Rigorous Movement of Convex Polygons on a Path Using Multiple Robots

By

Pierre Chamoun

A thesis submitted
to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science (OCICS)
School of Computer Science
Carleton University
Ottawa, Ontario, Canada

July 5, 2012

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-93498-2

Our file Notre référence
ISBN: 978-0-494-93498-2

Abstract

This thesis describes an approach for pushing a convex polygonal object, M, with rigor using multiple robots, along a desired rectilinear path, P, in a two-dimensional polygonal environment. The goal is to rigorously push M along P while preserving the orientation and the alignment of M, as well as precisely rotating it about its center when necessary. A path planning algorithm is presented which computes a shortest-path approximation P between two points in the environment. In general, the path requires both translations and rotations of M along the way. Robots are arranged into three groups, where each group is assigned a task of either pushing M towards its goal or adjusting M as it veers off from the desired path P. Each robot is computationally simple in that it merely moves towards a target point somewhere on the boundary of M. As the robots move towards these target points, they cooperatively push the object with no interaction between one another. The robots rely on only three parameters to push the object: the orientation of M, the current target point and the task they are required to perform. The target points are provided by a global control & monitoring system that monitors the progress and stability of the robots as they push M along P, providing direction to the robots in terms of tasks such as pushing, rotating, re-alignment, re-orientation or re-positioning commands. We verified our algorithm with a number of experiments that address the usefulness of the solution as well as the effects that an increase in robots number will have on the runtime and the data communication load.

Acknowledgments

First and foremost, I offer a special thanks to my heavenly Father Almighty God for giving me the strength and the patience throughout the years of my study.

I would like to offer my sincerest gratitude to my supervisor, Dr. Mark Lanthier, who coached me patiently throughout my thesis. I admire his professional guidance, effort and persistence that helped me to produce high quality work. It is an honor to work with Dr. Lanthier. During my research, I benefited from outstanding works by Lozano-Perez on path planning, Eric Bonabeau on swarm intelligence and Rodney Allen Brooks on behavioral models. Also, I would like to extent my appreciation for all the published work done by various researchers.

I would like to thank Adobe Systems for funding my study in full. In addition, I extend my appreciation to my managers for their support and understanding. Words fail me to express my appreciation to my wife Roula whose dedication, love and persistent confidence in me, has taken the load off my shoulder. She gave all the comfort and space to study and be successful. Also, I would like to thank my mother and my kids Rebecca, Charbel and Chanelle for their patience and the joy they bring into my life. I promise that I will dedicate enough time to you all in the years to come. Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of this thesis.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

C: Center of mass of M

d: Distance between adjacent vertices in same layer of graph G

G: Graph used to generate the shortest path approximation

GC: Global controller

L_i: Single layer of graph G

M: Object being pushed

O: Set of convex polygonal obstacles

O_i: Single obstacle in O

P: Path generated from graph G

p_i: Point on path P

PM: Path monitor

PP: Path planner module

R: Set of robots

R_i: Single robot in R

S_i: Single segment of P

s: Start point of P

t: End point of P

T: List of tasks

V_i: Single vertex in G

W: Workspace

Chapter 1

1 Introduction

One of the ultimate goals in robotics is to design robots that are capable of planning and accomplishing tasks independent of human intervention. For example, multiple robots can be assigned to push heavy items in a factory, transport dangerous goods, or push a boat into a port. The use of multiple robots to accomplish such tasks allows the object to be pushed rigorously. In addition, the object may be too heavy for just one or two robots to push. The problem addressed in this thesis is to design a strategy to enable multiple robots to rigorously push an object along a polygonal path in a two-dimensional environment containing polygonal obstacles (see Figure 1). To be more precise, the problem statement is as follows:

Let W be the workspace (2D Euclidean space) containing a set $O = \{O_1, O_2, \ldots, O_n\}$ of stationary convex polygonal obstacles. Let M be a convex polygon (called the *push object*) that needs to be transported from a start point s to a destination point t in W. Assume that M is to be pushed along a piecewise linear path $P = \{p_1, p_2, \ldots, p_k\}$, where $p_1 = s$ and $p_k = t$ by a set of robots $R = \{R_1, R_2, \ldots, R_r\}$. We assume that the system is noise-free in that each robot is able to perform point-to-point travelling without error. The robots attempt to translate M along P such that the center of mass of M (i.e., defined as a reference point C) remains on P at all times. During translation from s to t, rotations of M (about point C) are also allowed at any point p_i, $\forall\ 1 \leq i \leq k$. It is possible that M may deviate from P since the robots do not move

precisely in real life, however our algorithm is designed to minimize such path deviations by detecting and correcting misalignments along the way.

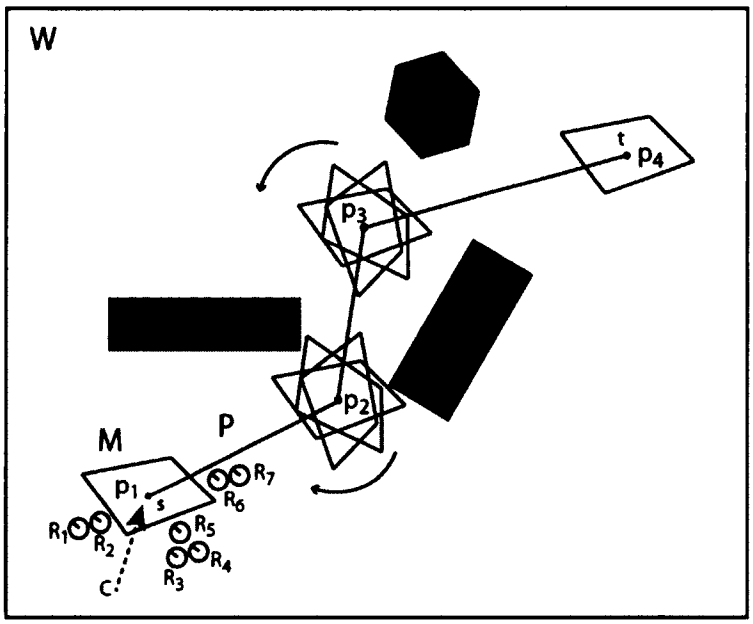


Figure 1: The workspace showing obstacle set O, push object M, robot set R and path P, showing start and target points s and t.

Traditional design approaches for multi-robot coordination systems are based on two strategies: (1) A *bottom-up strategy* typically presents a decentralized system in which robots communicate directly with each other and are capable of sensing the environment around them in order to accomplish certain tasks. (2) A *top-down strategy* presents a centralized control system in which robots retrieve global knowledge from a central controller that in turn is aware of the state of task completion. Both systems can produce the same results, but each has its advantages and disadvantages [1][2][10][14][15]. We chose to design a *hybrid system* that endorses a centralized strategy to monitor the overall goal progress, but also takes into consideration the bottom-up strategy (as found in social insects), in that our robots retrieve minimal information from the environment in order to accomplish the task.

Our overall control system is shown in Figure 2. It consists of a *global controller* (GC) that monitors the motion of M and provides the robots with the necessary information in order to push M with rigor along each segment of P. We assume that all robots remain active and do not become incapacitated during the course of the operation. Communication between the robots and the GC is minimal and limited to providing (a) target points in W for the robots to move towards and (b) the orientation of M (c) the task to be performed. It is assumed that the GC has complete global knowledge of W at all times including stationary obstacle set O, trajectory path P, position and orientation of M and source/target points s and t. The GC initiates a *Path Monitor* (PM) process which plays the role of a sensor system (e.g., an overhead camera or GPS) that keeps track of $M's$ location and orientation while being pushed along P. We assume that the path monitor is able to accurately measure the location, shape and orientation of M as well as each of the robots without error at all times. The path monitor, however, is more sophisticated in that it also issues commands to the robots as the progress of the pushing task unfolds. The movement along P is broken down into individual translations of M along each segment of the path $\overrightarrow{p_i p_{i+1}}$, $\forall\, 1 \leq i < k$ as well as potential rotations at each point p_i, $\forall\, 1 \leq i \leq k$ to re-orient M for the next segment.

The *Path Planner* (PP) is used to generate P as an efficient (e.g., shortest) path. As will be discussed later, it makes use of a graph that is constructed from all of the collision-free configurations of M. *Collision-free configurations* are a subset of what is called *configuration space* (C-Space), that was first proposed by Lozano-Perez in 1979 [20][21], where each configuration is a set of all the rotational poses of M at a specific reference point C. Collision-free configurations are denoted by C_{free}. Path planning is necessary in this work so that the robots follow a strict path while pushing M with rigor and avoiding collisions with the stationary obstacles. The *Stabilizer Tool* process in the GC is used to ensure that corrections are made along the way to try and keep C on P.

Noted as well, in Figure 2, that each robot has its own set of behaviors, which includes taxiing, avoiding, pushing, rotating, repositioning and adjusting; each of which is activated based on certain conditions. The behavior design of the robots is based on the subsumption architecture model proposed by Brooks in 1986 [14]. We followed the rule model set by Brooks by layering the behaviors such that they allow the robot to switch between them based on the position and orientation of *M* as well as the relative positions of the robots.

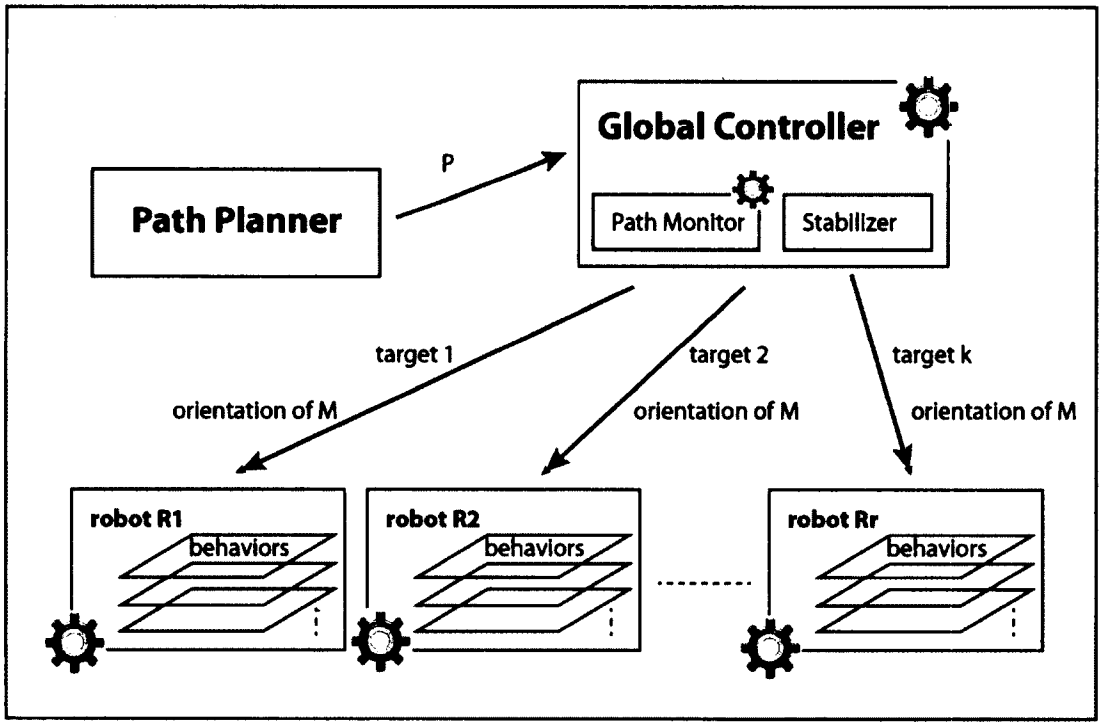


Figure 2: The overall system design.

1.1 Contributions

In this thesis, we implemented a hybrid system, which is partly a centralized system where a global controller provides minimal information to the robots. Each robot has a set of behaviors that communicate minimally with the controller hence they operate independently. We designed a path planning algorithm that provides a shortest path approximation for moving an object to a destination by allowing translations, as well as rotations, in a two-dimensional polygonal environment.

Before beginning our work, we did a thorough investigation into previous work related to the problem of multi-robot object pushing. To our knowledge, in the previous work that we encountered the robot's task was limited to pushing a rectangular box on a simple trajectory where rigorous movement was not taken into consideration [1][2][3][4][5][6][7][8][9] [11][35][44]. This means that the object could stray off path while being pushed by the robots, making it possible to deviate significantly from the desired path. Several papers studied the extent to which the object deviated from its path before it reached its destination, while other research focused on the behavior and ability to get the robots to communicate with their surroundings. The way that our work differs is identified below in our contributions. The contributions of this thesis are as follows:

- To our knowledge, this is the first research that combines the area of computational geometry motion planning algorithms with multi-robots systems.

- We implemented a shortest path approximation algorithm for the convex polygon motion planning problem that allows rotation of the push object. Our work allows arbitrary convex polygons to be pushed, not just rectangular boxes.

- In previous work, the focus was on transporting an object to a destination without giving enough attention to rigorous movement. We implemented a multi-robot behavioral system that pushes an object along a path with rigor. In practice, robots do not move along precise paths, so our algorithm also has the capability of adjusting the robot-pushing strategy so that the robots "get back on track" when the push object begins to deviate from the desired path.

- We simulated our algorithms using an integrated physics engine in order to obtain more realistic and natural interaction.

1.2 Thesis Layout

In Chapter 2 we present some related work on path planning, swarm intelligence, behavior implementation and physics engines, assessing how it relates to our work here, as well as defining terms that will be needed in subsequent chapters.

In Chapter 3 we present our algorithm that generates the path P. Our work is based on that proposed by Lozano-Perez [20][21]. In our design, we make use of the C-Space presented by Lozano in order to generate a grid of allowable poses for the push object on the workspace. Dijkstra's algorithm is then used on this grid to compute a shortest path approximation that allows rotations. Finally, we show how the path structure will change (i.e., the obstacle dimensions as well as that of M will grow) when multiple robots need to be used to solve the problem.

In Chapter 4 we present the strategy used by the robots to push and rotate M along P. We will describe how the target-balance points around M are calculated and how the individual robot behaviors change based on the location of these points. We also discuss factors that can cause the M to deviate from P or change its orientation. In addition, we discuss the behavioral system for the robots. We describe each of the behavior inputs, outputs and activation conditions. Furthermore, we describe the capabilities of the global controller, the path monitor and the stabilizer tool and show how they are used to find target points around M in order to push it with rigor.

In Chapter 5, we will present our experiments and the validity of the results which includes the deviation distance of M from the path while being pushed, the collisions that could cause M to deviate, the speed of pushing M as the number of robots increase and the data communication between the robots and the GC.

Finally, in Chapter 6 we summarize our design and we present ideas for future work and some potential solutions, such as enhancing the performance when finding the shortest path approximation, strategies that can be used when M is curved and enhancing the repositioning of the robots around M.

Chapter 2

2 Background and Previous Work

A number of research papers have been written in the areas of motion planning, cooperative transportation, multi-robot behaviors and collision avoidance. This stems from the need to automate real-life tasks by applying human-like behaviors in the science of dynamics and kinematics. The need to find solutions to these problems has encouraged computer scientists to study and experiment in these areas. During a research survey, we noticed quite a bit of work that had been done in the field of cooperative transportation and more specifically pushing an object from one location to another in a complex environment. In our experiments, we were unable to compare data with previous work in this area since the authors did not provide data that measured deviation distance from the pushing path. Our research concentrated on five areas: path planning, swarm intelligence and multi-robot systems, object pushing, the implementation of robot behaviors in centralized and decentralized environments and physics engines. Following is a brief survey of previous work in these areas.

2.1 Path Planning

The subject of path planning has been extensively researched, but the work most applicable to ours is that of Lozano-Perez et al. in 1979 [20]. They describe a collision avoidance algorithm for planning a safe path for a polygonal object moving among

known polyhedral objects that are considered forbidden regions. Later Lozano-Perez [21] extended his idea and defined what is known as *configuration space* (C-Space) where each pose represents a degree of freedom in the position and orientation of the object being moved. Only the poses that do not overlap obstacles are preserved. C-Space is normally partitioned into two mutually exclusive subsets: C_{obs} and C_{free}. Figure 3 shows a Cartesian map that represents collisions where C_{free} denotes the rotational poses of M that do not overlap obstacles (shown solid line polygons). On the other hand, the rotational poses of M that overlap obstacles are denoted as C_{obs} (shown as dashed line polygons). In the figure the reference point RF of M is translated to various locations.

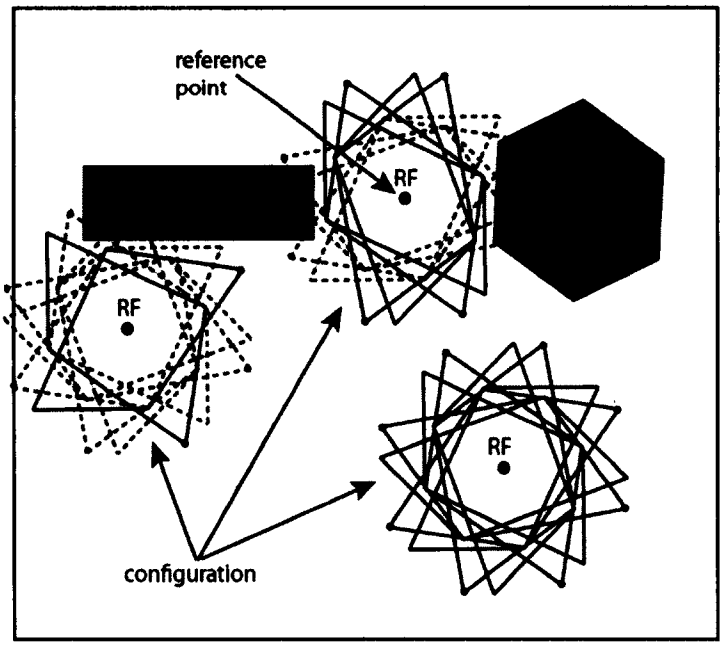


Figure 3: Cartesian map that represents collisions. The dotted line poses are part of the C_{obs} subset.

It was shown by Reif [44] that computing C-Space is PSPACE-hard, which implies NP-hard. The main problem is that the dimension of C-Space is unbounded. An excellent chapter by Steven M. LaValle [19] on C-Space explains this notion in more detail. Here is a mathematical interpretation of C-Space as formulated by Lozano-Perez:

Let O be the set of all obstacles in workspace W, where C-Space $\in W$ and n is a fixed number of points $p_1, p_2, \ldots, p_n$ in W. When M is placed at position p_i where $0 \leq i \leq n$ we can say:

$$C_{obs_i} = \{ \forall M\alpha_j \in W, (\alpha_j \cap O) \neq \emptyset \} \text{ and}$$

$$C_{free_i} = \{ \forall M\alpha_j \in W, (\alpha_j \cap O) = \emptyset \}$$

where $0 \leq j < 360^0$, α_j is a rotational pose. Thus, for all configurations in W we can say:

$$C\text{-}Space = \{ C_{obs} \cup C_{free} \}$$

The work by Lozano-Perez et al. triggered additional work by Rodney Brooks [22][23] to find the shortest path from a start position of an object to a goal point using C-Space. In 1982, Lozano-Perez and Brooks joined efforts to implement an algorithm based on C-Space for polygonal obstacles and a moving object with two degrees of freedom. Efforts in finding an accurate algorithm for C_{free} started with Schwartz and Sharir [25], who proposed to decompose C_{free} into a collection of non-overlapping cells and to represent cell connectivity using a graph. In 1988, J.F. Canny [26] proposed an algorithm that iteratively seeks a low-dimensional retraction of C_{free} by employing the techniques from differential topology. This algorithm was more efficient and less complex than that of Schwartz and Sharir. A recent paper by deBerg et al. [27] on computing push plans for an object by robots, makes use of C-Space to find the shortest path. Unlike deBerg et al. who push a disk-shaped object, in our system we use an arbitrary polygonal shape which requires more work for calculating M's configurations. We used a similar approach to that of Lozano-Perez and Brooks for translating M to equally spaced locations in W and using the poses at each location to generate a graph.

In our work, after generating a path P, we introduce robots that maneuver around M to push it. Since the robots travel with M, this essentially alters the shape of M. Hence M will "grow" larger to accommodate a buffer around it in which the robots walk around so as to be able to push from various points along M's boundary. As a consequence the "growing" of M will alter C-Space for our problem at hand. This results in a different path P being computed.

Our path planner generates an approximation of the shortest path P between two poses s and t by using an undirected graph. There are several efficient shortest path algorithms such as Dijkstra's algorithm [49] that solves the single-source shortest path problem for a graph with nonnegative edge weights, the Bellman–Ford algorithm [48] which is used primarily for graphs with negative edge weights, the Floyd–Warshall algorithm [50] which can be applied to weighted graphs with negative and non-negative edges, and others. We chose to use Dijkstra's algorithm as it is widely used.

2.2 Swarm Intelligence

Several papers have been written, and simulations implemented, to study the underlying behavior of social insects, multi-robot and single-robot systems [14][15][34]. Social insects such as ants can transport a prey to their nest with minimal interaction with each other or the environment [39]. They self-organize to accomplish such tasks where their behavior is not directed by a central controller. In our thesis, it is imperative to study this technique and try to apply some of its features while keeping in mind that our main target is to push an object rather than carrying it in order to transport it. A human can sense if an object is moving and also discern whether or not it is moving with the intended orientation and in the intended direction. A study done by Deneubourg et al.

[10] investigating the transport of a worm in the ant *Formica polyctena*, shows that the transport became suddenly successful after a number of unsuccessful attempts. Deneubourg et al. attribute the outcome to the forces applied cooperatively by the ants which finally aligned and caused the worm to be transported in the right direction [39]. Hence, the underlying mechanism of social insects uses trial and error to accomplish a task. As a result, task accomplishment is not typically done in a precise manner. However, careful accuracy may be crucial in some robotic applications where the margin of error is narrow. It may be necessary to use methods that enforce careful accuracy while accomplishing a task via multiple robots. We are proposing a hybrid system that takes advantage of precise cooperative transportation while having the flexibility of collective behavior as found in social insects. The study of cooperative transportation is divided into two main sections: centralized and decentralized control systems [3][5][15].

2.2.1 Centralized vs. Decentralized Systems

Most of the previous work related to swarm intelligence has been based on decentralized systems. In decentralized systems, the behavior of a group of agents emerges from the collective behaviors of the individual agents, which leads to what is called *self-organization*. This is very common amongst social insects. Some tasks may be too complex to be accomplished by a single agent but the collective efforts of many agents working together may allow completion of the task. E. Bonabeau et al. [15] presented a decentralized system to solve cooperative transportation by a group of robots that act like ants. The benefits of their work is that it simulates problems such as deadlocking when an equal number of robots surround a box resulting in an even distribution of the box forces. Such problem can be resolved with repositioning and realigning the robots. Li & Chen [3] used a swarm intelligence model where self-organized systems of homogeneous robots were built around simple behaviors, obtaining

a decentralized and intelligent global behavior. Wang et al. [35] proposed a decentralized system where the object is surrounded with series of robots and the position of the object is controlled by the position of each robot that encloses and pushes the object. In order to find suitable push points around the object, the robots would wander around the object to find such points using their sensors. In our design, instead of surrounding M with the robots and then allowing them to search for latching points around M, we enabled the centralized GC to quickly and efficiently pick three suitable points on the boundaries of M. One point is located on the left boundary of M (with respect to a particular push direction) and another on the right boundary. These are used for adjusting the orientation. A third point is chosen as one in which to push M in a particular direction. In the above simulations [3][15][35], a box shape was used as a push object along a trajectory with minimal concentration on rigorous motion and more focus on accomplishing the task.

Decentralized systems are considered robust and flexible against changes in the environment but do not give enough consideration to rigorous movement or speed in accomplishing a task. The focus is more on successfully finishing the task. If accuracy in the motion while pushing an object is important, then centralized systems are more likely to be used in such situation. Usually, they are used if there is no room for trial and error and it is expensive to get the robots to act as their own controller [45] due to three main factors: extensive calculations, bandwidth and proper assessment of overall task completion. For example, if a robot is required to do extensive calculations to generate the variables needed to preserve acceptable precision then it is wiser to use a powerful central controller to perform such calculations. Moreover, it is often more efficient and viable to do robot-to-controller communication since explicit communication between robots would require higher bandwidth. Robot-robot communication has another problem in that the amount of communication required can grow quadratically while robot-to-controller communication grow linearly. It is also difficult to get a proper assessment of

overall task completion and to assess the degree of error that can grow as the task is being performed.

2.2.2 Stagnation Recovery

When pushing objects, it is possible that *stagnation* can occur. This is caused by a deadlock on movement enforced by external forces or the cancellation of forces as agents in the opposite direction push at the same time. To avoid stagnation, social insects may try to change positions and realign or may seek help from other agents to overcome stagnation. The time overhead required for repositioning and realignment can be reduced by introducing the *division of labor* or what is called *task allocation*. Kube and Zhang [40] have devised a mechanism for repositioning and realignment inspired by social insects where the robots reposition randomly around the box. Their mechanism is triggered by the box's cessation of motion. If the box has not moved during a time greater than the realignment timeout threshold, the corresponding behavior randomly changes the direction of the applied force. When realignment is not sufficient to move the box, the repositioning behavior is activated. In our design we used a similar mechanism for division of labor and repositioning so as to balance the pushing. The characteristics of the stabilizer system we developed proved to be efficient in providing target points at the boundary of M based on mathematical calculations of where the robots should latch to balance-push M and avoid stagnation.

2.2.3 Homogeneity and Simplicity

Most multi-robot system studies use homogeneous systems where the structure is uniform across all involved robots [4][5][6][11][15][42]. As stated by Hales [41], comparing the performance of heterogeneous and homogeneous swarms, it was determined that heterogeneous swarms outperformed homogenous swarms if the weights of the center of mass attribute were heterogeneous in the population. In our implementation, we used a homogenous set of robots, but due to the robustness of our algorithm, it is possible to have heterogeneous robots as each robot is assigned to one of three tasks: "left adjuster", "right adjuster" or "pusher". As multiple robots with a single task attempt to reach a target point, they cluster around it and their collective behaviors force them to act as a solid mass. The total force of each cluster does not have to be equal as its duty is simple and that is to give M a little push from the right or left sides to move it back on track or to push it from behind to advance it. The presence of the path monitor and the collective behaviors of all the robots will ensure that M is moving rigorously.

It was determined by Goldberg and Mataric [47], that performance can degrade due to interference among robots. Therefore, it is apparent that the quality of the motion can be affected negatively by the quantity of the robots. A balanced number of robots needs to be determined based on the static friction, size, shape and weight of M and the power of the individual robots. This problem has been addressed by Deneubourg et al. [10] where their system adopted a task allocation mechanism that automatically determines the optimal size of a group of robots that cooperate in a foraging application. In our work, we have not addressed the problem of computing the optimal number of robots.

2.3 Behavior-Based Programming

One of the main goals of this thesis was to create robots that collect minimal information from the environment and react simply on this information. Similar behavior has been observed by social insects [39]. It is known that insects can easily navigate in the real world where they can wander, avoid objects, push objects, and move towards goals. Even though the computational power of an insect is minimal compared to a robot agent, it can be more efficient. This dilemma triggered the *Subsumption Architecture* Model proposed by Brooks (1986) [14], which is a great fit for associating behaviors with robots. In this model, the robots react to self-managed behaviors that are subsumed by other higher-level behaviors of the architecture. In our model, the behavior-based system is reactive in the sense that the robots do not know anything about M. Rather, they rely on primitive input (i.e., a target point to move towards or angle values) to decide which behavior to activate.

Work by Trojanek et al. [1] focuses on the cooperative transportation task carried out by teams of robots. Each robot uses a set of behaviors to execute a general task that consists of pushing a box by two mobile robots, that are unaware of each others' actions, along a partially specified trajectory. The value in this work is that it presents a method of division of a task into independent behaviors. However, unlike our design, this work is limited to only two robots for pushing, does not perform in-place rotations of the box and the behavior model is based on the collective information from both robots.

Yamada et al. [5] also designed mobile robots that were programmed to act using a behavior-based approach. Each robot determines its next behavior without explicit communication with other robots. In addition, a robot activates a suitable *situated behavior set* (SBS) to the current situation, and acts using the activated SBS. The behavior model in this work is interesting and useful. We based some of the behavior

model in our design on it where we get the robots to activate certain behaviors based on the position of a target point's visibility or the presence of a certain path segment.

Our design is also similar to that of Lewis et al. [6] where the concept of a *virtual structure* is introduced. Using the virtual structure approach, a general control strategy was developed to force an ensemble of robots to behave as if they were particles embedded in a rigid structure. The target points provided by our stabilizer system are in fact a virtual structure with two points for adjusting and one for pushing M.

One of the most studied implementations is that of Tang et al. [7], where pusher robots are tightly coupled and a virtual robot is instructed to help the robots to keep a static distance and orientation when pushing. The helper robot in this sense is similar to the behavior of our global controller where, through its stabilizer system and path monitor, it is capable of controlling the positions of the target points that the robots need to reach around M and monitor the orientation of M.

2.4 Physics Engines

We integrated a physics engine in our implementation in order to have a simulation that more closely represents reality. A *Physics Engine* has the capability of manipulating the motion of objects programmatically and providing acceptable accurate calculations that can help simulate and approximate important physics notions such as linear and angular velocity, force, and integration methods. Moreover, it has the ability to simulate gravity and different elasticity types of collisions. Various physics engines have two important distinguishing characteristics: *collision detection* and *integration methods*.

To simulate collisions programmatically, it is necessary to detect collisions against the edges of the involved objects and reverse the velocity in the offending axis. In addition, when objects collide they exchange forces for a short time interval and they undergo a change in velocity. The collisions may be *elastic* or *inelastic* [31][32]. Normally, most of the collisions are inelastic where total kinetic energy of all the objects involved in the collision does not remain constant. Momentum is conserved in inelastic collisions where, when two objects collide they stick together after the collision. In computer programs, this behavior is controlled by what is called *damping* and *restitution.* Adding damping to the collision is similar to increasing the friction force between the colliding objects and decreasing the restitution coefficient causing loss of *kinetic energy* [32] which leads to less bouncy collisions. For example, consider a box and a sphere. When the sphere collides with the box if the damping and restitution are zero then the box will slide as if it is floating (especially if the sphere is heavier than the box). If the damping is increased then the box will slide for a shorter distance. This factor will have an effect on the *linear velocities* (rate of change of linear speed) and *angular velocities* (rate of change of rotational speed). In our implementation, the values of the damping, restitution, linear velocity and angular velocity are controlled through a physics engine so that we are able to examine the effect when each parameter is changed.

A moving object takes a certain number of time-steps in order to reach a certain location. This kind of motion can be approximated with different types of integration methods such as Euler [30], Verlet [31], Runge–Kutta [32] and many others. All these methods are based on a mathematical notion called *ordinary differential equation* (ODE) [30], also known as *numerical integration*, which can approximate solving differential equations rather than solving them analytically. Accuracy of the integration method is important when choosing a physics engine. If the physics engine is using weak integration methods [32] such as Euler then it could lead to undesirable behaviors and unrealistic motion when the robots attempt to push M.

Several 2D physics engine were evaluated during our research survey such as Box2d [28], myphysicslab [37], Phys2d, and chipmunk [38]. We decided to use a popular 2D physics engine called JBox2d [29] (based on Box2d but is implemented in Java). Any 2D physics engine is suitable for our implementation as long as it has "good" collision detection mechanism and uses a good integration method.

2.5 Summary

Our work combines computational geometry with multi-robot systems by integrating work from four main areas: path planning based on C-Space, swarm-based division of labor, behavioral model and the integration of a physics engine. We took advantage of the C-Space notion by Lozano-Perez et al. [21], where we used the idea of collecting non-colliding rotational poses of M to generate our graph. Dijkstra's algorithm is then used to find a shortest path approximation from the generated undirected graph. Moreover, we applied some swarm intelligence characteristics used by Deneubourg et al. [10] and Bonabeau et al. [15] where robots are unaware of the presence of each other and act like social insects. In addition, we applied the division of labor mechanism to the robots in a similar manner to that of Kube and Zhang [40] for realignment and repositioning. Our design enables the robots to self-organize through a set of behaviors by applying the behavioral model by Brooks [14]. Finally, to make the simulation more realistic, we integrated a 2D physics engine that allowed us to dynamically change the physics properties of the workspace's components.

Chapter 3

3 Planning the Object's Path

Our path planning approach is based on the C-Space algorithm proposed by Lozano-Perez [21] which includes the C_{free} set of non-overlapping configurations of M with the obstacles in W. The C_{free} construct is used to produce a weighted, undirected graph G for finding a shortest path approximation P. The generated path P is rectilinear with segments $S_1, S_2, \ldots, S_k$ each with equal length. Each segment's endpoint of P represents a potential set A of sequential rotation angles of M such that $A = \{\alpha_r, \alpha_{r+1}, \ldots, \alpha_s, \alpha_{s+1}, \ldots, \alpha_t\}$ where $\alpha_{r+1} - \alpha_r = a$, for a fixed value of a, $\alpha_t \leq 360^o$ and $0 \leq r \leq s \leq t \leq 360/a$.

The path planning algorithm presented in this chapter is comprised of two main sub-sections: graph construction and path construction. First, two-dimensional *grid graphs* are constructed where each graph denotes a layer (Figure 6). The graphs are interconnected with edges to create the final graph G. In each layer, the vertices represent a fixed orientation (i.e., rotation) of M based on its center C. Once G is constructed and its edges are assigned weights, Dijkstra's algorithm is used to generate a shortest path approximation for M's trajectory.

Figure 4 shows a graph with only three layers. M starts at the bottom layer and then traverses G until it reaches the destination point. As M moves along P, it traverses edges in the graph G. Travel along edges within the same layer of G represents a translation of

M while travelling along edges of G that interconnect two layers represents a rotation of M about its reference point C. Figure 5 shows how M moves and rotates along P as the graph G is traversed.

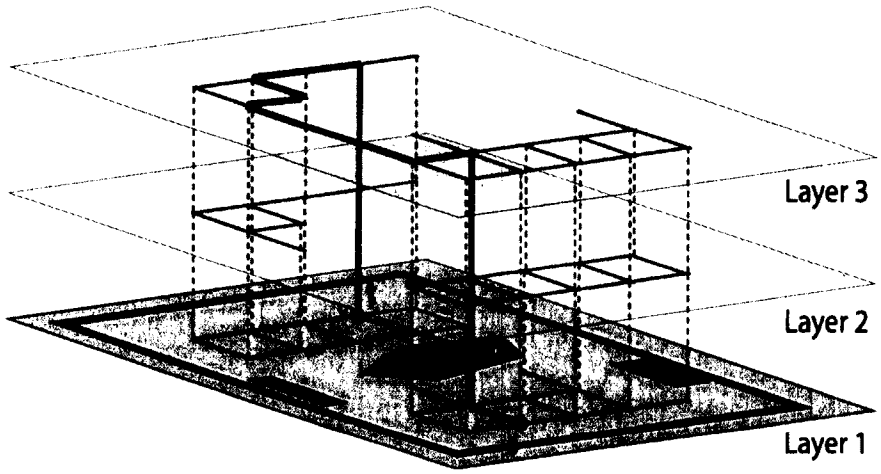


Figure 4: Graph with only three layers.

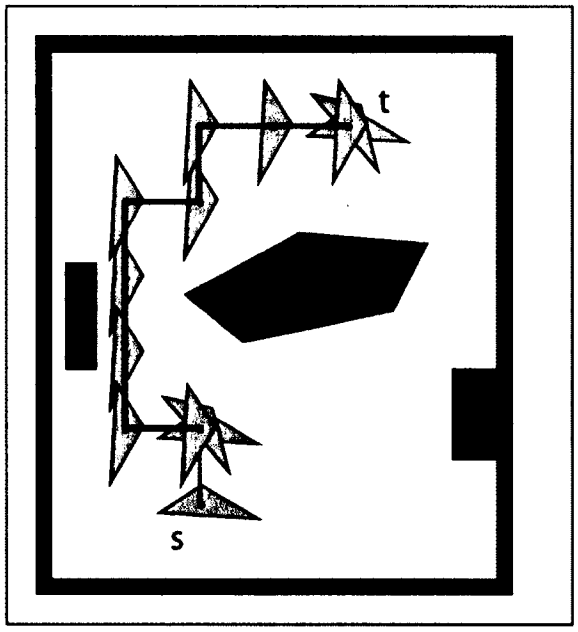
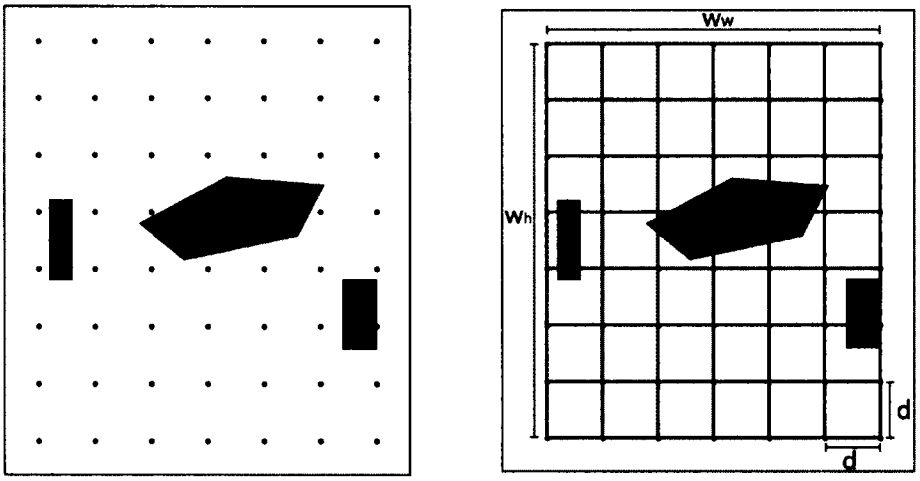


Figure 5: A fully generated P.

3.1 Algorithm Details

Let L_i be a *grid graph* with $m \times n$ vertices $V_L$and edges E_L such that $\{\forall\, u, v \in V_L, e \in E_L \mid u \neq v, |e| = \mathrm{d}\}$ where $1 \leq i \leq 360$, d is the length of every edge $e \in E_L$, $u = (x_u, y_u)$ and $v = (x_v, y_v)$ are adjacent and connected with an edge e *iff* $|uv| = d$ and $(x_u = x_v \; and \; y_u \neq y_v)$ or $(x_u \neq x_v \; and \; y_u = y_v)$. Let W_w and W_h be respectively the width and height of W where $W_w = W_h = nd - 1$.

Figure 6a shows an *upright square lattice* of $m \times n$ vertices in W where each point represents the center point C of M and is positioned at equal distance d from vertically and horizontally adjacent vertices. The top-left vertex is located at the coordinate (0,0) relative to W. The initial step in constructing graph L_i is shown in Figure 6b, where the vertices are connected with edges. In the figure, note the length d of each edge in L_i and the width and height of W.



(a) (b)

Figure 6: (a) Set of $m \times n$ vertices in W. (b) The grid graph.

Subsequently, consider translating M to each vertex in L_i. It is possible that some of the translated copies of M might overlap obstacles in W, as shown in Figure 7a. Such

vertices are removed along with their incident edges to get the final form of the graph L_1 which would denote the first layer in our algorithm (see Figure 7b). The subsequent layers from L_2 to L_l are constructed in a similar manner to L_1 but by rotating M for each layer by a certain angle denoted by α_l where $1 \leq l \leq 360$ (see Figure 7c-d). Moving up in the layers, the rotation of M increases such that $\alpha_1 = 0$ at L_1, $\alpha_2 = a$ at L_2, $\alpha_3 = 2a$ at $L_3, \ldots, \alpha_l = (l-1)a$ at L_l.

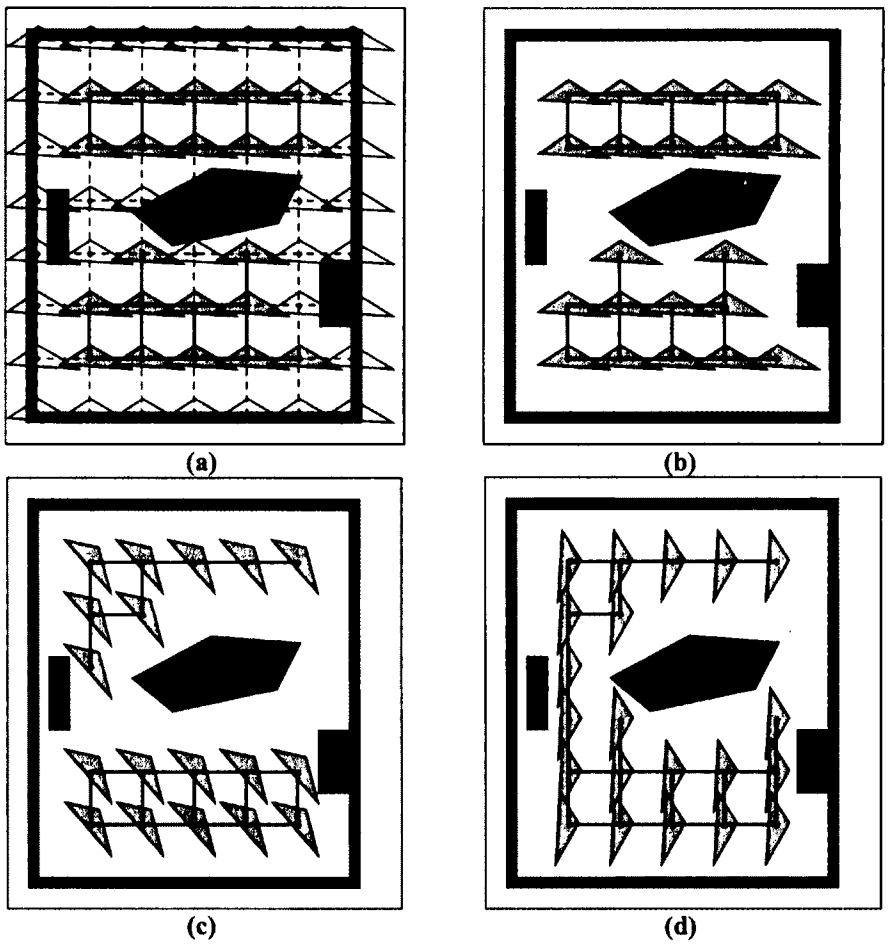


Figure 7: (a) Overlapped copies of M with the obstacles. (a) Removing the vertices along with their incident edges to get 1st layer. (c) Applying a rotation to M to construct the 2nd layer. (d) Applying another rotation to M to construct the 3rd layer.

Now that the layers are constructed, the next step is to connect them with edges. Let I be the set of *layer-interconnecting edges* between $L_{k-1}L_k$ where $1 < k \leq 360/a$ and v_{ij} on L_k corresponds to v_{ij} on L_{k-1}. Therefore, the vertices that can make an edge between two adjacent layers have the same x and y coordinates. The graph $G(V, E)$ is an undirected graph with $m \times n \times l$ vertices such that $G = \{L_1, L_2, \ldots, L_l\} \cup I$ (see Figure 8).

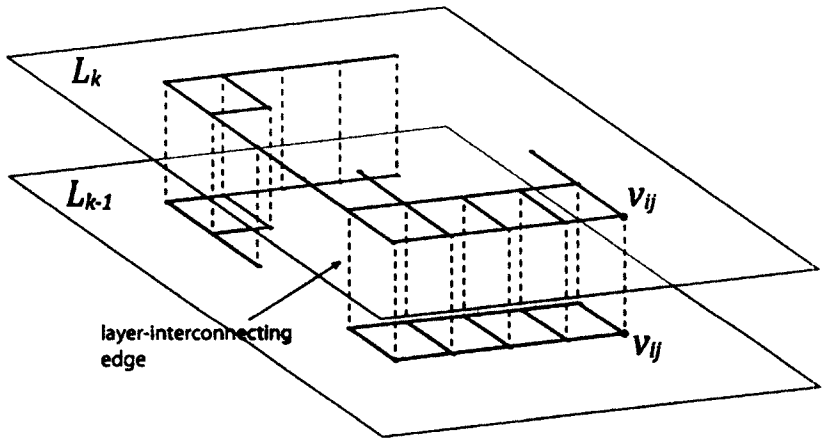


Figure 8: Layer-interconnecting edges.

Each of the edges in G is assigned a weight. Edges that are in the same layer are assigned a weight of 1 and layer-interconnecting edges are assigned a weight of 10. The reasoning behind assigning a weight of 10 for the layer-interconnecting edges is that the cost for translating and rotating M is greater than merely translating M in the same layer. The cost for layer-interconnecting edges can be any number greater than 1 but we chose a higher number to keep M in the same layer as much as possible. In Figure 9, the cost of traveling around the left side of the obstacle is much less than passing between the two obstacles where each rotation has a cost of 10 at vertices u and v. Each turn at u and v costs an extra 20 each (i.e., 2 turns each). Therefore, the cost for using the solid path is 49. The distance between the vertices u and v is the same but to avoid the overhead of rotating M we set a high cost for rotating. The result is an undirected, weighted graph G that will be used for constructing the shortest path approximation P.

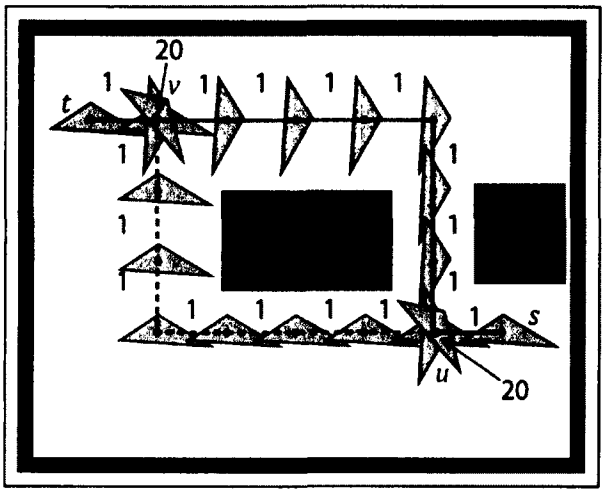
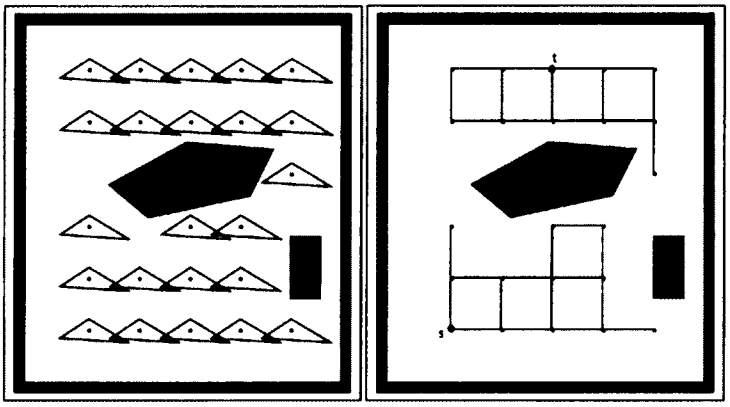


Figure 9: The edges weights. Cost of the dotted line path is 9. Cost of the solid line path is 49 .

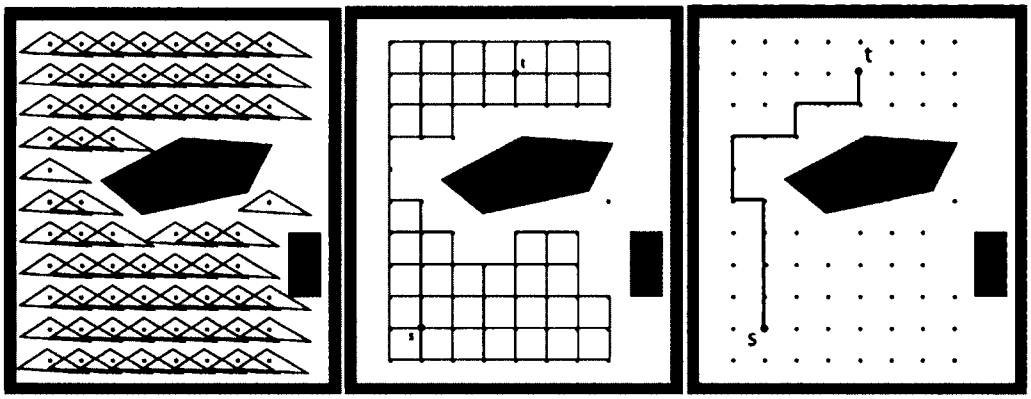
The start and end points of M (i.e, s and t) may be arbitrary points in W. Path P consists of a segment from s to $P_i \in P$, followed by a path in G from P_i to P_j followed by a segment P_j to t, where P_i (P_j) is the vertex in G that is closest to s (t) in the layer L_s (L_t) represented by the initial (final) rotation of M.

The length of d could affect the possibility of finding a path P in G. Figure 10b shows that L_1 is disconnected where s lies in one component and t in another. Therefore, it is not possible to create a continuous path P from s to t in this single layer. This will force the algorithm to seek vertices in the adjacent layers of G. However, by decreasing d, the solution can be enhanced and a path P may be found from s to t as shown in Figure 11.



(a) **(b)**

Figure 10: (a) All possible placements of M at some fixed orientation.
(b) Creating the edges. The result is no valid P.



(a) **(b)** **(c)**

Figure 11: Decrease d to find a solution.

In some cases, even if d is decreased, it is not possible to find a path P in a single layer as shown in Figure 12a, where a new obstacle is added to W (left-middle). The algorithm seeks vertices in the adjacent layers in order to find a solution to this problem (Figure 12b-c).

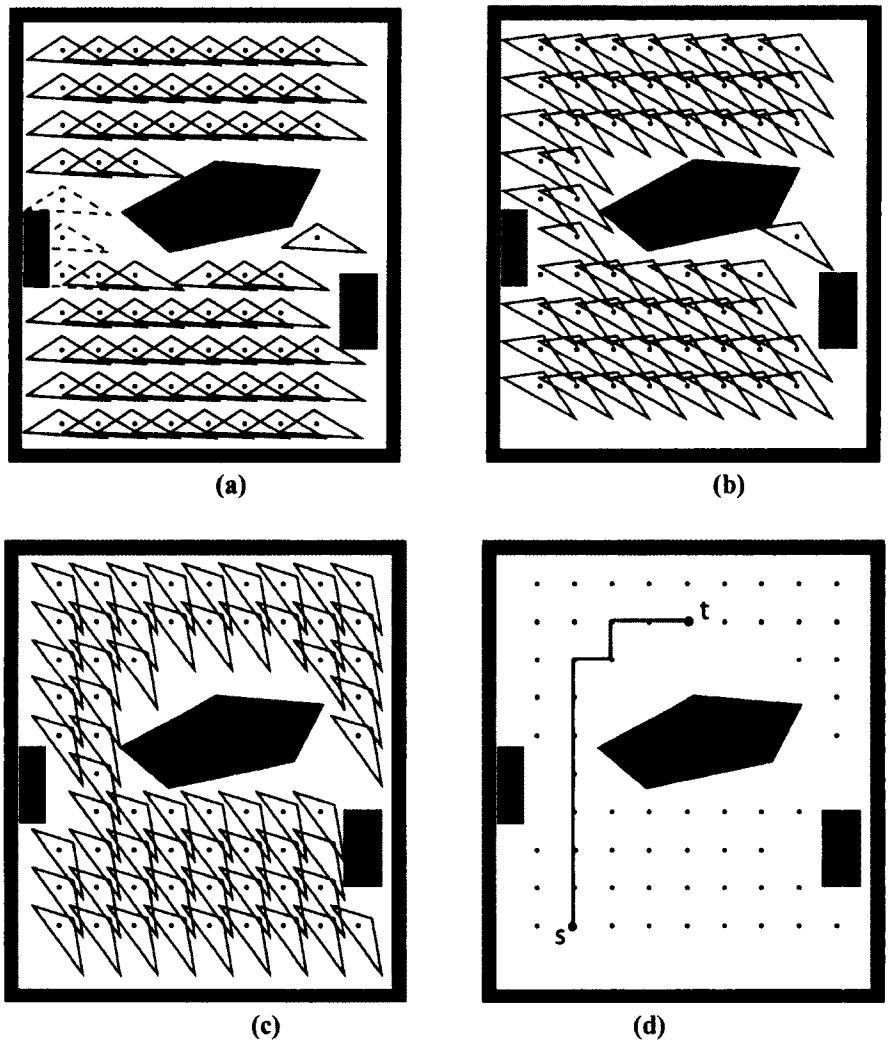


Figure 12: After adding another obstacle to W, a rotation is required to find a path from s to t.

In the generated path P, rotating M clockwise represents an upward move in the layers of G while rotating M counter-clockwise represents a downward move in the layers of G. Figure 13a shows that if the last translated M was to move ahead in the same layer then it would intersect with the obstacle. The clockwise rotation indicates the need to move upward to the adjacent layer. Figure 13b shows that there is still a need to rotate. Figure 13c shows that after two rotations, M moves ahead in the same layer until there is a need to rotate counter clockwise (i.e., goes back down the layers). Figure 13d shows

that there is still a need to rotate counter clockwise and move downwards toward the next layers, in order to match the destination orientation.

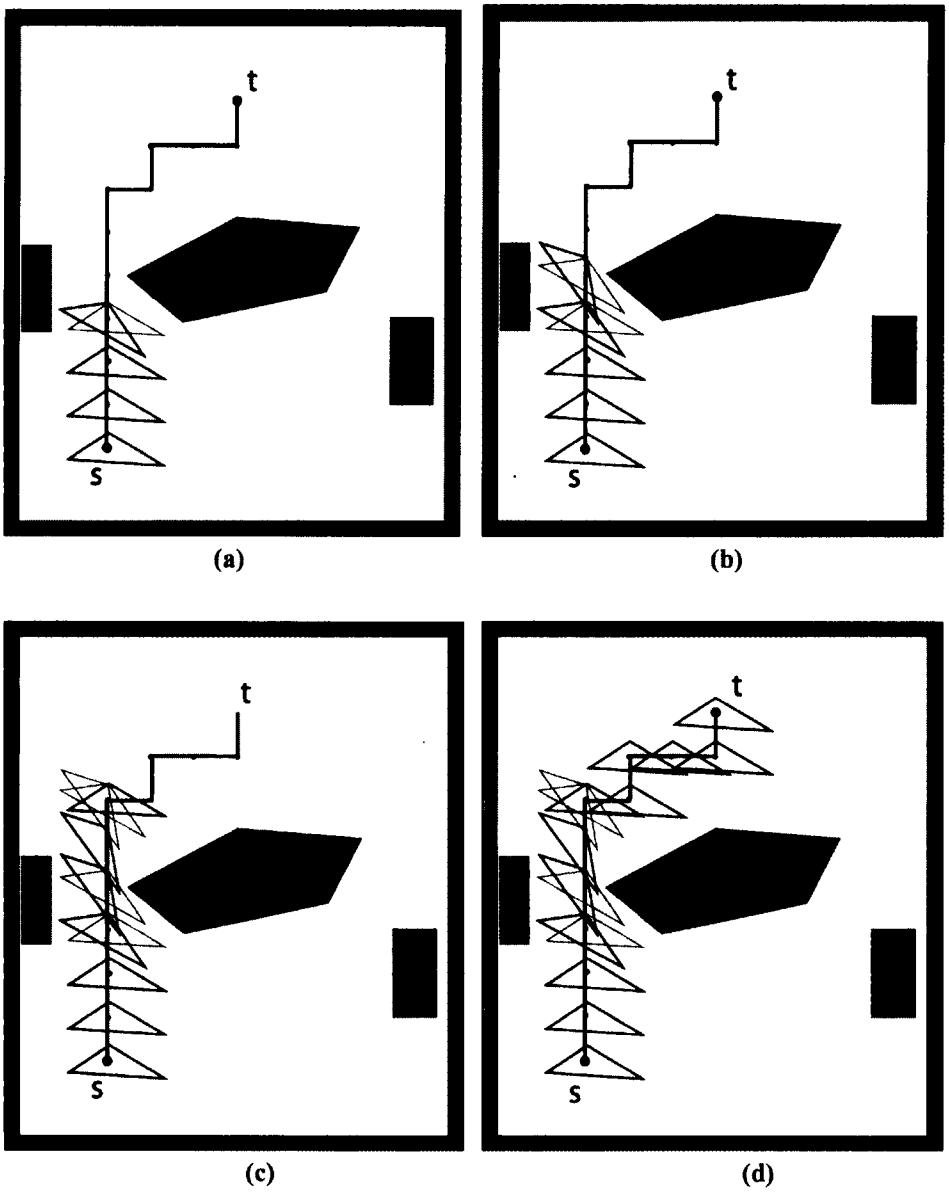
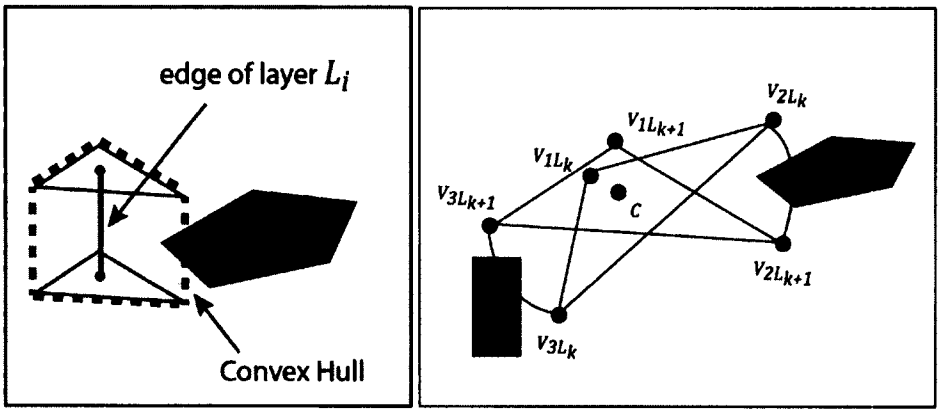


Figure 13: (a, b) Clockwise rotation. (b, d) Counter-clockwise.

3.2 Limits and Special Cases

In our algorithm, if d is too large and an obstacle is small enough then the generated path might ignore an obstacle and pass over it, as shown in Figure 14a. Even though we did not address this issue in our implementation, it can be solved by taking the convex hull of every two adjacent M's in the layer L_i (i.e., corresponding to edge e in that layer) and performing an intersection test of that hull with all obstacles in W. If an intersection occurs then the edge e must be removed from layer L_i.

A similar problem to the one above can occur when rotating M, as shown in Figure 14b. This problem can be addressed as follows: Let $V_{1L_k}, \ldots, V_{nL_k}$ be vertices of M in layer L_k where n is the number of vertices of M. Let $V_{1L_{k+1}}, \ldots, V_{nL_{k+1}}$ be vertices of M in layer L_{k+1} after rotating it by a certain angle about its center C, where V_{iL_k} corresponds to $V_{iL_{k+1}}$ and $1 \leq i \leq n$. Projecting V_{iL_k} and $V_{iL_{k+1}}$ onto L_1 (i.e., the same plane as the obstacles), if an obstacle in W intersects the arc between V_{iL_k} and $V_{iL_{k+1}}$ then the layer-interconnecting edge e between L_k and L_{k+1} must be removed from G. In Figure 14b, we can see that two obstacles intersect the arcs between corresponding vertices in M in layers L_k and L_{k+1}.



(a) (b)

Figure 14: (a).Translation special case. (b) Rotation special case.

3.3 Implementation Details

In Algorithm 3.1, the graph construction is performed by translating M to each vertex v in L_i and rotating M at v by α_l, where $1 \leq l \leq 360$, such that $\alpha_1 = 0$ at L_1, $\alpha_2 = a$ at L_2, $\alpha_3 = 2a$ at L_3, ... , $\alpha_l = (l-1)a$ at L_l. Only the poses that do not overlap obstacles in W and edges of W are added to the *objectMap* along with their "ids" to the *adjacency matrix*. This verified using `isValid()` in line 15. The algorithm has three nested loops where the first loop (line 10) is terminated once the rotation angle of M reaches 360. The second loop (line 12) terminates once the translated M reaches the bottom edge of W. The third loop (line 13) terminates once the translated M reaches the right edge of W. In our implementation, we pass the objectMap and the adjacency matrix to Dijkstra's algorithm where each entry in the matrix is either the *"id"* of the translated copy of M to a vertex in G or a zero value for every translated copy of M that overlaps an obstacle in W. The objectMap and the adjacency matrix are updated in line 16-17 of Algorithm 3.1. Table 1 demonstrates an adjacency matrix with only three layers shown. Note that in layer 3 and column 4 the ids are in sequence which denotes a possible continuous path portion of the path P. In the next section we present Dijkstra's algorithm, which takes as input the graph G, the adjacency matrix A, the start point s and the destination point t.

Layer 1							Layer 2							Layer 3						
0	1	2	3	4	0	0	0	19	20	21	22	23	0	0	34	35	**36**	37	38	39
0	0	5	6	0	0	0	0	0	24	25	26	0	0	0	0	40	**41**	42	0	0
0	0	7	0	0	0	0	0		0	27	0	0	0	0	0	43	**44**	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**45**	0	0	0
0	0	0	8	0	0	0	0	0	0	28	0	0	0	0	0	0	**46**	0	0	0
9	10	11	12	13	0	0	0	29	30	31	32	33	0	0	47	0	**48**	49	50	51
14	15	16	17	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1: Adjacency matrix showing only three layers.

Algorithm 3.1

CONSTRUCT-GRAPH(pushObject, shift, angle)

```
1  shiftVert ← 0
2  shiftHoriz ← 0
3  a ← 0
4  id ← 1
5  cols ← workspaceWidth/shift
6  rows ← workspaceHeight/shift
7  depth ← 360/angle
8  objectMap ← ∅
9  adjacencyMatrix ← ∅
10 FOR(k←0; k<depth; k++)
11    resultObject ← RotatePolygon(pushObject, a)
12    FOR(i←0; i<rows; i++)
13       FOR(j←0; j<rows; j++)
14               resultObject ← Translate(resultObject, shiftHoriz, shiftVert)
15               IF isValid(resultObject) THEN
16                  objectMap ← objectMap ∪ {resultObject}
17                  adjacencyMatrix ← adjacencyMatrix ∪ {id}
18                  id ← id + 1
19               ELSE adjacencyMatrix ← adjacencyMatrix ∪ {0}
20               shiftHoriz ← shiftHoriz + shift
21       ENDFOR
22       shiftVert ← shiftVert + shift
23       shiftHoriz ← 0
24    ENDFOR
25    a ← a+angle
26 ENDFOR
```

3.3.1 Dijkstra's Algorithm

Dijkstra's algorithm (see Algorithm 3.2) solves the single-source shortest-path problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. We assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weight from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the algorithm, a min-priority queue Q of vertices, keyed by their distance d values is used. Each time through the while loop of lines 4-8, a vertex u is extracted from $Q = V - S$ and added to set S, thereby maintaining the invariant. The first time through this loop, $u = s$. Vertex u, therefore, has the smallest distance estimate of any vertex in $V - S$.

Algorithm 3.2

DIJKSTRA(G, A, s, t)

```
1  InitializeSingleSource(G, s)
2  S ← ∅
3  Q ← V[G]
4  WHILE Q ≠ ∅ DO
5        u ← ExtractMin(Q)
6        S ← S ∪{u}
7        FOR each vertex v ∈ A[u]
8            Relax(A, u, v, t)
9        ENDFOR
10 ENDWHILE
```

3.4 Extending The Algorithm to Accommodate Robot Bodies

We have shown how P is generated by the path planner. However, before the GC makes the request to get P, it is required to assign multiple disk-shaped robots to push M. In order to maintain a precise balanced push by the robots, we expand the size of M denoted by M' such that the robots have enough space to maneuver around it. We expand M by 6 times the diameter of the robot to leave enough space (i.e., defined as *buffer zone*) so that the robots will stay away from M as much as possible while pushing or repositioning. The buffer zone can be divided in three areas: 1) Area Z_1 is closer to the outer edges of the buffer zone. This area is a resting zone for the robots. 2) Area Z_2 is a pathway for robots in a repositioning task. 3) Area Z_3 is a zone for robots in a repositioning task but avoiding collision with other robots. We will discuss this in more detail in next chapter. We use the *Minkowski sum* algorithm [52] to expand M, which is defined as follows: Given two sets $A,B \in \mathbb{R}^d$, their Minkowski sum, denoted by $A \oplus B$, is the set $\{ a + b \mid a \in A, b \in B \}$. To simplify the calculation, we convert the disk-shaped robot to a square (denoted by A) where its radius is scaled four times (Figure 15). Then, we translate A based on its center point to every vertex of B, where $B = M$. One way to get $A \oplus B$ is to get the *convex hull* [52] of all the translated A. The Minkowski sum algorithm is described in Algorithm 3.3. M' is used to compute P.

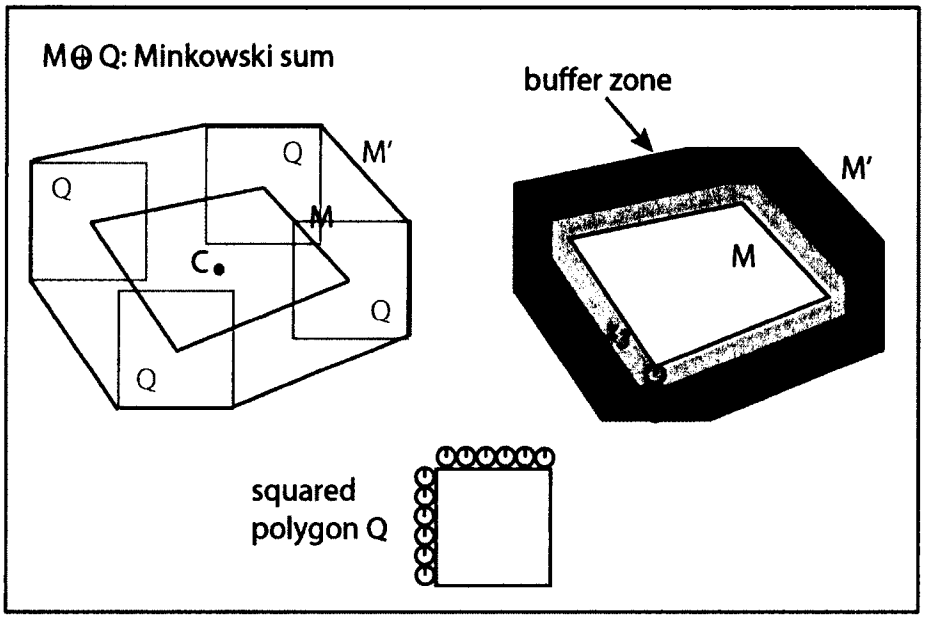


Figure 15: Creating the buffer zone for M. Z_1, Z_2, Z_3 are areas in the buffer zone.

Algorithm 3.3

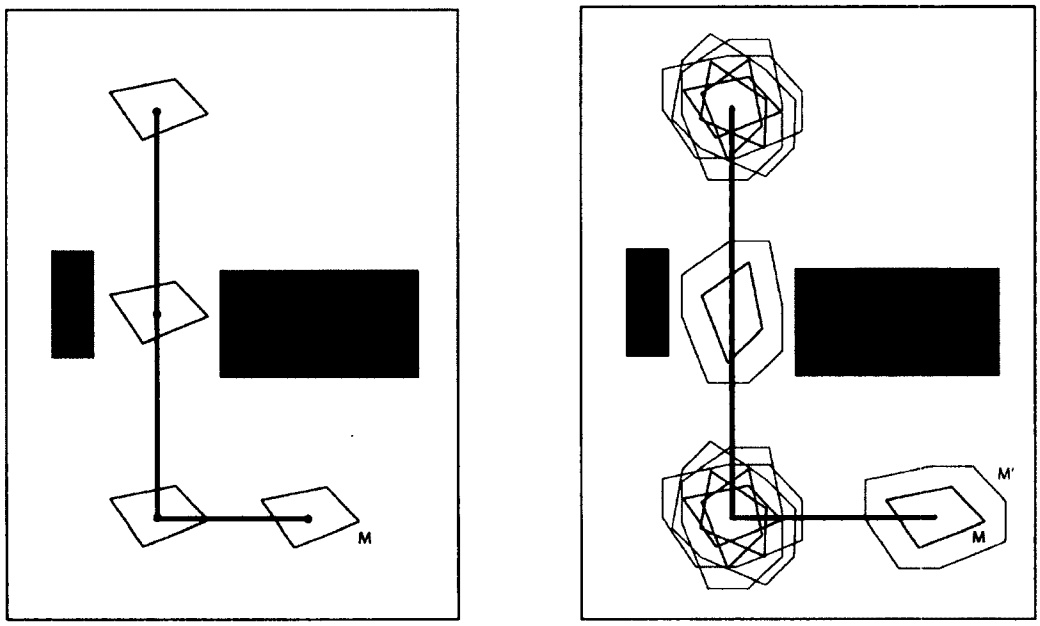
MINKOWSKISUM(A,B)

```
1   i←1; j←1
2   v_{n+1} ← v_1; v_{n+2} ← v_2; w_{m+1} ← w_1; w_{m+2} ← w_2
3   WHILE i=n+1 and j=m+1 DO
4         Add v_i + w_j as a vertex to A ⊕ B
5            IF angle(v_i v_{i+1}) < angle(w_j w_{j+1}) THEN
6               i←(i+1)
7            ELSE IF angle(v_i v_{i+1}) > angle(w_j w_{j+1}) THEN
8                     j←(j+1)
9                ELSE i←(i+1); j←(j+1)
10  ENDWHILE
```

The input to Algorithm 3.3 is a convex polygon A with vertices $v_1, \ldots, v_n$, and a convex polygon B with vertices $w_1, \ldots, w_m$. The lists of vertices are assumed to be in counter-clockwise order, with v_1 and w_1 being the vertices with the smallest y-coordinate

(and smallest x-coordinate in case of ties). The output is Minkowski sum $A \oplus B$. The algorithm loops through the vertices of A and B and adds their sum to the set $A \oplus B$. In the algorithm, the notation angle(vw) denotes the angle that the vector $\overrightarrow{pq}$ makes with the positive x-axis.

Since M' is larger than M then the new path P could change. In Figure 16, we show an example where after expanding M, the new shape M' needs to rotate in order to pass between the obstacles. M' is only used to generate the path P in the graph in order to determine when rotations are necessary due to the extra space needed for the robots to maneuver around M. Once P has been generated, M' is no longer needed.



(a) (b)

Figure 16: (a) Original size of M. (b) Growing M to M' where a rotation is required

Chapter 4

4 Multi-Robot Object Pushing

In this chapter, we will present our approach to solving the problem of pushing an object with rigor in a polygonal environment. The overall algorithm is described in Algorithm 4.1. In the first step, the GC uses the path planner presented in chapter 3 to find a path P from s to t. In addition, the GC assigns multiple robots to push M (see step 5 of Algorithm 4.1). Each robot is able to push M by seeking a target point positioned at the boundary of M. As the robot reaches the target point, it collides with M causing it to move. The force applied to M by the robots while pushing could cause it to rotate or stray away from the original path. To avoid these issues, our algorithm (see step 3 and 4 of Algorithm 4.1) creates a *stabilizer tool* and a *path monitor* (see section 4.2) which instruct the robots when an action is needed to adjust the motion of M, thereby minimizing the margin of error. In our approach, only pushing behavior, rather than pulling behavior, is applied to M. Although it is much easier to push M without respecting its orientation, the challenge resides in both pushing and keeping a precise orientation of M along P.

The path provided by the path planner is broken down into a set of tasks denoted by a *task path list* , T, that the robots are required to perform in order to push M along the path with rigor (see step 2 of Algorithm 4.1). A task has the following properties: task type, orientation of M and path segment. Three groups of tasks are used: "push", "rotate" and "reposition" (see Figure 17). Define a *sub-task* to be a temporary task that has more

priority than a task. When pushing M, three types of robots (i.e., denoted by *robots specialties*) are assigned to the task: "pushers", "left adjusters" and "right adjusters" (see step 6 of Algorithm 4.1). Moreover, while pushing, the orientation and alignment of M could deviate from their true value. Therefore, two sub-tasks are used: orientation sub-task and alignment sub-task. In addition, when rotating M, left and right adjusters are used to perform the task where one group of robots push from the left and another from the right side of M. The robots from both sides are positioned at the boundary of M based on the angle of the edge they need to push against to effectively perform an in-place rotation (see section 4.2). While rotating, the robots unintentionally translate M from its center of rotation. Therefore, a push sub-task is used to translate M back to its original center of rotation. The reposition task is performed when the robots need to change positions in order to push in a different direction. The path monitor observes the progress of each individual task and upon task completion, it informs the robots of their next task.

Algorithm 4.1

GlobalController(s, t, M)

```
1  P ← GetPathFromPathPlanner(s, t)
2  Construct the list T of tasks from the path P
3  Initialize the PathMonitor
4  Initialize the Stabilizer
5  R ← InitializeRobots()
6  Assign specialties to the list R of robots
```

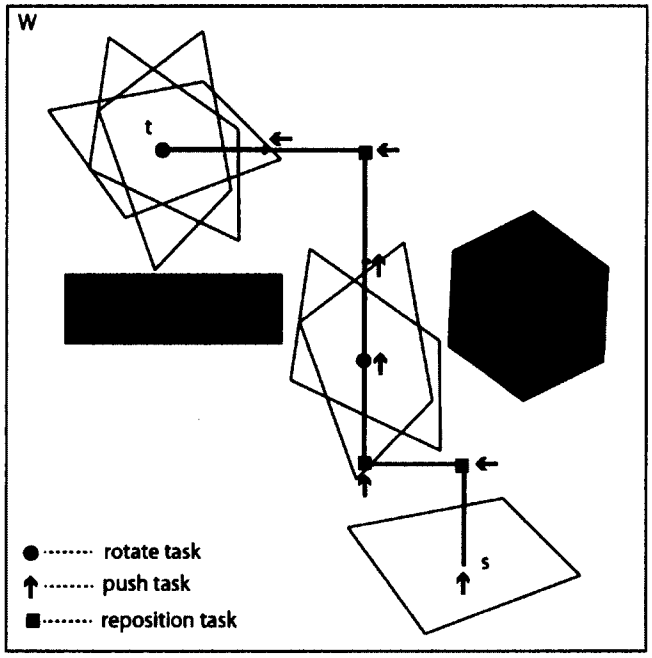


Figure 17: The path showing the various push, rotate and reposition tasks.

4.1 Preliminaries

The stabilizer tool is a data structure defined as follows: Let S be the smallest enclosing circle of M centered at C. Define $L1$ as a ray from C to the destination point N. Define $L2$ as a ray that extends $L1$ from C. Define $L3$ as a ray perpendicular to $L1$ which starts from C and extends to the left. Define $L4$ as a ray perpendicular to $L1$ which starts from C and extends to the right. Note that the length of each of the rays extends beyond S to make sure that the ray intersects one of the edges of M (see Figure 18).

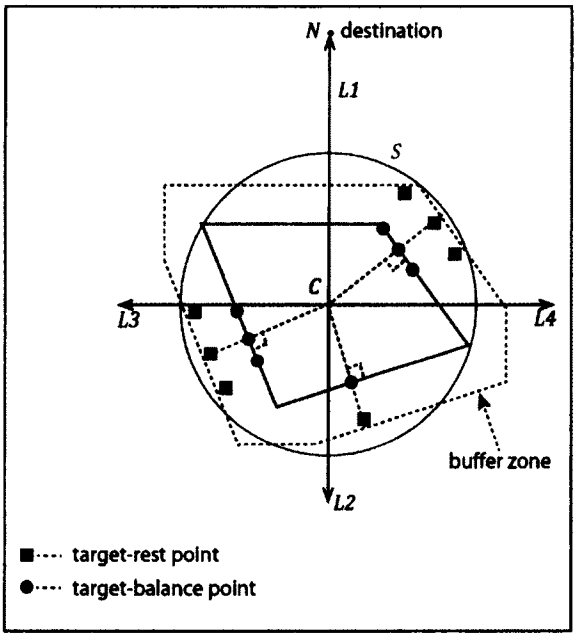


Figure 18: The stabilizer showing the target points and the buffer zone .

Consider a robot moving towards a *target point* provided by the stabilizer tool. A target point could be a point p_i of path P where $1 \leq i \leq k$, a point on the boundary of M denoted as a *target-balance point* used for pushing or rotating M or a point close to a target-balance point denoted as a *target-rest point.* A target-rest point keeps the robot away from M but at a close distance (i.e., $2.5d$, where d is the diameter of the robot) as a "standby" position (see Figure 18), as well as away from other robots that are in a repositioning task to avoid prolonged stagnation situations. The target-rest point falls in the z_3 zone of the buffer zone described in section 3.4). The computation of these target points depends on the type of task to be performed.

Consider a push task, the target point is defined as follows: Let e_2 be the edge of M that intersects $L2$. Let p_u be the push target-balance point on e_2 defined by the perpendicular line to e_2 passing through C. Let e_3 be the edge of M that intersects $L3$. Let l_a be the left-adjust target-balance point on e_3 defined by the perpendicular line to e_3

passing through C. Let e_4 be the edge of M that intersects $L4$. Let r_a be the right-adjust target-balance point on e_4 defined by the perpendicular line to e_4 passing through C (see Figure 20a). Define the associated target-rest points as, pusher resting point p_r, right-adjuster resting point r_{ar}, and the left-adjuster resting point l_{ar} (see Figure 20a). Target-rest points are created by extending the line from C to each of the target-balance points by 2.5 times the diameter of a robot. This extension distance allows the robots to stay away from other robots during a repositioning task (see Figure 22). Note that, if a target point lies on vertex v_{i+1} of M or at a distance from v_{i+1} equals to the diameter of the robot, where v_{i+1} is the end vertex of edge e_i, then the edge e_{i+1} is considered as the intersecting edge of M (see Figure 19). The reason is that when the robots maneuver around M, the constant interference with M could cause the target-balance points to quickly switch back and forth between the edges e_i and e_{i+1}.

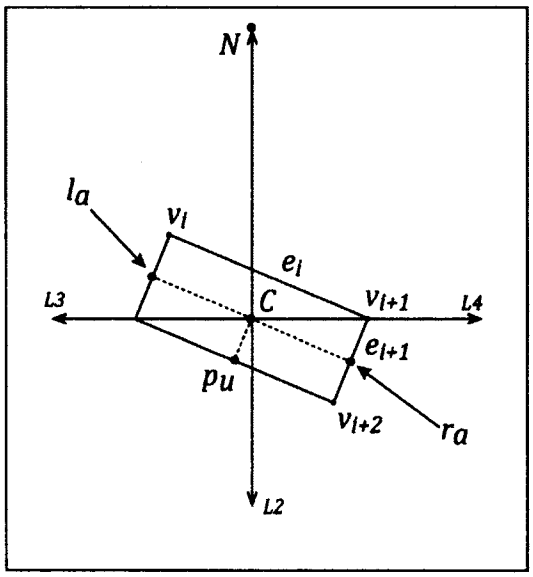


Figure 19: Special case the target-balance point falls on point of two intersecting edges of M.

The orientation sub-task could become active during a pushing task and has two additional target-balance points on each side of M which are calculated as follows: The edge e_3 that is intersected by the directed line from l_a to C is considered for the orientation target-balance points. The vertex of e_3 to the left of the line from C to l_a is denoted by l_o and the vertex of e_3 to the right is denoted by r_o. The target point l_o can be used for rotating M clockwise and r_o for rotating M counter- clockwise (see Figure 20b).

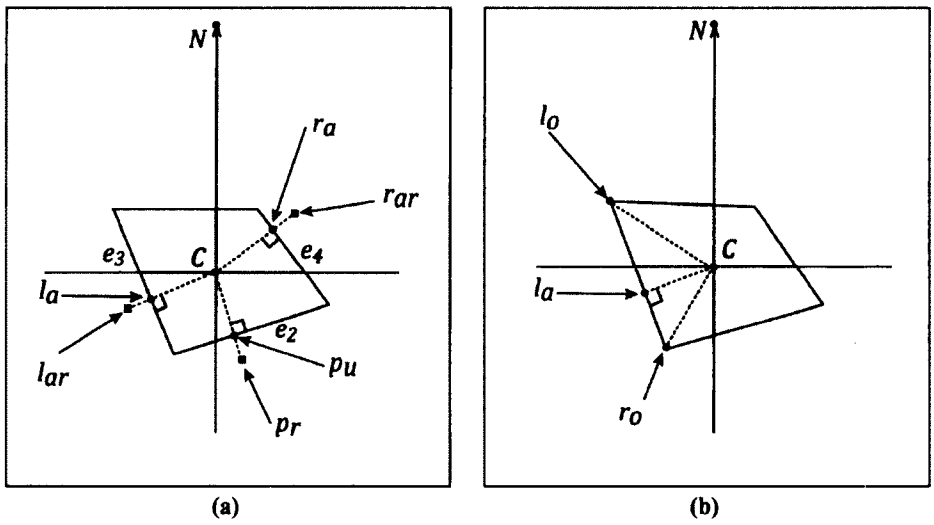


Figure 20: (a) The target-balance points used for pushing/aligning M and their resting points. (b) The target-balance points used for adjusting the orientation of M.

For a rotation task, the target-balance points are calculated from r_a and l_a by rotating the lines from C to each of r_a and l_a by some fixed degrees γ (i.e., $\pm 20^o$) (see Figure 21). For the left side, either l_r or l_l is chosen based on the required direction of rotation. The same is applied for the right side. If the target-balance rotation point does not fall on the same edge of l_a or r_a then the in-place rotation might not be stable and the temporary push sub-task will constantly become active to push M back to its center of rotation. Figure 21b shows r_r target-balance point not falling on the same edge as r_a target-balance point. In this situation, the angle γ should be as small as possible so that r_r is always on the same edge as r_a. The smaller γ is the slower the rotation becomes. In

addition, if γ has a different value on one side of M than the opposite side then the frequency of activating the temporary push sub-task will increase and accomplishing the rotation will become slower due to the back and forth switching between a rotate task and the temporary push sub-task.

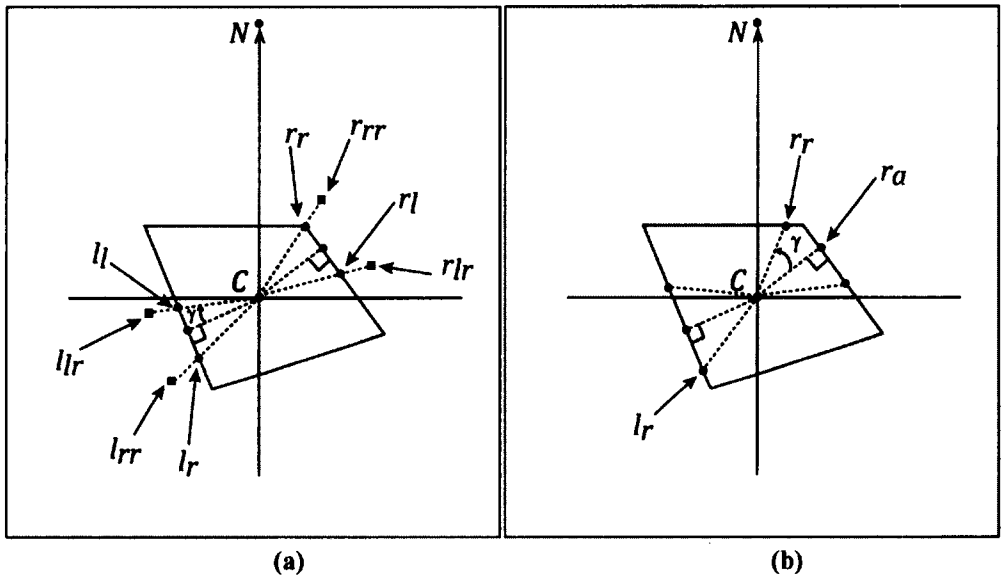


Figure 21: (a) The rotating target-balance points and their resting points. (b) Example showing r_r point not falling on the same edge as r_a point.

The *repositioning path* is a path for the robots to follow while performing the repositioning task, such that, q_0 is the first point on the path such that angle $\angle q_0 p_1 p_2 = 180^o$. Let $\overrightarrow{u_i u_{i+1}}$ be a segment in an n-segment repositioning path, $1 \leq i \leq n$. Each segment $\overrightarrow{u_i u_{i+1}}$ can be divided into equally spaced sub-segments $\overrightarrow{q_{(i,j)} q_{(i,j+1)}}$, $1 \leq j \leq k$ where $k = 2d$. Figure 22 shows that the repositioning path falls in the z_2 zone between the outer boundary of the buffer zone and the boundary of M to keep the robots away from M while repositioning. This allows enough space for a robot to pass beside a repositioning robot in between M and the repositioning path. If the robots touch M while repositioning they could cause it move unintentionally.

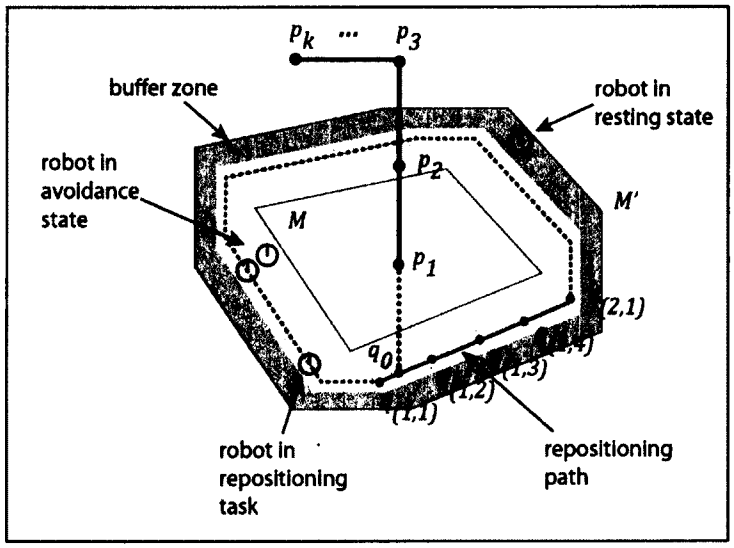


Figure 22: Repositioning path around M.

Let h be a point in W representing the "entrance" point of all robots. That is, each robot will begin at some location in W, but will head towards h in turn as they are assigned the task of pushing M. Assume that M begins with its center C at point p_1 of P. A *taxiing path* is a path from h to q_0 (i.e., the first point on the repositioning path) calculated using any shortest path algorithm, perhaps based on the *visibility graph* [52] (see Figure 23).

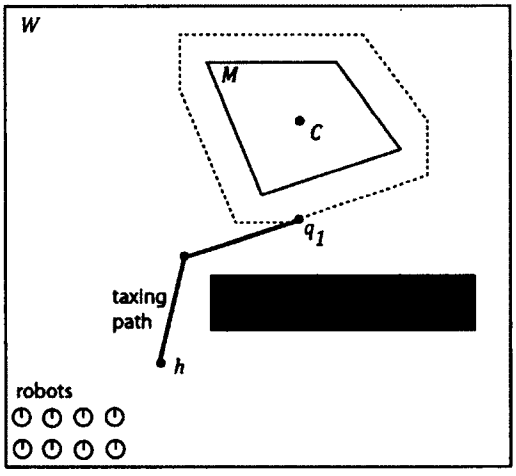


Figure 23: Taxiing path.

Our robot model assumes that the robots are circular and can move and push in any direction. The robots have mass and are simulated using friction, angular damping, but with minimal restitution to avoid bouncing.

4.2 Algorithm Details

The algorithm presented in this section will assume that there exists a valid path P between s and t. The initial step in this algorithm is to initialize the GC which retrieves P from the path planner module and constructs the path task list that the robots are required to perform. In order to push M and maintain its fixed orientation, three sets of robots are needed (see Figure 18). One set for pushing M forward by seeking one target-balanced point from the "back" and two sets for adjusting M (one from the left and another from the right; each by seeking one target-balanced point). Additionally, for a rotate task, one target-balance point is assigned on each side of M to perform the in-place rotation (see Figure 27). Each of the target-balance points has an associated target-rest point; just to keep the robots close enough to M while it is moving.

The GC initializes the *path monitor* to ascertain whether or not M is being moved rigorously along P. The path monitor process (see Algorithm 4.2) monitors the current position of M along the path P and is capable of determining if a push or a rotation is needed and in what direction. Moreover, it monitors if the endpoint of a path segment is reached or a rotation angle has reached a limit. Once these limits are reached, then it retrieves the next task from the path task list.

Algorithm 4.2

PathMonitor(T, M)

```
1  FOR each task t of T
2      WHILE t has not been completed DO
3          c ← GetPushObjectCurrentPosition(M)
4          IF t is PUSH THEN
5              s ← GetCurrentSegmentEndPoint(t)
6              IF s equals c THEN
7                  Mark t as completed
8          ELSE IF t is ROTATE THEN
9              a ← GetPushObjectCurrentAngle(M)
10             IF a equals task angle THEN
11                 Mark t as completed
12         ELSE IF t is REPOSITION THEN
13             IF RobotsReady() THEN
14                 Mark t as completed
15     ENDWHILE
16     RemoveTask(t)
17     UpdateStabilizer()
18 ENDFOR
```

The algorithm loops through all the path tasks and for each push task it monitors the distance between c, which is the current position of M and s, which is the endpoint of the current path segment (line 6). Once s and c are close enough, the task is removed from the list of tasks and the `UpdateStabilizer()` recalculates the stabilizer tool and computes new target points. Line 10 verifies whether or not the in-place rotation of M is properly performed where if the rotation is stable, the orientation of M is compared to the task's angle (i.e., orientation of M). If that is within a certain threshold then the task is removed and the stabilizer tool is recomputed. In line 13, `RobotsReady()` verifies whether the robots reached their assigned target points.

The GC initializes the stabilizer at the beginning of the algorithm but it is recomputed regularly to correspond to the movement of M. The stabilizer tool is defined with respect to M and takes as input a destination point as well as the vertices of M. It then provides target points and orientation angles of M. The number of target points could vary from one task type to another.

Algorithm 4.3 shows each robot's main loop where in line 2 and 3, the robot's specialty and its current position parameters are retrieved and passed to the GC through `TargetPointsProvider()` to retrieve target points. The robot iterates through the target points and seeks them one by one. Algorithm 4.4 facilitates providing target points to the robots based on their current position and their specialty. This algorithm retrieves the current task from the path monitor (line 2) and then checks whether the target point that the robot is trying to reach is visible (i.e., the line from the center of the robot to the target point crosses the interior of M). If not visible then a repositioning path is provided (line 3). Note that only the set of points $q_{(i,1)}$ are sent to the robot and the robot handles creating the set $\{q_{(i,j)} - q_{(i,1)}\}$ which is the rest of the points (see Figure 24). Line 6 calls `CheckPushStability()` to verify whether or not a deviation occurred while pushing M. A new target point is retrieved from the stabilizer tool (see Algorithm 4.5) based on whether the deviation is an orientation or alignment sub-task. In lines 8 and 12, `GetTargetPointFromStabilizer()` retrieves the target point from the stabilizer tool based on the task type and the robot specialty. In this case, Q will contain only on point. In line 16, if the task is to reposition then Q will contain the set of points $q_{(i,1)}$ of the repositioning path.

Algorithm 4.3

RobotMainLoop()

```
1  WHILE robotIsRunning() DO
2     robotSpecialty ← getRobotType()
3     pos ← getCurrentPositon()
4     Q ← TargetPointsProvider(robotSpecialty, pos)
5     FOR each p of Q
6        SeekTargetPoint(p)
7     ENDFOR
8  ENDWHILE
```

Algorithm 4.4

TargetPointsProvider(robotSpecialty, pos)

```
1  p ← nil
2  t ← GetCurrentTaskFromPathMonitor()
3  Q ← GetRepositioningPathFromGC(pos,t,robotSpecialty)
4  IF Q is empty THEN
5     IF t is PUSH THEN
6        p ← CheckPushStability(robotSpecialty)
7        IF p is nil THEN
8           p ← GetTargetPointFromStabilizer(t,robotSpecialty)
9        Q ← Q ∪ {p}
10    ELSE IF t is ROTATE THEN
11       CheckRotationStability()
12       p ← GetTargetPointFromStabilizer(t,robotSpecialty)
13       Q ← Q ∪ {p}
14    ELSE IF t is REPOSITION THEN
15       task ← GetNextTaskFromPathMonitor()
16       Q ← GetRepositioningPathFromGC(pos,task,robotSpecialty)
17 RETURN Q
```

In the next three sections we will present each of the tasks in detail. We will show how each of the target points is calculated and in what cases the repositioning path is constructed for an individual robot.

Reposition Task

If the direction from $\overrightarrow{p_{i-1}p_i}$ to $\overrightarrow{p_ip_{i+1}}$ changes or the target point that the robot is trying to reach is not visible (i.e., the line from the center of the robot to the target point crosses the interior of M) then the robot is required to reposition around M to reach the newly repositioned target points. As a result, if the current task is to "reposition" then a repositioning path is calculated where the start point of the robot would be the closest point on the repositioning path to its current location and the end point is the closest point on the repositioning path to the new target point. Figure 24 shows an example where a robot is required to reposition since the line from its center to the new target point intersects M. The start point of the robot is $q_{(1,3)}$ and the destination point $q_{(7,4)}$.

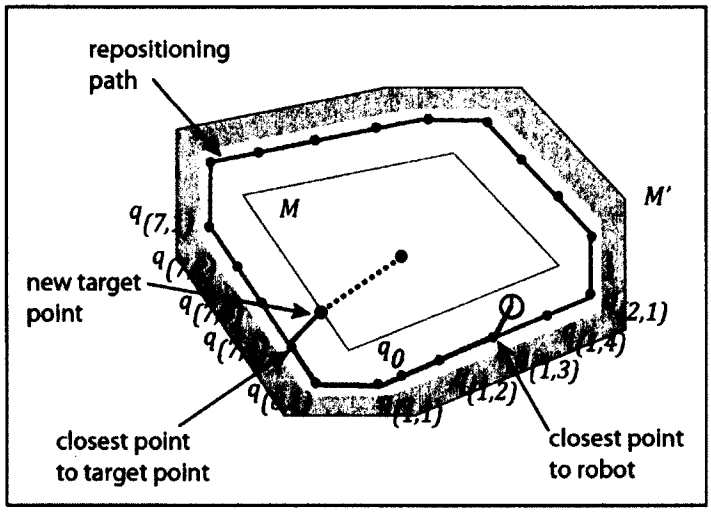


Figure 24: Example of a robot repositioning on the path from point $q_{(1,3)}$ to point $q_{(7,4)}$.

Push Task

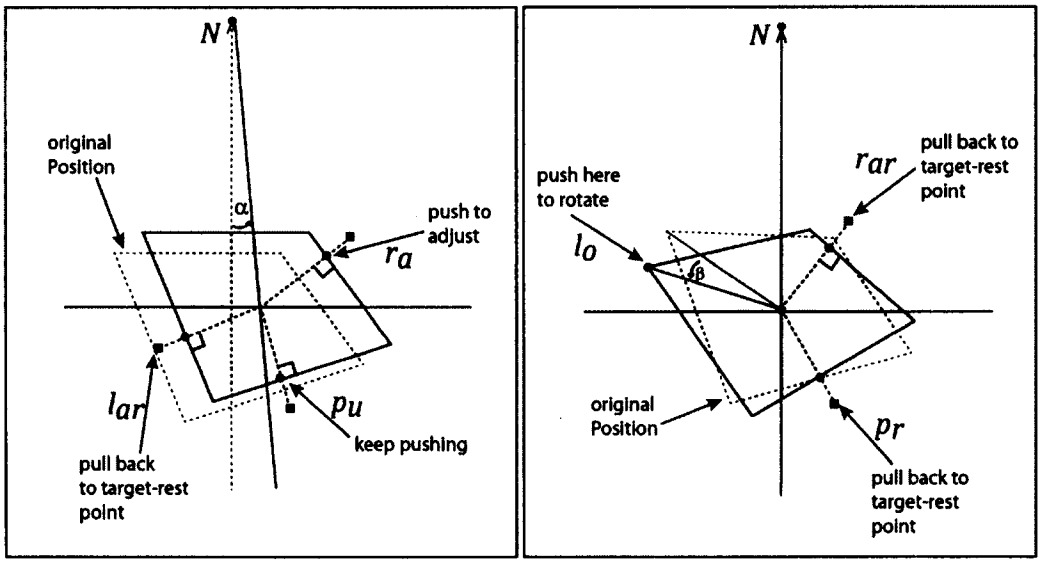
If the task is to "push" then the path monitor determines whether or not M has deviated from its current alignment or orientation. If no deviation occurs, then the target point is retrieved from the stabilizer tool based on the task type and the specialty of the robot. Algorithm 4.5 shows that if a deviation in alignment occurs then `GetAlignmentTargetPoint()` retrieves the appropriate target point from the stabilizer tool based on the task and the robot's specialty. In line 6 and 8, if M is deviating by a constant number ε, (i.e., $\varepsilon = 0.1$ was chosen experimentally) then an adjustment is required. In line 7, if the alignment of M is deviating to the right then for a left-adjuster robot, $p = l_{ar}$ and for a right-adjuster robot $p = r_a$. In addition, if M is deviating left then for a right-adjuster robot, $p = r_{ar}$ and for a left-adjuster robot $p = l_a$. Figure 25a shows M deviating to the right where the right robot seeks the right target-balance point r_a to adjust M and the left robot goes back to its target-rest point l_{ar} to stay away from M while maintaining a close distance.

Moreover, the algorithm shows that if a deviation in orientation occurs then `GetOrientationTargetPoint()` retrieves the appropriate target point from the stabilizer tool based on the task and the robot's specialty. In line 9, if the orientation of M is deviating clockwise then for a left-adjuster robot, $p = r_o$ and if the orientation of M is deviating counter-clockwise then for a left-adjuster robot, $p = l_o$. In either case, a right-adjuster robot goes back to its target-rest point, thus, $p = r_{ar}$. Figure 25b shows M rotating counter-clockwise. The left-adjuster robots seek the l_o target-balance point while the right-adjuster and pusher robots go back to the r_{ar} and p_r target-rest points, respectively.

Algorithm 4.5

***CheckPushStability*(robotSpecialty)**

```
1  p ← nil
2  a ← current orientation angle of M from the stabilizer
3  b ← orientation angle of M from the task properties
4  c ← current alignment angle of M from the stabilizer
5  d ← alignment angle of M from the task properties
6  IF absolute(c - c) < ε THEN
7      p ← GetAlignmentTargetPoint(robotSpecialty)
8  IF absolute(a - b) < ε THEN
9      p ← GetOrientationTargetPoint(robotSpecialty)
10 RETURN p
```



(a) (b)

Figure 25: (a) Example showing M deviating from P to the right. (b) Example showing M rotating counter-clockwise, the left orientation l_o target is used to adjust.

The alignment sub-task is used to keep M moving along a straight line segment. To handle this, a left turn or a right turn angle is determined between the segment's endpoints and the main axis of the stabilizer. Let S_1 and S_2 be the endpoints of the

segment that M is required to follow. Let A_1 and A_2 be the endpoints of the stabilizer tool rays L_1 and L_2. In Figure 26, the points S_2, S_1 and A_2 produce a right turn, indicative of M deviating to the right. Additional force is needed to push M from the right side to return to its normal path. The same type of operation is applied if the angle is a left turn where the force should be applied from the left side of M. While pushing M forward on a path, the pusher's target-balance point may be inaccurate causing M to deviate to either the left or the right. As a result the adjuster robots on each side of M are pulled back to their target-rest points based on M's deviation angle. For instance, if M is to be pushed to the right, the left robots will seek the target-balance point on M and the right robots will pull back to their target-rest point removing any potential interference of M.

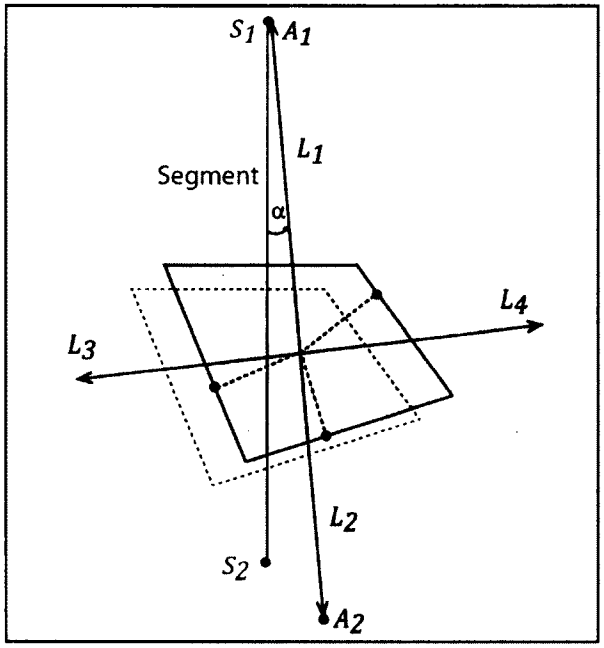


Figure 26: Re-aligning M as it deviates from its path using left turn angle test.

Rotate Task

If the task is to "rotate", then the path monitor determines whether or not M is being rotated perfectly about its center point C. Algorithm 4.6 describes how this is done. Assume that M is being rotated at point p_i of P, if M is deviating from p_i by a distance equals to 10 (line 3) then the rotate task is placed on hold while a temporary stabilizer tool is constructed (line 4) with p_i as a destination point and a temporary push task is used (line 5). A distance of 10 was chosen experimentally so as to prevent M from deviating dramatically from p_i. If the distance is too small (e.g., 1) then the frequency of switching to temporary push tasks would increase. Every simple collision between the robots and M would cause back and forth switching between rotating and pushing M. In line 5, the GC is informed by `CreatePushTask()` to create a new task and insert it at the beginning of the task list. (Note that, the first task in the task list is always the current task to be processes and after it is accomplished then it is deleted). Once M is back to p_i then the temporary push task is deleted and the original rotate task becomes the first task in the tasks list again. If no deviation occurs, then the target point is retrieved from the stabilizer tool based on the task type and the specialty of the robot. Friction and angular damping may interfere with the stability of the rotation of M. Therefore, by performing a push from two opposite sides of M, the in-place rotation will be more balanced. Figure 27 depicts an example where M must rotate counter-clockwise. Both l_r and r_r are used for rotating M where the pusher robots go back to their resting point.

Algorithm 4.6

CheckRotationStability(M)

```
1   p ← GetCenterPointOfMAt(pi)
2   c ← GetCurrentCenterPointOfM()
3   IF absolute(p - c) < 10 THEN
4       CreateTempStabilizer(p, M)
5       CreatePushTask()
```

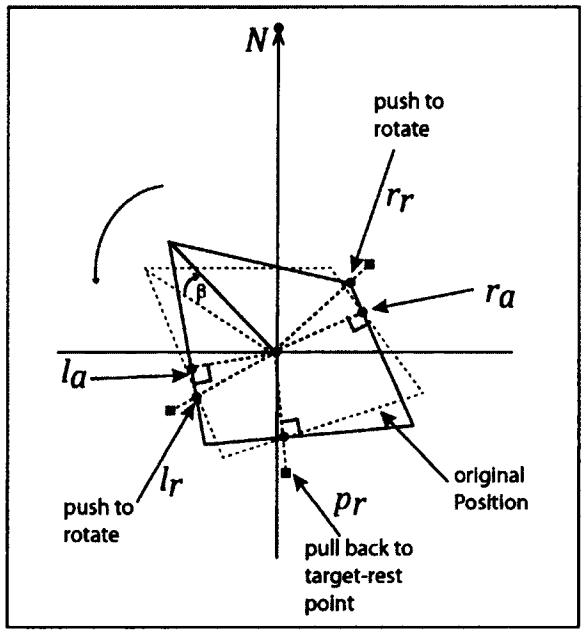


Figure 27: Example showing M during a rotate task. Both l_r and r_r are used to rotate M counter-clockwise.

4.3 Implementation Details

Figure 28 shows the interaction between the GC and the robots. The GC initializes the stabilizer tool and the path monitor and also it generates the task list from the path P for the robots to perform. After the robot registers with the GC, it is assigned a specialty such as a pusher, left-adjuster or right-adjuster. In addition, the GC uses a visibility graph algorithm to create the taxiing path. The robot requests target points from targetPointsProvider() (see Algorithm 4.4). Based on the task type, the GC retrieves target points from the taxiing path, the repositioning path or the stabilizer tool and transmits them to the robot. Each of the robots has a set of behaviors that runs concurrently and are activated based on the current task type retrieved from the path monitor, as well as the target points and the orientation of M supplied by the stabilizer tool. In the algorithm details, we gave a description of each of the tasks and sub-tasks.

The robots use a behavior arbitration scheme that iterates through the task-related behaviors in prioritized order and retrieves a command (i.e., *seek command*) and target point values (see Algorithm 4.7).

Algorithm 4.7

BehaviorArbitration()

```
1  task ← GetCurrentTaskFromPathMonitor()
2  FOR each behavior b in behaviors
3      executeBehavior(b, task)
4      c ← GetCommandFromBehavior(b)
5      p ← GetTargetPointFromBehavior(b)
6      SetCurrentCommandAndTarget(c, p)
7  ENDFOR
```

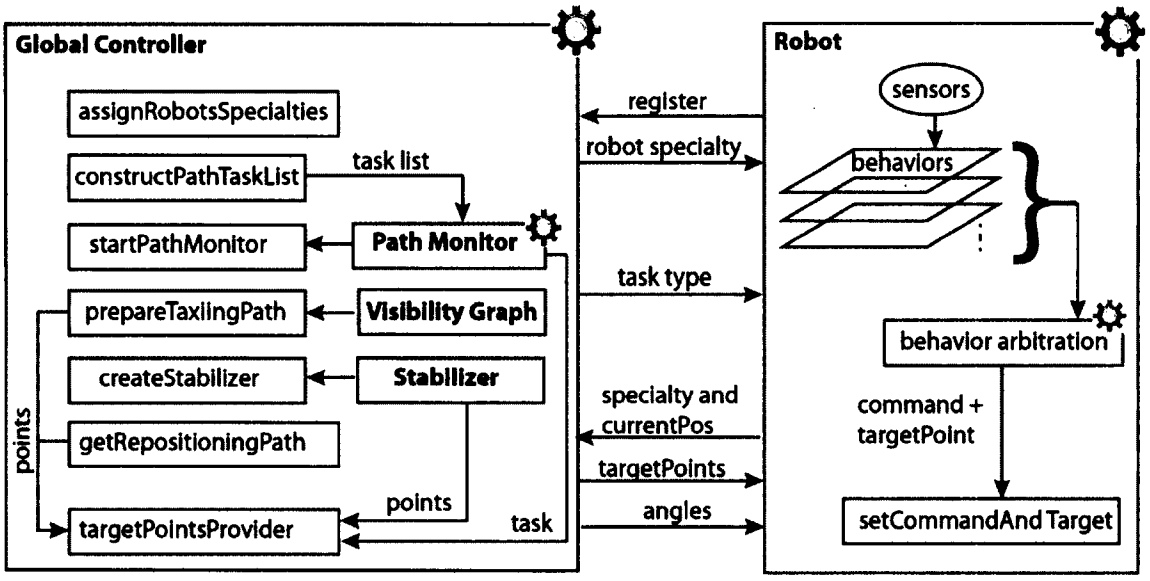


Figure 28: Global Controller interaction with a robot.

The motion of the robots is based on seeking target points. In general, if the intention is to move the robot towards a specific destination, then a path can be generated where each point along the path acts as a target point until the final destination is reached. The repositioning path is an example of that. In order to push M, the robot receives a target point on the boundary of M. When reaching this target point, the robot collides with M causing the latter to move. Our implementation uses the seeking algorithm presented by Craig Reynolds [51] where at each time-step a *steering force* is applied to perform a smooth turn by the robot towards a target point (see Figure 29). In this seek algorithm, presented in Algorithm 4.8, the desired velocity vector is the fastest way to reach the target point. In line 3, the desired velocity vector is the normalized difference between the target point and current position of the robot, scaled to be at most the maximum speed (i.e., MAX_SPEED) of the robot. In line 4, the steering force is the difference between the desired velocity vector and the current velocity vector which is always limited to the maximum force. In line 5, the current velocity vector has a new value which is the combination of the current velocity vector and the determined steering force vector which is also limited to the maximum speed.

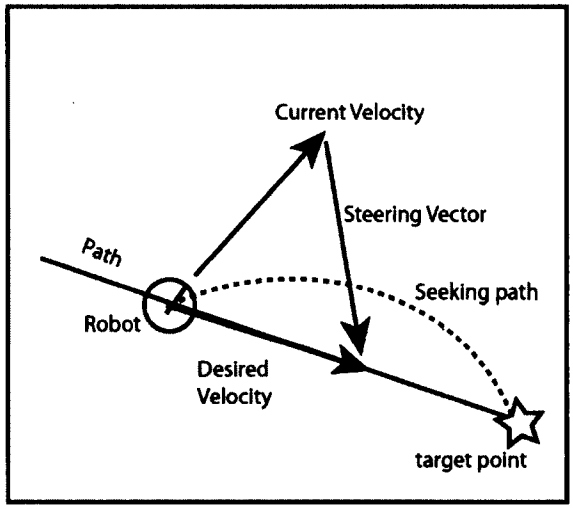


Figure 29: Seek operation where the steering force is applied to direct the robot towards a target point.

Algorithm 4.8

Seek(target)

```
1  p ← GetRobotPosition()
2  current_velocity ← GetRobotLinearVelocity()
3  desired_velocity ← Normalize(target - p) * MAX_SPEED
4  steering_force ← MIN(desired_velocity - current_velocity, MAX_FORCE)
5  current_velocity ← MIN(current_velocity + steering_force, MAX_SPEED)
```

4.3.1 Robot Behaviors

Our algorithm has been partly implemented using the behavioral model perceived by Brooks [13][14]. The flow of the model creates a simple behavioral system that removes the notion of central control system and enforces self-organizational behavior. In our design, due to the immense amount of processing that would be needed by a single robot in order to retrieve the path, monitor the motion of M, and calculate the locations of the target points, we decided to use a global controller. Using the behavior model, a robot may pursue multiple goals simultaneously allowing it to move, avoid collision, push, rotate, and reposition as shown in Figure 30. Our implementation, as in Brooks' behavioral model, separates and prioritizes the problem into various behaviors.

This architecture enables the robots to be robust and efficient. The behavior model is more responsive to higher priority behaviors like collision avoidance and repositioning. For instance, the collision avoidance and taxiing behaviors are needed when the robots are traveling towards M. Once a robot reaches the end of its taxiing path then the repositioning behavior is activated to drive the robot to a target-rest point based on its assigned specialty. Before the pushing or rotating behaviors are activated, the robots need to reposition at their target-rest points, thus allowing enough time for other robots to

reach their target-rest points in order for the pushing task to begin. The following seven sections will describe each behavior used in our model.

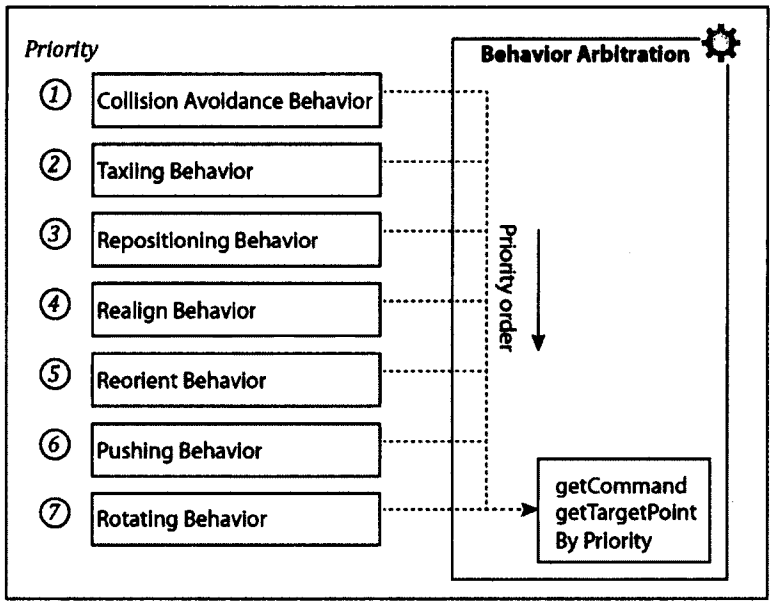


Figure 30: Behavior model

4.3.1.1 Repositioning Behavior

Behavior 4.9 shows this behavior which obtains a repositioning path from the *GC* (line 3). It is activated when the current target point that the robot is supposed to reach is not visible anymore. This is verified by sending a request (line 2) to the *GC* through `isTargetVisible()`. The algorithm iterates through the repositioning path points (lines 3-11) where the robot attempts to seek each point until the end of the path is reached (see Figure 24). Once the current target point is visible to the robot then the command is set to NONE and the target point is set to **nil**. In the next section we will present the collision avoidance behavior which has a higher priority than the repositioning behavior. Note that,

since the behaviors are concurrent then the repositioning loop in line 4 keeps processing while the collision avoidance behavior is active.

The data communication between this behavior and the *GC* is limited to a request to the *GC* to verify the visibility of the target point where the current position of the robot is sent in the request. In addition, a request to the *GC* is made to calculate the repositioning path where the current position and specialty of the robot are sent in the request. The response from *GC* is a **true** or **false** value (denoting the visibility of the target point) and the repositioning path target points.

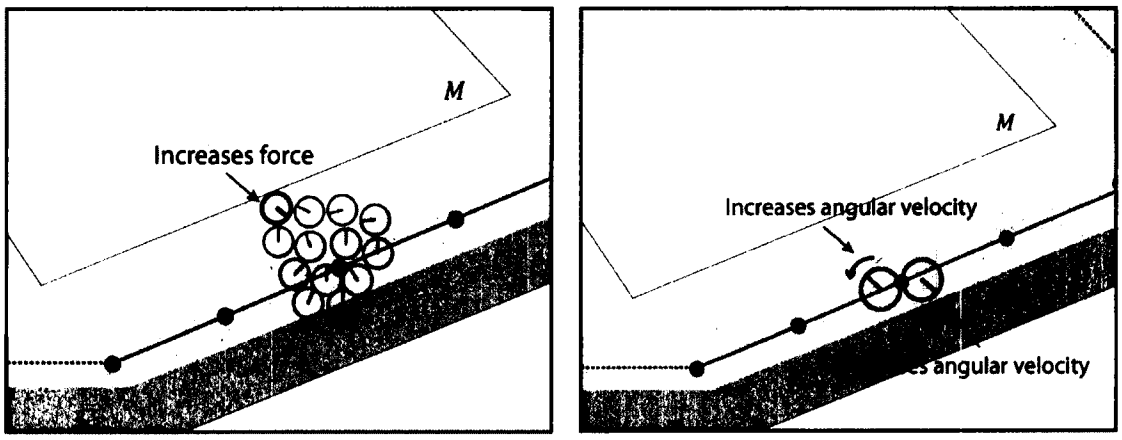
Behavior 4.9

RepositioningBehavior()

```
1   pos ← CurrentRobotPosition()
2   IF not isTargetVisible(pos) THEN
3       Q ← GetRepositioningPathFromGC(pos, robotSpecialty)
4       FOR each p of Q
5           pos ← CurrentRobotPosition()
6           IF pos is p THEN
7               RemoveFirstPoint(Q)
8           ELSE
9           SetTarget(p)
10          SetCommand(SEEK)
11      ENDFOR
12  ELSE
13      SetTarget(nil)
14      SetCommand(NONE)
```

4.3.1.2 Collision Avoidance Behavior

The aim of this behavior is to prevent robots from colliding with each other while taxiing or with M while repositioning. In Behavior 4.10, if the GC detects that the distance between the robot and other robots is within some threshold (i.e., the distance between the two robots' center points, which equals the diameter of the robot), then the robot increases its angular velocity in order to spin away from the other robots. In Line 2, the angular velocity is increased and in line 3, it is set back to normal when the distance between the two robots is greater than the diameter. As the robot is repositioning, it may collide with other robots that also are in a repositioning behavior. The collision could cause the robot to collide with M and move it unintentionally. To avoid these unnecessary collisions, the robot increases its force (line 6) so as to push through crowds of other robots (see Figure 31a). Once the robot reaches the intended target point the force is reset back to normal (line 7). The main reason for increasing the angular velocity is that when multiple robots are seeking the same target point, (i.e., while taxiing from their home location to M or while repositioning), they tend to stagnate at that target point (see Figure 31b) . By applying a spin, the robots will stay in motion and attempt to reach the target point one by one. As the robot reaches its target point, the spin will again allow it to escape from the crowded area and continue taxiing along the path seeking the next target point. The data communication between this behavior and the GC is limited to a request to the GC to detect collisions with other robots or M and the response from the GC is a **true** or **false** denoting possible collision.



(a) (b)

Figure 31: (a) increase force on robots colliding with M. (b) Increase angular velocity on stagnating robots

Behavior 4.10

AvoidingBehavior()

```
1  IF DetectPotentialCollisionWithRobots() THEN
2      IncreaseAngularVelocity()
3  ELSE
4      ResetAngularVelocity()
5  IF in repositioning mode THEN
6      IF DetectCollisionWithWall() THEN
7          IncreaseForce()
8      ELSE
9         ResetForce()
```

4.3.1.3 Taxiing Behavior

In Behavior 4.11, this behavior receives the taxiing path Q from the GC (line 1) and remains active until the end of the path is reached. At every iteration, this behavior tests the current path point against the current position of the robot (line 4). Once a target point is reached then it is removed from the taxing path (line 5). If the taxiing path list is empty

then the command is set to NONE and the target point is set to nil. The data communication between this behavior and the GC is limited to requesting the taxing path from the GC and the response from the GC is a list of target points that form the taxiing path.

Behavior 4.11

TaxiingBehavior()

```
1  Q ← GetTaxiingPath()
2  FOR each point p of Q
3      pos ← CurrentRobotPosition()
4      IF pos is p THEN
5          RemoveFirstPoint(Q)
6      ELSE
7          SetTarget(p)
8          SetCommand(SEEK)
9  ENDFOR
10 SetTarget(nil)
11 SetCommand(NONE)
```

4.3.1.4 Reorient Behavior

This behavior adjusts the orientation of M by seeking the target-balance point on the left side of M (see Figure 20b). As the robot reaches the target-balance point, it gives M a slight push causing a rotation clockwise or counter-clockwise based on the angle input from the stabilizer tool (see section 4.2). Behavior 4.12 is activated if the task type is a PUSH. In line 2, it shows that if the orientation of M requires an adjustment, then the pusher and right adjuster robots are pulled back to their target-rest points p_r and r_{ar} (lines 4-7). Based on the rotation angle needed, the left adjuster robots seek the target-balance points l_o or r_o (lines 11-13). If the current task type is not PUSH then the command is set

to NONE and the target point is set to **nil**. The data communication between this behavior and the GC is limited to requesting a target point, the orientation of M and the angle of L_1 of the stabilizer tool (see Figure 18) where the target point type is sent in the request.

Behavior 4.12

ReorientBehavior(task)

```
1  IF task is PUSH THEN
2      IF IsAdjustNeeded() THEN
3          IF robotSpecialty is PUSHER THEN
4              GetTargetFromStabilizer(pr)
5              SetCommand(SEEK)
6          ELSE IF robotSpecialty is RIGHT_ADJUSTER THEN
7              GetTargetFromStabilizer(rar)
8              SetCommand(SEEK)
9          ELSE IF robotSpecialty is Left_ADJUSTER THEN
10             IF IsLeftRotationNeeded() THEN
11                 GetTargetFromStabilizer(ro)
12                 SetCommand(SEEK)
13             ELSE
14                 GetTargetFromStabilizer(lo)
15                 SetCommand(SEEK)
16 ELSE
17     SetTarget(nil)
18     SetCommand(NONE)
```

4.3.1.5 Realign Behavior

This behavior maintains a fixed direction of M along the path P. In Behavior 4.13, this behavior is activated if the task type is a PUSH. In lines 2-4, the pusher robots will seek the target-rest point p retrieved from the stabilizer tool. In line 5-11, if a right alignment of M is needed (see Figure 25a) then the left-adjust robots pull back to the target-rest point l_{ar} and the right-adjust robots seek the target-balance point r_a. In lines

12-18, if a left alignment of M is needed then the left-adjust robots pull back to the target-balance point l_a and the right-adjust robots seek the target-rest point r_{ar}. If the current task type is not PUSH then the command is set to NONE and the target point is set to **nil**. The data communication between this behavior and the GC is limited to requesting a target point, the angle of the current path segment and the angle of L_1 of the stabilizer tool (see Figure 18) where the target point type is sent in the request.

Behavior 4.13

RealignBehavior(task)

```
1  IF task is PUSH THEN
2      IF robotSpecialty is PUSHER THEN
3          GetTargetFromStabilizer(p)
4          SetCommand(SEEK)
5      IF RightAdjustIsNeeded() THEN
6          IF robotSpecialty is LEFT_ADJUSTER THEN
7              GetTargetFromStabilizer (lar)
8              SetCommand(SEEK)
9          ELSE IF robotSpecialty is RIGHT_ADJUSTER THEN
10             GetTargetFromStabilizer(ra)
11             SetCommand(SEEK)
12     IF LeftAdjustIsNeeded() THEN
13         IF robotSpecialty is LEFT_ADJUSTER THEN
14             GetTargetFromStabilizer(la)
15             SetCommand(SEEK)
16         ELSE IF robotSpecialty is RIGHT_ADJUSTER THEN
17             GetTargetFromStabilizer(rar)
18             SetCommand(SEEK)
19 ELSE SetTarget(nil)
20      SetCommand(NONE)
```

4.3.1.6 Pushing Behavior

The collective pushing of all the robots will move M towards a certain goal while maintaining a fixed direction of M. In Behavior 4.14, this behavior is activated if the task type is a PUSH. In lines 2-4, the pusher robots will seek the target-rest point p retrieved from the stabilizer tool. If the current task type is not PUSH then the command is set to NONE and the target point is set to **nil**. Note that the realign and reorient behaviors run concurrently with this behavior and have higher priority. This behavior will deactivate until M is reoriented or realigned back to its correct position. The data communication between this behavior and the GC is limited to requesting a target point where the target point type is sent in the request.

Behavior 4.14

PushingBehavior(task)

```
1  IF task is PUSH THEN
2      IF robotSpecialty is PUSHER THEN
3          GetTargetFromStabilizer(p)
4          SetCommand(SEEK)
5  ELSE
6      SetTarget(nil)
7      SetCommand(NONE)
```

4.3.1.7 Rotating Behavior

In this behavior only two sets of robots are to seek target points around M depending on the rotation task direction of M (i.e., right or left). In Behavior 4.15, this behavior is activated if the task type is a ROTATE. In line 5-11, if the rotation task is to rotate M counter-clockwise then the left-adjuster robots seek the target-balance point l_r

and the right-adjuster robots seek the target-balance point r_r. In line 12-18, if the rotation task is to rotate M clockwise (by requesting the rotation type from the path monitor `RotationType()`) then the left-adjuster robots seek the target-balance point l_l and the right-adjuster robots seek the target-balance point r_l. The pusher robots seek the target-rest point p_r during the rotation task. If the current task type is not ROTATE then the command is set to NONE and the target point is set to **nil**. The data communication between this behavior and the *GC* is limited to requesting a target point, the direction of rotation where the target point type is sent in the request.

Behavior 4.15

RotatingBehavior(task)

```
1  IF task is ROTATE THEN
2      IF robotSpecialty is PUSHER THEN
3          GetTargetFromStabilizer(pr)
4          SetCommand(SEEK)
5      IF RotationType() is LEFT_ROTATION THEN
6          IF robotSpecialty is LEFT_ADJUSTER THEN
7              GetTargetFromStabilizer(lr)
8              SetCommand(SEEK)
9          ELSE IF robotSpecialty is RIGHT_ADJUSTER THEN
10             GetTargetFromStabilizer(rr)
11             SetCommand(SEEK)
12     IF RotationType() is RIGHT_ROTATION THEN
13         IF robotSpecialty is LEFT_ADJUSTER THEN
14             GetTargetFromStabilizer(ll)
15             SetCommand(SEEK)
16         ELSE IF robotSpecialty is RIGHT_ADJUSTER THEN
17             GetTargetFromStabilizer(rl)
18             SetCommand(SEEK)
19 ELSE
20     SetTarget(nil)
21     SetCommand(NONE)
```

4.4 Summary

In this chapter, we presented an algorithm for pushing M with multiple robots along path P retrieved from the path planner by the GC. The robots have specialties such as pusher, left-adjuster and right-adjuster. The path P is parsed into a set of tasks for the robots to perform where a task can be to push, rotate or reposition around M. The GC initializes the path monitor to monitor the motion of M along the path P as well as the stabilizer tool to provide target points around M for the robots to seek. The target points provide a means for the robots to push and rotate M with rigor as well as to travel from one location to a destination location in W following a strict path. A repositioning path is created to allow a robot to travel around M to a target point. Initially, the robots use a taxiing path from their home location to the first point on the repositioning path. Once the robots reach M then they seek target points provided by the stabilizer tool based on the current task provided by the path monitor. In our implementation, we used a behavior-based system where each behavior has a priority. For instance, a repositioning behavior has a higher priority than the push or rotate behaviors. While the robot is pushing, the orientation and alignment behaviors have higher priority than the push behavior. In the next chapter, we will present representative experiments that we performed to verify our algorithm.

Chapter 5

5 Experiments

In this chapter, we describe the experiments performed to verify our algorithms. The objective of our experiments is to show rigorous movement of M while multiple robots push M as well as to investigate how data communication load and runtime change as more robots are assigned to the problem. We believe that the experiments presented in this chapter are sufficient to prove that our algorithm is successful in reaching its goal. Our experiments are based on four shapes for M: *experiment 1* is a rectangular box, *experiment 2* is a diamond shape, *experiment 3* is a shape with corners not equal to 90^o and *experiment 4* is a somewhat circular shape. For each experiment, the robot team size was set to 3, 12, 24 and 48. Although our algorithm requires just three robots, in practice the force provided by just three robots may not be enough to push M if it is too heavy. By choosing various amounts of robots in our experiments, we can observe and quantify the effects that multi-robot interference can have both between the robots themselves and also their effect on M's deviation throughout the pushing task. We performed an additional experiment in which the path P takes different routes based on the distance d of edges in L_i as well as the rotation angle of M. Moreover, we performed an experiment with a maze-like workspace where M had to be rotated multiple times in order to move it through the maze. Finally, we describe an experiment that revealed details about the data communication load between the robots and the GC. The project has been implemented in Java using Eclipse and has been tested in the following environment: Windows 7 Enterprise 64-bit, Intel ® Core ™ i7 CPU, Q820 @ 1.73GHz, 16GB RAM.

In the following sections, we present the results from each of the experiments. Since collisions with M during the regular pushing and rotating phases are part of the task-at-hand, we are interested in determining the amount of interference which is a direct result of applying more robots to the task. Therefore, in each experiment, we present a table showing the number of collisions between the robots and M per team of robots during the repositioning tasks only. Charts are also presented showing the deviation distance of M from the path P during push and rotation tasks. The deviation distance is measured in pixels but it can be converted into any appropriate units (e.g., cm, mm, inches). In addition, measurement units can be chosen with respect to the diameter of the robots or the size of M. To get a feel for whether or not the robots are making efficient use of their time, we present tables depicting the average number of times each behavior was active during each of the experiments. Although the robots are simulated as separate threads, the time measurements for our experiments are based on the synchronous time-steps of the GC where each time-step represents a unit of time with respect to an iteration loop of the GC. The speed of the iteration loop depends on various factors such as the processor's clock frequency, the simulated robot's internal clock and the data communication rate between the robots and the GC.

5.1 Experiment 1 – Rectangular Shape

This experiment depicts the most basic of the experiments performed. The aim of this experiment is to show the efficiency of the algorithm when the line from C to the right and left adjuster target-balance points are perpendicular to segments of P as well as the line from C to the pusher target-balance point coincides with the segments of the path P (see Figure 32). The diameter of M is the smallest enclosing circle around M which is 120 pixels in this experiment.

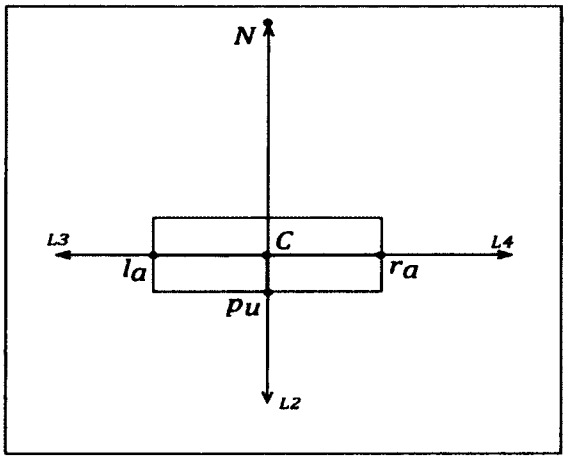
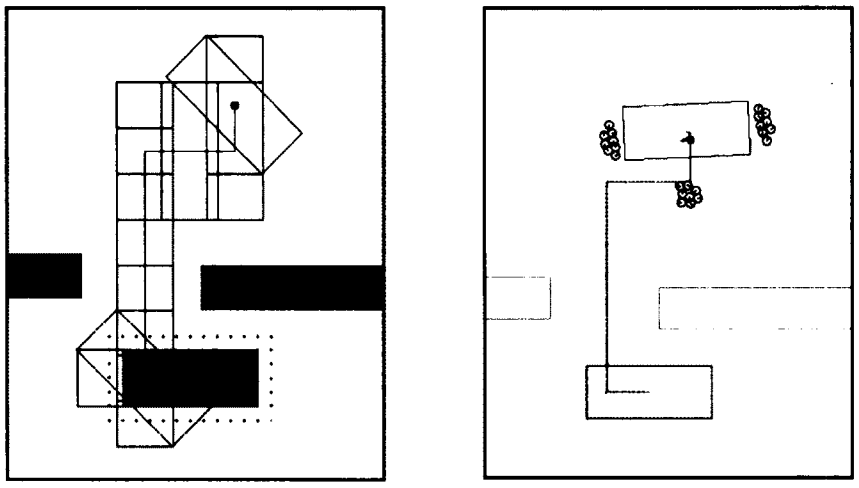


Figure 32: Rectangular shape showing target-balance points p_u, l_a and r_a being perpendicular to the edges of M.

Overall, this experiment shows minor deviation distance from P regardless of what team size we used. Figure 33a shows the translations of M along P and Figure 33b shows the trace of the center point of M along P while being pushed by 24 robots. The trace shows acceptable results where during the push tasks the deviation distance of M from P was minimal, only a small amount of deviation occurred during the first rotate task. The most stable result was achieved using three robots.



(a) (b)

Figure 33: (a) Translations of M along path P. (b) Trace along P while M is being pushed by 24 robots.

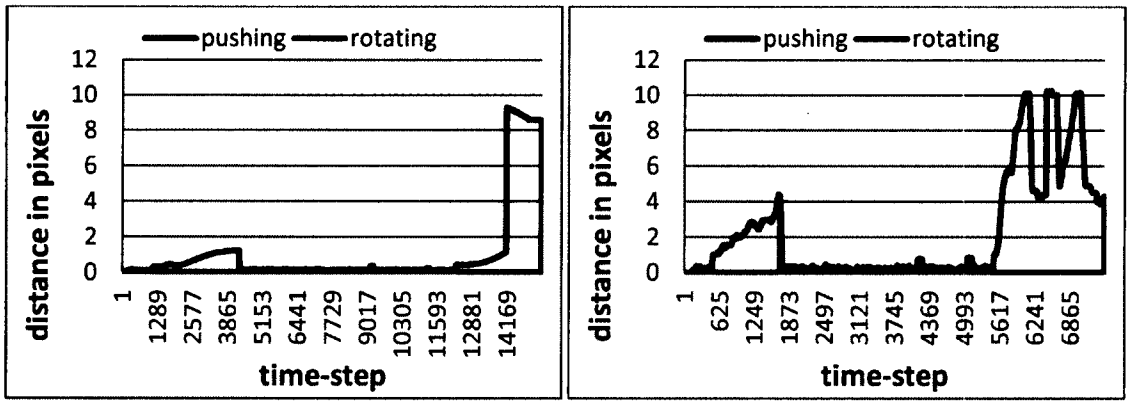
Table 2 shows the percentage of the average amount of times a behavior was active during the simulation. In addition, the table shows the average waiting time for each team of robots as the robots wait at their target-rest points. We can see from the table that the average percentage each behavior was active is consistent during the simulation for each behavior. As more robots are added then less time is spent pushing and realigning. There is a greater requirement for reorienting, repositioning and waiting.

Behaviors	**3 robots**	**12 robots**	**24 robots**	**48 robots**
Pushing	20.03%	20.87%	19.05%	18.57%
Rotating	26.58%	21.86%	24.95%	24.35%
Realignment	20.03%	19.70%	18.73%	18.18%
Reorientation	0.00%	2.62%	1.35%	2.90%
Repositioning	0.04%	0.15%	0.24%	0.30%
Waiting	33.33%	34.79%	35.68%	36.69%

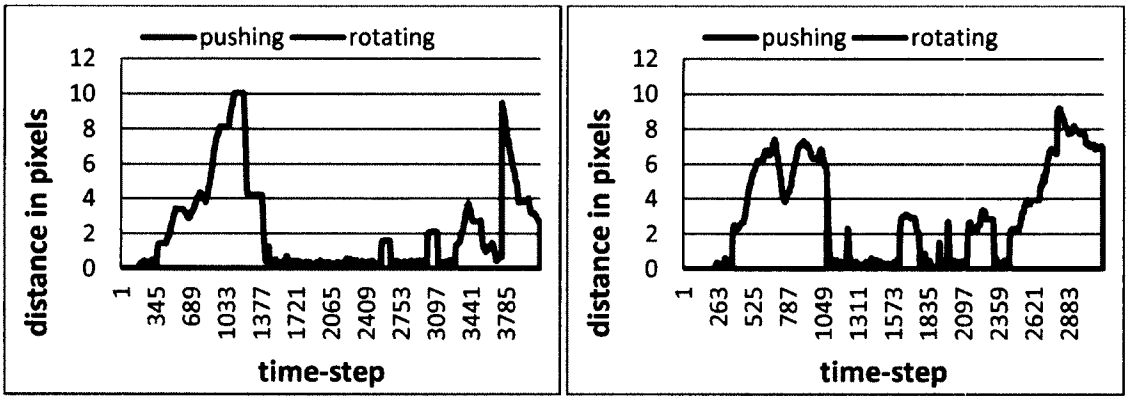
Table 2: Average amount of time each behavior was active, for various team sizes during experiment 1.

Figure 34 shows the deviation distance during pushing and rotating tasks for each time-step throughout the simulation, for 4 team sizes. The figure shows that the deviation distance for the push task is well in the range between 0 and 3 pixels in most cases. This is at most 2.5% of the diameter of M. We can see that the rotation tasks require more time-steps than that of the push tasks. This is due to the time it takes to rotate M, where during rotation tasks, M tends to deviate more from the rotation point and more readjustments are needed. The combinations of results from each of the tasks and the speed of pushing M when using a team size of 24 robots, show minor deviation from the path P where the number of time-steps are reasonably low compared to with other simulations having less number of robots. The sharp decrease in deviation distance at the end of each rotation is due to the switch in tasks from rotating to pushing as seen around time-step 1873 in Figure 34b. During a rotation task, M could deviate from the center of its rotation and after switching from a rotate task to a push task, the center of M could

still be far from *P*. When the push task activates, the robots push *M* back on track but this operation is quite fast. Figure 35 shows a magnified view of this time-step where the push task's deviation distance decreases at around time-step 1706. Notice that after the push task was activated, it took around 35 time-steps to realign *M*. The gap at time-step 1706 is due to the fact that the push task was activated just one time-step before the rotate task was deactivated. In addition, notice the sharp deviations in Figure 34b between time-steps 5617 and 6241. This is due to deviations during the rotation of *M* associated with temporary pushing tasks to push back *M* to its center of rotation. A magnified view of this portion of the graph is shown in Figure 36 where the temporary push task is activated at round time-step 6136. Once the deviation distance is back within the threshold (i.e., around 4 pixels) at around time-step 6191 then the rotate task is reactivated to continue rotating *M*. In Figure 34d, the "sharp" deviation at around time-step 1311 is somewhat misleading since this deviation spans many time-steps, making it more gradual than it appears.



(a) 3 robots (b) 12 robots



(c) 24 robots (d) 48 robots

Figure 34: Deviation distance per time-step during experiment 1.

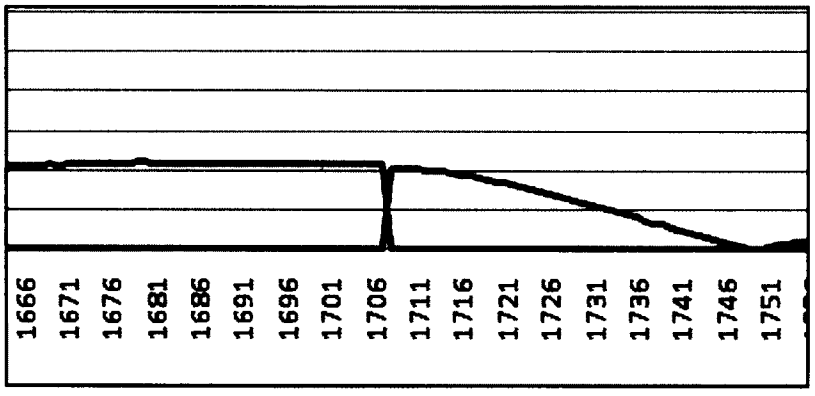


Figure 35: Switching from a rotate task to a push task.

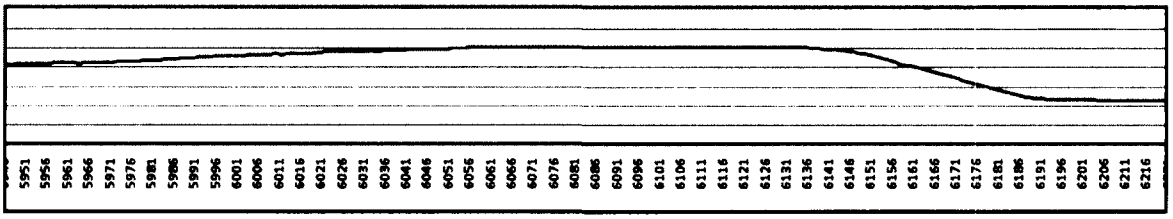


Figure 36: Temporary push is activated during rotation to push M back to its center of rotation.

Table 3 shows the average number robot-to-robot collisions as well as the collisions between the robots and M, while repositioning only. Note that we are not counting collisions during pushing, rotating, reorienting and realigning tasks since they are necessary to provide additional force for moving M. As the robots gather around a target-rest point or during repositioning, the number of inter-robot collisions increases causing their cumulative force as they collide to potentially push them towards M producing unnecessary collisions with M, thereby causing M to stray away from P. The effect of the collisions in this experiment is not as high and thus all the results in Figure 34 show consistent deviation distance as the number of robots increase. This is due to the rectangular shape of M. We could see more deviations with other shapes. The data in Table 3 does not include normal collisions with M during pushing, rotating, realignment or reorientation tasks.

Number of collisions	**3 robots**	**12 robots**	**24 robots**	**48 robots**
Robot-to-Robot	6.33	684.58	642.63	601.02
Robot-to-M	6.67	14.25	24.46	43.48

Table 3: Number of collisions during the repositioning tasks for experiment 1.

5.2 Experiment 2 – Diamond Shape

In this experiment, we used the diamond shape for M in order to show the efficiency of the algorithm when M has 90^o corners and edges that are not parallel to the segments of P (see Figure 37). In addition, the pusher target-balance point falls on top of one of the adjusters target-balance points. One more important characteristic of this experiment is that the target-balance points are calculated based on which edge of M the stabilizer tool axis intersects. Therefore, a slight deviation in the orientation or direction of M could cause the wrong edge to be chosen, especially if the axis of the stabilizer tool passes through the vertices of M. The diameter of M is the smallest enclosing circle around M which is 100 pixels in this experiment.

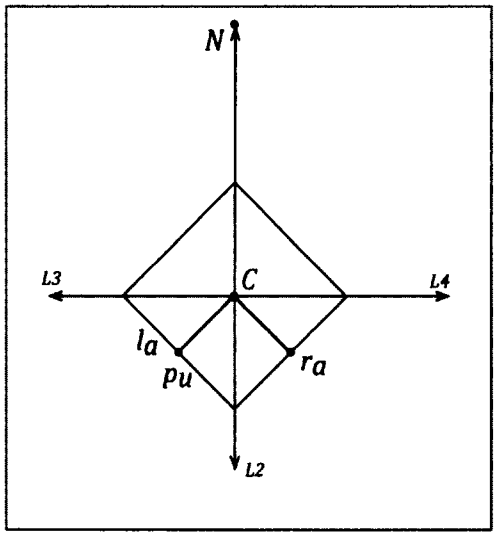
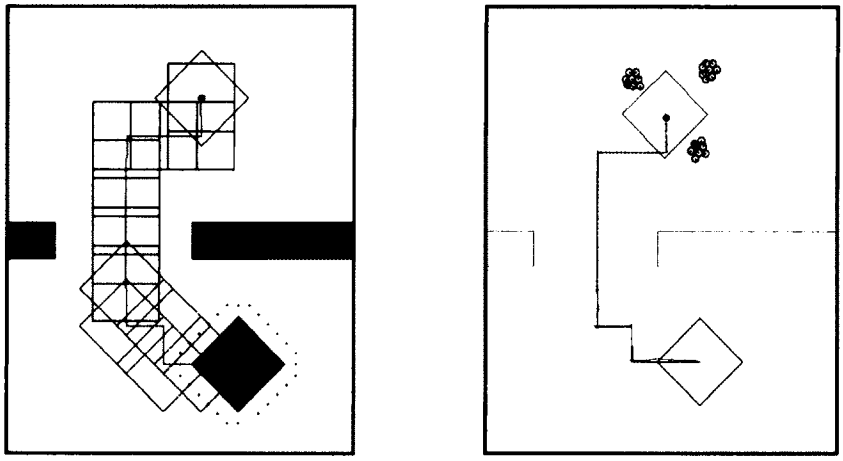


Figure 37:Diamond shape showing overlap of target-balance points p_u, l_a and the stabilizer tool's axis passing through the corners of M.

Figure 38a shows the translations of M along P and Figure 38b shows the trace of the center point of M along P while being pushed by 24 robots. The trace shows good results where during the push tasks the deviation distance was minimal and only a small amount of deviation occurred.



(a) (b)

Figure 38: (a) Translations of M along P. (b) Trace along P while M is being pushed by 24 robots.

Figure 39 shows the deviation distance per time-step. Notice that when we used teams of 3, 12, and 24 robots the pushing and rotating deviation distance was in the range between 0 and 5 which is at most 5% of the diameter of M. Using 48 robots the deviation distance climbed to around 10 which is 10% of the diameter of M. Table 4 shows the number of collisions that occurred during repositioning. In the first few segments of P, the deviation was higher due to unnecessary interference between the robots and M as they attempted to reposition around M causing it to move away from P. In order for M to shift back on track, the robots must stabilize around M. From this experiment, we can conclude that it is essential for the number of robots and the size of M to be determined prior to the operation in order to obtain rigorous results.

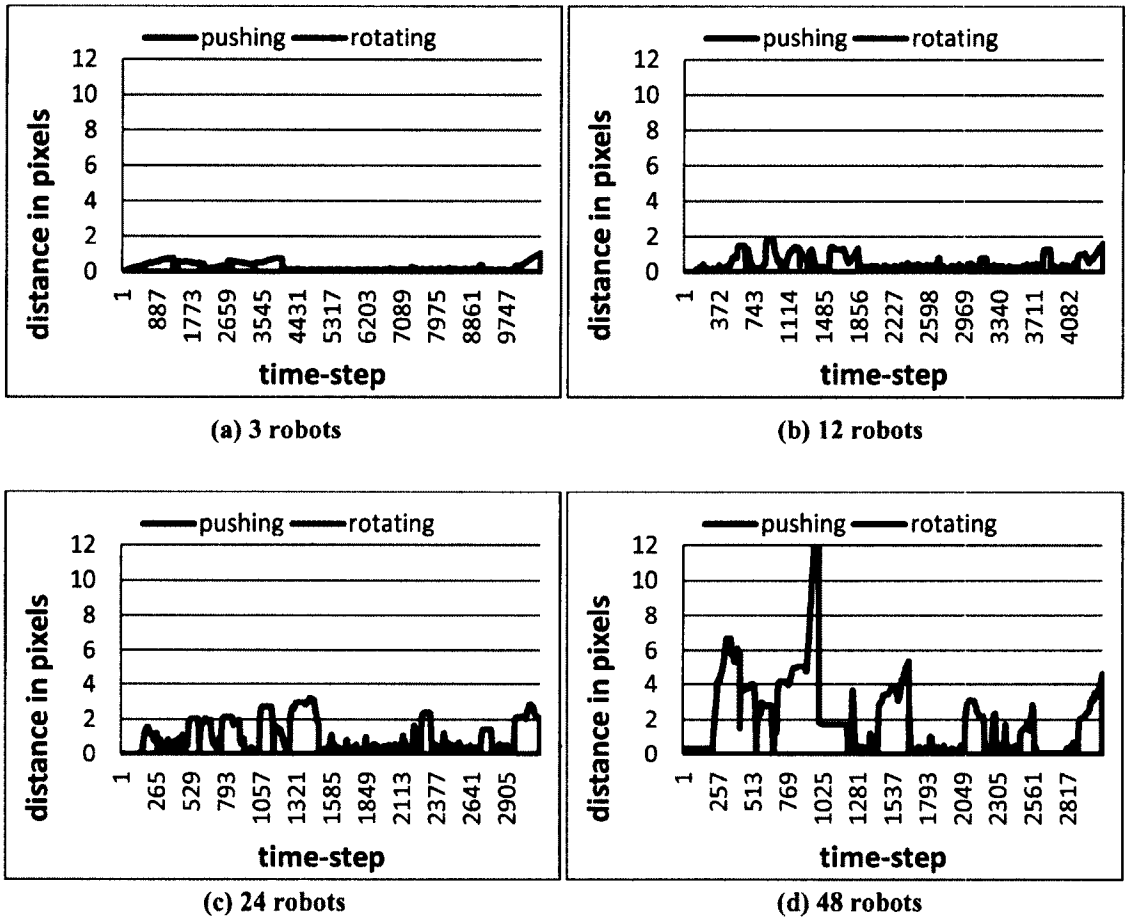


Figure 39: Deviation distance per time-step during experiment 2.

Number of collisions	3 robots	12 robots	24 robots	48 robots
Robot-to-Robot	12	434.50	495.83	727.48
Robot-to-M	6	10.33	10.96	56.17

Table 4: Number of collisions during the repositioning tasks for experiment 2.

Table 5 shows the average amount of time that each behavior was active during the simulation. In addition, the table shows the average waiting time for each team of robots. We can see from the table, that the average amount is somewhat consistent during the

simulation for each behavior with each robot spending 1/3 of its time waiting for other groups of robots to complete their rotating, re-alignment or pushing tasks.

Behaviors	**3 robots**	**12 robots**	**24 robots**	**48 robots**
Pushing	29.36%	28.86%	27.10%	25.83%
Rotating	7.91%	7.16%	6.61%	7.47%
Realignment	29.35%	28.78%	27.14%	26.98%
Reorientation	0.00%	0.98%	2.61%	2.35%
Repositioning	0.07%	0.19%	0.29%	0.47%
Waiting	33.31%	34.03%	36.25%	36.89%

Table 5: Average amount of time a behavior was active, for various team sizes during experiment 2.

5.3 Experiment 3 – Shape With Uneven Corner Angles

In this experiment M had unequal corner angles which were not equal to 90^o. In this case, the lines from C to the target-balance points are never perpendicular to the edges of M. The target-balance points are calculated based on the angle of the edges that intersect with the stabilizer tool axis (see Figure 40). The diameter of M is 120 pixels in this experiment.

Figure 41a shows the translations of M along P and Figure 41b shows the trace of the center point of M along P while being pushed by 24 robots. The trace shows reasonably accurate results. The defining characteristic of this experiment is that even though the shape of M is not a square (meaning the line from C to the target-balance points is not perpendicular to the edges on M), the operation showed minor deviations from the path P while pushing and rotating.

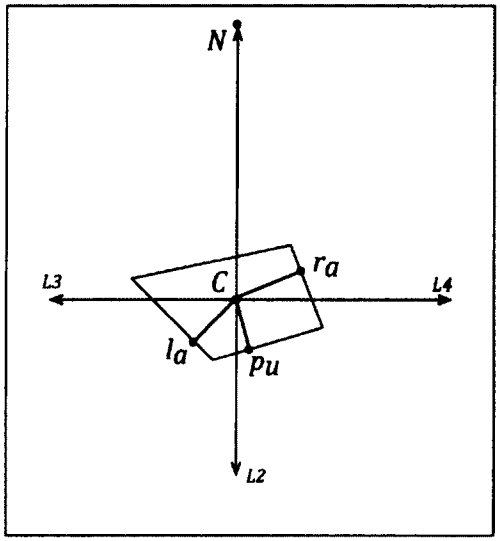
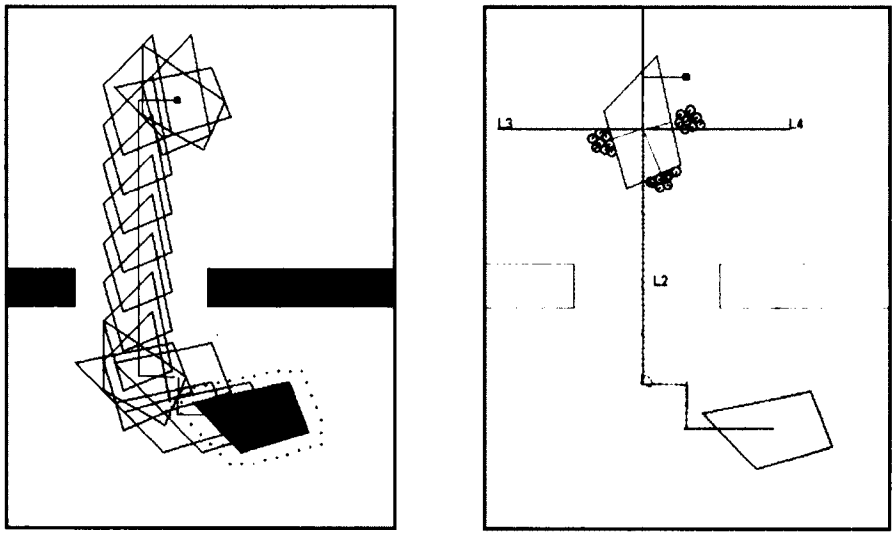


Figure 40: Uneven shape showing target-balance points being non-perpendicular to edges of M.



(a) (b)

Figure 41: (a) Translations of M along P. (b) Trace along P while M is being pushed by 24 robots.

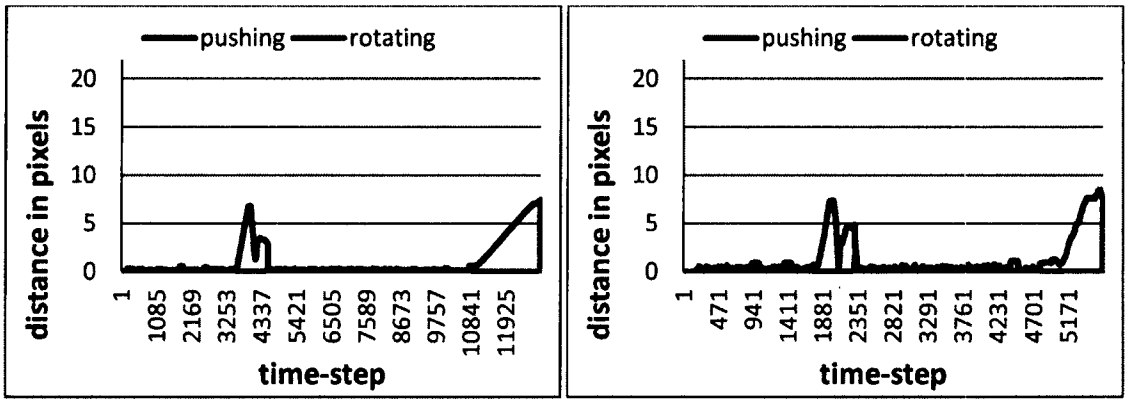
Table 6 shows the repositioning collisions during the experiment. Notice that when 48 robots were used, the interference between the robots and *M* was higher and that caused *M* to stray away from the path multiple times.

Number of collisions	**3 robots**	**12 robots**	**24 robots**	**48 robots**
Robot-to-Robot	19.00	417.00	860.71	2516.88
Robot-to-M	11.67	12.83	31.50	191.94

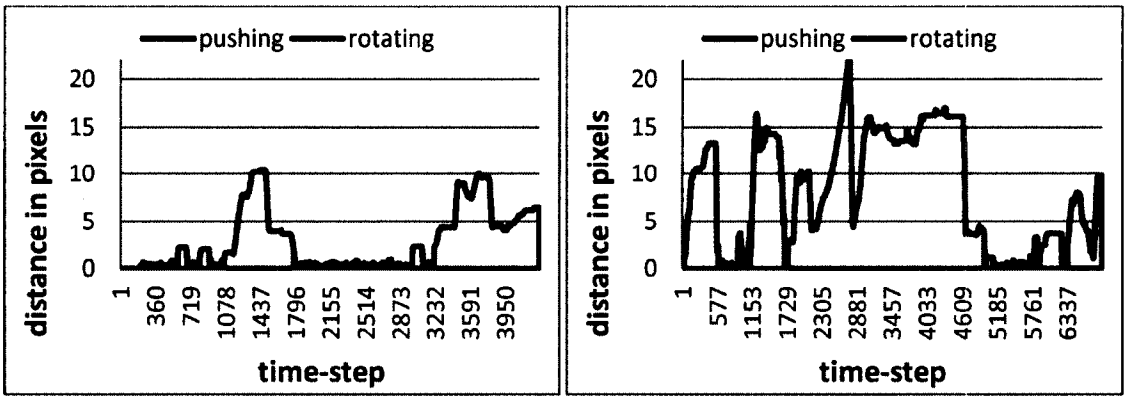
Table 6: Number of collisions during the repositioning tasks for experiment 3.

The charts in Figure 42 show the deviation distance during rotating and pushing tasks. Notice that the deviation distance during push tasks was between 0.4% and 2% of *M*'s diameter when using 3,12 and 24 robots but it grew to 14% when using 48 robots. During the rotation tasks, the deviation remained between around 6% and 9% for 3, 12 and 24 robots but then increased to 19% for 48 robots (see Figure 42 d). It is evident that the rotation tasks were not as stable as the previous experiments and that the shape of *M* plays an important role in performing rigorous movements.

Table 7 shows the average amount of time that each behavior was active during the simulation. Notice that as the number of robots increase, the amount of time spent pushing decreases, due to the increase in re-orientation, re-positioning and rotating behaviors. We can see again that the average time spent in each behavior is somewhat consistent.



(a) 3 robots (b) 12 robots



(c) 24 robots (d) 48 robots

Figure 42: Deviation distance per time-step during experiment 3.

Behaviors	3 robots	12 robots	24 robots	48 robots
Pushing	25.62%	26.20%	24.64%	22.64%
Rotating	15.35%	12.61%	15.50%	10.39%
Realignment	25.62%	26.20%	24.64%	23.61%
Reorientation	0.00%	0.82%	0.76%	7.51%
Repositioning	0.06%	0.15%	0.38%	0.97%
Waiting	33.34%	34.02%	34.07%	34.88%

Table 7: Average amount of time a behavior was active, for various team sizes during experiment 3.

5.4 Experiment 4 – Circular Shape

In this experiment, we used a near-circular shape for M with small edges that were not perpendicular to the stabilizer tool axis. The target balance points never coincide with the stabilizer tool axis. (see Figure 43). The diameter of M is the smallest enclosing circle around M which is 85 pixels. Figure 44a shows the translations of M along P and Figure 44b shows the trace of the center point of M along P while being pushed by 24 robots. The trace shows acceptable results where during the push tasks the deviation distance is again minimal (i.e., around 0.3% for a 3-robot team and 3.5% for a 48-robot team). Figure 45 shows the deviation distance for each time-step. Notice that there are no rotate tasks, since M did not need to be rotated for a solution.

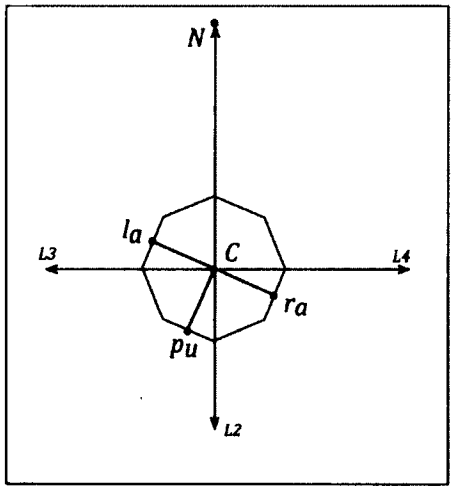
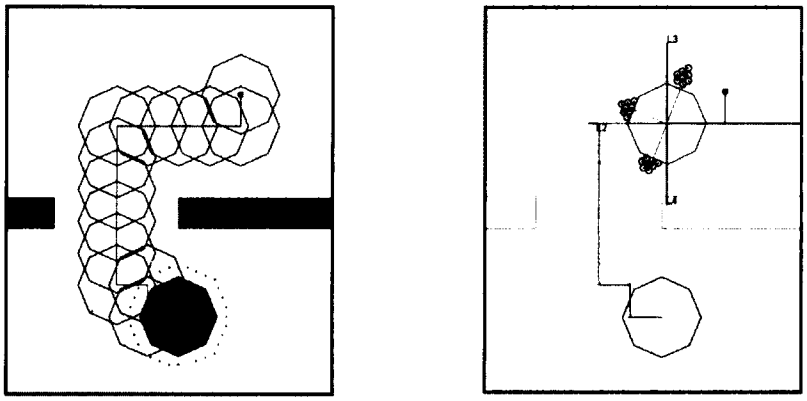
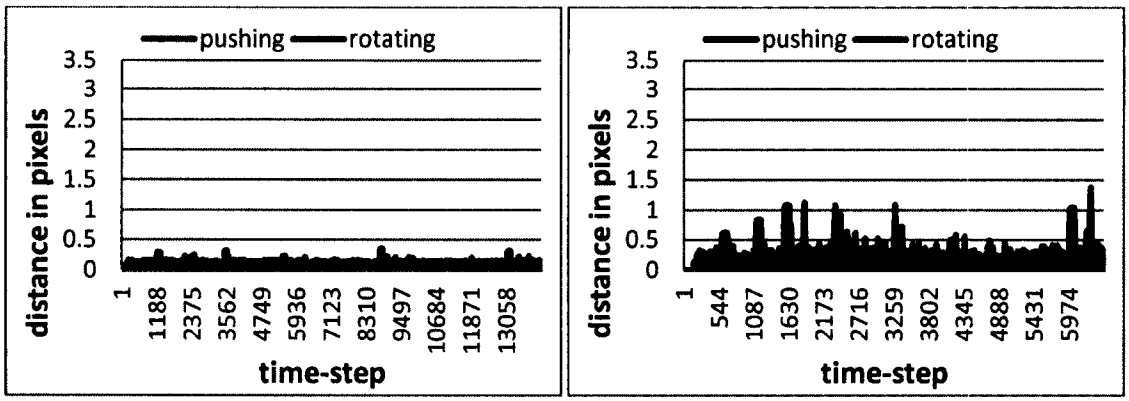


Figure 43: Circular shape with small edges showing that the target balance points do not coincide with the stabilizer tool axis.

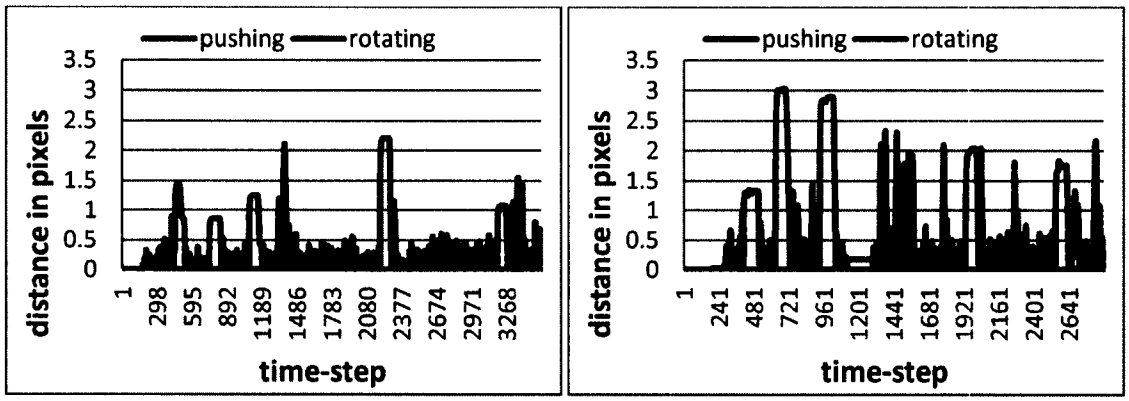


(a) (b)

Figure 44: (a) Translations of M along P. (b) The trace along P while M is being pushed by 24 robots.



(a) 3 robots (b) 12 robots



(c) 24 robots (d) 48 robots

Figure 45: Deviation distance per time-step during experiment 4.

Table 8 shows that the average number of collisions between the robots and *M* are much higher when we used 48 robots. This is reflected in Figure 45d where the deviation distance reaches 6. In contrast, when just 3 robots were used, the deviation distance did not even reach 0.3 pixels (see Figure 45a). Since there are no rotations, Table 8 shows zeros for Robot-to-*M* collisions for 3 and 12 robots. However, there are still chances for collisions during the repositioning tasks, and so when the number of robots increased to 24 and 48, there were many collisions once again.

Number of collisions	**3 robots**	**12 robots**	**24 robots**	**48 robots**
Robot-to-Robot	5	340	383.79	442.94
Robot-to-M	0	6.25	21.00	38.60

Table 8: Number of collisions during the repositioning tasks for experiment 4.

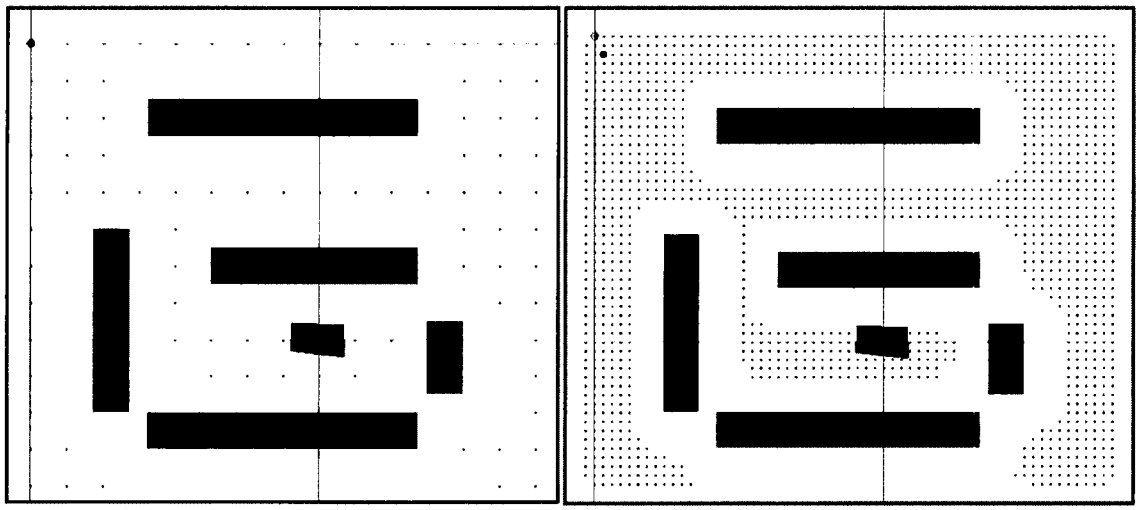
Table 9 shows the average amount of time that each behavior was active during the simulation. In this simulation, since *M* did not have to rotate, the rotating behavior showed a consistent zero value. In addition, when we used a team of 3 robots, the orientation of *M* was stable enough such as zero reorientations were required. However, the realignment activation decreased as the number of robots increased. That is due to less pushing tasks as we added more robot which causes *M* to reach its destination faster. Notice that the average percentage of waiting increased when we had a team of 48 robots. That is because more deviations and additional time-steps were required in order to push *M* back on track by one group of adjusters as the other group waits for the realign task to finish.

Behaviors	3 robots	12 robots	24 robots	48 robots
Pushing	33.32%	30.78%	30.31%	29.68%
Rotating	0.00%	0.00%	0.00%	0.00%
Realignment	33.32%	30.71%	30.61%	28.82%
Reorientation	0.00%	2.57%	2.79%	3.96%
Repositioning	0.04%	0.09%	0.16%	0.21%
Waiting	33.32%	35.85%	36.12%	37.33%

Table 9: Average amount of time a behavior was active, for various team sizes during experiment 4.

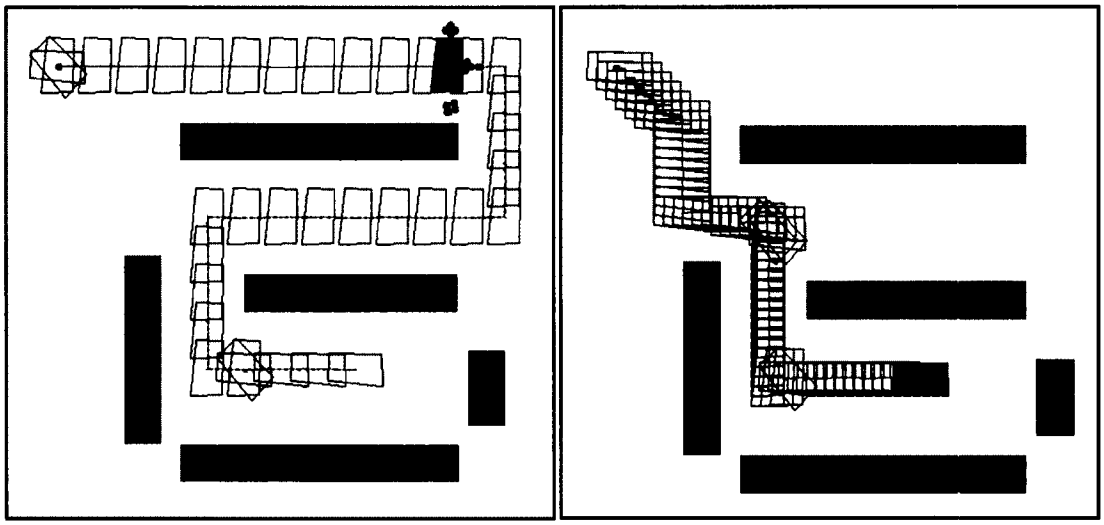
5.5 Experiment 5 - Effects of Changing Graph Size

The aim of this experiment was to show the effects of changing the edge length d in the layers of G and the rotation angle of M which can affect the calculation of P. In Figure 46a, we chose $d = 40$ and $a = 45$ and in Figure 46b, we chose $d = 10$ and $a = 10$. In Figure 47, we show the translations and rotations of M along P. By decreasing d, the generated path P took a shorter route. Notice that with smaller d, P required more rotations in order to pass in between the two obstacles at the top/left corner of the workspace. Figure 47b shows that when we used a smaller d, the deviation distance increased to almost 13 due to additional rotations at the end of the second segment of the path where many adjustments were required to accomplish a successful rotation.



(a) (b)

Figure 46: Graph vertices with (a) larger and (b) smaller values of d.



(a) (b)

Figure 47: Translation of M along P for (a) a large value of d and (b) a smaller value of d. The path takes a longer route in (a) due to the inability to rotate through the left side opening.

In Table 10, the number of collisions for the Robot-to-Robot and Robot-to-*M* was much higher when we used a smaller *d* since the path *P* has more rotation tasks and requires more repositioning around *M*. The numbers in this table reflect the unnecessary collisions with *M* during repositioning.

Number of collisions	**12 robots with longer path**	**12 robots with shorter path**
Robot-to-Robot	362.92	1038.08
Robot-to-M	9.58	27.50

Table 10: Number of collisions during the repositioning tasks using large and small values of *d*.

Figure 48 and Figure 49 show the deviation distance for each time-step until the completion of the task. Notice that the number total number of time-steps are less even though the path in Figure 49 is shorter. This is due to a number of rotation tasks in the shorter path. The time spent on readjustments at the rotation tasks much higher. Eventually, if the rotation accuracy can be further be enhanced then the time-steps would be decreased.

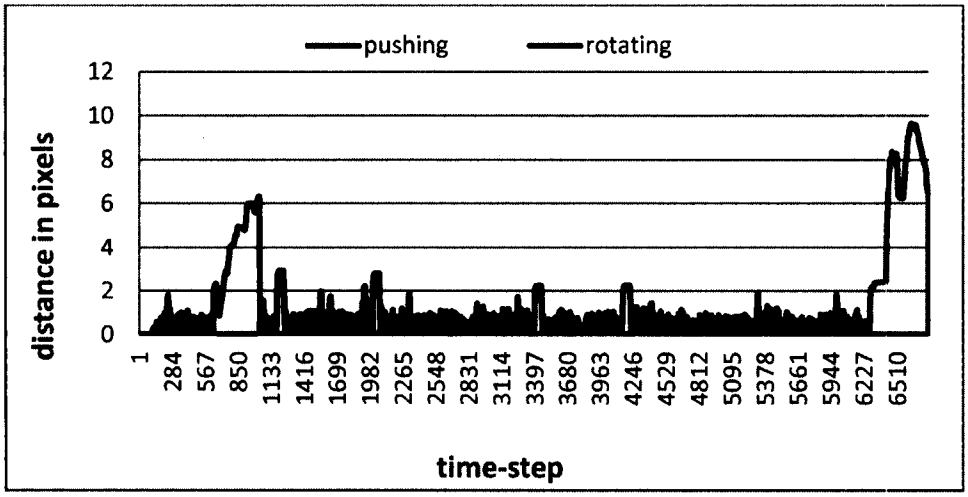


Figure 48: Deviation distance per time-step during experiment 5 for large value of *d*.

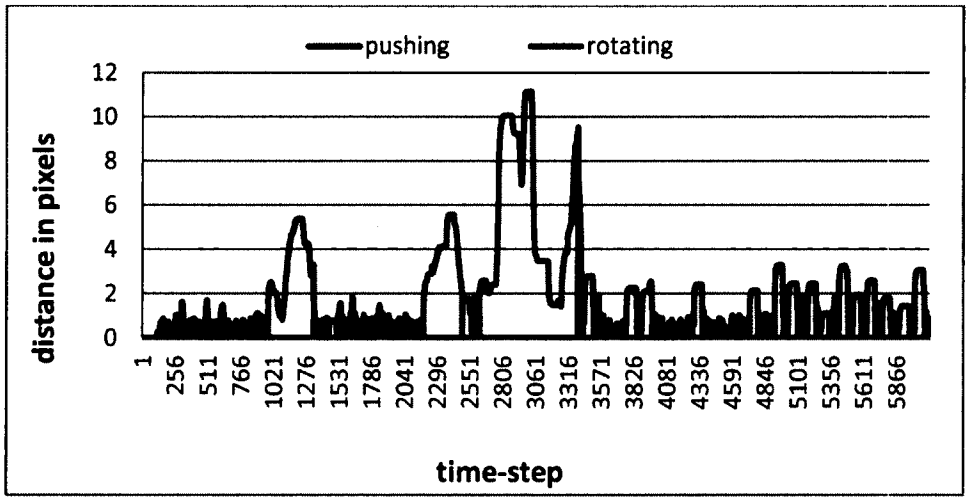


Figure 49: Deviation distance per time-step during experiment 5 for small value of d.

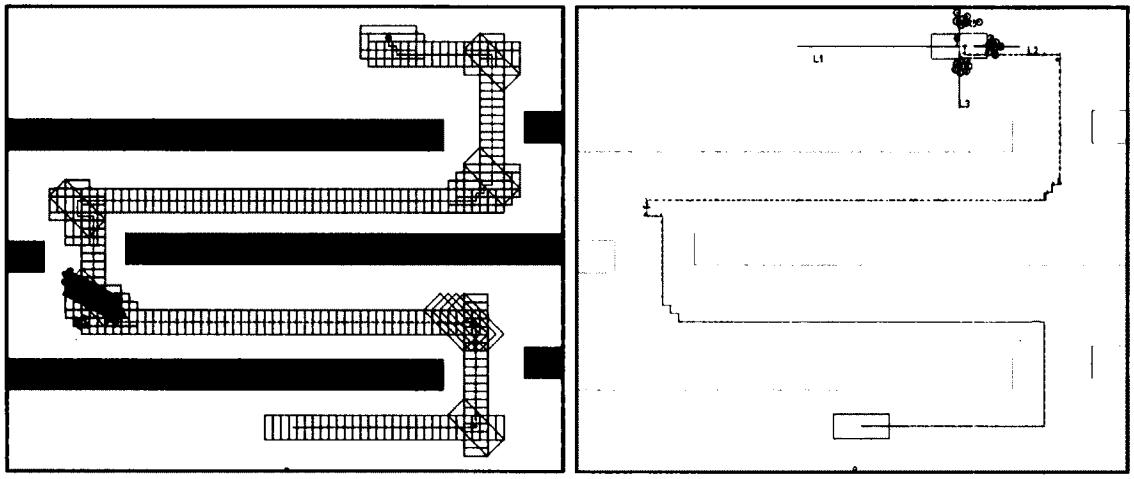
Notice that in Table 11, the number of realignments for the longer path is higher than the shorter path. That is caused by an increased number of push tasks which affects the number of realignments and reorientations. The percentage of realignments with the shorter path P is less due to a decreased number of pushes. In general, smaller paths require less realignments to complete the operation. Since there are more rotations in the shorter path, the table shows more waiting time, since during rotation the push robots pull back to their resting point.

Behaviors	**12 robots with longer path**	**12 robots with shorter path**
Pushing	27.84%	25.61%
Rotating	6.08%	9.13%
Realignment	27.77%	25.61%
Reorientation	2.37%	2.77%
Reposition	0.11%	0.53%
Waiting	35.83%	36.35%

Table 11: Average amount of time each behavior was active during experiment 5.

5.6 Experiment 6 – Effect of Multiple Rotations

We performed this experiment to show a number of reorientations while trying to move in a maze-like workspace. In addition, this simulation demonstrates *M* being push while its orientation is not perpendicular to the segment of the path *P*. Figure 50a shows the translations of *M* along *P* and Figure 50b shows the trace of the center point of *M* along *P* while being pushed by 24 robots. We can see that the robots succeeded in pushing through the maze using teams of 3, 12 and 24 robots. The simulation was not successful when we used more robots as the interference between the robots and *M* was very high which caused *M* to stray away from the path *P* and it never recovered. If a larger *M* dimension was used with the same size of robots then we would get better results for a team of 48 robots.

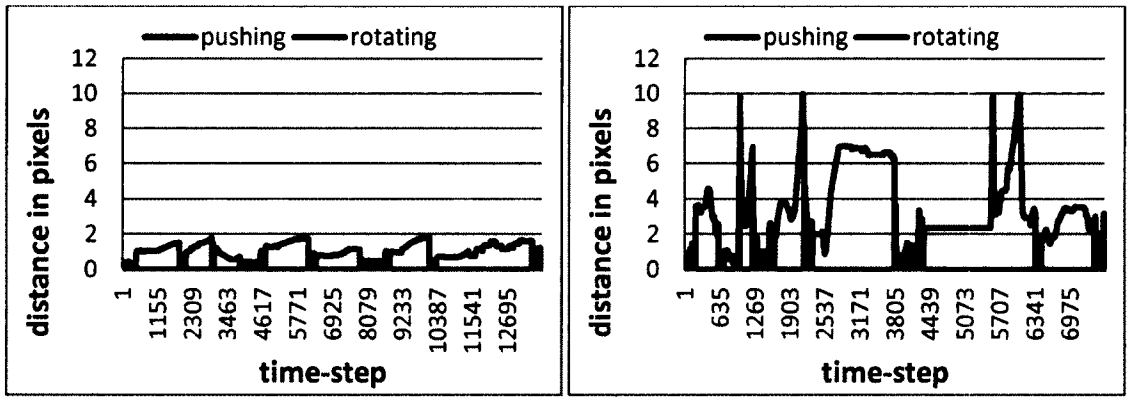


(a) (b)

Figure 50: (a) Translations of *M* along *P*. (b) Trace along *P* while *M* is being pushed by 24 robots.

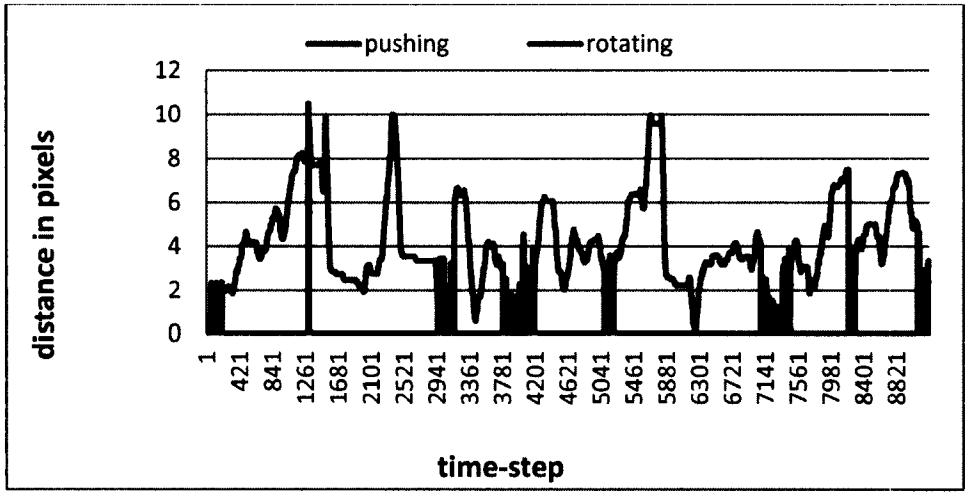
Figure 51a shows that the deviation distance with a team of the 3 robots was very low compared to the teams of 12 and 24 robots although using these team sizes any deviations were very quickly recovered (i.e., many spikes).

In Table 12, the number of collisions between the robots and M was much higher when we used a team of 24 robots, due to the fact the robot-to-robot collision is very high which causes the robots to unintentionally push very closely towards M. The number of collisions for 48 robots is not available as M was irrecoverable and was pushed far from the path P.



(a) 3 robots

(b) 12 robots



(c) 24 robots

Figure 51: Deviation distance per time-step during experiment 6.

Number of collisions	**3 robots**	**12 robots**	**24 robots**
Robot-to-Robot	6.67	1470.83	2472.71
Robot-to-*M*	1.06	23.38	57.29

Table 12: Number of collisions during the repositioning tasks for experiment 6.

Table 13 shows the average amount of time that each behavior was active during the simulation. In this simulation, when we used a team of 3 robots, the orientation of *M* was stable enough such as zero reorientations were required. Adding more robots, *M* slides faster and requires less pushing tasks which causes the number of realignments to be less. Also, in this experiment, the average percentage of waiting increased when we used a team of 48 robots. That is due to more deviations and additional time-steps to push *M* back on track by one group of adjusters as the other group waits for the realign task to finish.

Behaviors	**3 robots**	**12 robots**	**24 robots**
Pushing	26.83%	25.62%	22.21%
Rotating	13.55%	11.93%	14.11%
Realignment	26.34%	23.38%	22.64%
Reorientation	0.00%	3.15%	3.13%
Reposition	0.15%	0.28%	0.41%
Waiting	33.14%	35.65%	37.49%

Table 13: Average amount of time each behavior was active during experiment 6.

5.7 Experimental Consistency and Repeatability

In order to show the consistency and repeatability over multiple runs, we performed 10 runs on experiment 3 using teams of 3, 12 and 24 robots. We choose experiment 3 over other experiments as *M* has a "non-square" bounding box and could causing more

deviations during the run. Figure 52 and Figure 53 show the maximum deviation distance during push and rotate tasks, respectively, for the three teams of robots over 10 runs. We can see that for the push tasks, the deviation distance generally increases as the number of robots increases. The deviation distance was consistently minimal when we used a team of 3 robots whereas with more robots the deviation distance was less consistent throughout the runs. As we saw previously in experiment 3 that the number of robot-to-robot collisions and robots with increase as we added more robots which cause to deviate unintentionally from the path . The same type of deviation distance increase happened for the rotate tasks. We can see in Figure 53 that the deviation distance increased as we added more robots. Again, the unintentional collisions is the cause.

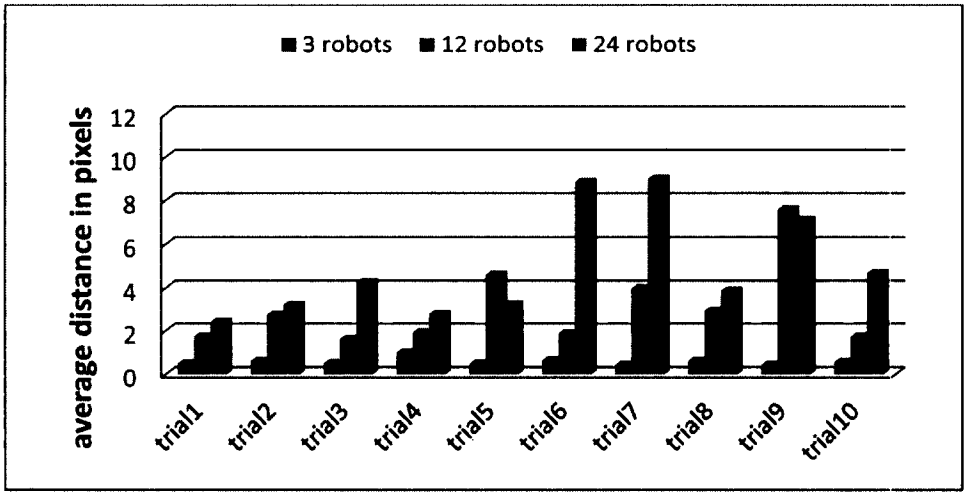


Figure 52: Maximum deviation distance during **pushing** tasks for teams of 3, 12 and 24 robots.

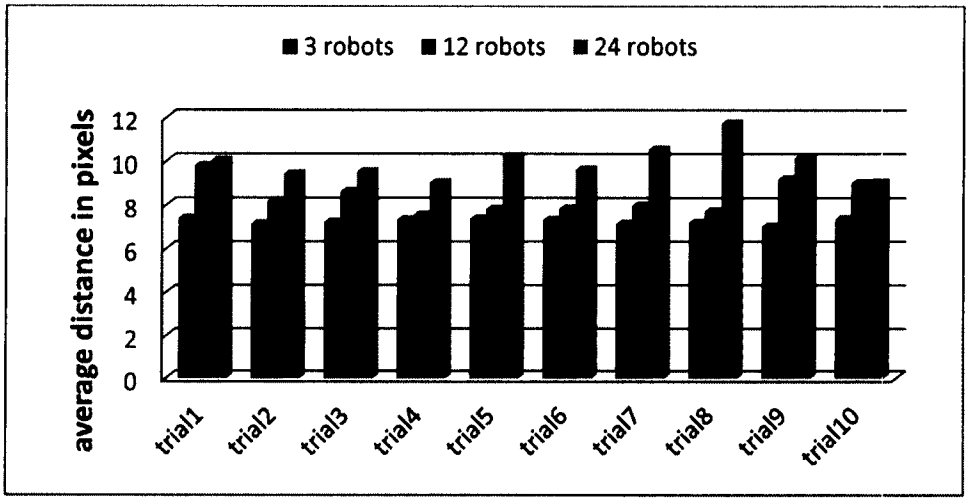


Figure 53: Maximum deviation distance during **rotating** tasks for teams of 3, 12 and 24 robots.

Figure 54 and Figure 55 show the maximum, average and standard deviation of distance for ten runs during push and rotate tasks, respectively. For each team of robots, the maximum deviation distance for each run was taken as a set, then the maximum value from this set was calculated. The maximum of these maximum values was calculated for each team. The same type of steps are applied to calculate the average and the standard deviation. When calculating the average distance, the average of the average is calculated and when calculating the standard deviation, the maximum of the maximum standard deviation is calculated. Notice that, on average, the deviation distance during the push tasks did not exceed 2.0 pixels. In our experiment it is 2% of the diameter of , which indicates rigorous movement. In addition, the average for rotate tasks was below 3.5 pixels which is only a few pixels away from the center of rotation. The graphs prove that the results did not vary much over the 10 runs of the same experiment.

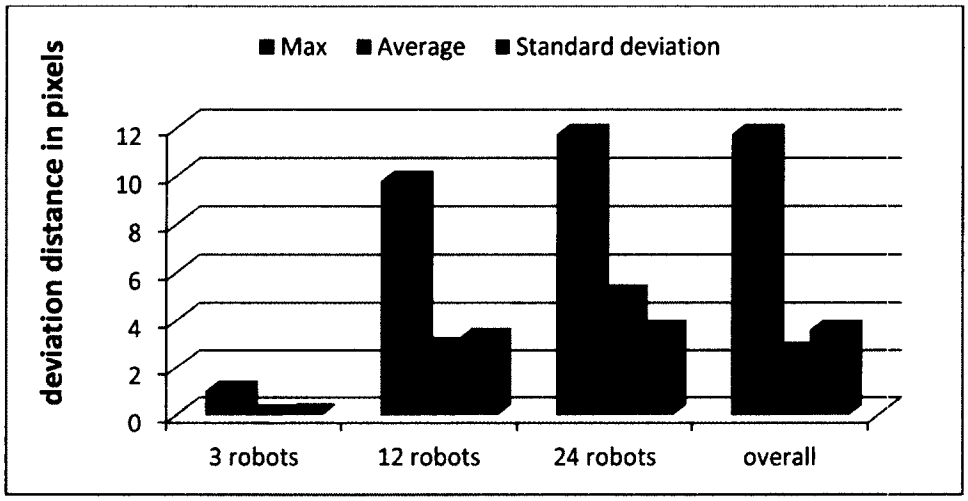


Figure 54: Maximum and average distances as well as standard deviation during **push** tasks for experiment 3 using 3, 12 and 24 robots.

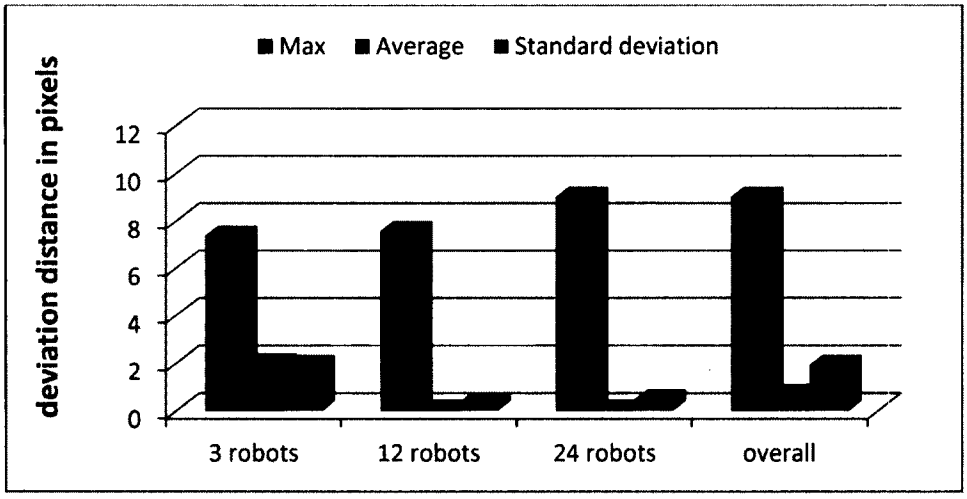


Figure 55: Maximum and average distances as well as standard deviation during **rotate** tasks for experiment 3 using 3, 12 and 24 robots.

5.8 Run-time Analysis

Figure 56 displays the amount of time required to push completely along P in each of the six experiments showing the effects of increasing the number of robots. In addition, the figure shows the average amount of time required for all the experiments combined, representing a typical scenario. In general, the figure shows that the average number of time-steps decrease when the number of robots increase due to the cumulative force from each of the robots. The time-steps are registered from the moment the robots leave their home location until reaches the destination location . The taxiing time-steps are insignificant and range from 50 to 70 time-steps per trial. Notice that in experiments 3 and 6, the number of time-steps for 24 and 48 robots increased to above the time-steps for the smaller team size runs. This is due to the increased interference between the robots and as they crowd around it which causes greater deviations and thus more time is needed to readjust to push it back on track.

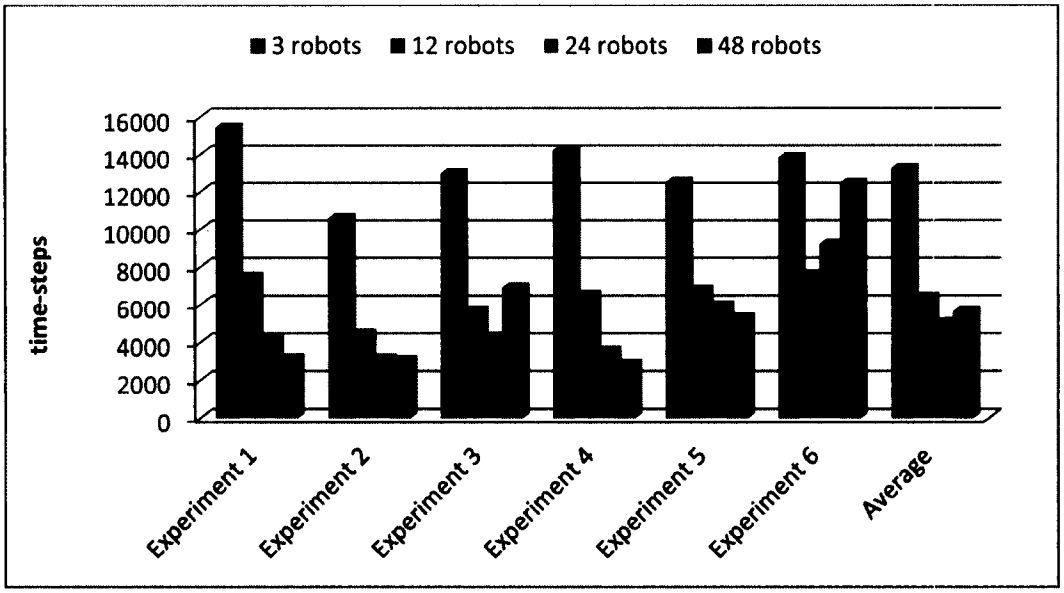


Figure 56: Running time for each experiment with team sizes of 3, 12, 24 and 48 robots. The average over all experiments is also shown.

Although the graph shows that time-steps decrease as robots are added, Figure 57 shows nowhere near *optimal* speedup. Since the algorithm requires a minimum of 3 robots, the graph shows speedup in terms of multiples of 3 robots. For example, with 48 robots, this is 16 times more robots than the 3-robot case. Hence, optimal speedup would be 16 times quicker when using 48 robots. The reasons for non-optimal speedup are clear. As has been shown through the experiments, on average, the robots do not make the most efficient use of their time since 1/3 of their time is spent in a "waiting state" resting until the other two groups have completed their rotation, re-alignment, etc.. Also, due to the increase in collisions as the number of robots increase, optimal speedup is unobtainable in practice. Lastly, as multiple robots push one another to combine their forces to push M, some of the force is absorbed, so each group of robots does not provide optimal force. This less-than-ideal speedup, is not of concern in this thesis, as the goal of the algorithm was not to move M faster but to push it rigorously. Additional robots are only used to provide more force when M is too heavy to push with just a few robots.

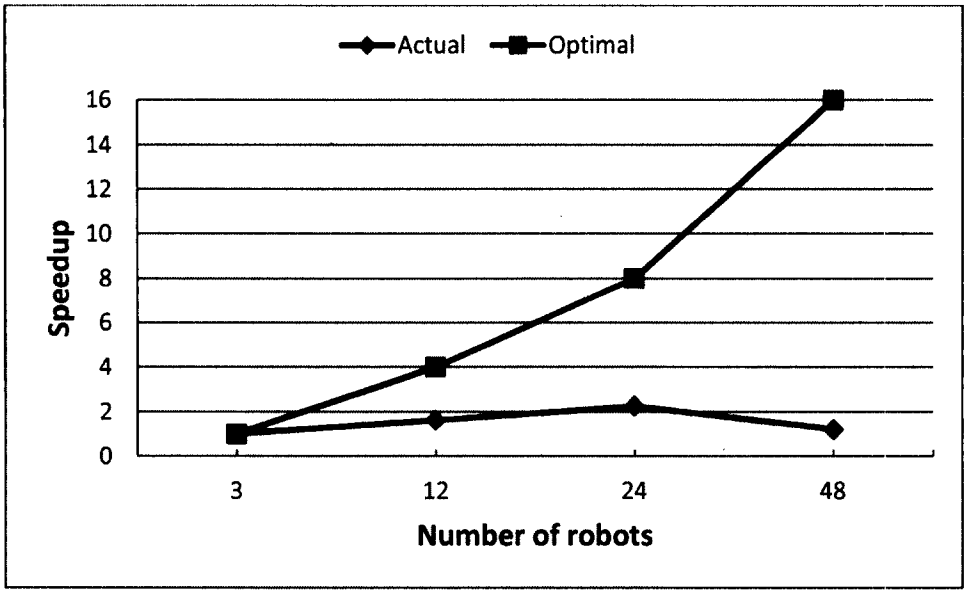


Figure 57: Typical speedup (with respect to 3 robots) as more robots are added.

5.9 Data Communication Load

The purpose of this experiment was to show the data communication between the robots and the *GC*. We performed the experiment with teams of 3, 12, 24 and 48 robots using the same workspace as in experiment 1, where we collected the average of the total data from all robots. The data packets sent and received are in units of bytes. A call to request a certain type of data from the *GC* requires 1 byte, a transmitted coordinate requires 2 bytes, a boolean value requires 1 byte and a transmitted angle requires 1 byte. Table 14 shows the amount of data sent and received between the robots and the *GC*. To get a better comparison of data communication amounts as the number of robots increase, it is necessary to determine how much communication is done at each time step, on average. The average number of bytes per time-step is calculated by multiplying the average amount of bytes sent and received from all the behaviors by the number of robots and then divide that by the number of time-steps for the simulation. The average amount of data sent and received between a team of 3 robots and the *GC* is 659319 bytes where the average number of bytes sent and received per time-step for total robots was 123.99 bytes. The whole operation was accomplished in 15952 time-steps. In addition, the average amount of data sent and received between a team of 48 robots and the *GC* is 114746.31 bytes where the average number of bytes sent and received per time-step for total robots was 1459.80 bytes. The whole operation was accomplished in 3773 time-steps. Notice that, as the number of robots decrease, the average number of bytes per time-step decreases. However, the average send/receive bytes per time-step per robot decreases as we add more robots (i.e., for 48 robots the average number of bytes per time-step per robot was 30.41 and for 3 robots it is 41.33). This is due to the fact during repositioning for example, some robots might exchange more data with the *GC* than other robots since it depends on their distance from the next target point they are required to reach. also, the number of points to follow on the repositioning path could vary from one robot to another. Notice that the behavior arbitration also sends and receives bytes from

the *GC*. This data communication is limited to the current task to be performed which is one byte for the request and one byte for the response.

Average send/Receive	**3 robots**		**12 robots**		**24 robots**		**48 robots**	
Behaviors	Send	Receive	Send	Receive	Send	Receive	Send	Receive
Pushing	26520	17680	10098	6732	5674.13	3782.75	3276.00	2184.00
Rotating	64752	47089	16677.75	12123.25	13582.75	9859.75	9219.38	6697.06
Realignment	70720	53040	26841	20130.75	14679.00	11009.25	8630.75	6473.06
Reorientation	89578	8986	35977.25	4818.5	20903.00	3127.63	13654.31	1692.50
Reposition	139074	47524	53990.25	19491.25	41233.50	15693.00	30160.69	11602.56
Behavior Arbitration	47178	47178	18729	18729	14361.00	14361.00	10578.00	10578.00
Average total in bytes per robot	437822	221497	162313.3	82024.75	110433.4	57833.38	75519.13	39227.19
Time-steps	15952		6307		4908		3773	
Average send/receive bytes per time-step per robot	41.33		38.74		34.28		30.41	
Average send/receive bytes per time-step for all robots	123.99		464.89		822.82		1459.80	

Table 14: Amount of data sent and received between the robots and the *GC*. The amount of average bytes per time-step is also shown.

Chapter 6

6 Conclusion

In this thesis, we presented a strategy for pushing, with rigor, a polygonal object M on a path P using multiple robots in a 2D environment with polygonal obstacles. Our strategy is based on previous research in three areas: path planning and configuration space as described by Lozano-Perez [20][21], the design of behavioral models as described by Brooks [13][14], and cooperative transport and swarm intelligence as described by Bonabeau et al. [15][39]. When compared to previous work, our work is unique in three ways. First, it focuses on pushing the object with rigor along a computed path that incorporates rigorous rotations and re-alignments. Second, it allows pushing and rotating of arbitrary convex polygons and can be easily extended to non-convex polygons as well, whereas previous work only attempted the pushing of a box of rectangular shape on a trajectory [1][2][3][4][5][6][7][8][9] [11][35][44]. Our solution can even be applied to curved convex polygonal objects. Third, in previous work, the focus was on transporting an object to a destination without giving enough attention to rigorous movement. Our algorithm also has the capability of adjusting the robot-pushing strategy so that the robots "get back on track" when the push object begins to deviate from the desired path.

In chapter 3, we presented a path planning algorithm in which we described the C_{free} set of non-overlapping configurations of M with the obstacles in W. In addition, we created a graph G that represented the configuration space which was used to calculate a rectilinear shortest path approximation P. The graph consists of a set of layers L_i where

each layer contains edges of length d (which is an adjustable parameter). A layer represents a rotation of M by a certain angle α_l (also an adjustable parameter) where $1 \leq l \leq 360$. The layers in G are interconnected by edges with weights higher than edge weights within the same layer so as to discourage rotations whenever possible. The length of d affects the likelihood of finding a path P in G from s to t.

In chapter 4, we presented our algorithm for pushing M with multiple robots, along the path P that is provided by the path planner in chapter 3. Our design consists of a hybrid system that is based on a global controller and a multi-robot behavior-based system, designed to alleviate the processing load from the robots and supply increased rigorous motion. Previous research has shown the efficiency of such behavior-based systems which concentrate on the movement of an object (traditionally a box shape) along a small trajectory to a destination represented by an area or a light bulb. Our design is empowered by a *Global Controller* (GC) which acts as a global satellite looking over the motion of M as well as the locations of the robots within the workspace W. The path P is broken into a set of tasks for the robots to perform, where the tasks are managed by a *Path Monitor*. There are three types of tasks (i.e., *push*, *rotate*, and *reposition*) as well as two sub-tasks (i.e., *reorient* and *realign*). Our design shows that these tasks are sufficient to push M along P with rigor. Our solution assigns one of three specialties to each of the robots resulting in three groups: pushers, left-adjusters, and right-adjusters. Each of the robots has a set of behaviors that control its motion based on its location and the task that it needs to perform. We designed a *Stabilizer Tool* data structure which adjusts itself based on the dimensions of M. It provides three things: (1) target points around the boundary of M for the robots to move towards, (2) the orientation of M and (3) the stabilizer tool axis angles. The data communication between the robots and the GC is limited to these three types of data as well as the current task type. Our design is sufficient for pushing a convex polygonal shape but we believe that with certain adjustments, it can be applied to any shape, even with curved edges (see section 6.1).

We ran various experiments that showed that our design is capable of pushing M along P with minor deviation. The results (shown in chapter 5) indicate that the deviation of M from P remains small since any large deviations were prevented by the realign and reorient behaviors. In our experiments, we found that the motion of M is more stable when the number of robots around M is reasonable where they can maneuver freely inside the buffer zone. The results show that the number of collisions between the robots and M becomes higher as the number of robots increases. That produces higher interference with M and as a consequence can cause it to stray away from P. The experiments provided analysis on the data communication between the robots and the GC. We found that as the number of robots increases, the data sending and receiving between the robots and the GC decreases per robot. This is due to the fact that adding more robots, the cumulative force of the robots against M increases and causes it slide faster and reach its destination in less time-steps. In addition, the experiments presented analysis on the average number of time-steps needed to complete the task. The results showed that M indeed moves faster as more robots are added however, the speedup is not optimal. None of the previous work that we encountered provided data related to deviation distance of M for the path P and the data communication between the robots and their centralized controller. For that reason, we did not provide a quantitative comparison to previous work.

6.1 Future Work

The generated graph G can be enhanced further to minimize the area of examination on the workspace. In section 3.1, the C_{free} set could contain configurations that are not of any help when generating P. Visually, these configurations represent empty space in W, and the generated path P would never pass through these spaces.

Therefore, the graph G would contain vertices that are never used, thereby increasing storage space and runtime complexity with no benefit. The graph G can be enhanced to avoid calculations on these vertices.

Another area of future work is to handle curved-shaped push objects. Our current implementation works only for convex polygons. However, the stabilizer tool design is flexible enough to work with any object shape including objects with curved edges (see Figure 58). In the current design, M has straight edges and the axis of the stabilizer tool intersects these edges to provide the target-balance points around the boundary of M. Recall that these target points are calculated based on the angle that these edges make with the axis of the stabilizer tool (see section 4.2). We believe that same strategy can be applied if M has curved edges, but with minor changes in how the stabilizer tool finds the target-balance points. As shown in Figure 58, the tangent line angles at the points where the stabilizer axis intersects with the curves will play a role in finding the best placement of the target-balance to the left or to the right of the intersection point. The center C of M can be projected onto the tangent line creating the required target-balance point.

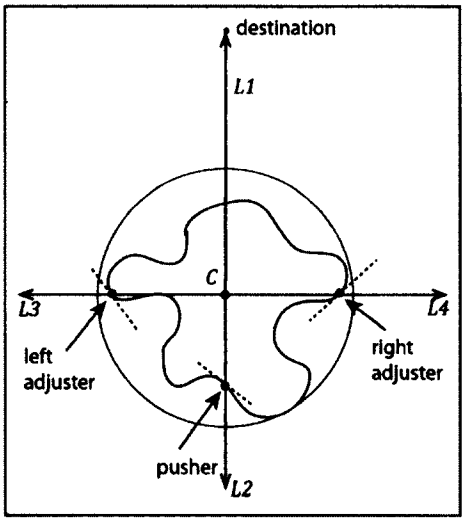


Figure 58: Curve-shaped object with proposed balanced points for pushing and adjusting.

Additional ideas for future work could be to enhance the robots maneuvering behaviors by spreading out the robots along the boundary of M. This can be accomplished by increasing the number of target-balance points so as to minimize collisions between the robots and M. In regards to data load, an improvement can be made by implementing a mechanism to enable robots within the same team to share the same information so as to reduce the data communication between the robots and the GC.

The simulation sometimes fails to keep M from deviating too far off the path P. However, we believe that such failures can be prevented with further adjustments to ensure rigorous movements. For instance, if the angular or linear damping are set very low (i.e., close to zero) then M would appear to float, where a simple push by the robots could cause it to slide too far from a goal point. This follows Newton's law of physics, whereby an object can move forever at constant speed if no resisting force is applied to the object. Therefore, to control the motions of M in such a situation, we can add a feature where some robots are always tasked to apply force on M from the opposite direction of where other robots are pushing. Additionally, the simulation can fail if simultaneously a *pusher* robot pushes from one side of M and an *adjuster* robot adjusts from the opposite side, potentially causing a stagnation where M moves a step forward, followed by a step backward. It is likely that if the pushers apply more force in such a situation or for the adjuster robots to adjust less frequently, then this can be prevented, provided that the situation can be detected or predicted beforehand.

A final area of future work is to allow robots switch specialties on-the-fly, possibly decreasing the minimum number of robots to two or even one by reusing a robot to perform multiple tasks in sequence, such as pushing, adjusting, and rotating. This could reduce the repositioning requirements of the robots and as a result decrease the number of unnecessary collisions with M. Currently on average, each robot spends one third of its time in a resting or waiting state. Allowing robots to switch specialties can cause them to

make better use of their time, providing a more efficient solution with a reduced overall completion time and hence better speedup. Of course, this will increase an individual robot's workload which could have adverse affect of a robot's energy resources and potential for overheating.

7 References

[1] Trojanek, P., Szynkiewicz, W., and Zieliński, C. Definition and composition of individual robot behaviours in cooperative box pushing. In Proceedings of the 13th IEEE IFAC International Conference on Methods and Models in Automation and Robotics. Technical University of Szczecin. 29-30. (2007).

[2] Parra González, E.F., Ramírez-torres, J., and Toscano-Pulid, G. A New Object Path Planner for the Box Pushing Problem. Electronics, Robotics and Automotive Mechanics Conference. 119-124. (2009).

[3] Parra González, E.F. Ramírez-torres, J., and Toscano-Pulid, G. Motion Planning for Cooperative Multi-robot Box-Pushing Problem. Advances in Artificial Intelligence, Iberamia. (2008).

[4] Chen, X., and Li, Y. Modeling and simulation of a swarm of robots for box-pushing task. 12th Mediterranean Conference on Control and Automation, Kusadasi, Aydin, Turkey. (2004).

[5] Yamada, S., and Saito, J. Adaptive action selection without explicit communication for multi-robot box-pushing. Systems, Man and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 31(3). 398–404. (2001).

[6] Lewis, M.A., and Tan, K.H. High Precision Formation Control of Mobile Robots Using Virtual Structures. Journal Autonomous Robots archive, Volume 4 Issue 4. (1997).

[7] Liu, S., Liu, F. and Tang, F. Cooperative transport strategy for formation control of multiple mobile robots. Journal of Zhejiang University, Science C, volume 11, 1-13. (2010)

[8] Gerkey, B.P., and Mataric, M.J. Pusher-Watcher: An Approach to Fault-Tolerant Tightly-Coupled Robot Coordination. In Proceedings of IEEE International Conference on Robotics and Automation, ICRA, Volume 1, 464-469. (2002).

[9] Adouane, L., and Le Fort-Piat, N. Methodology For Parameters Optimization Of Hybrid Architecture Of Control. In Proceedings of the 16th World Congress of the International Federation of Automatic Control, Prague, Czech Republic. (2005).

[10] Deneubourg, J.L., Dorigo, M., and Labella, T.H. Self-Organised Task Allocation in a Group of Robots. In Proceedings of the 6th International Symposium on Distributed Autonomous Robotic Systems, Tokyo, Japan. (2004).

[11] Karigiannis, J.N., Rekatsinas, T.I., and Tzafestas, C.S. Fuzzy Rule Based Neuro-Dynamic Programming for Mobile Robot Skill Acquisition on the basis of a Nested Multi-Agent Architecture. 2010 IEEE International Conference on Robotics and Biomimetics - RoBio. (2010).

[12] Payton, D.W. and Rosenblatt, J. A Fine-Grained Alternative to the Subsumption Architecture for Mobile Robot Control. In Proceedings of the IEEE/INNS International Joint Conference on Neural Networks, Washington DC, Volume 2, 317-324. (1989).

[13] Brooks, R.A. A robot that walks: Emergent behaviors from a carefully evolved network. In Beer, R., et. al. (eds.), Biological Neural Networks in Invertebrate Neuroethology and Robotics. Academic Press. (1993).

[14] Brooks, R.A. A Robust Layered Control System for a Mobile Robot, IEEE Journal of Robotics and Automation, Volume 1 RA-2, 14-23. (1986).

[15] Bonabeau, E, and Kube, C. Cooperative transport by ants and robots. Journal of Robotics and Autonomous Systems, Volume 30, 85–101. (2000).

[16] Togelius, J. Evolution of a Subsumption Architecture Neurocontroller. Journal of Intelligent and Fuzzy Systems Volume 15-1, 15-20. (2004).

[17] Jacobsen, C.L., Jadud, M.C., and Simpson, J. Mobile Robot Control The Subsumption Architecture and occam-pi. In Proceedings of the Communicating Process Architectures, 225-236. (2006).

[18] Igarashi, T., Inami, M., and Kamiyama, Y. A dipole field for object delivery by pushing on a flat surface, In Proceedings of 2010 IEEE International Conference on Robotics and Automation, 5114–5119. (2010).

[19] LaValle, S.M. Planning Algorithms. University of Illinois, *The Configuration Space*, Chapter 4. (2006).

[20] Lozano-Perez, T., and Wesley, M.A. An algorithm for planning collision-free paths among polyhedral obstacles. Communication of ACM Volume 22, 560-570. (1979).

[21] Lozano-Perez, T. Spatial Planning: A Configuration Space Approach. IEEE Transactions on Computers, Volume C-32-2, 108-120. (1983).

[22] Brooks, R.A. Solving the Find-Path problem by representing free space as generalized cones. M.I.T. Artificial Intelligence Lab., Rep. AIM-674. (1982).

[23] Brooks, R.A. and Lozano-Perez, T. A subdivision algorithm in configuration space for Find path with rotation. M.I.T. Artificial Intelligence. Lab., Rep. AIM-684. (1982).

[24] Liu, G.F., Milgram, R.J., and Trinkle, J.C. Toward Complete Path Planning for Planar 3R-Manipulators Among Point Obstacles. In Algorithmic Foundations of Robotics VI, STAR 17, Springer-Verlag, 329-344. (2005).

[25] Schwartz, J.T., and Sharir, M. On the piano movers II. General techniques for computing topological properties on real algebraic manifolds. Advances in Applied Mathematics, Volume 4, 298-351. (1983).

[26] Canny, J.F. The complexity of robot motion planning. Institute of Technology Cambridge, Massachusetts, MIT Press. (1988).

[27] De Berg, M., and Gerrits, D.H.P. Computing push plans for disk-shaped robots. In Proceedings of 2010 IEEE International Conference on Robotics and Automation - ICRA, 4487-4492. (2010).

[28] Catto, E. Box2D Physics Engine, April 2011, http://www.box2d.org/. (2012)

[29] JBox2D: Java Physics Engine, April 2011, http://www.jbox2d.org/. (2012)

[30] Rabin, S. Introduction to Game Development. Second Edition, Course Technology PTR, *Introduction to Numerical Physics Simulation*, Chapter 4.3. (2009)

[31] Shabana, A.A. Computational Dynamics. Third Edition, Numerical Integration Schemes, Chapter 14. (2010).

[32] Dobre, A., and Ramtal, D. The Essential Guide to Physics for Flash Games - Animation and Simulations. Apress. Chapter 5,11,14. (2011).

[33] Ericson, C., and Kaufmann, M. Real-Time Collision Detection. Chapter 4-7. (2005).

[34] Kube, C. R., and Zhang, H. Collective Robotics: From Social Insects to Robots. Adaptive Behavior, Volume 2, 189-218. (1993).

[35] Wang, Z., and Kumar, V. Object closure and manipulation by multiple cooperating mobile robots. In Proceedings of IEEE International Conference on Robotics and Automation - ICRA, Volume 1, 394–399. (2002).

[36] Drogoul, A., and Melendez, A.M. Analyzing multi-robot box-pushing. Advances in Science Computation, Memory Workshops 5th International Meeting of Computation. University of Colima, Volume 1, 530-539. (2004).

[37] Neumann, Erik, myphysicslab, June 2012, http://www.myphysicslab.com/. (2012).

[38] chipmunk, June 2012, http://chipmunk-physics.net/. (2012).

[39] Bonabeau, E., Dorigo, M., Guy Theraulaz, G. Swarm Intelligence: From Natural to Artificial Systems. Oxford. (1999)

[40] Kube, R.C., and Zhang, H. Stagnation Recovery Behaviors for Collective Robotics. In Proceedings 1994 IEEE/RSJ/GI International Conference on Intelligent Robots and Systems, Los Alamitos, 1883-1890. (1995).

[41] Hales, J.A. Comparing the performance of heterogeneous and homogeneous swarms. Mississippi State University, 78-79. (2008).

[42] Quinn, M. The Evolutionary Design of Controllers for Minimally-Equipped Homogeneous Multi-Robot Systems. University of Sussex. (2004).

[43] Braitenberg, V. Vehicles, Experiments in Synthetic Psychology. MIT Press, Cambridge, MA, USA. (1986).

[44] Reif, J. H. Complexity of the mover's problem and generalizations. Annual IEEE Symposium on Foundations of Computer Science, 421–427. (1979).

[45] Anderson, C., and Bartholdi, J.J. Centralized versus decentralized control in manufacturing: Lessons from social insects. In Complexity and Complex Systems

in Industry (I.P. McCarthy and T. Rakotobe-Joel, Eds.), University of Warwick, UK, 92-105. (2000).

[46] Physics processing unit, February 2012, http://en.wikipedia.org/wiki/Physics_processing_unit. Wikipedia. (2012).

[47] Goldberg, D., and Mataric, M.J. Interference as a tool for designing and evaluating multi-robot controllers. In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97), MIT Press, Cambridge, MA, USA, 637–642. (1997).

[48] Bellman, R.E. On a Routing Problem. Quart. Applied Math, Volume 16, 87-90. (1958).

[49] Dijkstra, E.W. A Note on Two Problems in Connection with Graphs. Number Math, Volume 1, 269-271. (1959)

[50] Floyd, R.W. Algorithm 97. Communications of the ACM 5-6, 345. (1962).

[51] Reynolds, C.W. Steering Behaviors For Autonomous Characters. In the proceedings of Game Developers Conference 1999 held in San Jose, California. Miller Freeman Game Group, San Francisco, California, 763-782. (1999).

[52] De Berg, M., Cheong, O., Kreveld, M.V., and Overmars, M. Computational Geometry: Algorithms and Applications. 3rd Edition, Chapter 12, 13, 14. (2008).