paessens: The savings algorithm for the vehicle routing problem. *European Journal of Operational Research*, 34(3):336–344, 1988. http://ideas.repec.org/a/eee/ejores/v34y1988i3p336-344.html. [S. 6]

[PS96] P. Prosser und P. Shaw: Study of greedy search with multiple improvement heuristics for vehicle routing problems. Technischer Bericht, University of Strathclyde, Glasgow, Scotland, 1996. [S. 6]

[Sei91] R. Seidel: A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. *Comput. Geom. Theory Appl*, 1:51–64, 1991. [S. 24]

[Sol87] M. M. Solomon: Algorithms for the vehicle routing and scheduling problems with time window constraints. *Oper. Res.*, 35:254–265, 1987. http://portal.acm.org/citation.cfm?id=41439.41448. [S. 5, 11, 19]

[Sol05] M. M. Solomon: VRPTW Benchmark Problems, 2005. http://w.cba.neu.edu/~msolomon/problems.htm. [S. 44]

[Spo07] S. Spoorendonk: Optimal Solutions for Solomon problems, 2007. http://www.diku.dk/hjemmesider/ansatte/spooren/solomon/r1r2solu.htm. [S. 44]

[SSSWD00] G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt und G. Dueck: Record breaking optimization results using the ruin and recreate principle. *J. Comput. Phys.*, 159:139–171, 2000. http://portal.acm.org/citation.cfm?id=344464.344466. [S. 6, 25]

[Č85] V. Černý: Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985. [S. 25]

F. Erklärung der selbständigen Bearbeitung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und dabei keine anderen Hilfsmittel und Quellen als die angegebenen benutzt zu haben.

Würzburg, den ______________, ____

(Philipp Kindermann)