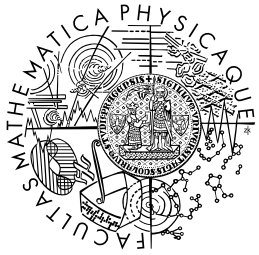

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jiří Krejčí

Multi-agent picker routing problem

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: prof. RNDr. Roman Barták, Ph.D.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

First and foremost, I am grateful to my supervisor, prof. RNDr. Roman Barták, Ph.D., for his guidance, patience and support. I would like to thank you very much for the valuable advice you gave me throughout the process.

I would like to express my deepest gratitude to my parents, who provided me with unfailing support throughout my years of study. Special thank you goes to my grandfather, who introduced me to the diverse world of science and sparked my interest in mathematics, physics and technology.

Finally, I am sincerely thankful to all my friends. I'm glad to have you all in my life. Your endless encouragement and support really mean a lot to me.

Title: Multi-agent picker routing problem

Author: Jiří Krejčí

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: An important part of warehouse operations is order picking, which is the process of collecting products from stocking locations. In our case, a scattered storage warehousing strategy is assumed. Items are stored at multiple locations scattered through the warehouse. Usually, multiple order pickers are responsible for a quick collection of items. That can result in order pickers blocking each other, reducing the picking throughput. Most of the existing picker routing algorithms are not concerned with picker blocking, even though its effect on picking efficiency can be substantial. It is the objective of this thesis to address the picker routing problem in a multi-agent environment. First, the literature is surveyed and then a multi-agent picker routing algorithm is presented. The algorithm is based on the idea of prioritized planning. Results of the empirical evaluation indicate that the multi-agent approach leads to better quality solutions.

Keywords: multi-agent, logistics, route planning

Contents

Introduction

Order picking, which is the process of retrieving items from storage to meet customer demand, and warehouse replenishment are critical functions to each supply chain. In prior studies, these tasks have been identified as the most labor-intensive and costly activities for almost every warehouse [1]. The cost of order picking is estimated to be as much as 55% of the total warehouse operating expense [2]. Even though the use of automated equipment in the distribution centers is on the rise and has the potential to lower the cost of order picking over time, manual order picking is still prevalent. This thesis is motivated by manual warehouses, but the results may be relevant for automated warehouses as well.

In its essence, the order picking problem is a variant of the well-known Traveling Salesman Problem. If we think of pick locations as cities, then the task becomes: given a pick list, plan a tour through the warehouse that visits every location on the pick list exactly once. In the past, researchers have developed many approaches to solve this special case of TSP, so that we could solve real-world instances fast enough. One of the most influential is the exact polynomial time algorithm that was developed by Ratliff and Rosenthal in 1983 [3]. There is a caveat though, it solves the problem for one specific warehouse layout only. Because the algorithm lacks the flexibility to be used in many real-world scenarios, it has been extended multiple times by other authors. Even though the exact algorithm is developed, it is the heuristic approach that is still arguably the most commonly used in practice. These routing policies [4] — despite frequently leading to suboptimal routes — are very quickly computed, easy to implement, and can be followed effortlessly by order pickers. Heuristic algorithms can often be described by a simple, easy-to-remember rule.

However, in a typical warehouse, there are multiple order pickers working at the same time. Up until this point, none of the routing policies considered interactions between pickers. When multiple pickers are given similar or intersecting routes, the pickers start to block each other, which results in congestion. In recent years, many studies have been conducted on the effects of picker blocking. Authors often develop an analytical model or simulation to study, which factors are the most contributing to picker blocking. Unsurprisingly, the number of pickers, storage location assignment policy, and warehouse layout are on top of the list [5].

But how much efficiency is lost due to picker blocking? First, we have to define the efficiency of order picking. One way to define it is as throughput — the number of completed orders (per period of time). An alternative metric could be the sum of the time spent waiting by order pickers. With efficiency defined, we can look at specific studies. One of the most comprehensive publications on the topic is written by Huber [5]. He concludes that congestion accounts for the throughput loss of about 30% on average and the majority of experiments had throughput loss of at least 15%. In a recent study, Klodawski et al. [6] were more interested in a relationship between the number of agents in a warehouse and picker blocking. The time spent blocking was roughly 5% when only three agents were in a warehouse, but it rose to ca. 30% when the number of agents was 9. Finally, Elbert et al. [7] compared different routing heuristics in their work.

The choice of routing heuristic alone had a big impact on throughput times and blocking. Use of the worst performing heuristic resulted in more than 30% drop in throughput compared to the best.

Given the significant throughput loss due to congestion, surprisingly little research has been done on picker routing with congestion consideration. It is the goal of this thesis to survey existing picker routing algorithms, the algorithms from the multi-agent path finding realm, and to develop an algorithm based on this research that will compute routes for order pickers while avoiding mutual blocking at the same time. We have determined that prioritized planning with a custom solver for individual tours offers the best trade-off between computation times and solution quality.

The structure of this thesis directly mirrors the goals we have set. In the first chapter, we will describe the real-world warehouse layout and processes to give us the necessary understanding of the warehousing domain. The problem to be solved will be defined and related problems will be introduced. In the second chapter, we will go through related work, including picker routing and multi-agent path finding. In the third chapter, we will analyze the possible solution approaches and we will give reasons why or why not to use the particular approach to solve our problem. In the fourth chapter, we will introduce the custom GTSP solver. This solver is used in a prioritized planning algorithm. We will introduce and test a few heuristics to improve the performance of the prioritized planning algorithm. Finally, in the fifth chapter, the performance of the algorithm will be evaluated empirically.

1. Multi agent picker routing

In this chapter, we describe the real-world warehouses that motivate this thesis and we introduce the necessary terminology. Next, the formulation of the problem will follow. Last, we introduce related problems in this chapter, namely the variants of the Travelling Salesman Problem and the Multi-agent path finding.

1.1 Real-world warehouse specification

Consider a parallel-aisle warehouse as shown in the Figure 1.1 for illustration. The warehouse has narrow vertical aisles, which means that order pickers cannot pass each other in an aisle, and several horizontal cross aisles. The area between two neighboring cross-aisles is called a block. Based on the number of cross-aisles, we distinguish between single-block and multi-block warehouse layouts. The warehouse has a high bay storage area, meaning its storage racks have pallets stored at multiple height levels. In addition to the main storage area, the warehouse may also contain special zones, e.g. freezers and regions separated by walls that are connected to the main grid. There are two areas that are connected to the Input/Output ports of the warehouse — the inbound and outbound staging areas. The inbound staging area serves as a buffer for new products arriving to the warehouse that are waiting to be restocked. The outbound area has a similar purpose, but in reverse — it's where completed customer orders are taken after they have been picked up in the storage area and are waiting to be loaded into trucks for delivery. In the literature, a single depot is more common for this purpose. Products are stored in aisles on both sides. One distinct product type (SKU - stock keeping unit) can be stored at multiple locations in the warehouse, which is referred to as scattered storage in the industry.

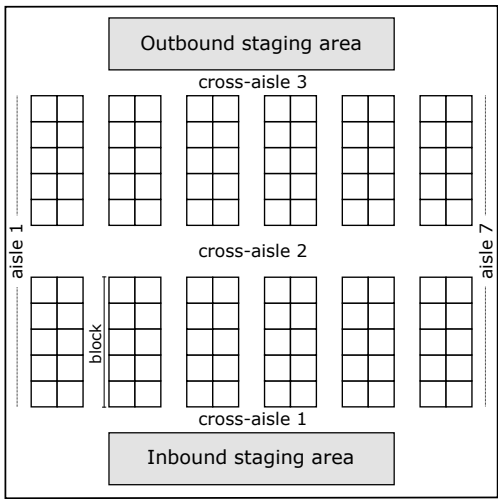

Figure 1.1: Warehouse layout

Figure 1.2: High bay storage

People or autonomous agents navigate through the warehouse. We assume, that agents travel at a constant speed. There are two kinds of tasks we consider — order picking and warehouse replenishment. To recapitulate, order picking is the process of retrieving items from storage, usually given to the agent on the

so-called pick list. The reverse process is called warehouse replenishment. From our perspective, the only difference between the two processes is the staging area the agent begins or ends his tour at. Therefore, to simplify the terminology, we will refer to all people performing these two tasks as order pickers, or agents more generally.

During the workday, order pickers are assigned customer orders. We assume, that the orders are already assigned to individual pickers, including the sequence for each picker, in which the orders have to be completed. It is worth noting, that the order assignment (or order batching) alone is generally a hard task, which can have a big impact on the efficiency of the operation. For further reading, see the article by Scholz and others [8]. Agents walk or ride through the warehouse with a picking device and pick items from a pick list, or replenish items respectively. We assume that the picking device has enough capacity to carry all items specified in any customer order. As previously mentioned, the aisles are narrow, meaning that the order picker can reach product racks on both sides of an aisle without any time penalty, but one order picker cannot pass another in the aisle.

Even though we don't have any particular warehouse instances specified, part of the task specification is the upper bound on the size of the problem. There can be up to 100 000 pick locations in the warehouse. The number of order pickers working in the warehouse at the same time can range from 2 to 100, depending mostly on the warehouse size. With approximately 10 customer orders per picker per workday, the number of customer orders can reach up to ca. 1000 per day. Any single customer order can consist of at most 14 unique items, but most of the orders will consist of less than 10 different items. In some instances, where the items stored in the warehouse are large (e.g. pallets of building material), the number of items in an order will be usually close to one. The number of agents and the number of items per order will be particularly important for the algorithm. As we will see later, these two values have the biggest impact on the complexity of the problem.

1.2 Problem input

We will neglect the real-world specifics for now and focus on the abstract view of the problem only. This includes representing the physical warehouse as a graph as illustrated in Figure 1.3. The input to the abstract multi-agent picker routing problem is:

1. A directed graph $G = (V, E)$, which is assumed to represent a 4-connected grid with obstacles (storage racks, walls, etc.). The vertices of the graph are all possible locations for the agents, and the edges are possible transitions. All edges have unit weight.

2. Set of m tasks $O = \{O_1, ..., O_m\}$. Each task is specified by a start vertex, $start_{O_i} \in V$, a goal vertex, $goal_{O_i} \in V$ and a collection of sets, where the sets contain vertices from G. For each vertex and an item pair (v, i), such that item i is stored at vertex v, pick time $t_{vi} \geq 0$ is defined, which is the time needed to pick item i at vertex v.

3. k agents $a_1, ..., a_k$.

4. For each agent $a_k \in A$, an ordered list of tasks R_k is given, such that every task $O_i \in O$ is contained precisely in one list R_j, $j \in \{1, ..k\}$.

Time is discretized into time steps. We assume without loss of generality that the start vertex of any task is identical to the goal vertex of the previous task in the corresponding ordered list if such task exists.

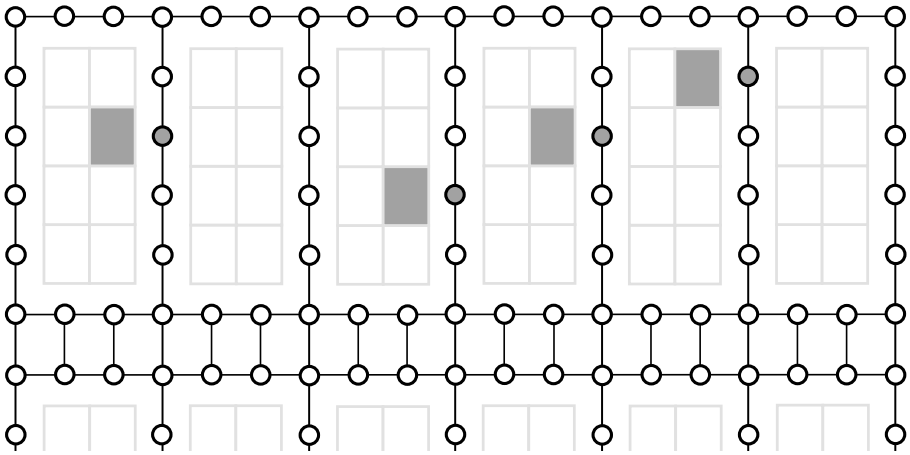

Figure 1.3: Graph representation of a warehouse

1.3 Problem formulation

Let the input to the problem be of the same format as defined above. Each agent occupies a single vertex at any given time and similarly, each vertex and edge can be used by at most one agent at any given time step. Agents can perform one of the three actions at a time - *stay* at the vertex, *move* to an adjacent vertex and finally, perform a *pick* action.

Every agent is given a unique list of tasks. Each task contains a start location, a goal location, and a collection of sets. Elements of the sets are pick locations. From each set, exactly one location has to be *picked*. Picking an item at a location takes some positive time t_{vi} specific for each vertex and item pair (v, i) — an agent must stay at least t_{vi} time units at the vertex. For the other vertices and regular movement actions, such conditions do not apply and an agent can stay at the vertex for arbitrarily many time steps.

Agents perform the tasks one at a time and after an agent finishes a task, he immediately begins to perform the next one, if there is one. We assume that the agent disappears after his last task is finished. The order in which the tasks are performed is given.

The goal is to find a tour for each task and agent, such that exactly one vertex from each set is visited for a given task and the tours are minimizing the sum of costs objective function (defined in the next section).

1.4 Related problems and definitions

For a single agent and task, the problem of finding a tour is closely related to Travelling Salesman Problem, even though there are some substantial differences. Let us consider graph representation of a warehouse (see Figure 1.3). The gray nodes are pick locations of one task. If we neglect the scattered storage assumption and we assume that each item is stored at exactly one location instead, then we can classify the problem as a variant of TSP called Steiner TSP.

Definition 1 ([9] Steiner TSP)**.** *Given a list of cities, some of which are required to be visited, and the lengths of roads between them, the goal is to find the shortest possible walk that visits each required city and then returns to the origin city. Vertices may be visited more than once and edges may be traversed more than once as well.*

In the definition, an agent must return to the origin city. Generally, agents in the warehouse may receive orders, that have different start and finish locations. We can neglect this difference from a theoretical standpoint, as it doesn't change the complexity of the problem. The number of permutations of cities is still the same, only the route to the final location is different. For practical purposes, we could add a zero cost edge from the final node to the start node, and an edge with an infinite cost from the final node to all other nodes, which would force the final node as the last node, before returning to the start.

Steiner TSP can be converted into a classical TSP instance. First, shortest paths between all required nodes are computed, which yields a new edge set. The TSP instance is a complete graph of required nodes and this new edge set. The conversion is polynomial, since the algorithm computes a quadratic number of shortest paths and each path takes polynomial time to compute.

Let us return to the original assumptions of the scattered storage. Even though there is no direct equivalent to the Steiner TSP applicable to scattered storage that would also keep the intermediate nodes, there is a TSP variant that can be used, if we keep only the picking nodes – Generalized TSP (GTSP). Generalized TSP is also known as the Set TSP. The problem is NP-hard since the TSP is a special case of GTSP when the size of all sets is equal to one.

Definition 2 (GTSP)**.** *Let $G = (V, E)$ be a weighted complete directed graph on N vertices and let $V = V_1 \bigcup ... \bigcup V_m$ be a partition of the vertices into M disjoint sets. The objective is to find a minimum weight cycle containing exactly one vertex from each set of the partition.*

Again, we cannot just create a GTSP instance from the input without any modifications. We can solve the problem with different start and finish locations in the same way as was mentioned with STSP. Another obvious step would be to calculate the shortest distances between all pick vertices. Since the storage is high bay and one node can correspond to two racks, there can be multiple different items stored at a single pick location. This can be resolved by making a copy of the vertex for each SKU and assigning them into the sets of the given SKU. An example of a GTSP instance is illustrated in Figure 1.4.

However, our problem is defined for multiple agents, so we need to define terminology and problems from the multi-agent research area as well.

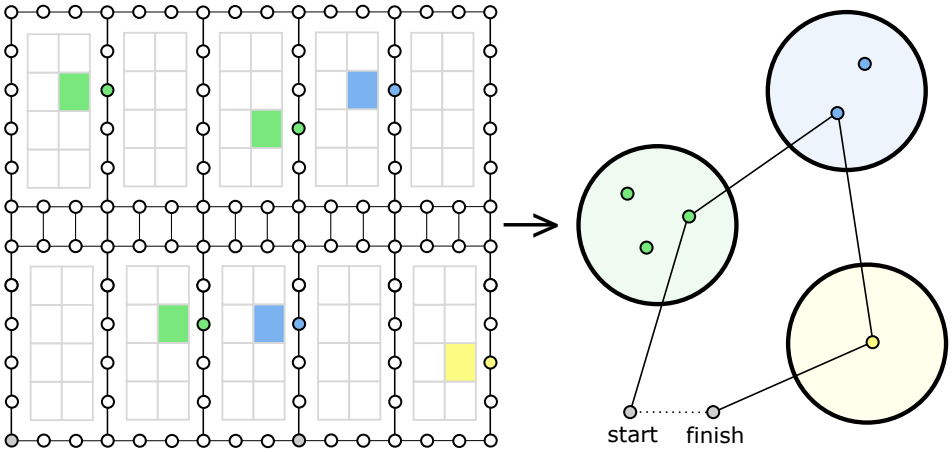

Figure 1.4: GTSP illustration with possible solution

Definition 3 ([10] Multi agent path finding)**.** *The input to a classical MAPF problem with k agents is a tuple $\langle G, s, t\rangle$, where $G = (V, E)$ is an undirected graph, $s : [1, ..., k] \to V$ maps an agent to a source vertex, and $t : [1, ..., k] \to V$ maps an agent to a target vertex. Time is assumed to be discretized, and in every time step each agent is situated in one of the graph vertices and can perform a single* action*. An action is a function $a : V \to V$, such that $a(v) = v'$ means that if an agent is at vertex v and performs a then it will be in vertex v' in the next time step. Each agent has two types of actions:* wait *and* move*. A* wait *action means that the agent stays in its current vertex for another time step. A* move *action means that the agent moves from its current vertex v to an adjacent vertex v' in the graph.*

For a sequence of actions $\pi = (a_1, ..., a_n)$ and an agent i, we denote by $\pi_i[x]$ the location of the agent after executing the first x actions in π, starting from the agent's source $s(i)$. Formally, $\pi_i[x] = a_x(a_{x-1}(...a_1(s(i))))$. A sequence of actions π is a single-agent plan for agent i iff executing this sequence of actions in $s(i)$ results in being at $t(i)$. A solution is a set of k single-agent plans, one for each agent.

The concept of collision will be introduced separately now. For the purposes of this thesis, we are interested in two types of *conflicts* - the *vertex conflicts* and the *swapping conflicts*. Finally, we have yet to define an objective function for future comparison and evaluation of the solutions. We will work with two objective functions throughout the thesis — the makespan, and the sum of cost.

Definition 4 ([10] Vertex conflict)**.** *A* vertex conflict *between π_i and π_j occurs iff according to these plans the agents are planned to occupy the same vertex at the same time step.*

Definition 5 ([10] Swapping conflict)**.** *A* swapping conflict *between π_i and π_j occurs iff the agents are planned to swap locations in a single time step.*

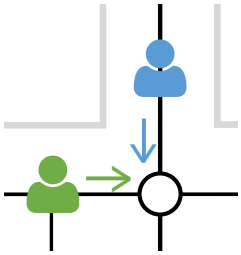

Figure 1.5: Vertex conflict

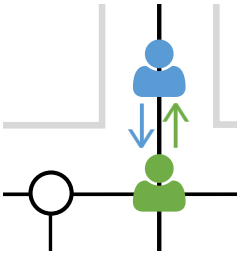

Figure 1.6: Swapping conflict

Definition 6 ([10] MAPF objective funtctions)**.** *Let $\pi = \{\pi_1, ..., \pi_k\}$ be a MAPF solution.*

1. ***Makespan.*** *The number of time steps required for all agents to reach their target. The makespan of π is defined as $max_{1 \leq i \leq k} |\pi_i|$.*

2. ***Sum of costs.*** *The sum of time steps required by each agent to reach its target. The sum of costs of π is defined as $\sum_{1 \leq i \leq k} |\pi_i|$.*

The multi-agent path finding problem has many variations and extensions, but the definitions above will suffice in providing the terminology and concepts needed for the purposes of this thesis. MAPF might be the most closely related problem of the three mentioned. Suppose that for some agent a_i, $\pi = \{\pi_1, ..., \pi_k\}$ is a permutation of vertices that are required to be visited by the agent a_i in a given task. To convert the GTSP solution into a single-agent plan, we just need to solve $k+1$ single-agent path finding problems and concatenate the paths into a single path, or, to be more precise, a walk. Likewise, if we look at the problem from a different perspective, the difference lies in the fact, that instead of simple and non-conflicting routes, we search for non-conflicting tours for the agents instead.

Algorithm A*

Now, we will take a small sidestep to introduce an algorithm, that is not directly related to the main problem, but it will show up later in the thesis on multiple occasions. The algorithm we are talking about is the A* search algorithm, first published in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael [11]. It can be seen as an extension of Dijkstra's algorithm that selects the next node to expand not only based on the cost of the path from the start node, but based on the heuristic approximation of the cost to get to the goal node as well. Written mathematically, it selects a node that minimizes $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the start node to n and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.

An important property is the admissibility of a heuristic function. We say the heuristic function is admissible if it never overestimates the actual cost to get to the goal. If we choose such heuristic function, then the A* algorithm always finds an optimal solution.

2. Related work

In the previous chapter, we have established that generally, the picker routing problem can be classified as a variant of the Traveling Salesman Problem. More precisely the Steiner TSP variant without the assumption of scattered storage and Generalized TSP with scattered storage. Furthermore, we have identified the Multi-agent path finding research area as helpful when dealing with multiple agents, providing the needed terminology and as we will see soon, general frameworks for designing multi-agent algorithms as well. In this chapter, we will survey the literature on all related problems.

The goal of this survey is to get familiar with the wide variety of approaches used in literature. As we will see, the diversity of approaches is astonishing — ranging from exact algorithms designed for special kinds of instances only to soft-computing techniques such as Genetic algorithms. The focus is not on providing a detailed description of the individual approaches, but on covering as much of the approaches as possible while trying to take away the key concepts. That should help us with a proposal of an algorithm to solve the Multi-agent picker routing problem. We will analyze the approaches with this task in mind in the next chapter.

2.1 Picker routing - specialized

The first group of algorithms we will look at are the specialized algorithms for the picker routing problem. Specialized in this context means that the algorithms are often designed for a single purpose only, which might give them an advantage over more general algorithms, but at the cost of versatility. Nevertheless, algorithms of this kind seem to be the most widely used in practice to this date.

There are number of issues we have to look out for when reviewing the literature on the picker routing problem that stem from the narrow specialization of the algorithms. Usually, the algorithm is developed for a certain warehouse layout and cannot be used with any other. Most of the time, even less significant aspects like depot position matter, even though small modifications of the algorithm might fix some of the lesser issues. But the biggest issue we are facing is that most of the algorithms in the literature are not designed for scattered storage policy. And something that should be considered is that the performance of the algorithms can vary from instance to instance. For example, some heuristic algorithm might calculate near-optimal routes when customer orders are larger, but it could perform poorly otherwise.

2.1.1 Exact algorithms

As it was mentioned in the previous chapter, the picker routing problem is generally NP-hard. Nevertheless, optimal algorithm linear in the number of aisles was developed by Ratliff and Rosenthal [3] for the single-block warehouse layout. The authors assumed a low-level storage rack, single depot in the front cross-aisle and narrow picking aisles. Ratliff and Rosenthal used a dynamic programming approach, gradually building *partial tour subgraphs* (PTS) from left to right. In

short, PTSs can be described as subgraphs, that can be extended into a valid tour. The authors have noticed, that the PTSs can be divided into a fixed number of equivalence classes. Therefore, the algorithm needs to consider only an fixed number of minimum length PTSs at any given step and the number of steps is polynomial in the number of aisles in the warehouse.

Since warehouses often have more complex layouts, the extension of the algorithm to three cross aisles followed [12]. The latest contribution was introduced by Pansart et al. [13], who applied a dynamic programming algorithm for the rectilinear TSP developed by Cambazard and Catusse [14] and also formulated a compact Integer Linear Program for the problem. However, the complexity of the dynamic programming algorithm increases rapidly with the number of cross aisles. More specifically, the time complexity of the algorithm is $\mathcal{O}(nh7^h)$, where n is the number of vertices and h is the number of horizontal lines (cross aisles). The authors experimentally compared the ILP approach, solving the TSP with Concorde solver (more about Concorde later) and the dynamic programming approach. The dynamic programming algorithm was the fastest for up to 6 cross-aisles, but it failed to solve instances with 11 cross-aisles, while the Concorde solved all test instances within one minute. Concorde performed better on the test instances than CPLEX Optimizer that solves linear programs.

2.1.2 Heuristics

Another possible, and arguably the most frequently used practice for calculating picking routes is by the means of heuristic algorithms. Unlike the exact algorithms, heuristics are not guaranteed to find the shortest tour possible, but this attribute is offset by the ease of implementation and faster calculation times. Moreover, heuristic rules are usually easy to remember and follow for order pickers, who might struggle with following more complex routes.

For narrow-aisle warehouses, Hall proposed and evaluated performance of three simple heuristic routing strategies [4], namely the *Traversal* (also called *S-Shape*) strategy, *Midpoint* and *Largest Gap* strategy. We will briefly describe the traversal strategy for illustration. The traversal strategy tells the picker to cross through the entire length of any aisle containing at least one pick location. The other strategies can be described in an analogous manner. Later, the *return* and the *composite* heuristics were further developed [15]. What all of these heuristics have in common is that they were originally developed for a single-block layout only.

An extension to a multi-block layout for S-Shape and Largest gap heuristics was introduced by Roodbergen and de Koster, along with the new *combined* heuristic [16]. Chen et al. developed two modifications of S-shape heuristics [17], that attempt to avoid congestion. In the first modification, this is achieved by the condition that if an aisle is already occupied by one picker, the other must wait at the entrance of the aisle. The other modification considers spatial relationships between the picked item and the next item to determine the travel time and the waiting time. If congestion occurs, it can dynamically reroute the picker. These heuristics were introduced as a benchmark for the Ant Colony Optimization algorithm, which was developed in the same paper as well.

Recently, Weidinger developed a new heuristic approach to address routing

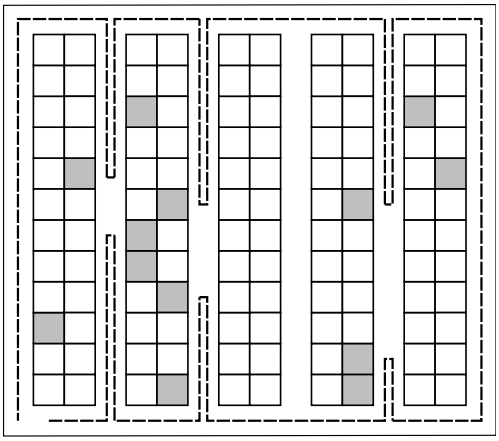

Figure 2.1: Midpoint heuristic

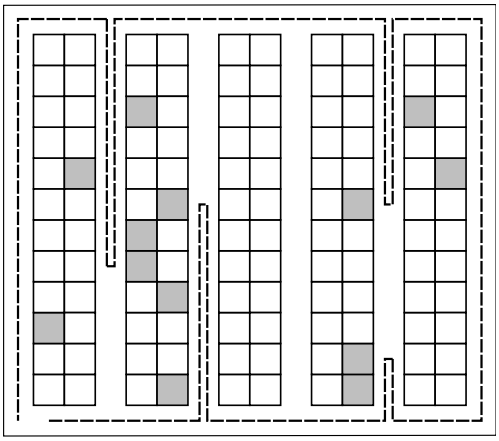

Figure 2.2: Largest Gap heuristic

in scattered storage warehouses [18]. His approach proves to be very efficient, the author states just 0.5% mean optimality gap in the paper. Because of the complexity of the problem, the author partitioned the problem into two parts — storage positions selection and picker routing. The positions selection is done by three simple priority rules, that compute route distances implicitly and are capable of finding the shortest tour for the given positions.

The choice of the best heuristic depends on many factors, among the most important are pick density, number of cross aisles, and storage policy. The difference between heuristic and optimal route in the single-block case can be as low as 1%, but it can get much higher as well [16], especially if the wrong heuristic for the situation is chosen. There are more rule-based heuristics, but as an overview, the list provided is sufficient.

2.1.3 Meta-heuristics

To complete the list of all approaches to the picker routing problem found in literature, we must not forget the meta-heuristics, which gained popularity mostly in recent years. In the single block case, meta-heuristic approaches are used predominantly to solve complex combined optimization problems, for example, the joint order batching and picker routing problem. The exception is a paper, in which the authors used a hybrid genetic algorithm to solve picker routing with product returns and even considered interactions between the order pickers [19].

For multi-block warehouses, the use of meta-heuristic algorithms is more frequent for solving picker routing independently, but the combined approaches are still prevalent. One of the earliest uses of meta-heuristic was by Chen et al. [17], whose work was already mentioned in the heuristics subsection. They applied an Ant Colony Optimization approach to a multi-block narrow-aisle warehouse with two order pickers while also accounting for congestion. The work was extended in 2016 to include an online routing method under non-deterministic picking time [20]. The authors concluded, that the new method can reduce the order service time by coping with the congestion.

Another algorithm, that uses ACO optimization, is FW-ACO [21]. The algorithm is a combination of the ACO meta-heuristic and Floyd-Warshall algorithm. Authors claim, that the algorithm is best suited for complex warehouse configura-

tions, where the shortest path generated by the FW-ACO is usually significantly better than the path returned by regular heuristic algorithms.

2.2 Picker routing - TSP

As we have seen, in literature, authors often present specific algorithms that solve picker routing, but only for instances that satisfy a narrow set of assumptions, e.g. two-block warehouses with a single depot and narrow aisles. Because the assumptions we have are much broader, it is possible that we will have to use a general TSP solver for the picker routing problem.

Solving TSP optimally is no easy feat. One of the earliest attempts was by Dantzig et al. published in the 1954 study [22], who used a cutting-plane algorithm to compute an optimal TSP tour going through 49 American cities. It took 15 more years to find an algorithmic approach for creating these cuts. Other approaches include branch-and-cut or branch-and-bound algorithms. Applegate et al. used these techniques to create a well-known Concorde TSP solver [23]. Concorde was used to solve a TSP instance as large as 85900 cities, and it is probably the fastest exact TSP solver to this date.

There are many heuristic approaches as well. The most prominent are the *improvement heuristics*. The idea is to generate some initial solution and then iteratively improve it using local search operators. Examples of iterative heuristics are the *2-opt* and the *3-opt* (see Figure 2.3) local searches or their generalization — the *Lin-Kernighan* TSP heuristic [24], that can use a general *k-opt* move and tries to find the best choice of k for each move. The general *k-opt* move is carried out by removing k edges from the tour, then the algorithm searches through all of the ways to reconnect the tour and picks the best option. The LK heuristic is said to be one of the most effective methods for generating optimal or near-optimal TSP solutions. The most widely used implementation of the LK heuristic is the Helsgaun's [25], which is referred to as the LKH heuristic. The LKH heuristic is applied in the paper written by Theys et al. [26] for routing order pickers. They compared the tours computed by the LKH heuristic to optimal tours computed by the Concorde solver and the results were very good. The optimality gap was just 0.1% on average.

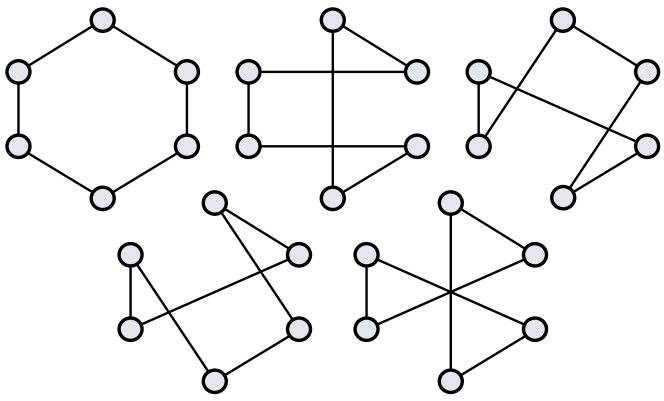

Figure 2.3: 3-opt moves illustration

Special kind of heuristic algorithms are the so-called approximation algorithms. What characterizes the approximation algorithms is that there is a fixed upper bound on the solution quality compared to the optimal solution. For example, the Minimum Spanning Tree (MST) heuristic is a 2-approximation algorithm, meaning that the tour it finds is always shorter than two times the length of the optimal tour. Another example is the $\frac{3}{2}$-approximation Christofides Algorithm. These algorithms are well suited to be used in combination with the improvement heuristics, providing some initial tour in a short time.

The last class of methods used to solve TSP are the Soft-Computing techniques, that include meta-heuristic algorithms. Hore et al. used an improved Variable neighborhood search [27]. Comparison between Ant colony optimization and Genetic algorithm was performed by Hore et al. [28], they recommend GA for low computational resources and smaller problems, while the ACO performed better on larger and complex problems (very sensitive to parameters variation, rely on authors experience to fine-tune the parameters).

2.2.1 GTSP

GTSP instances can be converted to a regular TSP. This was exploited by Helsgaun [29], who used the conversion algorithm along with the LKH TSP solver to produce a GLKH solver. He used it to solve GTSP instances from GTSPLIB. The GTSPLIB is a benchmark library [30], that was developed from TSP-LIB instances by performing clustering on the vertices. Small benchmark instances (tens of sets, up to 200 cities) were solved to optimality very quickly within one second. All large benchmark instances (up to 217 sets and 1084 cities) were solved in a reasonable time (maximal time was 2054 sec). Most of the large instances were solved to optimality, too. Only some of the tours were very slightly longer than optimal. Because the GTSPLIB instances were not challenging enough, the author extended the library with 44 new instances with up to 85900 vertices. The solver found some solution to all of these new instances in a reasonable time, only the optimality gap was much bigger – up to 5%.

There are dedicated GTSP solvers in the literature as well. One such solver is using the branch-and-cut approach [30]. Unfortunately, we cannot compare benchmark results to the GLKH solver, because different test instances are used. But the former approach is most likely superior. The computation times of the dedicated GTSP solver are longer by orders of magnitude even on instances, that would be labeled as medium by Helsgaun (roughly 400 cities). Another dedicated GTSP solver is the memetic solver called GK [31]. To cite the authors, memetic algorithms combine the powers of genetic and local search algorithms. The GK solver is not the first iteration of the memetic algorithm, but more of an evolution of the approach. The last dedicated solver we will mention is the heuristic GLNS [32], which operates under the general framework of adaptive large neighborhood search.

All three heuristic solvers, the GK, the GLNS, and the GLKH yield impressive performance on GTSPLIB with fast solve times and a consistently small optimality gap. The authors of the GLNS presented a comparison on other benchmark libraries as well and they have come to a conclusion, that the GLNS solver outperforms both the GK and the GLKH on most of the instances.

2.2.2 Steiner TSP

Even Steiner TSP got some attention from researchers recently. Letchford et al. proposed compact integer linear programming formulation [33]. For larger instances, a heuristic approach might be more viable. A greedy algorithm with heuristic was developed by Interian and Ribeiro [34]. Finally, there is a possibility to convert Steiner TSP to classical TSP. That is exactly what Eduardo Álvarez-Miranda and Markus Sinnl [35] did in their work, using the state-of-the-art TSP solver Concorde. They also provided a comparison of the three mentioned approaches. The Concorde-based approach outperformed both the exact and heuristic approach by a large margin. Compared to the heuristic approach, not only was the computation time by three orders of magnitude smaller on large instances, but it achieved this speed-up while providing optimal solutions. The optimality gap ranged between 0% and 3.5%.

2.3 Multi-agent path finding

The multi-agent path finding problem has many variations. Fortunately, we are interested in the classical formulation of the problem, because we need an algorithm that is as general as possible. Therefore, we can take advantage of the wide array of algorithms, both heuristic and optimal. Some algorithms are more like frameworks, providing an abstract high-level view on the problem, which might help us with the necessary modifications. There are more ways to divide the MAPF algorithms, one that can be found in literature is based on the way that the algorithms solve the problem.

But first, we will introduce one of the simplest algorithms for MAPF, that doesn't quite fit in any category of our division. It is the *prioritized planning* algorithm [36]. It computes the paths for agents sequentially, usually using the A* single-agent algorithm, beginning with agents with the highest priority. Each subsequent and lower priority agent must not block any other agent with higher priority. This reduces the search space significantly, but the algorithm is neither optimal nor complete. The incompleteness is caused by the classical MAPF stay-at-target condition and hence is not an issue for us.

2.3.1 Search-based solvers

As the name suggests, algorithms from this family are searching some state space. But the algorithms in the literature are not similar at all, the authors have come up with interesting ideas for search trees and other improvements as the MAPF, and solving it optimally in particular, has gained more attention. Originally, the algorithms were often based on A* [37] — a state consists of locations of all agents at some time step and to get to the next state, all agents execute some action at once. The problem is, that the branching factor grows exponentially with the number of agents, so the researchers started to focus on more efficient methods. Because such methods are developed already, we won't discuss the A* based algorithm and its numerous extensions any further.

One of the novel algorithms is the *Increasing cost tree search* (ICTS) [38]. The main difference between ICTS and A* is, that ICTS searches in the *increasing*

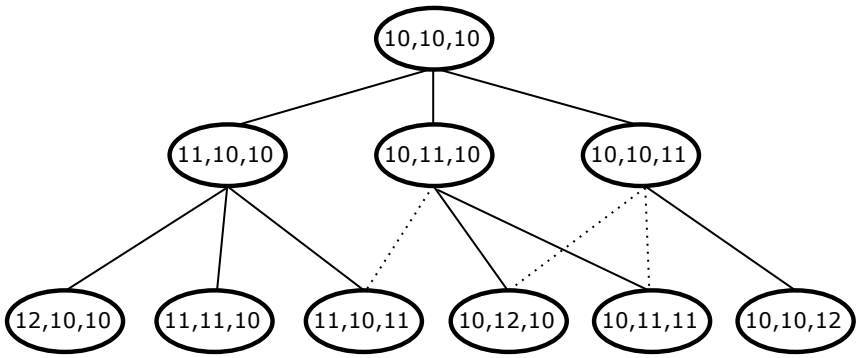

Figure 2.4: Increasing cost tree for three agents

cost tree, which is illustrated in Figure 2.4. In this tree, each node is defined by a set of costs (cost per agent). Each child node has the same costs as the parent node for all agents but one, whose cost is increased by one. If there doesn't exist any non-conflicting solution in the parent node, child nodes are created. For each agent, there is one child node with his cost increased. ICTS proved to be more efficient than the variants of the A* algorithm in many situations — especially when the number of agents is higher, since A* has branching factor exponential in the number of agents, and also when the cost difference between optimal single-agent paths and the optimal non-conflicting path is low, because ICTS is exponential in the depth of its search tree.

Finally, the last and the most recent search-based algorithm for the MAPF is the *Conflict based search* [39]. CBS also uses unique search tree on the high level, the *Conflict Tree* (see Figure 2.5). Each node of the Conflict Tree contains a set of constraints. A constraint restricts a movement of a single agent at some time step. A conflict is a case where a constraint is violated. At each Conflict Tree node, low level search is performed for all agents. Returned single-agent routes must be consistent with all constraints of the node. If there is a conflict in the solution of some node, the node is declared a non-goal node and child nodes are created with additional constraints, that resolve the new conflict. Child nodes also inherit all of the constraints of the parent.

The low-level search can be done by any algorithm, that solves the single-agent problem with constraints. Mostly, A* is used as a low-level search algorithm. CBS outperforms the algorithms above on benchmark maps, where the number of conflicts is not very high. Authors proposed an additional enhancements to the CBS, namely *Meta-Agent CBS* in the original paper, *merge and restart* and *prioritizing conflicts* that further improve performance in the next iteration of CBS, called Improved CBS [40]. The first improvement can merge conflicting agents into one meta-agent, whose path is solved by some MAPF algorithm at once. The merge and restart improvement restarts the search from scratch, when a decision to merge agents is made. The prioritizing conflicts improvement adds three different priorities for conflicts — cardinal, semi-cardinal and non-cardinal. Cardinal conflicts always cause an increase in the solution cost and they are always prioritized for resolution. The last improvement is called *bypassing conflicts* [41]. It tries to prevent a split action by modifying the path of one of the agents participating in a conflict.

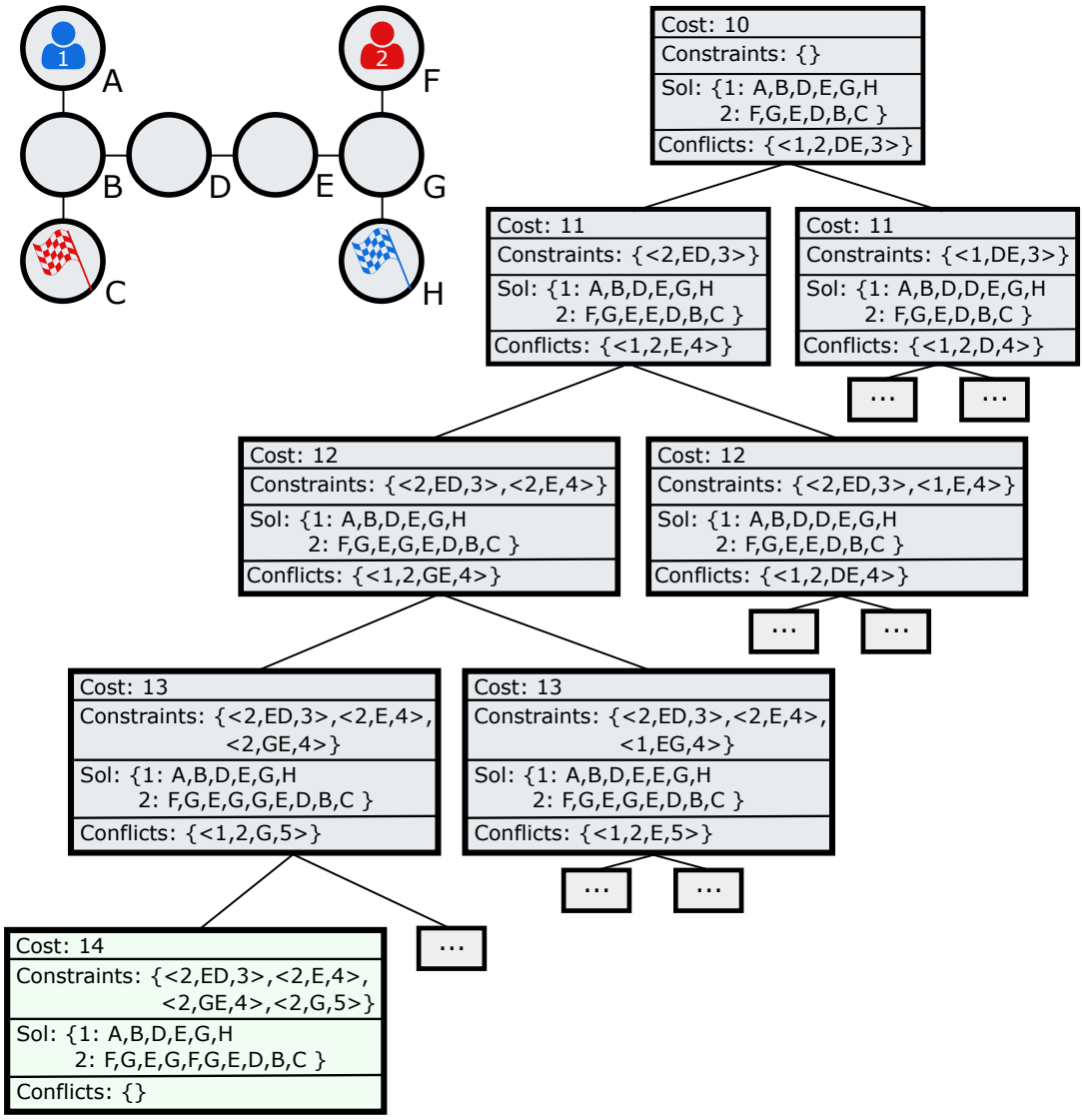

Figure 2.5: Conflict Tree

In another recent paper [42], *corridor symmetry* resolution was introduced, preventing exponential explosion in the space of possible collision resolutions. Another symmetry tackled in the paper is the *target symmetry*, which solves the problem when the path of one agent traverses the target vertex of a second agent who has already arrived at the target vertex and stays there forever.

We will need to understand the corridor symmetry resolution for the next chapter, so we will use the example from the original paper [42] to explain it, written almost word-for-word to preserve clarity. Consider a corridor C of length k with endpoints b and e. Assume that a shortest path of agent a_1 traverses the corridor from b to e and a shortest path of agent a_2 traverses the corridor from e to b. They conflict with each other inside the corridor. Let t_1 be the earliest time step when agent a_1 can reach e and t_2 be the earliest time step when agent a_2 can reach b.

Let us first assume, that there are no bypasses (alternative paths from start vertex to target vertex that don't use vertices of the corridor). In this case, one

of the agent must wait. If the agent a_1 has higher priority, the agent a_2 has to wait. The earliest when agent a_2 can start to traverse the corridor from e is $t_1 + 1$. Therefore, the earliest time step when agent a_2 can reach b is $t_1 + 1 + k$. Similarly, if we prioritize agent a_2, then the earliest time step when agent a_1 can reach e is $t_2 + 1 + k$. Therefore, any paths of agent a_1 that reach e before time step $t_2 + 1 + k$ must conflict with any paths of agent a_2 that reach b before the time step $t_1 + 1 + k$.

Now assume agent a_1 has a bypass to reach e and the earliest time step when he can reach e using the bypass is t'_1. And similarly for an agent a_2, suppose there is an bypass to reach b at time step t'_2 at earliest. If we prioritize agent a_1, agent a_2 can now use the bypass or wait, so the earliest time step when agent a_2 can reach b is $min(t'_2, t_1 + 1 + k)$. If we prioritize agent a_2, agent a_1 can reach e the earliest at $min(t'_1, t_2 + 1 + k)$. Therefore, any paths of agent a_1 that reach e before or at time step $min(t'_1 - 1, t_2 + k)$ must conflict with any paths of agent a_2 that reach b before or at time step $min(t'_2 - 1, t_1 + k)$. In other words, for every pair of conflict-free paths for the two agents, at least one of the two following constraints hold:

- $\langle a_1, e, [0, min(t'_1 - 1, t_2 + k)]\rangle$ or
- $\langle a_2, b, [0, min(t'_2 - 1, t_1 + k)]\rangle$,

where $\langle a_i, v, [t_{min}, t_{max}]\rangle$ is a range constraint that prohibits agent a_i from being at vertex v at any time step form t_{min} to t_{max}. To resolve this corridor conflict, we split the current conflict tree node and generate two child nodes, each with one of the two range constraints as an additional constraint. The values of t_1, t'_1, t_2, t'_2 can be computed using the A* algorithm. In order for the conflict resolution to work, constrained agent must wait before the corridor. This holds because CBS breaks ties by preferring the path that has the fewer conflicts with the paths of other agents. We can use this branching method only when the path of agent a_1 in the current conflict tree node violates the new range constraint of agent a_1 and at the same time, the path of agent a_2 violates the new range constraint of agent a_2. This guarantees that the paths in both child nodes are different from the paths in the current node.

2.3.2 Reduction-based solvers

A fundamentally different approach is to reduce the MAPF problem to some other, well-studied problem. Those often include SAT [43], ILP [44] or compilation to Answer Set Programming [45]. Because solvers for these problems are well optimized, this approach can be efficient in some cases, mostly when the number of conflicts is high, which is shown by Surynek et al. [43], who compiled the problem into SAT, or by Lam et al. with the *Branch-and-Cut-and-Price* hybrid method [46].

2.3.3 Rule-based algorithms

Algorithms from this category are fast, but not optimal. One such algorithm is *Push and Swap* [47], which even guarantees completeness. The algorithm is repeating two simple operations in order to get all agents to their targets. While

no blocking occurs, the algorithm pushes agents towards their goals. If there is a conflict, the Swap function is called. Swapping is implemented by searching for a vertex with a degree greater than two, pushing one of the agents into that vertex, letting the other agent pass the vertex, which enables agents to swap their locations and continue on their path. Another example of a rule-based algorithm is the *BIBOX* [44].

3. Analysis

We have identified the closely related problems and reviewed the literature to get an overview of possible approaches and to get inspiration. In this section, we will go through the list of approaches from the literature and we will assess, whether the approach is suitable for modification to be used for solving the multi-agent picker routing problem. In the end, we will choose to use a MAPF algorithm on the high level, which will handle conflicts between agents. On the low level, a custom GTSP solver will be used. This chapter provides a peek into how the algorithms were chosen.

One of the issues we face is that both sub-problems are NP-hard, as was already stated. As a consequence, we have to be careful when considering optimal algorithms, as the time needed to find a solution could easily exceed any limits we might impose. In other words, the use of heuristics might be required in order to get reasonable computation times. On the other hand, we benefit from the extensive research in both areas that has already been done.

The first idea might be to express the whole problem as a mixed integer programming problem. Unfortunately, even though the solvers are powerful, it can be assumed based on the previous research [13], that real-world instances would be too large to be solved, or even too large to be loaded into the computer memory. Just solving a linear program for one agent without scattered storage can take more than 30 minutes for large instances, even though such examples are not the norm.

3.1 Picker routing - specialized

We will keep the same division of algorithms as in the previous chapter for clarity. As we have seen, most of the algorithms from this category need a special kind of instance and are single-agent only. Even if we don't find any algorithm suitable for solving the Multi-agent picker routing problem as a whole, we might find an algorithm, that will be used for finding the single-agent tours in the final solution.

3.1.1 Exact algorithms

All algorithms in this category that we have mentioned in the previous chapter are based on ideas presented by Ratliff and Rosenthal [3]. They also share the biggest benefit — the polynomial time complexity. If the number of cross-aisles is fixed, that is. The most interesting algorithm for us is the most general introduced by Pansart et al. [13], since it's the only algorithm not limited by the number of cross-aisles. But our assumption of no specific warehouse layout poses a problem nevertheless, since a rectilinear warehouse layout is required by the algorithm. This condition should hold for most instances we encounter, but it cannot be guaranteed. This problem could be resolved by dividing the warehouse into more areas — the main rectangular grid and the special areas. Then, the algorithm could be used in the main storage area and the special areas could be solved separately.

Another incompatibility lies in the fact, that the algorithm is not designed for scattered storage. There are two possible solutions to this problem. The first solution is heuristic. Customer orders can be preprocessed so that only one location per item is chosen. The obvious advantages are, that this solution can be applied to all algorithms, that are not designed for scattered storage and the computation of the tour would be less hard. The disadvantage is, that a lot of variability in the tour generation is lost, which reduces the possibilities to dodge congested aisles and other agents. This might be compensated for to a certain degree by the choice of heuristic. The heuristic could be designed to pick just the locations, that are in the least occupied parts of the warehouse, but this could result in longer tours overall. The second solution would be to modify the algorithm for use with scattered storage. Unfortunately, even if we managed to modify the algorithm, the polynomial time complexity would not be preserved. It has been proven by Weidinger [18], that finding the shortest picking tour satisfying a customer order in a rectangular scattered storage warehouse is strongly NP-hard. All of the arguments made with respect to the warehouse layout and scattered storage are largely applicable to all algorithms, so we won't repeat them again in the latter parts of this thesis.

The last thing to note is that the algorithm is single-agent only and there is almost no way to affect the solution it finds. Therefore, it doesn't seem to be suitable for our application, not even as a single-agent tour solver for some multi-agent algorithm.

3.1.2 Heuristics

Most of the rule-based heuristic algorithms are single-agent only and cannot be used with scattered storage. There is just one heuristic developed by Weidinger [18], that is designed for scattered storage. As we have seen, its performance seems to be good. We could also use the idea of splitting the problem into two parts, as we have mentioned a few paragraphs earlier. But, the heuristic can be used to compute single-agent tours only.

On the other hand, there already exists a multi-agent heuristic. In the modified S-shape heuristics [17], pickers wait at the entrance of the subaisle if the subaisle is already occupied. This prevents congestion, but at the expense of time spent waiting. This heuristic was developed only to serve as a comparison for the Ant Colony Optimisation algorithm the authors presented in the same paper. Nevertheless, it shows a possible way of implementing rules for congestion mitigation. A better alternative might be one-way subaisles. This way, pickers would still have to wait, but only until their pick location is accessible and not until the subaisle is completely vacant. It would require a mathematical model or tests to tell, which approach works better.

To sum up, it is possible to design both a heuristic for scattered storage and a multi-agent heuristic. If we combined the ideas, we should be able to design a heuristic that can do both. The usual advantages and disadvantages of rule-based heuristics would apply — such approach would definitely be very fast, but at the cost of solution quality. There would still be an issue with non-standard warehouse layouts, but this could be resolved. It is an approach worth considering, but there might be a better way still to achieve our goals.

3.1.3 Meta-heuristic

The last category to consider are the meta-heuristic algorithms. Because the meta-heuristic approach is versatile by design, there probably exists some way, how to make an algorithm that performs reasonably well. However, existing research is not that helpful. The single-agent approaches are not easily extendable and the multi-agent approach by Chen and Wang [20] assumes non-deterministic pick times. This assumption lead authors to the use of online conflict resolution instead of planning conflict-free routes in the first place, which is a completely different task.

We could compare the performance of individual approaches in the single-agent scenario at least. However, we would like to resort to meta-heuristic algorithms as the last option. The main issue is, that developing and fine-tuning a meta-heuristic algorithm requires a considerable amount of experience [28], especially when dealing with a problem as complex as ours.

3.2 Picker routing - TSP

It is no surprise, that algorithms from this category are more general. Therefore, we don't have to come up with ways, how to make them work with some special warehouse layout or depot location. We have found out, that the two most promising ways of solving TSP are the optimal Concorde solver and the heuristic LKH solver. Both are very powerful, yet the heuristic solver has an edge when it comes to speed. We can use results from the paper of Theys [26] to get an idea, how fast the solvers are. With orders of 15 items, the Concorde solver found a solution in 70 ms on average, while the LKH found a solution in 20 ms on average. The gap in computation times got much wider with 240 items per order — it took the Concorde solver 27.24 seconds on average to find a solution, but the LKH has found a solution in only 600 ms on average. Since we assume the maximal number of items per order to be 14, the Concorde solver could be fast enough, even given the added complexity of multiple storage locations per item (and conversion of GTSP to TSP). In any case, the LKH scales much better to larger instances, so its use would guarantee good computation times for all of our instances.

So there is an efficient way of solving GTSP, but what about picker blocking? This is where things get complicated. The information about location of other order pickers, whose tours are already planned, has to be passed on somehow to the solver when finding a new tour. Alternatively, the tours could all be searched for at once, so the solver would have complete information about the location of all agents at all times. We haven't invented a way, how to implement the second idea and due to the added complexity of multi-agent problems, it might not be realistic at all for now. So we have to come up with a way of providing additional location information of other agents to the solver. The logical way is to block actions, that would result in picker blocking (e.g. a CBS constraint). Then, the MAPF frameworks for avoiding conflicts could be applied.

3.2.1 GTSP instance representation

In this section, we will analyze possible instance representations with the prohibition of certain edges. The simplest representation is probably a full, time-expanded graph — every vertex has its own copy in each time step and the edges to neighboring vertices go from time t into $t+1$. This would give us the ultimate control over all edges. However, defining a GTSP instance on such a graph would not be straightforward. For example, how should all the non-picking vertices be distributed into disjoint sets, so that any tour covers all the sets? And even if we managed to define an instance, it would be very memory inefficient and too time-consuming to solve. The warehouses we might encounter could have tens of thousands of locations and the number of time steps could be in thousands as well. The resulting graph representing an instance could therefore have tens of millions of vertices, which is too many for any GTSP solver.

To make the GTSP solvable, we need a more compact formulation. The number of vertices can be reduced by omitting all non-picking vertices for a given task. However, we still need the time dimension and control over all edges. This could be achieved by generating paths between the picking vertices using an A* search algorithm with a list of constraints as an input. The size of the graph will be reasonable now if represented efficiently. The division of the vertices into sets is easy as well — all copies of the vertex in the time dimension can be assigned to the set of the SKU. Of course, for more than one SKU picked at a single location, there would be a copy of the vertex for each SKU.

Unfortunately, some GTSP solvers like GLNS accept only full adjacency matrices as an input, which makes them unusable for any larger instances. Consider a customer order of 10 SKUs, which might result in ca. 100 different pick vertices and an upper bound on time steps of 1000. Then, there will be 100 000 vertices in the time-expanded graph. Full adjacency matrix would have 10 billion entries, which means that it would take ca. 40GB of memory to store with 4 bytes per entry. We have tried to come up with GTSP formulation of a time expanded graph nevertheless, but the heuristic GLNS solver is not optimized to work well on directed acyclic graphs (without the returning edge) and it failed to solve even small instances. But this experiment resulted in an interesting observation — to generate a graph for a solver, all possible paths are enumerated, which is time-consuming on its own. That led to an idea to store visited subsets of SKUs for each vertex and to use this information to solve the GTSP directly. In other words, we have decided to make a custom GTSP solver and we will describe it in the next chapter. Since it solves GTSP with a time component, it actually solves a problem, that differs from the standard definition of GTSP. Therefore, the solver is not directly comparable to other solvers.

3.2.2 GTSP solver performance evaluation

Our GTSP solver has an exponential time complexity, so it should be empirically verified, that it is feasible to use the solver in practice. To put the results into perspective, we will make a rough estimate of a sensible computation time. Assume that we want a solution to a problem with 10 agents. Each agent has to complete 10 customer orders and we want the computation time to be less than 10 minutes. This gives the solver ca. 6 seconds per order. We also assume, that

prioritized planning is used to mitigate picker blocking and it slows down the computation by a factor of six. Therefore, we estimate a good computation time per unconstrained order to be at around one second.

The tests were carried out in a small warehouse in accordance with the methodology outlined in the next chapter. All tours are computed in isolation, so there are no constraints. Reported solve times are calculated as a mean value from a total of 150 solved GTSPs for each number of SKUs. In order to minimize the effect of a specific item distribution, the tests were carried out on ten different warehouse instances. For each warehouse instance and SKU count, 15 GTSPs were solved.

SKUs	Solve time (ms)
4	2.93
5	7.81
6	16.95
7	32.21
8	55.81
9	93.95
10	171.51
11	314.92
12	641.96
13	1488.85
14	2964.77
15	6515.92

Table 3.1: GTSP solve times

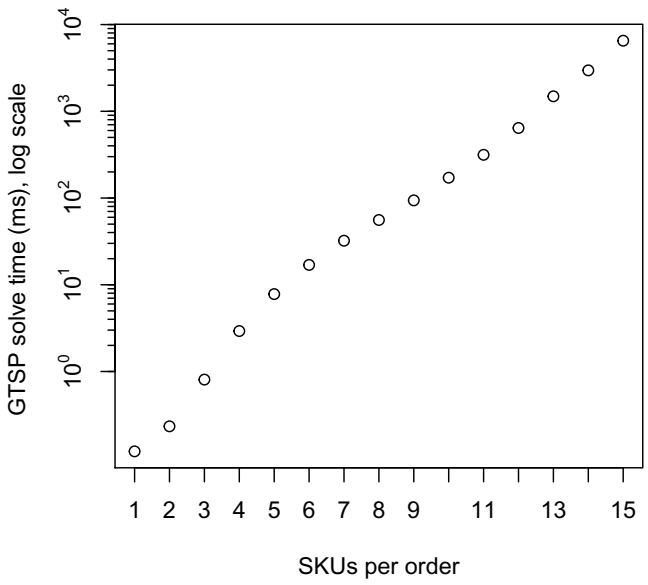

Figure 3.1: GTSP solve times

For up to 12 SKUs per order, the solve times are better than we have expected. Since the mean number of SKUs per order in real-world data is expected to be below 10, we can state that the solver seems to be fast enough according to our guess. Naturally, there are other variables than the number of SKUs that influence solve times. For example, the higher overhead of the MAPF algorithm than expected could increase the computation time by a very large factor, which would make our approach hardly usable. But in conclusion, the intermediate results are good enough to carry on with the development.

3.3 Multi-agent path finding

We have already introduced a GTSP solver, that is suitable for use with some of the MAPF algorithms. In reality, it was the other way around — we have identified the most promising MAPF approaches and then tried to come up with low level search algorithm to complement them. In this section, we will provide some reasoning behind this decision. As with picker routing algorithms, we will keep the division from the previous chapter, so it is possible to easily look up information.

First, let us cover the heuristic approach that we haven't included in any category. The prioritized planning is promising. Because of its minimal overhead, it would most likely enable us to solve large instances. Furthermore, the scattered storage assumption works well with prioritized planning, because the GTSP algorithm still has a lot of freedom when searching for tours, even for the last agents that are the most constrained. In order to improve the quality of solution or computing time, we could design heuristic rules for determining an order of agents, or recalculate the solution with different ordering of agents, if one of the tours was far longer than optimal for the given agent.

3.3.1 Search-based solvers

Due to the added complexity of searching for GTSP tours instead of paths, it's probably safe to say, that A* based approaches wouldn't perform well. It is not even clear what the heuristic function of A* should look like, since there is not a single location the agent is trying to reach.

The ICTS seems to be performing reasonably well on the MAPF test instances. Its two-level approach fits our scenario better, since we would have to modify the low-level part of the algorithm only. The issue we would face is, how to efficiently enumerate all single-agent solutions of some fixed cost. Another issue is, that picking may take up to hundreds of time steps and in the worst-case scenario, an agent would have to wait until some other agent finishes picking, causing many nodes on the high level to be opened.

CBS

The CBS is one of the most recent algorithms for MAPF. It is also a two-level algorithm with many possible extensions, that could improve performance. On the low level, we can use the GTSP solver mentioned previously. After the low level solver is in place, the high level CBS could work just as if it was solving a MAPF instance. The only issue is, that solving a GTSP at the low level is slower by orders of magnitude compared to finding the shortest path. The use of improvements of the basic CBS algorithm on a high level might alleviate the problem. For example, we could implement modified prioritizing conflicts according to some heuristic rules. But the most promising improvement is the efficient corridor conflict resolution based on the concept of corridor symmetry [42], since a typical warehouse layout mostly consists of long corridors.

Corridor conflict symmetry

The use of the corridor symmetry resolution in combination with GTSP solver at low level has some issues, though. The important idea behind CBS and conflict tree is that when a solution of some conflict tree node includes a conflict $C_n = (a_i, a_j, v, t)$, then in any valid solution, at most one of the conflicting agents may occupy vertex v at time t. Therefore, at least one of the constraints (a_i, v, t) or (a_j, v, t) must be added to the set of constraints of the current node. To guarantee optimality, the current node is split into two children and both possibilities are examined. Standard corridor conflict resolution is based on the same principle — the two child nodes cover the whole state space. In a warehouse, the situation is much more complex. Agents don't have fixed goal locations and they have an additional pick action. The algorithm can choose for some agent to pick at some corridor in one iteration and in the next, it can choose completely different pick locations. See the Figure 3.2 for illustration of some conflict situations. But we will show that the invariant we have mentioned for the standard corridor conflict resolution doesn't hold for GTSP tours with counterexample based on the Figure 3.3. It will show, that using the range constraints can lead to the loss of some solutions, even optimal. We will use the simplest corridor conflict possible to demonstrate it — the situation a) from the Figure 3.2, where two agents want to traverse the corridor in opposite directions. The other conflict types in the figure are shown just for illustration.

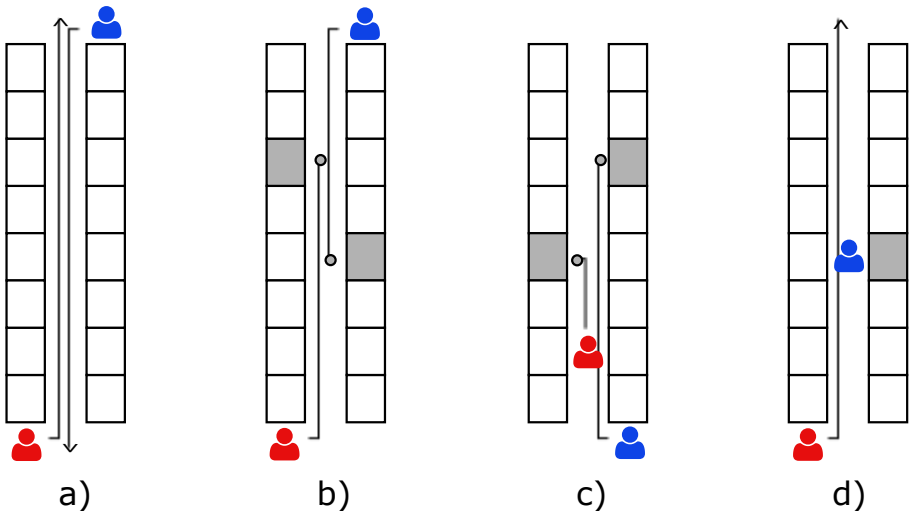

Figure 3.2: Examples of conflicts in a corridor

First, we will describe the Figure 3.3. There are two important agents — a_1 and a_2. Other agents are static in the timespan of the example. There is agent a_3 at location $v3$ and agent a_4 located at $v4$. The reason why they are not moving might be that the SKUs they are picking have high pick times. Therefore, the vertices $v3$ and $v4$ are effectively blocked. Similarly, vertices $b1, b2, b3, b4$ are blocked as well. The difference between blocked vertices and static agents is, that the solver has already constrained the blocked vertices, so they act as a wall, while the vertices occupied by static agents seem to be vacant to the solver until a conflict is found. Vertices $p1$ to $p6$ are pick vertices of agents a_1 and a_2. More specifically, agent a_1 has to pick green and orange SKU, which are located at

vertices $p1, p4, p5$. Agent a_2 has blue and red SKU on the pick list, which are located at vertices $p2, p3, p6$. The numbers next to pick vertices indicate pick times of a given SKU stored at that pick vertex. The tour of agent a_1 starts at location $s1$ and the goal location is $g1$. The tour of agent a_2 starts at $s2$ and his goal location is $g2$. The only way to get from the start location to the goal location for both agents is by crossing the corridor spanning from b to e. Besides the blocked vertices, there are no further constraints in the beginning.

The example might look complicated at first, but the idea is simple. Both agents have only two different choices of pick vertices. First, the shorter tour is chosen by the GTSP algorithm in the beginning. The CBS algorithm will try to resolve the corridor conflict between the two agents, which it does by introducing a range constraint. After the corridor conflict is resolved, other conflicts force both agents into switching the pick vertices to the other pairs. New corridor conflict arises, but one time step earlier. But because there already is the old range constraint, one of the agents will have to wait one time step longer next to the entrance to the corridor than it would be optimal. Now, we will describe this situation more precisely with steps of one possible CBS computation. Computations might differ in the order, in which the conflicts are found and resolved.

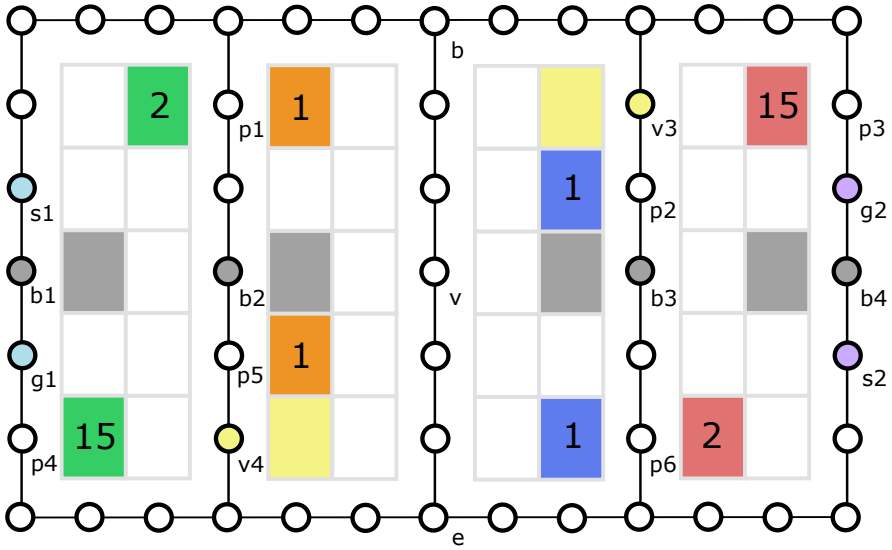

Figure 3.3: Corridor conflict counterexample

1. The algorithm finds optimal tours for both agents — agent a_1 picks green SKU at $p1$ and proceeds to go to vertex $p5$, where he picks orange SKU. After picking, he finishes the tour at $g1$. Agent $a2$ picks red SKU at $p6$ and blue SKU at $p2$, after which he goes to goal vertex. Both tours have the length of 31 and are optimal, given the constraints. Then, the algorithm looks for conflicts and it finds the corridor conflict $C_n = (a_1, a_2, v, 15)$. The node is declared non-goal and two children are created. The constraints that will be added are $\langle a_1, e, [12, 24]\rangle$ into the first child and $\langle a_2, b, [12, 24]\rangle$ into the second child node.

2. Since the example is symmetrical, we can choose any of the two child nodes. Let us choose the left, that constraints agent a_1. His tour is recalculated

with the new range constraint. The second best option is to wait next to vertex b for 7 time steps, until agent $a2$ traverses the corridor. Then, the algorithm looks for conflicts. It finds conflict $C_{n+1} = (a_2, a_3, v3, 22)$. Prioritizing agent a_2 in this conflict doesn't lead to any solution, so we look only at the other conflict tree node that prioritizes agent a_3. In the child node, this process repeats. At time step 23, new conflict at the same vertex is found and resolved in the same way as the last conflict. This is repeated until the other two pick vertices are chosen and agent a_2 doesn't try to cross vertex $v3$, which happens after conflict $C_{n+10} = (a_2, a_3, v3, 31)$ is resolved. The tour for agent a_2 will include the blue SKU at vertex $p6$, the red SKU at vertex $p3$ and it will be 40 time steps long.

3. Now, the conflict $C_{n+11} = (a_1, a_4, v4, 29)$ is found. It happens 7 time steps later than for agent a_2, because agent a_1 is still constrained by the range constraint. The conflict is resolved in the same way as in step 3, just like the next conflict $C_{n+12} = (a_1, a_4, v4, 30)$. Then, conflict $C_{n+13} = (a_1, a_4, v4, 31)$ is found. At this point, the pick vertices of the tour could be swapped for orange SKU at $p1$ and green SKU at $p4$. If the algorithm prioritizes the old tour over the new, one more conflict at the same vertex is found and resolved, before it switches to the new tour.

It is exactly at this point, when the algorithm fails to find the optimal solution, which would be for one of the agents to wait 7 time steps before the corridor and then proceed with the tour. But this solution is not reachable, because of the old range constraint. Since we have prioritized agent a_2 in the first step, there is the $\langle a_1, e, [12, 24]\rangle$ constraint. So agent a_2 has to enter the corridor first in an optimal solution. But agent a_1 cannot benefit from the new situation, that vertex e is free at time step 24 instead of 25, because the blue SKU at vertex $p6$, that is part of the final tour of agent a_2, takes one step less to pick than the red SKU of the original tour. Consequently, his tour will be one step longer than it could have been. It doesn't help to choose the other node in the first step and prioritize agent a_1, because the result would be identical except that it would be agent a_2, whose tour would be one step longer. This is the case since the example is symmetrical for both agents.

It is not difficult to imagine, that a similar situation might actually arise during computation. And one time step delay is the best-case scenario. This example was chosen, because it didn't even include any pick vertices in the corridor. It can be assumed, that in a real warehouse, most conflicts will happen near pick vertices, since the real pick times are much higher than in our example. Modifying the corridor conflict resolution, so that it accounts for picking in corridors is a hard task with an uncertain payoff. Of course, there could still be rare situations, where the corridor conflict resolution would save a lot of computational power, but it's probably not worth it.

Even though we didn't find a way to use the corridor conflict resolution, we have implemented the CBS algorithm anyway to test its performance. On the low-level, the GTSP solver was used. But the intermediate results that we have obtained during the development phase were not good. Even with just a couple of agents, some instances were solved fast, but some took really long to solve, or they were not solved at all. The CBS algorithm had to expand thousands of

nodes in order to find some solutions even with just five agents. Therefore, we have concluded that the CBS is not suitable in our case.

3.3.2 Reduction-based solvers

Even though the most recent reduction-based approaches show very good results that are comparable to the best search-based solvers, we would rather not choose the reduction-based approach. The conversions are complicated even for the MAPF and applying the findings to our case would likely be very hard. Also, the warehouse instances can be large and that will likely require a heuristic, more flexible approach. We will only mention the hybrid method, that seems especially promising — the Branch-and-Cut-and-Price algorithm. Authors have shown, that it has the potential to outperform CBS, especially on larger instances. Furthermore, this approach is also capable of domain-specific reasoning to improve its performance. To incorporate GTSP into the framework, we would need to implement a *pricer* — an algorithm, that solves a single-agent problem with a modified objective function that reflects the locations of other agents to find improving tours. Nevertheless, using some search-based approach seems to be more attainable, so we will not pursue the hybrid approach as well.

3.3.3 Rule-based algorithms

The rule-based approaches are not suitable to be used in our case. The rules are stated for the path finding problem and therefore cannot be used to solve GTSPs. Even though we cannot use any of the rule-based algorithms directly, we could introduce some rules inspired by these algorithms to improve the performance of some other algorithm. For example, we could implement rule-based dodging in cross-aisles.

4. Solution methods

In this chapter, an algorithm for the multi-agent picker routing problem is introduced. But first, we will describe our GTSP solver, which is crucial for integration with the multi-agent algorithm used to mitigate picker blocking. Then the multi-agent algorithm will be introduced — it is the prioritized planning algorithm. Some options that affect the performance of the prioritized planning, such as the choice of priority heuristic, will be evaluated empirically.

4.1 Search-based GTSP solver

To fully utilize the scattered storage assumption, solving GTSP instances is necessary to find picking tours for customer orders. At the same time, the most promising multi-agent approaches need control over individual edges and vertices at a certain time in order to avoid picker blocking when generating tours. It poses a great challenge to solve such problems quickly as standard off-the-shelf solvers include an option for adding constraints neither on pick vertices (cities) nor on pass through vertices. Without the time component, it would be near impossible to generate conflict-free tours from customer orders, so we have made a GTSP solver, that can block individual vertices or edges at a specific time.

The generalized traveling salesperson problem is a touring problem. When searching for a solution, a state of a touring problem includes not only the current location, but all visited locations on the path leading to the node as well. In the context of order picking, a state contains a set of already picked SKUs. Beyond the usual definition of a touring problem, the state space has to be extended by the time component as well.

Consider two identical states apart from the time component. The states may have different sets of successors. For example, a path to the next pick vertex might be blocked by some order picker in the first state, but not in the other.

4.1.1 GTSP formulation

We have already talked about compact GTSP instance representation in the previous chapter. In this chapter, we will follow up on the last chapter and formalize GTSP as a problem for a search algorithm. More specifically, we will define the initial state, actions, the transition model and successors, a goal test, and a path cost. After the formal formulation of the problem, the search algorithm will be introduced.

- **Initial state:** An initial state is given by an agent's location and a current time step, because the algorithm requires absolute time in order to correctly block occupied vertices. Naturally, there are no visited/picked vertices in an initial state.
- **Actions:** We have defined three atomic actions — move, wait and pick. For the purpose of a search algorithm, we will combine sequences of these three atomic actions into a new *travel* action. If an agent leaves a pick vertex, the travel action includes picking at that vertex before traveling. The actual

movement part of the travel action can contain any valid sequence of move and wait atomic actions that end at the right vertex. Therefore, for any two different vertices, there are infinitely many travel actions. We will usually search for the shortest one. Pick on leave was chosen, because then it has to be checked only once when expanding a node, whether picking at that node is possible, or if it is blocked by some constraint.

- **Transition model:** Let $d(v_1, v_2, t, C)$ be a distance function that returns the cost of the shortest non-conflicting path from vertex v_1 to v_2, with departure from v_1 at time t and with constraints C. Let $p(v)$ be the pick time of SKU stored at vertex v. The value of $p(v)$ is defined for every pick vertex of the GTSP instance, because there is exactly one SKU at each vertex (see Chapter 1). Let C be a set of constraints and (v, t, S) be a state, where v is a pick vertex, t is a time step and S is the set of already picked vertices. Successor to the state (v, t, S) is any state $(x, t_x, S \cup \{SKU_x\})$, such that x is reachable from v at time $t_x = t + p(v) + d(v, x, t + p(v), C)$ and SKU_x is different from the SKUs stored at all previous pick vertices on the path, which could be written as $SKU_x \notin S$ in a mathematical notation.
- **Path cost:** The path cost function $g(n)$ is defined as the number of time steps required to get from the root node to the node n while picking at all nodes of the path. Only with the exception of the very first node and n, because picking is part of a transition to the next node.
- **Goal test:** The goal test succeeds, if the state's location is a goal vertex and if the set of picked SKUs contains all SKUs of the customer order.

4.1.2 Uniform-cost search

With the problem formulated, we can start with the verbal description of the search algorithm. Later in this chapter, there is also a description in form of pseudocode (see Algorithm 1 and Algorithm 2). We encourage the reader to see the pseudocode if something is not clear.

The algorithm is a modified Uniform-cost search. Just as the regular Uniform-cost search, the algorithm expands nodes with the lowest path cost $g(n)$ first. But there is not any explicit priority queue. Instead, for each (v, t) pair, the algorithm remembers whether the node is open or not in an array. The GTSP solver iterates over time steps chronologically, therefore in an order of an increasing path cost. For each time step, it iterates over all pick vertices. If some node is not open, the algorithm skips it.

One difference between the two approaches is in the memory complexity of the data structure. Since the branching factor can be a very big number, there could be a lot of open nodes. But storing whether a node is open for all nodes in an array is not free either. There will be a break-even point, where our approach becomes more memory efficient than the standard approach with a priority queue. Another difference is that finding a node in an indexed array is easier than in a priority queue. As we will see, this works well with another modification that we will introduce soon.

On the expansion of an open node, the algorithm tries whether picking at the node's vertex is possible and if it is, it searches the shortest paths to all successor

nodes and marks them as open. It should be noted that in reality, the algorithm tries to find the shortest path after picking to all pick vertices that store a different SKU than the current node's vertex and it tries to open the corresponding nodes as well. Usually, this might result in visiting a pick vertex of some picked SKU for the second time, but we have come up with a way, how to prevent this from happening.

4.1.3 Subset array

As we have noted, the state space of our problem has three dimensions — time, vertex, and a set of picked SKUs. Storing all of this information for all open and explored nodes would be very costly. Assume that there are N SKUs on the pick list of the current customer order. For some (v, t) pair, where t is a time step and v is a vertex, there are 2^{N-1} possible subsets of SKUs, since the SKU stored at v will be included in all of them. Therefore, there are 2^{N-1} different states with the same vertex and time. Let us consider a hypothetical customer order, that has 100 pick vertices in total split between 15 different SKUs and the tour could be up to 1000 time steps long. Then the upper bound on the number of different search nodes is approximately 1.6 billion. That is way too much to store in computer memory, even if the search nodes were only a couple of bytes large.

That leads us to the biggest difference between our GTSP solver and the standard Uniform-cost search. To reduce the memory complexity of the search, we will merge all of the search nodes with the same vertex and time coordinate into one node with a single shared subset memory, which we will call *the subset array*. For example, if there were two search nodes, (v_1, t_1, $\{SKU_1$,$SKU_2\}$) and (v_1, t_1, $\{SKU_2$, $SKU_3\}$), then they would be merged into a single node (v_1, t_1, $\{\{SKU_1$, $SKU_2\}$, $\{SKU_2$, $SKU_3\}\}$). Notice, that both sets of the original nodes are represented in the subset array of the merged node, therefore no information about any path leading to the node was lost. Every (v, t) pair up to the time limit has its own subset array.

As we have seen on the example, the sets stored in the subset array correspond to legal paths. For each set stored in the subset array, there must exist at least one path, on which the SKUs from the set are picked. The other implication, that for every legal path leading to the node there must be a set of its pick vertices in the node's subset array, does not generally hold. It is because the algorithm is not complete if there are constraints, which will be shown later. Now, we will describe the operations on the subset array and we also want to show that all of the operations are correct, as well as the invariant stated in this paragraph.

Operations on the subset array

Every node begins with an empty set. Before the node is expanded, two operations are performed on the subset array of the node. The first operation is the filter operation. Consider a node n with SKU_i stored at the node's vertex with set of sets S stored in its subset array. There might be sets containing SKU_i in S, because generally, the nodes are allowed to open any node with different SKU from theirs. Because the agent will pick SKU_i on leave, SKU_i would be picked twice on some of the paths after leaving node n, which would make those paths invalid. That is why all sets containing SKU_i are removed from S.

The second is the add operation. The SKU_i is added to all remaining sets in S, signaling to all successors, that the valid paths leading to n were just extended by picking SKU_i at n.

Now that the subset array of node n is consistent, the node is ready for expansion. The algorithm generates successors (as defined for this algorithm) and it wants to copy the subset array into all of them. Copying the subset array gives the successors the information, which vertices were already picked on the paths leading to them through the current node. But it also needs to keep all of the sets already stored in successor's subset array intact. Therefore, the algorithm unifies subsets of the current node with subsets of the successor and saves the resulting set into the successor. When the successor itself is expanded, it will contain a union of all possible subsets of picked SKUs, that are on some path to the node.

Correctness of the operations

The only time an SKU is added to the sets is when the SKU will be picked at the node's vertex. The only time a set is removed from the subset array is to prevent double picking on the paths. Finally, the unification transfers all of the sets of the expanded node to successors, while preserving all of the sets that are already saved in successors. The sets that would correspond to illegal paths will be removed in some future step. Therefore, all sets of some node after performing all operations correspond to some legal paths and no valid paths are lost when performing the operations on the subset array.

4.1.4 Finding a solution

Suppose the algorithm reached some node that contains a set with all SKUs of the order in its subset array. Therefore, the only thing left to get a full tour would be to find a path to the goal vertex. But the tour that includes this pick vertex as the last pick location doesn't have to be the shortest tour possible. Therefore, the algorithm only marks this node as a possible last pick vertex. We will call a node, that is a possible last pick vertex *a candidate*. Along with the path to the goal vertex, the algorithm gets an upper bound on tour length. This bound is used to adjust the time limit and then the search for a better solution continues.

Reverse search

After reaching the time limit and if some candidate was found, the algorithm knows only the last pick vertex, path to the goal vertex, and the time step at which the tour has begun. Because the predecessors are not stored in the node, the algorithm has to search for the predecessors iteratively node after node, until the root node is reached. That is quite easy since the information needed for the reverse search — the subsets of SKUs that can be picked of all nodes up to the candidate — is stored in the subset array and the actions are reversible. At least in a sense, that the algorithm can find the predecessors.

Unlike the forward search, the reverse search begins with all SKUs picked. The goal is to reach the initial state. In other words, it tries to reach the root node with an empty set of SKUs to be picked. The reverse search starts from

the best candidate node, which is the last pick vertex of the tour. Therefore, the algorithm knows which SKU is picked at this node and it can remove it from the set of SKUs to be picked.

Now the algorithm searches for a suitable predecessor. Suitable predecessor means, that the node contains a set equal to the set of SKUs to be picked. That equality serves as a certificate, that there exists some path to the node, on which the missing SKUs are picked. Therefore, the algorithm searches for backward paths (including picking) to all vertices of the SKUs left to be picked and it searches for the set of missing SKUs in their subset arrays. If there is a matching set then the algorithm has found one more predecessor. It removes the SKU of the predecessor node from the set of SKUs to be picked and the process repeats.

Reverse search with constraints

When there are no constraints, forward and backward paths between nodes are symmetrical and the reverse search is trivial. However, the paths are not always symmetrical, when there are some constraints. For example, assume the agent finished picking at vertex v_1 at time t and the only neighboring vertex of v_1 is blocked at time $t + 1$. After picking, he finds the shortest path to v_2. Since the neighbor of v_1 is blocked at time $t+1$, the agent waits for one time step at v_1, until the neighbor is not constrained anymore and then the agent proceeds to travel on the shortest path to the vertex v_2. He arrives at the time step $t + d_{opt}(v_1, v_2) + 1$, where d_{opt} is a function that returns the length of an optimal unconstrained path between two vertices. Assume that the algorithm is in the reverse search phase now and it is at vertex v_2 at time step $t + d_{opt}(v_1, v_2) + 1$. It tries to find a shortest backward path to v_1 and it finds a path of length $d_{opt}(v_1, v_2)$. Therefore, the reverse search would not check the predecessor node at time step t, but only at a time step $t + 1$, which might cause that the solution is not found.

Fortunately, the solution to this problem is easy. If any predecessor with the needed set of SKUs is not found on optimal backward paths, we repeat the search for predecessor with all backward paths extended by one. Extending the paths by one time step at a time will continue until the predecessor node with the exact set needed for a solution is found. The algorithm knows that the predecessor exists from the sets of the current node. Since the algorithm starts from the lengths of optimal paths and the extending doesn't end until the predecessor is found, this modification is also correct and the algorithm finds the solution.

Algorithm 1: GTSP solver

```
Input: Instance of an customer order, warehouse graph, time limit
// Initialize paths from start, since explicit priority queue is not used.
foreach vertex v in pick vertices do
    path := Shortest path from start vertex to v with constraints
    Mark v as visited at time length(path)
    Initialize sets[v][cost]
end
for time := 0 to time limit do
    foreach vertex v in pick vertices do
        if (v,time) not visited then
            continue
        end
        if cannot pick at (v,time) then
            continue
        end
        Filter subsets containing SKU(v) from sets[v][time]
        Add SKU(v) to all subsets at sets[v][time]
        if sets[v][time] contains all SKUs then
            path := Shortest path from v to goal with constaints at t
            newLimit := time + length(path)
            if limit > newLimit then
                limit := newLimit
                Mark v as a candidate
            end
            break
        end
        foreach vertex u in pick vertices do
            if SKU(v) = SKU(u) then
                continue
            end
            path := Shortest path from v to u with constraints at t
            arrival := time+pickTime(v)+length(path)
            Mark u as visited at arrival
            sets[u][arrival] := sets[u][arrival] ⋃ sets[v][time]
        end
    end
end
if candidate exists then
    solution := Reverse search the solution
end
return solution, or failure
```

Algorithm 2: Reverse search - GTSP solver

```
Input: Last pick location with time (v_l, t_l),start v_s, goal v_g, tour start
       time t_0
solution.AddFirst(v_g)
solution.AddFirst(v_l)
toPick.AddAll()
toPick.Remove(SKU(v_g))
currentTime := t_l
currentVertex := v_l
LOOP:
for i := 0 to toPick.Count do
    for extension := 0 to currentTime − t_0 do
        foreach vertex v in pick vertices do
            if SKU(v) not in toPick then
                continue
            end
            // extension is the length of the path beyond optimal
            path := FindReversePath(currentVertex, v,
             extension,currentTime)
            if path doesn't exist then
                continue
            end
            if toPick is in sets[v][currentTime-length(path)-pickTime(v)]
             then
                solution.AddFirst(v)
                toPick.Remove(SKU(v))
                currentTime := currentTime - length(path) - pickTime(v)
                currentVertex := v
                goto LOOP
            end
        end
    end
end
solution.AddFirst(v_s)
return solution
```

4.1.5 Example of the search

Now that we have covered how the algorithm works, we can illustrate it with an example. Consider an instance from the Figure 4.1. Start and goal locations are both at the depot and the tour begins at the time step zero. Pick list consists of three SKUs, we will call them the red, the blue, and the yellow SKU. The yellow SKU can be picked at vertex $y1$, the blue SKU at vertex $b1$, and the red SKU at vertices $r1$ and $r2$. Pick times are all zero. The goal is to find the shortest tour to pick all three SKUs.

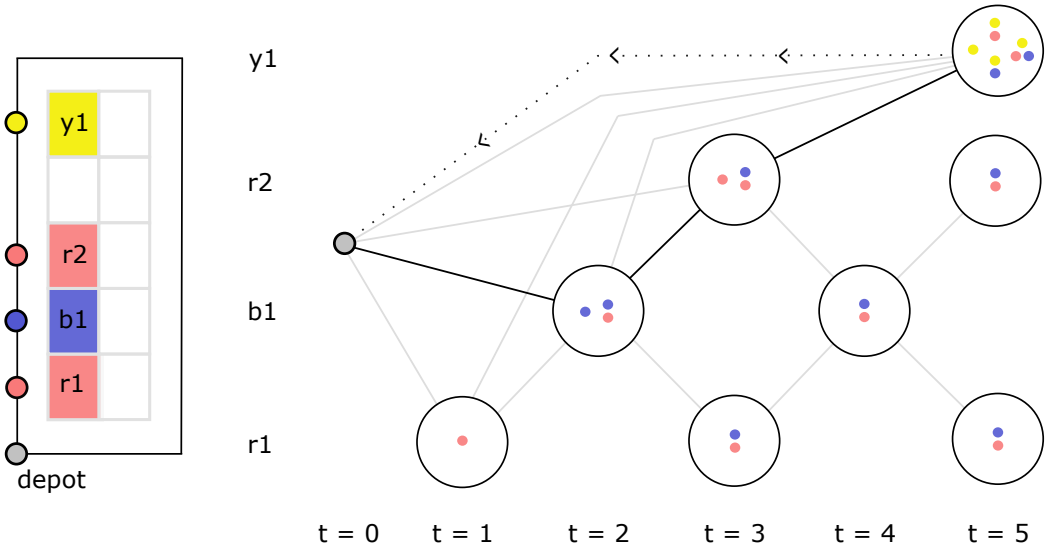

Figure 4.1: GTSP search with zero pick times

The search space is represented as follows. On the horizontal axis are the time steps. The pick vertices are on the vertical axis and the subsets of picked SKUs are represented by the clusters of dots inside of nodes. One cluster of dots represents one set of SKUs, that are picked on some path to the node, including the SKU picked at the node's vertex, because they represent the state of the node after expansion. The gray edges lead to successor nodes. For simplicity, we have omitted all edges that end beyond the time step 5. The dotted edge represents the final path from the last search node's vertex to the goal vertex. There is one path with black lines. That is the path of one optimal solution of the two possible.

4.1.6 Properties

The correctness of the algorithm follows from the correctness of the Uniform-cost search algorithm and the correctness of our modifications. The algorithm always stops, when it is run on a machine with finite memory, because the main loop is an iteration over all (v, t) pairs and there is a limited number of pairs. Theoretically, the algorithm doesn't have to stop when the memory is infinite. The maximal number of steps in a reverse search is always limited if the algorithm reaches the reverse search phase.

Without any constraints, the algorithm is complete, because it eventually tries all possible combinations, if any solution is not found early. It is also optimal

without constraints, because the search continues at least until all nodes with time step one less than the cost of the best candidate solution are expanded. This guarantees, that the algorithm doesn't stop before it finds an optimal solution, because no optimal candidate can have path cost greater than the cost of the whole optimal solution.

With constraints, it is neither optimal nor complete. This is because when expanding a node, only the shortest route to a successor is found. If the successor is blocked for picking at the arrival time, then no further successors are generated and the branch is not expanded further. Therefore, if the only solution requires a longer than optimal path between two vertices because picking at the end vertex of the path is blocked after the arrival on the optimal path, then the algorithm won't find it.

4.1.7 Complexity

Let us denote t the maximal number of time steps, C the number of unique SKUs in the pick order, I_{c_i} the number of picking locations of $i-th$ SKU and I is the total number of all picking locations. The memory complexity is asymptotically determined by the size of the subset arrays, which is $\theta(2^C * I * t)$. For every picking location and time step, the algorithm keeps in memory all visited subsets. The algorithm also uses some memory for finding the shortest paths and constraints, but that amount is negligible for larger C. Also, there is the optional cache of shortest routes between pick vertices. Again, depending on the value of C, the size of the cache will be negligible and it can be disabled if it caused any problem. For the sake of completeness, the size of the cache is $\theta(I^2 * K)$, where K is some constant dependent on the warehouse size.

The time complexity of a naive algorithm, that tries every pick location combination and every permutation, is $\theta(C! * C_{i_1} * ... * Ci_n)$. Our GTSP solver has the time complexity of the search $\mathcal{O}(I^2 * t)$. At every pick vertex and time step from the range, the solver tries to find a path to every other pick vertex and unifies the subset arrays, if the path exists. The time t, in this case, is only equal to the length of the shortest tour, not to some predetermined constant. But this complexity formula omits an important action. The key operation is the manipulation with the subset arrays, that have exponential memory complexity. So the actual time complexity of the solver is $\mathcal{O}(2^C * I^2 * t)$, which is exactly what we have seen in the experiment from the previous chapter.

	Small	Medium I	Medium II	Large
C	10	12	14	16
I	100	120	140	200
t	3000	3000	4000	4500
Memory (MB)	38	184	1146	7372

Table 4.1: Real-world memory usage examples

4.2 Single-agent approach

The single-agent approach solves the multi-agent picker routing problem in a single-agent way. The main goal of this algorithm is to simulate a regular warehouse with some common single-agent routing policy. By simulating we mean that the algorithm not only computes single-agent routes but also ensures that the routes are conflict-free and therefore could be performed step-by-step in a physical warehouse. In a manual real-world warehouse, conflicts are resolved during the execution by order pickers. If planned picking tours contain any conflict, order pickers dodge each other or just wait, until the other order picker makes way.

We can think of the single-agent approach as a two-step algorithm. In the first step, the algorithm plans optimal single-agent picking tours for all agents independently, or in other words, without any knowledge of other agent's locations. In a real-world warehouse, the tours could be computed by the optimal algorithm introduced by Ratliff and Rosenthal. Since this algorithm is optimal as well, the resulting tours should be comparable.

Conflict resolution

As we have said, in a typical manual warehouse, conflict resolution happens naturally. The single-agent approach does conflict resolution in the second step. Input for the second step is a set of independent single-agent solutions and output is a single conflict-free multi-agent plan for all agents.

Now, we will describe the second step of the algorithm in detail. First, priorities are assigned to the individual customer orders. Then the algorithm iterates over the single-agent tours computed for the orders in decreasing order of priorities. For each tour, the goal is to resolve any conflicts with the higher priority tours. The higher priority tours cannot be modified, so it is the currently processed tour that has to adapt.

The conflict resolution procedure for a single picking tour maintains both the pick locations and their order in a tour. Therefore, the only possibility is to adjust the paths between pick locations (and the start/goal locations). Conflicts are resolved one path at a time, iterating through the paths chronologically. After a path is non-conflicting with other agents, then the algorithm starts processing the subsequent path and so on, until the path to the goal vertex is conflict-free as well.

The conflicts of a single path are resolved by iterative extension of the path. First, the algorithm tries to replace the original path with an optimal path that is consistent with constraints imposed by other agents. The optimal path might or might not be of the same length, depending on the constraints. All subsequent extensions prolong the path by a single step. The path is extended, if it doesn't meet any of the following criteria:

1. the path must exist,
2. picking at the last vertex at the time of arrival must be possible (excluding the path to the goal vertex of the tour),
3. after picking at the last vertex, there must exist a path to goal vertex.

The necessary condition number one says, that the path must exist. This might seem trivial, but the existence of a path of length n doesn't imply the existence of a path of length $n + 1$ in a multi-agent environment. The second criterion is self-explanatory. The last criterion ensures, that the agent is free to leave after picking. In our implementation, we have chosen the goal vertex of the tour as the target vertex for the tested path, because it should be located in a staging area outside of all picking aisles. This criterion is necessary because it might happen that the agent is squeezed by higher priority agents inside of a picking aisle and such a solution would be invalid. The first and shortest path that meets all three criteria is substituted in the tour instead of the original single-agent path.

Properties

After all conflicts are resolved, we obtain a solution with the lowest cost possible without changing pick vertices of tours or priorities. The algorithm is not optimal, nor complete. In any case, the third criterion should ensure that some solution is found most of the time. Its computation time will be determined asymptotically by the computation of initial tours, which is exponential. The conflict resolution is done exactly once for each path and the number of path extensions is bounded by the makespan of the partial solution, after which there will be no constraints and all paths after this point will be valid. Therefore, the algorithm always stops.

Algorithm 3: Single-agent algorithm

```
Input: Customer orders, warehouse graph
tours := Find shortest single-agent tours for all orders.
Assign priority to all tours
foreach tour t in tours do
    foreach path p in tour.paths do
        l := length(path)-1
        while Non-conflicting path not found do
            l := l + 1
            Find path of length l to the next pick vertex with constraints
            if Path does not exist then
                continue
            end
            if Next pick vertex is blocked during picking then
                continue
            end
            if Path to depot after picking does not exist then
                continue
            end
            Add path to the solution
        end
    end
    Add modified tour into constraints
end
return solution, or failure
```

4.3 Prioritized planning picker routing

Prioritized planning with the GTSP solver on the low level is the main algorithm we introduce for solving the multi-agent picker routing problem. There is a similarity to the previous algorithm in the sense, that in order to avoid conflicts between agents, it also prioritizes some tours over others. The main advantage of this approach is, that it avoids branching of the high level search, which could possibly result in an exponential number of search nodes.

The main difference between prioritized planning and the previous single-agent algorithm is that prioritized planning solves the GTSPs with existing tours of other agents as constraints, while the single-agent approach uses the GTSP solver only to determine the pick vertices and their order without any constraints. Therefore, prioritized planning resolves conflicts directly when computing the GTSPs, while the single-agent approach resolves conflicts only by modifying paths between predetermined pick vertices.

We have said, that the algorithm assigns priorities to tours and not agents. That is not a mistake, but the priorities have to be assigned with caution. The only constraint is that for one agent, any single tour must have a higher priority than all subsequent tours of the same agent. This constraint must hold because in order to plan a tour, all previous tours have to be already computed so that the algorithm knows when the current tour starts. This finer priority assignment will enable us to use more efficient heuristics later.

The prioritized planning algorithm is not optimal, but it is able to solve instances much bigger than it would be possible with any optimal algorithm. It is not complete either. Due to the limitations of our GTSP solver, all it takes is for the other agents to block some pick vertex for long enough. The blocked pick vertex should be the only pick vertex of the given SKU and the current agent must have this SKU in his pick list. Then the GTSP solver will eventually fail because it won't be able to pick at that vertex from any node and it doesn't try to find an alternative time to pick. Therefore, the open nodes will run out and blocking the pick vertex for a finite amount of time is sufficient.

The memory complexity of the prioritized planning algorithm is completely determined by the memory complexity of the GTSP solver. The only additional memory is needed for solutions and constraints. The number of constraints is limited by the solution length. The time complexity is also determined by the GTSP solver, as the priority assignment for the tours is usually polynomial in the number of orders and there is almost no other overhead other than managing the constraints.

Algorithm 4: Prioritized planning algorithm

```
Input: Customer orders, warehouse graph
Assign priority to all customer orders
foreach order o in prioritized orders do
    tour := FindTour(o, constraints)
    solution.AddTour(tour)
    constraints.AddConstraints(tour)
end
return solution, or failure
```

4.3.1 Gradual constraint addition

The usual way of adding constraints of previously planned tours is to add all constraints at once. This means that every step some agent takes on an already planned tour is added as a constraint. Therefore, the GTSP solver has complete information about the movement of other agents right away. But there is another way. Alternatively, the tour could be computed without any constraints first. Then, the algorithm would check, whether there is a conflict with some other tour. In case of a conflict, one constraint is added and the GTSP is computed again. This process is repeated until a non-conflicting solution is found.

Algorithm 5: Gradual prioritized planning algorithm

```
Input: Customer orders, warehouse graph
Assign priority to all customer orders
foreach order o in prioritized orders do
    constraints.Clear()
    while Non-conflicting tour not found do
        tour := FindTour(o, constraints)
        conflict := FindConflict(tour, solution)
        if conflict found then
            constraints.AddConstraint(conflict)
        end
    end
    solution.AddTour(tour)
end
return solution, or failure
```

There is a good reason to use this alternative approach. It will become clear after we write a few words about the standard approach. When the task is to find a path between two points, the number of constraints has little impact on the computation time. But in our case, the GTSP solver relies heavily on caching of paths between pick locations. Therefore, if some cached path is blocked, the solver has to find a new path by using the A* search algorithm. This is a much more demanding operation than just returning a path from the cache. Just a single constraint can result in a big number of paths being blocked and recalculated. This is slowing the GTSP computation down significantly and the more constraints there are, the slower the computation. Hundreds or thousands of constraints that are present with the standard prioritized planner will slow down the solver substantially. The advantage of the standard prioritized planner is that each GTSP is solved exactly once.

Comparison of the prioritized planning approaches

The gradual approach relies on solving the GTSPs much faster due to the small number of constraints in order to mitigate this issue. But it comes at a high cost — many recalculations of the same GTSP instance. The upper bound on the number of recalculations is equal to the sum of costs of planned tours and solving the GTSP gets slower and slower as the constraints are added one by one. This leads us to the following hypotheses, that we will test:

1. the gradual approach will perform better than standard prioritized planner when the number of agents is relatively small with respect to the size of the warehouse,

2. the gradual approach will slow down disproportionately to the number of agents, as the number of agents increases.

The tests were carried out on two different instance sizes, small (1) and large (2). The instances and methodology are described in Chapter 5. For each instance size and algorithm, there will be three tests with different numbers of agents in a warehouse. The number of agents will range from 3 to 12. The number of trials (solved instances) is 40. The standard prioritized planner is labeled as PP, while the gradual prioritized planner is labeled as PP-G. Time is a shortened form of the mean solve time and we report it in seconds.

Algorithm	Size	Agents	Time (sec)
PP-G	1	3	0.8
PP-G	1	6	3.8
PP-G	1	12	57.8
PP	1	3	121.7
PP	1	6	586.3
PP	1	12	2724.2
PP-G	2	3	0.6
PP-G	2	6	1.6
PP-G	2	12	9.5
PP	2	3	98.2
PP	2	6	356.5
PP	2	12	1457.7

Table 4.2: Gradual prioritized planner experiment

The experiment clearly shows that in our tests, the gradual constraint addition is superior to the standard prioritized planner with respect to solve times. Therefore, the first hypothesis likely holds. The evidence even suggests that PP-G performs better even for the more densely populated instances, such as the small warehouse instance with 12 agents. Gradual prioritized planner solves instances with twelve agents faster on average, than the PP solves instances with just three agents.

The data also support the second hypothesis. For a small instance size, when the number of agents doubled from 6 to 12, the average computation time increased more than fifteen times. The increase was more modest for large instances, but the average solve time increased more than four times with the same numbers of agents. It seems that the increases are driven by some other factor as well, maybe it is the density of agents in a warehouse.

Minor optimizations

We have tried some smaller optimizations as well, but without greater success. One of the tries was to improve the performance by adding some constraints

in advance. This should have reduced the number of recalculations. For every picking done by some other agent at a pick vertex of the current order, two constraints at the start and end times of the pick were added. This almost ensures that the solver will not find a solution with blocked picking and it is achieved with a small number of constraints. The results of this optimization were that it helps with some harder instances, but the overall effect is positive only by a small margin.

We have also tried to add several constraints at once by finding more than one conflict, which resulted in a hardly measurable effect. Another way to add more constraints in one iteration could be by looking at the blocking agent's locations in some time window close to the conflict and adding these constraints as well. This could help because there often are many constraints close to one point in space and time.

4.3.2 Priority heuristics comparison

The performance and solution quality can be influenced by the order, in which the algorithm solves the tours of customer orders. To be clear, the algorithm cannot change the order in which the agent completes his customer orders, but there is some freedom in order, in which the tours are solved. We will compare the default order to the SKU heuristic. There are two variants of the SKU heuristic. The first variant is to prioritize customer orders with the most SKUs on a pick list (labeled as PP-SH, which stands for prioritized planner-SKU high), the other variant is to prioritize customer orders with the least SKUs to pick (labeled as PP-SL). We will test several related hypotheses. All of the hypotheses are meant relative to the default order without heuristic.

1. The heuristic PP-SH reduces the computation time.
2. The heuristic PP-SH leads to lower quality solutions.
3. The heuristic PP-SL leads to higher quality solutions.

The reasoning behind the first hypothesis is that the most time-consuming tours to solve will be computed first, when the number of previously planned tours and constraints is low. That should result in a better computation time than without any heuristic. Hypotheses two and three are mirror images of each other. The reasoning is, that PP-SH leaves the customer orders with the least different items, and therefore usually the least pick locations as well, to be computed last when the number of constraints is the highest. That could leave very few optimization options for the last tours, resulting in a higher solution cost. The same logic in reverse applies to the third hypothesis.

The tests were repeated 40 times with different random generator seed numbers. A small warehouse instance was chosen for the tests. The tests were carried out for three different agent counts. For more information on the methodology see Chapter 5.

Algorithm	Agents	Time (sec)	SoC	Makespan
PP	3	1.0	6006	2338
PP-SH	3	0.8	6006	2338
PP-SL	3	1.8	6007	2340
PP	6	13.0	12194	2636
PP-SH	6	5.3	12200	2637
PP-SL	6	18.5	12193	2635
PP	12	178.5	24296	2728
PP-SH	12	55.8	24306	2720
PP-SL	12	160.9	24280	2726

Table 4.3: Comparison of prioritized planner heuristics

We can conclude, that the first hypothesis most likely holds. There is a significant drop in computation times when using the PP-SH heuristic. The drop is pretty consistent for 6 and 12 agents at approximately one third of the average solve time without any heuristic. There is only a slight hint, that the effect increases with the number of agents. Of course, the effect will also be dependent on the structure of customer orders, especially on the number of SKUs in the orders. Even though there are indications that the choice of a heuristic has some effect on solution quality, the evidence doesn't seem to be strong enough to reject the null hypothesis for both the second and third hypotheses.

Because the difference in computation times was much more significant than the difference in solution quality, we will use the PP-SH variant of the algorithm in the final evaluation.

5. Empirical evaluation

The purpose of this chapter is to empirically evaluate the performance of the algorithms in several different scenarios. By performance, we mean mostly the computation times and solution quality. The main criterion for the computation times is the requirement, that the algorithm should be able to provide results after an overnight computation. The solution quality of the prioritized planning approach is compared to the single-agent algorithm to see, whether the difference is significant enough to justify the potentially higher computation times.

5.1 Setup and implementation

The tests were run on a server with Intel Xeon 8270 CPU at 2.7GHz and with 8GB of RAM. The virtual machine had 2 CPU cores available. The operating system used was Windows 10. No other CPU-intensive applications were running on the virtual machine. The algorithms are implemented in the C# programming language and run in the .NET Core 3.1 Runtime. Each test is run several times with a different random number generator seed. The default number of trials per test is 40, if not specified otherwise.

5.2 Test problem instances

Since any library of standard test instances for rectangular scattered storage warehouses doesn't exist and this thesis is motivated by real-world logistic operations that decide the ranges of parameters, we generate our own test instances.

There are three model warehouse sizes — small, medium, and large. General assumptions about the warehouse type and layout are identical as stated in Section 1.1. The size parameters such as the number of aisles and cross-aisles were chosen to be consistent with the assignment and other research in this area. The specific values of parameters that we have chosen for the three standard warehouse sizes are in Table 5.1.

	Aisles	Cross-aisles	Shelf Length	Shelf Height	Storage Locations
S	10	3	10	5	2000
M	20	6	10	5	10000
L	50	6	20	5	50000

Table 5.1: Test instance parameters

The number of unique items stored in a warehouse was chosen so that each item is stored in ca. 10 locations. Therefore, it is different for each of the instance sizes. Items are distributed across the warehouse at random. For every storage location, one item is sampled from a uniform distribution of all items. Every item is stored in at least one location. The average number of locations per item is 10 in our test instances.

Each warehouse size is tested with three different numbers of agents. The numbers of agents are 3, 6, 12 for the small warehouse, 12, 24, and 48 agents

for the medium warehouse size and finally, 25, 50, and 100 agents for the large warehouse. Each agent is assigned three orders. Orders are generated at random. The length of a pick list is chosen randomly from a uniform distribution for each order and it ranges from 2 to 8 SKUs for most of the tests. The only exception is the comparison of the prioritized planner to the gradual prioritized planner, where the pick list length is fixed to 5 SKUs, which helps with computation times.

It should be noted that there is not a single value or distribution applicable to all real-world warehouses. Some store large items only, so the number of SKUs per order is close to one, while other warehouses might contain a mix of small and large items.

5.3 Results

Three algorithms will be compared. The lower bound on multi-agent solution quality is set by an algorithm labeled as SA-NB. This stands for single-agent no block, because the agents don't interact and cannot block each other. All tours calculated by SA-NB are optimal in a single-agent environment. Therefore, it really is a lower bound for any multi-agent algorithm, since preventing agents from blocking each other can only add to the cost. How much will the cost increase when agents do block each other is approximated by the algorithm labeled as SA, which stands for single-agent and it is the algorithm we have introduced in Section 4.2. The algorithm labeled as PP is the gradual prioritized planning algorithm.

We will compare the algorithms in computation times and solution quality. All of these values are calculated as a mean value of all instances with corresponding parameters. The unprocessed results of the experiments can be found as an attachment of this thesis. In the column Time, we report the time needed to find a complete solution, excluding initialization of the solver (memory allocation, etc.). The column SoC stands for Sum of Cost. For easier comparison of the algorithms, there are relative values of cost in parentheses, where 100% is always the lower bound approximation of the SA-NB algorithm for the given number of agents.

Small instances

The number of trials for small instances is set to 200. This was possible, because most of the instances were solved within one minute.

The computation time of the prioritized planner rises steeply as the number of agents increases. Doubling the number of agents from six to twelve caused a more than tenfold increase in the average computation time.

Regarding the solution quality, it seems that the more agents there are, the bigger the optimality gap of the single-agent algorithm. This was expected, as the number of conflicts increases as well. On the other hand, prioritized planner shows results well within 1% from the single-agent optimum for the Sum of Cost function and it is just a couple of time steps behind the lower bound estimate for the makespan function in all three test scenarios.

In addition to the main experiment, we have performed another set of tests with the prioritized planner algorithm to see the limit number of agents, for which the instances are still solvable. The number of trials for this test was set to 20

Algorithm	Agents	Time (sec)	SoC	Makespan
SA-NB	3	0.2	6089 (100.0)	2394 (100.0)
SA	3	0.2	6205 (101.9)	2425 (101.3)
PP	3	1.0	6095 (100.1)	2395 (100.0)
SA-NB	6	0.3	12090 (100.0)	2578 (100.0)
SA	6	0.6	12694 (105.0)	2681 (104.0)
PP	6	5.7	12122 (100.3)	2580 (100.1)
SA-NB	12	0.6	23925 (100.0)	2697 (100.0)
SA	12	3.6	26827 (112.1)	2940 (109.0)
PP	12	63.4	24100 (100.7)	2700 (100.1)

Table 5.2: Small instance results

only, because we wanted to solve instances with as many agents as possible and we didn't need the results to be very precise. The results are visualized in Figure 5.1.

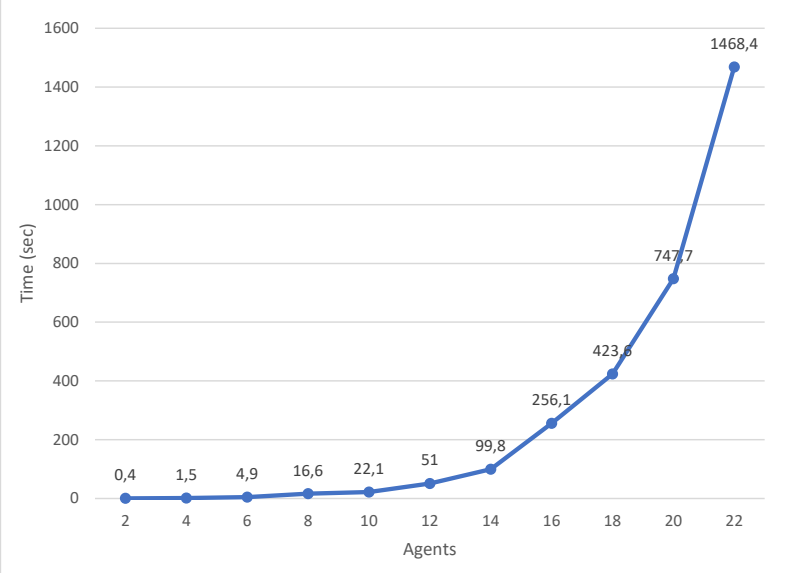

Figure 5.1: Prioritized planner on small instance agents limit

It seems that the limit on small instances is somewhere at around 22 agents. For 24 agents, it took the algorithm hours to find a solution, so the tests didn't even finish in time.

Medium instances

The number of trials for the medium instances is set to the standard value of 40. This should still be enough for a good approximation of the real mean value.

Algorithm	Agents	Time (sec)	SoC	Makespan
SA-NB	12	1.1	25844 (100.0)	2903 (100.0)
SA	12	2.1	26559 (102.8)	2953 (101.7)
PP	12	7.4	25899 (100.2)	2904 (100.0)
SA-NB	24	2.2	52049 (100.0)	3032 (100.0)
SA	24	5.5	55042 (105.8)	3153 (104.0)
PP	24	52.7	52304 (100.5)	3036 (100.1)
SA-NB	48	4.3	104107 (100.0)	3142 (100.0)
SA	48	33.8	117268 (112.6)	3390 (107.9)
PP	48	712.7	105399 (101.2)	3147 (100.2)

Table 5.3: Medium instance results

First, let us verify the claim about the good approximation of the real mean value, at least on the last test. The 95% confidence interval of the mean computation time of the PP algorithm with 48 agents is $[591, 834]$. For the same test scenario and the sum of costs values, the confidence interval is $[104419, 106377]$. We have checked the confidence intervals for the other PP tests as well and the relative deviations were similar. Even though the deviations of the mean computation times are quite large, the differences between the approaches are even larger. Therefore, the results are precise enough to make some conclusions in the discussion.

The prioritized planner displays similar behavior as on the small instances. If we compare the computation times between 24 and 48 agent instances, we notice a more than tenfold increase in the computation times. Even the single-agent approach doesn't seem to scale linearly with the number of agents, most likely because more paths need rerouting as the number of agents increases.

The solution quality of the prioritized planner is still very good, even though for 48 agents, the approximation of optimality gap has risen slightly above 1%. Single-agent approach with conflict resolution yields tours with Sum of Cost up to 12.6% larger, than the single-agent algorithm without conflict resolution. For 48 agents, there is a 243 time steps difference in makespan between SA and PP, which would be more than 4 minutes, if one time step was equal to one minute.

Large instances

Even though some large instances took more than one hour to solve, the number of trials is still 40. The confidence intervals of the mean computation times are a bit wider than they were on medium instances.

Computation times of the prioritized planner seem to increase consistently with what we have seen already. Doubling from 50 to 100 agents resulted in a little over eightfold increase in computation times on average. The single-agent approach still finds a solution in a couple of minutes most of the time.

The solutions produced by the single-agent approach are closer to the optimum than on smaller instances. The average optimality gap in the sum of costs is smaller than 8.1% for all tests. For the makespan, it is even below 6%. Prioritized planner displays similar solution quality as in medium instance tests. For the sum of costs, the solution costs differ less than 1.3% from optimum on average, while the makespan is within 0.2% of the optimal makespan.

Algorithm	Agents	Time (sec)	SoC	Makespan
SA-NB	25	2.8	61800 (100.0)	3402 (100.0)
SA	25	5.4	62909 (101.8)	3451 (101.4)
PP	25	55.0	61979 (100.3)	3405 (100.1)
SA-NB	50	5.6	123879 (100.0)	3480 (100.0)
SA	50	24.0	128745 (103.9)	3563 (102.4)
PP	50	430.8	124562 (100.6)	3485 (100.1)
SA-NB	100	11.2	248055 (100.0)	3568 (100.0)
SA	100	189.7	268098 (108.1)	3771 (105.7)
PP	100	3578.4	251400 (101.3)	3574 (100.2)

Table 5.4: Large instance results

5.4 Discussion

We wanted to test, whether prioritized planning runs in a reasonable time and whether it outperforms some usual single-agent approach in solution quality. That single-agent approach is, in our case, the single-agent algorithm with conflict resolution. We have also included the results of the single-agent algorithm without conflict resolution, because the results can serve as a lower bound on optimal solution cost. That enables us to judge the solution quality even better. In this section, we will evaluate the results and discuss, whether we have succeeded in those goals.

Let us start with the computation times. If we look at the different tests separately, we can see, that the prioritized planner doesn't scale well with the number of agents. The maximal computation time on 100 agents was a bit over two hours. But the customer orders consisted of 5 SKUs on average only. If this average was higher, the GTSP solver would probably take much more time to compute the individual customer orders. That alone could prolong the computation time over the 12 hours of overnight limit. On the other hand, on most instances of small and medium size, the prioritized planner will likely finish computation within the set time limit. On large instances, it will be highly dependent on the number of agents, their customer orders, and some other factors, such as the distribution of items in a warehouse.

If we compare the computation times between different instance sizes, an interesting phenomenon occurs. Solving an instance with the same number of agents in a larger warehouse is faster on average, than in a smaller warehouse. This is illustrated in Figure 5.2, which shows the average time it took to solve an instance divided by the number of agents. The blue line corresponds to small instances, the orange line to medium instances, and the gray line to large instances. We observe that with 12 agents, solving a small instance takes much longer per agent than it does for a medium instance. A similar result can be seen with 48 agents. It is probably caused by the increased density of agents, which causes more blocking and potential congestion. This in turn increases the number of constraints that have to be added, before a GTSP tour is conflict-free. This suggests that agent density will be one of the most important factors that determine the computation times of prioritized planner. More research would be needed to specify and confirm this hypothesis.

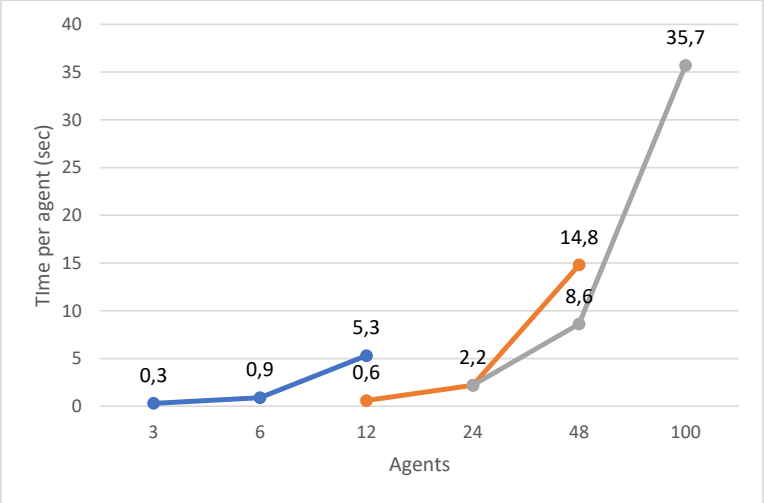

Figure 5.2: Prioritized planner solve time per agent

The results show clearly that in solution quality, the prioritized planner is significantly better than the single-agent approach with conflict resolution. The difference is the most notable when the number of agents is high. For example, on medium instance with 48 agents, the difference in average sum of costs was more than 10%. It was less noticeable on the instances with relatively few agents, where the difference was only ca. 1.5%.

The experiments have some limitations, too. The variety of the test scenarios could be higher. They don't even come close in covering the wide variety found in real warehouses. Another limitation is that the single-agent approach is only an approximation of a real-world warehouse routing policy. More experiments would have to be carried out for specific warehouses to determine precisely the benefit of using the multi-agent routing policy. But probably the biggest limitation is in the algorithm itself. We assume manual warehouses with people performing the order picking. But the tours, that the algorithm produces, rely on the absolute precision of agents in order to be conflict-free. We can only speculate, whether the inaccuracy and inconsistency of human order pickers wouldn't eliminate any benefit of multi-agent planning.

Conclusion

The work on multi-agent picker routing algorithms is in its infancy. We have carried out a comprehensive literature review of this problem and we have found only couple of multi-agent approaches. On the other hand, there are plenty of research papers proving that the effect of picker blocking and congestion is real and that it is significant.

There exists a large variety of picker routing algorithms as well as multi-agent path finding algorithms. In order to come up with an algorithm that solves multi-agent picker routing, we have explored many of them. We have identified the prioritized planning approach from the multi-agent path finding domain as the most promising, because it offers a good compromise between the solution quality and computation times. Since the algorithm was originally designed to solve multi-agent path finding instances, we had to come up with a way to adapt it to the picker routing problem. In the end, we have developed a custom GTSP solver that finds picking tours and that allows the prioritized planner to block already occupied vertices. Together, these two algorithms solve the multi-agent picker routing problem. In order to improve the performance of the algorithm, we have tested the use of a heuristic, that determines the order in which the tours are computed. We have shown that the right heuristic can significantly decrease computation times.

Finally, the new algorithm was empirically tested and compared to a single-agent routing policy. Even though the algorithm has some limitations regarding its computation times, it performed well in solution quality. The solutions it produced were nearly optimal, unlike the solutions produced by the single-agent approach after all conflicts were resolved.

But the main takeaway is, that there is still a lot more to be done. We have identified several other promising approaches to solve the problem. For the largest of instances, a purely heuristic approach could be more fitting. On the other hand, for smaller instances, some approach based on the CBS or the hybrid Branch-and-Cut-and-Price method might work well. Other areas that could be explored include, how human order pickers actually execute the planned tours and whether the benefit of conflict-free tours is not lost by human inaccuracy. Alternatively, the use of the algorithm in automated warehouses could be examined as well, since the focus shifts from manual warehouses to automated. There is definitely a lot of ongoing research fueled by automation but unfortunately, not all of this research is made public. Nevertheless, manual warehouses are probably here to stay for some time, too.

Bibliography

[1] Jolyon Drury. Towards more efficient order picking. *IMM monograph*, 1(1):1–69, 1988.

[2] J.A. Tompkins, J.A. White, Y.A. Bozer, and J.M.A. Tanchoco. *Facilities Planning*. Wiley, 2010.

[3] H Donald Ratliff and Arnon S Rosenthal. Order-picking in a rectangular warehouse: a solvable case of the traveling salesman problem. *Operations research*, 31(3):507–521, 1983.

[4] Randolph W Hall. Distance approximations for routing manual pickers in a warehouse. *IIE transactions*, 25(4):76–87, 1993.

[5] Christian Huber. *Throughput analysis of manual order picking systems with congestion consideration*, volume 76. KIT Scientific publishing, 2014.

[6] M Klodawski, R Jachimowski, I Jacyna-Golda, and M Izdebski. Simulation analysis of order picking efficiency with congestion situations. *International Journal of Simulation Modelling*, 17(3):431–443, 2018.

[7] Ralf Elbert, Torsten Franzke, Christoph H Glock, and Eric H Grosse. Agent-based analysis of picker blocking in manual order picking systems: Effects of routing combinations on throughput time. In *2015 Winter Simulation Conference (WSC)*, pages 3937–3948. IEEE, 2015.

[8] André Scholz, Daniel Schubert, and Gerhard Wäscher. Order picking with multiple pickers and due dates–simultaneous solution of order batching, batch assignment and sequencing, and picker routing problems. *European Journal of Operational Research*, 263(2):461–478, 2017.

[9] Gérard Cornuéjols, Jean Fonlupt, and Denis Naddef. The traveling salesman problem on a graph and some related integer polyhedra. *Mathematical programming*, 33(1):1–27, 1985.

[10] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Barták. Multi-agent pathfinding: Definitions, variants, and benchmarks. *CoRR*, abs/1906.08291, 2019.

[11] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[12] Kees Jan Roodbergen and René De Koster. Routing order pickers in a warehouse with a middle aisle. *European Journal of Operational Research*, 133(1):32–43, 2001.

[13] Lucie Pansart, Nicolas Catusse, and Hadrien Cambazard. Exact algorithms for the order picking problem. *Computers & Operations Research*, 100:117–127, 2018.

[14] Hadrien Cambazard and Nicolas Catusse. Fixed-parameter algorithms for rectilinear steiner tree and rectilinear traveling salesman problem in the plane. *CoRR*, abs/1512.06649, 2015.

[15] Charles G Petersen. An evaluation of order picking routeing policies. *International Journal of Operations & Production Management*, 1997.

[16] Kees Jan Roodbergen and RenÉde Koster. Routing methods for warehouses with multiple cross aisles. *International Journal of Production Research*, 39(9):1865–1883, 2001.

[17] Fangyu Chen, Hongwei Wang, Chao Qi, and Yong Xie. An ant colony optimization routing algorithm for two order pickers with congestion consideration. *Computers & Industrial Engineering*, 66(1):77–85, 2013.

[18] Felix Weidinger. Picker routing in rectangular mixed shelves warehouses. *Computers & Operations Research*, 95:139–150, 2018.

[19] Albert H Schrotenboer, Susanne Wruck, Kees Jan Roodbergen, Marjolein Veenstra, and Arjan S Dijkstra. Order picker routing with product returns and interaction delays. *International Journal of Production Research*, 55(21):6394–6406, 2017.

[20] Fangyu Chen, Hongwei Wang, Yong Xie, and Chao Qi. An aco-based online routing method for multiple order pickers with congestion consideration in warehouse. *Journal of Intelligent Manufacturing*, 27(2):389–408, 2016.

[21] Roberta De Santis, Roberto Montanari, Giuseppe Vignali, and Eleonora Bottani. An adapted ant colony optimization algorithm for the minimization of the travel distance of pickers in manual warehouses. *European Journal of Operational Research*, 267(1):120–137, 2018.

[22] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[23] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Tsp cuts which do not conform to the template paradigm. In *Computational combinatorial optimization*, pages 261–303. Springer, 2001.

[24] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[25] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[26] Christophe Theys, Olli Bräysy, Wout Dullaert, and Birger Raa. Using a tsp heuristic for routing order pickers in warehouses. *European Journal of Operational Research*, 200(3):755–763, 2010.

[27] Samrat Hore, Aditya Chatterjee, and Anup Dewanji. Improving variable neighborhood search to solve the traveling salesman problem. *Applied Soft Computing*, 68:83–91, 2018.

[28] Sabry Ahmed Haroun, Benhra Jamal, et al. A performance comparison of ga and aco applied to tsp. *International Journal of Computer Applications*, 117(20), 2015.

[29] Keld Helsgaun. Solving the equality generalized traveling salesman problem using the lin–kernighan–helsgaun algorithm. *Mathematical Programming Computation*, 7(3):269–287, 2015.

[30] Matteo Fischetti, Juan José Salazar González, and Paolo Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45(3):378–394, 1997.

[31] Gregory Gutin and Daniel Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9(1):47–60, 2010.

[32] Stephen L Smith and Frank Imeson. Glns: An effective large neighborhood search heuristic for the generalized traveling salesman problem. *Computers & Operations Research*, 87:1–19, 2017.

[33] Adam N Letchford, Saeideh D Nasiri, and Dirk Oliver Theis. Compact formulations of the steiner traveling salesman problem and related problems. *European Journal of Operational Research*, 228(1):83–92, 2013.

[34] Ruben Interian and Celso C Ribeiro. A grasp heuristic using path-relinking and restarts for the steiner traveling salesman problem. *International Transactions in Operational Research*, 24(6):1307–1323, 2017.

[35] Eduardo Álvarez-Miranda and Markus Sinnl. A note on computational aspects of the steiner traveling salesman problem. *International Transactions in Operational Research*, 26(4):1396–1401, 2019.

[36] R. K. Dewangan, A. Shukla, and W. W. Godfrey. Survey on prioritized multi robot path planning. In *2017 IEEE International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*, pages 423–428, 2017.

[37] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, and Jonathan Schaeffer. A* variants for optimal multi-agent pathfinding. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[38] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013.

[39] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[40] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. Icbs: improved conflict-based search algorithm for multi-agent pathfinding. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[41] Eli Boyarski, Ariel Felner, Guni Sharon, and Roni Stern. Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In *ICAPS*, pages 47–51, 2015.

[42] Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J Stuckey, Hang Ma, and Sven Koenig. New techniques for pairwise symmetry breaking in multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 193–201, 2020.

[43] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient sat approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 810–818, 2016.

[44] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *2009 IEEE International Conference on Robotics and Automation*, pages 3613–3619. IEEE, 2009.

[45] Rodrigo N Gómez, Carlos Hernández, and Jorge A Baier. Solving sum-of-costs multi-agent pathfinding with answer-set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9867–9874, 2020.

[46] Edward Lam, Pierre Le Bodic, Daniel Damir Harabor, and Peter J Stuckey. Branch-and-cut-and-price for multi-agent pathfinding. In *IJCAI*, pages 1289–1296, 2019.

[47] Ryan J Luna and Kostas E Bekris. Push and swap: Fast cooperative pathfinding with completeness guarantees. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
SKU	Stock keeping unit
MAPF	Multi-agent path finding
GTSP	Generalized traveling salesperson problem