Spring 2021

Multi-Robot Coordination with Environmental Disturbances

Adem Coskun

Multi-Robot Coordination with Environmental Disturbances

by

Adem Coskun

Bachelor of Science
Inonu University, 2008

Master of Science
Clemson University, 2013

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2021

Accepted by:

Marco Valtorta, Major Professor

Jason O'Kane, Committee Member

John Rose, Committee Member

Yan Tong, Committee Member

David Hitchcock, Committee Member

Tracey L. Weldon, Interim Vice Provost and Dean of the Graduate School

Dedication

This dissertation is dedicated to my beloved parents for their endless love.

Acknowledgments

I would like to express my grateful appreciation to my major advisor Prof. Marco Valtorta for his support.

I also would like to thank to my dissertation committee members, Dr. John Rose, Dr. Jason O'Kane, Dr. Yan Tong, and Dr. David Hitchcock for their valuable suggestions.

Last but not the least, I would like to thank my beloved wife for her support and patience.

Abstract

Multi-robot systems are increasingly deployed in environments where they interact with humans. From the perspective of a robot, such interaction could be considered a disturbance that causes a well-planned trajectory to fail. This dissertation addresses the problem of multi-robot coordination in scenarios where the robots may experience unexpected delays in their movements.

Prior work by Čáp, Gregoire, and Frazzoli introduced a control law, called RMTRACK, which enables robots in such scenarios to execute pre-planned paths in spite of disturbances that affect the execution speed of each robot while guaranteeing that each robot can reach its goal without collisions and without deadlocks. We extend that approach to handle scenarios in which the disturbance probabilities are unknown when execution starts and are non-uniform across the environment. The key idea is to 'repair' a plan on-the-fly, by swapping the order in which a pair of robots passes through a mutual collision region (i.e. a coordination space obstacle), when making such a change is expected to improve the overall performance of the system. We introduce a technique based on Gaussian processes to estimate future disturbances, and propose two algorithms for testing, at appropriate times, whether a swap of a given obstacle would be beneficial. Tests in simulation demonstrate that our algorithms achieve significantly smaller average travel time than RMTRACK at only a modest computational expense.

However, deadlock may arise when rearranging the order in which robots pass collision regions and other obstacles. We provide a precise definition of deadlock using a graphical representation and prove some of its important properties. We show how

to exploit the representation to detect the possibility of deadlock and to characterize conditions under which deadlock may not occur. We provide experiments in simulated environments that illustrate the potential usefulness of our theory of deadlock.

Table of Contents

List of Figures

Chapter 1

Introduction

As multi-robot systems become more reliable and more widespread, it is becoming increasingly common for them to share their operating environments with humans. For example, it is more likely that we will have more household cleaning robots in our houses. Also, it seems that we will have more multi-robot systems in the era of Industry 4.0 [40], such as, for example, warehouse management systems as pointed out in [65]. Moreover, the number of autonomous cars is increasing every day on the roads and coordinating those autonomous cars can help to reduce collisions and ensure that they reach their destinations faster. Coordinating multi-robot systems in environments like the ones just described when humans are present can be a challenging problem for several reasons. One specific issue is that the motions of the robots may be interrupted or delayed by humans. In such a scenario, the robot may be prevented from progressing along its path for some period of time, an event we refer to as a *disturbance*.

Prior work by Čáp, Gregoire, and Frazzoli [12] showed how to handle these kinds of unexpected disturbances effectively, by introducing an approach that first generates a suite of trajectories that is collision free in the absence of disturbances, and then controls the forward movements of the robots to ensure that the coordinated motions remain free of both collisions between robots and of deadlocks, even if some or all of the robot experience disturbances. This control rule is called Robust Multi-Robot Trajectory Tracking Strategy (RMTRACK).

The essential idea of the RMTRACK approach is for a robot to stop and wait, before entering a portion of its path on which a collision with another robot might occur, if the other robot would have passed through this collision region first, according to the original (undisturbed) trajectories.

The RMTRACK approach is very effective, especially in scenarios where the expected amount of disturbance experienced by each robot is approximately equal. The approach shows its limitations, however, in scenarios where the disturbance probabilities are unknown at the start and non-uniform across the environment.

For example, consider the simple problem illustrated in Figure 1.1. Two robots attempt cross the room, one from left to right and the other from right to left. The robots are named R_1 and R_2 respectively. Without loss of generality, suppose that the initial planned trajectory instructs R_1 to pass through the narrow central region first.

However, unbeknown to the robots and the trajectory planner, at the time of plan execution the left side of the room is filled with human workers that interrupt the motion of that robot. Then, R_2 reaches the collision region first. Using the RMTRACK algorithm, robot R_2 would wait at that position until R_1 fully navigated the left side and passed through the collision region. Clearly, robot R_2's wait is a waste of time.

We propose to resolve this kind of problem by repairing the plan on-the-fly. In the example of Figure 1.1, when robot R_2 arrives the intersection and robot R_1 has not yet cleared the intersection, robot R_2 has a choice: Does it wait until the robot R_1 clears the intersection, or does it continue forward, hoping to pass through the collision region before the robot R_1 arrives? We propose a two-phase approach to answering this question: First we estimate the probability of each robot experiencing disturbances along the relevant section of its path, based on disturbances observed by the robots in those regions during the current execution. Then, using those estimated

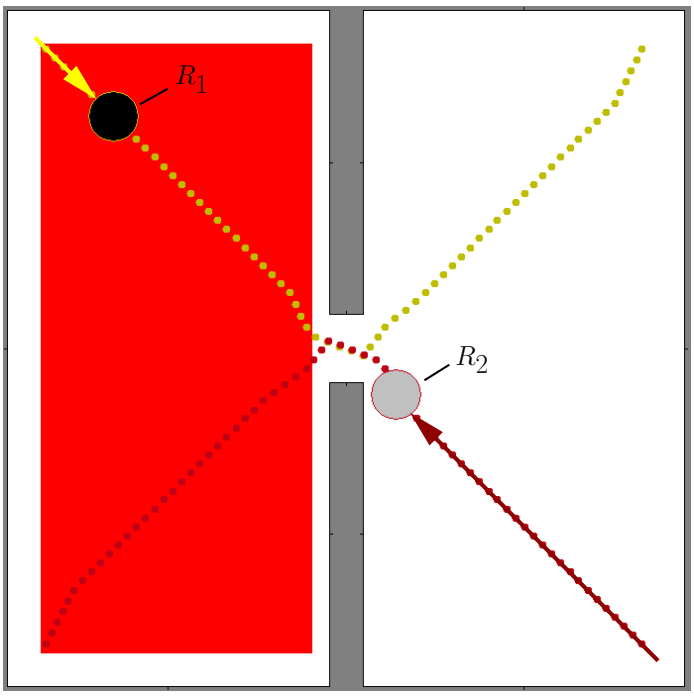


Figure 1.1: Two robots, R_1 and R_2, are attempting to move from one side to another side of the environment. Dotted lines show the path of the robots, and arrows point from each robot's start position to its current position. The probability of having disturbance in the red zones is 0.8. R_2 needs to wait until R_1 passed through the collision region.

probabilities, robot R_2 can decide whether it is likely to safely pass through the collision region before the arrival of robot R_1. If so, we 'flip' that obstacle and resume the RMTRACK controller, at which point robot R_2 will continue through the collision region immediately.

1.1 Survey

The problem of coordinating multiple robots in a shared workspace is one of the best studied problems in the field. Approaches for this problem are generally classified as either *reactive* or *planning*.

1.1.1 Reactive Approach

Each robot follows its own shortest path, and collisions are resolved locally by observing the other robots. These algorithms are practicable because of the computational efficiency. However, they do not guarantee that all robots reach their goal positions.

One of the first proposed reactive techniques is called the cocktail party model [46]. In this model, each robot knows its own current and goal position, but there is no communication between robots, and the only information from the other robots comes from their sensors when they are nearby. Their algorithm is based on maze-searching techniques. The term, the cocktail party model, is inspired by the behavior of a guest in a crowded place. When a guest wants to talk to someone from another table, he dynamically plans his movements by travelling between all tables, chairs, and other guests with a minimal distance. If the guest senses that one person is drunk, then the guest increases the distance from that person for safety.

Also, there are other reactive techniques [60, 33, 59] based on the velocity obstacle (VO) approach [25], so the collisions are avoided observing not only the positions of the robots but also their velocities. Among them, the optimal reciprocal collision avoidance (ORCA) [59] formulation is used in practice due to its efficiency in calculating the velocity obstacles. A generalized velocity obstacle approach for non-holonomic robots is proposed in [1, 5].

The main disadvantage of the reactive approaches is the possibility of having a deadlock [18, 30]. Figure 1.2 illustrates two deadlock scenarios in multi-robot systems. Deadlock avoidance [2, 41] is also studied for traffic systems in [49, 50, 51] by using a resource allocation system (RAS), and in operating systems where a resource allocation graph is sometimes used, as described in [58].

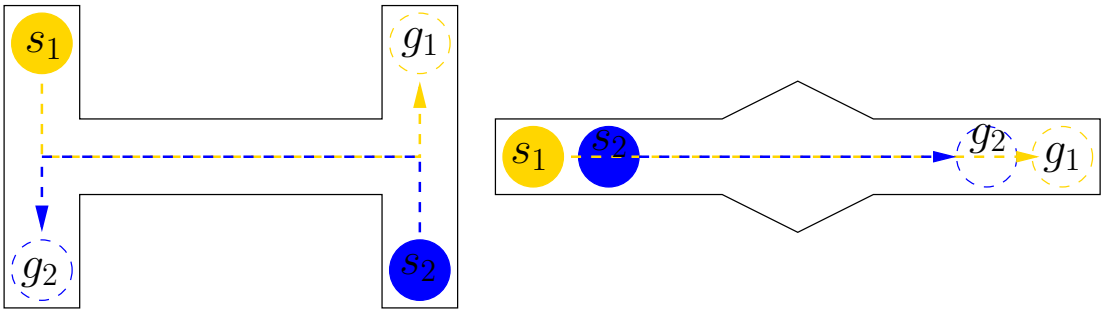


Figure 1.2: Two deadlock examples for reactive approaches. In the left environment, the robots are moving at the same speed and therefore occupy the narrow corridor at the same time. In the right environment, the reactive behavior of the robots prevents then from recognizing the need to switch positions in the wider part of the corridor. Both deadlock cases can be avoided by using a planning approach.

1.1.2 Planning Approach

In the planning approach, all robot trajectories are generated by planning the coordination between each robot before the robots start to execute their paths. These approaches guarantee that all robots reach their goal positions. However, their complexity increases exponentially with the number of coordinated robots. According to [55], planning for circular robots moving amidst static polygonal obstacles is strongly NP-hard. When planning for rectangular robots moving in a rectangle environment, it is PSPACE-hard [37], which is also the case for square robots [54].

A configuration space [44] for a robot represents all points that the robot can reach, so planning for a robot is basically finding a path between its start and goal positions in the configuration space. A configuration space for a multi robot system is the Cartesian product of the configuration spaces for each robot. In order to reduce the complexity of the planning approach, the path-velocity decomposition described in [38] is used. The decomposition consists of planning the path to avoid collisions with static obstacles, and planning the velocity to avoid collisions with dynamic obstacles. Such decomposition leads to the notion of *coordination space* originally

proposed in [45], with the geometry-based approach described in [42], and widely used, as described in [44, 26, 32, 53, 8, 15, 27, 17, 28, 16].

We can differentiate planning globally with a joint configuration space, or separately for each robot. In the literature, the two approaches are referred to as *coupled* [35, 57, 56, 63, 64, 24, 48] and *decoupled* [23, 61, 6, 13, 14].

Also, we can define *centralized* approaches, in which all information is contained in a central computer that computes plans and distributes them to each robot. Approaches that are not centralized are called *decentralized* [62]. Decentralized approaches can be used within decoupled algorithms, but one disadvantage of the decoupled algorithms is that they are incomplete, which means they may fail to find trajectories even though they exist.

Moreover, there are autonomous intersection management systems for multi agent systems [22, 20, 21, 36, 3] based on the reservation-based approach. In this approach, each agent (typically a vehicle) sends a request to a central agent, which is an intersection manager. Then the intersection manager decides to reserve a space-time region in the intersection. Clearly, this a special kind of centralized approach, in which coordination is addressed only at special locations (the intersections).

Planning under uncertainty and sequential decision making is also studied as decision-theoretic planning, and modeled as a Markov decision process (MDP) or dynamic Bayesian network (DBN) [11, 9, 10, 31]. Bayesian network approaches are used to predict the movement of the dynamic surroundings for collision avoidance under uncertainties. In [43], a dynamic Bayesian network, which is a directed graphical model with a conditional probability distribution for each child node, is used for maneuver prediction. In [39], an object-oriented Bayesian network (OOBN) is used to predict lane change maneuvers. Similarly, maneuver prediction with traffic interaction is studied in [29, 52, 4].

Another planning approach is categorized as Multi-Agent Path Finding (MAPF). In this approach, the problem is represented with a graph where each vertex corresponds to an agent's location and edges correspond to transitions between two locations. The agents can stay on their current vertex with a wait action. Otherwise, they move to their next vertices with a move action. Having a random delay on the agent's execution is studied in [47], and more recently an Action Dependency Graph (ADG) is proposed in [7] for reordering each agent's execution schedule when an agent has a large delay.

The most closely related work, that of Čáp, Gregoire, and Frazzoli [12], is specifically targeted to enable the reliable execution of centrally-generated joint plans, in cases where the robots cannot necessarily follow those paths without temporal disruption. In our work, we extend from that previous work, considering a similar problem, but allowing the robots greater freedom to adjust the plan based on conditions observed during the actual execution.

1.2 Summary of Revised Prioritized Planning

This section briefly describes revised prioritized planning, which was proposed and studied by Čáp in [13]. This planning approach is used to generate trajectories of the robots for the simulation.

Classical prioritized planning is efficient when used in practical applications because the trajectories of each robot are planned one after another instead computing all robots at once. However, classical prioritized planning is incomplete because it may fail to find trajectories for the robots. For example, a state-space based planner will find trajectories for the scenario shown on the right part of Figure 1.2, but classical prioritized planning will fail.

Classical prioritized planning assigns each robot a distinct priority. Then, each robot's trajectory is computed in such a way that it avoids the higher priority robots'

trajectories. The algorithm finds a solution if all robots' trajectories are generated successfully. However, the algorithm may fail to find a solution in some cases such that the robots cannot avoid the trajectory of one or more other robots, so that it fails even though there exists a solution.

There are two types of scenarios where classical prioritized planning fails. They are named Type A and Type B in [13]. Type A is illustrated in Figure 1.3 where robot 1 has the higher priority and blocks robot 2 when it reaches its goal position. This situation can be avoided, for example, if robot 2 has a trajectory where it can go around the goal position of robot 1.

Type B is illustrated in Figure 1.4 where the trajectory of robot 2 overlaps that of robot 1 while robot 1 is moving. This happens mainly because robot 1 is faster than robot 2 and robot 1 has a higher priority. This situation can be avoided, for example, if robot 1 has a trajectory where it can go around the start position of robot 2, so robot 2 can wait at its start position and then it can move after robot 1 does not block robot 2's trajectory. If robot 2 can go around the goal position of robot 1, then a scenario of Type A can also be avoided.

According to [13], *a valid infrastructure* is an environment where the following conditions can be satisfied: all possible goal positions of higher priority robots can be avoided by the lower priority robots, and all possible start positions of lower priority robots can be avoided by the higher priority robots. A valid infrastructure is illustrated in Figure 1.5. If those conditions are satisfied within an environment, then revised prioritized planning is guaranteed to find the trajectories for each robot as proved in [13].

On the other hand, revised prioritized planning has some limitations. Recall the scenario depicted in Figure 1.4. In this case, assume that two robots have the same speed. Then, it is obvious that revised prioritized planning algorithm fails because robot 1 cannot avoid the start position of robot 2, but classical prioritized planning

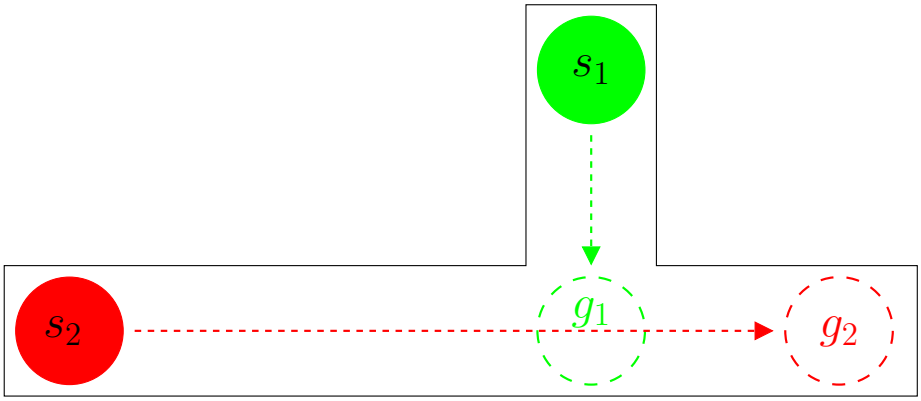


Figure 1.3: The first robot, colored green, has a higher priority, and its start and goal positions are represented as s_1 and g_1, respectively. Also, s_2 and g_2 represent the start and goal positions of the second robot, colored red. The robots travels at the same speed. Since the first robot has a higher priority, the path of the second robot is not considered by the first robot. Thus, when the first robot reaches the goal position, it blocks the second robot. This is a Type A scenario. Figure adapted from [13].

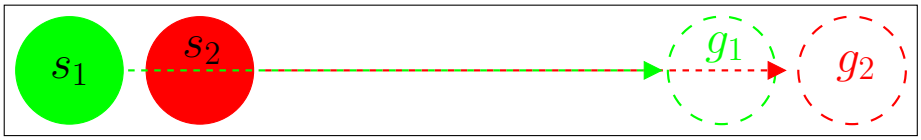


Figure 1.4: The start and goal positions of two robots are depicted in the environment. The first robot, colored green, travels at twice the speed of the second robot, colored red. Also, the first robot has a higher priority, so there is no trajectory for the second robot to avoid the conflict of the first robot's path. Thus, classical prioritized planning fails. Figure adapted from [13].

successfully finds the trajectories for both robots by allowing the two robots to move straight with the same speed. Moreover, classical prioritized planning can find shorter trajectories than revised prioritized planning. For example, in Figure 1.6, the first robot has a curved trajectory instead of a straight one. Because of revised prioritized planning condition, it needs to avoid the start position of the second robot.

Overall, it has been proved in [13] that revised prioritized planning always finds a solution within a valid infrastructure environment, but we also need to consider some limitations or disadvantages when comparing revised prioritized planning with classical prioritized planning.

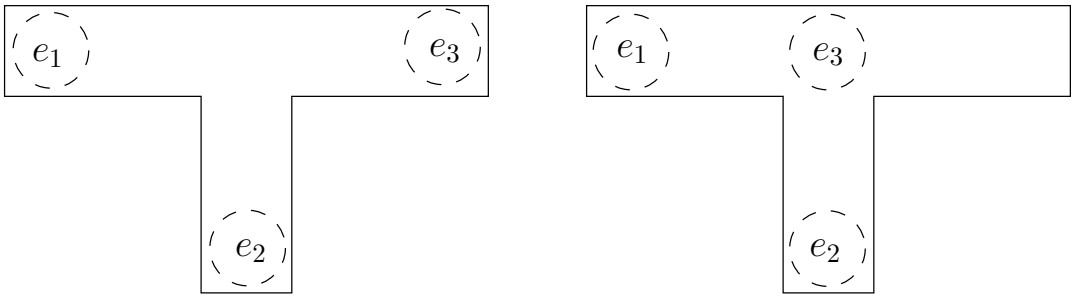


Figure 1.5: e_1, e_2, and e_3 represent endpoints, which are the possible start or goal positions of some robots. The environment depicted on the left is an example of a valid infrastructure. However, the environment depicted on the right is not a valid infrastructure because e_3 blocks a path between e_1 and e_2. For example, assume that e_1 and e_2 are the start and goal positions for the higher priority robot, respectively. Also, assume that e_2 and e_3 are the start and goal positions for the other lower priority robot, respectively. Then, it is not possible that the higher priority robot avoids the start position of the lower robot, e_3. Thus, this environment depicted on the right is not a valid infrastructure.

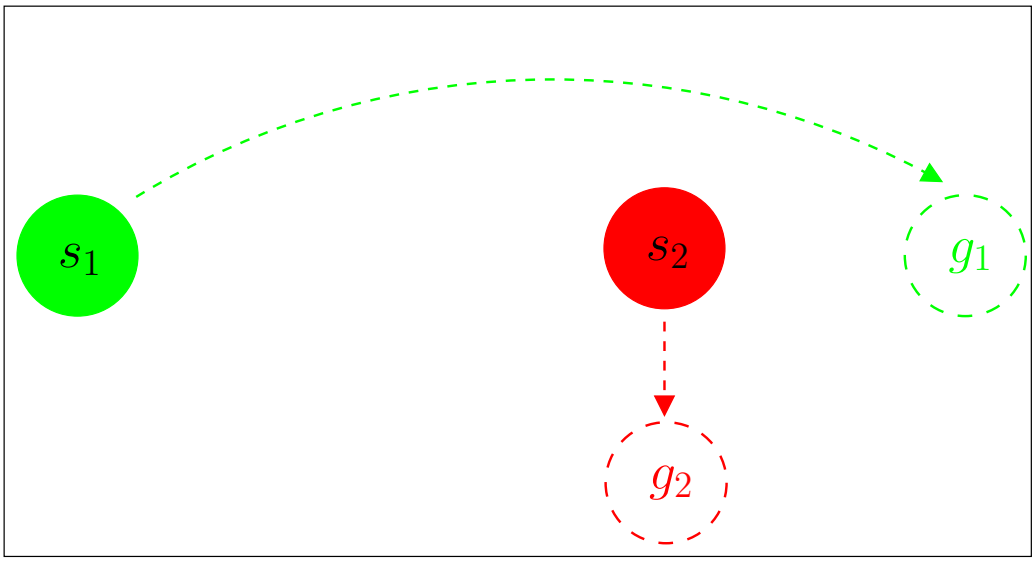


Figure 1.6: Revised prioritized planning generates a longer trajectory for the first robot, colored green, than classical prioritized planning, because the first, higher priority, robot must go over the start position of the second, lower priority, robot. Figure adapted from [13].

Chapter 2

Problem Statement

This chapter formalizes the problem we address in this paper. The treatment is based upon work by Coskun and O'Kane [19] and generalizes the model used by Čáp, Gregoire, and Frazzoli [12].

2.1 Environment, Robots, and Trajectories

We assume that n identical holonomic robots, indexed $1, \ldots, n$, operate in a shared 2d environment, $\mathcal{W} \subseteq \mathbb{R}^2$. The robots are disc-shaped with body radius r. We model time as a sequence of discrete stages indexed by $t \in \mathbb{N}$. Each robot starts at a *start position* and travels within $\mathcal{W}$ to a *goal position*. We assume that feasible collision-free trajectories for each robot, $\pi_1, \ldots, \pi_n$, from their respective start positions to their goals are generated by a multi-robot trajectory planner, such as one that uses prioritized planning [13]. Each trajectory $\pi_i : \{1, \ldots, K_i\} \to \mathcal{W}$ is a function mapping integers to locations in the environment, in which trajectory π_i for robot i has K_i steps. We model the robots' execution of these paths in discrete time, writing $x_i(t)$ to denote number of steps of π_i executed by robot i up to time t. If robot i experiences a disturbance or a delay in its execution, we will have $x_i(t) < t$. Thus, the actual position of robot i at time t is $\pi_i(x_i(t)) \in \mathcal{W}$.

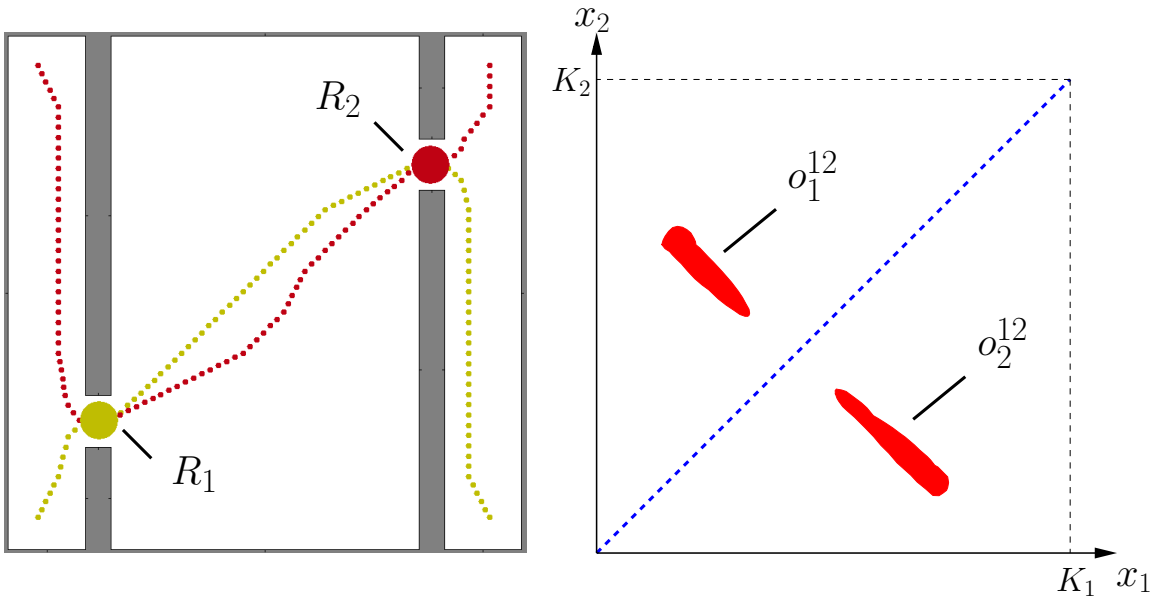


Figure 2.1: Two robots, R_1 and R_2, in an environment where depicted on the left have two collision regions. The dotted lines are representing their paths. The coordination space of these two robots is depicted on the right with two obstacles, o_1^{12} and o_2^{12}. The blue dotted line represent the planned path. The label of $\ell(o_1^{12})$ is 1, which means R_1 should pass the obstacle o_1^{12} first. Also, the label of $\ell(o_2^{12})$ is 2, which means R_2 should pass the obstacle o_2^{12} first.

2.2 Coordination Spaces

For each pair of distinct robots (i, j), we define the *coordination space* $C_{ij} \subseteq C$ as

$$C_{ij} = \{(k_i, k_j) \mid ||\pi_i(k_i) - \pi_j(k_j)|| \geq 2r\}. \tag{2.1}$$

The intuition is that a single point in C_{ij} is determined by the positions of robots i and j along their paths, and that pairs of positions that would place robot i in collision with robot j are excluded from the coordination space. See Figure 2.1 for an example. Within each coordination space C_{ij}, we can identify the obstacle region $O_{ij} = \{1, \ldots K_i\} \times \{1, \ldots, K_j\} - C_{ij}$. We partition O_{ij} into maximal connected regions, $o_1^{ij}, \ldots, o_m^{ij}$, so that $O_{ij} = o_1^{ij} \cup \cdots \cup o_m^{ij}$. Each o_k^{ij} is called a *coordination space obstacle*.

Each coordination space obstacle represents a region in the environment that both robots must pass through, but in which a collision may possibly occur if both robots

occupy it at the same time. Notice the execution of robots i and j generates a path through C_{ij} from $(1,1)$ to (K_i, K_j) and that the main diagonal in coordination space always corresponds to the planned path. For each obstacle o_k^{ij}, this path must pass either above o_k^{ij} or below o_k^{ij}. The former case corresponds, in the workspace (i.e., the real environment), to robot j passing through the collision region before robot i; in the latter case, robot i passes through the collision region before robot j.

In addition to the trajectories $\pi_1, \ldots, \pi_n$, we also assume that the trajectory planner assigns to each coordination space obstacle o_k^{ij} a *label* $\ell(o_k^{ij}) \in \{i, j\}$, indicating which of the two robots is planned to pass through the collision region first. Robot i passes through a collision region first and robot j passes second, if the corresponding obstacle in coordination space is above the main diagonal. Similarly, robot j passes through the collision region first if the corresponding obstacle is below the main diagonal.

In one of the examples, depicted in Figure 2.2, two robots are shown in six different positions, located in the environment and coordination space. Robot 1 passes the collision region first, so the label of the obstacle, $\ell(o_1^{12})$, is 1, and the corresponding path in coordination space, the blue dashed line, passes below the obstacle. In Figure 2.2 (a), the robots execute two path steps from their start position so far, and robot 1 is one step away from the collision region as it approaches it. In Figure 2.2 (b), robot 1 enters the collision region, but robot 2 is still three steps away from the collision region. In Figure 2.2 (c), robot 1 is almost passed the collision region, and robot 2 has almost arrived the the collision region. In Figure 2.2 (d), robot 1 has just passed the collision region, and robot 2 just arrives. In Figure 2.2 (e), robot 1 is two steps past the collision region, and robot 2 is on the center of the collision region. In the last frame, Figure 2.2 (f), robot 2 also has passed the collision region.

In one of the another examples, depicted in Figure 2.3, robot 2 follows robot 1 in a narrow corridor as shown. Robot 1 is one step away from robot 2, so robot 2 may

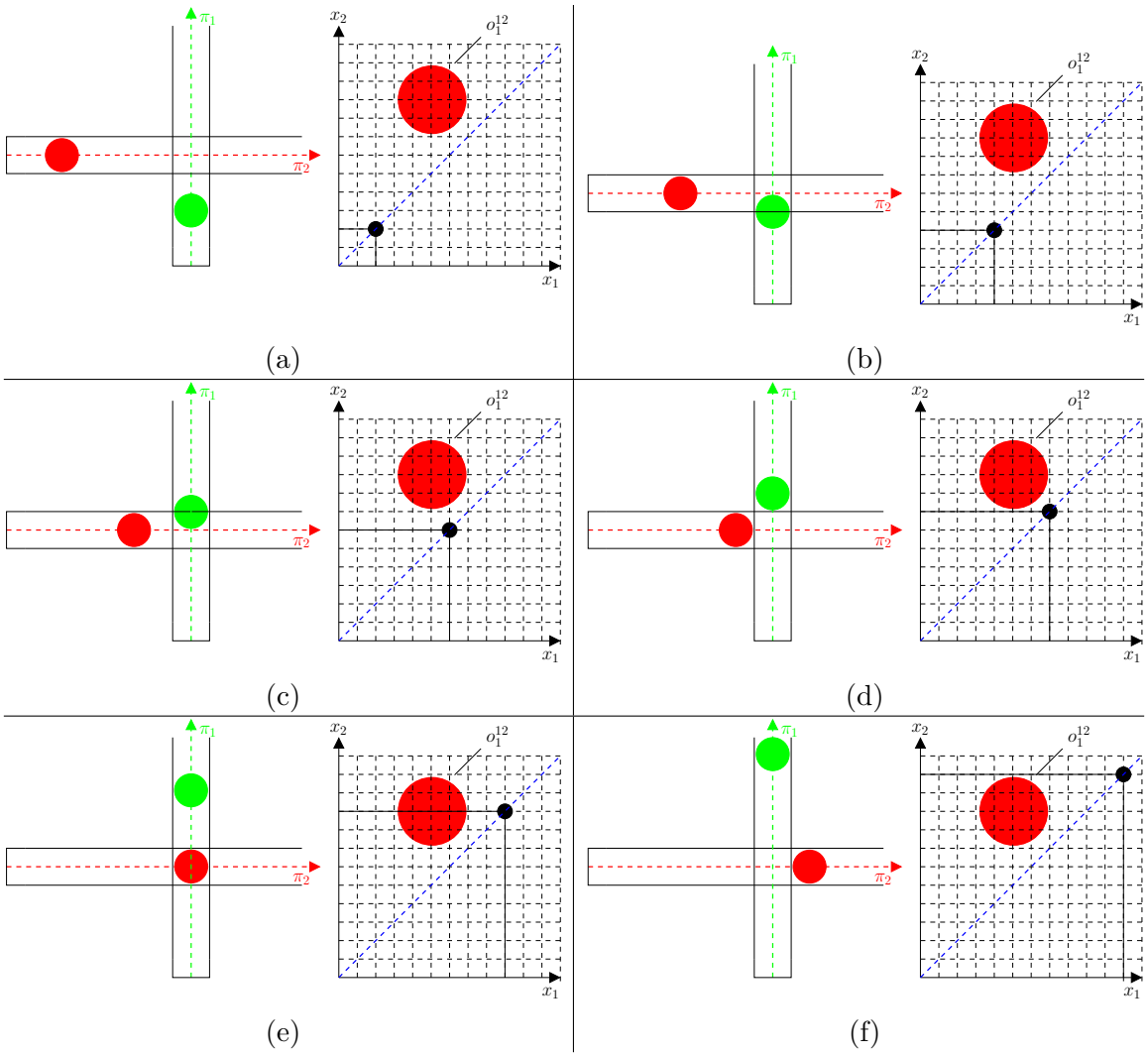


Figure 2.2: On the left side of the figures, two robots are shown traveling through two intersecting narrow corridors, according to trajectories π_1 and π_2, respectively. On the right side of the image, the position of the robot pair is a function of the coordinates (x_1, x_2) in the coordination space. The axes are indexed by path steps. It is assumed that the origin is the beginning of the path. The corresponding positions of the robots in the coordination space is shown by the black point, and the dashed blue line shows the planned trajectory.

collide with robot 1 when it gets closer more than one step. The corresponding coordination space obstacle and three different positions of the robots in the environment and coordination space are illustrated in the sub-figures.

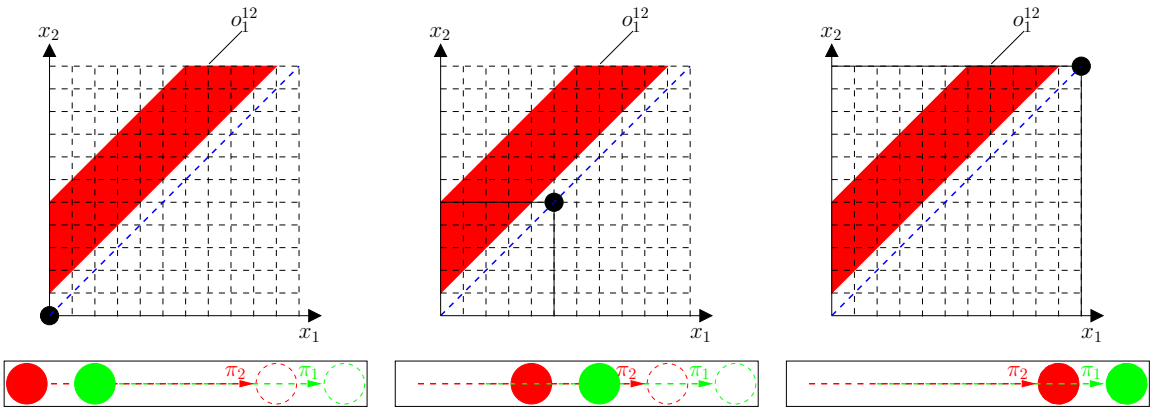


Figure 2.3: On the bottom side of the figures, two robots are shown following each other in a narrow corridor, according to trajectories π_1 and π_2, respectively. On the top of the figures, the corresponding positions of the robots are represented in the coordination space.

2.3 Commands and Disturbances

At each time step, each robot may decide to attempt to either move forward along its path, or to voluntarily remain where it is. If the robot decides to move forward, that movement may be prevented by a *disturbance* of some kind from within the environment. We model these options using a control variable $a_i : \mathbb{N} \to \{0, 1\}$ and a disturbance variable $\delta_i : \mathbb{N} \to \{0, 1\}$ for each robot. Then each robot's progress through its path is governed by the transition equation

$$x_i(t + 1) = x_i(t) + a_i(t)\delta_i(t). \tag{2.2}$$

We assume that the robots are subject to a disturbance at each point $q \in W$, which is modeled by a Bernoulli probability distribution whose parameter is a function of position, and the probability of any robot experiencing a disturbance at position q is $p(q)$. The function p is unknown to both the robots and the trajectory planner. For simplicity, we assume that the same p governs the disturbances for all of the robots, and that p does not vary as time passes.

2.4 Objective

The goal is to establish an efficient control strategy for each robot i to select $a_i(t)$ at each time t. The control strategy should ensure that the robots do not collide with each other, and that all of the robots reach their goals, that is, there exist some time t such that $x_i(t) = K_i$ for all robots $i = 1, \ldots, n$.

In other words, the control strategy decides for each robot i, at each time t, whether to move or wait (that is, $a_i(t) = 0$ or $a_i(t) = 1$), in such a way that each robot reaches its goal, without any collisions between robots, in minimal total time.

2.5 Summary of RMTRACK

In this section, we summarize the existing RMTRACK algorithm.

Our description of RMTRACK necessarily differs from that of Čáp, Gregoire, and Frazzoli because their formulation parameterizes the configuration space in a way that ensures that its diagonal is collision-free, which implies that the obstacle labels can be inferred by whether each obstacle is above or below the diagonal. Since we intend to modify the obstacle labels during execution, we introduce the following functionally equivalent presentation of RMTRACK.

The control law for RMTRACK, which for robot i at time t selects $a_i(t)$, is:

$$a_i(t) = \begin{cases} 0 & \exists j \neq i, \text{ s.t. } \exists k : \ell(o_k^{ij}) = j \text{ and} \\ & o_k^{ij} \cap (\{x_i(t)+1\} \times \{x_j(t), ..., K_j\} \neq \emptyset \\ 0 & \text{if } x_i(t) = K_i \\ 1 & \text{otherwise} \end{cases} \tag{2.3}$$

The top portion of Figure 2.4 illustrates the intuition. If there exists at least one coordination space obstacle o_k^{ij} representing a collision region that robot j should pass through first, that is, for which $\ell(o_k^{ij}) = j$, then robot i may need to wait for

robot j to pass. To determine whether this is the case, we extend a line segment upward in C_{ij} from the next position along the path for robot i, and check whether this line segment intersects with o_k^{ij}. If so, then robot i should stop and wait for robot j to make some progress, ensuring that the robots' path passes above o_k^{ij} in C_{ij}. Naturally, when robot i has completed its path, that is, when $x_i(t) = K_i$, it should stop. If neither of these two stopping conditions holds, then robot i chooses $a_i(t) = 1$, attempting to make progress toward its goal.

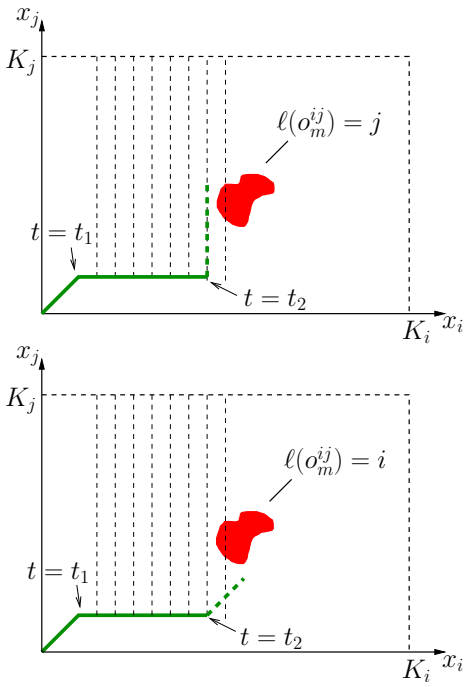


Figure 2.4: An illustration of the behavior of RMTRACK. Robot i and robot j share a collision region in the coordination space C_{ij}. In this example, robot j begins to experience a lengthy disturbance starting at time t_1. The path through this coordination space until time t_2 is shown in green; the dotted green lines show possible future trajectories for the robots. The key question is: What should robot i do at time t_2? [top] If the obstacle o_k^{ij} has label $l(o_k^{ij}) = j$, then robot j is planned to pass through this collision region first. Equivalently, the coordination space path should travel over o_k^{ij}. Robot i must wait, choosing $a_i(t_2) = 0$. This wait, shown in C_{ij} as upward vertical movement, lasts until robot j has advanced far enough to clear o_k^{ij}. [bottom] If the obstacle o_k^{ij} has label $l(o_k^{ij}) = j$, then robot i is planned to pass through this collision region first; the coordination space path should travel under o_k^{ij}. Robot i can continue immediately, choosing $a_i(t) = 1$, without regard to the progress of robot j.

Chapter 3

TestFlipFast and TestFlipAggressive Algorithms

This chapter describes our approach. The essential motivation can be seen on the left side of Figure 2.4. In this example, robot j has experienced a lengthy disturbance, whereas robot i has been able to progress through its path steadily. Notice that the original, offline trajectory planner formed a global plan in which robot j should cross the collision region before robot i. This decision was reasonable in the absence of disturbances, but disturbance probabilities across the environment are unknown when the plan is generated. Consequently, as it executes the RMTRACK control law (Equation 2.3), robot i will reach the start of the collision region within its path, and then wait until robot j overcomes its disturbances to pass first.

One readily notices, however, that if robot j's progress has been slowed much more than that of robot i, then *robot i might attempt to pass this collision region immediately*, thereby 'flipping' the coordination space obstacle from $l(o_k^{ij}) = j$ to $l(o_k^{ij}) = i$. The right part of Figure 2.4 illustrates the result of this change: Robot i continues to use RMTRACK to govern its movements, but because of the altered obstacle label, robot i can proceed immediately. By the time robot j finally reaches this region, it is likely that robot i will be safely out of the way.

The essence of our approach is to detect when opportunities for these kinds of on-the-fly changes to the coordination space obstacle labels may be beneficial to the overall performance of the system. We note that the alternative of simply re-executing

the global trajectory planner in such situations is not generally a feasible option, since that sort of joint planning scales, as a general rule, quite poorly as the number of robots increases.

3.1 When to Check for Obstacle Flips

Before we address the question of how to determine *if* flipping an obstacle might be helpful, we first consider *when* during their execution the robots might reasonably consider this kind of change. Recall that the advantage of an obstacle flip derives from enabling a robot whose progress might have been delayed because of the first case in Equation 2.3 to proceed immediately instead of waiting for the other robot to pass a certain collision region. Thus, robot i performs a flip check at most once for each obstacle o_k^{ij}, specifically the first time that obstacle triggers the first condition in Equation 2.3.

3.2 Estimating the Disturbance Probabilities

Our goal is to change the label of a coordination space obstacle, allowing a robot to pass through without waiting, only when doing so is unlikely to delay the other robot. To make such a decision requires an estimate of the disturbance probability function p, at positions along each of the two robots' paths, from their current positions through to the end of the collision region. We write $\hat{p} : W \to [0, 1]$ to denote this estimate of the disturbance probability.

The robots use their own observations of the actual disturbances, realized during the current execution, to compute $\hat{p}$. Each robot keeps track of its last s time steps, in which s is a tunable parameter, and tracks both its position π_i and whether it experienced a disturbance δ_i, in those time steps. Based on those observations, robot i can compute a position-probability pair, which estimates the probability of a

disturbance at the centroid of its positions across the last s time steps:

$$\left(\pi_i\left(\left\lfloor\frac{1}{s}\sum_{i=0}^{s-1}x_i(t-i)\right\rfloor\right),\frac{1}{s}\sum_{i=0}^{s-1}\delta_i(t-i)\right) \tag{3.1}$$

The system then uses these estimates of p at various places within W, to form its global estimate $\hat{p}$, using a Gaussian Process regression model. We also use a g-means clustering approach [34] to reduce the size of the observation set, to ensure that the Gaussian Process learning is completed efficiently. Figure 3.2 illustrates an example of this process.

For the Gaussian Process Regression, we also need to tune the regularization parameter, λ. We calculate the RMSE (root mean squared error) to validate the model with different values of λ, so that we can estimate the ideal value of λ which produces a model that generalizes well to new, previously unseen data. Figure 3.3 illustrates the Gaussian Process Regression with different values of the regularization parameter λ; the value 0.1 was chosen for all our simulations, because it always resulted in the smallest RMS error.

3.3 Testing Whether Flipping an Obstacle Is Helpful

Finally, we can establish conditions under which we expect the average travel time for the robots to benefit from changing the label of one of the obstacles on-the-fly. We propose two methods for this. The first method, TestFlipFast (Section 3.3.2) is very efficient, but overly conservative for some types of coordination space obstacles; the second, TestFlipAggressive (Section 3.3.3) is more computationally expensive, but can identify flipping opportunities overlooked by the first method. Throughout this section, we consider the case in which robot i has begun to wait for robot j because of obstacle o_k^{ij} with label $l(o_k^{ij}) = j$; the question is whether to change this label to i. First, in Section 3.3.1 we derive the calculation of the expected travel time.

Figure 3.1: The robot moves through two regions, shown in red, in which the probability of disturbance is elevated. The robot does not know of these regions beforehand, and must estimate the disturbance probability based on its own experience of disturbances.

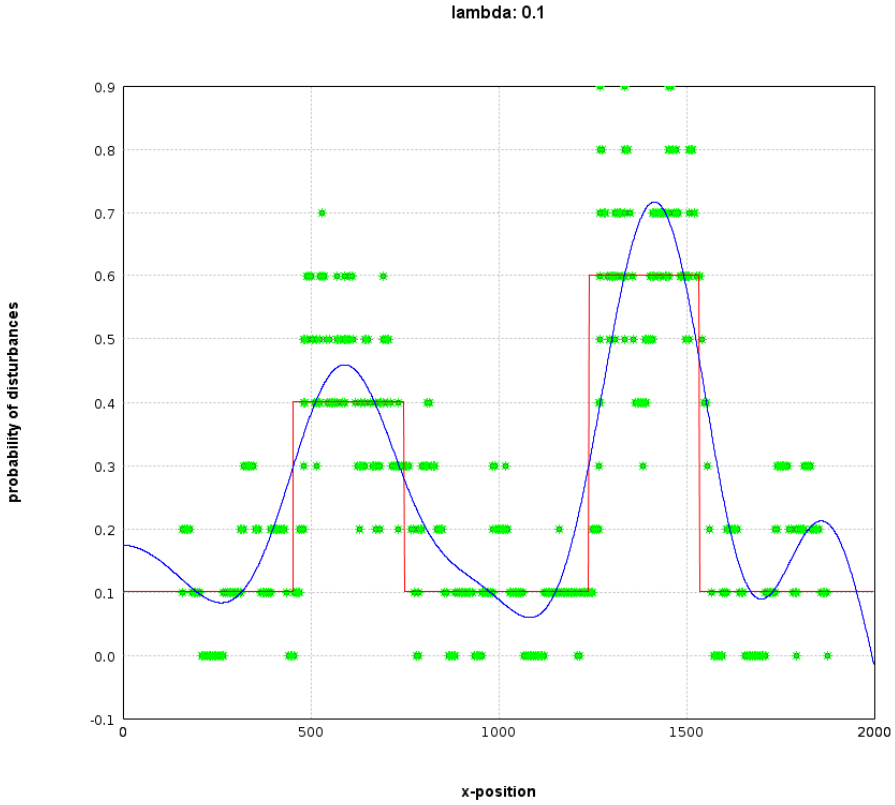


Figure 3.2: Results of the process of estimating the disturbance probability. Green points mark the observations, computed via Equation 3.1. The blue curve shows $\hat{p}$, as computing via Gaussian Process regression over these observations. For comparison, the actual disturbance probability p is plotted in red. Note that this illustration shows only a one-dimensional slice of the estimated disturbance probability function $\hat{p}$, along the robot's actual path. Our approach computes $\hat{p}$ across the full 2-dimensional domain.

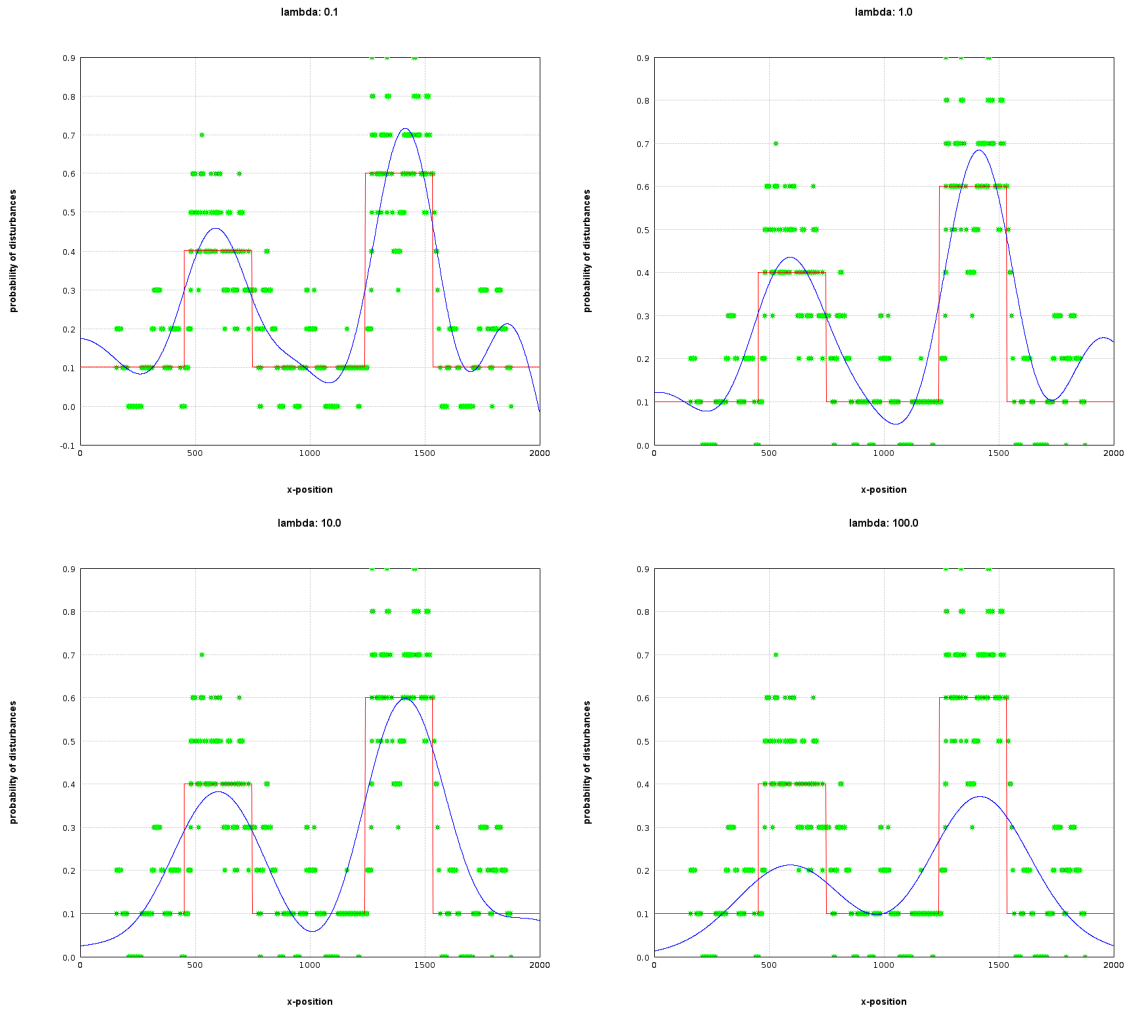


Figure 3.3: [top-left] if $\lambda = 0.1$, then $RMSE = 0.131$. [top-right] if $\lambda = 1$, then $RMSE = 0.135$. [bottom-left] if $\lambda = 10$, then $RMSE = 0.149$. [bottom-right] if $\lambda = 100$, then $RMSE = 0.222$.

3.3.1 How to Calculate The Expected Travel Time

Let assume robot i is at path step k_i, the predicted probability of having a disturbance at the current path step, k_i, be $\hat{p}(k_i)$. Then, the predicted probability of not having a disturbance at the current path step, k_i, is $1 - \hat{p}(k_i)$. Also, let t be the unit time step, and let the random variable $X = \{t, 2t, 3t, ...\}$ be the set of possible travel times to reach the next path step, k_{i+1}.

With probability $1 - \hat{p}(k_i)$, the travel time to reach the next path step is one t. This means that the robot did not have a disturbance, and it moved at its first attempt.

$$P(X = t) = \hat{p}(k_i)^0(1 - \hat{p}(k_i))$$

With probability $\hat{p}(k_i).(1 - \hat{p}(k_i))$, the travel time to reach the next path step is two t. This means that the robot could not move because of a disturbance at first, and then moved.

$$P(X = 2t) = \hat{p}(k_i)^1(1 - \hat{p}(k_i))$$

With probability $\hat{p}(k_i).\hat{p}(k_i).(1 - \hat{p}(k_i))$, the travel time to reach the next path step is three t. It means the robot had two time disturbances, then reached the next step.

$$P(X = 3t) = \hat{p}(k_i)^2(1 - \hat{p}(k_i))$$

In general, we can express the probability of having nt travel times to reach the next path step as following (the robot has $(n-1)$ times disturbance before moving):

$$P(X = nt) = \hat{p}(k_i)^{(n-1)}(1 - \hat{p}(k_i))$$

Thus, the expected travel time to reach the next path step, k_{i+1} is:

$$\begin{aligned} E_i(k_i) &= t(1 - \hat{p}(k_i)) + 2t\hat{p}(k_i)(1 - \hat{p}(k_i)) + 3t\hat{p}(k_i)^2(1 - \hat{p}(k_i)) + \dots \\ &= t(1 - \hat{p}(k_i)) \sum_{n=1}^{\infty} n\hat{p}(k_i)^{n-1} \end{aligned} \tag{3.2}$$

Then, by using the derivative of the summation of geometric series, which is

$$\sum_{n=1}^{\infty} n.x^{n-1} = \frac{1}{(1-x)^2}, |x| < 1,$$

the expected travel time is reduced to the following equation:

$$E_i(k_i) = t(1 - \hat{p}(k_i))\frac{1}{(1 - \hat{p}(k_i))^2} \tag{3.3}$$
$$= \frac{t}{1 - \hat{p}(k_i)}$$

This formula allows constant-time estimation of travel time.

3.3.2 TestFlipFast - RMTRACK+TFF

Our first method decides to flip a coordination space obstacle if the expected time for robot i to clear o_k^{ij} is less than the expected time for robot j to arrive at o_k^{ij}.

The idea to making this approach efficient is to consider only the axis-aligned bounding box of o_k^{ij}, rather than its precise shape. This simplifying assumption means that we can consider the movements of robot i independently of those of robot j. Let k_i^{clear} be the path step at which robot i clears the obstacle, and k_j^{reach} be the path step at which robot j reaches the obstacle. Then the first method decides in favor of flipping if Equation 3.4 holds.

$$\sum_{k=k_i}^{k_i^{clear}} E_i(k) < \sum_{k=k_j}^{k_j^{reach}} E_j(k) \tag{3.4}$$

Figure 3.4 illustrates the TestFlipFast method.

3.3.3 TestFlipAggressive - RMTRACK+TFA

We also consider an alternative to TestFlipFast, which considers the interactions between robot i and robot j as they travel near the obstacle. These kinds of interactions are important if, for example, the obstacle is long, narrow, and diagonal in the coordination space, as would occur if the paths for robot i and robot j travel in parallel for some distance. See Figure 3.5. Using TestFlipFast would be unlikely to flip such an obstacle, since the time at which robot i fully clears the obstacle will be far in the future. Figure 3.6 illustrates this case.

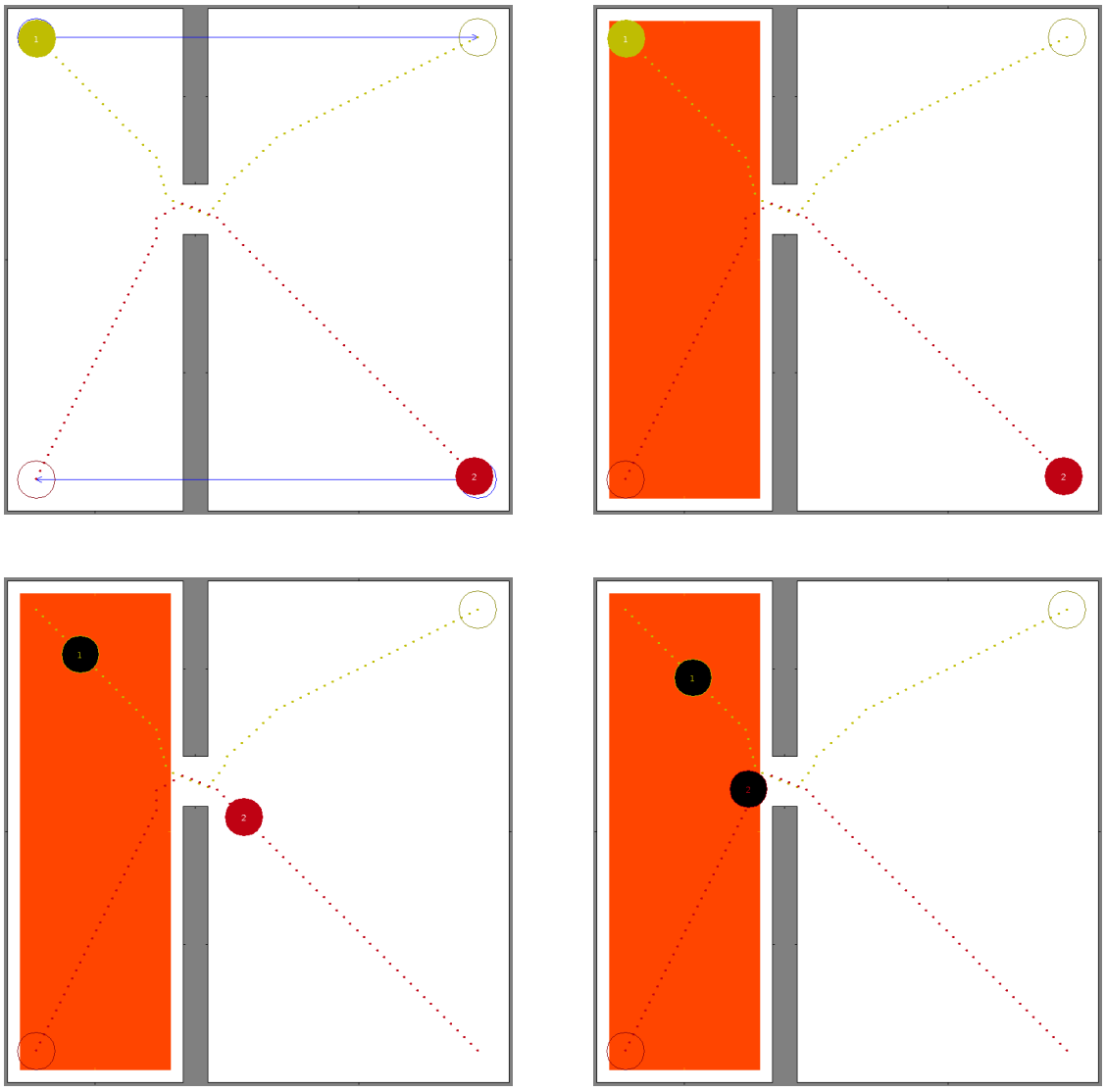


Figure 3.4: [top-left] Robot 1 moves from the top left corner to top right corner. Robot 2 moves from bottom left corner to bottom right corner. Also, robot 1 should pass the intersection first. [top-right] The probability of having disturbances in the red zones are assigned as 0.7. The disturbance probability everywhere else is 0.02. [bottom-left] Since robot 1 experienced much disturbance, it did not reach and pass the intersection yet. Therefore, robot 2 reaches the intersection first and checks if the expected time to clear the intersection is less than the expected time for robot 1 to arrive the intersection. [bottom-right] RMTRACK+TFF method allows robot 2 to pass the intersection first instead of waiting until robot 1 clears the intersection.

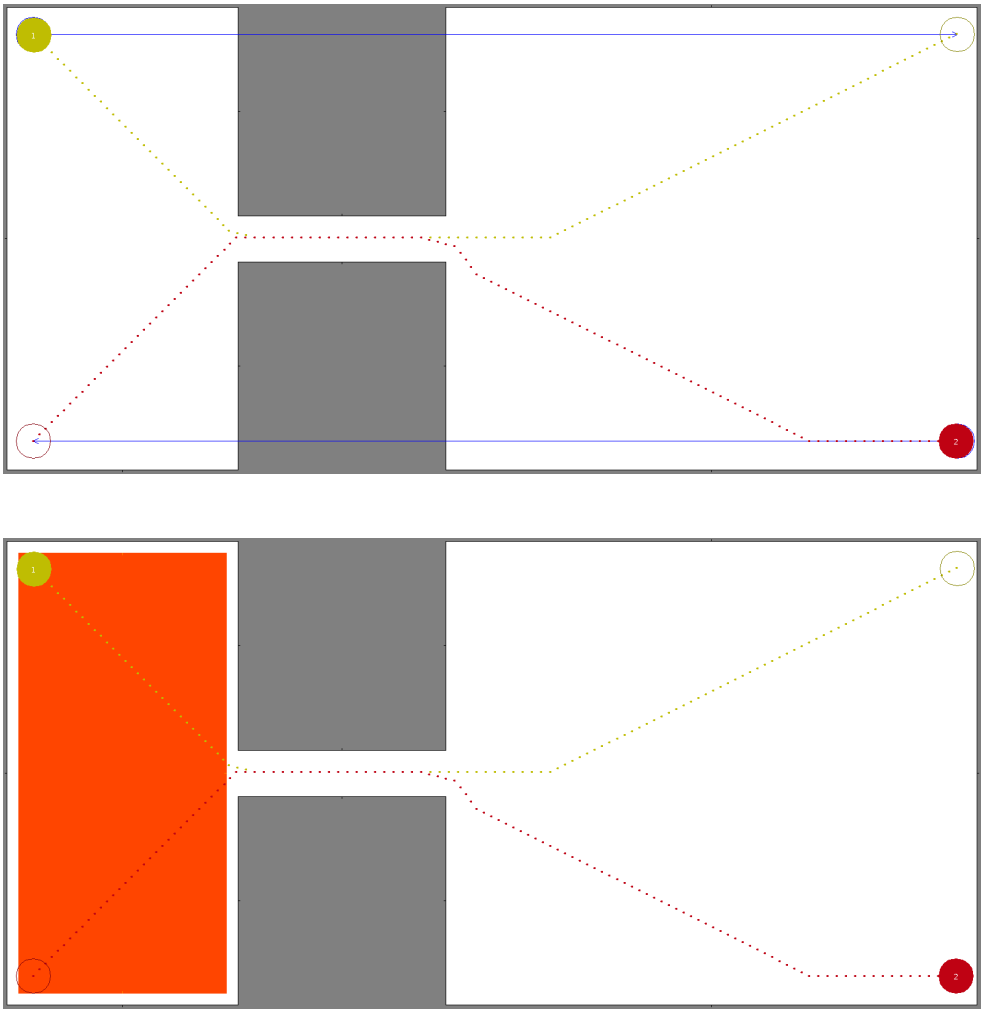


Figure 3.5: [top] Robot 1 moves from the top left corner to top right corner. Robot 2 moves from bottom left corner to bottom right corner. Also, robot 1 should pass the intersection first. The probability of having disturbances in the red zones are assigned as 0.7. The disturbance probability everywhere else is 0.02. [bottom] Since robot 1 had many disturbances, it did not reach and pass the intersection yet. Therefore, robot 2 reaches the intersection first and checks if the expected time to clear the intersection is less than the expected time for robot 1 to arrive at the intersection.

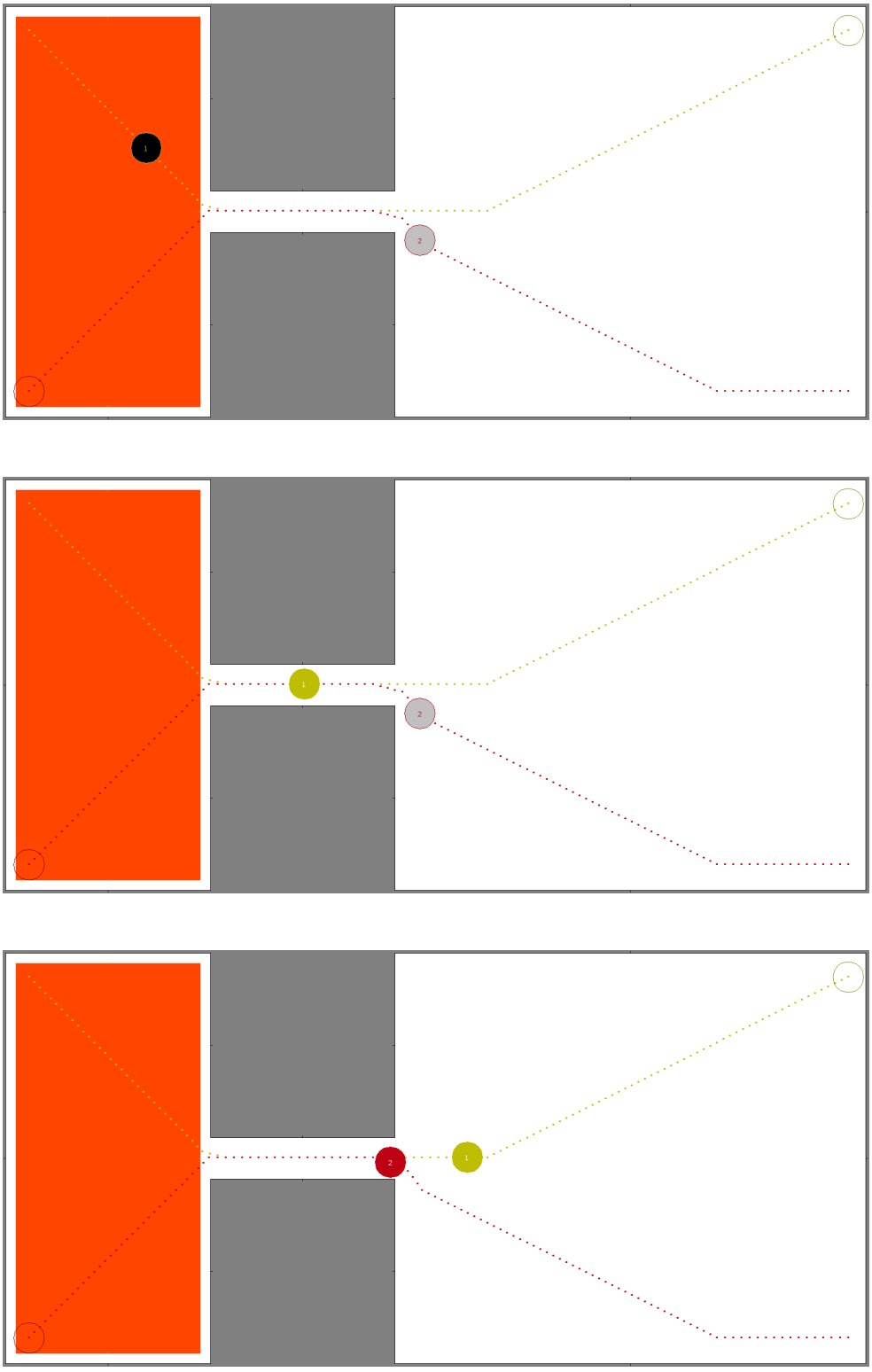


Figure 3.6: [top and middle] In this case, RMTRACK+TFF method does not allow robot 2 to pass the intersection first, so robot 2 waits until robot 0 clears the intersection. [bottom] When robot 1 clears the intersection, then robot 2 starts to move right after robot 0.

To account for those kinds of interactions, we propose TestFlipAggressive as an alternative. The core calculation is denoted as $E_{ij}(k)$, which represents the expected travel time for robot i to clear the obstacle, accounting for the fact that its motion may be delayed, according to Equation 2.3. Because of the additional states (since we consider joint positions of both robot i and robot j, rather than robot i individually) and because of the time needed to evaluate Equation 2.3 at each point, this approach can be slower than the expected time computation in FlipCheckFast.

Finally, we can use these sorts of expected time computations to decide whether to flip obstacle o_k^{ij}. Because we want to consider the effects of this obstacle's label, we compute four different expected times:

- The expected time for robot i to clear o_k^{ij}, using the current label:

$$\sum_{k=k_i}^{k_i^{clear}} E_{ij}(k)$$

- The expected time for robot i to clear o_k^{ij}, using the opposite label, which it would acquire if it were flipped:

$$\sum_{k=k_i}^{k_i^{clear}} \mathcal{I}_{ij}(k)$$

- The expected time for robot j to clear o_k^{ij}, using the current label.

$$\sum_{k=k_j}^{k_j^{clear}} E_{ji}(k)$$

- The expected time for robot j to clear o_k^{ij}, using the opposite label, which it would acquire if it were flipped.

$$\sum_{k=k_j}^{k_j^{clear}} \mathcal{I}_{ji}(k)$$

Using these estimates of the consequences of an obstacle flip, we choose to carry out that flip if the anticipated benefit (that is, the anticipated reduction in travel time)

for robot i outweighs the anticipated cost to robot j. That is, if

$$\sum_{k=k_i}^{k_i^{clear}} E_{ij}(k) - \sum_{k=k_i}^{k_i^{clear}} \mathcal{I}_{ij}(k) > \sum_{k=k_j}^{k_j^{clear}} \mathcal{I}_{ji}(k) - \sum_{k=k_j}^{k_j^{clear}} E_{ji}(k)$$

or

$$\sum_{k=k_i}^{k_i^{clear}} \mathcal{I}_{ij}(k) + \sum_{k=k_j}^{k_j^{clear}} \mathcal{I}_{ji}(k) < \sum_{k=k_i}^{k_i^{clear}} E_{ij}(k) + \sum_{k=k_j}^{k_j^{clear}} E_{ji}(k) \tag{3.5}$$

then we change the label of o_k^{ij}. Figure 3.7 and Figure 3.8 illustrates the TestFlipAggressive method.

That completes our discussion of the approach. In summary, as the robots execute RMTRACK, the system attempts to identify times at which it can opportunistically modify the label of an obstacle, to repair the initial trajectory, to recover from large, unexpected disturbances with full replanning.

3.4 Experimental Results

We have implemented these algorithms in Java, building upon the original RMTRACK implementation.[1] For Gaussian Process regression, we use the Statistical Machine Intelligence and Learning Engine (SMILE).[2] The experiments were conducted on an Ubuntu 20.04 computer with a 2.9GHz processor.

We conducted experiments in two distinct environments, each with certain large regions designated as high-disturbance zones. The environments are shown in Figure 3.9; the high disturbance zones are shown in red. The disturbance probability in these red zones are 0.85. Outside the red zones, the disturbance probability is 0.05.

For each environment, we varied the number n of robots in increments of 5 and selected random starting and goal positions for each robot. For each n, we conducted twenty trials, executing three algorithms for each configuration of state and

[1] `https://github.com/mcapino/rmtrack`

[2] `https://haifengl.github.io/`

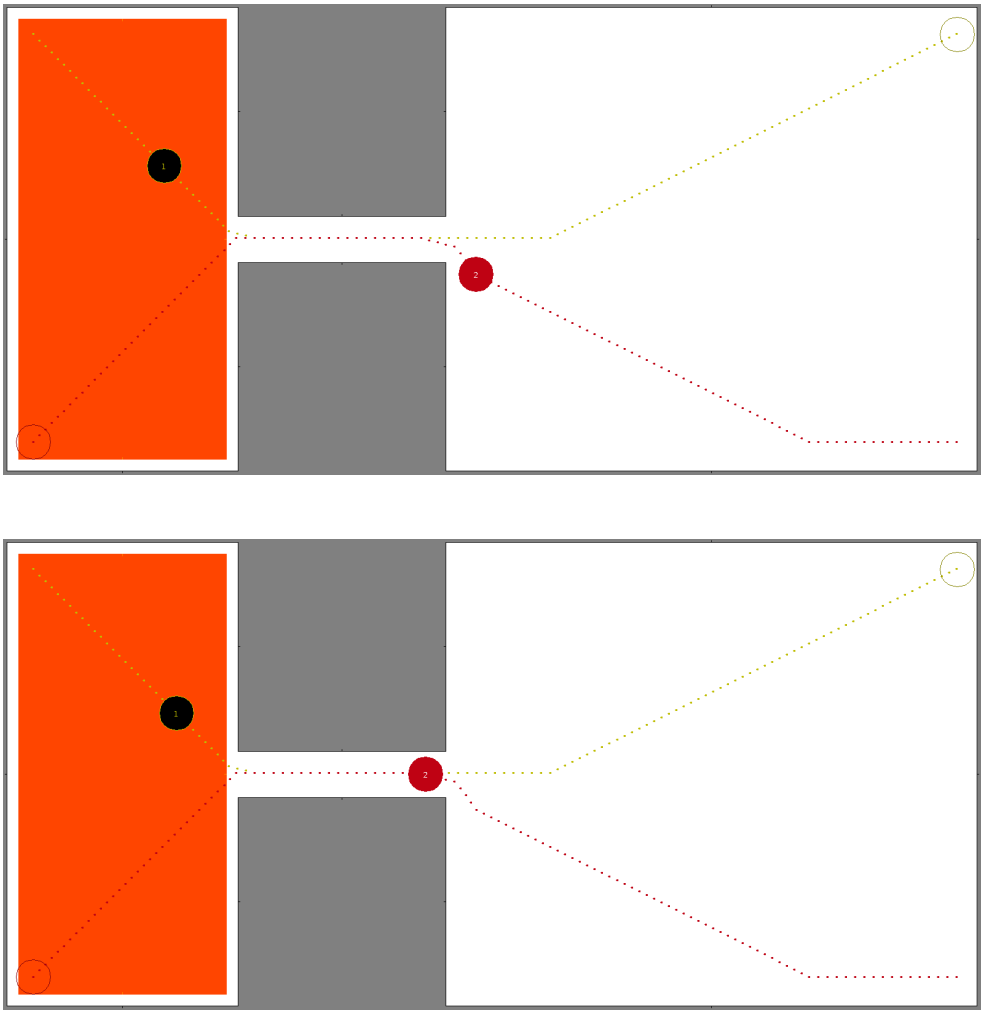


Figure 3.7: RMTRACK+TFA method allows robot 2 to pass the intersection first because the anticipated benefit is bigger than anticipated cost.

goal positions: (1) vanilla RMTRACK, (2) RMTRACK with obstacle flipping via TestFlipFast (RMTRACK+TFF), and (3) RMTRACK with obstacle flipping via TestFlipAggressive (RMTRACK+TFA).

For each algorithm, we measured the average completion time for the robots that completed each trial successfully, i.e., without having deadlock. These results appear in Figure 3.10. We observe that, for these problem instances, both obstacle flipping approaches can generate sizable decreases to the average travel time for the robots. We also computed the standard deviation for the same (environment, algorithm) pairs. We observe that in almost all cases the standard deviation, shown in

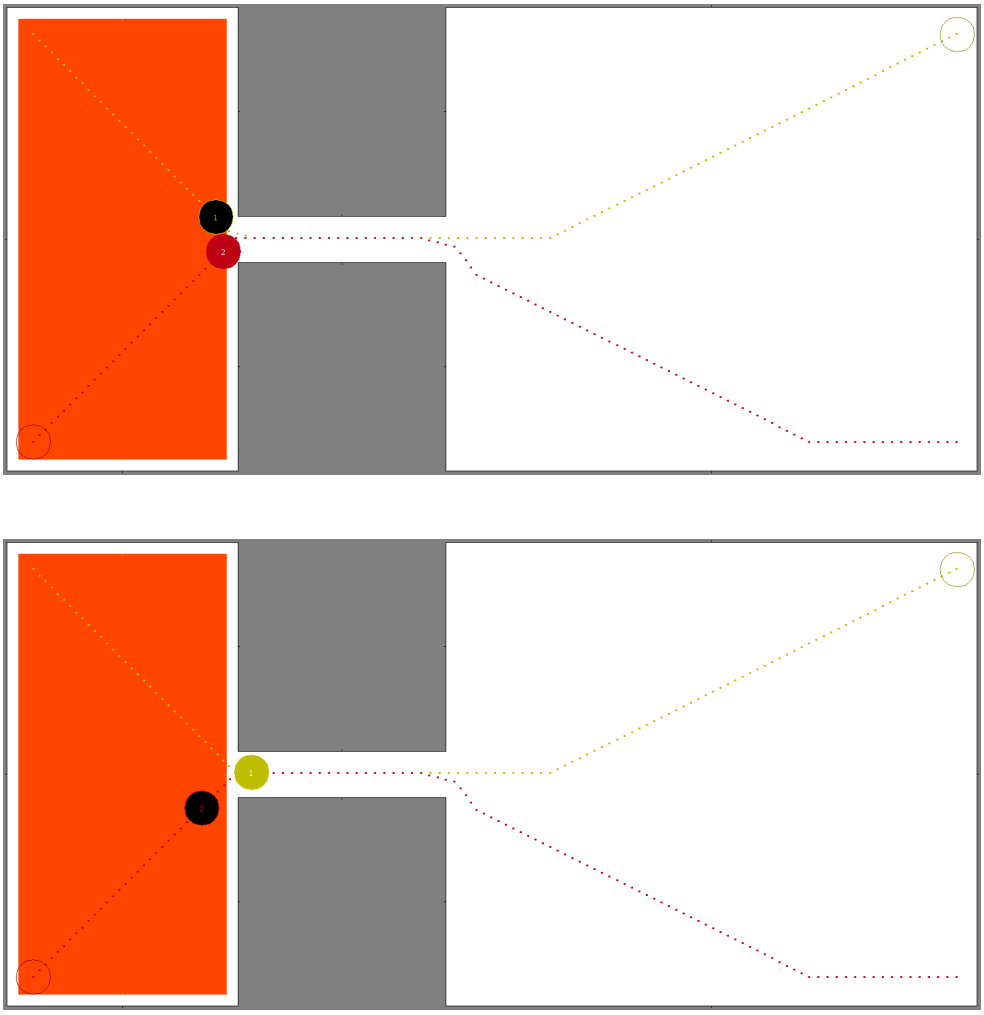


Figure 3.8: Because of the flip, when robot 2 clears the intersection first, then robot 1 enters the intersection.

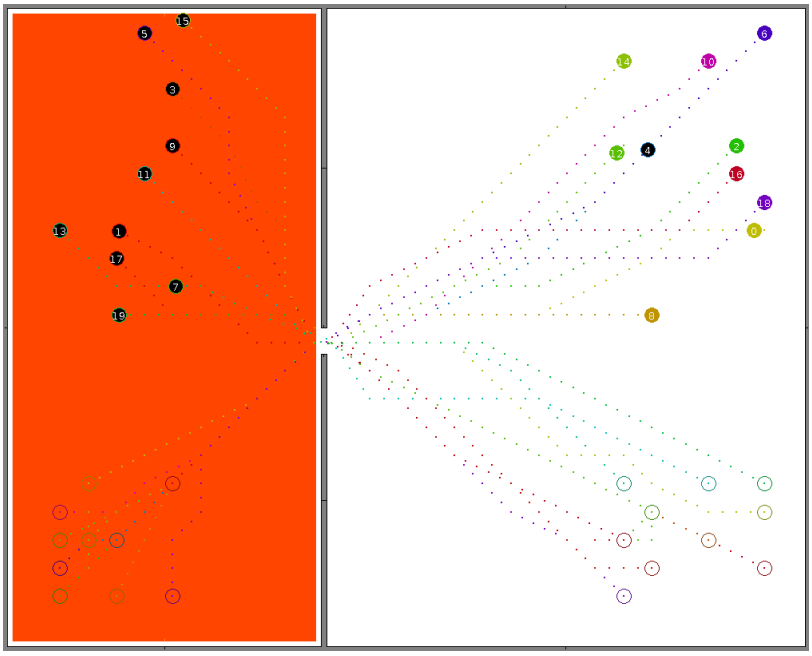
Figure 3.11, is smaller for the modified algorithms than for vanilla RMTRACK and that it is no more than 10% of the average travel time. We report these results in a different way in Figure 3.12 and Figure 3.13.

For the obstacle flipping algorithms, we also measured the amount of computation time consumed by the TestFlip algorithms in calculating the expected travel time. From these results, which are in Figure 3.14, we conclude that this computation is nearly negligible, in comparison to the savings in robot travel time.

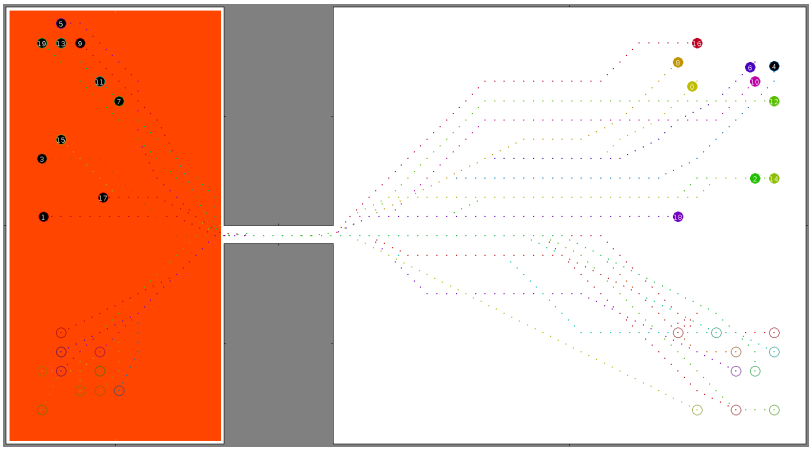
We also counted the number of flips executed by each approach, as shown in Figure 3.15. These results confirm that the extra computation time invested in TestFlipAggressive does indeed identify more opportunities to flip the obstacle labels.

We report the percentage of successful trials in Figure 3.16, where a trial is successful if all robots reach their goal position without encountering deadlock.

In all cases, for a sufficiently large number of robots, deadlock took place. For the environment with a short passage, both TestFlip algorithms perform similarly. As expected (see section 3.3.3), the RMTRACK+TFA algorithm performs better in the environment with a long passage.

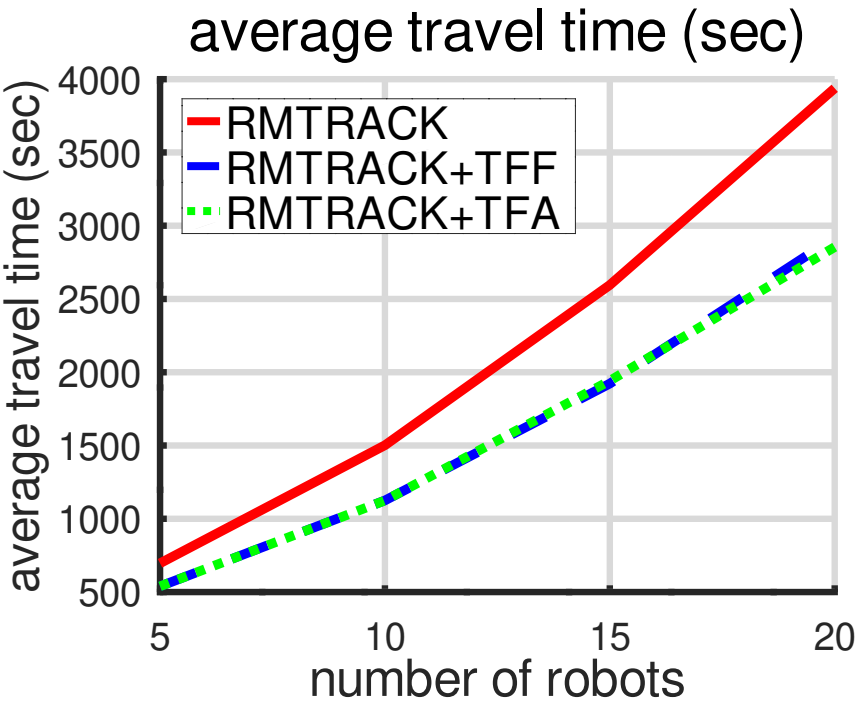


(a) An environment with a short passage

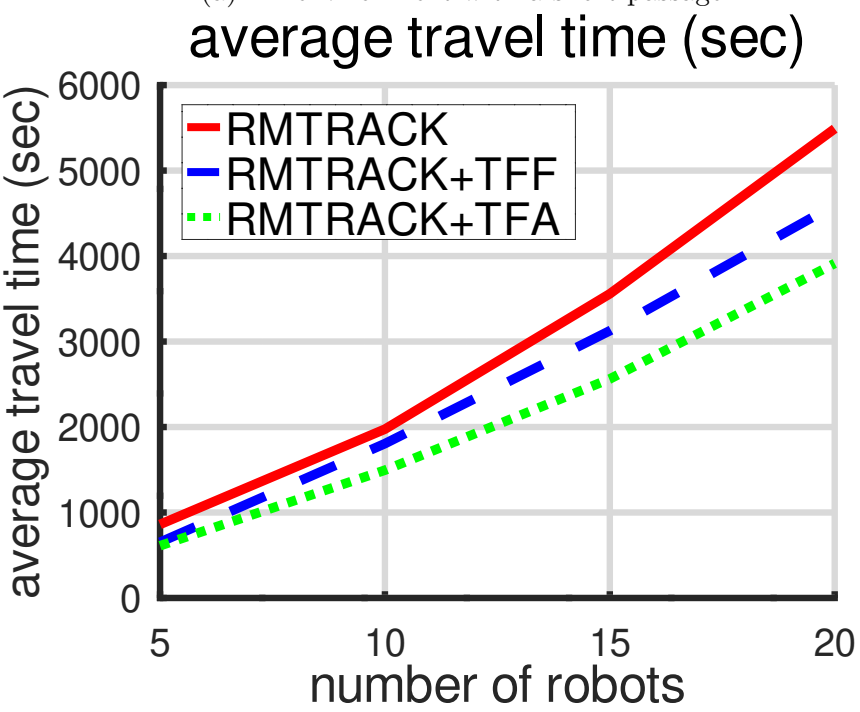


(b) An environment with a long passage

Figure 3.9: Environments

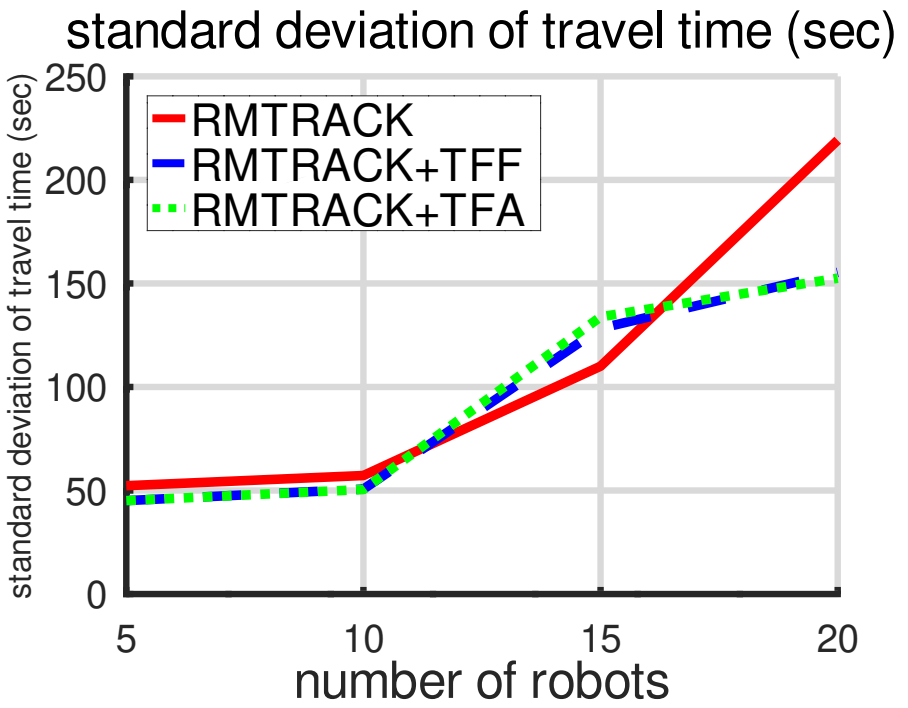


(a) An environment with a short passage

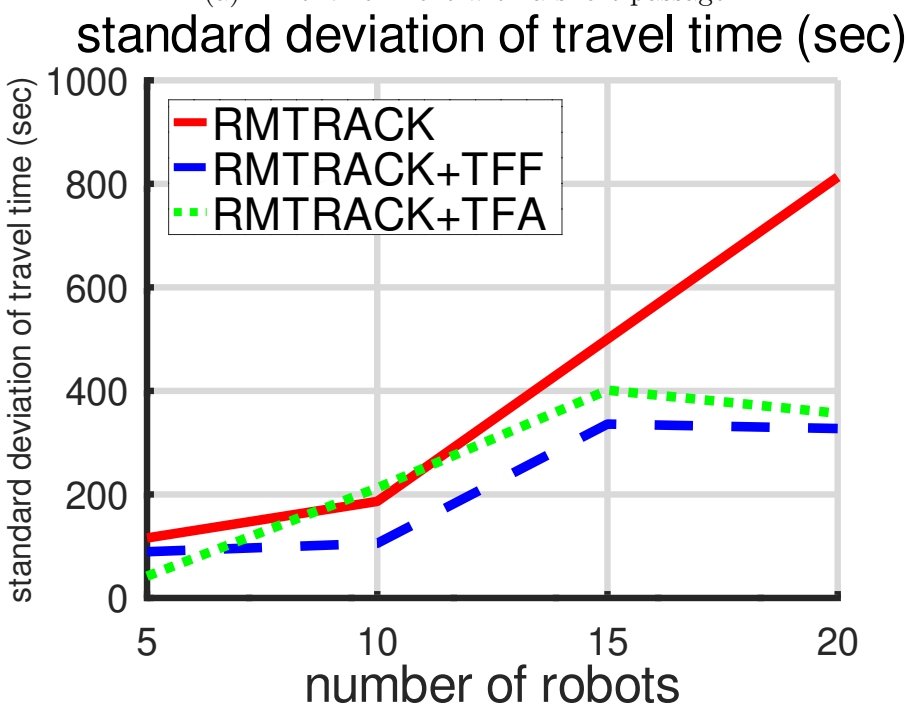


(b) An environment with a long passage

Figure 3.10: Average Travel Times

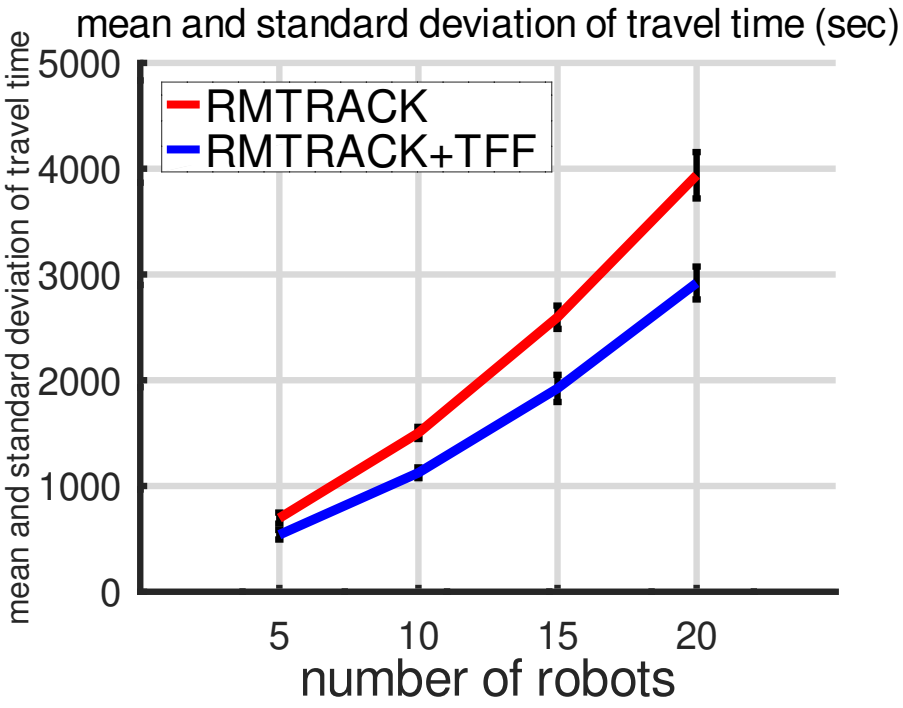


(a) An environment with a short passage

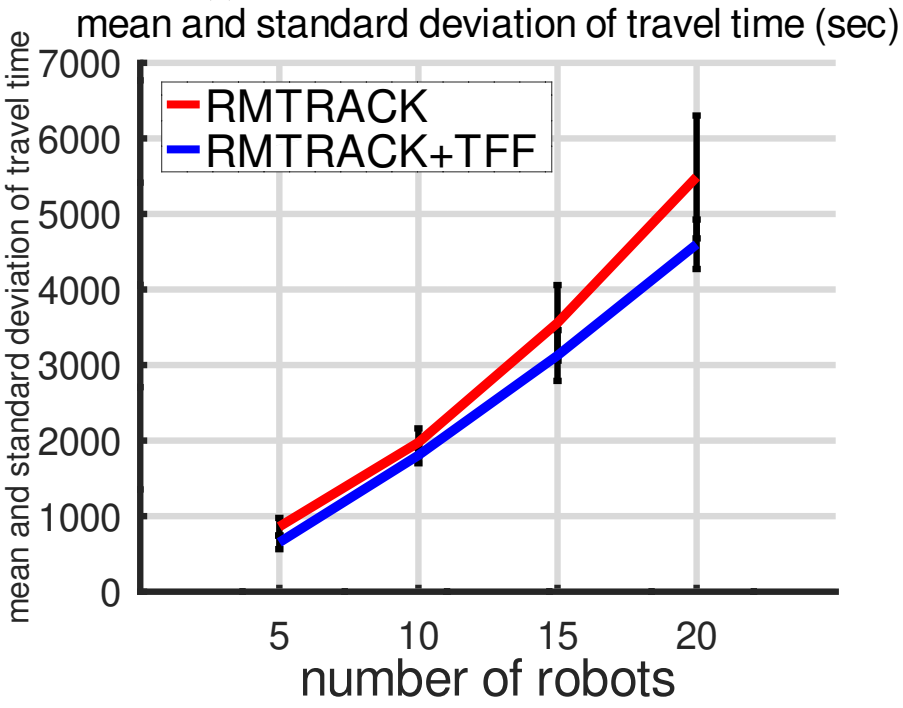


(b) An environment with a long passage

Figure 3.11: Standard Deviation of Travel Times

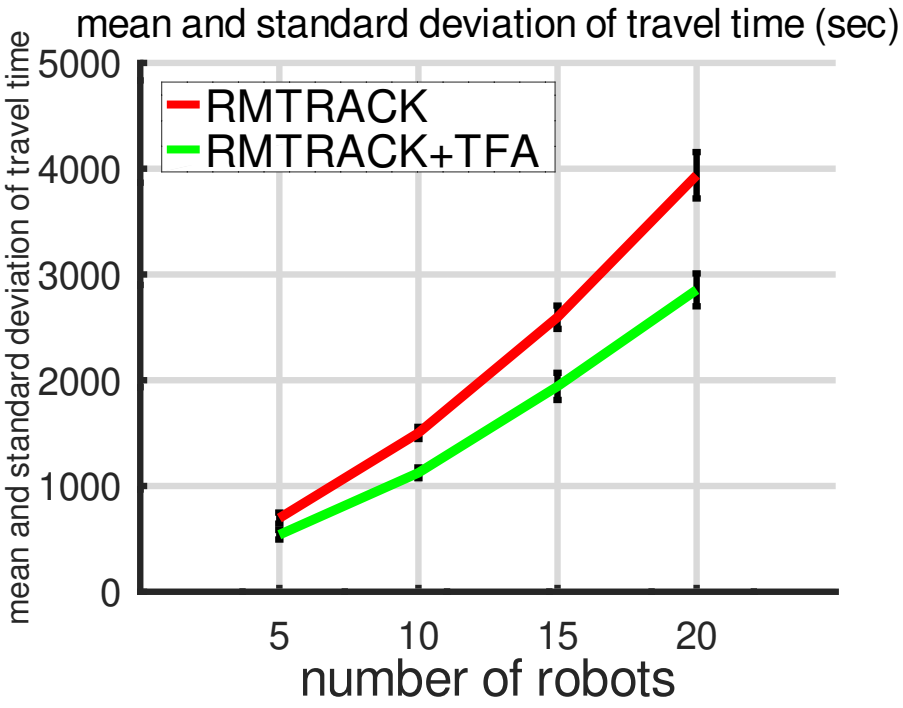


(a) An environment with a short passage

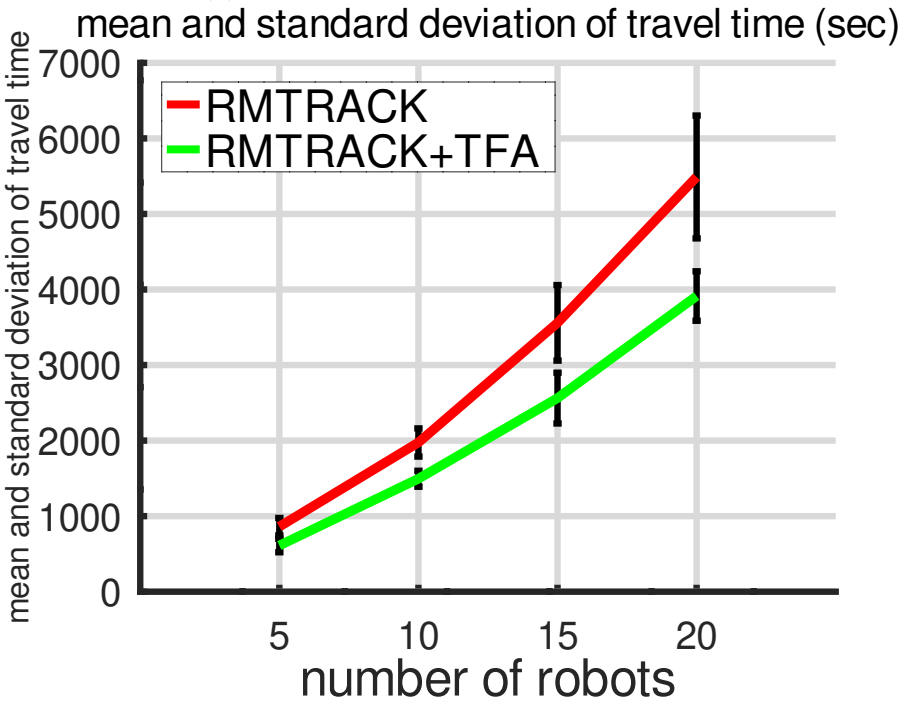


(b) An environment with a long passage

Figure 3.12: Mean and Standard Deviation of Travel Times for RMTRACK and RMTRACK+TFF

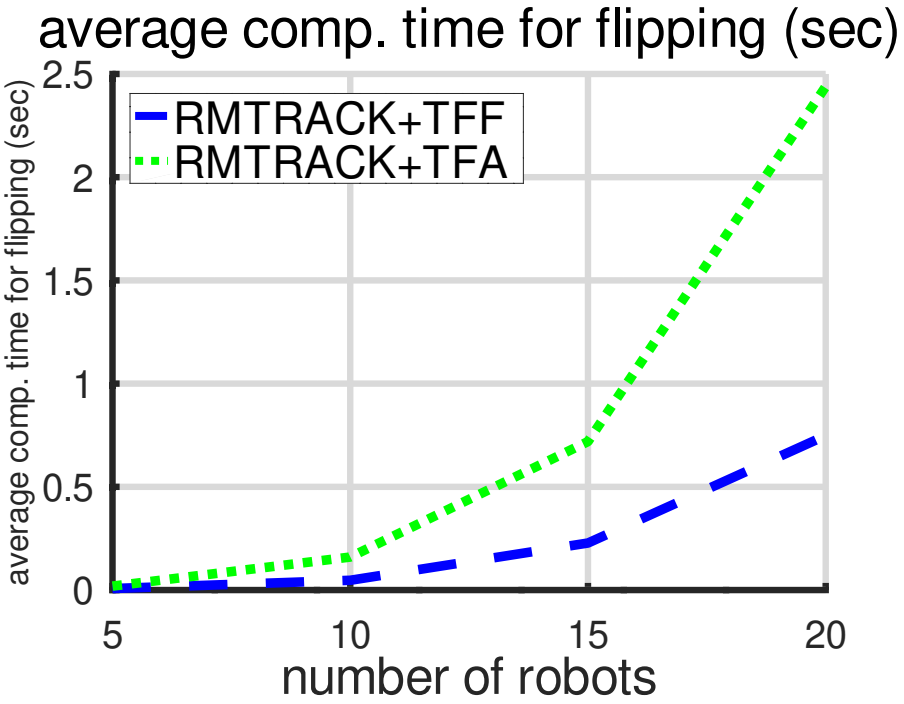


(a) An environment with a short passage

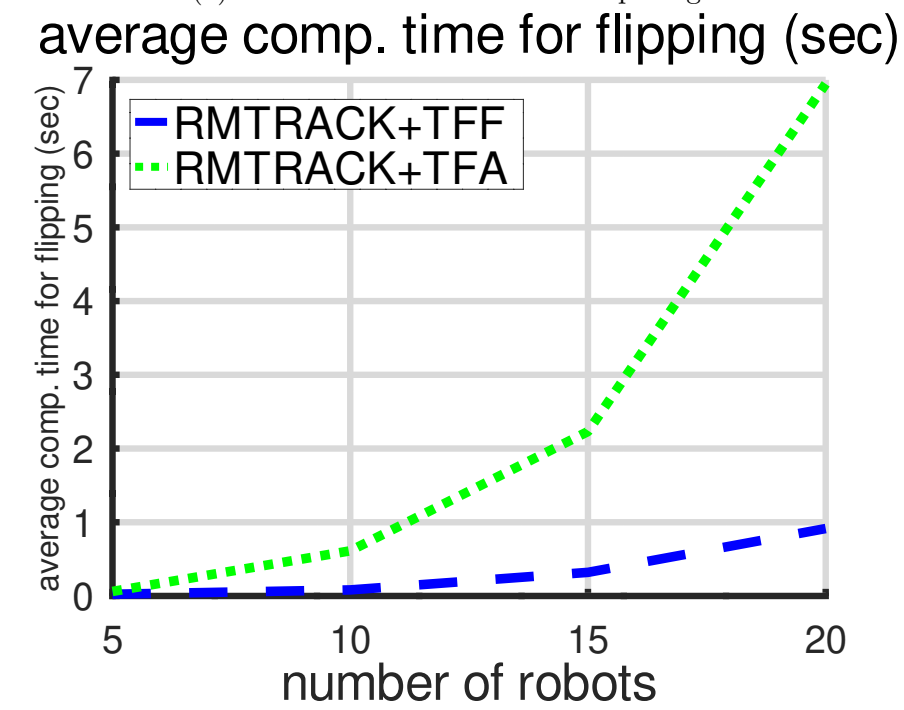


(b) An environment with a long passage

Figure 3.13: Mean and Standard Deviation of Travel Times for RMTRACK and RMTRACK+TFA

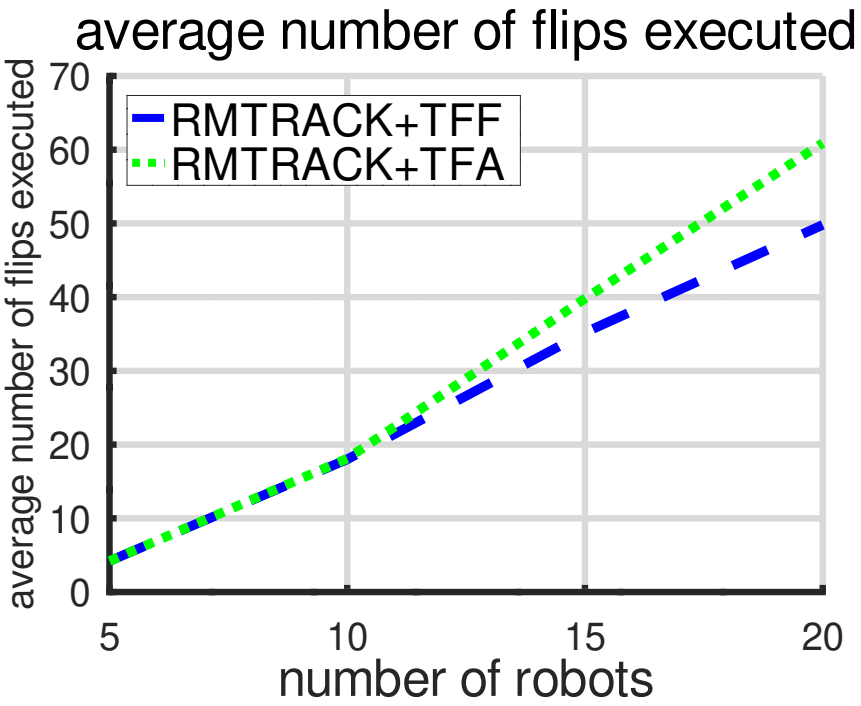


(a) An environment with a short passage

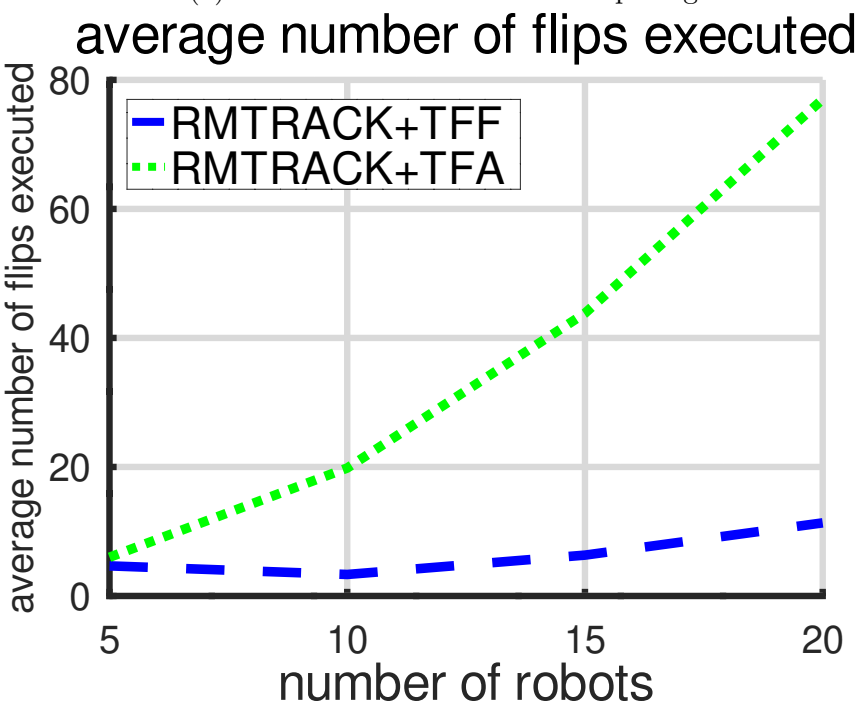


(b) An environment with a long passage

Figure 3.14: Computational Times for Flipping

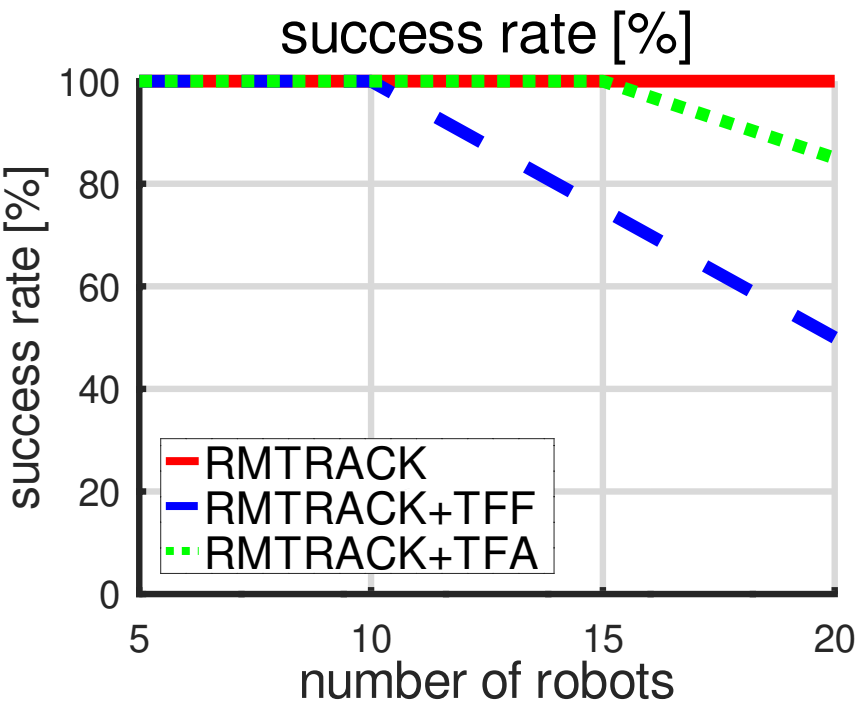


(a) An environment with a short passage

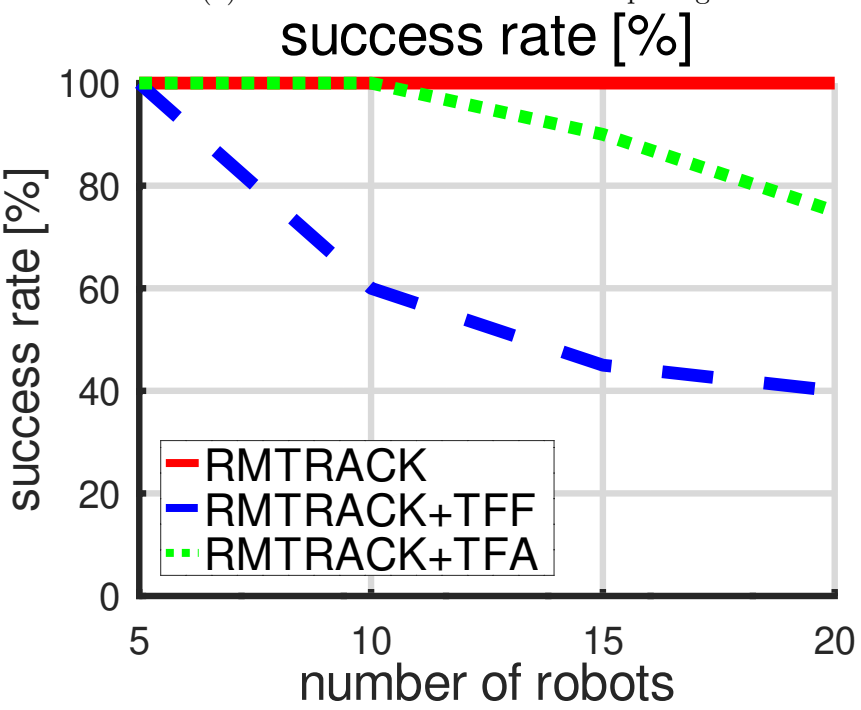


(b) An environment with a long passage

Figure 3.15: Number Of Flips Executed



(a) An environment with a short passage



(b) An environment with a long passage

Figure 3.16: Success Rate

Chapter 4

Deadlock

Though RMTRACK+TFF and RMTRACK+TFA can effectively modify the coordination plan (by judiciously changing obstacle labels) to improve overall efficiency, there are circumstances in which those changes can lead to deadlock conditions, wherein none of the robots can make progress. In this chapter, we provide a precise definition of deadlock in this context, and prove that, under certain reasonable but not universal conditions, deadlock does not occur. These results then form the foundation for the deadlock-free coordination scheme.

Let's recall the control law for RMTRACK, which for robot i at time t selects $a_i(t)$, is:

$$a_i(t) = \begin{cases} 0 & \exists j \neq i, \text{ s.t. } \exists k : \ell(o_k^{ij}) = j \text{ and} \\ & o_k^{ij} \cap (\{x_i(t)+1\} \times \{x_j(t), ..., K_j\} \neq \emptyset \\ 0 & \text{if } x_i(t) = K_i \\ 1 & \text{otherwise} \end{cases} \qquad \text{(2.3 revisited)}$$

The three cases of Equation 2.3 correspond to three distinct states for each robot:

- *Waiting (first case in Eq. 2.3):* If there exists a coordination space obstacle, o_k^{ij} for robot i and j with label j, so that $\ell(o_k^{ij}) = j$, then we check whether there is an intersection between the obstacle and the line from $(x_i(t)+1, x_j(t))$ to $(x_i(t)+1, K_j)$. The intersection indicates that robot j did not pass the obstacle yet, and robot i should therefore wait; thus the action variable is 0.

- *Finished (second case in Eq. 2.3):* When robot i reaches its destination, it should not move any further. The action variable then becomes 0.

- *Moving State (third case in Eq. 2.3):* If neither of the first two cases applies, then robot i is free to move, taking the action variable as 1.

We begin with a definition of deadlock. The three different states —waiting, moving, and finished— for each robot defined in Eq. 2.3 form the basis of the definition.

Definition 4.1 (deadlock)**.** The system has a *deadlock* if at least one robot is in the waiting state and no robots are in the moving state.

Our concern is to understand the conditions under which deadlocks can occur, so that we can ensure that those conditions do not arise. To that end, we first introduce *collision segments* in coordination spaces.

Definition 4.2 (collision segment)**.** For a given coordination space obstacle o_k^{ij}, the *collision segment* c_k^{ij} for that obstacle is the set of indices in $\{1, \ldots, K_i\}$ along the path for robot i between the path step at which robot i reaches that obstacle, denoted $s(c_k^{ij})$, and the path step at which robot i clears that obstacle, denoted $f(c_k^{ij})$. That is, $c_k^{ij} = \{s(c_k^{ij}), \ldots, f(c_k^{ij}) - 1\}$.

Figure 4.1 shows an example environment with the coordination spaces and collision segments. Such segments are important because they capture the structure of how the movements of the various robots affect each other. Notice in particular that the s and f functions induce a partition of the path of robot i into segments delimited by the start and end points of all of its collision segments with all other robots. Using this partition, we can express the relationship between paths of the robots and labels of the obstacles.

Definition 4.3 (segment graph)**.** The *segment graph* is a directed graph containing:

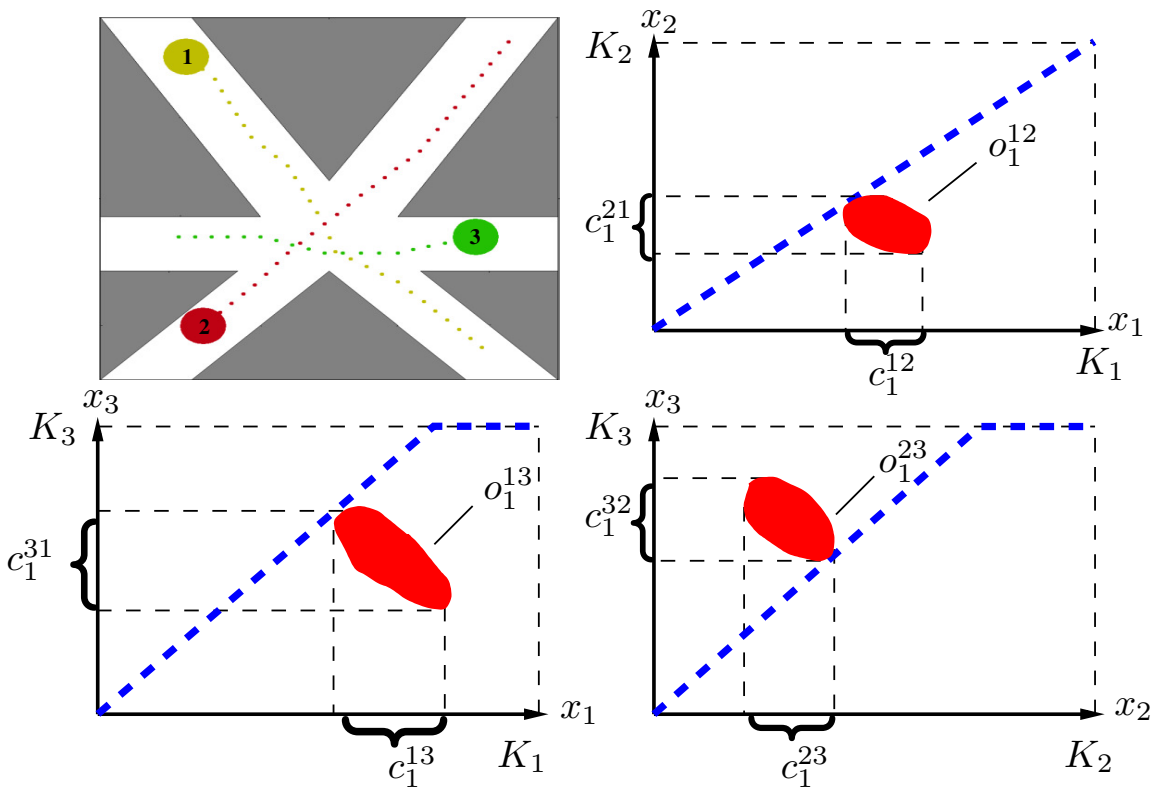


Figure 4.1: Three robots that have pairwise collisions in the environment are depicted on the top left. The coordination spaces, C_{12}, C_{13}, and C_{23}, are depicted on the top right, on the bottom left, and on the bottom right, respectively. The collision segments correspond to the obstacles shown in the coordination spaces.

1. vertices corresponding to the maximal path segments for each robot delimited by the start and the finish of its collision segments with all other robots, denoted $v_1^{(i)}, v_2^{(i)}, \ldots, v_{m_i}^{(i)}$ for each robot i;

2. edges called *path sequence edges* between each successive pair of vertices $v_k^{(i)}$ and $v_{k+1}^{(i)}$; and

3. for each coordination space obstacle o_k^{ij} with label i, an edge called an *obstacle-label edge* from the vertex for robot i corresponding to the final path segment overlapping o_k^{ij} to the vertex for robot j corresponding to the first path segment overlapping o_k^{ji}.

It will be convenient later to refer to the starting and ending points of the path segment for each vertex. We (re-)use the letters s and f for this purpose, so that vertex $v_k^{(i)}$ in the segment graph corresponds to the range of steps $s(v_k^{(i)}), \ldots, f(v_k^{(i)})$ within the path for robot i. Notice that, in general, the finish of one vertex is immediately before the start of another: $f(v_k^{(i)}) = s(v_{k+1}^{(i)})$.

In addition, we write $V(o_k^{ij}) = \{v_p^{(i)}, v_{p+1}^{(i)}, \ldots v_{p+q}^{(i)}\} \subset V$ for the set of robot i vertices containing obstacle o_k^{ij}.

As the robots execute their paths, they move in sequence through the segments of their paths. Thus, we can refer to the *present vertex* in the segment graph for each robot. Any vertex corresponding to a path segment that the robot has not yet begun to execute is a called a *future vertex* for that robot. The subgraph of the segment graph induced by the present and future vertices is called the *active subgraph*.

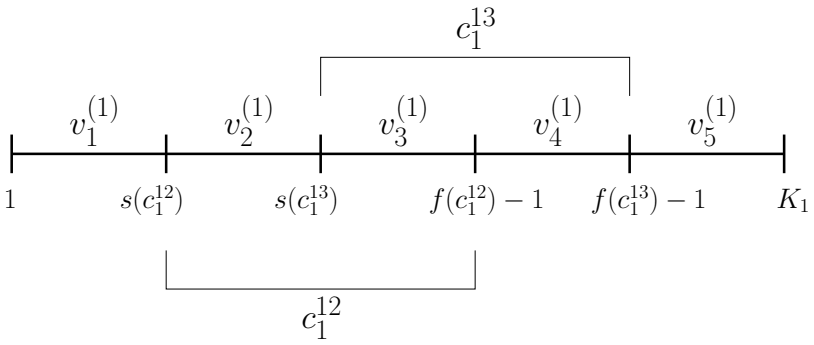
The intuition is that each edge in segment describes a constraint wherein one segment of some robot's path must be completed before another segment can begin. Path sequence edges encode the constraint that each robot must execute its path sequentially; obstacle label edges encode the waiting behaviour required by the first case in Equation 2.3.

For the scenario depicted in Figure 4.1, Figure 4.2 shows the vertices for each robot, Figure 4.3 shows the path sequence edges, and Figure 4.4, Figure 4.5, Figure 4.6 shows each obstacle-label edge.

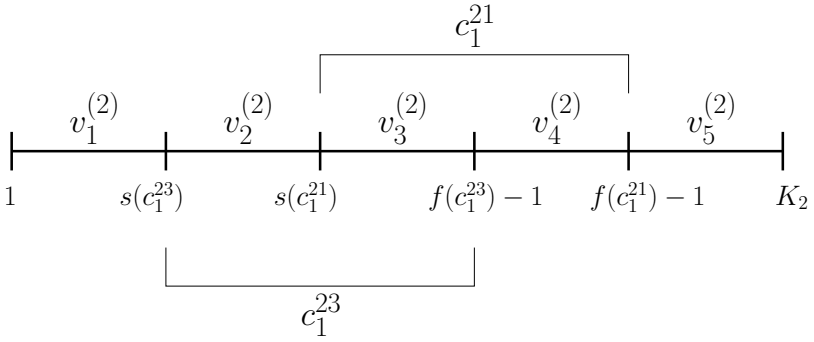
The next definition is needed to connect segment graphs to the possibility of deadlocks in the future.

Definition 4.4. A set of trajectories is *non-conflicting* if, for every pair of i, j of distinct robots, for all $1 \leq k \leq K_i$, we have

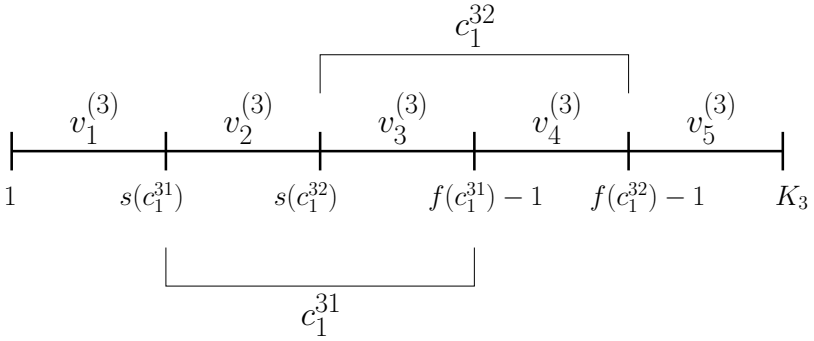
$$||\pi_i(k) - \pi_j(1)|| \geq 2r,$$



(a) There are five vertices for robot 1 as the path steps are delimited by the start and the finish of the collision segment with robot 2 and the collision segment with robot 3.



(b) There are five vertices for robot 2 as the path steps are delimited by the start and the finish of the collision segment with robot 1 and the collision segment with robot 3.



(c) There are five vertices for robot 3 as the path steps are delimited by the start and the finish of the collision segment with robot 1 and the collision segment with robot 2.

Figure 4.2: The collision segments for the robots in Figure 4.1 are depicted above. Each robot has two different collision segments, and the collision segments are not disjoint, so they contain two vertices. For example, c_1^{13} contains $v_3^{(1)}$ and $v_4^{(1)}$.

and

$$||\pi_i(k) - \pi_j(K_j)|| \geq 2r.$$

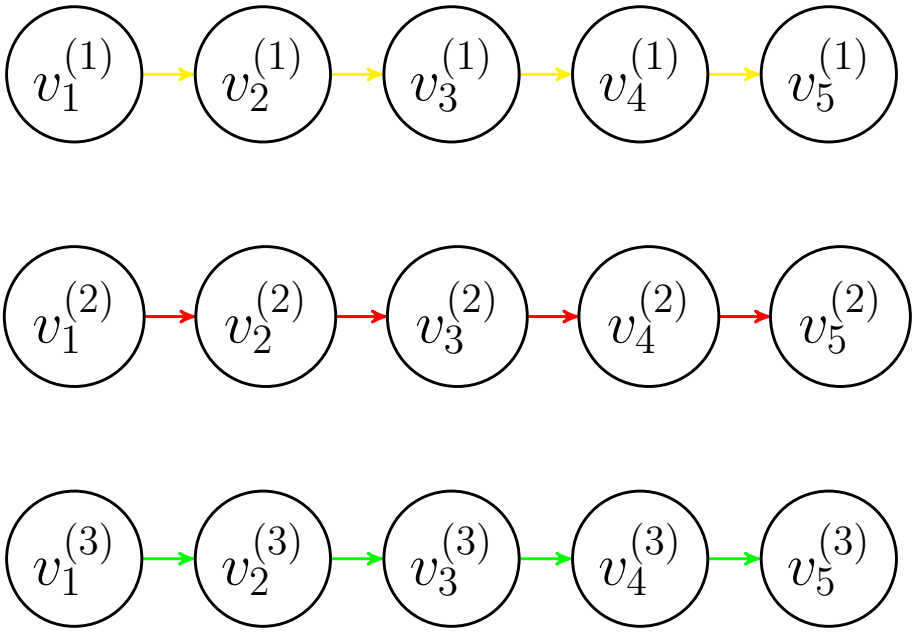


Figure 4.3: All horizontal edges represent the path sequence edges for each robot.

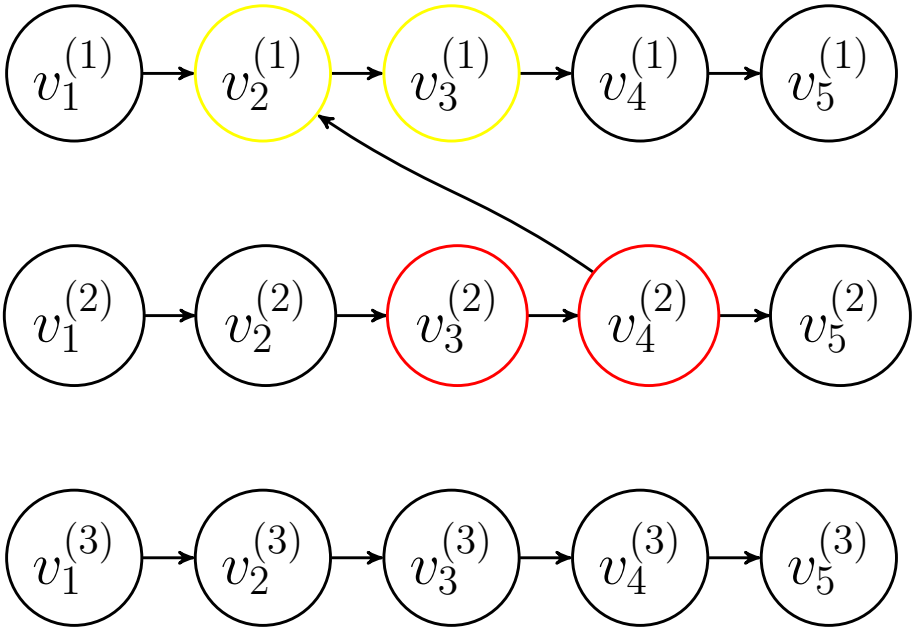


Figure 4.4: The coordination space obstacle o_1^{12} has label $\ell(o_1^{12}) = 2$. $V(o_1^{12}) = \{v_2^{(1)}, v_3^{(1)}\}$ and $V(o_1^{21}) = \{v_3^{(2)}, v_4^{(2)}\}$. The *obstacle-label edge* is from the last vertex containing o_1^{21}, which is $v_4^{(2)}$, to the first vertex containing o_1^{12}, which is $v_2^{(1)}$.

That is, when we have non-conflicting trajectories, the path for each robot is disjoint from the start and goal positions of all other robots. Figure 4.7 shows an example of conflicting trajectories and the corresponding segment graph.

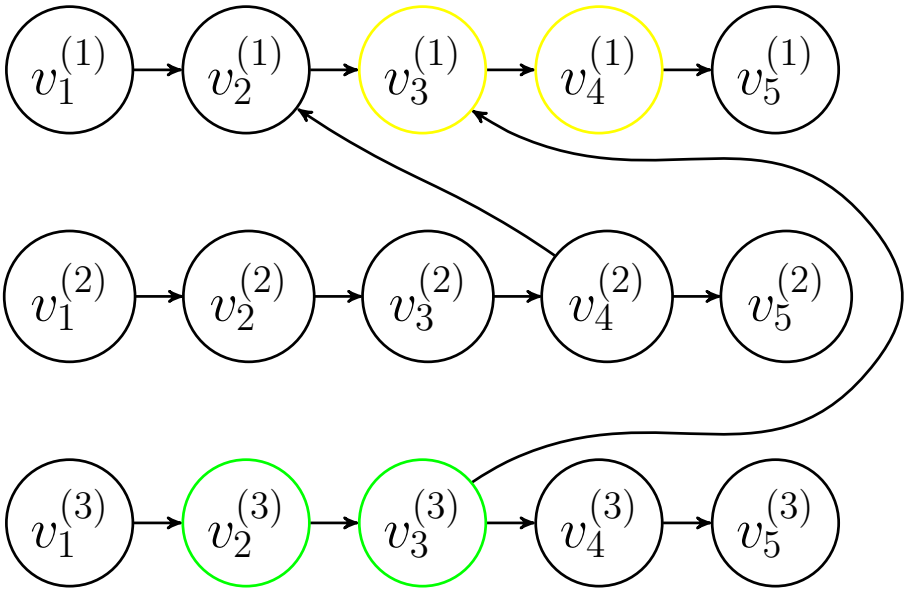


Figure 4.5: The coordination space obstacle o_1^{13} has label $\ell(o_1^{13}) = 3$. $V(o_1^{13}) = \{v_3^{(1)}, v_4^{(1)}\}$ and $V(o_1^{31}) = \{v_2^{(3)}, v_3^{(3)}\}$. There is an *obstacle-label edge* from the last vertex containing o_1^{31}, which is $v_3^{(3)}$, to the first vertex containing o_1^{13}, which is $v_3^{(1)}$.

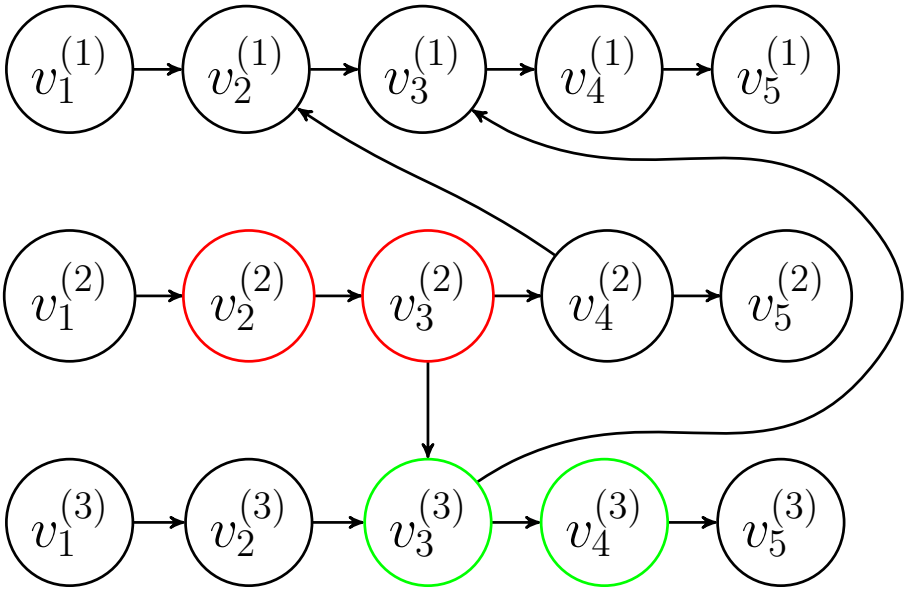
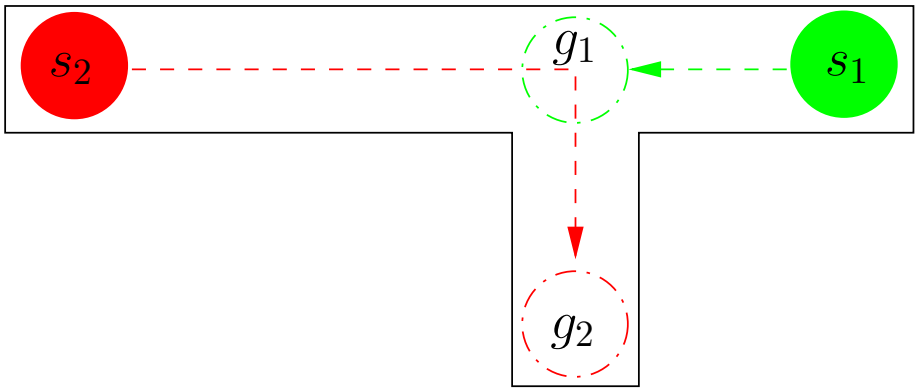


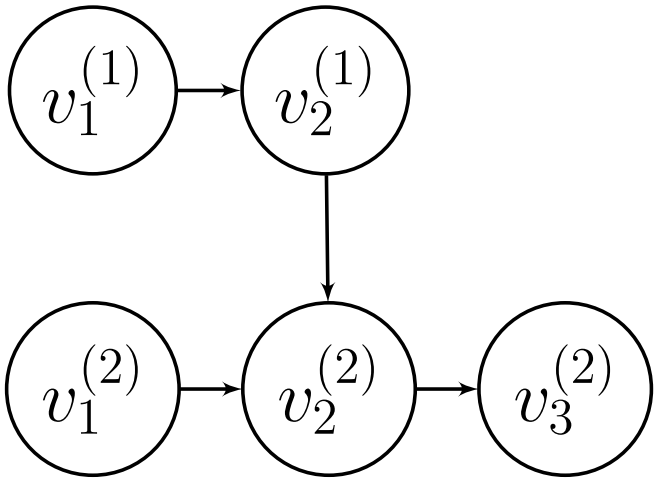
Figure 4.6: The coordination space obstacle o_1^{23} has label $\ell(o_1^{23}) = 2$. $V(o_1^{23}) = \{v_2^{(2)}, v_3^{(2)}\}$ and $V(o_1^{32}) = \{v_3^{(3)}, v_4^{(3)}\}$. There is an *obstacle-label edge* from the last vertex containing o_1^{23}, which is $v_3^{(2)}$, to the first vertex containing o_1^{32}, which is $v_3^{(3)}$.

4.1 Deadlock Conditions

Now we will show that cycles in the segment graph correspond to potential deadlocks.



(a) The green dotted line shows the path of robot 1, from the start position, s_1, to the goal position, g_1, and the red dotted line shows the path of robot 2, from the start position, s_2, to the goal position, g_2.



(b) The segment graph with the label, $\ell(o_1^{12}) = 1$.

Figure 4.7: If robot 1 reaches its goal position, g_1, before robot 2 clears the intersection, then the system has a deadlock. Even though the environment has a deadlock, the corresponding segment graph does not have a cycle.

Theorem 4.1. If the system has non-conflicting trajectories and is in a deadlock, then the active subgraph has a cycle.

Proof. Definition 4.1 ensures that at least one robot is in the waiting state. Let i denote the index of this robot, and let $v_p^{(i)}$ denote its present vertex. Because of Definition 4.3, there must therefore be at least one edge incoming to $v_{p+1}^{(i)}$, i.e. the vertex corresponding to the next segment for robot i from a future vertex of some other robot. Let j denote the index of this other robot. Robot j cannot be in

the moving state because the system is in a deadlock. Moreover, robot j cannot be in the finished state because being in the finished state, in a system with non-conflicting trajectories, does not prevent any other robots —robot i in particular— from moving. Thus, robot j is in the waiting state. Because robot j is waiting, there must be another incoming obstacle-label edge from another robot's future vertex to a future vertex of the robot j, and so on. This produces an infinite sequence of vertices in the graph, each connected by a directed edge to its predecessor in the sequence. Since the number of robots is finite, the vertices in this sequence cannot all be distinct. Therefore, the sequence must eventually repeat, forming a cycle in the active subgraph. □

Theorem 4.2. If the active subgraph of a system has a cycle, and the labels of the obstacles of the robots in the cycle do not change, then the system will eventually be in deadlock.

Proof. It is obvious that if the system has one robot, then the segment graph has only path sequence edges, so a cycle cannot exist. Also, a cycle cannot exist when there are only two robots. Note that there must be only one incoming or one outgoing obstacle-label edge between two robots' vertices for each coordination space obstacle, so those obstacle-label edges do not lead to a cycle.

Now, assume that there are at least three robots in the system and that the active subgraph has a cycle between robot i, robot j, ... , and robot k, whose present vertices are $v_p^{(i)}, v_q^{(j)}, \ldots, v_r^{(k)}$, respectively.

Consider a cycle between future vertices of the robots in the segment graph: $v_{p+1}^{(i)} \rightarrow v_{q+1}^{(j)} \rightarrow \ldots \rightarrow v_{r+1}^{(k)} \rightarrow v_{p+1}^{(i)}$

Since there is an obstacle-label edge from robot k to robot i, robot i cannot start the future vertex, $v_{p+1}^{(i)}$ before robot k finishes its future vertex, $v_{r+1}^{(k)}$.

Similarly, there is also an obstacle-label edge from robot i to robot j, so robot j cannot start the future vertex, $v_{q+1}^{(j)}$ before robot i finishes its future vertex, $v_{p+1}^{(i)}$.

Therefore, no robots in the cycle start their future vertex and stay in the waiting state, so the system has a deadlock. □

The main contribution of the proposed algorithm is that it allows testing to determine whether changing an obstacle label leads to a deadlock. The algorithm first generates the segment graph by using a Java library, JGraphT[1]. Then, if the segment graph detects a cycle with the updated label, the algorithm does not change that label. For example, let us recall the example depicted in Figure 4.1. Also, Figure 4.9 shows the initial positions of the robots with the segment graph by indicating their present vertices. There are three robots, and there are three pairwise obstacles between each robot. According to the label of the obstacles, robot 2 should pass the obstacle before robot 1 passes, as shown in C_{12}, on the top right of Figure 4.1. However, robot 2 is subject to more disturbance than the others, and robot 1 reaches the obstacle before robot 2 as depicted on Figure 4.9. There are two scenarios for Robot 1, staying in the waiting state until robot 2 clears the obstacle, or flipping the label of obstacle and continuing to execute its path. If robot 1 changes the label, then it can move, but it needs to stop to avoid collision with robot 3, which is in the waiting state by waiting for robot 2, as depicted on Figure 4.10. Then, when robot 2 reaches the obstacle, a deadlock ensues as shown in Figure 4.11. In order to avoid this deadlock, when robot 1 needs to change the label between robot 2, the segment graph is updated, and checked for the occurrence of a cycle. If a cycle occurs as shown on the right of Figure 4.10, then the algorithm does not change the label, and robot 1 stays in the waiting state by waiting for robot 2 so that the system does not have a deadlock.

Corollary 4.1. If, in a system with non-conflicting trajectories, the active subgraph contains no cycles, then the system is not in a deadlock.

[1] `https://jgrapht.org/`

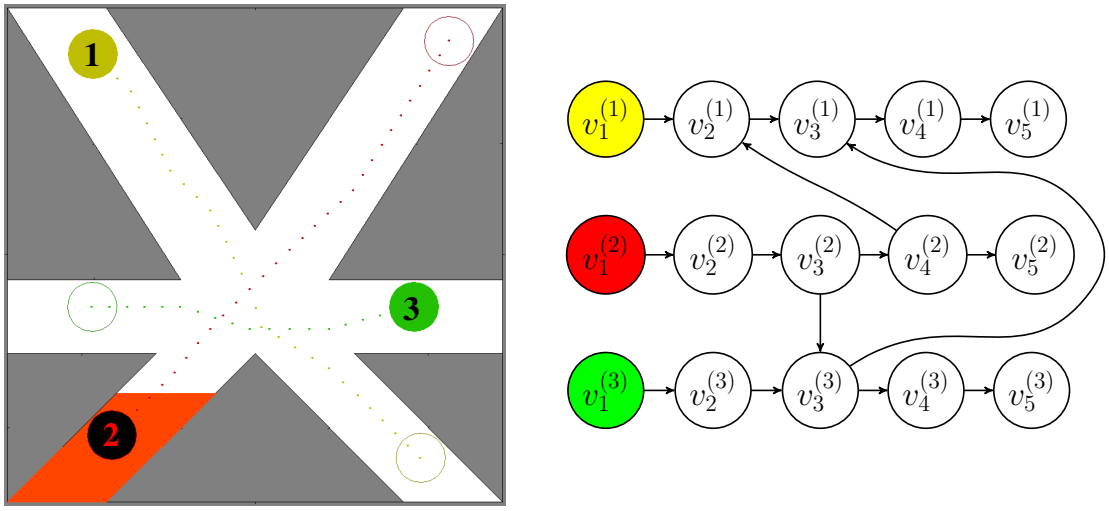


Figure 4.8: All robots are in their start positions. The present vertices of the robots are $v_1^{(1)}$, $v_1^{(2)}$, and $v_1^{(3)}$, respectively.

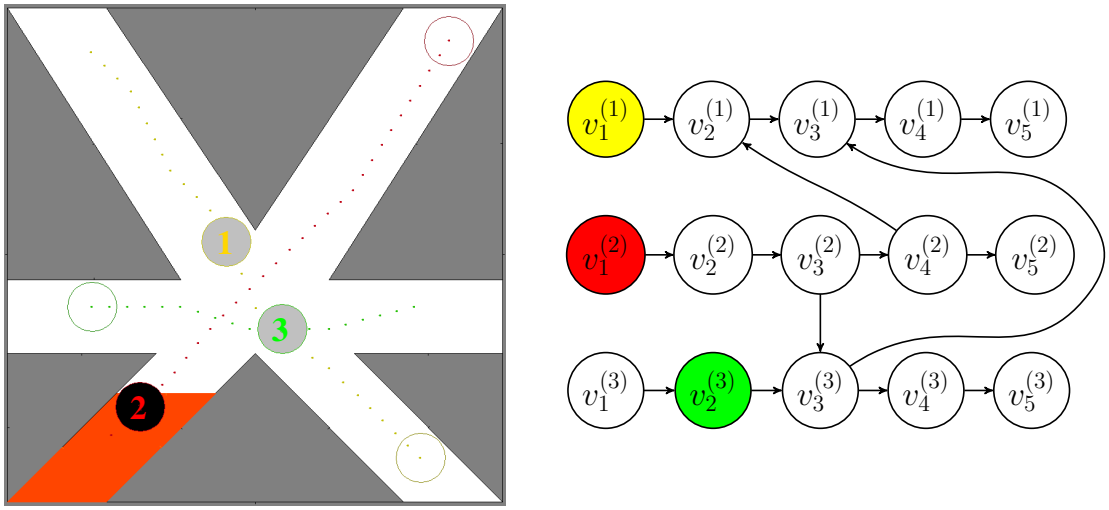


Figure 4.9: Robot 1 reaches the intersection before robot 2, but needs to wait for robot 2 to clear the intersection because of the label, $\ell(o_1^{12}) = 2$. Robot 2 is delayed because of the disturbances. Robot 3 enters the intersection before robot 1, but cannot clear the intersection with robot 2 because of the label, $\ell(o_1^{23}) = 2$.

This last result is of particular interest because, since the initial trajectories are collision free, the only way a cycle can be introduced in the segment graph is by modifying one of the obstacle labels, which in turn rewires one of the obstacle label edges. This idea underlies the deadlock free coordination approach.

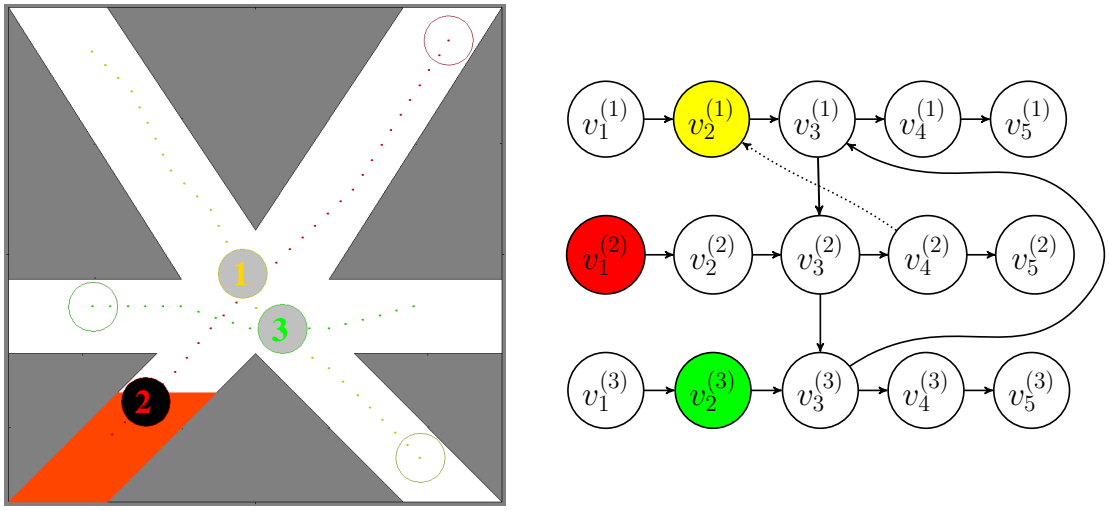


Figure 4.10: Robot 1 flips the label of the obstacle it shares with robot 2, so the label $\ell(o_1^{12})$ becomes 1, and we need to update the *obstacle-label edge* between robot 1 and robot 2. The updated *obstacle-label edge* is from the last vertex containing o_1^{12}, which is $v_3^{(1)}$, to the first vertex containing o_1^{21}, which is $v_3^{(2)}$. Now, robot 1 starts the intersection between robot 2, but needs to wait until robot 3 clears the intersection because of the label, $\ell(o_1^{13}) = 3$. Note that the updated obstacle-label edge results in a cycle in the segment graph.

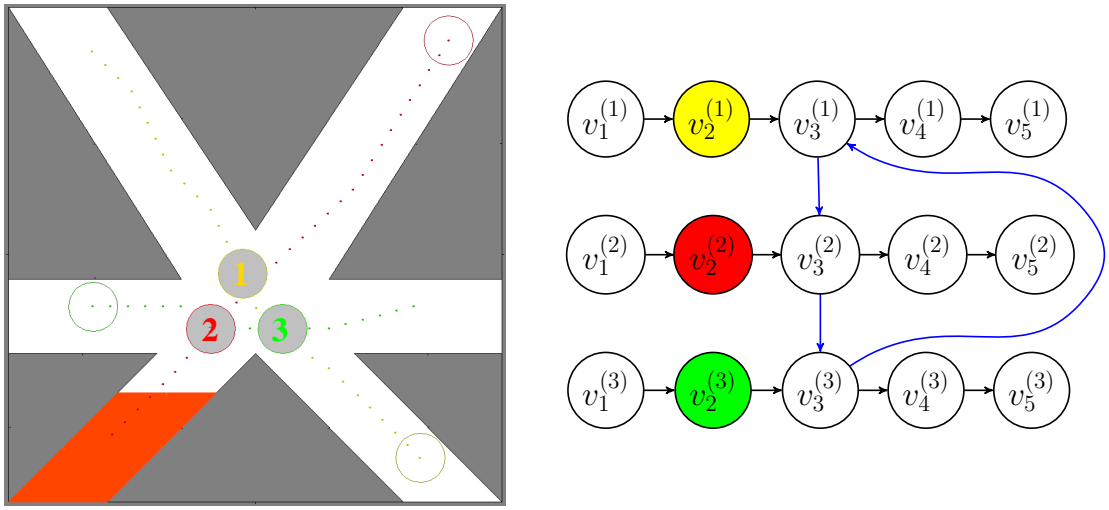


Figure 4.11: When robot 2 starts the intersection with robot 3, it needs to stop because of the updated label, $\ell(o_1^{12}) = 2$. All robots are in the waiting state. Thus, the system has a deadlock.

Moving beyond that observation, we now also provide a sufficient condition, which results a deadlock-free system, even in cases were the obstacle labels can be freely modified.

4.2 Deadlock-Free Coordination

We first examine the disjointedness of collision segments.

Theorem 4.3. If the collision segments are disjoint, then each set $V(o_k^{ij})$ contains only a single vertex.

Proof. Suppose $|V(o_k^{ij})| \neq 1$. The existence of more than one vertex in the set $V(o_k^{ij})$ indicates there exists at least one more collision segment reached or cleared by robot i before robot i clears o_k^{ij}. This contradicts the assumption that collision segments do not intersect, so $|V(o_k^{ij})| = 1$. □

The existence of these singleton vertex-segment sets is of interest because it allows us to demonstrate that the corresponding segment graph does not have a cycle.

Theorem 4.4. If the collision segments are disjoint, then the segment graph does not contain a cycle.

Proof. Suppose the segment graph has a cycle such that $v_p^{(i)} \to v_q^{(j)} \to \cdots \to v_r^{(k)} \to v_p^{(i)}$ with $V(o_l^{ij}) = \{v_p^{(i)}\}$, $V(o_l^{ji}) = \{v_q^{(j)}\}, \ldots, V(o_m^{ki}) = \{v_r^{(k)}\}$, and $V(o_m^{ik}) = \{v_p^{(i)}\}$.

Since $V(o_l^{ij}) = \{v_p^{(i)}\}$, $s(v_p^{(i)})$ must be the number of path steps to reach obstacle o_l^{ij}. However, since $V(o_m^{ik})$ contains only a single vertex, $s(v_p^{(i)})$ is also the number of path steps to reach the obstacle o_m^{ik}. This contradicts the assumption that the collision segments are disjoint. Therefore the segment graph does not have a cycle. □

Finally, viewing Theorem 4.4 in light of Corollary 4.1, one sees conditions which, though not directly leveraged in the framework of this chapter, may nonetheless be useful when designing multi-robot trajectory planners. If the planner generates paths that meet that condition — in this context, the requirement is that intersection points between paths must be at least distance $2r$ apart from each other, then deadlocks can be avoided, even if a method such as RMTRACK+TFF or RMTRACK+TFA modifies obstacle labels.

A deadlock-free environment is illustrated in Figure 4.12. Figure 4.13 shows the vertices for each robot. Note that collision segments are disjoint in this environment. Figure 4.14 shows the path sequence edges, and Figure 4.15, Figure 4.16, and Figure 4.17 shows each obstacle-label edge. Figure 4.18 and Figure 4.19 show the robots' positions with their current vertices in the segment graph, and all robots are in the moving state. However, robot 1 needs to wait robot 2 because of the label $\ell(o_1^{12}) = 2$ when it reaches the intersection as shown in Figure 4.20. If robot 1 decides to flip the label of the obstacle, then the obstacle label edge between robot 1 and robot 2 is updated as shown in Figure 4.21. Notice that the segment graph still does not have a cycle, and robot 1 continues to execute its path as shown in Figure 4.22. The robot 1 and robot 2 keeps moving, but robot 3 still waits robot 2 as shown in Figure 4.23. After robot 2 clears the intersection between robot 3, robot 3 starts to move as shown in Figure 4.24. All robots keep moving as shown in Figure 4.25 and Figure 4.26. Eventually, all robots reach their goal position as shown in Figure 4.27.

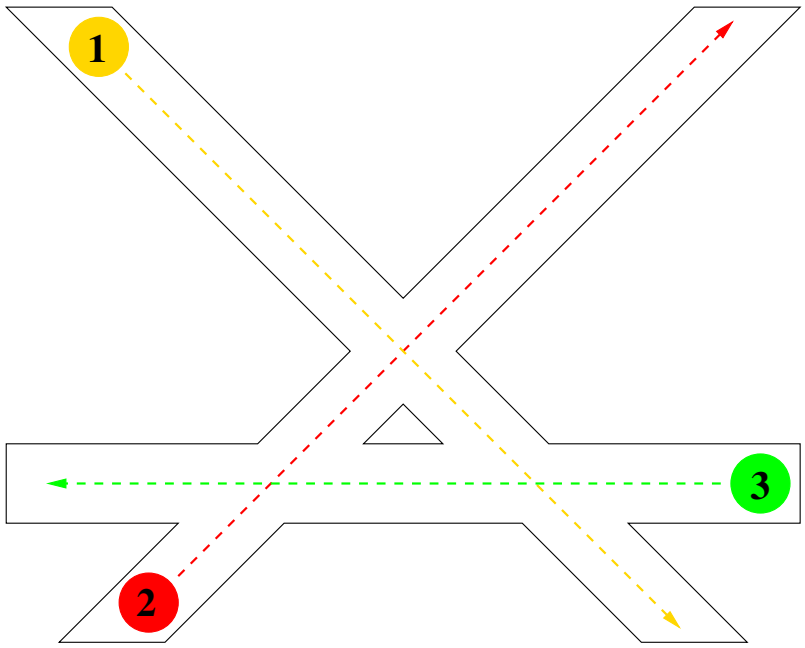
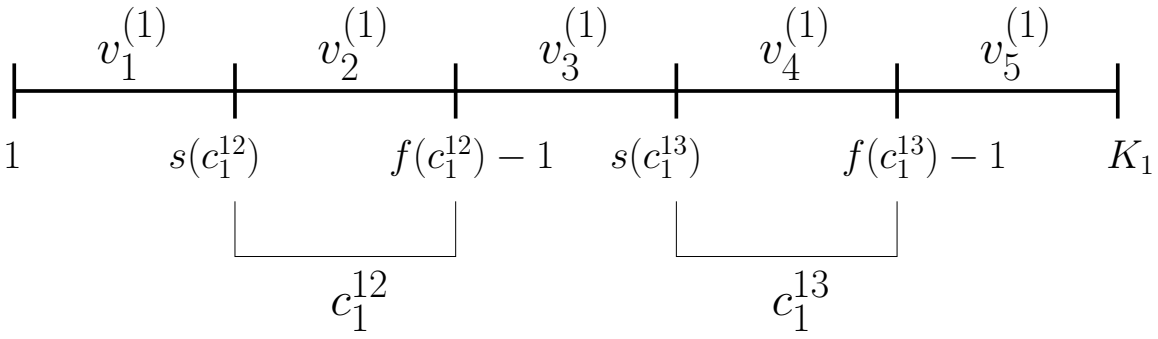
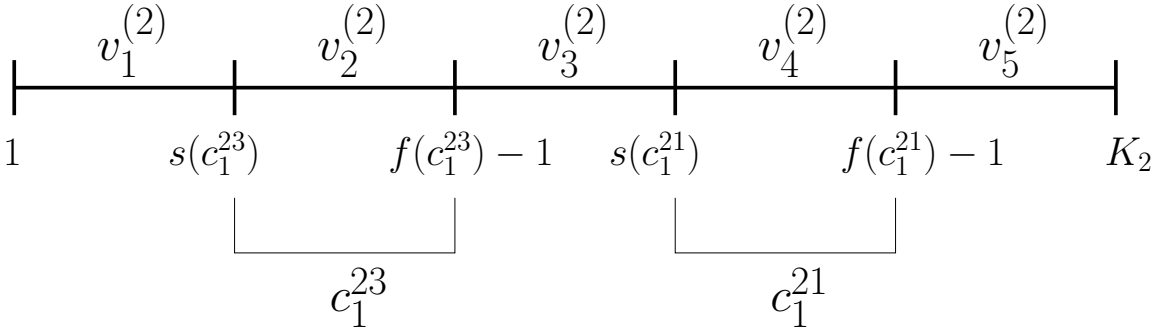


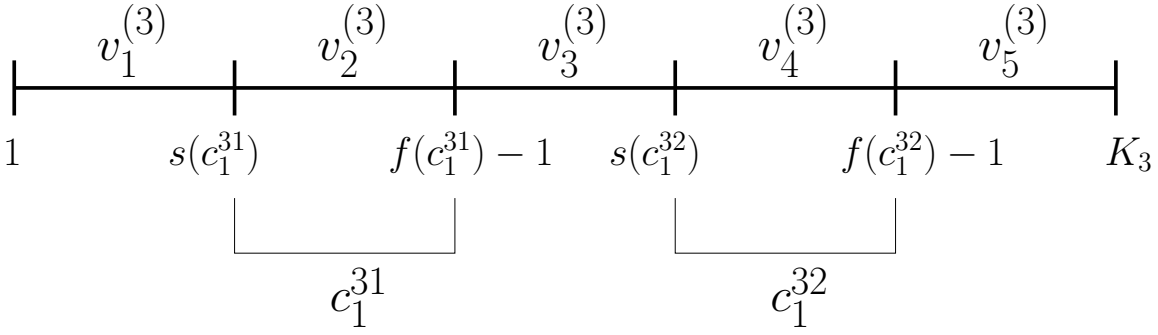
Figure 4.12: Three robots have pairwise collisions in the environment. Dotted lines show the path of the robots.



(a) There are five vertices for robot 1 as the path steps are delimited by the start and the finish of the collision segment with robot 2 and the collision segment with robot 3.



(b) There are five vertices for robot 2 as the path steps are delimited by the start and the finish of the collision segment with robot 1 and the collision segment with robot 3.



(c) There are five vertices for robot 3 as the path steps are delimited by the start and the finish of the collision segment with robot 1 and the collision segment with robot 2.

Figure 4.13: The collision segments for the robots in Figure 4.12 are depicted above. Each robot has two different collision segments, and the collision segments are disjoint, so each collision segment contains only one vertex.

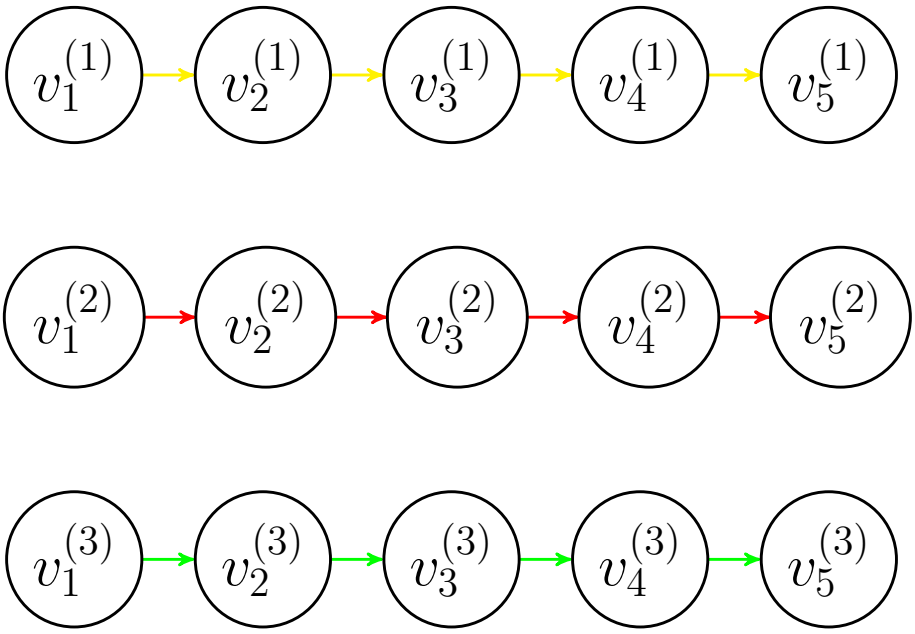


Figure 4.14: All horizontal edges represent the path sequence edges for each robot depicted in Figure 4.12.

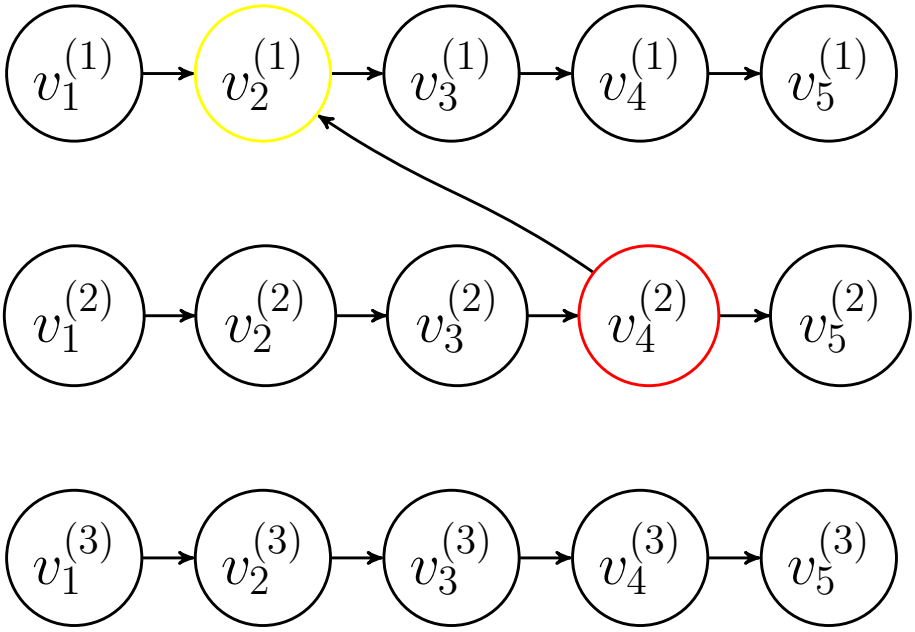


Figure 4.15: The coordination space obstacle o_1^{12} which has label $\ell(o_1^{12}) = 2$. $V(o_1^{12}) = \{v_2^{(1)}\}$ and $V(o_1^{21}) = \{v_4^{(2)}\}$. Then, we have the *obstacle-label edge* from $v_4^{(2)}$ to $v_2^{(1)}$.

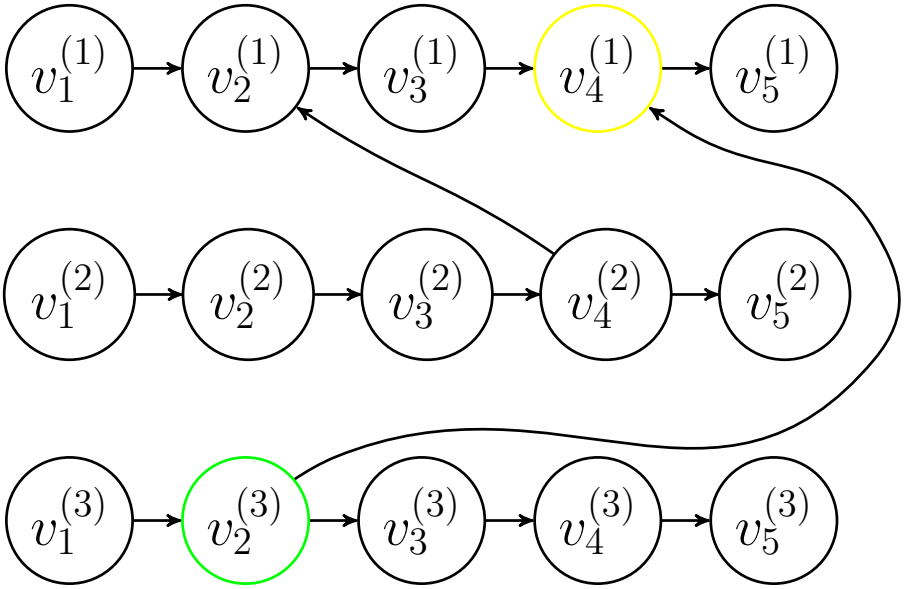


Figure 4.16: The coordination space obstacle o_1^{13} has label $\ell(o_1^{13}) = 3$. $V(o_1^{13}) = \{v_4^{(1)}\}$ and $V(o_1^{31}) = \{v_2^{(3)}\}$. There is an *obstacle-label edge* from $v_2^{(3)}$ to $v_4^{(1)}$.

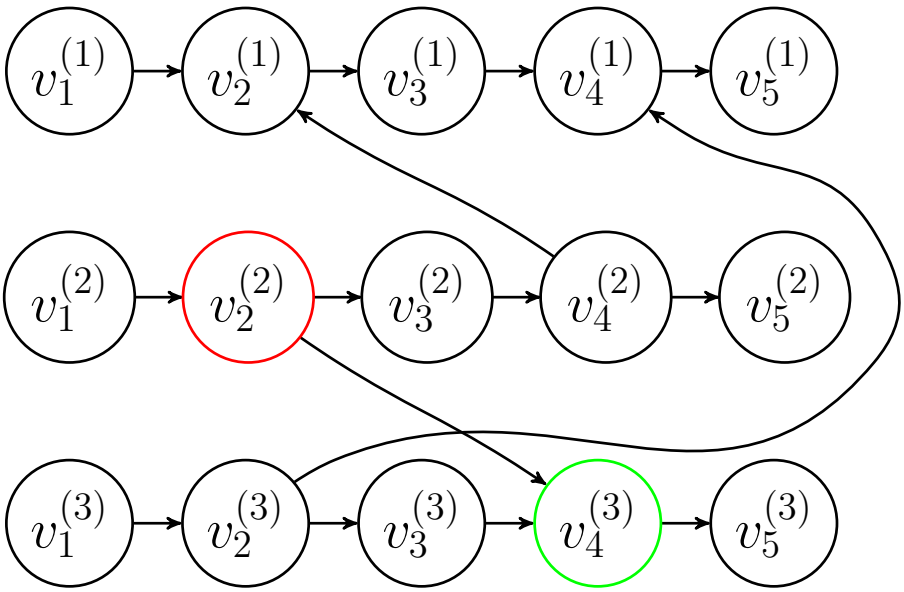


Figure 4.17: The coordination space obstacle o_1^{23} has label $\ell(o_1^{23}) = 2$. $V(o_1^{23}) = \{v_2^{(2)}\}$ and $V(o_1^{32}) = \{v_4^{(3)}\}$. There is an *obstacle-label edge* from $v_2^{(2)}$ to $v_4^{(3)}$.

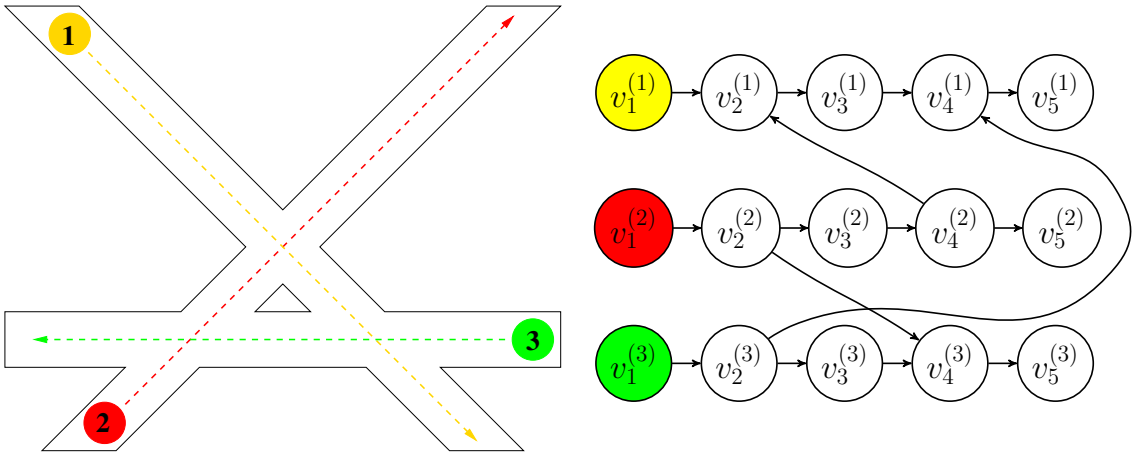


Figure 4.18: All robots are in their start positions. The present vertices of the robots are $v_1^{(1)}$, $v_1^{(2)}$, and $v_1^{(3)}$, respectively.

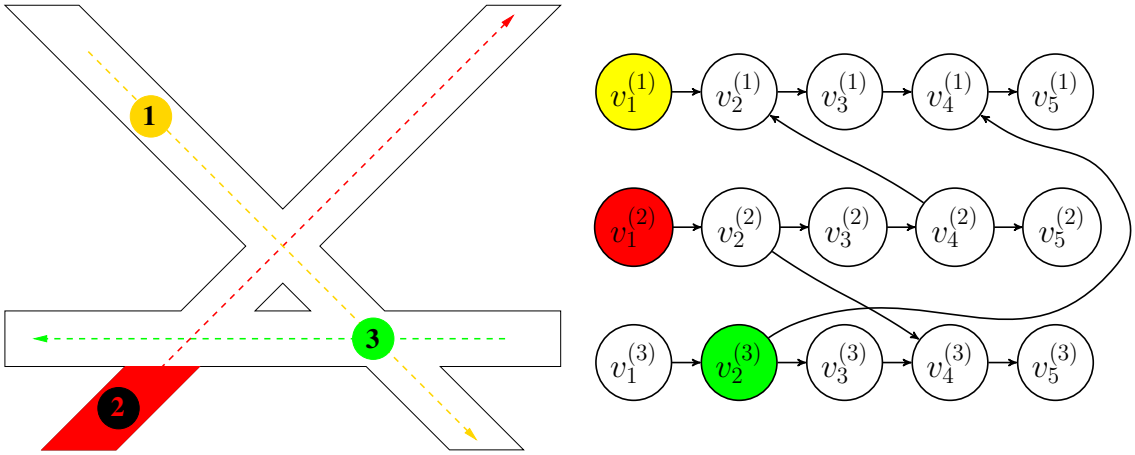


Figure 4.19: All robots are in the moving state. The present vertex of robot 1 is still $v_1^{(1)}$. The present vertex of robot 2 is also still $v_1^{(2)}$, and robot 2 has some delay because of the disturbances. Robot 3 starts the intersection with robot 1, so the present vertex of robot 3 is $v_2^{(3)}$.

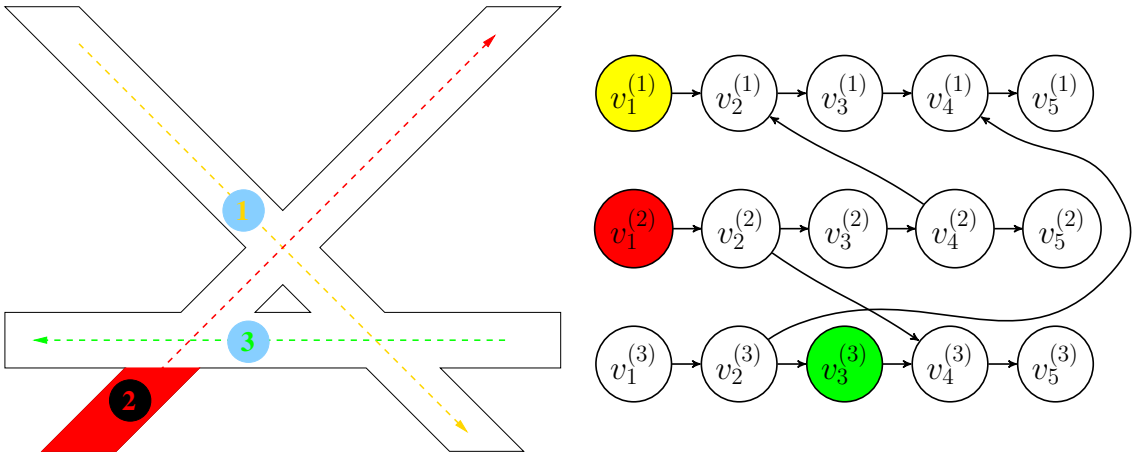


Figure 4.20: Robot 1 reaches the intersection with robot 2, but needs to wait for robot 2 because of the label, $\ell(o_1^{12}) = 2$. Robot 2 has disturbances, so it is delayed before clearing the intersection. Robot 3 also reaches the intersection with robot 2, but needs to wait for robot 2 because of the label, $\ell(o_1^{23}) = 2$. Robot

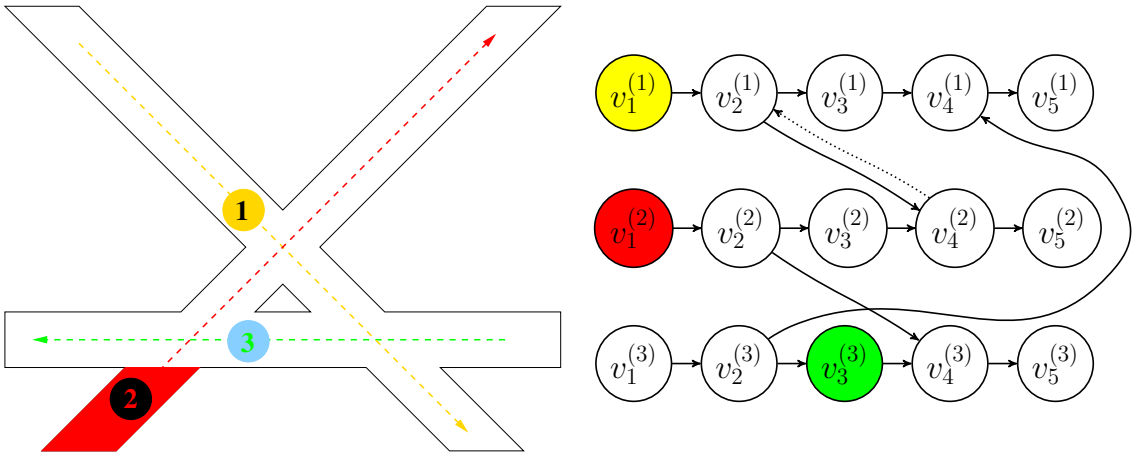


Figure 4.21: Robot 1 flips the label of the obstacle it shares with robot 2, so the obstacle label $\ell(o_1^{12})$ becomes 1, and we need to update the *obstacle-label edge* between robot 1 and robot 2. The updated *obstacle-label edge* is from $v_2^{(1)}$ to $v_4^{(2)}$. Note that the updated obstacle-label edge does not result a cycle in the segment graph.

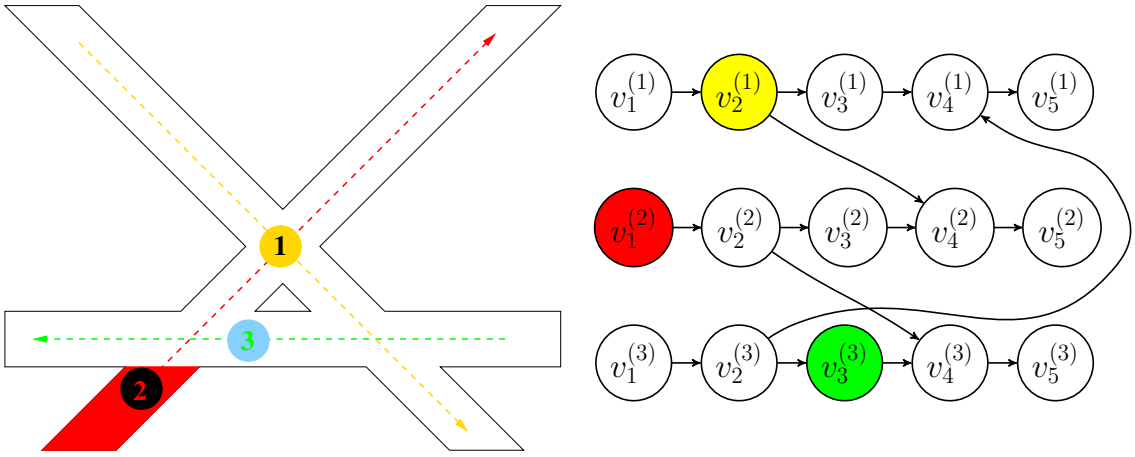


Figure 4.22: Robot 1 starts the intersection with robot 2, so the present vertex of robot 1 is $v_2^{(1)}$. Robot 2 is still having disturbances, and robot 3 still waits for robot 2.

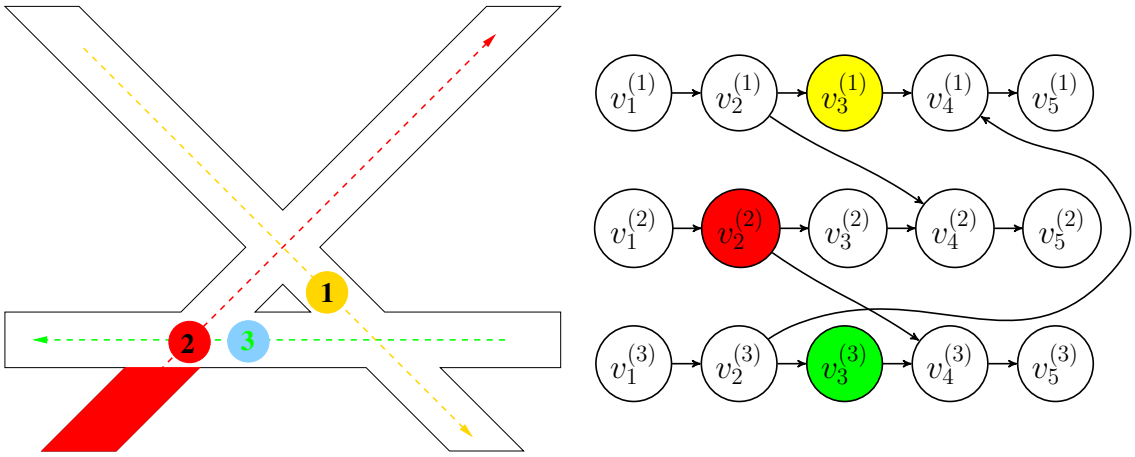


Figure 4.23: Robot 1 clears the intersection with robot 2, so the present vertex of robot 1 is $v_3^{(1)}$. Robot 2 starts the intersection with robot 3, so the present vertex of robot 2 is $v_2^{(2)}$. Robot 2 does not clear the intersection yet, so robot 3 stays in the waiting state.

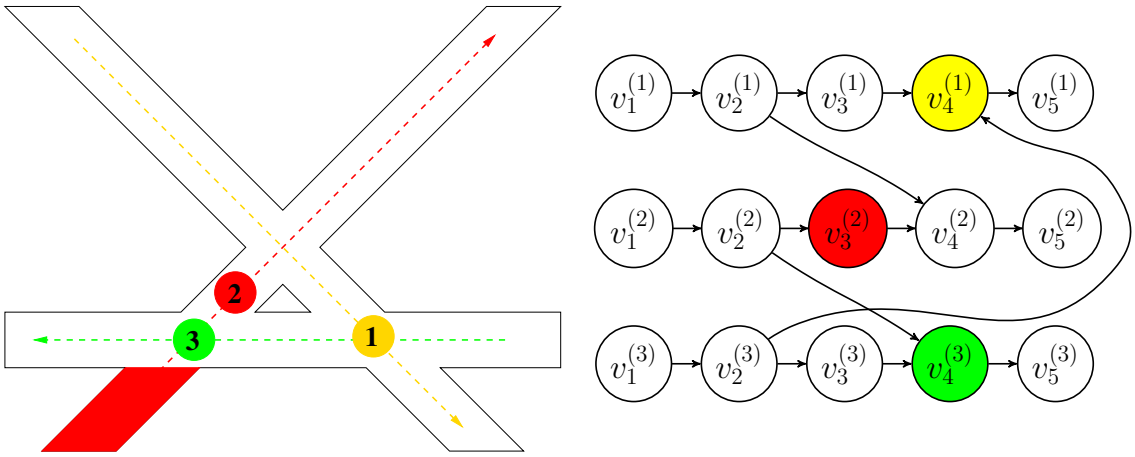


Figure 4.24: Robot 1 starts the intersection with robot 3, so the present vertex of robot 1 is $v_4^{(1)}$. Robot 2 clears the intersection with robot 3, so the present vertex of robot 2 is $v_3^{(2)}$. Robot 3 now starts the intersection with robot 2, so the present vertex of robot 3 is $v_4^{(3)}$.

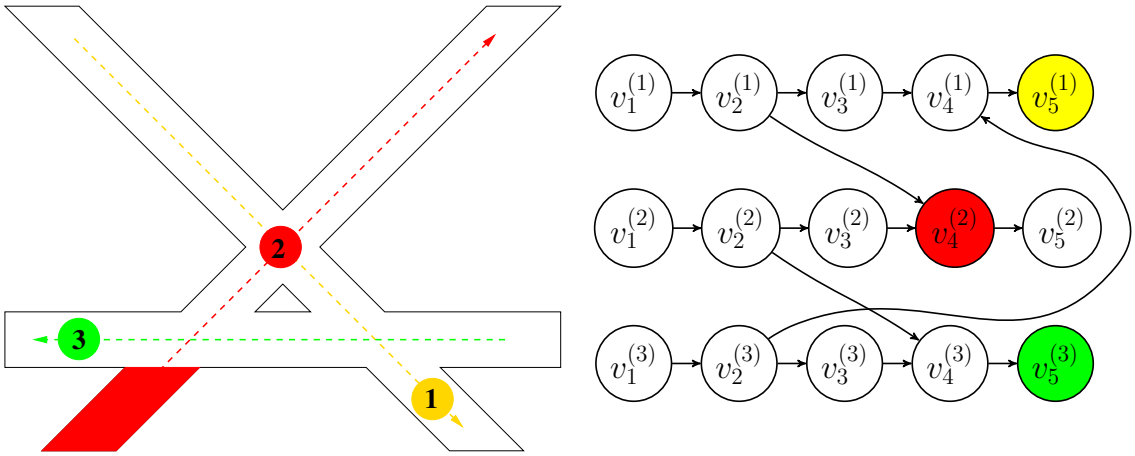


Figure 4.25: Robot 1 clears the intersection with robot 3, so the present vertex of robot 1 is $v_5^{(1)}$. Robot 2 starts the intersection with robot 1, so the present vertex of robot 2 is $v_4^{(2)}$. Robot 3 clears the intersection with robot 2, so the present vertex of robot 3 is $v_5^{(3)}$.

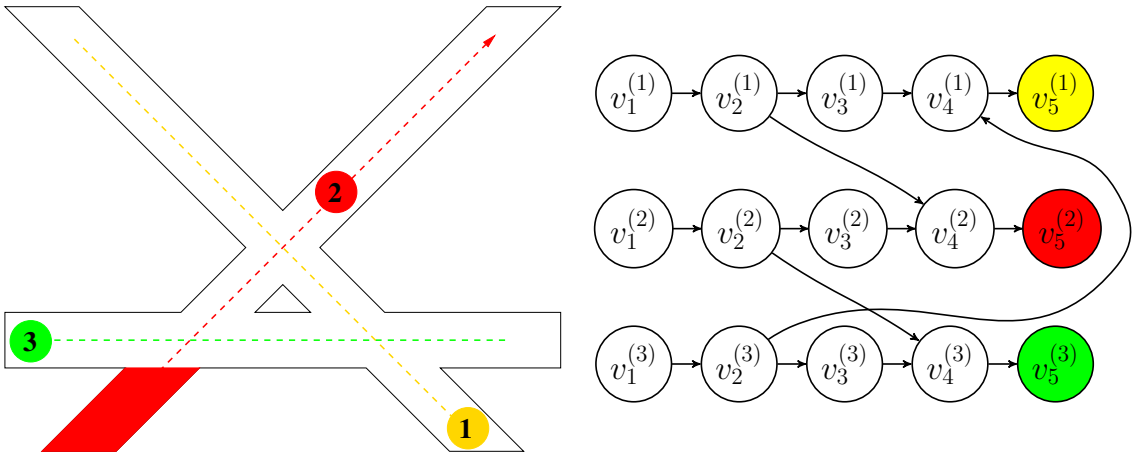


Figure 4.26: Robot 1 and robot 3 reach the goal position, so they are in the finished state. Robot 2 clears the intersection with robot 1, so the present vertex of robot 2 is $v_5^{(2)}$.

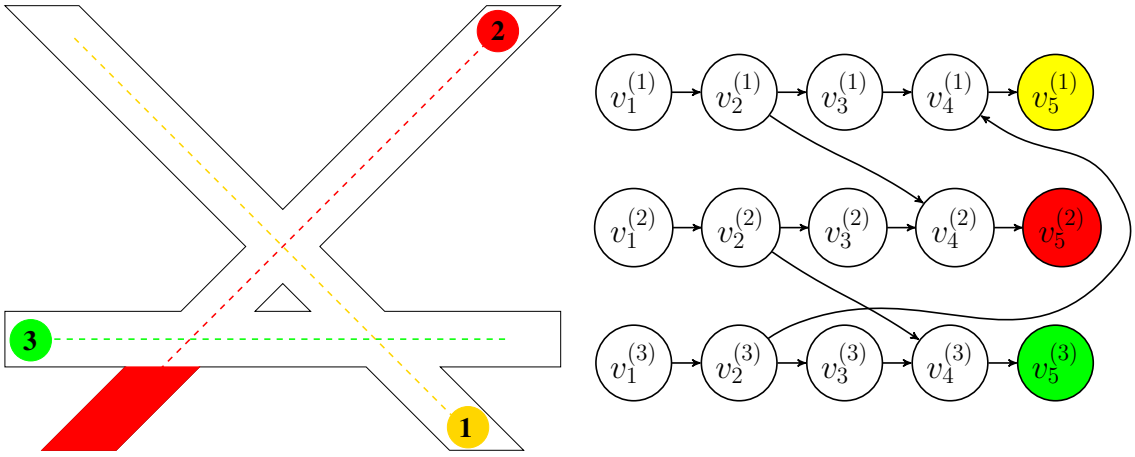


Figure 4.27: Robot 2 also reaches the goal position, so all robots are in the finished state.

4.3 Experimental Results

We have extended the original code for RMTRACK simulation[2] by adding flipping algorithms and our new deadlock detection method, and implemented it in Java with an Ubuntu 20.04 computer and a 2.9GHz processor.

The environment for the experiments in this chapter is shown in Figure 4.28. The probability of having disturbance in the red area is 0.85, and for the rest of the environment, it is 0.05. For each number of robots (5, 10, 15, and 20), we have 10 different randomly generated start and goal positions.

The average travel times of RMTRACK and two flipping algorithms are shown in Figure 4.29. The flipping algorithms reduced the travel times and avoided deadlock, so that all robots reached their goal positions as shown in Figure 4.32.

The average computation time for RMTRACK+TFA is longer than the average computation time for RMTRACK+TFF as shown in Figure 4.30 because RMTRACK+TFA calculates the expected travel times by considering the movements of both robots with or without flipping the label of the obstacle. Also, note that computation time in general is small when compared with travel time.

The number of flips is almost the same for two flipping algorithms in Figure 4.31 because the collisions between two robots may happen only in a small area of their trajectories. On the other hand, if two robots travels in a long and narrow passage, then collisions may happen in a large area of their trajectories, and RMTRACK+TFA performs better than RMTRACK+TFF as shown in Chapter 3.

Simulation videos are provided in the link.[3]

[2] `https://github.com/mcapino/rmtrack`

[3] `https://www.dropbox.com/sh/n3wk1wuxn875nw2/AAA3fmOdM41zPTJTXMOgZzAQa?dl=0`

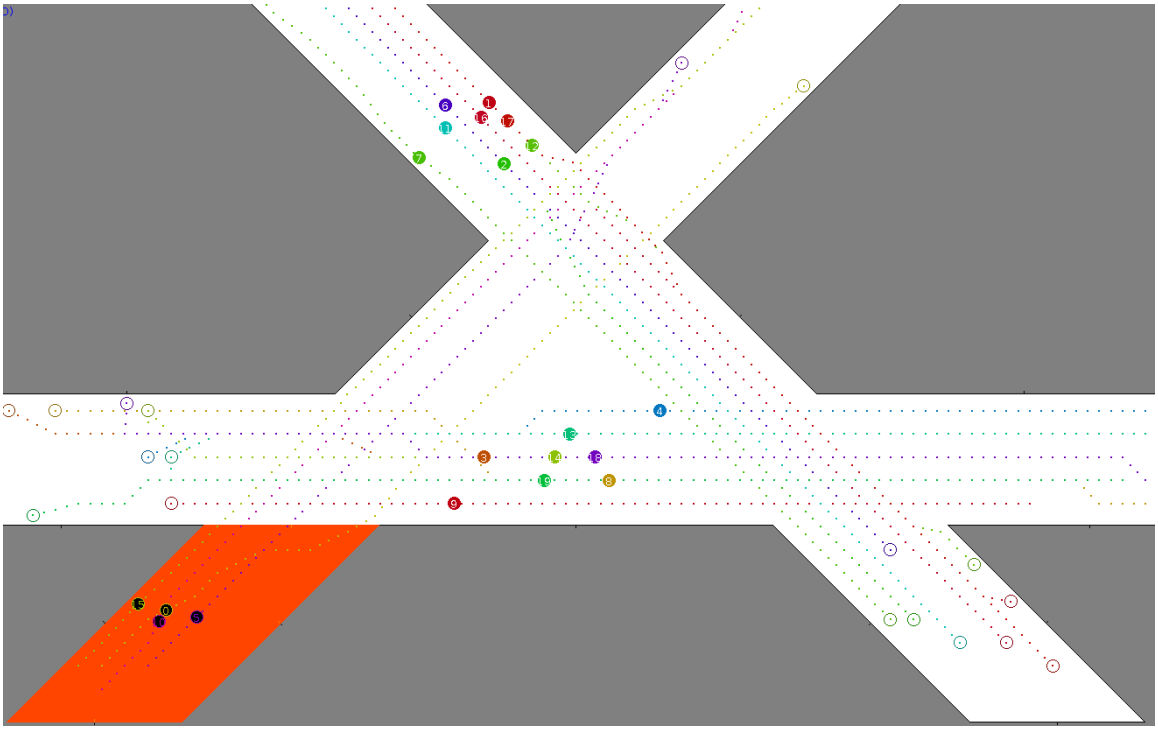


Figure 4.28: An environment with 20 robots. The dotted lines represent the planned trajectories of the robots. The robots in the red area are subject to a disturbance, which may be caused by interaction with humans or other objects and results in a delay in their progress.

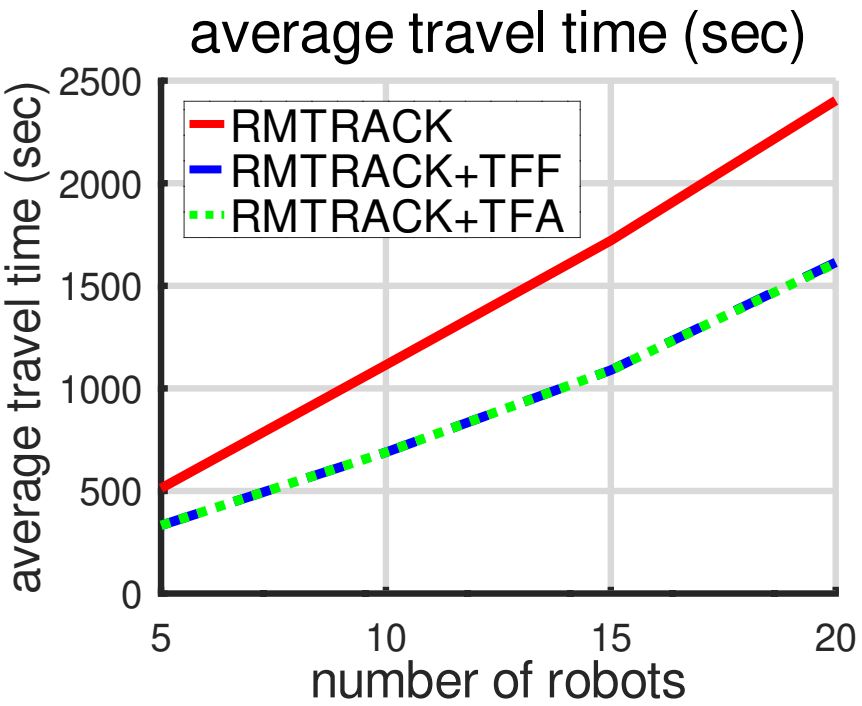


Figure 4.29: Average travel time for the number of robots (5, 10, 15, 20)

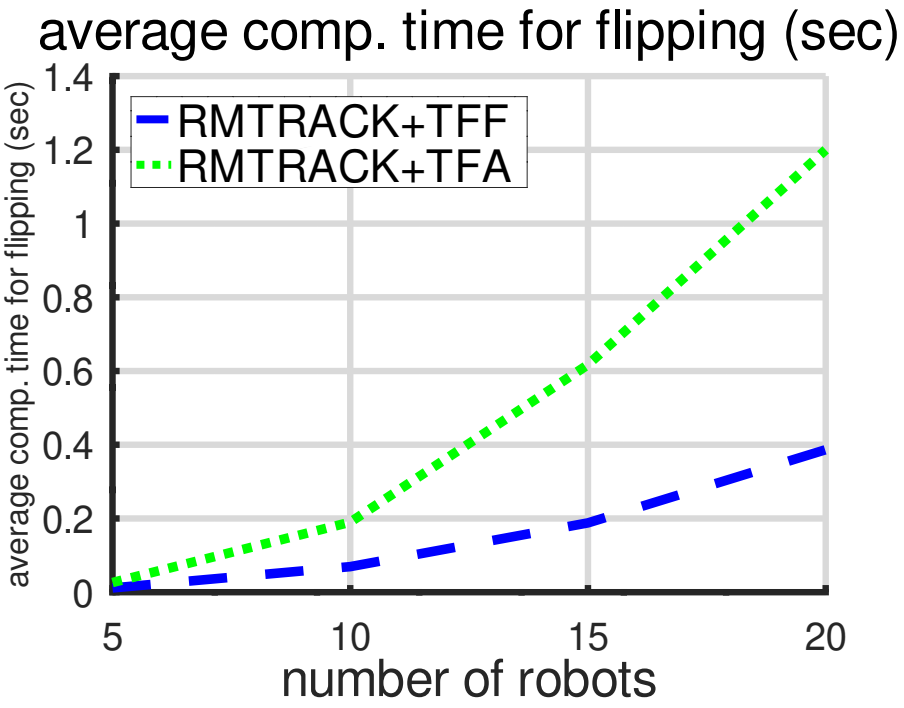


Figure 4.30: Average computational time for the flipping algorithms

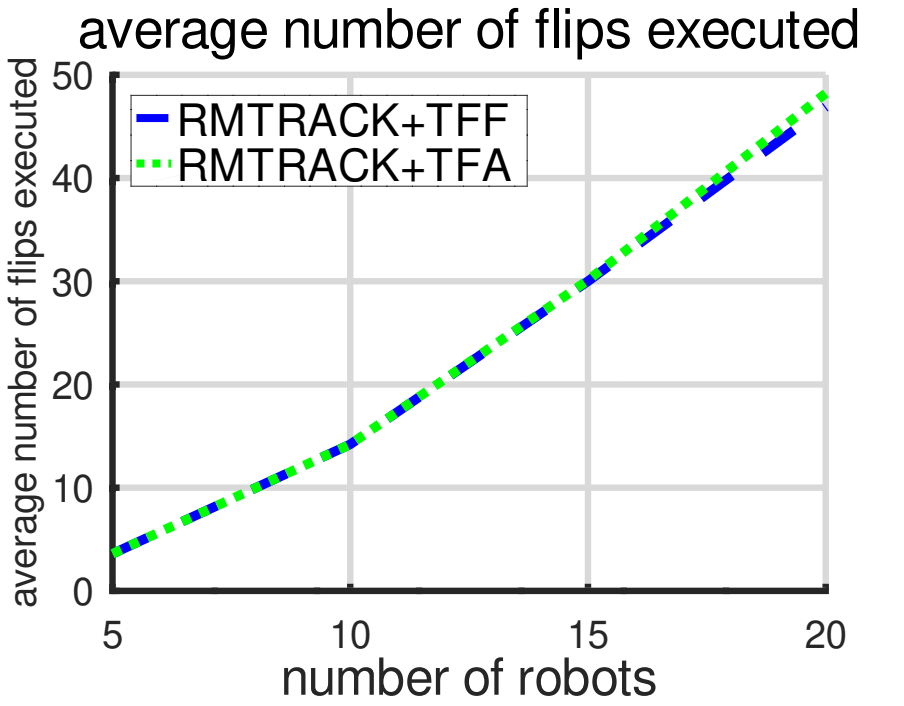


Figure 4.31: Average number of flips executed for the flipping algorithms

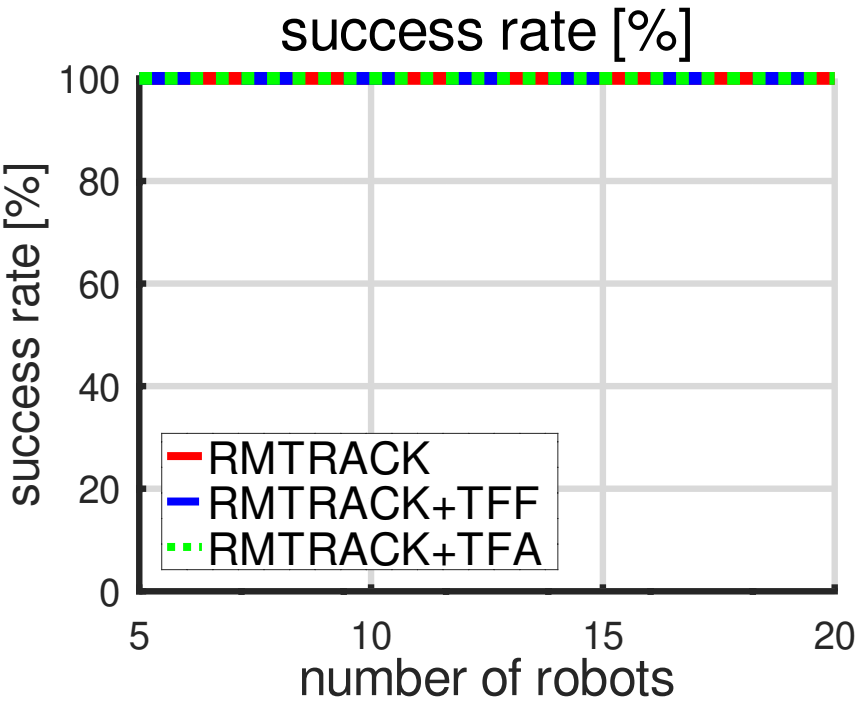


Figure 4.32: Success rate of RMTRACK and the flipping algorithms, so all robots reach their goal positions.

Chapter 5

Conclusion and Future Work

This dissertation presented a technique for online repair of multiple-robot coordination plans. The idea is to start from a planned joint trajectory for the robots, but to adjust the path of each robot by 'flipping' the order in which pairs of robots should pass through their shared collision regions. These decisions are made on-the-fly, without full replanning, in response to unexpected disturbances that result in changes of the execution speed of some of the robots along their paths.

We also provided a theoretical analysis of the conditions under which flipping algorithms lead to deadlock and conditions for deadlock-free environments even in the presence of flips. Our simulation results complement the theoretical ones by showing that flipping algorithms significantly reduce robot travel times when the difference in disturbance probability between different areas is large. However, since flipping algorithms may lead to deadlock in some situation, we have proposed an algorithm based on the segment graph data structure to detect and avoid deadlocks before flipping, thus combining the efficiency of flipping algorithms with a theoretical guarantee of deadlock avoidance.

We provide an appendix with the documentation of the simulation software and a link to its source code.

In future work, we plan to consider a decentralized version of this problem, in which robots have limited information about the progress made by the other robots but must nevertheless decide how to proceed. We intend to use segment graphs to detect and avoid deadlocks in such decentralized approaches to multi-robot co-

ordination. We plan to implement and test the flipping algorithms in a real-world environment with real robots, exploit their available sensors, and compare relative advantages and disadvantages. On the theoretical side, we conjecture that our flipping algorithms are Pareto optimal and plan to investigate this conjecture.

Bibliography

[1] Javier Alonso-Mora, Andreas Breitenmoser, Martin Rufli, Paul Beardsley, and Roland Siegwart. Optimal reciprocal collision avoidance for multiple non-holonomic robots. In *Distributed autonomous robotic systems*, pages 203–216. Springer, 2013.

[2] Toshiro Araki, Yuji Sugiyama, Tadao Kasami, and Jun Okui. Complexity of the deadlock avoidance problem. In *2nd IBM Symp. on Mathematical Foundations of Computer Science*, pages 229–257. IBM, 1977.

[3] Tsz-Chiu Au, Neda Shahidi, and Peter Stone. Enforcing liveness in autonomous traffic management. In *AAAI*, 2011.

[4] Mohammad Bahram, Constantin Hubmann, Andreas Lawitzky, Michael Aeberhard, and Dirk Wollherr. A combined model-and learning-based framework for interaction-aware maneuver prediction. *IEEE Transactions on Intelligent Transportation Systems*, 17(6):1538–1550, 2016.

[5] Daman Bareiss and Jur van den Berg. Generalized reciprocal collision avoidance. *The International Journal of Robotics Research*, 34(12):1501–1514, 2015.

[6] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and autonomous systems*, 41(2-3):89–99, 2002.

[7] Alexander Berndt, Niels Van Duijkeren, Luigi Palmieri, and Tamas Keviczky. A feedback scheme to reorder a multi-agent execution schedule by persistently optimizing a switchable action dependency graph. *arXiv preprint arXiv:2010.05254*, 2020.

[8] Zeungnam Bien and Jihong Lee. A minimum-time trajectory planning method for two robots. *IEEE Transactions on Robotics and Automation*, 8(3):414–418, 1992.

[9] Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, pages 195–210, 1996.

[10] Craig Boutilier. Sequential optimality and coordination in multiagent systems. In *IJCAI*, volume 99, pages 478–485, 1999.

[11] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[12] Michal Čáp, Jean Gregoire, and Emilio Frazzoli. Provably safe and deadlock-free execution of multi-robot plans under delaying disturbances. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 5113–5118. IEEE, 2016.

[13] Michal Čáp, Peter Novák, Alexander Kleiner, and Martin Seleckỳ. Prioritized planning algorithms for trajectory coordination of multiple mobile robots. *IEEE transactions on automation science and engineering*, 12(3):835–849, 2015.

[14] Michal Čáp, Jiří Vokřínek, and Alexander Kleiner. Complete decentralized method for on-line multi-robot trajectory planning in well-formed infrastructures. In *Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling*, pages 324–332, 2015.

[15] Cheol Chang, Myung Jin Chung, and Bum Hee Lee. Collision avoidance of two general robot manipulators by minimum delay time. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(3):517–522, 1994.

[16] Hamid Chitsaz, Steven M. LaValle, and Jason M. O'Kane. Exact Pareto-optimal coordination for two translating polygonal robots on a cyclic roadmap. In *Proc. Canadian Conference on Computational Geometry*, 2008.

[17] Hamid Chitsaz, Jason M. O'Kane, and Steven M. LaValle. Exact Pareto-optimal coordination for two translating polygonal robots on an acyclic roadmap. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3981–3986, 2004.

[18] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[19] Adem Coskun and Jason M O'Kane. Online plan repair in multi-robot coordination with disturbances. In *International Conference on Robotics and Automation (ICRA)*, pages 3333–3339. IEEE, 2019.

[20] Kurt Dresner and Peter Stone. Multiagent traffic management: A reservation-based intersection control mechanism. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 530–537, 2004.

[21] Kurt Dresner and Peter Stone. Mitigating catastrophic failure at intersections of autonomous vehicles. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1393–1396, 2008.

[22] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research*, 31:591–656, 2008.

[23] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 2(1-4):477, 1987.

[24] Cornelia Ferner, Glenn Wagner, and Howie Choset. Odrm* optimal multirobot path planning in low dimensional search spaces. In *2013 IEEE International Conference on Robotics and Automation*, pages 3854–3859. IEEE, 2013.

[25] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998.

[26] Thierry Fraichard and Christian Laugier. Planning movements for several coordinated vehicles. In *IROS*, volume 89, pages 466–472, 1989.

[27] Robert Ghrist, Jason M. O'Kane, and Steven M. LaValle. Pareto optimal coordination on roadmaps. In *Proc. International Workshop on the Algorithmic Foundations of Robotics*, pages 185–200, 2004.

[28] Robert Ghrist, Jason M. O'Kane, and Steven M. LaValle. Computing pareto optimal coordinations on roadmaps. *International Journal of Robotics Research*, 24(11):997–1010, November 2005.

[29] Tobias Gindele, Sebastian Brechtel, and Rudiger Dillmann. Learning driver behavior models from traffic observations for decision making and planning. *IEEE Intelligent Transportation Systems Magazine*, 7(1):69–79, 2015.

[30] Jaskaran Grover, Changliu Liu, and Katia Sycara. Deadlock analysis and resolution in multi-robot systems (extended version). *arXiv preprint arXiv:1911.09146*, 2019.

[31] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored mdps. In *Advances in neural information processing systems*, pages 1523–1530, 2002.

[32] Yi Guo and Lynne E Parker. A distributed and optimal motion planning approach for multiple mobile robots. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, volume 3, pages 2612–2619. IEEE, 2002.

[33] Stephen J Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming Lin, Dinesh Manocha, and Pradeep Dubey. Clearpath: highly parallel collision avoidance for multi-agent simulation. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 177–187. ACM, 2009.

[34] Greg Hamerly and Charles Elkan. Learning the k in k-means. *Advances in neural information processing systems*, 16:281–288, 2004.

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

[36] Matthew Hausknecht, Tsz-Chiu Au, and Peter Stone. Autonomous intersection management: Multi-intersection optimization. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4581–4586. IEEE, 2011.

[37] John E Hopcroft, Jacob Theodore Schwartz, and Micha Sharir. On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman's problem". *The International Journal of Robotics Research*, 3(4):76–88, 1984.

[38] Kamal Kant and Steven W Zucker. Toward efficient trajectory planning: The path-velocity decomposition. *The international journal of robotics research*, 5(3):72–89, 1986.

[39] Dietmar Kasper, Galia Weidl, Thao Dang, Gabi Breuel, Andreas Tamke, Andreas Wedel, and Wolfgang Rosenstiel. Object-oriented Bayesian networks for

detection of lane change maneuvers. *IEEE Intelligent Transportation Systems Magazine*, 4(3):19–31, 2012.

[40] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[41] Mark Lawley and Spyros Reveliotis. Deadlock avoidance for sequential resource allocation systems: Hard and easy cases. *International Journal of Flexible Manufacturing Systems*, 13(4):385–404, 2001.

[42] Stephane Leroy, Jean-Paul Laumond, and Thierry Siméon. Multiple path coordination for mobile robots: A geometric algorithm. In *IJCAI*, volume 99, pages 1118–1123, 1999.

[43] Junxiang Li, Bin Dai, Xiaohui Li, Xin Xu, and Daxue Liu. A dynamic Bayesian network for vehicle maneuver prediction in highway driving scenarios: Framework and verification. *Electronics*, 8(1):40, 2019.

[44] Tomas Lozano-Perez. Spatial planning: A configuration space approach. In *Autonomous robot vehicles*, pages 259–271. Springer, 1990.

[45] Tomas Lozano-Perez et al. Deadlock-free and collision-free coordination of two robot manipulators. In *1989 IEEE International Conference on Robotics and Automation*, pages 484–489. IEEE, 1989.

[46] Vladimir J. Lumelsky and KR Harinarayan. Decentralized motion planning for multiple mobile robots: The cocktail party model. *Autonomous Robots*, 4(1):121–135, 1997.

[47] Hang Ma, TK Satish Kumar, and Sven Koenig. Multi-agent path finding with delay probabilities. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[48] Raz Nissim and Ronen Brafman. Distributed heuristic forward search for multi-agent planning. *Journal of Artificial Intelligence Research*, 51:293–332, 2014.

[49] Spyros A Reveliotis and Elzbieta Roszkowska. On the complexity of maximally permissive deadlock avoidance in multi-vehicle traffic systems. *IEEE Transactions on Automatic Control*, 55(7):1646–1651, 2010.

[50] Spyros A Reveliotis and Elzbieta Roszkowska. Conflict resolution in free-ranging multivehicle systems: A resource allocation paradigm. *IEEE Transactions on Robotics*, 27(2):283–296, 2011.

[51] Elzbieta Roszkowska and Spyros A Reveliotis. On the liveness of guidepath-based, zone-controlled dynamically routed, closed traffic systems. *IEEE Transactions on Automatic Control*, 53(7):1689–1695, 2008.

[52] Matthias Schreier, Volker Willert, and Jürgen Adamy. An integrated approach to maneuver-based trajectory prediction and criticality assessment in arbitrary road environments. *IEEE Transactions on Intelligent Transportation Systems*, 17(10):2751–2766, 2016.

[53] Kang G Shin and Qin Zheng. Minimum-time collision-free trajectory planning for dual-robot systems. *IEEE Transactions on Robotics and Automation*, 8(5):641–644, 1992.

[54] Kiril Solovey and Dan Halperin. On the hardness of unlabeled multi-robot motion planning. *The International Journal of Robotics Research*, 35(14):1750–1759, 2016.

[55] Paul Spirakis and Chee K Yap. Strong np-hardness of moving many discs. *Information Processing Letters*, 19(1):55–59, 1984.

[56] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. In *IJCAI*, pages 668–673, 2011.

[57] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, volume 1, pages 28–29. Atlanta, GA, 2010.

[58] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[59] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer, 2011.

[60] Jur Van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1928–1935. IEEE, 2008.

[61] Jur P Van Den Berg and Mark H Overmars. Prioritized motion planning for multiple robots. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 430–435. IEEE, 2005.

[62] Prasanna Velagapudi, Katia Sycara, and Paul Scerri. Decentralized prioritized planning in large multirobot teams. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4603–4609. IEEE, 2010.

[63] Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *2011 IEEE/RSJ international conference on intelligent robots and systems*, pages 3260–3267. IEEE, 2011.

[64] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, 2015.

[65] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9–9, 2008.

Appendix A

Simulation

This appendix contains the documentation for the simulation software, used for the experiments described in Section 3.4 and Section 4.3, and available at `https://github.com/acoskunUSC/flipping.git`.

A.1 Creating an Instance

Problem instances are described in an XML (eXtensible Markup Language) file. Figure A.1 shows an example XML file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<multiagentproblem>
    <environment>
        <boundary>0,0 0,1000 800,1000 800,400 850,400 850,1000 1000,1000 1000,0 </boundary>
        <obstacles>
            <obstacle>150,150 400,150 400,400 150,400 </obstacle>
            <obstacle>450,650 650,650 650,850 450,850 </obstacle>
        </obstacles>
        <disturbances>
            <polygon probability="0.6">860,400 990,400 990,990 860,990</polygon>
            <circle probability="0.7" center="625,275" radius="200"/>
        </disturbances>
    </environment>
    <graph>
        <vertices>
            37,37 62,37 87,37 112,37 137,37 162,37 187,37 212,37 237,37 262,37 287,37 312,37 337,37
        </vertices>
        <edges>
            37,37 42,27 ;37,37 27,27 ;37,37 37,62 ;37,37 62,37 ;37,37 62,62 ;37,37 27,76 ;37,37 37,87 ;3
        </edges>
    </graph>
    <agents>
        <agent maxspeed="0.0500" radius="25" start="37,37" target="973,973"/>
        <agent maxspeed="0.0500" radius="25" start="973,27" target="27,973"/>
    </agents>
    <docks/>
</multiagentproblem>
```

Figure A.1: An example XML file. Only the first several vertices and edges are shown.

The root element, *multiagentproblem*, must have three child elements, which are *environment*, *graph*, and *agents*. One other child element, *docks*, is optional. The

docks element in Figure A.1 is empty, which means it has no content, so this is represented as <docks/>.

A.1.1 Environment

Environment element needs to have three child elements, *boundary*, *obstacles*, and *disturbances*.

Boundary

Boundaries are polygons. The coordinates of the corners of a boundary are specified as an (x, y) pairs, separated by spaces. Note that boundary is drawn counterclockwise, starting at any corner. Figure A.2 shows the environment, whose boundary points are listed in Figure A.1.

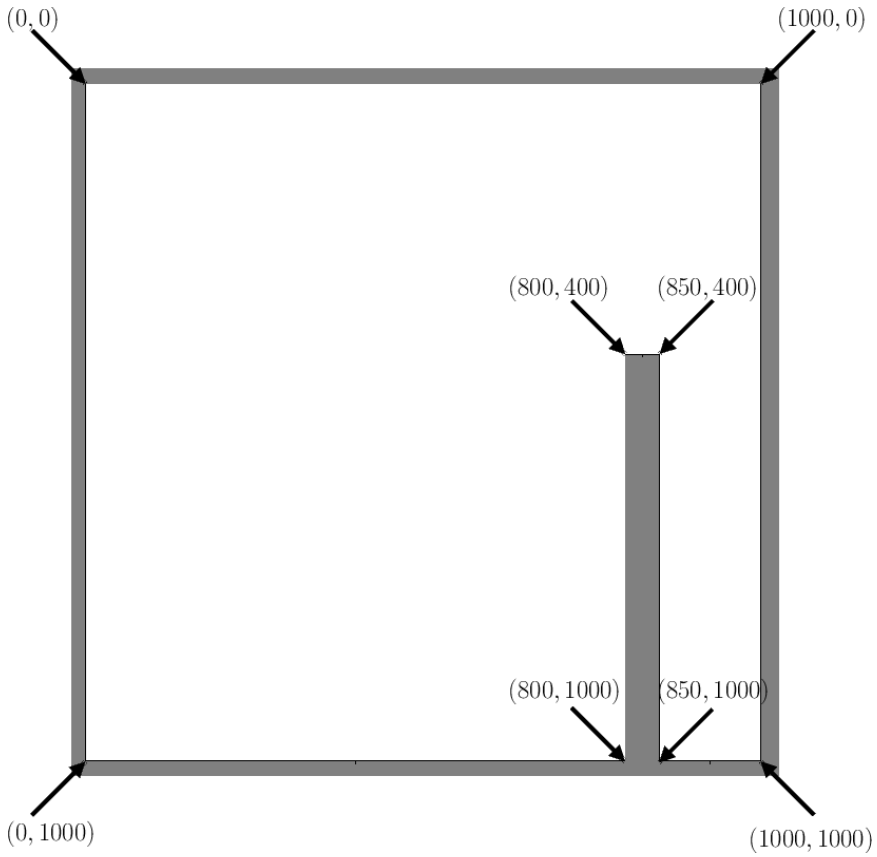


Figure A.2: The environment with the boundary points.

Obstacles

Obstacles are represented as a polygon. Note that obstacles are drawn clockwise, starting at any corner. Even when no obstacles are specified, an empty obstacles element, denoted <obstacles/>, needs to be included. Figure A.3 illustrates two obstacles.

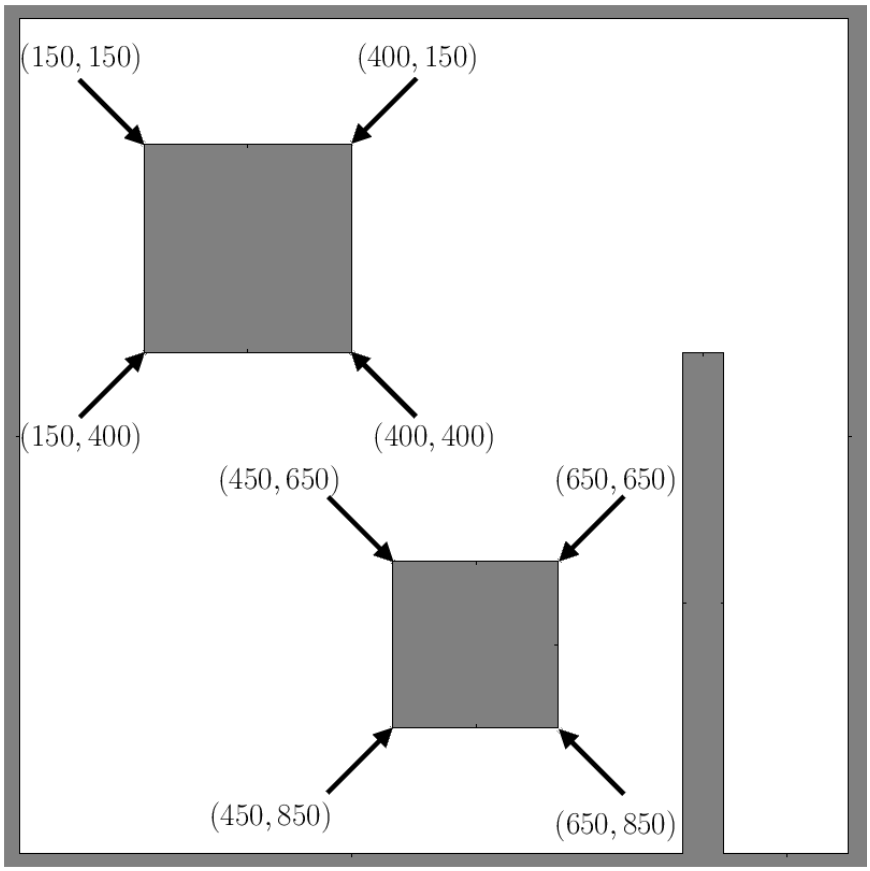


Figure A.3: The environment with two obstacles.

Disturbances

Disturbances can be represented as a polygon or a circle. Each shape needs to take a probability attribute. Circle is a void element, but has two additional attributes, center and radius. Figure A.4 illustrates two different disturbances.

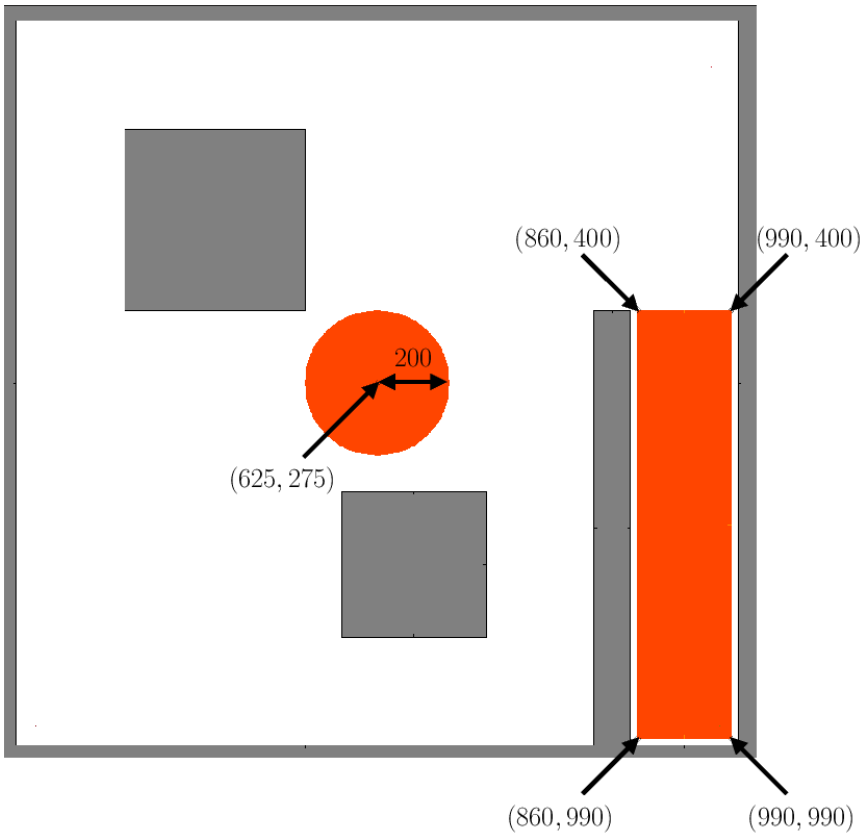


Figure A.4: The environment with disturbances.

A.1.2 Graph

To generate the graph element, which consists of vertices and edges elements, you need to run the *TriangulationGenerator* method with the following parameters:

- -problemfile: an XML file that contains the environment element
- -bodyradius: radius of the agent
- -dispersion: density of the points, which is $\sqrt{2} * gridstep$
- -connectionradius: distance between two vertices
- -generateinfrastructure: generates non-conflicting trajectories
- -outfile: an XML file that contains the environment and graph elements

Figure A.5 shows the generated vertices and edges of the example environment.

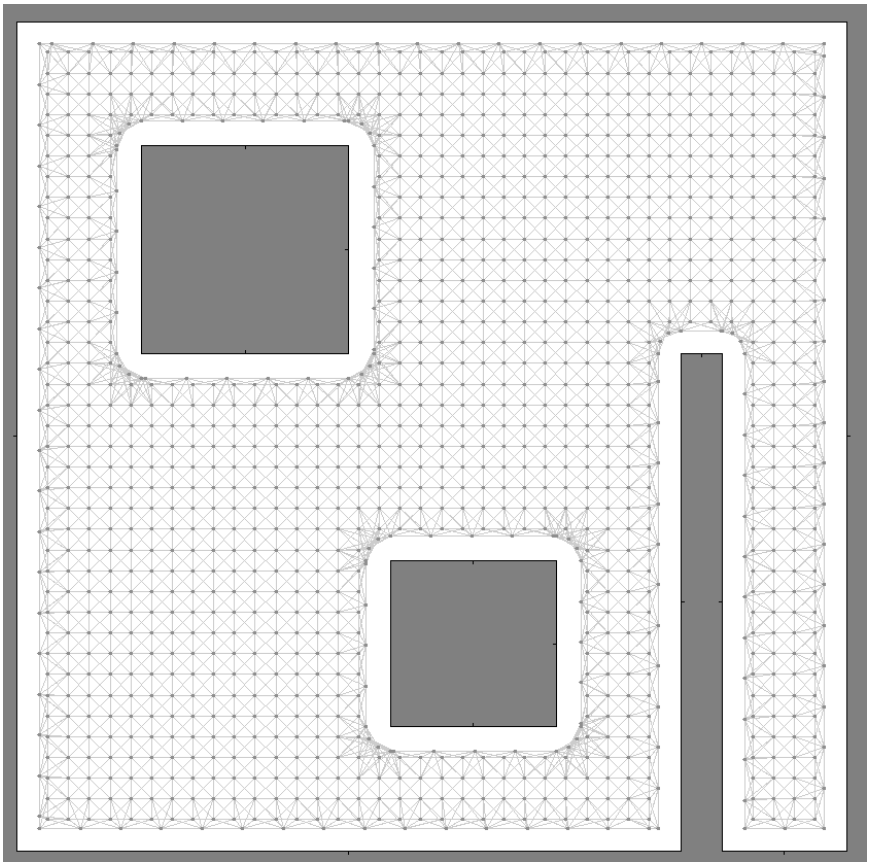


Figure A.5: The environment with the graph.

A.1.3 Agents

Each agent elemtent contains maximum speed, radius, start position, and target position as attributes. Each start and goal position must be a vertex. You can also generate a random start and target position by calling the *GenerateEAInstance*[1] method with the following methods:

- -env: an XML file that contains the environment and graph elements.
- -nagents: number of agents
- -radius: radius of the agent
- -maxspeed: maximum speed of the agent

[1]EA stands for Earliest Arrival.

- -seed: a number to initialize a pseudorandom number generator
- -sgnooverlap: start and goals can overlap
- -sgavoiding: start and goals are separated
- -outfile: an XML file that contains the environment, graph, and agents elements.

Figure A.6 shows two agents with their start and goal positions.

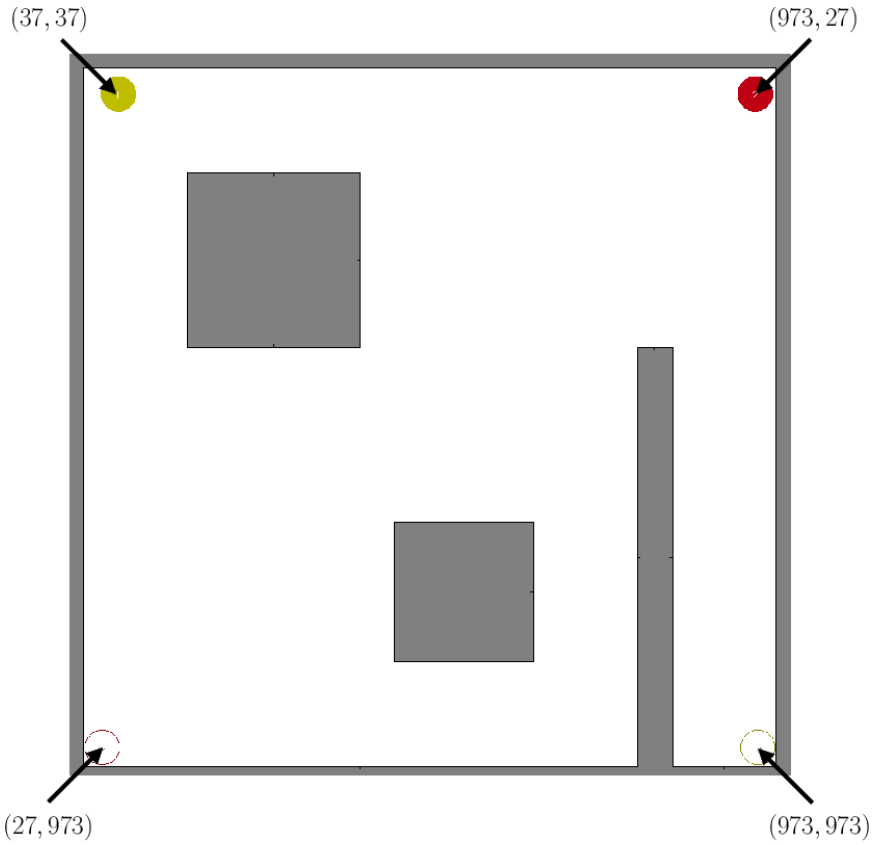


Figure A.6: The environment with two agents.

A.1.4 Problem Instance Designer

Instead of specifying obstacles and agents pixel by pixel, one can draw them by calling *ProblemInstanceDesigner* method with the -outfile parameter. Figure A.7 shows the interface of the application.

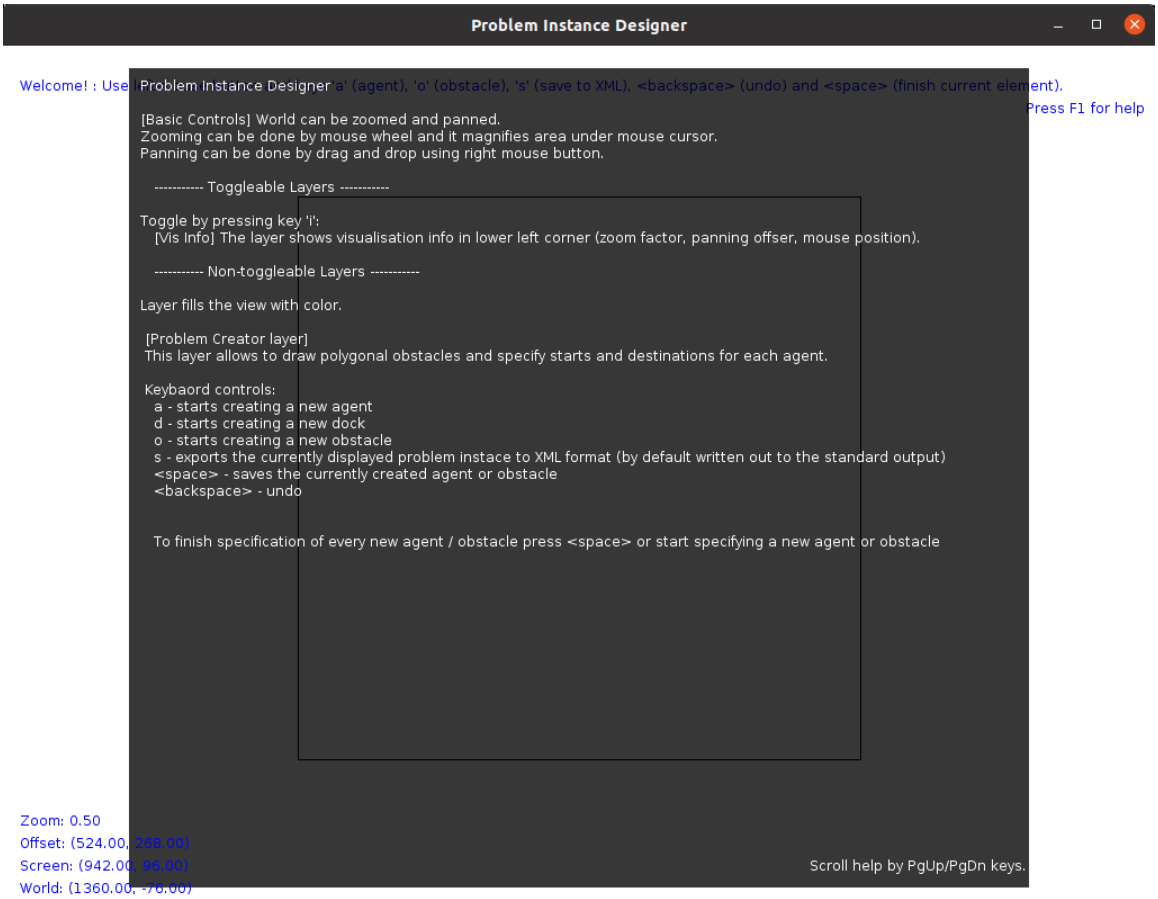


Figure A.7: Problem Instance Designer

A.2 Running the Instance

After you create the problem instance, one can generate the trajectories of each robot and run the flipping algorithms by calling the *ScenarioCreator* method with the following methods:

- -problemfile: an XML file that contains the environment, graph, and agents elements. Figure A.1 shows a complete example.
- -method: a parameter that specifies one of five available methods for controlling the actions.
  - ORCA: One of the most practicable reactive method, the optimal reciprocal collision avoidance.

  - ALLSTOPS: Stops all robots whenever any robot has a disturbance.
  - RMTRACK: Described in Section 2.5
  - RMTRACK_TFF: Described in Section 3.3.2
  - RMTRACK_TFA: Described in Section 3.3.3

- -maxtime: the maximum time (in milliseconds) for trajectory planning
- -timestep: the timeout time (in milliseconds) when the simulation will be terminated after absence of progress
- -dprob: the disturbance probability for the part of the environment that does not belong to any specified disturbance elements
- -avoidDeadlock: for RMTRACK_TFF and RMTRACK_TFA methods
- -showvis: toggles for the following visualization options
  - m: shows the missions of agents with an arrow from start position to target position
  - t: shows the trajectories of the agents
  - g: shows the planned graph
  - p: shows the polygons for obstacles
  - d: shows the areas for disturbances

Figure A.8 shows the generated trajectories for two agents.

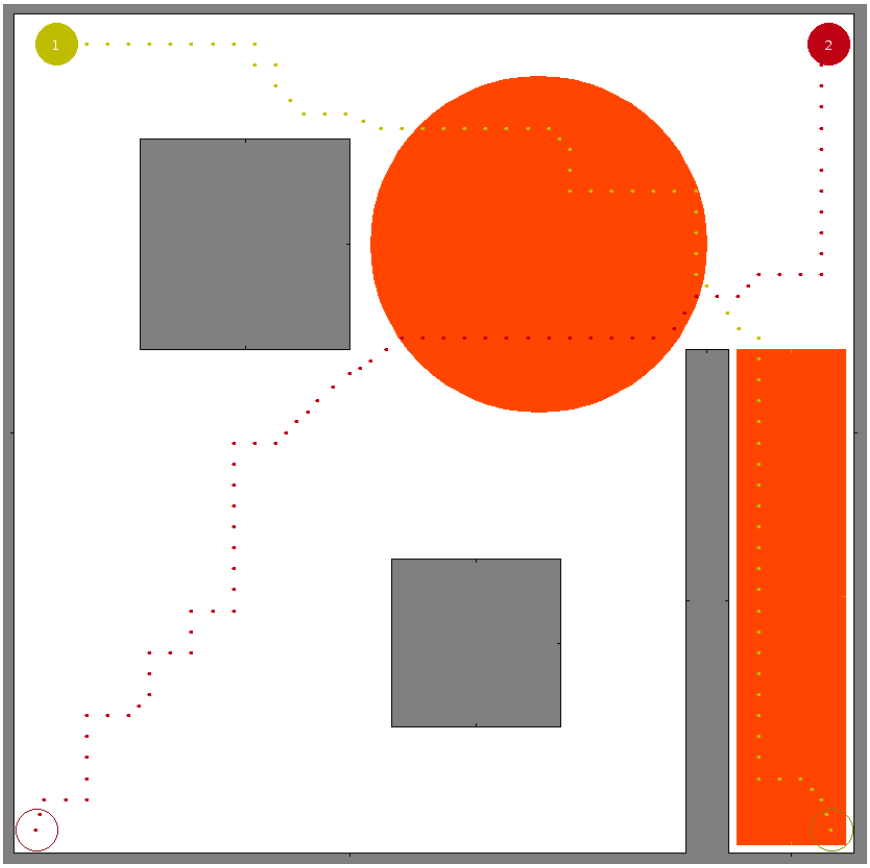


Figure A.8: Trajectories for two robots generated using *ScenarioCreator*.