Bar-Ilan University
Department of Computer Science

ROBUST AND EFFICIENT MULTI-ROBOT COVERAGE

by

Noam Hazon

Submitted in partial fulfillment of the requirements for the Master's degree
in the department of Computer Science

Ramat-Gan, Israel
October 2005

This work was carried out under the supervision of

Dr. Gal A. Kaminka

Department of Computer Science, Bar-Ilan University.

Abstract

Area coverage is an important task for mobile robots, with many real-world applications. Motivated by potential efficiency and robustness improvements, there is growing interest in the use of multiple robots in coverage. Previous investigations of multi-robot coverage algorithms focused on the improved efficiency gained from the use of multiple robots, but did not formally addressed the potential for greater robustness. We address robustness and efficiency in a family of multi-robot coverage algorithms, based on spanning-tree coverage of approximate cell decomposition. We present off-line and on-line algorithms and analytically show that the algorithms are complete and robust, in that as long as a single robot is able to move, the coverage will be completed. We analyze the assumptions underlying the algorithms requirements and present a number of techniques for executing it in real robots. We show extensive empirical coverage-time results of running the algorithms in a number of different environments and several group sizes.

Acknowledgments

Special gratitude to Dr. Gal A. Kaminka for advising my research. He succeeded to teach me some basic principles and methods of research which can not be found in any textbook.

I would like to thank Dr. Moshe Lewenstein for his help in developing the Optimal Backtracking MSTC algorithm; Fabrizio Mieli, for his help in evaluating the On-Line MSTC algorithm with the simulated environment; and Noa Agmon-Segal, for her contribution of the approximation algorithm for efficient spanning tree. I would also like to thank the MAVERICK group at the Computer Science department, Bar Ilan University, for their support. It was obviously a great research environment.

I wish to express my gratitude to my parents, Shmuel and Tzipi, for their support and encouragement of my academic education.

Further, I want to thank my beloved wife Shira for standing by me throughout the long way, which sometimes required some concessions from her side.

Finally, thanks to G-d.

Contents

List of Figures

Chapter 1

Introduction

Area coverage is an important task for mobile robots, with many real-world applications such as floor cleaning [6], lawn mowing [15], de-mining [19], harvesting [20], painting, and hazardous waste cleaning [13]. In these, a robot is given a bounded work-area, possibly containing obstacles. The robot is assumed to have an associated *tool* of a given shape [9]—often corresponding to the robot's relevant sensors' and/or actuator range—that must visit every point within the work-area. Since the tool size is typically much smaller than the work-area, the robot's task consists of finding and moving along a path that will take the tool over the entire work-area. This is sometimes referred to as exhaustive geographical search [24], or sweeping [10].

In recent years, there is growing interest in the use of multiple robots in coverage, motivated by efficiency and robustness. First, multiple robots may complete the task more quickly than a single robot, by dividing the work-area between them. Second, multi-robot algorithms may succeed in face of failures, since even if a robot fails, its peers might still be able to cover its assigned area. Formally, a coverage algorithm is said to be *complete* if, for any work-area, it produces a path that completely covers the work-area. We want multi-robot algorithms to be not only complete, but also *efficient* (in that they minimize the time it takes to cover the area), and *robust* (in that they can handle catastrophic robot failures). We may additionally want the algorithm to be *non-redundant* (*non-backtracking*), in that any portion of the work area is covered only once.

Previous investigations that examine the use of multiple robots in coverage mostly focus on completeness and non-redundancy. However, much of previous work does not formally consider robustness. Moreover, while completeness and non-backtracking properties are sufficient to show that a single-robot coverage algorithm is also efficient (in coverage time), it turns out that this is not true in the general case. Surprisingly, in multi-robot coverage, non-backtracking and efficiency are independent optimization criteria: Non-redundancy algorithms may be inefficient, and efficient algorithms may use backtracking. Finally, the initial position of robots in the work-area significantly affects the completion time of the coverage, both in backtracking and non-backtracking algorithms. Yet no bounds are known for the coverage completion time, as a function of the number of robots and their initial placement.

This thesis examines robustness and efficiency in multi-robot coverage. We first focus on coverage

using a map of the work-area, sometimes referred to as *off-line coverage* [4]. We assume the tool to be a square of size D. The work-area is then approximately decomposed into cells, where each cell is a square of size $4D$, i.e., a square of four tool-size sub-cells. As with other approximate cell-decomposition approaches ([4]), cells that are partially covered by obstacles, or the bounds of the work-area, are discarded from consideration. We use an algorithm based on a spanning-tree to extract a path that visits all sub-cells (i.e., it is complete). Previous work developed algorithms for generating used such a path (called *STC* for Spanning-Tree Coverage) for single-robot coverage and showed it to be complete and non-backtracking [9].

We present a family of novel algorithms, called MSTC (*Multirobot Spanning-Tree Coverage*) that addresses these challenges. First, we construct a non-backtracking MSTC algorithm that is guaranteed to be *robust*: It guarantees that the work-area will be completely covered in finite time, as long as at least a single robot is functioning correctly. We analyze the best-case and worst-case completion times for this algorithm, and find that in the worst-case, the coverage time for k robots is essentially equal to that of a single robot. Unfortunately, this worst-case scenario is common in coverage applications: This is where robots all start from approximately the same position (e.g., doorway to the work-area). We further prove that this result holds for any non-backtracking algorithm that uses STC paths.

We then present a second robust MSTC algorithm, which allows for some simple backtracking: It may have a robot visit a cell twice, but no more. We show that surprisingly, even though this algorithm involves backtracking, its worst-case coverage time for $k > 2$ robots is half that of a single robot. These results show that coverage algorithms must distinguish between non-backtracking and efficiency properties. These two criteria converge only in the single-robot case, but are distinct (and may be mutually exclusive) in the general k-robot case.

The simple backtracking MSTC algorithm guarantees a better worst-case coverage time than the non-backtracking algorithm, but it does not generate an optimal allocation of robots to assigned trajectories and thus its average performance can be improved. We present the third robust MSTC algorithm, optimal backtracking MSTC, which achieves the best time for a given initial configuration. This algorithm has another useful advantage: it continues to provide an optimal solution even if it runs on heterogenous robots with different speeds or different amounts of fuel.

We then focus on coverage without using a map of the work-area, sometimes referred to as *on-line coverage* [4]. Some applications lend themselves easily to *off-line* coverage, where the robots are given a map of the work-area, and can therefore plan their paths ahead of deployment. For instance, in many outdoor operations, aerial photos or maps might be available. However, many applications must utilize *on-line* coverage algorithms. Here, the robots cannot rely on apriori knowledge of the work-area, and must construct their movement trajectories step-by-step, addressing discovered obstacles (and/or collisions, in the case of multiple robots) as they move.

We present a *guaranteed robust* multi-robot on-line coverage algorithm, called ORMSTC (*On-line Robust Multi-robot Spanning-Tree Coverage*). The algorithm is based on the use of spanning tree coverage paths [9]. It runs in a distributed fashion, using communications to alert robots to the positions

of their peers. Each robot works within a dynamically-growing portion of the work-area, constructing a local spanning-tree covering this portion, as it moves. It maintains knowledge of where this spanning-tree can connect with those of others, and selects connections that will allow it to take over the local spanning trees of others, should they fail.

We also address the challenge of using ORMSTC algorithm with physical vacuum cleaning robots. We present techniques useful in approximating the assumptions required by STC algorithms (e.g., known positions, within an agreed-upon coordinate system) to allow them to work in real world situations. We then show the effectiveness of our implemented ORMSTC algorithm in extensive experiments.

Chapter 2

Background and Related work

Choset [4] provides a recent survey of coverage algorithms, which, while mostly focused on single-robot settings, distinguishes important classes of coverage algorithms. First, the survey distinguishes between *offline* algorithms, in which a map of the work-area is given to the robots, and *online* algorithms, in which no map is given. The *online* algorithms are also divided to two sub-types: The regular approach, which remembers the area that was covered thus requires a lot of memory, and an ant-like approach which uses marks on the covered area to overcome this problem.

To achieve *completeness* many coverage algorithms use a cellular decomposition of the free space. A cellular decomposition breaks down the target region into cells such that coverage in each cell is a simple task. Provably complete coverage is attained by ensuring the robot visits each cell in the decomposition. Choset [4] distinguishes between three types of work-area decompositions: *Approximate*, *semi-approximate* and *exact*.

In the approximate cellular decomposition, first introduced by Moravac and Elfes [18], the work-area is divided to cells which are all the same size and shape. Cells that are partially covered by obstacles, or the bounds of the work-area, are discarded from consideration thus the union of the cells to cover only approximate the target region. Typically the cell size is determined by the size of the robot's tool or sensor range. Zelinsky et al. [29] uses the regular wavefront algorithm to determine a coverage path for a single-robot. Our algorithms are build on the single-robot STC (Spanning-Tree Coverage) family of algorithms[9] which also uses the approximate cellular decomposition.

Hert et al. [14] introduced the semi-approximate cellular decomposition approach. In that the area is decomposed to fixed width cells but with varying top and button.

In the exact cellular decomposition, which involves computational geometry methods, the area is divided to a non-intersecting cells whose union fills the work-area. One popular approach is the trapezoidal decomposition [17], in which the space is divided to trapezoidal cells. A more suitable for robots approach is the boustrophedon decomposition [5].

Another division should obviously be also between single-robot and multi-robot algorithms. recent years are seeing much interest in multi-robot coverage algorithms, thanks to two key features made possible by using multiple robots: (i) enhanced productivity, thanks to the parallelization of sub-tasks,

and (ii) robustness in face of single-robot catastrophic failures. We want to note that robustness can be also interpreted in a different manner. Acar and Choset [1] presented a robust on-line single robot coverage algorithm while their robustness quality is the ability to filter bad sensors readings.

Our off-line algorithms build on the single-robot off-line STC algorithm [9] that is based on an approximate cellular decomposition. A different approach to extending the STC algorithm to multiple robots can be found in [7], but does not carry the robustness and performance guarantees we provide below.

Most related to our off-line algorithms is the work by Spires and Goldsmith [24], that shows an off-line multi-robots algorithm based on an approximate cellular decomposition. The algorithm uses a Hilbert space-filling curve which guarantees a robust coverage path. Unfortunately, this works only in obstacle-free work-areas. The algorithms we describe handle obstacles. Spires and Goldsmith argue that the initial positions of the robots within the work-area significantly affect the coverage time, but do not provide guarantees on the performance of their algorithm. In contrast, we provide the two backtracking MSTC algorithms that are guaranteed to reduce the coverage time (compared to the single-robot case) regardless of initial positions.

Another off-line multi-robot coverage algorithm was introduced by Kurbayashi et al. [16]. Their off-line algorithm based on an exact cellular decomposition. However, no guarantees on robustness are provided. Furthermore, the algorithm is centrally executed, and thus involves a single point of failure.

Our on-line algorithm is also based on the approximate cellular decomposition, like the single robot on-line STC algorithm [9].

There have been additional investigations of on-line multi-robot coverage, but these do not guarantee a complete coverage if one of the robots failed. Wagner et al. [26–28] proposes a series of multi-robots ant-based algorithms which use approximate cellular decomposition. The algorithms involve little or no direct communications, instead using simulated pheromones, or traces of robots. Some of these algorithms solve only the discrete coverage problem and the others can not guarantee robustness due to their heuristic nature.

Rekleitis et al. [21] uses two robots to cover an unknown environment, using a visibility graph-like decomposition (sort of exact cellular decomposition). The algorithm use the robots as beacons to eliminate odometry errors, but does not address catastrophic failures (i.e., when a robot dies). In a more recent article, Rekleitis et al. [22] extends the Boustrophedon approach [4] to a multi-robot version. Their algorithm also operates under the restriction that communication between two robots is available only when they are within line of sight of each other, but has many points of failure, i.e., it could stop functioning if one of the key robots fails.

Butler et al. [3] proposed an on-line multi-robot coverage, in a rectilinear environment, which based on the exact cellular decomposition. They do not prove their robustness, and the robots could cover the same area many times.

Chapter 3

Off-line coverage algorithms

Building on definitions in previous single-robot investigations e.g. [4], [9], we focus in this chapter on the off-line coverage case, where the robots have a-priori knowledge of the work-area, i.e. they have a complete map of the work-area, its boundaries and all the obstacles (which are assumed to be static). Each robot has an associated tool shaped as a square of size D. The objective is to cover the work-area using this tool. In real-world applications, the tool may correspond to sensors that must be swept through the work-area to detect a feature of interest, and the size D may be determined by the effective range of the sensors. Or, in vacuum cleaning application, the tool may correspond to the opening of the vacuum itself, typically underneath the robot. As with previous work [9], we assume robots can move (with the tool) continuously in the four basic directions (up/down, left/right), and can locate themselves within the work-area to within a sub-cell of size D.

We divide the area into square cells of size $4D$ (each one consists of 4 sub-cells of size D), while discarding cells which are partially covered by obstacles. We define a graph structure, G(V,E). V is the nodes set, which are the center points of each cell, and E is the edges set, which are the line segments connecting centers of adjacent cells. Then we build a spanning tree for G using any spanning-tree construction algorithm. We can affect the shape of the covering path by adding weights to the edges and building a minimum spanning tree [25]. This can be used, for instance, to reduce the number of turns, by assigning horizontal edges greater weights than those of vertical edges [9].

We can now define the MSTC problem: We are given an STC path for a given work area, and a set of k robots. We assume that the robots have initial positions $S_0, \ldots, S_{k-1}$ within the cell decomposition of the work-area. In this, we depart from previous work on multi-robot coverage which does not take into account the initial positions of the k robots. The challenge is to assign k portions of the STC path to the different robots, such that when all the robots complete their assigned sub-paths, the entire work-area is covered.

In this chapter we examine an instance of this problem, where robots are assumed to be homogeneous in their same speed and tool size D. We use N to denote the number of cells in the grid, and n to denote the number of sub-cells. We further assume that the work-area is contiguous, i.e., all cells of the work-area are accessible from any starting position.

Following the efficiency criteria in single-robot coverage algorithms which requires the least possible backtracking we begin with our multi-robot non-backtracking algorithm (Section 3.1). We then introduce the backtracking algorithm and show that backtracking can lead to a lower worst-case coverage time (Section 3.2).

3.1 Non-backtracking MSTC

The coverage works in two phases. First, Algorithm 1 builds an STC path using the method in [9] (briefly described above). Then, to carry out the coverage, each robot uses its copy of this STC path, and its initial position on the path, to follow a sub-path that is assigned to it (Algorithm 2). This is done while making sure that robots make up for catastrophic failures of their peers. Note that the execution of Algorithm 2 is complete decentralized, as each robot executes its own independent copy.

Algorithm 1 MSTC Path Plan(work-area W, robots' initial positions $S_0, \ldots, S_{k-1}$)

1: Arbitrarily pick the starting point S_0
2: Starting from S_0, construct P, an STC path of W (as described above).
3: Order the positions $S_0, \ldots, S_{k-1}$ along the STC, starting from S_0 and moving in a counter-clockwise direction.
4: Return P, ordered list of positions $S_0, \ldots, S_{k-1}$.

Starting from S_0, Algorithm 1 constructs a spanning tree for G. Moving along a path which circumnavigates the spanning tree along a counterclockwise direction orders the starting points as shown in Fig. 3.1. The construction of the spanning-tree in this pre-process phase can be done by one robot and broadcast to the others, or it can be done by every robot independently while they use the same algorithm for the building of the tree.

Once the path has been constructed and divided into sections, Algorithm 2 is executed in a distributed fashion by all robots. After the initialization phase (lines 1–2), each robot starts to cover its section $[S_i, \ldots, S_j)$, from its current location S_i to the initial position S_j of the next robot, along the STC in a counterclockwise direction (lines 3–4, see Fig. 3.1). Lines 6–11 guarantee the robustness: If one robot fails, the robot behind it takes the responsibility to cover its section (see below for formal proof). To ease the notation, we denote $(a + b) \mod k$ as $a \oplus b$, and $(a - b) \mod k$ as $a \ominus b$, where k is the number of robots.

Note that the algorithm addresses communication requirements in general form. In practice, communications can be implemented in many different ways. For example, the status of liveness (lines 6, 8) can be determined by the robots' sending of an "I am alive" message every period of time. When a message is not received by a robot after a defined timeout period, it is considered dead. Alternatively, liveness can be checked when reaching the initial position of another robot. Similarly, the announcement of section completion (line 5) can be communicated in various ways.

We analyze these algorithms. First, to prove completeness and optimality we remind the reader that circumnavigating the spanning tree produce a closed curve which visits all the sub-cells exactly one

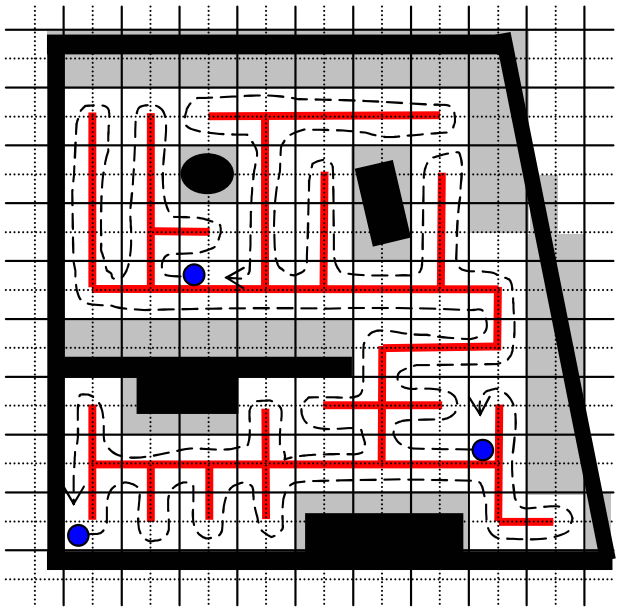


Figure 3.1: The grid, the spanning tree and the paths for three robots.

Algorithm 2 non-backtracking MSTC(STC path P, ordered positions $S_0, \ldots, S_{k-1}$)

1: Let $i \leftarrow$ my own id (in the range $0 \ldots k-1$)
2: Let $t \leftarrow i \oplus 1$ // next robot's position, cyclically
3: **while** current position $\neq S_t - 1$ **do**
4: Move toward S_t along STC, in counter-clockwise direction // this changes current position
5: Announce completion of $[S_i, S_t)$
6: **while** R_t is alive and $[S_0, \ldots, S_{k-1}, S_0]$ incomplete **do**
7: Wait
8: **if** R_t is not alive and $[S_0, \ldots, S_{k-1}, S_0]$ incomplete **then**
9: $i \leftarrow t$
10: $t \leftarrow t \oplus 1$
11: Goto 3 // Take over role of failing robot
12: Stop.

time [9]. In Algorithm 1 the STC curve is partitioned into k sections whose union is the whole path. That leads to the completeness theorem below.

Theorem 3.1.1 (Completeness)**.** *Algorithm 1 generates k paths that together cover every cell accessible from the starting cell S_0.*

Proof. Previous work has shown that step 2 produces a path that covers all cells (Lemma 3.3 in [9]). Step 3 partitions this path into k sections. Therefore, the union of the k sections covers every cell accessible from S_0. □

Given the set of paths produced, Algorithm 2 makes sure the robots visit all these cells only once (if no failure has occurred). The following theorem applies.

Theorem 3.1.2 (Non-Backtracking)**.** *If all robots use Algorithm 2, and no robot fails, no cell is visited more than once.*

Proof. If no robot fails, then each robot i only covers the section $[S_i, S_{i\oplus 1})$ of the STC path (where if $i = k$, then cyclically $i + 1 = 0$). Thus every cell is covered only by a single robot. Since robots never backtrack, every point is only covered once. □

Robustness. As one key motivation for using multiple robots comes from robustness concerns, we prove that Algorithm 2 above is robust to catastrophic failures, where robots fail and can no longer move. This result relies on an assumption that robots which fail do not block live robots.

Theorem 3.1.3 (Robustness)**.** *Algorithm 2 guarantees that the coverage will be completed in finite time even with up to $k - 1$ robots failing.*

Proof. The path is divided to k sections. We will prove that each section will be covered. Due to the nature of the path generated, all the robots are topologically moving in a circle, so the robot that is responsible to cover a section has $k - 1$ robots behind it. This is correct for any section i. We will prove that this section i will be covered, by induction on the number of robots k.

Induction Base ($k = 3$). If robot R_i that is responsible to cover this section is not dead before the completion of the cover of this section, then this section is covered. Else, $R_{i\ominus 1}$ or $R_{i\ominus 2}$ is alive. If $R_{i\ominus 1}$ is alive, according to line 6 in the algorithm it will return to step 3 and cover this section. If only $R_{i\ominus 2}$ is alive, according to line 6 in the algorithm it will return to step 3 and cover section $i - 1$ (because $R_{i\ominus 1}$ is not alive). Then the condition will be true again because R_i is dead, and $R_{i\ominus 2}$ will cover also section i.

Induction Step. Suppose it is known that if at least one of k robots is alive section i will be covered. We will prove it for $k + 1$ robots.

If robot R_i that is responsible to cover this section is not dead before the completion of the cover of this section, then this section is covered. Otherwise, there is at least one of k robots behind it that is alive. According the induction step, every section within k sections behind R_i will be covered, including the section behind it. The robot that will cover this section will cover also section i (according to line 6 in the algorithm, because R_i is not alive). □

Robustness is guaranteed with a simple mechanism. There is no need to reconfigure the group after a robot failed. It also does not matter which robot fails or how many robots failed at the same time.

Robustness against collisions is an additional concern with multiple robots. Normally, as each robot only covers its own section, theorem 3.1.2 also guarantees that no collisions take place, as the STC path never crosses itself. In practice, localization and movement errors may cause the robot to move away from its assigned path, and thus risk collision. Despite this, the separation between the paths of different robots decreases the chance of collisions.

Efficiency. Additional important motivation for using multiple robots is the possibility of reducing the coverage time by parallelizing portions of the coverage. In single-robot settings, guarantees of completeness and non-backtracking are sufficient to show (in combination) optimality of coverage time, since every cell is visited, but only once (the minimum). Thus n cells are covered in n steps.

To analyze the number of steps required to complete the coverage, we have to take into account the initial configuration. We define the *running time* as the maximum over the steps that each robot has to go, $\max_{i \in k} step(i)$, where $step(i)$ is the total number of steps taken by robot i.

Using multiple robots, the hope is to reduce the coverage time to approximately n/k. Indeed, the following theorem shows this to be a best-case scenario for Algorithm 2.

Theorem 3.1.4 (MSTC Non-Backtracking Best Case)**.** *The best running time for Algorithm 2 is* $\lceil \frac{n}{k} - 1 \rceil$

Proof. The best-case scenario is when the starting positions $S_0, \ldots, S_{k-1}$ place the robots at equal distance from each other, thus partitioning the STC path into k sections, each of size n/k. □

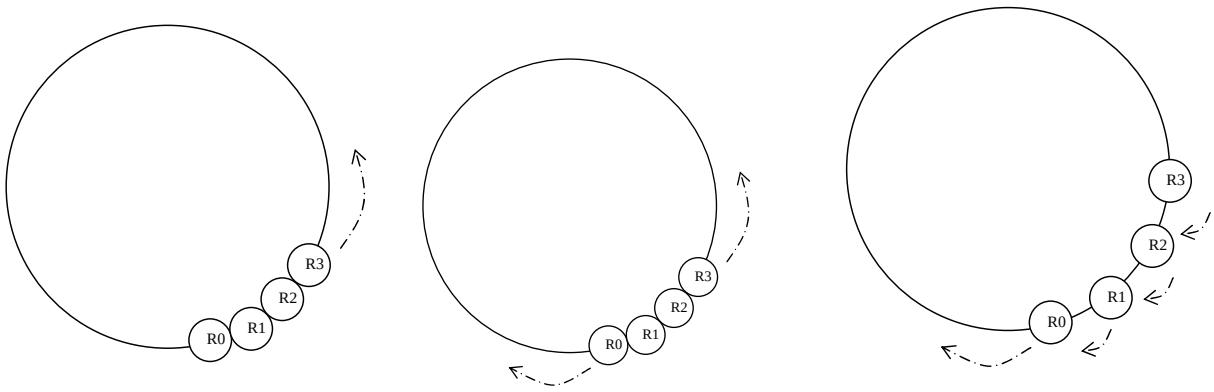
Unfortunately, it turns out that the running time is critically dependent on the initial positions of the robots. Indeed as the following theorem shows, the worst case scenario for Algorithm 2 has a running time that is almost equivalent to that of a single robot.

Theorem 3.1.5 (MSTC Non-Backtracking Worst Case)**.** *The worst running time for Algorithm 2 is* $n - k - 1$.

Proof. The worst-case scenario is where all the robots start next to each other, on adjacent cells. Since all robots move in the same direction, all but one robot will only cover the cell they are on before reaching the end of their assigned section. One robot will have a section assigned to that contains all $n - k$ remaining sub-cells (Fig. 3.2(a)). □

The result demonstrates that the initial position of the robot within the work-area can adversely affect the coverage time. Unfortunately, the worst-case scenario is common in real-world applications, e.g., vacuuming (all robots start from a single doorway), de-mining (all robots start from a single entry point to the mine field), or lawn mowing (all robots start at the mower storage area).

This worst-case scenario may appear deceptively simple to address. One may reason that by allowing another robot to head in the opposite direction, two robots may cover the $n - k$ section in parallel, thus completing the coverage in approximately $n/2$ (Fig. 3.2(b)). However, it turns out that this is incorrect.



(a) Worst case, our non-backtracking algorithm (b) dual-direction solution to our non-backtracking algorithm worst-case (c) Worst case, any non-backtracking algorithm

Figure 3.2: Non-backtracking worst-cases

A more general result is proven below, and shows that the worst-case scenario is in fact much more general than for Algorithm 2. Indeed, it is applicable to any STC-based algorithm that is non-backtracking.

Theorem 3.1.6 (General Non-Backtracking Worst Case)**.** *Any non-backtracking covering algorithm based on partitioning the spanning tree path to sections, has a worst-case running time of* $n-2(k-1)-1$.

Proof. Consider the case where robots are positioned such that a single empty sub-cell separates each pair (Fig. 3.2(c)). Because no backtracking is allowed, only one of the extreme robots can cover the big part of the path. The others, including the extreme robot from the other side, can cover only the empty sub-cell next to them, regardless the method that the algorithm chooses for deciding on a direction for movement. So we get $k-1$ robots that cover two squares (their square, and the square next to them), and one robot that has to cover the rest of the path $n-2(k-1)-1$. □

In other words, there is no non-backtracking algorithm for setting the coverage direction of the coverage for different robots such that the worst case above is eliminated. We remind the reader that the requirement for non-backtracking movement is inherited from the single-robot STC algorithm [9], where it also leads to optimality in coverage time. The next section examines what happens when we remove the requirement of non-backtracking movement while preserving the STC movement rules.

3.2 Backtracking MSTC

Let us examine an instance of the worst-case scenario of a non-backtracking algorithm, with only two robots that are positioned such that there is a single empty sub-cell between them. Without backtracking, one of the robots would have to commit to covering the single sub-cell, while the other would then be forced to cover the remaining $n-3$ sub-cells. However, if we allow robots to backtrack, then the robot that covers the single sub-cell would be able to cover it, then backtrack, and head in the other direction.

The two robots would then meet approximately in the middle of the $n-3$ section, thus halving the coverage time.

Naturally, a new worst-case scenario can be found for this back-tracking case. In this scenario, the initial positions of the two robots separated by are a third of the STC path. One robot thus covers $2/3$ of the path, while the other robot goes a $1/3$ of the path in one direction and then backtracks, but it can't help the first one in its section. The overall coverage time will be $2n/3$.

To define a general back-tracking algorithm, let us first define a few helpful notations. sec_i is the section that robot R_i is responsible to cover. Unlike in the non- backtracking algorithm sometimes $sec_i \neq [S_i, S_{i\oplus 1})$ (The section which starts at S_i and ends just before $S_{i\oplus 1}$ when moving in a counter-clockwise direction along the STC path, as defined before). We use $|[S_l, S_j]|$ to denote the length of the section $[S_l, S_j]$, taken along the shortest path along the STC cycle. The point $S_i + L$ is the point in a distance of L from S_i when moving in a counterclockwise direction along the STC path. $D1_i$ is the initial direction of movement for robot i, while $D2_i$ is the direction of movement for robot i if it has to backtrack.

We now turn to describing the MSTC backtracking algorithm. The first phase of building the STC and ordering the starting point is the same as in the non-backtracking case (Algorithm 1). We add another initialization phase where the robots re-divide the sections if backtracking is needed. The case where there are only two robots is somewhat different from the general case, so we present two initializations algorithms: Algorithm 3 if there are only two robots and Algorithm 4 for $k > 2$ robots. After this initialization phase the robots follow the backtracking algorithm (Algorithm 5).

The idea of the initialization phase is to allocate sections and directions of movement to the robots. With two robots, if there is no part of the path that is longer than 2/3 of the entire STC path, all the sections and directions of movement are the same as in the non-backtracking algorithm (Algorithm 3, lines 1–4). Otherwise, the robots share the coverage of their sections. They first cover the shortest section between them (thus one of them will have to go in a clockwise direction) and then backtrack to cover together the other section.

Algorithm 3 2 robots initialization phase(STC path P, ordered positions S_0, S_1)

1: Let $sec_0 \leftarrow [S_0, S_1)$
2: Let $sec_1 \leftarrow [S_1, S_0)$
3: Let $D1_0, D1_1 \leftarrow$ counterclockwise
4: Let $D2_0, D2_1 \leftarrow$ null
5: **if** there is h such that $sec_h > \frac{2}{3}(|[S_0, S_1)| + |[S_1, S_0)|$ **then**
6: $\quad i \leftarrow h \oplus 1$
7: $\quad sec_h \leftarrow [S_i + \lceil \frac{|[S_i, S_h)|}{2} \rceil, S_h + \lceil \frac{|[S_h, S_i)|}{2} \rceil)$
8: $\quad sec_i \leftarrow [S_h + \lceil \frac{|[S_h, S_i)|}{2} \rceil, S_i + \lceil \frac{|[S_i, S_h)|}{2} \rceil)$
9: $\quad D1_h \leftarrow clockwise$
10: $\quad D2_h \leftarrow counterclockwise$
11: $\quad D1_i \leftarrow counterclockwise$
12: $\quad D2_i \leftarrow clockwise$

Algorithm 4 general initialization phase(STC path P, ordered positions $S_0, \ldots, S_{k-1}, k > 2$)

for all i such that $0 \leq i \leq k-1$ **do**
  Let $sec_i \leftarrow [S_i, S_{i\oplus 1})$
  Let $D1_i \leftarrow$ counterclockwise
  Let $D2_i \leftarrow$ null
if there is h such that $sec_h > \frac{1}{2}(\sum_0^k |[S_i, S_{i\oplus 1})|)$ **then**
  $i \leftarrow h \oplus 1$
  $j \leftarrow i \oplus 1$
  $f \leftarrow j \oplus 1$
  if $|[S_i, S_j)| < |[S_j, S_f)|$ **then**
    $sec_h \leftarrow [S_h, S_h + \lceil \frac{|[S_h,S_i)|+|[S_i,S_j)|}{2} \rceil)$
    $sec_i \leftarrow [S_h + \lceil \frac{|[S_h,S_i)|+|[S_i,S_j)|}{2} \rceil, S_i + \lceil \frac{|[S_i,S_j)|}{2} \rceil)$
    $D1_i \leftarrow counterclockwise$
    $D2_i \leftarrow clockwise$
    $sec_j \leftarrow [S_i + \lceil \frac{|[S_i,S_j)|}{2} \rceil, S_f)$
    $D1_j \leftarrow clockwise$
    $D2_j \leftarrow counterclockwise$
  else
    $D1_j \leftarrow counterclockwise$
    $D2_j \leftarrow clockwise$
    if $h = f$ **then**
      $sec_h \leftarrow [S_j + \lceil \frac{|[S_j,S_h)|}{2} \rceil, S_h + \lceil \frac{|[S_h,S_i)|+|[S_j,S_h)|}{2} \rceil)$
      $D1_h \leftarrow clockwise$
      $D2_h \leftarrow counterclockwise$
      $sec_i \leftarrow [S_h + \lceil \frac{|[S_h,S_i)|+|[S_j,S_h)|}{2} \rceil, S_i)$
      $D1_i \leftarrow clockwise$
      $sec_j \leftarrow [S_i, \lceil \frac{|[S_j,S_h)|}{2} \rceil)$
    else
      $sec_h \leftarrow [S_h, S_h + \lceil \frac{|[S_h,S_i)|}{2} \rceil)$
      $sec_i \leftarrow [S_h + \lceil \frac{|[S_h,S_i)|}{2} \rceil, S_i)$
      $D1_i \leftarrow clockwise$
      $sec_j \leftarrow [S_i, S_j + \lceil \frac{|[S_j,S_f)|}{2} \rceil)$
      $sec_f \leftarrow [S_j + \lceil \frac{|[S_j,S_f)|}{2} \rceil, S_{f\oplus 1})$
      $D1_f \leftarrow clockwise$
      $D2_f \leftarrow counterclockwise$

With more than two robots, only if there is no part of the path that is longer than **half** of the entire STC path, all the sections and directions of movement are the same as in the non-backtracking algorithm (Algorithm 4, lines 1–4). Otherwise, the two robots that have this section between them share its coverage. One of them will have to go in a clockwise direction, leaving to the robot next to it (from the other side) to also cover the distance between them. To avoid the case that this robot will have to cover more than half of the path because of the backtracking, this robot gets help from one of its neighbors—the one closest to it. They both cover half of the distance between them and return to cover their original part of the path. See Fig. 3.3 for example of this situation.

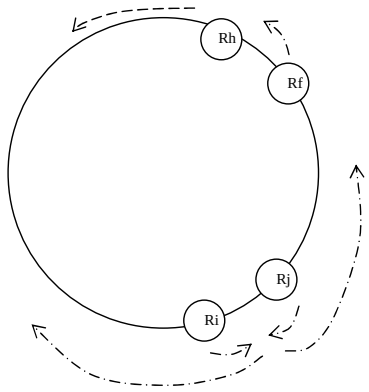


Figure 3.3: An execution example of Algorithm 5. The indexes used are the same as in the general initialization phase.

The backtracking algorithm (Algorithm 5) follows the re-divided sections generated in the initialization phase, similarly to the way the non-backtracking algorithm does. Algorithm 5 also ensures that only after a robot finishes to cover its section, even if it includes going in one direction and then backtrack, it covers sections of dead robots. Thus this algorithm is also robust.

The algorithm's completeness and robustness can be proven similarly to the completeness and robustness of the non-backtracking Algorithm 2. With respect to its backtracking, it can be easily shown that any point that is covered more than once, is covered by the same robot, and that there is no point that is covered more than twice.

The best-case coverage time for the backtracking MSTC algorithm is the same as for the non-backtracking version, i.e., $\frac{n}{k} - 1$. This is because in the best case, the initial positions of the robots are equidistant, and the robots can cover their sections without backtracking. The worst-case coverage time is analyzed below:

Theorem 3.2.1 (MSTC Backtracking Worst Case)**.** *The worst-case running time for Algorithm 5 is* $n/2 - 1$ *when* $k > 2$*, and* $\lceil 2n/3 - 1 \rceil$ *when* $k = 2$*.*

Proof. There are two cases, depending on the value of k. **Case 1 (**$k = 2$**).** In the worst case, one of the robots has a section x that is less than or equal to half the path. If x is longer than a third $(1/3)$ of the entire path, the other robot covers a section less than $2/3$ of the path, and we are done. If x is

Algorithm 5 backtracking MSTC(STC path P, ordered positions $S_0, \ldots, S_{k-1}$)

Require: initialization phase

1: Let $s \leftarrow$ my own id (in the range $0 \ldots k-1$)
2: Let $t \leftarrow s \oplus 1$ // next robot's position, cyclically
3: **while** current position $\neq$ one edge of your sec **do**
4: Move towards edge of your sec along STC, according your $D1$ argument
5: **if** your $D2 \neq null$ **then**
6: your $D1 \leftarrow$ your $D2$
7: your $D2 \leftarrow null$
8: Go to 3
9: **else**
10: Announce completion of your sec
11: **while** R_t is alive and there is i such that sec_i incomplete **do**
12: Wait
13: **if** R_t is not alive and and there is i such that sec_i incomplete **then**
14: $s \leftarrow t$
15: $t \leftarrow t \oplus 1$
16: Go to 3 // Take over role of failing robot
17: Stop

equal to a $1/3$ of the path, then the other robot covers $2/3$ of the path, i.e., $\lceil 2n/3 - 1 \rceil$, and we are done. Otherwise, x is shorter than a $1/3$ of the path, i.e., $x = n/3 - y, y > 0$. The robot that covers x backtracks over it. In this time the other robot passes twice that length, i.e., $2(n/3 - y) = 2n/3 - 2y$. At this point, the portion of the path remaining uncovered is $n - (n/3 - y) - (2n/3 - 2y) = 3y$. The two robots cover it together so each of them covers half of it. Hence, the total time taken by each is $2n/3 - 2y + 1.5y = 2n/3 - y/2$. If y is even, then, this is at most $2n/3 - 1$. If y is odd, then one robot covers $\lfloor y/2 \rfloor$ and the other $\lfloor y/2 \rfloor + 1$; i.e., the worst time in this case is $2n/3 - \lfloor y/2 \rfloor - 1 = 2n/3 - 1$.

Case 2 ($k > 2$). If there is no section that is longer than half of the path, then when every robot covers its section, no robot covers more than half of the path. On the other hand, if there is a section longer than half the path, then necessarily it is the only one. We denote it as $[S_h, S_i)$ (as in the algorithm). There are three possible cases:

- $|[S_i, S_j)| < |[S_j, S_f)|$. $|[S_i, S_j)| + |[S_j, S_f)| <$ half of the entire path $\Rightarrow |[S_i, S_j)| < 1/4$ of the entire path. R_j passes twice over half of $[S_i, S_j)$ and over $[S_j, S_f)$ so the its total path is: $|[S_i, S_j)| + |[S_j, S_f)| <$ half of the entire path. R_i passes twice over half of $[S_i, S_j)$ and over $[S_h, S_i)$ until it meets R_h. In the time that R_j passes half of $[S_i, S_j)$ and backtracks, R_h passes this distance ($|[S_i, S_j)|$) to $R_j \Rightarrow$ The remaining area to cover is $(|[S_h, S_i)| + |[S_i, S_j)|)/2 \Rightarrow$ The number of steps for every one of them is: $\frac{|[S_h,S_i)|+|[S_i,S_j)|}{2} + |[S_i, S_j)| = |[S_h, S_i)|/2 + |[S_i, S_j)|/2$. $|[S_h, S_i)| \leq n - (|[S_i, S_j)| + |[S_j, S_f)|) \Rightarrow |[S_h, S_i)|/2 + |[S_i, S_j)|/2 \leq n - |[S_j, S_f)|/2 \leq n/2 - 1$
- $|[S_i, S_j)| \geq |[S_j, S_f)|$ **and** $h = f$. This case could only happen with three robots. The proof is analogous to that of the previous case.

- $|[S_i, S_j)| \geq |[S_j, S_f)|$ **while** $h \neq f$. R_f passes twice over half of $[S_j, S_f)$ and over $[S_f, S_{f\oplus 1})$, so its total path is $|[S_j, S_f)| + |[S_f, S_{f\oplus 1})|$. R_j passes twice over half of $[S_j, S_f)$ and over $[S_i, S_j)$, so its total path is $|[S_i, S_j)| + |[S_j, S_f)|$. $|[S_h, S_i)| >$ half of the entire path $\Rightarrow |[S_i, S_j)| + |[S_j, S_f)| + |[S_f, S_{f\oplus 1})| \leq$ half of the entire path $\Rightarrow$ R_j and R_f covered less than half of the entire path. R_h and R_i passes half of $[S_h, S_i)$. $|[S_h, S_i)| <$ length of the entire path $\Rightarrow$ R_h and R_i covered less than half of the entire path.

Thus in all cases, three or more robots take no more than $n/2 - 1$ to complete coverage. □

The key insight offered by these results is that non-backtracking, the property that no portion of the work-area is covered more than once, is a distinct performance criteria from that of efficiency. These converge in the single robot case, but not in general. Indeed, it can be shown that *only* utilizing some backtracking can we guarantee improved coverage time. To see this, consider a case where the two robots are behind each other, in a corridor leading into the work area. Unless the second robot covers at least a portion of the area covered by the first robot, there is no way for the robots to split the covering task between them. Without some redundancy, the first robot will necessarily have to cover almost all of the work area by itself.

To see this, consider the situation in Fig. 3.4. Here, the two robots are in a corridor leading into the work area. Unless the second robot covers at least a portion of the area covered by the first robot, there is no way for the robots to split the covering task between them. Without some redundancy, the first robot will necessarily have to cover almost all of the work area by itself.

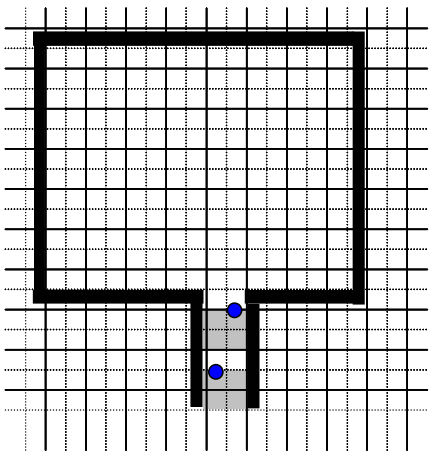


Figure 3.4: A situation example when backtracking is a must to increase efficiency

This result is not intuitive. For instance, in their 2004 paper on multi-robot coverage using limited communications, Rekleitis et al. write in the introduction [22]:

> "...we seek to reduce repeat coverage, which is defined as any robot covering previously covered space. Any reduction in repeat coverage increases the performance, as more robots cover simultaneously new space."

We offer a revised version of this goal. Reduction in redundancy may cause an *increase* in coverage time, and thus reduce the performance. The deployment of multiple robots must take this into account, and balance redundancy and efficiency as required.

Chapter 4

Efficiency in off-line coverage

In the previous chapter we introduced two off-line algorithms and analyzed their characteristics. We now introduced an improved algorithm and discuss different efficiency aspects of the three algorithms. We first introduce the polynomial-time *optimal* backtracking MSTC algorithm, and show that its coverage time is significantly better than the simple backtracking algorithm (Section 4.1). We then examine the possibility to use our algorithms with heterogenous robots with different speeds or different fuel/battery time (Section 4.2). In the last section we discuss the option to build a more efficient spanning tree which can improve the overall coverage time of our three off-line algorithms (Section 4.3).

4.1 Optimal Backtracking MSTC

Algorithm 5 in the previous chapter allows backtracking for only two robots and only in the case where one robot has to cover more than half of the entire working area. It does not generate an optimal allocation of robots to assigned sections and directions, and thus, while it guarantees a better worst-case coverage time than the non-backtracking algorithm, its average performance can be improved.

The optimal backtracking MSTC initialization algorithm below (Algorithm 6) allows all robots to backtrack over any number of steps, in order to achieve the best time for the given initial configuration. It is intended as a drop-in replacement for Algorithm 4, initializing tasks for each robot. While intuitively it may seem that the run-time complexity will grow combinatorically, with the number of possible allocations, it turns out that given the discretization of the problem, a polynomial-time solution exists.

The algorithm is described below. It assigns a *solution* to each robot, where this solution is a tuple $\langle R, L_1, L_2, D_1, D_2 \rangle$. R is the index of the robot in question, $0 \leq R \leq k-1$. L_1 is the length of the section to take before switching directions. L_2 is the length of the section to take after switching directions. If no switching is needed, its value will be zero. D_1 is the first direction to take; D_2 is the direction to take after travelling length L_1 along the STC, in direction D_1. D_1, D_2 can therefore be cw (clockwise), ccw (counterclockwise), or $null$ (not switching direction). As before, we use $|[S_l, S_j]|$ to denote the length of the section $[S_l, S_j]$, taken along the shortest path along the STC cycle.

The key to the algorithm resides in the discretization of the problem. Because we divide the working

Algorithm 6 Optimal(STC path P, ordered positions $S_0, \ldots, S_{k-1}$)

```
1: t ← ∅
2: for i ← 0 to k − 1 do
3:    for l ← 0 to |[S_i, S_{i⊖1}]| do
4:       if not Check(i, l, |[S_i, S_{i⊕1}]|) then
5:          continue to next l
6:       else if Check(i, l, 0) then
7:          t ← Solution(i, l, 0, t)
8:          break inner loop
9:       else
10:         r ← Search(0, |[S_i, S_{i⊕1}]|, i, left,' right search')
11:         t ← Solution(i, l,r, t)
12:   for r ← 0 to |[S_i, S_{i⊕1}]| do
13:      if not Check(i, |[S_i, S_{i⊖1}]|, r) then
14:         continue to next r
15:      else if Check(i, 0, r) then
16:         t ← Solution(i, 0, r, t)
17:         break inner loop
18:      else
19:         l ← Search(0, |[S_i, S_{i⊖1}]|, i, right,' left search')
20:         t ← Solution(i, l, r, t)
21: return t
```

area to cells and sub-cells, each robot can move a finite number of steps. The algorithm checks all options for an optimal assignment of paths, using a binary search.

For each robot (line 2), Algorithm 6 checks all the possible steps that the robot can move in a counterclockwise direction along the spanning tree path, until it reaches the robot next to it (line 3). For each possible step, we check whether the robot should backtrack and move in the opposite direction. The total movement duration is the value of that solution. We first check that it is a valid solution, meaning that within the solution duration time all the robots can complete to cover the rest of the area (the procedure Check in lines 4–5 and in the function Search). We then store this solution if it is the best so far (the procedure Solution in lines 7 and 10). We similarly test the other side, i.e. check all possible steps in the clockwise direction along the spanning tree path, until it reaches the robot next to it (lines 12–20).

The *Check* procedure works as follows. We get a configuration of robot, i, and its movements, and have to calculate if the other robots can complete the coverage in the same time it takes to robot i to complete its sections. The idea is that when fixing the movement of one robot to determine overall coverage time, all other robots have only one opportunity for movement within the same time frame, so the check for the validity of the solution is linear in the number of robots. First, robot i's coverage time is calculated. To minimize the total coverage time, if robot i has to backtrack, it will do so in the smaller section (line 1). We then check what is the distance that the next robot can cover in the same time. If there is a remaining area between the robots which both did not cover it is not a valid solution (line 4). If

not, this robot has a remaining area between it and the robot next to it (lines 7,10) that has to be covered in the same time frame, so we repeat the check between them. The check is done cyclically for all the robots (line 3), and if the total area can be covered within the time frame which was determined by the given configuration, the solution is valid.

Algorithm 7 Check(robot i, ccw movement amount $left$, cw movement amount $right$)

1: $time \leftarrow \min(left, right) \cdot 2 + \max(left, right)$
2: $area \leftarrow |[S_i, S_{i\oplus 1}]| - right$
3: **for** $r \leftarrow [i \oplus 1, i \oplus 2, \ldots, i \ominus 1]$ **do**
4: **if** $area > time$ **then**
5: return $false$
6: **else if** $area \cdot 2 \geq time$ **then**
7: $area \leftarrow |[S_r, S_{r\oplus 1}]|$
8: **else**
9: $best \leftarrow \max(time - 2 \cdot area, \frac{time-area}{2})$
10: $area \leftarrow \max(|[S_r, S_{r\oplus 1}]| - best, 0)$
11: **if** $area > right$ **then**
12: return $false$
13: **else**
14: return $true$

The *Solution* procedure (Algorithm 8) creates a new solution tuple, if it is better than the existing best solution t. First, in lines 1–3, the new solution length is compared to the existing solution length. If the existing solution is better or equal to the new solution, the existing solution is returned (line 4). Otherwise, the new solution tuple is calculated. If the amount of CCW movement $left$ is zero, then the only movement is in the CW direction. The length of that movement L_1 is equal to the amount of movement in the CW direction, $right$, and the first direction to take is in the CW (clockwise) direction. This is done in lines 5–8. A similar check is done for the opposite direction (lines 9–12). However, if the solution calls for moving bi-directionally, than the algorithm first checks which direction involves less movement, CCW (lines 13–17) or CW (18–22). In either case, the direction involving less movement is taken first, since it will be backtracked-over in the counter direction.

The *Search* procedure (Algorithm 9) performs a binary search over the length of a section between one robot to another. It gets a fixed movement length in one direction for a specific robot, and searches for the shortest length that this robot can move in the other direction, while still enabling the other robots to complete the coverage of the remaining area in the same time it takes to this robot to complete its two direction movement. This check is done with *Check* procedure (lines 6, 11), recursively, until the length is found (lines 1–2). The direction of search is determined by the 'type' argument: a **right search** means that the 'movement' argument is a fixed counterclockwise movement along the spanning tree, with a length of **movement**, so the search is done for the length of the clockwise movement. A **left search** means the opposite. Before the call to this procedure we examine that there is a valid solution with this robot with the maximal possible movement (lines 4–5 in Algorithm 6) to guarantee that the procedure will not be stuck in an endless loop.

Algorithm 8 Solution(robot i, ccw movement $left$, cw movement $right$, current best solution t)

$VAL \leftarrow min(left, right) \cdot 2 + max(left, right)$
$t_{VAL} \leftarrow min(t_{left}, t_{right}) \cdot 2 + max(t_{left}, t_{right})$
if $t_{VAL} \leq VAL$ **then**
    return t
if $left = 0$ **then**
    $L_1 \leftarrow right$
    $D_1 \leftarrow clockwise$
    $D_2 \leftarrow null$
else if $right = 0$ **then**
    $L_1 \leftarrow left$
    $D_1 \leftarrow counterclockwise$
    $D_2 \leftarrow null$
else if $left \leq right$ **then**
    $L_1 \leftarrow left$
    $L_2 \leftarrow right$
    $D_1 \leftarrow counterclockwise$
    $D_2 \leftarrow clockwise$
else
    $L_1 \leftarrow right$
    $L_2 \leftarrow left$
    $D_1 \leftarrow clockwise$
    $D_2 \leftarrow counterclockwise$
return $\langle i, L_1, L_2, D_1, D_2 \rangle$

Algorithm 9 Search(low border low, high border $high$, robot index i, one side movement $movement$, type of search $type$)

if low = high **then**
    return low
else
    $half \leftarrow \lfloor \frac{low+high}{2} \rfloor$
    if $type$ = 'right search' **then**
        if Check(i, $movement$, $half$) **then**
            Search(low, $\frac{low+high}{2}$, i, $movement$, 'right search')
        else
            Search($half + 1$, $high$, i, $movement$, 'right search')
    else
        if Check(i, $half$, $movement$) **then**
            Search(low, $\frac{low+high}{2}$, i, $movement$, 'left search')
        else
            Search($half + 1$, $high$, i, $movement$, 'left search')

Algorithm 6 generates the optimal allocation of robots to sections (and their backtracking, if necessary), such that coverage is achieved at the best possible time. This is proven in Theorem 4.1.1 below. The optimality is according to the MSTC movement rules which introduced before: all the robots move along the same spanning tree without crossing it, and every robot backtracks only on its own steps. The only case where a robot has to cover another robot's cell is where the latter failed and its entire allocated section has to be covered by another robot.

Theorem 4.1.1 (Optimal MSTC Optimality)**.** *Algorithm 6 generates an optimal allocation of sections such that the overall coverage time is minimal.*

Proof. The value of an solution for a given initial configuration is its overall coverage time. If this is optimal, then for each robot, its individual coverage time is less than or equal to this value; and there exists at least one robot whose coverage time is exactly that (otherwise this is not an optimal solution). By choosing the best valid solution of each robot and comparing it to the other robots' best solutions, we guarantee finding the optimal solution. □

The run-time of the allocation itself is polynomial in n, the number of sub-cells, and k, the number of robots. This is shown in Theorem 4.1.2 below.

Theorem 4.1.2 (Optimal Backtracking Run-Time)**.** *Algorithm 6 runs in time* $\mathcal{O}(nk^2 \log n)$.

Proof. The main loop is executed k times. In each phase there are 2 loops, both executed at most $O(n)$ times because this is the maximum number of possible steps. In each loop the function Check is executed twice and then the function Search and Solution (in the worst case). In the function Check there is only one loop which runs $k-1$ times thus its running time complexity is $O(k)$. The function Search runs a binary search on one section of the spanning tree path, and uses Check function in each phase so its running time complexity is $O(k \log n)$. The function Solution uses only a constant number of check so its running time complexity is $O(1)$. So, the overall running time complexity is $\mathcal{O}(nk^2 \log n)$. □

Typically the number of robots is much smaller than the number of cells in the area to be covered, i.e., $k << n$. Thus in applications, we expect the runtime to be mostly affected by the $n \log n$ factor.

The actual running time can be further improved if the algorithm is modified to skip checking values which are bigger than the largest initial section. However, while this is a useful implementation note, it does not affect theoretical runtime complexity.

4.2 Heterogeneous Robots

So far we assumed homogeneous robots, with equal speeds and fuel capacities. Thus, the coverage time for a given section, between neighboring robots, was considered equal regardless of which robot was chosen to traverse it. We now describe how heterogeneous robots, in speed and/or fuel capacity, can be taken into account.

The non-backtracking algorithm (Algorithm 2) will work in the same manner for robots with different speeds, as no optimization is attempted. The simple backtracking algorithm (Algorithm 5), however, will not function correctly. It might even result in a worse coverage time than the non-backtracking algorithm for the same initial configuration. For example, suppose we have four robots as described in figure 4.1. The values near the braces describe the number of sub-cells between the robots. Suppose R0 is a slow robot that can move in a speed of 1 sub-cell/second and R1 is a fast robot that can move in a speed of 10 sub-cells/second. The other robots can move at a speed of 5 sub-cells/second. When following the non-backtracking algorithm the coverage will be completed in 20 seconds, which is the time for R3 to cover its section. In that time the other robots can complete to cover their sections too. If the backtracking algorithm will be executed, the coverage will take 105 seconds. This long time is due to the slow movement of R0 which has to cover half of the section between it and R1 (distance of 5 sub-cells), backtrack (again distance of 5 sub-cells) and cover almost half of the section between it and R3 (distance of 95 sub-cells). The backtracking algorithm has a simple mechanism which considers only distances between the robots thus it fails when we change the assumption of homogenous robots with equal speeds.

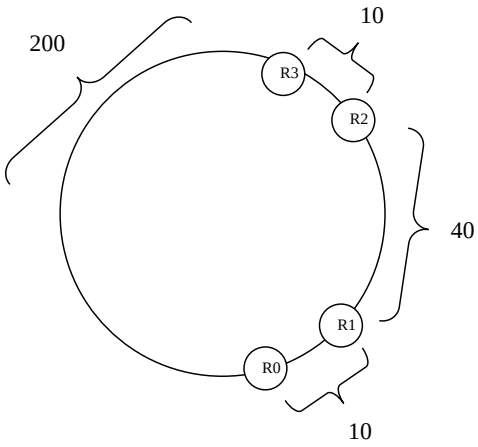


Figure 4.1: An example where the backtracking algorithm generates bad solution if the robots has different speeds.

The optimal backtracking algorithm (Algorithm 6) works can be easily extended to address robots with heterogeneous speeds. The extension needed involves modifying the way coverage time is calculated in the Check procedure (Algorithm 7) and in the Solution procedure (Algorithm 8). Instead of calculating coverage time based on the length of the section, it should be calculated based on the length given the maximum speed of the robot in question (lines 7,9-10 in Algorithm 7). With this modification, Algorithm 6 is guaranteed to return a solution that is optimal in coverage time, even taking into account heterogeneous speed limits.

We will also want to consider the case where the robots are equipped with different amount of fuel or different battery capacity. The simple algorithms (Algorithms 2, 5) do not address this case, and may

return a planned path that cannot be executed by the robots, given their fuel capacity.

However, the optimal backtracking algorithm provides a solution in this case as well. Algorithm 7 requires a modification in the calculation of the distance that a robot can cover within a given time frame, such that the calculation also takes into account the fuel available. With this change, the optimal backtracking algorithm is guaranteed to produce a solution that is feasible given the robots' fuel or battery constraints. However, this solution still minimizes coverage time, rather than fuel consumption.

4.3 Optimal Spanning Tree

Our off-line multi-robot algorithms as well as the single-robot version in [9] use a spanning tree to create a circular path which completely cover the area. When building this spanning tree in a single robot system, the influence of the structure of the tree is almost irrelevant for the coverage time. This results from the fact that coverage time is linear in the size of the grid, since each cell except for the boundary cells is covered once, hence the total coverage time is n (the number of sub-cells). The structure of the tree may only affect efficiency due to the number of turns it requires, and other similar issues. On the other hand, in our multi-robot systems, the structure of the tree can have crucial consequences on the coverage time of the terrain. The choice of the spanning tree can change the robots' initial positions from being concentrated, i.e., placed as a bundle, to being scattered along the spanning tree path - all without actually changing the physical initial position of the robots. That is mean that if the tree is appropriately built, the structure of the tree itself can substantially decrease the coverage time obtained by algorithms based upon it.

When constructing this tree we try to minimize the maximal distance between every two consecutive robots along the spanning tree path. If such tree is obtained, all versions of the MSTC algorithm ran on these tree will achieve substantially better coverage time. An illustration of the importance of the right choice of spanning tree is given in Figure 4.2. The figure presents an example for a terrain in which $n = 120$, $k = 3$ and two different trees are suggested as base for coverage. The spanning tree is described by the bold lines, and we use the different kinds of dashed lines to describe the spanning tree path, each dashed line represents the distance between two adjacent robots along the path. In order to clarify the example, the section between each two adjacent robots is given a different background as well. Note that in both grids the robots are initially located in the same positions. The tree in Figure 4.2a. places the robots almost uniformly along the tree path, thus non-backtracking algorithm will cover the area in 50 steps, the backtracking algorithm will cover it also in 50 steps and the optimal algorithm will cover it in 44 steps. However, in Figure 4.2b. the robots are placed arbitrarily along the tree path, thus the non-backtracking algorithm will cover the area in 112 steps, the backtracking algorithm will cover it in 57 steps and the optimal algorithm will cover it in 56 steps.

To formally define our problem, we are given a graph structure, G(V,E) of our cellular decomposition. V is the nodes set, which are the center points of each cell, and E is the edges set, which are the line segments connecting centers of adjacent cells. Each cell is decomposed to 4 sub-cells. We are also

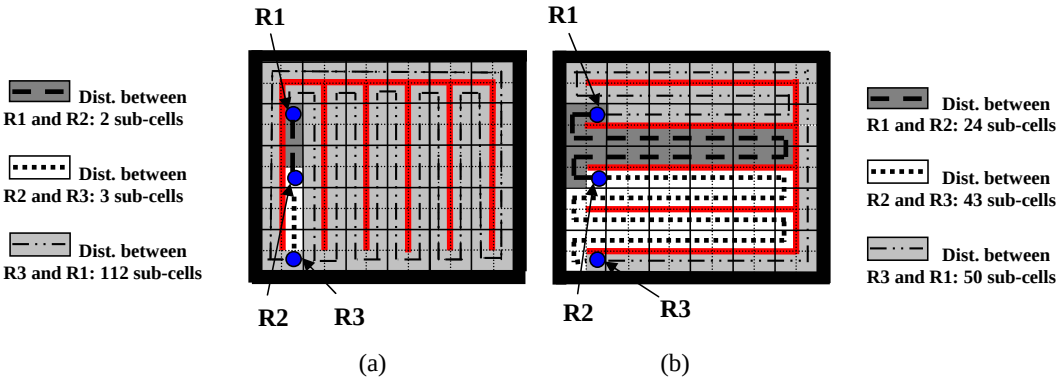


Figure 4.2: Illustrating how different trees can influence coverage time.

given initial locations of k robots on sub-cells of G' cells. For every spanning tree of G, ST_G, moving along the path which circumnavigates ST_G orders the robots. Let D_{ij} be the distance along the spanning tree path between two consecutive robots. Out problem is to find ST_G that minimizes $\max_{i \in k} D_{ij}$.

The construction of an optimal spanning tree is believed to be $\mathcal{NP}$-Hard [30]. However, an approximation algorithm (co-authored with Noa Agmon and Gal Kaminka) exists [2]. It is presented in Appendix A.

Chapter 5

On-line coverage algorithm

We focus in this chapter on the on-line coverage case, in which the robots do not have apriori knowledge of the work-area, i.e., the exact work-area boundaries and all the obstacles locations (which are assumed to be static), but their absolute initial positions. We first introduce the algorithm, and prove that it is complete, non-redundant and robust (Section 5.1). We then analyze the assumptions underlying the algorithmic requirements and present various approximation techniques for these requirements, to allow the algorithm to work in real world situations (Section 5.2).

5.1 On-line MSTC Algorithm

Our on-line algorithm basic assumptions are the same as with previous work [9] and with our off-line case in chapter 3. Each robot has an associated tool shaped as a square of size D. The objective is to cover the work-area using this tool. In real-world applications, the tool may correspond to sensors that must be swept through the work-area to detect a feature of interest, and the size D may be determined by the effective range of the sensors. Or, in vacuum cleaning application, the tool may correspond to the opening of the vacuum itself, typically underneath the robot. We also assume robots can move (with the tool) in the four basic directions (up/down, left/right), and can locate themselves within the work-area to within a cell of size D.

We divide the area into square cells of size $4D$, each one consists of four (4) sub-cells of size D. Denote the number of cells in the grid by N, and denote the number of sub-cells by n,i.e., $N = 4n$. The area occupancy in the beginning is unknown so every cell is initially considered to be empty.

The algorithm starting point is the work-area and k robots with their absolute initial positions: $A_0, \ldots, A_{k-1}$. The initial position of every robot is assumed to be in an obstacle-free cell, and the robot should know its position. Indeed, one assumption the algorithm makes—as previous work [9] does—is that robots can locate themselves within an agreed-upon grid decomposition of the work area. In practice, of course, this assumption is not necessarily satisfied. Section 5.2 below discusses methods for approximating this assumption in practice, which we utilize in our work with physical robots.

We seek algorithms that are *complete*, *non-redundant*, and *robust*. An algorithm is complete if, for k

robots, it produces paths for each robot, such that the union of all k paths complete covers the work area. An algorithm is non-redundant if it does not cover the same place more than one time. The robustness criteria ensures that as long as one robot is still alive, the coverage will be completed.

The algorithms below are run in a distributed fashion, and generate on-line coverage that is complete, non-redundant, and robust. Each robot runs the initialization algorithm first (Algorithm 10), and then executes (in parallel to its peers) an instance of the ORMSTC (On-line Robust Multi-robot STC, Algorithm 11). Each ORMSTC instance generates a path for its controlled robot on-line, one step at a time. It is the union of these paths that is guaranteed to be complete, non-redundant, and robust.

We begin by describing Algorithm 10. The initialization procedure constructs the agreed-upon coordinate system underlying the grid work area. It then allows each robot to locate itself within the grid, and update its peers on the initial position of each robot.

Algorithm 10 On-line MSTC initialization()

1: Decompose the working area into a $2D \times 2D$ grid, agreed among all the robots.
2: Decompose each $2D \times 2D$ cell to 4 $D \times D$ sub-cells
3: $i \leftarrow$ my own robot ID
4: **if** $A_i \neq$ the middle of a sub-cell **then**
5: $\quad s_i \leftarrow$ the closest sub-cell
6: $\quad$ Move to s_i
7: **else**
8: $\quad s_i \leftarrow A_i$
9: $S_i \leftarrow$ the cell that contains s_i
10: Announce S_i as your starting location to the other robots
11: Update your map about $S_0, \ldots, S_{k-1}$ // the other robots starting cells
12: Initialize $connection[0 \ldots k-1][0,1] \leftarrow null$

Once the grid is constructed—though of course, not traversed, nor mapped—and robots know their initial positions, coverage can begin. This is carried out by Algorithm 11, executed in a distributed fashion by all robots. This recursive algorithm gets two parameters: X, the cell that the robot just entered, and W, the cell from which the robot have arrived. In the first recursive call to the algorithm, the argument X is the robot's starting cell S_i, and W can theoretically be any neighboring cell of S_i. In order to be consistent with the other algorithm steps we choose W as the cell which is closest to the robot sub-cell starting position and if it enter to that cell it will be its right bottom sub-cell from the robot point of view. See Figure 5.1 for the four possible states. Note that for clarity purpose in the algorithm pseudo code we denote a cell with an obstacle in one (or more) of its 4 sub-cells, or with the robot's own spanning tree edge, as a *blocking* cell.

The main idea behind the algorithm is that every robot gradually builds a local spanning tree of the uncovered cells that it discovers, while tracking the state of its peers which it meets. The spanning tree is built by a depth-first-like procedure: Scan for a non-occupied neighboring cell (Lines 1–2), build a tree edge to it (Line 18), enter that cell (Line 19) and continue recursively with this cell (Line 20). If there is not any free cell, the robot goes back along its local spanning tree to the previous covered cell

Algorithm 11 ORMSTC(W, X)

$N_{1..3} \leftarrow$ the 3 neighboring cells of X in clockwise order, starting after W
for $i \leftarrow 1$ to 3 **do**
  if $N_i =$ blocking cell **then**
    continue to the next i
  if N_i has already a tree edge of another robot j **then**
    ask robot j if it is alive
    if robot j answered **then**
      if $connection[j][0] = null$ **then**
        $connection[j][0] = connection[j][1] =$ the edge from X to N_i
        continue to the next i
      else
        $connection[j][1] =$ the edge from X to N_i
        continue to the next i
    else
      // robot j is not alive
      remove robot j from your connections array and broadcast it
      mark its tree cells as empty on the map and broadcast it
  construct a tree edge from X to N_i and broadcast it
  move to a sub-cell of N_i by following the right-side of the tree edges
  execute ORMSTC(X, N_i)
if $X \neq S_i$ **then**
  move back from X to W along the right-side of the tree edges
  return from recursive call
if $W \neq$ blocking cell **then**
  execute ORMSTC(X, W)
move to your starting sub-cell s_i along the right-side of the tree edge
broadcast completion of your work
while not all the robots announced completion of their work **do**
  check periodically that the robots which you a have a connection with are alive
  if robot j is not alive **then**
    mark robot j tree cells as empty on the map and broadcast it
    broadcast that you didn't complete your work
    decide which connection: $connection[j][0]$ or $connection[j][1]$ is closer to your place when moving in clockwise or counter-clockwise direction along your tree edges
    move to this connection in the appropriate direction
    $X \leftarrow$ your connection cell
    $Y \leftarrow$ robot j's connection cell
    remove robot j from your connections array and broadcast it
    construct a tree edge from X to Y and broadcast it
    move to a sub-cell of Y by following the right-side of the tree edges
    execute ORMSTC(X, Y)

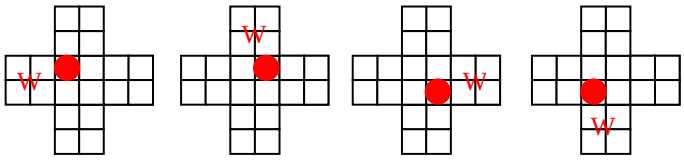


Figure 5.1: The 4 possible initial positions, and the recommended W choose for them

(Line 22) and the recursion is folding back (Line 23).

During this exploration and covering process, the first time a robot i meets a cell with robot j's tree-edge ($i \neq j$), it examines its peer's state (Lines 5–6). If robot j is still alive, robot i saves the edge which connects its tree to robot j's tree as $connection[j][0]$ (Lines 8–9). From this point on, if robot i meets again a cell with robot j edge it examines again its peer's state and if it is still alive, robot i saves this edge also as $connection[j][1]$ (Line 12). This will occur whenever robot i meets a cell with robot j's tree-edge; it updates $connection[j][1]$ to save the last edge which connects its tree to robot j's tree. If, during this phase, robot i discovers that robot j is not alive anymore, it announces to the other robots that robot j is dead. Then all robots delete the entries for robot j from their $connection$ arrays, and the cells which robot j was responsible for are considered as empty cells (Lines 15–17). Robot i and the other robots can now build their spanning tree edges to these cells and cover them.

When a robot has no neighboring cells to cover, and it is back in its initial position cell (line 24) it checks if it did not leave the cell which was first declared as W uncovered (line 24). If so, it turns to cover W in the same recursive covering procedure (line 25). if not, the robot finishes to cover its starting cell and announces to the other robots about completion of its work (lines 26–27). The coverage process is not completed until all the robots announce completion of their work. Until then, a robot who finishes its work monitors the state of all the robots with who it has a non-empty $connection$ entry (line 28–29). If one of them (suppose it is j) is not alive anymore, the robot updates the other robots that robot j is dead, thus the cells which have its tree edges should be considered as empty cells without tree edges (line 31). It then turns to cover robot j's cells, thus announces to the other robots that it did not finish its work (line 32). The robot has two possibilities to reach robot j's cells: move along the left side of its spanning tree edges till it reaches $connection[j][0]$, the first connection edge between it and robot j, or to move in an opposite direction along the right side of the spanning tree edges till it reaches $connection[j][1]$, the last connection edge between it and robot j. The robot chooses the best option and moves to the chosen connection edge (lines 33–34). Now it can delete robot j from the connection array (line 37), and continue to construct the spanning tree edges for the new free cells with the previous described recursive way (lines 38–40).

In lines 1–3, the robot has to explore its three neighboring cells. Each robot must have the ability to sense and determine if its three neighboring cells are free from obstacles. If the cell is partially occupied with an obstacle it will not be covered.

The ORMSTC algorithm requires reliable communication to operate well. Each informative mes-

sage that a robot receives (a cell that is now occupied with a tree edge, a dead robot, etc.) updates the map and overall world state (in the memory of its peers). Obviously, there is also an assumption here that robots are cooperative, in that when a robot is asked if it is alive, it broadcasts truthfully if it can.

In lines 18 and 38 the robot builds a local spanning tree edge. A synchronization problem could occur if more than one robot wants to construct a tree edge in the same cell. It can be solved by any synchronization protocol, and we chose a simple one (for our algorithm implementation): before the robot builds the edge it notifies the other robots. If one or more robots wants this cell too, all of them decide between them who will take it (by the highest ID number or by the smallest number of cells covered so far). The robots which lose the cell should treat it as a cell with another robot's spanning tree edge and continue with the algorithm.

Lines 15–17 and 30–40 guarantee the robustness. If one robot fails, there is always at least one robot that will detect it and will take the responsibility to cover its section (see below for formal proof).

We now address the completeness of the ORMSTC algorithm. Each robot constructs its own spanning tree and circumnavigate it to produce a closed curve which visits all the sub-cells of the tree cells. Completeness is achieved by ensuring that every cell (within the area boundaries) will have a tree edge connection from one of the trees. Note that this approach to achieve completeness is different from out previous off-line approach (Theorem 3.1.1 in chapter 3): In the on-line algorithm, if not stopped from going into certain cells (e.g., because they have already been visited), the robot will expand its tree to cover the entire work-area. This is in contrast to the off-line algorithms where a single spanning tree is constructed, and every robot covers only a portion of this spanning tree path.

Theorem 5.1.1 (Completeness)**.** *Given a grid work-area W, and k robots, Algorithm 11 generates k paths k_i, such that $\bigcup_i k_i = W$, i.e., the paths cover every cell within the work-area.*

Proof. By induction on the number of robots k.
Induction Base ($k = 1$). with only one robot, ORMSTC operates exactly like the On-line STC Algorithm which was proven to be complete (Lemma 3.3 in [9]).
Induction Step. Suppose it is known that $k - 1$ robots completely cover the area. We will prove it for k robots. Without loss of generality, let us consider robot i. Executing ORMSTC, i will build its local spanning tree edges, and generate a path to cover some cells. The other robots treat these cells as occupied, exactly as if they were filled with obstacles. Therefore all the other cells will be part of $k - 1$ pathes and covered by the $k - 1$ robots, according the induction relaxation. Robot i treat all the cells of the other $k - 1$ robots as occupied cells, so it will completely cover its cells according to the induction base case. □

One result which emerges from Theorem 5.1.1 is that each robot can use the algorithm in any direction, and the algorithm will still function correctly. In other words, Algorithm 11 will work even if one or more robots reverse the scan, i.e., the scan is done in a counter-clockwise direction and the movement is along the left side of the tree edges.

We now turn to examining ORMSTC with respect to coverage optimality. In our off-line algorithm chapter (Chapter 3) we discussed several optimization criteria, one of which is *redundancy*, the number of times a subcell is visited. We showed that the two efficiency and redundancy are not the same. Indeed multi-robot coverage is more efficient under some conditions, when some redundancy is allowed. Specifically, some backtracking of a robot over its own previously-visited cells (upto, and beyond, its initial position) can be beneficial to overall coverage time. However, we believe that in on-line settings, backtracking is not advisable. Before the robot reaches again its initial position it is does not know if the backtracking will improve or decrease the coverage time because it does not has a complete map of the area. After it reaches a border of its region, it makes no sense for it to go back towards its initial position, because any cell which is accessible from the robot initial position would have already been covered.

ORMSTC can be shown to be non-redundant. Theorem 5.1.2 below guarantees that the robots visit all the cells only once (if no failure has occurred—see below for a discussion of robustness). This guarantee is in fact a feature of many spanning-tree coverage algorithms, as circumnavigating a tree produce a closed curve which visits all the sub-cells exactly one time [9].

Theorem 5.1.2 (Non-Redundancy)**.** *If all robots use Algorithm 11, and no robot fails, no cell is visited more than once.*

Proof. If no robot fails, then each robot only covers the cells for which it builds a tree edge. If there is already a tree edge to a cell, the robot will not enter it (Line 5). Thus every cell is covered only by a single robot. Since robots never backtrack, every point is only covered once. □

As key motivation for using multiple robots comes from robustness concerns, we prove that Algorithm 11 above is robust to catastrophic failures, where robots fail and can no longer move.

Theorem 5.1.3 (Robustness)**.** *Algorithm 11 guarantees that the coverage will be completed in finite time even with up to* $k-1$ *robots failing.*

Proof. Based on the completeness theorem (Theorem 5.1.1), every number of robots can cover the work area so if one or more robots fail all the cells that were not occupied by tree edges of the failing robots and are accessible to other live robots will be covered. So all we have to prove is that cells with tree edges of a dead robot, or cells which are accessible only to a robot that have died will be covered by another robot. It can be possible to have such cells because of the work area structure or because the robot can create a line of covered cells which blocks the access for other robots to a group of free cells.

Cells with tree edges of a robot are treated by the other robots as cells with obstacles. According the completeness theorem, there is at least one robot that will cover a neighboring cell of one of these cells, thus will have a connection to this cell. There are two possible cases:

1. A robot failed before a robot that has a connection with it reach the connection- In that case, lines 16–17 ensures that the dead robot's covered cells will be declared free so they and the cells which were accessible only to the dead robot will be covered by other robots.

2. A robot fails after all the robots that have a connection with it reach the connection- In that case, lines 31 and 37 ensures that the robot's covered cells will be declared free so they and the cells which were accessible only to the dead robot will be covered by other robots.

In both possible cases, the algorithm is thus robust. □

5.2 From Theory to Practice

In real-world settings, some of the assumptions underlying ORMSTC can not be satisfied with certainty, and can only be approximated. This section examines methods useful for such approximations, and their instantiations with physical robots.

In particular, we have implemented the ORMSTC algorithms for controlling multiple vacuum cleaning robots, the RV-400 manufactured by Friendly Robotics [8]. Each commercial robot was modified to be controlled by an small Linux-running computer, sitting on top of it. A generic interface driver for the RV-400 robot was built in Player[11], and a client program was built to control it. Each robot has several forward-looking sonar distance sensors, as well as sideways sonars. One robot is shown Figure 5.2.

Figure 5.2: RV-400 robot used in initial experiments.

The ORMSTC algorithm (indeed, many of the STC algorithms) make several assumptions. First, there are assumptions as to the work area being provided as input. ORMSTC assumes, for instance, that the work-area has known bounds, and that it is divided into a grid that is known by all robots (i.e., all robots have the same division). ORMSTC assumes robots can communicate reliably, and locate themselves within a global coordinate system. Finally, ORMSTC makes assumptions about the sensory information available to the robots. In particular, ORMSTC makes the assumption that each robot can sense obstacles within the front, left, and right $4D$ cells.

One challenging assumption is that of a global coordinate system that all robots can locate themselves within. In outdoor environments, a GPS signal may in principle be used for such purposes (note

that the position only has to be known within the resolution of a sub-cell). However, in circumstances where a global location sensor (such as the GPS) is unavailable, a different approach is needed. In particular, this is true in the indoor environments in which the vacuum cleaning RV-400 is to operate.

For the purposes of the experiments, we have settled on letting the robots know their *initial* location on an arbitrary global coordinate system. Once robots began to move, however, they relied solely on their odometry measurements to position themselves. In the future, we hope to use a dynamic initialization procedure, where robots will sense identify their relative positions. Then, a translation/rotation matrix can be computed for each robot, given an arbitrarily selected global origin for a coordinate system.

One advantage of ORMSTC in this regard is that its movements are limited to turns of 90° left or right, and to moving forward a fixed distance. This offers an opportunity for both reducing errors by calibration for odometry errors specific to this limited range of movements, and by resetting after each step, thus avoiding accumulative errors. Indeed, this was the approach taken in the experiments (see Chapter 6).

Given a global coordinate system, ORMSTC also requires robots to agree on how to divide up the work-area into a grid. This agreement is critical: Differences in the division may cause grids created by different robots to be mis-aligned, or overlap. To do this, the bounds of the grid have to be known, in principle. Once the bounds are known, the robots only have to decide on the origin point for the approximate cell decomposition.

Here again a number of approximating solutions were found to be useful. First, one can have the robots use a dynamic work-area. During the initialization phase, the robots determines the maximal distances, X_{max} and Y_{max} (along the X- and Y- axes, respectively), over all pairs of robots. They then build a temporary rectangular work-area around them, with sides greater or equal to X_{max}, Y_{max} (see Figure 5.3). As the robots move about, they will push the boundaries of the work-area into newly discovered empty cells that lie beyond the bounds, or they will encounter the real bounds of the work area, which will be regarded as obstacles. A related approximation is to provide the robots with an initial work-area that is known to be too big, and allow the robots to discover the actual bounds. This was the technique we utilized.

Robustness against collisions is an additional concern in real-world situations. Normally, as each robot only covers the path along its own tree, Theorem 5.1.2 guarantees that no collisions take place. This separation between the paths of different robots decreases the chance of collisions. In practice, localization, movement errors, and the way the grid is constructed may cause the robot to move away from its assigned path, and thus risk collision. We utilized our bumps sensors to cope with this problem as they are often used as a key signal in vacuum-cleaning robots. Our heuristic is to simply respond to a bump by moving back a little, waiting for a random (short) period of time and trying again. If bumps occur three times in a row in the same location, the location is marked as a bound or obstacle. A more complicated solution which requires more communication is to coordinate between the robots that have adjacent tree edges when a collision is likely to occur.

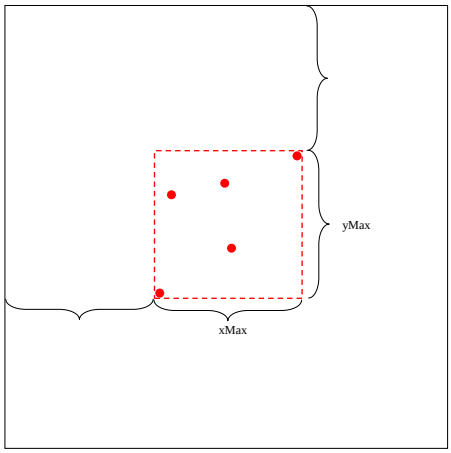


Figure 5.3: Initial dynamic work area

A final challenge was offered by the robots' limited sensor range. The robot is equipped with ten sonar sensors which are not capable of sensing all three neighboring cells of the robot cell at the same time as described in the algorithm requirements before. We solved this problem by dividing the original sensing and movement phases to three steps. The robot first senses its first cell by turning its sensors towards it. If it is empty, it continues with the regular algorithm flow. If not, in moves forward to be as close as possible to the border of the next require-sensing cell and only then it turns to sense it and continues with the algorithm. The same procedure is performed to the third neighboring cell. Although it slowed down the algorithm performance this fix enabled us to run the algorithm in the simulation with the robots constraints, so it can be applied also to run the algorithm on different real robots with limited sensors.

Chapter 6

Experiments

To empirically evaluate the performance of the algorithms, we conducted an extensive set of experiments. We first compare our off-line algorithms with different environments and different starting positions 6.1. We then describe our implementation of the on-line algorithm for controlling multiple vacuum cleaning robots 6.2.

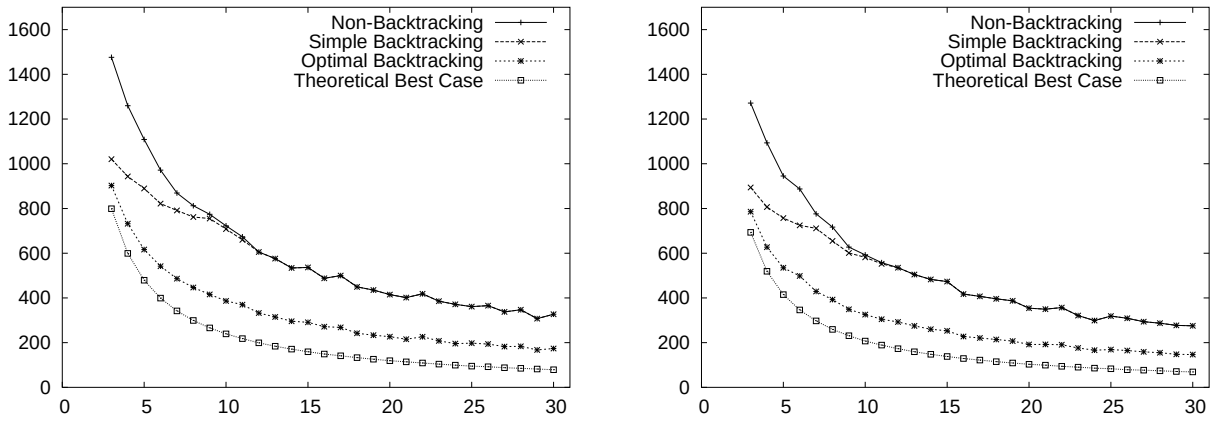
6.1 Off-line algorithms experimental results

We empirically evaluated the performance of the different off-line algorithms. In a first set of experiments, 3 to 30 robots were assigned for covering a grid of size 30×20 cells, i.e., 2400 D-size subcells. In each trial of the experiments, the number of robots was fixed, and their initial positions were randomly generated. Then the different coverage algorithms were run to calculate the coverage time. Each such trial was repeated 100 times. We repeated these experiments for both an empty grid, as well as grid with 80 $4D$ obstacle cells, whose position was randomly generated.

Figure 6.1-a shows the results of these experiments, in the empty grid case. In the figure, the X-axis shows the number of robots, while the Y-axis shows the running time. The figure shows several curves. The worst-case curves were calculated analytically, and show the worst-case coverage-times for the backtracking and non-backtracking algorithms. The best-case curve was also calculated analytically, and is shown so as to provide a benchmark against which to interpret the actual algorithms running times. Figure 6.1-b shows similar results, but for the grid with obstacles.

The figure shows that the simple non-backtracking and backtracking algorithms have a difference in performance for small teams, but converge and show the same run-time for larger teams. However, the optimal backtracking algorithm is clearly superior to the two techniques. This shows that the run-time can be significantly improved by carefully considering how the initial positions of the robots affect their planned coverage paths.

We thus wanted to explore further the affect of the initial positions of robots on their performance. In the experiments above, the initial positions were randomly generated, and thus in the limit, their average position would have been a distance of $\frac{n}{k}$ from each other, i.e., the best case. However, real-



(a) Empty grid.

(b) Grid with obstacles.

Figure 6.1: Results of experiments with different MSTC coverage algorithms. Each data point is the average of 100 trials

world settings typically do not have the flexibility of landing robots in their perfect initial positions.

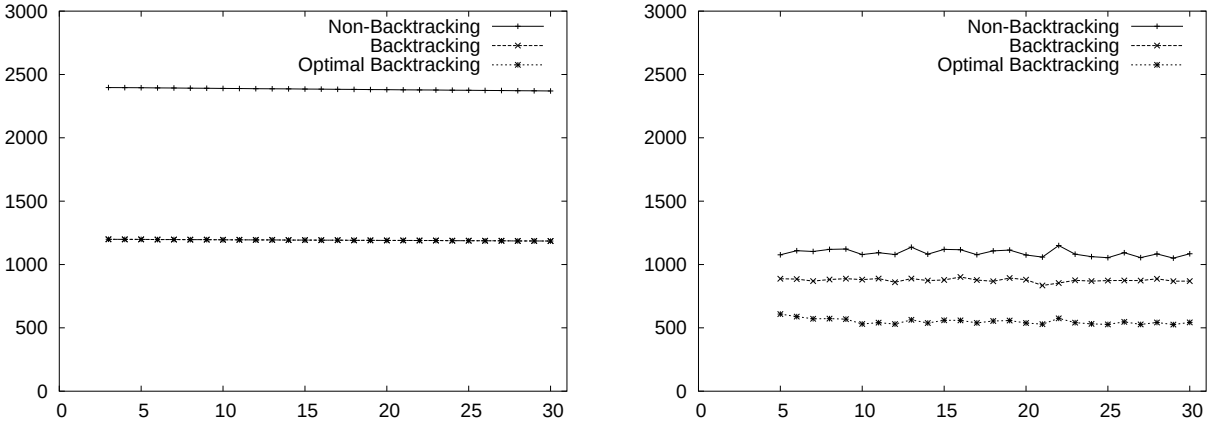
To better simulate real-world conditions, we ran a second set of experiments, where 3–30 robots were assigned to bases, and the number of bases (and their positions) were controlled. For a team of k robots, we allowed for b bases, where $1 \leq b \leq k$. We then split the k robots into the b bases, and randomly selected the position of each base. When $b = 1$, it is the worst case for the non-backtracking case (or close to it), where all robots start from the same position. When $b = k$, it is the case of the experiments above.

Figure 6.2 shows a subset of the results of these experiments. In all subfigures, the X axis shows the total number of robots in the bases, and the Y axis shows coverage time. In Figure 6.2-a, all robots leave from a single base. The two backtracking algorithms converge to a value much below that of the non-backtracking algorithm, whose faced with its worst case (approximately). In Figure 6.2-b, the performance of the three robots is clearly differentiated, yet in Figures 6.2-c,d the simple backtracking and non-backtracking algorithms converge (as we saw in the first set of results, above). In all figures, however, the optimal algorithm significantly outperforms its competitors.

6.2 On-line algorithm experimental results

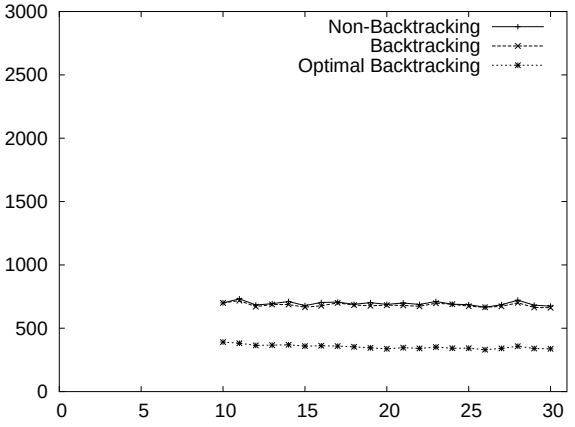
We conducted systematic experiments with our implementation of our on-line algorithm ORMSTC (Algorithm 11), to measure its effectiveness in practice with the RV400 robot. The experiments were conducted using the Player/Stage software package [11], a popular and practical development tool for real robots. Initial experiments were carried out with physical RV400 robots, to test the accuracy of the simulation environment used. However, to measure the coverage results accurately, the experiments below were run in the simulation environment. Figure 6.3 shows a screen shot of running example with six robots in the one of the simulated environments used in the experiments.

In the first experiment, we focused on demonstrating that the ORMSTC algorithm—and our im-

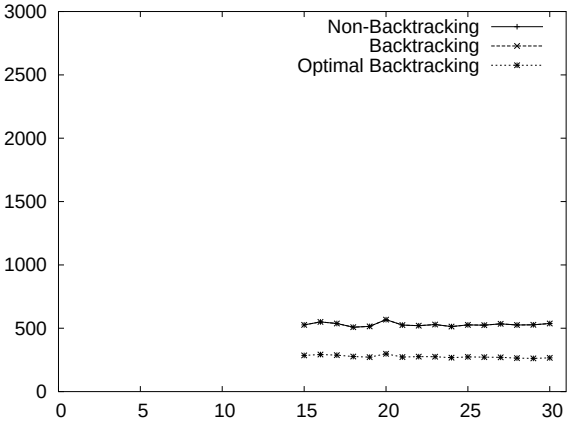


(a) $b = 1$.

(b) $b = 5$.



(c) $b = 10$.



(d) $b = 15$.

Figure 6.2: Coverage runtime when robots operate from b bases. Each base holds $\frac{k}{b}$ robots. Each data point is an average of 100 trials

.

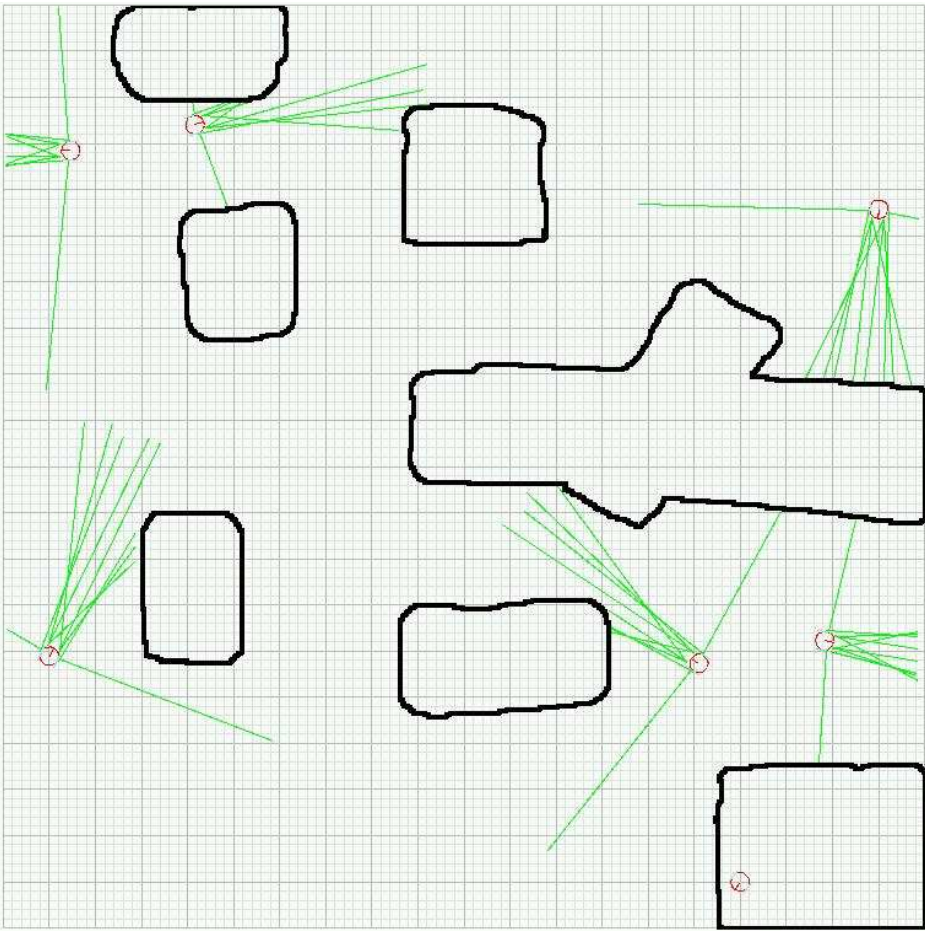


Figure 6.3: Simulation screen shot of six robots covering an outdoor environment

plementation of it for real robots—indeed manages to effectively use multiple robots in coverage. We run our algorithm with 2,4,6,8 and 10 robots. Each team was tested on two different environments: an outdoor and indoor environments. For each team size and environment type, 10 trials were run. The initial positions were randomly selected.

The results are shown in Figure 6.4. The X-axis measures the number of robots in the group. The Y-axis measures the coverage time. The two curves represent the two different environments. Every data point represents the average ten trials, and the horizontal line at each point is the standard deviation.

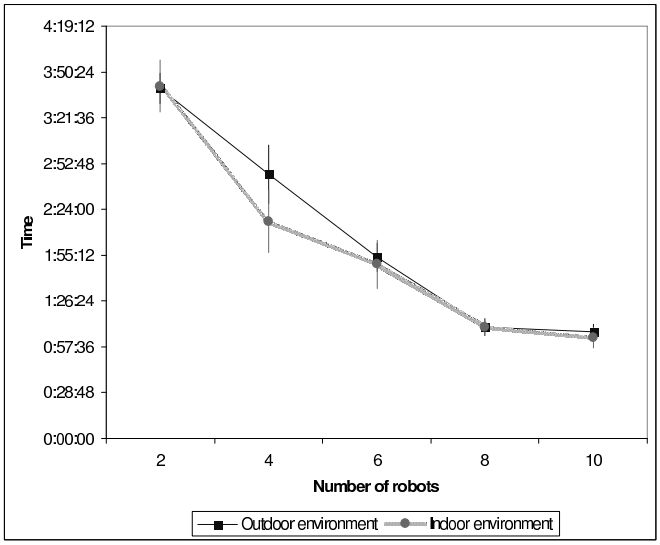


Figure 6.4: Overall coverage time

The results show that in both environments, coverage time decreases in general when increasing the group size. However, we can also see that the marginal coverage decreases with the addition of new members. This is a well-known phenomenon (in economics, but also in robotics [12, 23]). It is due to the overhead imposed on a bigger group of robots, in collisions avoidance and communication load. The overhead cost can be also seen when comparing the indoor and outdoor environment coverage times. Although the indoor environment is smaller than the outdoor one, the coverage time is almost the same because there are more obstacles and doors to pass and there is a bigger chance to collide with walls or team mates.

A second experiment compared the performance of ORMSTC with our off-line backtracking MSTC algorithm 5 in bad initial positions scenarios. In general, off-line algorithms have an advantage, in that they can optimize the generated paths for the obstacles, as these are known to them. To compute its performance, we calculated the path for a given initial position, and used the empirically-derived average velocity of the robot, to estimate the off-line coverage time.

We empirically discovered that when placed in off-line worst-case conditions (all the robots start at adjacent cells 3.2.1), the on-line algorithm performs significantly better as additional group members were used. This is in contrast to the off-line MSTC algorithm, whose performance does not improve noticeably as additional robots are used for coverage. The results are shown in Figure 6.5. Again, the

X-axis shows the number of robots in the group, while the Y-axis shows the coverage time. Every point is the average over 10 trials.

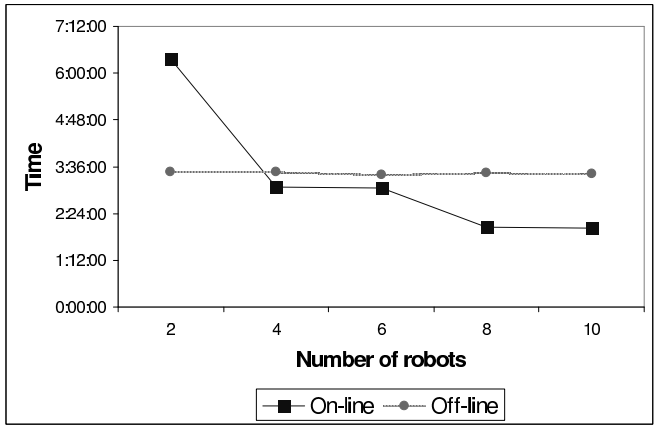


Figure 6.5: Coverage time of worst-case positions. Off-line results are calculated based on empirically-measured average velocity.

When comparing ORMSTC's performance in the worst-case initial positions with our previous results (Fig.6.4) it can be seen that it achieved longer coverage time. But when comparing it to the off-line algorithm, our algorithm performs better where there are more than two robots. The cause for this phenomena is that in contrast to the off-line algorithm the on-line spanning trees are being build dynamically along the algorithm execution, thus the robots are always trying to break into uncovered areas as long as they have ones.

Chapter 7

Conclusion and Future Work

Many real-world coverage applications require multiple robots to completely cover a given work-area. Some applications lend themselves easily to *off-line* coverage, where the robots are given a map of the work-area, and can therefore plan their paths ahead of deployment [4]. For instance, in many outdoor operations, aerial photos or maps might be available. Examples of such applications include de-mining operations [19], harvesting [20], or aerial surveillance. We presented a family of off-line algorithms and have shown that these algorithms are complete and robust in face of catastrophic individual robot failures. We examined the efficiency of these algorithms in terms of coverage time, and have shown that the initial positions of the robots have significant impact on the coverage time. In particular, while all algorithms carry the potential for best-case coverage in time n/k (where n is the number of cells, and k the number of robots), non-backtracking coverage has a worst-case time essentially equal to that of a single robot. Unfortunately, this is the common case where robots start right next to each other. In contrast, the backtracking algorithm is guaranteed to halve the coverage time of a single robot. We have also introduced a novel polynomial-time optimal backtracking coverage algorithm, capable of handling heterogeneous robots with different speeds and fuel capacity. We have shown in systematic experiments that its performance is a significant improvement over the simpler backtracking and non-backtracking algorithms.

However, many applications must utilize *on-line* coverage algorithms. Here, the robots cannot rely on apriori knowledge of the work-area, and must construct their movement trajectories step-by-step, addressing discovered obstacles (and/or collisions, in the case of multiple robots) as they move. Some applications that currently utilize on-line algorithms include vacuum cleaning [6], lawn mowing [15], and hazardous waste cleaning [13]. We presented the ORMSTC, an on-line multi-robot coverage algorithm and analytically showed that it is complete and robust in face of catastrophic robot failures. As there is always a gap between theory and practice, we analyzed the assumptions underlying the algorithmic requirements. We discuss various approximation techniques for these requirements, to allow the algorithm to work in real world situations. Based on early trials with real-robots, we conducted systematic experiments with our implementation, to measure the ORMSTC's effectiveness in practice. The results show that the algorithm works well in different environments and group sizes. ORMSTC

also seems to work well in some of off-line coverage worst cases.

In future work we intend to expand the algorithms we presented. Here we suggest some of the areas that should be explored:

Optimal backtracking algorithm run-time improvements. The optimal backtracking algorithm checks all options for an optimal assignment of paths, using a binary search. In its current implementation it may check the same case twice. For instance, one check for the optimal solution is when one robot goes to the half of the distance between it and its neighboring peer. This check is done for every robot for both directions, although the case when one robot goes this distance in one direction is identical to the case where its neighboring peer goes the same distance in the opposite direction. We would like to decrease the number of checks by identifying the repeated cases and check them once. We would also like to explore the relation between the given initial configuration and its optimal solution coverage time. We noticed that for some initial configurations there is more than one optimal solution. All of them have the same overall coverage time, but the assignment of paths among the robots is different. Sometimes there is only two solutions, but sometimes there are many optimal solutions. A better understanding of this link between the initial configuration and the number and structure of the optimal solutions could give as the possibility to forecast the expected optimal coverage time. This can improve our algorithm significantly, by reducing the number of options to check.

Classification of the optimal tree problem and our heuristic. We would like to establish the complexity class which the problem of building optimal tree belongs to. It is believed (strengthened by [30]) that this problem is $\mathcal{NP}$-Hard, although it is not proven yet. We would also like to guarantee something about the optimality of the heuristic presented in the appendix. Is it within a certain epsilon of the optimal?

On-line algorithm improvements. We would like to improve the algorithm to generate paths with less turns and to cover also cells which are partially covered by obstacles. Another important improvement is to enable the algorithm to work with less communication.

Effectiveness of our approximation techniques. We would like to empirically test the effectiveness of our approximation technics. We introduced some approximation technics and used them in our on-line algorithm implementation. We would like to run the algorithm without using these technics and compare the affect on the algorithm performance.

Repeated coverage We would like to explore how will our algorithms change if we redefine the problem such that:

- The objective is not to visit every cell at least once, but to visit every cell at least m times. Or
- The objective is to visit every cell with uniform frequency (the time between visits is equal).

Integration with search theory. One of the branches of operational research is the search theory. The aim of the algorithms from search theory is to find one or more targets in a specified area. They get information about the possibility of the targets to be in the different regions and have to decide at which order to search and how much time to spend at each region. One case is where the algorithm has to find all the targets but it does not get the information of how many targets exists and it has unlimited time. It can be seen that this is just another version of the our coverage problem. It may be interesting to combine their ideas with our algorithms to solve this type of problems.

Appendix A

Approximation for efficient spanning tree

In this appendix we describe the algorithm Create_Tree. This algorithm creates spanning trees while considering the initial location of all robots in the team and trying to minimize the maximal distance between any two adjacent robots on the tree. As before, N is the number of cells in the grid approximation and k is the number of robots.

The algorithm is composed of two stages. First, a subtree is created gradually for each robot starting from the initial position of the robot, such that in each cycle either one or two cells are added to each subtree. Denote the subtree originated in robot R_i by T_{R_i}. The cells are chosen in a way that maximizes the distance from current expansion of all other trees. First, the algorithm tries to find the longest possible path for the tree. When it fails to continue, it tries to perform Hilling, in which it looks for two joint unoccupied cells adjacent to the path. If it found such cells, then it adds them to the path as demonstrated in Figure A.1. If the algorithm failed to find more hills, then it expands the tree, from both sides of the path, as symmetrically as possible. First it attempts to add one cell to its right, then one cell to its left, and so on, until the entire grid is covered by all k disjoint subtrees.

After such k subtrees are generated, it is only left to connect them (second stage). Denote an edge connecting two different trees T_{R_i} and T_{R_j} by $bridge(T_{R_i}, T_{R_j})$. As we are given k subtrees to be connected to one tree covering the entire grid, it is required to find $k-1$ bridges. These bridges should be chosen in a way that the resulting tree does not contain cycles or, equivalently, cover the entire grid. For example, if $k = 4$ then possible valid choice of bridges are $\{bridge(T_{R_1}, T_{R_2}), bridge(T_{R_1}, T_{R_3}), bridge(T_{R_3}, T_{R_4})\}$, where $\{bridge(T_{R_1}, T_{R_2}), bridge(T_{R_2}, T_{R_3}), bridge(T_{R_1}, T_{R_3})\}$ is invalid, as T_{R_4} remains disconnected. Create_Tree picks a valid choice of $k-1$ bridges at random, and calculates the maximal sub-cells distance between two adjacent robots when moving along the tree. It repeats the process k^2 times, and reports the best tree it observed, according to the above criterion.

Clearly, the algorithm provides complete coverage of the terrain, as the first stage of constructing subtrees does not end before every cell is occupied by some subtree. The first stage terminates, as in each cycle at least one cell is added to at least one subtree, hence given a finite terrain the algorithm halts.

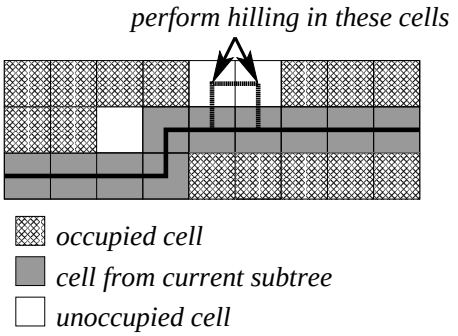


Figure A.1: Illustration of the Hilling procedure.

Algorithm Create_Tree **(work-area** W**, robots initial positions** $S_0, \ldots, S_{k-1}$**)**

1. Build k subtrees as follows.
   For every robot R_i, $1 \leq i \leq k$ **Do**:
   (a) **For** each possible next cell (up, down, right, left), compute the Manhattan distance from the current location of all other robots.
   (b) **If** more than one possible next move exists, **then** pick the one whose minimal distance to any other robot is maximized.
   (c) **If** there is no next possible move, **then** perform Procedure Hilling from the last main branch.
   (d) **If** failed to find an unoccupied cell in Hilling, **then** branch out, as symmetric as possible, from the main branch to all possible directions.
2. Find all possible bridges between the k trees.
3. **For** $i = 0$ to k^2 **Do**:
   (a) At random, find a valid set of bridges B_i between trees such that they create one tree with all N vertices.
   (b) Compute the set S_i of distances between every two consecutive robots on the tree.
   (c) $BestRes$ is initialized with S_0.
   (d) **If** the maximal value in S_i is lower than the maximal value in $BestRes$, **then** $BestRes \leftarrow S_i$.
4. Return the tree associated with $BestRes$.

Theorem A.0.1. *The time complexity of* Create_Tree *algorithm is* $\mathcal{O}(N^2 + k^2N)$.

Proof. In the stage where k subtrees are created, in the worst case when adding one cell to a subtree the algorithm runs over all current cells in the subtree (during Hilling or while branching out), hence the complexity is at most $\mathcal{O}(N^2)$. In the second stage, where the trees are connected, k^2 different choices of trees are examined, each time the entire tree is traversed, thus the complexity of this stage is $\mathcal{O}(k^2N)$. Hence the entire complexity of the algorithm is $\mathcal{O}(N^2 + k^2N)$. □

Bibliography

[1] E. U. Acar and H. Choset. Robust sensor-based coverage of unstructured environments. In *International Conference on Intelligent Robots and Systems*, pages 61–68, Maui, Hawaii, USA, 2001.

[2] N. Agmon, N. Hazon, and G. A. Kaminka. Constructing spanning trees for efficient multi-robot coverage. In *International Conference on Robotics and Automation*, 2006.

[3] Z. Butler, A. Rizzi, and R. L. Hollis. Complete distributed coverage of rectilinear environments. In *Workshop on the Algorithmic Foundations of Robotics*, March 2000.

[4] H. Choset. Coverage for robotics—a survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31:113–126, 2001.

[5] H. Choset and P. Pignon. Coverage path planning: The boustrophedon decomposition. In *International Conference on Field and Service Robotics*, Canberra, Australia, December 1997.

[6] J. Colegrave and A. Branch. A case study of autonomous household vacuum cleaner. In *AIAA/NASA CIRFFSS*, 1994.

[7] N. Correll, K. Easton, A. Martinoli, and J. Burdick. Distributed exploration and coverage. www.coro.caltech.edu.

[8] Friendly Robotics®, Ltd. Friendly robotics vacuum cleaner. http://www.friendlyrobotics.com/friendly_vac/.

[9] Y. Gabriely and E. Rimon. Spanning-tree based coverage of continuous areas by a mobile robot. *Annals of Mathematics and Artificial Intelligence*, 31:77–98, 2001.

[10] D. W. Gage. Command control for many-robot systems. In *The nineteenth annual AUVS Technical Symposium (AUVS-92)*, 1992.

[11] B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal, Jul 2003.

[12] D. Goldberg and M. J. Mataric. Interference as a tool for designing and evaluating multi-robot controllers. In *AAAI/IAAI*, pages 637–642, 1997.

[13] S. Hedberg. Robots cleaning up hazardous waste. *AI Expert*, pages 20–24, May 1995.

[14] S. Hert, S. Tiwari, and V. Lumelsky. A terrain-covering algorithm for an auv. In *Journal of Autonomous Robots Special Issue on Autonomous Underwater Robots*, volume 3, pages 91–119, 1996.

[15] Y. Huang, Z. Cao, and E. Hall. Region filling operations for mobile robot using computer graphics. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 1607–1614, 1986.

[16] D. Kurabayashi, J. Ota, T. Arai, and E. Yoshida. Cooperative sweeping by multiple mobile robots. In *Int. Conf. on Robotics and Automation*, 1996.

[17] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, July 1991.

[18] H. Moravec and A. Elfes. High resolution maps from wide angle sonar. In *IEEE Int. Conference on Robotics and Automation*, pages 116–121, St. Louis, March 1985.

[19] J. Nicoud and M. Habib. The pemex autonomous demining robot: Perception and navigation strategies. In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robot Systems*, pages 1:419–424, 1995.

[20] M. Ollis and A. Stentz. First results in vision-based crop line tracking. In *IEEE International Conference on Robotics and Automation*, 1996.

[21] I. Rekleitis, G. Dudek, and E. Milios. Multi-robot exploration of an unknown environment, efficiently reducing the odometry error. In *International Joint Conference in Artificial Intelligence (IJCAI)*, volume 2, pages 1340–1345, Nagoya, Japan, August 1997. Morgan Kaufmann Publishers, Inc.

[22] I. Rekleitis, V. Lee-Shue, A. P. New, and H. Choset. Limited communication, multi-robot team based coverage. In *IEEE International Conference on Robotics and Automation*, pages 3462–3468, New Orleasn, LA, April 2004.

[23] A. Rosenfeld, G. A. Kaminka, and S. Kraus. Adaptive robot coordination using interference metrics. In *Proceedings of the 16th European Conference on Artificial Intelligence*, August 2004.

[24] S. V. Spires and S. Y. Goldsmith. Exhaustive geographic search with mobile robots along space-filling curves. In *Proceedings of the First International Workshop on Collective Robotics*, pages 1–12. Springer-Verlag, 1998.

[25] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983.

[26] I. Wagner, M. Lindenbaum, and A. Bruckstein. Efficiently searching a graph by a smell-oriented vertex process. *Ann. Math. Artif. Intell.*, 24:211–223, 1998.

[27] I. Wagner, M. Lindenbaum, and A. Bruckstein. Distributed covering by ant-robots using evaporating traces. *IEEE Trans. Robotics Autom.*, 15(5):918–933, 1999.

[28] I. Wagner, M. Lindenbaum, and A. Bruckstein. Mac vs. pc determinism and randomness as complementary approaches to robotic exploration of continuous unknown domains. *International Journal of Robotics Research*, 19(1):12–31, 2000.

[29] A. Zelinsky, R. Jarvis, J. Byrne, and S. Yuta. Planning paths of complete coverage of an unstructured environment by a mobile robot. In *International Conference on Advanced Robotics*, pages 533–538, Tokyo, Japan, November 1993.

[30] X. Zheng, S. Jain, S. Koenig, and D. Kempe. Multi-robot forest coverage. In *Proc. IROS*, 2005.