DECENTRALIZED AND COMPLETE MULTI-ROBOT MOTION PLANNING IN CONFINED SPACES

Dexter R. R. Scobee '12 and Adam T. Wiktor '12
Advisor: Christopher M. Clark

Submitted to the
Department of Mechanical and Aerospace Engineering
Princeton University
in partial fulfillment of the requirements
of Undergraduate Independent Work

Final Report

May 3, 2012

Reader: Naomi Leonard
MAE 440
78 pages
Color Images
Advisor Copy

To the Indestructible

Acknowledgements

First and foremost, we would like to thank our advisor, Christopher Clark, for helping us throughout the process and up until the very last minute. He managed to achieve the difficult balance of steering us towards a challenging, interesting problem that was actually solvable in the time frame available to us. We are extremely grateful for his support. We would also like to thank the Mechanical and Aerospace Engineering department for all we have learned, and special thanks to Jo Ann Love, without whom not a single MAE would graduate.

Adam would like to thank his parents, Stefan and Terry, for supporting him throughout his undergraduate years. They have provided him with an incredible amount of love and support, and have never failed to be there whether he needed someone to talk to or just wanted to put off working. He also appreciates his friends at Princeton for offering him a much-needed distraction from academics. Finally, he'd like to thank Dexter for making the senior thesis process infinitely more enjoyable. The late-night thesis sessions spent in Charter will always remain one of his defining memories of Princeton.

Dexter would like to thank his parents, Richard and Teresa, whose love and support throughout his entire lifetime have made it possible for him to realize his dreams. He thanks his sister, Cristi, whose unrelenting enthusiasm and infectious disposition never cease to bring a smile to his face, even on the cloudiest days. He would be remiss if he did not thank his girlfriend, Ilina, who, in her limitless patience, has been a source of peace and tranquility throughout his entire Princeton career. If it were possible, he would enumerate all of the friends and family who have helped him along the way. Thank you, all.

Finally, Dexter would like to thank Adam for accompanying him in this endeavor. His partnership has made the many long hours spent on this project more than enjoyable, and there is no one with whom he would rather swap bots.

Abstract

This paper presents the Push-Swap-Wait algorithm, a decentralized and complete approach for multi-robot motion planning in confined spaces. The algorithm builds upon a "push and swap" paradigm that has been used effectively in centralized navigation. This push and swap approach was expanded to apply to decentralized planning by adding a waiting mode to handle situations in which communication between robots is lost.

A proof is presented that guarantees the completeness of the Push-Swap-Wait algorithm in cases where the environment can be modeled as a tree T for which the number of leaf nodes is greater than the number of robots navigating through it. The algorithm also relies on the formation of ad-hoc communication networks among robots, such that robots can share information with a subset of other robots in the tree.

Finally, the algorithm is implemented in MATLAB to test its efficacy in a simulated environment populated with virtual robots. In systems of up to 30 robots navigating a randomly generated 10x10 graph, each simulated robot performs on average only one to two swaps before all robots reach their goal states. The algorithm was also found to have a time complexity of $O(R^2)$, indicating that this algorithm is well suited for scaling to large systems of robots.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

$\delta(n,t)$	Set of nodes adjacent to node n at time t that are available for a pushed robot
δ_L	Lowest priority node adjacent to node n at time t that is available for a pushed robot
γ	Twig node
$\Gamma(b)$	Set of all twig nodes of branch node b
γ_{end}	First node on the path from branch node b to the swapping robots at the time the swap is initialized
$\hat{\pi}(r,t)$	Planned next node of r at time t
$\overline{r}^*$	Higher priority robot of the two swapping robots $\overline{r}^*$ and $\underline{r}^*$
$\Phi(r)$, $\Phi(n)$	Priority of robot r or node n
Π	Sequence of moves
$\pi(r,t)$	Change in position of robot r at time t
ρ	Radius of communication
$\underline{r}^*$	Lower priority robot of the two swapping robots $\overline{r}^*$ and $\underline{r}^*$
ε	Set of all edges in the tree
$A(r,t)$	Assignment (position of robot r at time t)
B	Set of all branch nodes
b	Branch node
$C(r)$	Set of all robots in the communication network of r

$c(r)$	Set of all robots in direct communication with r
E	Set of all edges in the graph
e	Individual edge
G	Graph
$g(r)$	Goal node of robot r
L	Set of all leaf nodes
l	Leaf node
N	Set of all nodes
n	Individual node
$P(n)$	Set of all ancestors of node n
R	Set of all robots
r	Individual robot
r^*	Highest priority unsolved robot
r_{follow}	Swapping robot that begins farther from the target branch node b
r_{leader}	Swapping robot that begins closer to the target branch node b
$s(n_a, n_b)$	First node on the path from n_a to n_b
$S_{a,b}$	Path (set of nodes) leading from a to b
T	Spanning tree
t	Time

Chapter 1

Introduction

1.1 Background

Robotics holds the potential to solve many practical problems in everyday life that would otherwise require intensive human effort, but in order to fully realize this goal, the robots must be able to make decisions and move automatically without human intervention. Planning the motion of even a single robot can be quite complicated as movements become more intricate, environments change over time, and measurement uncertainties become significant [1]. When multiple robots are involved in a system and must either avoid interfering with one another or actively collaborate, the problem becomes harder still. The field of multi-robot motion planning has many direct applications to real-world problems. If driverless vehicles ever become common, for example, they will need to be able to interact and react to a dynamically changing environment [2]. Unmanned aerial vehicles (UAVs) could likewise benefit by coordinating their actions to accomplish missions using less complicated and expensive systems than a vehicle performing the same task alone [3]. Farther in the future, teams of rovers on other planets might need to work together to explore the surroundings, collect data, or build structures as precursors to manned exploration [4].

In general, algorithms to plan the motion of groups of robots can be categorized as either a centralized control architecture, in which a single computer controls all robotic agents, or a decentralized architecture in which each robot calculates its own motions. Decentralized control offers several advantages over a centralized algorithm[5]. First, it can be difficult for a central computer to control a robot when distance or obstacles limit communication. Second, centralized controllers tend not to scale well as the number of robots increases because a single computer must calculate the paths for a large number of robots.

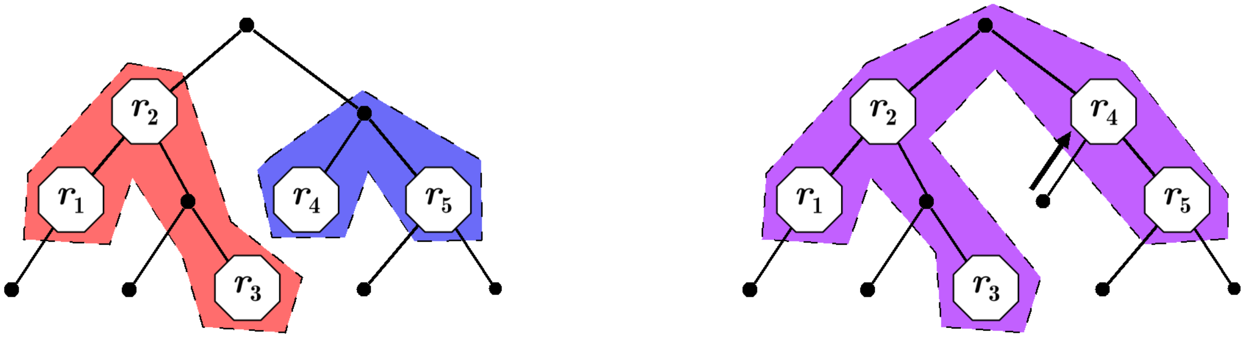
One class of problem where decentralization offers significant advantages is in the navigation of confined spaces. Path planning in confined spaces such as tunnels or hallways is particularly challenging because the passageway can be so narrow that robots are unable to pass one another. If two robots attempt to use the same narrow corridor, one may have to move off its planned path to let the other pass. This problem can arise for mining robots, which must be able to navigate in small tunnels without colliding. Similarly, warehouse management robots need to be able to navigate narrow aisles along predefined tracks [6]. In the first case, decentralization offers the advantage of avoiding a challenging and potentially intermittent communication link to a central computer [7]. In the second case, large numbers of warehouse robots could make centralized control computationally difficult. A decentralized algorithm for the navigation of confined spaces could therefore be extremely beneficial.

Several centralized algorithms for robot navigation in confined spaces already exist. Some of these algorithms have the extremely desirable property of being complete - that is, they guarantee that a solution will be found if it exists [8, 9, 10]. Centralized algorithms can also be classified as either optimal or non-optimal. Optimal algorithms, such as search algorithms like A*, are capable of computing the shortest set of paths that solve the problem (if a solution exists), but the computation is NP-complete [11]. Others, like the push-swap algorithm proposed by Luna, are not guaranteed to find the shortest path, but are capable of finding a solution in much less time [10].

Unlike these centralized algorithms, decentralized architectures do not necessarily have total information on all robots, so it is difficult to guarantee that a solution is always found. For this reason, all known decentralized algorithms to date are not complete and suffer from the possibility of deadlocks [11]. This paper addresses this issue by proposing the Push-Swap-Wait approach, a decentralized algorithm for navigating in confined spaces that is guaranteed to be complete under certain conditions.

1.2 Problem Formulation

Consider a set of nodes N and a set of bi-directional connecting edges between them E which form a graph $G(N, E)$. Occupying G is a set of autonomous robotic agents R. At each timestep t, there is an assignment A that maps each robot $r \in R$ to its location in G, such that $A(r, t) \in N$. All agents have knowledge of $G(N, E)$ and each has a unique assigned goal $g(r) \in N$ such that $g(r_i) \neq g(r_j)$ if $i \neq j$. Each node can contain only one robot at a time, meaning that $\forall r_i, r_j \in R$, if $i \neq j$, then $A(r_i, t) \neq A(r_j, t)$. Between timesteps, robots may move from node n_o to node n_p provided that $\exists\, e \in E : e = (n_o, n_p)$. However, two robots cannot traverse the same edge between the same timesteps, so $\forall r_i, r_j \in R$, if $A(r_i, t+1) = A(r_j, t)$, then $A(r_j, t+1) \neq A(r_i, t)$. The change from one assignment $A(R, t)$



(a) Two separate networks (b) One unified network

Figure 1.2.1: *Formation of networks among robots.* Here, the radius of communication ρ equals 2. As robot r_4 moves in the tree, it enters the communication range of r_2, thus enabling communication between all five robots.

to another $A(R, t+1)$ is determined by the individual position change made my each robot, $\pi(r, t)$. At each timestep t, every robot r computes which move $\pi(r, t)$ to make, which may take the robot along an edge e to a new node n (provided the conditions given above hold) or keep the robot at its current node. The goal is to rearrange the robots from an initial assignment $A(R, 0)$ to a final assignment $A(R, t_{final})$ where $\forall r \in R, A(r, t_{final}) = g(r)$.

In order to make informed decisions about where to go, robots are able to detect and communicate with other robots within a certain radius ρ, measured in the number of edges between agents. All robots r_i within ρ nodes of r are considered to be in direction communication with r, such that $r \in c(r)$. Robots will transmit information about themselves and any other robots of which they are aware. In this way, two robots well outside of individual communication may still be aware of one another thanks to the formation of an ad-hoc communication network among a larger group of robots (see Figure 1.2.1). The set $C(r)$ includes all robots in communication with r, whether direct or indirect.

For the algorithm presented here, robotic motion is restricted to a spanning tree T of G, such that $T = T(N, \varepsilon)$, where $\varepsilon \subseteq E$. With this tree framework, three special kinds of nodes can be identified: leafs, branch nodes, and twigs.

Definition. LEAF NODE: A *leaf is defined as a node* l *such that* $\exists! \, n : (l, n) \in \varepsilon$, *or in other words, a node connected to only one other node.*

The set of nodes L contains the leaf nodes of T, such that $L \subseteq N$.

Definition. BRANCH NODE: *Branch nodes are those nodes* b *for which the number of nodes* n *satisfying* $(b, n) \in \varepsilon$ *is greater than or equal to three, and they correspond to nodes which are connected to three or more edges.*

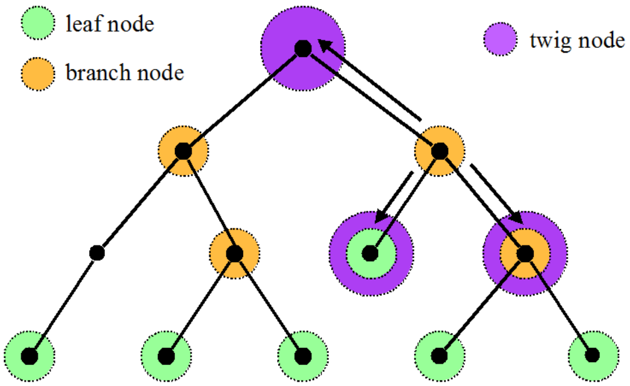


Figure 1.2.2: *Special types of nodes.* This figure shows leaf and branch nodes highlighted for a typical tree structure. For one particular branch node, arrows are drawn pointing from it to its twig nodes. Note that branch nodes, leaf nodes, and regular nodes can all be categorized as twig nodes.

As with leaf nodes, branch nodes of T are contained in a set $B : B \subset N$.

Definition. TWIG NODE: *A node γ is considered to be a twig of branch node b if node γ is adjacent to b such that $\exists e \in \varepsilon : e = (b, \gamma)$.*

Each individual branch node b has an associated set $\Gamma(b)$ which contains all twig nodes γ of b. Figure 1.2.2 illustrates examples of where leaf, branch, and twig nodes appear graphically.

The completeness guarantee of this algorithm is restricted to those cases where $|R| \leq |L| - 1$ and $\rho \geq 2$.

Chapter 2

Push-Swap-Wait Algorithm

2.1 Overview

The Push-Swap-Wait (PSW) algorithm presented here draws inspiration from the push-swap algorithm presented by Luna and Bekris [10]. A third mode, waiting, is added to guarantee completeness for the decentralized problem. This mode is used to ensure that a solution can be found even in cases where communication is lost between swapping robots and pushed robots. Like the push-swap algorithm, robots use two different modes to reach their goal positions. In swap mode, two robots decide to switch positions and move through the tree T to find a branch node at which they can complete the swap. In pushed mode, all robots move out of the path of a swapping pair to allow the swap to take place. The PSW algorithm assigns a priority value to each robot, and then allows the robot with the highest priority to perform any swaps necessary until it reaches its goal. At this point, the robot with the next highest priority receives these same privileges and proceeds towards its goal in the same manner. In this way, PSW successively solves one robot at a time until the overall problem is solved.

2.2 Description

Before any motion planning or movement occurs at time t_0, each robot must analyze the graph G of their environment and calculate the spanning tree T. Each robot r will perform this operation in the same manner, such that each robot has an identical copy of tree T off of which to base decisions. Once the tree T has been formed, every node $n \in N$ will be assigned a priority value $\Phi(n)$ based on a postorder traversal of the tree. This priority ordering assures that no two nodes are given the same priority, such that $\forall n_i, n_j \in N$, if

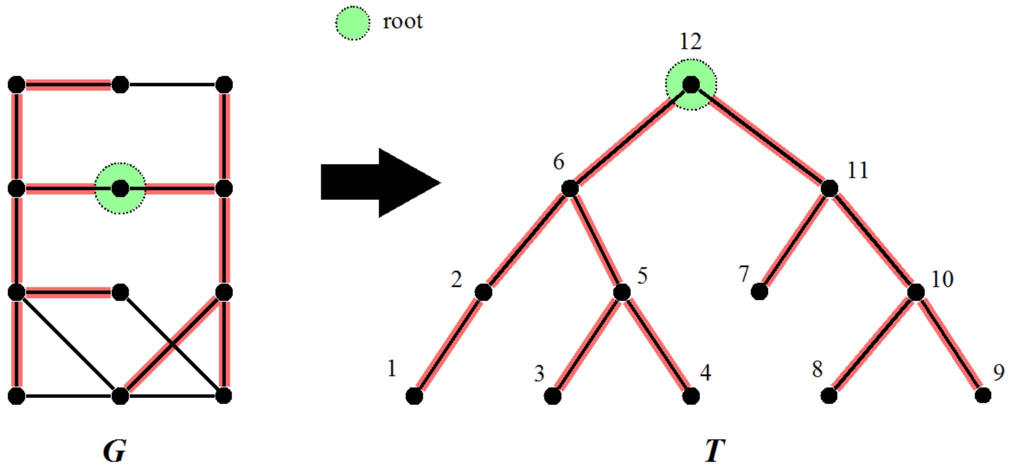


Figure 2.2.1: *Tree formation and priority assignment.* An arbitrary graph G can be transformed into a tree T by choosing a root node and selecting edges according to a breadth-first search. The nodes n of T can then each be assigned a priority $\Phi(n)$ by following a postorder traversal of the tree, as shown in the figure. Note that lower-numbered nodes are considered to be higher-priority.

$i \neq j$, then $\Phi(n_i) \neq \Phi(n_j)$. Figure 2.2.1 illustrates the formation of a tree and the assignment of priority to nodes on that tree. By assigning priority in this way, each robot $r \in R$ can also be given a priority equal to the priority of its goal $g(r)$, such that $\Phi(r) = \Phi(g(r))$. *The ordering of robots by priority is central to the guarantee of completeness* (see Section 2.4)

Definition. PRIORITY*: In reference to nodes, the priority $\Phi(n)$ of node n is the position of node n in a postorder traversal of the tree T. In reference to robots, the priority $\Phi(r)$ of robot r is equal to the priority $\Phi(g(r))$ of its goal and places it in an order relative to all other robots.*

The algorithm dictates that robots behave in such a way that they become solved in order of their priority.

Definition. ANCESTORS: *The set of ancestors of a node $n \in N$ is the set of nodes $P(n) \in N$ such that $P(n) = parent(n) + P(parent(n))$ and is empty for $n = root(T)$.*

Definition. SOLVED: *A robot r is solved at time t when the following conditions are met:*

1. *for some time $t_1 < t$, $A(r, t_1) = g(r)$*
2. *$\forall r_L \in R$ such that $\Phi(r_L) < \Phi(r)$ and $\forall t' : t_1 \leq t' \leq t$ it holds that $A(r, t') \notin P(A(r_L, t'))$*
3. *and $\forall r_H \in R$ such that $\Phi(r_H) > \Phi(r)$, robot r_H is also solved.*

Status	**Description**
NORMAL	Robot is heading towards goal
PAUSED	Robot is temporarily not moving
WAITING	Robot is awaiting the return of r^*
PUSHED	Robot is being pushed by another robot
STUCK	Robot was pushed, but could not move
SWAP_SET	Robot is initializing a swap
SWAP_CONTINUE	Robot is continuing a swap
SWAP_FINISH	Robot is finishing a swap
SWAPPING	Robot is swapping (whether set, continue, or finish)

Table 2.1: *List of possible robot statuses*

At each time t, all robots $r \in R$ decide which move $\pi(r,t)$ to make by executing the $Plan(r,t)$ algorithm. By performing logical checks based on robot r's knowledge of itself and of all other robots $r_i \in C(r)$, $Plan(r,t)$ will determine $A(r,t+1)$ by setting $\pi(r,t)$ as well as set the status of robot r (see table 2.1).

As time progresses, robots will become solved in order of their priority, until some time t_{final} when all robots $r \in R$ have been solved, and by the definition of being solved, $A(r,t_{final}) = g(r)\ \forall r \in R$, meaning that a solution to the problem has been found.

In describing the logic of the algorithm, two definitions related to movement on the tree structure will prove useful: "up" the tree and "down" the tree.

Definition. UP THE TREE: *A node $n_2 \in N$ is up the tree from node $n_1 \in N$ if there exists $n' \in N$ on the path $S_{1,2} \subset \varepsilon$ from n_1 to n_2 such that $n' \in P(n_1)$*

Definition. DOWN THE TREE: *A node $n_2 \in N$ is down the tree from node $n_1 \in N$ if there exists $n' \in N$ on the path $S_{1,2} \subset \varepsilon$ from n_1 to n_2 such that $n \in P(n')$*

2.2.1 Plan

At each time t, each robot $r \in R$ calls the $Plan()$ function to decide on its next move based on its knowledge of other robots in the local communication network $C(r)$. Algorithm 2.1 first checks if r or any robot $r_i \in C(r)$ is waiting for a swapping robot $r^* \in R : r^* \notin C(r)$. If r is the one waiting for r^*, $status(r)$ gets set to WAITING so that r does not move and all other robots in $C(r)$ will remain motionless. If another robot r_i is the one waiting for r^*, r remains motionless to allow r^* to return, but does not set $status(r)$ to WAITING to avoid a loop where other robots remain frozen even after r^* returns because $status(r) =$ WAITING and vice versa.

Algorithm 2.1 *Plan*(r, t)

```
1  if [$\exists r_i \in [r, C(r)] : status(r_i) = \text{WAITING}$] and $r^* \notin C(r)$
2      if $status(r) = \text{WAITING}$
3          $status(r) \leftarrow \text{WAITING}$
4      else
5          $status(r) \leftarrow \text{PAUSED}$
6      end
7      $\pi(r,t) \leftarrow A(r,t)$
8  elseif $r \in [\overline{r}^*, \underline{r}^*] \leftarrow CheckSwap(r,t)$
9      if $\exists \underline{r}^*$
10         $Swap(r,t)$
11     else
12         $\pi(r,t) \leftarrow s(A(r,t), g(r))$
13         $status(r) \leftarrow \text{NORMAL}$
14     end
15 elseif $r^* \in C(r)$
16     $Pushed(r,t)$
17 elseif $\exists r_i \in C(r) : \Phi(A(r_i,t)) > \Phi(A(r,t))$ and $\hat{\pi}(r,t) \in path(r_i)$
18     $status \leftarrow \text{PAUSED}$
19 else
20     $\pi(r,t) \leftarrow s(A(r,t), g(r))$
21     $status \leftarrow \text{NORMAL}$
22 end
```

Algorithm 2.1 next checks if robot r should be swapping. The algorithm calls the *CheckSwap*() function (algorithm 2.2), which returns the two robots that should be swapping, or just the highest priority unsolved robot if it does not need to swap, or NULL if there are no valid swaps. If r is one of the two robots that should be swapping, the algorithm calls the *Swap*() function (algorithm 2.3) to handle the details of the swap. If r is the only robot returned by *CheckSwap*() (i.e. it is the highest priority unsolved robot and does not need to swap), r sets its path to $g(r)$ and $status(r)$ to NORMAL so that it pushes other robots out of its way as it moves to its goal. If *CheckSwap*() does not return any robots, the algorithm moves on.

Next (line 15) the *Swap*() function checks for a swapping robot $r^* \in C(r)$. r^* can be either of the swapping robots $\overline{r}^*$ or $\underline{r}^*$, or it can be the highest priority unsolved robot that is moving towards its goal without needing to swap. If r sees a robot r^*, the algorithm calls the *Pushed*() function (algorithm 2.8) which makes sure that r moves out of the way of r^*.

Finally, the algorithm handles the case where the robot is moving without swapping or being pushed. Since the movement of the highest priority unsolved robot is handled earlier with the call to *CheckSwap*() and *all other robots are stationary unless swapping or being pushed*, this section handles the movement when all robots in $C(r)$ are solved. The algorithm checks if there is a robot $r_i \in C(r)$ on a higher priority branch than r, and r *pauses if* $\hat{\pi}(r,t)$, *the planned next node for* r, *is on the path of* r_i.

Definition. PLANNED NEXT NODE*: if* $\pi(r, t-1) \leftarrow s(A(r, t-1), n)$*, the planned next node of* $r \in R$ *at time* t*,* $\hat{\pi}(r,t)$*, is the next node after* $\pi(r, t-1)$ *on the path* $S(A(r, t-1), n)$*.*

Since robots always choose the lowest priority branch available when getting pushed, this ensures that a robot that got pushed down a higher priority branch moves back up first, preserving the order of solved robots. If there are no robots meeting this criterion, r moves towards its goal with $status(r)$ set to NORMAL.

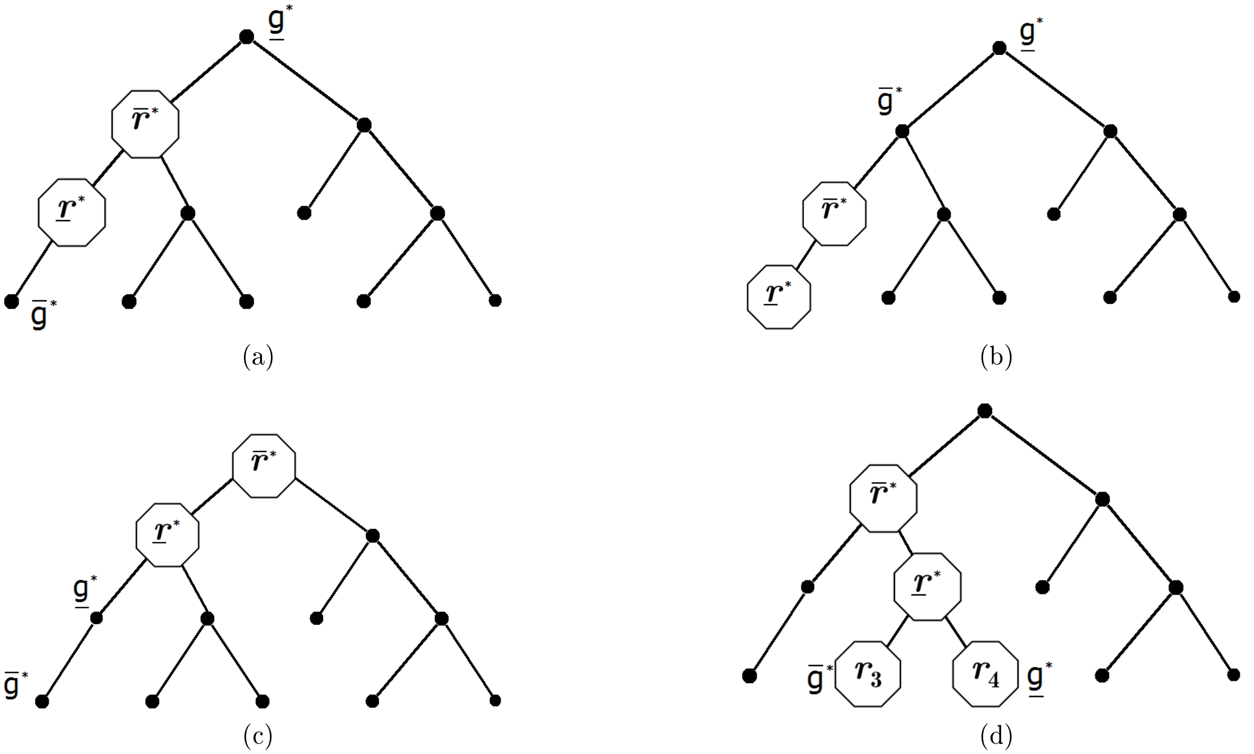
2.2.2 CheckSwap

Algorithm 2.2 *CheckSwap*(r,t) returns $[\overline{r}^*, \underline{r}^*]$

```
$\overline{r}^* \leftarrow r_i \in [r, C(r)] : \Phi(r_i) \geq \Phi(r_j) \forall r_j \in [r, C(r)]$

if $status(\overline{r}^*) = $ SWAPPING
    return $[\overline{r}^*, \underline{r}^*]$
elseif $\overline{r}^*, r_L \in [r, C(r)]$ should swap and $\overline{r}^*, r_L$ are adjacent
    if $\exists r_s \in R : g(r_s) \in P(A(\overline{r}^*, t))$ and $r_s$ is solved
        return $[$NULL, NULL$]$
    else
        $\underline{r}^* \leftarrow r_L$
        $status(\overline{r}^*), status(\underline{r}^*) \leftarrow$ SWAP_SET
        return $[\overline{r}^*, \underline{r}^*]$
    end
else
    return $[\overline{r}^*,$ NULL$]$
end
```

The *CheckSwap*() algorithm determines which robots in a communication network should be swapping, if any. The function first finds the highest priority unsolved robot $\overline{r}^*$ in the set $[r, C(r)]$. If $\overline{r}^*$ is already swapping with a robot $\underline{r}^*$, the pair of robots $[\overline{r}^*, \underline{r}^*]$ is returned to allow the swap to finish. Otherwise, the algorithm checks for a robot on a node adjacent to $\overline{r}^*$ that needs to swap with $\overline{r}^*$. Since $\rho \geq 2$, *the adjacency condition ensures that the two swapping robots will not loose communication with one another.* The *CheckSwap*() algorithm calls a function *ShouldSwap*() to determine if two robots need to swap. The four possible conditions for two robots $\overline{r}^*$ and $\underline{r}^*$ needing to swap are:



(a) (b)

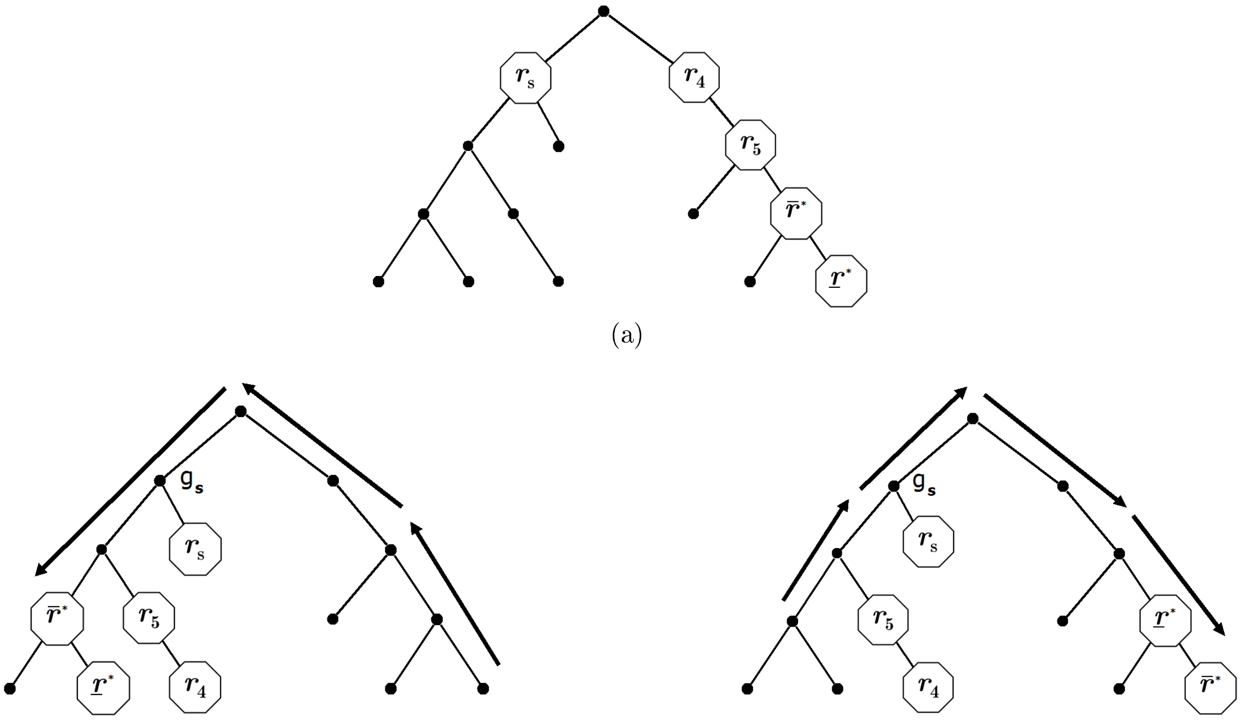
(c) (d)

Figure 2.2.2: *Four swapping conditions.* Figures (a), (b), (c), and (d) each demonstrate one of the four conditions which two robots must satisfy in order to need to swap with one another. In (a), $\underline{r}^*$ is on the path from $\overline{r}^*$ to $g(\overline{r}^*)$ and $\overline{r}^*$ is on the path from $\underline{r}^*$ to $g(\underline{r}^*)$. In (b), $\underline{r}^*$ and $g(\underline{r}^*)$ are on the path from $\overline{r}^*$ to $g(\overline{r}^*)$, and vice versa for (c). Figure (d) demonstrates the case where $\underline{r}^*$ is stuck and blocking the path from $\overline{r}^*$ and $g(\overline{r}^*)$.

1. if $\underline{r}^*$ is on the path from $\overline{r}^*$ to $g(\overline{r}^*)$ and $\overline{r}^*$ is on the path from $\underline{r}^*$ to $g(\underline{r}^*)$
2. if both $\underline{r}^*$ and $g(\underline{r}^*)$ are on the path from $\overline{r}^*$ to $g(\overline{r}^*)$
3. if both $\overline{r}^*$ and $g(\overline{r}^*)$ are on the path from $\underline{r}^*$ to $g(\underline{r}^*)$
4. if $\overline{r}^*$ is heading to its goal without swapping and $status(\underline{r}^*)$ is STUCK (see figure 2.2.2).

The list of robots $r_i \in [r, C(r)]$ is sorted by decreasing priority, so if $\overline{r}^*$ is not already swapping it will choose to swap with the next highest priority robot satisfying the above conditions.

After identifying the swapping robots, the algorithm checks if the swapping robot is at a child node of the goal of a solved robot $r_s \in R$. Note that r_s is in R rather than $C(r)$, meaning that each robot must maintain a list of all solved robots it has seen at any time. If $g(r_s) \in P(A(\overline{r}^*, t))$, *the new swap is suppressed to ensure that any robots that were pushed down the tree past the goal of a solved robot will return to* $C(r_s)$ *before starting a swap* (see Figure 2.2.3). If $g(r_s) \notin P(A(\overline{r}^*, t))$ the algorithm returns the two swapping robots $\overline{r}^*, \underline{r}^*$.



(a)

(b)

(c)

Figure 2.2.3: *Swap suppression at child of* $g(r_s)$*.* In Figure (a), two robots, $\overline{r}^*$ and $\underline{r}^*$, initiate a swap that will take them to the other side of the tree, where they will push two unsolved robots, r_4 and r_5, and one solved robot r_s. In Figure (b), r_s is waiting for the return of the swappers before returning to its goal g_s, preventing it from becoming unsolved. Figure (c) shows a hypothetical situation in which the swappers have moved up the tree from r_s, but unsolved robots remain below. Swap suppression ensures that all robots will move up the tree together, so no robots will be stuck under r_s.

Finally, if the highest priority unsolved robot $\overline{r}^*$ does not need to swap with any other robots, the function returns only $\overline{r}^*$ so it can drive straight to its goal.

2.2.3 Swap

Based on the swapping robot's status, *Swap*() decides which phase of the swap it is in, and calls the appropriate function (Algorithms 2.4, 2.5, 2.6, and 2.7).

StartSwap

StartSwap() is called to initialize a new swap or to pick a new branch point once a pair of swapping robots realize that their original branch point is unavailable. The algorithm first selects the branch node $b \in B$ that is closest to the higher priority robot $\overline{r}^*$ and has not yet been visited by the swapping pair. b is then added to the list of visited nodes, and *all*

Algorithm 2.3 *Swap*(r, t)

```
if $status(r) = $ SWAP_SET
    $StartSwap(r, t)$
elseif $status(r) = $ SWAP_CONTINUE
    $ContinueSwap(r, t)$
elseif $status(r) = $ SWAP_FINISH
    if $r = r_{leader}$
        $FinishSwapLeader(r, t)$
    else
        $FinishSwapFollower(r, t)$
    end
end
```

Algorithm 2.4 *StartSwap*(r, t)

```
$b^* \leftarrow b_i \in B : b_i \notin visited(r)$ and $b_i$ is closest branch node to $\overline{r}^*$
$visited(r) \leftarrow b^*, n \; \forall n \in visited(r) : n \notin P(b^*)$

if $A(\overline{r}^*, t) = s(\underline{r}^*, b^*)$
    $r_{leader} \leftarrow \overline{r}^*$
    $r_{follow} \leftarrow \underline{r}^*$
else
    $r_{leader} \leftarrow \underline{r}^*$
    $r_{follow} \leftarrow \overline{r}^*$
end
$\gamma_{end} \leftarrow \gamma_i \in \Gamma(b^*) : \gamma_i = s(b^*, A(r_{follow}, t))$
$[\gamma_1, \gamma_2] \leftarrow [\gamma_i, \gamma_j] \in \Gamma(b^*) : \gamma_i, \gamma_j \neq \gamma_{end}$

if $r = r_{leader}$
    $\gamma_r \leftarrow \gamma_1$
    $\pi(r, t) \leftarrow s(A(r, t), \gamma_1)$
elseif $r = r_{follower}$
    $\gamma_r \leftarrow \gamma_2$
    $\pi(r, t) \leftarrow s(A(r, t), \gamma_2)$
end

$status(r) \leftarrow$ SWAP_CONTINUE
```

Algorithm 2.5 *ContinueSwap*(r, t)

```
if $status(r') =$ SWAP_SET
    $StartSwap(r)$
elseif $A(r,t) = \gamma_r$
    $\pi(r,t) \leftarrow A(r,t)$
    $status(r) \leftarrow$ SWAP_FINISH
elseif $\exists r_i \in C(r) : A(r_i,t) = \gamma_r$ and $status(r_i) =$ STUCK or SWAP_FINISH
    if $\exists \gamma_{new} \in \Gamma(b) : \gamma_{new} \neq \gamma_{end}, \gamma_r$
        $\pi(r,t) \leftarrow s(A(r,t), \gamma_{new})$
        $status(r) \leftarrow$ SWAP_CONTINUE
    else
        $\pi(r,t) \leftarrow A(r,t)$
        $status(r) \leftarrow$ SWAP_SET
    end
end
```

parents of b are removed so that the swapping robots will check these nodes again on their way back up the tree. This behavior is necessary to guarantee that the swapping robots will be able to find an available branch node even if they lose communication with robots they pushed out of the way.

The algorithm next determines which robot is the leader in the swap, that is, which of $\overline{r}^*, \underline{r}^*$ is closer to b. The twig that the robots must pass to reach b, γ_{end}, is set to be the first node on the path from b to the farther robot r_{follow} so that it is defined even if $A(r_{leader}, t) = b$. The robot r then finds two additional twigs $\gamma_1, \gamma_2 \neq \gamma_{end}$ and sets its path to one of them (see figure 2.2.4). Finally, $status(r)$ is set to SWAP_CONTINUE so that on the next iteration r will move to the twig it selected.

ContinueSwap

ContinueSwap() handles the movement of swapping robots after they choose a branch node and until they reach their destination twig γ_r. If robot r sees that its swap partner r' has reset its status to SWAP_SET, r' must have realized that the branch node b is no longer valid because not enough twigs are available. Therefore r' will pick a different branch node, so r calls *StartSwap*() to also pick a different branch node. *StartSwap*() must be called immediately rather than on the next iteration so that r' does not misinterpret a change in $status(r)$ to mean that another new branch node is needed. Otherwise, the algorithm checks if r has reached its twig, in which case its path is set to its current location so it does not move and $status(r)$ is set to SWAP_FINISH so it calls *FinishSwap*() on the next iteration.

The algorithm finally checks if the current destination of r is still available. This is done by checking for a robot $r_i \in C(r)$ with $A(r_i, t) = \gamma_r$ and whose status is STUCK or SWAP_FINISH. If there are any more twigs of b available (other than γ_r and γ_{end}), r sets its path to this new twig γ_{new} and calls *ContinueSwap*() again on the next iteration. Note that this could cause $\gamma_{new} = \gamma_{r'}$, but this would only occur if $r = r_{leader}$ because *StartSwap*() sets $\gamma_{leader} = \gamma_1$. In this case r_{leader} would reach γ_{leader} and set $status(r_{leader})$ to SWAP_FINISH, and on the next iteration r_{follow} would realize its twig was occupied and pick a new twig. In this way the swapping robots iterate through all twigs of b, and only pick a new branch point if there are not enough available twigs of b. When robot r realizes that there are insufficient twigs it sets $status(r)$ to SWAP_SET, and on the next iteration both swapping robots pick a new branch node.

FinishSwap

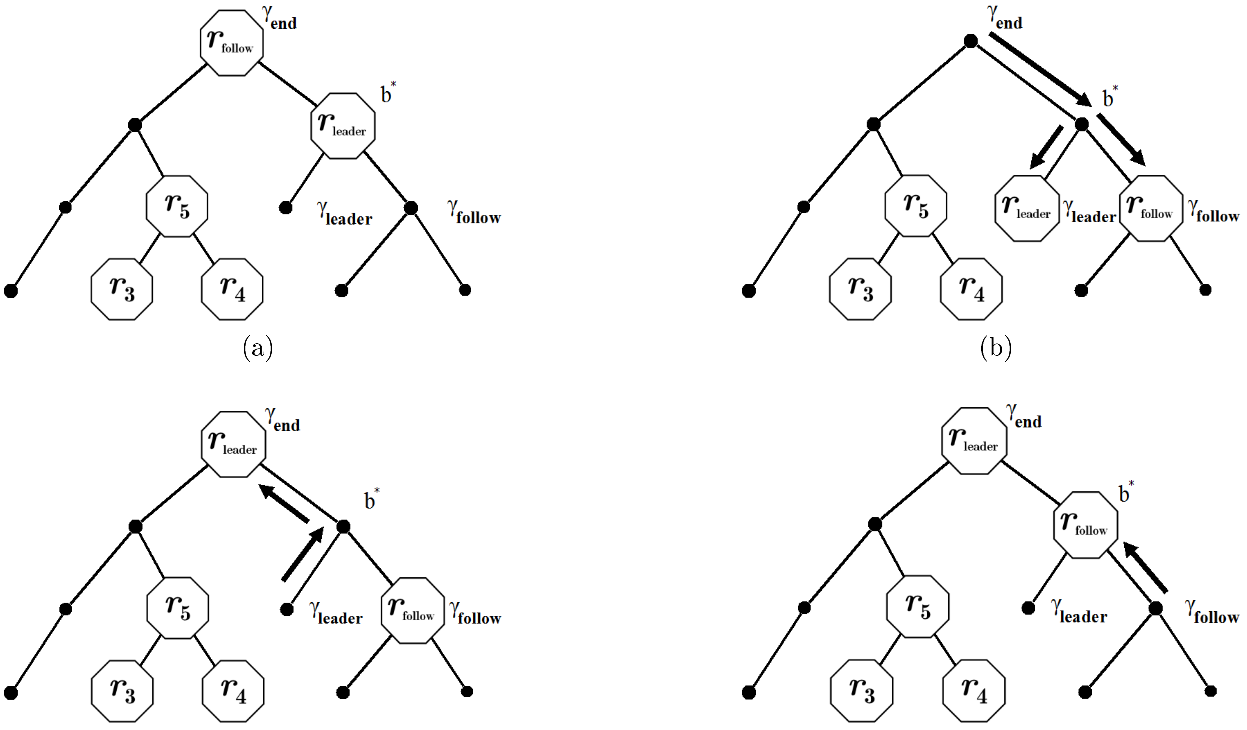
The *FinishSwap*() function handles the details of a swap once the robots have reached an available branch node. The algorithm ensures that robots leave their twigs in the proper order to complete swapping. There are two different functions depending on the order of the robots as they arrive at the branch node: *FinishSwapLeader*() is called if robot r is the first to reach the branch node, and *FinishSwapFollower*() is called if r is the second robot. Figure 2.2.4 demonstrates the steps involved in completing a swap.

FinishSwapLeader If robot $r = r_{leader}$, there are three possible states to consider: r could be at its twig γ_r, or r could be at the end position γ_{end}, or r could be on the branch node b (see figure 2.2.4). In the first case, where $A(r, t) = \gamma_r$, the algorithm checks if r sees that its swap partner robot r' was unable to reach an available twig and needs to use a different branch node. If this is the case, r immediately calls *StartSwap*() to pick a new branch node. Next the algorithm checks if the swap partner r' has reached its twig $\gamma_{r'}$, in which case r sets its path to move to the end twig γ_{end}. Finally, if neither condition holds r assumes that r' is on b moving towards γ_r, so r does not move.

In the second case, where $A(r, t) = \gamma_{end}$, r has reached the end twig γ_{end} so it is waiting for the swap partner r' to reach the branch node b before the swap is complete. r therefore checks if $A(r', t) = b$, and if so sets its path to $g(r)$ and $status(r)$ to NORMAL. Otherwise, r assumes that r' is still moving towards b and does not move.

Finally, if robot r is heading towards γ_{end} it simply continues moving.

FinishSwapFollower If $r = r_{follower}$ there are two possible states: it can be at its twig γ_r, or it can be at the branch node b. If $A(r, t) = \gamma_r$, r first checks if its swap partner r'



(a) (b)

(c) (d)

Figure 2.2.4: *Process of finishing a swap.* In Figure (a), the robots r_{leader} and r_{follow} have just arrived at $b^*$and γ_{end} and will soon begin finishing their swap. In (b), r_{leader} and r_{follow} arrive at their respective twigs, and each will call the function $FinishSwap()$ in order to calculate their next position. Figure (c) shows r_{leader} reaching γ_{end}. In Figure (d), r_{follow} reaches b^* and the swap is over. Notice that r_{follow} and r_{leader} have swapped positions from (a) to (d).

Algorithm 2.6 *FinishSwapLeader*(r, t)

```
if $A(r,t) = \gamma_r$
    if $status(r') =$ SWAP_SET
        $StartSwap(r)$
    elseif $A(r',t) = \gamma_{r'}$
        $\pi(r,t) \leftarrow s(A(r,t), \gamma_{end})$
        $status(r) \leftarrow$ SWAP_FINISH
    else
        $\pi(r,t) \leftarrow A(r,t)$
        $status(r) \leftarrow$ SWAP_FINISH
    end
elseif $A(r,t) = \gamma_{end}$
    if $A(r',t) = b$
        $\pi(r,t) \leftarrow s(A(r,t), g(r))$
        $status(r) \leftarrow$ NORMAL
    else
        $\pi(r,t) \leftarrow A(r,t)$
        $status(r) \leftarrow$ SWAP_FINISH
    end
elseif $r$ heading to $\gamma_{end}$
    $\pi(r,t) \leftarrow s(A(r,t), \gamma_{end})$
    $status(r) \leftarrow$ SWAP_FINISH
end
```

Algorithm 2.7 *FinishSwapFollower*(r, t)

```
if $A(r,t) = \gamma_r$
    if $A(r',t) = \gamma_{end}$
        $\pi(r,t) \leftarrow s(A(r,t), b)$
        $status(r) \leftarrow$ SWAP_FINISH
    else
        $\pi(r,t) \leftarrow A(r,t)$
        $status(r) \leftarrow$ SWAP_FINISH
    end
elseif $A(r,t) = b$
    $\pi(r,t) \leftarrow s(A(r,t), g(r))$
    $status(r) \leftarrow$ NORMAL
end
```

Algorithm 2.8 *Pushed*(r, t)

```
if $\exists r_i \in C(r) : A(r,t) \in path(r_i)$ and $status(r_i) =$ PUSHED or SWAPPING
    if $\exists n \in \delta(A(r,t), t)$
        $\pi(r,t) \leftarrow \delta_L(A(r,t), t)$
        $status(r) \leftarrow$ PUSHED
        return
    else
        $\pi(r,t) \leftarrow A(r,t)$
        $status(r) \leftarrow$ STUCK
        return
    end
elseif $r^* \in c(r)$
    if $\hat{\pi}(r^*, t) = s(A(r^*, t), g(r^*))$
        $\pi(r,t) \leftarrow A(r,t)$
        $status(r) \leftarrow$ PAUSED
        return
    else
        $\pi(r,t) \leftarrow A(r,t)$
        $status(r) \leftarrow$ WAITING
        return
    end
elseif $status(r) =$ WAITING
    $\pi(r,t) \leftarrow A(r,t)$
    $status(r) \leftarrow$ WAITING
    return
else
    $\pi(r,t) \leftarrow A(r,t)$
    $status(r) \leftarrow$ PAUSED
    return
end
```

has reached γ_{end}. Otherwise r does not move. In the second case, if $A(r,t) = b$ then robot r' must have already reached γ_{end}, so the swap is complete. r sets its path to its goal, and $status(r)$ is set to NORMAL.

2.2.4 Pushed

The *Pushed*() algorithm governs the behavior of robot r in the case where r is neither swapping nor the highest priority robot in $[r, C(r)]$, and r is not waiting to regain communication with r^*. First, robot r checks whether it is on the path of of any other robot r_i that is being affected by a swap (either one of the swappers, or a robot being pushed by the swappers).

If so, r examines the set of nodes $\delta(A(r,t))$ that are adjacent to its current position and available for a pushed robot.

Definition. AVAILABLE FOR PUSHED ROBOT: *A node n is available for a pushed robot r if it is not on the path between the pushed robots and the swapping robots ($n \neq s(A(r,t), A(r^*,t))$) and it is not occupied by a stuck robot.*

If there are adjacent nodes available and $\delta(A(r,t),t)$ is not an empty set. then $\pi(r,t)$ is set to the lowest priority node in this set, $\delta_L(A(r,t),t)$, and the status of r is set to PUSHED since it is being pushed to this new node. If $\delta(A(r,t),t)$ is an empty set, then robot r has nowhere to be pushed and sets its status to STUCK while remaining on its current node.

When r is not on the path of a swapping robot or a pushed robot, the algorithm checks whether r is in direct communication with the highest priority unsolved robot r^*, that is, $r^* \in c(r)$. If so, and r^* is heading towards its goal such that its predicted move $\hat{\pi}(r^*,t)$ is the next node on the path between its current location and its goal (that is $s(A(r^*,t), g(r^*))$) then r will remain on its current node and set its status to PAUSED. If, however, r^* is not heading towards its goal, r will remain on its current node and set its status to WAITING so that, if at time $t+1$ $r^* \notin C(r)$, r will wait for the return of r^*. *This waiting ensures that robot r remains in the correct order with respect to r^*, such that, if r and r^* have swapped, they will never need to swap again.* If $r^* \notin c(r)$, then robot r will set its status based on its previous status, WAITING if it was previously set to WAITING, and PAUSED otherwise. The continuity of the WAITING status allows r to continue preparing to wait when r^* is still in $C(r)$ but not in $c(r)$.

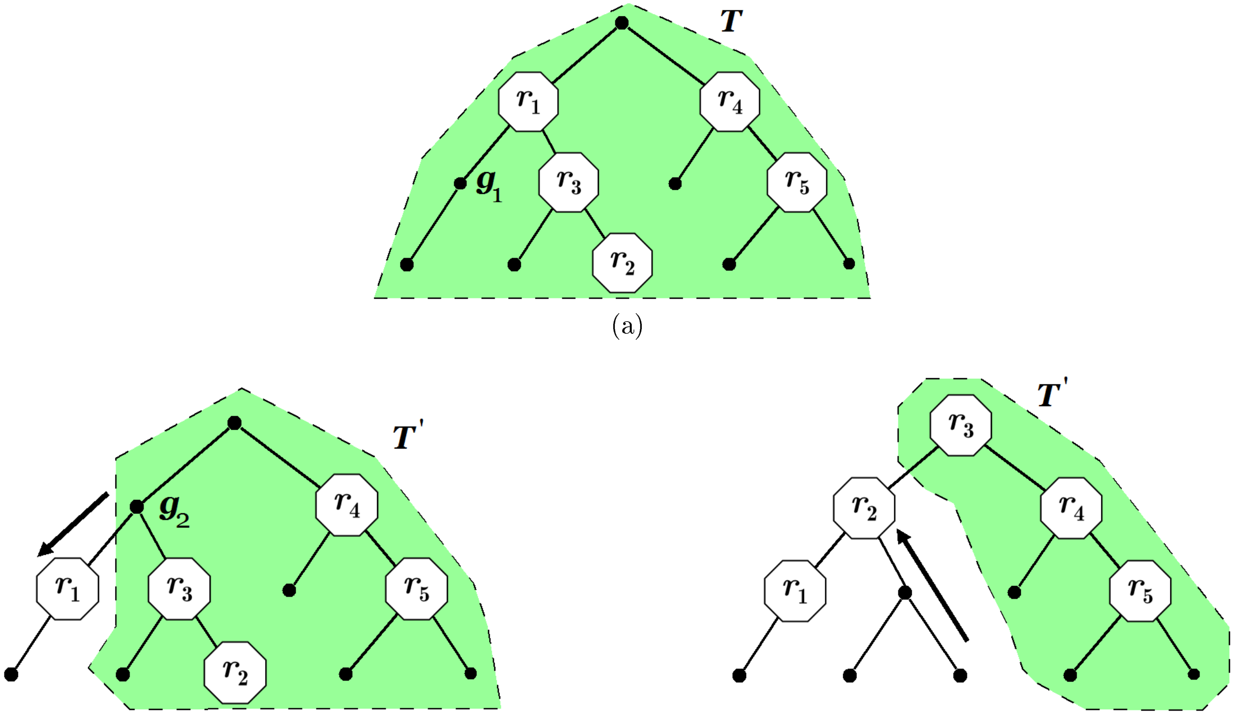
2.3 Key Features

After looking at each component in detail, some important characteristics of the Push-Swap-Wait algorithm will be highlighted. First, in order to make this difficult problem more manageable, robot motion is restricted to a spanning tree T instead of allowing robots to traverse any edge in the graph G. While this change eliminates potential shortcuts between nodes, the nature of the tree structure can be exploited in order to guarantee the completeness of the algorithm in spite of the decentralization of motion planning. Second, one robot $r \in R$ is permitted to reach its goal at a time. To that end, in any given network of communication, there can only be one pair of swapping robots, while all other robots will only move in order to accommodate the swappers. This restriction allows the algorithm to focus on sequentially finding solutions to smaller subsets of the full problem at the cost of overlooking potential simultaneous solutions.

The nature of the PSW algorithm allows for the guarantee that a solution to any given instance of the problem can be found by solving sub-problems and ensuring that they remain solved. The guarantee that a solved sub-problem is not disturbed by any subsequent swaps is based on several key components of the algorithm. By assigning robot r a priority $\Phi(r)$ based on a postorder traversal of the tree, and only allowing the highest priority unsolved robot to reach its goal, the problem can be successively reduced into smaller subtrees where solved robots are effectively removed from the overall problem (see Figure 2.3.1). In reality, there are some situations in which a solved robot must be disturbed if there are insufficient leaf nodes in the reduced problem to guarantee a solution. It is therefore necessary to ensure that any such solved robot that is pushed by a swap is able to return to its goal position without becoming unsolved. To achieve this, the algorithm forces all pushed robots to move to the lowest priority node possible, and after being pushed to give the right of way to robots on higher priority nodes, ensuring that solved robots recover from a push operation.

The decentralization of decision making in this problem lead to some of the greatest challenges in developing a complete algorithm, namely, handling situations in which robots lose communication with one another and are therefore forced to plan their motion based on incomplete information. For instance, in the case mentioned above where solved robots are displaced from their goals, problems can arise if the swapping robots leave the network of communication of the solved robots and become trapped when the solved robots return to their goals. To address this and other issues, pushed robots - solved or unsolved - wait for swappers to return if the swapping robots were last seen moving away from the goal of the highest-priority swapper. Additionally, the algorithm prevents new swaps from being initialized by robots that are at a descendant node of a solved robot's goal. These precautions correct for the previous issue and ensure that unsolved robots cannot become trapped behind a solved robot.

Loss of communication also presents potential problems for two robots attempting to complete a swap. If the two swapping robots are not in communication with one another when a critical event, such as discovering that a certain branch node cannot be used to swap, takes place, it is possible that the two would make different decisions about how to proceed and the swap would be unsuccessful. The algorithm prevents this undesirable situation by only allowing swaps to be initiated by two robots occupying adjacent nodes. Since the radius of communication ρ is required to be at least two edge lengths, the swapping robots will be able to maintain constant communication as they traverse the tree. Similarly, if the swapping robots lose communication with robots that they have already pushed, it is possible that the availability of branch nodes in the tree could change when the previously pushed robots move (see Figure 2.4.3). To compensate for the dynamic nature of branch node availability,



(a)

(b)

(c)

Figure 2.3.1: *Successive problem reduction.* The problem begins with all robots needing to reach their goals, and all nodes on the tree T being possible locations for any robot. (Figure (a)). When robot r_1 reaches its goal node g_1 and becomes solved, it is known that no robots occupy any lower priority nodes, and the problem space is reduced to a subset of robots and a subset of the tree, T' (Figure (b)). When robot r_2 reaches its goal, the remaining problem space is even further reduced (Figure (c)). While solved robots may later be pushed down the tree, whenever a new robot becomes solved at some time t, there can be no lower priority robots on higher priority nodes at that time t.

swapping robots will re-check all potential branch nodes as they make their way towards the root. Through these corrective measures, robots compensate for the limitations on their information inherent in a decentralized system and maintain the completeness guarantee of the algorithm.

2.4 Proof of Completeness

The completeness guarantee of this algorithm is based on several lemmas. First, for any given tree T and set of robots R such that $|R| \leq |L| - 1$, there exists a branch node b such that for any single pair of two robots $r_i, r_j \in R$, r_i and r_j can swap. Second, there exists a sequence of moves Π over some time period t_1 to t_2 such that $\forall r_i, r_j \in R$, it is possible that $A(r_i, t_2) = A(r_j, t_1)$ and $A(r_j, t_2) = A(r_i, t_1)$. That is, any pair of two robots can swap positions in the tree T. Third, through a series of these swaps, the highest priority robot yet to be solved, r^*, will reach his goal at some time t^* such that $A(r^*, t^*) = g(r^*)$ and it becomes solved. Finally, once a robot r has been solved, no future swaps will cause it to become unsolved. That is, it will never need to swap to return to its goal.

Definition. AVAILABLE BRANCH NODE: *For a branch node b to be available for two swapping robots r_i and r_j at time t, it must satisfy the conditions that $\forall r_x \in R : r_x \neq r_i$ and $r_x \neq r_j$, $A(r_x, t) \neq b$ and that there are at least three nodes $n \in N$ such that edge $(b, n) \in \varepsilon$ and $A(r_x, t) \neq n$.*

2.4.1 Lemma One: Branch Availability

It will first be shown that there is a branch node available for two robots to swap. Considering an instance of the problem where $|R| = 2$ (the minimum number of robots for a nontrivial solution) and $|L| = 3$, each node $l \in L$ has exactly one edge connecting it to the rest of the tree. If two leafs l_1 and l_2 are part of the same tree $T(N, \varepsilon)$, then they must be somehow connected by a path defined by a set of nodes $S_{1,2}$. Since all nodes but l_1 and l_2 on this path must be connected to at least two other nodes by edges $e \in \varepsilon$, and l_1 and l_2 are connected to exactly one node, the path from a third leaf node l_3 to l_1, $S_{1,3}$, must overlap with $S_{1,2}$ such that they have at least one node $b \neq l_1$ in common. This node b will therefore be connected to a node n_{com} which is common to both $S_{1,2}$ and $S_{1,3}$, as well as to two additional nodes, one in $S_{1,2}$ and one in $S_{1,3}$. Because it has at least three edges associated with it and there are no non-swapping robots to occupy node b or its adjacent nodes, node b satisfies the criteria for a branch nodes and is available for swapping. Figure 2.4.1 represents this relationship graphically.

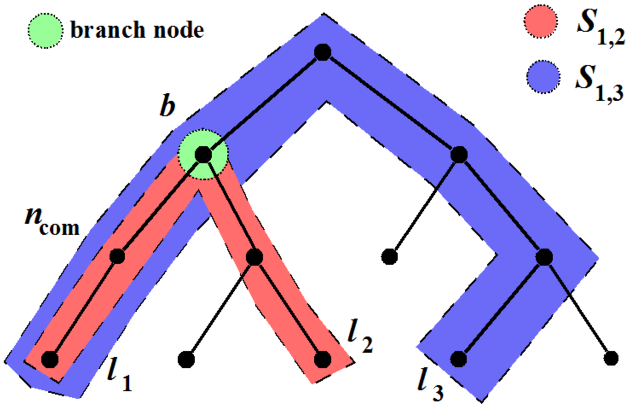


Figure 2.4.1: *Existence of a branch node.* The paths from l_2 and l_3 to l_1, $S_{1,2}$ and $S_{1,3}$ must intersect at some branch node b such that b is connected to a node in common between $S_{1,2}$ and $S_{1,3}$, n_{com}, as well as two other nodes, each unique to one of the two sets.

In applying this to a tree T of arbitrary size containing a group of robots R such that the condition $|R| \leq |L|-1$ holds, in the case of global communication ($\rho \to \infty$), swappers will be able to push all other robots to leaf nodes and the other robots will not move until the swap is complete. The swappers themselves can then navigate to leaf nodes such that at some time time t, $\forall r \in R$, $A(r,t) \in L$, and at least one leaf node will remain unoccupied. With all robots on leaf nodes, there exists an available branch node based on the logic introduced for the case where $|R| = 2$. To prove this, let l_i be the leaf node occupied by robot r_i, then $\forall r_i, r_j \in R$, let the leaf nodes mentioned above, l_1 and l_2, equal l_i and l_j, respectively, and allow l_3 to be one of the unoccupied leafs. As discussed for the case when $|L| = 3$ and $|R| = 2$, there must be a branch node b between these three leafs. Additionally, the paths from this node b to l_1, l_2, and l_3 can not contain any node $n = A(r,t), \forall r \in R$, since no nodes on these paths may be leaf nodes, including node b itself. This fact means that branch node b satisfies the condition for availability since it is adjacent to at least three nodes $n \in N$ such that $(b,n) \in \varepsilon$ and $A(r_x,t) \neq n$ for $r_x \in R : r_x \neq r_i$ and $r_x \neq r_j$. Therefore, there is always a potential branch point that any robots r_i and r_j could use to swap.

2.4.2 Lemma Two: Ability of Robots to Swap

Next, it will be shown that even for cases where the radius of communication is limited ($\rho < \infty$), a single pair of swapping robots will still be able to reach an available branch point. Limited communication presents two possible cases for robots trying to swap. First, it is possible that all robots, swappers and others, maintain communication throughout the swap. Second, it is possible that the swappers lose communication with other robots before the swap is completed. Since robots only decide to initiate swaps with robots on adjacent

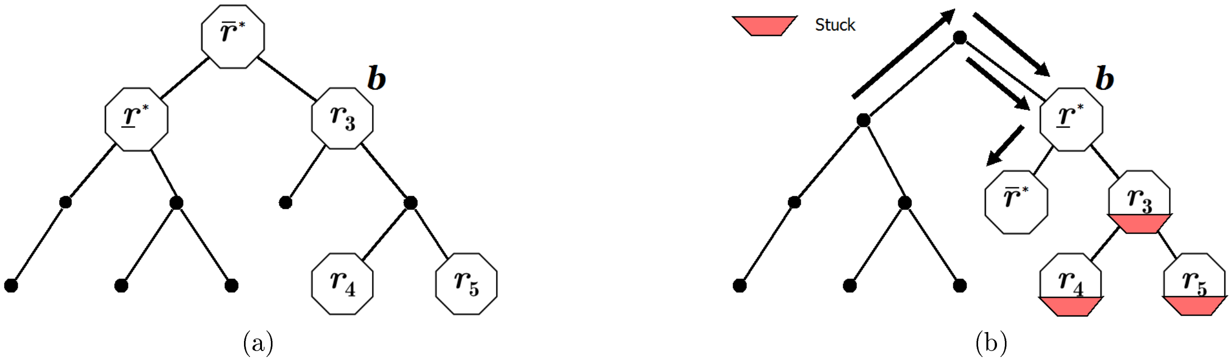


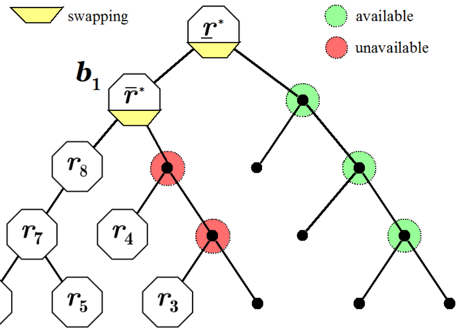
Figure 2.4.2: *Swappers encounter stuck robots.* In Figure (a), the swapping robots $\overline{r}^*$ and $\underline{r}^*$ select branch node b as their branch to complete a swap. After pushing robot r_3 down the tree, r_3, r_4, and r_5 become stuck and the swapping robots know that branch node b is unavailable for swapping.

nodes and swappers always choose to head towards the same branch node, the algorithm dictates that swapping robots will never lose communication with one another.

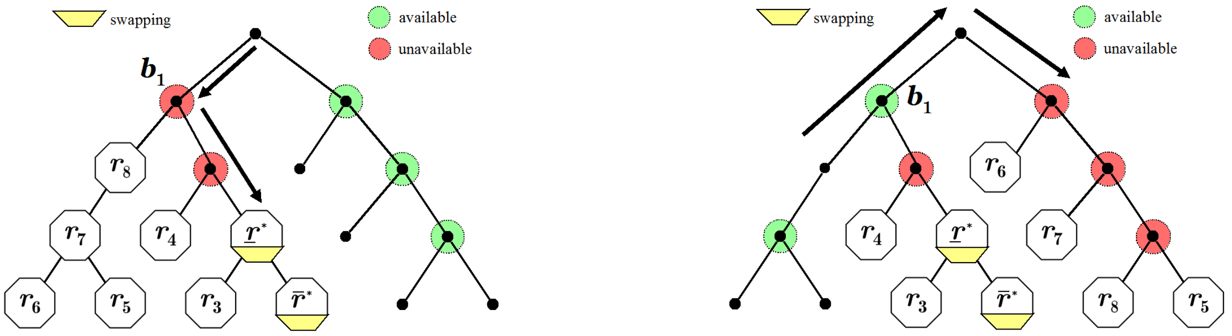
Case 1: Persistent Communication Following the algorithm presented here, once two robots $\overline{r}^*, \underline{r}^* \in R$ have identified themselves as the highest priority swapping pair and have selected a branch node b_1, they will push any other robot r that they encounter to the lowest priority nodes possible until $\overline{r}^*$ and $\underline{r}^*$ either reach twigs of b_1 or realize that branch node b_1 cannot be made available.

Since pushed robots will be instructed to stay in place for the remainder of the swap (enabled by persistent communication), the problem then reduces to the case where all nodes occupied by stuck robots are removed from the tree. Considering only the remaining set of nodes $N' \subset N$ and the remaining robots $R' \subset R$ that occupy N' such that $\overline{r}^*, \underline{r}^* \in R'$, it must be the case that $|R| - |R'| = |N| - |N'| \geq |L| - |L'|$ since each stuck robot removed from R corresponds to a node removed from N, but not all nodes removed from N are necessarily leaf nodes. Therefore, $|L'| - |R'| \geq |L| - |R|$. From the original constraint that $|R| \leq |L| - 1$, $|L| - |R| \geq 1$ and hence $|L'| - |R'| \geq |L| - |R| \geq 1$. The new tree $T'(N', \varepsilon')$ will therefore also still satisfy the condition that $|R'| \leq |L'| - 1$. Knowing this, Lemma One gives that there is a branch node b' in T' which can be made available for $\overline{r}^*$ and $\underline{r}^*$ to use for swapping.

The swapping robots will then begin looking for branch nodes in T', knowing that the other portion of T is completely occupied by stuck robots. In the worst case, $\overline{r}^*$ and $\underline{r}^*$ will continue pushing other robots and refining their search to smaller and smaller subtrees until they are the only two robots which remain on some subtree T^x, in which case there can be



(a) Assignment at time t_1



(b) Assignment at time $t_2 > t_1$

(c) Assignment at time $t_3 > t_2$

Figure 2.4.3: *Dynamic availability of branch nodes.* If robots $\overline{r}^*$and $\underline{r}^*$ decide to swap at time t_1 (Figure (a)), they would first select b_1 as a branch node, discover that it is unavailable, and continue exploring branch nodes down the tree until time t_2 (Figure (b)). At this point, if the swapping robots have lost communication with robots r_5 through r_8, it would be possible for those robots to navigate to the other side of the tree and change the availability of branch nodes in the tree (Figure (c)). In particular, notice that branch node b_1 has been made available for swapping after having been previously identified as unavailable by the swapping robots.

no other robots which will prevent them from using a branch node b_x found in T^x to swap.

Case 2: Loss of Communication In the case where the swapping robots $\overline{r}^*$ and $\underline{r}^*$ lose communication with robots that they have already pushed, there is no guarantee that those robots will stay in place. That is, a robot r that became stuck on some node n_1 at time t_1, such that $n_1 \notin N'$, may at some future time t_2 occupy some node $n_2 : n_2 \in N'$. This complication means that two swapping robots cannot simply iterate through all possible branch nodes if they want to be guaranteed to be able to swap, as it may be possible that different branch nodes can be made available at different times (see Figure 2.4.3 for more detail).

To correct for the dynamic nature of the set of available branch nodes, *whenever the swapping pair $\overline{r}^*$ and $\underline{r}^*$ select a new target branch node b, they remove any parent branch* *nodes* $b_i : b_i \in P(b)$ *from their list of previously visited branch nodes.* This change takes advantages of the tree structure of T to ensure that, as $\overline{r}^*$ and $\underline{r}^*$ make their way up T from an unavailable branch node b, they check all branch nodes which may now be made available since last they were examined, that is all nodes $b_i \in P(b)$. This behavior guarantees that $\overline{r}^*$ and $\underline{r}^*$ will eventually be able to swap because they will either find that a previously unavailable branch node can be made available, or they will find that it is still unavailable, in which case there must be a branch node that can be made available elsewhere in T, following the logic for the case of persistent communication.

2.4.3 Lemma Three: Goal Reachability

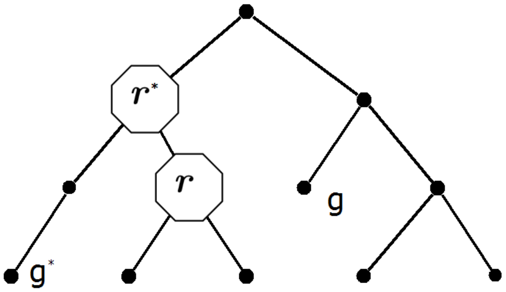
Given that any two robots can swap positions, it will now be shown that through a series of swaps, the highest priority unsolved robot $r^* \in R$ will reach its goal at some time t^* such that $A(r^*, t^*) = g(r^*)$ and become solved. This property follows from the fact that once r^* has swapped with another robot $r \in R$, the algorithm prevents r from coming between r^* and its goal $g(r^*)$, as will be shown below. Taking advantage of the properties of the tree structure, it can be shown that this fact holds regardless of which direction r^* travels.

Suppose r^* completes a swap with some robot $r \in R$ at time t_1. Since r and r^* have just swapped, the two robots must be correctly positioned with respect to one another (that is, if $g(r^*)$ is down the tree from $A(r^*, t_1)$, then $A(r, t_1)$ is up the tree from $A(r^*, t_1)$, and vice versa). As r^* begins to move again, it will either head towards or away from its goal $g(r^*)$. If r^* is heading away from its goal, and r and r^* lose communication with one another, r *will wait in place until it regains communication with* r^*, ensuring that the ordering of r and r^* is maintained. If r^* moves towards its goal, no movement by r can place it on the path between $r^*$and $g(r^*)$, because to do so r would need to pass through r^* (see Figure 2.4.4).

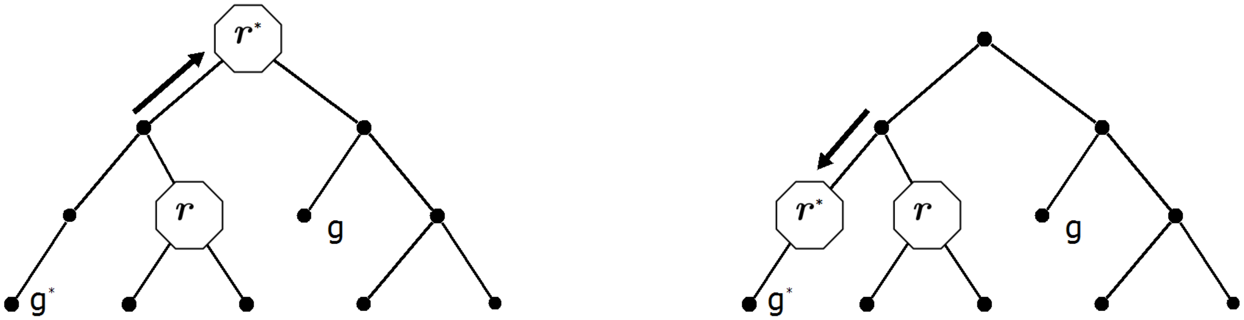
By the property that after swapping with r^* robot r can never come between r^* and $g(r^*)$, it follows that once r^* has swapped with any robot r, there will never be a time $t_2 : t_2 > t_1$ at which r^* will again need to swap with r. In the worst case, r^* can swap with every other robot $r \in R$ before having an unobstructed path to its goal. Therefore, when it finishes swapping and reaches its goal, r^* will satisfy all the conditions to be solved.

2.4.4 Lemma Four: Solved Robots Never Swap

Next it will be proved that a solved robot $r \in R$ will not swap with any other robots and can only be pushed down the tree. Considering first the case of the highest priority robot $r_1 \in R$ such that $\Phi(r_1) > \Phi(r_L)\ \forall r_L \in R : r_L \neq r_1$, if r_1 is solved all robots must be up the tree from r_1, so if r_1 is pushed it can only be pushed down the tree. Since at some



(a) Completion of Swap



(b) r^* moves away from $g(r^*)$

(c) r^* moves towards $g(r^*)$

Figure 2.4.4: *Permanence of swaps.* Figure (a) shows a just completed swap between r and r^*. In (b), r^* moves away from its goal. In this case, r will stay in place until r^* reaches its goal or it loses communication with r^*. However, since r^* is heading away from its goal, if communication is lost, r will wait for $r^*$to return, preventing r from coming between $r^*$and $g(r^*)$. In (c), r^* is moving towards its goal, which physically blocks r from coming between r^* and $g(r^*)$.

time $t_1 < t$ $A(r_1, t_1) = g(r_1)$, if r_1 is pushed down the tree it will always be the case that both r_L and $g(r_L)$ are up the tree from r_1. Also, $g(r_L)$ must be up the tree from $g(r_1)$ since $\Phi(r_1) > \Phi(r_L)$, so if r_1 is solved it does not meet any of the conditions for a swap. Applying the same logic to other solved robots r, any robots down the tree from r will be solved and will not swap. All lower priority robots $r_L \in R : \Phi(r) > \Phi(r_L)$ will be up the tree from r, so if r is pushed it can only be pushed down the tree. Once again, this means that both r_L and $g(r_L)$ are up the tree from r, and since $g(r_L)$ is up the tree from $g(r)$, a solved robot will never swap.

2.4.5 Lemma Five: Solution Monotonicity

Finally, it will be shown that once a robot is solved it remains solved regardless of other swaps. That is, if time t_1 is the time that robot $r \in R$ is first solved, there is no time $t_f : t_1 < t_f$ at which robot r becomes unsolved.

Consider a set of solved robots $r_1, r_2, ..., r_n \in R$ such that $\Phi(r_1) > \Phi(r_2) > ... > \Phi(r_n)$. The definition of a solved robot dictates that the only way r_n could be unsolved at some time t_f is if for some $r \in r_1 ... r_n$ and some $r_L \in R : \Phi(r) > \Phi(r_L)$ $A(r, t_f) \in P(A(r_L t_f))$. By Lemma Four, at any time $t : t_1 \leq t$ robot r can only get pushed down the tree or move back up the tree to its goal. Since r will stop moving up the tree when it reaches $g(r)$, it could only become unsolved if at some time t, $g(r) \in P(A(r_L, t))$. Also, since *robots choose the lowest priority branch available when getting pushed,* a low priority branch must fill completely before pushed robots move on to a higher priority branch, and therefore r could only become unsolved if at time t, $\Phi(A(r, t)) < \Phi(A(r_L, t))$.

It will now be shown that even in situations where the conditions that $g(r) \in P(A(r_L, t))$ and $\Phi(A(r, t)) < \Phi(A(r_L, t))$ are met, the algorithm will prevent robot r from becoming unsolved. There are two cases to consider: robot r maintains communication with all robots r_L satisfying $\Phi(A(r, t)) < \Phi(A(r_L, t))$ and $g(r) \in P(A(r_L, t))$, and r loses communication with some robots r_L.

Case 1: Persistent Communication In the first case, the algorithm dictates that $A(r, t_f) \notin P(A(r_L, t_f))$ because *robots give right of way to other robots on a higher priority branch.* Therefore r will wait until $\Phi(A(r, t)) > \Phi(A(r_L, t))$ before moving back up the tree, so r will not be unsolved if communication is maintained.

Case 2: Loss of Communication In the second case, where r loses communication with some robots r_L such that $\Phi(A(r, t)) < \Phi(A(r_L, t))$ and $g(r) \in P(A(r_L, t))$, it must be the case that r also loses communication with both of the swapping robots $r^* \in R$, and r^*

also satisfies $\Phi(A(r,t)) < \Phi(A(r^*,t))$ and $g(r) \in P(A(r^*,t))$. This is due to the fact that *robots are only pushed one node at a time* and $\rho \geq 2$, so r will always have a communication network at least one node beyond the branch node where r_L took a different path than r. This means that r^* must have pushed r_L down past the branch node and is also out of communication in the same direction.

Since $\rho \geq 2$, at some point r will have communication with $r^*$and will see it heading down the tree. It must be the case that r^* is heading away from its goal because $g(r) \in P(A(r^*,t))$ and $\Phi(r) > \Phi(r^*)$, so *r will wait until r^* returns to the communication network before moving.* Once r^* begins moving back up the tree, *no robots are allowed to initiate swaps when $g(r) \in P(A(r_L,t))$*, and any robots r_L that were pushed by r^* will also move back up the tree and follow r^* back into the communication network. Once r regains communication with r_L, the argument presented above demonstrates that r will remain solved.

It is therefore not possible for any sequence of moves to cause $A(r,t_f) \in P(A(r_L t_f))$, so there is no t_f at which robot r_n becomes unsolved.

2.4.6 Theorem: Completeness of Algorithm

By Lemma One, for any given tree T and set of robots R such that $|R| \leq |L| - 1$, for any two robots $r_i, r_j \in R$, there exists a branch node b such that r_i and r_j can swap. Second, by Lemma Two any two robots will be able to reach an available branch point and swap positions in the tree T. The four criteria for a swap to take place (see algorithm 2.2) reduce to testing for a robot $r_i \in R$ between robot $r \in R$ and $g(r)$ that cannot move off the path, so the criteria will successfully pick the correct swaps to perform. Through a series of these swaps, Lemma Three states that the highest priority robot yet to be solved, r^*, will reach its goal at some time t^* such that $A(r^*,t^*) = g(r^*)$ and become solved. By Lemma Five, once a robot r has been solved, no future swaps by other robots will cause it to become unsolved. Therefore, by successively allowing the highest priority unsolved robot to swap and become solved, every robot will eventually meet the definition of being solved. At that point, every robot can drive unobstructed to its goal and the problem is solved.

Chapter 3

Implementation and Experiments

3.1 Implementation

The algorithm was implemented and tested in MATLAB. The eight algorithms were written as detailed above, with several important differences. First, robots moved at a given velocity rather than jumping from node to node. This allowed for a smooth animation, but also necessitated the implementation of code to handle cases where robots are between nodes. The number of computations also increased significantly since the *Plan*() function was called each time a robot moved.

Another result of the asynchronous nature of robot motion is that it is possible for swapping robots to lose communication with one another. For the purposes of the algorithm, robots are approximated as being at the node closest to their actual position (referred to as the box the robot is in). If the two swapping robots are each at the outer edge of their respective boxes, the algorithm could consider them to be on adjacent nodes while they actually are at a distance of ≈ 2. Since robots can take time to turn corners and are not synchronized when they move, this could cause two swapping robots to lose communication while moving towards a branch. This problem is handled by incorporating a series of tests into *Swap*() to ensure that the swapping robots have communication during the crucial swap maneuvers. For example, if a swapping robot realizes that it needs to pick a new branch point, it first checks if it is in communication with its swap partner. If not, the swap is canceled and the robot moves back towards its goal. Similarly, robots cancel a swap if they reach their twig and do not see their partner. However, if swapping robots lose communication while driving to their goals they do *not* cancel the swap. In this way, the loss of communication is acceptable because robots will be closer to the branch point when they cancel the swap than when it began, so eventually they will reach the branch point and finish the swap.

Finally, the method implemented to handle communication between robots differs from the ideal communication network assumed by the algorithm. First, since robots are not always exactly at nodes the communication radius is defined such that a robot $r_i \in c(r)$ is within the radius of communication of r if the distance from the node closest to r_i is less than ρ away from the node closest to r. This can lead to r and r_i being in communication up to a distance of $\rho + 1$ if they are on opposite sides of their nodes, but this is acceptable because it still meets the minimum criteria for communication. More significantly, the communication network is not ideal because there is a lag as information propagates from one robot to another. Each robot chooses the most up-to-date information on other robots when making decisions, but if two robots are far apart and transferring information through several intermediaries it could take several timesteps for information to reach the other robot. While it is extremely unlikely, this could result in problems if a robot moves so that a communication network is shortened at precisely the wrong moment, leading to an important signal being lost. This risk is minimized by the fact that only adjacent robots can swap, so swapping robots should always have direct communication and not have to worry about signal delay. However, there remains a chance that the delay could cause other unanticipated problems.

3.2 Testing

The implementation was tested by running two hundred random simulations as well as several planned cases designed to test specific aspects of the code. The algorithm successfully solved one hundred problem instances with a random graph of 5x5 nodes and ten robots with randomized positions and goals (see figure 3.2.1). These simulations were meant to test the implementation in a densely populated environment, since on average there were barely more leaf nodes than the minimum requirement. The algorithm also successfully solved problems in a sparsely populated map, this time solving one hundred random problem instances with a 10x10 node graph and ten robots.

Several problem instances were specifically designed to test certain aspects of the implementation, and once again the algorithm successfully solved them all. These included a map with only one branch node and many leafs designed to test the ability of robots to choose and execute swaps, as well as one with a single long branch and a distant branch node designed to test the ability of robots to push others out of the way (see figure 3.2.2).

3.3 Results

The algorithm was tested by generating a set of ten randomized 10x10 node graphs, then running ten simulations with random robot positions for each number of robots $|R| =$

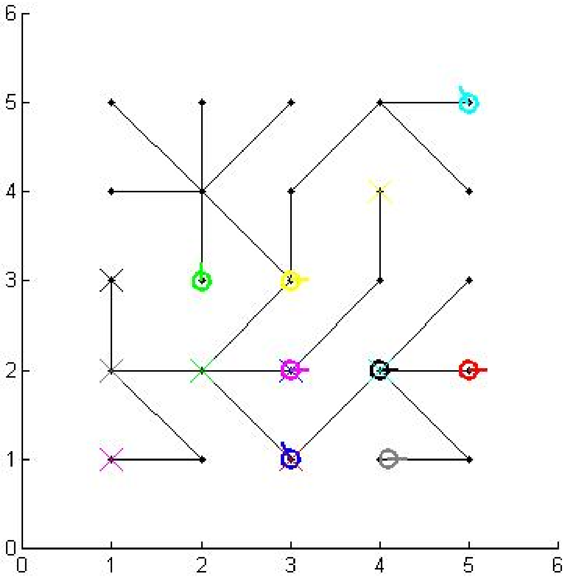
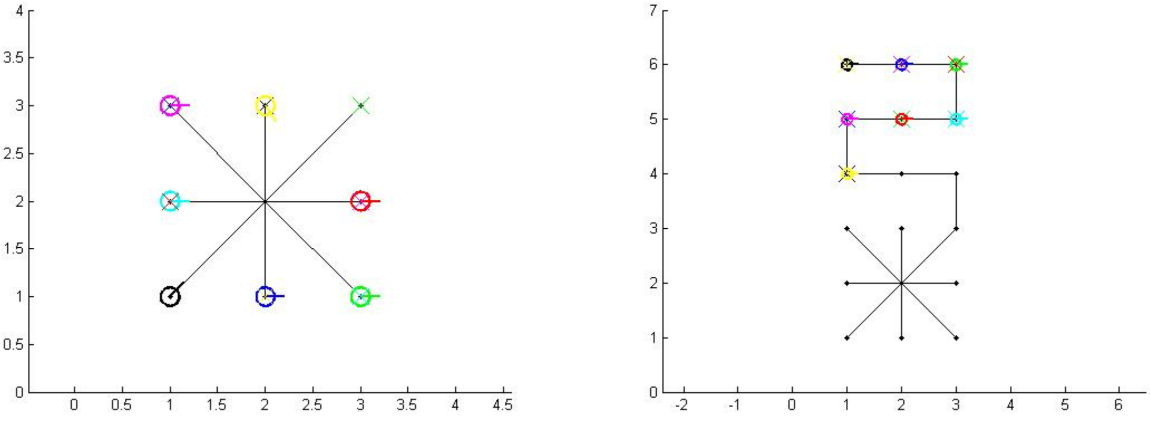


Figure 3.2.1: *Randomly generated tree and robots.* Map generated by computer code for stress testing of algorithm.



(a) “Star” (b) “Broom”

Figure 3.2.2: *Sample test cases.* Purposefully created to test algorithm on corner cases.

$5, 10, 15, 20, 30$. Data was collected on the distance each robot traveled, the number of swaps it performed, the maximum amount of time taken for one call to $Plan()$, and the total amount of time to solve the problem.

3.3.1 Path Length

Path length data was collected by tracking the difference between the total distance the robot traveled and the distance it would have traveled if no other robots were present. This data is presented in figure 3.3.1. As expected, robots are pushed further off their path as the number of robots increases. This is because robots in sparse graphs can for the most part drive directly to their goal, whereas densely populated graphs require multiple swaps and push operations. Besides having a higher average distance traveled, densely populated graphs like $|R| = 30$ have a larger variation in distance. This indicates that some robots do not have to change their course very much to accommodate other robots they encounter, whereas others get pushed to many different nodes. This behavior is likely due to the fact that the higher priority robots that get solved first do not have to move far from their goals, whereas the low priority robots are pushed for a long period of time before finally being solved. In the worst case, a robot in a problem where $|R| = 30$ can be pushed to over 70 nodes. However, it is important to note that even in this worst case scenario the robot is not traveling all over the map, but rather is being pushed back and forth between the same set of nodes. By tracking the number of distinct nodes that each robot explores, it is revealed that in this worst case for $|R| = 30$ where a robot has a path of over 70 nodes, the robot only visits a total of 12 unique nodes. It may be the case that in some applications this behavior of moving back and forth between several nodes is acceptable as long as the robot is not driving across the entire tree. Alternatively, it is possible that some optimizations could allow a robot to remain in place instead of moving back and forth.

3.3.2 Number of Swaps

Figure 3.3.2 shows the total number of swaps robots must complete before solving the problem. As the figure shows, robots perform an average of two swaps even in densely populated graphs. The maximum number of swaps seen - nine swaps when $|R| = 20$ - is still significantly below the total number of robots in the problem. This is advantageous because swaps take a long time to complete, especially if robots must travel a long distance to reach a branch node. By Lemma Three of the proof, in the worst case each robot would have to swap with every other robot in order to be solved. As the figure shows, though, robots in reality swap far less than this upper limit.

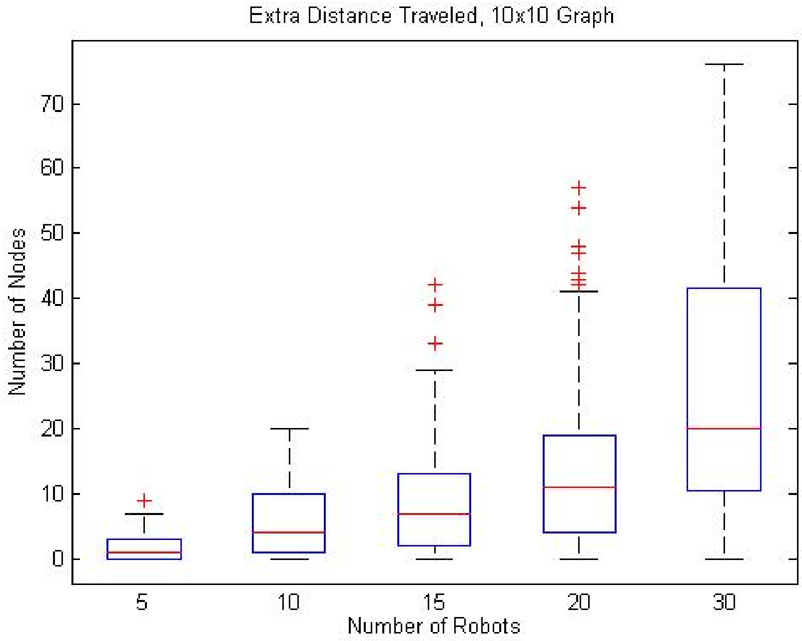


Figure 3.3.1: *Extra distance traveled.* The extra distance is defined as (Total distance traveled) - (Distance from start node to goal node), and is shown against an increasing number of robots in a 10x10 grid. The horizontal red line indicates the median number of nodes, the box encloses the 2nd and 3rd quartiles, and the dashed vertical line extends to the minimum and maximum values not judged to be outliers.

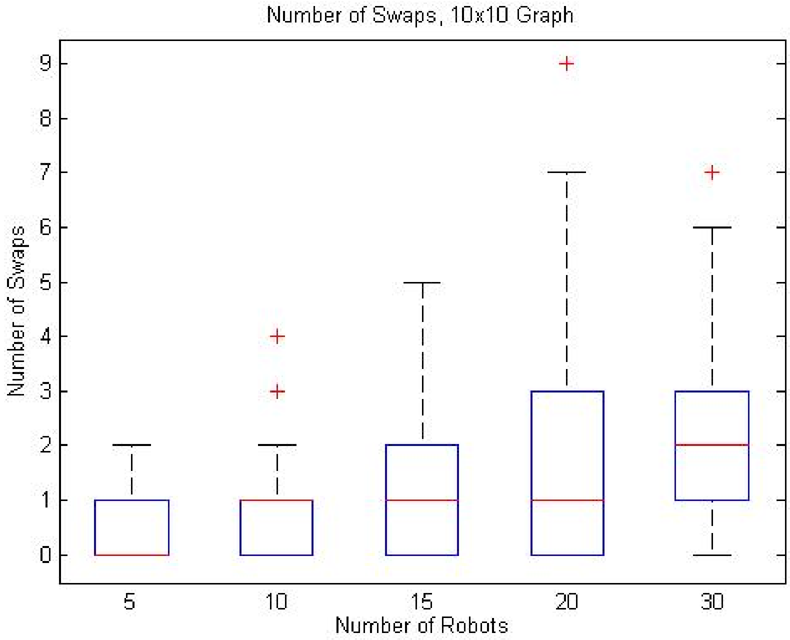


Figure 3.3.2: *Number of swaps.* Total number of swaps performed by each robot before a solution is reached, shown against an increasing number of robots on a 10x10 grid.

3.3.3 Algorithm Complexity

Figure 3.3.3 is a log-log plot showing the runtime required to solve the entire problem, as well as the maximum runtime for a single call to $Plan()$. The log-log plot shows that the runtime for the whole problem grows by three orders of magnitude as the number of robots grows by approximately one order of magnitude, meaning that the complexity of the whole problem is roughly $O(|R|^3)$. However, this is partly due to the fact that one computer is simulating all R robots, meaning that the actual complexity of the algorithm should be roughly $O(|R|^2)$. Additionally, the number of moves required to solve the problem grows as $|R|$ increases, meaning that the planning algorithm is called many more times for large $|R|$. In many cases, the computational complexity of the algorithm itself could therefore be less than $|R|^2$ if the number of calls to $Plan()$ is accounted for. This can also be seen by examining the code in Appendix B, since $Plan()$ contains one nested *for* loop that breaks when a value is found. Indeed, figure 3.3.3 shows that the maximum runtime for a single call to $Plan()$ grows by one order of magnitude as the number of robots goes from $|R| = 5$ to $|R| = 30$, so in fact it is true that the time complexity of the algorithm is between $|R|$ and $|R|^2$.

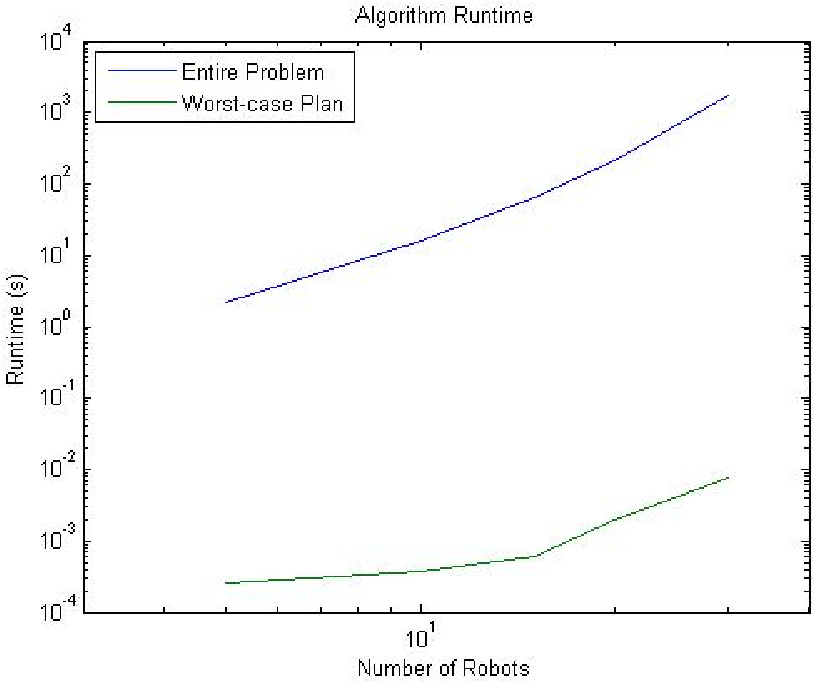


Figure 3.3.3: *Runtime.* Average runtime to reach a solution for every robot in the grid, along with the maximum runtime for one call to $Plan()$

Chapter 4

Conclusion

4.1 Summary

The Push-Swap-Wait algorithm presented here represents a reliable and complete solution to the problem of effectively coordinating the motion of many autonomous agents navigating a graph structure G in real time without reliance on global communication. The decentralized nature of the algorithm allows each robot $r \in R$ to plan its next move without full knowledge of the current state of the problem, but with a subset of information based on its current network of communication $C(r)$. Even with this limited information, it can be guaranteed that, in those cases where G can be transformed into a tree T such that $|R| \leq |L| - 1$ and the radius of communication ρ is greater than or equal to two edge lengths on this tree, a solution can be found such that all robots will reach their goals at some time t_{final}. This coordinated behavior is achieved by taking advantage of *a priori* information available to each robot (the structure of the graph G) and having them process and utilize the information in a consistent manner. Additionally, robots are able to predict the future behavior of other robots based on their reported positions, current status, directions of motion, and the locations of their goals in the tree.

While the resulting behavior of PSW may appear to be centrally organized, it is important to remember that each robotic agent is independently making decisions at each time t, and it is these individual decisions, computed continuously as the problem develops, that lead to the final solution. This is in sharp contrast to previous work on the subject, which either relied on centralized control to guarantee completeness, or implemented a decentralized algorithm that was susceptible to deadlock[11]. The fact that PSW computes a solution in real time rather than pre-computing a path can also be advantageous, as it can be more flexible and robust against disturbances. In fact, the dynamic nature of the information available to each robot in this formulation of the problem would make most pre-computed solutions useless, as they could not be guaranteed to take into account information on all of the robots on the graph. The real time nature of the algorithm also ensures that the amount of computation required at each time step is independent of the amount of time that passes before a solution is found. The Push-Swap-Wait algorithm is therefore able to scale to larger problems without incurring costs in time beyond those inherent in traveling further distances.

4.2 Suggestions for Future Work

While the Push-Swap-Wait algorithm is both complete and decentralized, there are several constraints that limit its effectiveness. First, it is by no means optimal, and in some cases robots can be forced to traverse over 70 extra nodes before reaching their goal. Second, there are some problem instances that do not meet the constraint that $|R| \leq |L| - 1$ and therefore PSW is not guaranteed to find a solution even if one exists. Finally, robots are slow to reach their goals because of the constraints that pushed robots do not move unless instructed to by a swapping robot.

Future research offers the opportunity to address these and other limitations. The algorithm could certainly get closer to the optimal solution by taking advantage of specific situations as they arise in the course of solving the problem. The simplest case would be making a more intelligent choice of twigs when swapping. This optimization would not interfere with the completeness guarantee, and has the potential to speed up swaps by relaxing the requirement that the swapping robots move back to γ_{end} and b to complete the swap. Another possibility is the case where two swaps could occur simultaneously without interfering with one another. Additional thought would need to go towards deciding exactly which conditions would allow for this behavior and how to detect when they are satisfied. It may also be possible to check for cases where non-swapping robots can continue moving towards their goals if they do not interfere with an ongoing swap. More generally, it may be useful to explore easing the restriction of robotic motion to a tree structure and investigate situations in which it is not only possible but advantageous for a robot to traverse an edge $e \notin \varepsilon$ that is not part of the tree. If done carefully, such changes could maintain the completeness of the algorithm while reducing both the time taken and the distance traveled before each robot reaches its goal.

Beyond optimizations to the algorithm, testing an implementation designed for physical robots will be necessary to determine its final usefulness. While the theoretical treatment supplied here provides guarantees on the completeness of the algorithm, those guarantees are contingent upon a certain set of requirements that may be difficult to satisfy in practical applications.

References

[1] Jean-Claude Latombe. *Robot Motion Planning.* Kluwer international series in engineering and computer science: Robotics. Kluwer Academic Publishers, 1990.

[2] Zhihua Qu, Jing Wang, and Clinton Plaistedl. A new analytical solution to mobile robot trajectory generation in the presence of moving obstacles. *IEEE Transactions on Robotics*, December 2004.

[3] Richard A. Wise and Rolf T. Rysdyk. Uav coordination for autonomous target tracking. In *Proceedings of the AIAA Guidance Navigation and Control Conference*, August 2006.

[4] Terry Huntsberger et al. Tightly-coupled coordination of multi-robot systems for mars exploration. *IEEE Transaction on Robotics and Automation: Special Issue on Multi-Robot Systems*, April 2001.

[5] Christopher M. Clark. *Dynamic Robot Networks: A Coordination Platform for Multi-Robot Systems.* PhD thesis, 2004.

[6] Erico Guizzo. Three engineers, hundreds of robots, one warehouse. *IEEE Spectrum*, July 2008.

[7] Vladimir Konyukh. Strategy of automation for underground mining. In *Strategic Technology, 2007. IFOST 2007. International Forum on*, pages 615 –618, oct. 2007.

[8] Mike Peasgood, Christopher M. Clark, and John McPhee. A complete and scalable strategy for coordinating multiple robots within roadmaps. *IEEE Transactions on Robotics*, 24.2, 2008.

[9] Ryan Luna and Kostas E. Bekris. Efficient and complete centralized multi-robot path planning. In *SOCS*, 2011.

[10] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *International Joint Conference on Artificial Intelligence*, pages 294–300, 2011.

[11] Ryan Luna. Efficient multi-robot path planning in discrete spaces. Master's thesis, University of Nevada, May 2011.

Appendix A

Data Storage and Transfer

A.1 Data Storage

Robots store many variables describing themselves, other robots, and their environment. While some are redundant, they are stored to avoid recomputing values unnecessarily. The stored values are listed in table A.1.

A.2 Data Transfer

The information passed between robots is summarized in table A.2. The robot's priority is actually redundant given the botNum and swap, and the boxNum is redundant given X and Y position, but both of these variables are used frequently enough that they merit being transferred. The rest of the variables are specifically needed by the algorithm at some point. Note that the path transmitted from robot to robot is different from the path variable that each robot stores about itself in that the path in knowledge begins at the robot's last node.

Variable	Explanation
botNum	Robot's ID number (priority)
swap	ID of this robot's swap partner
priority	Maximum priority of this robot and swap partner
status	State of this robot when swapping or pushed
leader	Is this robot the leader in the swap?
visited	List of unavailable branch nodes already visited
oldTwig	γ_{end}
otherSwap	Swapping robot to wait for
solved	Is this robot solved?
solvedBots	List of all solved robots seen so far
boxNum	Node nearest this robot's current position
path	Set of nodes this robot is planning to take
last	Index of last node in path that this robot was on
xPos	X coordinate of this robot's position
yPos	Y coordinate of this robot's position
xGoal	X coordinate of this robot's goal
yGoal	Y coordinate of this robot's goal
goalNum	Node number of this robot's goal
theta	Orientation (counter-clockwise from right, in rad)
time	Simulation time
map	Data type storing environment (graph and tree)
color	Used for drawing this robot in the animation
knowledge	Information on all other robots in $C(r)$

Table A.1: *Stored Data*

Variable	Explanation
botNum	Robot's ID number
xPos	X coordinate of robot's position
yPos	Y coordinate of robot's position
xGoal	X coordinate of robot's goal
yGoal	Y coordinate of robot's goal
priority	Maximum priority of robot and swap partner
path	Set of nodes robot is planning to take
boxNum	Node nearest robot's current position
swap	Robot's swap partner
status	State of robot when swapping or pushed
solved	Is this robot solved?
TimeOfReceipt	Simulation time this data was generated

Table A.2: *Transferred Data*

Appendix B

MATLAB Code

B.1 animation.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decentralized and Complete Multi-Robot Motion Planning        %
%                                           in Confined Spaces  %
% Dexter Scobee and Adam Wiktor                                 %
% Top-level animation code:                                     %
%                                                               %
% Initializes each robot. Next, calls functions to pass         %
% messages between robots, have them plan their paths and move, %
% and draw the map and their current location. Continues        %
% looping until all robots have reached their goal.             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all
close all
clc

dt = 0.1; % time step
radius = 2; % radius of communication

% initialize the map and robots
map = MapMaker('maptest.txt', radius);
bot = BotMaker('MapTestBots.txt',map);
numBots = length(bot);

done = 0;
while done == 0
    clf
    hold on
    % map.draw();
    map.drawTree();
    done = 1;

    % each robot communicates with neighbors
    for i=1:numBots
        bot(i).getInfo(checkNeighbors(i, bot));
    end

    % each robot moves
    for i=1:numBots
        botDone = bot(i).move(dt);
        bot(i).draw();
        if (bot(i).solved ~= 1) || (botDone ~= 1)
            done = 0; % loop again if any robot is not done
        end
    end
    axis equal

```

```
    hold off

    pause(dt/10);
end % while
```

B.2 Robot.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decentralized and Complete Multi-Robot Motion Planning        %
%                                           in Confined Spaces  %
% Dexter Scobee and Adam Wiktor                                 %
% Robot datatype:                                               %
%                                                               %
% Datatype to represent one robot, along with methods for       %
% moving the robot toward its goal and avoiding collisions.     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Robot < handle

    properties
        % Swap parameters
        botNum = 0;     % robot's ID number
        swap = 0;       % ID of the bot this bot is swapping with
                        % (0 if no swap)
        priority = 0;   % priority of swap pair OR depth of goal node
        status = 0;     % 1 if moving, -1 if can't move, 0 if could move
        leader = 0;     % is this bot the leader in the swap
        visited = 0;    % branch nodes already visited
        oldTwig = 0;    % twig that bot came from in a swap
        otherSwap = 0;  % the pair of swappers that you're waiting for
        solved = 0;     % has this bot (and all higher priority bots)
                        % been properly sorted
        solvedBots =0;  % list of solved robots seen

        % Bot position and goal
        boxNum = 0;     % node nearest the robot's current position
        path = 0;       % array of nodes for robot to travel along
        last = 0;       % index in path of the last node the robot was on
        xPos = 0;       % x position
        yPos = 0;       % y position
        xGoal = 0;      % x coordinate of goal node
        yGoal = 0;      % y coordinate of goal node
        goalNum = 0;    % node number of goal
        theta = 0;      % orientation (ccw from right)

        % Other parameters
        time = 0;       % simulation time
        map = 0;        % map
        color = 'b';
        knowledge = struct('botNum', 0, 'xPos', 0,...
            'yPos', 0,'xGoal', 0, 'yGoal', 0,...
            'priority', 0, 'path', 0, 'boxNum', 0,...
            'swap', 0, 'status', 0, 'solved', 0, 'ToR', 0);
    end

    properties (Constant = true)
        radius = 0.1; % robot's radius when drawing
        turn = 10; % turning speed (rad/s)
        vel = 1;  % velocity (units/s)
    end

    methods

        %*********************************************************%
        % Constructor. Takes a map, the robot's ID number,        %
        % current x-y position, goal x-y position, and color as   %
        % arguments and returns a robot object. The color         %
        % argument is optional and defaults to blue.              %
        %*********************************************************%
        function bot = Robot(map, botNum, x, y, xdest, ydest, color)
            %initialize variables
```

```
            bot.botNum = botNum;
            bot.xPos = x;
            bot.yPos = y;
            bot.xGoal = xdest;
            bot.yGoal = ydest;
            bot.map = map;
            bot.time = 1;
            bot.swap = 0;
            bot.status = 0;
            bot.solved = 0;

            bot.goalNum = map.xy2node(xdest,ydest);
            bot.priority = map.nodeDepth(bot.goalNum);

            if nargin > 6
                bot.color = color;
            end
            bot.boxNum = map.xy2node(x,y);
            bot.path = map.makePath(bot.boxNum, bot.goalNum);
            bot.last = 1;
            % bot.initialize(map.xy2node(xdest,ydest));
            bot.botNum = bot.priority;
        end

        %*********************************************************%
        % signal:                                                 %
        % Pass bot’s current state and knowledge to a neighboring %
        % robot.                                                  %
        %*********************************************************%
        function signal = signal(bot)
            % pass bot’s current state
            signal(1) = struct(...
                ’botNum’, bot.botNum,...
                ’xPos’, bot.xPos,...
                ’yPos’, bot.yPos,...
                ’xGoal’, bot.xGoal,...
                ’yGoal’, bot.yGoal,...
                ’priority’, bot.priority,...
                ’path’, bot.path(bot.last:length(bot.path)),...
                ’boxNum’, bot.boxNum,...
                ’swap’, bot.swap,...
                ’status’, bot.status,...
                ’solved’, bot.solved,...
                ’ToR’, bot.time);

                signal(2:length(bot.knowledge)+1) = bot.knowledge;
        end

        %*********************************************************%
        % getInfo:                                                %
        % Receive data from neighboring robots and store it to    %
        % bot’s knowledge.                                        %
        %*********************************************************%
        function getInfo(bot, data)
            bot.knowledge = data;
        end

        %*********************************************************%
        % draw:                                                   %
        % Draw bot as a circle with a line indicating the         %
        % direction bot is facing.                                %
        %*********************************************************%
        function draw(bot)
            % Draw the robot at its current position
            alpha = 0:0.1:2*pi;
            x = bot.xPos + bot.radius*cos(alpha);
            y = bot.yPos + bot.radius*sin(alpha);
            plot(x,y,’Color’,bot.color, ’LineWidth’,2);
            x = [bot.xPos bot.xPos+2*bot.radius*cos(bot.theta)];
            y = [bot.yPos bot.yPos+2*bot.radius*sin(bot.theta)];
            plot(x,y,’Color’,bot.color,’LineWidth’,2);

            % Draw the goal
            plot(bot.xGoal,bot.yGoal,’x’,...
```

```
                'Color',bot.color,'MarkerSize',15);
            axis square
        end


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % These methods change the robot's positon / orientation %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


        %*******************************************************%
        % move:                                                 %
        % Plans a path and moves bot along it. Takes the time   %
        % step as an argument and returns 1 if bot has reached  %
        % the goal or 0 otherwise.                              %
        %*******************************************************%
        function done = move(bot, dt)
            % plan bot's path
            bot.time = bot.time + 1;
            done = bot.plan();
            done = bot.checkLockBox(done);

            if done == 1 % reached goal
                return;
            end

            if done == -1 % do not move
                done = 0;
                return;
            end

            if bot.last >= length(bot.path) - 1
                return;
            end

            next = bot.path(bot.last+1);
            [nextX nextY] = bot.map.node2xy(next);
            dx = nextX - bot.xPos;
            dy = nextY - bot.yPos;
            if (dx == 0) && (dy == 0)
                nextTheta = bot.theta;
            else
                nextTheta = atan2(dy,dx);
            end

            dTheta = nextTheta - bot.theta;
            % check for shortest turning direction
            if abs(dTheta - 2*pi) < abs(dTheta)
                dTheta = dTheta - 2*pi;
            else
                if abs(dTheta + 2*pi) < abs(dTheta)
                    dTheta = dTheta + 2*pi;
                end
            end

            % turn
            if abs(dTheta) > 1e-14
                if abs(dTheta) <= bot.turn*dt % close enough
                    bot.theta = nextTheta;
                    return;
                end
                bot.theta = bot.theta + bot.turn*dt*sign(dTheta);
                return;
            end

            % move
            if (abs(dx) <= abs(bot.vel*dt*cos(bot.theta))) && ...
                    (abs(dy) <= abs(bot.vel*dt*sin(bot.theta)))
                % close enough
                bot.xPos = nextX;
                bot.yPos = nextY;
                bot.last = bot.last + 1;
            else
                bot.xPos = bot.xPos + bot.vel*dt*cos(bot.theta);
```

```
                bot.yPos = bot.yPos + bot.vel*dt*sin(bot.theta);
            end

            % check if bot has entered a different boxNum
            if next ~= bot.boxNum
                nextDist = bot.map.xyDist(bot.xPos, bot.yPos,...
                    nextX, nextY);
                [lastX lastY] = bot.map.node2xy(bot.path(bot.last));
                lastDist = bot.map.xyDist(bot.xPos, bot.yPos,...
                    lastX, lastY);
                if nextDist < lastDist
                    bot.boxNum = next;
                end
            end

        end % function move
    end % methods

    methods (Access = private)

        %*********************************************************%
        % checkLockBox:                                           %
        % Check for lock box violations. If one is found, stop    %
        % moving. Otherwise continue as planned                   %
        %*********************************************************%
        function done = checkLockBox(bot,done)
            % check for lock box violations
            if bot.knowledge(1).botNum ~= 0 % know about other robots
                for i = 1:length(bot.knowledge)
                    if bot.knowledge(i).boxNum == bot.path(bot.last+1)
                        % another robot is at bot’s next node
                        done = -1;
                        return;
                    end
                end
            end
        end

        %*********************************************************%
        % plan:                                                   %
        % Plan a path from bot’s current position to the goal,    %
        % avoiding collisions if necessary. Return 1 if bot has   %
        % reached the goal, -1 if bot should stop moving, and 0   %
        % otherwise.                                              %
        %*********************************************************%
        function done = plan(bot)
            done = 0;

            % get this bots solved state
            bot.solved = bot.checkSolved();
            waitForSwappers = 0;
            getPushed = 0;

            % get info on other bots in network
            if bot.knowledge(1).botNum ~= 0
                for i = 1:length(bot.knowledge)
                    % check if there was a high-priority swap to wait for
                    if bot.knowledge(i).priority == bot.otherSwap || ...
                            (bot.knowledge(i).swap ~= 0 && ...
                            bot.knowledge(i).priority < bot.otherSwap)
                        bot.otherSwap = 0;
                    end

                    % check if anyone else is waiting
                    % for swappers to return
                    if bot.knowledge(i).status == -2
                        waitForSwappers = 1;
                    end

                    % Only get pushed if you’re lower priority than
                    % the swappers OR if you’re already solved
                    if bot.knowledge(i).swap ~= 0 && ...
                            (bot.botNum > bot.knowledge(i).priority || ...
                            bot.solved == 1)
```

```
                    getPushed = 1;
                end

                % Check for solved bots to add to solvedBots
                goal = bot.map.xy2node(bot.knowledge(i).xGoal,...
                                           bot.knowledge(i).yGoal);
                if bot.knowledge(i).solved == 1
                    if bot.solvedBots == 0
                        bot.solvedBots = goal;
                        continue;
                    end
                    foundGoal = 0;
                    for j=1:length(bot.solvedBots)
                        if bot.solvedBots(j) == goal
                            foundGoal = 1;
                            break;
                        end

                        if bot.map.nodeDepth(bot.solvedBots(j)) > ...
                                bot.map.nodeDepth(goal)
                            solvedList = ...
                                zeros(1,length(bot.solvedBots)+1);
                            if j > 1
                                solvedList(1:j-1) = ...
                                    bot.solvedBots(1:j-1);
                            end
                            solvedList(j) = goal;
                            solvedList(j+1:end) = ...
                                bot.solvedBots(j:end);
                            bot.solvedBots = solvedList;
                            foundGoal = 1;
                            break;
                        end
                    end
                    if ~foundGoal
                        bot.solvedBots(end+1) = goal;
                    end
                elseif bot.solvedBots ~= 0
                    % remove any unsolved bots in solvedBots
                    unsolvedBot = 0;
                    for j=length(bot.solvedBots):-1:1
                        if bot.map.nodeDepth(goal) <= ...
                                bot.map.nodeDepth(bot.solvedBots(j))
                            unsolvedBot = j;
                        else
                            break;
                        end
                    end
                    if unsolvedBot > 1
                        bot.solvedBots = ...
                            bot.solvedBots(1:unsolvedBot-1);
                    elseif unsolvedBot == 1
                        bot.solvedBots = 0;
                    end
                end
            end
        end

        if bot.otherSwap ~= 0
            done = bot.getStopped();
            return;
        end

        if bot.status == -2
            bot.status = 0;
        end


        if waitForSwappers
            done = -1;
            return;
        end

        % check if there is a higher priority bot to swap with
```

```matlab
            [swapBot1 swapBot2] = bot.checkSwap();

            if bot.botNum == swapBot1
                if bot.swap ~= swapBot2
                    % suppress new swaps if below a solved bot
                    belowSolved = 0;
                    for i = 1:length(bot.solvedBots)
                        if bot.solvedBots == 0
                            break;
                        end
                        goal = bot.solvedBots(i);
                        goalPriority = bot.map.nodeDepth(goal);
                        if bot.botNum < goalPriority
                            break;
                        end

                        node = bot.boxNum;
                        nodePriority = bot.map.nodeDepth(node);
                        while nodePriority < goalPriority
                            if goal == bot.map.tree(node)
                                belowSolved = 1;
                            end
                            node = bot.map.tree(node);
                            nodePriority = bot.map.nodeDepth(node);
                        end
                    end

                    if ~belowSolved
                        bot.resetSwap();
                        bot.priority = min([swapBot1 swapBot2]);
                        bot.swap = swapBot2;
                    else
                        getPushed = 0;
                        bot.resetSwap();
                    end
                end
            elseif bot.botNum == swapBot2
                if bot.swap ~= swapBot1
                    % suppress new swaps if below a solved bot
                    belowSolved = 0;
                    for i = 1:length(bot.solvedBots)
                        if bot.solvedBots == 0
                            break;
                        end
                        goal = bot.solvedBots(i);
                        goalPriority = bot.map.nodeDepth(goal);
                        if bot.botNum < goalPriority
                            break;
                        end

                        node = bot.boxNum;
                        nodePriority = bot.map.nodeDepth(node);
                        while nodePriority < goalPriority
                            if goal == bot.map.tree(node)
                                belowSolved = 1;
                            end
                            node = bot.map.tree(node);
                            nodePriority = bot.map.nodeDepth(node);
                        end
                    end

                    if ~belowSolved
                        bot.resetSwap();
                        bot.priority = min([swapBot1 swapBot2]);
                        bot.swap = swapBot1;
                    else
                        getPushed = 0;
                        bot.resetSwap();
                    end
                end
            elseif bot.swap ~= 0
                bot.resetSwap();
            end

```

```
            % if I need to swap
            if bot.swap ~= 0 && bot.swap ~= bot.botNum
                % enter swap mode
                done = bot.getSwapped();
                return;
            end

            % if I don't need to swap

            if bot.swap == bot.botNum
                bot.status = 0;
                done = 0;
                return;
            end


            if getPushed
                % enter get pushed mode
                done = bot.getPushed();
                return;
            end


            % in normal mode

            % want to give right of way to bots on higher priority branch
            % (this reverses the 'push')
            if bot.knowledge(1).botNum ~= 0
                for i = 1:length(bot.knowledge)
                    if bot.boxNum > bot.knowledge(i).boxNum
                        for j = 1:length(bot.knowledge(i).path)
                            if bot.knowledge(i).path(1) ~= ...
                                    bot.knowledge(i).boxNum
                                continue;
                            end
                            if bot.knowledge(i).path(j) == bot.boxNum
                                break;
                            end
                            if bot.knowledge(i).path(j) == ...
                                    bot.path(bot.last+1)
                                done = -1;
                                return;
                            end
                        end
                    end
                end
            end

            % at this point, bot is not involved in any swaps
            [xDest yDest] = bot.map.node2xy(bot.path(end));
            if (bot.xPos == xDest) && (bot.yPos == yDest)
                % reached last node on current path
                if (bot.xPos == bot.xGoal) && (bot.yPos == bot.yGoal)
                    done = 1; % reached goal
                    return;
                end
                % plan a new path to the goal
                bot.initialize(bot.map.xy2node(bot.xGoal,bot.yGoal));
            end
        end % function plan

        %*********************************************************%
        % checkSolved:                                            %
        % Checks to see if a robot and subtree are solved         %
        %*********************************************************%

        function solved = checkSolved(bot)
            solved = bot.solved;
            if (bot.xPos == bot.xGoal) && (bot.yPos == bot.yGoal)
                solved = 1; % reached goal
            end
            if bot.knowledge(1).botNum ~= 0
               for i=1:length(bot.knowledge)
                  if bot.knowledge(i).botNum < bot.botNum ...
```

```
                          && bot.knowledge(i).solved == 0
                      solved = 0;
                  end
               end
            end
        end

        %*********************************************************%
        % checkSwap:                                              %
        % Checks to see if a robot needs to swap                  %
        %*********************************************************%

        function [swapBot1 swapBot2] = checkSwap(bot)
            if bot.knowledge(1).botNum == 0
                swapBot1 = 0;
                swapBot2 = 0;
                return;
            end

            botList = bot.signal();
            botRank = zeros(size(botList));
            botNumList = zeros(size(botList));
            for i=1:length(botList)
                botNumList(i) = botList(i).botNum;
                botRank(i) = botList(i).priority;
            end

            % first sort by botNum so this bot is in the right order
            [~,I] = sort(botNumList);
            botList = botList(I);
            botRank = botRank(I);

            % next sort by priority
            [~,I] = sort(botRank);
            botList = botList(I);

            for i = 1:length(botList)
                % if highest priority robot who needs to swap is already
                % swapping with someone, let them continue
                if (botList(i).swap ~= 0) && ...
                        (botList(i).swap ~= botList(i).botNum)
                    swapBot1 = botList(i).botNum;
                    swapBot2 = botList(i).swap;
                    return;
                end

                for j = i+1:length(botList)
                    % j must be adjacent to i for them to swap
                    if ~bot.checkAdjacent(botList(i),botList(j))
                        continue;
                    end

                    [split1, split2, onPath1G, onPath2G] = ...
                        bot.checkSplit(botList(i),botList(j));
                    if (split1 && split2) || (split1 && onPath2G) || ...
                            (split2 && onPath1G) || ...
                            (botList(i).swap == botList(i).botNum && ...
                             split1 && botList(j).status == -1)
                        % assumes knowledge is ordered
                        % with highest priority bots first
                        swapBot1 = botList(i).botNum;
                        swapBot2 = botList(j).botNum;
                        return;
                    end
                end

                if botList(i).solved ~= 1
                    swapBot1 = botList(i).botNum;
                    swapBot2 = botList(i).botNum;
                    return;
                end
            end

            swapBot1 = 0;
```

```
            swapBot2 = 0;
        end

        %**********************************************************%
        % checkSplit:                                              %
        % Checks to see if two robots split eachother from their   %
        % goals.
        % splitX = 1 if bot(X) is separted from his goal           %
        % onPathXG = 1 if bot(X)’s goal is on the path between     %
        % bot(Y) and bot(Y)’s goal                                 %
        %**********************************************************%

        function [split1, split2, onPath1G, onPath2G] = ...
                checkSplit(bot, bot1, bot2)

            bot1Goal = bot.map.xy2node(bot1.xGoal, bot1.yGoal);
            bot2Goal = bot.map.xy2node(bot2.xGoal, bot2.yGoal);

            pathBot = bot.map.makePath(bot1.boxNum, bot2.boxNum);
            pathGoal1 = bot.map.makePath(bot1.boxNum, bot1Goal);
            pathGoal2 = bot.map.makePath(bot1.boxNum, bot2Goal);

            if (pathBot(2) ~= pathGoal2(2)) || (bot1.boxNum == bot2Goal)
                split2 = 1;
            else
                split2 = 0;
            end

            if ~isempty(find(pathGoal1 == bot2Goal, 1))
                onPath2G = 1;
            else
                onPath2G = 0;
            end

            pathBot = bot.map.makePath(bot2.boxNum, bot1.boxNum);
            pathGoal1 = bot.map.makePath(bot2.boxNum, bot1Goal);
            pathGoal2 = bot.map.makePath(bot2.boxNum, bot2Goal);

            if (pathBot(2) ~= pathGoal1(2)) || (bot2.boxNum == bot1Goal)
                split1 = 1;
            else
                split1 = 0;
            end

            if ~isempty(find(pathGoal2 == bot1Goal, 1))
                onPath1G = 1;
            else
                onPath1G = 0;
            end
        end

        %**********************************************************%
        % checkAdjacent:                                           %
        % Check if two robots are on adjacent nodes                %
        %**********************************************************%

        function isAdjacent = checkAdjacent(bot, bot1, bot2)
            % check if bot2 is on bot1’s parent
            if bot.map.tree(bot1.boxNum) == bot2.boxNum
                isAdjacent = 1;
                return;
            end

            %check if bot1 is on bot2’s parent
            if bot.map.tree(bot2.boxNum) == bot1.boxNum
                isAdjacent = 1;
                return;
            end

            isAdjacent = 0;
        end

        %**********************************************************%
        % getPushed:                                               %
```

```
        % Robot moves out of the way of swapping bots             %
        %*********************************************************%
        function done = getPushed(bot)

            if bot.knowledge(1).botNum ~= 0
                % find the swapping pair with highest priority
                swapCheck = 0;
                for i=1:length(bot.knowledge)
                    if bot.knowledge(i).swap ~= 0
                        if swapCheck == 0 % first swapping pair found
                            swapBots(1) = bot.knowledge(i);
                            swapCheck = 1;
                        elseif bot.knowledge(i).botNum == swapBots(1).swap
                            swapBots(2) = bot.knowledge(i);
                        end
                    end
                end

                % set otherSwap to keep track of highest priority swapBot
                if swapCheck ~= 0
                    % if swapBots(1) is higher priority bot and bot is
                    % in direct communication with swapBots(1)
                    if swapBots(1).botNum == swapBots(1).priority &&...
                            swapBots(1).ToR == bot.time - 1
                        bot.otherSwap = 0;
                        if bot.botNum == 14
                            a=1;
                        end
                        % check if swapBots(1) is at a parent of his goal
                        node = bot.map.xy2node(swapBots(1).xGoal,...
                            swapBots(1).yGoal);

                        atParent = 0;
                        foundFirstNode = 0;
                        while node ~= bot.map.root
                            if (node == swapBots(1).path(1)) || ...
                               (node == swapBots(1).path(2)) &&...
                               ~foundFirstNode
                                foundFirstNode = 1;
                            end
                            if (node == swapBots(1).path(1)) || ...
                               (node == swapBots(1).path(2)) &&...
                               foundFirstNode
                                atParent = 1;
                                break;
                            end

                            node = bot.map.tree(node);
                        end

                        % if swapBots(1) is heading up the tree
                        if bot.map.tree(swapBots(1).path(1)) == ...
                                swapBots(1).path(2) && ...
                                swapBots(1).path(1) ~= bot.map.root
                            if atParent
                                bot.otherSwap = swapBots(1).botNum;
                            end

                        else % if swapBots(1) is not heading up the tree
                            if ~atParent
                                bot.otherSwap = swapBots(1).botNum;
                            end
                        end
                    end
                end

                % add pushed robots to swapBots array
                for i=1:length(bot.knowledge)
                    if (bot.knowledge(i).status == 1) ...
                            && (bot.knowledge(i).swap == 0)
                        swapBots(end+1) = bot.knowledge(i);
                        swapCheck = 1;
                    end
                end
```

```

                if swapCheck == 0
                    bot.status = 0;
                    done = 0;
                    return
                end

                freeNodes = zeros(bot.map.n,1);
                % check if bot is on the path of the swapping robots
                for i=1:length(swapBots)
                    if ~isempty(find(swapBots(i).path(2:end) ==...
                            bot.boxNum,1))
                        for j=1:length(swapBots(i).path)
                            if swapBots(i).path(1) ~= swapBots(i).boxNum
                                continue;
                            end
                            freeNodes(swapBots(i).path(j)) = 1;

                            if swapBots(i).path(j) == bot.boxNum
                                break;
                            end
                        end
                    end
                end


                % check if bot has reached destination
                [xDest yDest] = bot.map.node2xy(bot.path(end));
                if bot.xPos == xDest && bot.yPos == yDest
                    bot.status = 0;
                end

                if freeNodes(bot.boxNum) == 0  % bot is not in the way
                    bot.initialize(bot.boxNum);
                    if bot.status == 1
                        done = 0;
                        return;
                    else
                        bot.status = 0;
                        done = -1;
                        return;
                    end
                end

                % check for nodes that are free
                for i=1:length(bot.knowledge)
                    if bot.knowledge(i).status == -1 || ...
                            bot.knowledge(i).status == 2
                        freeNodes(bot.knowledge(i).boxNum) = 1;
                    end
                end

                index = 1;
                % add all children to neighbors
                for i=1:bot.map.n
                    if (bot.map.tree(i) == bot.boxNum) && ...
                            (i ~= bot.map.root)
                        neighbors(index) = i;
                        index = index + 1;
                    end
                end
                % add parent to neighbors
                if (bot.boxNum ~= bot.map.root)
                    neighbors(index) = bot.map.tree(bot.boxNum);
                end

                % sort neighbors based on node depth
                [~,I] = sort(-bot.map.nodeDepth(neighbors));
                neighbors = neighbors(I);



                dest = 1;
                if bot.status == 1
```

```
                if freeNodes(bot.path(bot.last+1)) == 0
                    % current path is still good, continue moving
                    done = 0;
                    return;
                else % need to find new destination
                    currentDest = find(neighbors ==...
                        bot.path(bot.last+1),1);
                    if ~isempty(currentDest)
                        dest = currentDest + 1;
                    end
                end
            end

            while dest <= length(neighbors)
                if freeNodes(neighbors(dest)) == 0
                    bot.initialize(neighbors(dest));
                    bot.status = 1;
                    done = 0;
                    return;
                end
                dest = dest + 1;
            end

            % no free nodes available
            bot.status = -1;
            bot.initialize(bot.boxNum);
            done = -1;
            return;
        end
    end

    %*********************************************************%
    % getStopped:                                             %
    % Robot moves out of the way of swapping bots             %
    %*********************************************************%
    function done = getStopped(bot)
        bot.status = -2;
        done = -1;
        return;
    end

    %*********************************************************%
    % getSwapped:                                             %
    % Robot swaps with another robot                          %
    %*********************************************************%
    function done = getSwapped(bot)
        bot.solved = 0;
        if bot.status == 0 % find a new branch point
            done = bot.startSwap();
            return;
        end

        if bot.status == 1
            done = bot.continueSwap();
            return;
        end

        if bot.status == 2
            done = bot.endSwap();
            return;
        end
    end

    %*********************************************************%
    % startSwap:                                              %
    % Initialize the swap, picking a branch point and planning%
    % a path.                                                 %
    %*********************************************************%
    function done = startSwap(bot)
        foundPartner = 0;
        bot.leader = 0;
        bot.oldTwig = 0;
        count = bot.map.findBranches();

```

```
            for i=2:length(bot.visited)
                if bot.visited(i) ~= 0
                    count(bot.visited(i)) = 0;
                end
            end

            priorityBot = bot; % bot with higher ID number
            otherBot = bot;

            if bot.knowledge(1).botNum ~= 0
                for i=1:length(bot.knowledge)
                    if bot.knowledge(i).botNum == bot.swap
                        foundPartner = 1;
                        otherBot = bot.knowledge(i);
                    end

                    % find lower ID bot
                    if (bot.swap == bot.knowledge(i).botNum) && ...
                            (bot.swap < bot.botNum)
                        priorityBot = bot.knowledge(i);
                        otherBot = bot;
                    end
                end
            end

            %revert to normal mode if partner not found during
            %branch reassignment
            if ~foundPartner
                bot.resetSwap();
                done = 0;
                return;
            end

            minLength = inf;
            goBranch = 0;
            noTwig = 0;

            for i=1:length(count)
                if count(i) >= 3 % node is a viable branch
                    route = bot.map.makePath(priorityBot.boxNum,i);
                    if length(route) < minLength
                        minLength = length(route);
                        goBranch = i;
                        if length(route) > 2
                            noTwig = route(end-2);
                        else
                            noTwig = 0;
                        end
                    end
                end
            end

            %didn't find available branch; reset visited and try again
            if goBranch == 0
                bot.visited = 0;
                done = 0;
                return;
            end

            %%%%if here, you are in contact with your partner, and
            %%%% there are branch points available

            bot.visited(end+1) = goBranch;

            % remove the next "parent branch" from visited
            branch = goBranch;
            while branch ~= bot.map.root
               found = find(bot.visited == bot.map.tree(branch),1);
               if ~isempty(found)
                   bot.visited(found) = 0;
                   break;
               end
               branch = bot.map.tree(branch);
            end
```

```

            %if priorityBot was on the chosen branch point, look at
            %otherBot
            route = bot.map.makePath(otherBot.boxNum, goBranch);
            if noTwig == 0
                if length(route) > 2
                    noTwig = route(end-2);
                end
            else
                if length(route) > 2
                    %if robots are on opposite sides of a branch point
                    if route(end-2) ~= noTwig
                        noTwig(2) = route(end-2);
                        %lower priority bot waits on his twig
                        if bot.botNum == otherBot.botNum
                            bot.initialize(noTwig(2));
                            bot.leader = 1;
                            bot.status = 1;
                            bot.oldTwig = noTwig(1);
                            done = 0;
                            return;
                        end
                    end
                end
            end

            % build list of twigs off of branch point
            % add parent to list of twigs
            if goBranch ~= bot.map.root && ...
                    isempty(find(noTwig == bot.map.tree(goBranch), 1))
                twigList = bot.map.tree(goBranch);
            else
                twigList = [];
            end
            for j=1:bot.map.n
                if bot.map.tree(j) == goBranch && ...
                        isempty(find(noTwig==j,1)) && ...
                        j ~= bot.map.root
                    twigList(end+1) = j;
                end
            end

            if ~isempty(find(route(2:end)==priorityBot.boxNum,1))
                %priorityBot is leader
                if bot.botNum == priorityBot.botNum % bot is priorityBot
                    bot.initialize(twigList(1));
                    done = 0;
                    bot.leader = 1;
                    bot.status = 1;
                    bot.oldTwig = noTwig(end);
                    return;
                else % non-split case, pick second twig
                    bot.initialize(twigList(2));
                    done = 0;
                    bot.leader = -1;
                    bot.status = 1;
                    bot.oldTwig = noTwig(1);
                    return;
                end

            else %otherBot is leader
                if bot.botNum == otherBot.botNum % bot is otherBot
                    bot.initialize(twigList(1));
                    done = 0;
                    bot.leader = 1;
                    bot.status = 1;
                    bot.oldTwig = noTwig(1);
                    return;
                elseif length(noTwig) > 1 % bot is priorityBot
                    % split case, pick first twig
                    bot.initialize(twigList(1));
                    done = 0;
                    bot.leader = -1;
                    bot.status = 1;
```

```
                bot.oldTwig = noTwig(1);
                return;
            else % non-split case, pick second twig
                bot.initialize(twigList(2));
                done = 0;
                bot.leader = -1;
                bot.status = 1;
                bot.oldTwig = noTwig(1);
                return;
            end
        end
    end

    %*********************************************************%
    % continueSwap:                                           %
    % Bot continues to move unless it has reached its         %
    % destination or there are other bots blocking the path   %
    %*********************************************************%
    function done = continueSwap(bot)
        done = 0;
        foundPartner = 0;

        if bot.knowledge(1).botNum ~= 0
            for i=1:length(bot.knowledge)
                if bot.knowledge(i).botNum == bot.swap
                    foundPartner = 1;
                    swapPartner = bot.knowledge(i);
                    break;
                end
            end
        end

        % if regaining communication with partner, need to
        % reinitialize all swap parameters (oldTwig, visited, etc.)
        % becuase he may have already cleared them
        if foundPartner && swapPartner.swap ~= bot.botNum
            bot.visited = 0;
            bot.status = 0;
            % call immediately to avoid confusion
            done = bot.startSwap();
            return;
        end

        % if partner is picking a new branch while he knows
        % he's swapping with you
        if foundPartner && swapPartner.status == 0 ...
                && swapPartner.swap == bot.botNum
            bot.leader = 0;
            bot.status = 0;
            % call start swap right away so partner
            % doesn't misinterpret status = 0
            done = bot.startSwap();
            return;
        end


        % check if bot has reached destination twig
        [xDest yDest] = bot.map.node2xy(bot.path(end));
        if bot.xPos == xDest && bot.yPos == yDest
            bot.status = 2;
            done = -1;
            return;
        end

        % if not at destination, make sure that destination
        % is still available, else, choose a different twig.
        if bot.knowledge(1).botNum ~= 0
            for i = 1:length(bot.knowledge)
                if (bot.knowledge(i).status == -1 || ...
                        bot.knowledge(i).status == 2) && ...
                        bot.knowledge(i).boxNum == bot.path(end)

                    % build list of twigs off of branch point
                    % add parent to list of twigs
```

```
                goBranch = bot.path(end-2);
                if goBranch ~= bot.map.root
                    twigList = bot.map.tree(goBranch);
                else
                    twigList = [];
                end
                for j=1:bot.map.n
                    if bot.map.tree(j) == goBranch && ...
                            j ~= bot.map.root
                            %isempty(find(noTwig==j,1)) && ...
                        twigList(end+1) = j;
                    end
                end

                % current twig should only appear once
                % in twigList
                current = find(twigList == bot.path(end));
                if current < length(twigList)
                    if twigList(current+1) ~= bot.oldTwig
                        bot.initialize(twigList(current+1));
                        return;
                    elseif current+1 < length(twigList)
                        bot.initialize(twigList(current+2));
                        return;
                    end
                end
                % either no more twigs to check, or remaining twig
                % is oldTwig
                % move on to next branch

                if foundPartner
                    bot.status = 0;
                    bot.leader = 0;
                    done = 0;
                    return;
                end

                bot.resetSwap();
                done = 0;
                return;

            end
        end
    end
end


%*********************************************************%
% endSwap:                                                %
% Send the bot back to the branch point                   %
%*********************************************************%
function done = endSwap(bot)
    foundPartner = 0;

    if bot.knowledge(1).botNum ~= 0
        for i=1:length(bot.knowledge)
            if bot.knowledge(i).botNum == bot.swap
                foundPartner = 1;
                swapPartner = bot.knowledge(i);
                break;
            end
        end
    end

    if bot.leader == 1 % bot is the leader
        if bot.oldTwig == bot.path(end) % heading to oldTwig
            % follower has reached twig
            [xDest yDest] = bot.map.node2xy(bot.path(end));
            if bot.xPos == xDest && bot.yPos == yDest
                % bot is at oldTwig
                if foundPartner
                    [xBranch yBranch] = ...
                        bot.map.node2xy(bot.visited(end));
                    if swapPartner.xPos == xBranch...
```

```
                                    && swapPartner.yPos == yBranch...
                                    && swapPartner.path(end) == ...
                                    bot.visited(end)
                                % follower is at branch, end swap
                                % swap is complete!
                                % Huzzah, Huzzah for Charter Club!
                                bot.resetSwap();
                                done = 0;
                                return;
                            end
                            % follower is not at branch, wait
                            done = -1;
                            return
                        end

                        % no communication, return to normal mode
                        bot.resetSwap();
                        done = 0;
                        return;
                    else
                        % heading to oldTwig
                        % check if oldTwig is blocked
                        if bot.knowledge(1).botNum ~= 0
                            for i=1:length(bot.knowledge)
                                if bot.knowledge(i).status == -1 && ...
                                        bot.knowledge(i).boxNum == ...
                                        bot.oldTwig
                                    bot.status = 0;
                                    bot.leader = 0;
                                    done = 0;
                                    return;
                                end
                            end
                        end

                        % keep going to oldTwig
                        done = 0;
                        return;
                    end
                end

                % at twig, check if other bot is in position
                if foundPartner
                    % check if other bot needs new branch
                    if swapPartner.status == 0
                        bot.status = 0;
                        bot.leader = 0;
                        % call start swap right away so partner
                        % doesn't misinterpret status = 0
                        done = bot.startSwap();
                        return;
                    end
                    % check if other bot is at his twig
                    [xTwig yTwig] = bot.map.node2xy(swapPartner.path(end));
                    if swapPartner.xPos == xTwig && ...
                            swapPartner.yPos == yTwig
                        bot.initialize(bot.oldTwig);
                        done = 0;
                        return;
                    else
                        done = -1;
                        return;
                    end
                end


            else % bot is the follower
                if bot.visited(end) == bot.path(end) % heading to branch
                    [xDest yDest] = bot.map.node2xy(bot.path(end));
                    if bot.xPos == xDest && bot.yPos == yDest
                        % swap is complete
                        bot.resetSwap();
                        done = 0;
                        return;
```

```
                    end

                    % heading to branch
                    done = 0;
                    return;
                end

                if foundPartner
                    [xTwig yTwig] = bot.map.node2xy(bot.oldTwig);
                    if swapPartner.xPos == xTwig && ...
                            swapPartner.yPos == yTwig
                        % leader is at oldTwig
                        bot.initialize(bot.visited(end));
                        done = 0;
                        return;
                    else
                        % check if other bot needs new branch
                        if swapPartner.status == 0
                            bot.status = 0;
                            bot.leader = 0;
                            % call start swap right away so partner
                            % doesn’t misinterpret status = 0
                            done = bot.startSwap();
                            return;
                        end
                    end
                    done = -1;
                    return;
                end
            end


            % at this point, no communication with swap partner
            bot.resetSwap();
            done = 0;
        end

    %*********************************************************%
    % resetSwap:                                              %
    % Reset bot variables involved in swaps.                  %
    %*********************************************************%
    function resetSwap(bot)
            bot.swap = 0;
            bot.visited = 0;
            bot.status = 0;
            bot.leader = 0;
            bot.oldTwig = 0;

            bot.initialize(bot.goalNum);
            bot.priority = bot.botNum;
        end

        %*********************************************************%
        % initialize:                                             %
        % Initializes bot’s path.                                 %
        %*********************************************************%
        function initialize(bot, destNode)
            startNode1 = bot.path(bot.last);
            startNode2 = bot.path(bot.last+1);

            path1 = bot.map.makePath(startNode1, destNode);
            path2 = bot.map.makePath(startNode2, destNode);
            if path1(2) == path2(1)
                bot.path = path1;
            else
                bot.path = path2;
            end
            bot.last = 1;
        end
    end % private methods
end % classdef
```

B.3 Map.m

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decentralized and Complete Multi-Robot Motion Planning        %
%                                           in Confined Spaces  %
% Dexter Scobee and Adam Wiktor                                 %
% Map datatype:                                                 %
%                                                               %
% Datatype to store the map that robots travel on. Consists of  %
% nodes and the edges that connect them.                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Map < handle

    properties
        nX = 0;        % number of nodes along X axis
        nY = 0;        % number of nodes along Y axis
        n = 0;         % total number of nodes
        graph = 0;     % matrix storing connections between nodes
        tree = 0;      % tree
        root = 0;      % root of the tree
        nodeDepth = 0; % matrix containing the depth of each node
        rho = 0;
        comm = 0;
    end

    properties (Constant = true)
        dx = 1; % X distance between nodes
        dy = 1; % Y distance between nodes
    end

    methods

        %*********************************************************%
        % Constructor. Takes the number of x nodes and the number %
        % of y nodes as arguments, and returns a map object       %
        %*********************************************************%
        function map = Map(x, y)
            map.nY = y;
            map.nX = x;
            map.n = map.nX*map.nY;
            map.graph = zeros(map.n, map.n);

            map.root = map.xy2node(ceil(map.nX/2), ceil(map.nY/2));
            map.tree = map.bfs(map.root);
            map.dfs(map.root, 1);
            map.makeComm();
        end

        %*********************************************************%
        % addEdge:                                                %
        % Adds an edge to the map. Takes the x and y coordinates  %
        % of each node to be connected.                           %
        %*********************************************************%
        function addEdge(map, x1, y1, x2, y2)
            v = map.xy2node(x1,y1);
            w = map.xy2node(x2,y2);
            map.graph(v,w) = 1;
            map.graph(w,v) = 1;
        end

        %*********************************************************%
        % makePath:                                               %
        % Calculates the shortest path between the start node and %
        % destination node using breadth-first search. Takes the  %
        % start node and destination node numbers as arguments    %
        % and returns an array of nodes representing the path.    %
        %*********************************************************%
        function path = makePath(map, startNode, destNode)
            % Build a tree using BFS with startNode as the root

            s = startNode;
            i = 1;
            while (map.tree(s(i)) ~= s(i))
```

```
            s(i+1) = map.tree(s(i));
            i = i+1;
        end

        d = destNode;
        i = 1;
        while (map.tree(d(i)) ~= d(i))
            d(i+1) = map.tree(d(i));
            i = i+1;
        end

        for i=1:length(d)
            pathD(i) = d(length(d)-i+1);
        end

        for i=1:length(s)
            for j=1:length(pathD)
                if s(i) == pathD(j)
                    break;
                end
            end
            if s(i) == pathD(j)
                break;
            end
        end

        path = s(1:i);
        path(i+1:i+(length(pathD)-j)) = pathD(j+1:end);


        path(end+1) = path(end);
    end

    %*********************************************************%
    % draw:                                                   %
    % Draws the map                                           %
    %*********************************************************%
    function draw(map)
        hold on
        for i=1:map.n
            % draw the nodes
            [x1 y1] = map.node2xy(i);
            plot(map.dx*x1,map.dy*y1,'.k','MarkerSize',10);

            % draw the edges
            for j=1:map.n
                if (map.graph(i,j) == 1)
                    [x2 y2] = map.node2xy(j);
                    plot(map.dx*[x1 x2], map.dy*[y1 y2],'k');
                end
            end
        end

        axis([0 map.dx*(map.nX+1) 0 map.dy*(map.nY+1)]);
        axis square
    end


    %*********************************************************%
    % drawTree:                                               %
    % Draws only edges that are part of the tree              %
    %*********************************************************%
    function drawTree(map)
        hold on
        for i=1:map.n
            % draw the nodes
            [x1 y1] = map.node2xy(i);
            plot(map.dx*x1,map.dy*y1,'.k','MarkerSize',10);

            % draw the edges
            for j=1:map.n
                if j == map.root
                    continue;
                end
```

```
                [x1 y1] = map.node2xy(j);
                [x2 y2] = map.node2xy(map.tree(j));
                plot(map.dx*[x1 x2], map.dy*[y1 y2],'k');
            end
        end

        axis([0 map.dx*(map.nX+1) 0 map.dy*(map.nY+1)]);
      %  axis square
    end


    %*********************************************************%
    % makeTree:                                               %
    % Build the tree using BFS                                %
    %*********************************************************%
    function makeTree(map)
        map.tree = map.bfs(map.root);
    end

    %*********************************************************%
    % findBranches:                                           %
    % Find branch nodes in the tree                           %
    %*********************************************************%
    function count = findBranches(map)
        count = ones(map.n, 1);
        for i=1:map.n
            if i ~= map.root
                count(map.tree(i)) = count(map.tree(i)) + 1;
            end
        end

        count(map.root) = count(map.root) - 1;
    end

    %*********************************************************%
    % xy2node:                                                %
    % Converts the x-y coordinates of a node to a node        %
    % number.                                                 %
    %*********************************************************%
    function node = xy2node(map,x,y)
        node = (y-1)*map.nX + x;
    end

    %*********************************************************%
    % node2xy:                                                %
    % Converts a node number to x-y coordinates.              %
    %*********************************************************%
    function [x y] = node2xy(map,node)
        y = floor((node-1)/map.nX)+1;
        x = node - (y-1)*map.nX;
    end

    %*********************************************************%
    % nodeDist:                                               %
    % Calculates the cartesian distance between two nodes     %
    % given the node numbers.                                 %
    %*********************************************************%
    function distance = nodeDist(map, n1, n2)
        [x1 y1] = map.node2xy(n1);
        [x2 y2] = map.node2xy(n2);

        xDist = x1 - x2;
        yDist = y1 - y2;

        distance = sqrt(xDist*xDist + yDist*yDist);
    end



    %*********************************************************%
    % testMap:                                                %
    % Adds edges to a sample map. Requires a map at least 5x4 %
    % nodes.                                                  %
    %*********************************************************%
```

```
        function testMap(map)
            map.addEdge(1,1,2,1);
            map.addEdge(2,1,2,2);
            map.addEdge(2,2,1,2);
            map.addEdge(1,2,1,3);
            map.addEdge(1,3,2,3);
            map.addEdge(2,3,2,4);
            map.addEdge(2,4,1,4);
            map.addEdge(3,2,2,3);
            map.addEdge(3,1,3,2);
            map.addEdge(3,2,4,2);
            map.addEdge(4,2,4,1);
            map.addEdge(4,1,5,1);
            map.addEdge(3,2,3,3);
            map.addEdge(3,2,4,3);
            map.addEdge(3,3,3,4);
            map.addEdge(3,4,4,4);
            map.addEdge(4,4,4,3);
            map.addEdge(4,3,5,3);
            map.addEdge(5,3,5,2);
            map.addEdge(5,3,5,4);
        end

        %*********************************************************%
        % makeComm:                                               %
        % Performs breadth-first search to complete the           %
        % communication matrix                                    %
        %*********************************************************%
        function makeComm(map)
            r = ceil(map.rho);
            map.comm = zeros(map.n, map.n);
            for i=1:map.n
               map.bfsComm(i, r);
            end
        end

        %*********************************************************%
        % bfsComm:                                                %
        % Performs breadth-first search to build a tree from the  %
        % start node to every other node on the map.              %
        %*********************************************************%
        function bfsComm(map, startNode, r)

            q = zeros(1,map.n);
            q(1) = startNode;
            pos = 1;
            len = 1;
            dist = zeros(1,map.n);

            while (len > 0 && dist(q(pos)) <= r)
                v = q(pos);
                map.comm(startNode, v) = 1;

                pos = pos+1;
                len = len - 1;
                for i=1:map.n
                    if i ~= v && ((map.tree(i) == v) ...
                            || map.tree(v) == i) && dist(i) == 0
                        dist(i) = dist(v) + 1;
                        q(pos + len) = i;
                        len = len + 1;
                    end
                end
            end
        end

        %*********************************************************%
        % bfs:                                                    %
        % Performs breadth-first search to build a tree from the  %
        % start node to every other node on the map.              %
        %*********************************************************%
        function tree = bfs(map, startNode)
            tree = zeros(1,map.n);
            tree(startNode) = startNode;
```

```matlab

            q = zeros(1,map.n);
            q(1) = startNode;
            pos = 1;
            len = 1;
            while (len > 0)
                v = q(pos);
                pos = pos+1;
                len = len-1;
                for i=1:map.n
                    if (map.graph(v,i) == 1) && (tree(i) == 0)
                        q(pos + len) = i;
                        len = len + 1;
                        tree(i) = v;
                    end
                end
            end
        end

        %*********************************************************%
        % dfs:                                                    %
        % Performs depth-first search to determine the            %
        % priority ranking ("depth") of each node in the tree     %
        %*********************************************************%
        function depth = dfs(map, parent, depth)

            for i=1:map.n
                if (map.tree(i) == parent) && (i ~= parent)
                    depth = map.dfs(i, depth);
                end
            end

            map.nodeDepth(parent) = depth;
            depth = depth+1;
        end

    end % public methods

    methods (Static)
        %*********************************************************%
        % xyDist:                                                 %
        % Calculates the cartesian distance between two points on %
        % the map given cartesian coordinates.                    %
        %*********************************************************%
        function distance = xyDist(x1, y1, x2, y2)
            xDist = x1 - x2;
            yDist = y1 - y2;

            distance = sqrt(xDist*xDist + yDist*yDist);
        end
    end % static methods

end % classdef
```

B.4 checkNeighbors.m

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decentralized and Complete Multi-Robot Motion Planning        %
%                                           in Confined Spaces  %
% Dexter Scobee and Adam Wiktor                                 %
% checkNeighbors function:                                      %
%                                                               %
% Checks if a given robot has neighbors close enough to        %
% communicate with, and collects data from any close neighbors. %
% Takes the radius of communication, the botNum of the given   %
% robot, and the array of all robots as input arguments.       %
% Returns the array of data from neighboring robots, or a      %
% structure with a botNum of 0 if there are no neighbors.      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function data = checkNeighbors(botIndex, bot)
```

```
neighbor = 1;

for i=1:length(bot)
    if i == botIndex
        continue; % do not check if a robot is its own neighbor
    end

    % check if the robots are close enough to communicate
    if bot(botIndex).map.comm(bot(botIndex).boxNum,bot(i).boxNum) == 1
        signal = bot(i).signal(); % get data from robot i
        tempData(neighbor:neighbor+length(signal)-1) = signal;
        neighbor = neighbor + length(signal);
    end

end % for

if neighbor == 1 % no neighbors found, return an empty struct
    data = struct(’botNum’, 0, ’xPos’, 0,...
        ’yPos’, 0,’xGoal’, 0, ’yGoal’, 0,...
        ’priority’, 0, ’path’, 0, ’boxNum’, 0,...
        ’swap’, 0, ’status’, 0, ’solved’, 0, ’ToR’, 0);

else % neighbors were found
    % check for duplicate information in tempData

    botNumList = zeros(1,length(tempData));
    torList = zeros(1,length(tempData));
    for i=1:length(tempData)
        botNumList(i) = tempData(i).botNum;
        torList(i) = tempData(i).ToR;
    end

    [Y I] = sort(-torList);
    botNumList = botNumList(I);
    tempData = tempData(I);

    [Y I] = sort(botNumList);
    tempData = tempData(I);
    checkBot = zeros(1,bot(botIndex).map.n);

    if bot(botIndex).knowledge(1).botNum ~= 0
        for i=1:length(bot(botIndex).knowledge)
            checkBot(bot(botIndex).knowledge(i).botNum) =...
                bot(botIndex).knowledge(i).ToR;
        end
    end

    index = 2;
    num = 1;
    while tempData(num).botNum == 0 ||...
            tempData(num).botNum == bot(botIndex).botNum ||...
            (tempData(num).ToR <= checkBot(tempData(num).botNum) ...
                        && (checkBot(tempData(num).botNum) ~= ...
                        bot(botIndex).time - 1))
        num = num + 1;
    end
    data(1) = tempData(num);

    for i = num+1:length(tempData)
       if tempData(i).botNum ~= tempData(i-1).botNum && ...
               tempData(i).botNum ~= bot(botIndex).botNum && ...
               tempData(i).botNum ~= 0 &&...
               (tempData(i).ToR > checkBot(tempData(i).botNum) ...
                        || (checkBot(tempData(i).botNum) == ...
                        bot(botIndex).time - 1))
          data(index) = tempData(i);
          index = index + 1;
       end
    end

end % if
end % function checkNeighbors
```

This paper represents our own work in accordance with University regulations.