Bachelorarbeit

Systematischer Vergleich heuristischer, approximativer und parametrisierter Algorithmen zur Routenoptimierung in der Lagerkommissionierung

Constantin Dietrich

21.08.2026

Vorgelegt an der Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Lehrstuhl für Algorithmen und Datenstrukturen

universität freiburg

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Lehrstuhl für Algorithmen und Datenstrukturen

Autor	Constantin Dietrich, Matrikelnummer: 5126278
Zeitraum	21.05.2026 – 21.08.2026
Prüfer	Prof. Dr. Hannah Bast, Technische Fakultät Lehrstuhl für Algorithmen und Datenstrukturen
Betreuer	Dr. Patrick Brosi, Technische Fakultät Lehrstuhl für Algorithmen und Datenstrukturen

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.

Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, 21.08.2026 — Ort, Datum

C. Dietrich — Unterschrift

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Lehrstuhl für Algorithmen und Datenstrukturen

Erklärung zur Nutzung von KI-Werkzeugen

Als Hilfsmittel habe ich den KI-Assistenten Claude (Anthropic) eingesetzt: für Literaturrecherche, Strukturvorschläge, Korrekturen, LaTeX-Hilfe sowie Feedback und Formulierungsvorschläge zu meinen Entwürfen. Die zugehörige Implementierung habe ich mit Hilfe des Claude Code CLI entwickelt, das dort für Implementierungsvorschläge, Refactoring, Testfälle und die Docker-Konfiguration genutzt wurde.
Alle übernommenen Formulierungen und Codeteile habe ich geprüft und überarbeitet. Konzeption, Auswahl und Verifikation der Algorithmen, Durchführung und Auswertung der Experimente sowie alle Schlussfolgerungen stammen von mir. Die Verantwortung für den gesamten Inhalt liegt bei mir.

Zusammenfassung

Das Optimieren von Routen in Lagersystemen ist ein zentrales Problem in der Lagerlogistik und wird auch als Order Picking Routing Problem (OPRP) bezeichnet. Die vorliegende Arbeit vergleicht drei Lösungsverfahren für das OPRP in konfigurierbaren rechteckigen Lagern. Untersucht werden die Nearest-Neighbor-Heuristik, der Christofides-Algorithmus und das parametrisierte Verfahren von Cambazard und Catusse (2018). Motiviert ist der Vergleich durch den Zielkonflikt zwischen Berechnungsaufwand und Routenlänge. In der gesichteten Literatur findet sich kein systematischer Vergleich dieser drei Paradigmen auf einem einzigen, dynamisch konfigurierbaren Lagermodell. Für die Berechnung der benötigten Kennzahlen wurde eine mit Docker containerisierte Webanwendung entwickelt. In 10.200 Läufen zeigt sich, dass der Berechnungsaufwand von unterschiedlichen Größen getrieben wird. Beim parametrisierten Verfahren bestimmt ihn die Anzahl der Regalreihen r, bei den beiden anderen Verfahren die Anzahl der Produkte. Dadurch bleiben diese beiden im Millisekundenbereich. Das exakte Verfahren bleibt bis $r = 7$ praktikabel. Dort erreicht die mittlere Berechnungszeit 59 Sekunden. Die wirtschaftliche Grenze liegt früher, nämlich zwischen $r = 6$ und $r = 7$. Ab dort übersteigt die Berechnungszeit die Gehzeit, die durch die kürzere Route eingespart wird. Die Lösungsqualität der Heuristiken wird durch die Anzahl der Regalreihen und die Auftragsgröße beeinflusst. Gegen die exakte Referenz liegt Nearest Neighbor je nach Layout 4,1 % bis 13,7 % über dem Optimum, Christofides 3,6 % bis 7,9 %. In mehrblockigen Layouts vergrößert eine steigende Auftragsgröße diesen Abstand zusätzlich. Im Maximum erreicht Nearest Neighbor 50,0 %, Christofides 24,2 % bei einer garantierten Güte von 3/2. Die Arbeit liefert diesen Dreiervergleich, ein wiederholbares Benchmark-Werkzeug und eine Entscheidungsmatrix, die auf r und dem verfügbaren Zeitbudget beruht.

Abstract

Optimising routes in warehouse systems is a central problem in warehouse logistics and is known as the Order Picking Routing Problem (OPRP). This work compares three solution approaches to the OPRP in configurable rectangular warehouses: the nearest neighbour heuristic, the Christofides algorithm, and the fixed-parameter tractable approach by Cambazard and Catusse (2018). No systematic comparison of these three paradigms on a single, dynamically configurable warehouse model appears in the reviewed literature.

To compute the required metrics, a Docker-containerised web application was developed. Across 10,200 runs, computation cost is driven by different factors: for the fixed-parameter implementation by the number of shelf rows r, for the other two by the number of products, which keeps them in the millisecond range.

The exact approach remains viable up to $r = 7$, where mean computation time reaches 59 seconds. The economic limit is reached earlier, between $r = 6$ and $r = 7$, where computation time exceeds the walking time saved on the shorter route. Solution quality of the heuristics is influenced by the number of shelf rows and the order size. Against the exact reference, nearest neighbour lies 4.1 % to 13.7 % above the optimum depending on layout, Christofides 3.6 % to 7.9 %. In multi-block layouts, a growing order size widens this margin further. At maximum, nearest neighbour reaches 50.0 %, Christofides 24.2 % with a guaranteed ratio of 3/2.

The work contributes this three-way comparison, a repeatable benchmarking tool, and a decision matrix based on r and the available time budget.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation & Problemkontext 1
1.2 Problemstellung 2
1.3 Zielsetzung & Forschungsfrage 3
1.4 Methodik 4
1.5 Aufbau der Arbeit 4

2 Grundlagen **7**

2.1 Grundlagen der Graphentheorie 7
2.1.1 Pfade, Zyklen und Kreise 7
2.1.2 Spannbaum und Minimum Spanning Tree 8
2.1.3 Matching 9
2.2 Lagerlayout und Kommissionierung 9
2.2.1 Graphenmodelle des Lagers 13
2.3 Komplexitätstheorie 14
2.3.1 Fixed-Parameter Tractability 15
2.3.2 Baumweite und Pfadweite 16
2.4 Das Travelling Salesperson Problem (TSP) 18
2.4.1 Symmetrisches und asymmetrisches TSP 19
2.4.2 Das Order Picking Routing Problem (OPRP) 19
2.5 Lösungsparadigmen für das TSP 20
2.6 Kürzeste-Wege-Algorithmen 21

3 Stand der Forschung **23**

3.1 Heuristiken für das TSP 23
3.2 Approximationsalgorithmen für das TSP 24
3.3 Parametrisierte Verfahren 25
3.4 Routenplanung in der Lagerkommissionierung 26

3.5 Forschungslücke 27
3.6 Abgrenzung der Arbeit 28

4 Methodik & Implementierung 29
4.1 Systemarchitektur 29
4.2 Lagermodellierung 32
4.3 Distanzberechnung 33
4.4 Nearest-Neighbor-Heuristik 34
4.5 Christofides-Algorithmus 35
4.5.1 Verfahren und Gütegarantie 35
4.5.2 Implementierung 36
4.5.3 Laufzeit und Grenzen der Implementierung 39
4.6 Fixed-Parameter-Algorithmus 40
4.6.1 Einordnung und Grundidee 40
4.6.2 Reduktion auf einen Tour-Teilgraphen 41
4.6.3 Der Frontier-Zustand 42
4.6.4 Sweep-Reihenfolge und Transitionen 46
4.6.5 Ablauf und Terminierung 50
4.6.6 Rekonstruktion der Route 52
4.6.7 Laufzeit und Grenzen der Implementierung 53
4.7 Benchmark-System 57
4.7.1 Validierung und Reproduzierbarkeit 59

5 Evaluation 63
5.1 Versuchsaufbau 63
5.2 Einfluss der Lagerbreite 65
5.3 Einfluss der Regalreihen 70
5.4 Skalierung bei quadratischen Lagerlayouts 78
5.5 Optimalitätsgap 83
5.6 Statistische Absicherung 84

6 Diskussion 89
6.1 Beantwortung der Forschungsfrage 89
6.2 Praktische Implikationen 92
6.3 Limitationen 95
6.4 Einordnung in die Literatur 98

7 Fazit & Ausblick **101**
7.1 Zusammenfassung 101
7.2 Ausblick 103

Literatur **107**

A Anhang **A**
A.1 Reproduktion der Ergebnisse . . . . . A
A.2 Softwareumgebung der Benchmarks . . . . . D
A.3 Optimalitätsgap je Konfigurationsstufe . . . . . D

Symbolverzeichnis

Symbol	Bedeutung
Graphen	
$G = (V, E)$	Graph mit Knotenmenge V und Kantenmenge E
u, v	Knoten eines Graphen. Ohne Argument bezeichnet v im Lagerlayout zusätzlich die Zahl der vertikalen Gänge
$w(e)$	Gewicht der Kante e. Ohne Argument steht w für die Lagerbreite
$\deg_T(v)$	Grad des Knotens v im Graphen T
T	minimaler Spannbaum (Christofides), im Fixed-Parameter-Kapitel ein Tour-Teilgraph
M	minimales perfektes Matching
O	Menge der Knoten ungeraden Grades, nicht die Landau-Notation am Ende dieser Liste
C	Eulerscher Kreis, im Versuchsplan die Menge der Lagerkonfigurationen
π	Knotenreihenfolge einer Tour
$\mathcal{T}$, X_i	Zerlegungsbaum und zugehörige Taschen
$\mathrm{tw}(G)$, $\mathrm{pw}(G)$	Baumweite und Pfadweite von G
Lagerlayout	
r	Anzahl der Regalreihen
h	Anzahl der Quergänge; im vollen Lagergitter $h = r + 1$
h_{eff}	effektive Quergangzahl eines Auftrags, also die Quergänge bis zum obersten zu besuchenden Regalblock, $h_{\text{eff}} \leq r + 1$
w	Anzahl der Regalspalten (Lagerbreite)
v	Anzahl der vertikalen Gänge, $v = w + 1$
B, H	Breite und Höhe des Lagergitters in Zellen

Symbol	Bedeutung
n_{isles}, n_{rows}	Konstruktorparameter des Gitters, die Implementierungsnamen von w und r
(x, y)	Koordinate im Zellengitter
Distanzen und Routen	
n	Auftragsgröße, also Anzahl der zu besuchenden Lagerorte. In den Komplexitätsschranken allgemein die Eingabegröße
$d(a, b)$	Manhattan-Distanz zwischen a und b
$d_{\text{Lager}}(a, b)$	Lagerdistanz, kürzester Weg unter Berücksichtigung der Regale
c_u, c_o	Distanz zum unteren bzw. oberen Quergang
$L_{\text{Algorithmus}}$, L_{Optimum}	Routenlänge des Verfahrens bzw. des Optimums
L_A, L_B	mittlere Routenlängen im Break-even-Vergleich
t_A, t_B	mittlere Berechnungszeiten im Break-even-Vergleich
v_{Geh}	Gehgeschwindigkeit der Kommissionierkraft
v_0	Depot
Fixed-Parameter-Algorithmus	
ω	Frontier-Zustand, Paar aus Paritäts- und Komponententupel
x_i	Parität der i-ten Kreuzung, $x_i \in \{0, U, E\}$, nicht die Koordinate (x, y)
c_i	Komponentenlabel der i-ten Kreuzung
Λ, Λ'	Schicht der dynamischen Programmierung (Zustand $\to$ Kosten)
$\Omega(h)$	Menge der möglichen Frontier-Zustände
S_k	Super-Catalan-Zahl (kleine Schröder-Zahl)
L	Länge eines Gangsegments, im Pseudocode die Menge der Lagerplätze. Routenlängen tragen stets einen Index
γ	größte Lücke zwischen aufeinanderfolgenden Picks
d_u, d_o	Distanz zum entferntesten Pick unten bzw. oben
Komplexität	
k	struktureller Eingabeparameter (FPT). Im Beweis zum Zustandsraum ein Summationsindex, im Batching-Ausblick die Zahl der Produkte je Auftrag

Symbol	Bedeutung
m	Zahl der Kundenaufträge im Batching-Ausblick
$f(k)$	berechenbare Funktion in der FPT-Laufzeitschranke
c	Konstante in der FPT-Laufzeitschranke, nicht zu verwechseln mit c_u, c_o und c_i
α	Approximationsfaktor
O, Θ	asymptotische obere bzw. scharfe Schranke, gleiches Zeichen wie die Knotenmenge oben
Versuchsplan	
R	Gesamtzahl der Läufe eines Versuchsplans
P	Menge der Produktanzahlen, im Pseudocode des Fixed-Parameter-Algorithmus die Pickmenge
I, A	Iterationen je Kombination und Menge der getesteten Algorithmen

Abbildungsverzeichnis

2.1 Terminologie des Lagerlayouts 11
2.2 Lagerlayout in der Webanwendung 12
2.3 Die drei Modelle desselben Lagers 14

4.1 Aufbau der Webanwendung 31
4.2 Der Frontier-Zustand beim Sweep 43
4.3 Zulässige und unmögliche Komponentenmuster 45
4.4 Die sechs Traversierungen eines Gangsegments 48

5.1 Berechnungszeit über der Lagerbreite, Benchmark B1 67
5.2 Routenlänge über der Lagerbreite, Benchmark B1 68
5.3 Optimalitätsgap über der Lagerbreite, Benchmark B1 69
5.4 Berechnungszeit über der Regalreihenzahl, Benchmark B2 74
5.5 Berechnungszeit über der Auftragsgröße, je Regalreihenstufe . . . 75
5.6 Routenlänge über der Regalreihenzahl, Benchmark B2 76
5.7 Optimalitätsgap über der Regalreihenzahl, Benchmark B2 77
5.8 Berechnungszeit über der Lagergröße, Benchmark B3 80
5.9 Routenlänge über der Lagergröße, Benchmark B3 81
5.10 Optimalitätsgap über der Lagergröße, Benchmark B3 82

Tabellenverzeichnis

4.1 Laufzeitanteile der Christofides-Phasen 40
4.2 Die neun Transitionen des Fixed-Parameter-Algorithmus 50
4.3 Anzahl der Frontier-Zustände $|\Omega(h)|$ 56
4.4 Verteilung der Testsuite auf die Komponenten 60

5.1 Berechnungszeit über der Lagerbreite, Benchmark B1 69
5.2 Effektive Quergangzahl terminierter und abgebrochener Läufe, Benchmark B2 71
5.3 Berechnungszeit je Produktanzahl, Benchmark B2 73
5.4 Routenlänge über der Regalreihenzahl, Benchmark B2 74
5.5 Berechnungszeit und Timeout-Anteil über der Regalreihenzahl, Benchmark B2 77
5.6 Berechnungszeit und Timeout-Anteil über der Lagergröße, Benchmark B3 82
5.7 Optimalitätsgap über alle drei Benchmarks 83
5.8 Varianzanteile von Auftragsgröße und Layout 86

6.1 Empfohlener Algorithmus je Regalreihenzahl und Zeitbudget . . . 93

A.1 Python-Paketversionen der Benchmark-Umgebung . . . . . D
A.2 Mittlerer Optimalitätsgap je Lagerbreite in Benchmark B1 ($r = 1$) E
A.3 Mittlerer Optimalitätsgap je Regalreihenzahl in Benchmark B2 (Breite 10) . . . . . E
A.4 Mittlerer Optimalitätsgap je Lagergröße in Benchmark B3 . . . . E

Kapitel 1

Einleitung

1.1 Motivation & Problemkontext

Die Berechnung kürzester Wege ist in der Algorithmentheorie ein historisch prominentes Problem. Das Shortest Path Problem (SPP), welches für einen gegebenen Start- und Endpunkt den kürzesten Weg berechnet (Dijkstra 1959), oder auch das Travelling Salesperson Problem (TSP), welches für n gegebene Orte die kürzeste Route berechnet (Dantzig, Fulkerson und S. Johnson 1954), werden in dem Forschungsfeld schon seit vielen Jahren untersucht.

Durch die Vielzahl an praktischen Anwendungen dieser Algorithmen besteht insbesondere in der Lagerlogistik ein großes wirtschaftliches Interesse an hocheffizienten Ansätzen, die in möglichst geringen Berechnungszeiten kurze Routen berechnen (De Koster, Le-Duc und Roodbergen 2007). Eine solche Anwendung ist das Kommissionieren von Produkten in einem Lagersystem, bei dem Kommissionierkräfte eine gegebene Menge von Produkten einsammeln und verpacken. Das Optimieren der Routen ist dabei von hoher Relevanz, da so Laufwege und daraus bedingt Kosten reduziert werden können (Petersen 1997). Das Lagersystem wird hierfür als Graph modelliert, in dem die begehbaren Gänge und Kreuzungen den zulässigen Bewegungsraum darstellen (De Koster, Le-Duc und Roodbergen 2007). Die Kommissionierkraft sucht für eine gegebene Anzahl von Produkten in verschiedenen Lagerorten die optimale Route (Ratliff und Rosenthal 1983).

1.2 Problemstellung

Das beschriebene Problem ist in der Literatur als das Order Picking Routing Problem (OPRP) bekannt. Dies ist eine Variante des TSP, bei der nicht alle Knoten des Graphen besucht werden müssen, sondern nur jene, an denen ein Produkt kommissioniert wird. Alle anderen Knoten dürfen zur Traversierung genutzt werden (Abschnitt 2.4.2).

Das klassische TSP ist NP-schwer (Cormen u. a. 2009, Kap. 34.5.4). Die Herleitung findet sich in Abschnitt 2.4. Da die Eingabe des OPRP auf rechteckige Lagergraphen beschränkt ist, war lange offen, ob das OPRP auch der Klasse der NP-schweren Probleme angehört. Prunet, Absi und Cattaruzza (2025) bestätigen in ihrer Arbeit diese Klassenzugehörigkeit für Probleme mit einer beliebigen Anzahl an Quergängen im Lagerlayout. Für eine begrenzte Anzahl an Quergängen zeigen Cambazard und Catusse (2018) eine Lösung in polynomieller Zeit auf. Die Komplexität des OPRP ist also unmittelbar von der Anzahl der Quergänge abhängig.

Exakte Lösungsverfahren für das allgemeine TSP, etwa die dynamische Programmierung von Held und Karp (1962), haben eine exponentielle Laufzeit in der Zahl der zu besuchenden Orte. Für das OPRP existieren dagegen mehrere exakte Verfahren, die die Struktur des Lagers ausnutzen (Kapitel 3). Das in dieser Arbeit implementierte Verfahren von Cambazard und Catusse (2018) bindet die Laufzeit an die Zahl der Quergänge. Sie wächst exponentiell in der Quergangzahl, aber nur linear in der Zahl der zu besuchenden Lagerorte. Die Berechnungsdauer der Route steht somit in direkter Abhängigkeit vom gegebenen Layout des Lagers.

In der Praxis kommen exakte Verfahren dennoch selten zum Einsatz. De Koster, Le-Duc und Roodbergen (2007) nennen dafür drei Gründe. Für viele Layouts existiert überhaupt kein exaktes Verfahren, optimale Routen erscheinen den Kommissionierkräften unlogisch, sodass sie davon abweichen, und Staus in den Gängen lassen sich mit einem exakten Verfahren nicht berücksichtigen. Verbreitet sind daher regelbasierte Heuristiken.

1.3 Zielsetzung & Forschungsfrage

Um kürzeste Wege in Lagersystemen bestimmen zu können, wurden bereits einige Ansätze vorgestellt. Diese teilen sich in verschiedene Paradigmen auf. Heuristische Algorithmen schränken den Suchraum durch feste Regeln ein. Sie berechnen dadurch schnell eine Route, die jedoch meist länger ist als die Optimallösung. Approximative Algorithmen stützen sich auf bewiesene Schranken der Routengüte und geben damit eine Qualitätsgarantie. Parametrisierte Algorithmen berechnen die kürzeste Route exakt, binden ihre Laufzeit aber nicht an die Eingabegröße, sondern an einen Parameter des Problems. Ein empirischer Vergleich dieser drei Paradigmen an variablen Lagerlayouts fehlt bislang (siehe Abschnitt 3.5). Diesen Vergleich leistet die vorliegende Arbeit.

Das primäre Ziel der Arbeit ist es, den Trade-off zwischen Berechnungszeit und Routenlänge systematisch für variierende Lagerlayouts zu untersuchen. Implementiert werden die Nearest-Neighbor-Heuristik (Rosenkrantz, Stearns und Lewis 1977), der Christofides-Algorithmus (Christofides 1976) und der Fixed-Parameter-Algorithmus nach Cambazard und Catusse (Cambazard und Catusse 2018). Die drei Algorithmen sind bewusst gewählt, da jeder von ihnen eines der genannten Paradigmen repräsentiert. Die Heuristik, die Approximation und das parametrisierte Verfahren. Als Problemgröße werden in dieser Arbeit zwei Größen unterschieden. Die Auftragsgröße, also die Anzahl der zu besuchenden Lagerorte, und die Lagerdimensionen, also Lagerbreite und Regalreihenzahl. Durch diese Auswahl lässt sich nicht nur beurteilen, wie sich die Problemgröße in diesem Sinne auf die Berechnungsdauer und Routenlänge auswirkt, sondern auch, welches algorithmische Paradigma für spezifische Lagerlayouts den besten Kompromiss aus Lösungsqualität und Rechenzeit bietet.

Daraus leitet sich die zentrale Forschungsfrage ab: “Wie unterscheiden sich der Christofides-, Nearest-Neighbor- und Fixed-Parameter-Algorithmus nach Cambazard und Catusse hinsichtlich ihrer Lösungsqualität und Laufzeit bei steigender Problemgröße und variierenden Lagerlayouts des OPRP?”

1.4 Methodik

Um die Eigenschaften der genannten Algorithmen zu untersuchen, wurde eine interaktive, Flask-basierte Webanwendung entwickelt. In dieser sind Regalreihen und Regalspalten dynamisch anpassbar. Zusätzlich können Kommissionieraufträge randomisiert generiert und die berechneten Routen visualisiert und verglichen werden. Ergänzend wurde eine Benchmarking-Funktion implementiert, die durch systematische Variation von Lagerdimensionen und Produktanzahlen die Algorithmen unter reproduzierbaren Bedingungen evaluiert. Erfasst werden Routenlänge und Berechnungszeit. Daraus wird die Abweichung von der optimalen Lösung berechnet. Die Berechnungszeit umfasst allein die Routenberechnung, also die Bestimmung der Besuchsreihenfolge und ihrer Länge. Die Expansion dieser Reihenfolge in einen begehbaren Gitterpfad dient nur der Darstellung und liegt außerhalb der Messung. Die Benchmarks laufen in einem Docker-Container, sodass die Ergebnisse unabhängig von der lokalen Softwareumgebung reproduzierbar sind. Aus den Messungen wird abschließend eine Entscheidungsregel abgeleitet, die in Abhängigkeit von Regalreihenzahl und Zeitbudget angibt, welches Verfahren zu wählen ist. Diese wird um eine Wirtschaftlichkeitsrechnung ergänzt. Die genaue Beschreibung der Methodik und des Versuchsaufbaus folgt in den Kapiteln 4 und 5.

1.5 Aufbau der Arbeit

Diese Arbeit führt zunächst in Kapitel 2 in die relevanten theoretischen Grundlagen ein. Dazu gehören die Graphen-, die Komplexitätstheorie, die formale Definition des TSP sowie dessen Lösungsparadigmen und die Lagerlayout-Modellierung. In Kapitel 3 wird der aktuelle Stand der Forschung präsentiert und die bestehende Literatur eingeordnet. Es werden die klassischen Routenstrategien sowie graphenbasierte und parametrisierte Ansätze eingeordnet. Die Methodik und Implementierung werden in Kapitel 4 genau behandelt. Hier werden der Aufbau der Webanwendung und die Implementierung der Algorithmen beschrieben. Die experimentellen Ergebnisse und der Versuchsaufbau sowie der Einfluss des Lagerlayouts werden in Kapitel 5 evaluiert. In Kapitel 6 werden die Ergebnisse und die Forschungsfrage diskutiert und die Limitationen dieser Arbeit benannt. Ein Fazit

und Ausblick wird in Kapitel 7 gegeben. Hier werden die Kernerkenntnisse und der Ausblick auf zukünftige Forschungsrichtungen zusammengefasst.

Kapitel 2

Grundlagen

2.1 Grundlagen der Graphentheorie

In der Informatik werden Routen häufig auf Graphen modelliert. Der Graph stellt hier das Netzwerk an möglichen Wegen dar und die Route ist ein gegebener Pfad durch ein solches Netzwerk. Nach Cormen u. a. (2009, Anh. B.4) wird zwischen gerichteten und ungerichteten Graphen unterschieden.

Ein ungerichteter Graph G wird als ein Paar $G = (V, E)$ beschrieben. V bildet hier eine endliche Menge an Knoten ab und $E \subseteq \{\{u, v\} \mid u, v \in V\}$ bildet die Menge der Kanten ab, wobei jede Kante durch ein ungeordnetes Paar von Knoten dargestellt wird. Bei einem gerichteten Graphen ist die Kantenmenge dagegen $E \subseteq V \times V$, die Kanten sind also geordnete Paare. Damit ist die Kantenrelation nicht symmetrisch. Aus $(u, v) \in E$ folgt nicht, dass auch $(v, u) \in E$ gilt.

Besonders relevant für die Routenfindung sind gewichtete Graphen. Ein gewichteter Graph erweitert G um eine Gewichtsfunktion $w : E \to \mathbb{R}$, die jeder Kante ein Gewicht zuordnet. Das Kantengewicht kann beispielsweise die Distanz zwischen zwei Lagerorten repräsentieren.

2.1.1 Pfade, Zyklen und Kreise

Ein Pfad ist durch eine Folge von Knoten definiert, bei der aufeinanderfolgende Knoten durch Kanten verbunden sind. Knoten und Kanten dürfen dabei mehrfach

vorkommen, da eine Kommissionierroute denselben Gang mehrmals durchlaufen kann. So kann ein Pfad durch einen Graphen eine Route darstellen. Pfade können hierbei unterschiedliche Eigenschaften erfüllen. Ein Pfad ist zyklisch, wenn der Start- und Endknoten des Pfades identisch ist. Eine besondere Unterform des Zyklus ist der Hamiltonsche Kreis. Dies ist ein Zyklus, der jeden Knoten genau einmal besucht und dann wieder zum Ausgangsknoten zurückkehrt. Hierbei sind nur die Knoten von Relevanz. Es muss nicht jede Kante einmal genutzt werden.

Wird das Routenfindungsproblem auf einem Distanzgraphen abgebildet, entspricht eine Kommissionierroute genau einem Hamiltonschen Kreis, da jeder seiner Knoten ein Lagerort modelliert, der besucht werden muss. Diese Reduktion auf einen Distanzgraphen wird in Abschnitt 4.3 durchgeführt.

Der Eulersche Kreis stellt eine bestimmte Form eines Zyklus dar und spielt eine zentrale Rolle in dem Christofides-Algorithmus (siehe Abschnitt 4.5.1). Ein Eulerscher Kreis ist ein Zyklus, der jede Kante im Graphen genau einmal durchläuft. Dieser existiert genau dann, wenn der gegebene Graph zusammenhängend ist und jeder Knoten einen geraden Grad aufweist (Diestel 2017). Der Grad eines Knotens ist definiert durch die Anzahl der mit ihm verbundenen Kanten.

Um den Eulerschen Kreis zu berechnen, wird in dieser Arbeit der Hierholzer-Algorithmus verwendet. Dieser erhält einen Graphen als Eingabe, der zusammenhängend ist und in dem alle Knoten eine gerade Anzahl an Kanten aufweisen. Die Laufzeit ist durch $O(|E|)$ beschränkt, wobei $|E|$ die Anzahl der Kanten bezeichnet (Diestel 2017). Der Ablauf der implementierten Variante wird in Abschnitt 4.5.2 beschrieben.

2.1.2 Spannbaum und Minimum Spanning Tree

Ein Teilgraph eines Graphen $G = (V, E)$ ist ein Graph, dessen Knoten- und Kantenmenge in V bzw. E enthalten sind. Ein Graph heißt zusammenhängend, wenn zwischen je zwei Knoten ein Pfad existiert.

Der Spannbaum ist ein wichtiges Konzept in der Graphentheorie. Dieser ist definiert als ein Teilgraph eines zusammenhängenden Graphen $G = (V, E)$, der alle Knoten aus V enthält und ein Baum ist. Ein Baum ist definiert als ein

zusammenhängender Graph ohne Zyklen. Ein Spannbaum hat genau $|V| - 1$ Kanten (Cormen u. a. 2009, Kap. 23).

Der Minimum Spanning Tree (MST) ist eine besondere Unterform eines Spannbaums, dessen akkumulierte Kantengewichte minimal sind: $\min \sum_{e \in T} w(e)$. Die klassischen Berechnungsalgorithmen für einen MST sind die Verfahren nach Kruskal und Prim (Cormen u. a. 2009, Kap. 23). Der MST ist besonders relevant für den Christofides-Algorithmus, da das Gewicht des MST eine untere Schranke für die optimale Tour durch einen Graphen darstellt. Diese Schranke wird in Abschnitt 4.5.1 für die Herleitung der Gütegarantie verwendet.

2.1.3 Matching

Ein Matching eines Graphen $G = (V, E)$ ist definiert durch eine Kantenmenge $M \subseteq E$, bei der kein Knoten in mehr als einer Kante vorkommt. Ein perfektes Matching ist dadurch definiert, dass es jeden Knoten des Graphen abdeckt (Diestel 2017). Bei einem perfekten Matching mit minimalem Gewicht wird $\sum_{e \in M} w(e)$ minimiert.

In jedem Graphen existiert eine gerade Anzahl von Knoten mit ungeradem Grad. Dies folgt aus dem Handschlaglemma, nach dem die Summe aller Knotengrade eines Graphen gerade ist (Cormen u. a. 2009, Übung B.4-1).

2.2 Lagerlayout und Kommissionierung

Das Lagerlayout, welches in dieser Arbeit verwendet wird, basiert auf rechteckigen Lagerregalen, welche in Reihen und Spalten angeordnet werden (De Koster, Le-Duc und Roodbergen 2007). Durch diese Anordnung entstehen Gänge, welche im 90-Grad-Winkel aufeinander treffen. Das Depot, an dem jede Route beginnt und endet, liegt in dieser Arbeit an einer Ecke des Lagers.

Für die weiteren Kapitel werden diese Gänge unterschieden. Die vertikalen Gänge verlaufen zwischen den Regalspalten an denen die Lagerorte positioniert sind. Die horizontalen Gänge verbinden die vertikalen Gänge miteinander und werden als

Quergänge bezeichnet. Auf diesen liegen keine Lagerorte. Die Anzahl der Regalreihen wird mit r bezeichnet, die Anzahl der Quergänge mit h. Da oberhalb und unterhalb jeder Regalreihe ein Quergang verläuft und aufeinanderfolgende Reihen sich einen Quergang teilen, gilt $h = r + 1$. Für den Fixed-Parameter-Algorithmus zählt allerdings nicht die Anzahl der Quergänge des Lagerlayouts, sondern nur die Zahl derjenigen, die ein Auftrag erreicht, also die Quergänge vom Depot bis einschließlich des Quergangs oberhalb des obersten zu besuchenden Regalblocks. Wo im Folgenden von h als Parameter die Rede ist, ist diese auftragsabhängige Größe gemeint. $r + 1$ gibt lediglich ihre Obergrenze an (vgl. Abschnitt 4.6). Der Abschnitt eines vertikalen Gangs zwischen zwei benachbarten Quergängen wird als Gangsegment bezeichnet.

Die Anzahl der Regalspalten, im Folgenden auch als Lagerbreite bezeichnet, wird mit w bezeichnet. Links und rechts jeder Regalspalte verläuft ein vertikaler Gang, sodass es $v = w + 1$ vertikale Gänge gibt. Der Bereich zwischen zwei aufeinanderfolgenden Quergängen wird als Block bezeichnet. Ein Layout mit r Regalreihen besitzt damit r Blöcke und ist für $r = 1$ einblockig, für $r \geq 2$ mehrblockig. Abbildung 2.1 fasst diese Begriffe zusammen. Abbildung 2.2 zeigt ein solches Layout in der entwickelten Webanwendung.

Das Lager wird zunächst als zweidimensionales Zellengitter modelliert. Jede Zelle ist entweder begehbar (Gang oder Kreuzung) oder blockiert (Regal). Lagerorte liegen auf den Regalzellen. Da Regale nicht betreten werden, wird für die Wegberechnung jeweils die unmittelbar angrenzende begehbare Gangzelle verwendet (siehe Abschnitt 4.3). Dieses Zellengitter beschreibt den Bewegungsraum und dient als Grundlage der Distanzberechnung. Davon zu unterscheiden sind die beiden Graphen, auf die das Lager für die Algorithmen abgebildet wird. Diese werden in Abschnitt 2.2.1 eingeführt.

Jeder Lagerort erhält eine fortlaufende Nummer. Diese wird auf eine (x, y)-Koordinate im Gitter abgebildet, wobei die Nummerierung spaltenweise innerhalb der Regalblöcke erfolgt. Die genaue Berechnung wird in Kapitel 4 beschrieben.

Distanzen in dieser Art von Lagersystemen sind durch die Abwandlung der Manhattan Distanz einfach zu berechnen. Diese ergibt sich als Summe der absoluten Differenzen der x- und y-Koordinaten zwischen zwei Punkten in einem Koordinatensystem:

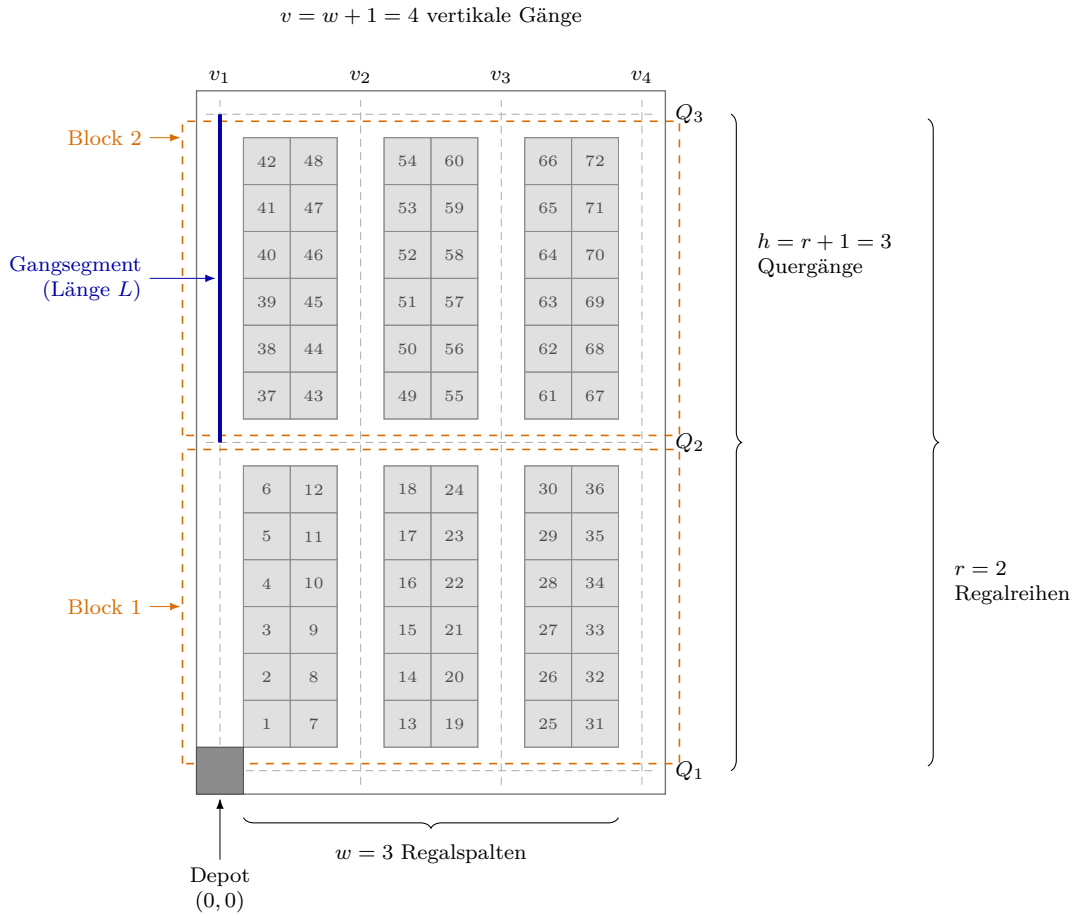


Abbildung 2.1: Terminologie des in dieser Arbeit verwendeten Lagerlayouts, hier mit $w = 3$ Regalspalten und $r = 2$ Regalreihen. Die grauen Flächen sind Regale, die gestrichelten Linien die Mittellinien der begehbaren Gänge. Aus w folgen $v = w + 1$ vertikale Gänge, aus r folgen $h = r + 1$ Quergänge, da benachbarte Regalreihen sich einen Quergang teilen. Ein Block ist der Bereich zwischen zwei benachbarten Quergängen, ein Gangsegment der Abschnitt eines vertikalen Gangs zwischen zwei benachbarten Quergängen. Jeder Regalblock ist in seine zwölf Lagerplätze unterteilt und mit deren Indizes beschriftet. Die Nummerierung verläuft spaltenweise innerhalb eines Regalblocks und innerhalb einer Spalte vom Depot weg aufsteigend, die Regalblöcke selbst werden zeilenweise ab der Regalreihe am Depot durchnummeriert (siehe Abschnitt 4.2). Die Zeichnung ist maßstäblich in Gitterzellen.

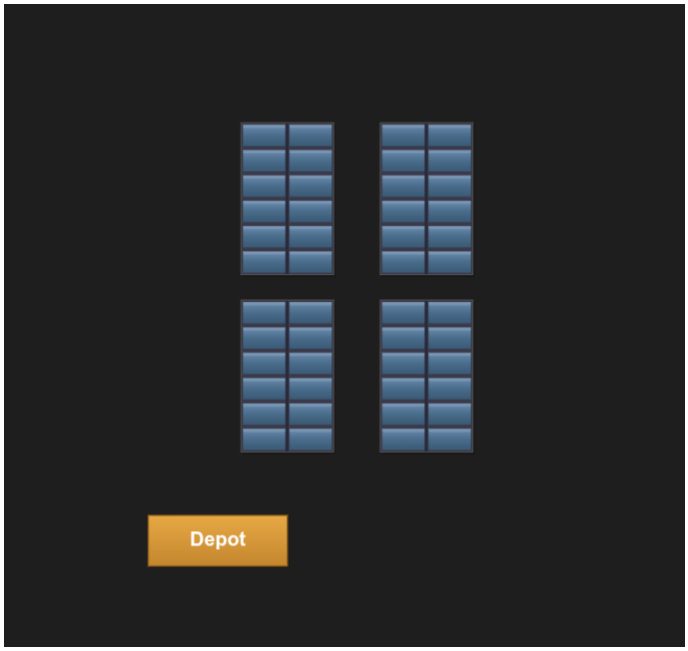


Abbildung 2.2: Lagerlayout in der entwickelten Webanwendung, hier mit zwei Regalspalten und zwei Regalreihen. Jeder Regalblock enthält zwölf Lagerplätze. Der hervorgehobene Kasten außerhalb des Gangnetzes zeigt das Depot in der unteren linken Ecke (vgl. Abschnitt 4.2).

$$d(a, b) = |x_a - x_b| + |y_a - y_b| \tag{2.1}$$

Befinden sich zwei Lagerorte a und b an derselben Gangspalte, also $x_a = x_b$ nach der Abbildung auf die angrenzende Gangzelle, so entspricht die Distanz direkt der Manhattan-Distanz. Der senkrechte Weg innerhalb einer Gangspalte ist immer begehbar. Liegen die Lagerorte jedoch in unterschiedlichen Gängen, muss die Kommissionierkraft zunächst den aktuellen Gang über eine Kreuzung verlassen. Dabei gibt es zwei Möglichkeiten. Entweder den Gang nach unten oder nach oben verlassen. Die Distanz ergibt sich dann als Minimum beider Wege:

$$d_{\text{Lager}}(a, b) = \begin{cases} d(a, b) & \text{wenn } x_a = x_b \\ \min(d_{\text{unten}},\ d_{\text{oben}}) & \text{sonst} \end{cases} \tag{2.2}$$

Dieselbe Fallunterscheidung liegt später auch der Konstruktion des im Frontend dargestellten Gitterpfades zugrunde (Abschnitt 4.3).

Hierbei bezeichnen d_{unten} und d_{oben} die Distanzen über die untere bzw. obere Gangkreuzung:

$$d_{\text{unten}} = c_{\text{unten}} + d(k_{\text{unten}}, b), \quad d_{\text{oben}} = c_{\text{oben}} + d(k_{\text{oben}}, b) \tag{2.3}$$

Dabei ist c_{unten} die Distanz vom Startort a zur unteren Kreuzung k_{unten} und c_{oben} die Distanz zur oberen Kreuzung k_{oben}. Von der jeweiligen Kreuzung aus wird die restliche Distanz zum Zielort b über die Manhattan-Distanz berechnet.

2.2.1 Graphenmodelle des Lagers

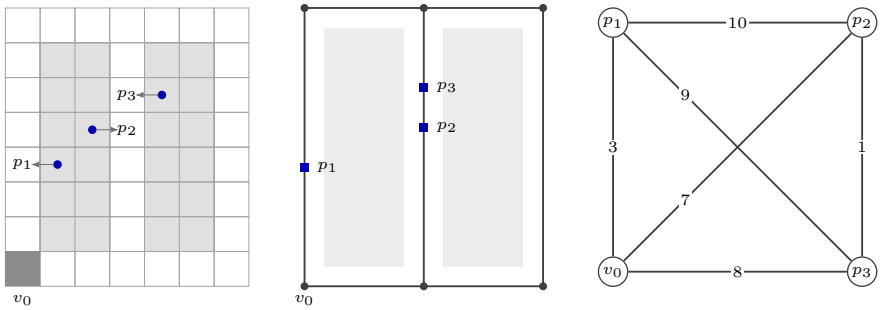
Das Lagerlayout wird in dieser Arbeit auf zwei verschiedene Graphen abgebildet. Beide modellieren dasselbe Lager, unterscheiden sich jedoch in der Representation der Kanten und Knoten. Welcher der beiden Graphen verwendet wird, hängt vom jeweiligen Algorithmus ab. Abbildung 2.3 stellt beide Graphen dem Zellengitter derselben Instanz gegenüber.

Der Gittergraph bildet das Lagerlayout direkt ab. Seine Knoten sind die Kreuzungen der Gänge, seine Kanten die Gangabschnitte zwischen zwei Kreuzungen. Die Lagerorte liegen damit nicht auf den Knoten, sondern auf den Kanten. Eine Kante muss bedient werden, wenn auf ihr ein Produkt des Auftrags liegt. Knoten und Kanten dürfen dabei mehrfach durchlaufen werden, da ein Gang auch als reiner Durchgang genutzt werden kann.

Der Distanzgraph abstrahiert von der Gangstruktur. Seine Knoten sind das Depot und die Lagerorte des jeweiligen Auftrags. Er ist vollständig, das heißt je zwei Knoten sind durch eine Kante verbunden. Das Gewicht einer Kante ist die Länge des kürzesten Weges zwischen den beiden Lagerorten im Lager. Da diese Gewichte exakte kürzeste Wege sind, erfüllt der Distanzgraph die Dreiecksungleichung (siehe Abschnitt 4.3). Welche Konsequenz das für die Wahl der Verfahren hat, wird in Abschnitt 2.5 behandelt.

Die Nearest-Neighbor-Heuristik und der Christofides-Algorithmus rechnen auf dem Distanzgraphen und bestimmen eine Reihenfolge der Lagerorte. Der Christofides-Algorithmus setzt dafür einen vollständigen Graphen voraus, dessen Gewichte die

Dreiecksungleichung erfüllen. Die Nearest-Neighbor-Heuristik benötigt lediglich eine Distanzfunktion, ihre Gütegarantie hängt aber ebenfalls an der Dreiecksungleichung. Der Fixed-Parameter-Algorithmus rechnet dagegen auf dem Gittergraphen (siehe Abschnitt 4.6).



(a) Zellengitter (b) Gittergraph (c) Distanzgraph

Abbildung 2.3: Dieselbe Instanz in den drei Modellen, die in dieser Arbeit verwendet werden: ein Lager mit $w = 2$ Regalspalten, $r = 1$ Regalreihe und drei zu besuchenden Lagerorten p_1, p_2, p_3. (a) Das Zellengitter beschreibt den Bewegungsraum. Graue Zellen sind Regale und nicht begehbar, die Lagerorte liegen auf Regalzellen und werden für die Wegberechnung auf die angrenzende Gangzelle abgebildet (Pfeile). (b) Im Gittergraphen sind die Knoten die Gangkreuzungen und die Kanten die Gangsegmente. Die Lagerorte liegen damit auf Kanten und nicht auf Knoten. Auf ihm rechnet der Fixed-Parameter-Algorithmus. (c) Der Distanzgraph enthält nur Depot und Lagerorte, ist vollständig und trägt als Kantengewichte die kürzesten Weglängen in Gittereinheiten. Auf ihm rechnen die Nearest-Neighbor-Heuristik und der Christofides-Algorithmus. Die Zeichenpositionen in (c) sind ohne Bedeutung, allein die Gewichte zählen.

2.3 Komplexitätstheorie

Die Komplexitätstheorie klassifiziert Probleme in verschiedene Komplexitätsklassen (Cormen u. a. 2009, Kap. 34). Für diese Arbeit sind die Klassen P und NP relevant sowie die über Reduktionen definierte Eigenschaft der NP-Schwere. Die parametrisierte Komplexitätstheorie erweitert diesen Rahmen um Klassen parametrisierter Probleme. Die hier zentrale Klasse FPT wird in Abschnitt 2.3.1 eingeführt.

Die Klasse P beinhaltet alle Entscheidungsprobleme, die mithilfe einer deterministischen Turingmaschine in polynomieller Zeit lösbar sind. Diese Probleme sind effizient lösbar und auch für große Eingaben berechenbar.

Die Klasse NP (nichtdeterministisch polynomiell) beinhaltet alle Entscheidungsprobleme, deren Lösung in polynomieller Zeit verifiziert werden kann. Ob P = NP gilt, also ob jedes effizient verifizierbare Problem auch effizient lösbar ist, ist eine der zentralen offenen Fragen der Informatik.

Die dritte Klasse ist die der NP-schweren Probleme. Ein Problem ist NP-schwer, wenn sich jedes Problem in NP in polynomieller Zeit auf dieses Problem reduzieren lässt. NP-schwere Probleme sind somit mindestens so schwer wie jedes Problem in NP.

Für die Praxis bedeutet NP-Schwere, dass unter der Annahme P $\neq$ NP kein Algorithmus existiert, der das Problem in polynomieller Laufzeit exakt löst. Stattdessen wächst die Berechnungszeit schneller als jedes Polynom in der Eingabegröße, weshalb für große Probleminstanzen auf Heuristiken oder Approximationsalgorithmen zurückgegriffen werden muss.

2.3.1 Fixed-Parameter Tractability

Fixed-Parameter Tractability erlaubt es, komplexe Probleme, die abhängig von ihrer Eingabegröße in der Berechnungszeit inakzeptabel ansteigen, in der Praxis dennoch effizient zu lösen. Hierfür wird die Komplexität des Problems neben der Eingabegröße n an einen Parameter k gebunden. Ein Problem gilt als Fixed-Parameter Tractable (FPT) bezüglich eines Parameters k, wenn es in einer Laufzeit $O(f(k) \cdot n^c)$ lösbar ist (Downey und Fellows 1999). Dabei ist f eine berechenbare Funktion, n die Eingabegröße und c eine Konstante. Das Ziel der FPT-Algorithmen ist es, die exponentielle Explosion der Laufzeiten auf den Parameter k zu beschränken. Bleibt k in der praktischen Anwendung klein, lässt sich das Problem trotz seiner generellen Komplexität effizient berechnen.

2.3.2 Baumweite und Pfadweite

Der Parameter k stellt für einen Fixed-Parameter-Algorithmus die zentrale Kenngröße der Eingabe dar. Für Routenprobleme auf Gittergraphen sind das die Baumweite und die verwandte Pfadweite. Beide werden im Folgenden eingeführt, weil der in dieser Arbeit implementierte Fixed-Parameter-Algorithmus seine Laufzeit an die Pfadweite des Lagergitters bindet (Abschnitt 4.6).

Jeder Graph hat eine Baumweite. Diese ist über die Baumzerlegung definiert. Eine Baumzerlegung eines Graphen $G = (V, E)$ ist ein Paar $(\mathcal{T}, \{X_i\})$. Jedem Knoten i des Baums $\mathcal{T}$ ist dabei genau eine solche Teilmenge zugeordnet. Diese Teilmengen werden als Taschen bezeichnet. Eine Tasche ist also ein Knoten des Zerlegungsbaums $\mathcal{T}$ und enthält eine Auswahl von Knoten aus G. Derselbe Knoten aus G darf dabei in mehreren Taschen vorkommen. Für die Taschen gelten drei Bedingungen (Bodlaender 1996):

1. Jeder Knoten aus V liegt in mindestens einer Tasche.
2. Für jede Kante $\{u, v\}$ aus E gibt es eine Tasche, die u und v gemeinsam enthält.
3. Für jeden Knoten v bilden alle Taschen, die v enthalten, einen zusammenhängenden Teilbaum von $\mathcal{T}$.

Die Weite einer Baumzerlegung ist die größte vorkommende Taschengröße minus eins. Die Baumweite $\mathrm{tw}(G)$ ist die kleinste Weite, die sich über alle möglichen Baumzerlegungen von G erreichen lässt. Der Abzug von eins ist Konvention und sorgt dafür, dass ein Baum mit mindestens einer Kante die Weite eins erhält.

Die Baumweite misst damit, wie baumähnlich die Struktur eines Graphen ist. Ein Baum mit mindestens einer Kante weist eine Baumweite von 1 auf, ein einfacher Kreis eine Baumweite von 2 und ein vollständiger Graph K_n eine Baumweite von $n - 1$. Ein $a \times b$-Gitter hat die Baumweite $\min(a, b)$ (Bodlaender 1996).

Je niedriger die Baumweite, desto einfacher ist die Struktur des Graphen. Viele NP-schwere Probleme auf allgemeinen Graphen werden FPT-lösbar, wenn sie auf die Baumweite parametrisiert werden können(Courcelle 1990; Arnborg, Lagergren und Seese 1991). Eine verwandte Kenngröße ist die Pfadweite $\mathrm{pw}(G)$. Sie beruht auf der Pfadzerlegung, einer Baumzerlegung, deren Baum ein Pfad ist und deren

Taschen damit eine lineare Folge bilden. Die Pfadweite ist die kleinste Weite über alle Pfadzerlegungen von G. Ein Pfad ist ein Baum, also ist jede Pfadzerlegung auch eine Baumzerlegung. Die Pfadzerlegungen sind damit nur ein Teil aller Baumzerlegungen, und das Minimum über diesen Teil kann nicht kleiner ausfallen. Für jeden Graphen gilt daher

$$\mathrm{tw}(G) \leq \mathrm{pw}(G) \tag{2.4}$$

(Bodlaender 1998, Lemma 3). Eine Schranke für die Pfadweite ist damit zugleich eine Schranke für die Baumweite, denn aus $\mathrm{pw}(G) \leq k$ folgt mit dieser Ungleichung unmittelbar $\mathrm{tw}(G) \leq k$. Für ein Lagerlayout führt die Baumweite dabei nicht weit. Die oben genannte Schranke $\min(a, b)$ eines $a \times b$-Gitters wächst mit der kleineren der beiden Seiten und wird für quadratische Lager entsprechend groß. Ausschlaggebend ist vielmehr die kleine Zahl der Quergänge. Für den Gittergraphen eines Lagers lässt sich eine Pfadzerlegung direkt angeben, und die lineare Anordnung ist hier die natürliche, da das Gitter sich Spalte für Spalte durchlaufen lässt. Die Menge der h Kreuzungen einer Gangspalte trennt den Graphen in einen linken und einen rechten Teil und kann als ein Separator betrachtet werden. Nimmt man beim Übergang zur nächsten Gangspalte jeweils den hinzukommenden Knoten in die Tasche auf, so entsteht eine Folge von Taschen der Größe höchstens $h + 1$ und damit eine Pfadweite von h (Cambazard und Catusse 2018; Pansart, Catusse und Cambazard 2018). Die Breite des Lagers erhöht nur die *Zahl* der Taschen, nicht deren Größe. Deshalb bleibt die Zerlegung schmal, solange h klein bleibt.

Bei Cambazard und Catusse (2018) bezeichnet h die Anzahl der horizontalen Linien des betrachteten Gitters. Pansart, Catusse und Cambazard (2018) übertragen den Ansatz auf das Lager, in dem diese Linien den Quergängen entsprechen. Für das volle Lagergitter beträgt die Pfadweite damit $r + 1$ (Abschnitt 2.2). Der Fixed-Parameter-Algorithmus rechnet jedoch nicht auf dem vollen Gitter, sondern nur bis zum obersten und rechtesten zu besuchenden Regalblock. Nach links und nach unten wird nicht beschnitten, da das Depot in der unteren linken Ecke liegt und in jeder Tour enthalten sein muss (vgl. Abschnitt 4.6.7). Für das so beschnittene Gitter ist die Pfadweite kleiner oder gleich $r + 1$. Welchen Wert sie annimmt, entscheidet allein die Lage des höchstgelegenen Pickortes eines Auftrags. Die Beschneidung nach rechts verkleinert dagegen nur die Zahl der Taschen und damit nicht die Pfadweite. Die Reihenfolge, in der die Picks besucht werden, spielt dabei

keine Rolle, denn das Gitter steht fest, bevor eine Route berechnet wird. Liegt der höchste Pick unterhalb der obersten Regalreihe, ist das effektive h_{eff} kleiner als $r + 1$. Eine Pfadweite von h bedeutet, dass beim spaltenweisen Durchlauf zu jedem Zeitpunkt nur h Kreuzungen offen sind. Der Fixed-Parameter-Algorithmus muss deshalb nur diese h Kreuzungen und die Zustände betrachten. Die Zahl der zu speichernden Zustände hängt damit allein von h ab und nicht von der Breite des Lagers (siehe Abschnitt 4.6).

2.4 Das Travelling Salesperson Problem (TSP)

Das Travelling Salesperson Problem (TSP) beschreibt das Problem einer Person, die n Städte besuchen muss. Dabei ist das Ziel die Reihenfolge der Städte so zu bestimmen, dass die Gesamtroute minimal wird (Cormen u. a. 2009, Kap. 35.2).

Das Problem wird definiert durch einen vollständigen, gewichteten Graphen $G = (V, E)$ mit $n = |V|$ Knoten und einer Gewichtsfunktion $w : E \to \mathbb{R}$. Gesucht ist eine Knotenreihenfolge π, sodass die Gesamtkosten der Tour minimal werden:

$$\min_{\pi} \sum_{i=1}^{n-1} w(\pi(i), \pi(i+1)) + w(\pi(n), \pi(1)) \tag{2.5}$$

Dies entspricht dem kürzesten Hamiltonschen Kreis in G (Dantzig, Fulkerson und S. Johnson 1954).

Karp (1972) klassifizierte das Bestimmen eines Hamiltonschen Kreises in einem gegebenen Graphen als NP-vollständig. Hieraus folgt die NP-Schwere des klassischen TSP (Cormen u. a. 2009, Kap. 34.5.4). Damit gilt für das TSP die in Abschnitt 2.3 beschriebene Konsequenz der NP-Schwere.

Neben dem allgemeinen TSP gibt es auch noch die Unterform des metrischen TSP. Dieses unterscheidet sich durch eine zusätzliche Erfüllung folgender Dreiecksungleichung: $w(u, v) \leq w(u, x) + w(x, v)$ für alle $u, v, x \in V$. Diese garantiert, dass der direkte Weg niemals länger ist als ein Umweg über einen Zwischenknoten. Der Christofides-Algorithmus setzt ein metrisches TSP voraus.

2.4.1 Symmetrisches und asymmetrisches TSP

Das TSP lässt sich zusätzlich anhand der Symmetrie der Gewichtsfunktion unterscheiden. Beim symmetrischen TSP gilt $w(u, v) = w(v, u)$ für alle $u, v \in V$. Die Distanz ist richtungsunabhängig. Beim asymmetrischen TSP ist dies nicht gegeben. In praktischen Anwendungen können asymmetrische Graphen beispielsweise genutzt werden, um Einbahnstraßen zu modellieren. In dieser Arbeit wird ausschließlich das symmetrische TSP betrachtet, da die Gänge im Lagersystem in beide Richtungen begehbar sind und die Wegkosten damit richtungsunabhängig sind.

2.4.2 Das Order Picking Routing Problem (OPRP)

Das Order Picking Routing Problem (OPRP) ist wie folgt definiert: Gegeben sind ein Lagerlayout, ein Depot und eine Menge von n Lagerorten, die besucht werden müssen. Gesucht ist die kürzeste Route, die alle n Orte besucht und zum Depot zurückkehrt. Im Gegensatz zum klassischen TSP müssen nicht alle Knoten im Graphen besucht werden, sondern nur eine Teilmenge. Die restlichen Knoten dürfen als Durchgangspunkte verwendet werden. Diese Verallgemeinerung des TSP ist in der Literatur als Steiner-TSP bekannt. Beim Steiner-TSP ist neben dem Graphen eine Teilmenge der Knoten ausgezeichnet, die besucht werden müssen. Alle übrigen Knoten dürfen als Durchgangsknoten genutzt werden. Das OPRP ist damit ein Steiner-TSP, dessen Eingabe auf Lagergraphen eingeschränkt ist. Diese Sichtweise liegt auch dem Ansatz von Cambazard und Catusse (2018) zugrunde, die das rektilineare TSP als Steiner-Problem auf dem Hanan-Graphen lösen. Liegen die zu besuchenden Punkte in der Ebene und verlaufen alle Wege achsenparallel, spricht die Literatur auch vom rektilinearen TSP. Der zugehörige Suchraum ist das *Hanan-Gitter*, also das Gitter aus allen waagerechten und senkrechten Linien durch die gegebenen Punkte. Im Lagerkontext ist dieses Gitter nicht aus den Punkten, sondern durch die Gangstruktur vorgegeben.

Die Komplexität des OPRP hängt davon ab, wie viele Quergänge das betrachtete Lager haben darf. Für ein einblockiges Lager mit zwei Quergängen ist es seit Ratliff und Rosenthal (1983) in Polynomialzeit lösbar. Ob das auch bei mehreren Blöcken gilt, galt lange als offen (Pansart, Catusse und Cambazard 2018). Prunet,

Absi und Cattaruzza (2025) zeigen, dass das Problem stark NP-schwer ist, sobald beliebig viele Quergänge zugelassen werden. Bei fester Quergangzahl bleibt es dagegen polynomiell lösbar. Pansart, Catusse und Cambazard (2018) zeigen Fixed-Parameter-Tractability in der Zahl der Quergänge. Der in dieser Arbeit implementierte Algorithmus von Cambazard und Catusse (2018) nutzt denselben Effekt, parametrisiert jedoch über die Anzahl der Regalreihen r.

2.5 Lösungsparadigmen für das TSP

Da das TSP schon viele Jahre besteht und eine große Anwendung in der Praxis findet, wurden viele unterschiedliche Lösungsansätze entwickelt. Zu den klassischen exakten Verfahren gehören das Branch & Bound und die dynamische Programmierung nach Held und Karp. Sie berechnen das Optimum, sind aber in der Eingabegröße exponentiell. Held-Karp hat eine Laufzeit von $O(n^2 \cdot 2^n)$ (Held und Karp 1962). Praktikabel sind sie deshalb nur für kleine Instanzen. Die drei Paradigmen, die in dieser Arbeit verglichen werden, lassen sich davon abgrenzen als heuristische, approximative und parametrisierte Verfahren. Heuristiken vereinfachen die Problemlösungen durch vereinfachte Grundannahmen, um schneller eine Lösung zu berechnen. Diese sorgen dafür, dass die Berechnungen sich deutlich vereinfachen, jedoch auch die Lösungsqualität stark variieren kann. Dadurch gibt es keine konstante Gütegarantie für die Abweichung vom Optimum. Ein Beispiel für einen solchen heuristischen Ansatz für das TSP ist die Nearest-Neighbor-Heuristik. Für metrische Instanzen ist ihre Güte im schlechtesten Fall nicht durch eine Konstante beschränkt. Das Verhältnis zur optimalen Tour wächst wie $\Theta(\log n)$, wobei die bewiesene obere Schranke $\frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}$ lautet (Rosenkrantz, Stearns und Lewis 1977). Ohne die Dreiecksungleichung existiert überhaupt keine Schranke. Die berechnete Tour kann also mit wachsender Knotenzahl immer weiter vom Optimum abweichen. Approximationsalgorithmen hingegen haben beweisbare Schranken für ihre Gütegarantie. Das heißt, die berechnete Lösung ist maximal α-mal so lang wie die optimale Lösung. Ein Beispiel für einen Approximationsalgorithmus wäre der Christofides-Algorithmus, der mit einer nachgewiesenen Gütegarantie von 3/2 Lösungen für ein metrisches TSP berechnet (Christofides 1976). Sie brauchen dafür meist länger als Heuristiken. Parametrisierte Algorithmen bilden das dritte Paradigma. Sie binden die Komplexität nicht an die Eingabegröße, sondern an

einen Parameter des Problems (vgl. Abschnitt 2.3.1) und liefern dabei exakte Lösungen. Ein Beispiel ist der in dieser Arbeit implementierte Fixed-Parameter-Algorithmus nach Cambazard und Catusse (2018), der die Komplexität an die Quergangzahl h bindet.

2.6 Kürzeste-Wege-Algorithmen

Da in einem Lagersystem die Routenfindung durch die Regale beschränkt ist, muss ein Routenfindungsalgorithmus verwendet werden, der mit diesen Beschränkungen im Bewegungsraum umgehen kann. Hierfür wird in dieser Arbeit die Fallunterscheidung der Lagerdistanz-Formel aus Abschnitt 2.2 wiederverwendet. Die Formel bestimmt bereits, über welchen Quergang der kürzeste Weg zwischen zwei Lagerorten führt, und legt damit zugleich den Verlauf des Weges fest. So können die Abfolgen von Lagerorten im Frontend als Weg durch das Lager dargestellt werden. Wie der Pfad daraus ohne weitere Suche entsteht, beschreibt Abschnitt 4.3. Dieses Vorgehen wird mithilfe des A*-Algorithmus validiert (Abschnitt 4.7.1). A* nach Hart, Nilsson und Raphael (1968) erweitert den Algorithmus von Dijkstra (1959) um eine Heuristikfunktion $\hat{q}$, die die Restdistanz zum Ziel schätzt. Für einen Knoten u wird der nächste zu besuchende Knoten nach $f(u) = g(u) + \hat{q}(u)$ ausgewählt, wobei $g(u)$ die bisherige Distanz vom Start und $\hat{q}(u)$ die geschätzte Restdistanz zum Ziel ist. Dadurch sucht A* bevorzugt in Richtung des Ziels.

Als Heuristik innerhalb von A* wird die Manhattan-Distanz verwendet. Sie ist nicht zu verwechseln mit der Lagerdistanz aus Abschnitt 2.2, die die Distanz zwischen zwei Lagerorten angibt. Auf dem Lagergitter ist die Manhattan-Distanz zulässig, denn jeder Schritt verändert genau eine Koordinate um eins, sodass kein Weg zwischen zwei Zellen kürzer sein kann als ihre Manhattan-Distanz. Regale können einen Weg nur verlängern, niemals verkürzen. Sie ist zudem konsistent, weil sie selbst die Dreiecksungleichung erfüllt und jede Kante des Gitters die Kosten eins hat. Für benachbarte Zellen u und v gilt damit $\hat{q}(u) \leq 1 + \hat{q}(v)$. Beides zusammen garantiert, dass A* den kürzesten Gitterpfad findet (Hart, Nilsson und Raphael 1968).

Kapitel 3

Stand der Forschung

Für das TSP und seine Spezialisierungen wie das OPRP existiert eine umfangreiche Forschungslandschaft. Diese Arbeit betrachtet davon drei Lösungsparadigmen: Heuristiken, Approximationsalgorithmen und parametrisierte Verfahren. In den folgenden Abschnitten werden die relevanten Arbeiten aus diesen Bereichen vorgestellt und in das in dieser Arbeit untersuchte OPRP eingeordnet.

3.1 Heuristiken für das TSP

Die Nearest-Neighbor-Heuristik löst das TSP mit einem sehr einfachen Ansatz. Rosenkrantz, Stearns und Lewis (1977) untersuchten diesen in ihrer Arbeit und erkannten, dass dieser auf metrischen Instanzen im schlechtesten Fall eine Approximationsgüte von $\Theta(\log n)$ aufweist. Praktisch ist diese Heuristik sehr gut einsetzbar, da sie die Tour greedy in $\Theta(n^2)$ konstruiert und damit selbst für große Eingabegrößen überschaubare Berechnungszeiten aufweist. Die Analyse von Rosenkrantz, Stearns und Lewis (1977) zeigt jedoch eine Schwäche auf. Da die Heuristik immer den nächstliegenden noch nicht besuchten Knoten besucht, kann es zu Umwegen kommen, wenn entfernte Knoten erst spät besucht werden. Die Qualität der Lösung ist daher stark von der räumlichen Verteilung der zu besuchenden Knoten abhängig.

Neben der Nearest-Neighbor-Heuristik hat sich in der Literatur auch das Lin-Kernighan-Verfahren von Lin und Kernighan (1973) etabliert, ein Local-Search-

Ansatz. Ausgehend von einer bestehenden Tour werden Kanten ausgetauscht, wobei die Anzahl der pro Schritt veränderten Kanten nicht fest ist, sondern während der Suche bestimmt wird. Dadurch wird die Tour als Ganzes berücksichtigt, was zu deutlich besseren Ergebnissen führt als bei Nearest-Neighbor. Obwohl diese Heuristik größere Berechnungszeiten aufweist, gilt das Verfahren als eine der effektivsten praktischen Heuristiken für das TSP. Helsgaun (2000) präsentiert eine Implementierung dieses Ansatzes, die auch für sehr große Instanzen Touren nahe am Optimum findet.

D. S. Johnson und McGeoch (1997) verglichen in einer umfassenden experimentellen Studie verschiedene TSP-Heuristiken, darunter 2-Opt (iteratives Ersetzen zweier Tour-Kanten), 3-Opt, Simulated Annealing und genetische Algorithmen. Dabei zeigte sich, dass Lin-Kernighan-basierte Verfahren bei vergleichbarem Zeitbudget die besten Touren liefern und sich in iterierter Form auch auf sehr große Instanzen skalieren lassen.

Heuristiken bieten somit effiziente Lösungsansätze für das TSP, haben jedoch keine konstante Gütegarantie. Diesen Nachteil adressieren Approximationsalgorithmen.

3.2 Approximationsalgorithmen für das TSP

Christofides (1976) stellte den in Abschnitt 2.5 eingeführten 3/2-Approximationsalgorithmus für das metrische TSP vor. Serdyukov (1978) entdeckte unabhängig denselben Algorithmus, weshalb dieser heute auch als Christofides-Serdyukov-Algorithmus bezeichnet wird. Über 45 Jahre blieb 3/2 die beste bekannte Schranke. Erst Karlin, Klein und Oveis Gharan (2021) unterboten sie um ein $\varepsilon > 10^{-36}$. Für die Praxis bleibt 3/2 damit die relevante Garantie. Eine exakte Lösung liefern Approximationsalgorithmen damit nicht. Die parametrisierte Komplexität verfolgt dafür einen anderen Weg.

3.3 Parametrisierte Verfahren

Exakte Lösungsansätze gehen meist mit exponentiellen Berechnungszeiten einher. Die parametrisierte Komplexität ermöglicht es, NP-schwere Probleme effizient zu lösen, sofern der Parameter klein ist. Downey und Fellows (1999) formalisierten das FPT-Framework. Es erlaubt, NP-schwere Probleme exakt zu lösen, sofern sie in der Klasse FPT liegen, sich ihre Komplexität also auf einen strukturellen Eingabeparameter k zurückführen lässt. Praktikabel ist die Laufzeit dabei nur, wenn k klein bleibt und die Funktion $f(k)$ moderat wächst.

Die Baumweite eines Graphen bietet sich bei Routenfindungsalgorithmen oft als Parameter k an. Ist sie nicht bekannt, muss zunächst eine Zerlegung berechnet werden. Bodlaender (1996) zeigte, dass sich für jeden festen Wert k in Linearzeit entscheiden lässt, ob ein Graph die Baumweite höchstens k besitzt, und in diesem Fall eine passende Zerlegung ausgegeben werden kann. Für das in dieser Arbeit betrachtete Lagerlayout wird dieser Schritt allerdings nicht benötigt, da sich die Zerlegung wie in Abschnitt 2.3.2 gezeigt direkt aus der Gangstruktur ergibt.

Diese Zerlegungen sind auch der Hebel für Courcelle (1990). In seiner Arbeit wird gezeigt, dass sich auf Graphen mit beschränkter Baumweite für jedes feste k eine ganze Klasse von Eigenschaften in Linearzeit entscheiden lässt. Dazu gehört jede Eigenschaft, die sich allein über Knoten- und Kantenmengen des Graphen ausdrücken lässt, etwa die Frage nach einem Hamiltonkreis. Formal ist das die monadische Logik zweiter Stufe. Arnborg, Lagergren und Seese (1991) übertrugen das Ergebnis auf Optimierungsprobleme und damit auf die Frage nach der kürzesten Tour. Für Lagergraphen mit beschränkter Baumweite ist die Existenz eines effizienten Verfahrens damit theoretisch gesichert. Beide Ergebnisse zeigen jedoch nur, dass ein solches Verfahren existiert, und nicht, wie ein für die Praxis effizienter Algorithmus konkret aussieht. Für das OPRP braucht es deshalb einen auf die Gangstruktur zugeschnittenen Algorithmus.

Cambazard und Catusse (2018) entwickelten einen Fixed-Parameter-Algorithmus für das rektilineare TSP mit einer Laufzeit von $O(n \cdot h \cdot 7^h)$. Der Parameter h bezeichnet dort die Anzahl der horizontalen Gitterlinien. In der Lageranwendung entspricht das der Anzahl der Quergänge (Pansart, Catusse und Cambazard 2018). Solange diese Zahl klein bleibt, liefert der Algorithmus exakte Lösun-

gen in praktikablen Berechnungszeiten. Wie sich diese Ansätze konkret auf die Lagerkommissionierung übertragen lassen, zeigen die folgenden Arbeiten.

3.4 Routenplanung in der Lagerkommissionierung

Die Routenfindung in der Lagerkommissionierung wurde in der Literatur umfassend untersucht.

De Koster, Le-Duc und Roodbergen (2007) geben einen umfassenden Überblick über den Stand im Lagerdesign und der Kommissionierung. Es werden die klassischen regelbasierten Strategien S-Shape (auch Transversal), Return, Midpoint, Largest-Gap und Composite (auch Combined) verglichen. Diese Heuristiken sind schnell, erreichen aber nicht die Qualität eines exakten Verfahrens. Vorgestellt wird dazu ein Vergleich von sechs Methoden unter zufälliger Einlagerung (Petersen 1997). Die besten Einzelheuristiken, Composite und Largest-Gap, liegen dort 9 bis 10 % über dem Optimum, während die je Instanz beste der fünf Heuristiken im Mittel rund 5 % darüber liegt.

Petersen (1997) verglich die Routing-Strategien experimentell. Sein zentrales Ergebnis ist, dass das optimale Routing alle untersuchten Heuristiken deutlich schlägt. Unter den Heuristiken hängt die geeignete Wahl vor allem von der Größe der Pickliste ab. Bei großen Aufträgen schneiden Composite und Transversal besser ab, bei kleinen Largest-Gap und Midpoint. Die Lagerform verändert vor allem die absolute Routenlänge. Die Qualität der Strategien bleibt über die Lagerlayouts weitgehend stabil, mit Ausnahme des breitesten untersuchten Layouts.

Ratliff und Rosenthal (1983) stellten den ersten exakten Algorithmus vor, der das OPRP in Polynomialzeit löst. Der Aufwand ist dabei linear zu der Zahl der Gänge. Der Algorithmus ist auf einfache Lagerlayouts mit zwei Quergängen beschränkt, zeigt aber, dass die spezielle Struktur von Lagergraphen die Komplexität des Problems reduzieren kann.

Roodbergen und De Koster (2001b) erweiterten diesen Ansatz auf Lager mit einem mittleren Quergang und damit auf drei Quergänge. In einer weiteren

Arbeit verglichen Roodbergen und De Koster (2001a) sechs Routing-Methoden miteinander, darunter auch ein Optimalverfahren. Beide Arbeiten zeigen, dass komplexere Layouts angepasste Algorithmen erfordern.

Die Arbeit von Theys u. a. (2010) stellt die Brücke zwischen der TSP-Forschung und der praktischen Anwendbarkeit dar. Sie zeigen, dass der Lin-Kernighan-Helsgaun-Solver bis zu 47 % kürzere Routen liefert als gängige Lagerheuristiken. Ein allgemeiner TSP-Solver schlägt die regelbasierten Strategien also deutlich.

Scholz u. a. (2016) entwickelten eine MIP-Formulierung für das Single-Picker Routing Problem und zeigen damit, dass sich das Problem auch mit mathematischer Programmierung exakt lösen lässt. Die Größe der Formulierung ist dabei unabhängig von der Anzahl der Pick-Standorte.

Pansart, Catusse und Cambazard (2018) präsentierten zwei exakte Algorithmen speziell für das OPRP: einen Dynamic-Programming-Ansatz und einen Mixed-Integer-Programming-Ansatz. Der Dynamic-Programming-Ansatz verallgemeinert das Verfahren von Ratliff und Rosenthal (1983) auf beliebig viele Quergänge. Den Algorithmus von Cambazard und Catusse (2018) setzen sie als Vergleichsbasis ein.

Die von Pansart, Catusse und Cambazard (2018) noch offen gelassene Komplexitätsfrage ist inzwischen teilweise beantwortet. Prunet, Absi und Cattaruzza (2025) zeigen, dass das OPRP stark NP-schwer ist, sobald man Lager mit beliebig vielen Blöcken zulässt. Damit ist die Härte an die unbeschränkte Blockzahl gebunden. Für jede feste Quergangzahl bleibt das Problem polynomiell lösbar, wie der in Abschnitt 3.3 beschriebene Algorithmus zeigt. Genau in diesem Spannungsfeld bewegt sich der Vergleich der drei Paradigmen in dieser Arbeit. Ein Verfahren, das die Struktur des Lagers ausnutzt, steht Verfahren gegenüber, die ohne diese Annahme auskommen.

3.5 Forschungslücke

Obwohl sich die Literatur umfassend mit dem TSP und dem OPRP beschäftigt hat, besteht eine Forschungslücke. Masae, Glock und Grosse (2020) legen ein systematisches Literaturreview zum Picker-Routing vor und dokumentieren die

Breite der entwickelten Verfahren. Ihre Bestandsaufnahme ist nach Lösungsmethoden und Layouttypen gegliedert, ein methodenübergreifender Vergleich auf einheitlicher Datenbasis findet sich darin jedoch nicht. Daraus ergeben sich drei Lücken:

1. Vergleiche zwischen Heuristiken und exakten Verfahren sind etabliert (Petersen 1997; Roodbergen und De Koster 2001a). Ein Vergleich, der zusätzlich ein Approximationsverfahren mit beweisbarer Gütegarantie und ein parametrisiertes Verfahren einbezieht, also alle drei in dieser Arbeit betrachteten Paradigmen zugleich, findet sich in der gesichteten Literatur jedoch nicht.
2. Die Layoutvariation wird in der Literatur jeweils innerhalb eines Paradigmas gefahren. Ob und wie stark die Gütelücke der approximativen und heuristischen Verfahren mit der Lagerform variiert, ist in der gesichteten Literatur nicht systematisch untersucht.
3. Die in der gesichteten Literatur berichteten Vergleiche sind nur eingeschränkt nachrechenbar, da Instanzen und Zufallsstartwerte nicht durchgängig veröffentlicht werden. Es fehlt ein Werkzeug, das Layout-Sweeps mit fixierten Seeds erzeugt und die Ergebnisse paradigmenübergreifend auswertet.

Diese drei Lücken adressiert die vorliegende Arbeit mit einer eigens implementierten Webanwendung, die reproduzierbare Benchmarks über alle drei Paradigmen hinweg erzeugt und auswertet.

3.6 Abgrenzung der Arbeit

Diese Arbeit schränkt den Vergleich auf drei Algorithmen ein: den Christofides-, den Nearest-Neighbor- und den Fixed-Parameter-Algorithmus von Cambazard und Catusse (2018). Klassische Lagerheuristiken wie S-Shape oder Largest-Gap sowie Metaheuristiken bleiben ausgeklammert, da sie sich keinem der drei verglichenen Paradigmen zuordnen lassen. Betrachtet wird ausschließlich ein Single-Picker-Szenario. Eine Kommissionierkraft besucht eine gegebene Menge von Lagerorten in einem konfigurierbaren, rechtwinkligen Lagerlayout mit parallelen Gängen. Order Batching und Zoning werden nicht behandelt, ebenso nicht das Gewicht und das Volumen der Produkte oder das Verhalten bei mehreren Kommissionierkräften.

Kapitel 4

Methodik & Implementierung

Dieses Kapitel beschreibt die Implementierung der Webanwendung und der drei Algorithmen, die in dieser Arbeit untersucht werden. Zunächst wird die gewählte Systemarchitektur beschrieben. Dann wird die Modellierung des Lagerlayouts eingeführt und die Funktionsweise der drei Algorithmen genau erklärt. Zum Schluss wird das Benchmarking-System vorgestellt, welches die Algorithmen bezüglich der Forschungsfrage testet und die Daten aggregiert.

4.1 Systemarchitektur

Die Webanwendung besteht aus einem Python-Flask-basierten Backend und einem nativen JavaScript-Frontend.

Das Flask-Backend kommuniziert mithilfe einer REST-API mit dem Frontend. Folgende API-Endpunkte wurden im Backend implementiert:

- `/generate-test-locations` (POST): Generiert n zufällige Lagerplätze mit einem optional hinzugegebenen Seed. n ist hier durch den gegebenen Input spezifiziert.
- `/calculate-route` (POST): Berechnet die Routen für die im Input spezifizierten Algorithmen und Lagerplätze.
- `/benchmark` (POST): Führt das vollfaktorielle Benchmarking durch.
- `/benchmark/export` (GET): Gibt die Ergebnisse des letzten Benchmark-Laufs in CSV-Form aus. Die Ergebnisse werden dazu im Prozess gehalten,

sodass Export und Benchmark-Lauf im selben Serverprozess erfolgen müssen.

Das Frontend entspricht der Model-View-Controller-Architektur (MVC) mit einem AppController-Objekt als Orchestrator. Dieser steuert das WarehouseModel, welches die Datenhaltung des Lagerlayouts verwaltet. Hierzu gehören die Dimensionen, Lagerplätze und Routen des aktuellen Lagers. Der WarehouseRenderer ist eine Canvas-basierte Visualisierung des Lagerlayouts und beinhaltet alle Methoden, die notwendig sind, um dieses auf dem HTML-Canvas zu rendern. Das BenchmarkCharts-Objekt erstellt die statistischen Diagramme, die nach dem erfolgreichen Benchmarking durch die ermittelten Daten im CSV geplottet werden. Hierfür wird das Modul Plotly.js importiert.

Die Backend-Klassen sind nach dem Strategy Pattern aufgebaut. Es wurde eine BaseRoute implementiert, die als abstrakte Basisklasse dient. Alle Algorithmenklassen erben von dieser und implementieren die Methode `compute_route()`. Die Basisklasse stellt zusätzlich die Methode `expand_route()` bereit. Sie expandiert die von `compute_route()` gelieferte Besuchsreihenfolge in eine Folge von Pfadabschnitten. So kann die Route in dem Frontend visualisiert werden. Die Visualisierung liegt außerhalb der gemessenen Berechnungszeit (Abschnitt 4.7). In der Hauptdatei `app.py` werden alle Algorithmen importiert und dem SOLVERS-Dictionary hinzugefügt. Dies ermöglicht eine einfache Erweiterung. Neben den drei verglichenen Verfahren enthält das Repositorium dort eine ältere MILP-Formulierung nach Pansart, Catusse und Cambazard (2018) (`routes/scfs_plus.py`). Sie diente der Entwicklung, ist nicht Gegenstand dieser Arbeit und wurde in keinem der Benchmarks ausgeführt. Auf sie gehen auch die PuLP-Abhängigkeit und ein Teil der Testsuite zurück. Das Backend nutzt folgende Bibliotheken:

- Flask: Für den Server und die REST-API-Integration
- NetworkX: Für das Blossom-Matching (Christofides) und den Euler-Kreis (Fixed-Parameter). Der MST (Prim) und der Hierholzer-Algorithmus sind eigenimplementiert

Die genaue Architektur ist in Abbildung 4.1 zu sehen.

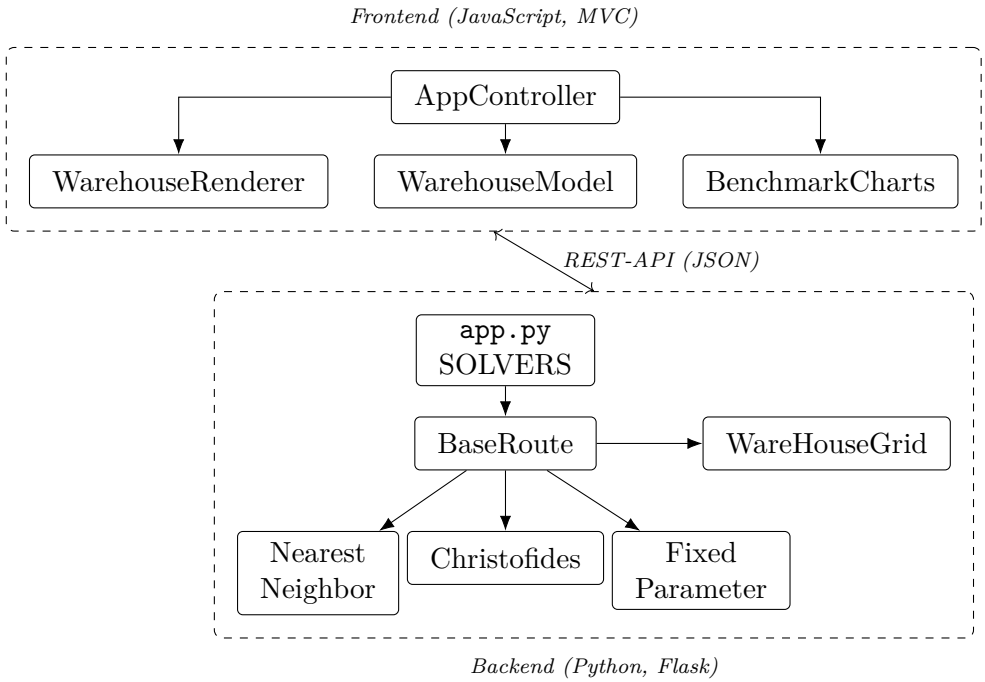


Abbildung 4.1: Aufbau der Webanwendung. Das Frontend folgt dem MVC-Muster, das Backend organisiert die drei Algorithmen nach dem Strategy Pattern über die gemeinsame Basisklasse BaseRoute. Beide Seiten kommunizieren ausschließlich über die REST-API.

4.2 Lagermodellierung

Das Lagersystem wird als 2D-Binärgitter modelliert. Die `WareHouseGrid`-Klasse verwaltet dieses Gitter. Eine 1 markiert begehbare Felder (Gänge, Kreuzungen), eine 0 blockierte Felder (Regale). Es gibt zwei Hauptkonfigurationsparameter für das Lagerlayout. Der Parameter `num_isles` definiert die Anzahl der Regalspalten, `num_rows` die Anzahl der Regalreihen. Jeder Regalblock hat eine feste Größe von 2 Spalten × 6 Zeilen. Dies ergibt pro Lagerregal 12 Lagerplätze, die potenziell besucht werden müssen. Die Gitterdimensionen folgen daraus unmittelbar. Auf jede der n_{isles} Regalspalten entfallen 2 Zellen Breite, dazu kommt je ein vertikaler Gang sowie ein abschließender Gang am Rand. Analog entfallen auf jede Regalreihe 6 Zellen Höhe plus je ein Quergang und ein abschließender Quergang:

$$B = 2\,n_{\text{isles}} + (n_{\text{isles}} + 1) = 3\,n_{\text{isles}} + 1 \tag{4.1a}$$

$$H = 6\,n_{\text{rows}} + (n_{\text{rows}} + 1) = 7\,n_{\text{rows}} + 1 \tag{4.1b}$$

Dabei bezeichnet B die Breite und H die Höhe des Lagerbereichs in Gitterzellen, jeweils einschließlich der Regalzellen. Das erzeugte Gitter ist um drei Zeilen höher als H. Diese zusätzlichen Zeilen nehmen den Packtisch auf. Die Gesamtzahl der Lagerplätze ergibt sich zu $12 \cdot n_{\text{isles}} \cdot n_{\text{rows}}$.

Start- und Endpunkt jeder Kommissionierung ist das Depot. Es liegt für alle drei Algorithmen einheitlich auf der begehbaren Zelle $(0, 0)$, also am linken Ende des unteren Quergangs. Jeder Lagerort hat einen spezifischen Index, der ab 1 indiziert ist. Die Funktion `location_to_coordinate()` berechnet aus dem Index die zugehörige Koordinate wie folgt. Sei $\ell \geq 1$ der Index eines Lagerplatzes. Zunächst werden die Regalnummer s und die Regalreihe z bestimmt:

$$s = \left\lceil \frac{\ell}{12} \right\rceil, \qquad z = \left\lceil \frac{s}{n_{\text{isles}}} \right\rceil \tag{4.2}$$

Daraus ergeben sich die Koordinaten:

$$y = 7\,(z - 1) + 1 + ((\ell - 1) \bmod 6) \tag{4.3a}$$

$$x = \left(\left\lceil \frac{\ell}{6} \right\rceil + s - 1 \right) \bmod 3\,n_{\text{isles}} \tag{4.3b}$$

4.3 Distanzberechnung

In der Applikation kommen zwei Verfahren für die Distanzberechnung zum Einsatz. Das erste ist die Lagerdistanzformel, die für die schnelle Berechnung von Lagerdistanzen in den Algorithmen verwendet wird. Das zweite ist eine geschlossene Pfadkonstruktion, die eine gegebene Besuchsreihenfolge von Lagerorten in Pfadsegmente Expandiert. Dies ermöglicht die Darstellung in dem Frontend als Pfad durch das Lagersystem.

Die Lagerdistanzformel ist in der Methode `calculate_warehouse_distance()` implementiert und berechnet die optimale Distanz zwischen zwei Lagerplätzen unter Berücksichtigung der Gangstruktur in $O(1)$ pro Lagerpaar. Die Funktion berechnet die Distanz durch eine Fallunterscheidung. Die Funktion `location_to_coordinate()` liefert die Koordinate der Regalzelle selbst, nicht die der begehbaren Gangzelle. Da Regale nicht betreten werden, wird jeder Lagerplatz zunächst auf die unmittelbar angrenzende Gangzelle abgebildet. Für eine Regalzelle mit der Spalte x ist das die Spalte $x-1$, wenn $x \bmod 3 = 1$ gilt, und andernfalls die Spalte $x+1$. Die Zeile bleibt unverändert. Das Depot auf $(0,0)$ liegt bereits auf einem Gang und wird von dieser Abbildung ausgenommen. Alle folgenden Distanzen beziehen sich auf diese Gangkoordinaten.

Seien $p_1 = (x_1, y_1)$ und $p_2 = (x_2, y_2)$ die so bestimmten Gangkoordinaten der beiden Lagerplätze. Die zugrunde liegende Fallunterscheidung wurde bereits in Abschnitt 2.2 hergeleitet. Hier wird sie in der konkret implementierten Form angegeben, wobei d wie dort die Manhattan-Distanz bezeichnet:

$$d_{\text{Lager}}(p_1, p_2) = \begin{cases} d(p_1, p_2) & \text{falls } x_1 = x_2 \\ \min \begin{cases} c_u + d\left((x_1,\, y_1 - c_u),\, p_2\right) \\ c_o + d\left((x_1,\, y_1 + c_o),\, p_2\right) \end{cases} & \text{sonst} \end{cases} \tag{4.4}$$

Dabei ist $c_u = y_1 \bmod 7$ die Distanz zum unteren und $c_o = 7 - c_u$ die Distanz zum oberen Quergang. Die Konstante 7 ergibt sich aus dem Reihenabstand des Gitters (siehe Abschnitt 4.2). Mithilfe dieser Formel kann eine vollständige Distanzmatrix zwischen allen Lagerorten in $O(n^2)$ gebildet werden. Die Formel liefert exakte Kürzeste-Wege-Distanzen im Lagergraphen. Solche Metriken erfüllen die Dreiecksungleichung per Konstruktion. Für drei Lagerorte a, b und c ist der kürzeste Weg

von a nach c mindestens so kurz wie jeder andere Weg zwischen beiden, also auch wie derjenige, der über b führt. Dieser hat die Länge $d_{\text{Lager}}(a, b) + d_{\text{Lager}}(b, c)$, woraus $d_{\text{Lager}}(a, c) \leq d_{\text{Lager}}(a, b) + d_{\text{Lager}}(b, c)$ folgt. Dieser Distanzgraph erfüllt damit die Dreiecksungleichung, was dieses TSP zu einem metrischen TSP klassifiziert. Dies ist die Voraussetzung für die Christofides-Gütegarantie (Christofides 1976).

Die geschlossene Pfadkonstruktion liefert keine Distanzen, sondern allein den Pfad für die Routendarstellung im Frontend. Sie erhält die Besuchsreihenfolge der Lagerorte als Eingabe und liefert den diskreten Pfad, der für die `drawRoute`-Funktion im Frontend verwendet wird. Dabei greift sie auf die Fallunterscheidung der Lagerdistanz-Formel zurück und bildet denselben Weg nach, über den Gleichung 4.4 das Minimum bestimmt. In der Startgangspalte zum gewählten Quergang, entlang des Quergangs zur Zielgangspalte, dort zum Ziel. Der gezeichnete Pfad ist deshalb exakt so lang wie die berechnete Distanz. Alle drei Implementierungen nutzen diese Konstruktion. Beim Fixed-Parameter-Algorithmus liegen die Wegpunkte bereits achsenparallel auf den Gängen, sodass nur die Zellen zwischen ihnen aufgefüllt werden (siehe Abschnitt 4.6). Der Schritt erfolgt in `expand_route()` und liegt damit außerhalb der gemessenen Berechnungszeit.

4.4 Nearest-Neighbor-Heuristik

Der erste der implementierten Algorithmen ist die Nearest-Neighbor-Heuristik. Dies ist eine Greedy-Heuristik und von den drei implementierten Verfahren dasjenige mit der kürzesten Implementierung und der niedrigsten Laufzeit.

Der Algorithmus berechnet die kürzeste Route wie folgt. Von dem gegebenen Startknoten (Depot) werden die Entfernungen zu allen unbesuchten Knoten berechnet. Es wird der Knoten mit der geringsten Distanz bestimmt und dieser als Folgeknoten gewählt. Dies wird so lange wiederholt, bis alle Knoten besucht wurden. Abschließend wird zum Depot zurückgekehrt. Die Laufzeit der Nearest-Neighbor-Heuristik ist $O(n^2)$, da in jeder Iteration n Distanzen verglichen werden. Die Nearest-Neighbor-Heuristik hat keine *konstante* Gütegarantie. Für metrische Instanzen wächst das Verhältnis zur optimalen Tour im schlechtesten Fall wie $\Theta(\log n)$ (Rosenkrantz, Stearns und Lewis 1977). Da die Lagerdistanz eine Metrik ist (Abschnitt 4.3), greift hier die logarithmische Schranke. Der Ablauf ist in

Algorithmus 1 dargestellt.

Algorithmus 1 Nearest-Neighbor-Heuristik

Eingabe: Lagerplätze L, Depot v_0
Ausgabe: Besuchsreihenfolge R

1: $R \leftarrow [\, v_0 \,]$
2: $U \leftarrow L$ ▷ noch nicht besuchte Lagerplätze
3: **solange** $U \neq \emptyset$
4: $\quad p \leftarrow \arg\min_{q \in U} d(\text{last}(R),\, q)$
5: $\quad R \leftarrow R \,\|\, [\, p \,]$
6: $\quad U \leftarrow U \setminus \{p\}$
7: $R \leftarrow R \,\|\, [\, v_0 \,]$ ▷ Rückkehr zum Depot
8: **gib zurück** R

Die Distanzen werden mithilfe der Methode `WareHouseGrid.calculate_warehouse_distance()` berechnet. Die Routenlänge ergibt sich durch das Aufsummieren der Lagerdistanzen. Nachdem die Besuchsreihenfolge bestimmt ist, wird diese durch die geschlossene Konstruktion aus Abschnitt 4.3 zu einer darstellbaren Route umgewandelt. Dieser Schritt erfolgt in `expand_route()` und liegt außerhalb der gemessenen Berechnungszeit.

4.5 Christofides-Algorithmus

4.5.1 Verfahren und Gütegarantie

Der Christofides-Algorithmus ist ein Approximationsalgorithmus mit einer Gütegarantie von $\frac{3}{2}$ für ein metrisches TSP (Christofides 1976). Die Voraussetzung hierfür ist, dass die Dreiecksungleichung gilt. Diese ist in dem behandelten Fall erfüllt (siehe Abschnitt 4.3).

Christofides setzt einen vollständigen Graphen mit metrischen Kantengewichten voraus. Der Lagergraph selbst erfüllt das nicht, denn er ist ein Gittergraph, in dem die meisten Lagerorte nicht direkt benachbart sind, und das OPRP verlangt ohnehin nur den Besuch einer Teilmenge seiner Knoten (vgl. Abschnitt 2.4.2). Angewendet wird das Verfahren daher nicht auf den Lagergraphen, sondern auf den Distanzgraphen aus Abschnitt 4.3. Dieser ist vollständig, und seine Gewichte sind

kürzeste Wege im Lager. Ein kürzester Hamiltonkreis in diesem Distanzgraphen entspricht damit genau einer optimalen OPRP-Route, denn die Durchgangsknoten sind in den Kantengewichten bereits enthalten. Erst dadurch gilt die Gütegarantie gegenüber dem OPRP-Optimum und nicht bloß gegenüber einem abstrakten TSP.

Das Verfahren nutzt neben dem minimalen Spannbaum auch das Konzept des Matchings. Zunächst wird ein minimaler Spannbaum des Distanzgraphen berechnet. In diesem existiert eine gerade Anzahl von Knoten mit ungeradem Grad, was aus dem Handschlaglemma folgt (vgl. Abschnitt 2.1.3). Für die Menge dieser Knoten wird ein perfektes Matching mit minimalem Gewicht bestimmt und dem Spannbaum hinzugefügt. In dem so entstehenden Multigraphen hat jeder Knoten einen geraden Grad, weshalb dieser einen Eulerschen Kreis besitzt. Aus diesem wird durch Abkürzen bereits besuchter Knoten die Tour gebildet.

Die Gütegarantie lässt sich in vier Schritten skizzieren. Sei OPT die Länge einer optimalen Tour. Dabei bezeichnet $w(\cdot)$ die Summe der Kantengewichte eines Teilgraphen, gebildet über die Gewichtsfunktion des Distanzgraphen. Erstens gilt $w(T) \leq \mathrm{OPT}$ für den minimalen Spannbaum T. Entfernt man aus einer optimalen Tour eine Kante, so bleibt ein Spannbaum übrig, und der minimale Spannbaum ist höchstens so schwer wie dieser (vgl. Abschnitt 2.1.2). Zweitens gilt $w(M) \leq \mathrm{OPT}/2$ für das minimale perfekte Matching M. Die Knoten mit ungeradem Grad liegen auf der optimalen Tour in einer zyklischen Reihenfolge, und dieser verkürzte Kreis ist wegen der Dreiecksungleichung nicht länger als die optimale Tour. Nimmt man abwechselnd jede zweite seiner Verbindungen, so zerfällt er in zwei perfekte Matchings, deren Gewichte sich zu höchstens OPT addieren. Das billigere von beiden wiegt somit höchstens $\mathrm{OPT}/2$ und das minimale erst recht. Drittens kostet der Eulersche Kreis auf $T + M$ folglich $w(T) + w(M) \leq \frac{3}{2} \cdot \mathrm{OPT}$. Viertens verschlechtert das Abkürzen die Tour nicht, erneut wegen der Dreiecksungleichung. Damit gilt die Schranke auch für die ausgegebene Route. Der vollständige Beweis findet sich bei Christofides (1976).

4.5.2 Implementierung

Der Christofides-Algorithmus geht in den folgenden Schritten vor:

1. Vollständigen Distanzgraphen aufbauen

2. Minimum Spanning Tree berechnen
3. Knoten mit ungeradem Grad identifizieren
4. Minimum Weight Perfect Matching berechnen
5. Matching-Kanten hinzufügen, um den Eulerschen Multigraph zu erhalten
6. Eulerschen Kreis berechnen mit dem Hierholzer-Algorithmus
7. Doppelt besuchte Knoten entfernen, um Hamiltonkreis zu erhalten

Der implementierte Algorithmus ist in Pseudocodeform in Algorithmus 2 zu sehen.

Algorithmus 2 Christofides-Algorithmus

Eingabe: Lagerplätze L, Depot v_0
Ausgabe: Besuchsreihenfolge R

$V \leftarrow L \cup \{v_0\}$
$E \leftarrow \emptyset$
für alle $u, v \in V$ mit $u \neq v$
    $E \leftarrow E \cup \{(d(u,v),\ u,\ v)\}$
$T \leftarrow \textsc{Prim}(V, E)$ ▷ Minimum Spanning Tree
$O \leftarrow \{v \in V : \deg_T(v) \text{ ungerade}\}$
$M \leftarrow \textsc{MinWeightMaxCardMatching}(O)$ ▷ Blossom
$G \leftarrow T \uplus M$ ▷ Multigraph, alle Grade gerade
$C \leftarrow \textsc{Hierholzer}(G,\ v_0)$ ▷ Eulerscher Kreis
$R \leftarrow$ erstes Vorkommen jedes Knotens in reverse(C) ▷ Shortcutting
$R \leftarrow R \,\|\, [\,\text{first}(R)\,]$
gib zurück R

Der minimale Spannbaum wird mit dem Verfahren von Prim aufgebaut. Als Startknoten dient dabei ein beliebiger Knoten und nicht das Depot, was zulässig ist, weil das *Gewicht* des minimalen Spannbaums vom Startknoten unabhängig ist. Bei gleichen Kantengewichten kann der konkrete Baum variieren, was für die Gütegarantie unerheblich ist. Das Depot wird erst später beim Eulerschen Kreis als Startpunkt gesetzt. Veraltete Halde-Einträge werden nicht entfernt, sondern beim Entnehmen verworfen. Nach jeder akzeptierten Kante wird die vollständige Kantenliste erneut nach neuen Kandidaten durchsucht, woraus sich eine Laufzeit von $O(n \cdot |E|)$ statt $O(|E| \log |V|)$ ergibt.

Für das Matching wird die Funktion `min_weight_matching` aus NetworkX verwendet, die auf dem Blossom-Algorithmus von Edmonds (1965) beruht. Sie berechnet ein Matching minimalen Gewichts mit maximaler Kardinalität, indem sie jedes

Gewicht durch $(w_{\max} + 1) - w$ ersetzt und darauf ein Matching maximalen Gewichts bestimmt. Dass daraus das gesuchte perfekte Matching minimalen Gewichts wird, folgt in drei Schritten. Der Graph über der Knotenmenge O ist vollständig, und $|O|$ ist gerade (vgl. Abschnitt 2.1.3). Auf einem vollständigen Graphen mit gerader Knotenzahl ist ein Matching maximaler Kardinalität automatisch perfekt. Alle perfekten Matchings über O besitzen zudem genau $|O|/2$ Kanten, sodass sich der konstante Aufschlag für jedes von ihnen zum selben Wert addiert und die Rangfolge unverändert lässt. Das Maximum der transformierten Gewichte ist damit das Minimum der tatsächlichen. Die Laufzeit des Verfahrens liegt bei $O(n^3)$ (Gabow 1976).[1]

Der Eulersche Kreis wird als Stack-Variante des Hierholzer-Verfahrens berechnet (vgl. Abschnitt 2.1.1). Startknoten ist das Depot, damit die Route dort beginnt. Solange der aktuelle Knoten noch unbenutzte Kanten besitzt, wird eine davon verbraucht und der Nachbarknoten auf den Stack gelegt. Andernfalls wird der Knoten vom Stack in die Ausgabe geschrieben. Das Einhängen der Teilkreise geschieht dadurch implizit über die Stack-Reihenfolge. Dass der Lauf terminiert und alle Kanten verbraucht, folgt aus der Gradparität. Bei jedem Besuch eines Knotens werden zwei Kanten verbraucht, eine zum Hineingehen und eine zum Hinausgehen, sodass bei geradem Grad stets eine Auswegkante zur Verfügung steht. Das Ergebnis entsteht in umgekehrter Reihenfolge und wird abschließend gespiegelt.

Beim Abkürzen überlebt jeweils das erste Vorkommen eines Knotens im umgekehrten Kreis. Welche Vorkommen entfernt werden, beeinflusst die konkrete Route, weshalb diese Richtung festgehalten wird. Dass das Entfernen die Tourlänge nicht verschlechtert, folgt aus der Dreiecksungleichung.

Die Routenlänge wird auf der so entstandenen Besuchsreihenfolge über die Lagerdistanz aus Abschnitt 4.3 berechnet und nicht über den gezeichneten Gitterpfad. Die Expansion in Gitterzellen dient allein der Visualisierung im Frontend und liegt außerhalb der gemessenen Berechnungszeit. Dass die Gap-Werte aller drei Verfahren auf demselben Distanzbegriff beruhen, wird in Abschnitt 4.6.6 gezeigt.

[1] Die Implementierung enthält zusätzlich ein greedy Matching als Alternative. Diese ist über ein Klassenattribut deaktiviert und liefert kein garantiert perfektes Matching. Für alle Messungen dieser Arbeit wurde ausschließlich das Blossom-Matching verwendet.

4.5.3 Laufzeit und Grenzen der Implementierung

Theoretisch liegt der Christofides-Algorithmus bei $O(n^3)$, dominiert durch das Blossom-Matching. Für diese Implementierung trifft das jedoch nicht zu. Den Engpass bildet stattdessen der Aufbau der Kantengewichte. Die zugehörige Schleife läuft über alle n^2 geordneten Paare und berechnet damit jede Distanz zweimal, und der Test auf bereits vorhandene Kanten ist ein linearer Scan über eine Liste, die auf $n(n-1)/2$ Einträge anwächst. Hinzu kommt, dass der minimale Spannbaum mit dem Verfahren von Prim in $O(n \cdot |E|)$ statt in $O(|E| \log |V|)$ bestimmt wird. Daraus ergibt sich die Schranke dieser Phase. Jeder der n^2 Paardurchläufe prüft die bereits gesammelten Kanten linear, und diese Liste wächst auf $n(n-1)/2$ Einträge. Im schlechtesten Fall kostet der Aufbau der Kantengewichte damit $n^2 \cdot n(n-1)/2$ Vergleiche und liegt in $O(n^4)$. Diese Schranke folgt allein aus der hier beschriebenen Implementierung und ist keine Eigenschaft des Christofides-Algorithmus.

Tabelle 4.1 zeigt die Verteilung auf die vier rechenintensiven Phasen. Bei $n = 81$ entfallen rund 83 % der Gesamtlaufzeit auf den Kantenaufbau, während das Blossom-Matching nur etwa 9 % ausmacht. Zwischen $n = 31$ und $n = 81$ wächst der Kantenaufbau mit der dritten bis vierten Potenz der Auftragsgröße und bestätigt damit die obige Schranke, während die Gesamtlaufzeit im gemessenen Bereich von 5 bis 30 Produkten noch von der Grundlast und dem Matching mitbestimmt wird. Dieser Engpass ist reiner Implementierungsaufwand. Er beeinflusst ausschließlich die gemessenen Zeiten, nicht die berechneten Routen. Ein Duplikattest über eine Menge statt über eine Liste oder der Verzicht darauf durch Erzeugung nur ungeordneter Paare würde die Grundlast deutlich senken, ohne ein einziges Ergebnis zu verändern. Die in Kapitel 5 berichteten Berechnungszeiten des Christofides-Algorithmus sind daher als obere Schranke zu lesen.

Tabelle 4.1: Laufzeitanteile der vier rechenintensiven Phasen der Christofides-Implementierung. n bezeichnet die Knotenzahl im Distanzgraphen, also Auftragspositionen zuzüglich Depot. Die Phasenwerte stammen aus getrennt instrumentierten Läufen, die Spalte *gesamt* aus einem uninstrumentierten Lauf derselben Konfiguration (Breite 10, $r = 1$, Seed 42) auf demselben Rechner wie die Benchmarks. Die Zeilensummen weichen deshalb von der Gesamtspalte ab. Die Werte dienen allein der Identifikation der dominierenden Phase.

n	Kantenaufbau	Prim	Matching	Hierholzer	gesamt
11	0,15 ms	0,11 ms	0,90 ms	0,02 ms	0,91 ms
31	3,36 ms	0,72 ms	0,88 ms	0,03 ms	5,44 ms
51	20,78 ms	3,27 ms	4,05 ms	0,05 ms	28,57 ms
81	125,54 ms	12,44 ms	13,36 ms	0,07 ms	151,01 ms

4.6 Fixed-Parameter-Algorithmus

4.6.1 Einordnung und Grundidee

Das dritte Verfahren zur Berechnung einer Route in einem OPRP ist der von Cambazard und Catusse (2018) beschriebene Fixed-Parameter-Algorithmus. Als Ergebnis liefert dieser Algorithmus immer die optimale Route. Daraus folgt laut Definition, dass jede berechnete Route einen Optimalitätsgap von 0 vorweist (der Optimalitätsgap wird in Gleichung 5.1 formal definiert). So kann dieser Algorithmus als Referenz genutzt werden, um die Qualität der anderen Algorithmen zu beurteilen.

Der Algorithmus verfolgt einen dynamisch programmierten Frontieransatz. Auf den konkreten Ansatz wird in den Abschnitten 4.6.3 bis 4.6.6 eingegangen. Die Laufzeit ist so wie in Abschnitt 2.3.1 durch die Funktion $f(h) = h \cdot 7^h$, $c = 1$ gegeben, wobei h die Anzahl der Quergänge bezeichnet und damit die Pfadweite des Gittergraphen parametrisiert (siehe Abschnitt 2.3.2). Diese Schranke geht auf Cambazard und Catusse (2018) zurück. Welcher Wert tatsächlich eingeht, bestimmt der jeweilige Auftrag (vgl. Abschnitt 4.6.7). Die Laufzeit ergibt sich zu $O(n \cdot h \cdot 7^h)$.

Das Verfahren baut systematisch einen Zustandsraum anhand der gültigen Traversierungen auf. Nach jeder Iteration wird geprüft, ob die neu entstandenen Zustände

für eine optimale Route zulässig sind oder ob diese verworfen werden können. Die Frontier-Iteration trennt den Gittergraphen an der aktuellen Position der Frontier. Dies bewirkt, dass alle bereits geplanten Traversierungen links der Frontier nur durch die noch offenen horizontalen Quergänge besucht werden können. Es können keine neuen Wege links der Frontier entstehen und somit bleibt der Zustandsraum links einer gegebenen Frontier beschränkt.

In dem Ansatz von Held und Karp (1962) wächst der Zustandsraum exponentiell mit n, da gespeichert wird, welche Knoten bereits besucht sind. Im Frontier-Ansatz ist das nicht der Fall. Durch das Speichern, wie die Tour die Frontier kreuzt, vereinfacht sich hier die Berechnung substanziell.

4.6.2 Reduktion auf einen Tour-Teilgraphen

Der Fixed-Parameter-Algorithmus nutzt das Prinzip von Teilgraphen bzw. Teilrouten, um diese pro Iteration zu erweitern und daraus eine Gesamtroute und einen Endzustand zu konstruieren.

Im Gegensatz zum Nearest-Neighbor- und Christofides-Algorithmus rechnet der Fixed-Parameter-Algorithmus nicht auf einem Distanzgraphen, sondern auf einem Gittergraphen, der das Lagerlayout 1:1 darstellt. Hierbei stellen Kreuzungen die Knoten dar und Gänge die Kanten. Dabei wirkt die Nebenbedingung des Problems. Auf jeder Kante müssen alle Picks bedient werden. Die Kante selbst muss dafür nicht vollständig durchlaufen werden, sie darf aber nicht ausgelassen werden. Hierbei gilt, dass Knoten und Kanten auch mehrfach besucht werden dürfen. Ob ein Knoten gültig in die Tour eingebunden ist, wird über die Parität seines Grades erfasst. Die Parität sagt aus, ob der Knoten eine gerade Anzahl an Kanten (E, even) oder eine ungerade Anzahl an Kanten (U, uneven) im Teilgraphen aufweist. Als dritter Wert kommt 0 hinzu, wenn der Knoten bisher überhaupt nicht in den Teilgraphen eingebunden ist.

Es wird also primär keine Reihenfolge berechnet, aus der dann eine Route konstruiert wird, sondern ein Zustand einer Route, der alle Nebenbedingungen einhält und eine minimale Route darstellt. Erst wenn dieser Zustand gefunden wurde, wird eine Route daraus konstruiert.

Der Endzustand ergibt sich aus mehreren Teilzuständen, die mit zunehmender Iteration größer werden. Dabei ist folgendes Lemma zur Konstruktion eines gültigen Teilgraphen zu beachten:

Lemma 4.1 (Charakterisierung eines Tour-Teilgraphen)**.** *Ein Teilgraph T ist genau dann ein Tour-Teilgraph, wenn*

(a) alle Picks in T liegen,
(b) T zusammenhängend ist und
(c) jeder Knoten in T einen geraden Grad hat.

Das Lemma geht auf Ratliff und Rosenthal (1983) zurück und wird bei Cambazard und Catusse (2018) als Theorem 1 wiedergegeben. Es ist eine Spezialisierung des Satzes über Eulersche Kreise (siehe Abschnitt 2.1.1).

Aus dem Lemma ergeben sich verschiedene Bedingungen, die bei der Konstruktion eines Teilgraphen einzuhalten sind:

1. Ein Gangsegment mit zu besuchenden Lagerorten darf nicht ausgelassen werden.
2. Ein Knoten muss eine gerade Parität vorweisen, sobald alle mit ihm verbundenen Kanten von der Frontier eingeschlossen sind.
3. Bestehen mehrere nicht verbundene Komponenten in einem Teilgraphen, müssen diese spätestens in der letzten Iteration zu einer Komponente verschmelzen.

Zu beachten ist hier, dass Bedingung 3 nicht aus Bedingung 2 folgt. Als Gegenbeispiel genügt ein Teilgraph, der neben dem übrigen Tourverlauf einen abseits gelegenen, doppelt durchlaufenen Gang enthält. Die Endknoten haben beide Grad 2 und damit eine gerade Parität. Die Komponenten sind aber nicht verbunden und damit ergibt sich ein unzulässiger Teilgraph.

4.6.3 Der Frontier-Zustand

Mit jeder Iteration wird in dem Algorithmus der Frontier-Sweep angewendet, der in Abschnitt 4.6.4 beschrieben wird. Für die Berechnung werden alle Knoten, die sich

auf der aktuellen Sweep-Line befinden, als Zustand geschrieben. Dieser Zustand ist der Frontier-Zustand. Die Sweep-Line verläuft entlang eines senkrechten Gangs und schneidet dabei jeden der h Quergänge genau einmal. An diesen h Kreuzungen wird festgehalten, wie die Route die Sweep-Line überquert. Der Zustand wird wie folgt dargestellt:

$$\omega = ((x_1, \ldots, x_h),\ (c_1, \ldots, c_h)) \tag{4.5}$$

Die Tupel weisen immer die Länge h auf. Dabei bezeichnet x_i die Parität und c_i das Komponentenlabel der i-ten Kreuzung, gezählt von unten nach oben. Für die Parität gilt $x_i \in \{0, U, E\}$ mit der in Abschnitt 4.6.2 eingeführten Bedeutung. Kreuzungen, an denen die Route bisher nicht vorbeikommt, erhalten kein Komponentenlabel, was als $c_i = -$ notiert wird. Abbildung 4.2 zeigt einen solchen Zustand an einem Gittergraphen mit $h = 4$ Quergängen.

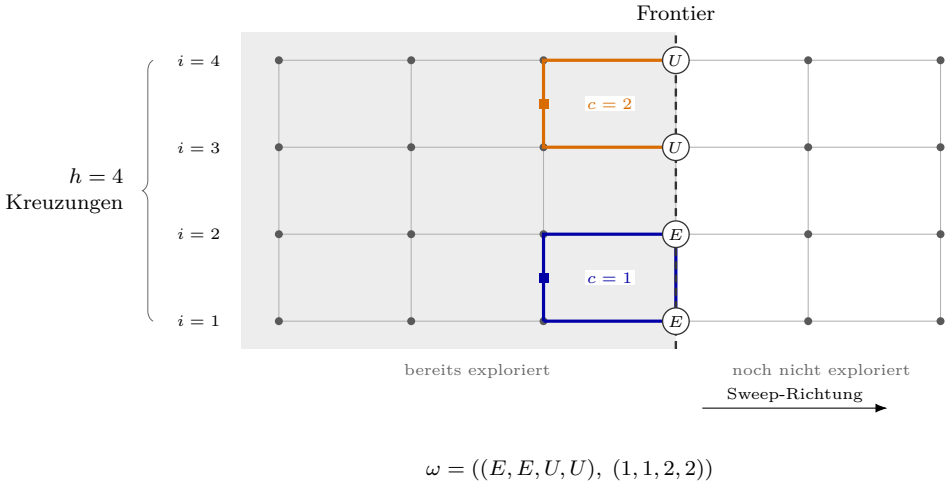


Abbildung 4.2: Momentaufnahme des Frontier-Sweeps auf einem Gittergraphen mit $h = 4$ Quergängen. Links der Frontier sind alle Kanten bereits exploriert, rechts noch keine. Festgehalten wird allein, wie der Tour-Teilgraph die h Kreuzungen der Frontier berührt: die Gradparität (E für gerade, U für ungerade) und die Zugehörigkeit zu einer Zusammenhangskomponente. Komponente 1 (blau) ist ein geschlossener Umlauf, ihre beiden Kreuzungen haben daher gerade Parität. Komponente 2 (orange) ist ein offener Pfad, ihre beiden Kreuzungen haben ungerade Parität und müssen nach rechts fortgesetzt werden. Die kleinen Quadrate auf den senkrechten Kanten sind Picks, die in dem jeweiligen Gangsegment zu bedienen sind. Der Zustand ist damit ein Paar von Tupeln der Länge h und unabhängig von der Lagerbreite und der Zahl der Picks.

Der Zusammenhang der einzelnen Komponenten innerhalb eines Frontier-Zustands ist daran zu erkennen, dass die Kreuzungen dieselbe Zahl als Bezeichner tragen. Im Folgenden sind 2 Zustände zu sehen, die dies anschaulich machen. Es kann also nicht vorkommen, dass 2 Komponenten sich individuell kreuzen. Diese Komponenten würden nach dem Zustand immer miteinander in eine Komponente verschmelzen.

Das folgende Beispiel zeigt für $h = 4$ das Prinzip:

$$\omega_1 = ((E, E, U, U),\ (1, 1, 2, 2)) \tag{4.6a}$$

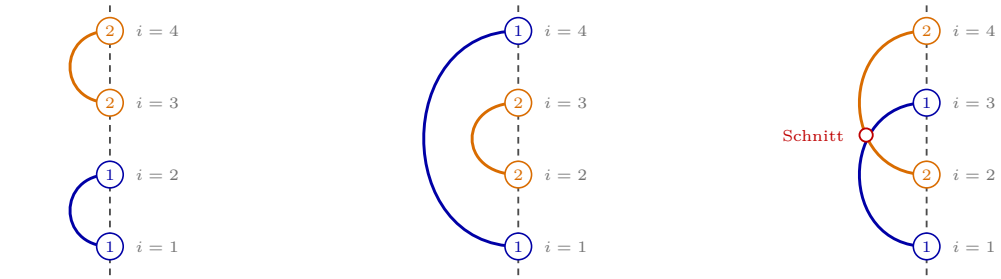
$$\omega_2 = ((E, U, U, E),\ (1, 2, 2, 1)) \tag{4.6b}$$

In ω_1 gehören die beiden unteren Kreuzungen zur einen und die beiden oberen zur anderen Komponente. In ω_2 liegt die zweite Komponente vollständig innerhalb der ersten. Beide Aufteilungen sind zulässig. Nicht auftreten kann dagegen ein Zustand wie $((E, E, E, E),\ (1, 2, 1, 2))$, in dem sich die Komponenten abwechseln. Ein Weg von der ersten zur dritten Kreuzung und ein Weg von der zweiten zur vierten müssten sich in der Ebene schneiden, und im Schnittpunkt wären beide Komponenten zu einer verschmolzen. Diese Eigenschaft wird bei Cambazard und Catusse (2018) als Lemma 4 gezeigt. Abbildung 4.3 stellt die drei Muster gegenüber.

In der Berechnung kann es vorkommen, dass über verschiedene Komponenten-Expansionen ein gleichwertiger Folgezustand entsteht, der sich jedoch in den Bezeichnern unterscheidet. Dies führt zu Ineffizienzen, da jeder Zustand auch als Schlüssel für das Dictionary dient, welches die Zustände einer Iteration verwaltet. Es könnte also dazu kommen, dass ein Zustand doppelt in dem Dictionary gespeichert wird. Um diesen Effekt zu vermeiden, werden die gleichwertigen Zustände normalisiert.

Die Kreuzungen werden in dem Normalisierungsverfahren von unten nach oben durchlaufen und es werden für jeden Zustand die Bezeichner neu verteilt. Der zuerst angetroffene Bezeichner wird zu 1, der nächste noch unbekannte zu 2 und so fort. So haben äquivalente Zustände die exakt gleichen Bezeichner und können als identischer Schlüssel in einem Dictionary verwendet werden.

So ergeben die äquivalenten Zustände $(3, 7, 7, -, 3)$ und $(5, 2, 2, -, 5)$ beide den



(a) getrennt
$(c_i) = (1, 1, 2, 2)$
zulässig

(b) verschachtelt
$(c_i) = (1, 2, 2, 1)$
zulässig

(c) kreuzend
$(c_i) = (1, 2, 1, 2)$
nicht möglich

Abbildung 4.3: Warum die Partition der Komponenten an der Frontier nichtkreuzend ist. Gezeigt sind drei Label-Muster für $h = 4$ Kreuzungen. Die Zahl in der Kreuzung ist ihr Komponentenlabel c_i, die Bögen deuten schematisch an, wie die zugehörigen Teilwege links der Frontier verlaufen. Die Paritäten sind für dieses Argument unerheblich und daher nicht eingetragen (vgl. Abbildung 4.2). (a) und (b) sind zulässig: die Komponenten liegen getrennt übereinander oder ineinander verschachtelt. (c) kann nicht auftreten. Ein Weg von Kreuzung 1 zu Kreuzung 3 und ein Weg von Kreuzung 2 zu Kreuzung 4 müssten sich in der Ebene schneiden, und im Schnittpunkt wären beide zu einer Komponente verschmolzen. Genau diese Einschränkung begrenzt die Zahl der Zustände und führt auf Gleichung 4.8.

normalisierten Zustand $(1, 2, 2, -, 1)$. Die konkreten Zahlen hängen nämlich davon ab, in welcher Reihenfolge Komponenten zuvor angelegt und verschmolzen wurden, nicht von der Aufteilung, die dabei entsteht. Ohne diesen Schritt wären zwei Zustände, die denselben Sachverhalt beschreiben, zwei verschiedene Einträge, und die Dominanzregel könnte sie nicht gegeneinander abwägen. Die Normalisierung setzt damit um, was Cambazard und Catusse (2018) in Abschnitt 2.1 voraussetzen, wenn sie zwei Teilgraphen als gleichwertig behandeln, sobald jede Vervollständigung des einen auch eine Vervollständigung des anderen ist.

4.6.4 Sweep-Reihenfolge und Transitionen

Der Fixed-Parameter-Algorithmus durchläuft den Gittergraphen mit einer Sweep-Line. Dies bedeutet, dass jede Kante des Gittergraphen genau einmal betrachtet wird. Zunächst wird die Spalte am linken Ende betrachtet und es werden alle ihre vertikalen Kanten exploriert. Dann folgen die horizontalen Kanten, die diese Spalte mit der nächsten verbinden. Somit wird der Gittergraph systematisch von links nach rechts durchlaufen. Für ein Gangsegment gibt es nach Ratliff und Rosenthal (1983) insgesamt sechs mögliche Traversierungsarten. Tabelle 4.2 fasst sie mit ihren Kosten zusammen. Abbildung 4.4 zeigt sie geometrisch. Welche davon zulässig sind, hängt von der Anzahl der Picks im Segment ab, sodass jeweils nur eine Teilmenge der sechs Fälle betrachtet werden muss.

Bei einem leeren Gangsegment sind es 3 Möglichkeiten:

1. Den Gang nicht in die Route einplanen
2. Den Gang einmal durchlaufen
3. Den Gang zweimal durchlaufen

Alle weiteren Möglichkeiten würden einer kürzesten Route widersprechen (Ratliff und Rosenthal 1983). Bei einem Gangsegment mit genau einem zu besuchenden Lagerort bestehen genau 4 Möglichkeiten der Traversierung:

1. Den Gang einmal durchlaufen
2. Den Gang zweimal durchlaufen
3. Den Gang von oben bis zum Lagerort laufen und wieder umkehren
4. Den Gang von unten bis zum Lagerort laufen und wieder umkehren

Gegenüber dem Gesamtkatalog entfallen hier zwei Fälle. Das Auslassen ist ausgeschlossen, weil der Lagerort bedient werden muss, und das beidseitige Teilen fällt ohne eine Lücke zwischen zwei Picks mit einem der beiden Stichgänge zusammen.

Bei 2 oder mehreren Lagerorten ergeben sich 5 Möglichkeiten, das Gangsegment zu traversieren. Hier entfällt allein das Auslassen.

1. Den Gang einmal durchlaufen
2. Den Gang zweimal durchlaufen
3. Von unten bis zum obersten Lagerort laufen und wieder umkehren
4. Von oben bis zum untersten Lagerort laufen und wieder umkehren
5. Von unten und oben bis zu dem jeweiligen Lagerort laufen, sodass die größtmögliche Lücke entsteht

Das Vorgehen bei dem letzten Punkt ergibt sich aus der Kostenformel. Diese zeigt im Folgenden sehr anschaulich, warum es bei der größten Lücke zu den größten Einsparungen kommt. Gespart wird genau das doppelte Lückenstück. Die Kosten betragen $2L - 2\gamma$, wobei L die Ganglänge und γ die größte Lücke darstellt. Die Komponenten bleiben bei dieser Strategie getrennt, da der Gang von beiden Seiten aus als eigener Stichgang bedient wird. Sie müssen durch eine später explorierte Kante zu einer Komponente verschmelzen.

Da für jeden Gang endlich viele Traversierungsmöglichkeiten bestehen und diese im Voraus feststehen, können die Gänge einer Spalte nacheinander durchlaufen werden und alle Zustände können pro Iteration erstellt und geprüft werden. Sollte ein Zustand den Pruning-Regeln nicht entsprechen, kann dieser direkt aussortiert werden. So werden zunächst alle vertikalen Gänge der jeweiligen Spalte durchlaufen.

Die horizontalen Gänge lassen deutlich weniger Möglichkeiten zu, da auf ihnen keine zu besuchenden Lagerorte liegen. So bestehen hier nur 3 Möglichkeiten der Traversierung, die abhängig von der Parität der jeweiligen Kreuzung gewählt werden:

1. Keine Traversierung: nur bei Parität 0 oder E
2. Einmalig traversieren: nur bei Parität U
3. Doppelt traversieren: nur bei Parität 0 oder E

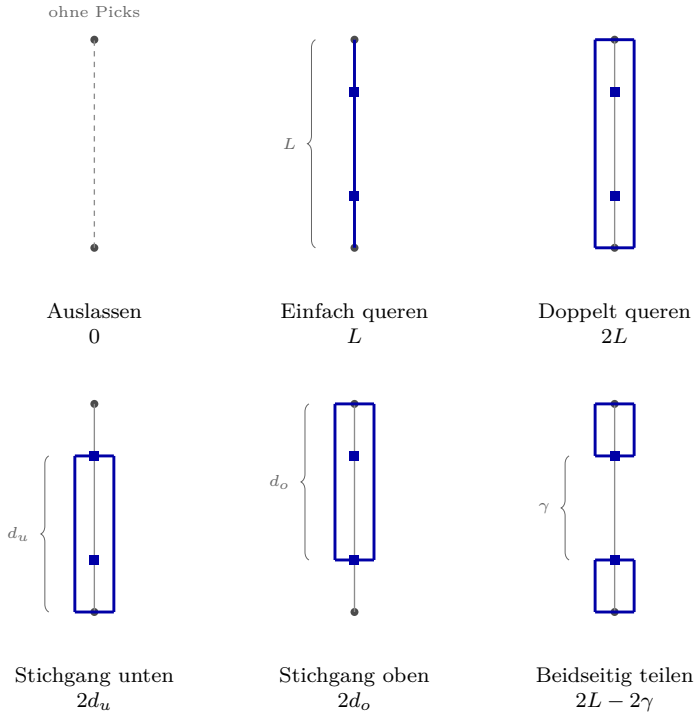


Abbildung 4.4: Die sechs möglichen Traversierungen eines Gangsegments nach Ratliff und Rosenthal (1983), jeweils mit den zugehörigen Kosten. Die grauen Punkte sind die beiden angrenzenden Quergangkreuzungen, die blauen Quadrate die zu bedienenden Picks, die blauen Linien der Wegverlauf. L ist die Länge des Segments, d_u und d_o die Distanz von der unteren bzw. oberen Kreuzung zum jeweils entferntesten Pick und γ die größte Lücke zwischen zwei aufeinanderfolgenden Picks. Welche Traversierungen zulässig sind, hängt von der Anzahl der Picks im Segment ab: Auslassen nur ohne Picks, beidseitiges Teilen erst ab zwei Picks mit einer Lücke dazwischen. Die Wirkung auf Paritäten und Komponenten des Frontier-Zustands ist in Tabelle 4.2 aufgeführt.

Eine horizontale Transition verschiebt den betrachteten Frontier-Slot von der aktuellen Gangspalte in die nächste. Die alte Kreuzung wird abschließend geprüft und durch die Kreuzung der Folgespalte ersetzt. Die Paritätsangaben in Tabelle 4.2 beziehen sich deshalb auf die neue Kreuzung.

Eine horizontale Kante kann also nur dann einfach traversiert werden, wenn die Kreuzung eine ungerade Parität aufweist. Die hinzugefügte Kante setzt die Parität der alten Kreuzung auf gerade und erfüllt damit Bedingung 2 aus Abschnitt 4.6.2. Die Kreuzung der Folgespalte erbt die ungerade Parität. Analog dazu verhalten sich die anderen Optionen. Aus einem gegebenen Zustand sind damit höchstens 2 der 3 Optionen zulässig, während an einem vertikalen Gangsegment bis zu 5 Folgezustände entstehen.

Bei der horizontalen Kantenexploration muss jede Komponente über mindestens eine weitere Kreuzung mit der Frontier verbunden bleiben. Sonst kann diese Komponente nicht weiter verbunden werden und es kann kein zulässiger Tour-Teilgraph mehr entstehen. Alle Zustände, die dies nicht erfüllen, werden verworfen. Dies ist Bedingung 3 aus Abschnitt 4.6.2.

Hier gibt es jedoch einen Sonderfall. Das Depot liegt ebenfalls auf einer Kreuzung mit einer horizontalen Kante. Es muss laut Definition besucht werden und stellt damit eine harte Bedingung für den resultierenden Zustandsraum dar.

Nachdem alle vertikalen Kanten der jeweiligen Spalte traversiert wurden, werden die horizontalen Gänge von unten nach oben traversiert. Auch hier werden nach jeder kombinatorischen Zustandserstellung in jeder Iteration die Zustände verworfen, die den Regeln nicht entsprechen.

Es werden alle Kanten nach diesem Sweep-Vorgehen exploriert, bis zur letzten Spalte.

Tabelle 4.2 fasst die möglichen Traversierungen mit ihren Kosten zusammen. Die Kosten ergeben sich aus den Konstanten des Lagermodells aus Abschnitt 4.2, die für alle Algorithmen identisch sind.

Die Fallunterscheidung der möglichen horizontalen und vertikalen Traversierungen ist vollständig. So kann jede optimale Tour in eine Folge dieser Strategien unterteilt werden. Deswegen verliert die Beschränkung auf neun Strategien keine optimale

Lösung (Ratliff und Rosenthal 1983; Pansart, Catusse und Cambazard 2018).

Strategie	Kosten	Paritäten	Komponenten
Vertikal (Gangsegment, sechs Fälle nach Ratliff und Rosenthal (1983))			
Auslassen	0	unverändert	unverändert
Einfach queren	L	unten +1, oben +1	verschmelzen
Doppelt queren	$2L$	unten +2, oben +2	verschmelzen
Stichgang unten	$2d_u$	unten +2	unten aktivieren
Stichgang oben	$2d_o$	oben +2	oben aktivieren
Beidseitig teilen	$2L - 2\gamma$	unten +2, oben +2	getrennt halten
Horizontal (Quergangkante zur nächsten Spalte)			
Abhängen	0	Kreuzung auf 0	Label freigeben
Einfache Kante	3	bleibt ungerade	unverändert
Doppelte Kante	6	wird gerade	ggf. neu

Tabelle 4.2: Die neun Transitionen des Fixed-Parameter-Algorithmus mit ihren Kosten und ihrer Wirkung auf den Frontier-Zustand. L bezeichnet die Länge des Gangsegments, d_u und d_o die Distanz zum jeweils entferntesten Pick, γ den größten Abstand zwischen zwei aufeinanderfolgenden Picks.

4.6.5 Ablauf und Terminierung

Der Algorithmus beginnt mit der Hauptschleife, die nach dem oben beschriebenen Iterationsmuster durch den Gittergraphen läuft. Im initialen Zustand sind alle Paritäten 0 und keine Kreuzung trägt ein Komponentenlabel. Mit den folgenden Iterationen bauen sich die Zustände auf. Erreichen zwei Wege denselben Zustand, dominiert immer der günstigere. Die Zulässigkeit dieser Regel wird in Abschnitt 4.6.7 begründet. Algorithmus 3 fasst den gesamten Ablauf zusammen.

Nach der letzten Iteration bestehen zulässige und unzulässige Zustände in der Ergebnismenge. Dies ist der Fall, da während des Sweeps bei einigen Zustandsabfolgen nicht frühzeitig zu erkennen ist, ob ein Zustand die Endbedingungen noch erfüllen wird. Ein Aussortieren ist deshalb erst nach der letzten Iteration möglich.

Die unzulässigen Zustände werden über die Verneinung der Paritäts- und der Zusammenhangsbedingung aus Abschnitt 4.6.2 herausgefiltert. Ein Zustand wird verworfen, wenn eine der folgenden Eigenschaften zutrifft:

1. Ein Endzustand hat an einer Kreuzung eine ungerade Parität.
2. Ein Zustand hat mehr als eine aktive Komponente.

Diese beiden Bedingungen allein stellen noch nicht sicher, dass die gefundene Tour das Depot enthält. Das Depot liegt auf dem untersten Quergang und ist damit selbst ein Kreuzungsknoten des Sweep-Graphen. Sein Besuch wird bereits während des Sweeps erzwungen. Bei den horizontalen Transitionen darf die Kreuzung des Depots nicht abgehängt werden, solange sie noch nicht durchlaufen wurde.

Liegen jedoch alle Picks einer Instanz in derselben, äußerst linken Gangspalte, so enthält der beschnittene Sweep-Graph gar keine horizontalen Kanten, und diese Regel greift nicht. Maßgeblich ist dabei nicht die Breite des Lagerlayouts, sondern die Lage der Picks. Da der Sweep-Graph nur bis zum rechtesten Pick aufgebaut wird (Abschnitt 4.6.7), kann dieser Fall auch in einem breiten Lager auftreten. Für ihn prüft der Endfilter zusätzlich, ob die Parität in der Zeile des Depots von null verschieden ist. Ohne diese Prüfung liefert das Verfahren dort eine Rundtour, die das Depot nicht enthält. Die Rekonstruktion setzt sie ab $(0, 0)$ an und erzeugt daraus einen offenen Pfad mit zu kleiner Länge, obwohl Paritäts- und Zusammenhangsbedingung erfüllt sind.

Es wird nicht geprüft ob alle Lagerorte in dem finalem Zustand enthalten sind, da diese aus den bisherigen Transitionen folgt. Ein Gangsegment mit Picks kann gar nicht ausgelassen werden.

Danach bleiben noch zulässige Zustände. Es werden alle Zustände nach den Kosten verglichen. Der Zustand mit den geringsten Kosten wird von dem Algorithmus ausgegeben.

Der Algorithmus terminiert, wenn durch alle Kanten nach dem Sweep-Line-Verfahren iteriert wurde. Solange die Größe des Gittergraphen endlich ist, terminiert somit der Algorithmus garantiert.

Zu zeigen bleibt, dass auch die Menge der mitgeführten Zustände endlich ist. Ein Zustand ist ein Objekt fester Größe. Jede Zustandsvariable hat endlich viele Werte. Die Parität hat drei mögliche Ausprägungen, das Label höchstens $h + 1$. Daraus ergibt sich die endliche Zustandsmenge von höchstens $3^h \cdot (h + 1)^h$ Zuständen. Es ist zu erkennen, dass die Zahl der Picks und auch die Lagerbreite keine Rolle spielen. So ist auch der Zustandsraum endlich und der Algorithmus terminiert

auch in der letzten Iteration garantiert.

Die Timeouts in den Benchmarks sind dem in den Experimenten auf 90 Sekunden gesetzten Timeout-Parameter (Abschnitt 5.1) geschuldet, der im Benchmark-Harness sitzt und nicht im Algorithmus.

Algorithmus 3 Schichtweise dynamische Programmierung über die Kanten des Gittergraphen

Eingabe: Kantenliste E in Sweep-Reihenfolge, Picks P, Depot v_0
Ausgabe: Kürzeste Tour als Wegpunktfolge

$\omega_{\text{init}} \leftarrow ((0,\ldots,0),\ (-,\ldots,-))$ ▷ alle Paritäten 0, kein Label
$\Lambda \leftarrow \{\, \omega_{\text{init}} \mapsto 0 \,\}$ ▷ Schicht als Abbildung Zustand → Kosten
für jede Kante $e \in E$
    $\Lambda' \leftarrow \emptyset$
    für alle $(\omega, c) \in \Lambda$
        für alle in ω zulässigen Strategien s für e
            $(\omega', c') \leftarrow$ Anwendung von s auf (ω, c)
            wenn $c' < \Lambda'(\omega')$ **dann** ▷ Dominanzregel, $\Lambda'(\omega') = \infty$ falls neu
                $\Lambda'(\omega') \leftarrow c'$
                speichere (ω, s) als Vorgänger von ω'
    $\Lambda \leftarrow \Lambda'$
$\Omega_{\text{gültig}} \leftarrow$ alle $\omega \in \Lambda$ mit geraden Paritäten und höchstens einer aktiven Komponente. Liegen alle Picks in einer Gangspalte, zusätzlich v_0 auf der Tour ▷ sonst bereits im Sweep erzwungen
wenn $\Omega_{\text{gültig}} = \emptyset$ **dann**
    gib zurück keine zulässige Tour
$\omega^* \leftarrow \arg\min_{\omega \in \Omega_{\text{gültig}}} \Lambda(\omega)$
gib zurück Rekonstruktion aus ω^* ▷ Abschnitt 4.6.6

4.6.6 Rekonstruktion der Route

Der Algorithmus gibt nach der Terminierung den günstigsten Endzustand zurück. Mit diesem Endzustand allein ist es nicht möglich, eine Route zu konstruieren. Deswegen wurde in dieser Arbeit ein Backtracking implementiert, sodass jede Erweiterung der Route zurückverfolgbar und damit konstruierbar ist.

Für jeden erreichten Zustand wird ein Tupel aus dem Vorgängerzustand und der Erweiterungsstrategie gespeichert. Es steht also nachvollziehbar fest, welche der neun Strategien der Kantenexploration gewählt wurde.

Aus diesen Strategien kann ein neuer Multigraph aufgebaut werden. Im Gegensatz zu einem normalen Graphen ist es in einem Multigraphen möglich, zwei Traversierungen zwischen den gleichen Knoten zu modellieren. Dies ist nötig, da einige Gänge doppelt abgelaufen werden.

Sollte ein Gang einmal traversiert werden, wird auch im Multigraphen dazu eine Kante erstellt. Analog dazu werden zwei Kanten erstellt, wenn der Gang zweimal traversiert worden ist. Bei Stichgängen wird eine Selbstschleife eingefügt, also eine Kante, die von der Kreuzung auf sich selbst zeigt. Als Kantenattribut trägt diese den Umkehrpunkt des Gangs.

So kann der komplette Tour-Teilgraph konstruiert werden. Aus diesem Multigraphen wird dann die Reihenfolge der Traversierung der Kanten bestimmt. Verwendet wird dafür die Euler-Kreis-Routine von NetworkX, die dem in Abschnitt 2.1.1 beschriebenen Verfahren von Hierholzer entspricht. Sie ist hier anwendbar, da der Endfilter aus Abschnitt 4.6.5 die beiden für einen Eulerschen Kreis nötigen Eigenschaften (b) und (c) aus Lemma 4.1 sicherstellt. Eigenschaft (a) folgt bereits aus den Transitionen.

Das Depot wird auf dem Gittergraphen an Punkt $(0, 0)$ simuliert. Dieser Punkt muss besucht werden und agiert wie ein Pflicht-Pick.

Die finale Routenlänge wird durch das Aufsummieren der Einheitsschritte des so entstandenen Gitterpfads ermittelt. Der Fixed-Parameter-Algorithmus verwendet dafür also nicht die Lagerdistanzformel aus Abschnitt 4.3, die bei der Nearest-Neighbor-Heuristik und beim Christofides-Algorithmus zum Einsatz kommt. Beide Größen stimmen überein. Die Wegpunkte des rekonstruierten Pfads liegen ausschließlich auf Gängen, und die Lagerdistanzformel liefert für solche Punkte exakte Kürzeste-Wege-Distanzen. Der Optimalitätsgap vergleicht damit zwei Längen, die auf demselben Distanzbegriff beruhen.

4.6.7 Laufzeit und Grenzen der Implementierung

Die Laufzeit des Algorithmus ergibt sich aus der Anzahl der Kanten des Gittergraphen und der Zahl der Zustände pro Iteration.

Die Kanten des Gittergraphen ergeben sich als $O(h \cdot v)$, wobei h wie bisher die Anzahl der Quergänge und v die Anzahl der vertikalen Gänge bezeichnet, also $v = w+1$ bei w Regalspalten (Abschnitt 2.2). Das Gitter besteht aus v senkrechten und h waagerechten Linien, die sich an $v \cdot h$ Kreuzungen schneiden. Von jeder Kreuzung gehen höchstens vier Kanten ab, nach oben, unten, links und rechts. Die Zahl der Kanten wächst deshalb genauso schnell wie die Zahl der Kreuzungen.

Ein Zustand ist keine einzelne Teillösung, sondern eine Äquivalenzklasse von Teillösungen. Zwei Teilgraphen gehören zur selben Klasse, wenn jede Vervollständigung des einen auch eine Vervollständigung des anderen ist (Cambazard und Catusse 2018). Daraus folgt unmittelbar, dass die Dominanzregel zulässig ist. Von zwei Teillösungen mit demselben Zustand genügt die kostengünstigere, denn die teurere kann in keiner Fortsetzung besser werden.

Die Zahl dieser Klassen hängt allein von h ab. Ein Zustand hält nur zwei Dinge fest, nämlich für jede der h Kreuzungen die Gradparität und die Zugehörigkeit zu einer Zusammenhangskomponente. Beides sind Tupel der Länge h. Was links der Frontier bereits geschehen ist, wird nicht gespeichert, und die Breite des Lagers geht nicht in den Zustand ein. Zusätzlich sind die möglichen Komponentenmuster beschränkt, da die Partition der Komponenten nichtkreuzend ist. Kreuzten sich zwei Wege, so hätten sie einen gemeinsamen Knoten und wären damit dieselbe Komponente (Abschnitt 4.6.3). Genau diese Beschränkung führt auf die Schranke in Gleichung 4.8.

Damit löst sich auch der scheinbare Widerspruch zu den mehreren Folgezuständen je Kante (vgl. Tabelle 4.2). Die Verzweigung multipliziert sich nicht auf, weil jede Transition wieder in einen Zustand derselben Menge führt. Die Iteration bildet die Zustandsmenge auf sich selbst ab und erzeugt keinen wachsenden Baum. Nach jeder Iteration bleiben höchstens so viele Zustände übrig, wie die Menge groß ist, unabhängig davon, wie viele Iterationen noch folgen.

Für den allgemeinen Fall gibt Cambazard und Catusse (2018) die Laufzeit mit $O(n \cdot h \cdot 7^h)$ an. Dort wird der Gittergraph aus den n Punkten der Instanz aufgebaut, sodass seine Größe von n abhängt. Im Lagerlayout dieser Arbeit ist das Gitter dagegen durch die Gangstruktur vorgegeben. Setzt man dessen Kantenzahl in

dieselbe Rechnung ein, ergibt sich (Pansart, Catusse und Cambazard 2018)

$$O(h \cdot v \cdot 7^h). \tag{4.7}$$

In beiden Formulierungen steht 7^h für die Zahl der Zustände pro Iteration und der übrige Faktor für die Zahl der Iterationen. Die Laufzeit ist damit exponentiell allein in h und linear in der Größe des Lagers. Damit ist die Definition des Fixed-Parameter-Algorithmus laut Abschnitt 2.3.1 erfüllt.

Die 7^h Zustände pro Iteration stellen die aufgerundete obere Schranke dar. Die exakte Zahl ist unhandlich und wird von Cambazard und Catusse (2018) angegeben als:

$$|\Omega(h)| = \sum_{k=0}^{h} \binom{h}{k} S_k \tag{4.8}$$

Es wird hier zunächst gezählt, wie viele k von den h Kreuzungen bereits im Teilgraphen liegen. S_k bezeichnet die Super-Catalan-Zahlen und zählt die gültigen Konfigurationen auf diesen k Kreuzungen, also die nichtkreuzende Partition *gemeinsam mit* der zugehörigen Paritätsbelegung (Cambazard und Catusse 2018, Lemma 5–7). Die nichtkreuzende Partition allein wird dagegen von den Catalan-Zahlen gezählt.

Laut Cambazard und Catusse (2018) wächst die Summe asymptotisch wie $(4 + 2\sqrt{2})^h$. Es gilt $4+2\sqrt{2} \approx 6{,}83$. Dieser Wert wird in dieser Arbeit auf 7 aufgerundet und als Schranke gesetzt.

Tabelle 4.3 zeigt klar, dass die Zustandszahl von $h = 7$ auf $h = 9$ um mehr als das Dreißigfache ansteigt. Quantitativ nachprüfbar ist dieser Zusammenhang am Übergang von $h = 7$ auf $h = 8$, also von $r = 6$ auf $r = 7$, weil beide Stufen noch ausnahmslos ohne Timeout terminieren. Die Zustandszahl wächst dort um den Faktor $95.200/16.768 = 5{,}68$, die gemessene mittlere Berechnungszeit um den Faktor 6,86 (Abschnitt 5.3). Der gemessene Faktor bezieht sich dabei auf r-Stufen, die auch Aufträge mit kleinerem effektiven h enthalten. Bei $r = 6$ und bei $r = 7$ erreichen jeweils 101 der 120 Läufe das nominale h. Die beiden Faktoren sind deshalb nur näherungsweise vergleichbar. Ab $h = 9$ ($r = 8$) lässt sich der Vergleich nicht mehr führen, da dort die Mehrzahl der Läufe in den Timeout fällt und die verbleibenden Mittelwerte verzerrt sind.

h	1	2	3	4	5	6	7	8	9
$\lvert\Omega(h)\rvert$	2	6	24	112	568	3.032	16.768	95.200	551.616

Tabelle 4.3: Anzahl der möglichen Frontier-Zustände $|\Omega(h)|$ nach Gleichung 4.8 für $h = 1$ bis $h = 9$.

Die Korrektheit des Algorithmus wird dadurch klar, dass sich jede optimale Route in die neun Strategien aus Abschnitt 4.6.4 zerlegen lässt und somit der Suchraum das Optimum enthält. Das Pruning steht der Korrektheit nicht im Weg, da von zwei Wegen zu demselben Zustand nur der teurere verworfen wird und beide für die Fortsetzung der Tour austauschbar sind (siehe Abschnitt 4.6.3). Der Algorithmus berechnet also garantiert aus dem Suchraum die optimale Lösung für das OPRP.

In dieser Arbeit wurde die lagerspezifische Variante des Algorithmus auf dem Gittergraphen implementiert. Dies ist klar abzugrenzen von den allgemeinen Implementierungen auf dem Hanan-Gitter, also dem in Abschnitt 2.4.2 eingeführten Gitter über der Punktmenge.

Der Parameter des Verfahrens ist nicht durch das Lager bestimmt, sondern durch den Auftrag. Der Sweep-Graph wird deshalb nicht über das gesamte Lagerlayout aufgebaut, sondern nur bis zum obersten und rechtesten Pick der jeweiligen Instanz. Dass dabei keine Lösung verloren geht, folgt aus der Struktur einer optimalen Tour. Diese betritt keinen Quergang oberhalb und keinen vertikalen Gang rechts des äußersten Picks, denn ein solcher Abschnitt müsste zweimal durchlaufen werden und ließe sich streichen, ohne einen Pick zu verlieren. Ein Graph über das volle Layout würde Zustände für Quergänge mitführen, die keine optimale Tour betritt. Da h im Exponenten der Laufzeitschranke steht, stiege der Aufwand mit jedem dieser Quergänge um ein Vielfaches, ohne dass sich die berechnete Route ändert. Die gemessene Zeit würde dann die Größe des Lagerlayouts darstellen aber nicht die Komplexität der Probleminstanz. Es gilt daher $h \leq r + 1$ und $v \leq n_{\text{isles}} + 1$, wie in Abschnitt 2.2 angelegt. Bei einem Layout mit zehn Regalreihen und nur fünf Produkten kann das effektive h deutlich unter 11 liegen. Diese Beschneidung erklärt das Terminierungsverhalten der Messreihen. Welche Instanzen innerhalb der Zeitschranke fertig werden, folgt daraus unmittelbar, und der Zusammenhang wird in Abschnitt 5.3 an den erhobenen Daten geprüft.

Der so aufgespannte Zustandsraum wird von der Implementierung auch ausge-

schöpft. Eine Instrumentierung der dynamischen Programmierung ergibt als größte Schichtgröße genau die Werte aus Tabelle 4.3, also 24 Zustände bei $h = 3$, 112 bei $h = 4$, 568 bei $h = 5$, 3.032 bei $h = 6$ und 16.768 bei $h = 7$. Die tatsächliche Zustandszahl entspricht damit $|\Omega(h)|$ und liegt um Größenordnungen unter der gerundeten Schranke 7^h.

Im Gegensatz zu den zugrunde liegenden Arbeiten wurde kein Preprocessing implementiert. Der implementierte Algorithmus ist bis zu einem effektiven $h = 8$ anwendbar. Läufe mit $h \leq 8$ terminieren ausnahmslos, Läufe mit $h \geq 9$ fallen ausnahmslos in den Timeout. Da h am Auftrag hängt und nicht am Layout, terminieren bei $r = 8$ noch 24 der 120 Läufe, bei $r = 9$ vier und bei $r = 10$ zwei, vergleiche Abschnitt 5.3.

4.7 Benchmark-System

Das implementierte Benchmarking-System ermöglicht es, alle Kombinationen aus Lagerdimensionen, Produktanzahlen, Iterationen und Algorithmen reproduzierbar zu testen. Die Gesamtzahl an Durchläufen ergibt sich durch folgende Formel.

$$R = |C| \cdot |P| \cdot I \cdot |A| \tag{4.9}$$

Dabei ist C die Menge der Lagerkonfigurationen, P die Menge der Produktanzahlen, I die Anzahl der Iterationen je Kombination und A die Menge der getesteten Algorithmen, abzüglich der übersprungenen Kombinationen mit mehr Produkten als Lagerplätzen.

Es können folgende Parameter von dem User gesetzt werden:

- Liste der zu testenden Lagerhausdimensionen (z. B. (3,2), (5,3), (8,4))
- Liste der zu testenden Produktanzahlen (z. B. 5, 10, 15, 20, 25, 30)
- Iterationen pro Kombination
- Optionaler Basis-Seed
- Timeout pro Solver

Durch die Seed-Integration ist es möglich, Testergebnisse reproduzierbar zu erstellen. Der Seed pro Iteration berechnet sich deterministisch aus `base_seed +`

`iteration`. Alle Algorithmen werden pro Testdurchlauf auf den gleichen Konfigurationen getestet. So ist eine Vergleichbarkeit gewährleistet. Jeder Algorithmus läuft in einem separaten Prozess. Bei Überschreitung der vom User definierten Timeout-Zeit wird der Testlauf als Timeout markiert. Dies schützt vor überlangen Berechnungszeiten exakter Verfahren. Zusätzlich begrenzt ein Speicherlimit von 12 GB pro Kindprozess den Arbeitsspeicher. Diese Grenze ist vorsorglich gesetzt, weil das zugrunde liegende Verfahren bei $h \geq 9$ laut Cambazard und Catusse (2018) an Speichergrenzen scheitert. In den vorliegenden Läufen griff jedoch stets zuerst die Zeitschranke, es trat kein einziger Lauf mit dem Status `memory` auf.

Die gemessene Berechnungszeit `computation_time_ms` ist für alle drei Verfahren identisch abgegrenzt. Gemessen wird von der Instanziierung des Solvers bis zu dem Zeitpunkt, an dem das Verfahren eine Besuchsreihenfolge der Lagerorte und deren daraus resultierenden Routenlänge vorliegen hat. Gemessen wird mit `time.perf_counter()`. Diese Methode gibt die Zeit in Sekunden aus, die für die Ausgabe in Millisekunden umgerechnet wird.

Die Erzeugung des Gitters, die für alle Algorithmen identisch ist und je Konfiguration nur einmal anfällt, der Prozess- und Übergabeaufwand des Messrahmens sowie die Expansion der Tour in einzelne Gitterzellen liegen außerhalb der Messung.

Vor jeder Zeitnahme wird das jeweilige Verfahren einmal auf einer Kleinstinstanz ausgeführt und dessen Ergebnis verworfen. Das geschieht für alle drei Verfahren gleichermaßen, und dieser Aufwärmlauf liegt außerhalb der Messung. Er stellt sicher, dass verzögerte Modulimporte und die erstmalige Ausführung der Codepfade abgeschlossen sind, bevor die Uhr startet. Dies ist notwendig, da der Messrahmen für jeden Lauf einen eigenen Prozess startet und diese Initialisierung sonst in jedem Kindprozess erneut anfallen würde. Die Verfahren wären zudem ungleich davon betroffen, weil nur Christofides und der Fixed-Parameter-Algorithmus auf NetworkX zurückgreifen. Somit würde dies unmittelbar die relative Vergleichbarkeit beeinflussen.

Eine Asymmetrie bleibt bewusst bestehen. Der Konstruktor des Fixed-Parameter-Algorithmus baut den beschnittenen Sweep-Graphen auf (Abschnitt 4.6) und leistet damit echte Arbeit innerhalb der Messgrenze, während der Konstruktor der beiden Heuristiken nur Werte zuweist. Die Grenze ist für alle drei Verfahren dieselbe, die dahinter liegende Arbeit nicht. Den Graphaufbau auszuklammern

hieße jedoch, das eigene Verfahren schönzurechnen, denn er ist Bestandteil des Algorithmus und nicht des Messrahmens.

Es werden für die Auswertung folgende Metriken pro Iteration festgehalten:

- Verwendeter Algorithmus (`algorithm`)
- Lagerbreite in Spalten (`num_columns`)
- Anzahl der Quergänge (`num_crossings`)
- Anzahl der Produkte (`num_products`)
- Nummer der Iteration (`iteration`)
- Routenlänge in Gittereinheiten (`route_length`)
- Berechnungszeit in Millisekunden (`computation_time_ms`)
- Verwendeter Seed (`seed`)
- Berechnungsstatus, ok / timeout / memory / error (`status`)

Der Optimalitätsgap ist kein Feld der CSV-Datei, sondern wird nachgelagert aus den Routenlängen berechnet. Er beschreibt die prozentuale Abweichung von der optimalen Route, die der Fixed-Parameter-Algorithmus liefert, und ist daher nur für Instanzen bestimmbar, in denen dieser ohne Timeout terminiert. Die zugehörige Formel steht in Abschnitt 5.2 (Gleichung 5.1). Nach erfolgreichem Durchlauf werden die Ergebnisse als CSV-Datei an das Frontend übergeben, wo diese heruntergeladen werden kann. Die Daten werden im UI direkt geplottet und können analysiert werden.

4.7.1 Validierung und Reproduzierbarkeit

Für die Validierung und Reproduzierbarkeit wurde in dieser Arbeit als Kontrollmechanismus für die optimalen Routen ein unabhängiger Referenzlöser verwendet. Dieser berechnet zunächst per Breitensuche die exakten Gitterdistanzen zwischen allen Lagerorten (metrischer Abschluss) und löst darauf das Problem mit dem exakten Held-Karp-Algorithmus (Held und Karp 1962). Er nutzt damit weder die Distanzformel noch den Graphaufbau der implementierten Verfahren und bestätigt, dass die Lösungen des implementierten Fixed-Parameter-Algorithmus in einer Stichprobe die optimalen Lösungen darstellen. Die Stichprobe umfasst 182 Zufallsinstanzen, bei denen alle Lösungen des Referenzverfahrens den Lösungen des Fixed-Parameter-Algorithmus gleichen. Sie ist über das Skript `verify_optimality.py`

im Code-Repository mit festem Basis-Seed deterministisch reproduzierbar.

Unabhängig davon wurde die Lagerdistanz-Formel selbst geprüft, weil sämtliche Routenlängen und damit alle Optimalitätsgaps anhand dieser berechnet werden. Dafür dienen zwei Referenzen. Die erste ist eine Breitensuche auf dem Gitter, die weder die Formel noch den Graphaufbau der Verfahren benutzt. Verglichen wurden alle rund 1,0 Millionen Lagerortpaare aus sieben Layouts zwischen 1×1 und 10×8, ohne eine einzige Abweichung. Die zweite ist der A*-Algorithmus aus Abschnitt 2.6, der dieselbe Formel suchbasiert prüft. Die zuvor bestehenden Kreuzvergleiche deckten dagegen nur das Layout 3×2 ab. Der Geltungsbereich der Optimalitäts-Stichprobe ist begrenzt. Sie umfasst Layouts mit höchstens 5 Regalspalten, 3 Regalreihen und 10 Produkten, weil der Held-Karp-Referenzlöser exponentiell in der Produktanzahl wächst. Validiert ist damit der Bereich $h \leq 4$, während die Benchmarks in Kapitel 5 bis $h = 11$ reichen. Eine Einschränkung der Unabhängigkeit bleibt zudem bestehen. Der Referenzlöser bildet die Lagerplätze mit derselben Hilfsfunktion auf die angrenzenden Gangzellen ab wie die implementierten Verfahren (Abschnitt 4.3). Ein Fehler in dieser Abbildung wäre für ihn nicht sichtbar. Als schwächere, dafür über den gesamten Parameterbereich reichende Absicherung dient, dass in keinem der 2742 auswertbaren Instanzpaare eine der beiden Heuristiken eine kürzere Route als der Fixed-Parameter-Algorithmus gefunden hat.

Des Weiteren wurde für jede Implementierung eine Testsuite implementiert, die die Funktionalität der Algorithmen sicherstellt. Sie umfasst 126 Tests. Tabelle 4.4 zeigt deren Verteilung auf die Komponenten.

Tabelle 4.4: Verteilung der Testsuite auf die Komponenten der Implementierung, Stand des Commits `2d660ee`.

Komponente	Tests
Fixed-Parameter-Algorithmus	38
Pfadkonstruktion und Expansion der Tour	33
MILP-Variante, nicht Gegenstand dieser Arbeit (Abschnitt 4.1)	21
Messgrenze und Aufwärmlauf des Benchmark-Rahmens	9
Heuristiken und Basisklasse	8
Lagergitter, davon 7 für die Gegenprüfung der Distanzformel	14
A*-Algorithmus	3
Summe	126

Um die Ergebnisse für Dritte reproduzierbar machen zu können und diese von der Softwareumgebung des PCs zu entkoppeln, wurden die Benchmarks in einem Docker-Container berechnet. Das Dockerfile dafür ist bei der Implementierung beigelegt. Die verwendeten Technologien wie Python, Flask und NetworkX wurden aufgrund der Kenntnis des Autors sowie der Zugänglichkeit gewählt. Zusätzlich liefert NetworkX erprobtes Blossom-Matching für die Christofides-Implementierung. Flask ist in dieser Arbeit genutzt worden, da es einen geringen Overhead aufweist und eine leichtgewichtige API bereitstellt. Die Requirements der Implementierung sind im Anhang A.2 dieser Arbeit aufzufinden.

Kapitel 5

Evaluation

5.1 Versuchsaufbau

Für die genaue Analyse der Algorithmen und der Berechnungszeit bezüglich der Eingabegrößen wurden drei systematische Benchmarks ausgeführt. Diese wurden wie folgt konfiguriert:

- Benchmark B1 – **Breite** variieren: In diesem Test wurde der Einfluss der Lagerbreite untersucht. Die Höhe wurde auf $r = 1$ festgesetzt. Dadurch ergeben sich die Konfigurationen: (1,1), (2,1), (3,1), (4,1), (5,1), (6,1), (7,1), (8,1), (9,1), (10,1)
- Benchmark B2 – **Höhe** variieren: Hier wurde die Höhe variiert bei gleichbleibender Breite. Die Konfigurationen ergeben sich wie folgt: (10,1), (10,2), (10,3), (10,4), (10,5), (10,6), (10,7), (10,8), (10,9), (10,10)
- Benchmark B3 – **Quadratische** Variation: Dieser Benchmark misst das Verhalten der Algorithmen gegenüber quadratischen Lagerlayouts. Die Konfigurationen ergeben sich zu: (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8), (9,9), (10,10)

Für alle drei Benchmarks wurden die Produktanzahlen für jedes Layout um 5 Produkte von 5 bis 30 erhöht. Pro Kombination wurde der Test 20-mal wiederholt. Produktanzahlen, welche die Zahl der verfügbaren Lagerorte überschreiten, wurden dabei übersprungen. Aus dem vollen Versuchsplan ergeben sich $3 \cdot 10 \cdot 6 \cdot 20 \cdot 3 = 10.800$ Läufe. Durch die Beschränkung der Lagerkapazität entfallen 600

davon, da die kleinsten Layouts nicht alle Produktstufen aufnehmen können. In Benchmark B1 sind bei der Konfiguration (1,1) nur die Stufen 5 und 10 möglich und bei (2,1) die Stufen 5 bis 20, in Benchmark B3 bei (1,1) ebenfalls nur die Stufen 5 und 10. Ausgewertet wurden somit 10.200 Läufe. Für alle drei Benchmarks beträgt der Basisseed 42. In der Payload-Datei ist nur dieser eine Wert hinterlegt. Die Anwendung leitet daraus die Seeds der 20 Iterationen einer Konfiguration ab, sodass Iteration i den Seed $41 + i$ erhält und sich die Reihe 42 bis 61 ergibt (vgl. Anhang A.1). Es wurde eine harte Timeout-Schranke von 90 Sekunden und ein Speicherlimit von 12 GB pro Kindprozess gewählt. Das Speicherlimit griff in keinem der Läufe. Es wurden alle drei Algorithmen für jeden Benchmark ausgewählt. Es wurden alle Benchmarks innerhalb eines Docker-Containers berechnet. Dies erlaubt ein hohes Maß an Reproduzierbarkeit der Berechnungen. Die Benchmarks wurden auf einem Lenovo Thinkpad ausgeführt. Dieses hatte folgende Spezifikationen:

- CPU: AMD Ryzen 7 PRO 5850U (8 Kerne, 16 Threads)
- RAM: 42 GB
- OS: Ubuntu 24.04.4 LTS (Kernel 6.8.0)
- Python: 3.12.13 (im Docker-Container, vgl. Anhang A.2)

Die Messreihen wurden mithilfe eines Skripts erhoben (vgl. Anhang A.1). Jeder Lauf wird in einem eigenen Prozess gemessen. Mithilfe des Pakets NetworkX wurden die Graphoperationen und das Blossom-Matching implementiert und berechnet. Die Messgrenze und der Aufwärmlauf vor jeder Zeitnahme sind in Abschnitt 4.7 beschrieben.

Die drei Benchmarks sind bewusst orthogonal angelegt. B1 variiert allein die Breite, B2 allein die Höhe und B3 beide gemeinsam. Dadurch lässt sich der Effekt jeder Dimension isoliert bestimmen, statt ihn aus einem gemeinsamen Wachstum zurückrechnen zu müssen. B3 dient anschließend als Kontrolle, ob sich die getrennt gemessenen Effekte im kombinierten Fall wiederfinden.

Drei Konfigurationen sind in jeweils zwei Benchmarks enthalten: (1,1) in B1 und B3, (10,1) in B1 und B2 sowie (10,10) in B2 und B3. Da alle Benchmarks denselben Basisseed verwenden, ergeben sich daraus 840 doppelt gemessene Läufe. Diese stimmen ausnahmslos in Status und Routenlänge überein. Allein die Berechnungszeit weicht ab, im Median um 0,014 bis 0,059 ms. In der Konfiguration (10,10) terminiert der Fixed-Parameter-Algorithmus nur in zwei der 120 Läufe, in beiden

Benchmarks bei denselben zwei Instanzen. Der Timeout fällt hier also unabhängig vom Messrauschen gleich aus, was zu der Erklärung aus Abschnitt 5.3 passt, nach der das effektive h_{eff} der Instanz über die Terminierung entscheidet.

5.2 Einfluss der Lagerbreite

Der Einfluss der Lagerbreite wurde gemessen, indem die Regalreihenzahl auf 1 fixiert wurde. Dabei wurde die Lagerbreite von 1 bis 10 erhöht. Damit gilt für den kompletten Durchlauf $r = 1$, und der Gittergraph hat mit $h = 2$ die kleinstmögliche Pfadweite (vgl. Abschnitt 2.3.2). Da der Fixed-Parameter-Algorithmus exponentiell in h wächst, sollten alle Berechnungen innerhalb dieses Benchmarks performant sein.

Zu erwarten ist, dass der Fixed-Parameter-Algorithmus mit einem fixierten $r = 1$ bei steigender Anzahl der zu besuchenden Lagerorte eine nahezu konstante Laufzeit aufweist. Die Messungen bestätigen das. Die Berechnungszeit bleibt über die Auftragsgrößen hinweg nahezu konstant und wächst mit der Breite annähernd linear, von 0,36 ms bei Breite 1 auf 1,63 ms bei Breite 10 (vgl. Tabelle 5.1). Beides entspricht der Schranke $O(h \cdot v \cdot 7^h)$. Das Verhalten ist linear in der Zahl der senkrechten Gänge und bei festem h unabhängig von der Zahl der Picks.

Für die beiden anderen Verfahren lässt sich eine andere Erwartung formulieren. Beide rechnen nicht auf dem Gittergraphen, sondern auf dem Distanzgraphen, der von der Gangstruktur abstrahiert und nur noch die Auftragspositionen als Knoten kennt (vgl. Abschnitt 2.4.2). Steht dieser Graph erst einmal, ist das Lagerlayout für den weiteren Ablauf ohne Bedeutung. Ihre Berechnungszeit sollte deshalb allein von der Zahl der Picks abhängen und über die Lagerbreite konstant bleiben.

Die Messungen bestätigen diese Erwartung im Kern. Beide Verfahren skalieren deutlich schwächer mit der Breite als der Fixed-Parameter-Algorithmus, ein vollständig konstanter Verlauf zeigt sich aber nicht. Die Nearest-Neighbor-Heuristik berechnet die Routen am schnellsten, ab Breite 2 zeigt der Christofides-Algorithmus die langsamsten Zeiten (vgl. Tabelle 5.1). Über die Breiten 3 bis 10 steigen die mittleren Berechnungszeiten um 9 % bei Nearest Neighbor und um 12 % bei Christofides (vgl. Abbildung 5.1). Erst ab Breite 3 fasst das Lager alle Auftragsgrößen

von 5 bis 30 Produkten, weshalb nur diese Stufen untereinander vergleichbar sind. Die Auftragsgröße wirkt weit stärker. Von 5 auf 30 Produkte steigen dieselben Zeiten um das 9- bis 11-Fache (Nearest Neighbor) beziehungsweise das 24- bis 27-Fache (Christofides). Der Einfluss der Breite ist daneben gering, aber nicht null. Wie sich die beiden Einflussgrößen quantitativ trennen lassen, zeigt Abschnitt 5.6. Dass Christofides insgesamt am langsamsten rechnet, liegt nicht am Blossom-Matching, sondern am Aufbau der Kantengewichte für den minimalen Spannbaum in der hier verwendeten Implementierung (vgl. Abschnitt 4.5.3).

Der Restanstieg der Berechnungszeit über die Breite hat seine Ursache nicht im Ablauf der Verfahren, sondern im Aufbau des Distanzgraphen. Die Lagerdistanzformel aus Abschnitt 4.3 unterscheidet zwei Fälle. Liegen zwei Lagerorte in derselben Gangspalte, genügt eine Manhattan-Distanz. Andernfalls müssen zwei Wege über den unteren und den oberen Quergang berechnet und verglichen werden. Mit wachsender Breite verteilen sich die Picks auf mehr Gangspalten, sodass der Anteil der Paare, die mit dem ersten und billigeren Fall auskommen, von 45,5 % bei Breite 1 auf 8,3 % bei Breite 10 zurückgeht. Die Kosten je Distanzabfrage bleiben konstant, ihr Mittelwert verschiebt sich lediglich zwischen zwei festen Werten. Damit ist der beobachtete Anstieg ein konstanter Faktor und keine Skalierung mit dem Layout. Die Routenlänge wächst über die Breite unabhängig davon, weil die Pickorte auf einer größeren Fläche verteilt liegen. Wie sich dabei die Lösungsqualität verhält, behandeln die folgenden Absätze.

Als Vergleichsgrundlage für die Routenlänge wird der Fixed-Parameter-Algorithmus verwendet, da dieser garantiert die kürzeste Route berechnet (Cambazard und Catusse 2018). Der relative Unterschied zur optimalen Routenlänge des Fixed-Parameter-Algorithmus wird als Optimalitätsgap bezeichnet und wie folgt berechnet:

$$\text{Gap} = \frac{L_{\text{Algorithmus}} - L_{\text{Optimum}}}{L_{\text{Optimum}}} \cdot 100\,\% \tag{5.1}$$

wobei $L_{\text{Algorithmus}}$ die vom jeweiligen Algorithmus berechnete Routenlänge und L_{Optimum} die optimale Routenlänge des Fixed-Parameter-Algorithmus bezeichnet. Der Gap ist nur dort bestimmbar, wo der Fixed-Parameter-Algorithmus innerhalb der Zeitschranke eine Lösung liefert. Hinsichtlich der Lösungsqualität (Abbildung 5.3) liegt der Optimalitätsgap der Nearest-Neighbor-Heuristik im Median bei 2,60 %, der des Christofides-Algorithmus bei 3,17 %. Der Mittelwert zeigt umgekehrte Werte (4,05 % vs. 3,56 %) (vgl. Tabelle 5.7). Instanzweise findet

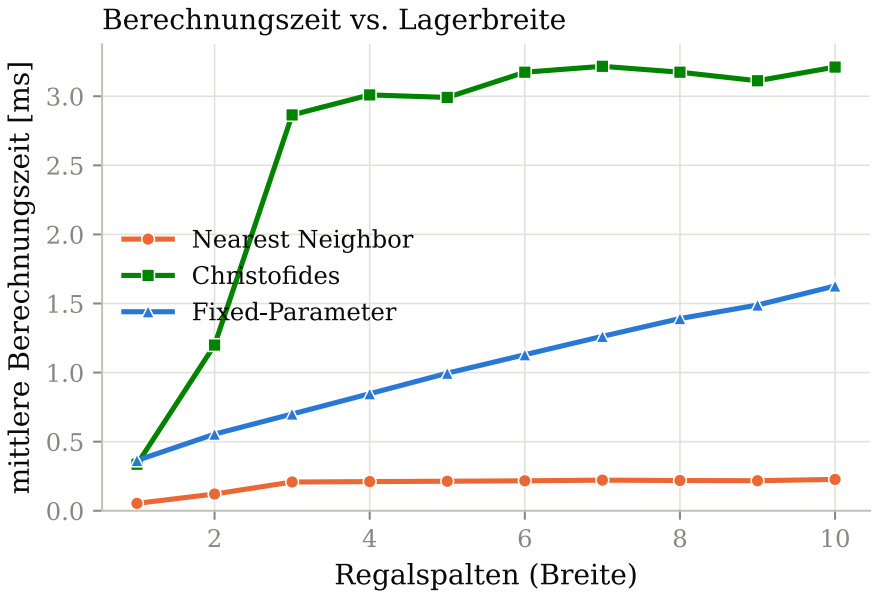


Abbildung 5.1: Mittlere Berechnungszeit je Algorithmus in Abhängigkeit der Lagerbreite bei fester Höhe $r = 1$.

Nearest Neighbor in 31,5 % der Fälle die kürzere Route als Christofides, bezogen auf alle 1080 Instanzen dieses Benchmarks.

Die absoluten Routenlängen wachsen dabei erwartungsgemäß mit der Breite, da mehr Lagerplätze auf größerem Raum verteilt sind (Abbildung 5.2). Über die Breite hinweg verhalten sich die beiden Verfahren unterschiedlich. Der Gap des Christofides-Algorithmus steigt vom Sonderfall Breite 1 auf ein Niveau von rund 4 % und bleibt ab Breite 4 ohne erkennbare Richtung, mit Werten zwischen 3,9 und 4,3 %. Der Gap der Nearest-Neighbor-Heuristik steigt dagegen mit Schwankungen weiter, von 2,16 % bei Breite 3 auf 5,05 % bei Breite 10. Die Rangkorrelation über diese Stufen beträgt 0,71 (vgl. Tabelle A.2 im Anhang). Breite 1 ist deshalb ein Sonderfall, weil dort nur 12 Lagerplätze und damit höchstens 10 Picks zur Verfügung stehen, sodass beide Verfahren nahezu optimal arbeiten. Wie bei den Berechnungszeiten ist der Anstieg von Breite 1 zu Breite 3 dabei teilweise ein Effekt der mitwachsenden Produktanzahl.

Aus diesen Daten ist zu schließen, dass bei flachen Lagern ($r = 1$) der Christofides-Algorithmus die längsten Berechnungszeiten aufweist, in der Lösungsqualität aber mit der Nearest-Neighbor-Heuristik gleichwertig ist. Christofides liegt im Mittel und im Maximum vorn (4,05 gegenüber 3,56 % und 35,71 gegenüber 15,79 %),

Nearest Neighbor im Median (2,60 gegenüber 3,17 %). Der Unterschied ist statistisch nicht signifikant (Wilcoxon, $p = 0{,}068$, vgl. Abschnitt 5.6). Der Vorteil des Christofides-Algorithmus liegt hier nicht im typischen Fall, sondern in der geringeren Streuung. Im Median ist Nearest Neighbor leicht besser, einzelne Ausreißer nach oben ziehen jedoch ihren Mittelwert und ihr Maximum über die Werte des Christofides-Algorithmus. Dieser vermeidet solche Ausreißer, ohne im Regelfall bessere Routen zu liefern.

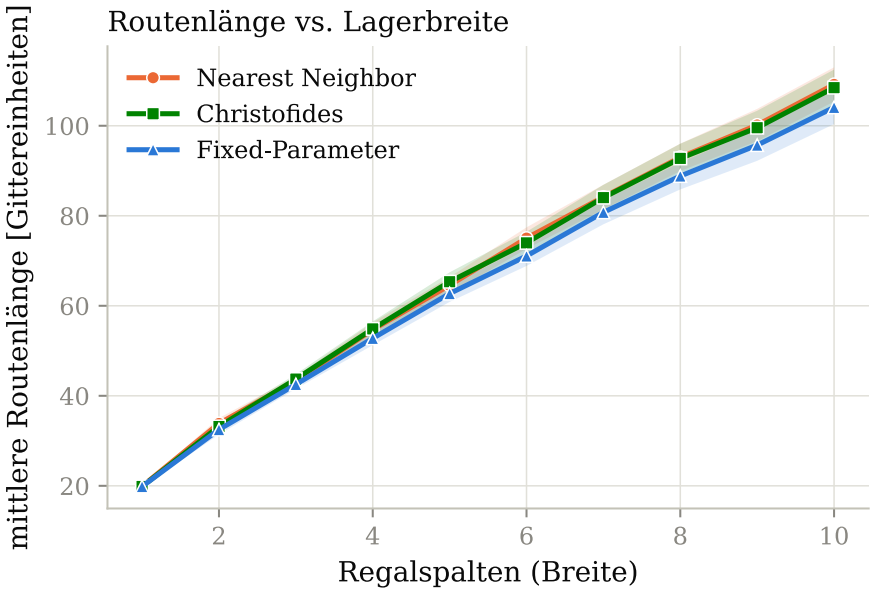


Abbildung 5.2: Mittlere Routenlänge je Algorithmus in Abhängigkeit der Lagerbreite, gemittelt über 20 Iterationen und alle verfügbaren Produktanzahlen. Das schattierte Band zeigt ±1,96 Standardfehler dieser gemischten Stichprobe. Wie bei den Berechnungszeiten mitteln Breite 1 und Breite 2 über weniger Produktstufen, weshalb ihre Punkte nicht unmittelbar mit den übrigen vergleichbar sind.

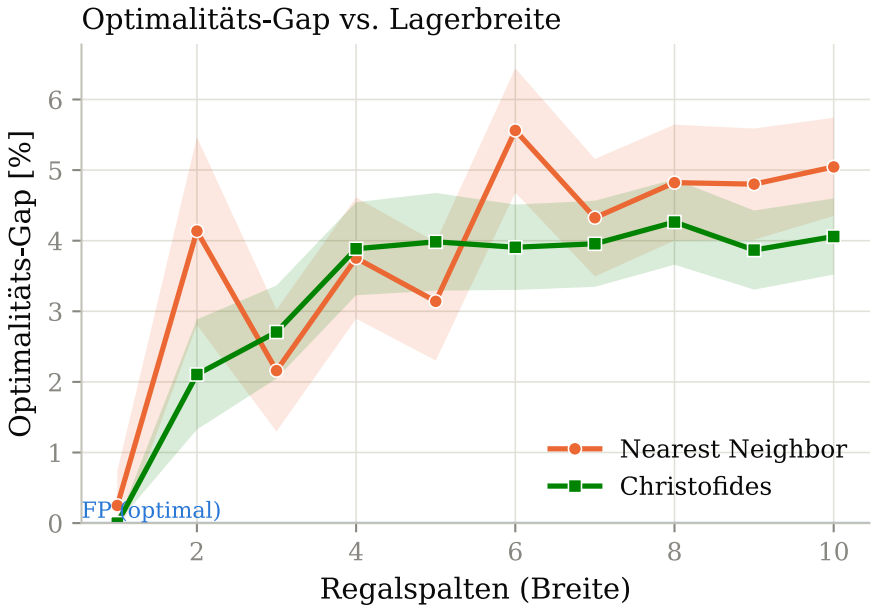


Abbildung 5.3: Optimalitätsgap von Nearest Neighbor und Christofides in Abhängigkeit der Lagerbreite, bezogen auf die exakte Lösung des Fixed-Parameter-Algorithmus.

Breite	Mittlere Berechnungszeit (ms)		
(Spalten)	Nearest Neighbor	Christofides	Fixed-Parameter
1	0,05	0,34	0,36
2	0,12	1,20	0,55
3	0,21	2,86	0,70
4	0,21	3,01	0,85
5	0,21	2,99	1,00
6	0,22	3,17	1,13
7	0,22	3,22	1,26
8	0,22	3,17	1,39
9	0,22	3,11	1,49
10	0,23	3,21	1,63

Tabelle 5.1: Mittlere Berechnungszeit je Algorithmus in Abhängigkeit der Lagerbreite bei fester Höhe $r = 1$, gemittelt über 20 Iterationen und alle Produktanzahlen. Die Lagerkapazität begrenzt die Produktanzahl bei den schmalsten Layouts. Breite 1 mittelt nur über 5 und 10 Produkte, Breite 2 über 5 bis 20, ab Breite 3 über 5 bis 30. Der Anstieg von Breite 1 zu Breite 3 ist daher teilweise ein Effekt der Produktanzahl.

5.3 Einfluss der Regalreihen

Im zweiten Benchmark wird die Anzahl der Regalreihen r systematisch erhöht, während die Breite konstant bleibt. Ein Layout mit r Regalreihen besitzt $h = r + 1$ Quergänge (vgl. Abschnitt 2.3.2). Die maximale Höhe beträgt $r = 10$ Reihen bei einer konstanten Breite von 10 Spalten.

Für den Fixed-Parameter-Algorithmus ist dies eine zentrale Auswertung, da h im Exponenten der Laufzeitschranke steht.

Zu erwarten ist, dass beim Fixed-Parameter-Algorithmus mit steigender Regalhöhe die Berechnungszeiten exponentiell ansteigen, während der Christofides- und der Nearest-Neighbor-Heuristik mit einem steigenden r keine Auffälligkeiten aufweisen sollten.

Beim Betrachten der Daten bestätigt sich die Vermutung des exponentiellen Anstiegs in Abhängigkeit von r (Abbildung 5.4). Es ist den Daten zu entnehmen, dass sich die mittlere Berechnungsdauer ab $r \geq 2$ mit jeder weiteren Regalreihe um das 4- bis 9-Fache vervielfacht (vgl. Tabelle 5.5). Der Faktor ist dabei nicht über alle Stufen gleich und variiert mit dem Inkrementieren von r. Erkennbar ist, dass bereits ab einer Höhe von $r = 2$ der Fixed-Parameter-Algorithmus im Mittel langsamer ist als der Christofides-Algorithmus. Die Timeout-Schranke von 90 Sekunden wird bei $r = 8$ schon in 80 % der Fälle erreicht und bei $r = 10$ in 98,3 % der Fälle. Die Berechnungsdauer ab $r = 8$ erweckt den Eindruck, der Fixed-Parameter-Algorithmus werde schneller. Dies ist nicht der Fall. Erklärbar ist dies durch die Zunahme an Timeouts, die nicht in die Daten aufgenommen werden. So sind bei $r = 9$ nur noch 4 von 120 Durchläufen in dem Datensatz, was nicht repräsentativ ist. Welche Läufe terminieren, entscheidet nicht die Regalreihenzahl des Layouts, sondern die Lage der Pickorte innerhalb des Layouts. Der Sweep-Graph wird nur bis zum obersten und rechtesten Pick einer Instanz aufgebaut (vgl. Abschnitt 4.6), sodass die für die Laufzeit maßgebliche Quergangzahl nicht $r + 1$ beträgt, sondern einen instanzabhängigen Wert $h_{\text{eff}} \leq r + 1$ annimmt. Über die 1200 Fixed-Parameter-Läufe aus Benchmark B2 hinweg terminieren *alle* 870 Läufe mit $h_{\text{eff}} \leq 8$ innerhalb der Zeitschranke, während *keiner* der 330 Läufe mit $h_{\text{eff}} \geq 9$ fertig wird. Die Grenze verläuft damit nicht ungefähr, sondern exakt zwischen $h_{\text{eff}} = 8$ und $h_{\text{eff}} = 9$, passend zum Sprung der Zustandszahl in Tabelle 4.3.

Tabelle 5.2 zeigt, was das auf den Stufen mit Timeouts bedeutet. Bei $r = 10$ hätte das Layout elf Quergänge. Terminiert haben dort ausschließlich Aufträge, deren Picks so tief und so weit links liegen, dass der beschnittene Sweep-Graph auf höchstens sieben Quergänge zusammenschrumpft. Die Timeouts sind umgekehrt genau die Instanzen, bei denen die Beschneidung kaum greift.

Zugleich zeigt sich, dass die Beschneidung im ausgewerteten Bereich $r \leq 7$ selten überhaupt zum Tragen kommt. Dort liegt h_{eff} nur in 6 % der Läufe unter $r + 1$, im Mittel um 0,08 Quergänge. Bei gleichverteilten Pickorten liegt fast immer einer in der obersten Reihe. Die berichteten Ergebnisse hängen also nicht an dieser Eigenschaft. Sichtbar wird sie erst dort, wo der Parameter über die Terminierung entscheidet. Damit bestätigt sich die Fixed-Parameter-Eigenschaft des Verfahrens auch empirisch. Für die Laufzeit ist der Parameter ausschlaggebend und nicht die Größe des Lagers. Timeouts treten dabei ausschließlich ab $r = 8$ auf. Auf den Stufen $r = 1$ bis $r = 7$ terminieren alle 840 Fixed-Parameter-Läufe innerhalb der Zeitschranke (vgl. Tabelle 5.2).

Tabelle 5.2: Effektive Quergangzahl h_{eff} der terminierten und der abgebrochenen Läufe, Benchmark B2. Je Stufe wurden 120 Fixed-Parameter-Läufe ausgewertet. Timeouts treten erst ab $r = 8$ auf.

r	h_{eff} terminiert	h_{eff} Timeout	terminiert (von 120)
1 bis 7	≤ 8	—	120
8	6 bis 8	9	24
9	5 bis 7	9 bis 10	4
10	6 bis 7	9 bis 11	2

Die Nearest-Neighbor-Heuristik und der Christofides-Algorithmus sind von r praktisch unabhängig. Über den gesamten Bereich von $r = 1$ bis $r = 10$ bleiben beide mittleren Zeiten innerhalb ihrer Streuung konstant (vgl. Tabelle 5.5 und Abbildung 5.5). Für den Christofides-Algorithmus ist der Effekt der Regalreihenzahl bei fester Auftragsgröße statistisch nicht nachweisbar (vgl. Abschnitt 5.6). Beim Fixed-Parameter-Algorithmus lässt sich der Vergleich nur bis $r = 7$ führen, weil darüber Timeouts die Mittelwerte verzerren. Dort steigt seine mittlere Berechnungszeit von 1,6 ms bei $r = 1$ auf 58,6 s bei $r = 7$ und damit um das rund 36.000-Fache (vgl. Tabelle 5.5).

Analog zu Benchmark B1 bestätigt sich die Layoutunabhängigkeit auch in dieser

Richtung, wenn die Produktanzahl festgehalten wird und die Regalreihenzahl wächst. Die Zeiten beider Heuristiken streuen über die zehn Stufen ohne erkennbare Richtung. Anders als über die Breite bleibt hier kein Restanstieg, denn die Breite ist auf 10 festgesetzt und damit auch der Anteil der beiden Fälle in der Lagerdistanzformel. Für Nearest Neighbor und Christofides ist somit allein die Auftragsgröße maßgeblich. Beim Optimalitätsgap (Abbildung 5.7) zeigt der Christofides-Algorithmus im Mittel 7,51 % längere Routen als die optimale Lösung, während Nearest Neighbor 11,60 % Abweichung aufweist (vgl. Tabelle 5.7). Die Optimalitätsgap-Angaben müssen in diesem Benchmark-Lauf auf $r \leq 7$ beschränkt werden, da bei größer werdendem r die Ergebnisse von den Timeouts verzerrt würden.

Nach der Produktanzahl aufgeschlüsselt zeigt sich stattdessen ein anderes Bild (Tabelle 5.3). Der Christofides-Algorithmus reagiert deutlich auf die Anzahl der Picks. Zwischen 5 und 30 Produkten steigt die Zeit im Mittel um den Faktor 26 von 0,33 ms auf 8,71 ms. Der Fixed-Parameter-Algorithmus ist dagegen beiden Größen ausgesetzt. Über die Stufen mit vollständiger Terminierung ($r \leq 7$) wächst seine mittlere Zeit von 4,4 auf 12,5 Sekunden, also um den Faktor 2,9. Auf der Stufe $r = 7$ allein steigt sie von 25,2 auf 75,0 Sekunden. Die Auftragsgröße wirkt damit polynomiell und bleibt im Faktor deutlich unter einem Regalreihenschritt (2,98 bei $r = 7$ gegenüber 6,86 für den Schritt von $r = 6$ auf $r = 7$). Absolut fällt sie im Break-even-Bereich dennoch ins Gewicht. Bei $r = 7$ kostet die Spanne von 5 auf 30 Produkte 49,8 Sekunden und damit ebenso viel wie der Schritt von $r = 6$ auf $r = 7$ mit 50,1 Sekunden. Der Schrittfaktor von rund 6,9 je zusätzlicher Regalreihe entspricht der Basis der Zustandsschranke $|\Omega(h)| \approx 6{,}83^{\,h}$: Eine zusätzliche Regalreihe erhöht h_{eff} um eins. Die Laufzeit wächst also exponentiell in h und nicht in r. Die Regalreihenzahl wirkt dabei nur als Treiber von h_{eff}.

Wie weit die Verfahren auseinanderliegen, zeigt der Durchlauf bei $r = 7$, der größten Ausprägung ohne Timeouts. Dort benötigt der Fixed-Parameter-Algorithmus im Mittel 58.613,2 ms gegenüber 0,24 ms bei Nearest Neighbor, also rund das 248.000-Fache. Nearest Neighbor berechnet auf dieser Stufe im Mittel 14,94 % längere Routen.

Die absoluten Routenlängen, auf denen diese Gap-Werte beruhen, zeigt Abbildung 5.6. Die zugehörigen Zahlenwerte sind in Tabelle 5.4 aufgeführt. Sie werden in Abschnitt 6.2 für die Abwägung zwischen gesparter Gehzeit und zusätzlicher

Produktanzahl	Nearest Neighbor	Christofides	Fixed-Parameter
5	0,05 ms	0,33 ms	4.371,4 ms
10	0,09 ms	0,73 ms	7.068,7 ms
15	0,17 ms	1,69 ms	11.151,6 ms
20	0,25 ms	3,04 ms	11.775,0 ms
25	0,36 ms	5,27 ms	12.130,2 ms
30	0,49 ms	8,71 ms	12.541,4 ms

Tabelle 5.3: Mittlere Berechnungszeit je Produktanzahl in Benchmark B2, gemittelt über die Regalreihenzahlen $r \leq 7$, für die der Fixed-Parameter-Algorithmus ausnahmslos ohne Timeout terminiert, und über 20 Iterationen.

Rechenzeit herangezogen.

Es ist final zu erkennen, dass der Fixed-Parameter-Algorithmus bis $r = 7$ zuverlässig terminiert. Ab $r = 8$ scheitern die Berechnungen mehrheitlich durch den Timeout. Ab $r = 8$ ist der Christofides-Algorithmus damit das genaueste noch praktikable Verfahren, und der Trade-off muss für Berechnungszeiten unter 90 Sekunden in Kauf genommen werden. In der Lösungsqualität bleibt der Fixed-Parameter-Algorithmus dort optimal, wo er terminiert.

Der Christofides-Algorithmus schneidet im Mittel klar besser ab, ist der Nearest-Neighbor-Heuristik aber nicht auf jeder einzelnen Instanz überlegen. In 26,5 % der 1200 Instanzen dieses Benchmarks liefert Nearest Neighbor die kürzere Route. Die Überlegenheit gilt also im Durchschnitt und im schlechtesten Fall, nicht instanzweise.

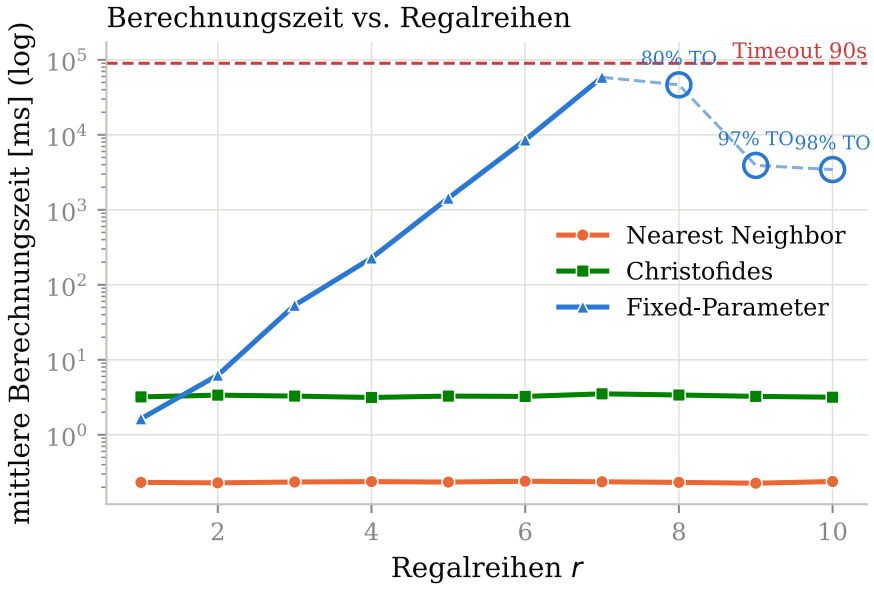


Abbildung 5.4: Mittlere Berechnungszeit (logarithmische Skala) in Abhängigkeit der Anzahl der Regalreihen r bei fester Breite von 10 Spalten. Die waagerechte gestrichelte Linie markiert die Timeout-Schranke von 90 s. Hohle Marker kennzeichnen Stufen mit Timeouts. Der dort eingezeichnete Mittelwert beruht nur auf den terminierten und damit leichtesten Instanzen und liegt deshalb unter dem Wert bei $r = 7$.

r	Nearest Neighbor	Christofides	Fixed-Parameter
1	109,18	108,47	104,08
2	138,75	135,52	126,10
3	159,50	152,42	141,10
4	177,90	170,37	156,92
5	200,25	190,33	175,17
6	213,88	203,87	187,90
7	235,75	223,18	203,05

Tabelle 5.4: Mittlere Routenlänge in Gittereinheiten je Algorithmus in Abhängigkeit der Anzahl der Regalreihen r (Benchmark B2, Breite 10), gemittelt über 20 Iterationen und alle Produktanzahlen 5 bis 30. Angegeben sind nur die Stufen $r \leq 7$, auf denen der Fixed-Parameter-Algorithmus ausnahmslos ohne Timeout terminiert. Darüber wären die Werte durch den Selektionseffekt verzerrt.

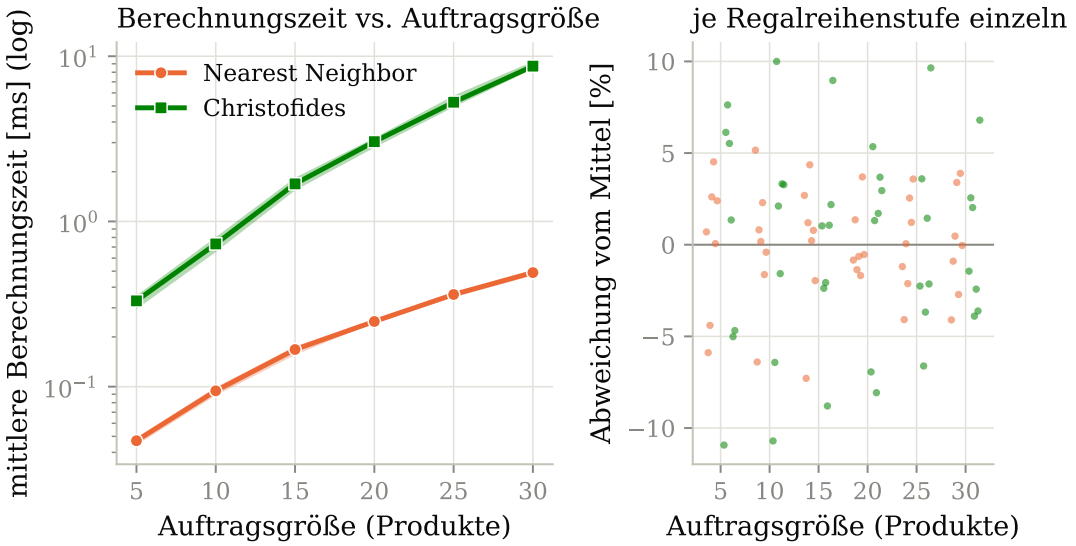


Abbildung 5.5: Mittlere Berechnungszeit in Abhängigkeit der Auftragsgröße für Nearest Neighbor und Christofides, Benchmark B2. Links die absoluten Zeiten auf logarithmischer Skala. Die Linie ist das Mittel über die Regalreihenstufen $r = 1$ bis $r = 7$, das Band deren Spannweite. Rechts dieselben Daten als Abweichung jeder einzelnen Stufe von diesem Mittel. Die Spannweite über die sieben Stufen bleibt an jedem Punkt unter 12 % (Nearest Neighbor) beziehungsweise 21 % (Christofides) und zeigt keine Richtung, während die Zeit über die Auftragsgröße um mehr als eine Größenordnung ansteigt. Die Layoutstufe ist damit für diese beiden Verfahren ohne nennenswerte Wirkung, die Auftragsgröße bestimmt die Berechnungszeit (vgl. Tabelle 5.8). Der Fixed-Parameter-Algorithmus ist nicht dargestellt, da er sich umgekehrt verhält (Abbildung 5.4).

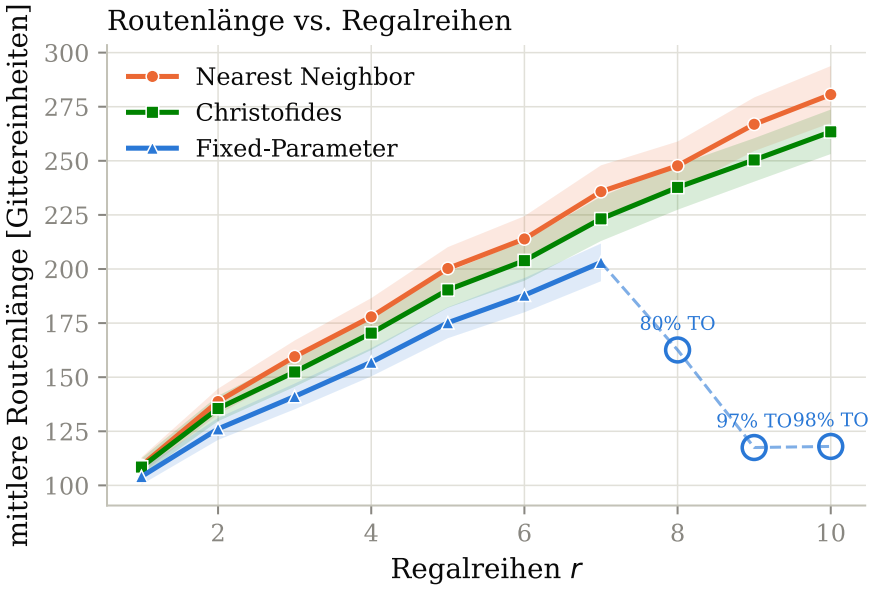


Abbildung 5.6: Mittlere Routenlänge in Abhängigkeit der Anzahl der Regalreihen r, je Algorithmus, gemittelt über 20 Iterationen und alle Produktanzahlen 5 bis 30. Das schattierte Band zeigt ±1,96 Standardfehler dieser gemischten Stichprobe und spiegelt überwiegend die Streuung der Produktanzahl. Die Fixed-Parameter-Werte ab $r = 8$ beruhen nur auf den ohne Timeout terminierten Instanzen (24, 4 und 2 von je 120) und sind nicht mit den übrigen Punkten vergleichbar. Der dortige Rückgang ist ein Selektionseffekt und keine kürzere Route.

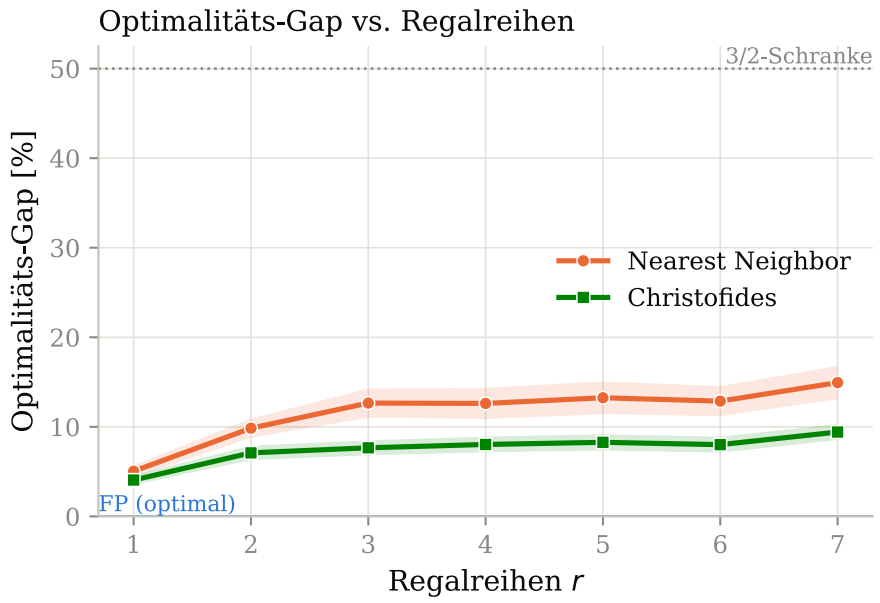


Abbildung 5.7: Optimalitätsgap von Nearest Neighbor und Christofides in Abhängigkeit der Anzahl der Regalreihen r, bezogen auf die exakte Lösung des Fixed-Parameter-Algorithmus (nur Instanzen ohne Timeout).

	Mittlere Berechnungszeit (ms)			FP-Timeouts
r	Nearest Neighbor	Christofides	Fixed-Parameter	(Anteil)
1	0,23	3,20	1,6	0 %
2	0,23	3,38	6,2	0 %
3	0,23	3,28	53,4	0 %
4	0,24	3,15	227,7	0 %
5	0,23	3,28	1 432,3	0 %
6	0,24	3,25	8 542,6	0 %
7	0,24	3,52	58 613,2	0 %
8	0,23	3,39	(46 395,0)	80,0 %
9	0,23	3,25	(3 913,2)	96,7 %
10	0,24	3,17	(3 442,7)	98,3 %

Tabelle 5.5: Mittlere Berechnungszeit je Algorithmus und Timeout-Anteil des Fixed-Parameter-Algorithmus in Abhängigkeit der Regalreihen r (Breite 10, Timeout-Schranke 90 s). Werte in Klammern beruhen nur auf den ohne Timeout abgeschlossenen Instanzen (bei $r = 8/9/10$ nur 24/4/2 von je 120) und sind daher nicht mit den übrigen Zeilen vergleichbar.

5.4 Skalierung bei quadratischen Lagerlayouts

Um im dritten Benchmark das Verhalten der Algorithmen unter quadratischen Lagerlayouts zu analysieren, wurden bei diesem Durchlauf die Lagerbreite und die Regalreihenzahl gleichermaßen erhöht. Der Durchlauf startet bei einer Lagerbreite von 1 und $r = 1$ und reicht bis zu Breite 10 und $r = 10$. In diesem Benchmark entspricht die Lagerbreite also stets der Regalreihenzahl r, sodass r im Folgenden zugleich die Kantenlänge des quadratischen Layouts bezeichnet.

Damit vereint dieser Benchmark die Breiten- und die Höheneffekte der beiden vorherigen Durchläufe in einem gemeinsamen Skalierungspfad.

In den Ergebnissen ist zu erkennen, dass die FP-Zeiten analog zu Benchmark B2 exponentiell ansteigen (vgl. Abbildung 5.8 und Tabelle 5.6). Die Zeiten steigen beim Schritt von $r = 2$ auf $r = 3$ um das 6,3-Fache und ab $r \geq 3$ pro Inkrement um das 6,8- bis 9,1-Fache. Genau wie bei Benchmark B2 gibt es bis $r = 7$ keine Timeouts. Ab $r = 8$ steigt die Timeout-Rate von 81,7 % auf 98,3 %. Die Timeouts setzen in beiden Benchmarks bei derselben Regalreihenzahl $r = 8$ ein, mit Raten von 81,7 % gegenüber 80,0 %, obwohl die Lagerbreite dort 8 gegenüber 10 Spalten beträgt. Der exponentielle Treiber ist die Zahl der Quergänge h, die über $h_{\text{eff}} \leq r+1$ mit der Regalreihenzahl wächst, nicht die Breite. Benchmark B1 bestätigt das im Umkehrschluss. Eine Verzehnfachung der Breite bei $r = 1$ führt zu keinem Timeout und zu einem Anstieg von 0,36 auf 1,63 ms. Über die untereinander vergleichbaren Breitenstufen 3 bis 10 sind es 0,70 auf 1,63 ms, also das 2,3-Fache bei einer Zunahme von 4 auf 11 vertikalen Gängen. Die Breite geht damit wie erwartet höchstens linear ein (Gleichung 4.7) und bleibt gegenüber r um Größenordnungen unkritisch, bleibt aber messbar. Bei $r = 7$ liegt die mittlere Zeit in Benchmark B2 bei Breite 10 um das 1,4-Fache über der in Benchmark B3 bei Breite 7 (58,6 s gegenüber 40,6 s).

Die Ergebnisse des Fixed-Parameter-Algorithmus (Tabelle 5.6) scheinen ab $r = 8$ schneller zu werden. Dahinter steht derselbe Selektionseffekt wie in Benchmark B2 (Abschnitt 5.3). Er verzerrt hier ebenso die mittleren Routenlängen in Abbildung 5.9.

Der Einfluss der Auftragsgröße stellt auch in diesem Benchmark einen Treiber dar. Auf der höchsten Stufe ohne Timeout ($r = 7$) steigt die mittlere Zeit des

Fixed-Parameter-Algorithmus zwischen 5 und 30 Produkten von 24,7 auf 49,6 s, also um den Faktor 2,01. Über die Stufen $r = 2$ bis $r = 7$ sind es 3,9 auf 9,5 s (Faktor 2,43). Die Auftragsgröße wirkt damit wie in Benchmark B2 polynomiell und bleibt deutlich unter einem Regalreihenschritt. Anders als dort erreicht sie absolut jedoch nicht mehr dessen Größenordnung. Die Spanne von 5 auf 30 Produkte kostet bei $r = 7$ 24,9 s, der Schritt von $r = 6$ auf $r = 7$ dagegen 35,4 s. Weil in diesem Benchmark Breite und Regalreihenzahl gemeinsam wachsen, fallen die Schrittfaktoren ab $r = 3$ mit 6,8 bis 9,1 höher aus als die 4,3 bis 6,9 in Benchmark B2, sodass der Layouteinfluss den der Auftragsgröße hier klarer dominiert.

Die Berechnungszeiten der Nearest-Neighbor-Heuristik und des Christofides-Algorithmus wachsen nur schwach mit r. Wie in Abbildung 5.8 zu sehen, verlaufen sie im Vergleich zum Fixed-Parameter-Algorithmus flach und bleiben über alle Stufen im Millisekundenbereich.

Nach der Produktanzahl aufgeschlüsselt zeigen beide Heuristiken dagegen dieselbe Abhängigkeit wie in Benchmark B2. Über die Stufen $r = 2$ bis $r = 7$ steigt die mittlere Zeit der Nearest-Neighbor-Heuristik zwischen 5 und 30 Produkten von 0,045 auf 0,442 ms und damit um den Faktor 9,9, die des Christofides-Algorithmus von 0,311 auf 8,097 ms und damit um den Faktor 26,0. In Benchmark B2 liegen die entsprechenden Faktoren bei 10,4 und 26,3. Bei fester Auftragsgröße von 30 Produkten bleiben die Zeiten beider Heuristiken über alle Layoutstufen dagegen innerhalb eines Faktors von 1,9. Für beide ist die Auftragsgröße somit auch unter quadratischen Layouts die allein maßgebliche Größe.

In der Lösungsqualität zeigt sich auch hier, dass Nearest Neighbor im Mittel 13,66 % über dem Optimum liegt, während Christofides nur 7,93 % aufweist (vgl. Abbildung 5.10 und Tabelle 5.7). Von allen Benchmarks liegt hier die größte Spreizung vor. Der Tabelle 5.7 ist auch zu entnehmen, dass Christofides maximal 24,24 % über dem Optimum liegt. Somit bleibt der Algorithmus deutlich unter der bewiesenen 3/2-Schranke. Die Nearest-Neighbor-Heuristik erreicht ein Maximum von 50,00 % und liegt damit genau auf der Schranke, die für den Christofides-Algorithmus bewiesen ist (vgl. Abschnitt 4.5.1). Der entscheidende Unterschied liegt jedoch nicht in der einzelnen Messung, sondern darin, dass Nearest Neighbor keine konstante Approximationsgüte besitzt. Die bekannte Worst-Case-Schranke wächst mit $O(\log n)$ (Rosenkrantz, Stearns und Lewis 1977) und ist damit für

große Instanzen unbeschränkt. Für Nearest Neighbor ist der gemessene Höchstwert folglich kein Beleg für eine Schranke, sondern gilt nur für die betrachtete Instanzmenge.

Obwohl Nearest Neighbor im Mittel schlechter ist, findet die Heuristik in 19,7 % der Fälle die kürzere Route, bezogen auf alle 1120 Instanzen dieses Benchmarks. Im Schnitt ist die Heuristik in der Lösungsqualität also schlechter, kann aber im Einzelfall bessere Ergebnisse liefern.

Als Fazit dieses Benchmarks bleibt festzuhalten, dass der Fixed-Parameter-Algorithmus bis $r = 7$, also $h_{\text{eff}} \leq 8$, praktisch einsetzbar ist. Ab $r = 8$ weist er eine zu große Berechnungsdauer auf, weswegen der Christofides-Ansatz dort einen geeigneten Kompromiss aus Berechnungsdauer und Routenlänge darstellt (vgl. Abbildung 5.9).

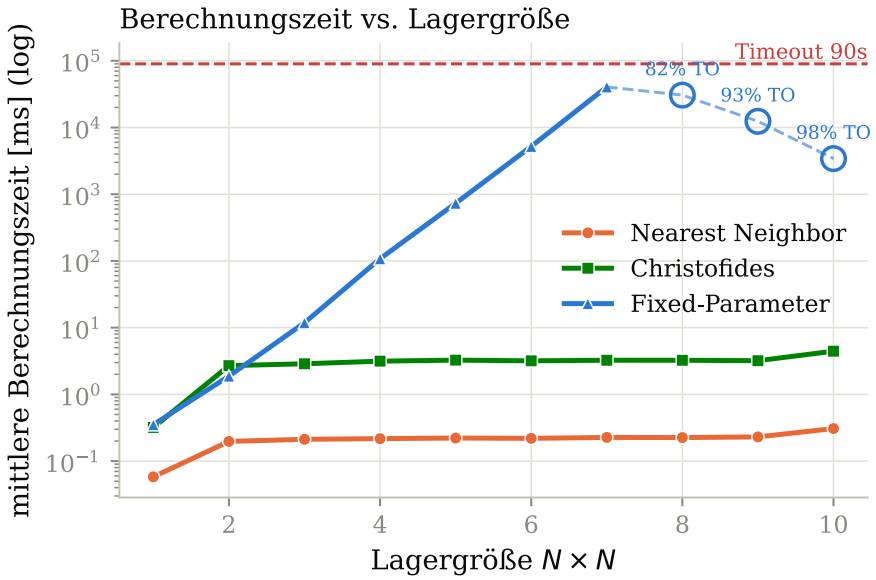


Abbildung 5.8: Mittlere Berechnungszeit (logarithmische Skala) in Abhängigkeit der Lagergröße $r \times r$. Die gestrichelte Linie markiert die Timeout-Schranke von 90 s.

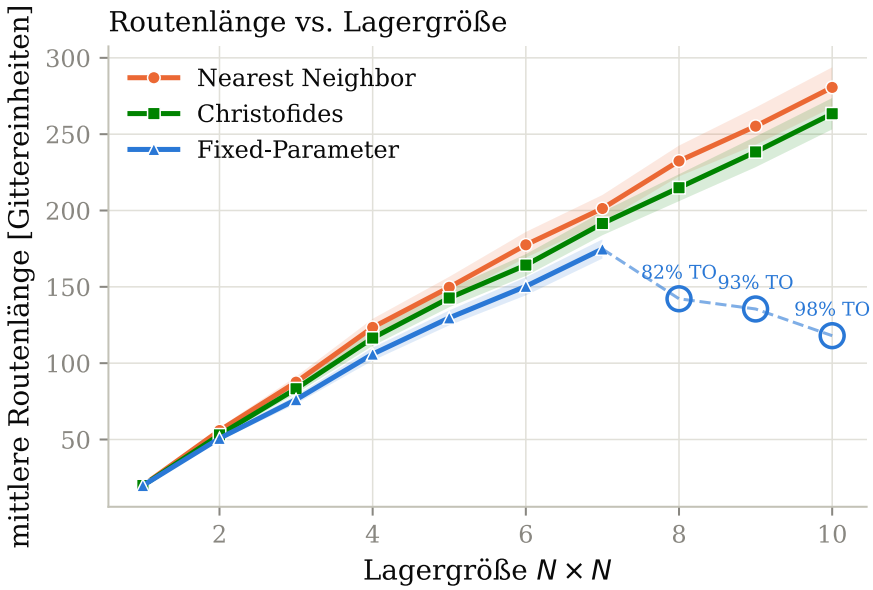


Abbildung 5.9: Mittlere Routenlänge in Abhängigkeit der Lagergröße $r \times r$, je Algorithmus, gemittelt über 20 Iterationen und alle Produktanzahlen 5 bis 30. Das schattierte Band zeigt ±1,96 Standardfehler dieser gemischten Stichprobe und spiegelt überwiegend die Streuung der Produktanzahl. Die Fixed-Parameter-Werte ab $r = 8$ beruhen nur auf den ohne Timeout terminierten Instanzen (22, 8 und 2 von je 120) und sind nicht mit den übrigen Punkten vergleichbar. Der dortige Rückgang ist ein Selektionseffekt und keine kürzere Route.

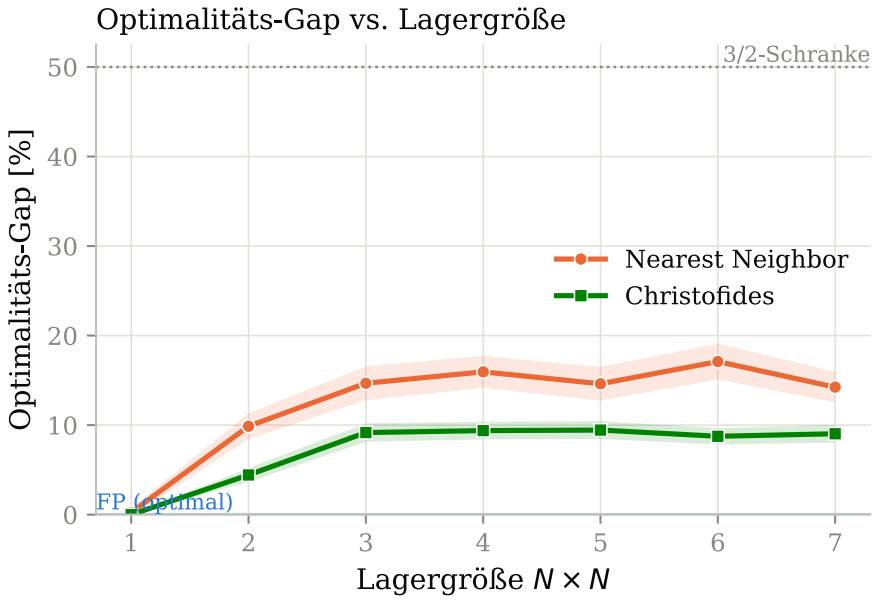


Abbildung 5.10: Optimalitätsgap von Nearest Neighbor und Christofides in Abhängigkeit der Lagergröße $r \times r$, bezogen auf die exakte Lösung des Fixed-Parameter-Algorithmus (nur Instanzen ohne Timeout).

	Mittlere Berechnungszeit (ms)			FP-Timeouts
r	Nearest Neighbor	Christofides	Fixed-Parameter	(Anteil)
1	0,06	0,32	0,4	0 %
2	0,20	2,70	1,9	0 %
3	0,21	2,88	11,9	0 %
4	0,22	3,15	107,9	0 %
5	0,22	3,26	731,7	0 %
6	0,22	3,18	5 203,6	0 %
7	0,23	3,25	40 589,6	0 %
8	0,23	3,25	(30 598,5)	81,7 %
9	0,23	3,20	(12 391,9)	93,3 %
10	0,31	4,44	(3 418,7)	98,3 %

Tabelle 5.6: Mittlere Berechnungszeit je Algorithmus und Timeout-Anteil des Fixed-Parameter-Algorithmus in Abhängigkeit der Lagergröße $r \times r$ (Timeout-Schranke 90 s). Werte in Klammern beruhen nur auf den ohne Timeout abgeschlossenen Instanzen (bei $r = 8/9/10$ nur 22/8/2 von je 120) und sind daher nicht mit den übrigen Zeilen vergleichbar.

5.5 Optimalitätsgap

Die in den vorherigen Abschnitten je Benchmark berichteten Optimalitätsgaps werden im Folgenden zusammengeführt und über alle drei Durchläufe hinweg verglichen. Betrachtet werden dabei nur die Konfigurationsstufen, auf denen der Fixed-Parameter-Algorithmus ausnahmslos ohne Timeout terminiert und damit für jede Instanz eine Referenzlösung vorliegt (vgl. Gleichung 5.1). Das sind 1080 Instanzen in Benchmark B1, 840 in B2 ($r \leq 7$) und 760 in B3 ($r \leq 7$). Die Stufen mit Timeouts entfallen, weil ihre verbleibenden Instanzen systematisch die leichten sind.

Die Werte sind der Tabelle 5.7 zu entnehmen. Es ist gut erkennbar, dass Christofides auf allen Instanzen mit exakter Referenzlösung nie über 25 % hinausgeht und damit deutlich unter der 3/2-Schranke bleibt, die einen Gap von bis zu 50 % zulässt. Nearest Neighbor erreicht hingegen bis zu 50 % und schöpft damit im Maximum genau den Spielraum aus, der für Christofides bewiesen ist. Für die Instanzen ohne Referenzlösung ($r \geq 8$) lässt sich der Gap nicht messen. Die theoretische Garantie des Christofides-Algorithmus gilt dort unabhängig davon. Der Abstand zwischen beiden Verfahren wächst dabei mit der Komplexität des Layouts. In Benchmark B1 trennen sie 0,5 Prozentpunkte, in Benchmark B2 bereits 4,1 und in Benchmark B3 5,7 Prozentpunkte.

Benchmark	Algorithmus	Mittel (%)	Median (%)	Maximum (%)
B1 (Breite)	Nearest Neighbor	4,05	2,60	35,71
	Christofides	3,56	3,17	15,79
B2 (Höhe)	Nearest Neighbor	11,60	9,66	44,36
	Christofides	7,51	7,30	23,53
B3 (quadratisch)	Nearest Neighbor	13,66	12,12	50,00
	Christofides	7,93	8,09	24,24

Tabelle 5.7: Optimalitätsgap von Nearest Neighbor und Christofides über alle drei Benchmarks, bezogen auf die exakte Lösung des Fixed-Parameter-Algorithmus. Berücksichtigt sind jeweils nur die Konfigurationen, für die der Fixed-Parameter-Algorithmus ausnahmslos ohne Timeout terminiert (B1 vollständig, B2 und B3 bis $r = 7$). Das entspricht 1080, 840 und 760 Instanzen. Die Mittel- und Medianwerte sind instanzgewichtet, also über alle Einzelinstanzen der jeweiligen Menge gebildet.

5.6 Statistische Absicherung

Um bei den Ergebnissen dieser Arbeit sicherstellen zu können, dass die Messpunkte auf die Algorithmen zurückzuführen sind und nicht auf den experimentellen Aufbau, wurden verschiedene Vorkehrungen getroffen.

Die einzige variierende Komponente innerhalb einer Konfiguration des Versuchsaufbaus stellt die Auswahl der Lagerorte dar. Das Lagerlayout sowie die Anzahl der Picks ist mit jedem Benchmark klar definiert. Hingegen ist die Verteilung der Picks zufällig. Für die Routenlängen misst die Standardabweichung daher allein, wie ein gegebener Algorithmus auf die Zusammensetzung dieser Picks reagiert. Für die Berechnungszeiten kommt Systemrauschen hinzu. Jede Wiederholung läuft auf einer anderen Pickliste und zu einem anderen Zeitpunkt. Beide Ursachen wirken damit gleichzeitig und lassen sich in diesem Aufbau nicht isolieren. Diese gehen gemeinsam in die Mittelung über die 20 Wiederholungen ein.

Ein Beispiel hierfür ist in Benchmark B2 gegeben. Bei $r = 5$ und 20 Produkten liefert die Nearest-Neighbor-Heuristik über die 20 Iterationen Routenlängen zwischen 182 und 272. Dies entspricht einem Unterschied von rund 50 % zwischen kürzester und längster Route bei identischen Faktoren. Der Variationskoeffizient ist berechenbar, indem die Standardabweichung durch den Mittelwert dividiert wird. Dieser ist dimensionslos und ermöglicht es, die Abweichungen von verschiedenen Konfigurationen untereinander zu vergleichen. So kann das Verhalten der Algorithmen über verschiedene Konfigurationen hinweg analysiert und verglichen werden. Im obigen Beispiel beträgt er bei einer Standardabweichung von 23,7 und einem Mittelwert von 222,6 rund 10,6 %.

Als zentrale Vorkehrung wurde jede Kombination aus Lagerlayout und Produktanzahl 20-mal mit variierendem Seed wiederholt. Berichtet werden Zeit- und Längenmittel über diese Wiederholungen. Die Gap-Kennzahlen (Mittel, Median, Maximum) und der gepaarte Vergleich werden dagegen instanzweise über alle Wiederholungen und Produktanzahlen gebildet.

Um den Vergleich zwischen Nearest Neighbor und Christofides abzusichern, wurde die Differenz der Optimalitätsgaps je Instanz gepaart ausgewertet. In Benchmark B1 beträgt der mittlere Vorsprung des Christofides-Algorithmus 0,49 Prozentpunkte. Die 20 Iterationen einer Konfiguration sind dabei nicht unabhängig

voneinander, weshalb das Konfidenzintervall auf Konfigurationsebene gebildet wird. Über die 54 Kombinationen aus Lagerbreite und Produktanzahl ergibt sich ein 95 %-Konfidenzintervall von $[-0{,}05;\ 1{,}04]$, das die Null enthält. Ein zweiseitiger Wilcoxon-Vorzeichen-Rang-Test über die Einzelinstanzen bestätigt dies mit $p = 0{,}068$. Nulldifferenzen wurden dabei verworfen, was hier ins Gewicht fällt, weil 386 der 1080 Differenzen exakt null sind. Routenlängen sind ganzzahlige Gittereinheiten, weshalb identische Ergebnisse häufig auftreten. Instanzweise gewinnt Christofides 354-mal, Nearest Neighbor 340-mal. Der Vorsprung ist damit bei flachen Lagern statistisch nicht belastbar. Die Verfahren sind bei $r = 1$ praktisch nicht zu unterscheiden. In den Benchmarks B2 und B3 fällt der Vorsprung mit 4,10 Prozentpunkten (840 Instanzen, $r \leq 7$) und 5,73 Prozentpunkten (760 Instanzen, $r \leq 7$) deutlich größer und signifikant aus ($p < 10^{-30}$). Da drei Vergleiche geführt werden, wäre eine Korrektur für multiples Testen angebracht. Sie ändert das Bild nicht, da beide Werte auch nach Bonferroni-Korrektur signifikant bleiben.

Auftragsgröße und Lagerdimension variieren im Versuchsplan gemeinsam. Für die Aussagen der vorangegangenen Abschnitte ist deshalb zu trennen, welcher Anteil der beobachteten Unterschiede auf welche der beiden Größen zurückgeht. Tabelle 5.8 gibt an, welchen Anteil der Gesamtvarianz jede von ihnen erklärt, getrennt für die Berechnungszeit und für den Optimalitätsgap. Ausgewertet werden nur die Layoutstufen, die alle Produktanzahlen von 5 bis 30 aufnehmen und auf denen der Fixed-Parameter-Algorithmus ausnahmslos terminiert, damit die Kappung an der Lagerkapazität die Zuordnung nicht verzerrt.

Für die Berechnungszeit ist das Ergebnis eindeutig. Bei der Nearest-Neighbor-Heuristik und beim Christofides-Algorithmus entfallen 94 bis 96 % der Varianz auf die Auftragsgröße und höchstens 0,6 % auf das Layout. Ein Kruskal-Wallis-Test bestätigt den Effekt der Auftragsgröße auf jeder einzelnen Layoutstufe ($p < 10^{-21}$). Umgekehrt ist der Effekt des Layouts bei fester Auftragsgröße beim Christofides-Algorithmus in Benchmark B2 in keiner der sechs Produktstufen nachweisbar und in Benchmark B1 in zwei von sechs, bei der Nearest-Neighbor-Heuristik dagegen in vier beziehungsweise fünf von sechs (jeweils $p < 0{,}05$). Letzteres ist der in Abschnitt 5.2 beschriebene Restanstieg über die Breite, der aus der Fallunterscheidung der Lagerdistanzformel folgt und nicht aus dem Ablauf des Verfahrens. Anschaulich wird das Verhältnis der beiden Größen an den Christofides-Zeiten aus Benchmark B2. Bei fester Auftragsgröße von 30 Produkten liegen

alle Stufen von $r = 1$ bis $r = 7$ zwischen 8,37 und 9,30 ms. Wird dagegen die Regalreihenzahl fixiert, steigt dieselbe Zeit zwischen 5 und 30 Produkten um das 26-Fache. Abbildung 5.5 zeigt denselben Sachverhalt grafisch. Beim Fixed-Parameter-Algorithmus kehrt sich das Bild um. Auf den Stufen ohne Timeout ($r \leq 7$) erklärt die Regalreihenzahl 96,2 % der Varianz der logarithmierten Berechnungszeit und die Auftragsgröße 1,0 %.

Für den Optimalitätsgap gilt diese klare Trennung nicht. Dort wirken beide Größen in derselben Ordnung, und in den Benchmarks B2 und B3 erklärt die Auftragsgröße sogar mehr als das Layout (Tabelle 5.8). Die Rangkorrelation zwischen Auftragsgröße und Gap beträgt in B2 und B3 zwischen 0,33 und 0,37 ($p < 10^{-18}$), die zwischen Layoutstufe und Gap zwischen 0,14 und 0,27. Zugleich bleiben mehr als zwei Drittel der Gap-Varianz unerklärt und entfallen damit auf die konkrete Pickliste. Die Lösungsqualität ist also deutlich schlechter vorhersagbar als die Berechnungszeit und die Auftragsgröße ist dafür ein eigenständiger Treiber. In Benchmark B2 wächst der mittlere Gap der Nearest-Neighbor-Heuristik bei $r = 7$ von 5,33 % bei fünf Produkten auf 23,28 % bei dreißig, der des Christofides-Algorithmus von 4,44 auf 10,35 %. Für die Berechnungszeit ist die Layoutabhängigkeit damit auf den Fixed-Parameter-Algorithmus beschränkt, für die Lösungsqualität gilt sie für alle drei Verfahren.

		Berechnungszeit (%)		Optimalitätsgap (%)	
Benchmark	Algorithmus	Produkte	Layout	Produkte	Layout
B1 (Breite)	Nearest Neighbor	94,6	0,1	5,3	5,0
	Christofides	95,7	0,2	6,5	1,6
B2 (Höhe)	Nearest Neighbor	96,3	0,0	11,2	10,8
	Christofides	94,1	0,1	16,1	10,1
B3 (quadratisch)	Nearest Neighbor	95,1	0,4	13,9	4,8
	Christofides	94,4	0,6	14,4	10,2

Tabelle 5.8: Anteil der Gesamtvarianz in Prozent, den die Auftragsgröße (Spalte *Produkte*) und die Layoutstufe (Spalte *Layout*) erklären, je Benchmark und Algorithmus. Der nicht ausgewiesene Rest entfällt auf die Streuung zwischen den Instanzen einer Konfiguration, also auf die konkrete Pickliste. Berücksichtigt sind nur Layoutstufen, die alle Produktanzahlen von 5 bis 30 aufnehmen und auf denen der Fixed-Parameter-Algorithmus ausnahmslos terminiert (B1 Breite 3 bis 10, B2 $r \leq 7$, B3 $r = 2$ bis 7). Das entspricht 960, 840 und 720 Instanzen je Algorithmus.

Der Vergleich zwischen den Algorithmen ist fair und möglich, da jeder Algorithmus für dieselbe Instanz eine Lösung berechnen muss. Zwar ist die Auswahl der Lagerorte zufällig, sie wird jedoch durch einen Seed gesetzt. Die Probleme sind damit für jeden Algorithmus identisch und reproduzierbar (vgl. Abschnitt 5.1).

Kapitel 6

Diskussion

6.1 Beantwortung der Forschungsfrage

In dieser Arbeit wurde empirisch untersucht, wie sich ein heuristisches, ein approximatives und ein parametrisiertes Verfahren für das Order Picking Routing Problem in rechteckigen Lagerlayouts unterscheiden. Die eingangs gestellte Leitfrage (Abschnitt 1.3) richtet sich dabei auf die Unterschiede der drei Verfahren in Lösungsqualität und Laufzeit bei steigender Problemgröße und variierenden Lagerlayouts.

Die Antwort fällt für die beiden Vergleichsgrößen unterschiedlich aus. In der Lösungsqualität liegen die Verfahren wenige Prozentpunkte auseinander, in der Laufzeit dagegen um bis zu fünf Größenordnungen. Der entscheidende Unterschied zwischen den Paradigmen liegt jedoch nicht nur in der Länge der Laufzeiten, sondern in den Parametern, die sie bedingen.

Hier unterscheiden sich die vorgestellten Algorithmen. Die Nearest-Neighbor-Heuristik und der Christofides-Algorithmus rechnen auf dem abstrahierten Distanzgraphen. Steht dieser einmal, ist das Lagerlayout für den weiteren Ablauf ohne Bedeutung. Die Auftragsgröße ist hier also der zentrale Komplexitätsparameter.

Der Fixed-Parameter-Algorithmus rechnet dagegen auf dem Gittergraphen und führt die Gangstruktur im Zustandsraum mit, weshalb sie dort unmittelbar in die Laufzeit eingeht.

Im Folgenden werden die drei Verfahren einzeln betrachtet und anschließend untersucht, ab welchen Eigenschaften eine Abweichung von der optimalen Route sinnvoll ist, um ein schnelleres Rechenergebnis zu erzielen.

Der Fixed-Parameter-Algorithmus von Cambazard und Catusse (2018) berechnet beweisgemäß immer die optimale Route, zum Preis hoher Berechnungskosten. Die Validierung in Abschnitt 4.7.1 bestätigt das stichprobenartig für Instanzen bis $h \leq 4$. Für die größeren Instanzen der Benchmarks stützt sich die Optimalität auf den Beweis der Vorlage. Mit dem Anstieg der Regalhöhe wächst die Berechnungszeit exponentiell. Ab einer Höhe von $r = 8$ kommt es mehrheitlich zu Timeouts (vgl. Abschnitt 5.3). Der Benchmark B1 (Abschnitt 5.2) zeigt bei fixierter Reihenzahl $r = 1$, dass die Lagerbreite gegenüber der vertikalen Expansion nur schwach wirkt. Die mittlere Berechnungsdauer bleibt dabei über alle zehn Breitenstufen unter 2 ms und steigt von 0,36 ms bei Breite 1 auf 1,63 ms bei Breite 10 (vgl. Tabelle 5.1). Über die Regalreihen wächst sie dagegen um das rund 36.000-Fache (vgl. Tabelle 5.5). Da r über $h_{\text{eff}} \leq r + 1$ die Zahl der Quergänge und damit die Pfadweite des Gittergraphen bestimmt, ist h der laufzeitbestimmende Parameter und die Regalreihenzahl wirkt als dessen Treiber.

Der Christofides-Algorithmus ist von diesem Parameter unabhängig. Seine mittlere Berechnungszeit bleibt über alle zehn Regalreihenstufen zwischen 3,15 und 3,52 ms und damit im Rahmen ihrer Streuung konstant (vgl. Tabelle 5.5). Über die Lagerbreite steigt sie um 12 % an (Abschnitt 5.2). An die Stelle der Optimalität greift hier eine bewiesene Schranke. Wie von Christofides (1976) gezeigt, erreicht die berechnete Route im schlimmsten Fall das 1,5-Fache der optimalen Route, übersteigt diese aber nicht. Im Mittel liegt der Christofides-Algorithmus je nach Benchmark 3,6 % bis 7,9 % über dem Optimum, gemessen als Optimalitätsgap gegenüber der exakten Lösung des Fixed-Parameter-Algorithmus (vgl. Tabelle 5.7).

Die Nearest-Neighbor-Heuristik ist das schnellste der drei Verfahren, weist jedoch auch die höchsten Abweichungen vom Optimum auf, im Mittel 4,1 % bis 13,7 % (vgl. Tabelle 5.7). Des Weiteren ist sie nur in 19,7 % bis 31,5 % der Instanzen besser als der Christofides-Algorithmus.

Die Lösungsqualität wird bedingt durch das Layout und die Auftragsgröße, welche abhängig voneinander wirken (vgl. Abschnitt 5.6). In den einblockigen Layouts von Benchmark B1 verbessert sich der Gap der Nearest-Neighbor-Heuristik mit

wachsender Auftragsgröße sogar leicht. Dieser sinkt von 4,35 % bei fünf Produkten auf 3,04 % bei dreißig Produkten. In den mehrblockigen Layouts der Benchmarks B2 und B3 kehrt sich das um. Dort steigt der Gap mit der Auftragsgröße deutlich, bei $r = 7$ von 5,33 auf 23,28 %. Der Nachteil wächst dabei nicht nur prozentual, sondern auch in der Weglänge. Auf derselben Stufe liegt die Nearest-Neighbor-Route bei fünf Produkten 6,7 Längeneinheiten über dem Optimum und bei dreißig Produkten 61,5, während der Christofides-Algorithmus von 5,6 auf 27,4 Längeneinheiten zunimmt. Die Auftragsgröße ist damit kein Nebeneffekt der Lagergröße, sondern eine eigene Einflussgröße auf die Lösungsqualität.

Aus den drei Verfahren und den beiden Einflussgrößen ergibt sich die Antwort auf die Trade-off-Frage in zwei Schritten. Der erste betrifft die Grenze des exakten Verfahrens, der zweite die Wahl zwischen den beiden übrigen.

Für Lagerlayouts mit einer geringen Anzahl von Regalreihen ($r \leq 7$) liefert der Fixed-Parameter-Algorithmus exakte Ergebnisse und terminiert zuverlässig innerhalb der Zeitschranke. Bei mittleren bis großen Systemen ist die Wahl eines anderen Algorithmus sinnvoll, um eine akzeptable Berechnungsdauer zu erzielen.

Ob der Einsatz der Nearest-Neighbor-Heuristik gerechtfertigt ist, hängt weniger vom Zeitdruck als von der Anzahl der Regalreihen ab. In flachen Lagern bleibt die Heuristik konkurrenzfähig. Über Benchmark B1 gemittelt liegt sie mit 4,05 % nur knapp hinter dem Christofides-Algorithmus mit 3,56 % (vgl. Tabelle 5.7). Bei den Lagerbreiten 3 bis 5 liegt sie sogar davor, bei allen übrigen Breiten der Christofides-Algorithmus (vgl. Tabelle A.2). Instanzweise gewinnt Christofides 354-mal, Nearest Neighbor 340-mal, und 386 der 1080 Instanzen sind gleich lang. Praktisch sind die Verfahren bei $r = 1$ nicht zu unterscheiden (vgl. Abschnitt 5.6). Mit wachsender Regalreihenzahl kehrt sich das Bild deutlich um. In Benchmark B2 stehen 11,60 % gegen 7,51 %, in Benchmark B3 13,66 % gegen 7,93 %. Erst hier wiegt der Qualitätsnachteil so schwer, dass die kürzere Berechnungsdauer ihn nur noch unter hohem Zeitdruck aufwiegt.

6.2 Praktische Implikationen

Mit den Ergebnissen dieser Arbeit lässt sich Lagerbetreibern eine Empfehlung aussprechen, welcher Algorithmus für das OPRP in Betracht gezogen werden kann. Sie gilt für die untersuchte Klasse von Lagersystemen: rechteckiges Layout aus parallelen Regalen mit einem Depot, bis zu 10 Regalspalten, bis zu 10 Regalreihen und Auftragsgrößen von 5 bis 30 Produkten. Innerhalb dieser Klasse muss zwischen kurzen Routen und geringen Berechnungszeiten abgewogen werden.

Eine Entscheidungsregel kann in Abhängigkeit von der Regalreihenzahl r und dem gegebenen Zeitbudget formuliert werden. Gemessen wurden allein die Layouts der drei Benchmarks, also 27 der 100 Kombinationen aus Breite und Regalreihenzahl, die diese Klasse zulässt. Weiter gelten die Annahmen des Versuchsaufbaus. Es bestehen gleichverteilte zufällige Pickorte und ein Einzelauftrag je Lauf und ein Depot. Ist r klein und ist eine optimale Route gewünscht, sollte der Fixed-Parameter-Algorithmus gewählt werden. Wenn es sich hingegen um ein großes Lager handelt, sollte der Christofides-Ansatz gewählt werden, der Qualität bei akzeptabler Zeit garantiert. Bei harten Echtzeitanforderungen, wenn die Route also innerhalb weniger Millisekunden vorliegen muss, empfiehlt sich der Nearest-Neighbor-Ansatz. Dieser ist in allen drei Benchmarks das schnellste Verfahren (vgl. Tabellen 5.1, 5.5 und 5.6). Der Qualitätsverlust muss hier bewusst in Kauf genommen werden. Tabelle 6.1 fasst diese Regel zusammen. Sie gilt für die mittlere Auftragsgröße dieser Benchmarks. Wie die Auftragsgröße die Grenze verschiebt, behandelt der Absatz zur Wirtschaftlichkeit weiter unten.

Die Matrix stützt sich bei den Zeitangaben auf die Mittelwerte der Messreihen. Das heißt, es existieren auch Läufe über den angegebenen Zeiten. Bei $r = 6$ liegt der Mittelwert mit 8,5 s zwar unter dem 10-Sekunden-Budget, 58 der 120 Läufe überschreiten es jedoch, im Maximum mit 11,5 s. Eine Ausnahme bildet der Randfall $r = 1$. Dort benötigt der Fixed-Parameter-Algorithmus im Mittel nur 1,64 ms und im Maximum 2,63 ms (vgl. Tabelle 5.5). Dieser liefert damit in allen 120 Läufen auch innerhalb des 5-Millisekunden-Budgets die optimale Route. Des Weiteren ist zu nennen, dass die verwendete Christofides-Implementierung bei Breite 10 im Mittel 3,21 ms benötigt. Dieser Wert ist implementierungsbedingt eine obere Schranke und wächst überproportional mit der Auftragsgröße (vgl. Abschnitt 4.5.3). Er gilt deshalb nur für die hier gemessenen Auftragsgrößen. Steht

nur eine sehr kurze Berechnungszeit zur Verfügung, muss daher auf die Nearest Neighbor Heuristik zurückgegriffen werden. Benchmark B1 zeigt, dass diese in flachen Lagersystemen mithält. Über den gesamten Benchmark gemittelt bleibt Christofides zwar vorn, in Benchmark B1 ist dieser Vorsprung jedoch klein und statistisch nicht signifikant (vgl. Abschnitt 5.6). Das betrifft allein die flachen Layouts dieses Benchmarks. In B2 und B3 ist der Unterschied deutlich und abgesichert. Steigt die Zeitschranke, verschiebt sich die Wahl wieder zugunsten des Christofides-Algorithmus.

		Zeitbudget pro Auftrag		
r	FP-Zeit	$\approx 5\,\mathrm{ms}$	$\approx 10\,\mathrm{s}$	$> 10\,\mathrm{s}$
≤ 4	bis 0,23 s	Nearest Neighbor	Fixed-Parameter	Fixed-Parameter
5	1,4 s	Nearest Neighbor	Fixed-Parameter	Fixed-Parameter
6	8,5 s	Nearest Neighbor	Fixed-Parameter†	Fixed-Parameter
7	58,6 s	Nearest Neighbor	Christofides	Christofides
≥ 8	Timeout	Nearest Neighbor	Christofides	Christofides

Tabelle 6.1: Empfohlener Algorithmus in Abhängigkeit der Regalreihenzahl r und des verfügbaren Zeitbudgets, abgeleitet aus den Mittelwerten von Benchmark B2 (Breite 10). Die Spaltenköpfe bezeichnen Budgets in der Größenordnung der angegebenen Werte. Die mitgeführte mittlere Berechnungszeit des Fixed-Parameter-Algorithmus (Spalte FP-Zeit) erlaubt es, dazwischenliegende Budgets selbst einzuordnen. Bei $r = 7$ bleibt der Fixed-Parameter-Algorithmus innerhalb eines großzügigen Zeitbudgets berechenbar, lohnt sich nach der Hochrechnung in diesem Abschnitt jedoch nicht mehr.

Die Wirtschaftlichkeit dieser Ergebnisse lässt sich überschlägig berechnen. Mahmoudinazlou u. a. (2024) gehen in ihrer Arbeit davon aus, dass eine Kommissionierkraft durchschnittlich mit einer Geschwindigkeit von 1 m/s unterwegs ist. Diese Annahme wird hier übernommen. Zusätzlich wird angenommen, dass eine Gittereinheit des Graphen einem Meter entspricht.

Unter diesen Annahmen lässt sich mit Gleichung 6.1 ermitteln, ab wann sich ein Algorithmus lohnt. Die Werte, mit denen im Folgenden gerechnet wird, stellen Mittelwerte aus Benchmark B2 dar.

$$\frac{L_B - L_A}{v_{\mathrm{Geh}}} > t_A - t_B \tag{6.1}$$

Dabei bezeichnen L_A und L_B die mittleren Routenlängen der beiden verglichenen Verfahren in Gittereinheiten, t_A und t_B ihre mittleren Berechnungszeiten in Sekunden und v_{Geh} die Gehgeschwindigkeit. Die linke Seite ist die durch die kürzere Route eingesparte Gehzeit, die rechte die zusätzlich benötigte Rechenzeit. Ist die Ungleichung erfüllt, lohnt sich das genauere Verfahren A. Mit $v_{\text{Geh}} = 1\,\text{m/s}$ und einer Gittereinheit pro Meter entspricht die Differenz der Routenlängen unmittelbar einer Zeitdifferenz in Sekunden.

Die zugrunde liegenden mittleren Routenlängen sind in Tabelle 5.4 aufgeführt. Es fällt auf, dass sich der Christofides-Algorithmus gegenüber der Nearest-Neighbor-Heuristik im Benchmark B2 im Mittel lohnt. Christofides spart bei $r = 1$ und einer Breite von 10 bereits 0,7 Sekunden und bei $r = 7$ 12,6 Sekunden im Mittel, bei einer Mehrrechenzeit von rund 3,1 ms (vgl. Tabelle 5.5).

Die Entscheidungsmatrix ist über alle Auftragsgrößen gemittelt und verschiebt die Grenze damit um eine Stufe. Wird der Break-even für jede Kombination aus Regalreihenzahl und Auftragsgröße einzeln ausgewertet, so lohnt sich der Fixed-Parameter-Algorithmus bis $r = 5$ bei jeder untersuchten Auftragsgröße. Bei $r = 6$ lohnt er sich erst ab 10 Produkten. Für einen Auftrag mit 5 Produkten stehen dort 3,8 Sekunden Ersparnis gegen 4,3 Sekunden Mehrrechenzeit, ab 10 Produkten dagegen 9,7 bis 22,9 Sekunden Ersparnis gegen 6,5 bis 10,8 Sekunden Mehrzeit. Bei $r = 7$ lohnt er sich in keiner einzigen Zelle. Die wirtschaftliche Grenze liegt damit bei $r = 6$ für Aufträge ab 10 Produkten und bei $r = 5$ für Aufträge von 5 Produkten.

Im Vergleich des Fixed-Parameter-Algorithmus mit Christofides zeigt sich, dass die exakte Lösung bei $r = 1$ 4,4 Sekunden einsparen würde, bei $r = 6$ 16 Sekunden und bei $r = 7$ 20,1 Sekunden. Hier wird jedoch auch die Mehrrechenzeit des Fixed-Parameter-Algorithmus relevant. Bei $r = 6$ wären das 8,5 Sekunden und bei $r = 7$ 58,6 Sekunden. Der Break-even liegt für den Fixed-Parameter-Algorithmus damit bei $r = 6$, wo sich Berechnungsdauer und eingesparte Gehzeit noch die Waage halten. Bei $r = 7$ steht die Ersparnis von 20,1 Sekunden einer Mehrrechenzeit von 58,6 Sekunden gegenüber.

Benchmark B3 stützt diese Einordnung unabhängig. Dort spart die exakte Lösung bei $r = 6$ rund 13,9 Sekunden bei 5,2 Sekunden Mehrrechenzeit und lohnt sich damit, während sie bei $r = 7$ 16,7 Sekunden Ersparnis gegen 40,6 Sekunden

Mehrzeit stellt und sich nicht mehr lohnt. Der Break-even liegt also auch bei mitwachsender Breite an derselben Stelle. Dieser Punkt hängt jedoch von den gewählten Parametern ab. Sollte die Kommissionierkraft zum Beispiel schneller gehen, könnte sich der Break-even-Punkt nach unten verschieben, während größere Zellabstände diesen nach oben verschieben könnten. Auch eine schnellere Hardware oder eine optimierte Implementierung würde die Berechnungszeit senken und den Break-even-Punkt damit nach oben verschieben. Falls die Routen vorberechnet werden, tritt der Break-even-Punkt in den Hintergrund und die Berechnungszeit kann nahezu vernachlässigt werden.

6.3 Limitationen

In dieser Arbeit wurde simuliert, wie sich die drei genannten Algorithmen in einem möglichst realistischen Lagermodell verhalten. Jede Simulation beruht jedoch auf vereinfachenden Annahmen, die den Geltungsbereich der Ergebnisse einschränken.

So wurde in dieser Arbeit nur ein Lagerlayout-Typ verwendet. Dieser ist rechteckig, hat parallele Regale und ein Depot. Dieser Typ ist zwar der Standardfall vieler realer Lager, die Erkenntnisse gelten jedoch nur innerhalb der in Abschnitt 6.2 genannten Grenzen. Auf abweichend aufgebaute Lager sind sie nicht ohne Weiteres übertragbar.

Auch die Regalbreite von zwei und die Höhe von sechs Gitterzellen sind in dieser Arbeit fest fixiert. Wie sich die Algorithmen bei anderen Regaldimensionen verhalten würden, bleibt offen. Aus dieser festen Blockgröße folgt zudem, dass ein gleiches Zahlenpaar aus Regalspalten und Regalreihen kein quadratisches Lager ergibt.

Ebenso wurde die Auftragsgröße nur zwischen 5 und 30 Produkten variiert. Da der Optimalitätsgap beider Heuristiken in den mehrblockigen Layouts mit der Auftragsgröße wächst (Abschnitt 5.6), lassen sich die berichteten Werte nicht auf größere Aufträge übertragen. Für Aufträge oberhalb dieses Bereichs ist ein größerer Gap zu erwarten, dessen Höhe hier nicht gemessen ist. Aus demselben Grund gilt die Entscheidungsmatrix in Abschnitt 6.2 nur innerhalb dieser Spanne.

Des Weiteren wurde sich auf die drei vorgestellten Algorithmen beschränkt. Die in Abschnitt 3.4 beschriebenen klassischen Routenstrategien wie S-Shape oder Largest-Gap wurden nicht in den Vergleich aufgenommen, obwohl sie in der Praxis verbreitet sind (De Koster, Le-Duc und Roodbergen 2007). Ein Vergleich zu diesen ist deswegen nur indirekt über die Literaturwerte in Abschnitt 6.4 möglich.

Ebenso festgelegt ist die Problemvariante. Das TSP, welches simuliert wird, ist symmetrisch und metrisch. Andere Formen des TSP werden nicht berücksichtigt.

Simuliert wurde zudem strikt nur mit einer Kommissionierkraft. Sehr verbreitete Verfahren wie Batching oder Zoning wurden nicht behandelt (Boysen, Koster und Weidinger 2019). Es wurden zudem keine Produktgewichte oder Volumina berücksichtigt, sodass keine Kapazitätsgrenze des Kommissionierwagens in das Modell eingeht.

Auch wurden die zu besuchenden Lagerorte gleichverteilt zufällig gezogen. Diese Annahme dürfte die Unterschiede zwischen den Verfahren eher vergrößern als verkleinern, da die Picks über das gesamte Lager streuen und sich keine kurzen Teilrouten anbieten. Belegt ist dieser Zusammenhang in dieser Arbeit jedoch nicht, da nur eine Verteilung untersucht wurde. Bei einer realen, korrelierten Einlagerung wäre zu erwarten, dass die Unterschiede geringer ausfallen.

Ebenso unverändert blieb die Lage des Depots. Es liegt in allen Läufen fest in der unteren linken Ecke des Lagers. Petersen (1997) zeigt auf, dass die Position des Depots einen Einflussfaktor auf die Routenlänge darstellt. Deswegen lassen sich die hier berichteten absoluten Längen nicht ohne Weiteres auf Lager mit mittig oder mehrfach angeordneten Übergabepunkten übertragen.

Eine weitere Einschränkung ergibt sich aus dem Verhalten des Fixed-Parameter-Algorithmus selbst. Ab einer Höhe von $h_{\text{eff}} = 9$, also $r = 8$, überschreitet dieser mehrheitlich die Zeitschranke von 90 Sekunden (vgl. Abschnitt 5.3), sodass in diesem Bereich keine exakte Referenzlösung mehr vorliegt. Das begrenzt die Aussagekraft der Ergebnisse für Lagersysteme mit $r \geq 8$, da hier kein belastbarer Optimalitätsgap mehr ermittelt werden kann und nur noch ein relativer Vergleich zwischen Nearest Neighbor und Christofides möglich ist. Auch die Ergebnisse des Durchlaufes für $r = 8$ sind verzerrt, da viele der Berechnungen nicht mehr innerhalb der Zeitschranke terminieren. So haben die terminierenden Ergebnisse

einen Survivorship-Bias.

Dieser Effekt betrifft nicht nur die Berechnungszeiten, sondern auch die Routenlängen. Die mittlere Länge der Fixed-Parameter-Lösungen sinkt in Benchmark B2 von 203,05 Gittereinheiten bei $r = 7$ auf 162,58 bei $r = 8$ und 117,50 bei $r = 9$, obwohl das Lager wächst. Ursache ist allein die Auswahl der überlebenden Instanzen. Terminieren können bei hohem r nur jene Aufträge, deren Pickorte das effektive h klein halten (vgl. Abschnitt 5.3). Mit steigendem r verschärft sich das. Bei $r = 8$ terminieren noch 24 von 120 Läufen, bei $r = 9$ nur 4 und bei $r = 10$ lediglich 2. Insgesamt entfallen 330 der 1200 Läufe in Benchmark B2 und 328 der 1120 in Benchmark B3 auf Timeouts. Um die beschriebene Verzerrung auszuschließen, werden die Optimalitätsgaps nur auf den Layoutstufen ausgewertet, auf denen der Fixed-Parameter-Algorithmus ausnahmslos terminiert. Sie beruhen deshalb auf 1080 Instanzen in B1, 840 in B2 ($r \leq 7$) und 760 in B3 ($r \leq 7$) statt auf den vollen Datensätzen.

Es wurden auch keine Metaheuristiken auf die Ergebnisse angewandt, um diese zu verbessern. Somit wäre ein potenziell kleinerer Optimalitätsgap für Nearest Neighbor und Christofides möglich.

Eine grundsätzliche Einschränkung betrifft die Messung von Berechnungszeiten. Die Berechnungszeit als Vergleichsmetrik zu nutzen ist als relativer Vergleich akzeptabel, als absoluter Wert dagegen nicht. So können Eigenschaften wie Implementierung, Maschine und Versuchsaufbau die Messung unmittelbar beeinflussen. Auch ist es schwer zu bestimmen, wo die Berechnungszeit beginnt und wo diese endet. Vorbereitende Schritte wie der Aufbau einer Datenstruktur, das Einlesen der Instanz oder die Aufbereitung des Ergebnisses lassen sich je nach Sichtweise dem Verfahren oder seiner Umgebung zurechnen. Beide Zurechnungen sind vertretbar. Entscheidend ist deshalb weniger, welche Grenze gewählt wird, als dass sie benannt und für alle verglichenen Verfahren identisch gezogen wird. Andernfalls misst der Vergleich zu einem Teil die Messung selbst (D. S. Johnson 2002). Diese Arbeit benennt ihre Messgrenze in Abschnitt 4.7 und zieht sie für alle drei Verfahren identisch. Die absoluten Werte bleiben davon unberührt an Maschine und Implementierung gebunden. An die Messgrenze schließt eine zweite, davon unabhängige Einschränkung an. Auch die Art der Implementierung in der Programmiersprache Python sorgt für höhere absolute Berechnungszeiten. Diese könnten mit einer anderen Programmiersprache deutlich kürzer ausfallen.

Hinzu kommt, dass die drei Implementierungen nicht denselben Reifegrad besitzen. Der Christofides-Algorithmus greift auf das etablierte Paket NetworkX zurück, während der Fixed-Parameter-Algorithmus für diese Arbeit eigenständig umgesetzt wurde. Dabei wurden das Preprocessing des Manhattan-Netzwerks und die Filterregel aus Cambazard und Catusse (2018) nicht übernommen. Beides senkt keine asymptotische Schranke, verringert aber den konstanten Faktor der Laufzeit. Der in Abschnitt 6.2 bestimmte Break-even-Punkt hängt unmittelbar von diesem Faktor ab und läge mit einer optimierten Implementierung bei einer höheren Regalreihenzahl. Der Vergleich der Verfahren ist damit nicht vollständig implementierungsunabhängig.

Schließlich beruhen alle berichteten Werte auf Mittelwerten aus 20 Iterationen je Konfiguration. Für die Benchmarks B2 und B3 ist der Qualitätsunterschied zwischen Nearest Neighbor und Christofides statistisch abgesichert, für Benchmark B1 hingegen nicht (vgl. Abschnitt 5.6). Aussagen über flache Lagersysteme sind daher zurückhaltender zu lesen als die übrigen.

6.4 Einordnung in die Literatur

In der Literatur werden Heuristiken und exakte Verfahren meist getrennt voneinander betrachtet. Der Beitrag dieser Arbeit besteht darin, sie unter denselben Bedingungen nebeneinanderzustellen. Im Folgenden wird geprüft, wo sich die Ergebnisse dieser Arbeit mit den publizierten Arbeiten decken oder widersprechen.

Die qualitativen Ergebnisse des Christofides-Algorithmus ordnen sich in die Literatur ein. Das gemessene Maximum von 24,2 % liegt deutlich unter den 50 %, die die 3/2-Schranke zulässt. Der Worst Case wird in dieser Arbeit nicht erreicht.

Die Laufzeitexplosion in Abhängigkeit des Parameters h, die in den Ergebnissen klar zu sehen ist, bestätigt die Ergebnisse von Cambazard und Catusse (2018). Die Zahl der Quergänge stellt sich auch hier als der kritische Parameter dar, während die Lagerbreite nach Gleichung 4.7 nur linear eingeht. Bei $r = 7$ steigt die mittlere Zeit von 40,6 s bei Breite 7 auf 58,6 s bei Breite 10. Dies deckt sich mit der in Abschnitt 4.6.7 hergeleiteten Laufzeit. Diese Übereinstimmung stützt sich nicht allein auf Zeiten einer einzelnen Maschine. Die Zustandszahl $|\Omega(h)|$ aus

Tabelle 4.3 ist eine maschinenunabhängige Kennzahl des Verfahrens, und der aus ihr vorhergesagte Wachstumsfaktor findet sich in den gemessenen Zeiten wieder (Abschnitt 5.3).

Bei der Nearest-Neighbor-Heuristik bleibt der Optimalitätsgap deutlich unter dem, was die Worst-Case-Schranke von Rosenkrantz, Stearns und Lewis (1977) zulässt. Für diesen Vergleich sind die gemessenen Maxima einschlägig, nicht die Mittelwerte. Diese liegen bei 35,71 % (B1), 44,36 % (B2) und 50,00 % (B3). Die konkrete Schranke $\frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}$ erlaubt dagegen bei $n = 30$ das Dreifache der optimalen Tour.

De Koster, Le-Duc und Roodbergen (2007) referieren einen Vergleich von sechs Routing-Methoden (Petersen 1997). Die besten Einzelheuristiken, Composite und Largest-Gap, liegen dort 9 bis 10 % über dem Optimum. Die je Instanz beste der fünf Heuristiken kommt im Mittel auf rund 5 %. In den Ergebnissen dieser Arbeit liegt Nearest Neighbor je nach Benchmark 4,1 % bis 13,7 % und der Christofides-Ansatz 3,6 % bis 7,9 % über dem Optimum. Die jeweils untere Grenze stammt aus Benchmark B1 mit nur einer Regalreihe, in dem das Routingproblem am einfachsten ist. In den mehrreihigen Layouts der Benchmarks B2 und B3 liegen die Werte bei 11,6 % bis 13,7 % beziehungsweise 7,5 % bis 7,9 %. Gegen die 9 bis 10 % der besten Einzelheuristiken liegt der Christofides-Algorithmus damit in allen drei Benchmarks besser, während Nearest Neighbor in den mehrreihigen Layouts darüber liegt.

Der Vergleich ist allerdings nur eingeschränkt tragfähig. Die von De Koster, Le-Duc und Roodbergen (2007) betrachteten Heuristiken sind auf feste Lagerlayouts zugeschnitten und folgen strikten Regeln, während die hier vorgestellten Lösungsansätze für das allgemeine OPRP gelten. Hinzu kommt, dass Petersen (1997) ein Layout mit lediglich zwei Quergängen untersucht, was in der Notation dieser Arbeit $r = 1$ entspricht. Strukturgleich ist damit nur Benchmark B1 mit 4,05 % (Nearest Neighbor) und 3,56 % (Christofides). Die Benchmarks B2 und B3 gehen über den dort untersuchten Layouttyp hinaus.

Kapitel 7

Fazit & Ausblick

7.1 Zusammenfassung

In dieser Arbeit wurden drei Algorithmen mit jeweils verschiedenen Ansätzen implementiert, einem heuristischen, einem approximativen und einem parametrisierten. Diese wurden auf demselben Lagermodell umgesetzt und miteinander verglichen. Dafür wurde eine Webanwendung entwickelt und das Verhalten der Algorithmen anhand von drei systematischen Benchmarks mit insgesamt 10.200 Läufen analysiert.

Der Vergleich der Algorithmen zeigte, dass die Laufzeit bei jedem Verfahren von unterschiedlichen Faktoren abhängig ist. Bei Nearest Neighbor und dem Christofides-Algorithmus ist diese von der Produktanzahl getrieben, wobei die Unterschiede im Millisekundenbereich bleiben. Christofides steigt zwischen 5 und 30 Produkten von 0,33 auf 8,71 ms (Benchmark B2). Beim Fixed-Parameter-Algorithmus ist es die Anzahl der Regalreihen r, und dort wächst die Berechnungszeit exponentiell an, von 1,64 ms bei $r = 1$ auf 58,6 s bei $r = 7$. Für Nearest Neighbor und Christofides ist die Lagerbreite praktisch ohne Belang. Beim Fixed-Parameter-Algorithmus geht sie nur linear ein, bleibt damit gegenüber r um Größenordnungen unkritisch, bleibt aber messbar. Bei $r = 7$ trennen die Breiten 7 und 10 rund 18 Sekunden (40,6 s gegenüber 58,6 s). Ebenso wirkt die Auftragsgröße auf den Fixed-Parameter-Algorithmus polynomiell. Bei $r = 7$ kostet die Spanne von 5 auf 30 Produkten 49,8 Sekunden (25,2 s auf 75,0 s) und damit absolut ebenso viel wie ein zusätzlicher Regalreihenschritt, im Faktor mit 2,98 gegenüber 6,86 jedoch deutlich weniger.

Die Qualitätsunterschiede sind moderat. Nearest Neighbor liegt im Mittel 4,1 % bis 13,7 % über dem Optimum, Christofides 3,6 % bis 7,9 % (vgl. Tabelle 5.7). Christofides überschreitet mit maximal 24,2 % auf allen 2680 Instanzen der Stufen mit vollständiger Terminierung die theoretische 3/2-Schranke nie. Der Unterschied in der Berechnungszeit wird bei dem Durchlauf $r = 7$ in Benchmark B2 deutlich. Dieser Durchgang hat die größte Ausprägung von r, bei der noch keine Timeouts auftreten. Hier braucht der Fixed-Parameter-Algorithmus rund 248.000-mal so viel Rechenzeit wie der Nearest-Neighbor-Ansatz, während Nearest Neighbor im Mittel 14,9 % längere Routen berechnet.

Aus den Ergebnissen lässt sich ableiten, bis zu welcher Größe von r es sich lohnt, exakte Routen zu berechnen. Bei $r = 8$ scheitern 80 % der Berechnungen an der Timeout-Schranke von 90 s, ab $r = 9$ sind es in Benchmark B2 über 96 % und in Benchmark B3 über 93 %. Die wirtschaftliche Grenze liegt unter den Annahmen aus Abschnitt 6.2 jedoch schon bei $r = 6$, denn bei $r = 7$ übersteigt die Berechnungszeit die eingesparte Gehzeit.

Die Forschungsfrage dieser Arbeit lautete: “Wie unterscheiden sich der Christofides-, Nearest-Neighbor- und Fixed-Parameter-Algorithmus nach Cambazard und Catusse hinsichtlich ihrer Lösungsqualität und Laufzeit bei steigender Problemgröße und variierenden Lagerlayouts des OPRP?” In der Lösungsqualität unterscheiden sich die Ansätze moderat, in der Laufzeit um Größenordnungen. Ausschlaggebend ist dabei die Zahl der Quergänge h, die über $h_{\text{eff}} \leq r + 1$ mit der Anzahl der Regalreihen wächst. Für Nearest Neighbor und Christofides blieben Produktanzahl und Lagerbreite in ihrer Wirkung auf die Laufzeit im Millisekundenbereich und damit praktisch ohne Belang. Beim Fixed-Parameter-Algorithmus wirken beide Größen polynomiell, im Break-even-Bereich aber im Sekundenbereich; sie verschieben die Wirtschaftlichkeitsgrenze auftragsgrößenabhängig.

Die Lösungsqualität hängt dabei ebenfalls vom Layout ab, allerdings unterschiedlich stark je Verfahren. Der Gap von Nearest Neighbor wächst mit der Komplexität des Layouts von 4,05 % in Benchmark B1 auf 13,66 % in Benchmark B3, während der Christofides-Algorithmus mit 3,56 % bis 7,93 % deutlich stabiler bleibt und in keinem Durchlauf die 3/2-Schranke erreicht. Mit steigender Layoutkomplexität wird die Wahl des Verfahrens damit nicht nur eine Laufzeit-, sondern auch eine Qualitätsentscheidung.

Der Beitrag dieser Arbeit liegt darin, die drei Ansätze in einem gemeinsamen Aufbau auf einem gängigen rechteckigen Lagermodell verglichen zu haben, und zwar auf denselben Instanzen, unter derselben Messgrenze und mit einem exakten Verfahren als Referenz. Die Optimalitätsgaps sind damit für alle Instanzen, auf denen dieses Verfahren terminiert, gemessen und nicht geschätzt. Erhoben wurden die Daten in einem Docker-Container, dessen Aufbau und Aufruf in Anhang A.1 dokumentiert sind.

Für die Praxis folgt daraus keine einzelne Kennzahl, sondern eine Reihenfolge von drei Entscheidungen. Ob das exakte Verfahren überhaupt in Frage kommt, entscheidet die Anzahl der Regalreihen, denn sie allein bestimmt, ob es innerhalb der Zeitschranke terminiert. Ob es sich lohnt, entscheiden Regalreihenzahl und Auftragsgröße gemeinsam. Unterhalb der in Abschnitt 6.2 bestimmten Grenze ist die exakte Route die wirtschaftlich bessere Wahl, oberhalb davon ist sie es nicht mehr, und der Approximationsalgorithmus liefert den tragfähigen Kompromiss. Wie teuer der Verzicht auf die exakte Route ausfällt, hängt schließlich von der Komplexität des Layouts und ebenso stark von der Auftragsgröße ab (vgl. Abschnitt 5.6). Die Wahl des Verfahrens ist damit eine Abwägung zwischen Berechnungszeit und Routenlänge, in die drei Größen eingehen und nicht nur die Größe des Lagers.

7.2 Ausblick

In dieser Arbeit wurde empirisch untersucht, welcher der drei untersuchten Algorithmen für welche Art von Lagerkonfiguration am besten geeignet ist.

Unmittelbar an die Ergebnisse anschließen lässt sich die Frage, wie weit sich die Wirtschaftlichkeitsgrenze von $r = 6$ durch eine schnellere Implementierung verschieben lässt. Die hier umgesetzte Variante verzichtet auf das Preprocessing und die Filterregel der zugrunde liegenden Arbeiten (vgl. Abschnitt 6.3). Beide senken nach Cambazard und Catusse (2018) den konstanten Faktor erheblich, ohne die asymptotische Schranke zu verändern. Zusammen mit einer kompilierten Implementierung anstelle der hier verwendeten Python-Umgebung wäre zu messen, um wie viele Regalreihen sich der Break-even dadurch nach oben verschiebt. Die hier gemessene Grenze deckt sich mit der Grenze der Vorlage. Cambazard und

Catusse (2018) berichten Ergebnisse bis $h = 8$ und scheitern bei $h = 9$ am Speicher, während hier bis $h_{\text{eff}} = 8$ zuverlässig terminiert wird und bei $h_{\text{eff}} \geq 9$ die Zeitschranke greift. Da h am Auftrag hängt, fällt diese Grenze nicht mit einer Regalreihenzahl zusammen. Bei $r = 8$ terminieren noch 24 der 120 Läufe, bei $r = 9$ vier und bei $r = 10$ zwei. Der geschätzte Aufwand liegt bei einigen Wochen. Preprocessing und Filterregel sind in den Vorlagen beschrieben und lassen sich in Tagen ergänzen. Den größeren Teil beanspruchen die Portierung des Zustandsraums in eine kompilierte Sprache und die erneute Vermessung.

Ebenso offen bleibt der Vergleich mit den klassischen Lagerheuristiken wie S-Shape oder Largest-Gap. Sie wurden hier nicht implementiert, weshalb sich die Ergebnisse nicht direkt an die Literatur zu diesen Strategien anschließen lassen. Für Lagerlayouts mit $r \geq 8$ fehlt zudem eine exakte Referenz. Ein Verfahren, das dort noch terminiert, würde die Gap-Messung auf den praktisch interessanten Bereich ausdehnen. Schließlich wurde das Seitenverhältnis der Layouts nie variiert, sodass der Einfluss der Regaldimensionen offenbleibt. Der geschätzte Aufwand liegt bei wenigen Tagen für S-Shape und Largest-Gap, da beide regelbasiert und ohne Graphsuche umsetzbar sind, und bei ebenso wenigen Tagen für die Variation des Seitenverhältnisses, die nur eine weitere Payload und einen Rechenlauf erfordert. Eine exakte Referenz für $r \geq 8$ ist der teuerste Punkt und liegt eher bei Wochen, weil dafür ein anderes Optimalverfahren angebunden werden muss.

Ein weiterer Ansatzpunkt, der für das hier behandelte Problem besonders interessant ist, wäre der Prozess des Batchings. Gegeben sind m Kundenaufträge mit jeweils k Produkten. Alle Produkte müssen in dem Lager eingesammelt werden. Wie können diese gruppiert werden, sodass die Summe der Routenlängen aller Batches minimal ist (Valle, Beasley und Cunha 2017)? Dieses Problem ist alltäglich in jedem Lager vorzufinden, da die Kommissionierkraft meist nur eine begrenzte Zahl an Produkten auf einmal transportieren kann (Boysen, Koster und Weidinger 2019). In dem Batching-Verfahren müssen sehr viele verschiedene Gruppierungen berechnet werden, wobei jede Gruppierung ein Routing-Problem wie das OPRP darstellt. Ein exaktes Lösungsverfahren würde hier die tatsächlichen Kosten einer Gruppierung liefern, während heuristische und approximative Verfahren lediglich einen Schätzwert dieser Kosten berechnen. Welcher der Algorithmen innerhalb eines Batching-Verfahrens genutzt werden könnte, ist nach den Ergebnissen dieser Arbeit abhängig von der Lagerkonfiguration. Der geschätzte Aufwand liegt bei

mehreren Monaten. Das Batching ist ein eigenes Optimierungsproblem, das das OPRP nur als Unterproblem enthält, und füllt damit eine eigene Arbeit.

In dieser Arbeit wurden die Lagerorte, die gepickt werden müssen, immer als gegeben angesehen. In der Praxis ist dies meist nicht der Fall. Produkte werden nach dem Chaosprinzip eingelagert. Dies bedeutet, dass ein Produkt genau dort eingelagert wird, wo gerade Platz frei ist. So kann es in einem Routing-Problem auch zu einem zusätzlichen Entscheidungsproblem kommen, bei dem die Kommissionierkraft den zu besuchenden Lagerort wählen muss. In der Literatur wird dies von Weidinger und Boysen (2018) als Scattered Storage definiert. Weidinger (2018) zeigt zudem, dass dieses Problem stark NP-schwer ist. Untersucht wird es dort in einem vergleichbaren rechteckigen Lagermodell. Ohne Scattered Storage ist das Problem in einblockigen Lagern polynomiell lösbar (Ratliff und Rosenthal 1983) und bei jeder festen Quergangzahl parametrisiert handhabbar (Cambazard und Catusse 2018). Mit Scattered Storage ist es stark NP-schwer. Es kommt also zu einem Komplexitätssprung. Für den Fixed-Parameter-Algorithmus würde dies bedeuten, dass der Frontier-Zustand zusätzlich mitführen müsste, welche Produkte bereits eingesammelt wurden. Damit wäre auch die Produktanzahl n unmittelbar relevant für den Zustandsraum, und die Beschränkung auf den Parameter h würde nicht mehr funktionieren. Außerdem wäre die Eingabe keine Liste von zu besuchenden Koordinaten, sondern eine Liste von Mengen möglicher Koordinaten, aus denen jeweils eine auszuwählen ist. Eine nächstgelegene Koordinate könnte global für die Gesamtroute die schlechteste Wahl sein. Hier wäre eine Untersuchung interessant, welche heuristische Lagerortwahl aus den möglichen Lagerorten kombiniert mit einem exakten Routing-Verfahren die besten Ergebnisse liefert. Der geschätzte Aufwand liegt bei mehreren Monaten. Der Frontier-Zustand des Fixed-Parameter-Algorithmus müsste um die bereits eingesammelten Produkte erweitert werden, womit die Beschränkung auf den Parameter h entfällt. Ein anderer Ansatz, die Gesamtroute des OPRP zu minimieren, wäre eine vorherige Optimierung der zu besuchenden Lagerorte. Anstatt eine gegebene Lagerortmenge zu akzeptieren und für diese die kürzeste Route zu berechnen, könnte auch schon initial die Verteilung der Produkte zu kürzeren Gesamtrouten führen.

Dieses Einlagerungsproblem wird in der Fachliteratur als Storage Location Assignment Problem (SLAP) bezeichnet. Zhang, Wang und Pan (2019) modellieren das SLAP anhand einer Nachfragekorrelation. Sobald Produkte oft zusammen

bestellt werden, wird versucht, diese möglichst nah aneinander einzulagern. So kommt es zu einer wahrscheinlich kürzeren Route, bevor ein Algorithmus anfängt zu rechnen. In dieser Arbeit wurden die zu besuchenden Lagerorte gleichverteilt zufällig gezogen. Es wäre für einen weiteren Ausblick interessant, inwiefern sich die Algorithmen bei optimierter Lagerverteilung im Optimalitätsgap angleichen. Der geschätzte Aufwand liegt bei einigen Wochen, sofern nur die Ziehung der Lagerorte auf eine korrelierte Verteilung umgestellt und die Benchmarks wiederholt werden. Ein eigenes Einlagerungsverfahren zu entwerfen, wäre erneut eine eigene Arbeit.

Die aktuelle Entwicklung in der Logistik zeigt, dass die Arbeit der Kommissionierkräfte immer mehr von robotisierten Lösungen übernommen wird (Azadeh, Koster und Roy 2019; Boysen, Koster und Weidinger 2019). Damit verändert sich auch das Problem. Roboter sind nicht an herkömmliche Lagersysteme gebunden und können sich an Hochregalen auch vertikal bewegen. Daraus ergibt sich ein komplexeres Problem, bei dem eine dreidimensionale Bewegung im Raum berücksichtigt werden muss. Die Distanzberechnung und auch Ansätze müssten für dieses neue Problem grundsätzlich angepasst oder verworfen werden. Der geschätzte Aufwand reicht von mehreren Monaten bis zu einer eigenen Arbeit.

Literatur

Arnborg, Stefan, Jens Lagergren und Detlef Seese (1991). "Easy Problems for Tree-Decomposable Graphs". In: *Journal of Algorithms* 12.2, S. 308–340. DOI: 10.1016/0196-6774(91)90006-K.

Azadeh, Kaveh, René de Koster und Debjit Roy (Aug. 2019). "Robotized and Automated Warehouse Systems: Review and Recent Developments". In: *Transportation Science* 53.4, S. 917–945. DOI: 10.1287/trsc.2018.0873.

Bodlaender, Hans L. (1996). "A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth". In: *SIAM Journal on Computing* 25.6, S. 1305–1317. DOI: 10.1137/S0097539793251219.

– (1998). "A Partial k-Arboretum of Graphs with Bounded Treewidth". In: *Theoretical Computer Science* 209.1-2, S. 1–45. DOI: 10.1016/S0304-3975(97)00228-4.

Boysen, Nils, René de Koster und Felix Weidinger (Sep. 2019). "Warehousing in the E-Commerce Era: A Survey". In: *European Journal of Operational Research* 277.2, S. 396–411. DOI: 10.1016/j.ejor.2018.08.023.

Cambazard, Hadrien und Nicolas Catusse (Okt. 2018). "Fixed-Parameter Algorithms for Rectilinear Steiner Tree and Rectilinear Traveling Salesman Problem in the Plane". In: *European Journal of Operational Research* 270.2, S. 419–429. ISSN: 03772217. DOI: 10.1016/j.ejor.2018.03.042. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221718302716 (besucht am 18. 03. 2026).

Christofides, Nicos (1976). *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem.* Technical Report 388. Neuveröffentlichung: Operations Research Forum 3, Art. 20 (2022), https://doi.org/10.1007/s43069-021-00101-z. Graduate School of Industrial Administration, Carnegie Mellon University.

Cormen, Thomas H. u. a. (2009). *Introduction to Algorithms.* 3. ed. Cambridge, Mass.: MIT Press. 1292 S. ISBN: 978-0-262-03384-8.

Courcelle, Bruno (1990). “The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs”. In: *Information and Computation* 85.1, S. 12–75. DOI: 10.1016/0890-5401(90)90043-H.

Dantzig, George, Ray Fulkerson und Selmer Johnson (1954). “Solution of a Large-Scale Traveling-Salesman Problem”. In: *Journal of the Operations Research Society of America* 2.4, S. 393–410. DOI: 10.1287/opre.2.4.393.

De Koster, René, Tho Le-Duc und Kees Jan Roodbergen (Okt. 2007). “Design and Control of Warehouse Order Picking: A Literature Review”. In: *European Journal of Operational Research* 182.2, S. 481–501. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.07.009. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221706006473 (besucht am 18. 03. 2026).

Diestel, Reinhard (2017). *Graph Theory*. 5. Aufl. Graduate Texts in Mathematics 173. Berlin, Heidelberg: Springer. ISBN: 978-3-662-53621-6. DOI: 10.1007/978-3-662-53622-3.

Dijkstra, Edsger W. (1959). “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1, S. 269–271. DOI: 10.1007/BF01386390.

Downey, Rod G. und Michael R. Fellows (1999). *Parameterized Complexity*. Monographs in Computer Science. New York: Springer. ISBN: 978-0-387-94883-6. DOI: 10.1007/978-1-4612-0515-9.

Edmonds, Jack (1965). “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17, S. 449–467. DOI: 10.4153/CJM-1965-045-4.

Gabow, Harold N. (1976). “An Efficient Implementation of Edmonds’ Algorithm for Maximum Matching on Graphs”. In: *Journal of the ACM* 23.2, S. 221–234. DOI: 10.1145/321941.321942.

Hart, Peter E., Nils J. Nilsson und Bertram Raphael (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, S. 100–107. DOI: 10.1109/TSSC.1968.300136.

Held, Michael und Richard M. Karp (1962). “A Dynamic Programming Approach to Sequencing Problems”. In: *Journal of the Society for Industrial and Applied Mathematics* 10.1, S. 196–210. DOI: 10.1137/0110015.

Helsgaun, Keld (2000). “An Effective Implementation of the Lin–Kernighan Traveling Salesman Heuristic”. In: *European Journal of Operational Research* 126.1, S. 106–130. DOI: 10.1016/S0377-2217(99)00284-2.

Johnson, David S. (2002). “A Theoretician’s Guide to the Experimental Analysis of Algorithms”. In: *Data Structures, Near Neighbor Searches, and Methodology:*

Fifth and Sixth DIMACS Implementation Challenges. Hrsg. von Michael H. Goldwasser, David S. Johnson und Catherine C. McGeoch. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 59. Providence, RI: American Mathematical Society, S. 215–250.

Johnson, David S. und Lyle A. McGeoch (1997). "The Traveling Salesman Problem: A Case Study in Local Optimization". In: *Local Search in Combinatorial Optimization*. Hrsg. von Emile H. L. Aarts und Jan Karel Lenstra. Chichester: Wiley, S. 215–310. ISBN: 0-471-94822-5.

Karlin, Anna R., Nathan Klein und Shayan Oveis Gharan (2021). "A (Slightly) Improved Approximation Algorithm for Metric TSP". In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2021)*. ACM, S. 32–45. DOI: 10.1145/3406325.3451009.

Karp, Richard M. (1972). "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. Hrsg. von Raymond E. Miller und James W. Thatcher. Plenum Press, S. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

Lin, Shen und Brian W. Kernighan (1973). "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". In: *Operations Research* 21.2, S. 498–516. DOI: 10.1287/opre.21.2.498.

Mahmoudinazlou, Sasan u. a. (2024). *Deep Reinforcement Learning for Dynamic Order Picking in Warehouse Operations*. arXiv:2408.01656 [math.OC], Version 2 (April 2025). arXiv: 2408.01656 [math.OC]. URL: https://arxiv.org/abs/2408.01656 (besucht am 18. 03. 2026).

Masae, Makusee, Christoph H. Glock und Eric H. Grosse (2020). "Order Picker Routing in Warehouses: A Systematic Literature Review". In: *International Journal of Production Economics* 224, S. 107564. DOI: 10.1016/j.ijpe.2019.107564.

Pansart, Lucie, Nicolas Catusse und Hadrien Cambazard (Dez. 2018). "Exact Algorithms for the Order Picking Problem". In: *Computers & Operations Research* 100, S. 117–127. ISSN: 03050548. DOI: 10.1016/j.cor.2018.07.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0305054818301862 (besucht am 18. 03. 2026).

Petersen II, Charles G. (Nov. 1997). "An Evaluation of Order Picking Routeing Policies". In: *International Journal of Operations & Production Management* 17.11, S. 1098–1111. ISSN: 0144-3577. DOI: 10.1108/01443579710177860. URL:

https://www.emerald.com/ijopm/article/17/11/1098-1111/147537 (besucht am 18. 03. 2026).

Prunet, Thibault, Nabil Absi und Diego Cattaruzza (2025). "A Note on the Complexity of the Picker Routing Problem in Multi-Block Warehouses and Related Problems". In: *Annals of Operations Research* 347, S. 1595–1605. DOI: 10.1007/s10479-025-06481-3. arXiv: 2312.01857.

Ratliff, H. Donald und Arnon S. Rosenthal (1983). "Order-Picking in a Rectangular Warehouse: A Solvable Case of the Traveling Salesman Problem". In: *Operations Research* 31.3, S. 507–521. DOI: 10.1287/opre.31.3.507. URL: http://www.jstor.org/stable/170620.

Roodbergen, Kees Jan und René De Koster (2001a). "Routing Methods for Warehouses with Multiple Cross Aisles". In: *International Journal of Production Research* 39.9, S. 1865–1883. DOI: 10.1080/00207540110028128.

– (2001b). "Routing Order Pickers in a Warehouse with a Middle Aisle". In: *European Journal of Operational Research* 133.1, S. 32–43. DOI: 10.1016/S0377-2217(00)00177-6.

Rosenkrantz, Daniel J., Richard E. Stearns und Philip M. Lewis II (1977). "An Analysis of Several Heuristics for the Traveling Salesman Problem". In: *SIAM Journal on Computing* 6.3, S. 563–581. DOI: 10.1137/0206041.

Scholz, André u. a. (2016). "A New Mathematical Programming Formulation for the Single-Picker Routing Problem". In: *European Journal of Operational Research* 253.1, S. 68–84. DOI: 10.1016/j.ejor.2016.02.018.

Serdyukov, Anatoliy I. (1978). "O nekotorykh ekstremal'nykh obkhodakh v grafakh [On Some Extremal Walks in Graphs]". In: *Upravlyaemye Sistemy* 17, S. 76–79. URL: http://nas1.math.nsc.ru/aim/journals/us/us17/us17_007.pdf.

Theys, Christophe u. a. (2010). "Using a TSP Heuristic for Routing Order Pickers in Warehouses". In: *European Journal of Operational Research* 200.3, S. 755–763. DOI: 10.1016/j.ejor.2009.01.036.

Valle, Cristiano Arbex, John E. Beasley und Alexandre Salles da Cunha (Nov. 2017). "Optimally Solving the Joint Order Batching and Picker Routing Problem". In: *European Journal of Operational Research* 262.3, S. 817–834. DOI: 10.1016/j.ejor.2017.03.069.

Weidinger, Felix (Juli 2018). "Picker Routing in Rectangular Mixed Shelves Warehouses". In: *Computers & Operations Research* 95, S. 139–150. DOI: 10.1016/j.cor.2018.03.012.

Weidinger, Felix und Nils Boysen (Dez. 2018). "Scattered Storage: How to Distribute Stock Keeping Units All Around a Mixed-Shelves Warehouse". In: *Transportation Science* 52.6, S. 1412–1427. DOI: 10.1287/trsc.2017.0779.

Zhang, Ren-Qian, Meng Wang und Xing Pan (März 2019). "New Model of the Storage Location Assignment Problem Considering Demand Correlation Pattern". In: *Computers & Industrial Engineering* 129, S. 210–219. DOI: 10.1016/j.cie.2019.01.027.

Anhang A

Anhang

A.1 Reproduktion der Ergebnisse

Sämtliche in dieser Arbeit berichteten Messwerte lassen sich aus dem beigelegten Quellcode neu erzeugen. Der Ablauf besteht aus zwei Schritten: Container bauen und Benchmarks ausführen.

Der Quellcode der Webanwendung ist öffentlich verfügbar unter `https://github.com/ConstiCode/bachelor_thesis`. Die in dieser Arbeit berichteten Messwerte wurden mit dem Stand des Commits `2d660ee` auf dem Branch `master` erzeugt. Spätere Commits betreffen das Makefile, den Entrypoint, die Dokumentation des Containers, die Weboberfläche und das Entfernen ungenutzter Codepfade. An Lagermodell, Benchmark-Runner und Payload-Dateien ändert sich nichts, die Messlogik der Algorithmen bleibt unverändert. Dem im SVN abgegebenen Quellpaket liegt der neuere Stand `64d8068` bei, zusammen mit den Payload-Dateien, dem Benchmark-Runner und dem Auswertungsskript. Der Container basiert auf dem Image `python:3.12-slim`. Das Dockerfile liegt im Wurzelverzeichnis und installiert die Abhängigkeiten aus `requirements.txt` und `requirements-dev.txt`. Die Befehle zum Bauen und Starten stehen zum Kopieren als Kommentare am Ende des Dockerfiles:

```
docker build -t oprp-bench .
```

Der Container erklärt bei jedem Start, wie es weitergeht. Welche Schritte möglich

sind, listet zudem das Makefile. Es nennt für jedes Ziel die gelesenen und erzeugten Dateien, die ungefähre Laufzeit und den ungefähren Speicherbedarf:

```
docker run --rm oprp-bench make help
```

Erhoben wurden die Messreihen mit `bench_night_runner.py`. Dieser Runner spiegelt die Schleifen des Endpunkts `/benchmark` aus `app.py`, ruft die Solver aber unmittelbar über `run_solver_capped` auf und umgeht damit die HTTP-Schicht. Die Zeitnahme ist auf beiden Wegen dieselbe, da sie innerhalb von `run_solver_capped` im Solver-Subprozess liegt:

```
mkdir -p results
docker run --rm -v "$PWD/results:/data" \
    oprp-bench make b1 BENCH_OUT_DIR=/data
```

Analog verhält sich dieser Befehl für `b2` und `b3`. Jeder Aufruf liest `benchmarks/payloads/<name>.json` und schreibt die ausgewertete Messreihe als `<name>.csv` in das über `BENCH_OUT_DIR` gesetzte Verzeichnis, standardmäßig `benchmarks/results/`. Ein vorhandener Datensatz wird dabei nie überschrieben, sondern ergänzt. Diese drei CSV-Dateien sind die Datengrundlage aller Tabellen und Abbildungen in Kapitel 5. Derselbe Versuchsplan lässt sich alternativ über die Weboberfläche fahren. Der erste Befehl startet die Anwendung, die unter `http://127.0.0.1:5000` den Endpunkt `/benchmark` bereitstellt, der zweite spielt die Payloads dagegen ein:

```
docker run --rm -p 127.0.0.1:5000:5000 oprp-bench
benchmarks/run_bench_http.sh b1
```

Dieser Weg erzeugt dieselben Messreihen, war an der berichteten Messung aber nicht beteiligt.

Die Payload-Dateien legen die Versuchsbedingungen fest und sind dem Abgabepaket beigelegt. Für alle drei Benchmarks gelten dieselben Parameter: Produktanzahlen von 5 bis 30 in Schritten von 5, 20 Iterationen je Kombination,

ein Basisseed von 42 und eine Timeout-Schranke von 90 Sekunden. Zusätzlich begrenzt die Anwendung jeden Solver-Subprozess auf 12 GB Adressraum. Dieser Wert ist in `app.py` hinterlegt und nicht über die Payload steuerbar. Aus dem Basisseed leiten sich die 20 Iterationen einer Konfiguration als Seeds 42 bis 61 ab. Dadurch erhalten alle drei Algorithmen in jeder Iteration exakt dieselbe Pickliste, und ein Wiederholungslauf erzeugt dieselben Ergebnisse.

Die Abbildungen des Evaluationskapitels werden aus den CSV-Dateien mit `make_plots.py` erzeugt. Dieses Skript benötigt zusätzlich `numpy`, `pandas` und `matplotlib`. Die verwendeten Versionen sind in `requirements-plots.txt` festgehalten und werden nicht im Benchmark-Container installiert, da sie für die Messung selbst nicht erforderlich sind.

Zur Absicherung der Implementierung dienen zwei weitere Aufrufe. Das Skript `verify_optimality.py` erzeugt deterministisch 182 Zufallsinstanzen mit bis zu 5 Regalspalten, 3 Regalreihen und 10 Produkten. Diese Obergrenze folgt aus dem exponentiellen Aufwand des Referenzlösers. Für jede Instanz prüft das Skript, ob der Fixed-Parameter-Algorithmus exakt das Optimum dieses Referenzlösers trifft. Dieser bestimmt die Distanzen per Breitensuche auf dem Gitter und löst darauf mit dem Verfahren von Held und Karp. Zusätzlich stellt das Skript sicher, dass Nearest Neighbor und Christofides nie unterhalb des Optimums liegen.

Die beiden Gegenprüfungen der Lagerdistanz-Formel aus Abschnitt 4.7.1 sind Teil der Testsuite und laufen mit ihr. Die Testsuite umfasst 126 Tests. Sie läuft im selben Container, dessen Image die Testabhängigkeiten bereits enthält. Die Test Suites werden mit zwei Befehlen gestartet:

```
python verify_optimality.py
pytest tests/
```

Ein erfolgreicher Lauf meldet `182/182 Instanzen: FixedParameter == Optimum` und endet mit Rückgabewert 0.

A.2 Softwareumgebung der Benchmarks

Alle Benchmarks wurden im Docker-Container auf Basis des Images `python:3.12-slim` mit Python 3.12.13 ausgeführt. Die folgende Tabelle protokolliert die verwendeten Paketversionen, erzeugt mit `pip freeze` am 06.08.2026. Direkt festgelegt sind darin `Flask`, `networkx` und `PuLP`. Die übrigen Einträge sind transitive Abhängigkeiten und dokumentieren den Stand des Messzeitpunkts.

Paket	Version
`blinker`	1.9.0
`click`	8.4.2
`Flask`	3.1.3
`itsdangerous`	2.2.0
`Jinja2`	3.1.6
`MarkupSafe`	3.0.3
`networkx`	3.6.1
`PuLP`	3.3.2
`Werkzeug`	3.1.8

Tabelle A.1: Python-Paketversionen der Benchmark-Umgebung

A.3 Optimalitätsgap je Konfigurationsstufe

Kapitel 5 berichtet den Optimalitätsgap je Benchmark aggregiert. Die folgenden Tabellen schlüsseln ihn zusätzlich nach den einzelnen Konfigurationsstufen auf und belegen damit die im Text genannten Verläufe über Breite, Regalreihen und Lagergröße. Berücksichtigt sind jeweils nur Instanzen, in denen der Fixed-Parameter-Algorithmus ohne Timeout terminierte. Die Spalte Instanzen weist die zugrunde liegende Anzahl aus. Die Stufen oberhalb von $r = 7$ gehen aus diesem Grund nicht in die aggregierten Tabellen des Evaluationskapitels ein. In Benchmark B1 tritt kein Timeout auf. Dass Breite 1 und 2 dort weniger Instanzen umfassen, liegt daran, dass in diesen Layouts nur 5 bis 10 bzw. 5 bis 20 Produkte Platz finden. Dasselbe gilt für $r = 1$ in Benchmark B3. Die Aufschlüsselung für Benchmark B1 findet sich in Tabelle A.2, für B2 in Tabelle A.3 und für B3 in Tabelle A.4.

Breite	NN-Gap (%)	CH-Gap (%)	Instanzen
1	0,25	0,00	40
2	4,14	2,10	80
3	2,16	2,71	120
4	3,75	3,89	120
5	3,14	3,98	120
6	5,56	3,91	120
7	4,33	3,96	120
8	4,82	4,26	120
9	4,80	3,87	120
10	5,05	4,06	120

Tabelle A.2: Mittlerer Optimalitätsgap je Lagerbreite in Benchmark B1 ($r = 1$)

r	NN-Gap (%)	CH-Gap (%)	Instanzen
1	5,05	4,06	120
2	9,85	7,10	120
3	12,65	7,66	120
4	12,61	8,03	120
5	13,25	8,27	120
6	12,87	8,01	120
7	14,94	9,42	120
8	11,83	6,93	24
9	0,00	0,89	4
10	0,00	1,79	2

Tabelle A.3: Mittlerer Optimalitätsgap je Regalreihenzahl in Benchmark B2 (Breite 10)

r	NN-Gap (%)	CH-Gap (%)	Instanzen
1	0,25	0,00	40
2	9,88	4,43	120
3	14,67	9,17	120
4	15,95	9,39	120
5	14,62	9,45	120
6	17,10	8,75	120
7	14,24	9,04	120
8	12,76	5,72	22
9	5,87	6,64	8
10	0,00	1,79	2

Tabelle A.4: Mittlerer Optimalitätsgap je Lagergröße in Benchmark B3