FAKULTÄT FÜR !NFORMATIK

Ein hybrides Verfahren basierend auf Variabler Nachbarschaftssuche und Dynamischer Programmierung zur Tourenfindung in einem Ersatzteillager mit domänenspezifischen Nebenbedingungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Misar
Matrikelnummer 0025068

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.Prof. Dipl.-Ing. Dr. Günther Raidl
Mitwirkung: Univ.Ass. Mag. Dipl.-Ing. Matthias Prandtstetter

Wien, 20.04.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Danksagung

Die vorliegende Arbeit durfte ich am Institut für Computergraphik und Algorithmen der Technischen Universität Wien erstellen und es freut mich diese nun fertig in Händen halten zu können.

Ich möchte mich für die Geduld und Mithilfe von Seiten Günther Raidls bedanken und ebenso Matthias Prandtstetter großen Dank für seine Betreuung aussprechen. In den vergangenen Monaten hat er einen besonders großen Beitrag zur Vervollständigung dieser Arbeit geleistet. Natürlich gilt meine Anerkennung auch allen anderen Personen, die von Seiten des Instituts ihren Anteil an der Entstehung dieser Arbeit hatten.

Es ist mir weiters ein Anliegen auch all jene zu erwähnen, die mich im Laufe des gesamten Studiums begleitet und unterstützt haben. Dazu zählen vor allem meine Studienkollegen Christian und Gerhard, sowie gleichermaßen auch meine Eltern, die mir zu jedem Zeitpunkt eine große Hilfe waren und Linda, der ich an dieser Stelle für ihre Ausdauer und Motivation danken will.

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, Thomas Misar, wohnhaft in 1070 Wien, Seidengasse 3/108, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit (einschließlich Tabellen, Karten und Abbildungen), die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20.04.2009

(Unterschrift Verfasser)

Kurzfassung

Im Rahmen dieser Arbeit wird eine konkrete Problemstellung aus dem Bereich der Lagerverwaltung behandelt. Dabei soll die benötigte Zeit zum Ausfassen von Artikeln aus dem Lager unter Berücksichtigung von domänenspezifischen Nebenbedingungen minimiert werden. Ausgehend von durch Kunden laufend aufgegebenen Bestellungen sollen feste Lieferzeiten eingehalten werden und Einschränkungen wie etwa Kapazitätslimits oder das Vermeiden von Kollisionen zwischen Arbeitern beachtet werden. Die für die gegebene Problemstellung zentrale Bestimmung effizienter Touren steht im Mittelpunkt der Arbeit, welche mit Ergebnissen aus einer konkreten Implementierung des vorgestellten Ansatzes abschließt.

Es wird ein Algorithmus vorgestellt, der ein eigens entwickeltes *Dynamisches Programm* zur Berechnung optimaler Wege durch das Warenlager mit der Umsetzung einer *Variablen Nachbarschaftssuche (engl.: Variable Neighborhood Search)* (VNS) verbindet. In mehreren Phasen werden dabei die vorliegenden Bestellungen zerlegt und davon ausgehend Touren gebildet, welche zuletzt auf alle verfügbaren Lagerarbeiter verteilt werden. Innerhalb der VNS kommt eine Variante des *Variable Neighborhood Descent* (VND) als lokale Verbesserungskomponente zum Einsatz. Während über die definierten Nachbarschaftsstrukturen unterschiedliche potentielle Lösungen erzeugt werden, erfolgt deren Bewertung durch die Berechnung von konkreten Touren mittels eines für diesen Zweck entwickelten Dynamischen Programms. Dabei werden spezielle Eigenschaften der zugrundeliegenden Lagerstruktur ausgenutzt, um so in polynomieller Zeit die bestmögliche Wegführung durch das Lager berechnen zu können. Für die Zuordnung von Arbeitern zu den auf diese Weise berechneten Touren wird schließlich eine zusätzliche VNS verwendet, deren Aufgabe es ist, die notwendigen Touren derart zu verteilen, dass der letzte Artikel zum frühest möglichen Zeitpunkt ausgefasst werden kann.

Die anhand des implementierten Programms durchgeführten Tests zeigen, dass die erfolgte Tourenplanung wertvolle Ergebnisse liefert und die notwendige Rechenzeit niedrig gehalten werden kann. Getestet wurde mit Bezug auf eine Referenzlösung, welche auf Basis eines aus der Literatur entnommenen Ansatzes erzeugt werden konnte. Eine ausführliche Auswertung der Testergebnisse zeigte, dass die Anwendung des hier vorgestellten Ansatzes im Echtbetrieb als sehr vielversprechend gilt und erhebliche Einsparungen bezüglich der benötigten Arbeitszeit erreicht werden können. Insgesamt betrachtet wird ein effizientes und zielführendes Verfahren zur Lösung des vorliegenden Problems vorgestellt.

Abstract

Within this thesis a real-world problem related to a warehouse for spare parts is considered. Regarding several constraints typically stated by spare parts suppliers the time needed to collect articles should be minimized. Based on continuously arriving orders by customers predefined delivery times and capacity constraints have to be met. To accomplish this task efficient pickup tours need to be determined which is the main issue covered by this work which comes to an end with experimental results of a concrete implementation of the proposed approach.

The algorithm presented embeds a specifically developed *dynamic program* for computing optimal walks through the warehouse into a general *variable neighborhood search* (VNS) scheme. Several stages are used for first splitting up all orders, then creating tours out of the results and finally assigning them to available workers. The VNS uses a variant of the *variable neighborhood descent* (VND) as local improvement procedure. While the neighborhood structures defined are intended to produce candidate solutions, a dynamic program specially designed to compute optimal order picking tours is used to evaluate them. For this purpose properties specific to warehouses are exploited such to compute optimal routes within polynomial time. The final assignment of workers to tours is realized by another VNS. The task is then to find an allocation such that the last article to be picked up will be collected as early as possible.

Evaluations of experimental results of a concrete implementation indicate that the presented approach provides valuable pickup plans and computation times can be kept low. Moreover the performed test runs have been compared to a reference solution which was computed based on an approach found in relevant literature. A detailed analysis of the obtained results showed that the application of the proposed approach to real-world instances is promising whereas the savings with respect to working time can be kept high. Overall an efficient as well as effective approach is introduced to solve this real-world problem.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit ist aus einer Zusammenarbeit des *Instituts für Computergraphik und Algorithmen der Technischen Universität Wien* mit der Firma *Dataphone GmbH* entstanden, welche sich mit Problemen der Lagerverwaltung auseinandersetzt und in diesem Fall Aufgaben innerhalb des Lagers eines Ersatzteillieferanten analysieren soll. Neben der Verwaltung sämtlicher Stammdaten und Auftragsdaten des Lagers gilt es, die tatsächliche Anordnung von Artikeln im Lager zu erfassen und davon ausgehend die benötigte Zeit zum Ausfassen von bestellten Artikeln (im Weiteren wird dieser Vorgang auch Kommissionierung genannt) zu minimieren. Dadurch müssen auch die Arbeitsschritte des Lagerpersonals berücksichtigt und im System abgebildet werden.

Der Aufbau des Lagers gleicht im Wesentlichen dem, was man sich gemeinhin beim Gedanken an ein Warenlager vorstellt. Es sind parallel zueinander angeordnete Regale vorhanden, zwischen denen jeweils Gänge verlaufen, um entsprechend lagernde Artikel ausfassen zu können. An den beiden Enden jedes dieser Regalgänge verlaufen orthogonal dazu etwas breitere Hauptgänge (siehe dazu Abb. 1.1). Innerhalb dieses Gangsystems bewegen sich dann meist mehrere Arbeiter gleichzeitig, die unter Zuhilfenahme von Kommissionierungswagen diverse Artikel einsammeln. Welche Artikel benötigt werden, ergibt sich aus den jeweils vorliegenden Bestellungen, welche im Laufe des Arbeitstages durch Kunden in Auftrag gegeben werden. Da der Ersatzteillieferant gewisse Lieferzeiten einhalten will und muss, ist eine entsprechend effiziente Bearbeitung der Aufträge notwendig. Sobald die verlangten Artikel eingesammelt wurden, werden sie zu einer zentralen Stelle im Lager, der Verdichtungszone, gebracht, wo sie verpackt und für den Versand an die Kunden vorbereitet werden.

Ohne Unterstützung durch ein entsprechendes System obliegt es nun dem Lagerleiter die vorhandenen Aufträge auf seine Mitarbeiter derart aufzuteilen, dass in möglichst kurzer Zeit die gewünschten Artikel in der Verdichtungszone bereitstehen. Die Reihenfolge, in der Artikel ausgefasst werden, ist nicht weiter vorgegeben und so entscheidet jeder Lagerarbeiter selbst über seinen Weg durch das Lager. Diese Wege oder Touren durch das Lager sind nun genau jener Teil innerhalb des gesamten Bestellablaufs, der großes Optimierungspotential bietet.

Ziel der Arbeit ist es, das Ressourcenmanagement innerhalb des Lagers dahingehend zu optimieren, dass zunächst die Zusammenstellung von einzusammelnden Artikeln geschickt gewählt wird und damit im Weiteren das Erstellen von dafür kürzest möglichen Touren einen erheblichen Vorteil in der Planung des Ablaufs bringt. Ein wesent-

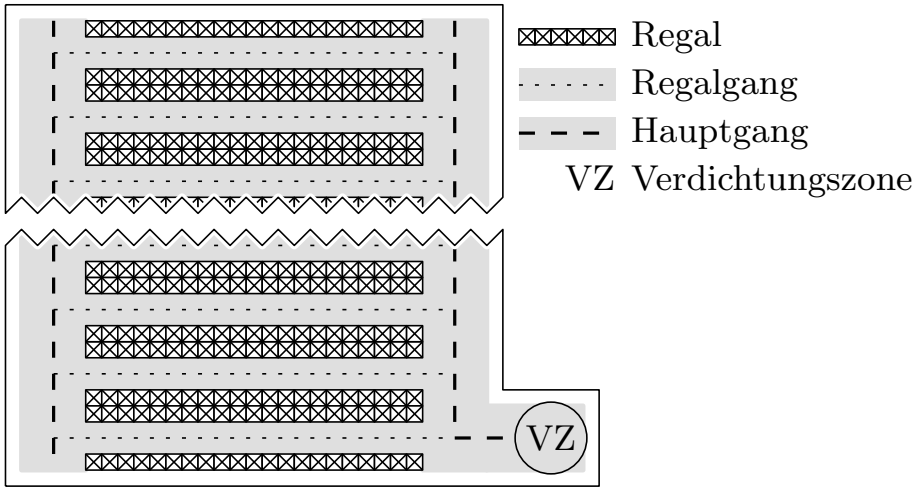


Abbildung 1.1: Schematische Darstellung des Lagers

licher Beitrag dazu ist ein von mir im Rahmen dieser Arbeit erstelltes Programm, das der Umsetzung sämtlicher Überlegungen innerhalb einer Heuristik dient. Dabei wird die vorhandene Problemstellung in vier algorithmischen Schritten bearbeitet. Es wird anhand bestimmter Kriterien, wie etwa Lieferzeit oder Platzbedarf eines Artikels, bestimmt, welche Artikel innerhalb einer Tour eines Arbeiters im Lager eingesammelt werden sollen. Der Schritt zur Berechnung von Touren ist dabei ein zentraler Bestandteil des Algorithmus und wurde mittels eines speziell entwickelten Dynamischen Programms umgesetzt. Sobald alle benötigten Touren berechnet wurden, können diese auf alle verfügbaren Lagermitarbeiter aufgeteilt werden, was im letzten Schritt erfolgt.

Im Anschluss an die Beschreibung des Algorithmus kann man anhand der Testergebnisse sehen, dass die Effizienz des Programms einen durchwegs positiven Eindruck vermittelt. Große Probleminstanzen können zwar sehr lange Laufzeiten im Bereich von mehreren Stunden benötigen, bis der Algorithmus keine weitere Verbesserung bringt, allerdings werden schon innerhalb der ersten Minuten gute Lösungen erzeugt und somit bleibt der Einsatz in der Praxis vielversprechend.

Zuerst werde ich nun eine detaillierte Problembeschreibung geben, die als Basis für meine Untersuchungen gedient hat. Im Anschluss daran möchte ich anhand von verwandten Arbeiten aus der Literatur einen Überblick über den Rahmen geben, in dem sich diese Arbeit bewegt. Ich werde im Weiteren beschreiben, aus welchen Teilen der letztlich verwendete Algorithmus aufgebaut ist und mit welchen Methoden der zuvor beschriebenen Arbeiten hierbei vorgegangen wird, wobei auch diese, wo sinnvoll und passend, im Detail behandelt werden. Im Zuge der Erläuterungen zum algorithmischen Verlauf, werde ich auch auf einige spezielle Probleme eingehen, die in diesem Zusammenhang zu lösen waren. Abschließend werden dann die Ergebnisse, die mit dem vorhandenen Programm erzielt werden konnten und aufgrund statistischer Auswertungen mehrerer Testläufe zustande gekommen sind, die Arbeit abschließen.

2 Problembeschreibung

Die folgende Aufgabenstellung stammt in ihrer ursprünglichen Form von der Firma *Dataphone GmbH* und wurde in Zusammenarbeit mit Mitarbeitern des *Instituts für Computergraphik und Algorithmen der Technischen Universität Wien* angepasst, um innerhalb eines gemeinsamen Projekts einen Lösungsansatz dafür zu entwerfen.

Im Zuge der Reorganisation und Erweiterung des Lagers eines Ersatzteillieferanten soll eine automationstechnisch unterstützte Lagerverwaltung eingeführt und ausgebaut werden, die unter anderem die Möglichkeit bieten soll, einzelne Prozessabläufe im Lageralltag zu rationalisieren. Zu diesem Zweck wurden sämtliche Artikel beziehungsweise jene Ladungsträger, auf denen sich die Artikel befinden, mit maschinenlesbaren Kodierungen, konkret mit Barcodes, versehen und eine zentrale Lagerverwaltungssoftware eingeführt. Auf Basis einer im Hintergrund eingerichteten Datenbank erlaubt die grafische Oberfläche der Software die Erfassung und Verwaltung von Stammdaten und Bestelldaten sowie den Zugriff auf den aktuellen Lagerstand.

In einer ersten Ausbaustufe soll vor allem die Zusammenstellung der einzelnen von Kunden in Auftrag gegebenen Bestellungen möglichst effizient realisiert werden, wobei folgende Schritte zu berücksichtigen sind:

1. Ein Kunde gibt eine Ersatzteillieferung in Auftrag.
2. Für jeden Auftrag fasst ein Lagerarbeiter die erforderliche Anzahl der bestellten Artikel aus dem Lager aus und bringt sie in eine so genannte Verdichtungszone.
3. Ein dieser Verdichtungszone zugeteilter Mitarbeiter packt alle zu den jeweiligen Bestellungen gehörenden Einzelteile in entsprechend dimensionierte Kisten, versieht diese mit Adresstickets und leitet sie an den Lieferanten (Paketdienst, Post, etc.) weiter.

Während der erste und der dritte Punkt dieser Abarbeitungsreihenfolge derzeit schon verlässlich und ohne gröbere Verzögerungen ablaufen, stellt der zweite Punkt den Flaschenhals in der Zusammenstellung der Aufträge dar, da hierbei die Mitarbeiter große Strecken im Lager zurücklegen müssen, um alle Einzelteile einzusammeln. Da im Normalfall bis zu sechs Mitarbeiter gleichzeitig an bis zu 1000 täglichen Aufträgen bestehend aus je fünf unterschiedlichen Artikeln arbeiten, gibt es durch geschickte Aufteilung der Aufträge ein großes Einsparungspotential, sofern eine geschickte Einteilung von Artikeln zu Touren vorgenommen wird. Folgender neu gestalteter Ablauf soll daher realisiert werden:

1. Mehrere Kunden geben (unabhängig voneinander) jeweils eine Ersatzteillieferung in Auftrag.
2. Alle derzeit im Lager tätigen Mitarbeiter bekommen jeweils eine Liste von Artikeln, die sie entlang einer vorberechneten Tour im Lager auf Kommissionierungswagen laden sollen. Diese Artikel können im Allgemeinen auch zu unterschiedlichen Aufträgen gehören.
3. Nach Abarbeitung dieser Liste stellt jeder Mitarbeiter den von ihm bedienten Kommissionierungswagen in der Verdichtungszone ab und entnimmt dort einen weiteren, allerdings leeren Kommissionierungswagen, um sich auf eine neue Tour zum Ausfassen weiterer Artikel zu machen.
4. Der Mitarbeiter in der Verdichtungszone verfährt weiterhin so, dass er alle Artikel eines Auftrags in entsprechende Schachteln verpackt und an den Botendienst übergibt. Dabei muss er allerdings beachten, dass nun die Artikel, die einer Bestellung zugeordnet sind, auf unterschiedlichen Kommissionierungswagen zwischengelagert sein können.

Die Verdichtungszone selbst (siehe Abb. 2.1) ist in drei Zonen unterteilt, die als eine Art Zwischenlager verstanden werden können. Von dort holen die Lagermitarbeiter leere Kommissionierungswagen, um Artikel aus dem Lager auszufassen und stellen diese befüllt wieder dort ab. Mitarbeiter, die für Verpackung und Versand zuständig sind, können dann von dort die entsprechenden Waren abholen und die Bestellabwicklung fortsetzen. Da diese einzelnen Zonen entsprechend weitläufig sind, um genügend Platz zu bieten, ist vorgesehen, dass alle zu einer Bestellung gehörigen Artikel innerhalb derselben Zone abgeliefert werden sollen. Dies verringert den Zeitaufwand für die Verpackung und macht die Arbeit in der Verdichtungszone wesentlich einfacher und effizienter.

Gegenstand dieser Arbeit ist nun die Entwicklung eines Lösungsansatzes, mit dessen Hilfe es möglich wird, die Aufteilung der Bestellungen auf unterschiedliche Mitarbeiter derart zu optimieren, dass die Touren der Kommissionierer möglichst kurz werden, wodurch gleichzeitig der zeitliche Abstand zweier Entnahmen einzelner Artikel aus dem Lager minimiert wird. Ausschlaggebend hierfür ist, dass durch diese Aufteilung der Bestellungen auf mehrere gleichzeitig im Lager arbeitende Personen bei entsprechender Optimierung große Einsparungen erreicht werden können, da für die Mitarbeiter besser organisierte Touren mit kürzeren Wegen möglich sind. Weiters soll erreicht werden, dass die jeweils zu einer gemeinsamen Bestellung gehörenden Artikel ungefähr zeitgleich in der Verdichtungszone bereitgestellt und im Weiteren verpackt werden, um mehrere Aufträge ohne den Einsatz größerer Zwischenlager gleichzeitig bearbeiten zu können.

Neben dieser prinzipiellen Aufgabenstellung müssen jedoch noch mehrere Nebenbedingungen berücksichtigt werden, die einen entscheidenden Einfluss auf die tatsächliche Realisierung der einzelnen Kommissionierungstouren haben:

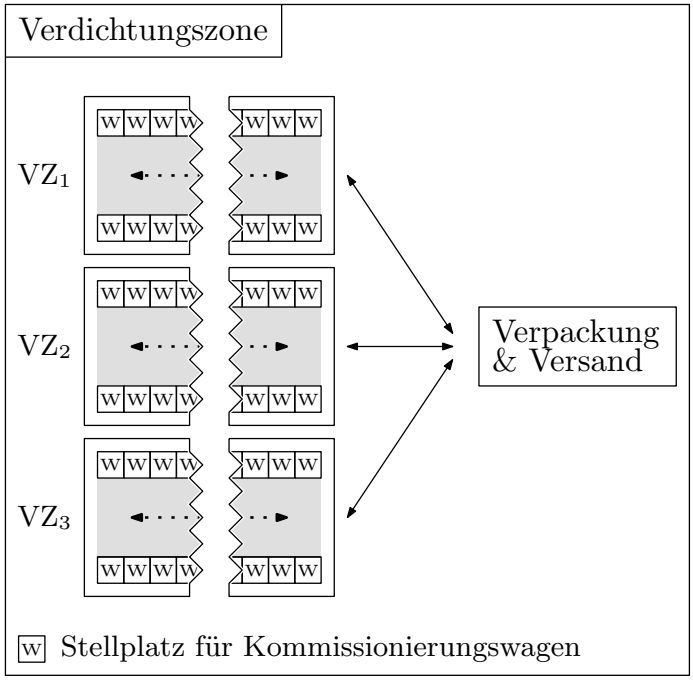


Abbildung 2.1: Schematische Darstellung der Verdichtungszone

- Die Kommissionierungswagen können einerseits nur bis zu einer gewissen Kapazität beladen werden und andererseits ist es aufgrund ihrer Bauweise nur vorgesehen, dass sie sich in eine Richtung bewegen. Wegen des schmalen Gangsystems ist ein Umkehren innerhalb eines Ganges nicht ohne Weiteres möglich.
- Die Überkreuzung zweier Kommissionierungstouren innerhalb eines Regalganges sollte möglichst vermieden werden, da aufgrund der Platzbeschränkung ein Überholen oder aneinander Vorbeifahren nicht möglich ist.
- Manche Artikel können an mehreren Positionen im Lager verfügbar sein. Abhängig von der bestellten Menge müssen oder können dann unter Umständen auch mehrere solcher Lagerplätze für einen Artikel angefahren werden.
- Einen weiteren Aspekt birgt die Tatsache, dass nicht alle an einem Tag zu bearbeitenden Bestellungen zu Arbeitsbeginn bereits bekannt sind, da noch im Laufe des Tages weitere Bestellungen eintreffen können. Dies entspricht sogar eher dem Normalfall, wodurch Anzahl und Struktur der zu bearbeitenden Aufträge im Laufe des Tages stark variieren. Ein Lösungsansatz soll das insofern berücksichtigen, als die Berechnungen jederzeit mit angepassten Eingabewerten wieder gestartet werden können sollen. Das heißt, es wird immer mit den aktuell offenen Bestellungen neu gerechnet.
- Die Bearbeitung von Bestellungen sollte nach Möglichkeit in der Reihenfolge ihres Eintreffens geschehen, um die Lieferzeiten möglichst sicher einhalten zu können.

- Die zuvor beschriebene Einteilung der Verdichtungszone in drei Zonen muss derart berücksichtigt werden, dass von den Kommissionierern ausnahmslos alle zu einer Bestellung gehörigen Artikel in derselben dieser drei Zonen abgeliefert werden.

Durch mobile Datenerfassungsgeräte soll der Optimierungsprozess unterstützt werden, da der Einsatz dieser Geräte es ermöglicht, die Entnahme eines Artikels beziehungsweise Abgabe eines Kommissionierungswagens in der Verdichtungszone in Echtzeit nachzuvollziehen. Dieses neue Lagerverwaltungssystem soll das derzeitige Verfahren ablösen, bei dem alle Entscheidungen durch Mitarbeiter getroffen werden.

Unter Berücksichtigung dieser Einschränkungen gilt es nun vorrangig die Berechnung von kürzesten Touren für die Mitarbeiter zu behandeln.

Mögliche Erweiterungen

Die folgende Auflistung eröffnet mögliche Erweiterungen der gegebenen Problembeschreibung, wobei deren Umsetzung aufgrund von massivem Mehraufwand nicht erfolgte. Überdies beeinflussen diese Aspekte den Kern der Arbeit nicht, wodurch die zentralen Ergebnisse des vorgestellten Verfahrens ihre Aussagekraft behalten.

Folgende Punkte sind als zusätzliche Funktionalitäten und Optionen denkbar:

- Die im Lager verfügbaren Kommissionierungswagen haben durchaus unterschiedliche bauartliche Voraussetzungen, was sich auf damit verbundene Eigenschaften, wie Geschwindigkeit, Ladefläche oder Manövrierfähigkeit, auswirkt. Ein entwickelter Lösungsansatz dafür sollte dann genügend Flexibilität aufweisen, um damit umgehen zu können. Die Palette an Transportmitteln reicht dabei von „einkaufswagenähnlichen“ Fahrzeugen bis hin zu Gabelstaplern.
- Die eingeschränkte Bewegungsfreiheit aufgrund der sehr schmalen Gänge im Lager verhindert es, dass zwei Kommissionierungswagen aneinander vorbeifahren können. Um Kollisionen auszuschließen, wäre es denkbar einen Algorithmus zu entwerfen, der entsprechende Zustände verbietet.
- Ein weiterer Punkt in Zusammenhang mit den im Lager verwendeten Fahrzeugen bezieht sich auf die Sicherheitsbestimmungen in Bezug auf Gabelstapler. Diese schreiben es prinzipiell vor, dass sich in einem Gang kein anderer Mitarbeiter gleichzeitig mit einem Gabelstapler aufhalten darf. Die Sperrung von solchen Gängen muss dann natürlich berücksichtigt werden.
- Es kann der Fall auftreten, dass der Lagerleiter befindet, einen Lagerplatz mit einem anderen Artikel belegen zu wollen. In einem solchen Fall muss die Möglichkeit gegeben sein, dass der anderweitig benötigte Lagerplatz möglichst schnell (gleichzeitig aber kosteneffizient) leer geräumt wird.

- Weiters liegt durch das ständige Eintreffen von Bestellungen im Laufe eines Arbeitstages der Anspruch an eine *Online-Optimierungsaufgabe* nahe. Das heißt, dass aktuell eintreffende Bestellungen jederzeit in den Optimierungsprozess aufgenommen werden können müssen. Im Unterschied zu einem kompletten Neustart der Berechnungen müssen in diesem Fall die bereits getätigten Optimierungen in den weiteren Verlauf des Programms einfließen.
- Außerdem kann es vorkommen, dass während der Zusammenstellung der einzelnen Aufträge unvorhersehbare Vorkommnisse auftreten, auf die entsprechend reagiert werden muss. Artikel können beispielsweise fehlerhaft, kaputt oder gar nicht vorhanden sein. In einem solchen Fall muss der entsprechende Artikel von einer anderen Position im Lager entnommen werden oder möglicherweise sogar neu eingelagert werden. Wenn ein Lagerarbeiter mehr Zeit benötigt als angenommen, verzögert sich natürlich auch die Kommissionierung und entsprechende Änderungen müssen berücksichtigt werden
- Eine zusätzliche Straffung der Nebenbedingungen ist durch die Garantie des Ersatzteillieferanten gegeben, fixe Lieferzeiten an seine Kunden einzuhalten. Diese sind auf alle Fälle einzuhalten. Es muss daher sichergestellt sein, dass jeder Auftrag bis zu einem vorgegebenen Termin verpackt ist, um verschickt werden zu können.

3 Verwandte Arbeiten

Aus der detaillierten Problemspezifikation folgt unmittelbar, dass es sich bei dieser Aufgabenstellung um ein mit dem *Vehicle Routing Problem* (VRP) [18] verwandtes Problem handelt, das zu den im klassischen VRP definierten Nebenbedingungen noch weitere domänenspezifische Einschränkungen enthält. Die ursprüngliche Variante des VRP verlangt, dass ausgehend von einem Depot Kunden mit unterschiedlichen Bestellungen beliefert werden müssen. Um den Transport des Ladeguts zu gewährleisten, steht eine Flotte von Lkws zur Verfügung. Gesucht ist eine Einteilung von Lkws zu Kunden, sodass die insgesamt zurückgelegte Wegstrecke der Lieferfahrzeuge minimiert wird. Auch Parallelen zu in der Literatur häufig auftretenden Varianten des VRP können festgestellt werden. Zum Beispiel darf beim *Capacitated VRP* jeder Lkw nur eine gewisse Last transportieren, beim *Split Delivery VRP* dürfen hingegen alle Kunden von beliebig vielen Lkws angefahren werden. Es konnte allerdings keine Arbeit gefunden werden, die alle diese Varianten ineinander vereint und zusätzlich noch die bereits beschriebenen domänenspezifischen Nebenbedingungen, wie etwa die beschränkte Breite von Gängen, berücksichtigt.

Neben der offensichtlichen Verwandtschaft mit Varianten des VRP, besteht auch eine Verbindung zu *Generalisierten Netzwerkproblemen* [9]. So kann die Möglichkeit einen Artikel von unterschiedlichen Lagerplätzen zu holen als implizite Clusterbildung verstanden werden. Jeweils nur ein Knoten aus einem solchen Cluster soll besucht werden. Unter diesem Blickwinkel ist eine Verbindung zum *Generalized Traveling Salesman Problem* (GTSP) [10] offensichtlich, bei dem es gilt, jeweils einen Knoten pro Cluster auszuwählen und anschließend eine Tour zu berechnen, sodass jeder Cluster genau einmal besucht wird.

Wie später genauer erklärt wird, basiert der hier vorgeschlagene Lösungsansatz auf *Variabler Nachbarschaftssuche (engl.: Variable Neighborhood Search)* (VNS) [12] mit integriertem *Variable Neighborhood Descent* (VND). Bei VNS/VND handelt es sich um Metaheuristiken, die auf der Idee aufbauen, dass ein globales Optimum stets ein lokales Optimum bezüglich aller möglichen Nachbarschaftsfunktionen ist. Unter einer Nachbarschaftsfunktion versteht man eine Rechenvorschrift, die es ermöglicht aus einer gegebenen Lösung x eine neue Lösung x' zu berechnen, wobei sich x und x' nur in einigen (wenigen) Merkmalen unterscheiden. VND durchsucht systematisch eine Menge von gegebenen Nachbarschaften, um so zu einer möglichst guten Lösung zu gelangen. Weiters wird durch VNS ein so genannter *Shaking*-Mechanismus angewendet, der dazu dient, festgefahrene Suchläufe durch das Einbringen von zufälligen Änderungen zu verbessern.

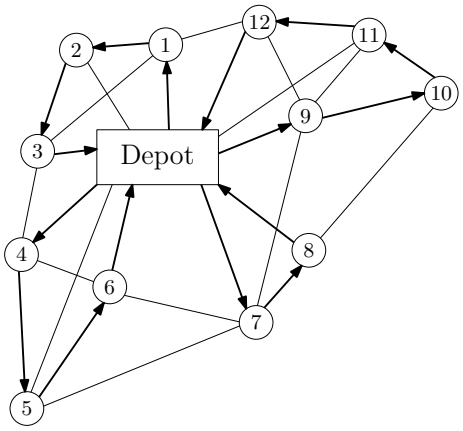


Abbildung 3.1: Vehicle Routing Problem (VRP)

3.1 Vehicle Routing Problem

Beim klassischen *Vehicle Routing Problem* (VRP) handelt es sich um eine aus der Transportlogistik stammende Problemstellung, welche die Verteilung von Gütern ausgehend von einem Depot an Kunden mit unterschiedlichen Standorten beschreibt (siehe Abb. 3.1). Dabei fordert jeder Kunde eine gewisse Menge eines bestimmten Gutes an. Das angestrebte Ziel hierbei ist es, Routen in der Form zu erstellen, dass die Gesamtkosten zur Anlieferung aller Güter minimiert werden. Natürlich kann diese Formulierung auch entsprechend umgekehrt werden und so nicht Kunden beliefert werden, sondern gewisse Güter eingesammelt werden. In der Regel unterliegt die Routenerstellung gewissen Kriterien, wie etwa der Kapazität des jeweiligen Transportmittels oder zeitlichen Einschränkungen. Das VRP gehört in die Klasse der $\mathcal{NP}$-schweren Probleme.

3.1.1 Capacitated Vehicle Routing Problem

Eine Variante des VRP ist das *Capacitated Vehicle Routing Problem* (CVRP) [17], welches die Kapazitäten der verwendeten Fahrzeuge einschränkt. Das in [17] beschriebene CVRP verlangt die Auslieferung eines einzigen Gutes an n Kunden ausgehend von einem Depot $\{0\}$ unter Verwendung von k unabhängigen Transportmitteln mit jeweils identischer Kapazität C, wobei jedem Kunden $i \in N = \{1, \ldots, n\}$ die Menge d_i zugestellt wird. Aus kombinatorischer Sicht handelt es sich bei der Lösung um eine Aufteilung $\{R_1, \ldots, R_k\}$ von N in k Routen, wobei jede dieser Routen $\sum_{j \in R_i} d_j \leq C$, $1 \leq i \leq k$ erfüllen muss.

Zum klassischen, bereits $\mathcal{NP}$-schweren VRP kommen also noch zusätzliche Einschränkungen hinzu, welche zu berücksichtigen sind. Das Problem wird dadurch allerdings nicht leichter.

3.1.2 Split Delivery Vehicle Routing Problem

Das *Split Delivery Vehicle Routing Problem* (SDVRP) [2] beschreibt jene Form des VRP, bei der ein Kunde mehrmals angefahren werden darf oder sogar muss. Wieder sind mehrere Lieferanten im Einsatz und jedes Transportmittel hat die gleiche Kapazität. Nun kann es allerdings vorkommen, dass die benötigte Menge eines Kunden die Kapazität eines einzelnen Transportmittels übersteigt oder aber auch einfach aus Effizienzgründen zwei Lieferanten einen Kunden mit Teillieferungen beliefern.

Wie bereits bei Dror und Trudeau in [7] und [8] gezeigt, können durch die Aufteilung von Lieferungen beträchtliche Einsparungen erzielt werden, sowie auch die Anzahl an Lieferanten reduziert werden. Dennoch bleibt das Problem an sich $\mathcal{NP}$-schwer.

Die meisten Lösungsansätze gehen von der Annahme aus, dass die so genannte Dreiecksungleichung erfüllt ist. Geht man von der Kostenberechnung auf Basis von Wegstrecken aus, so bedeutet dies, dass der direkte Weg zwischen zwei Knoten immer der kostengünstigste ist. Formal lässt sich das ganz leicht durch

$$c_{ij} + c_{jk} \geq c_{ik} \quad \text{, mit } i, j, k \in \{1, \ldots, n\}$$

ausdrücken, wobei n die Anzahl an Kunden repräsentiert und c_{ij} die Kosten für den Weg von Kunde i zu Kunde j angibt. Es wird dabei angenommen, dass $c_{ij} = c_{ji}$ gilt.

Ein interessanter Aspekt der zuvor erwähnten Arbeiten [7] und [8] ist die Erkenntnis, dass es bei geltender Dreiecksungleichung eine optimale Lösung gibt, bei der je zwei Routen nie mehr als zwei gemeinsame Zielorte mit Teillieferungen anfahren.

3.1.3 Vehicle Routing Problem with Time Windows

Eine weitere Variante des VRP behandelt jene Problemstellung, bei der die Kunden innerhalb eines definierten Zeitfensters beliefert werden müssen. Bei diesem Problem, dem *Vehicle Routing Problem with Time Windows* (VRPTW) [19], kann sich die Reihenfolge der zu beliefernden Kunden daher nicht ausschließlich aus den Routen mit kürzesten Wegen definieren, sondern wird zusätzlich von einer zeitlichen Beschränkung beeinflusst. Auch dieses kombinatorische Problem ist $\mathcal{NP}$-schwer und ist als Erweiterung des klassischen VRP eine sehr häufig vorkommende Problemstellung im Bereich der Logistik, wo es neben der Belieferung aller Kunden mit minimalen Kosten nun auch zu berücksichtigen gilt, dass jeder Kunde nur innerhalb eines Zeitfensters $[a_i, b_i]$ beliefert werden kann, wobei a_i der früheste und b_i der späteste Zeitpunkt ist, zu dem Kunde i beliefert werden kann.

Da das Ziel die Minimierung des notwendigen Zeitaufwandes ist, muss die Dreiecksungleichung in konkreten Szenarien meist nicht erfüllt werden, da kürzere Strecken nicht immer gleichbedeutend mit kürzerem Zeitaufwand sind. Das heißt, dass $t_{ij} + t_{jk} \geq t_{ik}$, wobei t_{ij} den Zeitaufwand für die Strecke von Kunde i zu Kunde j bezeichnet, in

diesem Fall nicht zwingend gilt.

3.2 Generalisierte Netzwerkprobleme

Zur Klasse der *Netzwerkprobleme* zählt man unter anderen das *Minimum Spanning Tree Problem*, das *Traveling Salesman Problem* sowie das Problem kürzester Wege. Allgemein besteht die Aufgabe bei einem *Netzwerkproblem* darin, einen optimalen Teilgraphen F eines Graphen G unter Einhaltung gewisser Randbedingungen zu finden. Wenn man nun von *Generalisierten Netzwerkproblemen* spricht, so werden dabei die Knoten des Graphen G in Gruppen (engl.: Cluster) eingeteilt und die Randbedingungen auf dieser Basis formuliert. Hier kann dann beispielsweise ein minimaler Spannbaum über alle Cluster oder ein *Hamiltonkreis* (siehe dazu auch Kapitel 3.3) für alle Cluster gesucht werden.

Formal ist bei einem *Generalisierten Netzwerkproblem* ein (un)gerichteter Graph $G = (V, E)$, bestehend aus einer Menge von Knoten $V = \{1, \ldots, n\}$ und der Kantenmenge $E \subseteq \{(i, j) : i, j \in V\}$, gegeben. $E(S) = \{(i, j) \in E : i, j \in S\}$ sei dann jene Teilmenge von Kanten, die ihre Endpunkte in $S \subseteq V$ haben. Je nachdem welche Randbedingungen definiert werden, entsteht ein entsprechendes konkretes Problem.

3.3 Traveling Salesman Problem

Beim sogenannten *Traveling Salesman Problem* (TSP) [1] handelt es sich um ein weiteres kombinatorisches Optimierungsproblem, das in der Klasse der $\mathcal{NP}$-vollständigen Probleme enthalten ist. Das Ziel hierbei ist es, alle vorhandenen Orte innerhalb einer Rundreise, bei der also der Startort gleich dem Zielort ist, zu besuchen und dabei eine möglichst kurze Strecke zu finden.

Um dieses Problem in ein mathematisches Modell zu bringen, bietet sich die Übersetzung in ein graphentheoretisches Problem an, wobei die Orte den Knoten und die Verbindungen der Orte den Kanten des Graphen entsprechen. Jede Kante besitzt eine bestimmte Länge, womit die mit ihrer Verwendung verbundenen Kosten definiert sind. Gesucht ist nun eine Tour, welche ein Kreis im Graphen ist, der jeden Knoten genau einmal enthält. Eine solche Tour wird auch *Hamiltonkreis* genannt.

Der Einfachheit halber wird für dieses Problem meist angenommen, dass der zugrunde liegende Graph vollständig ist, also je zwei Knoten durch eine Kante verbunden sind. Sollte der Graph nicht vollständig sein, kann man sich dadurch helfen, die fehlenden Kanten einzufügen und mit so hohen Kosten zu belasten, dass sie in einer minimalen Tour nicht vorkommen würden, es sei denn es wäre sonst keine Tour auffindbar. Allerdings ist dann zu beachten, dass möglicherweise unlösbare Instanzen durch diese Anpassung lösbar werden. Die Komplexität des Suchraumes unter

Berücksichtigung der Anzahl n an Knoten im Graphen ist dabei in $O(n^n)$.

Das *Generalized Traveling Salesman Problem* [10] behandelt die leicht abgeänderte Variante des eigentlichen Problems, welche alle zu besuchenden Orte in Gruppen einteilt und verlangt, dass genau ein Ort aus jeder Gruppe besucht werden muss. Welcher das ist, kann frei gewählt werden. Es sind dafür also zwei miteinander verknüpfte große Schritte notwendig, nämlich einerseits die Auswahl einer Teilmenge von Knoten des zugrunde liegenden Graphen, wobei aus jeder Gruppe von Orten jeweils genau einer in dieser Teilmenge enthalten ist und andererseits die Bestimmung einer Tour mit minimalen Kosten innerhalb des Teilgraphen, der aus den ausgewählten Knoten entstanden ist.

3.4 Bekannte Lösungsansätze

Zur Lösung des *Vehicle Routing Problem* gibt es einige Ansätze, die unter anderem *Variable Nachbarschaftssuche* oder *Tabu-Suche* einsetzen. Hierbei wird der Einsatz solcher heuristischer Verfahren gewählt, da aufgrund der Problemstellung bereits für kleine Probleminstanzen eine Vielzahl an möglichen Lösungen existiert, gleichzeitig aber kein Algorithmus bekannt ist, der in polynomieller Zeit eine optimale Lösung konstruiert.

In [13] wird etwa beschrieben, wie mittels *Tabu-Suche* nach einer Lösung für ein SDVRP kombiniert mit einem VRPTW gesucht werden kann, für die die Anzahl der verwendeten Fahrzeuge sowie die Länge der insgesamt zurückgelegten Strecke minimiert werden.

Auch exakte Verfahren wie *Ganzzahlige Lineare Optimierung (engl.: Integer Linear Programming)* (ILP) und *Branch & Bound* beziehungsweise *Branch & Cut* werden häufig zur Lösung herangezogen. Ausgehend von einer ILP-Formulierung beschreibt etwa [15] einen solchen Ansatz zur Lösung von CVRP-Instanzen mittels *Branch & Cut*.

In Zusammenhang mit Warenlagern, deren Anordnung in klassischer rechteckiger Form vorliegt, ist auch das Aufsuchen von Touren ein interessantes Teilproblem im Zuge der Optimierung von Lagerabläufen. In [5] wird ein Verfahren, die sogenannte *S-Shape*-Heuristik, betrachtet, das der Erstellung solcher Touren dient. Dabei bewegen sich die Lagerarbeiter S-förmig durch das Lager, das heißt, dass ein Gang komplett durchquert wird, sobald er einmal betreten wurde. Interessant ist dieser Ansatz besonders deshalb, da er später für Vergleiche zu dem in dieser Arbeit vorgestellten Verfahren herangezogen wird. Betrachtungen der *S-Shape*-Heuristik, sowie Varianten davon finden sich auch in [4], wo außerdem auf die Komplexität solcher Lagerabläufe hingewiesen wird und klar hervorgeht, dass entsprechende Probleme stets sehr speziell sind. Es gibt also kein Konzept und kein globales Optimierungsmodell für eine systematische Behandlung ähnlicher Situationen.

4 Dynamische Programmierung

Unter dem Begriff der Dynamischen Programmierung versteht man ein algorithmisches Verfahren, bei dem in mehreren voneinander abhängigen Schritten Entscheidungen getroffen werden, die zur optimalen Lösung eines Problems führen. Dabei ist stets die Lösung eines Problems unter Ausnützung des Wissens über bereits gelöste Teilprobleme ein wesentlicher Bestandteil. Die in den einzelnen Berechnungsschritten zu lösenden Teilprobleme sind außerdem immer abhängig von den zuvor schon gelösten. Sobald zu einem Teilproblem eine Lösung berechnet wurde, wird diese mitprotokolliert, um für die spätere Verwendung abgerufen werden zu können und nicht wiederholt berechnet werden zu müssen. So arbeitet man sich schrittweise anhand optimaler Teillösungen zur Lösung des Gesamtproblems vor.

Der Ansatz der Dynamischen Programmierung basiert auf folgendem ursprünglich von Bellman formulierten Postulat [3]:

> „Ein optimales Verfahren hat die Eigenschaft, dass, wie auch immer der Anfangszustand und die erste Entscheidung ausfielen, die folgenden Entscheidungen für eine optimale Lösung sich auf den Zustand, der aus der ersten Entscheidung resultiert, beziehen müssen."

Wesentlich dabei ist, dass alle Entscheidungen des Algorithmus von bereits zuvor getroffenen abhängen. Das bedeutet, dass die Lösung eines Teilproblems als Ausgangspunkt stets die Lösung eines vorangehenden Teilproblems heranziehen muss. Folgt man diesem Prinzip, dann versucht man das Problem derart zu zerlegen, dass jedes der entstehenden Teilprobleme optimal gelöst werden kann. Am Ende wird die optimale Lösung des initial formulierten Problems erreicht und kann durch die zuvor durchgeführten Schritte zusammengesetzt werden.

Ein ähnlicher Ansatz, der ebenso das Zerlegen eines Problems in Teilprobleme verfolgt, ist *Divide & Conquer*. Dieses Verfahren unterscheidet sich allerdings wesentlich von Dynamischer Programmierung, da hier per Definition keine Abhängigkeit zwischen den einzelnen Teilproblemen bestehen muss.

Kürzeste Wege in einem Graphen

Ein gutes Beispiel für Dynamische Programmierung bietet der Algorithmus für kürzeste Wege in einem Graphen, welcher 1959 von Dijkstra [6] vorgestellt wurde. Ge-

geben sei ein Graph $G = (V, E)$, bestehend aus einer Menge $V = \{v_1, v_2, \ldots, v_n\}$ von Knoten und einer Menge E von Kanten. Weiters bezeichne e_{ij} eine Kante von v_i nach v_j und c_{ij} deren Kosten. Der Algorithmus von Dijkstra berechnet in diesem Graph den jeweils kürzesten Weg von einem Knoten $v_{start} \in V$ zu allen anderen Knoten $v \in V$.

Anfangs wird für jeden Knoten im Graph dessen initial bekannte Distanz zum Anfangsknoten mit

$$\begin{aligned} w(V_{start}) &= 0 && \text{und} \\ w(V_i) &= \infty && V_i \in V, i \neq start \end{aligned}$$

festgehalten. Ausgehend vom ersten Knoten V_{start} des Weges und beginnend mit diesem wird nun jeweils immer ein Knoten V_f „fixiert“ und alle direkten Nachfolger zur Fixierung vorgemerkt, also „markiert“. Dabei wird für jeden Knoten V_i die Distanz $w(V_i)$ aktualisiert mit

$$w(V_i) = \min\{w(V_i), w(V_f) + c_{if}\}$$

und damit bestimmt, ob ein kürzerer Weg bis zum Knoten V_i existiert. Wenn alle Nachfolger von V_f in dieser Form bearbeitet wurden, wird der näheste davon fixiert und dessen Nachfolger untersucht. Ein Knoten, der bereits fixiert wurde, wird nicht mehr aktualisiert. Durch dieses Vorgehen nähert man sich Schritt für Schritt dem Zielknoten. Sobald dieser fixiert wird, hat man das Ende erreicht und einen optimalen Weg gefunden. Die Rekonstruktion des Weges ist durch Rückverfolgung über die Gleichung

$$w(V_i) + c_{ij} = w(V_j)$$

einfach möglich, womit sich natürlich auch mehrere mögliche kürzeste Wege ergeben können.

Anhand dieses ursprünglich von Dijkstra vorgestellten Algorithmus ist es daher möglich, durch schrittweises Lösen voneinander abhängiger Teilprobleme in einem Graphen den kürzesten Weg zwischen zwei Knoten zu bestimmen, sofern dieser existiert.

Die Dynamische Programmierung folgt also einer einfachen Strategie:

1. Aufteilung des Problems in kleinere, voneinander abhängige Probleme
2. Finden einer optimalen Lösung für die Teilprobleme
3. Kombinieren der optimalen Teillösungen zur Berechnung der optimalen Lösung für das Gesamtproblem

Es sei hier nochmals auf den wesentlichen Aspekt der Abhängigkeit von Teilproblemen hingewiesen, welcher etwa bei *Divide & Conquer* nicht Teil des Konzepts ist.

5 Variable Nachbarschaftssuche

Variable Nachbarschaftssuche (VNS) ist eine vergleichsweise junge Metaheuristik, die ihren Ursprung um das Jahr 1995 hat, in dem sie erstmalig vorgestellt wurde (siehe [16] und [11]).

Grundsätzlich kann ein Optimierungsproblem allgemein formuliert werden als

$$\min \{f(x)|x \in X, X \subseteq Z\}. \tag{5.1}$$

Z, X, x und f bezeichnen dabei den Lösungsraum, die Menge der gültigen Lösungen, eine konkrete Lösung und eine Funktion, die jeder Lösung $x \in X$ einen reellen Zielfunktionswert zuordnet. Meist ist eine explizite Durchsuchung von X auf Basis von (vollständigen) Enumerationsverfahren nicht applikabel. Zudem sind viele der in der Praxis relevanten Probleme $\mathcal{NP}$-schwer, was impliziert, dass die Existenz eines polynomiellen Algorithmus zur Lösung dieser Probleme höchst unwahrscheinlich ist. Daher kommen häufig (Meta-)Heuristiken zum Einsatz.

VNS bietet dabei ein sehr einfach gehaltenes Verfahren mit vielen Ausbaumöglichkeiten und folgt dem einfachen Prinzip, bereits vorhandene Lösungen einer leichten Veränderung zu unterziehen, um somit neue, möglicherweise bessere, Lösungen zu erhalten. Um dabei systematisch vorgehen zu können, werden so genannte Nachbarschaften definiert, welche im Wesentlichen auf Rechenvorschriften basieren, wie eine vorhandene Lösung abzuändern ist, um neue, möglicherweise bessere, zu erreichen.

Sei $\mathcal{N} = \{\mathcal{N}_1, \ldots, \mathcal{N}_{k_{max}}\}$, $k_{max} \geq 1$, als endliche Menge vorgegebener Nachbarschaftsstrukturen, sowie $\mathcal{N}_k(x)$ als Menge der Lösungen der k-ten Nachbarschaft von x definiert. Als globales Optimum wird jene Lösung x_{opt} bezeichnet, für die

$$x_{opt} = \min \{f(x)|x \in X, X \subseteq Z\}$$

gilt. Ein lokales Minimum $x' \in X$ in Bezug auf $\mathcal{N}_k$ ist gegeben, wenn es keine Lösung $x \in \mathcal{N}_k(x') \subseteq X$ mit $f(x) < f(x')$ gibt.

Die Variable Nachbarschaftssuche beruht auf folgenden drei einfachen Tatsachen [12]:

(i) Ein lokales Minimum bezüglich einer Nachbarschaftsstruktur ist nicht notwendigerweise auch ein lokales Minimum bezüglich einer anderen.

(ii) Ein globales Minimum muss gleichzeitig ein lokales Minimum bezüglich aller Nachbarschaftsstrukturen sein.

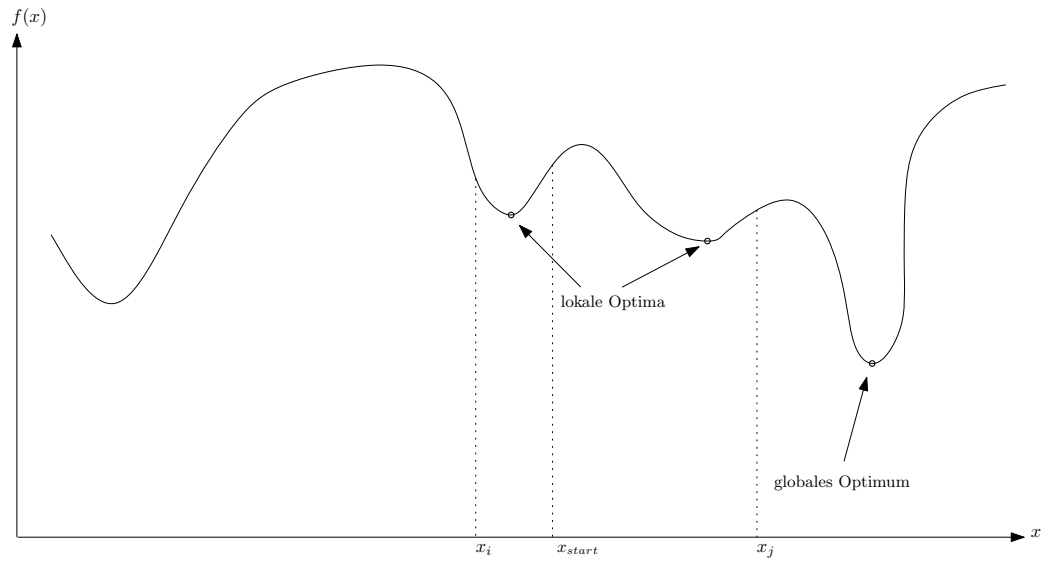


Abbildung 5.1: Lokale Optima und globales Optimum im Verlauf der Zielfunktion

(iii) Sehr viele Probleme bieten die Eigenschaft, dass lokale Optima nahe beieinander liegen.

Der letzte Punkt basiert auf vorhandenen Erfahrungswerten und lässt in vielen Fällen Rückschlüsse von einem lokalen auf das globale Optimum zu.

5.1 Lokale Suche und Shaking

Lokale Suche mittels Variable Neighborhood Descent Unter lokaler Suche versteht man ein Verfahren, bei dem der Lösungsraum in einem begrenzten Bereich in der Umgebung einer gültigen Lösung durchsucht wird. Der zu durchsuchende Bereich wird durch die Nachbarschaft dieser Lösung vorgegeben. Das bedeutet, dass durch festgelegte Rechenvorschriften Adaptionen der Ausgangslösung vorgenommen werden und auf diesem Weg neue Lösungen erreicht werden. Um entscheiden zu können, ob eine Lösung besser oder schlechter ist, wird eine Bewertungsfunktion verwendet.

In Abb. 5.1 ist der Verlauf einer möglichen Zielfunktion dargestellt, sowie die Position x_{start} einer Startlösung markiert, von welcher ausgehend innerhalb einer Nachbarschaft N_i beispielsweise alle Lösungen im Intervall $[x_i; x_{start}]$ oder innerhalb einer Nachbarschaft N_j alle Lösungen im Intervall $[x_{start}; x_j]$ erreichbar sind und die markierten lokalen Optima hierbei jeweils die besten Lösungen der Nachbarschaften darstellen.

Bei der konkreten Umsetzung der lokalen Suche durch den *Variable Neighborhood Descent* (VND) wird der Lösungsraum über alle Nachbarschaftsstrukturen deterministisch durchsucht. Ausgangspunkt sind eine vorhandene Startlösung sowie die vor-

definierte Reihenfolge der verfügbaren Nachbarschaftsstrukturen. Beginnend bei der ersten Nachbarschaft der Startlösung wird nun nach einer besseren Lösung gesucht, um auf Basis dieser die Suche fortzusetzen. Die Reihenfolge in der die Nachbarschaften einer Lösung durchsucht werden hängt davon ab, ob Verbesserungen gefunden werden können oder nicht. Wird Nachbarschaft N_i durchsucht und konnte eine Verbesserung erzielt werden, so wird die Suche mit Nachbarschaft N_1 fortgesetzt. Kann in N_i keine Verbesserung erzielt werden, so kommt Nachbarschaft N_{i+1} zum Einsatz. Der Algorithmus terminiert also erst, wenn keine weiteren Verbesserungen erzielt werden können. Auf diesem Weg werden alle Nachbarschaften durchsucht, wobei am Beginn der Kette üblicherweise jene Strukturen stehen, deren Definition ein schnelleres Durchsuchen erlaubt als bei später gereihten. In diesem Zusammenhang seien zwei Strategien erwähnt, anhand derer entschieden werden kann, wann das Durchsuchen einer Nachbarschaft beendet werden soll. Wird *Best Improvement* (für die beste Lösung einer Nachbarschaft) gewählt, dann werden die Nachbarschaften, wie eben beschrieben, komplett durchsucht. Unter Verwendung von *Next Improvement* (für die nächst bessere Lösung einer Nachbarschaft) wird die Suche innerhalb einer Nachbarschaft abgebrochen, sobald eine bessere Lösung gefunden werden konnte. Der erste Eindruck mag vermitteln, dass es für die Lösungsverbesserung über alle Nachbarschaften besser wäre diese stets komplett zu durchsuchen. Tatsächlich ist es aber schwierig zu beurteilen welche Lösung für nachfolgende Nachbarschaften größeres Optimierungspotential bietet. Sicher jedoch kann behauptet werden, dass *Next Improvement* kürzere Laufzeit hat als *Best Improvement*, sofern mindestens einmal abgebrochen werden kann, bevor eine Nachbarschaft komplett durchsucht wurde.

In Alg. 1 ist der Ablauf von VND dargestellt. Natürlich ist zu beachten, dass sich abhängig von der Anzahl der definierten Nachbarschaftsstrukturen auch die Laufzeit verändert. Eine größere Anzahl an Nachbarschaftsstrukturen erhöht den notwendigen Zeitaufwand, steigert aber gleichzeitig auch die Chance eine bessere Lösung zu finden.

Shaking Zusätzlich zur lokalen Suche kann der Einsatz von zufälligen Komponenten positiven Einfluss auf die Effizienz eines Algorithmus haben. Dies kann durch einen so genannten *Shaking*-Mechanismus erreicht werden, also einem zufälligen „Durchschütteln“, wodurch zufällige Veränderungen einer Lösung eine neue Lösung erzeugen. Angenommen alle vorhandenen Nachbarschaftsstrukturen alleine ermöglichen lediglich von der in Abb. 5.1 markierten Startlösung x_{start} aus innerhalb des Intervalls $[x_i; x_j]$ neue Lösungen zu finden, so würde dies bedeuten, dass die besten auffindbaren Lösungen die beiden gekennzeichneten lokalen Optima wären. Durch das *Shaking* kann man allerdings erreichen, dass dieses Intervall verschoben wird und somit in einem neuen Bereich der Zielfunktion nach Optima gesucht werden kann, womit die Wahrscheinlichkeit steigt, das globale Optimum zu finden.

Bei der *lokalen Suche* handelt es sich also um einen Prozess, der innerhalb eines definierten Teilbereichs des gesamten Lösungsraumes nach besseren Lösungen sucht. Die Definition von zu durchsuchenden Teilbereichen (Nachbarschaften) ist allerdings

Algorithmus 1 : VND

Input : eine Startlösung x_{start} für das Optimierungsproblem
Output : die beste gefundene (lokale) Lösung x
Data : sei N' eine endliche Menge vorgegebener Nachbarschaftsstrukturen

```
begin
    k ← 1 ;
    x ← x_start ;
    repeat
        Suche nach bestem Nachbarn x' der Lösung x innerhalb der k-ten
        Nachbarschaft (x' ∈ N'_k(x)) ;
        if f(x') < f(x) then
            x ← x' ;
            k ← 1 ;
        else
            k ← k + 1 ;
    until k = k'_max ;
    return x ;
end
```

abhängig von der aktuellen Lösung. Somit stellt sich natürlich stets die Frage, ob mit den zur Verfügung gestellten Nachbarschaften der gesamte Lösungsraum erreicht werden kann, beziehungsweise vor allem die optimale Lösung. Um nun auch möglicherweise nicht abgedeckte Bereiche des Lösungsraumes erreichen zu können, wird *Shaking* eingesetzt. Erzeugt werden diese Veränderungen durch zufälliges Generieren von Lösungen aus einer der Nachbarschaften der aktuellen Lösung.

5.2 Genereller Ansatz für die Variable Nachbarschaftssuche

Aus der Kombination von lokaler Suche und Shaking lässt sich nun der generelle Ansatz für VNS aufbauen. Dabei wird die lokale Suche eingebettet in einen wiederkehrenden Ablauf von Shaking und der Entscheidung darüber, ob mit einer neuen und besseren Lösung fortgesetzt wird, oder keine Verbesserung innerhalb der lokalen Suche möglich war. Diese Vorgehensweise wird in Alg. 2 gezeigt. Es handelt sich hier um die Grundstruktur einer VNS, wobei die allgemein als lokale Suche bezeichnete Phase durch VND abgedeckt wird. Darüber hinaus bildet dieser Ablauf zugleich das Grundgerüst der in dieser Arbeit angewandten Nachbarschaftssuche.

Algorithmus 2 : Genereller VNS-Ansatz

Input : eine Startlösung x_{start} für das Optimierungsproblem
Output : die beste gefundene Lösung x
Data : sei $\mathcal{N}'$ eine endliche Menge vorgegebener Nachbarschaftsstrukturen

```
begin
    k ← 1 ;
    x ← x_start ;
    repeat
        // Shaking: Erzeuge eine zufällige Lösung x' aus x
           innerhalb der k-ten Nachbarschaft (x' ∈ N'_k(x))
        x' ← shake(x, k) ;
        // Führe lokale Suche mittels VND durch
        x' ← VND(x') ;
        if f(x') < f(x) then
            x ← x' ;
            k ← 1 ;
        else
            k ← k + 1 ;
    until k = k'_max ;
    return x ;
end
```

6 Ein hybrides Verfahren

Die vorliegende Problemstellung macht es unmöglich ein konstruktives Verfahren zu entwickeln, dass in polynomieller Zeit zur optimalen Lösung führt, weshalb eine Alternative gefunden werden muss. Die Herausforderung dabei stellen die Probleminstanzen dar, durch welche bei einer Größenordnung, die im Echtbetrieb vorstellbar wäre, die Anzahl der gültigen Lösungen sehr groß wird. Würde man alle möglichen Lösungen untersuchen wollen oder mit exakten Verfahren wie Branch & Bound arbeiten, so würde dies zu nicht akzeptablen Rechenzeiten führen. Man kann davon ausgehen, dass es sich dabei um mehrere Stunden handeln würde, man im Echtbetrieb je nach Situation aber bei einer vertretbaren Zeitspanne im Bereich von Sekunden bis zu maximal wenigen Minuten liegen muss.

Aus diesem Grund schien es sinnvoll zu sein, metaheuristische Verfahren in Kombination mit exakten Methoden zur Lösung von Teilaufgaben beziehungsweise zur Bewertung von konkreten Lösungen zu verwenden. Durch den Einsatz von VNS als Metaheuristik ist es möglich in kurzer Zeit sehr viele unterschiedliche Lösungen zu erzeugen. Dabei werden zunächst gewisse Parameter ermittelt, die eine konkrete Lösung definieren. Allerdings muss diese auch bewertet werden, um deren Güte bestimmen zu können. Dies geschieht mittels Dynamischer Programmierung in einem weiteren Schritt des Algorithmus, in welchem auf Basis der durch die VNS vorgegebenen Parameter einer Lösung eine passende konkrete Tour durch das Lager berechnet wird.

6.1 Der grundsätzliche Ablauf

Der in dieser Arbeit entwickelte Algorithmus kann im Wesentlichen in vier Teilbereiche gegliedert werden, welche so lange wiederholt werden bis keine zu bearbeitenden Bestellungen mehr vorhanden sind:

1. **Partitionierung vorhandener Bestellungen** - Der erste Schritt soll aus allen vorhandenen Bestellungen jene auswählen, die im Weiteren bearbeitet werden. Dabei wird einerseits berücksichtigt welche Priorität eine Bestellung auf Basis ihres Liefertermins hat und andererseits welche Kapazitätsbeschränkungen in der Verdichtungszone vorherrschen. Es soll bereits durch diesen Schritt sichergestellt sein, dass Artikel aus einer Bestellung stets im selben Bereich der Verdichtungszone gelagert werden (siehe auch Kapitel 2).
2. **Berechnung von Artikelauswahlen** - In einem zweiten Schritt werden dann

die einzusammelnden Artikel in so genannte Artikelauswahlen zerlegt. Das heißt es werden Listen von Artikeln erstellt, die dazu dienen, die innerhalb einer Tour abzuarbeitenden Artikel zu definieren. Dieser Schritt wird durch eine VNS abgedeckt, die viele verschiedene solcher Artikelauswahlen erzeugt, um später konkrete Touren daraus zu berechnen.

3. **Berechnung von Kommissionierungstouren** - Wie bereits im vorangegangenen Schritt beschrieben, geht es hierbei um das Finden konkreter Touren durch das Lager. Für jede einzelne Artikelauswahl kann nun mittels eines Dynamischen Programms sehr schnell die optimale Tour in Bezug auf deren Länge berechnet werden. Die Summe der Längen aller Touren stellt dabei die Güte der gesamten Lösung in dieser Konstellation dar. Sobald dieser Schritt abgeschlossen ist, kann entschieden werden, ob das vorhandene Ergebnis an die Lagerarbeiter übergeben werden soll, oder nochmals zum vorigen Schritt zurückgegangen werden soll, um durch neue Artikelauswahlen ein möglicherweise besseres Gesamtergebnis erzielen zu können.

4. **Zuweisen der Touren zu Arbeitern** - Im letzten Schritt werden die vorhandenen Touren an die verfügbaren Lagerarbeiter vergeben. Hier wird wieder mittels einer weiteren VNS versucht eine möglichst optimale Aufteilung zu erreichen, damit die letzte Tour zum frühest möglichen Zeitpunkt beendet werden kann. Sollten nach diesem Schritt noch weitere Bestellungen im System vorhanden sein, beginnt einfach ein neuer kompletter Durchlauf des Algorithmus.

Zur Veranschaulichung dieses Ablaufs dienen sowohl die schematische Darstellung in Abb. 6.1, als auch die Skizzierung durch Alg. 3. Dabei ist zu beachten, dass die Auswertung der Zielfunktion des ersten Schritts abhängig ist von den Berechnungen des zweiten Schritts, wodurch ein starkes Zusammenspiel dieser beiden Phasen gegeben ist. Das endgültige Ergebnis aller Berechnungen soll allen verfügbaren Arbeitern des Lagers Touren so zuweisen, dass der Endzeitpunkt aller Touren so früh wie möglich ist.

6.2 Variable Nachbarschaftssuche und Dynamische Programmierung

Wie bereits erwähnt, handelt es sich bei dem Algorithmus um einen hybriden Ansatz, bestehend aus VNS und Dynamischer Programmierung. Dabei wird der durchsuchte Lösungsraum über die zuvor angesprochene Einteilung in Artikelauswahlen definiert und dieser Schritt des Algorithmus auf Basis von VNS durchgeführt. Ausgehend von einer ersten Auswahl werden also alle weiteren auf bessere Zielfunktionswerte hin untersuchten Artikelzusammenstellungen durch stete leichte Abänderungen, definiert durch das System an Nachbarschaften, generiert. Hierbei kann eine geringfügige Veränderung einer Artikelzusammenstellung gleichzeitig große Auswirkungen auf die

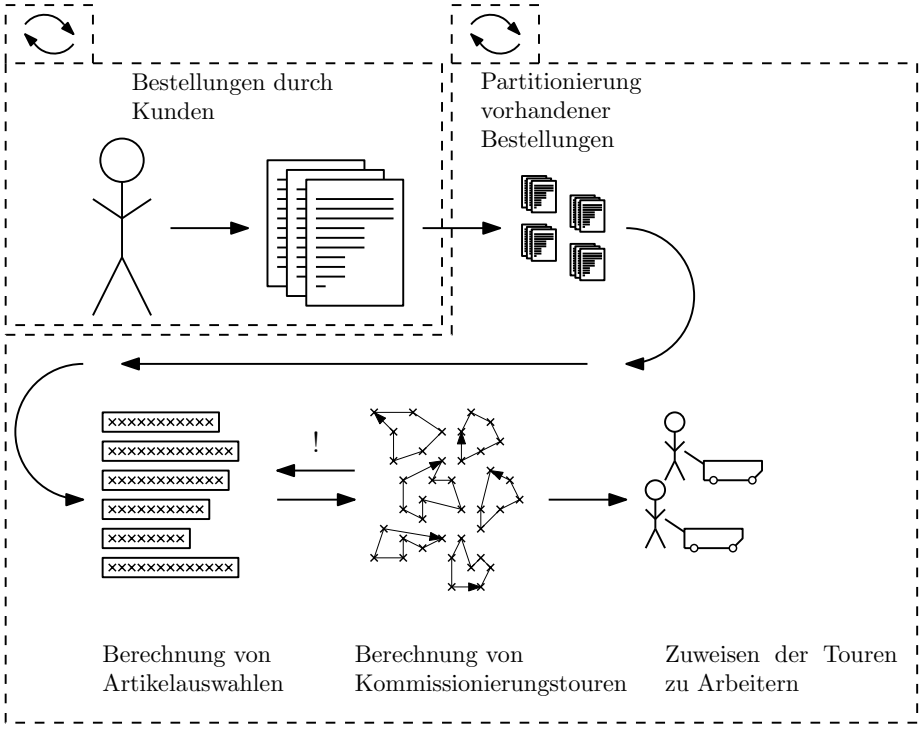


Abbildung 6.1: Schematische Darstellung des grundsätzlichen Ablaufs

Algorithmus 3 : Grundsätzlicher Algorithmus

Input : Bestellungen mit Listen aller einzusammelnder Artikel
Output : Eine Zuordnung von Arbeitern zu fertigen Touren

```
begin
    repeat
(1)     Erstelle Partitionierung vorhandener Bestellungen ;
        repeat
(2)         Berechne Zuordnung von Artikeln zu Touren ;
(3)         Berechne optimale Touren für die oben erstellte Zuordnung ;
        until keine Verbesserung konnte erzielt werden ;
(4)     Erstelle eine Zuordnung von Touren zu Arbeitern ;
        return die Zuordnung von Touren zu Arbeitern ;
    until keine weiteren Bestellungen sind abzuarbeiten ;
end
```

dadurch erhaltene Lösung haben. Das bedeutet, dass eine schlechte Lösung durch kleine Adaptionen schnell zu einer sehr guten Lösung werden kann.

Da dies alleine noch nicht ausreicht um die notwendigen Kommissionierungstouren bereitzustellen, ist als weiterer Bestandteil die Berechnung von optimalen Touren notwendig. Zu diesem Zweck schien es aufgrund der eingrenzbaren Möglichkeiten für einzelne Wegstrecken innerhalb des Lagers sinnvoll, mittels Dynamischer Programmierung nach einer Lösung zu suchen. Dabei werden für Auswahlen von zu besuchenden Lagergängen optimale Teiltouren berechnet, auf deren Basis die kürzeste Gesamttour erstellt werden kann.

7 Der Algorithmus im Detail

Die in Kapitel 6 beschriebene Struktur des in dieser Arbeit entwickelten Verfahrens verlangt nun noch nach einer konkretisierten Beschreibung. In diesem Kapitel möchte ich nun die genaue Funktionsweise des Algorithmus vorstellen und im Detail auf die zuvor vorgestellten einzelnen Schritte eingehen.

7.1 Partitionierung vorhandener Bestellungen

Wie bereits erwähnt wurde, handelt es sich bei diesem Teil des Ablaufs um eine erste Auswahl aus den vorhandenen Bestellungen. Es gibt also sozusagen einen Container, der mit Bestellungen von Seiten der Kunden befüllt wird. Aus diesem werden dann jeweils die nächsten zu bearbeitenden ausgesucht und an die folgenden Phasen übergeben. Um die Anforderungen an die möglichst kurz zu haltenden Lieferzeiten einhalten zu können, werden die verfügbaren Bestellungen nach deren Lieferdatum geordnet. Zusätzlich wird berücksichtigt, dass Artikel einer Bestellung im selben Bereich der Verdichtungszone abgelegt werden müssen und die damit verbundenen Kapazitätsbeschränkungen einzuhalten sind. Die Auswahl wird daher so getroffen, dass eine Partitionierung in ihrer Gesamtheit einen einzigen Bereich der Verdichtungszone erreicht. Es ist natürlich ersichtlich, dass es sich bei dieser Vorgehensweise nur um einen möglichen Ansatz handelt, mit diesen Randbedingungen beziehungsweise mit im Verlauf des Arbeitstages eintreffenden Bestellungen umzugehen. Es wird hierbei kein Anspruch auf Optimalität des Verfahrens gestellt, alle Nebenbedingungen werden dabei allerdings erfüllt.

7.2 Zuordnung von Artikeln zu Touren

Obwohl dieser Schritt des Algorithmus hauptsächlich die Entscheidung trifft welche der bestellten Artikel innerhalb einer Tour eingesammelt werden sollen, erfolgt die Bewertung dessen auf Basis der Länge von konkreten Touren und das macht bereits hier die Berücksichtigung mehrerer Einschränkungen notwendig. So auch bei Artikeln mit mehreren Lagerplätzen. Hier muss entschieden werden welche Menge von welchem Platz abgeholt werden soll, wobei natürlich auch die Kapazität der Kommissionierungswagen nicht überschritten werden darf.

Eine Lösung x für die Zuordnung von Artikeln zu Touren muss also mehrere Informationen beinhalten, welche über die Menge $\mathcal{S}$ der Zuordnungen von Artikeln zu Touren bestimmt sind. Das ist zunächst die Information, welche Mengen

$$\left(k_1, \ldots, k_{\bar{j}}\right) \subseteq \mathbb{N}^{\bar{j}}$$

für jeden Artikel $a_j \in \mathcal{A}$ einzusammeln sind, wobei $\mathcal{A}$ die Menge der Artikel bezeichnet ($|A| = j$) und $\bar{j}$ maximal gleich der Anzahl an unterschiedlichen Lagerplätzen für den Artikel a_j sein kann. Weiters wird definiert von welchen Lagerplätzen

$$\left(l_1, \ldots, l_{\bar{j}}\right) \subseteq \mathcal{L}^{\bar{j}}$$

diese entnommen werden sollen, wobei $\mathcal{L}$ die Menge der Lagerplätze beschreibt. Die sich daraus ergebende Anzahl an Touren, um alle Artikel einzusammeln, soll mit $m = |\mathcal{S}|$ bezeichnet werden.

Dies wiederum lässt folgende formale Definition zu:

$$\mathcal{S} = \bigcup_{i=1}^{m} \{S_i\} \tag{7.1}$$

$$S_i = \{1, \ldots, s_{\bar{i}}\}, \qquad \text{mit } \bar{i} \geq 1 \tag{7.2}$$

$$s_j = \left(a_j, \left(l_1, \ldots, l_{\bar{j}}\right), \left(k_1, \ldots, k_{\bar{j}}\right)\right), \qquad \text{mit } \bar{j} \geq 1 \tag{7.3}$$

Zusätzlich sei T_i definiert als ein konkreter Weg durch das Lager, auf welchem die durch S_i implizierten Lagerplätze besucht werden. Die Menge

$$\mathcal{T} = \bigcup_{i=1}^{m} \{T_i\} \tag{7.4}$$

bildet dann die Menge aller Touren T_i.

7.2.1 Konstruktionsheuristiken

Aufgrund der Tatsache, dass der Algorithmus ausgehend von einer Startlösung viele unterschiedliche Lösungen erzeugen und untersuchen soll, ist es notwendig einen schnellen Startpunkt zu generieren, was mittels Initialisierung per Konstruktionsheuristiken geschieht. Ausgehend davon kann dann in der Folge die vorhandene Lösung weiter verbessert werden. Da die Anzahl der gegebenen und zu erfüllenden Nebenbedingungen groß ist, bot sich die Entwicklung zweier unterschiedlicher Konstruktionsheuristiken an, von denen beide jeweils unterschiedliche Mengen von Einschränkungen abdecken.

Die erste dieser Heuristiken, die so genannte *Collision Avoiding Heuristic* (CAH), baut auf der Idee auf eine Menge von Touren erzeugen zu können, die nicht über-

Algorithmus 4 : CollisionAvoidingHeuristic

Input : eine Liste bestellter Artikel; Anzahl der Zonen $\overline{m}$
Output : eine Menge von Mengen bestehend aus einzusammelnden Artikeln innerhalb von $\overline{m}$ Touren
Data : Menge $\mathcal{S}$ von Mengen S_i, mit $i = 1 \dots \overline{m}$, von einzusammelnden Artikeln

```
begin
    foreach S_i ∈ S do
        S_i ← ∅ ;
        füge alle in Zone Z_i bestellten Artikel zu S_i hinzu ;
    return S ;
end
```

lappen. Es sollen dabei also keine Kreuzungen auftreten, sodass Arbeiter auf ihrem Weg durch das Lager nicht kollidieren. Um das zu erreichen, wird das Lager in $\overline{m}$ (physisch) nicht überlappende Zonen Z_i, mit $i = 1, \dots, \overline{m}$, eingeteilt. Nun wird für jede solche Zone Z_i eine Tour T_i errechnet, um alle Artikel einzusammeln, welche in Zone Z_i gelagert sind. Der Parameter $\overline{m}$ kann nach Belieben gewählt werden, es bietet sich aber durchaus an ihn mit der Anzahl der verfügbaren Lagerarbeiter zu initialisieren. Der Ablauf dieser Heuristik wird in Alg. 4 skizziert.

Die zweite Konstruktionsheuristik, *Time Saving Heuristic* (TSH) genannt, versucht anhand der streng einzuhaltenden Liefertermine entsprechende Zuordnungen zu finden. Erreicht wird dies hierbei, indem zunächst eine Reihung der Bestellungen nach deren spätest möglichen Lieferzeitpunkten erstellt wird, um anschließend mittels First-Fit Verfahrens Touren T_i, mit $i = 1, \dots, m$, zu berechnen, sodass die ersten benötigten und auszuliefernden Artikel in Tour T_1 eingesammelt werden. Erst wenn die Kapazität des ersten Kommissionierungswagens nicht mehr ausreichen sollte, wird eine zweite Tour T_2 für weitere Artikel erzeugt. Analog dazu entstehen eventuell benötigte weitere Touren $T_3, \dots, T_m$. Der Pseudo-Code für diesen Ablauf wird in Alg. 5 gezeigt.

Es ist offensichtlich, dass diese beiden Heuristiken alleine noch nicht ausreichen, um zulässige Lösungen des Problems zu produzieren. Bei CAH kann nicht zugesichert werden, dass etwa Lieferzeiten eingehalten werden und darüber hinaus werden die Kapazitäten der Kommissionierungswagen nicht beachtet. Andererseits kann bei TSH nicht garantiert werden, dass die Touren nicht überlappen. Daher kann nur dann mit diesen Methoden gearbeitet werden, wenn zusätzlich Reparaturalgorithmen zum Einsatz kommen. Trotzdem soll angenommen werden, dass die vorerst gefundenen Lösungen als *ad hoc*-Antwort durchaus akzeptabel sind, etwa wenn das System im Betrieb morgens gestartet wird. Der Hintergedanke dabei ist, dass für jeden einzelnen Mitarbeiter zunächst nur die Information über den nächsten zu holenden Artikel von Bedeutung ist. Alle weiteren Artikel können bestimmt werden während der Mitar-

Algorithmus 5 : TimeSavingHeuristic

Input : eine Liste L bestellter Artikel a_i
Output : eine Menge von Mengen bestehend aus einzusammelnden Artikeln innerhalb von m Touren
Data : Menge $\mathcal{S}$ von Mengen S_i, mit $i = 1 \dots m$, von einzusammelnden Artikeln; m wird durch diesen Algorithmus bestimmt

```
begin
    sortiere die Elemente in L nach dem spätesten Lieferdatum;
    S ← ∅ ;
    i ← 1 ;
    S_i ← ∅ ;
    foreach a ∈ L do
        if a passt nicht in S_i then
            S ← S ∪ {S_i} ;
            i ← i + 1 ;
            S_i ← ∅ ;
        füge a zu S_i hinzu ;
    S ← S ∪ {S_i} ;
    return S ;
end
```

beiter mit den ersten Artikeln beschäftigt ist. Damit hat das System Zeit, um weitere Rechenschritte durchzuführen und verbesserte Lösungen für die übrigen Routen zu finden. Artikel, die bereits eingesammelt wurden, müssen fix in den jeweiligen Touren eingeplant bleiben und nur nachfolgende Artikel sind in der Reihenfolge der Abholung noch variabel.

Auch wenn die Lösung x, gefunden von CAH oder TSH, gültig ist, heißt das noch nicht, dass deren Güte $f(x)$, im Speziellen also die Länge aller dadurch errechneten Touren insgesamt, die bestmögliche ist. Generell kann man aber davon ausgehen, dass dermaßen erzeugte Lösungen relativ schlechte Bewertungen haben werden. Der Grund dafür sind die Bestrafungen, die durch ein Überfüllen von Kommissionierungswagen entstehen. Sobald die Kapazitätsgrenze eines solchen erreicht wird, bestraft der Algorithmus dies so stark, dass es auf alle Fälle eine günstigere Lösung ist einen zusätzlichen Wagen zu verwenden. Es wird also unerlässlich sein mit Verbesserungsheuristiken weiterzuarbeiten.

7.2.2 Reparatur- und Verbesserungsheuristik

Zentraler Bestandteil des Programms ist die Umsetzung einer *Variablen Nachbarschaftssuche* (VNS) [12] mit Verwendung des *Variable Neighborhood Descent* (VND) als lokale Verbesserungskomponente, welche sowohl als Reparatur- als auch als Ver-

besserungsheuristik eingesetzt werden kann. Zur Reparatur ist das Verfahren einsetzbar, weil durch hohe Bestrafung von ungültigen Lösungen stets eine bessere, also kostengünstigere und gleichzeitig vor allem gültige Lösung gefunden werden kann. Wesentlich hierbei ist die Veränderung der Artikelauswahlen, die in weiterer Folge erst die Tourberechnung beeinflusst. Die nachfolgende Auflistung bezieht sich also stets auf eine Adaptierung von Artikelauswahlen.

Folgend den bereits in Kapitel 5 dargelegten Erläuterungen, kommen die anschließend vorgestellten Nachbarschaftsstrukturen zum Einsatz. Da diese Strukturen auf Basis von grundlegenden *Moves* definiert werden, folgt zuvor eine Auflistung eben dieser:

SwapMove(i, j, k, l) Dieser Schritt tauscht die Artikel $k \in S_i$ und $l \in S_j$, mit $S_i, S_j \in \mathcal{S}$, $1 \leq i < j \leq m$ und $k \neq l$. Das bedeutet, dass nach Anwendung dieses Schrittes Artikel k von Tour S_i entfernt und Artikel l eingefügt wurde. Analog wurde Artikel l von Tour S_j entfernt und Artikel k eingefügt.

ShiftMove(i, j, k) Wird dieser Schritt durchgeführt, so bedeutet dies, dass Artikel $k \in S_i$ aus S_i entfernt und in S_j eingefügt wird, wobei $1 \leq i, j \leq m$ und $i \neq j$.

MergeMove(i, j) Bei diesem Schritt werden zwei Touren S_i und S_j zu einer neuen Tour $S_{i'}$ zusammengeführt. Es werden also all jene Artikel, die zuvor von S_i und S_j eingesammelt wurden, von einer neuen Tour bearbeitet, wodurch S_i und S_j aus der Menge $\mathcal{S}$ entfernt werden können und stattdessen $S_{i'}$ in $\mathcal{S}$ eingefügt wird. Es vermindert sich dadurch also die Anzahl der Touren um eins.

SplitMove(i, R) Die Anwendung dieses Schrittes bringt die Tour S_i zu einer Aufteilung in zwei neue Touren S_i und $S_{i'}$, indem alle in R enthaltenen Artikel von S_i nach $S_{i'}$ verschoben werden. $S_{i'}$ muss dann natürlich zu $\mathcal{S}$ hinzugefügt werden und somit erhöht sich die Gesamtzahl an Touren um eins.

Formal betrachtet hat man hier 2^s Möglichkeiten die ursprüngliche Tour zu zerlegen, wenn die Anzahl an Elementen in S_i mit s bezeichnet wird, also $|S_i| = s$. Die Zerlegungen berechnen sich dann aus der Potenzmenge von S_i, nämlich $\mathcal{P}(S_i)$, beziehungsweise ergibt das $|\mathcal{P}(S_i)| = 2^s$ verschiedene Möglichkeiten.

SwapPositionMove(i, k, l) Dieser Move nützt die Tatsache aus, dass Artikel an mehr als nur einer Position im Lager verfügbar sein können. Sollte an einer weiteren Position im Lager ein bestimmter Artikel in der benötigten Menge vorhanden sein, so kann die Position vertauscht werden. Es wird dann der Ort des Artikels $k \in S_i$ auf den neuen Ort l geändert. Um den Ablauf hierbei einfach zu halten, wird der Artikel aus der Tour S_i entfernt und eine neue Tour S_i' gebildet, die zunächst nur diesen einen Artikel k mit der neuen Lagerplatzinformation enthält. Durch die spätere Anwendung eines *MergeMove* könnte diese Tour mit einer anderen wieder zusammengelegt werden.

SplitPositionMove(i, k, l, q) Dieser Schritt ermöglicht es in etwas anderer Form als bei den beiden bisherigen Varianten einen Artikel von anderer Stelle einzusam-

meln. Es ist generell natürlich nicht nur möglich die komplette Menge eines angeforderten Artikels von einem anderen Lagerplatz zu holen, sondern bei Bedarf auch nur eine Teilmenge davon. Diese Tatsache erlaubt die Definition eines Moves, durch den aus der Tour S_i eine bestimmte Teilmenge q für den Artikel k von einer zusätzlichen Position l im Lager innerhalb einer neuen Tour S_i' eingesammelt wird und gleichzeitig eine bleibende Restmenge von der ursprünglichen Position zu holen bleibt.

Aus diesen grundlegenden Moves werden nun folgende Nachbarschaftsstrukturen definiert, wobei $\mathcal{S}$ die Menge an Selektionen (Touren) bezeichnet, s_{max} die maximale (größte vorkommende) Anzahl an Artikeln einer Tour angibt und l_{max} für die maximale (größte vorkommende) Anzahl an alternativen Lagerplätzen eines Artikels aus dem Lager steht:

Swap Grundlage für diese Nachbarschaftsstruktur ist der zuvor eingeführte *SwapMove*. Ausgehend von einer konkreten Lösung sind dadurch alle Lösungen definiert, welche mittels eines *SwapMoves* erreichbar sind. Sollen also Artikel zwischen je zwei Touren vertauscht werden, so liegt die Anzahl an Möglichkeiten dafür in $O(|\mathcal{S}|^2 \cdot s_{max}^2)$.

Shift Diese Nachbarschaftsstruktur basiert auf dem *ShiftMove* und somit kann die Anzahl an möglichen Alternativen zu einer konkreten Lösung mit $O(|\mathcal{S}|^2 \cdot s_{max})$ abgeschätzt werden.

Merge Auch diese Struktur leitet sich aus nur einem Move ab, nämlich dem *MergeMove*. Hier kommt man für eine Lösung auf $O(|\mathcal{S}|^2)$ mögliche neue Lösungen. In dieser Nachbarschaftsstruktur sind alle Lösungen enthalten, die aus einer gegebenen Lösung durch Anwendung eines *MergeMoves* erzeugt werden können.

Split In der Anwendung hat es sich bewährt bei dieser auf dem *SplitMove* basierenden Nachbarschaftsstruktur beim Durchsuchen auf eine komplette Abdeckung zu verzichten. Die insgesamt möglichen Adaptierungen können mit $O(|\mathcal{S}| \cdot 2^{s_{max}})$ abgeschätzt werden. Tatsächlich werden nur jene Fälle berücksichtigt, bei denen die Anzahl an Artikeln in einer Tour auf die Hälfte reduziert werden. Bei einer geraden Anzahl an Artikeln kann genau in die Hälfte geteilt werden, bei ungerader Anzahl entsteht dementsprechend eine Tour mit einem Artikel mehr als bei der anderen. Somit bleibt ein stark verminderter Umfang von $O(|\mathcal{S}| \cdot s_{max})$ übrig.

SwapPosition Diese Nachbarschaftsstruktur stützt sich auf den vorgestellten *SwapPositionMove* und umfasst daher eine geschätzte Anzahl von $O(|\mathcal{S}| \cdot s_{max} \cdot l_{max})$ möglicher Nachbarlösungen zu einer konkreten Lösung.

SwapPositionMerge Hier handelt es sich um eine Struktur, die ähnlich wie *SwapPosition* aufgebaut ist. Allerdings wird versucht eine jeweils durch einen *SwapPositionMove* neu entstehende Tour mit den übrigen verfügbaren Touren zu kombinieren. Es kommt hierbei zur Kombination mit einem *MergeMove*. Da-

her entsteht eine Größe des in Frage kommenden Lösungsraumes von $O(|\mathcal{S}|^2 \cdot s_{max} \cdot l_{max})$.

SplitPositionMerge Grundlage für diese Nachbarschaftsstruktur bilden der *SplitPositionMove* sowie der *MergeMove*, welche nacheinander angewendet werden. Konkret wird versucht etwa die Hälfte der angeforderten Menge eines Artikels von einem alternativen Lagerplatz zu holen. Dabei entstehen für eine Lösung durch $O(|\mathcal{S}|^2 \cdot s_{max} \cdot l_{max})$ beschränkte weitere gültige Lösungen.

Die nun aufgelisteten Nachbarschaftsstrukturen sind im Wesentlichen aus mehreren Moves (deren Anzahl wird im Folgenden mit m bezeichnet) zusammengesetzt und benötigen daher weit mehr Rechenzeit, um durchsucht zu werden. Um dem entgegenzuwirken, wurde der durchsuchte Bereich teilweise eingeschränkt beziehungsweise mit einer vorgegebenen Anzahl an zufällig erzeugten Lösungen innerhalb der Strukturen gearbeitet. Daher gibt es in solchen Fällen deutliche Unterschiede zwischen dem theoretischen und dem praktisch durchsuchten Nachbarschaftsbereich einer Lösung.

Der Vorteil bei der Kombination von mehreren Moves ist, dass schlechtere Lösungen vorerst beibehalten werden können und erst nachdem alle definierten Moves durchgeführt wurden eine Bewertung erfolgt. Weiters ist auch zu beachten, dass die einzelnen dabei verwendeten Moves in keiner Abhängigkeit zueinander stehen müssen, nur Überschneidungen müssen beachtet werden. Es kann ein Artikel beispielsweise nicht zweimal hintereinander aus einer Tour genommen werden. Die Strukturen sind also wie folgt definiert:

DoubleShift Hier wird einfach die Anwendung von *ShiftMoves* aneinander gereiht und somit zweimal verschoben, um eine neue Lösung zu erzeugen. Die Menge an möglichen Lösungen hat dabei eine Mächtigkeit in der Größenordnung von $O(|\mathcal{S}|^4 \cdot s_{max}^2)$. Durch die zufällige Generierung von entsprechenden Moves und einer fix vorgegebenen Anzahl an Versuchen ergibt sich eine Laufzeitkomplexität von $O(1)$.

Dieser markante Unterschied entsteht hier, da entschieden wurde jeweils 1000 zufällig generierte Lösungen zu testen, um die Komplexität der Nachbarschaftsstruktur zu umgehen. Das komplette Durchsuchen dieser, wie auch einiger der folgenden Nachbarschaftsstrukturen würde zu lange dauern. Es schien durchaus vertretbar an dieser Stelle mit einer Zufallskomponente zu arbeiten.

MultipleShift Ähnlich wie beim zuvor beschriebenen *DoubleShiftMove*, ist auch hier eine Aneinanderreihung von Elementen aus *ShiftMoves* Gegenstand der Definition, allerdings ist die Anzahl an Verschiebungen hier frei definierbar. Dementsprechend definiert sich die Größenordnung der möglichen Lösungen hier als $O((|\mathcal{S}|^2 \cdot s_{max})^m)$, die Laufzeit liegt aber auch hier wieder in $O(1)$ (siehe auch bei *DoubleShift*).

MultipleSwap Hierbei handelt es sich um die Kombination mehrerer *SwapMoves*, welche in ihrer Gesamtheit dann zu mehrfachen Vertauschungen von Artikeln

innerhalb einer gesamten Veränderung führen. Ein Lösungsraum der Größe $O((|\mathcal{S}|^2 \cdot s_{max}^2)^m)$ steht hier einem durchsuchten Raum der Größe $O(1)$ gegenüber (siehe auch bei *DoubleShift*).

MultipleSplit Wie der Name bereits andeutet, werden hier einige *SplitMoves* miteinander verknüpft, was die Aufspaltung mehrerer Touren auf einmal zur Folge hat. Hier bewegen wir uns bei der Anzahl der möglichen Alternativen im Bereich von $O((|\mathcal{S}| \cdot 2^{s_{max}})^m)$. Durch die auch hier wieder gewählte Beschränkung auf zufällig erzeugte Kombinationen ergibt sich eine Laufzeit von $O(1)$ (siehe auch bei *DoubleShift*).

ShiftSplit Dieser Schritt soll die Vorzüge eines *SplitMoves* mit denen eines *ShiftMoves* kombinieren. Dem Verschieben eines Artikels folgt hier unmittelbar die Zerlegung einer Zuordnung. Damit ergeben sich $O(|\mathcal{S}|^3 \cdot s_{max}^2)$ Möglichkeiten, die wiederum in $O(1)$ durchsucht werden (siehe auch bei *DoubleShift*).

SwapSplit Wie auch hier bereits aus dem Namen hervorgeht, wird zuerst ein *SwapMove* durchgeführt und anschließend eine Zuordnung durch einen *SplitMove* zerlegt. Hierbei ergibt sich ein Ausmaß an Lösungen von $O(|\mathcal{S}|^3 \cdot s_{max}^3)$, das Durchsuchen selbst ist wieder auf $O(1)$ reduziert (siehe auch bei *DoubleShift*).

SplitMerge Diese Nachbarschaftsstruktur wurde in der Form umgesetzt, dass sie in Anlehnung an den *SplitMove* entsprechend durchsucht wird und nach einer Aufsplittung versucht wird beide neu erzeugten Touren mit allen anderen verfügbaren zu kombinieren. Dies geschieht in mehreren Schritten. Zunächst wird versucht den ersten Teil der Tour mit allen weiteren verfügbaren Touren zu kombinieren. Anschließend wird dies auch mit dem zweiten Teil probiert. Ein weiterer Schritt versucht den ersten und zweiten Teil gleichzeitig mit jeweils unterschiedlichen Touren zu kombinieren. Es entstehen hier also gleich drei unterschiedliche Ansätze pro Aufsplittung. Die Vorgehensweise unterliegt dabei weitaus weniger dem Zufall, viel eher wird versucht eine komplette Abdeckung zu erreichen. Lediglich das Aufsplitten von Touren erfolgt nach wie vor eingeschränkt, da sonst für ein geringes Maß an Verbesserung ein unverhältnismäßig hoher Aufwand betrieben werden müsste. Außerdem sind bei dieser Nachbarschaftsstruktur die einzelnen Moves abhängig voneinander. Es ergibt sich hier eine Komplexität von $O(|\mathcal{S}|^2 \cdot s_{max})$.

Manche Situationen machen es nötig, eine schnelle Verbesserung einer gegebenen Lösung x zu errechnen, zum Beispiel wenn ein einzusammelnder Artikel nicht an der zugeteilten Stelle aufzufinden ist, weshalb die standardmäßige Abfolge bei VND durch eine Laufzeitbeschränkung limitiert wurde. Es kann dann nach Bedarf eine zeitliche Grenze geben, nach deren Erreichen keine neue Nachbarschaft durchsucht wird. Der Pseudo-Code in Alg. 6 soll diese Umsetzung andeuten.

Wie bereits zuvor erwähnt, kann dieser hier vorgestellte VND Ansatz als Reparatur- und auch als Verbesserungsverfahren verwendet werden. Dies erfordert die Bewer-

Algorithmus 6 : VND mit zeitlicher Einschränkung

Input : eine Startlösung x
Data : Zähler l, Anzahl l_{max} der zu untersuchenden Nachbarschaften

```
begin
    l ← 1 ;
    repeat
        x' ← durchsuche Nachbarschaft N_l(x) ;
        if f(x') < f(x) then
            x ← x' ;
            l ← 1 ;
        else
            l ← l + 1 ;
    until l > l_max oder eine vordefinierte Zeit wurde erreicht ;
end
```

tung einer Lösung x und einer daraus abgeleiteten möglichen Verbesserung x'. Da die Güte einer Lösung im Wesentlichen von der Summe der Längen aller Touren abhängt, ist es also hier von Bedeutung sämtliche benötigte Touren einer Lösung so zu berechnen, dass diese auf kürzestem Weg durch das Lager führen. Kapitel 7.3 wird zeigen wie es möglich ist entsprechende Touren unter Berücksichtigung der speziellen Eigenschaften des Lagers in polynomieller Zeit zu berechnen.

Die Nachbarschaftsstrukturen zur Durchführung von *Shaking*-Phasen (also zufälligen Veränderungen einer Lösung) basieren auf den zuvor vorgestellten, jedoch werden hierbei anstelle eines Schrittes mehrere zufällig ausgewählte Schritte auf die entsprechende Lösung angewendet. In der letzten Version des Algorithmus kam dafür *Swap* zum Einsatz. Die zufälligen Veränderungen aus der Nachbarschaft $\mathcal{N}'_k(x)$ in Alg. 2 kommen daher jeweils nur aus *Swap*. Gleichermaßen wäre aber auch die Verwendung von *Shift*, *Split* oder *Merge* sowie jede Kombination dieser möglich gewesen, jedoch konnten keine klaren Vorteile für eine der Varianten im Vorfeld ausgemacht werden.

7.2.3 Durchsuchen der Nachbarschaften

Beim Durchsuchen der einzelnen Nachbarschaften unterscheidet man zwei Methoden zur Übernahme einer neuen und besseren Lösung, nämlich *Next Improvement* und *Best Improvement* (siehe auch Kapitel 5). Wird *Next Improvement* verwendet, so ist das gleichbedeutend mit einem Abbruch der Suche, sobald eine bessere Lösung gefunden wurde. Konkret auf *ShiftMove* bezogen wird also die Nachbarschaft systematisch durchsucht und beginnend mit dem ersten bis zum letzten Artikel versucht diesen einer anderen Tour zuzuordnen. Bei diesem Vorgehen sind aber jene Artikel klar bevorzugt, welche in der Liste am Anfang stehen. Daher wurde eine zufällige Durchsuchungsreihenfolge implementiert, sodass jede Lösung im Nachbarschaftsraum mit

gleicher Wahrscheinlichkeit zu einer Verbesserung beitragen kann.

7.2.4 Dynamische Reihenfolge der Nachbarschaften

Eine große Herausforderung bei der Umsetzung mittels VNS stellt die Reihung der definierten Nachbarschaftsstrukturen dar. Nachdem die Anwendung einer Nachbarschaftsstruktur durchaus auch Auswirkungen auf die Effizienz nachfolgender Nachbarschaftsstrukturen hat, ist diese Entscheidungsfindung nicht trivial. Aus diesem Grund wurde entschieden, eine Adaptierung der in [14] vorgestellten Methode zur dynamischen Reihung vorzunehmen. Im Folgenden gilt es stets eine Permutation

$$\lambda = (\lambda_1, \ldots, \lambda_k) \text{ der Nachbarschaftsstrukturen } \{N_1, \ldots, N_k\}$$

zu berechnen, wobei der Nachbarschaftsstruktur N_i, $i = 1, \ldots, k$, ein Prädikat $w_i > 0$ zugeordnet wird, welches zu Beginn mit einem konstanten Wert W belegt wird. Während nun die Nachbarschaft $N_{\lambda_i}(x)$ einer Lösung x durchsucht wird, wird w_{λ_i} in Abhängigkeit von Rechenzeit t_{λ_i} und Sucherfolg aktualisiert. Wurde eine verbesserte Lösung gefunden, so wird w_{λ_i} halbiert und $\frac{t_{\lambda_i}}{\alpha}$ hinzugefügt, wobei α ein Parameter ist, der den Einfluss der Auswertung auf die eigentliche Rechenzeit repräsentiert. Sollte keine bessere Lösung gefunden werden, so wird die Nachbarschaftsstruktur durch Hinzufügen der unverminderten Rechenzeit t_{λ_i} zum bestehenden Wert w_{λ_i} mit höheren Kosten belastet. Die Permutation λ wird erst neu erzeugt, wenn ein gerade aktualisierter Wert w'_{λ_i} kleiner als das bisher kleinste, $\min_{j=1,\ldots,k} w_j$, oder größer als das bisher größte, $\max_{j=1,\ldots,k} w_j$, solcher Prädikate ist. Dann wird anhand der aufsteigenden Werte w_i eine neue Reihenfolge bestimmt.

Das Konzept ist in Alg. 7 ersichtlich, wobei eine kleine Anpassung zum in [14] vorgestellten Algorithmus durchgeführt wurde, um zu vermeiden, dass Nachbarschaften ausgelassen werden können. Das kann vorkommen, da dort vor jeder Umreihung gespeichert wird mit welcher Nachbarschaft fortgesetzt werden soll und diese dann weiter hinten gereiht werden könnte als andere noch nicht besuchte. Außerdem soll verhindert werden, dass die Nachbarschaften ständig umgereiht werden. Dies geschieht im ursprünglich vorgestellten Verfahren viel häufiger, da auch dann eine Umreihung passieren kann, wenn keine Verbesserung der Lösung erzielt wurde. Der tatsächlich verwendete Algorithmus verändert die Reihenfolge der Nachbarschaften erst dann, wenn gerade eine verbesserte Lösung gefunden werden konnte.

Getestet wurde der gesamte Algorithmus sowohl mit dynamischer als auch mit statischer Nachbarschaftsreihenfolge. Die Ergebnisse dazu werden in Kapitel 8 präsentiert.

Algorithmus 7 : VND mit dynamischer Nachbarschaftsreihenfolge

Input : eine Startlösung x
Data : Zähler i, Anzahl k der zu untersuchenden Nachbarschaften

begin
- $w_1 \leftarrow w_2 \leftarrow \ldots \leftarrow w_k \leftarrow W$;
- $w_{min} \leftarrow w_{max} \leftarrow W$;
- $\lambda = (1, 2, \ldots, k)$;
- $i \leftarrow 1$;
- **repeat**
  - $x' \leftarrow$ durchsuche Nachbarschaft $N_i(x)$ mit Zeitaufwand t_{λ_i} ;
  - **if** $f(x') < f(x)$ **then**
    - $x \leftarrow x'$;
    - $w_{\lambda_i} \leftarrow \frac{w_{\lambda_i}}{2} + \frac{t_{\lambda_i}}{\alpha}$;
    - **if** $(w_{\lambda_i} < w_{min}) \vee (w_{\lambda_i} > w_{max})$ **then**
      - Sortiere $\lambda_1, \ldots, \lambda_k$, sodass $w_{\lambda_1} \leq w_{\lambda_2} \leq \ldots \leq w_{\lambda_k}$;
      - $w_{min} \leftarrow w_{\lambda_1}$;
      - $w_{max} \leftarrow w_{\lambda_k}$;
    - $i \leftarrow 1$;
  - **else**
    - $w_{\lambda_i} \leftarrow w_{\lambda_i} + t_{\lambda_i}$;
    - $i \leftarrow i + 1$;
- **until** $i > k$;

end

Algorithmus 8 : Berechnung einzelner Touren

```
Input : Menge an Zuordnungen S
Data : m = |S|
Output : Menge von Touren T
begin
    foreach S_i ∈ S do
        T_i ← kürzeste Tour für S_i ;
        T ← T ∪ {T_i} ;
    return T ;
end
```

7.3 Berechnung einzelner Touren

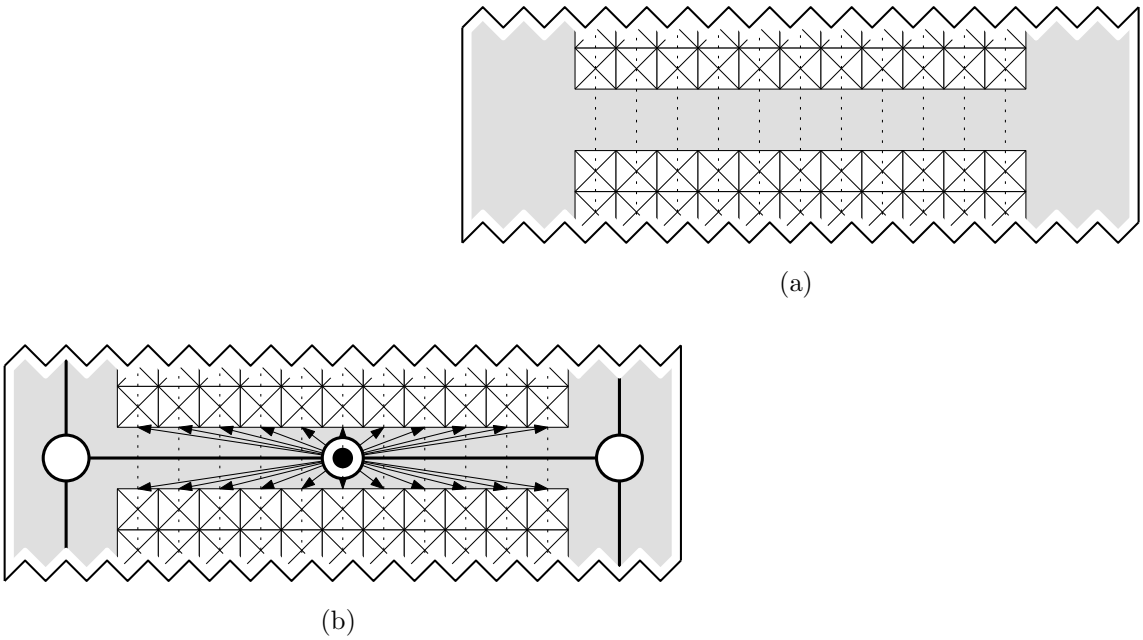
Die nachfolgende Phase des Algorithmus, wie in Alg. 3 vorgestellt, betrifft das Auffinden von Touren ausgehend von den zuvor festgelegten Artikelzuordnungen. Weiters ist die Berechnung von kürzesten Wegen ein Teilproblem des VNS, wie in Kapitel 7.2.2 beschrieben.

Zur effizienten Berechnung dieser Touren ist es zunächst notwendig das Lager in eine passende Struktur zu dessen Repräsentation zu bringen, was durch einen ungerichteten, zusammenhängenden Graphen $G = (V, E)$ passiert. Ausgehend davon kann nun ein Verfahren basierend auf Dynamischer Programmierung nach optimalen Touren suchen. Die dabei entscheidende Idee ist die Definition so genannter *Module*, welche entsprechend ihrer Verbindungsmöglichkeiten aneinander angeschlossen werden können. Der nun im Detail erläuterte Algorithmus berechnet für jede Zuordnung S_i eine konkrete Tour T_i. Diese Vorgehensweise ist im Pseudo-Code in Alg. 8 ersichtlich.

7.3.1 Repräsentation als Graph

Die Repräsentation des Lagers geschieht also mittels eines ungerichteten, zusammenhängenden Graphen $G = (V, E)$, wobei die Kantenmenge E die Gänge innerhalb des Lagers bildet. Unterschieden werden dabei *Regalgänge* und *Hauptgänge*. Während erstere jene Gänge sind, die sich zwischen den Lagerregalen befinden und parallel zu diesen sind, verbinden zweitere Regalgänge und wieder andere Hauptgänge miteinander (siehe auch Abb. 1.1 in Kapitel 1). Jeder Knoten $v \in V$ entspricht einem Punkt mit speziellen Eigenschaften. Das können die *Verdichtungszone* (VZ), Kreuzungen von Gängen und Positionen innerhalb von Regalen sein. Es kommt sehr häufig vor, dass mehrere Positionen im Lager (Lagerplätze) übereinander liegen, solche Vorkommen werden dann aber zusammengefasst in einem Knoten, der darüber hinaus den gesamten Regalgang enthält.

Man kann nun sehen, dass jeder Knoten $v \in V$ höchstens vier Nachbarn hat, das



(a)

(b)

Abbildung 7.1: Lagerrepräsentation (a) als Graph (b)

heißt, der Grad jedes Knotens ist von oben mit vier beschränkt. Weiters besitzen Knoten, die einen Lagerplatz bezeichnen, höchstens zwei Nachbarn. Das heißt, all jene Knoten innerhalb eines Regalganges können als lineare Liste betrachtet werden. Da dem Graph ein typisches Lager zugrunde liegt, ist er planar, also ohne Kreuzungen von Kanten zeichenbar.

Unter Ausnützung dieser Eigenschaften kann man den Graphen G einschränken und die Anzahl an Knoten stark reduzieren, indem für jeden Gang höchstens drei Knoten existieren (siehe Abb. 7.1). Dort repräsentiert der Knoten in der Mitte die (lineare Liste der) Lagerplätze und die beiden Knoten am Beginn und am Ende des Ganges bezeichnen die Kreuzungspunkte mit dem jeweiligen Hauptgang.

7.3.2 Ein Dynamisches Programm

Ohne Einschränkung der Allgemeinheit und mit Bezug auf Abb. 7.1 soll angenommen werden, dass Regalgänge und Hauptgänge orthogonal zueinander liegen. Hauptgänge verlaufen hierbei von oben nach unten, Regalgänge stets von links nach rechts. Eine Menge von zwei Hauptgänge gemeinsam mit den dazwischen liegenden Regalgängen wird weiters auch als *Block* bezeichnet.

Im Folgenden wird ein Algorithmus aufbauend auf Dynamischer Programmierung gezeigt, welcher in polynomieller Zeit die Berechnung kürzester Touren über alle Knoten von Interesse, also all jene Lagerplätze mit einzusammelnden Artikeln, in-

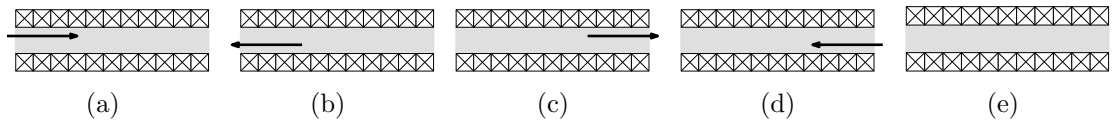


Abbildung 7.2: Grundlegende Gangoperationen.

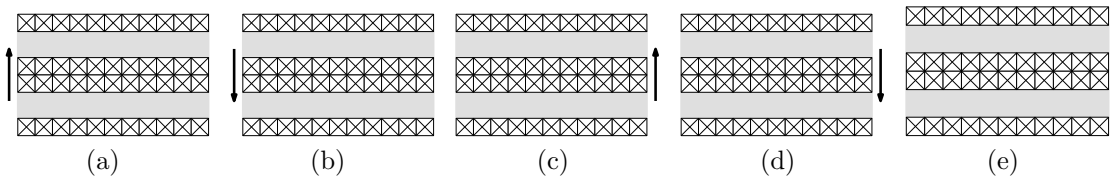


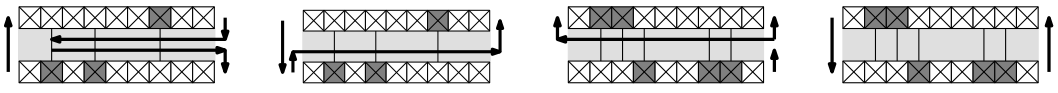
Abbildung 7.3: Grundlegende Zwischen-Gangoperationen.

nerhalb eines Blocks durchführt.

Zu diesem Zweck werden so genannte *Gangoperationen*, sowie *Zwischen-Gangoperationen* eingeführt. Gangoperationen werden herangezogen, um die Möglichkeiten für Wege innerhalb eines Regalganges abzudecken, wohingegen Zwischen-Gangoperationen Bewegungen innerhalb der Hauptgänge beschreiben.

Gangoperationen In Abb. 7.2 ist die Menge an grundlegenden Gangoperationen, welche von Arbeitern durchgeführt werden können, grafisch dargestellt. Es ist offensichtlich, dass Gänge entsprechend von links oder rechts betreten werden können (siehe Abb. 7.2a–7.2d), aber auch komplett ausgelassen werden können (siehe Abb. 7.2e). Ein Gang kann natürlich nur dann ausgelassen werden, wenn keine Artikel daraus benötigt werden. In diesem Fall ist eine solche Operation also auch gültig. Im implementierten Algorithmus kommt diese Operation nicht direkt zum Einsatz, da hier Gänge weggelassen werden, wenn sie nicht betreten werden müssen. Es werden dann einfach die Gänge davor und danach direkt miteinander verbunden.

Zwischen-Gangoperationen Zusätzlich zu den eingeführten grundlegenden Gangoperationen werden Zwischen-Gangoperationen benötigt. Diese wiederum beschreiben welche Möglichkeiten für die Arbeiter bestehen, um sich zwischen zwei aufeinander folgenden Gängen zu bewegen (siehe Abb. 7.3). Diese Aktionen lassen einen Wechsel über die Hauptgänge zu (siehe Abb. 7.3a–7.3d). Natürlich kann es auch vorkommen, dass kein weiterer Gang besucht werden muss, dann ist kein Gangwechsel mehr nötig, wie in Abb. 7.3e zu sehen ist. Die Umsetzung im Programm erfolgt auch hier durch Weglassen nicht benötigter Gänge, die theoretisch gültigen Gangoperationen sind dann nicht nötig.



(a) (b) (c) (d)

Abbildung 7.4: Gültige und ungültige Module.

Gangmodule Letztlich ist es hiermit möglich eine Menge an so genannten *Gangmodulen* zu definieren, welche wiederum auf den vorgestellten grundlegenden Operationen aufbauen. Daraus kann man nun jede erdenkliche Kombination erstellen, jedoch ist nicht jede davon auch gleichzeitig gültig. Abb. 7.4a und 7.4b zeigen zwei gültige Module. Das in Abb. 7.4c gezeigte Modul ist jedoch von vornherein ungültig, da der rechte Knoten, der die Kreuzung von Regalgang und Hauptgang repräsentiert zweimal verlassen aber nur einmal betreten wird. Das Modul in Abb. 7.4d ist im Prinzip gültig, aber vom Programm zu verwerfen, weil die Lagerplätze, von denen Artikel eingesammelt werden sollen (in der Abb. dunkelgrau markiert), gar nicht besucht werden.

Es sei nun definiert, dass zwei Module j und j' für die Gänge i und i' untereinander kompatibel sind, wenn die Module j und j' basierend auf ihren Zwischen-Gangoperationen verbunden werden können. Weiters ist Modul j' mit Modul j „von oben“ kompatibel, wenn die Zwischen-Gangoperationen von j um „nach unten“ beziehungsweise „nach oben“ zu gehen mit den entsprechenden Zwischen-Gangoperationen von j' zusammenpassen. Im Folgenden bezeichnet $\mathcal{M}_{\text{komp}}(j)$ die Menge aller Module j', die „von oben“ kompatibel mit j sind.

Einige Beobachtungen Generelles Ziel ist immer noch das Einsammeln aller angeforderten Artikel. Nachdem bereits Selektionen an Artikeln getroffen wurden, welche jeweils innerhalb einer Tour einzusammeln sind, wird mit dem Berechnen jeder dieser Touren ein Teilziel erreicht. Zusätzlich dazu suchen wir aber nach Touren durch das Lager, die insgesamt ein Minimum an Zeit benötigen sollen, weshalb eine entsprechende Bewertung jeder Tour erfolgen muss. Diese Bewertung kann auf den zuvor eingeführten Modulen zur Tourkonstruktion passieren, indem die Kosten $c_i(j) > 0$ für jedes Modul j basierend auf den Aktionen, die ein Arbeiter in Gang i durchführt, berechnet werden. Transformiert auf den erwähnten Dekodierungsgraph G' bedeutet das eine einfache Umlegung auf die Kosten für jede einzelne Kante $e \in E'$ gemäß den jeweiligen Längen derselben.

In Bezug auf die Anzahl von Modulen, die zur Zusammenstellung einer Tour relevant sein können, muss zum einen nocheinmal erwähnt werden, dass nicht alle Kombinationen an Gangoperationen und Zwischen-Gangoperationen gültige Module bilden. Zum anderen ist es aber darüber hinaus wesentlich zu beachten, dass auch gültige Module teilweise auszuschließen sind beziehungsweise manche gültige Module durch weitaus effizientere Module mit gleichen Möglichkeiten ersetzt werden können. Als

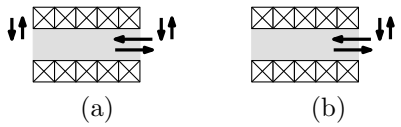


Abbildung 7.5: Ein gültiges Modul (a) kann durch ein effizienteres Modul (b) ersetzt werden.

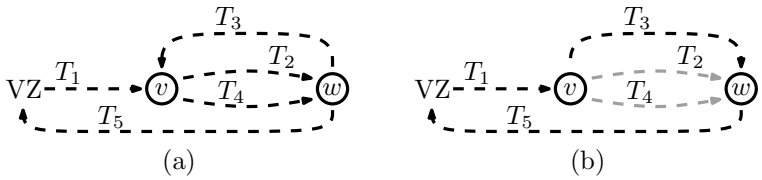


Abbildung 7.6: Konstuktion einer Tour T' aus einer Tour T unter der Annahme, dass T zweimal Knoten w direkt nach v besucht.

Beispiel für einen solchen Austausch im Sinne der Effizienzsteigerung soll Abb. 7.5 dienen. Dort verlangt Modul 7.5a das Erreichen und sofortige Verlassen des Ganges von links oben, weshalb diese beiden Kanten und damit auch nicht notwendiger Weg eingespart werden könnten.

Zu beachten ist außerdem, dass eine Tour, wie sie im Rahmen dieser Arbeit verstanden wird, sich von Touren in Zusammenhang mit dem *Travelling Salesman Problem* oder dem *Vehicle Routing Problem* unterscheidet. Die etwas andere Auffassung kommt daher, dass Knoten, wie zum Beispiel Kreuzungspunkte von Gängen oder die Verdichtungszone, mehr als nur einmal besucht werden dürfen. Das ist notwendig, da meistens keine direkte Verbindung zwischen zwei Knoten des Lagergraphs besteht. Somit können Pfade zwischen zwei Knoten mehrfach zurückgelegt werden, allerdings kann man die Anzahl an Verwendungen des selben Wegs von oben beschränken.

Theorem 7.3.1. *Gegeben ist eine Tour T von kürzester Länge in Bezug auf eine Menge an Punkten, die von T besucht werden. Weiters wird angenommen, es gibt zwei adjazente Punkte v und w, welche in T zweimal unmittelbar hintereinander besucht werden. Der Weg zwischen v und w wird dann in T einmal von v nach w passiert und einmal umgekehrt.*

Beweis. Angenommen der Weg zwischen v und w wird zweimal in der gleichen Richtung durchlaufen, dann kann man die Tour T in fünf Teil-Touren T_1, T_2, T_3, T_4 und T_5 aufteilen, wie in Abb. 7.6a zu sehen, wobei *VZ* für die Verdichtungszone steht. Man kann nun eine neue Tour T' erzeugen, welche beginnend bei *VZ* entlang von T_1 nach v verläuft, dann T_3 von v nach w folgt, um schließlich T_5 folgend von w aus zu *VZ* zurückzukehren (siehe Abb. 7.6b). Da v und w adjazent sind, also kein anderer Punkt auf dem Weg von v nach w passiert werden muss, besucht T' dieselben Punkte wie die Tour T. Weiters werden T_2 und T_4 innerhalb von T' ausgelassen, wodurch T' kürzer ist als T und das wiederum einen Widerspruch zu der Annahme darstellt,

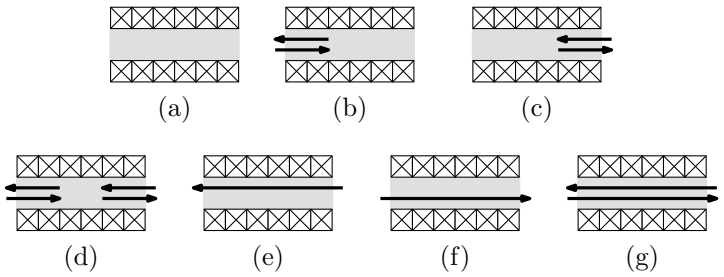


Abbildung 7.7: Sieben gültige und potentiell verwendete Gangoperationen.

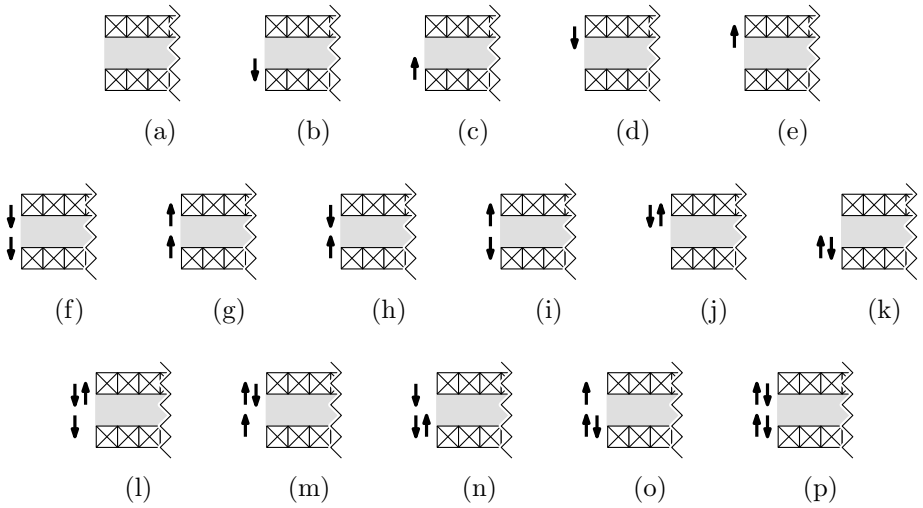


Abbildung 7.8: 16 gültige und potentiell verwendete Zwischen-Gangoperationen.

---

dass T optimal ist. □

Lemma 7.3.2. *Gegeben ist eine optimale Tour T in Bezug auf eine Menge an Punkten. Zwei adjazente Punkte v und w werden dann innerhalb von T höchstens zweimal unmittelbar hintereinander besucht.*

Beweis. Der Beweis dafür folgt direkt aus Theorem 7.3.1. Unter der Annahme, dass v und w mehr als zweimal unmittelbar hintereinander besucht werden, müßte der Weg von v nach w mindestens zweimal in derselben Richtung durchlaufen werden. □

Ausgehend von Theorem 7.3.1 und Lemma 7.3.2 kann man schließen, dass es sieben Kombinationen von Gangoperationen gibt, die in einer optimalen Tour Anwendung finden können (siehe Abb. 7.7). Natürlich kann ein Gang auch passiert (ausgelassen) werden, wenn keine Artikel entnommen werden müssen. Weiters kann die Anzahl an möglichen Zwischen-Gangoperationen, die für eine optimale Tour Relevanz haben können, von oben mit 16 eingeschränkt werden (siehe Abb. 7.8). Daher kann man mit einer festen Anzahl an generell möglichen und verwendbaren Modulen für optimale

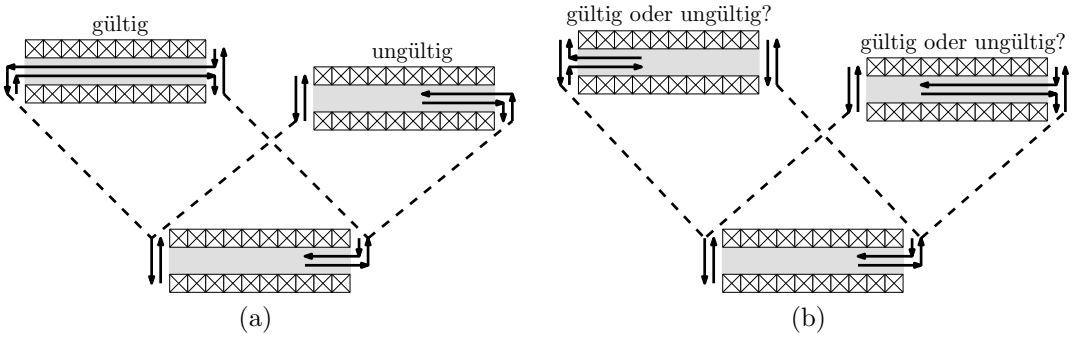


Abbildung 7.9: Sind diese Module gültig oder ungültig?

Touren arbeiten. An dieser Stelle sei nochmals darauf hingewiesen, dass leere Operationen, wie in Abb. 7.7a und Abb. 7.8a, theoretisch Gänge repräsentieren sollen, die nicht betreten oder erreicht werden müssen und im entwickelten Programm gar nicht zum Einsatz kommen. In solchen Fällen werden diese Gänge behandelt als würden sie für die jeweils konkrete Tour nicht existieren.

Obwohl nun ersichtlich ist, dass das Auffinden von Touren durch Auswahl geeigneter Module bewerkstelligt werden kann, kann man beobachten, dass das Zusammenspiel mancher Module zu letztlich ungültigen oder nicht rekonstruierbaren Touren führt (siehe Abb. 7.9a). Hier ist zu sehen, dass durch die Kombination zweier Module eigentlich zwei voneinander getrennte Touren entstehen würden. Wie in Abb. 7.9b zu sehen ist, kann die Entscheidung, ob die Kombination von Modulen gültig ist, nicht immer zu dem Zeitpunkt getroffen werden, zu dem ein Modul gewählt wird. Manchmal bestimmt erst ein später hinzugefügtes Modul darüber, ob die Kombination gültig oder ungültig ist. Sei $\mathcal{M}_{\mathrm{g}}(j)$ die Menge von Modulen $j' \in \mathcal{M}_{\mathrm{komp}}(j)$, wobei die Verwendung von Modul j in einer gültigen (Sub-)Tour resultiert.

Rekursive Update Funktion Folgend aus den zuvor erwähnten Beobachtungen, sollen zwei $(n+1) \times (\nu)$ Matrizen σ und τ eingeführt werden, wobei n die Anzahl an Gängen mit einzusammelnden Artikeln und ν die Anzahl an verwendbaren Modulen sind. Ein Eintrag σ_{ij}, mit $1 \leq i \leq n$ und $1 \leq j \leq \nu$, steht für die Länge einer gültigen Tour T', welche alle Gänge $1, \ldots, i$ besucht, das heißt alle benötigten Artikel aus den Gängen 1 bis i einsammelt und Modul j für Gang i verwendet. Analog dazu steht ein Eintrag τ_{ij} für die Länge einer Tour T'', welche wiederum die Gänge $1, \ldots, i$ besucht und Modul j für Gang i verwendet. Im Gegensatz zu Tour T' besteht aber bei Tour T'' die Möglichkeit durch Verwendung bestimmter Module in den Gängen $i+1$ bis n, ungültig zu werden.

Die Einträge für σ und τ berechnen sich mittels folgender rekursiver Funktion:

$$\sigma_{0\mu} = 0 \tag{7.1}$$

$$\tau_{0\mu} = 0 \tag{7.2}$$

$$\sigma_{0j} = \infty \qquad \text{für } j \in \{1, \ldots, \nu\} \setminus \{\mu\} \tag{7.3}$$

$$\tau_{0j} = \infty \qquad \text{für } j \in \{1, \ldots, \nu\} \setminus \{\mu\} \tag{7.4}$$

$$\sigma_{ij} = c_i(j) + \min \left\{ \begin{array}{l} \{\sigma_{i-1j'} : j' \in \mathcal{M}_{\mathrm{g}}(j)\} \cup \\ \{\tau_{i-1j'} : j' \in \mathcal{M}_{\mathrm{g}}(j)\} \end{array} \right\} \quad \begin{array}{l} \text{für } i \in \{1, \ldots, n\} \\ \text{für } j \in \{1, \ldots, \nu\} \end{array} \tag{7.5}$$

$$\tau_{ij} = c_i(j) + \min \left\{ \begin{array}{l} \{\sigma_{i-1j'} : j' \in \mathcal{M}_{\mathrm{komp}}(j)\} \cup \\ \{\tau_{i-1j'} : j' \in \mathcal{M}_{\mathrm{komp}}(j)\} \end{array} \right\} \quad \begin{array}{l} \text{für } i \in \{1, \ldots, n\} \\ \text{für } j \in \{1, \ldots, \nu\} \end{array} \tag{7.6}$$

In diesem Fall repräsentiert Modul μ die Verdichtungszone. Wenn ein Modul j für Gang i nur ungültige Touren bilden oder nicht kompatibel nach unten sein würde, so wird $\sigma_{ij} = \infty$, beziehungsweise $\tau_{ij} = \infty$ gesetzt.

Dekodierung einer optimalen Tour Sind nun die beiden Matrizen σ und τ wie beschrieben initialisiert worden, dann kann man die optimale Tour einfach daraus ableiten. Zuerst muss das Modul für Gang n in der optimalen Tour gewählt werden, welches jener Eintrag σ_{nj} repräsentiert, der minimal ist für alle j. Um die bereits gemachten Überlegungen zu ergänzen, sind alle gültigen Module in Gang n solche, die zwar mittels Zwischen-Gangoperationen zum Gang $n-1$ verbinden, nicht aber zu Gang $n+1$.

Zur Vereinfachung nehmen wir an dieser Stelle an, dass σ_{nJ} das Modul für Gang n in einer optimalen Tour bezeichnet. Es ist dann leicht zu zeigen, dass so ein Eintrag existieren muss. Nun kann man einfach innerhalb der Matrix zurückverfolgen, dass das Modul für Gang $n-1$ jenes Modul j' ist, für das die Gleichung (7.7) erfüllt ist:

$$\sigma_{nJ} - c_i(j) = \begin{cases} \sigma_{i-1j'} \\ \tau_{i-1j'} \end{cases} \tag{7.7}$$

Für alle weiteren Gänge lässt sich die optimale Tour entsprechend ableiten. Sollte der Fall eintreten, dass mehr als ein Modul die Gleichung (7.7) erfüllt, so ist das gleichbedeutend damit, dass es mehrere optimale Touren gibt. Aus diesen Touren ist dann eine beliebige wählbar.

7.3.3 S-Shape-Heuristik

Wie bereits in Kapitel 3.4 erwähnt, gibt es Arbeiten, welche zur Berechnung von Touren auf die so genannte *S-Shape*-Heuristik zurückgreifen. Das Verfahren folgt einem einfachen Prinzip, wonach sich die Arbeiter auf S-förmigen Touren durch das

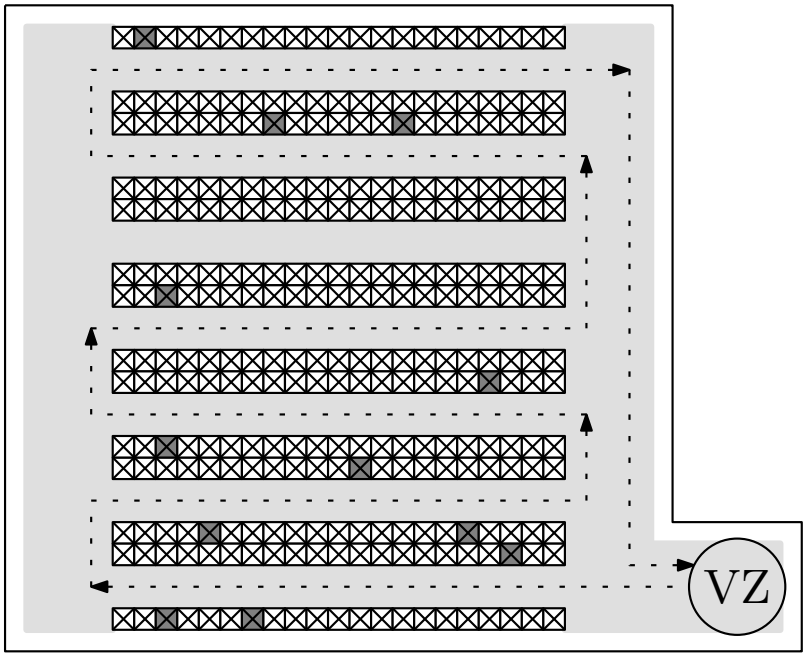


Abbildung 7.10: Veranschaulichung der S-Shape-Heuristik

Lager bewegen. Dabei wird ein Gang durchquert sobald er betreten wurde. Natürlich werden Gänge, aus denen kein Artikel geholt werden muss, ausgelassen. Zur Veranschaulichung dieser Vorgehensweise soll Abb. 7.10 dienen. Laut den bereits referenzierten Arbeiten [4] sowie [5] kann man auf diese Weise Lösungen finden, die für viele Probleminstanzen nahe am Optimum liegen. Das ist der Grund dafür, weshalb diese Heuristik in Kombination mit TSH im Rahmen dieser Arbeit als Referenzverfahren eingesetzt wurde. Damit kann man einen Vergleich zu den mittels des entwickelten hybriden Verfahrens berechneten Ergebnissen anstellen und deren Qualität einschätzen. Hierbei sollte darauf hingewiesen werden, dass es sich bei den mittels *S-Shape*-Heuristik gefundenen Werten nur um Richtwerte handelt. Im Gegensatz zum vorgestellten Algorithmus werden keine Nebenbedingungen geprüft. Daher werden beispielsweise weder Kollisionen vermieden und Artikel stets vom ersten gefundenen Lagerplatz geholt anstatt nach alternativen Lagerplätzen zu suchen.

7.4 Zuordnung von Arbeitern zu Touren

Zusätzlich zu den bisherigen Schritten ist es noch wichtig eine Aufteilung aller berechneten Touren auf die verfügbaren Lagerarbeiter zu erstellen. Das bedeutet, dass jedem Arbeiter eine Liste von Touren zugewiesen wird, die von diesem dann abgearbeitet werden muss. Nachdem sich aber alle Arbeiter gleichzeitig im Lager bewegen, müssen entsprechende Maßnahmen getroffen werden, um gegenseitige Behinderungen auszuschließen.

Algorithmus 9 : Initiale Arbeiter-Zuordnung

Input : eine Menge $\mathcal{T}$ von Touren T_i, mit $i = 1 \dots m$; die Anzahl an Arbeitern w

Data : Menge $\mathcal{W}$ von Mengen W_i, mit $i = 1 \dots m$, von abzuarbeitenden Touren, Zähler l

```
begin
    foreach W_i ∈ 𝒲 do
        W_i ← ∅ ;
    j ← 1 ;
    foreach T_i ∈ 𝒯 do
        W_j ← W_j ∪ {T_i} ;
        if l < m then
            l ← l + 1 ;
        else
            l ← 1 ;
    return 𝒲 ;
end
```

Eine Zuordnung von Arbeitern zu Touren soll also genau dann gültig sein, wenn es zu keinem Zeitpunkt zu Kreuzungen der Touren zweier Arbeiter innerhalb von Regalgängen kommt. Des weiteren sind Überholmanöver in Gängen nicht zulässig. Um das berücksichtigen zu können, muss die Geschwindigkeit der Kommissionierungswagen genauso beachtet werden, wie auch die Zeit, die notwendig ist, um die jeweiligen Artikel auszufassen.

Trotz all dieser formalen Überlegungen darf man nicht vergessen, dass immer noch Menschen in diesem Lager arbeiten, weshalb man entsprechende Zeitpuffer einplanen muss. Die Arbeitsgeschwindigkeit ist eben abhängig von sehr vielen Faktoren, wie etwa Alter und Erfahrung der Arbeiter, oder äußeren Einflüssen aus dem Umfeld des Lagers.

7.4.1 Konstruktionsheuristik

Zur Erzeugung einer Startlösung für weitere Berechnungen muss zunächst eine initiale Zuordnung von Arbeitern zu Touren erfolgen, indem die vorhandenen Touren gleichmäßig auf alle Arbeiter verteilt werden. Der einfache Ansatz einer First-Fit Heuristik wird zum Auffinden einer solchen Initiallösung herangezogen (siehe Alg. 9). Jedenfalls stellt eine solche erste Lösung keinerlei Anspruch auf Korrektheit. Zum einen können Kollisionen der Arbeiter innerhalb von Regalgängen auftreten, zum anderen kann nicht garantiert werden, dass alle angeforderten Artikel innerhalb des zulässigen Zeitrahmens zur Verdichtungszone gebracht werden. Es müssen also noch weitere Schritte folgen, um eine Zuordnung gültig zu machen. Auch hier hilft wieder

die Umsetzung einer VNS weiter.

7.4.2 Reparatur- und Verbesserungsheuristik

Zusätzlich zur Entscheidung, ob eine Zuordnung von Arbeitern zu Touren gültig ist, wird versucht eine Aufteilung zu finden, bei der alle Arbeiter mit ihren Arbeitsschritten so früh wie möglich fertig werden. Zu diesem Zweck bietet sich eine Zielfunktion an, die den letzten Endzeitpunkt aller Arbeiter auswertet.

Für ein sinnvolles VNS Verfahren muss nun noch eine Anzahl an Nachbarschaften (hier anhand von *Moves*) definiert werden. Sei $\mathcal{W} = \bigcup_{i=1}^{w}\{W_i\}$ die Menge aller Zuordnungen W_i von Arbeitern zu Touren. Weiters nehme man an, dass $\mathcal{T} = \bigcup_{i=1}^{m}\{T_i\}$ die Menge aller zuvor berechneter Touren beschreibt. Folgende *Moves* bilden die Basis für die verwendeten Nachbarschaften:

WorkerSwapMove(i, j, k, l) Dieser Schritt definiert einfach den Austausch von Tour $k \in W_i$ und $l \in W_j$, mit $1 \leq i < j \leq m$ und $k \neq l$. Das bedeutet schlicht, dass zwischen zwei Arbeitern je eine Tour ausgetauscht wird.

WorkerShiftMove(i, j, k) Hier wird lediglich eine Tour $k \in W_i$ aus W_i entfernt und in W_j an letzter Stelle angefügt, wobei $1 \leq i, j \leq m$ und $i \neq j$. Das kommt einem Verschieben einer Tour von einem Arbeiter zu einem anderen gleich.

WorkerShiftTourMove(i, j, k) Bei diesem Schritt wird Tour j, mit $j \in W_i$, um k Positionen verschoben, wobei $|k| < |W_i|$ und $1 \leq j + k \leq |W_i|$. Eine solche Umreihung kann in Bezug auf die termingerechte Abwicklung hilfreich sein.

WorkerSplitMove(i, R) Bei der Anwendung dieses Schrittes werden alle Elemente in R aus W_i entfernt und in eine neue Zuordnung W_i' eingefügt, welche dann einem neuen Arbeiter zugewiesen werden kann. Dieser Schritt kann natürlich nur angewendet werden, solange noch freie Arbeiter zur Verfügung stehen.

Aus diesen *Moves* ergeben sich die Nachbarschaftsstrukturen *WorkerSwap*, *WorkerShift*, *WorkerShiftTour* und *WorkerSplit*.

Nachdem das Hauptziel dieser VNS Prozedur das Auffinden einer gültigen Zuordnung von Touren zu Arbeitern ist, werden nur Moves berücksichtigt, deren Resultat eine gültige Lösung darstellt.

7.5 Erweiterter Algorithmus

Da die hier vorgeschlagenen Methoden insgesamt dafür ausgelegt sind in vergleichbar kurzer Zeit relativ gute Lösungen zu erzeugen, ist es durchaus wahrscheinlich, dass die gefundenen Lösungen noch weiter verbessert werden könnten. Zudem sind

Algorithmus 10 : Erweiterter Algorithmus

Input : Bestellungen mit Listen aller einzusammelnder Artikel
Output : Eine Zuordnung von Arbeitern zu fertigen Touren

```
begin
    repeat
(1)     Erstelle Partitionierung vorhandener Bestellungen ;
        repeat
            repeat
(2)             Berechne Zuordnung von Artikeln zu Touren ;
(3)             Berechne optimale Touren für die oben erstellte Zuordnung ;
            until keine Verbesserung konnte erzielt werden ;
(4)         Erstelle eine Zuordnung von Arbeitern zu Touren ;
            return die aktuelle Zuordnung von Arbeitern zu Touren ;
        until ein definiertes Abbruchkriterium ist erfüllt ;
    until keine weiteren Bestellungen sind abzuarbeiten ;
end
```

zu dem Zeitpunkt, zu dem der Algorithmus zu arbeiten beginnt, noch nicht alle Bestellungen bekannt, die im Laufe des Tages zu bearbeiten sein werden. Deshalb empfiehlt es sich eine erweiterte Variante des Alg. 3 wie in Alg. 10 beschrieben zu verwenden. Anstelle eines Abbruchs, sobald eine Zuordnung von Touren zu Arbeitern gefunden wurde, soll der Algorithmus hier wieder von vorne beginnen, allerdings unter Berücksichtigung von bereits bearbeiteten Aufteilungen. Das bedeutet etwa, dass bereits eingesammelte Artikel natürlich nicht mehr vertauscht werden können, oder, dass Arbeiter, die bereits auf dem Weg zu einem Artikel sind nicht mehr anders geschickt werden können. Sonstige Änderungen können natürlich noch durchgeführt werden, solange sie eine Verbesserung der Gesamtlösung bringen. Der Algorithmus sollte dann arbeiten, bis alle Bestellungen anforderungsgemäß bearbeitet wurden.

Um den Algorithmus möglichst effizient zu gestalten, müssen natürlich auch die darunterliegenden Datenstrukturen entsprechend gewählt werden und Bewertungsfunktionen oder im Speziellen die Funktion zur Berechnung optimaler Touren möglichst schnell arbeiten.

8 Testergebnisse

Um die Verwendbarkeit der vorgestellten Methode im industriellen Umfeld abschätzen zu können, wurden zahlreiche Testläufe durchgeführt. Als Hardware wurde ein zur Zeit aktuelles Arbeitsplatzsystem gewählt – konkret ein Dual Xeon mit 2.6 GHz und 8 GB RAM. Leider war es nicht möglich Echtdaten zu bekommen, da zum Zeitpunkt der Fertigstellung dieser Diplomarbeit noch keine hardwaremäßige Realisierung im Ersatzteillager unseres Industriepartners vorgenommen wurde, weswegen auf statistische Parameter, die im Laufe des vergangenen Jahres gesammelt wurden, zurückgegriffen werden musste, um zufällige Testinstanzen zu generieren. Hierzu wurde das Lager modelliert und die einzusammelnden Artikel den Parametern entsprechend zufällig im Lager verteilt. Zusätzlich wurden typische Kundenbestellungen erzeugt. Variable Parameter waren dabei die Summe der insgesamt bestellten Artikel und die Anzahl der Kundenbestellungen. Nach Rücksprache mit den Lagerleitern wurde festgesetzt, dass Rechenzeiten bis 20 Minuten als akzeptabel gelten, weswegen die maximal verfügbare Rechenzeit auf diese 20 Minuten gesetzt wurde.

8.1 Wahl der Konstruktionsheuristik

In einem ersten Vortest wurde untersucht wie die beiden Konstruktionsheuristiken zur Findung einer Ausgangslösung den gesamten Algorithmus beeinflussen. Die verwendeten Testinstanzen sind nicht ident mit jenen der späteren Testläufe, da durch die im Laufe der Entwicklung entstandenen Erweiterungen des Programms auch die Testdaten überarbeitet werden mussten. Deshalb sind die Ergebnisse dieses Vortests nicht direkt mit den späteren vergleichbar. In Tab. 8.1 sind die gesammelten Ergebnisse ersichtlich. Sie zeigen die absoluten Werte der Summe aller Tourlängen für die in Kapitel 7.2.1 vorgestellten Konstruktionsheuristiken CAH und TSH.

Natürlich sind diese Werte für sich noch nicht sehr aussagekräftig, vergleicht man sie allerdings miteinander, so lassen sich gewisse Beobachtungen machen. Zunächst fällt auf, dass die errechneten Werte der Tourlängen nicht in unmittelbarem Zusammenhang mit der gewählten Methode zur Erzeugung einer Startlösung gebracht werden können. Es gibt in beiden Fällen an manchen Stellen überdurchschnittlich große Abweichungen. Abgesehen von wenigen Ausnahmen schwanken die Werte für die Tourlängen im direkten Vergleich zwischen den beiden Konstruktionsheuristiken kaum. Das legt die Vermutung nahe, dass die Wahl der Konstruktionsheuristik, wenn überhaupt, nur einen vernachlässigbaren Einfluss auf die endgültige Lösung hat.

Tabelle 8.1: Absolute Werte der Tourlängen (Spalten CAH und TSH). Alle Werte sind Durchschnittswerte über 20 Durchläufe (in Klammern sind jeweils die Standardabweichungen angeführt).

		Vortest			
In	Ar	TSH		CAH	
S1	50	9450.11	(68.94)	9428.10	(71.05)
S2	50	6961.53	(68.51)	6960.26	(48.91)
S3	50	8810.44	(30.38)	8812.79	(34.52)
S4	50	6587.15	(83.36)	7631.04	(3149.92)
S5	50	7003.12	(87.93)	7010.59	(64.48)
M1	100	15724.93	(95.96)	15747.49	(80.36)
M2	100	12266.77	(5159.42)	15229.84	(3113.72)
M3	100	10187.26	(114.72)	10188.83	(122.95)
M4	100	13872.25	(105.74)	13839.25	(131.25)
M5	100	14485.47	(5062.99)	16471.11	(4194.89)
L1	150	14772.86	(190.65)	14778.72	(174.84)
L2	150	15712.18	(162.93)	17217.15	(3677.97)
L3	150	18052.87	(3706.13)	19439.24	(200.45)
L4	150	18357.51	(167.09)	18287.73	(121.11)
L5	150	20216.30	(4847.74)	21540.72	(5330.36)
X1	200	22441.44	(4263.34)	23766.84	(5223.52)
X2	200	20354.92	(4331.64)	22260.29	(189.05)
X3	200	25609.66	(361.38)	29492.64	(5359.09)
X4	200	18393.69	(3773.50)	25555.01	(3443.50)
X5	200	19685.29	(211.60)	23847.93	(5303.58)

8.2 Wahl von Berechnungsparametern

Der erste durchgeführte Testlauf sollte nun zeigen, wie sich gewisse Berechnungsparameter auf den Algorithmus auswirken. Zwei wesentliche Steuerungsmöglichkeiten sind zum einen das Erlauben oder Verbieten des Umkehrens innerhalb von Gängen, was bedeutet, dass ein Gang komplett durchquert werden muss, falls er betreten wird. Zum anderen ist das die Verwendung von statischen oder dynamischen Reihenfolgen für die Nachbarschaften der VNS. Wie bereits in Kapitel 7.2.4 beschrieben wurde, erfolgt bei dynamischer Nachbarschaftsreihenfolge laufend eine Umreihung dieser Strukturen, um Lösungsverbesserungen durch normalerweise später durchsuchte Nachbarschaften, die dann möglicherweise gar keine Verbesserungen erzielen, schon früher zu ermöglichen. In Tab. 8.2 (ohne Artikel auf alternativen Lagerplätzen) und Tab. 8.3 (mit Artikeln auf alternativen Lagerplätzen) sind die gesammelten Ergebnisse des Testlaufs (Testlauf 1) zusammengefasst. Dabei wird die mittels *S-Shape*-Heuristik (siehe Kapitel 7.3.3) gefundene Lösung als Referenz herangezogen. Die berechneten Werte werden dann prozentuell dazu angegeben, sodass ein direkter Vergleich möglich ist. Da alle Werte den Durchschnitt über 20 Durchläufe repräsentieren, wird auch die Standardabweichung angeführt. Zu beachten ist hierbei, dass die mittels *S-Shape*-Heuristik erzielten Werte nur als Anhaltspunkt dienen und keinerlei Nebenbedingungen berücksichtigt wurden. Zusätzlich sind noch die Durchschnittswerte für Laufzeit und Kapazitätsauslastung der Kommissionierungswagen angegeben. Verwendet wurden die Nachbarschaften *Split*, *Merge*, *Shift*, *Swap*, *SwapPosition*, *SwapPositionMerge*, *SplitPositionMerge* und *DoubleShift*. Diese Auflistung entspricht gleichzeitig der Anfangsreihenfolge der Nachbarschaften.

Man kann nun beim Untersuchen der Tabellen einige interessante Entdeckungen machen. Zunächst möchte ich auf den Vergleich von statischer und dynamischer Nachbarschaftsreihenfolge eingehen. Hier ist bei den kleinen Instanzen mit 25 Artikeln zu sehen, dass die Werte der Zielfunktionen sehr nahe beisammen liegen, bei dynamischer Nachbarschaftsreihenfolge dennoch leicht bessere Werte vorliegen. Die Laufzeit ist allerdings geringfügig länger, wenn auch nicht sehr viel. Interessant hingegen sind die Instanzen mit 50 Artikeln, bei denen sowohl ein besseres Laufzeitverhalten, als auch ein besserer Zielfunktionswert zu bemerken ist, wenn dynamische Nachbarschaftsreihenfolge verwendet wird. Auch bei jenen Instanzen mit 100 Artikeln bringt die Berechnung mittels dynamischer Nachbarschaftsreihenfolge stets ein besseres Ergebnis in beiden Bereichen. Für die größeren Instanzen mit 200 Artikeln wiederum muss man sagen, dass die dynamische Reihenfolge der Nachbarschaften keinen klaren Vorteil bringt. Das mag aber möglicherweise daran liegen, dass bei so vielen Artikeln große Teile des Lagers durchlaufen werden müssen, also kaum Wege eingespart werden können und somit die Optimierung weniger Optionen zur Verfügung hat. Außerdem werden nach der gegebenen zeitlichen Beschränkung die Berechnungen unterbrochen. Um die Auslastung der Kommissionierungswagen nicht außer Acht zu lassen, sei erwähnt, dass die Durchschnittswerte eine durchwegs hohe Belegung zeigen, was deutlich für die Effizienz der Berechnungen spricht. Gesamt betrachtet lässt

Tabelle 8.2: Ergebnisse der Testläufe 1 und 2 *ohne* Artikeln auf alternativen Lagerplätzen - Durchschnittswerte über 20 Testläufe mit 25, 50, 100 und 200 einzusammelnden Artikeln (Ar). Angeführt sind der Zielfunktionswert (f(x) in Prozent [%]) relativ zur mittels *S-Shape* konstruierten Referenzlösung ($f_{SS}(x)$) und mit absoluter Standardabweichung in Klammern, die Laufzeit (t in Sekunden [s]) und die Kapazitätsauslastung der Kommissionierungswagen (C in Prozent [%]). Getestet wurden die Instanzen (In) *ohne* Artikeln auf alternativen Lagerplätzen bei Berechnung *mit* und *ohne* Umkehren innerhalb eines Ganges und bei *statischer* und *dynamischer* Nachbarschaftsreihenfolge.

			Testlauf 1												Testlauf 2					
			statische Reihenfolge						dynamische Reihenfolge						dynamische Reihenfolge					
			ohne Umkehren			mit Umkehren			ohne Umkehren			mit Umkehren			ohne Umkehren			mit Umkehren		
In	Ar	$f_{SS}(x)$	f(x)	t	C	f(x)	t	C	f(x)	t	C	f(x)	t	C	f(x)	t	C	f(x)	t	C
S1	25	2135.47	75.2 (4.74)	92.05	80.4	72.6 (2.22)	119.63	87.1	74.8 (1.38)	105.77	80.6	72.5 (1.27)	156.23	86.1	74.7 (0.88)	300.90	86.9	72.6 (1.21)	401.12	86.7
S2	25	2150.77	68.6 (8.27)	81.05	72.5	67.4 (8.19)	111.65	80.8	67.6 (1.92)	83.64	69.9	66.6 (0.64)	152.07	77.5	67.3 (0.56)	342.29	74.4	66.5 (0.10)	420.56	79.2
S3	25	2368.25	73.0 (1.29)	108.64	85.6	70.2 (3.37)	135.41	85.6	72.9 (0.00)	119.23	85.6	70.1 (1.19)	148.48	85.6	70.9 (0.00)	270.27	86.2	69.5 (0.72)	381.24	80.2
S4	25	2320.21	68.5 (7.02)	89.24	72.7	68.0 (1.04)	104.24	72.7	68.3 (3.48)	90.77	72.7	67.8 (0.66)	129.77	72.7	67.6 (0.00)	241.96	74.2	67.6 (0.00)	299.53	68.2
S5	25	2352.01	66.3 (3.53)	82.75	78.2	63.4 (6.85)	92.97	83.7	66.2 (3.37)	103.56	78.1	63.1 (2.56)	146.03	84.0	66.1 (0.63)	286.96	85.2	63.0 (1.26)	391.50	82.8
M1	50	3658.73	225.1 (433.63)	829.65	99.9	210.6 (520.43)	1117.14	99.9	200.9 (485.90)	649.32	99.9	189.0 (411.04)	985.47	99.9	73.2 (14.86)	1185.23	99.6	68.9 (9.42)	>max	99.6
M2	50	3908.69	101.1 (192.75)	921.98	99.3	95.2 (143.38)	1158.00	99.3	98.1 (154.15)	638.38	99.3	87.4 (126.26)	957.56	99.3	71.9 (9.54)	1151.07	99.0	68.1 (6.47)	>max	99.0
M3	50	4325.10	133.8 (258.82)	832.84	99.8	127.8 (196.96)	1138.18	99.8	119.9 (152.27)	660.23	99.8	118.0 (149.04)	894.85	99.8	66.6 (13.52)	1137.23	99.6	61.2 (9.23)	>max	99.6
M4	50	3886.14	106.3 (286.22)	791.38	99.2	93.4 (191.30)	1133.87	99.2	96.2 (183.15)	685.06	99.2	101.7 (230.13)	947.45	99.2	68.2 (13.04)	1153.65	99.0	64.7 (6.59)	>max	99.0
M5	50	4160.70	196.8 (1133.36)	974.95	99.8	179.1 (889.70)	1189.09	99.8	135.4 (226.24)	651.23	99.8	120.1 (287.27)	1026.55	99.8	69.4 (7.65)	1158.30	99.6	64.7 (7.84)	>max	99.6
L1	100	6461.82	75.7 (37.55)	>max	99.1	72.7 (26.68)	>max	99.1	75.6 (31.22)	>max	99.1	71.2 (13.48)	>max	99.1	77.7 (41.05)	>max	99.0	68.7 (21.73)	>max	99.0
L2	100	6622.66	87.7 (156.68)	>max	99.9	110.8 (607.07)	>max	99.9	81.5 (173.85)	>max	99.9	81.6 (157.74)	>max	99.9	70.9 (49.67)	>max	99.6	66.5 (46.18)	>max	99.6
L3	100	6581.63	94.5 (174.52)	>max	99.8	95.0 (142.52)	>max	99.8	88.2 (153.86)	1188.82	99.8	82.1 (179.97)	>max	99.8	73.8 (40.10)	>max	99.6	68.1 (50.82)	>max	99.6
L4	100	6740.34	104.8 (303.67)	>max	99.8	115.3 (310.36)	>max	99.8	95.2 (196.42)	>max	99.8	102.1 (307.57)	>max	99.8	72.0 (68.85)	>max	99.6	67.4 (43.64)	>max	99.6
L5	100	6560.45	87.4 (223.85)	>max	99.8	99.7 (433.03)	>max	99.8	85.9 (136.25)	>max	99.8	79.6 (180.27)	>max	99.8	71.7 (36.90)	>max	99.6	67.4 (26.39)	>max	99.6
X1	200	10820.65	102.7 (285.88)	>max	99.9	115.2 (532.23)	>max	99.9	104.3 (507.35)	>max	99.9	694.9 (43803.86)	>max	96.8	81.9 (54.59)	>max	99.7	382.8 (23561.90)	>max	98.4
X2	200	11049.59	84.0 (146.84)	>max	99.5	93.8 (344.94)	>max	99.5	147.9 (4458.14)	>max	99.5	103.9 (1926.03)	>max	99.5	80.3 (51.32)	>max	99.3	118.7 (5157.68)	>max	99.3
X3	200	10987.27	129.1 (332.14)	>max	99.9	148.6 (531.30)	>max	99.9	121.5 (251.67)	>max	99.9	335.7 (12189.38)	>max	99.8	83.3 (66.47)	>max	99.6	72.9 (65.01)	>max	99.6
X4	200	11011.98	93.3 (177.47)	>max	99.7	107.3 (372.87)	>max	99.7	109.9 (1469.24)	>max	99.7	152.2 (5119.61)	>max	99.7	100.3 (1978.89)	>max	99.5	226.4 (10815.91)	>max	99.5
X5	200	11428.16	114.9 (258.96)	>max	100.0	122.5 (591.60)	>max	100.0	116.0 (703.96)	>max	100.0	100.0 (358.40)	>max	99.0	127.4 (5058.53)	>max	99.8	111.6 (4570.83)	>max	99.1

Tabelle 8.3: Ergebnisse der Testläufe 1 und 2 *mit* Artikeln auf alternativen Lagerplätzen - Durchschnittswerte über 20 Testläufe mit 25, 50, 100 und 200 einzusammelnden Artikeln (Ar). Angeführt sind der Zielfunktionswert (f(x) in Prozent [%]) relativ zur mittels *S-Shape* konstruierten Referenzlösung ($f_{SS}(x)$) und mit absoluter Standardabweichung in Klammern, die Laufzeit (t in Sekunden [s]) und die Kapazitätsauslastung der Kommissionierungswagen (C in Prozent [%]). Getestet wurden die Instanzen (In) *mit* Artikeln auf alternativen Lagerplätzen bei Berechnung *mit* und *ohne* Umkehren innerhalb eines Ganges und bei *statischer* und *dynamischer* Nachbarschaftsreihenfolge.

			Testlauf 1												Testlauf 2					
			statische Reihenfolge						dynamische Reihenfolge						dynamische Reihenfolge					
			ohne Umkehren			mit Umkehren			ohne Umkehren			mit Umkehren			ohne Umkehren			mit Umkehren		
In	Ar	$f_{SS}(x)$	f(x)	t	C	f(x)	t	C	f(x)	t	C	f(x)	t	C	f(x)	t	C	f(x)	t	C
S1	25	2147.27	74.8 (5.23)	62.30	76.9	72.2 (1.76)	104.79	80.5	74.4 (0.95)	136.75	80.0	72.1 (1.74)	178.48	84.5	74.4 (2.36)	454.55	83.5	72.0 (0.00)	523.18	90.2
S2	25	2150.77	68.1 (7.43)	59.76	69.7	66.8 (6.82)	91.76	73.3	67.5 (1.41)	117.70	70.2	66.2 (0.97)	143.19	74.6	67.3 (0.92)	360.31	74.8	66.2 (0.85)	498.57	78.2
S3	25	2368.25	72.9 (0.95)	111.35	85.6	70.3 (2.39)	131.97	85.6	73.0 (1.26)	92.90	85.6	70.1 (0.97)	170.86	85.6	70.9 (0.02)	259.24	86.2	69.4 (0.61)	360.82	86.2
S4	25	2320.21	68.3 (4.26)	90.92	72.7	67.9 (0.99)	122.70	72.7	68.1 (1.82)	94.71	72.7	67.8 (0.73)	134.27	72.7	67.6 (0.00)	250.91	74.2	67.6 (0.00)	325.90	74.2
S5	25	2310.23	67.8 (9.99)	20.82	79.3	64.2 (8.66)	20.82	82.5	66.7 (2.68)	91.48	78.1	63.5 (3.88)	102.28	79.4	66.5 (1.40)	265.08	81.6	63.2 (1.16)	472.63	84.5
M1	50	3658.73	188.6 (448.05)	946.93	99.9	200.4 (461.85)	1096.41	99.9	243.8 (477.00)	650.55	99.9	191.1 (396.50)	911.15	99.9	72.7 (14.80)	1197.71	99.6	69.2 (10.27)	>max	99.6
M2	50	3932.43	104.7 (141.95)	804.04	99.3	99.8 (179.04)	1118.80	99.3	92.3 (108.51)	797.88	99.3	85.5 (143.83)	901.94	99.3	71.2 (8.86)	1181.24	99.0	67.6 (7.70)	>max	99.0
M3	50	4256.39	130.5 (259.84)	694.81	99.8	123.8 (288.40)	1122.85	99.8	127.5 (164.63)	669.46	99.8	106.5 (132.44)	973.31	99.8	66.8 (10.71)	1173.11	99.6	62.1 (9.11)	>max	99.6
M4	50	3871.34	108.6 (228.32)	945.89	99.2	107.1 (226.38)	1153.29	99.2	95.0 (163.61)	669.68	99.2	89.5 (178.11)	973.45	99.2	70.3 (11.14)	1146.09	99.0	66.3 (9.61)	>max	99.0
M5	50	4160.70	176.7 (497.16)	950.52	99.8	169.6 (688.33)	1160.87	99.8	138.0 (263.29)	684.97	99.8	128.1 (210.22)	1081.84	99.8	68.9 (12.12)	1179.74	99.6	64.8 (9.14)	>max	99.6
L1	100	6448.32	75.5 (33.27)	>max	99.1	71.7 (25.18)	>max	99.1	75.4 (20.19)	1189.46	99.1	71.0 (22.46)	>max	99.1	74.1 (37.37)	>max	99.0	70.9 (56.67)	>max	99.0
L2	100	6548.31	94.4 (211.54)	>max	99.9	92.5 (366.95)	>max	99.9	83.9 (148.72)	1187.82	99.9	88.5 (225.18)	>max	99.9	70.0 (25.04)	>max	99.6	66.2 (26.12)	>max	99.6
L3	100	6588.51	88.1 (147.70)	>max	99.8	93.9 (255.14)	>max	99.8	88.9 (140.42)	>max	99.8	84.0 (170.39)	>max	99.8	75.1 (59.88)	>max	99.6	67.9 (25.49)	>max	99.6
L4	100	6740.34	98.3 (214.82)	>max	99.8	104.9 (286.70)	>max	99.8	94.4 (193.17)	1199.49	99.8	94.4 (171.96)	>max	99.8	71.8 (73.46)	>max	99.6	67.3 (46.97)	>max	99.6
L5	100	6560.45	84.5 (188.72)	>max	99.8	91.1 (262.57)	>max	99.8	82.6 (166.42)	>max	99.8	78.0 (163.04)	>max	99.8	72.2 (39.23)	>max	99.6	67.0 (43.27)	>max	99.6
X1	200	10925.71	98.6 (300.98)	>max	99.9	117.0 (392.63)	>max	99.9	101.1 (474.02)	>max	99.9	127.8 (4345.26)	>max	99.9	82.5 (55.80)	>max	99.7	141.1 (7227.99)	>max	100.0
X2	200	10997.52	86.9 (207.66)	>max	99.5	86.0 (296.29)	>max	99.5	84.1 (160.55)	>max	99.5	278.2 (15585.21)	>max	99.5	81.9 (69.20)	>max	99.3	108.5 (3542.61)	>max	99.3
X3	200	10932.39	135.9 (374.26)	>max	99.9	151.7 (528.37)	>max	99.9	123.9 (301.01)	>max	99.9	116.3 (312.00)	>max	99.9	141.9 (6124.02)	>max	99.6	418.9 (12399.40)	>max	99.6
X4	200	11011.98	94.3 (208.04)	>max	99.7	98.5 (291.13)	>max	99.7	88.8 (212.51)	>max	99.7	202.8 (10275.14)	>max	99.9	83.6 (70.99)	>max	99.5	116.6 (4782.30)	>max	99.5
X5	200	11360.11	112.5 (314.93)	>max	100.0	124.4 (393.85)	>max	100.0	106.5 (320.98)	>max	100.0	142.0 (4428.98)	>max	100.0	93.4 (1284.40)	>max	99.8	125.5 (5760.88)	>max	99.8

die Verwendung von dynamischen Nachbarschaftsreihenfolgen bessere Ergebnisse erwarten und sollte wohl im Echtbetrieb bevorzugt verwendet werden.

Nun zum Unterschied zwischen den Werten mit und denen ohne Umkehren. Vor allem bei kleineren bis mittleren Instanzen sind klare Vorteile durch das Zulassen des Umkehrens auszumachen. Auch hier sind interessanter Weise die Laufzeiten leicht über denen der Tests ohne Umkehren. Sobald die Instanzen eine Größe erreichen, bei der der Algorithmus abgebrochen wird, lässt sich nur schwer eine Aussage darüber machen, wie repräsentativ die Werte sind. Aber wie auch beim Vergleich zwischen statischer und dynamischer Nachbarschaftsreihenfolge sind keine klaren Vorteile für ein Verfahren auszumachen. Dennoch scheint man mit Umkehren eher bessere Ergebnisse erzielen zu können.

Weiters wurden die Instanzen hinsichtlich ihrer Lösbarkeit bei Verwendung von alternativen Lagerplätzen für bestimmte Artikel untersucht. Das heißt also, dass manche Artikel an mehreren Positionen im Lager verfügbar sind und damit zusätzliche Möglichkeiten zur Bildung von Touren vorliegen können. Wirklich klare Unterschiede sind hier nur schwer zu erkennen. Wenn überhaupt, so sind am ehesten in Kombination mit dem Erlauben des Umkehrens leichte Vorteile auszumachen. Das liegt aber sicherlich auch daran, dass nicht alle Artikel an mehreren Positionen verfügbar sind. Somit kann es auch passieren, dass für die angeforderten Artikel keine alternativen Lagerplätze existieren und somit dieselbe Problemstellung vorherrscht wie bei der Berechnung ohne alternative Lagerplätze.

Generell sind nach diesen Tests natürlich nur Tendenzen auszumachen, es ist aber zu beachten, dass durchaus die Werte für kleinere Instanzen ausschlaggebend sind, da, wie auch schon in der Problembeschreibung erwähnt, nicht alle angeforderten Artikel auf einmal bekannt sind. Es kann also angenommen werden, dass im Verlauf eines Arbeitstages immer wieder kleinere Instanzen verarbeitet werden. Unter diesem Gesichtspunkt bieten die zuvor gefundenen Unterschiede durchaus eine Entscheidungsgrundlage zur Wahl entsprechender Berechnungsparameter.

8.3 Effizienz einzelner Nachbarschaften

Neben den im vorigen Abschnitt erläuterten Ergebnissen, wurde auch das Verhalten der Nachbarschaften im Verlauf des Algorithmus ausgewertet. In Tab. 8.4 ist zu sehen wie hoch der Anteil an Verbesserungen gegenüber allen getesteten Schritten einer Nachbarschaft ist. Wurden also beispielsweise 100 Schritte innerhalb einer Nachbarschaft getestet und führten 80 davon zu einer Verbesserung, so würde in der Tabelle ein Wert von 80% vermerkt werden. Die ersten vier Nachbarschaften stellen grundlegende Operationen zur Lösungsverbesserung dar, weshalb sie auch zu Beginn gereiht werden. *SplitMove* ist an erster Stelle, um aus anfänglich meist großen Touren zunächst mehrere kleinere zu erzeugen. Die Laufzeiten der folgenden Nachbarschaf-

ten *ShiftMove* und *SwapMove* sind abhängig von der Anzahl der Touren insgesamt, weshalb vor diesen beiden noch *Merge* durchsucht wird. Das ermöglicht das Zusammenführen mancher Touren, falls dies eine Verbesserung bringt. Erst im Anschluss an diese vier Nachbarschaften folgen jene, deren Komplexität größer ist, wobei die Reihenfolge hier auf Basis von kleineren Vortests so gewählt wurde. Es folgen also noch *SwapPosition*, *SwapPositionMerge*, *SplitPositionMerge* und *DoubleShift*. Selbstverständlich wirkt sich diese Reihenfolge nur direkt bei Verwendung von statischer Nachbarschaftsreihenfolge aus. Das Durchsuchen der Nachbarschaften erfolgte außerdem unter Verwendung der Strategie *Next Improvement* (siehe dazu auch Kapitel 5).

Sofort fallen viele Einträge mit „0.0%“ auf. Vor allem Nachbarschaft N_7 ist hier sehr dominant, was aber durchaus zu erklären ist, da ein verteiltes Einsammeln eines Artikels von mehreren Positionen bedeutet, eine Position mehr im Lager anfahren zu müssen. Es ist zwar vorstellbar, dass in manchen Szenarien solch eine Aufteilung von Nutzen ist, die Tests haben allerdings gezeigt, dass damit nicht sehr viel verbessert werden kann. Das Verhalten im Echtbetrieb kann leider nur schwer eingeschätzt werden, denn hierzu müsste man auf Basis von echten Daten entsprechende Auswertungen erstellen. Weiters sind die Nachbarschaften N_5 und N_6 nur dann sinnvoll, wenn auch alternative Positionen von Artikeln im Lager verwendet werden, deshalb wurden die entsprechenden Einträge mit „-“ markiert. Zusätzlich dazu beeinflusst die dynamische Reihung der Nachbarschaften deren Effizienz entscheidend, was sehr deutlich bei N_8 zu sehen ist. Da bei statischer Reihenfolge der Nachbarschaften schon N_3 viele *ShiftMoves* durchführt, so bleiben für N_8 natürlich kaum mehr Möglichkeiten zur Verbesserung. Anders bei dynamischer Reihenfolge, wo die Nachbarschaften einigermaßen gleichmäßig verteilt zum Einsatz kommen. Eine weitere Beobachtung kann hinsichtlich N_2 gemacht werden. Der Prozentsatz ist hier vor allem bei größeren Instanzen sehr klein, was darauf zurückzuführen ist, dass *Merge* nur dann sinnvoll verwendet werden kann, wenn Kommissionierungswagen weit unter ihrer Kapazität ausgelastet sind, was lediglich zu Beginn des Algorithmus sehr wichtig ist, mit zunehmender Laufzeit aber kaum mehr vorkommen dürfte. Besonders effizient hingegen verhält sich N_1, wo ein sehr hoher Prozentsatz an erzeugten *SplitMoves* auch tatsächlich zu einer Verbesserung der Gesamtlösung führt. Dabei steht vor allem das Beseitigen von überfüllten Kommissionierungswagen und damit ungültigen Touren im Vordergrund.

8.4 Rechenzeit der Nachbarschaften

Repräsentativ für das Laufzeitverhalten der einzelnen Nachbarschaften soll Tab. 8.5 (Testlauf 1) zeigen, wie hoch der Rechenaufwand der einzelnen Nachbarschaften ausfällt.

Vor allem N_8 schlägt hier mit besonders hohem Aufwand zu Buche, was durch die einigermaßen hohe Effizienz bei Verwendung dynamisch veränderbarer Nachbarschafts-

Tabelle 8.4: Effizienz der Nachbarschaften in Testlauf 1 *ohne* bzw. *mit* Artikeln auf alternativen Lagerplätzen - Durchschnittswerte über 20 Testläufe mit 25, 50, 100 und 200 einzusammelnden Artikeln (Ar). Angeführt ist pro Nachbarschaft der prozentuelle Anteil (alle Werte in Prozent [%]) an erfolgreichen Schritten in Relation zu den insgesamt getesteten Schritten der jeweiligen Nachbarschaft. Getestet wurden die Instanzen (In) *ohne* bzw. *mit* Artikeln auf alternativen Lagerplätzen bei Berechnung *mit* und *ohne* Umkehren innerhalb eines Ganges und bei *statischer* und *dynamischer* Nachbarschaftsreihenfolge. (N_1: *Split*; N_2: *Merge*; N_3: *Shift*; N_4: *Swap*; N_5: *SwapPosition*; N_6: *SwapPositionMerge*; N_7: *SplitPositionMerge*; N_8: *DoubleShift*.)

			Testlauf 1																											
			statische Reihenfolge																dynamische Reihenfolge											
			ohne Umkehren								mit Umkehren								ohne Umkehren								mit Umkehren			
	In	Ar	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8
ohne alternative Lagerplätze	S1	25	46.9	49.2	49.4	45.6	-	-	-	1.0	40.4	41.9	55.7	49.5	-	-	-	3.1	54.3	39.1	71.9	76.9	-	-	-	47.1	48.5	32.3	76.1	78.7	-	-	-	50.1
	S2	25	44.0	46.1	60.8	37.2	-	-	-	3.3	33.2	32.6	71.7	42.5	-	-	-	0.6	52.9	32.1	77.9	72.0	-	-	-	45.8	42.8	29.0	79.9	74.1	-	-	-	48.6
	S3	25	100.0	6.9	77.3	67.6	-	-	-	0.0	100.0	7.8	75.9	65.6	-	-	-	0.0	100.0	8.7	76.2	88.7	-	-	-	34.0	100.0	6.5	80.4	86.8	-	-	-	39.3
	S4	25	100.0	15.6	71.6	58.2	-	-	-	4.0	93.4	16.0	77.0	53.5	-	-	-	0.0	100.0	10.1	74.7	82.9	-	-	-	36.7	94.3	17.8	75.8	82.1	-	-	-	51.1
	S5	25	67.5	38.6	44.3	55.9	-	-	-	0.0	48.9	40.0	62.4	41.6	-	-	-	0.0	71.5	33.6	66.0	80.5	-	-	-	40.0	55.8	31.6	75.2	77.0	-	-	-	42.8
	M1	50	100.0	2.3	47.2	96.0	-	-	-	0.0	100.0	1.5	43.6	97.1	-	-	-	0.0	100.0	2.7	69.6	97.1	-	-	-	14.8	100.0	0.3	65.9	97.4	-	-	-	28.9
	M2	50	100.0	2.4	54.5	95.6	-	-	-	0.0	100.0	1.4	57.4	96.3	-	-	-	0.0	100.0	3.9	72.7	97.1	-	-	-	20.2	100.0	1.5	68.6	97.8	-	-	-	34.0
	M3	50	100.0	2.2	62.3	94.7	-	-	-	0.0	100.0	1.3	63.5	95.7	-	-	-	0.0	100.0	3.7	74.5	96.9	-	-	-	31.3	100.0	0.7	75.5	97.4	-	-	-	40.6
	M4	50	100.0	2.1	57.1	94.5	-	-	-	0.0	100.0	1.3	51.7	96.1	-	-	-	0.0	100.0	1.5	75.2	96.6	-	-	-	23.3	100.0	1.6	71.3	97.4	-	-	-	34.6
	M5	50	100.0	2.1	57.5	95.0	-	-	-	0.0	100.0	1.7	60.5	96.1	-	-	-	0.0	100.0	4.3	73.5	97.1	-	-	-	22.4	100.0	0.9	69.8	97.6	-	-	-	35.6
	L1	100	100.0	2.2	48.7	96.2	-	-	-	0.0	100.0	1.7	51.6	99.5	-	-	-	0.0	100.0	3.8	75.4	96.0	-	-	-	15.4	100.0	2.1	73.2	98.5	-	-	-	55.4
	L2	100	100.0	2.9	47.7	97.3	-	-	-	0.0	100.0	1.1	57.8	99.5	-	-	-	0.0	100.0	3.7	79.2	97.1	-	-	-	32.7	100.0	2.7	78.0	98.5	-	-	-	55.9
	L3	100	100.0	2.6	54.6	96.3	-	-	-	0.0	100.0	1.4	59.2	99.3	-	-	-	0.0	100.0	3.1	78.8	96.5	-	-	-	30.7	100.0	0.8	86.7	98.2	-	-	-	58.4
	L4	100	100.0	2.7	50.8	96.0	-	-	-	0.0	100.0	1.4	51.2	99.8	-	-	-	0.0	100.0	2.3	77.8	97.0	-	-	-	19.8	100.0	1.4	73.8	99.0	-	-	-	62.2
	L5	100	100.0	3.1	53.1	96.5	-	-	-	0.0	100.0	1.3	57.8	99.2	-	-	-	0.0	100.0	1.4	71.9	97.0	-	-	-	34.0	100.0	2.0	75.3	98.8	-	-	-	63.0
	X1	200	100.0	1.1	74.9	100.0	-	-	-	0.0	100.0	0.7	86.1	100.0	-	-	-	0.0	100.0	5.7	81.4	98.6	-	-	-	69.2	100.0	3.3	95.2	100.0	-	-	-	94.9
	X2	200	100.0	0.9	78.4	99.4	-	-	-	0.0	100.0	0.5	86.2	100.0	-	-	-	0.0	100.0	1.8	84.5	98.7	-	-	-	79.8	100.0	3.7	94.9	100.0	-	-	-	91.5
	X3	200	100.0	0.9	74.5	99.6	-	-	-	0.0	100.0	0.5	87.8	100.0	-	-	-	0.0	100.0	2.9	74.1	98.7	-	-	-	77.5	100.0	6.0	95.7	100.0	-	-	-	97.4
	X4	200	100.0	1.1	76.2	99.4	-	-	-	0.0	100.0	0.4	85.5	100.0	-	-	-	0.0	100.0	3.9	84.1	98.3	-	-	-	69.9	100.0	2.5	90.6	100.0	-	-	-	90.7
	X5	200	100.0	1.1	73.0	99.9	-	-	-	0.0	100.0	0.5	85.7	100.0	-	-	-	0.0	100.0	6.1	74.6	99.0	-	-	-	81.2	100.0	1.9	92.7	100.0	-	-	-	91.4
mit alternativen Lagerplätzen	S1	25	50.3	53.8	62.9	34.9	0.0	2.8	0.0	3.9	38.6	43.3	67.8	46.5	0.0	9.1	0.0	4.7	46.2	44.1	67.6	75.2	14.7	12.8	0.0	62.8	40.6	33.5	71.7	74.8	12.9	16.7	0.0	58.5
	S2	25	44.8	47.6	68.5	36.0	0.7	12.3	0.0	6.5	33.2	40.0	73.6	38.4	0.0	10.4	0.0	1.7	46.5	41.5	74.1	68.4	15.1	20.3	0.0	64.8	41.3	36.1	79.9	69.5	10.9	16.6	0.0	62.3
	S3	25	100.0	8.1	77.9	64.0	0.0	0.0	0.0	0.0	100.0	10.5	74.4	63.8	0.0	0.0	0.0	0.8	100.0	7.2	76.9	89.2	0.0	0.0	0.0	35.5	100.0	6.4	75.0	87.4	0.0	0.0	0.0	37.3
	S4	25	100.0	14.9	71.1	58.1	0.0	0.0	0.0	4.6	93.3	15.9	74.4	55.1	0.0	0.0	0.0	0.7	100.0	13.4	73.6	83.7	0.0	0.0	0.0	31.5	97.8	14.7	76.9	82.1	0.0	0.0	0.0	47.6
	S5	25	85.1	57.4	66.4	79.1	0.0	0.0	0.0	62.7	66.4	53.4	84.0	83.9	0.0	0.0	0.0	73.3	69.1	33.6	67.8	81.7	39.1	10.9	0.0	75.4	52.2	38.8	80.1	77.2	25.2	19.1	0.0	68.3
	M1	50	100.0	2.2	44.5	96.5	0.0	0.0	0.0	0.0	100.0	1.5	46.0	96.8	0.0	0.0	0.0	0.0	98.0	2.5	66.3	97.0	62.8	49.9	0.0	46.8	100.0	0.1	69.5	97.3	66.3	47.3	0.0	43.6
	M2	50	100.0	2.0	53.7	95.3	0.0	0.0	0.0	0.0	100.0	1.4	55.6	96.5	0.0	0.0	0.0	0.0	100.0	3.3	70.0	97.3	0.0	0.0	0.0	17.4	100.0	1.1	71.8	97.6	0.0	0.0	0.0	32.8
	M3	50	100.0	2.5	65.7	94.5	0.0	0.0	0.0	0.0	100.0	1.7	64.4	95.8	0.0	0.0	0.0	0.0	100.0	1.7	67.3	96.8	55.3	47.7	0.0	61.3	100.0	0.8	72.4	96.8	58.6	57.9	0.0	56.2
	M4	50	100.0	2.1	50.2	94.8	0.0	0.0	0.0	0.0	100.0	1.6	51.5	95.8	0.0	0.0	0.0	0.0	99.0	0.8	65.5	97.1	52.8	58.6	0.0	48.2	100.0	0.3	64.5	97.6	55.0	51.1	0.0	47.8
	M5	50	100.0	1.9	57.3	95.3	0.0	0.0	0.0	0.0	100.0	1.6	58.9	96.2	0.0	0.0	0.0	0.0	100.0	0.4	61.4	96.9	37.8	13.9	0.0	63.5	100.0	0.2	68.2	97.2	34.2	11.6	0.0	50.8
	L1	100	100.0	2.8	47.8	96.0	0.0	0.0	0.0	0.0	100.0	1.6	52.9	99.0	0.0	0.0	0.0	0.0	100.0	3.8	56.8	97.2	76.7	71.4	0.0	67.6	100.0	0.9	81.9	98.3	62.9	82.6	0.0	61.9
	L2	100	100.0	2.6	53.6	97.0	0.0	0.0	0.0	0.0	100.0	1.7	56.2	99.6	0.0	0.0	0.0	0.0	100.0	1.3	72.1	97.3	54.6	70.4	0.0	63.0	100.0	1.2	83.9	98.6	58.1	82.6	0.0	59.0
	L3	100	100.0	3.0	52.8	96.5	0.0	0.0	0.0	0.0	100.0	1.4	59.2	99.1	0.0	0.0	0.0	0.0	99.6	2.9	60.7	97.1	70.8	63.8	0.0	70.4	99.6	1.0	87.5	98.1	74.0	72.3	0.0	66.8
	L4	100	100.0	2.1	50.6	96.8	0.0	0.0	0.0	0.0	100.0	1.4	53.9	99.7	0.0	0.0	0.0	0.0	100.0	0.9	60.9	97.2	53.7	42.0	0.0	64.4	98.8	2.0	71.7	98.7	62.1	33.7	0.0	75.2
	L5	100	100.0	2.2	54.4	96.4	0.0	0.0	0.0	0.0	100.0	1.6	54.4	99.2	0.0	0.0	0.0	0.0	100.0	1.2	67.1	96.9	49.4	52.9	0.0	58.4	100.0	0.1	76.2	98.8	44.5	55.6	0.0	59.1
	X1	200	100.0	1.3	75.4	99.1	0.0	0.0	0.0	0.0	100.0	1.1	84.1	100.0	0.0	0.0	0.0	0.0	100.0	8.8	75.5	98.5	69.3	87.0	0.0	82.8	100.0	1.7	91.1	100.0	65.7	93.0	2.6	90.8
	X2	200	100.0	0.8	76.4	98.9	0.0	0.0	0.0	0.0	100.0	0.7	86.1	100.0	0.0	0.0	0.0	0.0	100.0	8.2	83.3	98.6	72.1	85.5	0.0	86.4	100.0	2.9	96.1	100.0	65.9	91.7	0.0	88.8
	X3	200	100.0	0.8	70.8	99.8	0.0	0.0	0.0	0.0	100.0	0.4	85.7	100.0	0.0	0.0	0.0	0.0	100.0	7.4	66.1	98.6	64.8	84.7	0.0	82.4	100.0	4.3	89.8	100.0	65.2	91.5	0.0	88.7
	X4	200	100.0	1.1	76.8	99.0	0.0	0.0	0.0	0.0	100.0	0.5	85.5	100.0	0.0	0.0	0.0	0.0	100.0	2.4	60.7	98.5	58.1	49.9	0.0	89.5	100.0	1.9	86.6	100.0	61.2	53.3	0.0	92.0
	X5	200	100.0	1.3	74.0	99.8	0.0	0.0	0.0	0.0	100.0	0.5	86.6	100.0	0.0	0.0	0.0	0.0	99.0	18.9	68.5	98.9	88.0	99.1	0.0	92.1	100.0	24.2	83.4	100.0	80.7	99.6	0.0	95.6

Tabelle 8.5: Rechenzeit - Durchschnittswerte über 20 Testläufe mit 25, 50, 100 und 200 einzusammelnden Artikeln. Zu sehen ist der prozentuelle Anteil (alle Werte in Prozent [%]) an Rechenzeit der jeweiligen Nachbarschaft in Relation zur Rechenzeit für alle Nachbarschaften für Instanzen *ohne* bzw. *mit* Artikeln auf alternativen Lagerplätzen bei Berechnung *mit* Umkehren innerhalb eines Ganges und *dynamischer* Nachbarschaftsreihenfolge.
(N_1: *Split*; N_2: *Merge*; N_3: *Shift*; N_4: *Swap*; N_5: *SwapPosition*; N_6: *SwapPositionMerge*; N_7: *SplitPositionMerge*; N_8: *DoubleShift*; N_9: *ShiftSplit*; N_{10}: *SwapSplit*; N_{11}: *MultipleSwap*; N_{12}: *MultipleSplit*; N_{13}: *SplitMerge*.)

			Testlauf 1								Testlauf 2											
	In	Ar	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_1	N_2	N_3	N_4	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}
ohne alternative Lagerplätze	S1	25	1.1	3.0	9.2	11.2	0.2	0.1	0.1	75.1	0.4	1.0	3.5	3.1	0.1	0.1	22.9	0.1	0.1	4.2	18.3	46.3
	S2	25	1.2	2.7	7.8	11.6	0.2	0.1	0.1	76.4	0.4	0.9	3.3	3.7	0.1	0.0	23.1	0.1	0.1	5.1	18.5	44.7
	S3	25	0.6	5.0	9.3	7.7	0.2	0.1	0.1	77.1	0.3	1.2	3.2	2.2	0.1	0.0	17.3	0.1	0.1	2.7	8.5	64.2
	S4	25	0.7	5.5	11.4	9.0	0.2	0.1	0.1	73.0	0.2	1.4	3.8	2.5	0.1	0.0	17.6	0.1	0.1	2.8	8.1	63.3
	S5	25	0.9	3.6	9.4	10.0	0.2	0.1	0.1	75.9	0.3	1.1	3.5	3.4	0.1	0.0	22.2	0.1	0.1	3.5	11.0	54.7
	M1	50	0.4	3.9	16.1	9.6	0.1	0.0	0.0	69.8	0.2	0.6	2.2	1.7	0.0	0.0	7.8	0.1	0.0	1.3	2.8	83.2
	M2	50	0.4	4.1	16.1	8.8	0.1	0.0	0.0	70.5	0.2	0.6	2.1	2.1	0.0	0.0	8.0	0.1	0.0	1.2	2.6	83.0
	M3	50	0.4	4.3	16.0	10.0	0.2	0.1	0.1	69.0	0.2	0.7	2.6	2.4	0.0	0.0	7.3	0.1	0.1	1.7	3.4	81.4
	M4	50	0.5	4.2	16.1	9.7	0.1	0.1	0.1	69.3	0.2	0.7	2.4	2.0	0.0	0.0	8.4	0.1	0.0	1.3	3.7	81.2
	M5	50	0.5	4.1	16.7	9.3	0.2	0.1	0.0	69.1	0.2	0.7	2.1	2.1	0.0	0.0	8.1	0.1	0.1	1.4	2.3	83.0
	L1	100	0.6	6.3	24.6	20.7	0.1	0.0	0.0	47.7	0.1	0.4	1.4	3.5	0.0	0.0	3.0	0.0	0.0	0.4	3.9	87.2
	L2	100	0.5	3.9	22.3	23.7	0.1	0.1	0.1	49.3	0.1	0.4	0.7	2.3	0.0	0.0	3.0	0.0	0.0	0.4	2.8	90.3
	L3	100	0.7	4.2	17.0	27.0	0.1	0.1	0.0	50.8	0.1	0.3	0.8	2.7	0.0	0.0	2.4	0.0	0.0	0.3	2.6	90.7
	L4	100	0.5	4.0	25.3	20.4	0.1	0.0	0.0	49.6	0.1	0.3	0.8	2.6	0.0	0.0	2.4	0.0	0.0	0.3	3.1	90.3
	L5	100	0.7	8.0	24.8	22.5	0.1	0.1	0.1	43.8	0.1	0.4	0.8	3.1	0.0	0.0	2.6	0.0	0.0	0.3	3.9	88.8
	X1	200	2.1	6.8	27.2	39.1	0.1	0.1	0.1	24.4	0.1	0.4	0.5	1.6	0.0	0.0	0.8	0.0	0.0	0.1	1.4	95.0
	X2	200	1.9	7.2	25.5	35.1	0.1	0.1	0.1	30.0	0.1	0.4	0.8	5.5	0.1	0.1	1.5	0.0	0.0	0.2	2.3	88.8
	X3	200	1.9	7.6	30.7	41.7	0.1	0.1	0.1	17.8	0.1	0.3	0.6	2.5	0.0	0.0	1.0	0.0	0.0	0.1	2.3	92.9
	X4	200	1.4	6.2	36.5	28.1	0.1	0.1	0.1	27.5	0.2	0.4	0.7	5.2	0.0	0.0	1.3	0.0	0.0	0.2	1.9	90.0
	X5	200	1.8	6.8	31.8	28.9	0.1	0.1	0.1	30.4	0.1	0.4	0.8	5.0	0.0	0.0	1.8	0.0	0.0	0.2	3.5	88.1
mit alternativen Lagerplätzen	S1	25	1.5	3.7	12.7	14.0	0.3	1.3	1.1	65.5	0.4	0.9	3.8	3.5	0.3	0.3	19.2	0.1	0.1	4.0	18.3	49.0
	S2	25	1.6	3.5	10.6	16.1	0.4	1.4	1.1	65.3	0.4	0.9	3.7	3.3	0.4	0.3	18.0	0.1	0.1	4.9	16.7	51.1
	S3	25	0.6	5.1	10.1	7.2	0.3	0.1	0.1	76.5	0.2	1.3	3.3	2.1	0.1	0.0	17.5	0.1	0.1	2.8	9.0	63.3
	S4	25	0.7	5.5	11.0	8.8	0.3	0.1	0.1	73.4	0.2	1.3	3.8	2.3	0.1	0.0	17.4	0.1	0.1	2.7	8.2	63.8
	S5	25	1.5	5.7	15.0	16.1	0.5	3.0	3.0	55.1	0.3	1.0	3.7	3.0	0.5	0.5	14.5	0.1	0.1	3.4	9.1	63.7
	M1	50	0.4	4.4	17.4	10.9	0.4	2.0	2.9	61.6	0.2	0.6	2.7	2.0	0.2	0.4	6.7	0.1	0.1	1.0	3.7	82.5
	M2	50	0.4	4.0	15.3	9.1	0.1	0.0	0.0	71.0	0.2	0.6	1.9	1.8	0.0	0.0	7.5	0.0	0.0	1.1	3.2	83.7
	M3	50	0.9	5.0	20.3	12.5	0.4	1.5	2.3	57.1	0.2	0.7	2.9	2.2	0.2	0.3	6.6	0.1	0.1	1.3	3.5	82.1
	M4	50	0.7	4.5	20.6	10.6	0.3	1.7	2.1	59.5	0.2	0.7	2.7	2.0	0.2	0.3	6.7	0.1	0.1	1.1	4.9	81.2
	M5	50	0.7	4.9	20.6	12.3	0.3	0.6	0.7	60.0	0.2	0.7	2.7	1.9	0.1	0.1	6.4	0.1	0.0	1.1	2.0	84.6
	L1	100	0.6	4.3	21.1	26.1	0.4	1.3	3.2	42.9	0.1	0.3	1.4	2.7	0.1	0.2	1.4	0.0	0.0	0.3	4.2	89.2
	L2	100	0.6	4.4	18.9	24.8	0.3	1.0	2.0	47.9	0.1	0.4	1.6	3.0	0.1	0.2	1.8	0.0	0.0	0.3	4.2	88.2
	L3	100	0.5	8.6	16.0	31.0	0.5	2.9	4.1	36.5	0.1	0.4	1.3	5.0	0.1	0.3	2.6	0.0	0.0	0.3	3.6	86.2
	L4	100	0.7	6.0	29.7	27.7	0.3	2.5	2.9	30.2	0.1	0.4	1.5	3.9	0.1	0.2	1.7	0.0	0.0	0.3	4.2	87.4
	L5	100	0.9	4.0	24.0	24.9	0.2	1.0	1.4	43.6	0.1	0.3	1.6	3.1	0.1	0.1	1.7	0.0	0.0	0.3	3.5	89.1
	X1	200	0.9	5.7	33.0	36.2	0.6	2.1	6.2	15.3	0.1	0.7	1.2	6.4	0.1	0.5	1.3	0.0	0.0	0.4	3.6	85.6
	X2	200	1.1	7.0	25.0	33.3	0.7	3.6	6.9	22.4	0.2	0.2	0.8	3.6	0.2	0.4	0.7	0.0	0.0	0.3	2.3	91.3
	X3	200	1.1	5.4	38.7	25.0	0.6	2.5	6.3	20.5	0.1	0.2	0.6	3.2	0.1	0.3	0.6	0.0	0.0	0.3	2.4	92.1
	X4	200	1.0	5.3	40.2	32.2	0.5	2.4	3.3	15.0	0.2	0.4	1.3	6.1	0.1	0.2	0.7	0.0	0.0	0.2	3.4	87.5
	X5	200	0.9	4.8	49.3	21.7	0.6	2.0	12.1	8.6	0.1	0.3	0.8	0.9	0.1	0.6	0.3	0.0	0.0	0.3	2.4	94.2

strukturen durchaus vertretbar ist. Bei statischer Nachbarschaftsreihenfolge hingegen macht die Verwendung dieser Nachbarschaft keinen Sinn und sollte ausgelassen werden. Wie intuitiv außerdem zu erwarten war, benötigen N_3 und N_4 deutlich mehr Aufwand zum Durchsuchen, als die übrigen Nachbarschaften. Sind es bei kleinen Instanzen noch je etwa 10% der Rechenzeiten, so steigt der Anteil für große Instanzen auf 30–40% an.

8.5 Laufzeit und Lösungsverbesserung

Ein wesentlicher Aspekt des entwickelten Algorithmus ist das Auffinden von guten Lösungen innerhalb möglichst kurzer Zeit, da im Echtbetrieb nicht uneingeschränkt viel Zeit zur Verfügung stehen kann. Aus diesem Grund ist eine Visualisierung des Verlaufs der Lösungsverbesserung durchaus interessant, was anhand von Instanz M3 bei Berechnung mit alternativen Lagerplätzen und dynamischer Nachbarschaftsreihenfolge in Abb. 8.1 dargestellt wird. Es ist dabei eindeutig zu erkennen, dass innerhalb der ersten 100 Iterationen des VND bereits eine sehr gute Lösung erreicht wird, welche im weiteren Verlauf zwar noch geringfügig verbessert wird, aber nicht mehr in so hohem Maß wie zu Beginn. Es sollte darauf hingewiesen werden, dass die Verbesserungen innerhalb der zweiten 100 Iterationen bei weitem nicht so gravierend sind, da hier die veränderte Skalierung der y-Achse für die Tourlänge zwischen 0 und 50000 beachtet werden muss.

Dieses Verhalten ist äußerst günstig für die Verwendung im realen Umfeld, wo vor allem schnelle Antwortzeiten von Bedeutung sind. In manchen Fällen wird es dort nötig sein, schon nach kurzer Zeit den Algorithmus abzubrechen und die bis dahin vorliegende Lösung zu verwenden.

8.6 Lösungsqualität bei Verwendung zusätzlicher Nachbarschaften

Um das Verhalten des Algorithmus bei der Verwendung zusätzlicher Nachbarschaften beobachten zu können, wurde ein weiterer Testlauf durchgeführt, der unter den gleichen Voraussetzungen wie der in Kapitel 8.2 beschriebene durchgeführt wurde. Hierbei wurde allerdings nur mehr mit dynamischer Nachbarschaftsreihenfolge gerechnet, es kamen aber weitere Nachbarschaften zum Einsatz. Verwendet wurden hier neben *Split*, *Merge*, *Shift*, *Swap*, *SwapPositionMerge*, *SplitPositionMerge* und *DoubleShift* auch *ShiftSplit*, *SwapSplit*, *MultipleSwap*, *MultipleSplit* und *SplitMerge*. Diese Auflistung entspricht wieder gleichzeitig der Anfangsreihenfolge der Nachbarschaften. Weiters wurde *SwapPosition* ausgelassen, da diese bereits durch *SwapPositionMerge* abgedeckt ist.

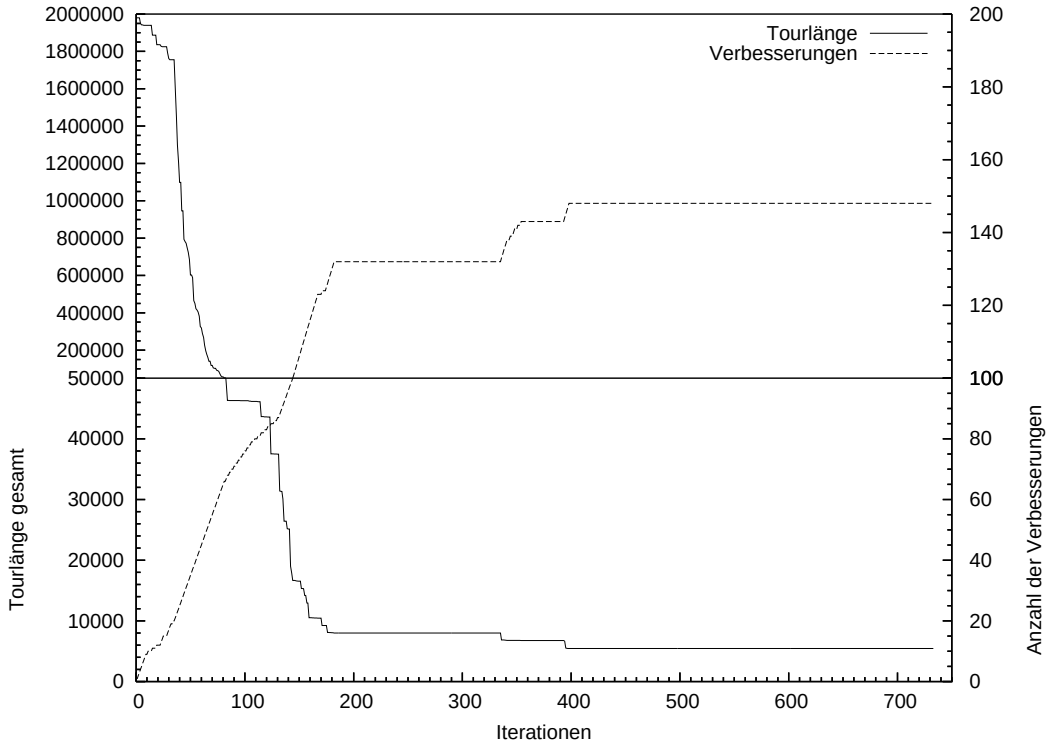


Abbildung 8.1: Verlauf der Zielfunktion im Vergleich zur Anzahl der gefundenen Verbesserungen (Achtung: Unterschiedliche Skalierung der y-Achse für $[0; 50000]$ und $(50000; 2000000]$)

In Tab. 8.2 und Tab. 8.3 sind die in diesem Test (Testlauf 2) erzielten Ergebnisse und Laufzeiten des Algorithmus zu sehen. Diese Tabellen enthalten auch die Ergebnisse aus Testlauf 1, was die Gegenüberstellung erleichtert. Es ist zu erkennen, dass von allen berechneten 80 Instanzen nur 11 nicht besser gelöst werden konnten und die Summe aller Zielfunktionswerte bei Instanzen ohne Umkehren innerhalb von Gängen um rund 20% und bei Instanzen mit Umkehren sogar um etwa 22% verbessert werden konnte. Gleichzeitig muss aber auch gesagt werden, dass die Berechnungen zum Teil mehr als die doppelte Laufzeit bei kleineren Instanzen benötigten. Dennoch konnten auch die Zielfunktionswerte großer Instanzen, deren Bearbeitung nach dem Grenzwert von 1200 Sekunden abgebrochen wurde, in derselben Zeit wie im vorigen Testlauf deutlich verbessert werden.

Die Auswertung der Effizienz der Nachbarschaften in Tab. 8.6 zeigt ein ähnliches Bild, wie in Tab. 8.4. Zur besseren Übersicht wurden nur die Werte für Berechnungen mit Umkehren innerhalb eines Ganges herangezogen. Zusätzlich ist hier ersichtlich, dass die neu hinzugekommenen Nachbarschaften N_9-N_{13} eine durchwegs hohe Verbesserungsquote aufweisen.

Die Rechenzeiten aus Testlauf 2, wie in Tab. 8.5 zu sehen ist, weisen Parallelen zu den in Testlauf 1 erhaltenen Werten auf. Im Wesentlichen bestätigt das Ergebnis die bereits zuvor ermittelten Zahlen, wobei es zu leichten Verschiebungen gekommen ist, da die neu hinzugefügten Strukturen mit Kombinationen aus zuvor einzeln angewandten arbeiten. Hier sind die Berechnungen für kombinierte und damit komplexere Strukturen natürlich auch aufwändiger, aufgrund des Erfolgs ist deren Anwendung aber wünschenswert. Besonders auffällig ist der hohe Rechenaufwand für *SplitMergeCombine*, wo im Vergleich dazu im vorangegangenen Testlauf *DoubleShift* den größten Anteil an Rechenzeit benötigte.

Es hat sich also gezeigt, dass die neu eingebundenen Nachbarschaftsstrukturen ein durchwegs positives Resultat bringen und die Lösung tatsächlich verbessern konnten. Teilweise konnten die Zielfunktionswerte sogar um 50% verringert werden und nur in wenigen Fällen wurden geringfügig schlechtere Ergebnisse erzielt, als beim vorherigen Testlauf.

Abschließend ist zu den Testergebnissen zu sagen, dass durchwegs positive Resultate erzielt werden konnten. Auch im Vergleich zur Lösung mittels *S-Shape*-Heuristik konnten mehrheitlich bessere Werte berechnet werden. Zusätzlich dazu sollte noch einmal darauf hingewiesen werden, dass unter der Verwendung der *S-Shape*-Heuristik keine Nebenbedingungen, wie etwa Kollisionen zwischen Arbeitern, geprüft werden konnten. Es wurde damit lediglich eine Referenzlösung erzeugt.

Es bleibt nun noch zu erwähnen, dass die hiermit geschilderten Beobachtungen darauf schließen lassen, dass weitere Verbesserungen etwa in Zusammenhang mit Anpassungen der Nachbarschaftsstrukturen durchaus möglich sind. In jedem Fall bietet sich ein Einsatz im Echtbetrieb an, um die Leistungsfähigkeit des entwickelten Ansatzes unter realen Bedingungen zu ermitteln.

Tabelle 8.6: Effizienz der Nachbarschaften in Testlauf 2 *ohne* bzw. *mit* Artikeln auf alternativen Lagerplätzen - Durchschnittswerte über 20 Testläufe mit 25, 50, 100 und 200 einzusammelnden Artikeln (Ar). Angeführt ist pro Nachbarschaft der prozentuelle Anteil (alle Werte in Prozent [%]) an erfolgreichen Schritten in Relation zu den insgesamt getesteten Schritten der jeweiligen Nachbarschaft. Getestet wurden die Instanzen (In) *ohne* bzw. *mit* Artikeln auf alternativen Lagerplätzen bei Berechnung *mit* Umkehren innerhalb eines Ganges und bei *dynamischer* Nachbarschaftsreihenfolge. (N_1: *Split*; N_2: *Merge*; N_3: *Shift*; N_4: *Swap*; N_5: *SwapPosition*; N_6: *SwapPositionMerge*; N_7: *SplitPositionMerge*; N_8: *DoubleShift*; N_9: *ShiftSplit*; N_{10}: *SwapSplit*; N_{11}: *MultipleSwap*; N_{12}: *MultipleSplit*; N_{13}: *SplitMerge*.)

			Testlauf 2 / dynamische Reihenfolge / mit Umkehren											
	In	Ar	N_1	N_2	N_3	N_4	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}
ohne alternative Lagerplätze	S1	25	22.4	26.4	63.7	79.5	-	-	48.5	18.2	12.6	72.9	15.6	56.2
	S2	25	21.3	28.3	67.5	73.7	-	-	50.9	15.5	9.5	68.4	21.2	62.4
	S3	25	95.2	14.0	65.0	79.7	-	-	39.5	52.6	56.0	76.2	23.8	42.1
	S4	25	86.2	15.1	62.2	75.3	-	-	47.0	52.3	37.1	78.3	21.6	45.4
	S5	25	27.1	27.4	63.9	69.5	-	-	41.9	16.8	15.9	73.8	17.6	53.3
	M1	50	100.0	0.7	71.4	95.6	-	-	43.1	79.2	56.3	72.0	70.3	31.8
	M2	50	100.0	0.6	79.1	94.9	-	-	47.6	78.7	53.3	76.7	79.0	42.0
	M3	50	100.0	1.5	77.8	93.7	-	-	57.1	78.3	75.4	66.4	73.0	48.8
	M4	50	100.0	0.3	75.4	94.3	-	-	44.3	81.9	60.0	75.8	66.0	44.5
	M5	50	100.0	0.2	77.4	94.6	-	-	45.8	81.9	57.8	70.2	82.3	42.7
	L1	100	100.0	2.3	89.7	95.9	-	-	72.0	76.6	54.5	90.9	42.1	45.8
	L2	100	100.0	3.5	95.5	97.5	-	-	70.6	75.8	53.9	89.4	51.5	32.1
	L3	100	100.0	0.1	94.3	96.6	-	-	77.5	74.7	70.2	93.3	53.2	33.9
	L4	100	100.0	0.1	95.1	96.9	-	-	73.5	73.8	48.9	93.4	46.0	27.5
	L5	100	100.0	1.1	95.8	96.3	-	-	74.3	76.3	61.0	93.5	36.2	39.4
	X1	200	100.0	4.2	99.1	100.0	-	-	98.1	67.4	44.8	98.9	77.9	64.7
	X2	200	100.0	4.2	99.2	99.3	-	-	88.7	71.0	41.2	97.4	62.3	69.5
	X3	200	100.0	2.1	99.2	99.9	-	-	93.1	68.2	41.6	98.1	57.7	61.6
	X4	200	100.0	6.3	99.3	99.4	-	-	93.6	66.9	49.6	98.5	74.7	73.8
	X5	200	100.0	2.1	99.0	100.0	-	-	90.3	66.7	49.1	97.9	56.0	78.0
mit alternativen Lagerplätzen	S1	25	18.3	26.4	58.7	73.7	34.7	0.0	50.6	13.6	9.5	74.4	9.8	62.7
	S2	25	20.9	30.3	63.0	76.0	32.6	0.0	58.8	13.5	10.0	69.7	16.2	63.3
	S3	25	95.4	13.3	62.7	79.4	0.0	0.0	38.6	63.5	54.7	75.6	18.3	42.4
	S4	25	92.6	17.9	61.0	76.4	0.0	0.0	46.6	45.9	46.2	77.9	12.8	42.6
	S5	25	27.7	31.0	63.4	72.5	46.5	0.0	46.3	15.6	12.9	75.0	18.6	58.7
	M1	50	100.0	0.2	65.8	94.0	83.8	0.0	52.8	70.9	68.6	80.7	60.0	33.9
	M2	50	100.0	0.4	80.5	94.3	0.0	0.0	44.1	78.6	66.7	76.5	70.2	33.7
	M3	50	100.0	1.8	72.4	93.4	82.1	0.0	63.1	64.7	69.3	77.5	70.4	47.4
	M4	50	100.0	1.0	70.0	94.9	81.4	0.0	55.8	71.2	69.5	78.5	49.4	39.8
	M5	50	100.0	0.4	71.9	95.2	49.6	0.0	61.0	70.0	74.5	80.7	86.7	39.5
	L1	100	100.0	0.2	89.5	97.7	91.2	0.0	90.6	73.4	58.1	94.4	39.6	38.3
	L2	100	100.0	0.8	87.8	96.8	90.3	0.0	85.3	69.5	52.6	91.7	32.1	36.5
	L3	100	100.0	0.0	92.1	93.4	94.0	0.0	72.8	76.3	63.8	95.8	50.0	47.7
	L4	100	100.0	0.0	90.7	96.3	86.2	0.0	89.8	68.2	57.3	95.5	40.8	42.4
	L5	100	100.0	0.0	88.4	96.5	77.2	0.0	87.1	67.9	50.1	95.0	43.3	33.7
	X1	200	100.0	10.0	98.0	100.0	96.9	0.0	88.6	63.1	41.2	96.0	56.6	82.3
	X2	200	97.5	2.7	99.1	99.4	94.3	0.0	94.8	62.6	42.1	91.2	56.9	66.1
	X3	200	100.0	7.4	99.3	100.0	96.6	0.0	95.3	66.9	32.3	97.9	60.8	70.1
	X4	200	100.0	0.0	98.1	100.0	78.2	0.0	97.3	63.6	43.6	98.3	47.7	73.2
	X5	200	100.0	32.7	97.9	100.0	99.4	0.0	99.0	21.9	21.1	94.4	56.7	63.4

9 Fazit

Inhalt dieser Arbeit war die Vorstellung eines hybriden Verfahrens, welches zum Lösen von Problemen der Tourenplanung im Echtbetrieb eines Ersatzteillagers eingesetzt werden kann. Das Grundgerüst bildet dabei eine *Variable Nachbarschaftssuche* mit integriertem *Variable Neighborhood Descent* (VND) als lokale Verbesserungsstrategie. Zum Lösen von Teilproblemen innerhalb dieses Algorithmus wird ein eigens entwickeltes Dynamisches Programm verwendet, mit dessen Hilfe es möglich ist, konkrete Touren optimal in polynomieller Zeit abhängig von der Anzahl der bestellten Artikel zu berechnen. Um abschließend eine Zuweisung von Lagerarbeitern zu den im vorhergehenden Schritt berechneten Touren zu berechnen, wird eine zweite VNS verwendet.

Aus den präsentierten Ergebnissen der Tests lässt sich ableiten, dass der gefundene Ansatz prinzipiell funktionstüchtig ist und schnell akzeptable Lösungen erzeugt. Sind die Arbeiter des Lagers erst einmal damit beschäftigt den ersten ihnen zugewiesenen Artikel auszufassen, kann die Gesamtlösung weiter optimiert werden. Dies ist möglich, da zu jedem Zeitpunkt eine gültige Lösung verfügbar ist. Von Bedeutung ist dies eventuell auch, wenn eine Erweiterung des Verfahrens zu einem Online-Algorithmus durchgeführt wird. Als solcher muss dieser auf laufend hinzukommende und im Vorhinein nicht bekannte Bestellungen reagieren können. Die Adaptierung von bereits ausgegebenen Touren würde in so einem Algorithmus einen wesentlichen Beitrag zur Flexibilität liefern. Dadurch, dass viele der zu liefernden Artikel erst im Laufe des Tages bestellt werden und zwischen den einzelnen Ausfassungsoperationen der Lagerarbeiter entsprechend viel Berechnungszeit zur Verfügung steht, wird die kontinuierliche Verbesserung einer anfangs schnell generierten Lösung möglich. Weiters scheint eine Aufteilung des Lagers in örtlich voneinander getrennte Bereiche, basierend auf der Lage der Verpackungszone, sinnvoll, wodurch die Größe der so entstehenden (Teil-)Instanzen entsprechend gering ist.

Obwohl die in dieser Arbeit präsentierten Testergebnisse implizieren, dass die Größe der tatsächlichen Instanzen mit bis zu 5000 bestellten Artikeln pro Tag zu komplex ist, kann man vermuten, dass nach weiterer Überarbeitung und Erweiterung der Nachbarschaftsstrukturen oder Anpassungen der Durchsuchungsstrategie die Ergebnisse weiter verbessert und selbst Instanzen von solcher Größe qualitativ hochwertig gelöst werden können.

Leider war es im Rahmen dieser Arbeit nicht möglich, die Auswirkungen von unvorhergesehenen Vorkommnissen (ein Artikel ist nicht in gewünschter Menge verfügbar,

Verspätung eines Arbeiters auf seiner Tour, etc.) aussagekräftig zu testen, da aufgrund des erst neu eingeführten Verwaltungssystems die Testdaten nicht dem Echtbetrieb entnommen werden konnten. Daher musste versucht werden, beim Generieren der Testdaten alle bekannten Aspekte und Eigenschaften der Lagerstruktur bestmöglichst zu berücksichtigen.

Anzumerken ist noch, dass die Methode zum Berechnen von optimalen Kommissionierungstouren teilweise auf der Annahme basiert, dass ein Mitarbeiter in einem Gang jederzeit umdrehen kann. Je nach Beschaffenheit des Lagers und der verwendeten Fahrzeuge kann es durchaus vorkommen, dass eine solche Richtungsänderung nicht, beziehungsweise nur schwer möglich ist. Sollte dies der Fall sein, kann das Dynamische Programm einfach angepasst werden. Die grundlegende Struktur des Ansatzes ändert sich dadurch nicht.

Im Rahmen der Tests hat sich gezeigt, dass die gefundenen Lösungen im Vergleich zu ebenfalls berechneten Referenzlösungen von guter Qualität sind und zusätzlich dazu alle Nebenbedingungen erfüllt werden konnten. Aber auch die Laufzeit des Verfahrens und vor allem die raschen Verbesserungen zu Beginn eines Durchlaufs vermitteln einen positiven Eindruck.

In Hinblick auf eine Beschleunigung des Algorithmus gäbe es die Möglichkeit auf die Berechnung exakter Touren zu verzichten. Natürlich würde man in diesem Fall zwar Geschwindigkeit gewinnen können, müßte aber bei der Lösungsqualität Einbußen hinnehmen. Auch ein nochmaliges Überarbeiten der Nachbarschaftsstrukturen könnte die eine oder andere Verbesserung bringen. Es sollte aber erwähnt werden, dass der präsentierte Ansatz bereits mit stabilem Verhalten und vielversprechender Leistung überzeugen kann.

Als Fazit lässt sich sagen, dass der Einsatz eines computerunterstützten Entscheidungssystems zur Planung von Kommissionierungstouren durchaus sinnvoll scheint, wenngleich aufgrund der Problemkomplexität auf eine besonders effiziente Implementierung geachtet werden muss. Besonders berücksichtigt werden muss dabei die Tatsache, dass dies ein System sein soll, dass darauf abzielt Menschen in gewisser Hinsicht Befehle zu erteilen, was einerseits soziale Schwierigkeiten mit sich bringt und andererseits aber auch bedeutet, dass Zustände erreicht werden können, die im Vorhinein nur schwer vorherzusehen sind. Das System sollte also nur als Entscheidungsunterstützung eingesetzt werden und kann die Kontrolle und Betreuung durch einen qualifizierten Lagerarbeiter nicht ersetzen.

Literaturverzeichnis

[1] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, January 2007.

[2] C. Archetti, M. G. Speranza, and A. Hertz. A tabu search algorithm for the split delivery vehicle routing problem. *Transportation Science*, 40(1):64–73, 2006.

[3] R. E. Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[4] R. de Koster, T. Le-Duc, and K. J. Roodbergen. Design and control of warehouse order picking: A literature review. *European Journal of Operational Research*, 182(2):481–501, 2007.

[5] R. de Koster and E. Van Der Poort. Routing orderpickers in a warehouse: a comparison between optimal and heuristic solutions. *IIE Transactions*, 30(5):469–480, 1998.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dezember 1959.

[7] M. Dror and P. Trudeau. Savings by split delivery routing. *Transportation Science*, 23:141–145, 1989.

[8] M. Dror and P. Trudeau. Split delivery routing. *Naval Research Logistics*, 37:383–402, 1990.

[9] C. Feremans, M. Labbe, and G. Laporte. Generalized network design problems. *European Journal of Operational Research*, 148(1):1–13, 2003.

[10] M. Fischetti, J. J. Salazar Gonzalez, and P. Toth. The symmetric generalized traveling salesman polytope. *Networks*, 26(2):113–123, 1995.

[11] P. Hansen, N. Mladenović, and L. C. D. Gerad. A tutorial on variable neighborhood search. Technical report, Les Cahiers du GERAD, HEC Montreal and GERAD, 2003.

[12] P. Hansen and N. Mladenović. Variable neighborhood search. In F. W. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 145–184. Kluwer Academic Publisher, New York, 2003.

[13] S. C. Ho and D. Haugland. A tabu search heuristic for the vehicle routing

problem with time windows and split deliveries. *Computers and Operations Research*, 31:1947–1964, 2004.

[14] B. Hu and G. R. Raidl. Variable neighborhood descent with self-adaptive neighborhood-ordering. In C. Cotta, A. J. Fernandez, and J. E. Gallardo, editors, *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics*, 2006.

[15] J. Lysgaard, A. N. Letchford, and R. W. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100(2):423–445, 2004.

[16] N. Mladenović. A variable neighborhood algorithm - a new metaheuristic for combinatorial optimization. *Abstracts of papers presented at Optimization Days*, page 112, 1995.

[17] T. Ralphs, L. Kopman, W. Pulleyblank, and L. T. Jr. On the capacitated vehicle routing problem. *Mathematical Programming Series*, 94(B):1–19, 2003.

[18] P. Toth and D. Vigo. *The Vehicle Routing Problem*. Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 2002.

[19] K. Q. Zhu, K. C. Tan, and L. H. Lee. Heuristics for vehicle routing problem with time. In *Windows, 6th AI and Math*, 2000.