EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Applied Algorithms Research Group

Strategies for Multi-Robot Motion Planning for Unlabeled Discs

Master Thesis

Koen van der Heijden

Supervisors:
Kevin Buchin
Irina Kostitsyna

Committee members:
Kevin Buchin
Irina Kostitsyna
Marcel Roeloffzen

Eindhoven, January 2020

Abstract

Recently, studies in multi-robot motion planning more and more often consider the unlabeled variant of the problem. Efficient algorithms that solve the unlabeled multi-robot motion planning problem have already been explored.

Most recently, Adler *et al.* considered the case of unlabeled multi-robot motion planning within a simple polygon workspace. They show that under certain conditions, a motion schedule can be computed efficiently. In this thesis, we investigate their algorithm experimentally and aim to extend it in two directions.

Firstly, we consider the case of non-simple workspace polygons. We show that, if we allow infinitely small holes, simultaneous movement of robots might be necessary.

Secondly, we aim to improve the quality of the motion schedule in terms of quality metrics like the lengths of the paths traversed by the robots or how often they need to be activated. To compute improved schedules, we present an alternative approach to solving the pebble motion problem on graphs. We compare the resulting motion schedules experimentally.

Contents

Chapter 1

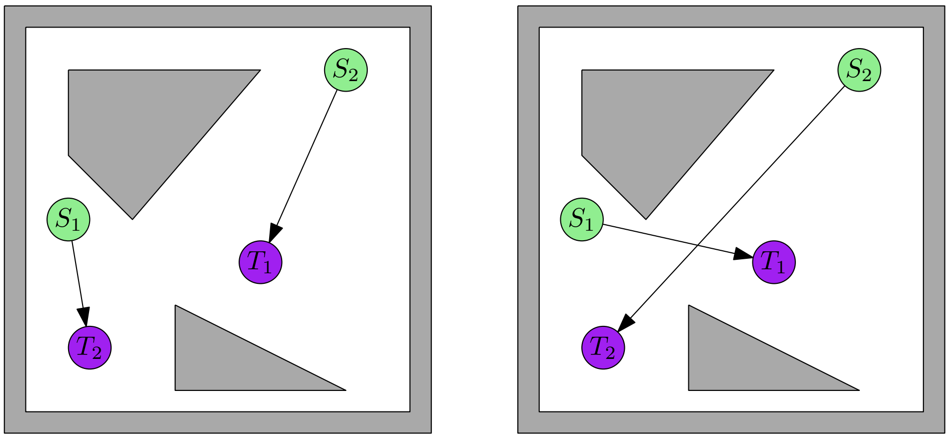
Introduction

Motion planning is a fundamental problem in robotics that is concerned with the planning of movements for autonomous objects in a workspace. In its most generic form, the goal of motion planning is to find a path in a workspace for a robot from a given starting position to some designated target position such that it does not collide with any obstacle in the workspace. There are many variants of the motion planning problem. A natural extension of the motion planning problem is the *multi-robot motion planning* problem [1]. In multi-robot motion planning, multiple robots have to move through a common workspace without colliding with an obstacle in the workspace or each other.

In cases where multiple robots move through a common workspace (i.e. multi-robot motion planning), the robots are often indistinguishable and thus interchangeable [2]. This setting of the problem is called *unlabeled* multi-robot motion planning (or in short, unlabeled planning). Given a set of identical robots inside a common workspace and a set of the same size consisting of destination positions, the goal is to find a motion plan such that every destination position is occupied by *some* robot. This is in contrast with the *labeled* variant (see Figure 1.1) of the multi-robot motion planning problem, in which every robot would have a designated, non-interchangeable destination position.

It is obvious that the (multi-robot) motion planning problem is relevant in robotics. However, the problem has various other applications, such as in computer game design [3] and crowd simulation [4]. Labeled planning has been studied for a longer time now (i.e. one of the first occurrences was by Schwartz and Sharir in 1983), while the unlabeled version has been looked at more recently [2].

When the number of robots is no longer constant, but rather variable, e.g. m, the problem becomes hard. This version with a variable number of robots is already shown to be PSPACE-hard in the relatively simple setting with m rectangular robots in a rectangular workspace [6] . The more general (labeled) multi-robot motion planning problem is already proven to be strongly NP-hard for disc robots (with varying radii) in a simple polygon workspace [7]. However, by adding constraints to the setting of the multi-robot motion planning problem, more efficient algorithms can be created, such as the algorithm described by Adler *et al.* [1], which we will discuss in more detail in Chapter 2.



(a) Unlabeled, the robot starting at S_1 moves to T_2 and the robot starting at S_2 moves to T_1

(b) labeled, the robot starting at S_i has to move to the corresponding target configuration T_i

Figure 1.1: The 2 variants of *multi-robot motion planning*.

1.1 Related work

Schwartz and Sharir were one of the first authors to study multi-robot motion planning from the geometric point of view in their series of papers *Piano Movers' Problem* in 1983 [5], [8]. The particular variant studied by Schwartz and Sharir was the labeled variant, which is described in [5] as follows: Given a collection of bodies B, which may be hinged, and given a region bounded by a collection of polyhedral or other simple walls, decide whether or not there exists a continuous motion connecting two given positions and orientations of the whole collection of bodies. Later, Schwartz and Sharir showed that the problem can be solved in $O(n^3)$ for the case where B contains 2 circular bodies in a 2D workspace, where n is the number of walls in the workspace. However, the algorithm was still exponential in the number of moving bodies [8]. Later, Yap [9] managed to create an algorithm using the *retraction method*, which has an improved complexity of $O(n^2)$ for two robots and $O(n^3)$ for three robots. Several years after that, Sharir and Sifrony [10] created an algorithm based on *cell decomposition* that has less constraints on the robot shapes and has a running time of $O(n^2)$ (again for 2 robots).

Different approaches have been developed to solve the multi-robot motion planning problem. Decoupled techniques (e.g. [1], [11]) try to partition the problem into multiple, easier to solve sub-problems. Centralized techniques (e.g. [10], [12]), however, work in the original configuration space, in which the number of degrees of freedom is normally higher. Therefore, the centralized techniques tend to be less efficient than the decoupled techniques, but centralized techniques often come with stronger guarantees [13].

Multi-robot motion planning is a continuous problem, however it can be transformed into a discrete problem on a so-called motion graph [14]. Robots can be represented as pebbles that are placed on the vertices of the graph. Movements of robots are done along the edges in the graph. The labeled setting for the discrete problem on a graph is shown to be testable for feasibility in linear time [15], and actual complete planners also show to be efficient [14], [16]. For the discrete problem in the unlabeled setting, complete and efficient planners have already been made which are able to generate the optimal solution [17]. While there are significant differences between the continuous motion planning problem and the discrete version, several continuous techniques make use of concepts from the discrete domain [1], [18].

The unlabeled setting of the problem for disc robots in a 2D polygonal workspace is solved for different assumptions and goals [1], [19]. Adler *et al.* [1] showed that the unlabeled setting can be transformed into a discrete motion planning problem on a graph. They show that if a solution to the motion planning problem exists, then such motion plan can be generated by solving the discrete motion planning problem on the graph. Specifically, a motion along an edge in the graph is translated to a movement of a robot in the 2D workspace. The algorithm by Adler *et al.*, however, only works under the assumptions that the workspace is a single polygon, the robots are unit discs and all source and destination positions for the robots are separated by at least 4 units. Also, the solution for the motion planning problem given by Adler *et al.* is not optimized. Solovey *et al.* [20] introduced an algorithm that computes a solution which minimizes the sum of lengths of the individual paths in the motion plan. They even show that the total length of the resulting motion plan is at most $OPT + 4m$ where OPT is the optimal solution cost, and can be computed in $O(m^4 + m^2n^2)$ time [20]. However, their algorithm has one more assumption compared to the algorithm by Adler *et al.*, namely that the center of every robot needs to be at least separated by $\sqrt{5}$ units from any obstacle.

1.2 Problem description

The problem that we will look at in this thesis, the *multi-robot motion planning problem for unlabeled unit disc robots*, can formally be described as follows:

Definition 1. *Let W be a simple workspace polygon with complexity n. Let S be a set of starting positions of size m and let T be a set of target positions of size m. For every pair $u, v \in S \bigcup T$ such that $u \neq v$ it holds that $|u - v| \geq 4$. Determine whether a collision free motion schedule exists in which unit disc robots from a given starting position $s \in S$ towards any unoccupied target position $t \in T$, and if such motion schedule exists, find it.*

Solutions to this problem have already been studied (see [1]) and efficient algorithms for it have been discovered. However, these algorithms are often not optimized for the quality of the motion schedule, or they are more constrained on the input. Adler *et al.* [1] show an algorithm that can verify whether a valid motion schedule exist for the problem defined in Definition 1 in $O((m + n) \log n)$ time, and if such schedule exist, it can be computed in $O(n \log n + mn + m^2)$ time.

In this thesis, we will look at strategies to solve the given problem when some constraints are loosened or removed. We will also try to discover strategies for improving the quality of the motion schedule.

Research questions In this thesis, we will address the following research questions related to the algorithms described by Adler *et al.* [1]: Can the algorithm be extended to the case of workspace polygons with holes? Can the resulting motion schedule be improved by using the motion graph differently? How does the algorithm perform experimentally?

1.3 Results

Adler *et al.* [1] considered the multi-robot motion planning problem for unlabeled unit-disc robots in a simple workspace polygon. They showed that under certain conditions, a motion schedule can be computed efficiently.

We first consider the algorithm they discussed in the case of non-simple polygons. In Chapter 3 we show if we allow infinitely small holes, simultaneous movement of robots might be necessary. We also conjecture that if holes in the workspace polygon are not infinitely small, there is a solution in which no simultaneous movement is necessary.

We also aim to improve the quality of the motion schedule in terms of quality metrics such as the lengths of the paths use by the robots, or how often robots need to be activated. In Chapter 4 we discuss a way of solving the motion graph using purple trees (Lemma 1). This algorithm shows that every robot needs to be activated at most once. Due to the robots activating only once, and thus moving towards their destination in one direct movement, the paths covered by the robots are also shorter on average when using purple trees.

The different parts of the algorithm are evaluated separately in Chapter 5. We will see that it is inefficient to compute the motion graph completely. However, we will see that the other parts of the algorithm seem to agree with their corresponding theoretical time complexity.

1.4 Outline

In Chapter 2 the preliminaries for this thesis will be described. Chapter 3 contains a discussion about how the algorithm performs when we drop the assumption that the workspace is a simply polygon but rather a polygon with holes. In Chapter 4, we discuss quality metrics of the algorithm and its output. We also discuss how the quality of the motion schedule can be improved. Chapter 5 contains an experimental evaluation of the different approaches of computing a motion schedule, based on different input data sets. Finally, we conclude with a discussion in Chapter 6.

Chapter 2

Preliminaries

Consider the multi robot motion planning with m unit disc robots and a simple workspace polygon $W \subset \mathbb{R}^2$ of complexity n, and consider a starting position S_i and destination position T_i of these m robots. In the unlabeled variant of multi robot motion planning, the question then is how to move every robot from its starting position S_i to its target position T_i such that no robot collides with the workspace W or with any other robot.

This problem is given more formally in Definition 1.

2.1 Spaces

Define the spatial pose of a robot, i.e. the coordinates of a predefined anchor point of a robot and the rotation in every euclidean plane, as its *configuration*. For a 2D workspace, the configuration of a robot can thus be described by the x and y coordinates of an anchor point and the rotation of the robot in the xy plane. For a 3D workspace, the configuration of a robot would be described by the x, y and z coordinates and the rotation of the robot in the xy, yz and zx planes. The *configuration space* of a workspace can be described as the set of all possible configurations, i.e. all possible combinations of coordinates and angles of rotation in the workspace. Since the problem as described in Section 1.2 considers unit disc robots in a simple 2D workspace, a configuration can be simplified to just the x and y coordinates of a robot, where the anchor point is the center of the robot, since any rotation of a disc around its center does not change the spatial pose of the disc. Therefore, we can define the configuration space as $C = \mathbb{R}^2$.

The configuration space can now be split into two subsets: the *free space* and the *obstacle space*. Define $D_r(x)$ for every $x \in \mathbb{R}^2$ as the disc or radius r around x. Then, the free space can be defined as all configuration c in the configuration space C such that $D_1(c)$ (which is the spatial location of the robot) does not collide with the workspace W. More formally, $F = \{c \in C | D_1(c) \bigcap W = D_1(c)\}$ where F is the free space, C is the configuration space and W is the workspace. The obstacle space can be defined as the exact complement of the free space, i.e. $O = \{c \in C | D_1(c) \bigcap W \neq D_1(c)\}$ where O is the obstacle space, C is the configuration space and W is the workspace.

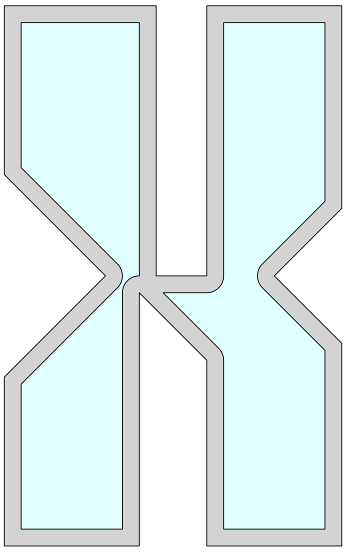


Figure 2.1: A 2D simple polygon in the euclidean space (i.e. the configuration space). Inside the gray workspace, the light blue area shows the free space.

2.2 Directed interference forest

The algorithm described by Adler *et al.* [1] considers all robots and their destinations to be *well separated*. The set of starting configurations of the robots in the algorithm is called $S \subseteq F$ and the set of target configurations is called $T \subseteq F$. We define 2 configurations $c_1, c_2 \in S \bigcup T$ to be well separated if $||c_1 - c_2|| \geq 4$. The distance between two robots in the workspace at configuration c_1 and c_2 would then be at least 2.

The free space F can be split into different simply connected components $F_i \in F$. It is known that a robot that is at configuration $c_i \in F_i$ cannot move to a configuration $c_j \in F_j$ if $F_i \neq F_j$ because there would be a configuration c_x on the path from c_i to c_j which is not in F. Thus, such c_x would be in O, and by definition, it would then collide with the workspace W. Therefore, it is clear that every free space component $F_i \in F$ can be solved as an independent subproblem.

However, it is possible that a robot at a configuration $c_i \in F_i$ does interfere with another free space component F_j. Let the disc around a configuration c, defined by $D_2(c)$, be the *collision disc* of c. If a robot at configuration c_j is inside of the collision disc of c_i, $D_2(c_i)$, then $||c_i - c_j|| \leq 2$. This would by definition imply that the robots at these configurations collide. Figure 2.1 shows an example in which the robot in one free space component interferes with the another free space component.

Such interference can be counteracted by solving specific free space components before others. To determine in which order free space components are solved, a *directed interference forest* is created. This is a directed, acyclic graph in which every vertex v_i refers to a free space component F_i. Let G be a directed interference forest. There exists an edge from vertex $v_i \in G$ to vertex $v_j \in G$ if either there is a configuration $s \in F_i$ that is in S such that $D_2(s) \bigcap F_j \neq \emptyset$ (see Figure 2.1) or there is a configuration $t \in F_j$ that is in T such that $D_2(t) \bigcap F_i \neq \emptyset$. Then, the free space components can be solved with respect to the topological ordering of G. This ensures that whenever a free space component F_i is selected for solving, every other free space component F_j that has a starting configuration interfering with F_i is already solved (i.e. the robot at that configuration is already moved away from it) and every other free space component F_j that has a target configuration interfering with F_i is not yet solved (i.e. the robot that will finish at that configuration is not yet moved towards it). Adler *et al.* [1] also show that G is indeed an acyclic graph if the workspace is a simple polygon.

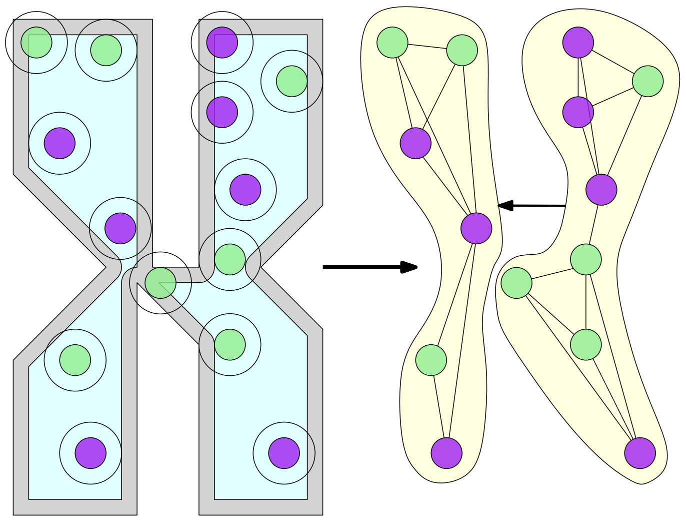


Figure 2.2: On the left, a workspace (in gray) with the corresponding free space (in light blue) and a set of robots at their start configurations S and target configurations T. The circle around a robot represent its collision disc $D_2(c)$. Since the $D_2(S_3)$ overlaps with the left free space component, the interference forest on the right contains an edge from the right vertex to the left vertex. In the vertices of the interference forest, a motion graph is shown.

2.3 Motion graphs

Solving a single free space component is done by means of a *motion graph*. A motion graph G_i is an undirected graph in which vertices represent configurations $c \in F_i$ in $S \bigcup T$. An edge from $v_i \in G_i$ to $v_j \in G_i$ exists if there is a path from the configuration c_i represented by v_i to c_j represented by v_j such that this path does not collide with the workspace or any collision disc of a configuration in $S \bigcup T$. This means that such edge exists if a path from one configuration in $U = S \bigcup T$ to another configuration U exists without colliding with any $D_2(u)$ for all $u \in U$.

2.4 Algorithm by Adler *et al.*

Adler *et al.* [1] showed that a motion graph G_i can be solved by determining a spanning tree of G_i and solving this using the *pebble motion problem* (described in more detail in Section 4.1). The pebble motion problem can be visualized as having a physical copy of a graph, and putting a pebble on every starting configuration. Then, select a leaf in the spanning tree that represents a destination in the original problem. From there, do a breadth first search to find the closest vertex in the tree containing a pebble, and move that pebble to the leaf. This represents a robot moving from a configuration (where the pebble was) towards a destination. If no leaf exists in the spanning tree that represents a destination in the original problem, select a random leaf and move the pebble on this leaf to the closest available empty vertex. After that, remove the previously selected leaf from the spanning tree. The movement made by the pebble is to be saved as a step in the motion plan. When all pebbles are moved to a destination configuration, the motion plan is done.

However, no claims are given about the efficiency of the resulting motion plan based on the spanning tree that is used.

Chapter 3

Polygons with holes

One of the preconditions of the previously described algorithm is that the workspace is a simple polygon. This requirement ensures that the directed interference forest is indeed a directed forest (see Lemma 3 and Lemma 4 in [1]).

Let $G = (V, E)$ be a directed interference forest representing workspace W and sets of start and target configurations S and T, respectively. If W is not a simple polygon, we cannot guarantee that E does not contain both (v_i, v_j) and (v_j, v_i) for any pair of vertices $v_i, v_j \in V$ such that $v_i \neq v_j$. We can also not guarantee that G does not contain an undirected cycle. Thus, we should no longer call G a directed interference forest, but rather a *directed interference graph* in this case.

Figure 3.1 shows an example of a workspace polygon W containing multiple holes. The corresponding interference graph (N.B.: This is no longer a forest) is also given.

We can see that the algorithm can no longer determine an ordering in which the free space components can be solved, because there is no topological ordering possible in this graph. There is no valid root in the directed interference graph (i.e. every vertex has at least one incoming edge). However, the algorithm could be extended in such way that this example is still solvable. Let $F_i, F_j \in F$ be two distinct free space components such that the corresponding vertices $G_i, G_j \in G$ are connected by an edge (G_i, G_j). Then there is either a starting configuration $s \in F_i$ such that $D_2(s) \bigcap F_j \neq \emptyset$ or a target configuration $t \in F_j$ such that $D_2(t) \bigcap F_i \neq \emptyset$.

For the former case, let $c \in F_i$ be a configuration which such that there exists a path π_{sc} from s to c. If for every configuration $p \in \pi_{sc}$, $D_2(p)$ does not overlap with any other free space component $F_k \in F \setminus (F_i \bigcup F_j)$, then the robot that is initially at s can freely move to c without interfering with another free space component. For the latter case, a configuration c can be determined in the same way.

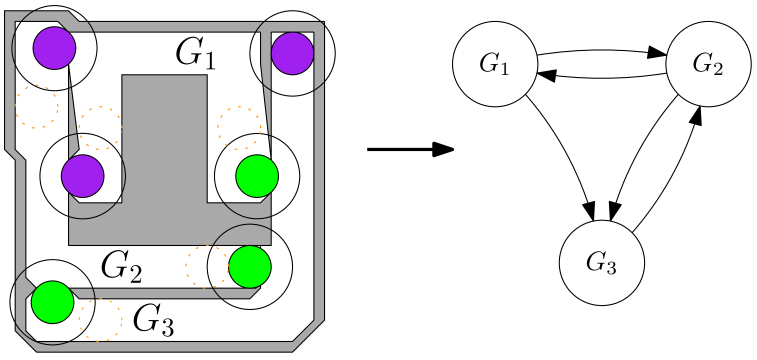


Figure 3.1: A polygon with 3 holes and its corresponding directed interference graph containing directed cycles. Starting configurations are shown in green and target configurations are shown in purple.

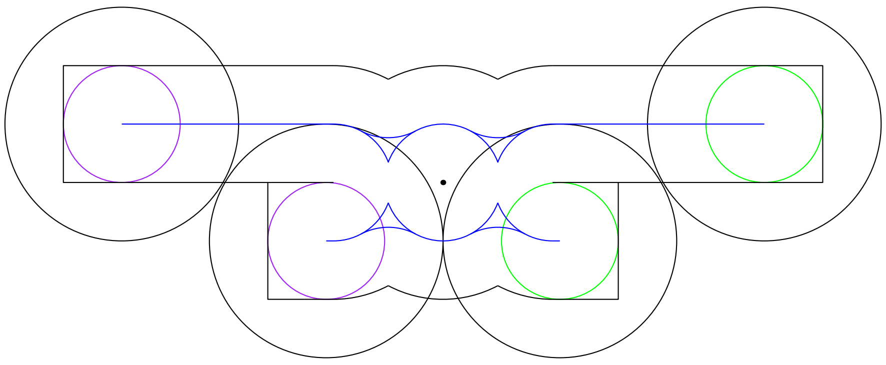


Figure 3.2: A workspace polygon with an infinitely small hole. The free space components are shown with a blue outline. Starting and target configurations for the robots are given using a green and purple outline, respectively.

In Figure 3.1, such configuration is given for every interfering configuration using a orange dashed circle. We can solve this example by replacing every starting or target configuration $u \in S \bigcup T$ with its corresponding non-interfering configuration u^*. We know that there is a path between u and u^* which does not interfere with other free space components. Thus, to solve the example, we first have to move every robot from the starting configuration $s \in S$ to its non-interfering counterpart s^*. Then we can solve the adapted version of the original problem in which every starting configuration $s \in S$ is replaced with s^* and every target configuration $t \in T$ is replace with t^*. Finally, we have to move every robot at its adapted target configuration t^* to the actual target configuration $t \in T$.

Point holes However, there are cases in which there is no configuration $c \in F_i$ for some $F_i \in F$ such that $D_2(c) \bigcap F_j \neq \emptyset$ for some $F_j \in F$, $F_i \neq F_j$. Figure 3.2 shows such case, containing two free space components, shown using a blue outline. For every configuration c in the bottom free space component F_{bottom} it holds that $D_2(c) \bigcap F_{top} \neq \emptyset$. This example is even stricter: For any $c \in F_{bottom}$ it holds that $|F_{top} \setminus D_2(c)| > 1$, i.e. F^*_{top} is always split into multiple connected components by a robot at any configuration $c \in F_{bottom}$.

Thus, there is no configuration c in either free space component such that the cycle in the interference graph is removed for the example given in Figure 3.2. In fact, this example is not solvable by the presented algorithm. The only way for the robot in the top free space component to move from its starting configuration to its target configuration is by crossing the robot in the bottom free space configuration. However, to cross the robot in the bottom free space configuration, both robots have to rotate around the zero-width hole simultaneously. N.B.: Simultaneous movement cannot be represented in a motion schedule, not even by adding intermediate configurations.

Conjecture However, let us point out that, if the size of the hole in the given example would not be arbitrary small, then it should be possible for a robot to move an amount proportional to the size of the hole if there would be a robot on the other side of the hole. Thus, if a hole is not arbitrary small, the solution of the problem would be described using a motion schedule (in which no simultaneous movement takes place). Therefore, we conjecture that the algorithm is adaptable such that it can solve cases containing holes with a size that is not arbitrary small.

Chapter 4

Computing efficient schedules

As previously described in Chapter 2, solving a motion graph can be done using a pebble game. A pebble game is a mathematical game in which "pebbles" are moved in a graph. There are many different types of pebble games. In the variant that we are looking at, pebbles can only be moved to neighbouring nodes. At any point during a pebble game, every node should have at most one pebble on it.

To use a pebble game to solve a motion graph, every robot is represented by a pebble. Initially, every node in the motion graph that represents a starting configuration should contain a pebble. The target is to move all pebbles to a node in the motion graph representing a target configuration.

A large part of solving the pebble game is about selecting which node can be processed. This can be done using a spanning tree. Solving the pebble game using a spanning tree is done by selecting a leaf and processing that, after which it can be removed (i.e. it will no longer be used in the next steps). The node that is selected for processing should not invalidate the motion graph (i.e. disconnect the motion graph such that the disconnected components can no longer be solved).

Determining how efficient a given motion schedule is is done based on different criteria. Efficiency can mean multiple things: efficiency of the algorithm itself (i.e. time complexity or space complexity), or quality of the resulting motion schedule (i.e. size of the motion schedule, sum of the length of the paths in the motion schedule, etc.).

4.1 Spanning trees

Generating spanning trees can be done in different ways. A minimum spanning tree, a spanning tree in which the edge weight is minimised, can be computed by using e.g. Kruskal's algorithm [21] or Prim's algorithm [22].

The definition of a minimum spanning tree is based on the edge weight. Different values as edge weight in a motion graph can be used, e.g. Euclidean distance, shortest path length in the free space component, or even a constant distance (i.e. an arbitrary spanning tree). Schedules can differ based on the edge weight chosen to generate the spanning tree, since different edges from the motion graph are used to create such spanning tree.

Adler *et al.* [1] do not actually compute a full motion graph, but rather compute a part of the motion graph. Figure 4.1 shows how the part of the motion graph is created using their method. Note that they do not create a spanning tree, but rather a sparse motion graph. However, to determine the effect of different edge weights, we choose to compute the full motion graph so no edges are left out when determining the minimum spanning tree.

The (minimum) spanning trees are used to compute efficient paths between different configurations. Since a spanning tree is connected, every vertex in it is reachable from every other vertex. The result of removing a leaf from a spanning tree is still a spanning tree. Therefore, selecting leafs from a minimum spanning tree to process alters the spanning tree in such way that it remains solvable by the algorithm.

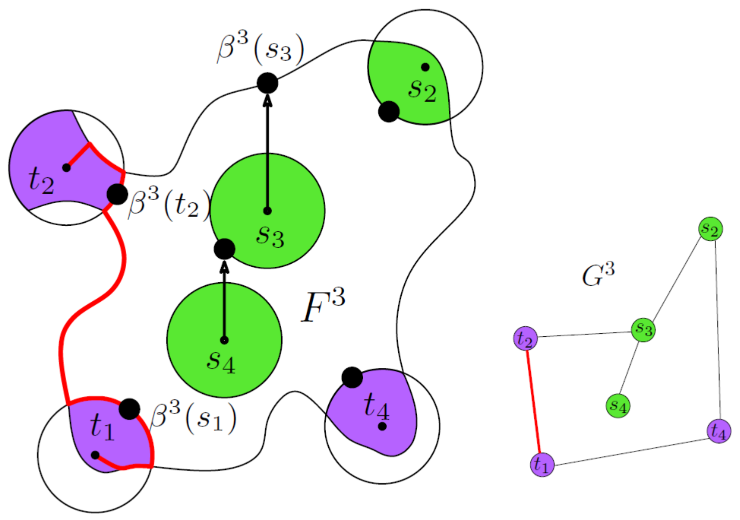


Figure 4.1: An example of how a free space component would be transformed into the relevant part of a motion graph using the method described by Adler *et al.* [1]. The boundary positions of F^3 consist of $B^3 = \{s_2, t_1, t_2, t_4\}$, while the hole positions consist of $H^3 = \{s_3, s_4\}$. For every $x \in H^3$ a ray is shot upwards to determine its boundary representative $\beta^3(x)$. If a boundary representative $\beta^3(x)$ lies on the boundary of F^3, it is connected as if x was positioned at $\beta^3(x)$. If a boundary representative lies on the boundary of another $D_2(y)$ for some $y \in S \bigcup T$, then an edge between x and y is added to the motion graph.

4.2 Strategies for moving pebbles

The strategy to solve the pebble game described by Adler *et al.* [1] finds a leaf, which preferably represents a target configuration. If such vertex exists, the closest pebble can be moved towards it, and that leaf can be removed from the spanning tree since the pebble on that leaf will never have to move away from it.

If such vertex does not exist, an arbitrary leaf (which will thus represent a start configuration) will be selected. If this leaf contains no pebble, it can be removed from the spanning tree since no pebble will ever have to move past it to go to a target configuration. If this leaf does contain a pebble, the algorithm finds the closest empty node in the spanning tree and moves all pebbles on the path to that node one step towards it, and the leaf can then be removed (since it now is an empty starting configuration).

This algorithm creates a motion schedule which makes robots move multiple times. Figure 4.2 shows an example in which the pebble which is initially placed on S_5 moves forward and backward up to $O(n)$ times before being placed on its target position. Let B_i be the branch of the tree which initially has S_i as leaf. Let S_1 be the leaf selected for processing (note that there are no blue leaves). The closest blue node would be in branch B_4. This can be done three times (i.e. by selecting the leaf from B_1) before two possible closest empty nodes are available. The fourth time a leaf l from B_1 is selected, the distance between l is $||l - S_i|| + 4$

In the worst case, the fourth time we select a leaf from branch B_1, the closest empty node that is found is thus in branch B_2. This would move the pebble which was initially at S_j towards S_k. Note that at this point, the distance from S_l to the closest empty node in B_2 is 2, while the distance from S_l to the closest empty node in B_4 is 7. Thus, we can select a leaf from B_3 and process it 5 times before we get another pair of nodes which are tied closest.

Again, in the worst case, the next time we select a leaf from branch B_3, the closest empty that is found is in branch B_4. This would push the pebble which started at S_j, which was currently at S_k, back to S_j. At this point, the distance from the S_i to the closest node in B_4 is 5, while the distance from S_i to the closest node in B_2 is 10.

Using this method, every time after the first time we switch from selecting a leaf from branch B_1 to selecting a leaf from branch B_3 or vice versa, the difference between the distance of the closest in branch B_2 and branch B_4 is 5. This implies that a total of 6 nodes have to be selected to get a new pair of tied closest nodes. In the worst case scenario, every time such node is found, the pebble starting at S_j is moved one place. However, the direction of this push is swapped every time this occurs. Therefor, the pebble starting at S_j will, in the worst case, move $O(n)$ times, while it could be placed only a constant distance away from its initial position. Thus, it might be significantly more efficient to determine a movement schedule in which only start and target configurations are linked, without any intermediate movements.

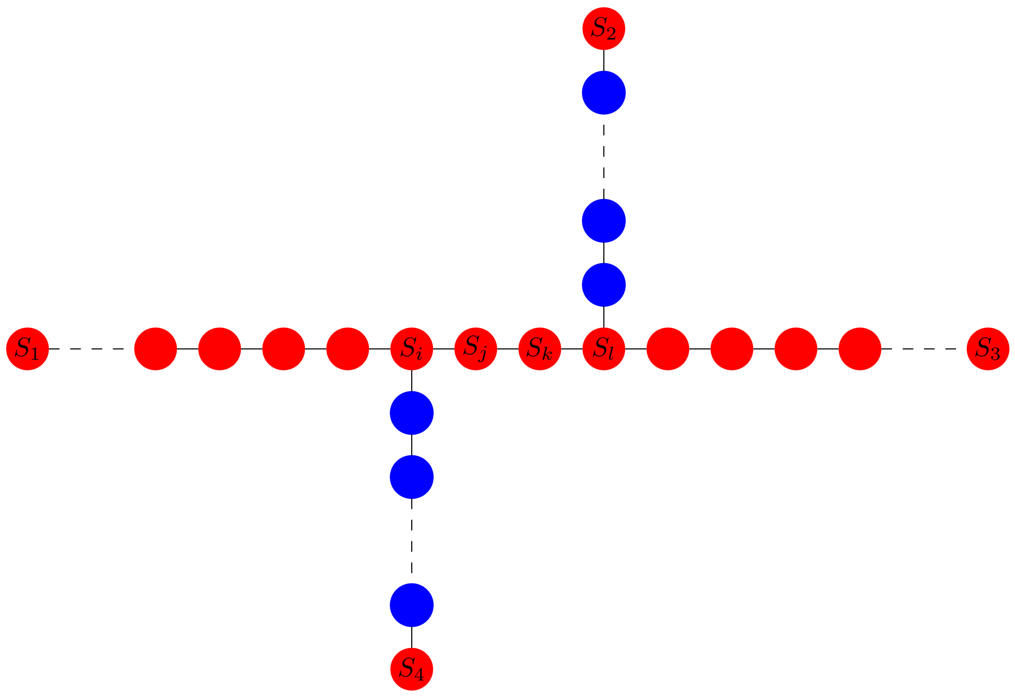


Figure 4.2: A motion graph, in which start configurations are colored red and target configurations are colored blue. The dashed lines between two vertices of the same color represent an arbitrary amount of vertices of the same color between them. Since an arbitrary red leaf is selected when solving a motion graph, there are cases in which the pebble at S_j or at S_k moves forward and backwards up to $O(n)$ times before being placed on its final position.

4.2.1 Purple tree

When selecting a leaf from the spanning tree which represents a target configuration, the closest pebble moves towards it before removing the leaf from the spanning tree. This implies it is the last move a pebble makes before being removed from the spanning tree. Also, when a leaf representing a starting configuration is selected which does not longer have a pebble it is removed without moving any pebble. Thus, if all leafs that are ever selected either represent target configurations or represent starting configurations which no longer have a pebble, every move of a pebble will be its last (and thus only) move. To formally use this property, we will first define a *purple tree* as follows:

Definition 2 (Purple tree)**.** *A purple tree $T = (V, E)$ is a colored tree in which every vertex $v \in V$ is colored blue, red or purple, and the number of blue vertices is equal to the number of red vertices.*

The spanning tree has four different types of nodes: one type represents a target configuration and has a pebble, one type represents a target configuration and has no pebble, one type represents a starting configuration and has a pebble and the last one represents a starting configuration and has no pebble. Then, we know that initially the tree has only starting configurations with pebbles and target configurations without pebble. Let the color of a starting configuration containing a pebble be red, and the color of a target configuration not containing a pebble blue. Both other types of configurations are represented by purple vertices. Then, initially the spanning tree is a purple tree with 0 purple vertices.

If the spanning tree has a blue leaf n, select it for processing. During processing, the closest pebble in the spanning tree can be moved to n [1]. After moving that pebble, n is a blue leaf containing a pebble. However, there is another red node somewhere in the spanning tree containing no pebble now. Since the blue leaf can be removed from the spanning tree (as it is processed), the spanning tree now has one more red leaf compared to the amount of blue leafs. To counteract this, the red node containing no pebble can be colored purple.

If the spanning tree has no blue leafs, but it has a purple leaf n, leaf n can just be removed from the spanning tree. Since leaf n is purple, the remaining spanning tree is still a purple tree according to Definition 2.

If the spanning tree has neither blue leafs nor purple leafs, Lemma 1 can be used to split the purple tree into two smaller purple trees.

Lemma 1. *A purple tree $T = (V, E)$ with only red-colored leafs contains an edge $e \in E$ such that $(V, E \setminus e)$ has 2 connected components, which are both purple trees.*

Proof. Let $T = (V, E)$ be a purple tree with only red leafs and a red root r. Note that r can also be a leaf. Define the value of a vertex $v \in V$ as the number of red vertices minus the number of blue vertices in the sub tree with root v.

Let $C = children(r)$ be the set of children of w. We know that $value(r) = 0$, since T is a purple tree. Since $color(r) = red$, we know that $\sum_{c \in C} value(c) = -1$. This implies that there exists at least one $c \in C$ with $value(c) < 0$ and c can be any color, i.e. red, blue or purple.

Let $w \in V$ be an arbitrary vertex with $value(w) < 0$. Let $C = children(w)$ be the set of children of w. Then, if w is a blue vertex, $\sum_{c \in C} value(c) \leq 0$. This implies that either all $c \in C$ have $value(c) = 0$ or there exists at least one $c \in C$ with $value(c) < 0$. If w is a red vertex, then $\sum_{c \in C} value(c) < -1$, which also implies $\sum_{c \in C} value(c) < 0$. If w is a purple vertex, then $value(w) = \sum_{c \in C} value(c)$ which implies $\sum_{c \in C} value(c) < 0$.

We now know that there must be some $c \in children(r)$ such that $value(c) < 0$, and that for every $v \in V$ with $value(v) < 0$ there exists some $c \in children(v)$ with either $value(c) = 0$ or $value(c) < 0$. Since, by definition all leafs $l \in V$ are colored red, we know that $value(l) = 1$. Therefor, we know that there must be a sub tree with root $s \in V$ with $value(s) = 0$ in T.

We can then remove $(parent(s), s) \in E$ from T, resulting in 2 connected components. Let $S = (V_s, E_s)$ be the sub tree of T with root s. Then we know that S is a purple tree, since $value(s) = 0$. Let m be the number of blue vertices in V_s. Then the number of red vertices in V_s is by definition also m. Let n be the number of blue vertices in V. Then the number of red vertices in V is by definition also n. Let $R = (V \setminus V_s, E \setminus \{E_s \cup (parent(s), s)\}$. Then the number of blue vertices in $V \setminus V_s$ is $n - m$ and the number of red vertices in $V \setminus V_s$ is also $n - m$. Therefore, R is also a purple tree. □

Figure 4.3 shows an example of a purple tree which has only red leaves and a red root. The edges which can be removed according to Lemma 1 such that the resulting trees are also purple trees.

Using Lemma 1, any purple tree with only read leafs can be split into two purple trees. It is possible that the resulting purple trees still do not have a blue leaf of course. However, then the purple tree can split into two smaller purple trees again. At some point, such purple tree contains only two leafs and by definition of a purple tree, it then contains exactly one red and one blue leaf. Thus, the smallest possible purple tree by definition has a blue leaf. Lemma 4.2.1 directly translates into an algorithm to solve the pebble motion problem.

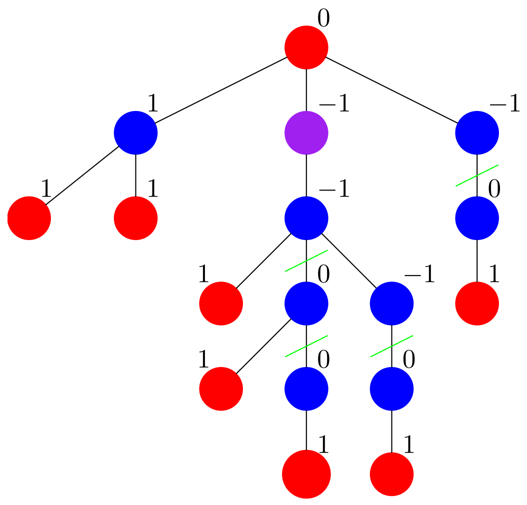


Figure 4.3: A purple tree with 8 red nodes, 8 blue nodes and 1 purple node. The edges which can be removed from the purple tree such that the resulting components are still purple trees are cut with a green line. The corresponding value of every node is also shown.

When applying this algorithm to the example in Figure 4.2, the first step would be to cut either S_2 or S_4 and its connected blue node from the rest of the tree. After this, one of the resulting trees contains two nodes, i.e S_2 or S_4 and its corresponding blue node. The other tree now also has a blue leaf, and can thus be solved by moving the closest pebble towards it.

4.3 Solutions without a spanning tree

Another option to solve the motion graph could not use a spanning tree. However, selecting the correct node to process in the motion graph is significantly harder than selecting a node from the spanning tree. The nice property of a leaf in a tree is that it will never disconnect the tree when removed. Though, there is a good chance that a motion graph does not have any leafs.

It is possible to select a node $n \in G$ such that G disconnects into more connected components. Though, this is only allowed when every connected component is still solvable (i.e. there are as many pebbles as target configurations in it). Thus, it is significantly easier to ensure no multiple connected components appear. This means that the selected node n should preferably not be a *bridge*. A bridge is a node $n \in G$, where G is a connected graph, such that $G \setminus n$ is still a connected graph.

For future work, a possible way to go would be finding an algorithm that is able to select either a node $n \in G$ such that n is not a bridge, or find a node $n \in G$ such that every connected component in $G \setminus n$ is still solvable. The former solution looks easier to solve, however, the latter solution might be more efficient.

Chapter 5

Experimental evaluation

In this chapter, we present an experimental evalution of the algorithms discussed in this thesis. The goal for these experiments is to find out how the algorithm performs on inputs of different sizes and shapes. We perform experiments to test the scalability of the algorithm, and the quality of its output.

To determine how well the algorithm scales, the running times of different parts of the algorithm are compared for a variable number of points in the input workspace and a variable number of robots. To properly test the effects of the workspace complexity and the number of robots on the algorithm, these amounts are varied independently (i.e. one is kept constant, while the other is variable).

Quality metrics The quality of the output of the algorithm can be described in different ways. The quality metrics we are looking at here are:

- The sum of the lengths of the paths of all robots according to the motion schedule. This provides an easy way to quantify how well the motion was planned.
- The total number of times a robot has to be activated (i.e. has to start moving). This metric is useful for cases where activating a robot would impose some sort of cost.

The algorithm consists of multiple components, which are all evaluated separately. First, the workspace is converted into the set of free space components. The free space is not necessarily a single component (see for example Figure 2.1). When a corridor in the workspace is smaller than the diameter of a robot, robots can not move through it, thus disconnecting the components at both sides of the corridor. Generating the free space components is done using the approximated inset algorithm provided by CGAL [23]. The next step is to transform every free space component into a corresponding motion graph. Then, the motion graphs are connected in a directed interference forest. Finally, the motion graphs are solved independently according to a topological ordering of the interference forest.

Edge weights for the motion graph The scalability tests are performed to determine the quality of the algorithm and the implementation. The output quality tests are performed to determine the quality of the algorithm. Specifically, different ways of solving the motion graph are compared. All versions make use of a spanning tree, but generating the spanning tree is done using different parameters. The spanning trees are generated using Kruskal's algorithm [21]. However, there are different ways of determining the edge weights, which change the resulting (minimum) spanning tree. The compared edge weights used are:

Constant. A constant edge weight is used to generate an arbitrary spanning tree. Computing a constant edge weight takes no time.

Euclidean distance. A Euclidean edge weight is relatively easy to compute (only constant time per edge). However, the results will most likely neither be optimal time-wise, nor in terms of the quality of the output. Though, Euclidean edge weights might be a good compromise.

Geodesic distance. Computing the geodesic distance takes relatively long. However, the resulting edge weight actually corresponds to the length of the path a robot has to move along.

Computing the geodesic edge weight is done using Dijkstra's shortest path algorithm on a triangulation of the free space components. The size of the triangulation scales linearly with the complexity of the workspace polygon, and also linearly with the number of robots. The time complexity of Dijkstra's algorithm is $O(V \log(V))$, where V is the number of vertices in the triangulation. To compute all edge weights, Dijkstra's algorithm is ran for every vertex in the motion graph (note that Dijkstra's algorithm computes the shortest paths to all nodes starting from a single source node). This implies that the time complexity of computing all geodesic edge weights is $O(m \cdot (m+n) \log(m+n))$.

Adler *et al.* [1] used an arbitrary spanning tree to solve the motion planning problem. Therefor, we decided to keep the arbitrary spanning tree (i.e. minimum spanning tree with constant edge weight) as reference in our comparison. For efficiency reasons instead of using the Euclidean distance, we will use the squared Euclidean distance. The minimum spanning trees for these distances are the same, but the squared Euclidean distance avoids computing square roots. We expect that the (squared) Euclidean distance will on average perform better then constant edge weights. The geodesic distance edge weight should perform significantly better than the other edge weights, however we expect the cost of computing the geodesic distance between every pair of configurations to be very high.

Also, note that, in the worst case, a single motion graph for any given data set can be a complete graph. This means that the number of edges in the motion graph can go up to $|E| = \frac{|V| \cdot (|V|-1)}{2}$ where $|V|$ is the number of vertices in the graph. This implies $|E| = O(|V|^2)$ for complete motion graphs. Note that the number of vertices in the motion graph is equal to $2 \cdot m = O(m)$, where m is the number of robots in the data set. Thus, the number of edges for which the weight has to be computed can go up to $O(m^2)$.

Solving the motion graph We also evaluate the algorithm described in Chapter 4, which made use of purple trees (Definition 2), as opposed to the algorithm to solve the motion graph as described by Adler *et al.* [1].

These different version of the algorithm will most likely show significant differences in running time and output quality. We expect that the algorithm based on purple trees will provide a higher output quality at the cost of higher running time. However, we hope that the increase of running time will be relatively small to the output quality improves significantly.

Size of the interference forest We are also interested in the effect of the size of the interference forest on the running time of the algorithm. One can think of different cases in which the interference forest would contain one singular motion graph, or many smaller motion graphs, based on the width of some corridor in the workspace. We expect many smaller motion graphs (and thus a bigger interference forest) to be easier to solve due to the worse than linear complexity of the algorithm.

Specification All described tests are performed on an HP EliteBook 8560w with an Intel Core i7-2670QM processor. For geometric functions in the algorithm, the CGAL library (version 4.14.3 [24]) is used. For graph functions, the Boost Graph Library (version 1.65.1 [25]) is used.

5.1 Inputs

In this section we will look at the different experimental settings that were used to evaluate the algorithm and the implementation. Every data set is discussed, as well as its purpose and expected results.

Random workspace polygon To evaluate the effect of the number of robots in the input on the running time of the implementation, we look at a random polygon consisting of 1000 points (i.e. $n = 1000$). We generated 1000 random points inside a square using a point generator provided by Castro *et al.* [26]. These 1000 points are then used to construct a simple polygon using a function also provided by Castro *et al.* [26]. We ensured that the free space corresponding to this polygon consisted of a single component, such that comparing different data sets is easier.

The generated polygon is kept unaltered between tests while varying the number of robots. This ensures that an arbitrary feature that this polygon may have (i.e. a single versus multiple corresponding free space components) does not influence the test results.

The number of robots is varied between the values $m = \{1, 2, 3, 5, 8, 10, 20, 30, 50, 80, 100\}$. These values are chosen to give a decent view of the scalability of the program with respect to the number of robots in the input.

Square workspace polygon with robots in a grid Since a random polygon can still have features which bias certain parts of the algorithm or different edge weights, we will also look at varying the number of robots inside a square workspace polygon, such that the robots and their target configurations are ordered in a grid. It is clear that specific parts of the algorithm will be very efficient as n is very low (i.e. generating the free space components, which does not depend on m).

Due to the separation distance being at least 4, a path between any pair of configurations $u_i, u_j \in S \bigcup T$ exists. This implies that the motion graph will be relatively large, even though there will be only one single motion graph. We look at the differences between the results seen in this paragraph compared to the results that were seen in Section 5.1

To compute the length of the sides of the square workspace polygon, we computed the square root of two times the number of robots, rounded up. Figure 5.1 shows an example in which two times the number of robots ($2 \cdot 50 = 100$) has a nice square root. However, when this is not the case (e.g. 100 robots, thus $2 \cdot 100 = 200$), the grid containing start and target configurations will have some empty spaces (see for example Figure 5.2). These empty spaces are located in the middle of the workspace, between the set of starting configurations and the set of target configurations.

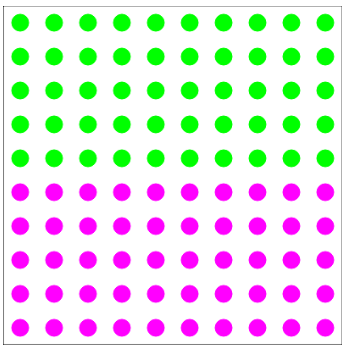


Figure 5.1: A grid setting with 50 start and target configuration (thus, a total of 100 configurations). With a separation distance of 4 units in mind, these configurations all fit in a square with sides of 40 units long.

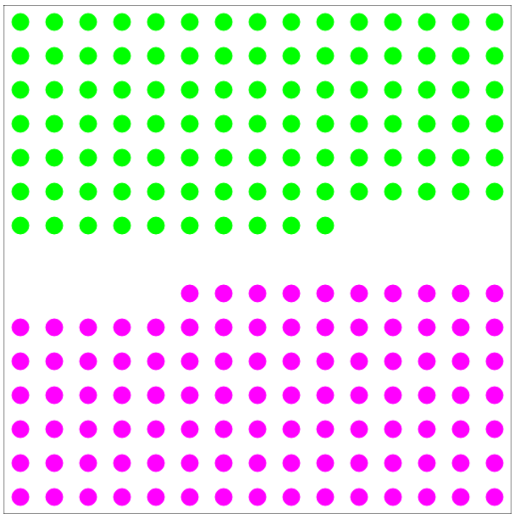


Figure 5.2: A grid setting with 100 start and target configuration (thus, a total of 200 configurations). With a separation distance of 4 units in mind, these configurations all fit in a square with sides of 60 units long.

Workspace polygon containing zig-zags To determine the effect of the complexity of the workspace polygon on the scalability of the algorithm, we consider a polygon with a fixed number of robots. The robots are again positioned like the robots in the grid (i.e. the starting configurations are at one side of the workspace, and the target configurations are at the other side of the workspace). However, as visible in Figure 5.3, the start and target configurations are now separated by a number of zig-zags in the workspace polygons.

If we define one zig-zag to be one cove on both sides of the workspace polygon, and one such cove consists of 3 points, then the complexity of the workspace can be defined as $n = 6 \cdot z + 6$ (mind the 6 workspace points that are not part of a cove), where z is the number of zigzags. The number of zigzags is varied between the values $z = \{4, 9, 19, 29, 39, 49, 59, 69, 79, 89, 99\}$. This means that the workspace complexity is varied between $n = \{30, 60, 120, 180, 240, 300, 360, 420, 480, 540, 600\}$.

This setting is used to determine the scalability of the algorithm with respect to the complexity of the workspace polygon. We expect to see this effect specifically on generating the free space components.

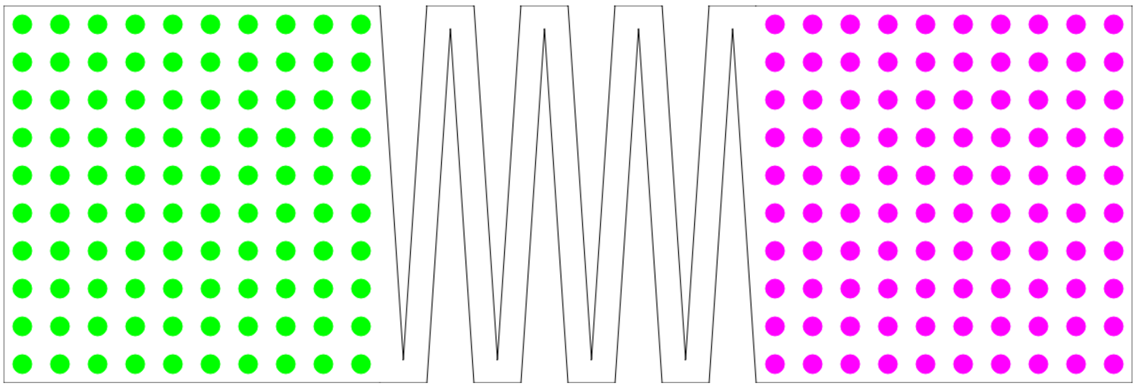


Figure 5.3: A zig-zag setting with 100 start and target configurations, which contains 4 zig-zags. The workspace complexity n is thus $6 \cdot 4 + 6 = 30$.

Comb-shaped workspace polygon As previously mentioned, there are multiple ways of determining the spanning tree used to solve the motion graph. The setting given in Figure 5.4 is used to show the worst case effect of choosing a bad edge weight (and thus a bad spanning tree).

Assume the following setting: Let h be the height of a tooth of the comb (i.e. from the top of the comb to the bottom of the tooth). Let e be the distance between two teeth of the comb. Let w be the width of a tooth, which is equal to the height of the base. The value of w should be at least 6 units, so every robot can pass another robot independently of the position of the other robot relative to the workspace. If w would be smaller than 6 units, since the diameter of a robot is 2 units, then $(w-2)/2 < 2$. This means that a robot could position itself in the middle of a corridor such that no robot would be able to pass it. Finally, let h be significantly larger than e (such that $w + e < h$ holds).

We can now look at the different minimum spanning trees created according to the different edge weights described in Chapter 5. The constant edge weight would still give us an arbitrary spanning tree, on which we cannot base the results too much. The Euclidean edge weight will give us a minimum spanning tree in which all configurations in the bottom of the teeth of the comb are connected. Note that $w + e < h$ implies that the Euclidean distance between these configurations in smaller than the Euclidean distance between two vertically aligned configurations. Also note that these edges are in the motion graph because there exists a path between those configurations since $w \geq 6$. The two connected components containing configurations that are created this way are finally connected by one single edge of length $h - w$. This results in a long detour for every robot, as they have to move along this single path connecting the start and target configurations. We will see that this is also a problem for the squared Euclidean edge weight, since $(w + e)^2 < h^2$ also holds because w, e and h are all positive numbers.

Finally, the geodesic edge weight will give us a minimum spanning tree that correctly identifies the comb structure and the edge weights between two configurations inside the teeth. This results in a minimum spanning tree in which every pair of start and target configuration which is vertically aligned (see Figure 5.4) is connected. All configurations in the base of the comb will also be connected pairwise to form a single minimum spanning tree.

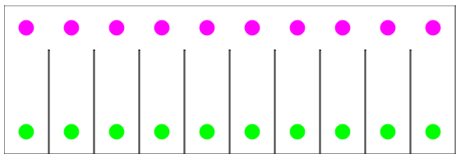


Figure 5.4: A comb setting with 10 teeth, and thus 10 starting and target configurations. Both the workspace complexity and the number of robots scale with the number of teeth the comb has.

The number of teeth in different tests is varied between $t = \{1, 2, 3, 5, 8, 10, 20, 30, 50, 80, 100\}$. The complexity of the workspace is $n = 4 \cdot t$ and the number of robots m is equal to t in this setting.

Corridor-shaped workspace polygon with small windows Finally, the last experimental setting is used to determine the effect of one large motion graph versus many small motion graphs on the running time of the algorithm. To test this, we look at the setting described in Figure 5.5.

Here, the polygon is shaped like a rectangle in which robots cannot pass each other (i.e. the width of the polygon is smaller than 4). At certain points, after a number of starting configurations followed by the same number of target configuration, a small window is placed in the workspace polygon. Two different versions of the workspace polygon are considered, namely one version in which the robots can pass through the windows and one version in which the robots cannot pass through the windows. When robots are not able to pass through the windows, multiple smaller free space components will be generated in the first step of the algorithm. When robots are able to pass through the windows, a single bigger free space component will be generated.

We generated the corridor workspace by starting at one end of the corridor. Here we placed m_i

starting configuration in a line, followed by m_i target configurations in a line. These configuration are all separated by 4 units. Then, a window is introduced in the workspace polygon (as visible in Figure 5.5). This process can then be repeated for m_{i+1}, starting from the previously introduced window. We generated this data set for $m_i \in \{2, 5, 9, 10, 4, 3, 7\}$. This means that the complexity of the workspace polygon is $n = 40$, and the number of robots is $m = 40$. This experiment was done using only this data set.

We expect to see the algorithm performing better when the free space consists of multiple components. Since the workspace complexity does not differ between the two versions of this setting, generating the free space components should be equivalent for both cases in terms of efficiency. However, the resulting free space consists of one component when the robots can move through the windows, while the free space consists of multiple components when the windows are too small. When the free space is split into multiple components like this, the complexity of every free space component is smaller. We suspect the running time of the algorithm to be worse than linear in the complexity of the free space components. Thus, we expect that running the algorithm on one free space component with a complexity of f is less efficient than running the algorithm on multiple free space components with a summed complexity f in most cases.

The test case used here is one long corridor containing small windows. When these windows are less than 2 units wide, robots can no longer pass through them. In the case of one long corridor, this will result in multiple motion graphs. These motion graphs will be connected in one line in the interference forest. However, this example could also be made with corners and junctions such that the interference forest would be a tree in which some nodes have more than 1 child node. Generating examples for such case would be harder, and solving the interference forest would be equally fast. This is due to the number of vertices and edges in the interference forest not changing. A handmade example of such setting is shown in Figure 5.6.

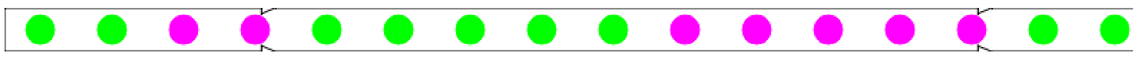


Figure 5.5: One end of a small corridor in which robots cannot move alongside each other. A few small windows are introduced in the workspace polygon. In this case, these windows are just too small for a robot to move through (e.g. 1.8 units wide). Another case would contain windows which are just big enough for robots to move through (e.g. 2.2 units wide). Note that since robots cannot move alongside each other, there is only one way to solve this setting, independent of the size of the windows.

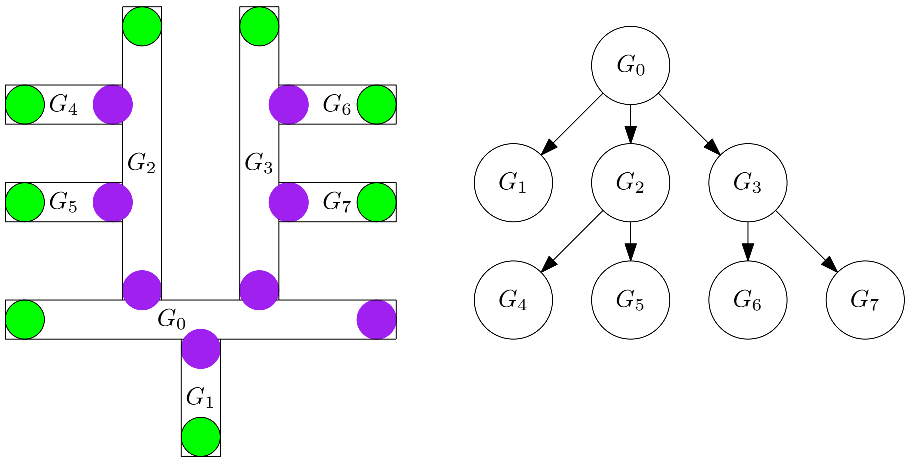


Figure 5.6: A floorplan-like setting where robots can not move from one room to another. The corresponding interference forest is given on the right. This example can be modified such that all doors are “open”, and robots can actually move between rooms. In this case, the interference forest would contain one single motion graph.

5.2 Results

This section contains the results of the conducted tests on all test cases. For every test case, the interesting results will be shown and explained.

Random workspace polygon The time complexity for generating the free space components only depends on the size and the shape of the workspace polygon. Therefore, there is no relation between the number of robots and the time it takes to generate the free space components.

Figure 5.7 shows us the time it takes to generate all motion graphs and combine them into a directed interference forest. The time complexity of this part of the algorithm looks to be worse than linear with respect to the number of robots. Also, note that the absolute time in seconds used by this part of the algorithm is significant compared to other parts of the algorithm.

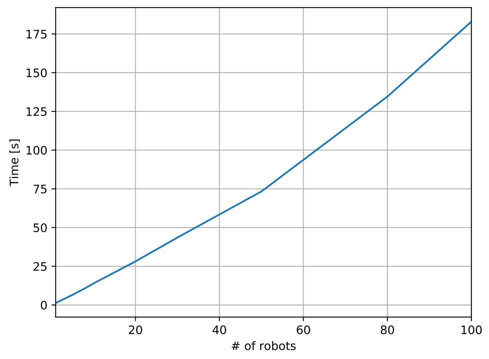


Figure 5.7: Time needed in seconds to generate all motion graphs and the directed interference forest in the random workspace polygon data set.

As expected, Figure 5.8a shows us that computing geodesic edge weights scales significantly worse than all the other methods of computing edge weights with respect to the number of robots. The complexity of computing all geodesic edge weights is, as previously discussed, $O(m \cdot (m + n) \log(m + n))$ which can be simplified to $O(m^2 \log m)$ since n is constant. However, it is hard to see the time complexity of computing the constant and Euclidean edge weights. According to Figure 5.8b it looks like both the constant and Euclidean edge weights can be computed in $O(m^2)$ time, albeit with a different constant factor. This can be explained by the fact that the number of edges in the motion graph is $|E| = O(|V|^2)$ and the number of vertices is $|V| = O(m)$.

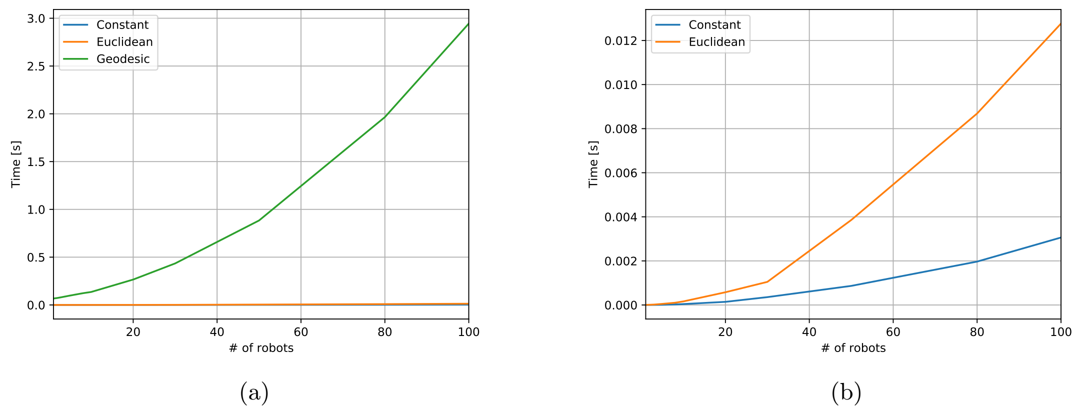


Figure 5.8: Time needed in seconds to compute the weights for all edges in every motion graph in the random workspace polygon data set.

However, Figure 5.9 shows us that choosing more sophisticated edge weights gives better running times for solving the motion graphs with respect to the number of robots. The running times of the part of the algorithm concerned with computing the edge weights are, however, more significant than the running times of this part of the algorithm. Thus, the combination of the running time needed to compute the edge weight with the running time needed to solve the interference forest still scales better for less sophisticated edge weight functions. Also note that the algorithm using the purple tree performs, in terms of scalability with respect to the number of robots, as good as the original algorithm.

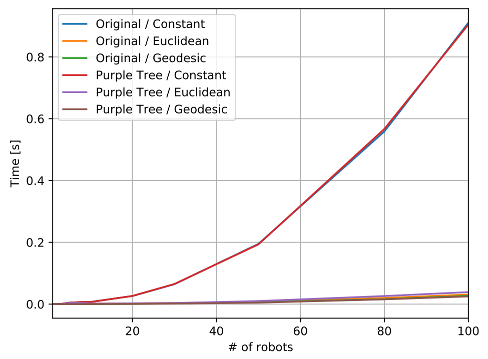


Figure 5.9: Time needed in seconds to solve every motion graph in topological ordering of the directed interference forest in the random workspace polygon data set.

In Figure 5.10, we can see that a significant part of the scalability of the implementation is influenced by the part which is concerned with generating the interference forest. Due to there only being one motion graph in this setting, and the interference forest is thus small, we can expect that generating the motion graph is the bottleneck in the scalability of the implementation.

Figure 5.11 shows a global trend in the sum of the distance covered by each robot for the different algorithms and edge weights. We can see that constant edge weights perform worse than an Euclidean edge weight function. Furthermore, the geodesic edge weight function performs better than its Euclidean counterpart.

The pebble motion algorithm as described by Adler *et al.* [1] produces a path along all configurations through which a pebble moves. Thus, the total distance covered by this algorithm is the sum of the distances of every segment on the path. Since this is not the shortest path, unless every starting configuration is directly connected to its corresponding target configuration in the spanning tree, the algorithm that uses purple trees to solve the motion graphs will report shorter total distances. Thus, the average distance covered per robot will also be smaller when using the algorithm that uses purple trees.

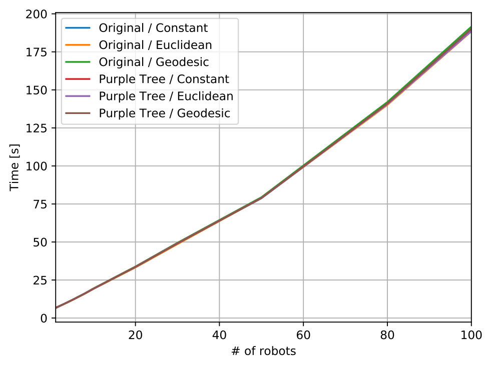


Figure 5.10: Time needed in seconds to run the complete algorithm on the random workspace polygon data set.

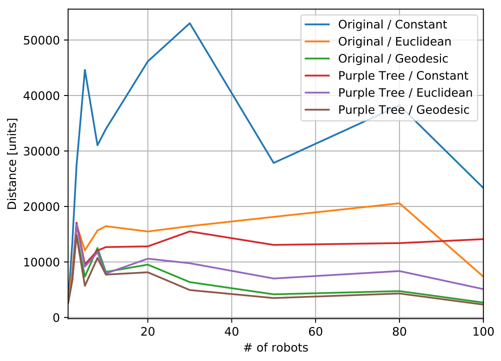


Figure 5.11: The average distance covered in units per robot in the random workspace polygon data set.

Square workspace polygon with robots in a grid The square workspace polygon with robots in a grid setting is used to show the relation between the number of robots and the time-complexity of the algorithm and the implementation. This data set has a workspace polygon which only consists of 4 points.

As previously mentioned, the time complexity of generating the free space components only depends on the complexity of the workspace polygon. Therefore, there is again no relation between the different data sets in this setting and the time it takes to generate the free space components.

Figure 5.12 shows the time it takes to generate all motion graphs, and connect them in a directed interference forest. Here, we can see a non-linear relation between the number of robots and the time it takes to generate the interference forest. This relation was not visible in the previous data set, because the size of the workspace polygon, which was significantly larger in the previous data set, influences the time it takes to generate the interference forest.

Figure 5.13 and Figure 5.14 show results that are comparable with the previous data set. The time complexity seems to scale in the same way as was the case with the random polygon. However, the rate with which they scale differ. This is due to a different complexity of the workspace polygon.

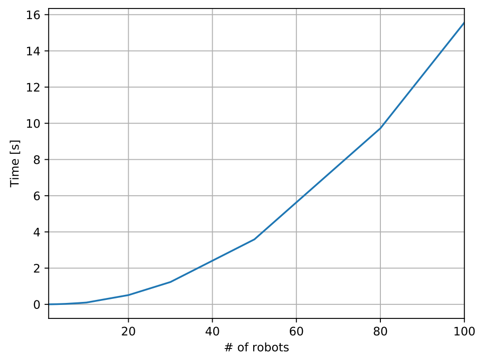


Figure 5.12: Time needed in seconds to generate all motion graphs and the directed interference forest in the grid data set.

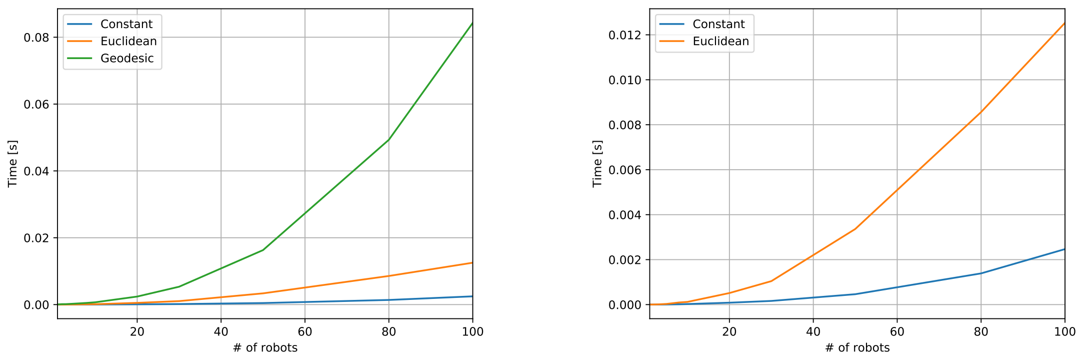


Figure 5.13: Time needed in seconds to compute the weight for all edges in every motion graph in the grid data set.

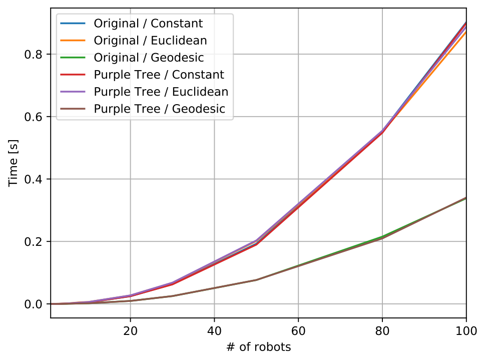


Figure 5.14: Time needed in seconds to solve every motion graph in topological ordering of the directed interference forest in the grid data set.

In Figure 5.15, we can see that the quality of the output schedule is the worse when using a geodesic or Euclidean edge weight function (note that geodesic distance inside a square polygon is equal to the Euclidean distance). This can be explained by reasoning on the set of start and target configurations. Due to the grid formation of all configurations, there are only a few locations on the grid where a starting configuration is positioned next to a target configuration. Moreover, there are cases in which there is no position in which a starting and a target configuration are located next to each other. Such example can be seen in Figure 5.2.

This example shows how the geodesic and Euclidean edge weight would ensure that the minimum spanning tree contains only one edge that connects a starting configuration with a target configuration. Though, when using constant edge weight, random edges are selected to be put in the minimum spanning tree. Thus, it is very likely that more than 1 edge exists which connects a starting configuration with a target configuration.

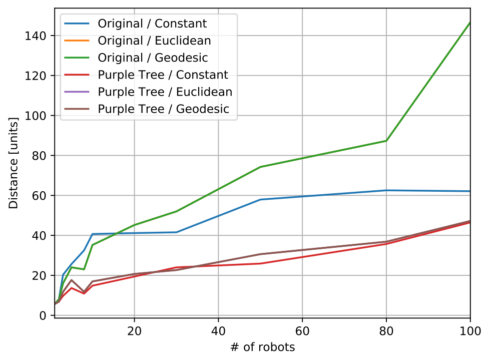


Figure 5.15: The average distance covered in units per robot in the grid data set.

Workspace polygon containing zig-zags This data set is used to show the relation between the number of points in the workspace polygon and the complexity of the algorithm and the implementation, while the number of robots is kept constant.

According to Adler *et al.* [1], the free space components can be computed in $O(n \log n)$ time. In Figure 5.16, we can see the time it took to generate the free space components. The scaling that we see in this figure is indeed a bit worse than linear.

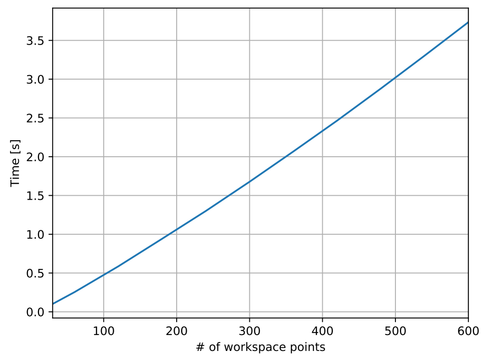


Figure 5.16: Time needed in seconds to generate the free space components in the zig-zag polygon data set.

Figure 5.17 shows that the time complexity of generating the motion graph scales a bit worse than linear with respect to the complexity of the workspace polygon. However, the time needed to generate the interference forest again shows to be significantly larger than other components of the algorithm.

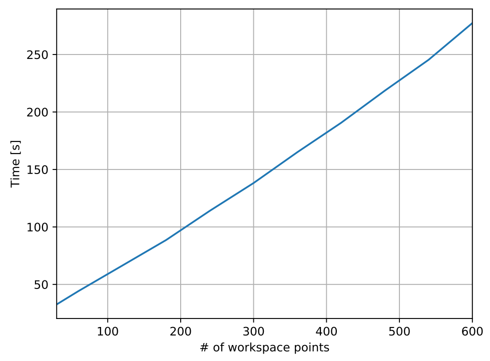


Figure 5.17: Time needed in seconds to generate all motion graphs and the directed interference forest in the zig-zag polygon data set.

The time needed to compute the edge weights for all edges in the motion graphs is shown in Figure 5.18. Since the number of vertices in the motion graph does not scale with the complexity of the workspace polygon, and the number of edges still being $|E| = O(|V|^2)$ in this data set, the number of times the edge weights are computed also does not scale with the complexity of the workspace polygon. This implies that the constant and Euclidean edge weights can be computed in constant time with respect to the complexity of the workspace. However, computing the geodesic edge weights still has a time complexity of $O(n \log n)$ where n is the complexity of the workspace polygon.

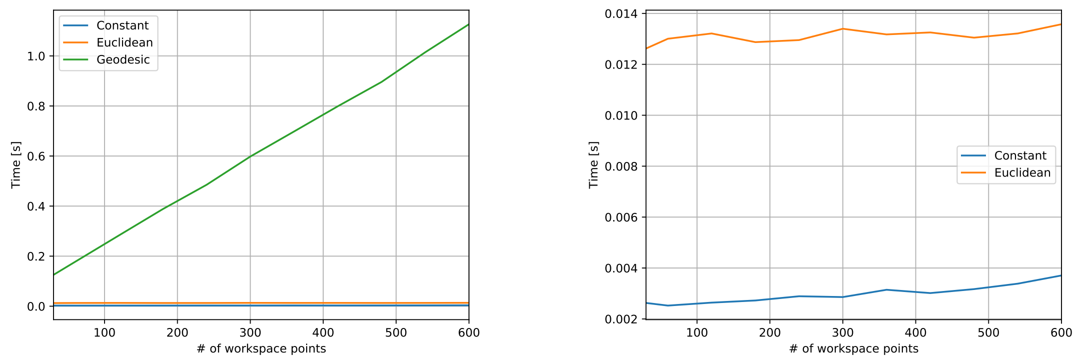


Figure 5.18: Time needed in seconds to compute the weights for all edges in every motion graph in the zig-zag polygon data set.

After computing the edge weights (or in the case of constant or Euclidean edge weights, after generating the interference forest), the workspace polygon is no longer used. Also, the size of the motion graphs or the interference forest is not influenced by the complexity of the workspace polygon. Therefore, the time it takes to solve the motion graphs in order according to the directed interference forest is not influenced by the complexity of the workspace polygon.

The absolute time needed to generate the interference forest is again significant with respect to the time needed for different components of the algorithm. Moreover, in this data set, due to the shape of the zig-zag polygon, the total distance covered by all robots is also heavily influenced by the number of zig-zags the workspace polygon contains.

Comb-shaped workspace polygon The results of the comb-shaped workspace polygon setting are given with respect to the number of teeth the comb has. In every tooth, one start configuration and one target configuration is placed, and the workspace complexity is increased by 4 per tooth. This means that the complexity of the workspace polygon is $n = O(t)$ and the number of starting and target configurations is $m = O(t)$, where t is the number of teeth in the comb.

Since the number of points in the workspace scales linearly with the number of teeth, the time complexity of generating the free space components, which can be seen in Figure 5.19, is equivalent to the complexity that we have seen in the previous paragraph.

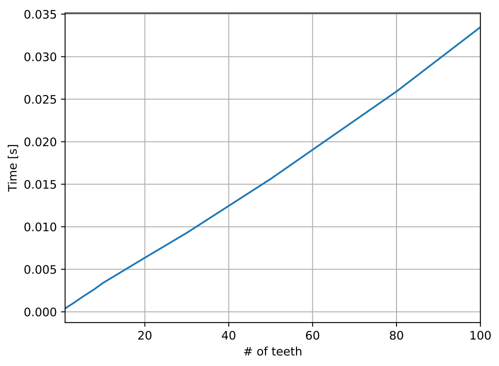


Figure 5.19: Time needed in seconds to generate the free space components in the comb-shaped workspace polygon.

Because the number of teeth in this data set scales linearly with the number of robots, we can expect a worse than linear scaling of the time complexity of generating the interference forest with respect to the number of teeth. This worse than linear scaling is visible in Figure 5.20.

Figure 5.21 shows the time needed to compute the edge weights for all edges in the motion graph. Since both the complexity of the workspace polygon and the number of starting and target configurations scale linearly with the number of teeth in this data set, the time complexity of computing all geodesic edge weights is $O(t \cdot (t + t) \log (t + t)) = O(t \cdot t \log t) = O(t^2 \log t)$. Computing the constant and Euclidean edge weights again takes $O(1)$ time for every edge in the motion graph. Since $m = O(t)$, the number of edges in the motion graph can be given as $O(t^2)$ in this data set. Thus, the complexity of computing all edge weights for the constant and Euclidean cases are $O(t^2)$.

The comb-shaped workspace polygon is engineered to show an example in which the geodesic edge weights show a significant improvement in terms of distance covered over the other edge weight functions. Figure 5.22 and Figure 5.23 show how the geodesic edge weights indeed perform better in terms of time needed to solve the interference forest as well as the total distance covered by all robots, respectively. Also note that, in this case, the Euclidean edge weights perform worse than the constant edge weights.

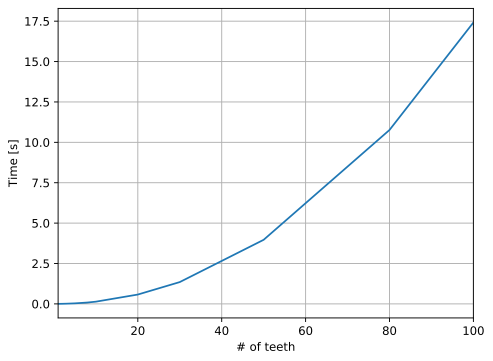


Figure 5.20: Time needed in seconds to generate all motion graphs and the directed interference forest.

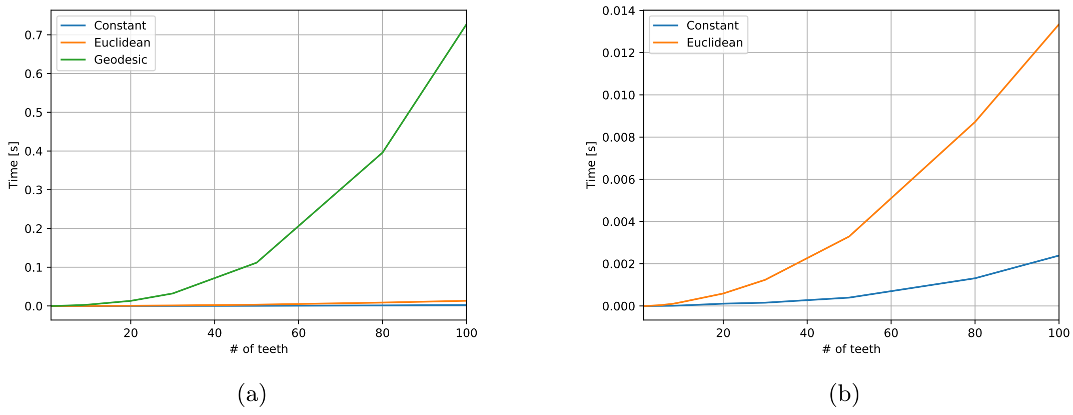


Figure 5.21: Time needed in seconds to compute the weights for all edges in every motion graph in the comb-shaped polygon data set.

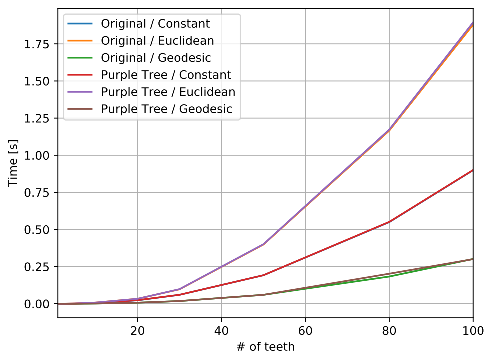


Figure 5.22: Time needed in seconds to solve all motion graphs in topological ordering of the directed interference forest in the comb-shaped polygon data set.

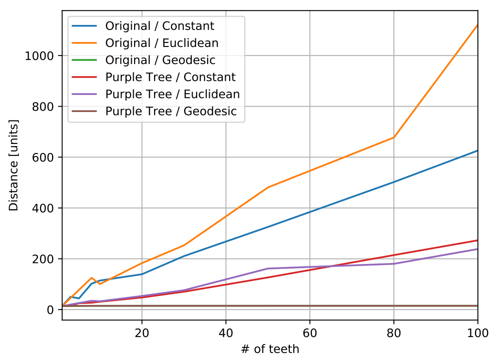


Figure 5.23: The average distance covered in units per robot in the comb-shaped workspace polygon data set.

Corridor-shaped workspace polygon with small windows The results of the corridor-shaped workspace polygon data set are compared in a different way. The bars with labeled "Single" represent results for the case in which the workspace polygon had windows through which robots are able to move. This means that a single free space component is generated in the first step of the algorithm. The bars with labeled "Multi" represent results for the case in which the workspace polygon had windows through which robots are not able to move. This means that multiple free space components are generated in the first step of the algorithm. Since the complexity of the workspace polygons is equal for both the "Single" and "Multi" case, there is no difference in time complexity for generating the free space components. In the different settings, the number of robots and the complexity of the workspace polygon remain the same.

In Figure 5.24, we can see that generating the motion graphs and the interference forest is faster when considering multiple smaller free space components than when considering one big free space component, knowing that the total number of vertices in the motion graphs in both cases is equal. In the previous data sets, we have seen that the time complexity of generating the motion graphs and interference forest scaled worse than linear with respect to the number of robots. Note that the number of robots in a free space component corresponds with the number of vertices in the corresponding motion graph. This means that the result that we see here was expected.

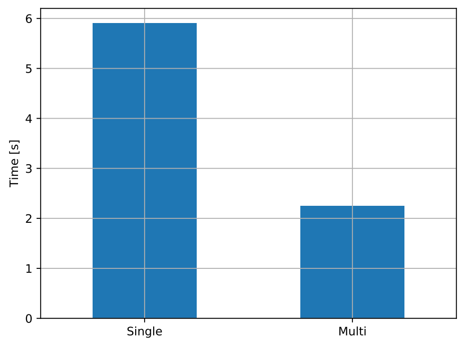


Figure 5.24: Total time needed in seconds to generate all motion graphs and the directed interference forest in the corridor-shaped workspace data set.

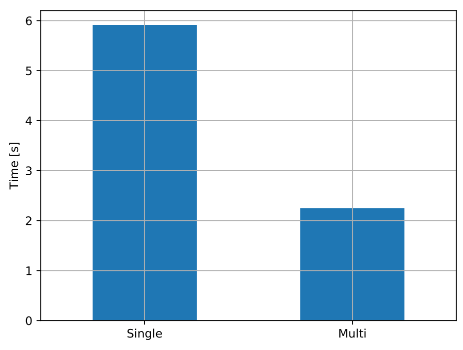


Figure 5.25: Time needed in seconds to solve all motion graphs in order according to the directed interference forest in the corridor-shaped workspace data set.

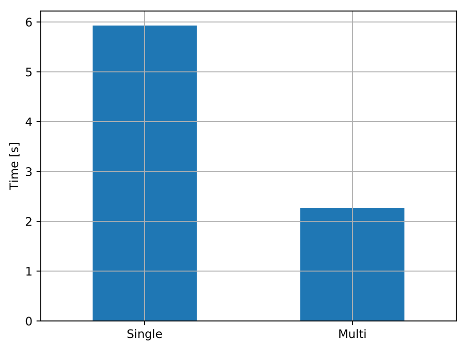


Figure 5.26: Time needed in seconds to run the complete algorithm in the corridor-shaped workspace data set.

As we can see in Figure 5.25, solving the multiple smaller motion graphs according to the directed interference forest is faster than solving the single bigger motion graph.

In Figure 5.26 we can again see that generating the motion graphs and the interference forest takes the most significant amount of time.

5.3 Conclusions

Adler *et al.* [1] stated that generating the free space components takes $O(n \log n)$ time, where n is the complexity of the workspace. This means that this part of the algorithm does not scale in terms of the number of robots, as we discussed in the random workspace polygon data set. The results that we are shown in Figure 5.16 (from the zig-zag data set) and Figure 5.19 (from the comb-shaped workspace data set) agree with a worse than linear scaling in terms of the complexity of the workspace polygon. However, it is hard to exactly determine the time complexity based on the results.

Generating the interference forest showed to be the part of the algorithm which scales the worst. In Figure 5.17 (from the zig-zag data set), a worse than linear scaling with respect to the complexity of the workspace polygon is shown. However, the scaling seems to be even worse in terms of the number of robots, as seen in Figure 5.12 (from the grid data set). We have seen that the scaling in Figure 5.7 (from the random workspace data set) is different from the scaling in

Figure 5.12 (from the grid data set), even with the same number of robots. However, because the complexity of the workspace polygon differs between those data sets, we can combine these results to conclude that there is also a factor in the time complexity of this part of the algorithm which depends on both the complexity of the workspace and the number of robots.

The time complexity of computing the edge weights is different per function used. The constant and Euclidean edge weight functions take a constant amount of time per edge for which the weight is computed. The motion graphs that were used in our examples were complete graphs. This means that the number of edges in these motion graphs is $|E| = O(|V|^2)$. Since our examples had only one motion graph, the number of vertices in this motion graph was $|V| = O(m)$, where m is the number of robots in the data set. Thus, computing the constant and Euclidean edge weights for our motion graphs took $O(m^2)$ time. Note that the constant edge weights can be computed a constant factor faster than the Euclidean edge weights.

Computing the geodesic edge weights took more time, and was depending on both the complexity of the workspace and the number of robots. The results that are shown in Section 5.2 seem to agree with the time complexity that we discussed in the beginning of this chapter.

Adler *et al.* show that a solution for a single motion graph G with a vertex set $S \bigcup T$ where $|S| = |T|$ can be found in $O(|S|^2)$ time. Again, note that the cases we discussed all consist of a single motion graph G, which implies that all m starting configurations and target configurations are represented in G. Thus $|S| = |T| = m$, and a solution for G can be found in $O(m^2)$ time. The results that we have seen in Section 5.2 seem to agree with a quadratic time complexity. Though, the different edge weights have an effect on the rate at which the running time grows.

Unfortunately, a big part of the total running time was used to generate the interference forest. This showed in every data set.

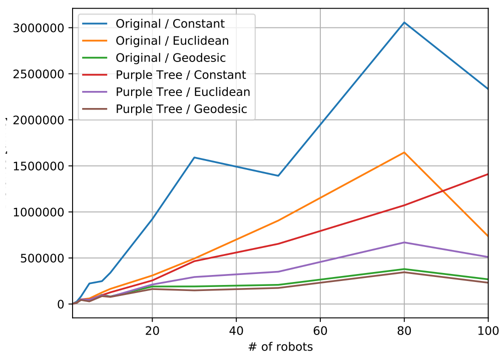
Bottleneck The bottleneck in the implementation of this algorithm is generating the motion graphs and the interference forest. This part of the algorithm takes a significant amount of time compared to all other steps of the algorithm. However, Adler *et al.* show that the full motion graph does not have to be computed first. Their method is described in Section 4.1 in detail.

We wanted to discover results for different spanning trees. Using the method by Adler *et al.*, we were not able to generate the spanning trees that we wanted to evaluate. Thus, we decided to compute the whole motion graph, which turned out to take a significant amount of time compared to other parts of the algorithm.

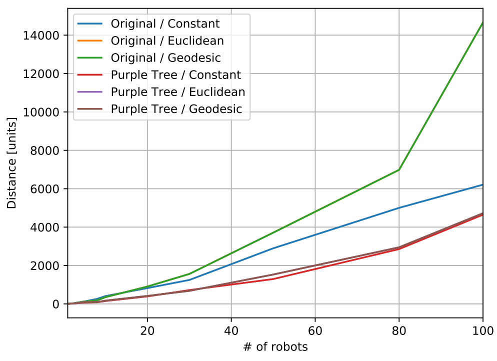
Purple tree In our experiments, the algorithm which uses purple trees performs better than the original solution for the pebble motion problem in terms of the covered distance per robot.

The covered distance per robot is often better when using the purple tree algorithm than when using the original algorithm. Figure 5.27 shows the difference in total covered distance. In every different setting, the purple tree algorithm shows to have a lower total covered distance. This is mostly due to the way the motion schedule is generated. Using the original algorithm, the motion schedule contains a path through many configurations for every robots. However, the purple tree algorithm generates a motion schedule in which every robot is activated once, and moves towards its target configuration in the shortest possible way.

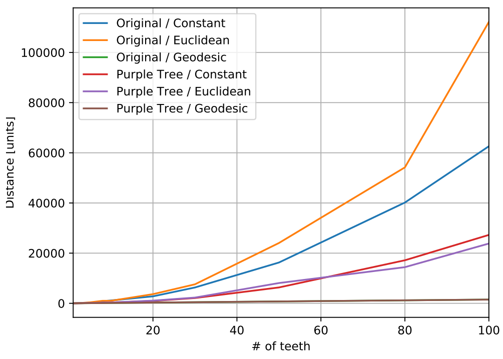
In terms of the number of times a robot has to be activated, the algorithm which uses purple trees is indeed optimal. This is visible in Table 5.1, which shows the total times any robot becomes activated in the motion schedule for the random polygon data set. These results correspond with results for all different data sets.



(a) The average distance covered in units per robot in the random polygon data set.



(b) The average distance covered in units per robot in the square polygon data set.



(c) The average distance covered in units per robot in the comb polygon data set.

Figure 5.27: A comparison between different data sets on the covered distance in units per robot.

	Original			Purple Tree		
	Constant	Euclidean	Geodesic	Constant	Euclidean	Geodesic
m						
1	1	1	1	1	1	1
2	4	3	2	2	2	2
3	9	3	4	3	3	3
5	12	7	7	5	5	5
8	29	22	24	8	8	8
10	36	18	23	10	10	10
20	55	65	73	20	20	20
30	79	96	90	30	30	30
50	121	255	150	50	50	50
80	182	654	445	80	80	80
100	223	349	336	100	100	100

Table 5.1: Total number of times a robot is activated in the random polygon data set

Chapter 6

Conclusions

In this thesis, we studied strategies of improving the algorithm devised by Adler *et al.* [1] in different ways. Adler *et al.* showed that, under certain conditions, the multi-robot motion planning problem can be solved efficiently in terms of running time. We discussed a way of extending the algorithm to the case of workspace polygons with holes. We also devised an algorithm to solve the pebble motion problem using purple trees. Finally, we evaluated the algorithm and its implementation experimentally, by making use of different ways of using the motion graph.

First, in Chapter 3, we have shown that, if we allow infinitely small holes, simultaneous movement of robots might be necessary. We also conjectured that, if holes in the workspace polygon are not infinitely small, there is a solution in which no simultaneous movement of robots is necessary.

The algorithm by Adler *et al.* is not optimized in terms of the quality of the motion schedule. One clear way of optimizing the quality of the motion schedule that this algorithm produces, is by optimizing the spanning tree that is used to solve the motion graphs. However, we have seen in Chapter 5 that there is no perfect solution for every setting, and that computing the whole motion graph is inefficient. Adler *et al.* described a method in which the motion graph is not computed explicitly, but this method was not optimal, nor was it easy to optimize. In general, using the geodesic distance is a good way of optimizing the distance covered by the robots when moving according to the motion schedule. Computing the Euclidean edge weights is only a constant factor slower than using constant edge weights, and generally performs better than the constant edge weights. However, the data set shown in Figure 5.1 made clear that there exist cases in which an arbitrary spanning tree performs better.

Another improvement to the algorithm can be made in the way of solving the motion graph. The way of solving this motion graph was originally done by using the pebble motion problem. However, we have seen a different solution using purples trees (Lemma 1). This solution optimizes the number of times a robot needs to be activated. Chapter 5 showed us that often, the total covered distance is also smaller using this solution. This is due to every robot taking the shortest possible path from its corresponding starting configuration to its corresponding target configuration, without moving to intermediary configurations.

We experimentally evaluated the algorithm and its implementation in Chapter 5. The time complexity that we have seen in these experiments seems to agree with the theoretical complexity that Adler *et al.* discussed.

6.1 Future work

We looked at workspace polygons that contained holes. We proposed an idea to solve the multi-robot motion planning problem for such workspace polygons by adding intermediate configurations which do not interfere with other free space components. However, it was also discussed that this idea would not work when holes have an area of zero units. The conjecture given in Chapter 3 might be an interesting idea to explore further.

Since free space components in our setting contained circular arcs (due to the robots being discs), the free space components also contained circular arcs on their boundaries. To compute the geodesic distance between two points in such free space components, the circular arcs on the boundary of a free space component were approximated. The resulting approximated free space component was then used to determine the geodesic distance. This could be improved to use the exact geodesic distance, which could also take into account other robots instead of only the free space component.

Finally, in Chapter 4 we discussed an idea to solve the problem without using (minimum) spanning trees. It might be possible to find an algorithm which is able to efficiently find a vertex $v \in V_i$ in a motion graph $G_i = (V_i, E_i)$ such that either removing v does not disconnect G_i, or removing v splits G_i into multiple smaller motion graphs which are still solvable.

Bibliography

[1] A. Adler, M. De Berg, D. Halperin and K. Solovey, ‘Efficient multi-robot motion planning for unlabeled discs in simple polygons’, in *Algorithmic Foundations of Robotics XI*, Springer, 2015, pp. 1–17.

[2] K. Solovey and D. Halperin, ‘On the hardness of unlabeled multi-robot motion planning’, *The International Journal of Robotics Research*, vol. 35, no. 14, pp. 1750–1759, 2016.

[3] G. Bottesi, J.-P. Laumond and S. Fleury, ‘A motion planning based video game’, *Technical Report 04576, LAAS-CNRS*, 2004.

[4] D. Thalmann, H. Grillon, J. Maim and B. Yersin, ‘Challenges in crowd simulation’, in *2009 International Conference on CyberWorlds*, IEEE, 2009, pp. 1–12.

[5] J. T. Schwartz and M. Sharir, ‘On the “piano movers” problem. ii. general techniques for computing topological properties of real algebraic manifolds’, *Advances in applied Mathematics*, vol. 4, no. 3, pp. 298–351, 1983.

[6] J. E. Hopcroft, J. T. Schwartz and M. Sharir, ‘On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman’s problem"’, *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[7] P. Spirakis and C. K. Yap, ‘Strong np-hardness of moving many discs’, *Information Processing Letters*, vol. 19, no. 1, pp. 55–59, 1984.

[8] J. T. Schwartz and M. 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*, vol. 2, no. 3, pp. 46–75, 1983.

[9] C. Yap, *Coordinating the motion of several discs*, English (US), ser. Robotics Report 16. Department of Computer Science, New York University, Feb. 1984.

[10] M. Sharir and S. Sifrony, ‘Coordinated motion planning for two independent robots’, *Annals of Mathematics and Artificial Intelligence*, vol. 3, no. 1, pp. 107–130, 1991.

[11] M. Erdmann and T. Lozano-Perez, ‘On multiple moving objects’, *Algorithmica*, vol. 2, no. 1-4, p. 477, 1987.

[12] B. Aronov, M. de Berg, A. F. van der Stappen, P. Švestka and J. Vleugels, ‘Motion planning for multiple robots’, *Discrete & Computational Geometry*, vol. 22, no. 4, pp. 505–525, 1999.

[13] K. Solovey, ‘Multi-robot motion planning: Theory and practice’, PhD dissertation, Tel Aviv University, 2018.

[14] D. Kornhauser, G. Miller and P. Spiralris, ‘Coordinating pebble motion on graphs, the diameter of permutation groups, and applications’, 1984.

[15] V. Auletta, A. Monti, M. Parente and P. Persiano, ‘A linear-time algorithm for the feasibility of pebble motion on trees’, *Algorithmica*, vol. 23, no. 3, pp. 223–245, 1999.

[16] R. Luna and K. E. Bekris, ‘An efficient and complete approach for cooperative path-finding’, in *Twenty-fifth AAAI conference on artificial intelligence*, 2011.

[17] J. Yu and S. M. LaValle, ‘Planning optimal paths for multiple robots on graphs’, in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 3612–3617.

[18] K. Solovey and D. Halperin, ‘K-color multi-robot motion planning’, *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 82–97, 2014.

[19] M. Turpin, N. Michael and V. Kumar, ‘Concurrent assignment and planning of trajectories for large teams of interchangeable robots’, in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 842–848.

[20] K. Solovey, J. Yu, O. Zamir and D. Halperin, ‘Motion planning for unlabeled discs with optimality guarantees’, *arXiv preprint arXiv:1504.05218*, 2015.

[21] J. B. Kruskal, ‘On the shortest spanning subtree of a graph and the traveling salesman problem’, *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[22] R. C. Prim, ‘Shortest connection networks and some generalizations’, *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[23] R. Wein, A. Baram, E. Flato, E. Fogel, M. Hemmer and S. Morr, ‘2D minkowski sums’, in *CGAL User and Reference Manual*, 4.14, CGAL Editorial Board, 2019. [Online]. Available: `https://doc.cgal.org/4.14/Manual/packages.html#PkgMinkowskiSum2`.

[24] The CGAL Project, *CGAL User and Reference Manual*, 4.14. CGAL Editorial Board, 2019. [Online]. Available: `https://doc.cgal.org/4.14/Manual/packages.html`.

[25] J. G. Siek, L.-Q. Lee and A. Lumsdaine, *Boost Graph Library, The: User Guide and Reference Manual*. Addison-Wesley Professional, 2001. [Online]. Available: `https://www.boost.org/doc/libs/1_65_1/libs/graph`.

[26] P. M. M. de Castro, O. Devillers, S. Hert, M. Hoffmann, L. Kettner, S. Schönherr, A. Tifrea and M. Gimeno, ‘Geometric object generators’, in *CGAL User and Reference Manual*, 4.14, CGAL Editorial Board, 2019. [Online]. Available: `https://doc.cgal.org/4.14/Manual/packages.html#PkgGenerators`.