Karlsruhe Institute of Technology

Coordinated Motion Planning for Multiple Square-Shaped Robots in a Grid

Submission to the „CG:SHOP 2021“-Challenge

Bachelor Thesis of

Julian Dinh

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisors:	Paul Jungeblut

Time Period: 24th December 2020 – 23rd April 2021

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 22nd April 2021

Abstract

Given are n square-shaped robots with start and target positions for each robot in an unbounded grid that may contain stationary obstacles. At each time step, a robot can move to an adjacent grid position or remain at its position. The task is to find a *schedule*, meaning a sequence of moves for each robot, so that every robot starts at its start position and reaches its target position without colliding with any other robot or obstacle. We show that, under mild restrictions, we always find a feasible schedule.

There are two objectives that we consider separately. The first objective aims at minimizing the overall time needed to move every robot to its target position. We prove that it is NP-complete to compute a schedule with a length that is lower than k, for a $k \in \mathbb{N}$. The second objective aims at minimizing the total distance traveled by all robots. We use different heuristics to obtain a schedule with good results concerning both objectives. Our solution is evaluated on many diverse instances. Especially for the second objective, we achieve decent results.

Deutsche Zusammenfassung

Gegeben seien n quadratische Roboter mit Start- und Zielpositionen für jeden Roboter. Die Roboter befinden sich in einem unbegrenzten Gitter, welches stationäre Hindernisse enthalten kann. In jedem Zeitschritt kann sich ein Roboter in eine benachbarte Gitterposition bewegen oder auf seiner Position bleiben. Die Aufgabe ist es, einen *Plan* für die Bewegungsabläufe zu finden, in dem jeder Roboter an seiner Startposition beginnt und seine Zielposition erreicht, ohne dabei mit anderen Robotern oder Hindernissen zu kollidieren. Wir zeigen, dass wir mit einer kleinen Einschränkung immer einen solchen Plan finden.

Es gibt zwei weitere Ziele, welche wir unabhängig voneinander betrachten. Das erste Ziel ist es, die Gesamtzeit zu minimieren, die benötigt wird, um alle Roboter an ihre Zielposition zu bringen. Wir beweisen, dass es NP-vollständig ist, einen Plan zu berechnen, welcher eine Gesamtzeit besitzt, die kleiner als k ist (für ein $k \in \mathbb{N}$). Das zweite Ziel ist es, die Strecke, die alle Roboter zusammen zurücklegen, zu minimieren. Wir benutzen verschiedene Heuristiken, um einen Plan zu erhalten, welcher beide Ziele möglichst gut berücksichtigt. Wir evaluieren unsere Lösung auf vielen verschiedenen Instanzen. Insbesondere für das zweite Ziel erzielen wir gute Ergebnisse.

Contents

1. Introduction

Since the second half of the 20th century, the domain of robotics has gained more and more relevance. Today, robots are used in many different areas. As robots tend to be more accurate and reliable than humans, they, for example, simplify processes in the industry. One of the key research topics in robotics is to plan a path for a robot from a given start position to a desired target position while avoiding obstacles in the environment. This is called *motion planning*. Planning the path of a single robot has been well studied over the last decades, whereas *coordinated motion planning* asks to coordinate many individual robots at the same time. In this scenario, the goal often is overall efficiency rather than individual navigation. There is still a vast demand for algorithms that solve the coordinated motion planning problem with provable performance guarantees. We participated in a challenge that stated the task to coordinate many square-shaped robots located in an unbounded grid.

1.1. „CG:SHOP 2021“-Challenge

This thesis is dedicated to the “CG:SHOP 2021”-Challenge[1] (Computational Geometry: Solving Hard Optimization Problems). Since 2019, this challenge has been hosted every year encouraging scientists to deal with solving hard computational geometry problems. This year’s challenge, called “Coordinated Motion Planning”, invites scholars to compute good solutions to a specific coordinated motion planning problem while minimizing different objective functions. The hosts of the challenge, Sándor P. Fekete, Phillip Keldenich, Dominik Krupke and Joseph S. B. Mitchell [FKKM21], published a paper providing further information on the 2021-challenge as well as the most notable results. In this context, 203 instances of the problem were provided and the quality of the submitted solutions to these instances determined the placement in this competition.

1.2. Related Work

Relocating many robots from a given start configuration into a desired goal configuration has been a topic of high interest for the past 20 years. More generally, this problem is also known as the *multi-agent pathfinding* problem (MAPF) and has many applications, for example in automated warehouses, as shown by Wurman et al. [WDM08]. However, Ma et al. [MKA+17] discuss issues that arise when generalizing MAPF methods to real-world

[1] Official website: `https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2021`

scenarios. Furthermore, Stern et al. [SSF$^+$19] provide a survey with variations on MAPF, for instance, they describe several types of conflicts between agents.

In coordinated motion planning, different variations can be applied as well. Objects are shaped in various ways and scenarios classify in *discrete* or *continuous* settings. On the one hand, in the discrete scenario, the input is a graph in which objects can transition from one vertex to another vertex if an edge exists between the two vertices. Objects are not allowed to use a vertex or edge at the same time. On the other hand, in the continuous case, the objects represent some shape and must move to their target positions, so that their bodies do not overlap at any time. All objects may be bounded to a certain region and there may be obstacles in the environment. In both settings, objects can further be *labeled*, *colored* or *unlabeled.* First, if all objects are labeled, they are distinguishable and each object has a unique target position that can only be covered by the object with the appropriate label. Second, in the colored case, the objects are partitioned into several groups by assigning a color to them. Objects of the same color belong to the same group and each target position can only be covered by an object with the corresponding color. Solovey and Halperin [SH14] present a sampling-based algorithm for this case. Third, in the unlabeled scenario, a target position can be covered by any object, because the objects are indistinguishable.

In the 1980s, Schwartz and Sharir [SS83] investigated the motion of multiple disk-shaped objects among obstacles in a bounded region. Their algorithms are polynomial in the number of obstacles, but exponential in the number of disks. Hopcroft et al. [HSS84] and Hopcroft and Wilfong [HW86] showed PSPACE-completeness for multi-object motion planning for rectangles in a rectangular boundary. The children's game "Rush Hour" is a sliding block game played on a grid board with the goal is to find a sequence of moves that allows a special target block to exit the grid. This problem was generalized to an $n \times n$ grid and proven PSPACE-complete by Flake and Braum [FB02]. In another famous game, called "15-puzzle", the goal is to slide 15 labeled unit squares into a target configuration within a 4×4 board. The 4×4 board was extended to an $n \times n$ board by Ratner and Warmuth [RW90], who showed that finding a shortest solution for the extended puzzle is NP-hard. All of these findings encourage the study of polynomial approximation algorithms for related robotics problems.

A common technique for solving motion planning problems is to depict the environment as a graph, where the cost of edges represents the cost to transition from one position to another. A related problem is the *k-disjoint shortest path problem*, which asks for the existence of k vertex-disjoint paths in a graph with k pairs of terminals that connect each pair of terminals. All paths are required to be shortest paths. Lochet [Loc21] shows that for any fixed k, the disjoint shortest paths problem admits a polynomial time algorithm.

Wagner and Choset [WC15] introduce a framework that initially plans paths for each robot individually and then coordinates motion among the robots as needed. This approach is a mix of so-called *decoupled* and *coupled path planning.* Decoupled path planning algorithms for multiple robots first compute paths individually for each robot and then adjust the paths to avoid collision with other robots. Coupled path planning methods search the joint configuration space guaranteeing to find an optimal path. An example of the latter approach is the famous A^* algorithm by Hart et al. [HNR68]. However, with a growing number of robots, the A^* algorithm becomes computationally infeasible. Making use of SAT solvers by recasting the coordinated motion planning problem as a SAT instance represents another coupled approach, see the article by Ruoyun et al. [HCZ10].

The „CG:SHOP 2021“-Challenge is based on an article by Demaine et al. [DFK$^+$19]. The authors prove that finding a schedule with minimal execution time is NP-hard, even for a grid arrangement without any stationary obstacles. Furthermore, they achieve constant-factor approximation for minimizing the overall execution time in the absence of obstacles, provided that some amount of separability is given.

In this thesis, we consider a grid setting with labeled sqaure-shaped robots. Within this specific context, robot motion behaves differently compared to the setting by Demaine et al. As with other teams who participated in the challenge, the solution we propose relies on the idea of a two-step process. Crombez et al. (Team Shadoks) [CdFG$^+$21] describe their method of solving the challenge's problem. They obtain a moderate solution and then optimize it by using different techniques and heuristics. Liu et al. (Team Gitastrophe) [LSJZZ21] follow a similar approach by first finding a good initial solution and then performing a k-opt local search phase which optimizes the solution. Yang and Vigneron (Team Unist) [YV21] take a simulated annealing approach in their optimization phase.

1.3. Contribution

This work provides a solution that can be used to compute schedules for the instances provided in the challenge. We propose different heuristics and show that, under mild restrictions, we always find a schedule. However, this schedule is not optimized. We introduce an algorithm that can potentially improve any given schedule.

Our work is structured as follows. In Chapter 2, we formally define Coordinated Motion Planning in a Grid. We explain key aspects of graph theory as they play an important role in our solution. In Chapter 3, we adapt a proof by Demaine et al. [DFK$^+$19] to show that it is NP-complete to compute a schedule with a length that is lower than k (for a $k \in \mathbb{N}$) by reducing from Monotone-3SAT. After that, Chapter 4 covers the methods we use to calculate solutions to the instances. Our solution and the heuristics are then evaluated on the instances that were provided in the challenge in Chapter 5. We further discuss the approaches by other teams and how they differ from ours.

2. Preliminaries

This chapter introduces the main problem *Coordinated Motion Planning for Multiple Square-Shaped Robots in a Grid* that we aim to investigate within our work. As a prerequisite, it is necessary to define the basic terms we use to describe the problem throughout the thesis. Furthermore, we refer to parts of graph theory, since we consider graph-related algorithms to solve our problem.

2.1. Coordinated Motion Planning in a Grid

The following definitions are inspired by the rules that were given in the context of the "CG:SHOP 2021"-Challenge as well as the article by Demaine et al. [DFK$^+$19] who dealt with a similar problem. We slightly adapt the setting of the authors to fit our needs. In the following, we go into detail for every component of our problem.

Grid. We consider an infinite rectangular grid. A *cell* in this grid is an area bordered by the lines of the grid. By labeling an arbitrary cell with $(0, 0)$, we can index each cell by a unique index $(x, y) \in \mathbb{Z} \times \mathbb{Z}$.

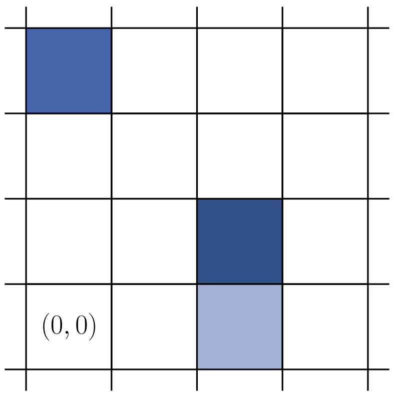


Figure 2.1.: Unbounded grid with three different robots at $(0, 3)$, $(2, 0)$ and $(2, 1)$

Robots. We have a set of n distinguishable robots that are located in the grid. Each robot is assigned a unique label i, which is an element of the set $L = \{1, \ldots, n\} \subseteq \mathbb{N}$. The robots are *unit-square-shaped*, as a consequence robots completely fill the cell they are currently located in (see Figure 2.1 for a visualization). A *position* of a specific robot with label $i \in L$ in a cell of the grid is denoted by a tuple $(x_i, y_i) \in \mathbb{Z} \times \mathbb{Z}$.

Start and Target Positions. Each robot has a unique *start position* and a unique *target position.* The start positions are described by a set $S = \{s_1, \dots, s_n\} \subseteq \mathbb{Z} \times \mathbb{Z}$, where s_i is the start position of robot i. The set $T = \{t_1, \dots, t_n\} \subseteq \mathbb{Z} \times \mathbb{Z}$ corresponds to the distinct target positions for each robot, where t_i depicts the target position of robot i. The sets S and T are not required to be disjoint. Therefore, the start position of robot i can potentially be the target position of any robot j and even $i = j$.

Obstacles. The set $O = \{o_1, \dots, o_l\} \subseteq \mathbb{Z} \times \mathbb{Z}$ represents l stationary obstacles located in the grid. Similar to robots, an obstacle completely fills out a cell in the grid. However, obstacles exist from the beginning and never change their positions during robot motion. The set of obstacles may be empty.

Robot Motion. Robots move in discrete time steps and all robots move at unit speed. At each unit of time, each robot can either stay at its position or move to an adjacent cell. There are four possible directions that a robot can move towards to: *west*, *north*, *east* and *south*. We do not allow diagonal movement. If a robot is at position (x, y) and decides to move west, its new position after completing the movement is $(x - 1, y)$. Analogously, this applies to the other three remaining directions as well. We denote such a *movement at one step of time* by $(x, y) \to (x - 1, y)$. A robot is only allowed to perform its motion as long as it does not collide with any other robot or obstacle.

Collisions. To define collisions, we allow time to be non-discrete only in this paragraph. A robot has to remain disjoint from all other robots and all obstacles at all times $t \in \mathbb{R}$. If a robot overlaps with a different robot or an obstacle, we call this a *collision.* We need to closely observe a special collision between robots that might occur because of their square shape. If there is a robot at position (x, y) and a different robot at position $(x + 1, y)$, then the robot at position (x, y) can only move east into position $(x + 1, y)$ if the robot at position $(x + 1, y)$ moves east as well. Visually speaking, the two robots remain in contact during their movement, but they never overlap (see Figure 2.2). If the robot at position $(x + 1, y)$ moves south or north in this scenario, the robot at position (x, y) would have to wait one time step before going east, otherwise we call this a *follow collision.*

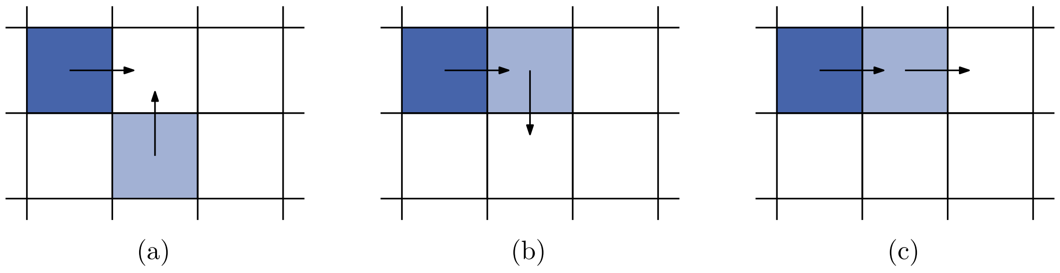


Figure 2.2.: (a) This motion is not legal as two robots end up at the same position. (b) This motion is not legal as a follow collision occurs. (c) This motion is collision-free.

We will subsequently define a decision problem, while related optimization problems are introduced at a later time. The decision problem itself will be investigated in Chapter 4 to show that it is decidable given a small restriction.

Definition 2.1. *Coordinated Motion Planning in a Grid*

Input: Given are n labeled robots, a set of start positions $S = \{s_1, \dots, s_n\}$, a set of target positions $T = \{t_1, \dots, t_n\}$ and a set of obstacles $O = \{o_1, \dots, o_l\}$.

Question: Is there a sequence of collision-free moves for all n robots so that robot i moves from s_i to t_i?

We further introduce the following terms and definitions to describe our problem. A *configuration* keeps track of the locations of all robots and obstacles at a time $t \in \mathbb{N}_0$. We define this as a mapping $C : \mathbb{Z} \times \mathbb{Z} \to \{1, \dots, n, \square, \blacksquare\}$. The restricted function $C|_R$ is injective, where $R \subseteq \mathbb{Z} \times \mathbb{Z}$ describes the positions of the robots. The empty square represents an empty cell and the black square denotes a cell blocked by an obstacle. The inverse image of a robot's label i is defined by $C^{-1} : \{1, \dots, n\} \to \mathbb{Z} \times \mathbb{Z}$. $C^{-1}(i)$ represents the position (x_i, y_i) of robot i in the grid. A configuration C_1 is *transformed at one time step* into another configuration C_2, if the motion of each robot during this time step is collision-free. This is particularly the case if each robot stays at its position or moves towards one of the four directions while respecting the rules set in the previous paragraphs. We denote this by $C_1 \to C_2$.

We refer to a *start configuration* as C_s and a *target configuration* as C_t. For $i \in L$, $C_s^{-1}(i) = s_i$ is the start position and $C_t^{-1}(i) = t_i$ is the target position of robot i. In the start configuration C_s, all robots are at their start positions and in the target configuration C_t, all robots are at their target positions. A *schedule* is a sequence of configurations $C_1 \to \dots \to C_M$, for $M \in \mathbb{N}$, where C_1 is the start configuration C_s and C_M is the target configuration C_t. Refer to Appendix B for an exemplary schedule. We call the number M of steps in a schedule the *makespan*. The *optimal makespan* is the minimum number of steps in any schedule. The number of steps a single robot performs in a given schedule is called the *distance*. The sum of all distances is defined as the *total distance* and the *optimal total distance* is the lowest possible total distance in any schedule. We also define the *optimal distance* of a single robot as the length of a shortest path between its start and target position. It should be noted that without obstacles, this distance is equal to the *manhattan distance* between the start and target position. Given two coordinates $p = (p_1, p_2)$ and $q = (q_1, q_2)$, the manhattan distance d is defined as $d = |p_1 - q_1| + |p_2 - q_2|$. If the set of obstacles is not empty, it is possible to use a breadth-first search to determine the optimal distance, which will be explained in Subsection 2.2.1.

Given the start and target positions of all robots, we call the integer axis-aligned rectangle that includes all start and target positions and all obstacles the *initial bounding box*. Furthermore, we define the *border* as the outer cells of this rectangle that are not part of the bounding box (see Figure 2.3 for an example). Robot motion is not limited by the initial bounding box. Robots can move further away in any direction if desired. Similar to the initial bounding box, the *bounding box at a time t* is the rectangle that includes all cells with robots currently located in, all target positions, and all obstacles.

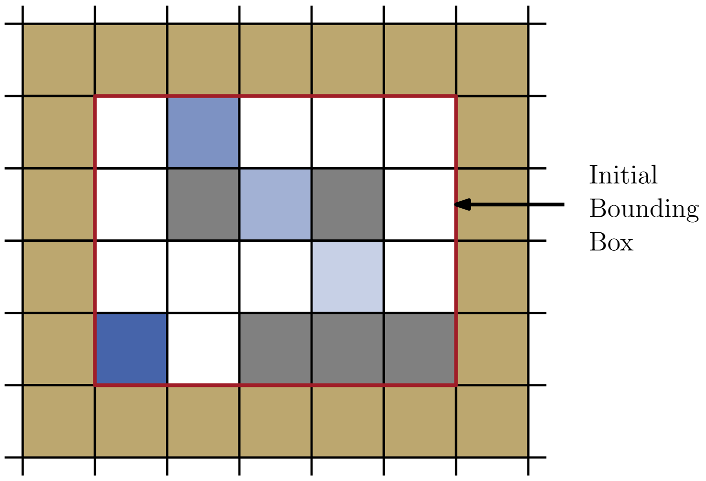


Figure 2.3.: Brown cells mark the border, the initial bounding box is highlighted red. The box includes all entities, namely four robots (blue) and five obstacles (gray).

We define another decision problem that asks to find a schedule with makespan at most k, for a $k \in \mathbb{N}$, and two optimization problems that ask to find optimal solutions for the makespan and total distance.

Definition 2.2. *Makespan*

Input: Given are n labeled robots, a set of start positions $S = \{s_1, \ldots, s_n\}$, a set of target positions $T = \{t_1, \ldots, t_n\}$ and a set of obstacles $O = \{o_1, \ldots, o_l\}$.

Question: Is there a schedule with makespan $M \leq k$, for a $k \in \mathbb{N}$?

Definition 2.3. *Optimal Makespan*

Input: Given are n labeled robots, a set of start positions $S = \{s_1, \ldots, s_n\}$, a set of target positions $T = \{t_1, \ldots, t_n\}$ and a set of obstacles $O = \{o_1, \ldots, o_l\}$.

Question: Find a schedule with optimal makespan.

Definition 2.4. *Optimal Total Distance*

Input: Given are n labeled robots, a set of start positions $S = \{s_1, \ldots, s_n\}$, a set of target positions $T = \{t_1, \ldots, t_n\}$ and a set of obstacles $O = \{o_1, \ldots, o_l\}$.

Question: Find a schedule with optimal total distance.

These problems seem difficult to solve. We prove NP-completeness for Makespan. The two objectives in this thesis are as follows:

1. Minimize the total distance
2. Minimize the makespan

In the context of the challenge, these two objectives are considered separately. However, in this thesis we propose one solution with the aim to minimize both objectives at the same time. Our solution consists of multiple steps, where each step contributes differently to the objectives. Sometimes, a step of our solution only considers one objective at a time. We can easily find lower bounds for the two objectives. For the makespan, a valid lower bound is the maximum distance between any robot's start and target position. If the makespan was lower than this distance, the robot with the maximum distance would not reach its target position. For the total distance, a lower bound is given by the sum of all distances between every robot's start and target position. In that case, every robot can move directly to its target position without having to make a detour.

2.2. Graph Theory

We will apply known algorithms that are related to graphs to solve our problem. Therefore, we preliminary define a few associated terms and introduce some graph problems and the corresponding algorithms.

A *graph* is a pair $G = (V, E)$ described by a finite set of vertices V and edges E between vertices. In a *directed graph*, the edges have a direction, meaning that edges are described by a set $E \subseteq \{(x, y) \mid x \in V \wedge y \in V\}$. On the contrary, the edges in an *undirected graph* do not have a direction. Hence, the set of edges is defined as as $E \subseteq \{\{x, y\} \mid x \in V \wedge y \in V\}$. We may assign a weight to each edge by defining a *weight function* $w : E \to \mathbb{Z}$ and call the resulting graph a *weighted graph*. A graph is called *bipartite*, if the vertices can be partioned into two disjoint sets so that every edge connects two vertices from different sets. A *complete bipartite graph* is a bipartite graph, where every vertex of the first set is connected to every vertex of the second set.

2.2.1. Breadth-First Search

As the *breadth-first search* (BFS) plays an important role within this thesis, we briefly describe the underlying mechanism. A detailed explanation can be found in the book by Cormen et al. [CLRS09]. Given a graph G, a source node s and a target node t, the breadth-first search traverses the graph starting at the source node, which is initially put into a queue. In a loop, the front element of the queue is popped and all of the neighbor nodes are explored. If they have not been visited yet, they are added to the queue. This process repeats until the queue is empty or the target node has been explored. The pseudocode is depicted in Algorithm 2.1. The time complexity of this algorithm is $O(n+m)$, where n is the number of vertices and m is the number of edges.

We also use the BFS to determine the distance between the nodes of a graph in relation to a source node s. We do not have an explicit target node as we aim to explore all nodes reachable from the source node. The source node is assigned a distance of 0 and the algorithm visits all the source node's neighbors and sets their distance to 1. Afterwards, the BFS explores all the unvisited neighbors of nodes that have distance 1 and assign those new nodes a distance of 2. The described process is repeated until all reachable nodes have been visited resulting in a distance for all reachable nodes in relation to a source node. In our case, the edges that were explored to reach a certain node describe a shortest path between two positions in the grid.

Algorithm 2.1: Breadth-first search

```
Input: Graph $G = (V, E)$, source node $s$, target node $t$
Data: Queue Q

// Initialization
1  Q.enqueue($s$)
2  mark $s$ as visited and all other nodes as not visited

// Main loop
3  while Q is not empty do
4      $v \leftarrow$ Q.dequeue()
5      if $v = t$ then
6          return
7      forall $(v, w) \in E$ do
8          if $w$ is not visited then
9              mark $w$ as visited
10             Q.enqueue($w$)
```

2.2.2. Minimum-Cost Maximum-Matching

An important term in relation to graphs is a *matching.* A matching in a graph is a set of pairwise non-adjacent edges, that means no two edges share a common vertex. We call the number of edges in the matching its *cardinality.* A *maximum matching* is a matching whose cardinality is maximum among all possible matchings in a given graph. We consider a directed, complete bipartite graph with the partitioned sets X and Y, where all edges go from X to Y. Furthermore, this graph has a weight function $w : E \to \mathbb{Z}$. We describe the calculation of a maximum matching, where the sum of the weights of all edges involved is minimal (*minimum-cost maximum-matching*), as we will refer to this calculation in Subsection 4.1.2.

We reduce this problem to a *minimum-cost maximum-flow* problem. Therefore, we add two nodes to the existing graph: a source s that has a directed edge to all vertices in X, and a sink t that has incoming edges from all vertices in Y. These newly added edges are assigned a cost of 0 and a capacity of 1. All other edges, meaning the ones that already existed, are assigned a capacity of 1 and the cost is equal to the weight.

In the following, we provide a quick overview about an algorithm that solves this problem, referring to [MCF] for a detailed implementation of the algorithm. Given our modified graph, for every edge, we add the reverse edge with capacity 0 and negative costs. Iteratively, we look for a shortest path from s to t considering the cost of the edges. The flow along the shortest path is augmented by 1 and the reverse edges on this path increase the residual capacity by 1. This iteration is repeated until no path exists from s to t, at that point the algorithm terminates and a solution is found. To obtain our desired minimum-cost maximum-matching, we simply take the edges that are part of the resulting flow. As we deal with edges with negative costs, we use the Bellmann-Ford algorithm to compute shortest paths. An elaborated explanation on the Bellman-Ford algorithm can be found in the book by Cormen et al. [CLRS09]. Its time complexity is $O(nm)$, where n is the number of vertices and m is the number of edges. Overall, this leads to a time complexity of $O(n^2m^2)$ for the minimum-cost maximum-flow.

It should be noted that this is not the fastest algorithm to compute a minimum-cost maximum-matching. More refined algorithms such as the Hungarian algorithm exist to deal with this problem in a shorter time, see the original publication by Kuhn [Kuh55]. Its time complexity is $O(n^3)$.

3. NP-Completeness

In this chapter, we show NP-completeness for Makespan (Definition 2.2). We mainly follow the proof that was given by Demaine et al. [DFK$^+$19], but slightly adapt their proof to fit our setting. In the setting of the authors, robots are not unit-square-shaped and as a consequence, different robot motions are possible. For their proof, Demaine et al. reduce from the problem Monotone 3-Sat:

Definition 3.1. *Monotone 3-Sat*

Input: Given is a formula ϕ of clauses where each clause consists of only positive or only negative literals and each clause contains three literals.

Question: Is there an assignment for the variables that satisfies the formula ϕ?

Monotone 3-Sat is known to be NP-complete, refer to problem L02 in the book of Garey and Johnson [GJ79] as well as the article by Gold [Gol78]. The reduction from Monotone 3-Sat works as follows. Given a formula ϕ, we construct an instance of the coordinated motion planning problem in a grid. This constructed instance has makespan M if and only if ϕ is satisfiable and $M + 1$ otherwise. The exact value of M will be determined at the end.

The high-level idea is to represent every variable as a *variable robot*, which is forced to move on exactly one of two paths. Forcing a variable robot to take one of two paths is accomplished by introducing two *auxiliary robots* for each variable robot. The two possible paths represent a truth assignment for the variable. Each clause has three *checker robots*, one for each literal in the clause. The checker robots cross paths with the corresponding variable robots and have to wait one time step if the truth assignment does not match. The checker robots for negative clauses start below all variable robots and move north, while the checker robots for positive clauses start above all variable robots and move south. Each clause further has a *clause robot* that crosses paths with the corresponding checker robots. The clause robot checks if its clause is satisfied by making sure that at least one variable in its clause is satisfied. In the end, a makespan M is feasible iff ϕ is satisfiable. A high-level sketch can be found in Figure 3.1.

For the remainder of this chapter, let the formula ϕ have n variables $\{x_0, \dots, x_{n-1}\}$ and m clauses $\{C_1, \dots, C_m\}$. Moreover, let the clauses be ordered in a way that negative clauses come before positive clauses. Formally, there is a c, so that C_i is a negative clause for all

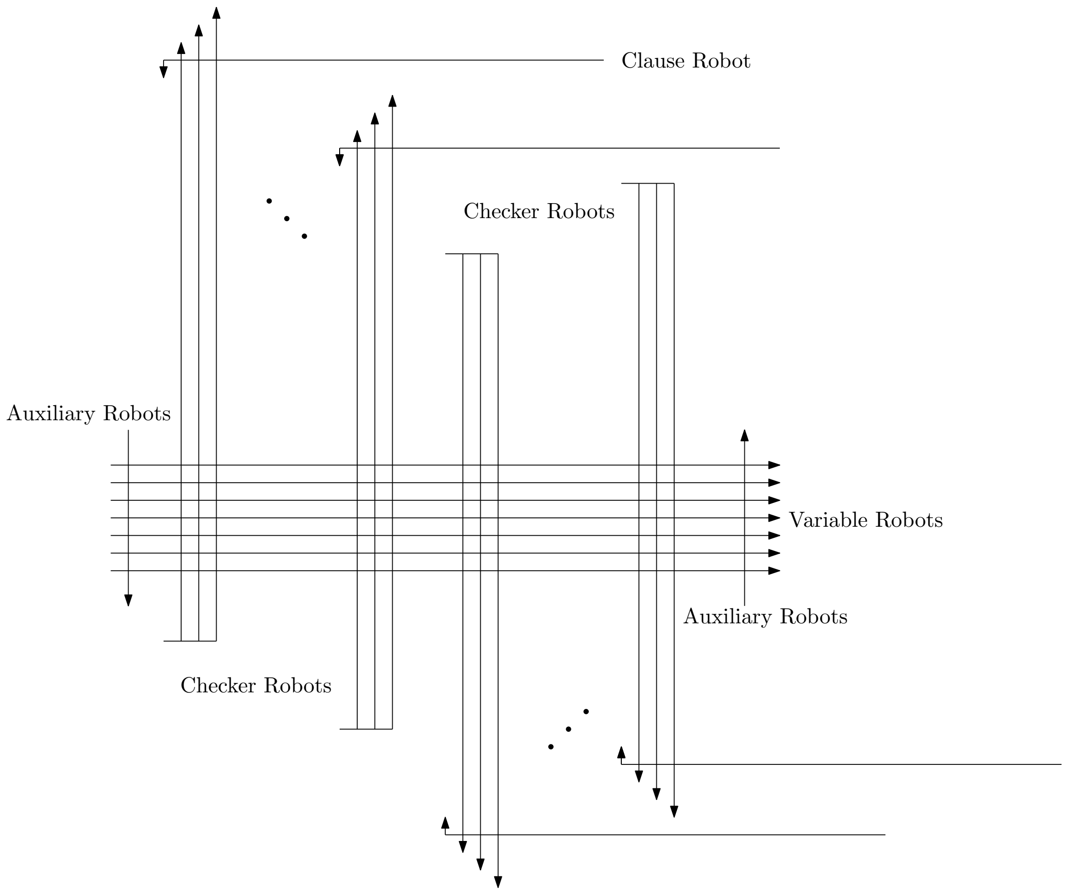


Figure 3.1.: High-level sketch of the proof, similar to the idea by Demaine et al. [DFK+19]. The variable robots move east. There are left and right auxiliary robots for each variable robot. Negative checker robots move north, while positive checker robots move south. Clause robots move west and cross paths with the corresponding checker robots.

$1 \leq i \leq c$ and C_i is a positive clause for all $c < i \leq m$. In the following, we go into detail for every component that is needed for the proof.

Variable Robots. For each variable x_j, we construct a *variable gadget* consisting of a *variable robot* and two *auxiliary robots*. The *left auxiliary robot* of a variable x_j has start position $(1, 6j)$ and target position $(1, 6j - M)$. The *right auxiliary robot* of a variable x_j has start position $(M - 3, 6j - M)$ and target position $(M - 3, 6j)$. The variable robot for variable x_j starts at position $(0, 6j)$ and moves to its target position $(M - 2, 6j)$.

Lemma 3.2. *Each variable robot moves on one of exactly two paths towards its target position.*

Proof. We consider the variable gadget for a variable x_j. The left auxiliary robot moves M steps south and the right auxiliary robot moves M steps north. As the distance they need to move is equal to the makespan, both auxiliary robots have to move at every time step without waiting. The variable robot's target position is located $M - 2$ cells east of its start position. Thus, the variable robot is allowed to wait two time steps.

The variable robot cannot move east at the first time step, as this would lead to a follow collision with the left auxiliary robot. The variable robot cannot move south or west either, because it would collide with the right auxiliary robot at the last time step.

Therefore, the variable robot is forced to wait or move north at the very first time step. Afterwards, the variable robot has to move east towards its target position. It cannot wait or move south, because this would result in a collision with the right auxiliary robot. As a consequence, in any schedule, each variable robot has x-coordinate $k-1$ after the kth time step ($1 \leq k \leq M-3$) and either y-coordinate $6j$ or $6j+1$. □

Figure 3.2 depicts the placement of the auxiliary robots and the variable robots. The latter are forced to move on one of two paths (top and bottom) that represent a truth assignment. The top path represents *false*, whereas the bottom path represents *true*.

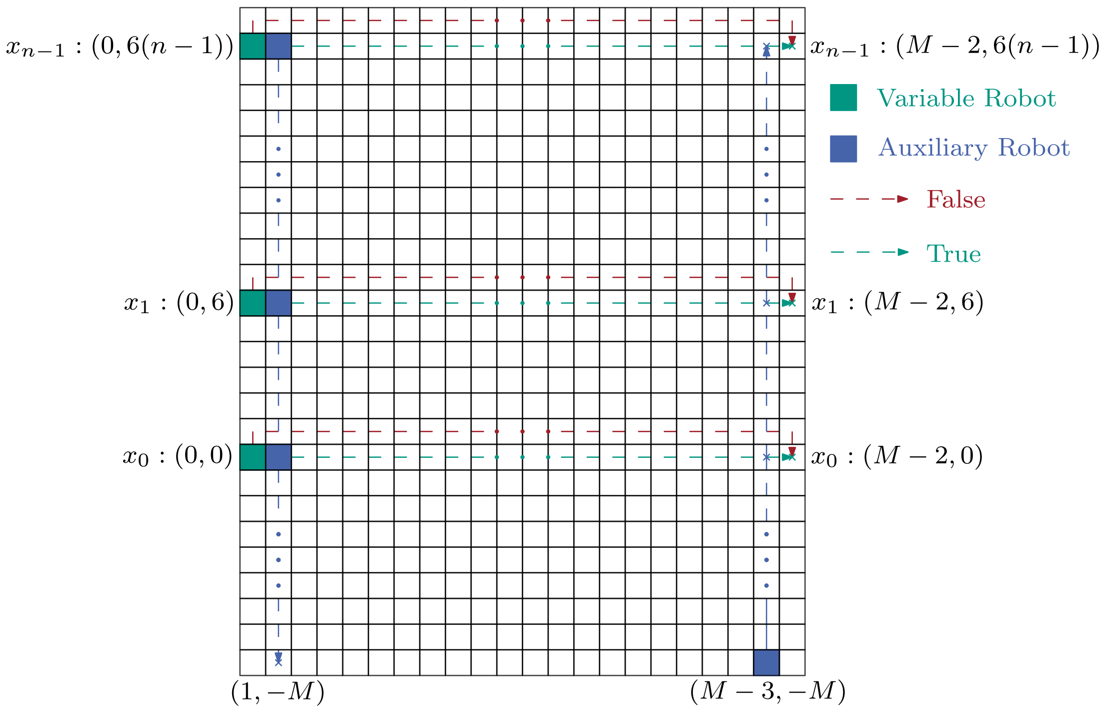


Figure 3.2.: A sketch of the start configuration for variable gadgets. Variable robots are forced to move on exactly one of two paths.

Checker Robots. We introduce *negative checker robots* for negative clauses and *positive checker robots* for positive clauses. First, we consider negative checkers. The general procedure for positive checkers remains the same, but we slightly adapt the start and target positions. For each negative clause $C_i = \{\overline{x}_{j_1}, \overline{x}_{j_2}, \overline{x}_{j_3}\}$ with $j_1 < j_2 < j_3$, there are three negative checker robots c_i^1, c_i^2, c_i^3. Their task is to check whether their corresponding literal satisfies the clause. The aim is to force a checker to wait one time step if the assignment is not satisfied. The start positions of negative checker robots are:

$$\alpha_i^1 := (6(ni + j_1), -6ni - 2)$$
$$\alpha_i^2 := (6(ni + j_2), -6ni - 2)$$
$$\alpha_i^3 := (6(ni + j_3), -6ni - 2)$$

The y-coordinates of the start positions for checkers of the same clause are identical. The x-coordinates differ depending on the variable the checkers are designated to check.

Checker c_i^3 moves $M-1$ steps north. Therefore, its target position is $t_i^3 := \alpha_i^3 + (0, M-1)$. The pathing of the other two checkers, c_i^1 and c_i^2, requires a more refined routing. They also move $M-1$ steps, but it is necessary to introduce a vertical offset between the checkers by performing side steps. This is crucial for the clause gadget, which is introduced later, to function as intended. Let $d_1 := 6(j_3 - j_1)$ and $d_2 := 6(j_3 - j_2)$. Visually speaking, d_1 is the horizontal distance between the initial positions of c_i^1 and c_i^3. Analogously, d_2 is the horizontal distance between the initial positions of c_i^2 and c_i^3. Since d_1 and d_2 are always even and at least twelve and six respectively as per construction, we define the sidesteps as $s_1 := d_1/2 + 4 < d_1$ and $s_2 := d_2/2 + 2 < d_2$. It should be noted that s_1 is greater than 9, s_2 is greater than 4 and both always have integer values. We force c_i^1 to make s_1 steps east and the remaining steps north. Therefore, the target position of checker c_i^1 is $t_i^1 := \alpha_i^1 + (s_1, M - 1 - s_1)$. Analogously, the target position of checker c_i^2 is $t_i^2 := \alpha_i^2 + (s_2, M - 1 - s_2)$. We force checkers to perform their sidesteps towards east before moving north by introducing auxiliary robots (see Figure 3.3). Moving east does not change the positions of checker robots relative to the variable robots, as they move east as well.

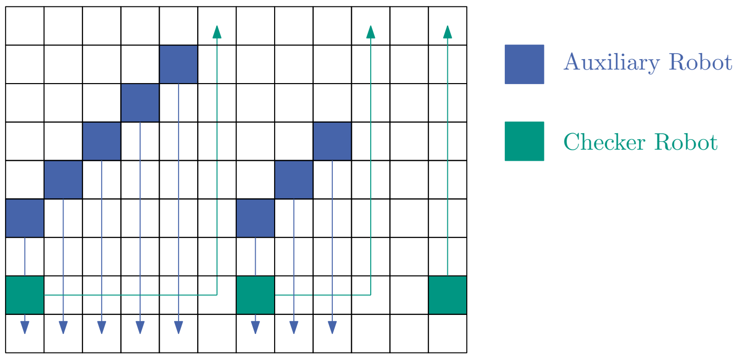


Figure 3.3.: Checker robots for a negative clause. The first two negative checker robots are forced to move east before going north because of the auxiliary robots.

Lemma 3.3. *Each negative checker robot has to wait one time step if the assignment of its corresponding literal is not satisfied.*

Proof. Let C_i be a negative clause. As the distance negative checkers need to move is equal to $M - 1$, they are allowed to wait one time step. All negative checkers start below the variable robots and move north after performing their sidesteps. At time $t = (6ni + j) + 1$, the checker for variable x_j and the corresponding variable robot are on the same x-coordinate $(6ni + j)$. The checker has y-coordinate $6j - 1$, whereas the variable robot has either y-coordinate $6j$ or $6j + 1$. The situation is depicted in Figure 3.4. Since we deal with a negative clause, the negative checker shall wait if the variable robot chose the bottom path (*true*). This is the case, because if the checker robot does not wait, a follow collision occurs. It cannot not go in any other direction, as it is only allowed to wait once. If the variable robot chose the top path (*false*), both robots can continue on their route without having to wait. Therefore, a negative checker robot has to wait one time step if its corresponding literal is not satisfied. □

We apply the same idea to specify the movements of positive checkers for a positive clause $C_i = \{x_{j_1}, x_{j_2}, x_{j_3}\}$ with $j_1 > j_2 > j_3$. Positive checkers start above all variable robots and move south. The start positions of the positive checker robots are:

$$\beta_i^1 := (6(ni + j_1), 6(n - 1) + 6ni + 3)$$
$$\beta_i^2 := (6(ni + j_2), 6(n - 1) + 6ni + 3)$$
$$\beta_i^3 := (6(ni + j_3), 6(n - 1) + 6ni + 3)$$

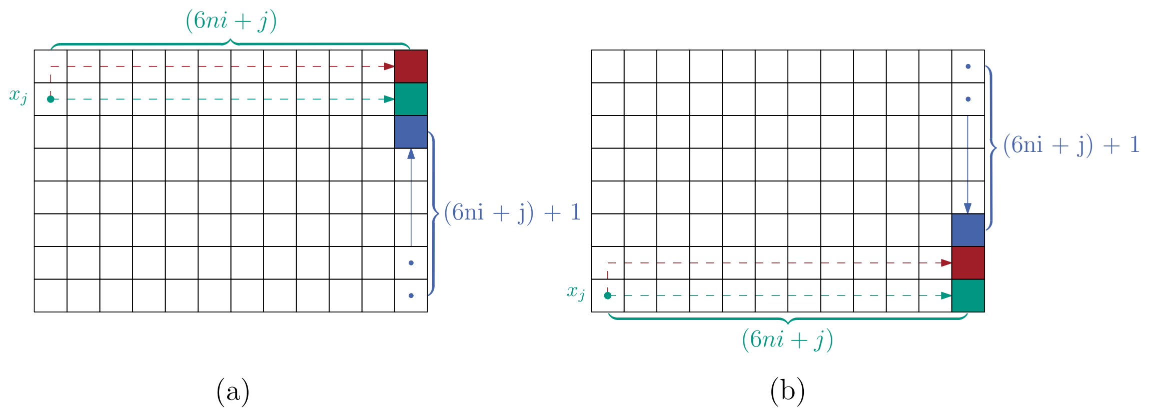


Figure 3.4.: (a) The situation of a negative checker robot and the corresponding variable robot at time $t = (6ni + j) + 1$. It needs to be remembered that the variable robot does not move east at the first time step. If the variable robot was on the *true* path, the checker would have to wait one time step to avoid a collision. If the variable robot was on the *false* path, the assignment would be satisfied and the checker robot would be able to pass without waiting. (b) The same situation for a positive checker and the corresponding variable robot. The procedure remains the same.

All positive checkers of the same clause start at the same y-coordinate. In contrast to negative checkers, the variable with the highest index is checked by checker c_i^1. The target positions of positive checkers are determined analogously to negative checkers and can be described by $u_i^3 = \beta_i^3 + (0, -M + 1))$ for c_i^3, $u_i^2 := \beta_i^2 + (s_2, -M + 1 + s_2)$ and $u_i^1 := \beta_i^1 + (s_1, -M + 1 + s_1)$ for c_i^2 and c_i^1 respectively. Positive checkers are also forced to move east before going south by introducing further auxiliary robots.

Lemma 3.4. *Each positive checker robot has to wait one time step if the assignment of its corresponding literal is not satisfied.*

Proof. Let C_i be a positive clause. The proof works analogously to Lemma 3.3. The term $(6(n-1))$ in the y-coordinate of a checker's start position represents the y-coordinate of the topmost variable robot. Therefore, at time $t = (6ni + j) + 1$, the checker for variable x_j and the corresponding variable robot are on the same x-coordinate. The checker robot has y-coordinate $6j + 2$, while the variable robot has either y-coordinate $6j$ or $6j + 1$. Refer to Figure 3.4 for a visualization. If the variable robot chose the top path (*false*), the positive checker would have to wait one time step to avoid a follow collision. If the variable robot was on the bottom path (*true*), both robots can continue moving. Therefore, a positive checker robot has to wait one time step if its corresponding literal is not satisfied. □

Clause Robots. For each clause C_i, we construct a *clause gadget* consisting of a *clause robot* that crosses paths with its corresponding checker robots. The clause robot makes sure that the clause is satisfied by ensuring that at least one variable in the clause is satisfied. This is the case if at least one of the three checkers for the clause did not have to wait when crossing paths with the variable robot. We introduce *negative clause robots* for negative clauses and *positive clause robots* for positive clauses. The start position of a negative clause robot is $t_i^1 + (M - 7, 0)$ and the target position is $t_i^1 - (3, 4)$. The start position of a positive clause robot is $u_i^1 + (M - 7, 0)$ and the target position is $u_i^1 - (3, -4)$.

Lemma 3.5. *Every clause robot moves M steps without waiting towards its target position if at least one checker robot of its clause did not wait when crossing paths with a variable robot.*

Proof. A negative clause robot is located to the right of the checkers and above the variable robots. It has to move $M-4$ steps west and four steps south (it does not have to move west first). Therefore, the clause robot has to move at each time step. Checker robots have to wait one time step if the assignment for their variable is not satisfied. The vertical offsets of the checkers c_i^1 and c_i^2 introduced earlier ensure that the clause robot may only pass through the checkers without waiting, if at least one of the checker robots did not wait. All three checkers are placed in a way to force the clause robot to move on one of three paths (see Figure 3.5 for a visualization). It should be noted that the clause robot cannot move in between these paths, as this causes one checker to wait, however, the checkers also have to move without waiting if their assignment is not satisfied. The positive clause robots follow the exact same procedure. □

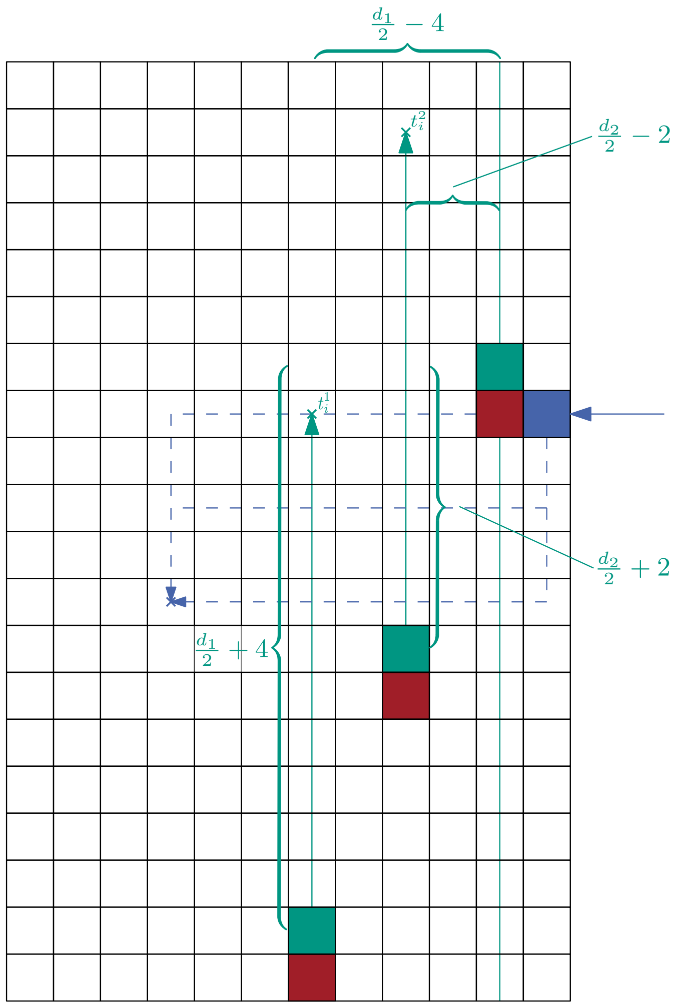


Figure 3.5.: A clause gadget for a negative clause. The clause robot (blue) crosses path with its corresponding checkers. The green squares represent the position of a checker that did not wait and the red squares are checkers that did wait one time step when crossing paths with the variable robot. The clause robot is allowed to pass a checker without waiting if the respective checker did not wait. After passing the checker, the clause robot can safely move to its target position. If checker c_i^3 did wait, the clause robot has to move south, as it is not allowed to wait. It cannot move west as this would cause a follow collision. If all three checkers did wait, the clause robot cannot reach its target position without waiting.

Theorem 3.6. *Makespan is NP-hard.*

Proof. We need to specify the critical makespan M. In the following, we introduce some conditions that M needs to meet and then set the makespan accordingly. On the one hand, the makespan must be large enough, so that the checkers of the last clause C_m can pass through the variable robots and their clause robots. On the other hand, the makespan must also allow the variable robots to cross paths with all checkers. Since the checkers of the last variable travel left of the line $x = 6n(m+1) - 6$, a makespan $M \geq 6n(m+1)$ is sufficient for the variable robots.

If the last clause is negative, the starting positions of its checkers are located on the line $y = -6nm - 2$. The variable robot of the last variable x_{n-1} travels below the line $y = 6(n-1) + 1$. We make sure that the negative clause gadgets are located strictly above all variables. Taking into account the positioning of the clauses, it is necessary for the checker that checks the first literal of the last clause C_m to have its target position above the line $y = 6(n-1) + 7$. Without taking the side steps performed by the checkers into consideration, the makespan has to be set as $M \geq (6nm+2)+(6(n-1)+7) = 6n(m+1)+7$. The number of side steps of each checker is less than $6n$. Hence, setting the makespan to $M \geq 6n(m+2) + 7$ is sufficient in this case.

To avoid any collisions between clause robots and checkers of a different clause, we ensure that the path of the clause robot of the last negative clause stays above the starting positions of all positive checkers. The critical clause is the last clause C_m. If it is a negative clause, the argumentation of the last paragraph holds. If it is a positive clause, the starting positions of its checkers are located on the line $y = 6(n-1) + 6nm + 3$. In the worst case, the clause C_{m-1} is a negative clause. The checkers of the clause C_{m-1} all start at $y = -6n(m-1) - 2$. Therefore, the clause robot of clause C_{m-1} has to move above the line $y = 6(n-1)+6nm+3$. In fact, the checkers of clause C_{m-1} need to have their target positions above the line $y = 6(n-1) + 6nm + 9$ due to the positioning of the clause gadget. All of these criteria lead to a makespan $M \geq (6(n-1)+6nm+9)+(6n(m-1)+2) = 6n(2m)+5$. Since the number of side steps of each checker is less than $6n$, the makespan has to be $M \geq 6n(2m+1)+5$. Therefore, in any case, we set the critical makespan as $M = 6n(2nm+2)+7$.

To show NP-hardness, we recap the construction of the coordinated motion planning instance. A makespan of M is feasible if and only if for every clause the clause robot reaches its target position without having to wait. Lemma 3.5 shows that this is only possible if at least one of the checkers for every clause does not wait when crossing paths with the variable robot. Lemma 3.4 and Lemma 3.3 then imply that the clause has a satisfied literal which results in the clause being satisfied. A makespan of M is feasible iff ϕ is satisfiable. □

Further remarks. All coordinates and the makespan M in the above proof are polynomial in the input size, implying strong NP-hardness. As we can easily check in polynomial time if a given schedule is legal (see Appendix A) and if its makespan is lower than k, Makespan is in NP. As a consequence of Theorem 3.6, Makespan is NP-complete. Furthermore, we do not make use of obstacles in this proof. The problem is NP-complete even when there are no obstacles.

4. Solving Coordinated Motion Planning in a Grid

In this chapter, we present an algorithm that solves the coordinated motion problem described in Section 2.1. The basic idea of our approach is based on a two-step process. First, we generate a feasible schedule, which is not of high quality with respect to minimizing the makespan and the total distance. To find such a schedule, we move all robots out of the initial bounding box. Once every robot is outside of the initial bounding box, we move the robots to their respective target position. We show that, if every robot has a path to the border without obstacles, we always find a schedule by routing robots sequentially. We slightly improve this initial solution by coordinating a subset of robots at a time rather than performing sequential motions. While creating the initial schedule we use different heuristics, which we will evaluate in Chapter 5. Second, we further improve the initial solution to obtain a schedule that yields optimized results concerning the makespan and the total distance. In the second phase, we keep the paths of all robots fixed except for one and improve the path of this specific robot. The procedure is repeated for all robots.

4.1. Moving Robots out of the Initial Bounding Box

We describe a method that provably moves all robots out of the initial bounding box. While robots are on their way out, they have to avoid collisions. This enables us to establish a certain order for the robots to move and we only move a subset of robots at a time. By routing robots according to the order, we are guaranteed to find a feasible schedule.

4.1.1. Calculating Different Parking Positions

We introduce two different ways of calculating *parking positions*. Parking positions are cells located outside of the initial bounding box. They can be considered as intermediate positions for robots: a robot starts at its start position and moves to its target position, but has to pass its parking position on its route. In the following, we focus on determining the parking positions to further elaborate on the decision regarding which parking position belongs to which robot in Subsection 4.1.2.

As our purpose is to keep the total distance as small as possible, the robots are not supposed to perform too many moves. Thus, we keep parking positions relatively close to the border. Parking positions shall be apart from each other to ensure that there are always paths on which the robots can move on. Another crucial factor for our computation is to keep the new bounding box, including all parking positions, at a small level.

For n robots, we need n parking positions. Let the bottom left cell inside the initial bounding box be the cell $(0, 0)$ and let the length of the box be odd. We call a cell *empty* if it does not become a parking position. All cells of the border stay empty.

Option A. We surround the border with parking positions in every direction while leaving certain cells empty at the same time. Cells positioned south or north of the bounding box with an odd x-coordinate do not become parking positions. The same rule applies to cells located west and east of the bounding box with an odd y-coordinate. The corners (southwest, etc.) are treated as if they were south or north of the box.

Option B. This method requires all robots to not have adjacent neighbors at all. The procedure is similar to Option A. To provide parking positions with even more space, we additionally leave cells positioned north and south with an odd y-coordinate empty. This applies to cells located west and east with an odd x-coordinate as well.

A visual example for the described arrangements of parking positions can be found in Figure 4.1.

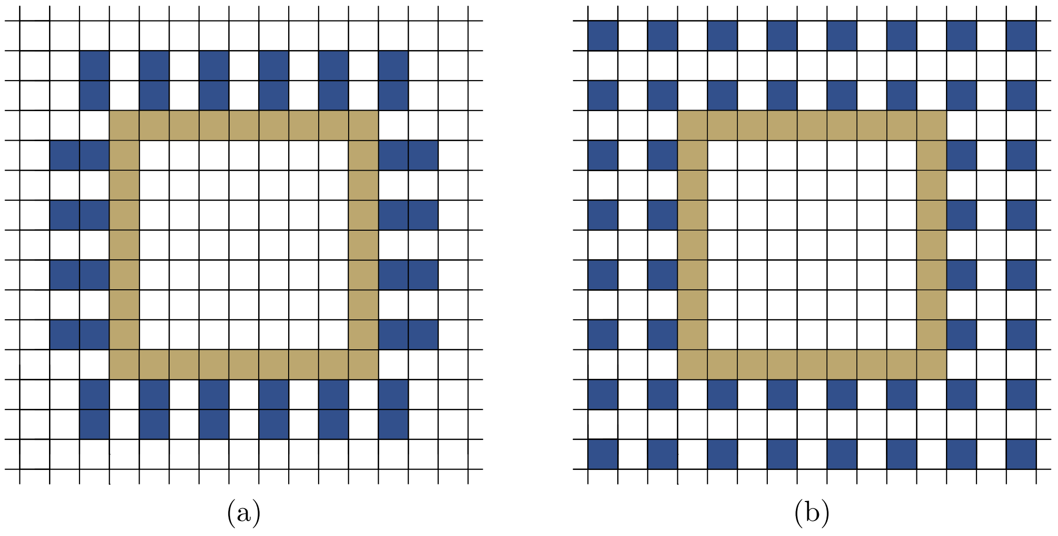


Figure 4.1.: The blue cells are parking positions and the brown cells mark the border. (a) Some parking positions are adjacent to each other. (b) No parking positions are adjacent to each other.

4.1.2. Assigning Robots to Different Parking Positions

Given n parking positions for n robots, we need to assign each robot to a different parking position. Although we could randomly assign robots to parking positions, we deliberately choose to keep the total distance as low as possible. Therefore, we introduce a more suitable approach building on the fundamentals of graph theory as explained in Subsection 2.2.2.

We define the start positions as a set X and the parking positions as a set Y. We further define a directed graph $G = (V, E)$ with vertices $V = X \cup Y$. Each vertex $x \in X$ has a directed edge to each vertex $y \in Y$, the set of edges is $E = \{(x, y) \mid x \in X \wedge y \in Y\}$. This graph is complete bipartite as per construction and represents our assignment problem. Each parking position is a possible parking position for every robot, but we can only assign at most one robot to a parking position. After defining a weight function $w : E \to \mathbb{Z}$, the edges in a minimum-cost maximum-matching in this graph represent our assignment.

We further deal with the question of how to set the weight function w. Let $d(x, y)$ be the obstacle-free distance between two positions x and y obtained by a breadth-first search. $C_t^{-1}(C_s(u))$ is the target position of the robot with start position u. A parking position is denoted by v. We propose three different heuristics for the weight function:

1. $(u, v) \mapsto d(u, v)$
2. $(u, v) \mapsto d(C_t^{-1}(C_s(u)), v)$
3. $(u, v) \mapsto d(u, v) + d(C_t^{-1}(C_s(u))$

The first possibility sets w as the distance between the start and parking position. Therefore, robots tend to have a short distance from their start position to their parking position, however, when they move from their parking position to their target position, they might have to perform considerably more moves. The second possibility functions the other way around and sets w as the distance between the target and parking position. The third possibility takes into account both distances described in the previous options. As a consequence, this might lead to less movement overall. All of these weight functions seem superior to assigning parking positions randomly.

4.1.3. Generating an Order

The next step is to move the robots to their parking positions. Instead of moving the complete number of robots simultaneously, we rather move several subsets at a time to avoid any collisions. We refer to subsets as *waves* of robots. The general procedure is to assign each robot to a wave based on an order. Once all robots of a wave reach their parking positions, the next wave is allowed to move.

To achieve our purpose of proving that we are able to move all robots out of the initial bounding box, we create an order for the robots to move. To this end, we determine a cell of the border as the *reference cell.* In particular, we choose the cell below the top right cell of the border as our reference cell. Since the border consists only of cells without robots and obstacles, a robot that has a path to the border automatically has a path to the reference cell as well.

As described in Subsection 2.2.1, a breadth-first search can be used to calculate the distance between each start position in the initial bounding box and the reference cell. The source node is the reference cell and the target node is a start position. It is vital for this BFS to not explore positions blocked by an obstacle. Therefore, we find a shortest path that goes around the obstacles. It should be noted that without obstacles, the distances can be obtained by calculating the manhattan distances between the reference cell and the start positions. The distance determines the wave that a robot belongs to, robots with the same distance belong to the same wave. The idea behind this approach is simple. Sometimes a robot is surrounded by obstacles, and there is a cell that the robot inevitably has to pass to reach the reference cell. To avoid that this specific cell is blocked by another robot, the robot covering the cell needs to move first. In this way, it creates new space for the robot that is "behind" it. Figure 4.2 exemplary illustrates this situation.

4.1.4. Moving Robots to their Parking Position

We move the robots to their parking positions according to their distances to the reference cell. The following Lemma 4.1 shows that we can route every robot sequentially to the parking positions via the reference cell.

Lemma 4.1. *Let every robot have path to the border that is not blocked by obstacles. We find a schedule to move every robot out of the initial bounding box.*

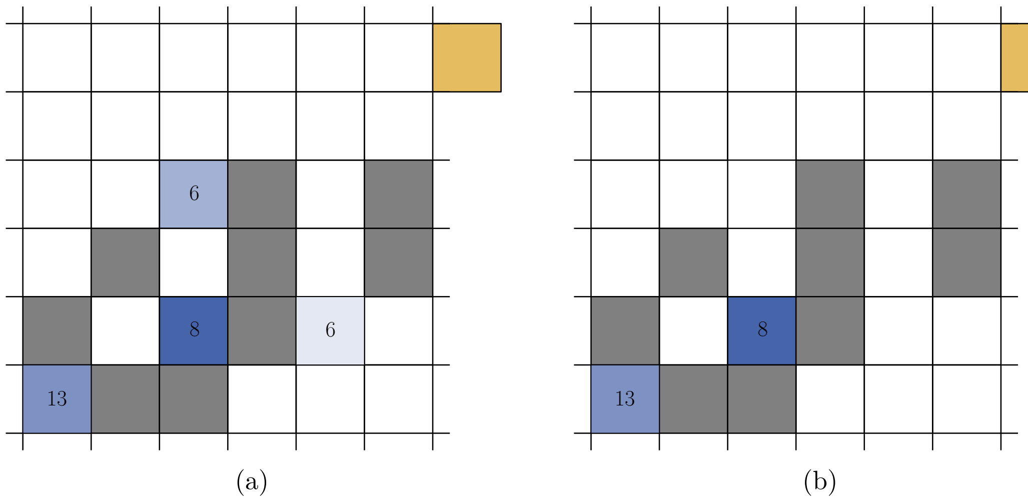


Figure 4.2.: The orange cell is the reference cell and the gray cells are obstacles. The numbers on the robots are equal to their distance obtained by a BFS. In (a), the dark blue robot cannot move out, because a light blue robot blocks its path. In (b), robots with a distance lower than 7 are already outside of the box. The order ensures that the light blue robots moves out of the initial bounding box first.

Proof. Let a breadth-first search calculate the obstacle-free distances between each robot's start position and the reference cell. Robots move sequentially, in ascending order according to the distance values. Hence, the robot with the lowest distance starts to move first. If there are multiple robots with equal distances, we choose one random robot (with that distance) to move. Once a robot is at the reference cell, it moves to its parking position. To keep it simple, the next robot starts moving once the previous robot has reached its parking position. In this way, not only is every robot guaranteed to move on a path without obstacles, but also on a path without other robots in its way. As a robot with distance k moves earlier than robots with distance $k+1$ and later than all robots with distance lower than k, a robot with distance k can always safely move to the reference cell by using the exact cells that have been explored in the BFS. □

Lemma 4.1 proves that we always find a schedule to move all robots to their parking positions. Still, this approach does not represent coordinated motion, since it does not involve parallel movements. For this reason, we send robots in waves. Every robot of a wave independently calculates a shortest path by performing a breadth-first search from its start position to its parking position. It should be noted that robots can move on the path via the reference cell that was originally determined by the other BFS. However, this simply ensures the existence of at least one path. This path is probably not ideal because it requires routing through the reference cell.

All robots of a wave start moving simultaneously according to their new path. If they reach their designated parking position, they wait and remain at their location. As multiple robots move at the same time, we need to prevent collisions. The robots' paths are calculated the moment they start moving, but if they strictly follow this path, two robots can possibly end up at the same position at the same time. We solve this problem as follows. If two robots move to the same cell at some point in time, we make one robot wait while the other robot continues on its path. By considering the remaining distance of those two robots, we determine that the robot with the longer remaining path is allowed to move first. In rare cases, a completely new path has to be calculated, for example, if a robot is at position (x, y) and another robot is at the adjacent position $(x, y+1)$ and the robot at position (x, y) aims to move north while the other robot aims to go south (see Figure 4.3 for an illustration). The robot with the longer remaining moves calculates a new path and treats the other robot as an obstacle. Therefore, it moves on a different path than initially calculated and bypasses the other robot.

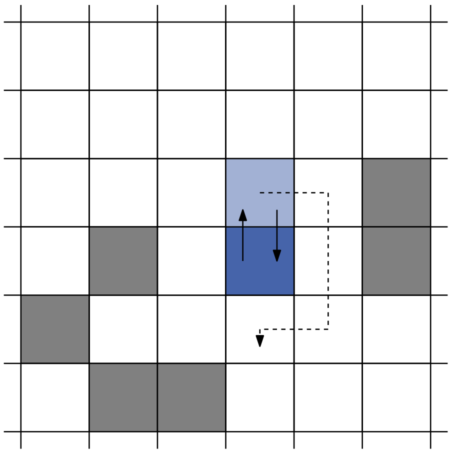


Figure 4.3.: The light blue robot calculates a new path while the dark blue robot waits.

4.1.5. Pushing Waves

While moving robots in waves rather than manipulating all robots simultaneously generates a feasible solution, it is not yet optimal. As described in the previous subsection, we need to wait until all robots of a wave have reached their desired parking position before we move the next wave of robots. In particular, this approach seems to be suboptimal for waves consisting of only one single robot. Hence, as a next step, we allow for "overlapping" waves.

We refer to the following procedure as *pushing waves*. We leave the first wave as it is while considering changes to the following waves. Let the second wave start at time t and end at time $t + l$. For every time step k between $t - 1$ and $t + l - 1$, we replace all robot movement of robots involved in the second wave of time step k with the robot motion of time step $k + 1$. We can safely apply this change to all following waves at time $k \geq t + l$ as well. If the new resulting schedule is valid, we lower the makespan by 1. We repeat the procedure until the schedule is not valid or we reach $t = 0$. As we overlap two waves, there is a chance for the schedule to be invalid. If this is the case, we revert the respective step, so that the schedule becomes valid again. We then repeat the whole procedure for the second and third wave and so on, until all waves are pushed. We do not change the movement of robots, only the time of when a robot moves. Thus, pushing waves lowers the makespan, but it is no improvement for the total distance objective. A method available to validate a given schedule can be found in Appendix A.

Further remarks. We improve the runtime by performing a binary search rather than simply pushing waves one by one. We determine the gap ℓ between two waves and then push a wave ℓ time steps to the front. If the resulting schedule is not valid, we revert the respective step and push the wave $\ell/2$ time steps to the front. This process is repeated until $\ell = 1$.

4.2. Moving Robots back into the Initial Bounding Box

All robots are located at their assigned parking positions and as a next step, we aim to move them back into the initial bounding box. This procedure resembles the procedure of moving robots out of the initial bounding box. In contrary to the setting in Section 4.1, every robot already has a designated target position.

4.2.1. Generating an Order

We aim to ensure that each robot can move to its target position without any complications. To accomplish this, we choose the cell below the top right corner of the border as the reference cell. It should be noted that the reference is reachable from all parking positions as per construction. The reference cell is the source node of a breadth-first search to calculate distances. While the fundamental idea remains the same, the target nodes of the BFS are the robots' target positions. When a cell needs to be passed although a target position is surrounded by many obstacles, this cell must stay empty to let the destined robots pass. A visualization of this scenario can be found in Figure 4.4.

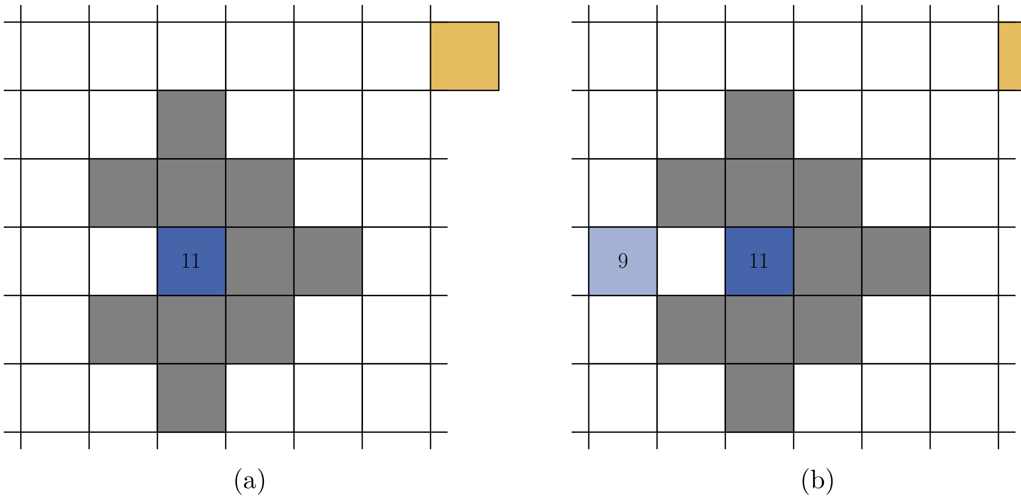


Figure 4.4.: The orange cell is the reference cell and the gray cells are obstacles. The numbers on the robots are equal to their distance obtained by a BFS. In (a), the dark blue has to be at its target position before in (b), the light blue robot reaches its target position.

4.2.2. Moving Robots to their Target Position

The argumentation of Lemma 4.1 can be applied in a similar manner to ensure that we obtain a schedule with all robots at their parking position in the start configuration and at their target position in the target configuration. Therefore, we may combine these two sequential schedules to obtain a schedule that moves all robots from their start position to their target position. By pushing waves, we further lower the makespan. We call the resulting schedule our *initial solution*.

4.3. Improving the Schedule

Given any schedule that solves Coordinated Motion Planning in a Grid, we aim to improve the schedule to further minimize the makespan and the total distance. The initial solution we obtained in the previous sections can still be optimized. We reduce the impact of unnecessary robot movements by performing a more complex breadth-first search in the three-dimensional space while maintaining feasibility.

4.3.1. Three-Dimensional Breadth-First Search

Given a schedule, we are able to extract each configuration $C_s, \ldots, C_t$. A single configuration C can be considered as a two-dimensional array, in which each entry (x, y) holds information about the entity currently located at position (x, y) in C. There may be free space, an obstacle, or a robot with label i. We add a new dimension to the array. Let the start configuration be the *bottom configuration*. We "place" the following configuration, which results by performing all moves of the first time step, onto the next layer. This procedure continues until the target configuration is the *top configuration*. The result is a three-dimensional array with a new dimension, which we call the *time dimension*.

Particularly in our solution, robots perform moves that do not seem to be necessary. We plan to "straighten" the existing paths of the robots. In the three-dimensional array, we perform a breadth-first search for a robot from its start position to its target position while respecting the paths of other robots. However, during the BFS, all paths of other robots are fixed (see 4.5 for a visualization). When the target position is reached, we trace back the nodes explored by the BFS and extract a new path for the robot. This path contains the same or fewer moves than before. Therefore, we potentially lower the total distance. As a robot may reach its position at an earlier time step than in the initial path, this method possibly lowers the makespan as well.

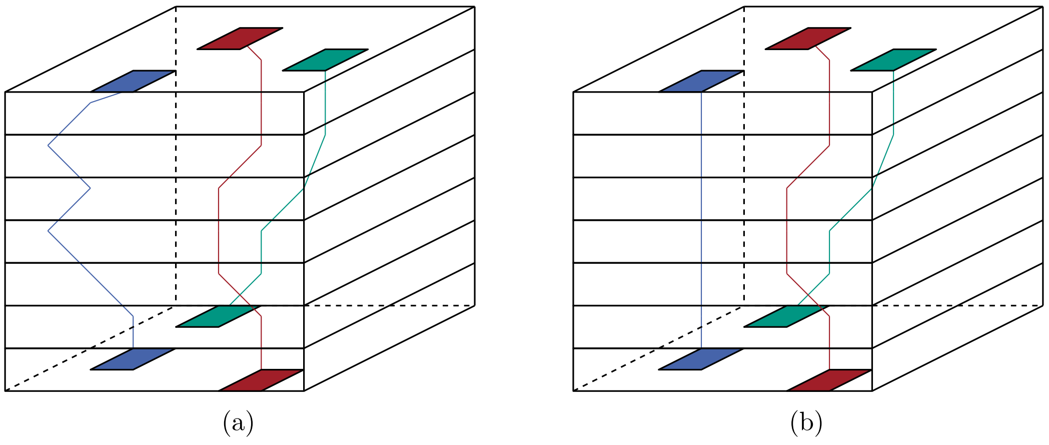


Figure 4.5.: A sketch of the general idea. Seven configurations are on top of each other. (a) Original routes of three robots. (b) The possible result of a BFS on the blue robot's path in (a). The other paths are fixed during the BFS. The blue path is "straightened" and contains fewer individual moves.

We shed light on the underlying mechanism of the three-dimensional breadth-first search and describe how to implement it. We define a graph $G = (V, E)$. A vertex is of the form (x, y, t), where x is the x-coordinate, y is the y-coordinate and t is the time step. The vertex contains information on whether there is a specific robot, an obstacle, or an empty cell at position (x, y) at time t. If a robot is located at (x, y, t), there are at most five possibilities for legal robot motions. The robot may stay at the position or move towards one of the four adjacent cells. In any case, its next position is located at the next time step $t + 1$, whereas x and y remain the same or either one of the coordinates is incremented or decremented. By ensuring that $(x, y, t + 1)$ (the same position at the next time step) is put first into the queue of the BFS, we urge the robot performs as few moves as possible. The resulting graph G is directed, acyclic and each node has out-degree 5. Edges represent possible motions towards the next time dimension. An edge is explored in the BFS, only if it does reflect a legal move, as we need to avoid any sort of collision. Specifically, the follow collision is not allowed to occur.

The procedure to check for collision works as follows. Let i be the label of the robot that performs a BFS and let (x, y, t) be the node that is currently popped from the queue. The five neighbors of this node are $(x, y, t + 1)$, $(x + 1, y, t + 1)$, $(x - 1, y, t + 1)$, $(x, y + 1, t + 1)$ and $(x, y - 1, t + 1)$. To check for obstacles and different robots, we simply check if the explored node contains an obstacle or a robot with label $j \neq i$. If the neighbor is not blocked, we further check for follow collisions by inspecting two cases (see Figure 4.6 for a visualization).

Case 1. The node $(x, y, t+1)$ is covered by a robot with label $j \neq i$. The robot with label i at (x, y, t) may only explore a node in the next time step if the directions of i and j are equal. Therefore, only one move is possible.

Case 2. We explore $(x+1, y, t+1)$ and $(x+1, y, t)$ is covered by a robot with label $j \neq i$. The robot with label i at (x, y, t) may only explore $(x+1, y, t+1)$ if the directions of i and j are equal. Analogously, this applies to all other neighbors too, except for $(x, y, t+1)$, as the robot i is covering (x, y, t).

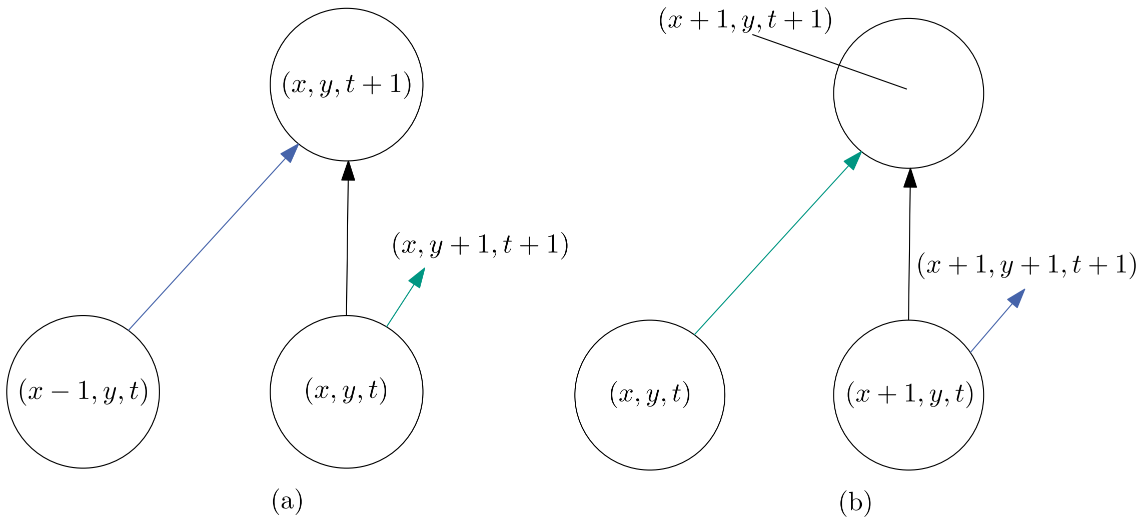


Figure 4.6.: Possible situation in a BFS. The node (x, y, t) currently explores the node denoted by the green edge. The blue edge represents the fixed path of a different robot. Only the relevant nodes and edges are depicted. (a) Case 1: The node $(x, y+1, t+1)$ (north) is not added to the queue as the other robot moves east. (b) Case 2: The node $(x+1, y, t+1)$ (east)is not added to the queue as the other robot moves north.

We perform the three-dimensional search for one robot at a time. The source node is the robot's start position $(x_1, y_1, 0)$ before the first time step. The target node is the robot's target position (x_2, y_2, t) at some time $t \leq M$. If the target position is explored in the BFS at any time $k < t$, we check if the robot is allowed to remain at the position. This is the case, if for all l $(k < l \leq t)$ no robot covers the position (x_2, y_2, l). We then abort the BFS early and the robot reaches its target position at an earlier point in time. Once the search is finished, we update our schedule and the graph according to the improved path. We continue to straighten the path of a different robot. The path generated by the previous breadth-first search is fixed. This process continues until a BFS is performed for all robots.

The goal is to run a three-dimensional search for each robot. We call this an *iteration*. We aim to do as many iterations as possible until there are no further improvements to the schedule. It should be noted that, after every iteration, optimized paths may be generated, as robots that perform the BFS at a later time during an iteration might have changed their pathing. The makespan and the total distance cannot increase as, in the worst case, our search would return the initial path. However, we do not have a fixed limit on the number of iterations. Therefore, we do not know in advance how many iterations will be performed.

We need to specify an order for the robots during one iteration. Instead of selecting the order randomly, we propose the following heuristics.

Option A. We sort robots in ascending or descending order according to their labels.

Option B. We sort robots by the time they reach their target position. AS the makespan is capped by robots that reach their target position at the last time step, we perform the BFS for these robots first.

Option C. We sort robots in descending order according to the obstacle-free distance between the reference cell and the target position obtained by a BFS. This option particularly takes into account how we create initial solutions. As described in previous sections, certain cells must stay empty to let robots pass. Therefore, if such a cell is the target position of a robot, this specific robot cannot reach its target position until the corresponding robots have passed the cell.

5. Evaluation and Discussion

In this chapter, we evaluate and discuss the proposed solution by examining the quality of the different heuristics for an initial solution and investigating how different initial solutions impact our three-dimensional search. The challenge provides 203 diverse instances with the number of robots ranging from 10 to 9000. The initial bounding box of the instances is a square in most cases and about half of the instances have no obstacles. Furthermore, the instances differ in different parameters like density, clusters of robots, and unblocked areas. Fekete et al. [FKKM21] give more insight into how they created the instances for the “CG:Shop 2021”-Challenge.

5.1. Different Arrangements of Parking Positions

We have proposed two possibilities for arrangements of parking position (see Figure 4.1). In both options, we surround the initial bounding box equally with parking positions, but Option A leaves less spacing between the parking positions compared to Option B, where no two parking positions are adjacent. The resulting makespans and total distances for some exemplary instances are depicted in Table 5.1. In this table, the makespan is the overall time all robots need to reach their parking position (not their target position). The same weight function for the assignment is used.

Table 5.1.: Impact of different arrangements of parking positions on the initial solution. Option A: Less spacing, Option B: No adjacent parking positions

		Option A		Option B	
Instance	n	Makepsan	Total Distance	Makespan	Total Distance
`small_free_001`	40	109	244	126	308
`small_015`	207	601	2 261	710	2 794
`medium_016`	1180	4 300	31 994	4 937	39 506
`large_free_001`	1688	6 652	55 484	7 731	67 290
`medium_free_019`	2250	5 805	77 702	6 777	101 290
`sun_00006`	3796	14 435	161 303	16 842	205 173
`london_night_00009`	6000	18 670	349 744	22 321	454 714
`clouds_00009`	7229	21 812	455 332	25 144	592 362
`large_free_009`	9000	23 153	594 887	27 489	791 755

We find that Option A has lower makespans and lower total distances overall. This is expected since the parking positions are located closer to the initial bounding box. Pursuing a three-dimensional BFS on the initial solution, we also consider Option B, where robots have more empty cells they can use. However, when performing the BFS, the arrangements of parking positions do not impact the final schedule in a significant way. The values for the objectives are mostly the same and no arrangement is always the superior one. Thus, we use both options and choose the arrangement that leads to a better result in the end.

Other teams in this competition move robots into intermediate positions as well. In their solutions, they introduce even more fine-grained arrangements of parking positions. Rather than uniformly surrounding the initial bounding box, they choose their parking positions differently. For example, team Shadoks [CdFG+21] introduces four different *storage networks* (arrangement of parking positions) and depending on the instance, one solution outperforms the other one. However, no storage network seems to be the best for all instances.

5.2. Different Assignments of Robots to Parking Positions

We have constructed a weighted, complete bipartite graph that describes our assignment problem. In this graph, we compute a minimum-cost maximum-matching. We have proposed three different weight functions which assign weights to the edges of this graph: the distance between a robot's start and parking position (start), the distance between a robot's target and parking position (target), or the sum of these two distances (start and target). Table 5.2 and Table 5.3 show eight selected instances and how different weight functions affect the quality of an initial solution. We choose these instances to be as representative as possible. They vary in density, number of robots, and number of obstacles.

Table 5.2.: Makespan with different weight functions

Instance	n	Start	Target	Start and Target
`small_free_001`	40	400	393	**372**
`small_015`	207	**2 399**	2 484	2484
`medium_016`	1 180	14 769	14 754	**13 827**
`large_free_001`	1 688	**26 437**	29 285	27 639
`medium_free_019`	2 250	18 761	18 638	**17 481**
`london_night_00009`	6 000	65 394	66 062	**63 844**
`clouds_00009`	7 229	71 620	70 117	**68 721**
`large_free_009`	9 000	77 264	77 820	**73 946**

Table 5.3.: Total distance with different weight functions

Instance	n	Start	Target	Start and Target
`small_free_001`	40	908	950	**858**
`small_015`	207	9 134	9 252	**8 560**
`medium_016`	1 180	116 429	116 937	**104 799**
`large_free_001`	1 688	227 719	233 981	**207 853**
`medium_free_019`	2 250	262 361	260 537	**240 497**
`london_night_00009`	6 000	1 243 908	1 248 464	**1 137 034**
`clouds_00009`	7 229	1 588 466	1 563 164	**1 431 344**
`large_free_009`	9 000	2 066 953	2 070 097	**1 890 321**

We observe that the third option, which takes into account both the start and the target position, results in a shorter makespan and total distance for the initial solution in most cases. Therefore, it appears to be the best choice to stick to this weight function.

5.3. Pushing Waves

In Table 5.4, we depict the effect of pushing waves on our initial solution. We compare the makespan between not pushing waves at all and pushing both sets of waves, meaning the waves of robots moving out of the initial bounding box as well as the waves of robots moving back to their target position.

Table 5.4.: The effect of pushing waves

Instance	No Pushing	All Waves Pushed	Improvement (%)
`small_free_001`	585	389	33.5
`small_015`	2 474	1 161	53.1
`medium_016`	14 573	7 247	50.3
`medium_free_019`	19 337	10 039	48.1
`large_free_001`	26 086	10 221	60.8
`london_night_00009`	64 623	36 782	43.1
`clouds_00009`	72 411	42 109	41.8
`large_free_009`	79 336	45 367	42.8

We find that the makespan is considerably reduced. Even for large instances, we obtain a makespan about half the size of a solution without pushing waves. This is a big improvement to our initial schedule. The total distance traveled by all robots is not reduced, as pushing waves does not change the paths of robots.

5.4. Three-Dimensional Breadth-First Search

Given our initial solution, we evaluate the impact of improving the schedule with our three-dimensional BFS. The makespan of our initial solution for the largest instances in the competition amounted to about 45 000. Without pushing waves, the makespan would be twice as large. Thus, pushing waves is crucial to reduce the search space. It needs to be remembered that the makespan is equal to the size of the time dimension. For a grid of size 200×200, this already leads to a search space on the order of 10^9, which is very expensive. A BFS for a single robot would take several minutes. As a consequence, for instances with thousands of robots, an iteration would take several weeks. Therefore, we rely on smaller instances and generalize our results if possible. We further provide numbers for the first few iterations of an instance with 3000 robots. It should be noted that there are techniques in route planning that might speed up our search.

First, we continue evaluating the impact of different weight functions on the three-dimensional search. Table 5.5 and Table 5.6 show the resulting makespans and total distances for eight exemplary instances. Surprisingly, we do not find any significant differences. Even though the initial solution is best when the weight function considers the start and target position of a robot, the quality of the resulting schedules after the BFS is very similar. While *start* is best for `small_free_001`, it is outperformed by *start and target* for `microbes_00000`. Therefore, we try all three options and for each instance, we choose the one performing best.

Second, we examine how the three-dimensional search improves the initial solutions. In Figure 5.1a, we observe how the makespan improves in every iteration of our BFS. The

Table 5.5.: Makespan after BFS with different weight functions

Instance	n	Start	Target	Start and Target
small_001	30	23	26	**22**
small_free_001	40	**26**	42	44
galaxy_cluster2_00000	80	37	**33**	34
socg2021_108	108	54	54	**53**
microbes_00000	110	189	180	**121**
redblue_00000	113	247	**181**	219
universe_bgradiation_00000	139	**195**	289	201
clouds_00000	160	239	254	**200**

Table 5.6.: Total distance after BFS with different weight functions

Instance	n	Start	Target	Start and Target
small_001	30	328	338	**302**
small_free_001	40	**414**	420	434
galaxy_cluster2	80	1 350	1 334	**1 310**
socg2021_108	108	2 442	2 414	**2 412**
microbes_00000	110	**3 005**	3 253	3 011
redblue_00000	113	**2 262**	2 400	2 296
universe_bgradiation_00000	139	2 394	2 456	**2 360**
clouds_00000	160	3 589	3 541	**3 471**

first iteration is usually a very large improvement. In most cases, we achieve a makespan of only half the size compared to the makespan of the initial solution. The next iterations still improve the makespan, but at some point, the makespan converges until there are no further improvements. It takes about 10-20 iterations until the three-dimensional search is finished. For small instances, the resulting makespan is about 10% – 20% of the initial solution.

The improvements of the total distance are depicted in Figure 5.1b. The progress is very similar to our observations for the makespan. The very first iteration is respectively a large improvement which approximately halves the total distance. The following iterations also lower the total distance, however, they do so at a slower pace. At some point, the distance stays on a constant level until the search comes to an end.

Third, we consider a larger instance, `universe_bgradiation_00006`, with 3000 robots. For reasons of time, we are only able to provide the first three iterations of the three-dimensional search. Table 5.7 shows the resulting values for makespan and total distance. Considering the makespan, the first iteration is not as impactful as we would expect. However, the improvements of the second and third iteration are larger than of the first iteration. Following this trend, the makespan will probably reach a lower level with more iterations. Considering the total distance, the first iteration represents a substantial improvement for the schedule and the following iterations still considerably lower the total distance.

The quality of our final solution depends on the order of the robots during an iteration of the BFS. Table 5.8 shows four different heuristics: ascending (asc) and descending (desc) order according to the labels, sort robots by the time they perform their last move (last robots), or sort robots by descending distance in relation to a robot's target position to the reference cell (highest distance). We find out that for instances like `small_001`, which are small and not densely packed, the order does not significantly change the makespan.

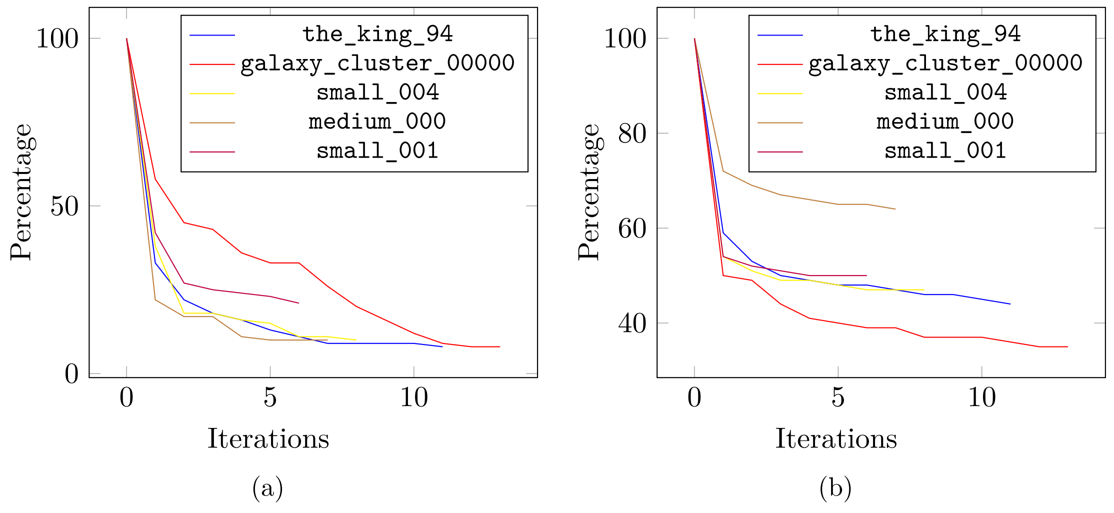


Figure 5.1.: (a) Improvements of Makespan (b) Improvements of Total Distance

Table 5.7.: Three iterations for `universe_bgradiation_00006`

Category	Initial	Iteration 1	Iteration 2	Iteration 3
Makespan	20 833	20 373	19 315	18 188
Total Distance	524 506	389 854	361 412	347 860

However, for most instances, the last heuristic seems to achieve the best results. These findings encourage the search for more good heuristics.

Table 5.8.: Makespan with different orders during a BFS

Instance	Asc.	Desc.	Last Robots	Highest Distance
`small_001`	23	22	**21**	22
`small_free_001`	85	76	68	**26**
`galaxy_cluster2`	35	**34**	**34**	**34**
`socg2021_108`	**53**	**53**	253	**53**
`microbes_00000`	364	368	384	**121**
`redblue_00000`	368	338	325	**219**
`universe_bgradiation_00000`	333	326	343	**201**
`clouds_00000`	521	450	450	**200**

We present some of the results we achieved, refer to Appendix C for an overview of all results. Table 5.9 compares the achieved values for makespan and total distance to the respective lower bound for eight instances. With regard to smaller instances, our solution performs well for minimizing the makespan. For `galaxy_cluster_00000` and two other instances, we even meet the trivial lower bound, meaning that we find an optimal solution concerning the makespan objective. However, for other small instances such as `small_free_011`, we obtain a makespan that is rather distant to the lower bound. In general, we find that if the instance is very dense, our approach seems not to work as intended. A possible explanation is the difficulty to find paths using the BFS while many of the positions are blocked. It should also be noted that the lower bound becomes a worse comparison as the density grows. Furthermore, our method does not scale well with the number of robots when considering the makespan. For example, `microbes_00004` achieves a makespan of 3 502, while team Gitastrophe [LSJZZ21] finds a schedule with makespan 126 for this instance.

Concerning the total distance traveled by all robots, our solution performs better, even for larger instances. Although we do not meet the trivial lower bound on any instance in this category, we are very close to an optimal total distance for small instances. Our three-dimensional search ensures that a robot tries to stay at its position if possible. Hence, most unnecessary moves in the initial solution are eradicated. For `microbes_00004`, we achieve a total distance that is about twice as large as the trivial lower bound.

Table 5.9.: Results in comparison to the lower bounds in both categories

		Makespan		Total Distance	
Instance	n	Lower Bound	Result	Lower Bound	Result
`small_free_000`	30	13	14	187	211
`small_002`	33	15	22	229	329
`small_004`	61	35	40	1116	1304
`buffalo_000`	63	54	59	1547	1801
`galaxy_cluster_00000`	80	27	27	845	965
`the_king_94`	94	31	36	1827	2017
`small_free_009`	160	34	213	2142	3218
`small_free_011`	200	31	365	2486	4214
`microbes_00004`	1250	91	3502	39 125	77 919

We briefly discuss how other teams approach the coordinated motion planning problem and why their results might have been superior to our results. The strategy of other teams is similar to ours: they find parking positions and compute a minimum-cost maximum-matching for the assignment. Afterwards, they perform a search in the three-dimensional space, but the details of this search differ.

Team Gitastrophe [LSJZZ21] wins the total distance category. To create an initial solution, they provide similar heuristics to ours, but introduce even more variations. Considering the search, they approach this problem with a k-opt technique. They choose k robots to improve their paths while the paths of other robots are fixed. The k robots are improved one-by-one. Therefore, it is crucial to choose a good set of k robots and determine an order for the robots within the set.

Team Shadoks [CdFG$^+$21] wins the makespan category. They pursue two different approaches. The first idea is very similar to ours, as they perform an A^* search in the three-dimensional space. They are able to tune this search, for example by using the sum of random weights for each cell the robot passes through to break ties. The second idea is a *Conflict Optimizer*. This strategy does not guarantee finding a solution, but it does result in a very good schedule in most cases. They modify their search so that it is allowed for a robot to go over another robot's path, see their paper for more details.

6. Conclusion

This thesis considered a coordinated motion planning problem as part of the “CG:SHOP 2021”-Challenge. Within this specific setting, robots are square-shaped and located in an unbounded grid. The goal was to compute a feasible schedule while separately following two objectives: minimize the makespan and minimize the total distance.

We adapted a proof by Demaine et al. [DFK$^+$19] to show that the related decision problem for the makespan is NP-complete in our setting, even if there are no obstacles in the environment. This encouraged us to search for a heuristic that minimizes the objectives as much as possible. Our solution consisted of two steps: finding an initial solution and improving the solution afterwards.

We showed that, if every robot can be routed to the border on a path that is not blocked by obstacles, we are guaranteed to find a feasible solution. We proved this by explaining how to obtain a sequential schedule based on calculating distances for each robot and establishing an order according to these distances. Therefore, all robots can be routed out of the initial bounding box, and with the same argumentation, we can move all robots back to their target position without collisions. By proposing different heuristics and pushing waves, we improved our initial solution, however, there still remains potential for improvement.

The three-dimensional breadth-first search had the most impact on improving our schedule. As we obtained a feasible initial solution, we simply removed one path of a robot while all other paths remained fixed. We found a new path for this robot, which was possibly shorter or involved fewer moves. Our evaluation showed that the final BFS worked well for smaller instances and for the objective that aims at minimizing the total distance, while it encountered difficulties with the makespan objective. Finding a good order for the robots during an iteration is relevant to achieve the best results. We provided a heuristic that operated well for our specific initial solution. It remains open to see if there are even further heuristics that considerably improve the three-dimensional search.

In the competition, we ranked 8th out of 17 in the total distance category and 11th out of 17 in the makespan category. While the strategy of other teams resembles ours, the three-dimensional search is executed differently. For example, Team Unist [YV21] added the idea of simulated annealing. Team Shadoks [CdFG$^+$21] provided a method that did not provably find a feasible solution, however, if the method found a legal schedule, it often beat the results of other teams in the makespan category. Whereas we focused on finding a decent *feasible* solution, an open question is to explore other methods that achieve good results without provably finding a schedule for all instances. We kept the paths of robots fixed while improving the path of a single robot. It is an interesting approach to investigate if this robot can negotiate with other robots to improve its path. Furthermore, recalculating paths for a set of k robots could also be an option. Team Gitastrophe [LSJZZ21] provided some sampling strategies for choosing k robots.

Even though the schedules of other teams may be overall superior, we still provide feasible solutions for all instances in addition to decent results for the total distance objective. Thus, this thesis contributes to the ongoing debate of possible solutions to the coordinated motion planning problem.

Bibliography

[CdFG+21] Loïc Crombez, Guilherme D. da Fonseca, Yan Gerard, Aldo Gonzalez-Lorenzo, Pascal Lafourcade, and Luc Libralesso. Shadoks Approach to Low-Makespan Coordinated Motion Planning. In *37th International Symposium on Computational Geometry (SoCG 2021)*, volume 189 of *LIPIcs*, 2021. To appear.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[DFK+19] Erik D. Demaine, Sándor P. Fekete, Phillip Keldenich, Henk Meijer, and Christian Scheffer. Coordinated Motion Planning: Reconfiguring a Swarm of Labeled Robots with Bounded Stretch. *SIAM Journal on Computing*, 48(6):1727–1762, 2019.

[FB02] Gary William Flake and Eric B. Baum. Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1):895–911, 2002.

[FKKM21] Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Joseph S. B. Mitchell. Computing Coordinated Motion Plans for Robot Swarms: The CG:SHOP Challenge 2021. In *37th International Symposium on Computational Geometry (SoCG 2021)*, volume 189 of *LIPIcs*, 2021. To appear.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979.

[Gol78] E. Mark Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.

[HCZ10] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A Novel Transition Based Encoding Scheme for Planning as Satisfiability. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, page 89–94. AAAI Press, 2010.

[HNR68] 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.

[HSS84] John E. Hopcroft, Jacob T. Schwartz, and Micha Sharir. On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE- Hardness of the "Warehouseman's Problem". *The International Journal of Robotics Research*, 3(4):76–88, 1984.

[HW86] John E. Hopcroft and Gordon T. Wilfong. Reducing Multiple Object Motion Planning to Graph Searching. *SIAM Journal on Computing*, 15(3):768–785, 1986.

[Kuh55] Harold W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[Loc21] William Lochet. A Polynomial Time Algorithm for the k-Disjoint Shortest Paths Problem. In *ACM-SIAM Symposium on Discrete Algorithms (SODA21)*, 2021.

[LSJZZ21] Paul Liu, Jack Spalding-Jamieson, Brandon Zhang, and Da Wei Zheng. Coordinated Motion Planning Through Randomized k-Opt. In *37th International Symposium on Computational Geometry (SoCG 2021)*, volume 189 of *LIPIcs*, 2021. To appear.

[MCF] Minimum-Cost Flow - Successive Shortest Path Algorithm. `https://cp-algorithms.com/graph/min_cost_flow.html`. Accessed: 2021-04-12.

[MKA$^+$17] Hang Ma, Sven Koenig, Nora Ayanian, Liron Cohen, Wolfgang Hoenig, T. K. Satish Kumar, Tansel Uras, Hong Xu, Craig Tovey, and Guni Sharon. Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios, 2017.

[RW90] Daniel Ratner and Manfred Warmuth. The (n^2-1)-Puzzle and Related Relocation Problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.

[SH14] Kiril Solovey and Dan Halperin. k-Color Multi-Robot Motion Planning. *The International Journal of Robotics Research*, 33(1):82–97, 2014.

[SS83] Jacob T. Schwartz and Micha Sharir. On the Piano Movers' Problem: III. Coordinating the Motion of Several Independent Bodies: The Special Case of Circular Bodies Moving Amidst Polygonal Barriers. *The International Journal of Robotics Research*, 2(3):46–75, 1983.

[SSF$^+$19] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Barták. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *12th International Symposium on Combinatorial Search*, 2019.

[WC15] Glenn Wagner and Howie Choset. Subdimensional Expansion for Multirobot Path Planning. *Artificial Intelligence*, 219:1–24, 2015.

[WDM08] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, 29(1):9–20, 2008.

[YV21] H. Yang and A. Vigneron. A Simulated Annealing Approach to Coordinated Motion Planninng. In *37th International Symposium on Computational Geometry (SoCG 2021)*, volume 189 of *LIPIcs*, 2021. To appear.

Appendix

A. Validating a Schedule

In Algorithm 6.1, we show how to validate a given schedule. The input is a schedule S consisting of steps $S_1, \dots, S_M$, where S_i encodes the steps of robots that move at time $t = i$.

Algorithm 6.1: SolutionValidator

```
Input: Schedule $S = (S_1, \dots, S_M)$
Data: Configurations $C_{curr}$, $C_{prev}$
Output: True, if schedule $S$ is valid. False, otherwise.
1  $C_{curr} \leftarrow C_s$ // Set $C_{curr}$ as start configuration
2  $C_{prev} \leftarrow C_s$
3  for $i \leftarrow 1$ to $M$ do
4      $C_{curr}$.UPDATE($S_i$) // Perform all moves of step $S_i$
5      if robot and obstacle at the same position in $C_{curr}$ then
6          return false
7      if two robots at the same position in $C_{curr}$ then
8          return false
9      forall $j \in S_i$ do
           // Check for follow collision
10         $p = (x, y) \leftarrow C_{curr}$.GETCOORDINATESOFROBOT($j$)
11         $k \leftarrow C_{prev}$.GETROBOTAT($p$)
12         if $k$ is robot and directions of robots $j$ and $k$ not equal then
13             return false // Follow collision occurs
14     $C_{prev} \leftarrow C_{curr}$
15 if at least one robot is not at its target position in $C_{curr}$ then
16     return false
17 return true
```

The algorithm works as follows. We set the current configuration C_{curr} and the previous configuration C_{prev} as the start configuration. In a loop, we update C_{curr} according to robot motion in step S_i. We check for collisions in C_{curr}. If a robot and an obstacle share the same position, the schedule is not valid. If two robots are located at the same position, the schedule is not valid. To check for follow collisions, we need the auxiliary configuration C_{prev} holding the configuration of the previous time step. If a robot $j \in S_i$ moves to a position p, we check if this position p has been occupied by a different robot k in the previous configuration. If that is the case, we check if both robots move towards the same direction. Otherwise, a follow collision occurs. We repeat this procedure for all steps in the solution. The last check ensures that all robots are positioned at their target positions.

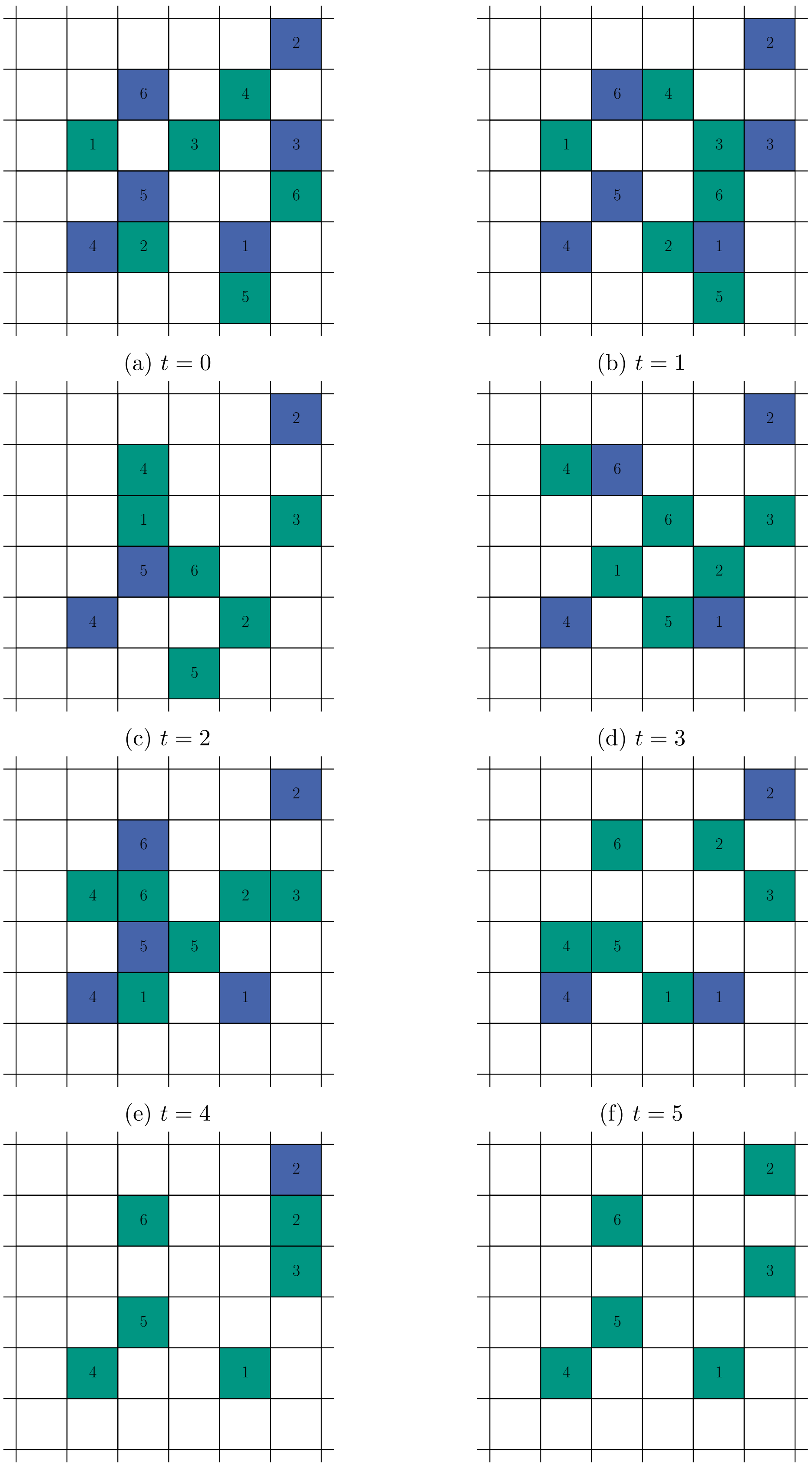
B. Complete Schedule

We show a full schedule we obtained by applying our solution in Figure B.1. The depicted instance ("Sprinkle") is taken from the problem description of the challenge[1]. There are six robots and the resulting schedule is optimal concerning both the makespan (7) and total distance (29).

C. Results for all Instances

The following Table C.1 contains the results for all 203 instances that we achieved. The instances are sorted according to the number of robots. Let n be the number of robots. It needs to be remembered that for large instances (approximately $n > 1250$), we did not perform a three-dimensional search. Medium instances ($250 \leq n \leq 1250$) performed a few iterations iterations, but we aborted the BFS early. Small instances ($n < 250$) are optimized.

[1]`https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2021`



(a) $t = 0$ (b) $t = 1$

(c) $t = 2$ (d) $t = 3$

(e) $t = 4$ (f) $t = 5$

(g) $t = 6$ (h) $t = 7$

Figure B.1.: A full schedule from start to target configuration. Robots are green and target positions are blue.

Table C.1.: Results for all 203 instances

		Makespan		Total Distance	
Instance	n	Lower Bound	Result	Lower Bound	Result
small_000	10	13	21	68	106
small_001	30	17	20	246	310
small_free_000	30	13	14	187	211
small_002	33	15	22	229	329
small_free_001	40	13	21	266	368
small_003	46	19	40	379	621
small_free_002	50	15	22	356	508
small_004	61	35	40	1 116	1 266
buffalo_000	63	54	59	1 547	1 759
small_005	63	18	40	577	1 037
small_free_003	70	14	50	469	937
small_006	71	**34**	**34**	1 013	1 103
galaxy2_cluster_00000	80	31	33	1 186	1 284
galaxy_cluster_00000	80	**27**	**27**	845	957
small_free_004	80	**30**	**30**	1 088	1 144
small_free_005	80	20	25	774	944
small_007	82	36	49	1 519	1 961
small_008	83	25	100	923	1 493
medium_000	90	57	62	2 682	2 962
small_free_006	90	15	117	618	1 528
small_free_007	90	14	82	559	1 361
the_king_94	94	31	36	1 827	2 017
socg2021_108	108	46	52	2 086	2 374
election_109	109	33	39	1 816	2 216
microbes_00000	110	38	146	1 927	2 985
redblue_00000	113	28	146	1 524	2 242
buffalo_free_000	125	34	36	1 871	2 095
universe_bg_00000	139	33	128	1 480	2 182
london_night_00000	140	30	171	2 126	3 610
sun_00000	143	27	120	1 484	2 716
medium_001	145	51	61	3 553	4 065
algae_00000	160	29	161	2 291	3 439
clouds_00000	160	27	177	2 151	3 481
small_free_008	160	31	232	2 286	3 540
small_free_009	160	34	213	2 142	3 212
medium_002	162	57	280	4 850	6 094
galaxy_cluster_00001	172	37	234	2 959	5 231
small_009	173	31	302	2 614	4 530
buffalo_001	174	41	416	3 322	6 038
small_010	175	31	177	2 213	4 073
medium_free_000	180	46	50	3 776	4 034
medium_free_001	180	33	136	2 755	3 347
medium_free_002	180	49	77	3 228	3 628
small_011	183	37	435	3 403	6 343
small_012	186	35	181	2 884	5 740
small_013	186	33	380	2 578	5 070

		Makespan		Total Distance	
Instance	n	Lower Bound	Result	Lower Bound	Result
galaxy_cluster2_00001	200	28	475	2 141	4 415
galaxy_cluster_00002	200	31	384	2 139	4 857
microbes_00001	200	28	421	2 227	4 569
small_014	200	39	324	3 236	6 702
small_free_010	200	32	275	2 529	4 591
small_free_011	200	31	333	2 486	4 214
small_015	207	40	314	4 606	7 644
london_night_00001	240	35	518	3 277	7 533
small_free_012	240	31	405	2 672	5 798
small_free_013	240	35	461	3 401	6 747
buffalo_free_001	250	31	652	3 419	6 401
galaxy_cluster2_00002	251	30	527	3 160	7 934
medium_003	256	100	377	17 624	20 822
small_016	276	36	532	3 905	9 261
algae_00001	278	32	503	3 943	8 735
small_free_014	280	30	593	3 830	9 194
small_free_015	280	32	631	3 893	8 555
medium_004	306	85	981	12 133	15 451
medium_free_003	320	64	291	7 873	8 903
small_free_016	320	36	605	4 789	11 073
small_free_017	320	32	624	4 537	10 689
small_017	322	34	610	4 685	11 943
small_018	324	34	585	4 985	11 973
small_019	329	34	753	4 573	11 381
small_free_018	360	32	700	4 709	13 155
small_free_019	360	32	678	4 714	12 480
medium_005	407	58	1 073	11 818	22 160
medium_free_004	480	59	925	10 506	17 126
medium_free_005	500	85	502	16 562	18 172
medium_006	502	72	1 887	15 651	29 205
redblue_00001	531	91	1 248	20 036	30 612
medium_free_006	540	52	1 354	11 134	21 404
sun_00001	571	91	1 788	25 030	36 758
medium_007	584	54	1 887	14 866	31 926
galaxy_cluster2_00003	625	82	2 470	18 487	36 859
medium_free_007	630	52	2 146	12 718	34 050
galaxy_cluster_00003	669	93	2 923	25 741	43 465
redblue_00002	669	78	2 319	24 097	40 661
galaxy_cluster2_00004	679	83	2 714	21 087	39 947
medium_008	693	52	1 900	13 769	40 245
buffalo_002	702	95	3 103	27 426	50 086
medium_009	706	93	1 998	31 459	52 525
algae_00002	720	88	3 908	27 840	52 540
buffalo_free_002	720	106	3 705	27 718	48 620
medium_010	726	75	3 711	26 699	52 783
medium_011	727	59	2 718	17 320	41 968
universe_bg_00001	740	86	4 374	26 443	53 435

		Makespan		Total Distance	
Instance	n	Lower Bound	Result	Lower Bound	Result
galaxy_cluster_00004	750	89	3 354	25 078	53 038
galaxy_cluster_00005	750	86	3 490	26 581	53 637
medium_free_008	750	79	3 401	28 792	50 382
redblue_00003	750	92	3 239	24 546	49 600
buffalo_003	776	68	3 191	20 056	48 338
medium_free_009	800	71	3 817	21 443	53 287
medium_free_010	810	53	2 608	15 914	47 182
medium_free_011	810	53	2 604	16 086	46 620
london_night_00002	825	84	5 104	30 494	6 6714
galaxy_cluster2_00005	860	84	3 418	33 807	69 541
large_000	911	178	2 089	67 853	95 813
clouds_00001	912	83	5 736	29 770	82 494
universe_bg_00002	914	83	4 382	31 528	62 972
medium_012	923	86	4 181	31 775	68 041
microbes_00002	958	89	6 523	32 944	88 920
london_night_00003	961	83	4 888	30 800	72 652
clouds_00002	963	87	4 778	30 538	68 108
algae_00003	969	82	6 304	33 754	88 054
clouds_00003	1 000	85	4 603	31 950	70 790
medium_free_012	1 000	94	6 030	34 703	96 327
sun_00002	1 000	82	4 537	34 412	74 676
sun_00003	1 000	84	4 173	32 599	74 699
universe_bg_00003	1 000	88	4 922	33 283	77 957
algae_00004	1 113	79	5 055	37 311	81 203
redblue_00004	1 125	82	5 990	37,007	86 755
redblue_00005	1 125	88	5 075	35 197	81 827
medium_013	1 141	99	5 819	50 814	98 088
medium_014	1 165	73	3 698	35 154	85 170
medium_015	1 167	85	5 926	38 909	99 551
medium_016	1 180	94	5 644	40 253	93 435
microbes_00003	1 186	92	6 718	40 923	99 157
medium_017	1 202	114	5 722	44 693	98 981
london_night_00004	1 250	88	5 931	41 539	107 741
medium_free_013	1 250	86	6 644	40 459	105 943
microbes_00004	1 250	91	3 502	39 125	77 919
buffalo_004	1 404	104	7 501	54 687	123 827
buffalo_free_003	1 440	78	6 416	44 607	114 811
medium_free_014	1 440	69	4 803	32 748	111 708
medium_free_015	1 440	69	5 679	37 364	111 718
large_001	1 563	137	12 501	78 545	194 635
large_free_000	1 688	140	13 056	83 806	209 574
large_free_001	1 688	112	10 583	83 021	227 827
large_002	1 692	189	22 746	151 430	278 828
sun_00004	1 707	92	9 453	58 661	183 537
universe_bg_00004	1 721	87	10 032	66 277	197 621
clouds_00004	1 745	84	8 048	50 543	193 723
medium_free_016	1 750	82	9 519	63 704	200 048

		Makespan		Total Distance	
Instance	n	Lower Bound	Result	Lower Bound	Result
microbes_00005	1 818	89	9 911	61 868	203 056
algae_00005	1 875	88	9 599	63 545	206 705
clouds_00005	1 875	86	9 216	60 370	209 902
london_night_00005	1 875	92	9 437	62 677	207 923
sun_00005	1 875	86	9 056	59 459	202 767
large_003	1 906	154	15 846	118 357	277 189
medium_018	1 993	91	10 149	68 126	230 302
galaxy_cluster2_00006	2 000	186	17 847	123 805	314 979
large_free_002	2 000	179	19 778	156 375	329 811
universe_bg_00005	2 000	82	9 802	66 145	226 937
large_004	2 034	185	21 344	156 549	465 463
medium_019	2 068	85	10 222	71 212	240 024
buffalo_free_004	2 160	114	11 897	83 747	277 269
medium_free_017	2 250	87	10 448	78 890	273 050
medium_free_018	2 250	89	10 329	75 836	266 440
medium_free_019	2 250	89	10 039	71 203	262 489
clouds_00006	2 374	169	20 980	153 148	405 426
london_night_00006	2 394	177	19 612	146 138	381 816
microbes_00006	2 500	175	23 419	168 266	407 044
redblue_00006	2 778	181	25 645	191 344	470 172
galaxy_cluster_00006	2 850	180	21 582	183 060	465 248
galaxy_cluster_00007	2 871	182	23 490	194 828	495 096
galaxy_cluster2_00007	2 878	167	24 813	184 239	485 227
redblue_00007	2 894	182	25 006	190 448	484 544
galaxy_cluster2_00008	3 000	163	23 819	196 812	516 158
redblue_00008	3 000	173	21 410	197 389	497 111
universe_bg_00006	3 000	177	20 833	212 076	540 948
large_005	3 223	141	22 925	189 702	565 074
large_free_003	3 375	124	18 833	157 310	515 120
sun_00006	3 796	178	28 379	256 667	677 381
universe_bg_00007	3 820	184	53 970	256 524	663 762
large_free_004	3 938	127	21 908	196 318	631 702
algae_00006	4 000	176	27 231	268 289	725 397
algae_00007	4 000	182	30 490	266 775	715 707
clouds_00007	4 000	165	55 258	278 973	738 157
london_night_00007	4 000	183	29 738	258 035	712 759
microbes_00007	4 000	192	29 373	270 084	731 772
sun_00007	4 000	174	23 724	222 919	709 049
universe_bg_00008	4 000	184	29 280	270 421	726 457
redblue_00009	4 500	189	56 331	303 078	818 004
large_006	4 599	138	24 473	233 159	799 731
large_007	4 706	215	70 572	371 285	979 159
sun_00008	4 805	171	58 232	324 115	898 523
algae_00008	5 000	182	60 489	331 244	957 270
clouds_00008	5 000	182	59 533	331 641	958 305
galaxy_cluster_00008	5 000	175	59 120	312 047	973 955
large_free_005	5 000	184	59 018	334 479	945 261
large_free_006	5 063	136	24 510	250 853	879 317

		Makespan		Total Distance	
Instance	n	Lower Bound	Result	Lower Bound	Result
microbes_00008	5 643	188	65 269	378 352	1 125 676
london_night_00008	5 648	179	63 301	361 184	1 135 742
large_008	5 682	182	63 369	402 994	1 143 500
large_free_007	6 000	189	65 842	401 362	1 224 374
london_night_00009	6 000	185	36 782	377 360	1 243 964
microbes_00009	6 000	183	65 525	402 329	1 218 991
clouds_00009	7 229	178	42 109	485 724	1 588 384
algae_00009	7 311	176	41 634	492 856	1 585 282
sun_00009	7 500	187	41 845	498 637	1 652 955
galaxy_cluster2_00009	7 555	186	42 362	497 764	1 651 382
galaxy_cluster_00009	7 838	189	42 879	516 728	1 745 800
large_free_008	8 000	184	40 935	524 085	1 773 341
universe_bg_00009	8 000	185	42 104	534 956	1 796 412
large_009	8 595	176	44 042	574 544	1 980 538
large_free_009	9 000	182	45 367	576 459	2 067 695