MASTERS THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF **MASTER OF MECHANICAL ENGINEERING**

TITLE: Auto-Generation and Real-Time Optimization of Control Software for Multi-Robot Systems

PRESENTED BY: Jill Goryca

ACCEPTED BY:

_____

Advisor, *Dr. Richard Hill* **Date**

_____

Department Chairperson, *Dr. Nassif Rayess* **Date**

APPROVAL:

_____

Dean, *Dr. Gary Kuleck* **Date**
College of Engineering and Science

Contents

List of Figures

List of Tables

1 Introduction

An event is something that happens. Discrete means separate, or distinct. Therefore, a discrete-event system consists of distinct happenings that change one state of the system to another. A discrete-event system can be as simple as a light bulb with a switch that can ‘turn on’ and ‘turn off.’ However, discrete-event systems can be large and complex: consider a software program, or a group of autonomous robots. The theory of discrete-event systems formalizes the process of representing, analyzing, and applying control logic to these systems.

This thesis applies discrete-event systems theory to robotic control systems. First, the theory is used to generate reliable control logic, which is represented by a finite state machine. Next, the finite state machine, along with user input, is used to generate real-time control software. The control software implements the logic represented by the finite state machine on simulated robots. The control software needs to be generated automatically, because a finite state machine can have a large number of states, which makes it difficult to code the software by hand.

Finite state machines contain all behaviors that satisfy a given set of requirements. Since multiple behaviors may be possible that satisfy the given requirements, the “best” set of control actions needs to be chosen from the set of allowed behaviors. An optimization algorithm is used to determine the best set of control actions, providing an optimum path through the finite state machine. The control software follows the optimized path to complete a mission. To implement the optimization algorithm in the control software, a cost needs to be assigned to each behavior.

The remainder of this chapter states the objectives of this project; background information on discrete-event systems, finite state machines and the Player/Stage simulation software used in this project; and the state of the art in control systems and robotics. Chapter 2 describes the robotics example that was used to develop and validate the real-time control software. It also explains the approach and shows the overall structure of the solution. Chapter 3 details the MATLAB tool and control software that was developed, and Chapter 4 presents a description of the optimization and the results of the simulation that were completed. Finally, Chapter 5 concludes the thesis, summarizing contributions and opportunities for future work. Appendices A-G provide technical details and include the MATLAB code that was developed as part of this project.

1.1 Objectives

There are two main objectives to this project. 1) Use a finite state machine to automatically generate software in MATLAB to control the actions of two robots in a simulation environment. 2) Optimize the actions of the two robots to complete a mission within the constraints of the finite state machine, demonstrating its effectiveness in a simulation environment.

1.2 Background Information

This section explains background information regarding the concepts of discrete-event systems and finite state machines. It also gives a description of the Player/Stage software.

1.2.1 Discrete-Event Systems

High-level robot control can be modeled as a discrete-event system (DES). A DES is a dynamic system which has distinct states such as a robot 'running' or 'stopped.' A DES transitions to a new state after an event, such as 'task finished' occurs. Because of their discrete states, DES are considered different than discrete-time systems, which are sampled versions of continuous-time systems. However, continuous-time models can be abstracted to a discrete-event model using a threshold value, which represents when an event (state change) has taken place [1]. DES can model a system at a higher level than discrete time systems. This allows them to be used for applications such as high-level robot control.

1.2.2 Finite State Machines

A finite state machine (FSM) is a graphical representation of states and events that can be traversed by a DES to reach a desired goal. Since several control actions may be valid at the same time, an FSM shows all of the valid state and event configurations. A sample FSM is shown in Figure 1-1.

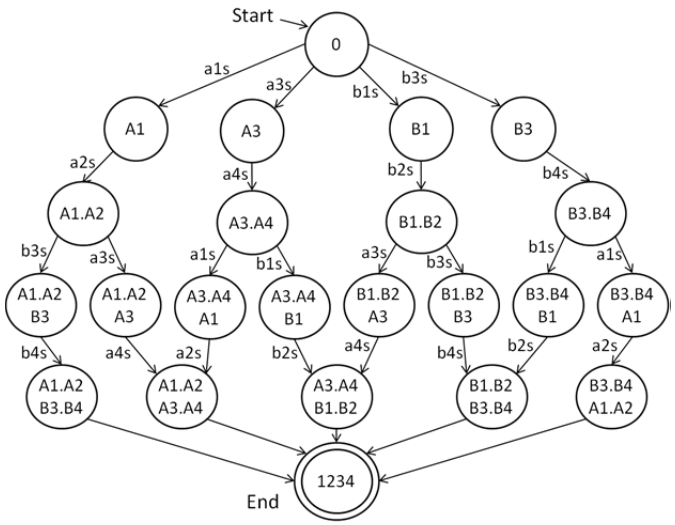

Figure 1-1 A finite state machine for two robots (A, B) completing four tasks (1-4).

In Figure 1-1, each circle represents a state, and each arrow between two states represents an event. The initial state is labeled “0”, and is indicated with an arrow. At the bottom of Figure 1-1, a double circle indicates a marked state, which is the end, or goal state. Since multiple events proceed from the initial state, there are several paths to choose from in order to achieve the goal state.

Note that since this FSM is a small example, all of the states and events for the scenario are not shown. Only the task-starting events are shown for simplicity. In DES, events are assumed to happen instantaneously, and a task cannot realistically be started and completed at the same time. Therefore, additional states and events occur between the time a task is started and when is is finished. In a larger FSM, additional events would be added to represent finishing tasks.

Events can be controllable or uncontrollable. A controllable event can be started by the control software. An example of a controllable event is a robot starting a task. An uncontrollable event can only be detected by the control program. It cannot be started (or prevented) by it. An example of an uncontrollable event is detecting that a robot has finished a task, assuming that the robot cannot be stopped after it starts a task.

Events can also be observable or unobservable. An observable event is one that can be detected, while an unobservable event cannot be detected. For example, an unobservable event could be that the wheels of the robot had slipped and its position on the map was no longer accurate. This happens frequently in real-life applications. However, since simulation was used in this project, all events are assumed to be observable.

1.2.3 Player/Stage Simulation Software

Player/Stage is open-source software that was used for testing the control code. This software consists of two pieces: “Player” and “Stage”. Player is a defined set of interfaces and drivers that can run in combination with Stage or an actual robot. Stage receives commands from Player and simulates the response of the robotic device. It displays an animation of the simulated robots. One limitation of the Stage simulator is that it does not simulate the dynamics of the robot. However, this software was considered sufficient for testing a high-level control algorithm. The details of the Player/Stage setup for this project can be found in *Appendix A: Player/Stage Simulation Setup and Instructions*.

1.3 State of the Art in Control Systems and Robotics

This section provides an overview of relevant work regarding control theory and robotics algorithms. Supervisory control theory is explained in the context of DES. Graph-search algorithms from robotics are explained, because the optimization algorithm used in this project is derived from a graph-search algorithm. The low-level robotics algorithms used in this project are also briefly described.

1.3.1 Supervisory Control Theory

High-level control logic can be synthesized using supervisory control theory for DES [2]. Supervisory control theory aims to keep a system safe by preventing unwanted behavior. It also verifies that the system is not blocked, so it can reach a desired goal state. Supervisory control theory is necessary because the complexity of modern systems makes it increasingly challenging to design custom controllers

for every new system. This theory provides rules to combine multiple controller models and methods to ensure that the control logic will meet a given set of requirements. Supervisory control theory continues to be developed to increase its effectiveness for a range of applications including robotics.

The main issue with supervisory control stems from building one large controller consisting of all possibilities. For large problems, this causes the size of the controller to grow exponentially, quickly using up available storage space and increasing computation time. A number of approaches exist that are designed to alleviate this problem by dividing the large controller into smaller parts. These include decentralized [3], hierarchical [4], and multi-modal [5] control.

1.3.2 Graph-Search Algorithms in Robotics

Graph-search algorithms exist for finding the minimum cost path through a graph, which is a structure containing nodes and edges. An FSM is a graph whose nodes and edges are equivalent to states and events, respectively. Each edge (or event) has a cost associated with it. The graph-search algorithm minimizes the total cost of all traveled edges between an initial state and a goal state to find the best path through the graph.

There are several different methods of solving the shortest path problem for graphs. These methods include genetic algorithms using a random search method [6] and element-by-element iteration algorithms such as Dijkstra's algorithm [7]. Several modifications to Dijkstra's algorithm have been made to improve its performance in dynamic situations including A* [8], which uses a heuristic to

quickly find a solution, and D*Lite [9], which reuses cost information that has not changed.

1.3.3 Low-Level Algorithms for Robotics

There are low-level control algorithms for robotic systems that are fairly well developed and robust. The algorithms used in this project were developed by the Advanced Mobility Laboratory at UDM. They have been validated through prior research activities and used in the International Ground Vehicle Competition. The algorithms used in this project include D*Lite, VFH, and a mapping algorithm. D*Lite receives goal locations from the main control file, finds a path that avoids all known obstacles, and calculates intermediate waypoints (or breadcrumbs) to the goal [9]. VFH provides velocity and turn rate commands to send the robot to the waypoints while avoiding contact with obstacles within the range of the robot's laser [10]. The mapping algorithm localizes the robot's position on a map and provides the map as an input variable for D*Lite and VFH. These algorithms are used by the actual vehicles in the Advanced Mobility Laboratory. Using the low-level algorithms in this project will facilitate implementation on actual hardware. However, these low-level algorithms do not address the high-level control problem.

2 Problem Definition and Proposed Solution

This chapter defines the problem and describes the motivating example used in this project. The proposed solution and approach that was used to solve the problem is given as well as the overall structure of the proposed solution.

2.1 Problem Definition

In general, the goal of this project is to establish a process to automatically synthesize control software for a range of applications. A robotics application in multi-robot control was used as the motivating example for this project.

The first challenge is to integrate the user's high-level desired mission for the robots with low-level control algorithms. Thus, one problem is to automatically generate real-time control software for the robots based on an FSM and user input. The control software must interact with low-level control algorithms such as D*Lite and VFH. It must be able to accept user input easily so that the states and events of the FSM can be defined in the simulated environment.

A second problem is to choose the optimal sequence of control actions within the constraints of the FSM. In this project, optimum is defined as the shortest time, which is assumed to be equivalent to the shortest distance. The optimized solution must be able to be updated easily as the robots move, since the distance between each robot and each task location changes. It must also take advantage of the fact that two robots can operate simultaneously.

2.2 Motivating Example

This project employed a simple test case that provided a concrete example to design to and test work against. Although the test case is not extremely realistic, it

captures the most important features, and is small enough to verify results by hand. In the test case, two robots must complete a mission, which consists of performing four tasks at various locations in a specified manner. A task is defined as a location (in x-y coordinates) that must be reached by the robot. The robots travel in rectangular regions (defined by x-y coordinates) on a map in the Player/Stage simulation environment. The starting positions of the robots and the locations of the tasks are shown in Figure 2-1.

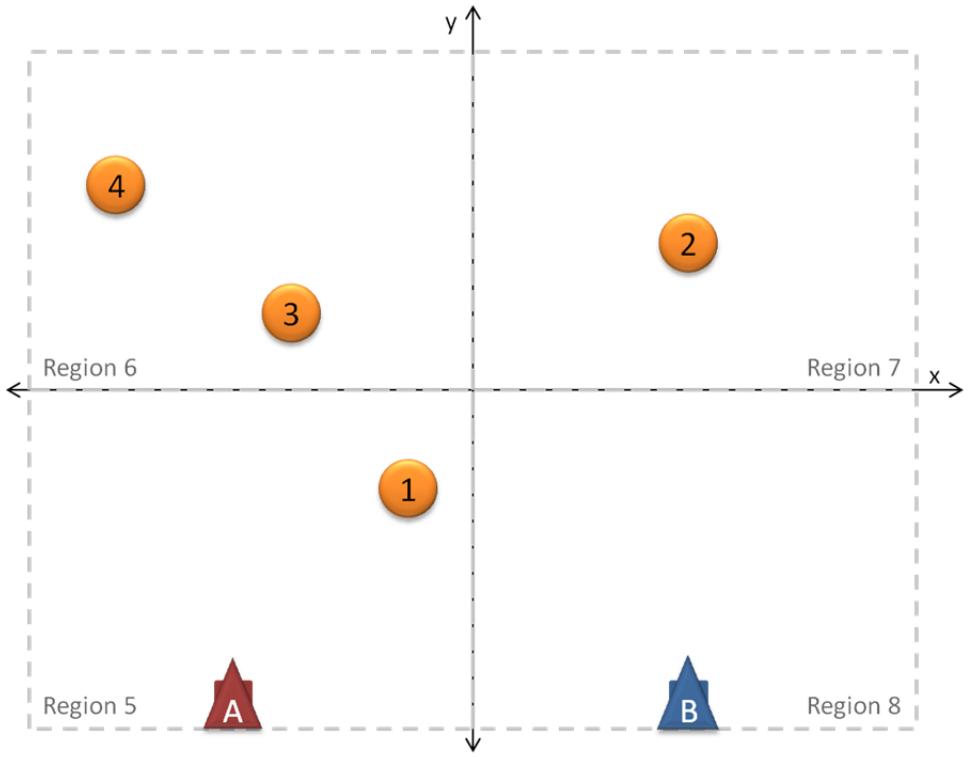

Figure 2-1 Map showing two robots, four tasks, and four regions.

In Figure 2-1, the two robots are labeled A and B, and the four tasks are numbered 1 through 4. The four regions, numbered 5 through 8, are defined as the four quadrants of a Cartesian-coordinate system. The robots start in different regions at the lower edge of the map, and the tasks are distributed in regions toward the upper edge of the map.

In an FSM, regions are represented by states such as “5A,” which corresponds to robot A in region 5. A robot moving from one region to another is defined as an uncontrollable event such as “a5e,” which corresponds to robot A entering region 5. Figure 2-2 shows the FSM for robot A’s region geometry. A similar FSM is used for robot B.

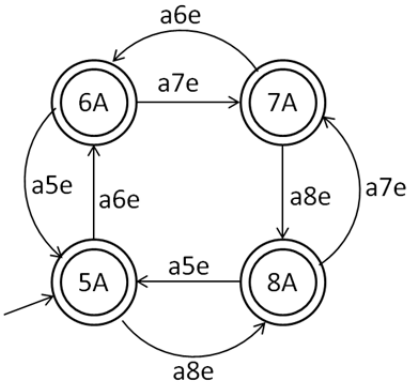

Figure 2-2 Robot A’s region geometry represented by an FSM.

Figure 2-2 shows that robot A may move between adjoining regions in either direction. It does not include moving diagonally from region 5 to region 7, so it is assumed that robot A will not encounter this situation.

A task in an FSM is represented by a state such as “4s.” Starting a task is a controllable event, and finishing a task is an uncontrollable event that occurs when the robot has reached the task location. For example, “a1s” is a task-start event which means that robot A has started task 1. Similarly, “a1f” is a task-finish event which indicates robot A has completed task 1. The task FSM shown in Figure 2-3 allows only one task to be started at a time, and the task must be finished before the next task can be started. This is because the tasks are defined as locations, and a robot can only travel to one location at a time. The locations of

the tasks are confined to specific regions by the region-entry events (“a7e”) that correspond to each task.

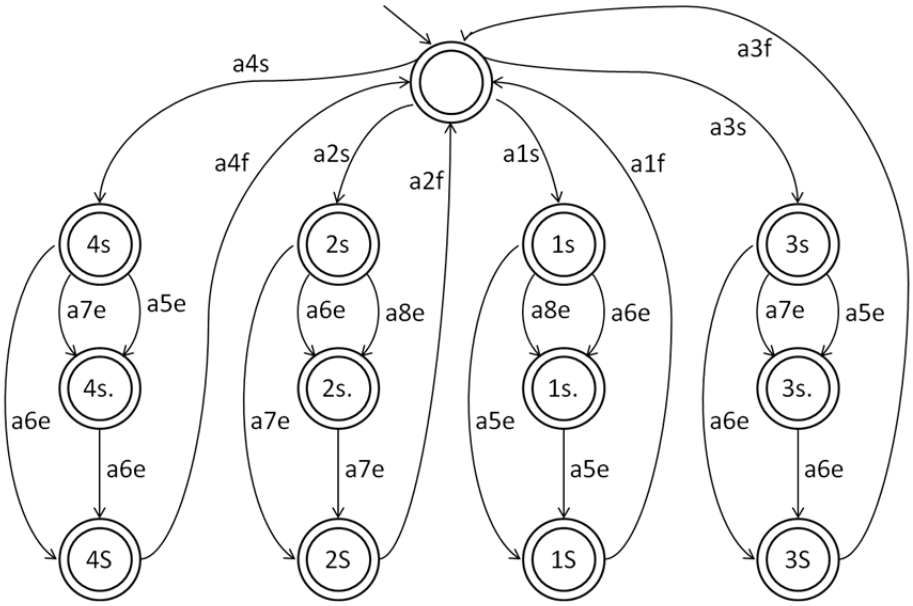

Figure 2-3 Robot A’s task order represented by an FSM.

The region and task FSMs are combined to obtain a “plant” FSM that represents the uncontrolled behavior of a robot. The plant FSM was automatically synthesized from the component FSMs using a program developed by the University of Michigan called UMDES/DESUMA [11]. Additional details for synthesizing FSMs can be found in reference [12]. The plant FSM for robot A is shown in Figure 2-4.

The plant FSM is similar for both robot A and B. The only difference is the region-entry events, since robot B starts in a different region than robot A. Note that the positions of the tasks and the starting locations of the robots can vary while using the same FSM, as long as the tasks and robots are located in the same region. If the tasks or robots are located in a different region, the FSM must be modified to reflect different region events.

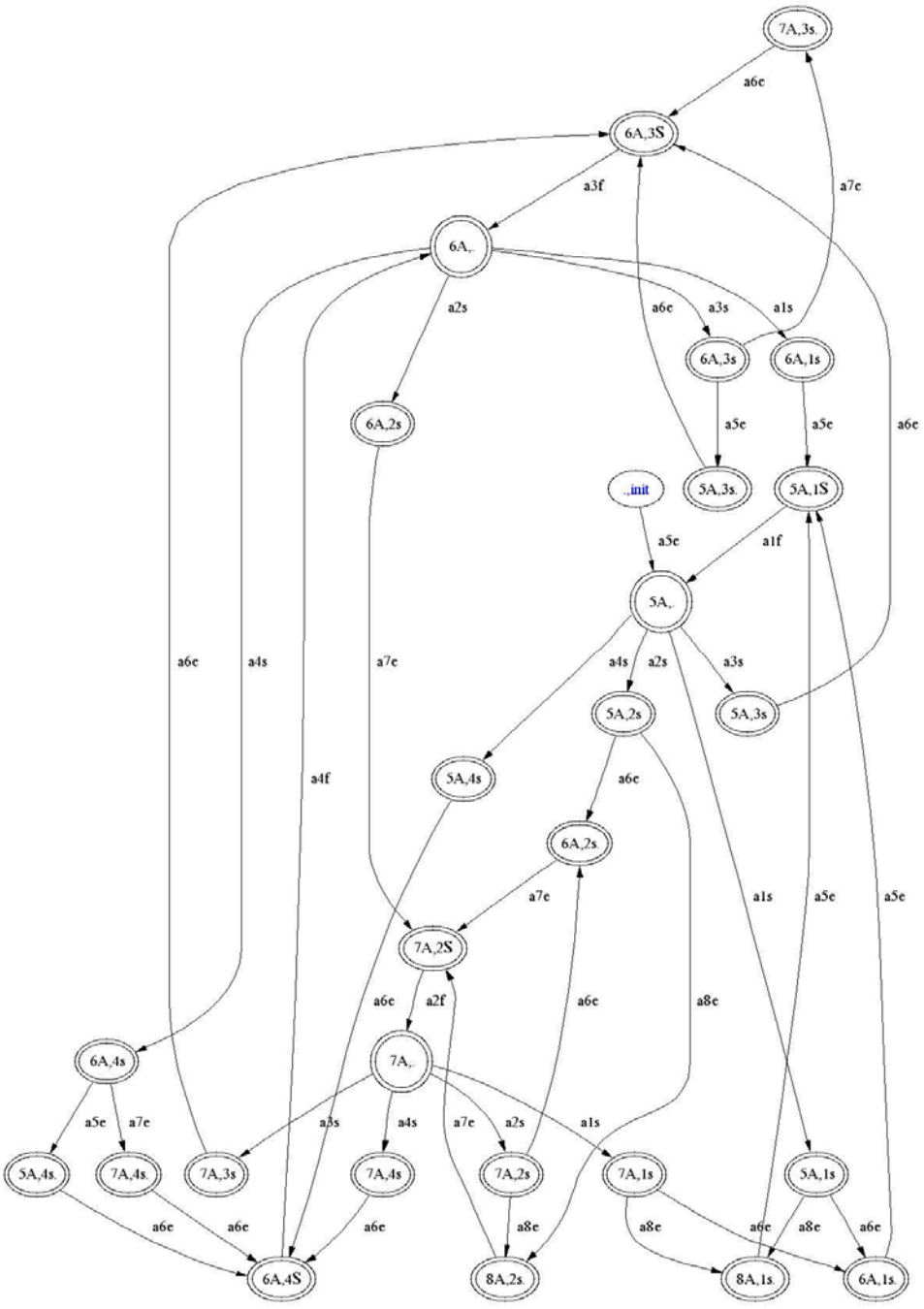

Figure 2-4 Robot A "plant" represented by an FSM.

In addition to defining the robots' behavior, the rules for completing tasks must be modeled. The avoidance rule states that the robots may not be in the same region at the same time to avoid the possibility of robots running into each other. Figure 2-5 shows the control logic for this avoidance rule in an FSM.

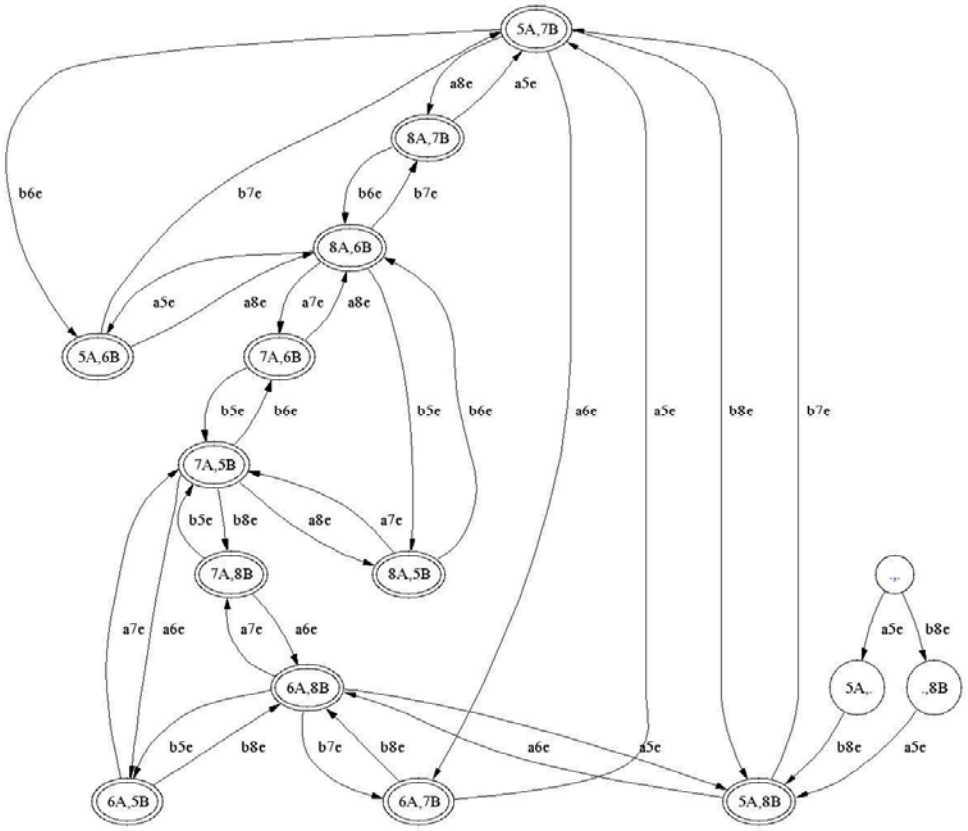

Figure 2-5 FSM showing avoidance control logic for robots A and B.

The task-completion rules are the order that the tasks must be completed. Task 1 must be completed before task 2 by one robot. Task 3 must be completed before task 4 by the other robot. It does not matter whether robot A or robot B completes any task sequence, as long as the same robot completes both tasks. For example, task 1 could represent picking up a package, and task 2 could be dropping it off. The same robot that picks up the package should deposit it at the correct spot. The task-completion FSM for robot A is shown in Figure 2-6.

The FSM that was used in this project was automatically synthesized using the avoidance and task-completion FSMs. It represents the controller for the system, and ensures that all requirements are met while always being able to reach the goal state. It is shown in Figure 2-7.

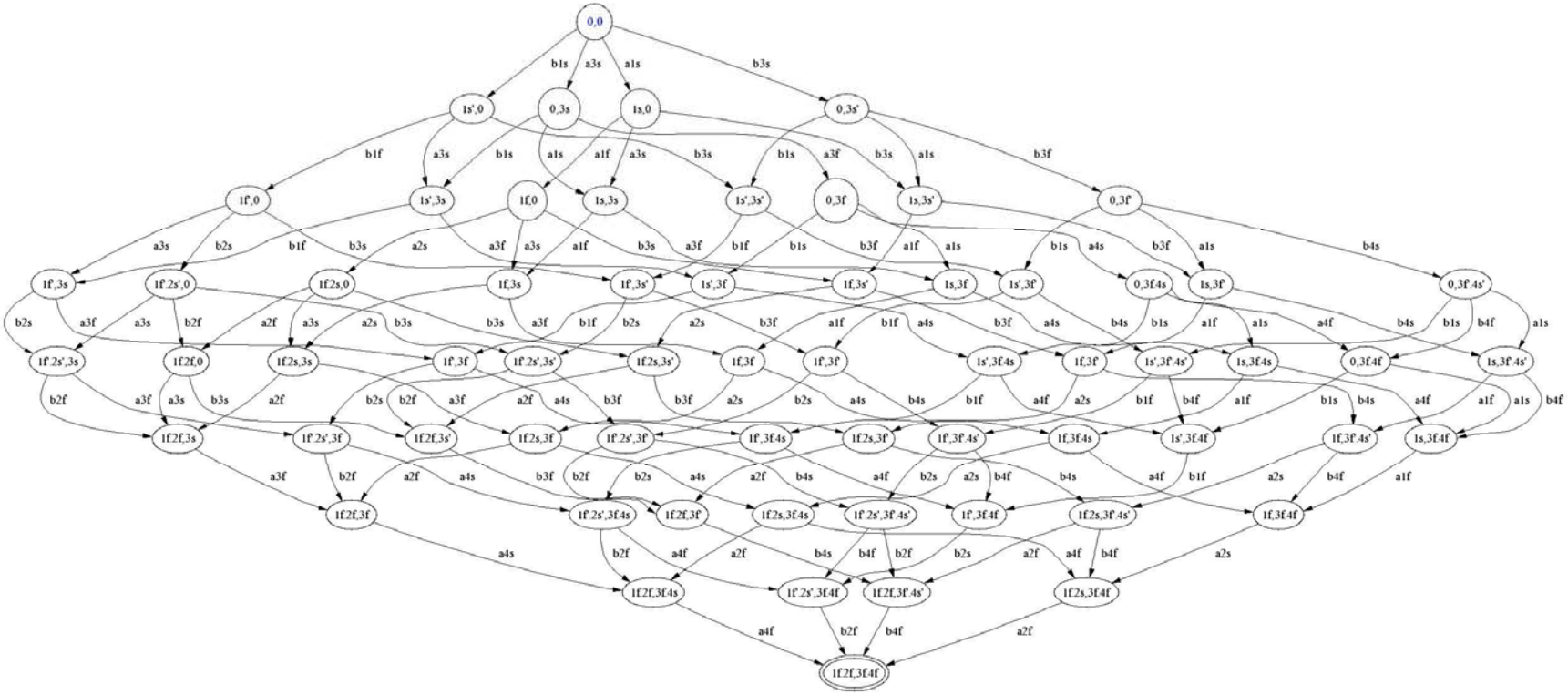

Figure 2-6 FSM showing task-completion control logic for robots A and B.

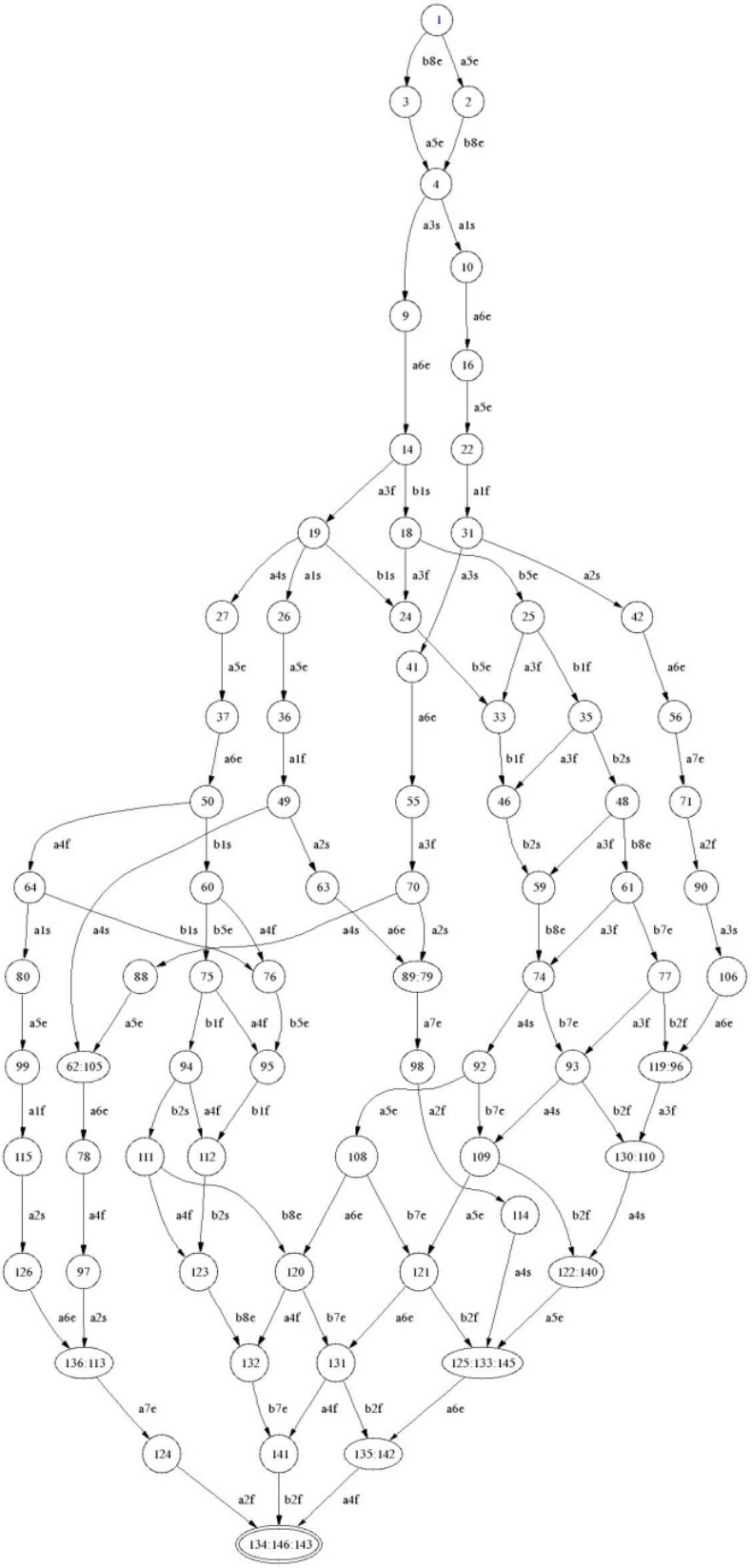

Figure 2-7 FSM used in this project showing control logic for both robots.

The FSM shown in Figure 2-7 has a choice at the state labeled “4”. The controllable events “a1s” and “a3s” at state 4 each have a cost associated with them. In this project, the cost for controllable events is based on the distance between the robot and the task location. The uncontrollable events have a zero cost assigned to them. The best choice from among the controllable events is determined by an optimization algorithm. The optimization is discussed in Chapter 4.

2.3 Proposed Solution

The proposed solution is an interface between the FSM control logic and the low-level control algorithms. The interface contains a MATLAB tool and high-level control algorithms. The high-level control algorithms are the software which control, detect, and optimize the actions of the robots. The MATLAB tool is a GUI that facilitates data entry for the user. The GUI displays the fields that the user must provide for the high-level control algorithms to translate the user’s desired mission into task locations for the low-level control algorithms. It automatically generates a file containing the user data in the format required by the high-level control algorithms. See Figure 2-8 for a diagram showing the relationship between the components of the proposed solution.

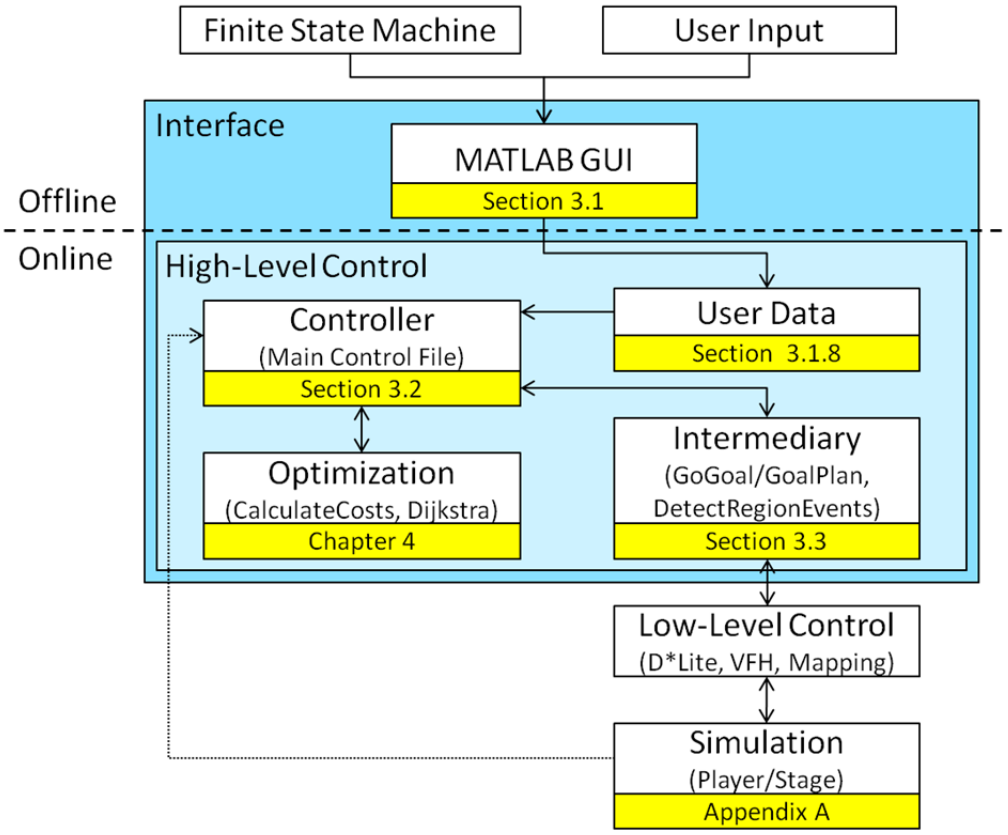

Figure 2-8 Relationship between components of the proposed solution.

As Figure 2-8 shows, the user provides the FSM file and input to the GUI. The GUI generates the User Data in a format compatible with the Main Control File. This happens offline, before any other control software is running. The Controller is the Main Control File that runs the other high-level software components. The Controller optimizes the controllable events in the FSM using Optimization functions including Calculate Costs and a modified Dijkstra's algorithm. The Controller calls the GoGoal or GoalPlan intermediate function, which interacts with the low-level algorithms D*Lite, VFH, and Mapping to send the robot to a goal location while simultaneously checking for new events to occur using the Detect Region Events function. When the next event is detected,

the Controller updates the current state of the FSM and determines the next set of control actions.

The overall structure of the proposed solution consists of many files that are related to one another. The details of each file are explained in the section indicated by each yellow box in Figure 2-8. The name of each file is shown in Table 2-1.

Table 2-1 File names of solution components.

Item	File Name
Finite State Machine	`control2.fsm`
MATLAB Tool	`FSM_Interpreter.m`
Controller/Main Control File (Robot A) (Robot B)	`Cer_VFSM_Controller_Start.m` `Cer1_VFSM_Controller_Start.m`
User Data	`UserData_control2_thesis.m`
Optimization/Calculate Costs	`func_calculateCosts.m`
Optimization/Dijkstra	`func_Dijkstra.m`
Intermediary/Detect Region Events	`func_detectRegionEvents.m`
Intermediary/Goal Finding	`func_GoGoal.m, func_GoalPlan.m`

2.3.1 Discussion of Proposed Solution

The proposed solution is different than the approach used by a previous student. In the previous approach, the equivalent of my Controller algorithm is based directly on the structure of the FSM. Custom code is created for each state, including the details of the specific example such as the location of the next task. This approach allows the code at each state to be modified for special cases. However, to automatically generate control software for the previous approach, a systematic method of code generation from user data would have to be developed. This would limit the adaptability of the resulting control software for special cases. The code at a state could still be modified by hand, but this would become

very unwieldy in a large FSM; the number of lines of code would be nearly proportional to the number of states in the FSM, and much of the code would be similar, making it difficult to find the position in the code to make a modification.

In my approach, the user data containing specific task locations is separated from the logic of the Controller algorithm that completes the mission of the FSM. The relevant user data are referenced by the Controller algorithm when needed. The same algorithm is used to evaluate each state, and the behavior varies based on the properties of the current state such as the number of controllable events. My approach requires all special cases to be considered and handled appropriately in the algorithm. However, excluding trivial examples, the resulting code is shorter, making it easier to read and debug. Since automatic code generation reduces the adaptability of the previous approach, the advantage of readability is the reason that prompted the adoption of my approach. Although my approach may be harder to change for special cases, it is more readable and correctable, leading to greater reliability of the resulting high-level control algorithm.

3 Auto-Generation of Control Software with MATLAB Tool

This chapter describes the efforts to complete the first objective. The features of the MATLAB tool are explained, and the resulting User Data file is described. The Main Control File is described as well as the Intermediate functions needed to control the low-level algorithms.

3.1 MATLAB Tool for User Input

A Graphical User Interface (GUI) was developed to facilitate data entry for the user. The GUI generates the User Data required to run an FSM file. A screen shot of the GUI is shown in Figure 3-1, and each numbered item is explained in its own section. The code can be found in *Appendix B: MATLAB Code for GUI*.

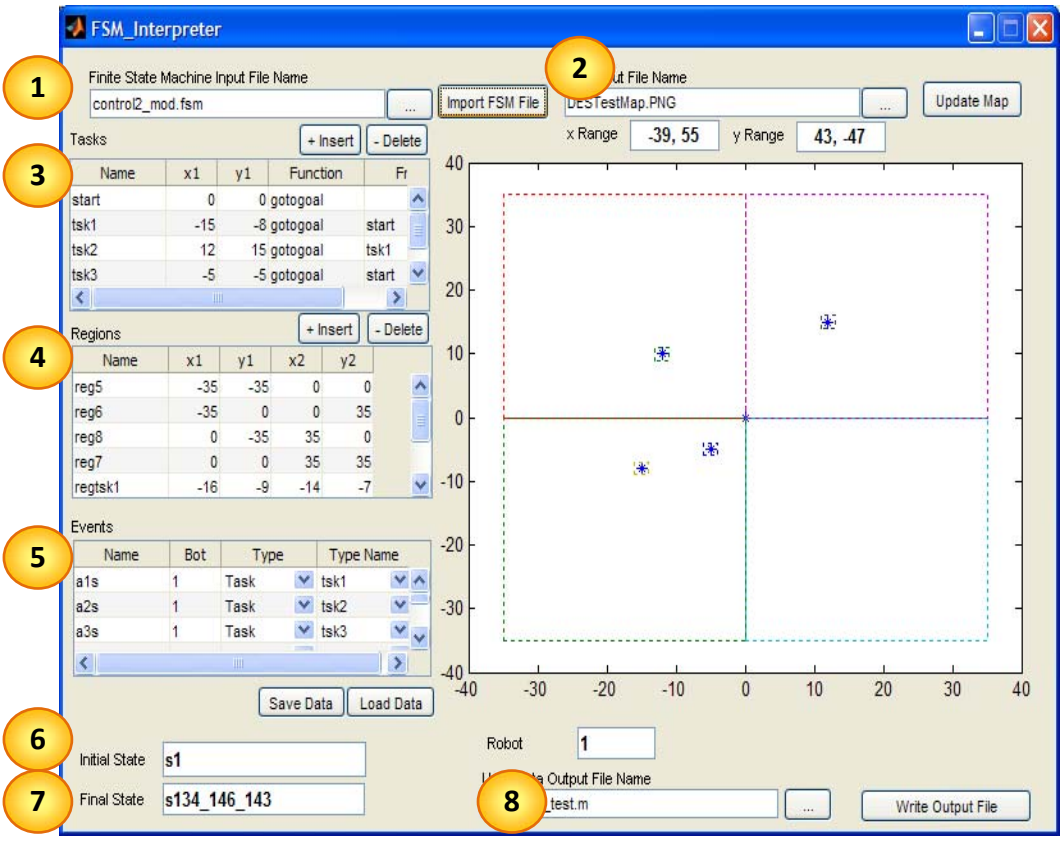

Figure 3-1 Screenshot of MATLAB GUI.

3.1.1 (1) Import Finite State Machine File

To use the FSM in the MATLAB-based control software, the *.fsm file must be imported into a MATLAB data structure. The text format of the input *.fsm file is shown in Figure 3-2.

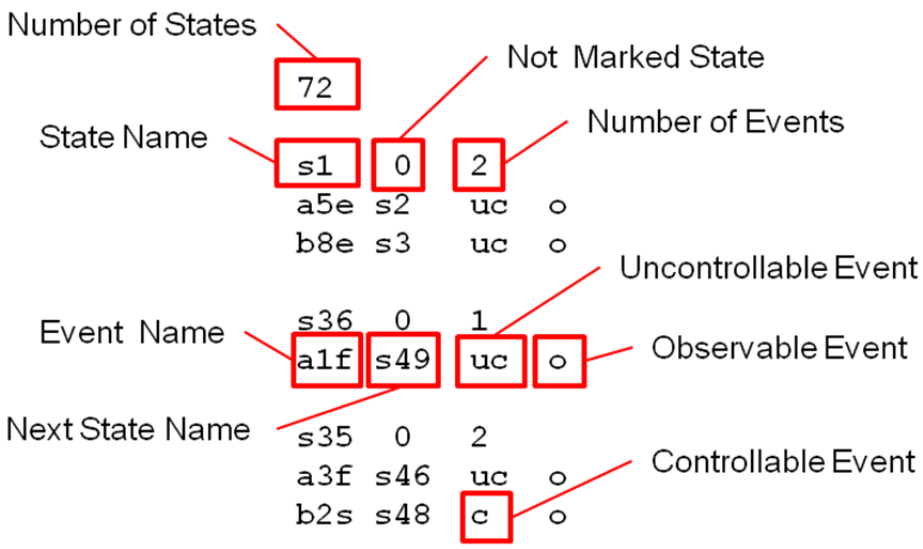

Figure 3-2 Format of the input FSM text file (*.fsm).

Figure 3-2 shows the total number of states on the first line of the *.fsm file. Each state is described in a text block. The first item in each block is the state name, `s1`, which is a string. The next value, `0`, is a boolean that indicates a non-marked state. The last item, `2`, is an integer: the number of valid events at this state. The following lines of the block describe each event. The first item in an event description is the event name, `a1f`. The next state name, `s49`, is the name of the state that is transitioned to after the event occurs. An uncontrollable event is designated with "`uc`" and a controllable event with "`c`". Similarly, an "`o`" designates an observable event.

The Import FSM function (`func_ReadFSM.m`) converts the text data contained in the *.fsm file into a MATLAB cell array entitled `states`. The format of the `states` cell array is shown in Figure 3-3.

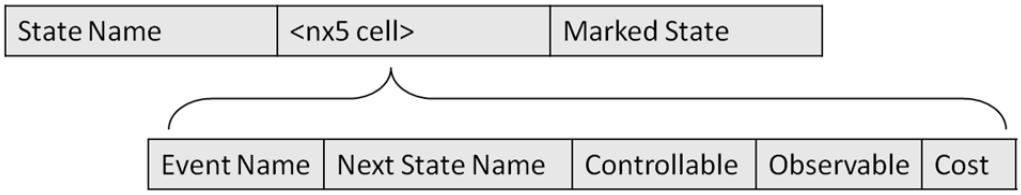

Figure 3-3 Data structure of `states` cell array.

Figure 3-3 shows that the valid events in the `states` cell array are listed as a nested cell array in the second column of `states`. The cost of an event is not included in the FSM file. Therefore, in the import process, the Cost column of the `states` array is left empty. It is filled in by the Calculate Costs function which is described in *Chapter 4*. The `states` cell array is filled in automatically by MATLAB based on the *.fsm file; no additional user input is needed.

The code for the Import FSM function is included in *Appendix D: MATLAB Code for Importing FSM*. The input to this function is the file name of the *.fsm file. The output is the `states` cell array and a list of unique event names.

To use the GUI to import the FSM file, enter the name of the file in the **Finite State Machine Input File Name** textbox. Click on the **Import FSM File** button to run the import function. When the import is complete, the list of event names will appear in the events table (table 4 in Figure 3-1). The default initial state and marked state will be automatically entered in textboxes 6 and 7 in Figure 3-1. To clear a previous FSM file, enter a new file name in the **Finite State Machine Input File Name** textbox, and re-import the FSM file.

3.1.2 (2) Enter map file name

To enable visualization of tasks, regions, and events, it is possible to upload a picture of the map used in Player/Stage. Enter the file name of the map picture in the **Map Input File Name** textbox. Click on the **Update Map** button to refresh the graph and display the map. The scaling of the map can be adjusted by editing the **x Range** and **y Range** textboxes.

Note that the map file name is not used in the auto-generation of control software. To use a different map in the robot simulation, edit the Player/Stage configuration file (*.cfg).

3.1.3 (3) Enter tasks data

The `tasks` cell array contains the data required for a low-level function to send a robot to a goal. It can be thought of as the details of a controllable event. The user defines the x- and y-coordinate of the location of each task in this table. The format of the table is shown in Figure 3-4.

Task Name	X1	Y1	Function

Figure 3-4 Data structure of `tasks` cell array.

The first column contains the task name, which can be any unique string. The second and third columns contain the x- and y-coordinate of the location of the task, respectively. After the x- and y-coordinates are entered, they will appear as blue stars on the map. The fourth column contains the name of the intermediate function (GoGoal or GoalPlan) that will send the robot to the task position.

All columns of the `tasks` array must be filled in by the user. To load data from previous saved data, use the **Load** and **Save** buttons on the GUI. To add or

delete tasks, after selecting a row, click on the **Insert** or **Delete** buttons above the table.

3.1.4 (4) Enter regions data

The `regions` cell array contains the data required to determine when a robot has entered a region. It can be thought of as the details of an uncontrollable event. The user defines the boundaries of each region in this table. Figure 3-5 shows the format of the `regions` table.

Region Name	X1	Y1	X2	Y2

Figure 3-5 Data structure of `regions` cell array.

The `regions` matrix has five columns and any number of rows. The first column contains the region name, which can be any unique string. The second thru fifth columns contain the upper and lower bounds of x-values and y-values for that region. It is not required to enter the minimum and maximum x- and y-values in a certain order, as long as the two points define opposing corners of a rectangle. After entering the x- and y-boundaries for a region, the map will display a dashed line outlining the boundaries of the region. It is important to ensure that the robots are always within at least one defined region, otherwise the control of the robot will not be defined, and the control software may crash. For this reason, the regions should be defined to cover the entire map that the robots will encounter.

All fields of the `regions` table must be filled in by the user. To add or delete regions, after selecting a row, click on the **Insert** or **Delete** buttons above the

table. To define a region graphically, after selecting a row, you can click twice on the map, once for each corner of a rectangle.

3.1.5 (5) Enter events data

The `events` cell array correlates each event with the details needed to either detect or control the event. The user defines these correlations in this table. The format of the `events` cell array is shown in Figure 3-6.

Event Name	Robot	Event Type	Type Name

Figure 3-6 Data structure of `events` cell array.

The first column of the `events` cell array is the event name such as `a1s`, which is extracted from the *.fsm file. The event name cannot be modified in the GUI; it must be edited in the *.fsm file. The rest of the fields are filled in by the user. Robot is an integer (either 1 or 2 in this example) which indicates which robot does the specified event. The third column specifies the event type as a string: “Task” for a task (controllable) event and “Region” for a region (uncontrollable) event. A task event means that a task is to be started, and the control software will access the `tasks` array to determine the goal to send the robot to, and the function to be used to send the robot to that goal. A region event means that the control software will detect when the robot is within the boundaries of the defined region. A region event can be used to define a task-completed (uncontrollable) event by creating a small rectangular region around the task location. The fourth column of the `events` array specifies the name of

the task or region event. This must match a previously defined item in the `tasks` and `regions` arrays.

3.1.6 (6) Enter initial state name

The **Initial State** textbox contains the initial state name. This serves as the starting point for the optimization algorithm to find a path through the FSM file. The default value is the first state in the *.fsm file, however, the user may modify the initial state using this textbox.

3.1.7 (7) Enter marked state name

The **Final State** textbox contains the marked state name. This is the end point for the optimization algorithm. When the Controller detects that this state is entered, the Main Control File will stop, and a “mission complete” message will be printed in the command window. If the mission actually continues after a marked state is reached, a new Main Control File should be started from a different initial state.

3.1.8 (8) Write user data to file

The user data consists of the `tasks`, `regions`, `events`, and `states` matrices. These matrices are written to an output file which is the User Data file. The User Data file is a text file that is run as a MATLAB script file at the beginning of the Main Control File to initialize the User Data matrices. The format of the user data is shown with the User Data file used for this example in *Appendix E: Sample User Data File.*

To generate the User Data, enter a file name in the **User Name Output File Name** textbox. Click on the **Write Output File** button to write the User Data file. To use the User Data file in a simulation, ensure that the file name is referenced in the Main Control File.

3.2 Controller: Main Control File

The Main Control File controls a single robot for a mission that is described by the User Data. To control multiple robots, use multiple instances of the Main Control File. Each instance of the file is the same except for the line which defines the robot (`bot = 1;`). The `bot` variable should be set to 1 for robot A and 2 for robot B. The main control file code (`Cer_VFSM_Controller_Start.m`) can be found in *Appendix F: MATLAB Code for Main Control File*.

The user input is generated off-line by means of a MATLAB tool as explained in *Section 3.1*. The user enters the data into the GUI, then presses a button to save the User Data file. The name of the User Data file (the file name that was entered or `UserData_Control2_thesis.m`) is referenced on the third line of the Main Control File. The same User Data file can be used for each instance of the Main Control File.

A flowchart of the Main Control File is shown in Figure 3-7. Yellow highlighting indicates a thesis section where the component is further explained.

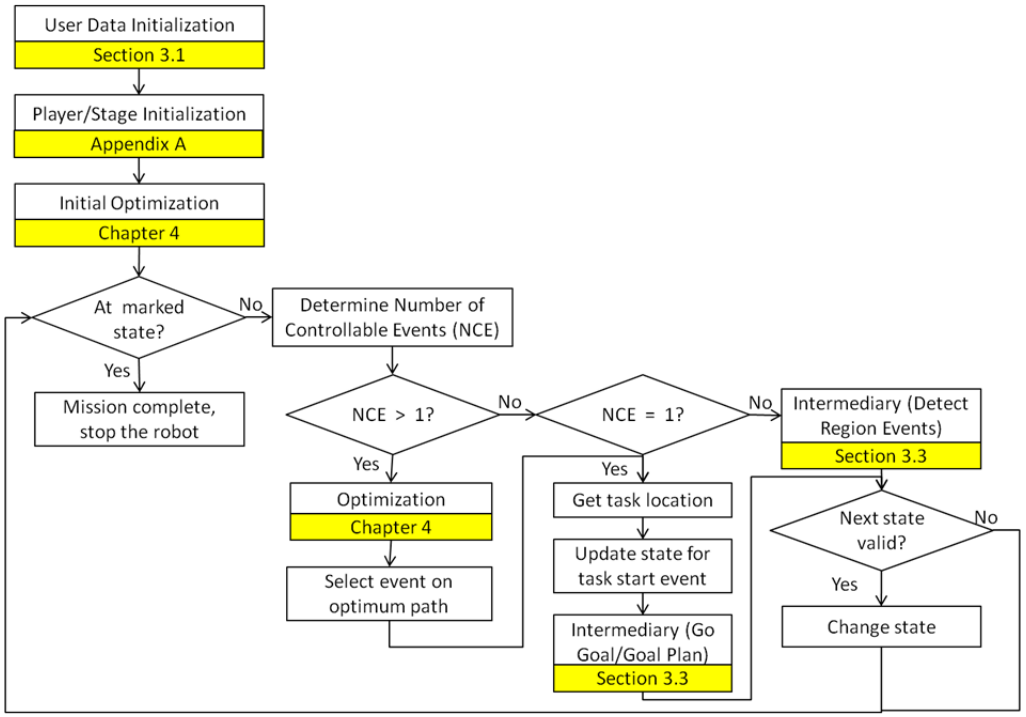

Figure 3-7 Flowchart for Main Control File.

As Figure 3-7 shows, when the Main Control File is started, the User Data and Player/Stage variables are initialized in MATLAB. Next, an initial path through the FSM is found using the Optimization algorithms. Then, the control code goes into the main loop beginning with the initial state selected by the user. Since this is not the marked state, the number of controllable events is determined for the current state by counting the controllable events in the `states` array. If there is more than one controllable (task-starting) event, the path is reoptimized to ensure that it is up-to-date, and the next controllable event on the path is chosen. When there is one controllable event (chosen or available), the task location is found from the User Data. The goal-finding (intermediate) function is used to send the robot to the task location until an event is detected, or the robot reaches the goal. The goal-finding function (Go Goal or Goal Plan) returns the event names that have been detected to the Main Control File. If there are no controllable events,

the program detects if any uncontrollable (region-crossing) events have occurred, returning these event names to the main program. If an event name is the next event on the path, the current state is updated. Regardless of whether the current state has been updated or not, the program cycles back to the beginning of the loop, checking whether this current state is the marked state. Once the path is completed, the algorithm stops, and a “mission complete” message is printed.

3.3 Intermediate Functions

The Intermediate functions are the Detect Region Events function and a goal-finding function (Goal Plan or Go Goal). The Detect Region Events function detects an uncontrollable region event for a robot. Both goal-finding functions send a robot toward a goal until an event is detected with the Detect Region Events function. When an event is detected, control is instantly returned to the calling function (Main Control File). The Goal Plan function uses low-level algorithms (running two additional MATLAB sessions) to send the robot to a goal, while the Go Goal function sends the robot to the goal using the same MATLAB session as the calling function. The user can choose to use either Goal Plan or Go Goal by editing the Function column of the `tasks` matrix in the MATLAB tool. In the UDM Control Lab, the Goal Plan function is the default.

3.3.1 Detect Region Events Function

The Detect Region Events function detects all regions that a robot is in, returning the name(s) of the corresponding uncontrollable event(s) such as “r6e.” It is called by the Go Goal and Goal Plan algorithms to determine if a change in

events has occurred. The code for this function (`func_detectRegionEvents.m`) is included in *Appendix G: MATLAB Code for Detect Region Events*.

The inputs to the Detect Region Events function include the robot the regions should be detected for, `bot`; the current position of the robot from Player/Stage, `pos2d`; the `regions` matrix; and the `events` matrix. Since different robots will correspond to different events (for example, a robot in region 5 could be the event 'a5e' or 'b5e.'), `bot` is used to pick the correct event from multiple choices. For robot A, `bot` is equal to 1; for robot B, `bot` is equal to 2. The output of the Detect Region Events function is `event_names`, a cell array containing a list of all events for the robot selected by `bot`. The function is called twice if all events are needed, one time for each robot.

In the Detect Region Events function, the position of the robot is compared with each user-defined region. If the position of the robot is within a region, the function adds the corresponding event name to the list of event names. The function does not detect controllable events such as starting a task because a task-start event does not correspond to a user-defined region. The function will return multiple event names if the robot is in multiple regions. To detect when an event has occurred, the calling function should initialize the event names with the robots' starting regions before commanding the robots to move. After some time, the Detect Region Events can be called again. The new list of event names should be compared to the previous list of event names. A change in the list of event names indicates that an event has occurred.

The Detect Region Events function does not refresh the robot position variable by executing a client read to Player/Stage. It expects that this will have already been done, and fresh data is being given to the function.

3.3.2 Goal Plan Function

The Goal Plan function sends the robot to a goal using the D*Lite and VFH low-level algorithms until an event is detected. It is called by the Main Control File after a goal for a robot has been chosen. The VFH algorithm and the mapping algorithm for D*Lite each run in a seperate MATLAB session. The code for the Goal Plan function (`func_GoalPlan.m`) can be found in *Appendix H: MATLAB Code for Goal Plan*.

The inputs to the Goal Plan function are the location of the task, `goal`; the position of robot A, `pos2dCer`; the position of robot B, `pos2dCer1`; the client for robot A, `clientCer`; the client for robot B, `clientCer1`; the map provided by the UDM mapping algorithm, `map`; the Player/Stage variable for VFH commands, `planner`; the robot (either 1 or 2), `bot`; the `events` matrix, and the `regions` matrix. The output is a consolidated list of both robot's event names from the Detect Region Events function, `all_events`.

The GoalPlan function uses D*Lite to plan a path toward the goal. The D*Lite algorithm plans a path from the goal location to the current position of a robot. It uses a map showing obstacles to determine intermediate waypoints (or breadcrumbs) to the goal. If an obstacle is detected, D*Lite does not have to re-plan the entire path; it can reuse previously calculated information.

After the path is planned, the Goal Plan function uses VFH to move the robot to each waypoint provided by D*Lite. VFH stands for Vector Field Histogram, and this low-level algorithm is a local motion planner. It provides velocity and turn rate commands to the robot to direct it to a waypoint. VFH includes obstacle avoidance. VFH is run by a separate MATLAB window in the UDM Control Lab setup.

After each waypoint has been reached, the Goal Plan function checks if an event has occurred using the Detect Region Events function. If an event is detected (whether the goal has been reached or not), the Goal Plan function returns all event names to the Main Control File. Since multiple events are detected by Detect Region Events, Goal Plan waits for a change in the event names array before returning control to the Main Control File. Figure 3-8 shows a flowchart of the Goal Plan function.

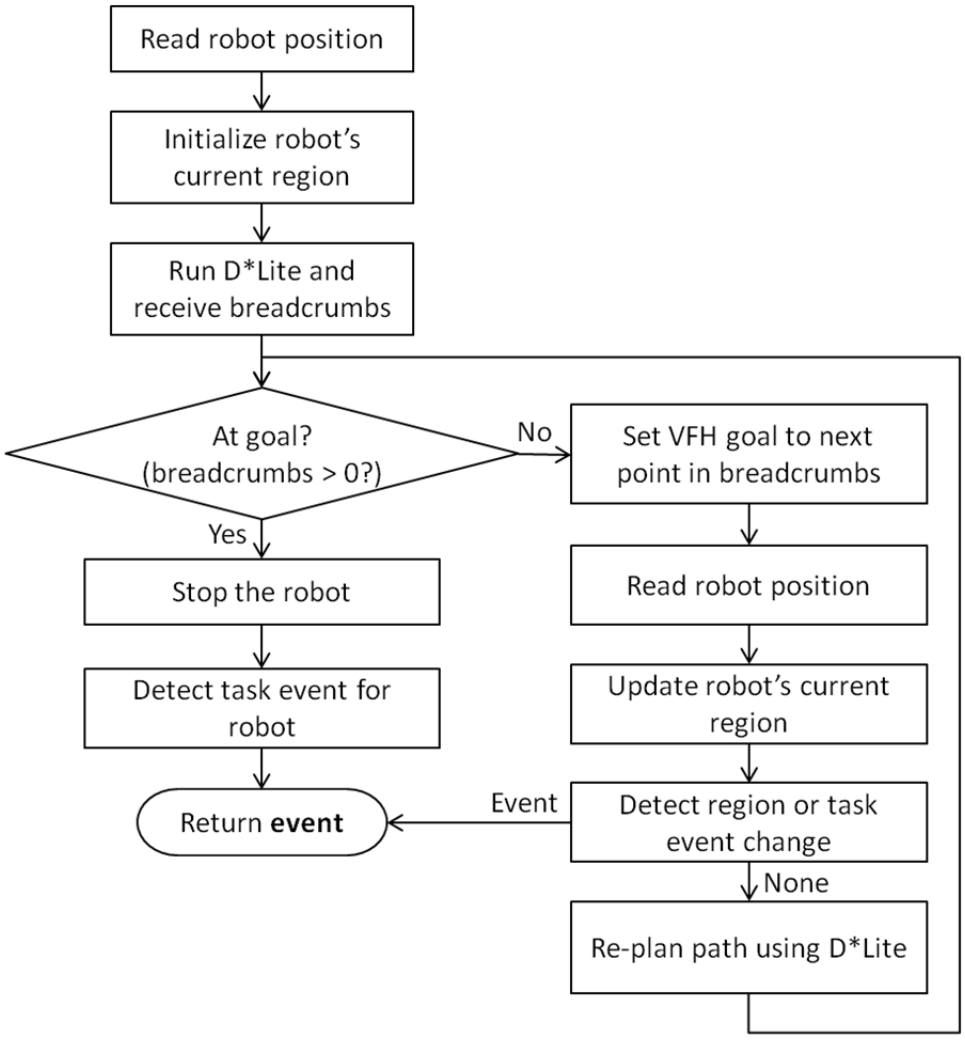

Figure 3-8 Flowchart for GoalPlan function.

As Figure 3-8 shows, the Goal Plan function calls the D*Lite algorithm to determine the best path toward the goal location. It then sets the VFH goal to the next breadcrumb on the path that D*Lite provides as output. It detects whether or not an event has occurred, until the robot reaches the goal. If an event occurs, the event names are returned to the Main Control File.

3.3.3 Go Goal Function

The Go Goal function sends the robot to a goal until an event is detected while avoiding obstacles without the use of low-level algorithms. It is an alternative to

the Goal Plan function, and it would be called similarly by the Main Control File after a goal for a robot has been chosen. The GoGoal function is a local path planner developed by using intuition. While it may not be as robust as VFH, it is useful for testing the high-level software, because it eliminates opening additional MATLAB windows to run low-level algorithms. The code for the Go Goal function (`func_GoGoal.m`) can be found in *Appendix I: MATLAB Code for Go Goal.*

The inputs to the Goal Plan function are the location of the task, `goal`; the Player/Stage client, `client`; the position of robot A, `pos2d1`; the position of robot B, `pos2d2`; the `laser` data of the robot from Player/Stage; the robot (either 1 or 2), `bot`; the `events` matrix, and the `regions` matrix. The output is `all_events`, which is a consolidated list of the event names from the Detect Region Events function.

The Go Goal function sends the robot in the direction of the goal using a turn-and-go strategy. If the initial angle is larger than 90 degrees, the robot turns in place before moving forward. When the robot senses an obstacle along its current direction, the function changes the default turn rate and forward speed in an attempt to go around the obstacle. In cases where the obstacle is directly in front of the robot, the direction (left/right) of the new turn rate is set by the sign (positive or negative) of the default turn rate. The magnitude of the turn rate is set by the proximity of the obstacle. As the distance to the obstacle decreases, the turn rate increases. The forward speed also varies based on the distance to the obstacle. As the distance to the obstacle decreases, the forward speed decreases. If

an obstacle is within range but not directly in front of the robot, the algorithm sets the turn rate to the first clear angle found in the laser data which is retrived from the robot in Player/Stage. It also reduces the forward speed.

Each time the Go Goal function recalculates the turn rate and forward speed, the function detects whether or not an event has occurred using the Detect Region Events function. If an event such as a region change has occurred, the Go Goal function returns to the Main Control File with the event names. If the robot reaches the goal, the task-finish event is returned to the Main Control File. Figure 3-9 shows the sequence of actions done by the Go Goal function.

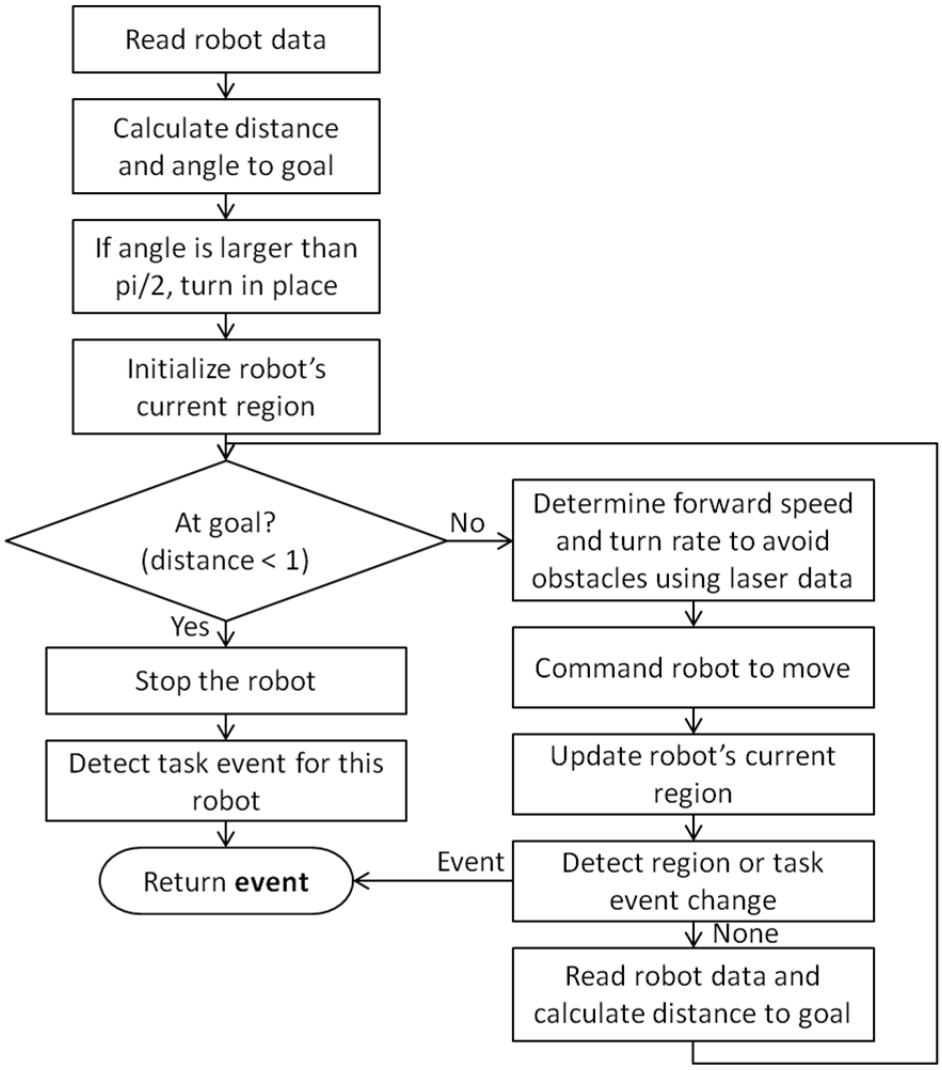

Figure 3-9 Flowchart for Go Goal function.

4 Optimization of Control Software with Graph-search Algorithm

This chapter describes the efforts to complete the second objective which is to optimize the actions of the two robots to complete a mission within the constraints of the finite state machine and demonstrate the solution in the Player/Stage environment. Two algorithms are used to plan the "best" path through multiple events of the FSM. The path through the finite state machine is initially calculated before the robots start moving, and is recalculated whenever a choice between controllable events needs to be made, because the costs may have changed since the initial optimization. The path is used to control the actions of the robots and to determine when the mission is complete.

The optimization algorithms are Calculate Costs and a modified Dijkstra's algorithm. Calculate Costs was developed as part of this project. The unmodified Dijkstra's algorithm was found on MATLAB Central File Exchange [7], and it was modified to optimize a path for two robots. The remainder of this chapter describes the Calculate Costs algorithm and the modified Dijkstra's algorithm.

4.1 Calculate Costs Algorithm

The Calculate Costs algorithm creates or updates the input matrices for Dijkstra's algorithm. The Main Control File runs the Calculate Costs algorithm before running the modified Dijkstra's algorithm. The code for the Calculate Costs algorithm (`func_calculateCosts.m`) is included in *Appendix J: MATLAB Code for Calculate Costs.*

The inputs to the Calculate Costs algorithm are the `states` matrix, `events` matrix, `tasks` matrix, and the current location of the two robots, `start`. The outputs are `A`, the adjacency matrix for Dijkstra's algorithm, and `C`, the cost matrix for Dijkstra's algorithm. The adjacency matrix and cost matrix are defined below.

The two input matrices for Dijkstra's algorithm redefine the `states` matrix in terms needed by a graph-search algorithm. The adjacency matrix defines the nodes (states) that are connected, and the cost matrix provides the weight of edges (events) in terms of distance. Each matrix is a square matrix the size of `states`. Each state is compared with every state in the `states` matrix, and the value of the corresponding entry is updated depending on the result of the comparison.

The value of the adjacency matrix is determined by whether two states, `i` and `j`, have an event connecting them or not. If the two states are connected, the value of the adjacency matrix `A(i,j)` is assigned `1`, otherwise it is assigned `0`.

The value of the cost matrix is determined by whether the event connecting two states is controllable or not. The cost of uncontrollable events is set to zero, even though its true cost is unknown and unpredictable. However, since the event cannot be controlled, its cost should not affect the control decisions of the controllable events. The cost of controllable events is the Euclidian distance between the locations of two tasks or the location of one task and the robot's current position (`start`). These costs change as the robot moves, and they would change if a task location is moved.

The tasks to use for the cost of controllable events are determined using the following assumptions and logic: Assume that an `event_name` from the `states`

matrix is a string of the form ‘a2s’ where the first character ‘a’ is robot A and ‘b’ is robot B, the second character ‘2’ is the task number (in this case from 1 to 4), and the third character ‘s’ signifies a task-start event and ‘f’ signifies a task-finish event.

For every task-start event where the cost is needed, there are two cases:

1) A previous task-finish event (signified by an ‘f’ in the third character of the `event_name`) exists that leads directly to the state containing the task-start event. For this case, the previous task is determined using the second character of the previous event name.
2) A previous event does not exist or it is not a task-finish event. In this case, use the current position of either robot A or B as determined by the first character of the `event_name`.

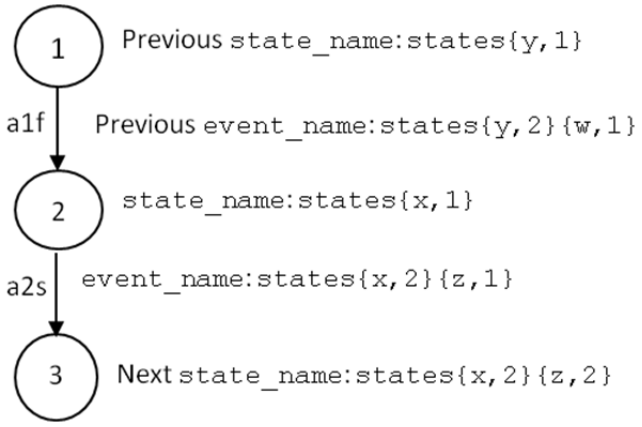

Figure 4-1 Excerpt of FSM for Calculate Costs example.

Figure 4-1 shows a fragment of an FSM. At state 2, the `event_name` is ‘a2s’. The previous `event_name` is ‘a1f’. The Calculate Costs algorithm calculates the distance from task 1 to task 2 to use as the cost for this event, because the second character of the previous `event_name` is ‘2’.

4.2 Modifications to Dijkstra's Algorithm

The modifications to Dijkstra's algorithm utilize the advantage of two robots working simultaneously to plan the shortest path though the FSM. The modified Dijkstra's algorithm is called by the Main Control File after the Calculate Costs algorithm when an updated path is needed. The code for the modified Dijkstra's algorithm (`func_Dijkstra.m`) can be found in *Appendix K: MATLAB Code for Dijkstra's Algorithm.*

The inputs to Dijkstra's algorithm are the adjacency matrix, `A`; the cost matrix, `C`; a vector of starting nodes, `SID`; and a vector of final nodes, `FID`. The adjacency matrix and the cost matrix are created by the Calculate Costs algorithm. In this implementation of Dijkstra's algorithm, one start node and one final node are used instead of vectors. However, it is possible to modify the Controller code to include multiple starting and final nodes. The outputs of Dijkstra's algorithm are the shortest path from the initial node (state) to the final node (marked state), `path`, and the total cost of all the edges (events) on the path, `totalCost`. The `path` contains the indices of the states in the `states` matrices that are on the shortest path.

To find the path with the minimum cost, Dijkstra's algorithm begins with the cost of the initial node set to zero. The cost to reach each neighboring node is found by adding the new edge cost to the cost of the open node ('Open' refers to the node that is currently being optimized.). Each neighboring node's cost is recorded, and the node with the lowest cost is added to the path. When this has been completed, the open node is settled ('Settled' means that the shortest path has been found for a node.). Dijkstra's algorithm continues by selecting the

neighboring node with the lowest edge cost, and repeating until the final node is settled.

In this project, the nodes are represented by states in the FSM file, and the edges are represented by events. For example, Figure 4-2 shows an initial cost of 15 at the state B3.A1 as a result of robot A completing task 1 and robot B completing task 3. If state B3.A1 is the open node, then the cost for its neighbors is calculated. The cost for robot A completing task 2 (event a2s) is 8, while the cost for robot B completing task 4 (event b4s) is 10. The unmodified Dijkstra's algorithm chooses path with the minimum cost, so at this step it would choose state B3.A1.A2 (highlighted in yellow) with a cost of 23.

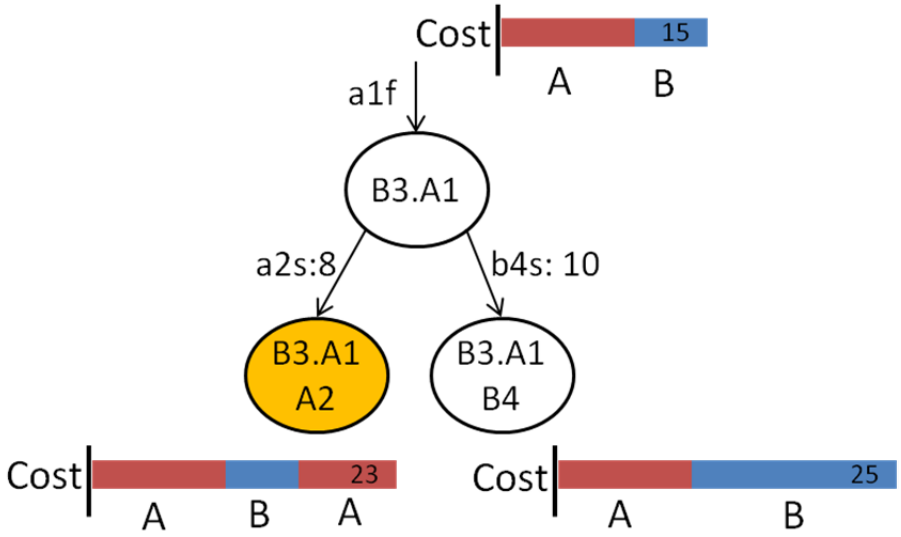

Figure 4-2 Example FSM for unmodified Dijkstra's algorithm.

The modification to Dijkstra's algorithm was related to the calculation of the edge costs. The Calculate Costs algorithm calculates the costs for each controllable event, but it does not take into account the effect of both robots traveling simultaneously. The modification keeps track of the accumulated cost for each robot separately. It minimizes the total cost of the robot with the

maximum accumulated cost. The robot with the maximum accumulated cost depends on the current state.

An example of this concept is provided in Figure 4-3. The shortest path to state B3.A1 has an accumulated cost for robot A of 10, while the accumulated cost for robot B is 5. The choice at this step of the algorithm is whether robot A should start task 2 with a cost of 8, or whether robot B should start task 4 with a cost of 10. The maximum cost at state B3.A1.A2 is 18 which is robot A's accumulated cost, while the maximum cost at state B3.A1.B4 is 15 which is robot B's accumulated cost. The minimum of these two costs is 15, so the modified Dijkstra's algorithm chooses robot B to complete task 4.

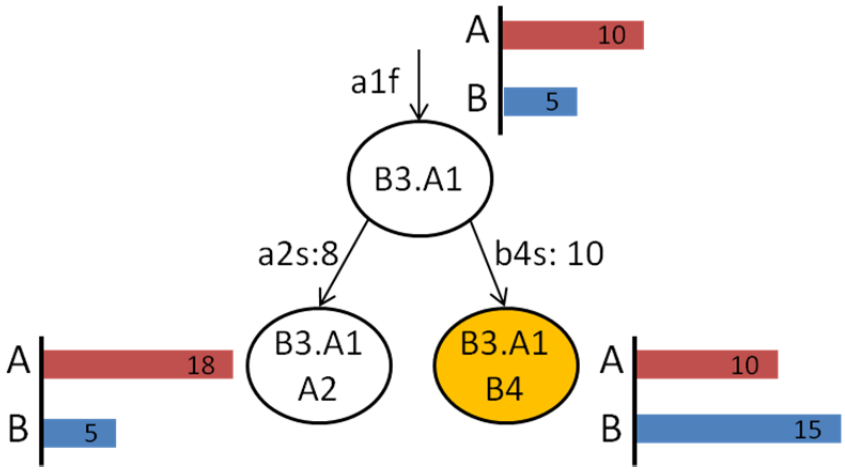

Figure 4-3 Example FSM for modified Dijkstra's algorithm.

4.3 Simulation Results

The control software was simulated in Player/Stage to verify that it worked as expected. The control software was tested during the development of the Main Control File, as well as after the optimization algorithms were completed. In this section, the results of a simulation are shown and discussed.

The location of the tasks and the starting position of the robots for the simulation are shown in Table 4-1. Regions 5-8 were defined as the four quadrants of a Cartesian-coordinate system. The task-finish events, which are uncontrollable events, were defined as a robot entering a 1 meter square around each task location. The user-defined regions are shown in Table 4-2. The `control2.fsm` file which represents the FSM shown in Figure 2-7 was used. The `control2.fsm` file is included in *Appendix C: FSM File*.

Table 4-1 Locations for Player/Stage Simulation.

Location	x	y
Start A	-9	-19
Start B	9	-19
Task 1	-2	-5
Task 2	7	8
Task 3	-5	3
Task 4	-13	8

Table 4-2 Regions for Player/Stage Simulation.

Region	x1	y1	x2	y2
reg5	-35	-35	0	0
reg6	-35	0	0	35
reg8	0	-35	35	0
reg7	0	0	35	35
regtsk1	-1	-4	-3	-6
regtsk2	6	7	8	9
regtsk3	-4	2	-6	4
regtsk4	-14	9	-12	7

The instructions to run a simulation in the UDM Control Lab are included in *Appendix A: Player/Stage Simulation Setup and Instructions*. The Go Goal function was used for this simulation, and similar results occur when using the Goal Plan function in the UDM Control Lab setup. Screenshots of the simulation are shown in Figure 4-4 through Figure 4-7.

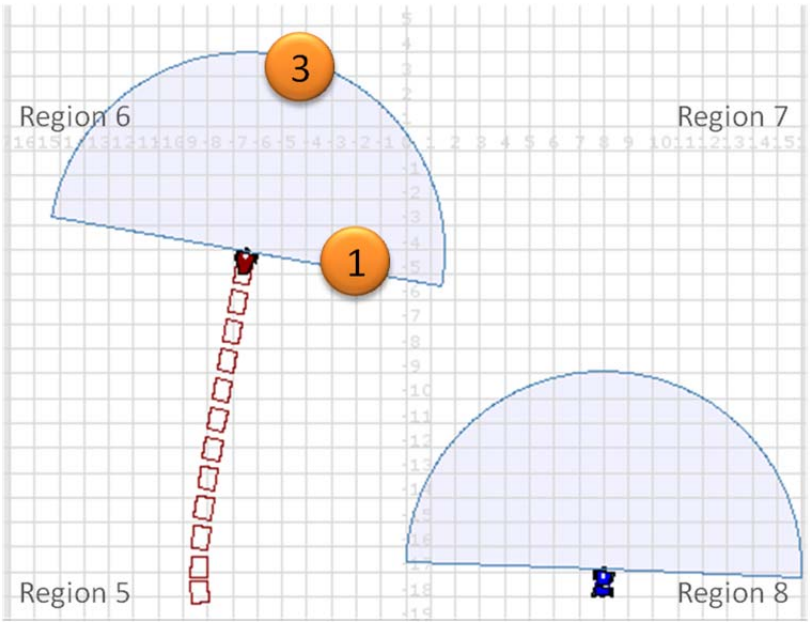

Figure 4-4 Robot A begins Task 3, and Robot B waits for Region 5 to clear.

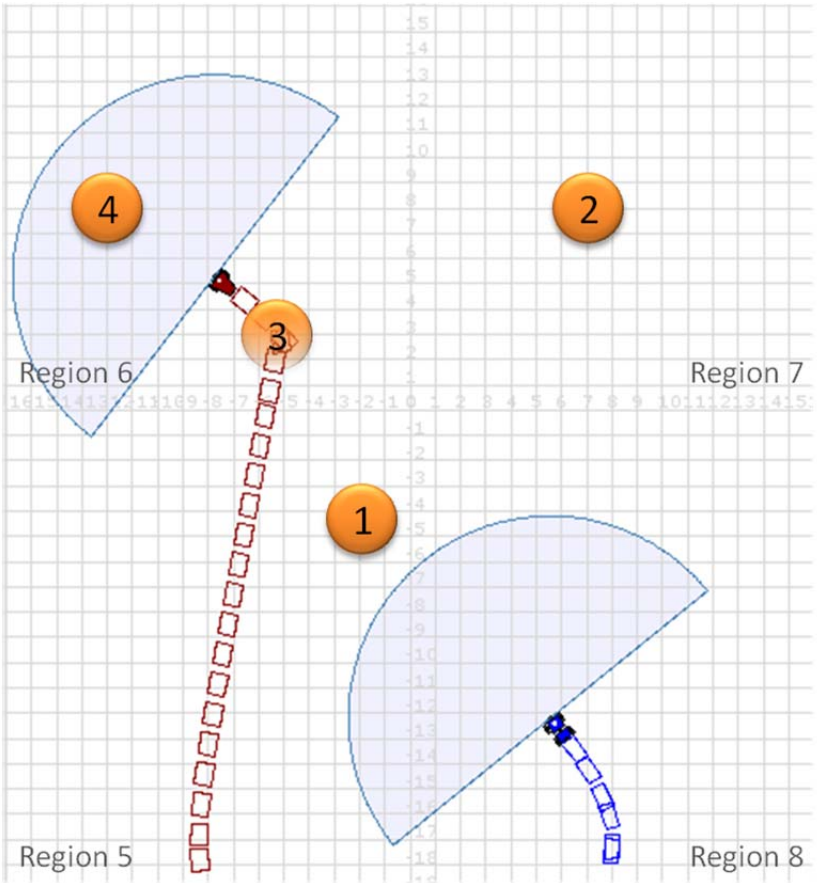

Figure 4-5 Robot A begins Task 4, and Robot B begins Task 1.

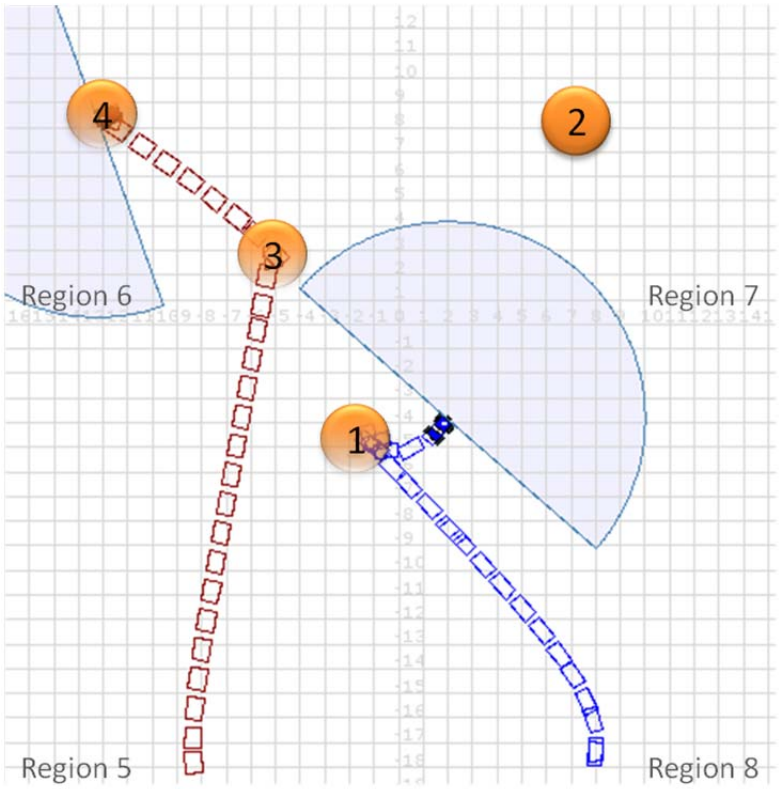

Figure 4-6 Robot A completes Task 4, and Robot B begins Task 2.

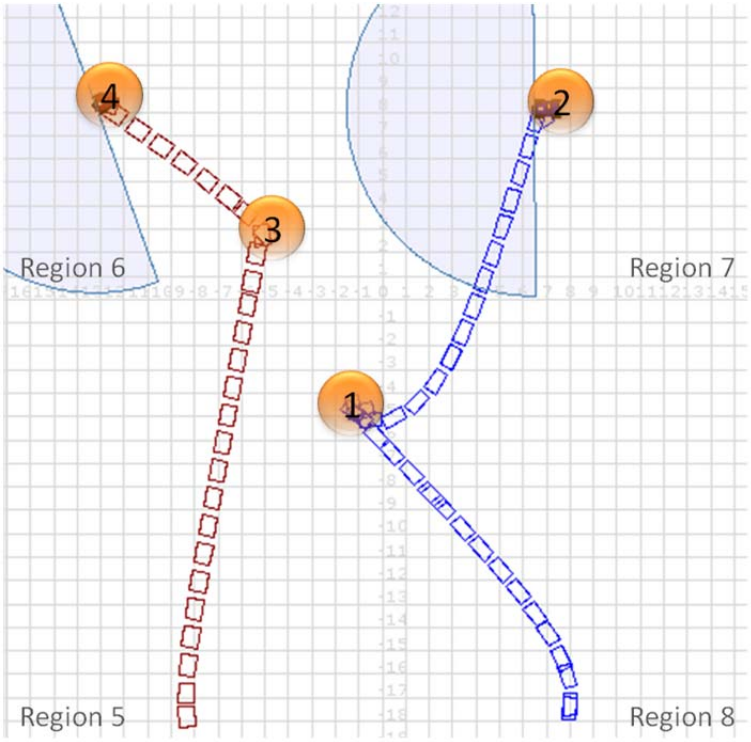

Figure 4-7 Robot A has completed Task 4, and Robot B has completed Task 2.

Figure 4-4 through Figure 4-7 show that the Controller chose robtot A to complete tasks 3 and 4, and robot B to complete tasks 1 and 2 as the optimum path. The total cost calculated by Dijkstra's algorithm was 33.6 meters. A hand calculation matches the optimization algorithm result. The MATLAB command window status for the simulation is shown in Table 4-3.

Table 4-3 MATLAB status output for both robots.

Robot A	Robot B
Elapsed time is 0.174457 seconds. Robot 1: Path planned from 's1' to 's134:146:143': Path = [s1 s3 s4 s9 s14 s19 s24 s33 s46 s59 s74 s93 s109 s121 s125:133:145 s135:142 s134:146:143] Total Cost: 21.77 1. State 1: s1 Detected Event: b8e 2. State 50: s3 Detected Event: a5e 3. State 49: s4 Elapsed time is 0.212016 seconds. Robot 1: Path planned from 's4' to 's134:146:143': Path = [s4 s9 s14 s19 s24 s33 s46 s59 s74 s93 s109 s121 s125:133:145 s135:142 s134:146:143] Total Cost: 21.77 Event Chosen: a3s (tsk3) GoToGoal: -5.00,3.00 2. State 48: s9 Detected Event: a6e 3. State 56: s14 Event Chosen: b1s (tsk1) Detected Event: a3f 4. State 52: s19 Elapsed time is 0.136695 seconds. Robot 1: Path planned from 's19' to 's134:146:143': Path = [s19 s27 s50 s64 s76 s95 s112 s123 s132 s141 s134:146:143] Total Cost: 19.96 Event Chosen: a4s (tsk4) GoToGoal: -13.00,8.00 2. State 39: s27 Detected Event: a6e 3. State 11: s50	Elapsed time is 0.173723 seconds. Robot 2: Path planned from 's1' to 's134:146:143': Path = [s1 s3 s4 s9 s14 s19 s24 s33 s46 s59 s74 s93 s109 s121 s125:133:145 s135:142 s134:146:143] Total Cost: 21.77 1. State 1: s1 Detected Event: b8e 2. State 50: s3 Detected Event: a5e 3. State 49: s4 Elapsed time is 0.146807 seconds. Robot 2: Path planned from 's4' to 's134:146:143': Path = [s4 s9 s14 s19 s24 s33 s46 s59 s74 s93 s109 s121 s125:133:145 s135:142 s134:146:143] Total Cost: 21.77 Event Chosen: a3s (tsk3) 2. State 48: s9 Detected Event: a6e 3. State 56: s14 Event Chosen: b1s (tsk1) GoToGoal: -2.00,-5.00 Detected Event: a3f 4. State 52: s19 Elapsed time is 0.166756 seconds. Robot 2: Path planned from 's19' to 's134:146:143': Path = [s19 s27 s50 s64 s76 s95 s112 s123 s132 s141 s134:146:143] Total Cost: 19.96 Event Chosen: a4s (tsk4) 2. State 39: s27 Detected Event: a6e 3. State 11: s50 Event Chosen: b1s (tsk1)

Event Chosen: b1s (tsk1) Detected Event: a4f 4. State 69: s64 Elapsed time is 0.132138 seconds. Robot 1: Path planned from 's64' to 's134:146:143': Path = [s64 s76 s95 s112 s123 s132 s141 s134:146:143] Total Cost: 6.38 Event Chosen: b1s (tsk1) 2. State 59: s76 Detected Event: b5e 3. State 33: s95 Detected Event: b1f 4. State 14: s112 Event Chosen: b2s (tsk2) 5. State 5: s123 Detected Event: b8e 6. State 67: s132 Detected Event: b7e 7. State 55: s141 Detected Event: b2f 8. State 17: s134:146:143 Mission complete!	GoToGoal: -2.00,-5.00 Event Chosen: b1s (tsk1) GoToGoal: -2.00,-5.00 Detected Event: a4f 4. State 69: s64 Elapsed time is 0.137905 seconds. Robot 2: Path planned from 's64' to 's134:146:143': Path = [s64 s76 s95 s112 s123 s132 s141 s134:146:143] Total Cost: 6.38 Event Chosen: b1s (tsk1) GoToGoal: -2.00,-5.00 2. State 59: s76 Detected Event: b5e 3. State 33: s95 Detected Event: b1f 4. State 14: s112 Event Chosen: b2s (tsk2) GoToGoal: 7.00,8.00 5. State 5: s123 Detected Event: b8e 6. State 67: s132 Detected Event: b7e 7. State 55: s141 Detected Event: b2f 8. State 17: s134:146:143 Mission complete!

As shown in Table 4-3, the status outputs for Robot A and B are nearly identical. It is important to realize that each MATLAB window is detecting events for both robots. This is necessary to ensure that the FSM file is followed, and no robot breaks any rules.

In addition, note that each time a controllable event occurs in the FSM file, the optimization is re-run to determine the best path before sending the robots to the goal. With a single processor, this is not a significant issue because all processes are computed in series. With multiple processors, it might be more efficient to begin sending the robots to the old optimized goal before recalculating the optimum path. If the new optimized goal is not the same as the old optimized goal, the goal of the low-level algorithm could be adjusted.

5 Conclusions

This project applies Discrete-Event Systems theory to multi-robot control systems. The example demonstrates the feasibility of the DES approach at a level that has not been performed before. The project provided lessons for modeling and implementation for this specific application. Tools were developed that will allow larger problems to be addressed.

5.1 Summary of Contributions

The two main objectives of this project were successfully completed. I developed an interface between the FSM control logic and the low-level robot algorithms. For the first objective, I developed a MATLAB tool to automatically generate the User Data. For the second objective, I created an algorithm to update the costs of the robots, and modified Dijkstra's algorithm to optimize a sequence of control actions for multiple robots operating simultaneously. Finally, I demonstrated that the control software works by simulating in Player/Stage.

The key to creating the interface shown in Figure 2-8 was to determine the required information that must be included in the matrices. This information may change based on the application, and the matrices (`states`, `events`, `tasks`, and `regions`) in this example are representative of the information that is needed for a robotics application. In this project, controllable events were mapped to task locations, and uncontrollable events were mapped to regions. To generalize this work, instead of tasks and regions, the controllable and uncontrollable events could be defined in terms that correlate with different low-level functions for other specific applications.

5.2 Future Work

There is always more that could be done in any project, and this one is no exception. Future work includes expanding the MATLAB tool for use with other applications, testing the control software with more complex models and on actual hardware, and improving the efficiency of the optimization algorithms. These efforts are described in the remainder of this section.

The MATLAB tool could be expanded for additional DES applications, such as manufacturing or communication systems, through generalizing the tasks and regions to controllable and uncontrollable events. Instead of determining whether the event is a “task” event or a “region” event, the Controller could determine whether it is an uncontrollable or controllable event, and adjust its behavior accordingly. In addition, the structure of the User Data file could be modified. Instead of four matrices (`states`, `events`, `tasks`, and `regions`), one matrix could contain the states and all of the corresponding event properties. The uncontrollable and controllable event properties (which are currently stored in the `regions` and `tasks` arrays) could be included in the nested events array as part of the `states` array.

Another improvement to the MATLAB tool that would allow it to be used for more complex models is to select multiple marked states. It is possible to have multiple marked states in a FSM, although in this example, only one marked state is used. Dijkstra’s algorithm has the capability to find paths to more than one final state. The MATLAB tool could be modified to include the selection of multiple marked states (perhaps with a text box and semicolons) and the result could be integrated with the Main Control File.

The control software could be implemented on actual hardware in UDM's Advanced Mobility Lab. Although the high-level control software was simulated with the current low-level algorithms, it was not tested on real robots. To improve the robustness of the control software, error handling could be further developed, and faults could be addressed.

The optimization algorithms could be improved by saving cost information that has not changed since the last optimization. A data structure would save the cost information from one run to another, so that if the costs had not changed, the path would not be recalculated. Or, if only some of the costs had changed, only the relevant portions of the path would need to be recalculated. The new cost function could then be verified by simulation and on actual robots.

The modified Dijkstra's algorithm could be improved to only choose between controllable events in certain cases. In a case with more than two robots, it is possible that a state could contain multiple controllable events and one or more uncontrollable events. For example, robot A start task 1, robot B start task 2, and robot C finish task 3. The current modified Dijkstra's algorithm would choose the uncontrollable event "robot C finish task 3" because it is defined with zero cost. Compared to the other tasks, which have a finite non-zero cost, the uncontrollable event would be chosen. This choice would not be desirable in terms of time because the controller cannot determine the amount of time it will take for an uncontrollable event to occur. To solve this problem, Dijkstra's algorithm could be further modified to choose the controllable event with the lowest cost, instead of any event with the lowest cost.

6 References

[1] R. Hill, "Discrete Event Systems," in *The Control Handbook, Second Edition*, CRC Press, 2010, pp. 5-81-5-82.

[2] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control and Optimization,* vol. 25, no. 1, pp. 206-230, 1987.

[3] K. Rudie and W. M. Wonham, "Think globally, act locally: Decentralized supervisory control," *IEEE Trans. Automatic Control,* vol. 37, no. 11, pp. 1692-1708, 1992.

[4] K. Schmidt, J. Reger and T. Moor, "Hierarchical control of structural decentralized DES," in *2004 International Workshop on Discrete Event Systems - WODES'04*, 2004.

[5] O. Kamach, L. Pietrac and E. Niel, "Multi-model approach to discrete-event systems: Application to operating mode management," *Mathematics and Computers in Simulation,* vol. 70, pp. 394-407, 2005.

[6] J. Kirk, "Fixed Start/End Point Multiple Traveling Salesman Problem - Genetic Algorithm," 07 Nov 2011. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/21299-fixed-startend-point-multiple-traveling-salesmen-problem-genetic-algorithm. [Accessed Apr 2012].

[7] J. Kirk, "Advanced Dijkstra's Minimum Path Algorithm," 01 May 2009. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/20025-advanced-dijkstras-minimum-path-algorithm. [Accessed Apr 2012].

[8] A. Patel, "Introduction to A*," Amit's Game Programming Site, 2012. [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. [Accessed Apr 2012].

[9] S. Koenig and M. Likhachev, "D*Lite," American Assiciation for Artificial Intelligence, 2002. [Online]. Available: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf.

[10] I. Ulrich and J. Borenstein, "VFH*: Local Obstacle Avoidance with Look-Ahead Verification," in *IEEE International Conference on Robotics and Automation*, San Francisco, CA, 2000.

[11] J. van Schuppen, "Mathematical Control and System Theory of Discrete Event Systems," 19 Jan 2008. [Online]. Available: http://homepages.cwi.nl/~schuppen/courses/lecturenotes/controldeslecturenotes.pdf. [Accessed 25 Jun 2012].

[12] S. Lafortune and D. Teneketzis, "UMDES-LIB: Library of Commands for Discrete Event Systems Modeled by Finite State Machines," 22 Aug 2000. [Online]. Available: http://www.eecs.umich.edu/umdes/publication_files/sldtwodes00.pdf. [Accessed 25 Jun 2012].

[13] K. Barry, "Player/Stage in coLinux with MATLAB," University of Detroit Mercy, 2009.

[14] J. Owen, "How to Use Player/Stage," 10 July 2009. [Online]. Available: http://www-users.cs.york.ac.uk/jowen/player/playerstage-tutorial-manual-2.1.pdf. [Accessed Feb 2012].

7 Appendix A: Player/Stage Simulation Setup and Instructions

7.1 Description of Setup

The Player/Stage open-source software was used for simulation of the test case. This setup consists of two pieces of software working together. The Player part is a defined set of interfaces and drivers that can run in combination with Stage, or an actual robot. The Stage simulation part receives the commands from Player and responds as the actual device would.

Player/Stage runs in Linux or the Mac OS. However, it can be run on Windows using a virtual machine. Two different Player/Stage configurations were used in testing. One was a personal laptop, which had a Player/Stage coLinux environment installed as detailed in Kevin Barry's tutorial [13]. The other was the Controls Lab at UDM which had Ubuntu installed as a virtual machine. On my laptop, Windows ran MATLAB, and Linux ran the Player/Stage simulation. In the Controls Lab, Ubuntu/Linux ran both MATLAB and Player/Stage.

Web tutorials are available that explain in detail how to use Player/Stage [14]. The referenced tutorial explains how to set up the Player/Stage configuration files. However, the control code used as an example in the tutorial is written in the C++ programming language. Since a MATLAB Mex software is used to connect to Player/Stage's C library in my setup, the syntax used in MATLAB is slightly different than the tutorial's examples. This makes it hard to apply the examples directly in MATLAB. The remainder of this appendix will focus on the key parts

of the Player/Stage initialization section of the main control file so that the user may debug some basic problems that may occur.

The Player/Stage initialization section of the code starts with initialization of the global variables. The global variables are shared between the main control file and the low-level control functions.

After the initialization of the global variables, a cleanup section attempts to destroy any variables that already exist in Mex or Stage. The syntax is as follows:

```
try, player('destroy', var_name); end;
```

This can be used for both laser references and position references. For client references, use ‘`client_disconnect`’ before attempting to destroy the variable.

Next, the client variable is established. The client is a reference variable to connect with the Player/Stage simulator. The syntax to create and connect a new client is as follows:

```
clientCer = player('client_create', 'localhost', 6665)
player('client_connect', clientCer)
```

The first argument `'client_create'` directs the Mex program what to do. The second argument needed to be changed from `'localhost'` to my laptop’s IP address, because Windows was operating MATLAB and Linux was operating Player/Stage. The last argument is referred to as the port. The port needs to match with the port in the Player configuration file (*.cfg) or the program will not work.

Note that either a single client or multiple clients can be used for multiple robots, depending on the setup of the simulation or actual hardware. For actual hardware, it is likely that multiple clients would be used, however, in simulation this doesn’t matter, so only one client can be used.

The next step is to initialize and subscribe to a robot position variable. This is a reference to read the robot's position. This variable is also used when commanding the robot to change position and go to a goal. The syntax is as follows:

```
pos2dCer = player('position2d_create', clientCer, 0);
player('subscribe', pos2dCer);
```

The `'position2d_create'` tells the Mex program what to do. The `clientCer` variable must match the name of the client variable that was established earlier. The `'0'` is an integer index variable. It must match the index of the robot's position2d in the Player configuration file.

A laser is created and subscribed to in the same way as a position variable. The laser returns readings from a laser device in Player Stage. The syntax is as follows:

```
laser = player('laser_create', clientCer, 0);
player('subscribe', laser);
```

It is also important that the laser index matches the index in the Player/Stage configuration file.

The key point to take away from this section is that if something goes wrong with the Player/Stage initialization section, there are several things that could be wrong. However some common ones include (1) the port/indices do not match the configuration file, (2) the Mex program is not working properly, or (3) Player/Stage is not setup right on your computer.

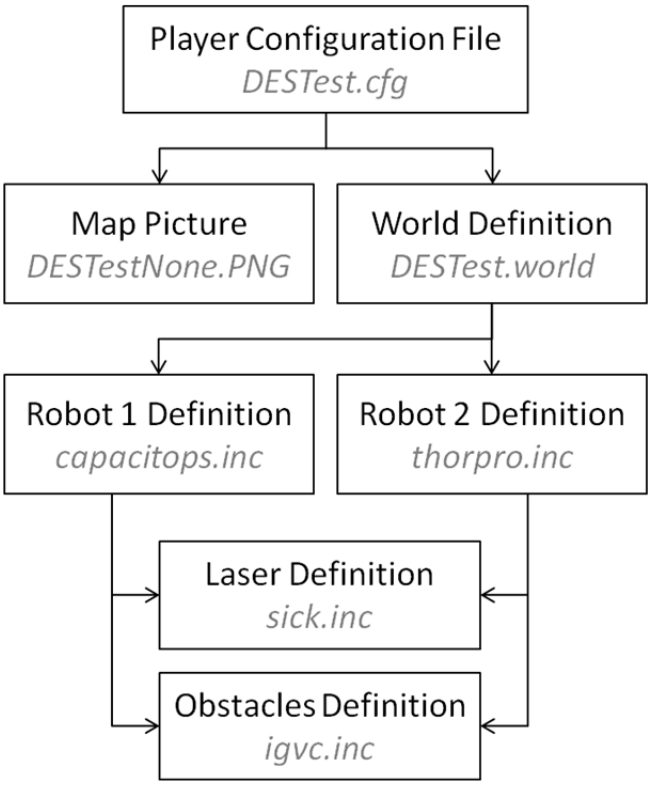

Figure 7-1 Player/Stage files interaction

As Figure 7-1 shows, the configuration file (*.cfg) relies on several other files in this setup to work properly. The world file is always needed, however, the number of subfiles can vary depending on the number of robots (with different dimensions and configurations), and the number of other subfiles such as the ones shown that define obstacles, and a laser for both robots.

7.2 Instructions to Run Player/Stage Simulation

The control software was tested in the Player/Stage simulation environment at UDM's Control Systems Lab. This section provides step-by-step instructions to run the code in UDM's Control Systems Lab.

1. Login to the “Wang” user on desktop computer, and open the Ubuntu Virtual Machine to access Linux.
2. Open a terminal window. This gives command line access to Linux. Basic Linux commands include the following:
    - `ls`: Lists all documents in the current folder.
    - `cd`: Change to a new directory (folder).
    - `player`: Start the Player/Stage simulator with a configuration file.
3. Enter the command: `cd Desktop/IGVC_Code/Stage` to switch to the Stage directory.
4. Enter the command: `player DESTest.cfg` to start the Stage simulator with the parameters in the DESTest.cfg configuration file. A new window will appear with two robots on a field as shown in Figure 7-2.

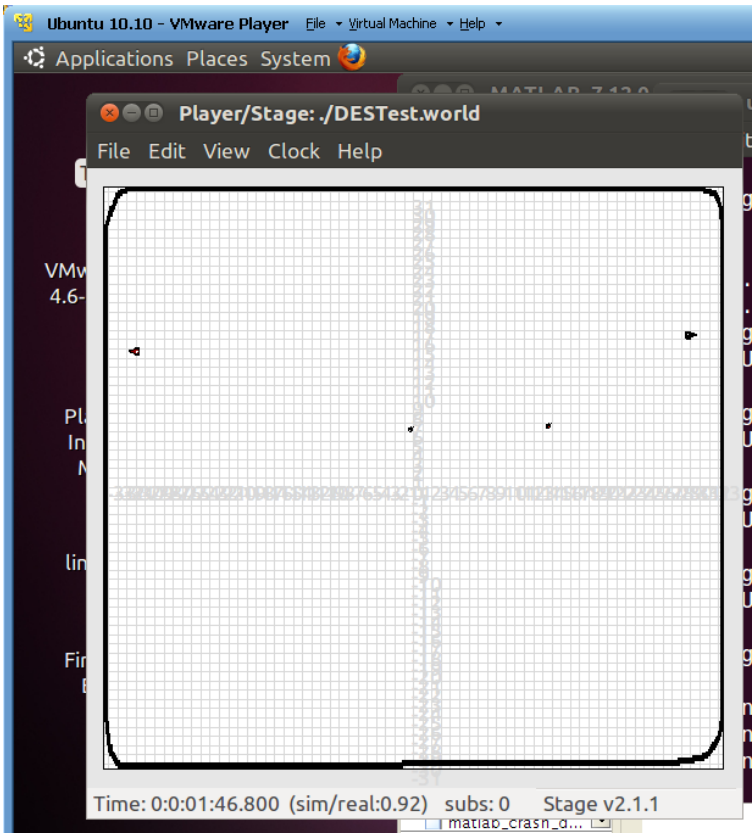

Figure 7-2 Screenshot of Player/Stage simulator.

As Figure 7-2 shows, there is are two robots and two obstacles. The black robot is Cer, or robot A; and the red robot is Cer1, or robot B. The number and position of obstacles can be changed in the configuration file as Section 7.1 describes.

5. Open 6 MATLAB sessions, one for each of the files listed below. The files can be found in the following location: Desktop/IGVC_Code/NavChallenge/CerberusDstar/NavDstar2/DES
    - CerMain/Cer_VFSM_Controller_Start.m (or the name of the main control file for robot A)
    - Cermapping/Cermapping2.m

- CerVFHPlanner270/CerVFHCerberusStart.m
- Cer1Main/Cer1_VFSM_Controller_Start.m (or the name of the main control file for robot B)
- Cer1mapping/Cer1mapping2.m
- Cer1VFHPlanner270/Cer1VFHCerberusStart.m

6. Ensure that all the files need to run the main control file are located in the Shared folder. These files are listed below:
    - func_calculateCosts
    - func_detectRegionEvents
    - func_Dijkstra
    - func_GoalPlan
7. The MATLAB windows containing the main control files can be positioned at the right top and bottom halves of the screen, and the Player/Stage animation window can be positioned to the left of the screen as shown in Figure 7-3. This allows you to see the command window and the animation window simultaneously.

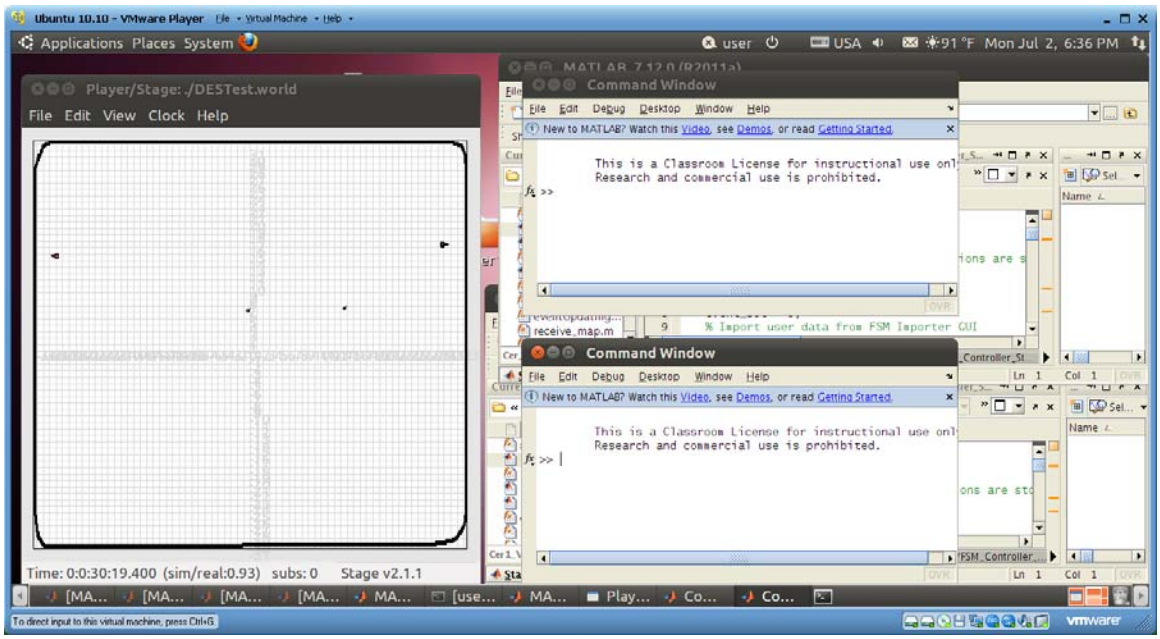

Figure 7-3 Screenshot of setup, showing arrangement of Player/Stage and MATLAB windows.

8. Start all six MATLAB files. If the robots move out of position when the VFH planner is started, use the MATLAB command `player('planner_set_cmd_pose', planner, desired_x, desired_y, 0)` to send the robots to the desired x- and y- coordinate locations.
9. A sample simulation result is shown in Figure 7-4.

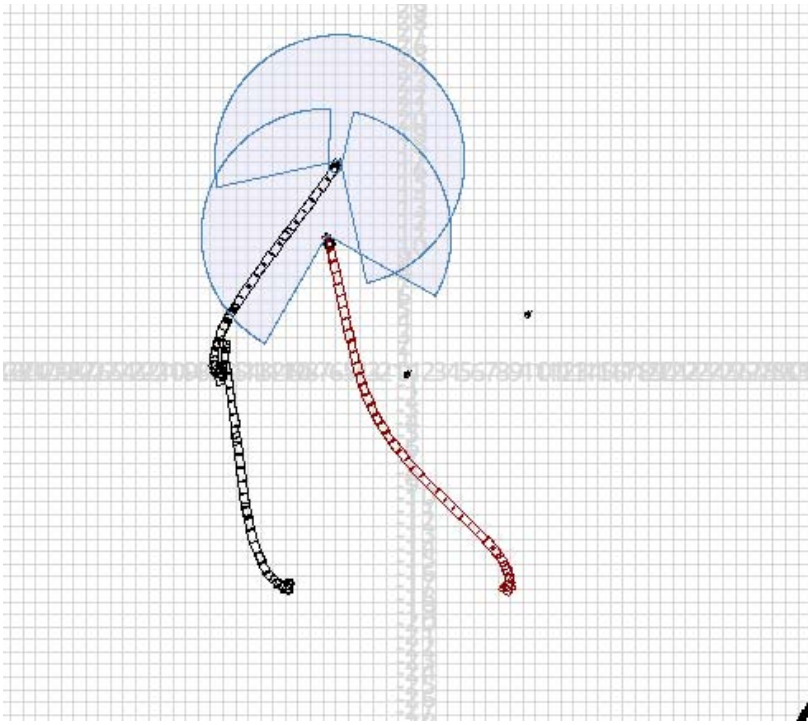

Figure 7-4 Player/Stage simulation results.

Appendix B: MATLAB Code for GUI

```
function varargout = FSM_Interpreter(varargin)
% FSM_INTERPRETER M-file for FSM_Interpreter.fig
%      FSM_INTERPRETER, by itself, creates a new FSM_INTERPRETER or
raises the existing
%      singleton*.
%
%      H = FSM_INTERPRETER returns the handle to a new FSM_INTERPRETER
or the handle to
%      the existing singleton*.
%
%      FSM_INTERPRETER('CALLBACK',hObject,eventData,handles,...) calls
the local
%      function named CALLBACK in FSM_INTERPRETER.M with the given
input arguments.
%
%      FSM_INTERPRETER('Property','Value',...) creates a new
FSM_INTERPRETER or raises the
%      existing singleton*.  Starting from the left, property value
pairs are
%      applied to the GUI before FSM_Interpreter_OpeningFcn gets
called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to FSM_Interpreter_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only
one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help FSM_Interpreter

% Last Modified by GUIDE v2.5 13-Aug-2012 20:22:51

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @FSM_Interpreter_OpeningFcn, ...
    'gui_OutputFcn',  @FSM_Interpreter_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
```

```
% End initialization code - DO NOT EDIT


% --- Executes just before FSM_Interpreter is made visible.
function FSM_Interpreter_OpeningFcn(hObject, eventdata, handles,
varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to FSM_Interpreter (see VARARGIN)

% Choose default command line output for FSM_Interpreter
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes FSM_Interpreter wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = FSM_Interpreter_OutputFcn(hObject, eventdata,
handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;



function txtFsmInputFn_Callback(hObject, eventdata, handles)
% hObject    handle to txtFsmInputFn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of txtFsmInputFn as
text
%        str2double(get(hObject,'String')) returns contents of
txtFsmInputFn as a double
fn = get(hObject, 'String');
if ~isempty(fn)
    updateTableData(handles);
end

% --- Executes during object creation, after setting all properties.
function txtFsmInputFn_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtFsmInputFn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in btnBrowseFsm.
function btnBrowseFsm_Callback(hObject, eventdata, handles)
% hObject    handle to btnBrowseFsm (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
fn = uigetfile({'*.fsm','FSM files';'*.*','All files'});
if ~isequal(fn,0)
    set(handles.txtFsmInputFn, 'String', fn)
end



function txtMapInputFn_Callback(hObject, eventdata, handles)
% hObject    handle to txtMapInputFn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of txtMapInputFn as
text
%        str2double(get(hObject,'String')) returns contents of
txtMapInputFn as a double
fn = get(hObject, 'String');
if ~isempty(fn)
    refreshGraph(handles);
end

% --- Executes during object creation, after setting all properties.
function txtMapInputFn_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtMapInputFn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in btnBrowseMap.
function btnBrowseMap_Callback(hObject, eventdata, handles)
% hObject    handle to btnBrowseMap (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
fn = uigetfile({'*.jpg;*.tif;*.png;*.gif;*.bmp','All Image Files';
'*.*','All Files'});
if ~isequal(fn,0)
    set(handles.txtMapInputFn, 'String', fn)
end

% --- Executes on button press in btnUpdateMap.
function btnUpdateMap_Callback(hObject, eventdata, handles)
% hObject    handle to btnUpdateMap (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% refresh graph data
refreshGraph(handles)


% --- Executes during object creation, after setting all properties.
function txtOutputFn_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtOutputFn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in btnWriteOutput.
function btnWriteOutput_Callback(hObject, eventdata, handles)
% hObject    handle to btnWriteOutput (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
fn = get(handles.txtFsmInputFn,'String');
[states, ~] = func_Read_FSM(fn);

fn = get(handles.txtOutputFn,'String');
tasks = get(handles.tblTasks,'Data');
regions = get(handles.tblRegions,'Data');
events = get(handles.tblEvents, 'Data');
bot = round(str2double(get(handles.txtBot,'String')));
initial_state = get(handles.txtInitialState,'String');
final_state = get(handles.txtFinalState,'String');
if ~isempty(fn)
   func_writeUserData(fn, bot, tasks, regions, events, states,
initial_state, final_state)
end

% --- Executes on button press in btnImportFsm.
function btnImportFsm_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to btnImportFsm (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
refreshGraph(handles)

set(handles.tblEvents,'Data',{})

updateTables(handles)


function txtxRange_Callback(hObject, eventdata, handles)
% hObject    handle to txtxRange (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of txtxRange as text
%        str2double(get(hObject,'String')) returns contents of
txtxRange as a double


% --- Executes during object creation, after setting all properties.
function txtxRange_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtxRange (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes during object creation, after setting all properties.
function txtyRange_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtyRange (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in btnSaveDataReg.
function btnSaveDataReg_Callback(hObject, eventdata, handles)
% hObject    handle to btnSaveDataReg (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% This saves the data to a MAT file. In future versions, this could
save to
% a text file or an excel file so that it can be edited outside of the
GUI.
%   bots = get(handles.tblBots,'Data');
   tasks = get(handles.tblTasks,'Data');
   regions = get(handles.tblRegions,'Data');
   events = get(handles.tblEvents,'Data');

   %uisave({'bots', 'tasks', 'regions', 'events'})
   uisave({'tasks', 'regions', 'events'})


% --- Executes on button press in btnLoadDataReg.
function btnLoadDataReg_Callback(hObject, eventdata, handles)
% hObject    handle to btnLoadDataReg (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename, pathname] = uigetfile('*.mat','MAT-files');
if isequal(filename,0) || isequal(pathname,0)
   %disp('User selected Cancel')
else
   %disp(['User selected',fullfile(pathname,filename)])
   s = load(filename);
 %  set(handles.tblBots,'Data', s.bots);
   set(handles.tblTasks,'Data', s.tasks);
   set(handles.tblRegions,'Data', s.regions);
   set(handles.tblEvents,'Data', s.events);

   updateTables(handles)
   refreshGraph(handles)
end



function txtFinalState_Callback(hObject, eventdata, handles)
% hObject    handle to txtFinalState (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of txtFinalState as
text
%        str2double(get(hObject,'String')) returns contents of
txtFinalState as a double


% --- Executes during object creation, after setting all properties.
function txtFinalState_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtFinalState (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called
```

```
% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes during object creation, after setting all properties.
function txtInitialState_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtInitialState (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes during object creation, after setting all properties.
function txtBot_CreateFcn(hObject, eventdata, handles)
% hObject    handle to txtBot (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes when entered data in editable cell(s) in tblTasks.
function tblTasks_CellEditCallback(hObject, eventdata, handles)
% hObject    handle to tblTasks (see GCBO)
% eventdata  structure with the following fields (see UITABLE)
%   Indices: row and column indices of the cell(s) edited
%   PreviousData: previous data for the cell(s) edited
%   EditData: string(s) entered by the user
%   NewData: EditData or its converted form set on the Data property.
Empty if Data was not changed
%   Error: error string when failed to convert EditData to appropriate
value for Data
% handles    structure with handles and user data (see GUIDATA)
if eventdata.Indices(2) > 2 && eventdata.Indices(2) < 4
    refreshGraph(handles)
end


% --- Executes when entered data in editable cell(s) in tblRegions.
```

```
function tblRegions_CellEditCallback(hObject, eventdata, handles)
% hObject    handle to tblRegions (see GCBO)
% eventdata  structure with the following fields (see UITABLE)
%   Indices: row and column indices of the cell(s) edited
%   PreviousData: previous data for the cell(s) edited
%   EditData: string(s) entered by the user
%   NewData: EditData or its converted form set on the Data property.
Empty if Data was not changed
%   Error: error string when failed to convert EditData to appropriate
value for Data
% handles    structure with handles and user data (see GUIDATA)
if eventdata.Indices(2) > 1 && eventdata.Indices(2) < 6
    refreshGraph(handles)
end

function refreshGraph(handles)
% refresh axes with current image data and region data

% % Display image to scale first
% fn = get(handles.txtMapInputFn,'String');
% x_bounds = get(handles.txtxRange, 'String');
% y_bounds = get(handles.txtyRange, 'String');
% if ~isempty(fn)
%     %read image data from file
%     img = imread(fn);
%     % scale image if bounds are specified
%     if ~isempty(x_bounds) && ~isempty(y_bounds)
%         % get individual values and convert from string to double
%         x = str2double(regexp(x_bounds,'[,]','split'));
%         y = str2double(regexp(y_bounds,'[,]','split'));
%     else
%         x = [0 size(img,1)];
%         y = [0 size(img,2)];
%     end
%     %plot image
%     image(x,y,img)
%     hold on;
% end

% Display tasks on top of image map
tasks = get(handles.tblTasks,'Data');

tName = 1;
tx = 2;
ty = 3;


for i =1:size(tasks,1)
    x = tasks{i,tx};
    y = tasks{i,ty};
    plot(x,y,'*');
    hold on
end

% Display regions on top of image map
```

```
regions = get(handles.tblRegions,'Data');

rName = 1;
rx1 = 2;
ry1 = 3;
rx2 = 4;
ry2 = 5;

for i =1:size(regions,1)
    try
        [x,y] = plotRectangle([regions{i,rx1} regions{i,rx2}], ...
            [regions{i,ry1} regions{i,ry2}]);
        hold all
        plot(x,y,':');
    end
end
hold off


function updateTables(handles)
fn = get(handles.txtFsmInputFn,'String');
if ~isempty(fn)
    [states, event_names] = func_Read_FSM(fn);

events = get(handles.tblEvents,'Data');

if isempty(events)
    % if logic to event names, such as 'a' = bot 1, 's' = task, 'e' =
region, enter code here
    for i=1:length(event_names)
        events{i,1} = event_names{i};

        % Determine bot
        if strcmp(event_names{i}(1),'a')
            events{i,2} = 1; % bot is 1
        else
            events{i,2} = 2; % bot is 2
        end

        %Determine controllable or uncontrollable event
        for n=1:size((states),1)
            for m=1:size((states{n,2}),1)
                if strcmp(states{n,2}{m,1},event_names{i})
                    if strcmp(states{n,2}{m,3},'c')
                        events{i,3} = 'Task'; % controllable event:
Task
                        break
                    else
                        events{i,3} = 'Region'; % uncontrollable event:
Region
                        break
                    end
                end
            end
        end
```

```
        events{i,4} = ''; % type_name which must be selected from
defined tasks & regions
    end
end

    % Set events table
    set(handles.tblEvents,'Data',events);

    % get task_names
    tasks = get(handles.tblTasks,'Data')
    task_names = tasks(:,1);

    % get region_names
    regions = get(handles.tblRegions,'Data')
    region_names = regions(:,1);

    % combine task_names and region_names into one list
    type_names = [task_names; region_names]';

    % update task_region_names in events table
    set(handles.tblEvents,'ColumnFormat',{'char','numeric',{'Task',
'Region'},type_names})

    % update initial state with default first entry of FSM
    set(handles.txtInitialState,'String',states{1,1})

    % update marked state with first marked state in FSM
    for n=1:length(states)
        if states{n,3}
            set(handles.txtFinalState,'String',states{n,1})
            break
        end
    end

end



% --- Executes on button press in btnAddTask.
function btnAddTask_Callback(hObject, eventdata, handles)
% hObject    handle to btnAddTask (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
tasks_id = get(handles.tblTasks,'UserData');
if ~isempty(tasks_id)
    tasks = get(handles.tblTasks,'Data');

    if tasks_id(1) == size(tasks,1)
        % insert new row at end of tasks
    tasks{tasks_id(1)+1,1} = 'tsk';
    tasks{tasks_id(1)+1,2} = 0;
    tasks{tasks_id(1)+1,3} = 0;
    tasks{tasks_id(1)+1,4} = 'gotogoal';
    tasks{tasks_id(1)+1,5} = 'start';
```

```
    else
        % insert new row in middle of tasks
    for i = (size(tasks,1)+1):-1:tasks_id(1)+1
        for j=1:size(tasks,2)
            tasks{i,j} = tasks{i-1,j};
        end
    end

    tasks{tasks_id(1)+1,1} = 'tsk';
    tasks{tasks_id(1)+1,2} = 0;
    tasks{tasks_id(1)+1,3} = 0;
    tasks{tasks_id(1)+1,4} = 'gotogoal';
    tasks{tasks_id(1)+1,5} = 'start';
    end
    set(handles.tblTasks,'Data',tasks)

end

% --- Executes on button press in btnAddRegion.
function btnAddRegion_Callback(hObject, eventdata, handles)
% hObject    handle to btnAddRegion (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
regions_id = get(handles.tblRegions,'UserData');
if ~isempty(regions_id)
    regions = get(handles.tblRegions,'Data');

    if regions_id(1) == size(regions,1)
        % insert new row at end of regions
    regions{regions_id(1)+1,1} = 'reg';
    regions{regions_id(1)+1,2} = 0;
    regions{regions_id(1)+1,3} = 0;
    regions{regions_id(1)+1,4} = 0;
    regions{regions_id(1)+1,5} = 0;
    else
        % insert new row in middle of regions
    for i = (size(regions,1)+1):-1:regions_id(1)+1
        for j=1:size(regions,2)
            regions{i,j} = regions{i-1,j};
        end
    end

    regions{regions_id(1)+1,1} = 'reg';
    regions{regions_id(1)+1,2} = 0;
    regions{regions_id(1)+1,3} = 0;
    regions{regions_id(1)+1,4} = 0;
    regions{regions_id(1)+1,5} = 0;
    end
    set(handles.tblRegions,'Data',regions)

end

% --- Executes on button press in btnDeleteTask.
function btnDeleteTask_Callback(hObject, eventdata, handles)
% hObject    handle to btnDeleteTask (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
tasks_id = get(handles.tblTasks,'UserData');
if ~isempty(tasks_id)
    tasks = get(handles.tblTasks,'Data');

    if tasks_id(1) == size(tasks,1)
        % delete last row of tasks
    tasks = tasks(1:tasks_id(1)-1,:);
    else
        % insert new row in middle of tasks
    for i = tasks_id(1):size(tasks,1)-1
        for j=1:size(tasks,2)
            tasks{i,j} = tasks{i+1,j};
        end
    end

    tasks = tasks(1:size(tasks,1)-1,:);
    end
    set(handles.tblTasks,'Data',tasks)

end


% --- Executes on button press in btnDeleteRegion.
function btnDeleteRegion_Callback(hObject, eventdata, handles)
% hObject    handle to btnDeleteRegion (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
regions_id = get(handles.tblRegions,'UserData');
if ~isempty(regions_id)
    regions = get(handles.tblRegions,'Data');

    if regions_id(1) == size(regions,1)
        % delete last row of regions
    regions = regions(1:regions_id(1)-1,:);
    else
        % insert new row in middle of regions
    for i = regions_id(1):size(regions,1)-1
        for j=1:size(regions,2)
            regions{i,j} = regions{i+1,j};
        end
    end

    regions = regions(1:size(regions,1)-1,:);
    end
    set(handles.tblRegions,'Data',regions)

end

% --- Executes when selected cell(s) is changed in tblTasks.
function tblTasks_CellSelectionCallback(hObject, eventdata, handles)
% hObject    handle to tblTasks (see GCBO)
% eventdata  structure with the following fields (see UITABLE)
%   Indices: row and column indices of the cell(s) currently selecteds
```

```
% handles    structure with handles and user data (see GUIDATA)
set(hObject,'UserData',eventdata.Indices)


% --- Executes when selected cell(s) is changed in tblRegions.
function tblRegions_CellSelectionCallback(hObject, eventdata, handles)
global regionIndices
% hObject    handle to tblRegions (see GCBO)
% eventdata  structure with the following fields (see UITABLE)
%   Indices: row and column indices of the cell(s) currently selecteds
% handles    structure with handles and user data (see GUIDATA)
set(hObject,'UserData',eventdata.Indices)
regionIndices = eventdata.Indices;


% --- Executes on mouse press over axes background.
function axisMap_ButtonDownFcn(hObject, eventdata, handles)
global firstpoint
global regionIndices
% hObject    handle to axisMap (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
if ~isempty(regionIndices)
    if ~isempty(firstpoint)
        % Set region interactively
        secondpoint = get(handles.axisMap,'CurrentPoint');

        regions = get(handles.tblRegions,'Data');

        % Set x1
        regions{regionIndices(1),2} =
min(firstpoint(1),secondpoint(1));
        % Set x2
        regions{regionIndices(1),4} =
max(firstpoint(1),secondpoint(1));
        % Set y1
        regions{regionIndices(1),3} =
min(firstpoint(1,2),secondpoint(1,2));
        % Set y2
        regions{regionIndices(1),5} =
max(firstpoint(1,2),secondpoint(1,2));

        set(handles.tblRegions,'Data',regions)

        refreshGraph(handles)
        % clear first point
        firstpoint = [];
        regionIndices = [];
    else
        % save first point
        firstpoint = get(handles.axisMap,'CurrentPoint');
    end
end
```

Appendix C: FSM File

```
72

s1   0   2
a5e s2   uc  o
b8e s3   uc  o

s130:110 0   1
a4s s122:140 c   o

s126 0   1
a6e s136:113 uc  o

s124 0   1
a2f s134:146:143 uc  o

s123 0   1
b8e s132 uc  o

s121 0   2
a6e s131 uc  o
b2f s125:133:145 uc  o

s120 0   2
a4f s132 uc  o
b7e s131 uc  o

s59  0   1
b8e s74  uc  o

s56  0   1
a7e s71  uc  o

s55  0   1
a3f s70  uc  o

s50  0   2
b1s s60  c   o
a4f s64  uc  o

s115 0   1
a2s s126 c   o

s114 0   1
a4s s125:133:145 c   o

s112 0   1
b2s s123 c   o

s111 0   2
b8e s120 uc  o
a4f s123 uc  o

s134:146:143 s1   0
```

```
s49  0   2
a4s s62:105  c   o
a2s s63  c   o

s48  0   2
b8e s61  uc  o
a3f s59  uc  o

s46  0   1
b2s s59  c   o

s42  0   1
a6e s56  uc  o

s41  0   1
a6e s55  uc  o

s109 0   2
a5e s121 uc  o
b2f s122:140 uc  o

s108 0   2
b7e s121 uc  o
a6e s120 uc  o

s106 0   1
a6e s119:96  uc  o

s37  0   1
a6e s50  uc  o

s36  0   1
a1f s49  uc  o

s35  0   2
a3f s46  uc  o
b2s s48  c   o

s99  0   1
a1f s115 uc  o

s33  0   1
b1f s46  uc  o

s98  0   1
a2f s114 uc  o

s97  0   1
a2s s136:113 c   o

s31  0   2
a3s s41  c   o
a2s s42  c   o

s95  0   1
b1f s112 uc  o
```

```
s94  0   2
b2s s111 c   o
a4f s112 uc  o

s93  0   2
a4s s109 c   o
b2f s130:110 uc  o

s92  0   2
b7e s109 uc  o
a5e s108 uc  o

s90  0   1
a3s s106 c   o

s135:142 0   1
a4f s134:146:143 uc  o

s119:96  0   1
a3f s130:110 uc  o

s27  0   1
a5e s37  uc  o

s26  0   1
a5e s36  uc  o

s25  0   2
b1f s35  uc  o
a3f s33  uc  o

s24  0   1
b5e s33  uc  o

s88  0   1
a5e s62:105  uc  o

s125:133:145 0   1
a6e s135:142 uc  o

s80  0   1
a5e s99  uc  o

s9   0   1
a6e s14  uc  o

s4   0   2
a1s s10  c   o
a3s s9   c   o

s3   0   1
a5e s4   uc  o

s2   0   1
b8e s4   uc  o

s19  0   3
```

```
a4s s27  c   o
b1s s24  c   o
a1s s26  c   o

s18  0   2
a3f s24  uc  o
b5e s25  uc  o

s141 0   1
b2f s134:146:143 uc  o

s14  0   2
b1s s18  c   o
a3f s19  uc  o

s78  0   1
a4f s97  uc  o

s77  0   2
a3f s93  uc  o
b2f s119:96  uc  o

s76  0   1
b5e s95  uc  o

s10  0   1
a1f s31  uc  o

s75  0   2
a4f s95  uc  o
b1f s94  uc  o

s74  0   2
a4s s92  c   o
b7e s93  uc  o

s71  0   1
a2f s90  uc  o

s70  0   2
a4s s88  c   o
a2s s89:79   c   o

s136:113 0   1
a7e s124 uc  o

s89:79   0   1
a7e s98  uc  o

s132 0   1
b7e s141 uc  o

s131 0   2
b2f s135:142 uc  o
a4f s141 uc  o

s64  0   2
```

```
b1s s76  c   o
a1s s80  c   o

s63  0   1
a6e s89:79   uc  o

s62:105  0   1
a6e s78  uc  o

s61  0   2
b7e s77  uc  o
a3f s74  uc  o

s60  0   2
b5e s75  uc  o
a4f s76  uc  o

s122:140 0   1
a5e s125:133:145 uc  o
```

Appendix D: MATLAB Code for Importing FSM

To reference a field of the `states` cell array, curly brackets `{}` are used in MATLAB. For example, to retrieve the first state name, the MATLAB syntax would be `states{1,1}`. To retrieve the first event name, use a double pair of curly brackets, one for the first cell array and another for the events nested cell array. The syntax for the first event name looks like this: `states{1,2}{1,1}`. Similarly to reference the next state name, the syntax is as follows:

`states{1,2}{1,2}`.

```
% ***************** func_Read_FSM.m ***************
% This function parses an FSM file, outputting a nx3 cell array
% of states, where n is the number of states, found on the first
% line of the *.fsm file. It also outputs a 1xn cell array of
% unique event names.

function [states, eventNames] = func_Read_FSM(fn)

% Open FSM file
fid = fopen(fn);

% Read number of states in file
[num_blocks, pos] = textscan(fid,'%u');

% Read first line of file
tline = fgetl(fid);

% Initialize loop variables
x=1; j=1; states = cell(num_blocks{1},2); eventList = {};

% Loop through file, collecting state and event data into
matrices
while ischar(tline)
    if ~isempty(tline)
        % Parse first line of block
        c = textscan(tline, '%s %*u %u');
        % Enter state name
        states{j,1} = c{1,1}{1,1};
        % Create new events matrix, size is last integer of line
        events=cell(c{1,2},5);
        % Loop through each event line
        for i = 1:c{1,2}
            if ~feof(fid) % if not end of file
                % Read an event line of data
                tline = fgetl(fid);
                % Parse the data
```

```
                d = textscan(tline, '%s %s %s %s');

                events{x,1} = d{1,1}{1,1}; % event name
                events{x,2} = d{1,2}{1,1}; % next state
                events{x,3} = d{1,3}{1,1}; % is controllable?
                events{x,4} = d{1,4}{1,1}; % is observable?

             % Add this event to the list of events
                eventList = [eventList events{x,1}];
                x=x+1;
            end
        end
        % Restart loop variables and input event matrix into
states matrix
        x = 1; states{j,2} = events; j=j+1;
    end
    % Get next line of text
    tline = fgetl(fid);
end
% Close FSM file
fclose(fid);

% Output list of event names
eventNames = unique(eventList);
```

Appendix E: Sample User Data File

```
******************** UserData_Control2_thesis.m ***************
% 1) ********** User Input **********

% botNames = {'Cer', 'Cer1'};
% botIP = {'6665', '6666');
% botIndex = [0, 2];
% bots = {'name', 'IP', Index}
% bots = {'Cer', '6665', 0;
%         'Cer1', '6666', 2};

initial_state = 's1';
final_state = 's134:146:143';

% tasks = {'name', x1, y1, 'function', 'from_task_name'}
tasks = { 'tsk1',  -2,  -5,  'gotogoal';
    'tsk2',  7,  8,  'gotogoal';
    'tsk3',  -5,  3,  'gotogoal';
    'tsk4',  -13,  8,  'gotogoal' };

% regions = {'name', x1, y1, x2, y2}
regions = { 'reg5',  -35,  -35,  0,  0;
    'reg6',  -35,  0,  0,  35;
    'reg8',  0,  -35,  35,  0;
    'reg7',  0,  0,  35,  35;
    'regtsk1',  -1,  -4,  -3,  -6;
    'regtsk2',  6,  7,  8,  9;
    'regtsk3',  -4,  2,  -6,  4;
    'regtsk4',  -14,  9,  -12,  7 };

% events = {'name', bot, 'type', 'type_name'}
events = { 'a1s',  1,  'Task',  'tsk1';
    'a2s',  1,  'Task',  'tsk2';
    'a3s',  1,  'Task',  'tsk3';
    'a4s',  1,  'Task',  'tsk4';
    'b1s',  2,  'Task',  'tsk1';
    'b2s',  2,  'Task',  'tsk2';
    'b3s',  2,  'Task',  'tsk3';
    'b4s',  2,  'Task',  'tsk4';
    'a5e',  1,  'Region',  'reg5';
    'a6e',  1,  'Region',  'reg6';
    'a7e',  1,  'Region',  'reg7';
    'a8e',  1,  'Region',  'reg8';
    'b5e',  2,  'Region',  'reg5';
    'b6e',  2,  'Region',  'reg6';
    'b7e',  2,  'Region',  'reg7';
    'b8e',  2,  'Region',  'reg8';
    'a1f',  1,  'Region',  'regtsk1';
    'a2f',  1,  'Region',  'regtsk2';
    'a3f',  1,  'Region',  'regtsk3';
    'a4f',  1,  'Region',  'regtsk4';
    'b1f',  2,  'Region',  'regtsk1';
    'b2f',  2,  'Region',  'regtsk2';
    'b3f',  2,  'Region',  'regtsk3';
```

```
    'b4f',  2,  'Region',  'regtsk4' };


% states = {'name', {'event_name', 'state_name', 'controllable',
'observable', 'cost';
%                    'event_name', 'state_name', 'controllable',
'observable', 'cost'}, 'marked'}
states = { 's1',  { 'b8e',  's3',  'uc',  'o';
                    'a5e',  's2',  'uc',  'o' },  0;
    's130:110',  { 'a4s',  's122:140',  'c',  'o' },  0;
    's126',  { 'a6e',  's136:113',  'uc',  'o' },  0;
    's124',  { 'a2f',  's134:146:143',  'uc',  'o' },  0;
    's123',  { 'b8e',  's132',  'uc',  'o' },  0;
    's121',  { 'a6e',  's131',  'uc',  'o';
               'b2f',  's125:133:145',  'uc',  'o' },  0;
    's120',  { 'b7e',  's131',  'uc',  'o';
               'a4f',  's132',  'uc',  'o' },  0;
    's59',  { 'b8e',  's74',  'uc',  'o' },  0;
    's56',  { 'a7e',  's71',  'uc',  'o' },  0;
    's55',  { 'a3f',  's70',  'uc',  'o' },  0;
    's50',  { 'a4f',  's64',  'uc',  'o';
              'b1s',  's60',  'c',  'o' },  0;
    's115',  { 'a2s',  's126',  'c',  'o' },  0;
    's114',  { 'a4s',  's125:133:145',  'c',  'o' },  0;
    's112',  { 'b2s',  's123',  'c',  'o' },  0;
    's111',  { 'a4f',  's123',  'uc',  'o';
               'b8e',  's120',  'uc',  'o' },  0;
    's49',  { 'a2s',  's63',  'c',  'o';
    'a4s',  's62:105',  'c',  'o' },  0;
    's134:146:143',  {},  [];
    's48',  { 'b8e',  's61',  'uc',  'o';
              'a3f',  's59',  'uc',  'o' },  0;
    's46',  { 'b2s',  's59',  'c',  'o' },  0;
    's42',  { 'a6e',  's56',  'uc',  'o' },  0;
    's41',  { 'a6e',  's55',  'uc',  'o' },  0;
    's109',  { 'a5e',  's121',  'uc',  'o';
               'b2f',  's122:140',  'uc',  'o' },  0;
    's108',  { 'a6e',  's120',  'uc',  'o';
               'b7e',  's121',  'uc',  'o' },  0;
    's106',  { 'a6e',  's119:96',  'uc',  'o' },  0;
    's37',  { 'a6e',  's50',  'uc',  'o' },  0;
    's36',  { 'a1f',  's49',  'uc',  'o' },  0;
    's35',  { 'a3f',  's46',  'uc',  'o';
              'b2s',  's48',  'c',  'o' },  0;
    's99',  { 'a1f',  's115',  'uc',  'o' },  0;
    's98',  { 'a2f',  's114',  'uc',  'o' },  0;
    's33',  { 'b1f',  's46',  'uc',  'o' },  0;
    's97',  { 'a2s',  's136:113',  'c',  'o' },  0;
    's31',  { 'a2s',  's42',  'c',  'o';
              'a3s',  's41',  'c',  'o' },  0;
    's95',  { 'b1f',  's112',  'uc',  'o' },  0;
    's94',  { 'b2s',  's111',  'c',  'o';
              'a4f',  's112',  'uc',  'o' },  0;
    's93',  { 'a4s',  's109',  'c',  'o';
              'b2f',  's130:110',  'uc',  'o' },  0;
    's92',  { 'b7e',  's109',  'uc',  'o';
```

```
'a5e',  's108',  'uc',  'o' },  0;
's90',  { 'a3s',  's106',  'c',  'o' },  0;
's135:142',  { 'a4f',  's134:146:143',  'uc',  'o' },  0;
's27',  { 'a6e',  's50',  'uc',  'o' },  0;
's119:96',  { 'a3f',  's130:110',  'uc',  'o' },  0;
's26',  { 'a5e',  's36',  'uc',  'o' },  0;
's25',  { 'a3f',  's33',  'uc',  'o';
'b1f',  's35',  'uc',  'o' },  0;
's24',  { 'b5e',  's33',  'uc',  'o' },  0;
's88',  { 'a5e',  's62:105',  'uc',  'o' },  0;
's22',  { 'a1f',  's31',  'uc',  'o' },  0;
's125:133:145',  { 'a6e',  's135:142',  'uc',  'o' },  0;
's80',  { 'a5e',  's99',  'uc',  'o' },  0;
's9',  { 'a6e',  's14',  'uc',  'o' },  0;
's4',  { 'a1s',  's10',  'c',  'o';
'a3s',  's9',  'c',  'o' },  0;
's3',  { 'a5e',  's4',  'uc',  'o' },  0;
's2',  { 'b8e',  's4',  'uc',  'o' },  0;
's19',  { 'a1s',  's26',  'c',  'o';
          'b1s',  's24',  'c',  'o';
          'a4s',  's27',  'c',  'o' },  0;
's18',  { 'b5e',  's25',  'uc',  'o';
'a3f',  's24',  'uc',  'o' },  0;
's16',  { 'a1f',  's31',  'uc',  'o' },  0;
's141',  { 'b2f',  's134:146:143',  'uc',  'o' },  0;
's14',  { 'a3f',  's19',  'uc',  'o';
'b1s',  's18',  'c',  'o' },  0;
's78',  { 'a4f',  's97',  'uc',  'o' },  0;
's77',  { 'a3f',  's93',  'uc',  'o';
'b2f',  's119:96',  'uc',  'o' },  0;
's76',  { 'b5e',  's95',  'uc',  'o' },  0;
's10',  { 'a6e',  's16',  'uc',  'o' },  0;
's75',  { 'a4f',  's95',  'uc',  'o';
'b1f',  's94',  'uc',  'o' },  0;
's74',  { 'a4s',  's92',  'c',  'o';
'b7e',  's93',  'uc',  'o' },  0;
's71',  { 'a2f',  's90',  'uc',  'o' },  0;
's70',  { 'a2s',  's89:79',  'c',  'o';
'a4s',  's88',  'c',  'o' },  0;
's136:113',  { 'a7e',  's124',  'uc',  'o' },  0;
's89:79',  { 'a7e',  's98',  'uc',  'o' },  0;
's132',  { 'b7e',  's141',  'uc',  'o' },  0;
's131',  { 'b2f',  's135:142',  'uc',  'o';
'a4f',  's141',  'uc',  'o' },  0;
's64',  { 'a1s',  's80',  'c',  'o';
'b1s',  's76',  'c',  'o' },  0;
's62:105',  { 'a6e',  's78',  'uc',  'o' },  0;
's63',  { 'a6e',  's89:79',  'uc',  'o' },  0;
's61',  { 'b7e',  's77',  'uc',  'o';
'a3f',  's74',  'uc',  'o' },  0;
's60',  { 'a4f',  's76',  'uc',  'o';
'b5e',  's75',  'uc',  'o' },  0;
's122:140',  { 'a5e',  's125:133:145',  'uc',  'o' },  0 };
```

Appendix F: MATLAB Code for Main Control File

```
**************** Cer_VFSM_Controller_Start.m ****************
% This file is used to control one robot in the UDM Lab
% 1) ********** User Input **********
clear all

% Add path to function locations (func_GoalPlan, func_Dijkstra, etc)
addpath '~/Desktop/Shared'

% The robot to control (either 1 or 2)
bot = 1;

% User Data file name (from FSM Interpreter GUI)
UserData_Control2_thesis
% UserData_A1A2B3B4FSM

% UserData must follow the following format:
% bots = {'name', 'IP', Index}
% tasks = {'name', x1, y1, 'function', 'from_task_name'}
% regions = {'name', x1, y1, x2, y2}
% events = {'name', bot, 'type', 'type_name'}
% initial_state = 'state_name';
% final_state = 'state_name';
% states = {'name', {'event_name', 'state_name', 'isControllable:
c_uc', 'cost';
%                    'event_name', 'state_name', 'isControllable:
c_uc', 'cost'}, 'isDone: 0_1'}




% 2) ********** Player/Stage Initialization *********
addpath '~/IGVC2010/MEXSupport/matlab_path/'

global clientCer1
global pos2dCer
global pos2dCer1
global laser
global planner
global image_opaque
global goal_opaque
global map_relay
global map
global Bread_crumbs
global time_plan

% Now if the variables exist, clean them up properly
try, player('destroy', laser); end;
try, player('destroy', pos2dCer); end;
try, player('client_disconnect', clientCer); end
try, player('destroy', clientCer); end;
try, player('destroy', pos2dCer1); end;
try, player('client_disconnect', clientCer1); end
```

```
try, player('destroy', clientCer1); end;

clientCer = player('client_create', 'localhost',6665);
player('client_connect', clientCer);

pos2dCer = player('position2d_create', clientCer, 0);
player('subscribe', pos2dCer);

clientCer1 = player('client_create', 'localhost',6666);
player('client_connect', clientCer1);

pos2dCer1 = player('position2d_create', clientCer1, 2);
player('subscribe', pos2dCer1);

laser = player('laser_create', clientCer, 0);
player('subscribe', laser);

planner = player('planner_create', clientCer, 0);
player('subscribe', planner);

image_opaque = player('opaque_create', clientCer, 9);
player('subscribe',image_opaque);

goal_opaque = player('opaque_create', clientCer, 3);
player('subscribe', goal_opaque);

map_relay = player('opaque_create', clientCer, 6);
player('subscribe',map_relay);

image_info = struct ('map',[],'BC', [], 'pos', [],...
    'time_plan', [],'distance', [], 'path_done', [], 'goal', []);


% initialize the time_plan
% the required time for planning and replanning
time_plan = 0;
% the array of the current bread crumbs
Bread_crumbs = [];
t = timer('StartDelay',2, 'Period', 1.0, 'ExecutionMode','fixedRate');
t.TimerFcn = {'receive_map', map_relay, image_opaque, image_info,
time_plan, goal_opaque};
%start(t)
%all_while=0;


% 3) ********** Static Optimization **********
% Send robot to initial positions
%player('planner_set_cmd_pose',planner,-9,-19,0)

% Get initial positions
player('client_read', clientCer);
player('client_read', clientCer1);

start_loc = [pos2dCer.px pos2dCer.py;
```

```
    pos2dCer1.px pos2dCer1.py];
% Calculate costs from/to each state
tic
[A,C,event_bots] = func_calculateCosts(states, events, tasks,
start_loc);

% Set current state index based on initial_state name
cur_state = find(strcmp(initial_state,states(:,1))); % *** this must be
an integer, not a string (1)
% Set end state index based on final_state name
end_state = find(strcmp(final_state,states(:,1)));   % *** this must be
an integer, not a string (3)

% Run optimization algorithm to find path through FSM file from
cur_state to end_state
[totalCost, path] = func_Dijkstra(A, C, cur_state, end_state,
event_bots);
toc

% Is there a way to share cost matrix between multiple matlab windows
so no need for multiple calculations of the same thing?

% Print path to MATLAB command window
fprintf('\nRobot %u: Path planned from ''%s'' to ''%s'': \nPath =
[',bot,states{cur_state,1},states{end_state,1})
fprintf('%s ',states{path,1})
%fprintf('%u ',path)
fprintf(']\nTotal Cost: %5.2f \n\n',totalCost)

% 4) ********** Virtual FSM with Dynamic Optimization **********
% Initialize goal to nothing
goal = [];
% Initialize cur_task to empty
cur_task = '';
% Initialize event_bot to not 1 or 2
event_bot = 0;
% Initialize path counter (pc) to be first point of path
pc = 1;
% Initialize current state (cur_state) to be first state on path
cur_state = path(pc);
% Display current state
fprintf('%u. State %u: %s \n',pc,cur_state,states{cur_state,1})
% now, I could change the task positions randomly by a certain amount

% This is for a finite state machine with an end state defined
while cur_state~=end_state
    % Update map
    while(isempty(map_relay.data))
        player('client_read', clientCer);
        player('client_read', clientCer1);
    end
    map = player('matlab_unpack', map_relay.data);


    % 1. Determine if controllable events or uncontrollable events
```

```
    NCE = 0; % Reset NCE (Number of Controllable Events) to zero
    % Loop through events in current state, checking for controllable
events
    for i = 1:size(states{cur_state,2},1)
        if strcmp(states{cur_state,2}{i,3}, 'c') % controllable event
            % Set event index to current index
            event_index = i;
            % Increment NCE counter
            NCE = NCE +1;
        end
    end

    % 2. Determine whether to change current goal based on NCE
    if NCE >= 1  % there are controllable events
        if NCE > 1 % more than one controllable event
            % Choose one controllable event based on next state_name in
path

            % Re-optimize path if costs have changed
            % Get initial positions
            player('client_read', clientCer);
            player('client_read', clientCer1);

            start_loc = [pos2dCer.px pos2dCer.py; pos2dCer1.px
pos2dCer1.py];

            tic
            [A,C,event_bots] = func_calculateCosts(states, events,
tasks, start_loc);


            %cur_state =
find(strcmp(states{cur_state,2}{event_index,2},states(:,1))); % ***
this must be an integer, not a string (1)
            end_state = find(strcmp(final_state,states(:,1)));   % ***
this must be an integer, not a string (3)

            % Run optimization algorithm to find path through FSM file
            [totalCost, path] = func_Dijkstra(A, C, cur_state,
end_state, event_bots);
            toc
            % Print path to MATLAB command window
            fprintf('\nRobot %u: Path planned from ''%s'' to ''%s'':
\nPath = [',bot,states{cur_state,1},states{end_state,1})
            fprintf('%s ',states{path,1})
            %fprintf('%u ',path)
            fprintf(']\nTotal Cost: %5.2f \n\n',totalCost)

            pc = 1;

            next_state_name = states(path(pc+1),1);
            event_index =
strcmp(next_state_name,states{cur_state,2}(:,2));
        end
```

```
        % Find event/task name
        event_name = states{cur_state,2}{event_index,1};
        task_name = events(strcmp(event_name, events(:,1)), 4);
        fprintf('   Event Chosen: %s (%s)\n',event_name,task_name{1})

        % Determine which bot is to do this event
        event_bot = events{strcmp(event_name, events(:,1)), 2};
        % Change goal if the controllable event is for this bot
        if bot == event_bot
            % Get goal from event/task name
            x = tasks{strcmp(task_name,tasks(:,1)),2};
            y = tasks{strcmp(task_name,tasks(:,1)),3};
            goal = [x y];
            fprintf('   GoToGoal: %3.2f,%3.2f\n',goal(1),goal(2))
            cur_task = task_name;
        end
        % Update path counter *** assume "start" event for each task
        % Check that next state_name is the next state on the path
        if strcmp(states{cur_state,2}{event_index,2},
states{path(pc+1),1})
            % The next state matches the planned next state, update pc
and continue path
            pc = pc+1;
            cur_state = path(pc);
            fprintf('%u. State %u: %s
\n',pc,cur_state,states{cur_state,1})
        end
    end

    % 3. Go to goal OR wait, until next event is detected
    % D*planner function
    % event_name = func_GoalPlan(goal, pos2dCer, pos2dCer1, clientCer,
clientCer1, map, planner, regions, events, bot);
    if bot == event_bot
        if bot == 1
            %event_names =
func_GoGoal(goal,clientCer,pos2dCer,pos2dCer1,laserCer,bot,events,regio
ns);
            event_names = func_GoalPlan(goal, pos2dCer, pos2dCer1,
clientCer, clientCer1, map, planner,regions,events,bot);
        elseif bot == 2
            %event_names =
func_GoGoal(goal,clientCer,pos2dCer,pos2dCer1,laserCer1,bot,events,regi
ons);
            event_names = func_GoalPlan(goal, pos2dCer1, pos2dCer,
clientCer1, clientCer, map, planner,regions,events,bot);
        end
        all_events = event_names;
    else
        if strcmp(cur_task, '') % no current task
            % wait for/detect next event

            player('client_read', clientCer);
            player('client_read', clientCer1);
```

```
            event_names_1 = func_detectRegionEvents(1,pos2dCer,
regions, events);
            event_names_2 =
func_detectRegionEvents(2,pos2dCer1,regions, events);

            all_events = [event_names_1 event_names_2];
        else % current task is not complete
            if bot == 1
                %event_names =
func_GoGoal(goal,clientCer,pos2dCer,pos2dCer1,laserCer,bot,events,regio
ns);
                event_names = func_GoalPlan(goal, pos2dCer, pos2dCer1,
clientCer, clientCer1, map, planner,regions,events,bot);

            elseif bot == 2
                %event_names =
func_GoGoal(goal,clientCer,pos2dCer,pos2dCer1,laserCer1,bot,events,regi
ons);
                event_names = func_GoalPlan(goal, pos2dCer1, pos2dCer,
clientCer1, clientCer, map, planner,regions,events,bot);
            end
            all_events = event_names;
        end
    end
    % event_index = strcmp(event_name,states{cur_state,2}(:,1));

    % Validate next state/event
    next_state_name = states(path(pc+1),1);
    next_state_index =
strcmp(next_state_name,states{cur_state,2}(:,2));
    next_event_name = states{cur_state,2}{next_state_index,1};

    % 4. Check if any all_events is the next state on the path
    if sum(strcmp(next_event_name, all_events(:)))>0
        fprintf('   Detected Event: ')
        fprintf('%s ',next_event_name)
        fprintf('\n')

        % The next state matches the planned next state, update pc and
continue path
        pc = pc+1;

        % Update state
        cur_state = path(pc);

        % Display current state
        fprintf('%u. State %u: %s \n',pc,cur_state,states{cur_state,1})

        % if the event is a task completion, update cur_task to ''
        if strcmp(next_event_name(3),'f')
            if strcmp(next_event_name(1),'a') % Turn in circle
                tic
                while toc <= 1
                    player('position2d_set_cmd_vel',pos2dCer, 0, 0,
pi/2);
```

```
                    pause(0.5)
                end
                player('position2d_set_cmd_vel',pos2dCer, 0, 0, 0);
                cur_task='';
            elseif strcmp(next_event_name(1),'b') % Turn in circle
                tic
                while toc <= 1
                    player('position2d_set_cmd_vel',pos2dCer1, 0, 0,
pi/2);
                    pause(0.5)
                end
                player('position2d_set_cmd_vel',pos2dCer1, 0, 0, 0);
                cur_task='';
            end

        end
    end

end    % End of Virtual FSM w/Optimization

% Display current state
fprintf('Mission complete!\n\n',pc,cur_state,states{cur_state,1})
```

Appendix G: MATLAB Code for Detect Region Events

********** **func_detectRegionEvents.m** **********

```
function event_names = func_detectRegionEvents(bot, pos2d, regions,
events)
% FUNC_DETECTREGIONEVENTS returns event name(s) from the events array
% based on the position of a robot in user-defined regions.

% BOT is the robot to detect the event for. Since different events
% correspond to each robot, BOT is used to pick one event from
% multiple_events. BOT is 1 for Cer and 2 for Cer1.

% POS2D is the position data that is provided by Player/Stage. It is a
% struct with at least two fields: px and py which provide the x and y
% location of the robot.

% REGIONS is a cell array with five columns and any number of rows. The
% first column contains the name of the state, the second thru fifth
% columns contain the upper and lower bounds of x-values and
% y-values in that region. Example:
% regions = { 'name',   x1,   y1,  x2,  y2}
% regions = { 'reg5',  -35,  -35,   0,   0;
%             'reg6',  -35,    0,   0,  35;
%             'reg8',    0,  -35,  35,   0;
%             'reg7',    0,    0,  35,  35};
%

% EVENTS is a cell array with four columns. Example:
% events = { 'name',  bot,  'event_type',  'type_name';
% events = { 'a1s',  1,  'Task',  'tsk1';
%    'a2s',  1,  'Task',  'tsk2';
%    'b1s',  2,  'Task',  'tsk1';
%    'b2s',  2,  'Task',  'tsk2';
%    'a5e',  1,  'Region',  'reg5';
%    'a6e',  1,  'Region',  'reg6';
%    'b5e',  2,  'Region',  'reg5';
%    'b6e',  2,  'Region',  'reg6'};


% These variables indicate which column of the cell array contains x
values
% and which columns contain y values.
name = 1;
x1 = 2;
x2 = 4;
y1 = 3;
y2 = 5;

% Shorten regions, x and y vaues of current position
r = regions;
px = pos2d.px;
py = pos2d.py;

j=1;
% Loop through regions
```

```
for i=1:size(regions,1)
    % Detect if robot position is within region boundaries
    if px < max(r{i,x1},r{i,x2}) && px > min(r{i,x1},r{i,x2}) && py <
max(r{i,y1},r{i,y2}) && py > min(r{i,y1},r{i,y2})
        region_name = regions{i,name};
        % Store all regions in a list of region_names
        region_names{j} = region_name;
        j=j+1;
    end
end

k=1;
% Loop through region_names
for i=1:size(region_names,2)
    % Get region_name matches in events
    multiple_events = events(strcmp(region_names(i),events(:,4)),1:2);

    % Loop through multiple_events
    for j = 1:size(multiple_events,1)
        % Match event_bot with bot
        if bot == multiple_events{j,2}
             event_name=multiple_events{j,1};

             % Add event_name to list of event_names
             event_names{k} = event_name;
             k=k+1;
             break
        end
    end
end
```

Appendix H: MATLAB Code for Goal Plan

This code was based on the work of a previous student. However, it was significantly modified to work with this project.

```
**************** func_GoalPlan.m ****************
function all_events = func_GoalPlan(goal, pos2dCer, pos2dCer1,
clientCer, clientCer1, map, planner, bot, events, regions)
addpath '~/Desktop/Shared'

% GOALPLAN sends the robot to the goal using the D*Lite path planner
% until an event is detected. Once an event is detected, the function
% returns to the calling function immediately with the event names.

% Inputs:
%   goal: [x y] coordinate of goal location
%   pos2dCer, pos2dCer1: Player/Stage position structs
%   clientCer, clientCer1: Player/Stage client pointers
%   map:
%   planner:
%   bot: integer for which robot, 1 for Cer (robot A), 2 for Cer1
(robot B)
%   events: User-defined list of events, format is {'name', bot,
'type', 'type_name'}
%   regions: User-defined list of regions, format is {'name', x1, y1,
x2, y2}

% Outputs:
%   all_events: array of event names

SAFETY=10; % Creates a safety boundary for the obstacle to avoid
collision
DISTANCE_TO_WAYPOINT = 1.5; % Triggers D*-Lite algorithm re-plan

while(isempty(goal))
    pause(1)
end

final_goal_x = goal(1,1);
final_goal_y = goal(1,2);
time_plan = 0;

% Read the robot position
while(~player('isfresh', pos2dCer))
    player('client_read', clientCer);
end
while(~player('isfresh', pos2dCer1))
    player('client_read', clientCer1);
end

if bot == 1
    x_start_player = double(pos2dCer.px);
    y_start_player = double(pos2dCer.py);
```

```
elseif bot == 2
    x_start_player = double(pos2dCer1.px);
    y_start_player = double(pos2dCer1.py);
% add additional bots here if needed
end

% Initialize region events
try
    event_names_1 = func_detectRegionEvents(1, pos2dCer, regions,
events);
    event_names_2 = func_detectRegionEvents(2, pos2dCer1, regions,
events);

    initial_events = [event_names_1 event_names_2];
    all_events = initial_events;
catch
    % If a robot position is outside one of the defined regions,
    % an error results. Inform the user of this error.
    Error_Data = func_printTable('regions',regions)

    fprintf('***** Error in Regions update!!! *****\nPlease ensure that
all robots are within a defined region!\nBot 1 pos2d: (%3.2f,
%3.2f)\nBot 2 pos2d: (%3.2f,
%3.2f)\n\n',pos2dCer.px,pos2dCer.py,pos2dCer1.px,pos2dCer1.py)
    return
end

% D*lite path planner
tic
[PPath, Bread_crumbs] = DESDstar_planner (x_start_player,
y_start_player, final_goal_x, final_goal_y, map, SAFETY);
time_plan = time_plan + toc;
% Bread_crumbs
[m,n] = size(Bread_crumbs);

while (n > 0) % It was (n > 1)

    % Define How far is the next point, I picked two points after
    % current location
    next_point_x=Bread_crumbs(1,1); % It was (1, 3)
    next_point_y=Bread_crumbs(2,1); % It was (2, 3)

    while(~player('isfresh', pos2dCer1))
        player('client_read', clientCer1);
    end
    while(~player('isfresh', pos2dCer))
        player('client_read', clientCer);
    end

    %% ---------------Start of the real time event detection---------
    % Detect events
    event_names_1 = func_detectRegionEvents(1, pos2dCer, regions,
events);
    event_names_2 = func_detectRegionEvents(2, pos2dCer1, regions,
events);

    all_events = [event_names_1 event_names_2];
```

```
    % Return to calling function if event is detected
    if ~isequal(all_events,initial_events)
        return
    end
    %% -----------------End of the real time event detection-----------


    %% -----------------D*-Lite re-plan path part---------------------
% If the stage not change, then the last goal hasn't changed. D*-Lite
% algorithm should keep on re-planning path till reach the goal
      player('planner_set_cmd_pose',planner,next_point_x, next_point_y,
0);
if bot == 1
    while(~player('isfresh', pos2dCer))
        player('client_read', clientCer);
    end
    % pdist=Pairwise distance
    dist=pdist([pos2dCer.px,pos2dCer.py;next_point_x,next_point_y]);
elseif bot == 2
    while(~player('isfresh', pos2dCer1))
        player('client_read', clientCer1);
    end
    % pdist=Pairwise distance
    dist=pdist([pos2dCer1.px,pos2dCer1.py;next_point_x,next_point_y]);
% add additional robots here, if needed
end
    if (dist > DISTANCE_TO_WAYPOINT)
       while(~player('isfresh', pos2dCer1))
           player('client_read', clientCer1);
       end
       tic
       if bot == 1
           [PPath, tmpBread_crumbs] = DESDstar_planner (pos2dCer.px,
pos2dCer.py, final_goal_x, final_goal_y, map,SAFETY);
       elseif bot == 2
           [PPath, tmpBread_crumbs] = DESDstar_planner (pos2dCer1.px,
pos2dCer1.py, final_goal_x, final_goal_y, map,SAFETY);
       % add additional robots here, if needed
       end

       % updating the new waypoints
       if (~isempty(tmpBread_crumbs));
           if (tmpBread_crumbs(1) ~= -1); % There are new waypoints
              time_plan = time_plan + toc;
              Bread_crumbs = tmpBread_crumbs;
           end
       end
    else
        Bread_crumbs(:,1)=[];   % Delete the next waypoints
    end
    [m,n] = size(Bread_crumbs);
end

    player('planner_set_cmd_pose', planner, final_goal_x,final_goal_y,
0);
```

```
end % End of GOALPLAN function
```

Appendix I: MATLAB Code for Go Goal

```
**************** func_GoGoal.m ****************
function all_events =
func_GoGoal(goal,client,pos2d1,pos2d2,laser,bot,events,regions)

% Initial default turn direction (either + or - 1)
si=1;

% Read robot position data
while(~player('isfresh', pos2d1))
    player('client_read', client);
end
while(~player('isfresh', pos2d2))
    player('client_read', client);
end

% Calculate robot's distance to goal & relative heading
if bot == 1
    dx = goal(1) - pos2d1.px;
    dy = goal(2) - pos2d1.py;
    da = atan2(dy,dx) - pos2d1.pa;
elseif bot == 2
    dx = goal(1) - pos2d2.px;
    dy = goal(2) - pos2d2.py;
    da = atan2(dy,dx) - pos2d2.pa;
end
% Calculate distance
dist = sqrt(dx^2 + dy^2);

% Correct for angles greater than +/- pi
if da > pi, da=da-2*pi; end
if da < -pi, da=da+2*pi; end

% Set initial heading (to angle) and velocity
turnRate = da;
forwardSpeed = 1;

% If angle is large, align along current heading before starting
while turnRate > pi/2 || turnRate < -pi/2
    % Turn in place if angle is large
    if bot ==1
        player('position2d_set_cmd_vel',pos2d1,0,0,turnRate);
    elseif bot ==2
        player('position2d_set_cmd_vel',pos2d2,0,0,turnRate);
    end

    % Read position data
    while(~player('isfresh', pos2d1))
        player('client_read', client);
    end
    while(~player('isfresh', pos2d2))
        player('client_read', client);
```

```
    end
    % Calculate relative heading
    if bot == 1
        turnRate = atan2(dy,dx) - pos2d1.pa;
    elseif bot == 2
        turnRate = atan2(dy,dx) - pos2d2.pa;
    end
end
% And stop
if bot ==1
        player('position2d_set_cmd_vel',pos2d1,0,0,0);
elseif bot ==2
        player('position2d_set_cmd_vel',pos2d2,0,0,0);
end

% Initialize events
try
        event_names_1 = func_detectRegionEvents(1, pos2d1, regions,
events);
        event_names_2 = func_detectRegionEvents(2, pos2d2, regions,
events);

        initial_events = [event_names_1 event_names_2];
        all_events = initial_events;
catch
    % If a robot position is outside one of the defined regions,
    % an error results. Inform the user of this error.
    Error_Data = func_printTable('regions',regions)

    fprintf('***** Error in Regions update!!! *****\nPlease ensure that
all robots are within a defined region!\nBot 1 pos2d: (%3.2f,
%3.2f)\nBot 2 pos2d: (%3.2f,
%3.2f)\n\n',pos2d1.px,pos2d1.py,pos2d2.px,pos2d2.py)
    return
end
% End of Region Initialization-----------------


%% Begin forever loop
while dist > 1 && all_events == initial_events % Loop while not at goal
    % Based on last laser scan, decide where to go next

    % Find angles where laser is clear
    isEight = laser.scan(1,:)==8;

    if sum(isEight) == 361 % Nothing in range
        % Go in direction of goal
        turnRate = da;
        forwardSpeed = 2;
        %fprintf('** Nothing in view, go to goal **\n')
    elseif sum(laser.scan(1,45:314) < 0.05)
        % obstacle very close, directly in front

        % back up to avoid hitting obstacle
        turnRate = si*rand;
        forwardSpeed = -1;
```

```
        %fprintf('** Obstacle less than 0.25 meters in front, back up
**\n')
    elseif sum(laser.scan(1,118:242) < .5)
        % obstacle 1 meter directly in front

        turnRate = si*pi/2;
        forwardSpeed = 0.25;
        %fprintf('** Obstacle less than 1 meter in front, turn pi/2
**\n')
    elseif sum(laser.scan(1,:) < 0.05)
        % obstacle very close, but not directly in front

        % back up to avoid hitting obstacle
        turnRate = si*pi/2;
        forwardSpeed = -1;
        %fprintf('** Obstacle less than 0.25 meters on side, back up
**\n')
    elseif sum(laser.scan(1,146:214) < 1)
        % obstacle 2 meters directly in front

        turnRate = si*pi/2;
        forwardSpeed = 0.5;
        %fprintf('** Obstacle 1-2 meters in front, turn pi/2 **\n')
    elseif sum(laser.scan(1,163:197) < 2)
        % obstacle 4 meters directly in front

        turnRate = si*pi/4;
        forwardSpeed = 1;
        %fprintf('** Obstacle 2-4 meters in front, turn pi/4 **\n')
    elseif sum(laser.scan(1,169:191) < 3)
        % obstacle 6 meters directly in front

        turnRate = si*pi/6;
        forwardSpeed = 2;
        %fprintf('** Obstacle 4-6 meters in front, turn pi/6 **\n')
    elseif sum(laser.scan(1,171:189) < 4)
        % obstacle 8 meters directly in front

        % Set turning direction
        si=sign(da);

        turnRate = si*pi/8;
        forwardSpeed = 2.5;
        %fprintf('** Obstacle 6-8 meters in front, turn pi/8 and set si
**\n')
    else % obstacle, but not directly in front
        % Go in goal direction if possible, otherwise go to first clear
        % angle.

        % Find closest index match to heading
        daInd = find(abs(da - laser.scan(2,:)) < 0.01);
        if ~isempty(daInd) % makes sure that this exists
            % If goal heading is not clear, then find the biggest clear
section and go that way.
            if ~isEight(daInd(1))
                % something is blocking goal direction, go in different
direction
```

```
                % Get edges of sections where laser is clear (including
end points)
                notFirst = ~isEight(1);
                notLast = ~isEight(length(isEight));
                edgeInd = find([isEight notLast] ~= [notFirst
isEight]);

                % Fix last index
                edgeInd(length(edgeInd)) = 360;

                % For right now, always choose first clear section
                % Future improvement: go to clear section closest to
goal angle,
                % provided that it is wide enough for robot to fit
                for i=1:length(edgeInd)
                    if laser.scan(1,edgeInd(i)+1) >= 8
                        turnRate = laser.scan(2,edgeInd(i)+1);
                        forwardSpeed = 1;
                        break
                    elseif i == length(edgeInd)
                        turnRate = laser.scan(2,edgeInd(i)+1);
                        forwardSpeed = 1;
                    end
                end
                si=sign(da);
                %fprintf('** Go to first clear section. **\n')
            else
                % nothing directly in front, continue toward goal

                %Set turn rate and forward speed
                turnRate = da;
                forwardSpeed = 1;
                %fprintf('** Nothing directly in front, go to goal.
**\n')
            end % *** End Go to Clear Section ***
        else
            % goal heading does not match an index because it is >
pi/2 or < -pi/2

            turnRate = si*pi/2;
            forwardSpeed = 0;
            %fprintf('** No match found for goal heading. (Line 182)
**\n')
            si=sign(da);
        end
    end
    %fprintf('turnRate = %1.4g; forwardSpeed = %1.4g; da = %1.4g
\n',turnRate, forwardSpeed, da)

    % Do whatever was decided
    if bot == 1
        player('position2d_set_cmd_vel',pos2d1, forwardSpeed, 0,
turnRate);
    elseif bot == 2
        player('position2d_set_cmd_vel',pos2d2, forwardSpeed, 0,
turnRate);
```

```
    end

    % Read position data
    while(~player('isfresh', pos2d1))
        player('client_read', client);
    end
    while(~player('isfresh', pos2d2))
        player('client_read', client);
    end

    if bot == 1
        % Calculate distance to goal
        dx = goal(1) - pos2d1.px;
        dy = goal(2) - pos2d1.py;
        % Calculate relative heading
        da = atan2(dy,dx) - pos2d1.pa;
    elseif bot == 2
        dx = goal(1) - pos2d2.px;
        dy = goal(2) - pos2d2.py;
        % Calculate relative heading
        da = atan2(dy,dx) - pos2d2.pa;
    end

    dist = sqrt(dx^2 + dy^2);

    % Correct angle
    if da > pi, da=da-2*pi; end
    if da < -pi, da=da+2*pi; end

% Detect events
        event_names_1 = func_detectRegionEvents(1, pos2d1, regions,
events);
        event_names_2 = func_detectRegionEvents(2, pos2d2, regions,
events);

        all_events = [event_names_1 event_names_2];


end

% stop the robot
if bot ==1
    player('position2d_set_cmd_vel',pos2d1, 0, 0, 0);
elseif bot ==2
    player('position2d_set_cmd_vel',pos2d2, 0, 0, 0);
end

end
```

Appendix J: MATLAB Code for Calculate Costs

********** **func_calculateCosts.m** **********

```
function [A,C] = func_calculateCosts(states, events, tasks, start)
% FUNC_CALCULATECOSTS returns an adjacency matrix and cost matrix based
% on states, events, and user-defined tasks.

% states = {'name', {'event_name', 'state_name', 'isControllable:
c_uc', 'cost'}, 'isDone: 0_1'}
% events = {'name', bot, 'type', 'type_name'}

% Valid types:
% tasks = {'name', x1, y1, 'function', 'From: task_name'}
% regions = {'name', x1, y1, x2, y2}
% *** be sure to have a task/event for starting location ***

% start(bot) = [pos2d_BOT.px pos2d_BOT.py]

n = size(states,1);

% Calculate costs based on event and task information

% Task finish Index
task_finished = cell(n,1);

% Loop through each state
for i = 1:n
    % Loop through the events of each state
    for k = 1:size(states{i,2},1)
        if strcmp(states{i,2}{k,3},'c')
            % is controllable event, calculate cost
            event_name = states{i,2}{k,1};

            distance = func_calcDistance(tasks, events, event_name,
start);

            states{i,2}{k,5} = distance;

            % check previous event before this state
            for j = 1:n
                for m = 1:size(states{j,2},1)
                    % check for matching state in other states
                    if ~strcmp(states{j,2}{m,1}(3),'e') % only options
are 'f','s', and 'e': 'e' is the one you don't want
                        % check for task finish or event start
                    if strcmp(states{i,1},states{j,2}{m,2})
                        % check if opposite bot
                        this_bot = events{strcmp(event_name,
events(:,1)),2};
                        event_bot = events{strcmp(states{j,2}{m,1},
events(:,1)),2};
                        if this_bot ~= event_bot
                            task_finished{i} = states{j,2}{m,1};
                            % what happens if multiple events (ie
task_finished not empty)?
                        end
```

```
                    end
                    end
                end
            end
        else
            % is uncontrollable event, cost is zero
            states{i,2}{k,5} = 0;

            % if task finish event, save event name
            if strcmp(states{i,2}{k,1}(3),'f')
                task_finished{i} = states{i,2}{k,1};
                % what happens if multiple events (ie task_finished not
                % empty)?
            end
        end
    end
end

% Update input matrices A and C for optimization algorithm
% n is number of states in *.fsm file
% A = adjacency matrix, NxN, is equal to 1 for connected states
% C = cost matrix, NxN, is equal to distance depending on event
A = zeros(n,n);
C = inf*ones(n,n);

for i = 1:n
    for j = 1:n
        for k = 1:size(states{i,2},1)
            if strcmp(states{j,1},states{i,2}{k,2})
                % new state is an allowed state
                A(i, j) = 1;   % states(i,j) are connected to each
other

                % use pre-calculated cost for state, unless
task_finished,
                % then calculate subtracted distance as cost
                if isempty(task_finished{i})
                    C(i, j) = states{i,2}{k,5};
                else
                    % Get event name
                    event_name = task_finished{i};
                    if strcmp(event_name(3),'f')
                        % convert a finished event to a task start
event
                        event_name = [event_name(1:2) 's'];
                    end
                    % Find cost of task_finished event
                    new_distance = func_calcDistance(tasks, events,
event_name, start);
                    % calculate subtracted distance
                    subtracted_distance = states{i,2}{k,5} -
new_distance;
                    if subtracted_distance > 0
                        C(i,j) = subtracted_distance;
                        %fprintf('C(%u,%u) = %3.2g instead of
%3.2g\n',i,j,subtracted_distance,states{i,2}{k,5})
                    else
```

```
                    C(i,j) = 0;
                end
            end
        end
    end
end
end

function distance = func_calcDistance(tasks, events, event_name,start)
% look in column 1 of events matrix for event_name match and return
task_name from column 4
task_name = events{strcmp(event_name,events(:,1)), 4};

% look in column 1 of tasks matrix for task_name match and return x1
and y1 from columns 2 and 3, respectively
x1 = tasks{strcmp(task_name,tasks(:,1)),2};
y1 = tasks{strcmp(task_name,tasks(:,1)),3};

% look in column 1 of tasks matrix for task_name match and return
from_event_name from column 5
from_task_name = tasks{strcmp(task_name, tasks(:,1)),5};

if strcmp(from_task_name,'start')
    % use current robot position
    bot = events{strcmp(event_name, events(:,1)),2};
    x2 = start(bot,1);
    y2 = start(bot,2);
else
    % look in column 1 of tasks matrix for task_name match and return
x1 and y1 from columns 2 and 3, respectively
    x2 = tasks{strcmp(from_task_name,tasks(:,1)),2};
    y2 = tasks{strcmp(from_task_name,tasks(:,1)),3};
end
distance = sqrt((x1-x2)^2+(y1-y2)^2);
end
```

Appendix K: MATLAB Code for Dijkstra's Algorithm

The code for Dijkstra's algorithm is shown below. The code was found on MATLAB Central [7].

```
*************** func_Dijkstra.m ***************
function [costs,paths] = func_Dijkstra(AorV,xyCorE,SID,FID,iswaitbar)
%DIJKSTRA Calculate Minimum Costs and Paths using Dijkstra's Algorithm
% http://www.mathworks.com/matlabcentral/fileexchange/20025-advanced-
dijkstras-minimum-path-algorithm
%   Inputs:
%     [AorV] Either A or V where
%         A   is a NxN adjacency matrix, where A(I,J) is nonzero (=1)
%               if and only if an edge connects point I to point J
%               NOTE: Works for both symmetric and asymmetric A
%         V   is a Nx2 (or Nx3) matrix of x,y,(z) coordinates
%     [xyCorE] Either xy or C or E (or E3) where
%         xy  is a Nx2 (or Nx3) matrix of x,y,(z) coordinates
%             (equivalent to V)
%               NOTE: only valid with A as the first input
%         C   is a NxN cost (perhaps distance) matrix, where C(I,J)
%             contains the value of the cost to move from point I to
%             point J
%               NOTE: only valid with A as the first input
%         E   is a Px2 matrix containing a list of edge connections
%               NOTE: only valid with V as the first input
%         E3  is a Px3 matrix containing a list of edge connections in
%             the first two columns and edge weights in the third
%             column
%               NOTE: only valid with V as the first input
%     [SID] (optional) 1xL vector of starting points
%         if unspecified, the algorithm will calculate the minimal path
from
%         all N points to the finish point(s) (automatically sets SID =
1:N)
%     [FID] (optional) 1xM vector of finish points
%         if unspecified, the algorithm will calculate the minimal path
from
%         the starting point(s) to all N points (automatically sets FID
= 1:N)
%     [iswaitbar] (optional) a scalar logical that initializes a
waitbar if nonzero
%
%   Outputs:
%     [costs] is an LxM matrix of minimum cost values for the minimal
paths
%     [paths] is an LxM cell array containing the shortest path arrays
%
%   Revision Notes:
%     (4/29/09) Previously, this code ignored edges that have a cost of
zero,
%     potentially producing an incorrect result when such a condition
exists.
```

```
%     I have solved this issue by using NaNs in the table rather than a
%     sparse matrix of zeros. However, storing all of the NaNs requires
more
%     memory than a sparse matrix. This may be an issue for massive
data
%     sets, but only if there are one or more 0-cost edges, because a
sparse
%     matrix is still used if all of the costs are positive.
%
%   Note:
%     If the inputs are [A,xy] or [V,E], the cost is assumed to be (and
is
%       calculated as) the point-to-point Euclidean distance
%     If the inputs are [A,C] or [V,E3], the cost is obtained from
either
%       the C matrix or from the edge weights in the 3rd column of E3
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using
[A,xy] inputs
%       n = 7; A = zeros(n); xy = 10*rand(n,2)
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       IJ = I + n*(J-1); A(IJ) = 1
%       [costs,paths] = dijkstra(A,xy)
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using
[A,C] inputs
%       n = 7; A = zeros(n); xy = 10*rand(n,2)
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       IJ = I + n*(J-1); A(IJ) = 1
%       a = (1:n); b = a(ones(n,1),:);
%       C = round(reshape(sqrt(sum((xy(b,:) - xy(b',:)).^2,2)),n,n))
%       [costs,paths] = dijkstra(A,C)
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using
[V,E] inputs
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]); E = [I(:) J(:)]
%       [costs,paths] = dijkstra(V,E)
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using
[V,E3] inputs
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]);
%       D = sqrt(sum((V(I(:),:) - V(J(:),:)).^2,2));
%       E3 = [I(:) J(:) D]
%       [costs,paths] = dijkstra(V,E3)
%
%   Example:
```

```
%       % Calculate the shortest distances and paths from the 3rd point
to all the rest
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]); E = [I(:) J(:)]
%       [costs,paths] = dijkstra(V,E,3)
%
%   Example:
%       % Calculate the shortest distances and paths from all points to
the 2nd
%       n = 7; A = zeros(n); xy = 10*rand(n,2)
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       IJ = I + n*(J-1); A(IJ) = 1
%       [costs,paths] = dijkstra(A,xy,1:n,2)
%
%   Example:
%       % Calculate the shortest distance and path from points [1 3 4]
to [2 3 5 7]
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]); E = [I(:) J(:)]
%       [costs,paths] = dijkstra(V,E,[1 3 4],[2 3 5 7])
%
%   Example:
%       % Calculate the shortest distance and path between two points
%       n = 1000; A = zeros(n); xy = 10*rand(n,2);
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       D = sqrt(sum((xy(I,:)-xy(J,:)).^2,2));
%       I(D > 0.75,:) = []; J(D > 0.75,:) = [];
%       IJ = I + n*(J-1); A(IJ) = 1;
%       [cost,path] = dijkstra(A,xy,1,n)
%       gplot(A,xy,'k.:'); hold on;
%       plot(xy(path,1),xy(path,2),'ro-','LineWidth',2); hold off
%       title(sprintf('Distance from 1 to 1000 = %1.3f',cost))
%
% Web Resources:
%   <a
href="http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm">Dijkstra's
Algorithm</a>
%   <a
href="http://en.wikipedia.org/wiki/Graph_%28mathematics%29">Graphs</a>
%   <a href="http://en.wikipedia.org/wiki/Adjacency_matrix">Adjacency
Matrix</a>
%
% See also: gplot, gplotd, gplotdc, distmat, ve2axy, axy2ve
%
% Author: Joseph Kirk
% Email: jdkirk630@gmail.com
% Release: 1.1
% Date: 4/29/09

% Process Inputs
error(nargchk(2,5,nargin));
all_positive = 1;
[n,nc] = size(AorV);
```

```
[m,mc] = size(xyCorE);
[E,cost] = processInputs(AorV,xyCorE);
if nargin < 5
    iswaitbar = 0;
end
if nargin < 4
    FID = (1:n);
end
if nargin < 3
    SID = (1:n);
end
if max(SID) > n || min(SID) < 1
    eval(['help ' mfilename]);
    error('Invalid [SID] input. See help notes above.');
end
if max(FID) > n || min(FID) < 1
    eval(['help ' mfilename]);
    error('Invalid [FID] input. See help notes above.');
end

isreversed = 0;
if length(FID) < length(SID)
    E = E(:,[2 1]);
    cost = cost';
    tmp = SID;
    SID = FID;
    FID = tmp;
    isreversed = 1;
end

L = length(SID);
M = length(FID);
costs = zeros(L,M);
paths = num2cell(nan(L,M));

% Find the Minimum Costs and Paths using Dijkstra's Algorithm
if iswaitbar, wbh = waitbar(0,'Please Wait ... '); end
for k = 1:L
    % Initializations
    if all_positive, TBL = sparse(1,n); else TBL = NaN(1,n); end
    min_cost = Inf(1,n);
    settled = zeros(1,n);
    path = num2cell(nan(1,n));
    I = SID(k);
    min_cost(I) = 0;
    TBL(I) = 0;
    settled(I) = 1;
    path(I) = {I};

    while any(~settled(FID))
    % Update the Table
    TAB = TBL;
    if all_positive, TBL(I) = 0; else TBL(I) = NaN; end
    nids = find(E(:,1) == I);
    % Calculate the Costs to Neighbor Points and Record Paths
    for kk = 1:length(nids)
        J = E(nids(kk),2);
```

```
        if ~settled(J)
        c = cost(I,J);
        % *** Calculate incremental cost, if appropriate
        if size(path{I},2)>1
            prevI = path{I}(length(path{I})-1); % Find previous state
on path
            if bot(prevI,I) ~= bot(I,J) % Opposite bot
            % Incremental cost = <cost> + <this bot sum> - <previous
bot sum>
            incremental_cost = c + TB(I,bot(I,J)) - TB(I,bot(prevI,I));
            c = incremental_cost;
            end
        end
        if all_positive, empty = ~TAB(J); else empty = isnan(TAB(J));
end
        if empty || (TAB(J) > (TAB(I) + c))
            TBL(J) = TAB(I) + c;
            TB(J,1) = TB(I,1);
            TB(J,2) = TB(I,2);
            TB(J,bot(I,J)) = TB(I,bot(I,J)) + cost(I,J);
            if isreversed
            path{J} = [J path{I}];
            else
            path{J} = [path{I} J];
            end
        else
            TBL(J) = TAB(J);
            TB(J,1) = TB(I,1);
            TB(J,2) = TB(I,2);
        end
        end
    end

        if all_positive, K = find(TBL); else K = find(~isnan(TBL)); end
        % Find the Minimum Value in the Table
        N = find(TBL(K) == min(TBL(K)));
        if isempty(N)
            break
        else
            % Settle the Minimum Value
            I = K(N(1));
            min_cost(I) = TBL(I);
            settled(I) = 1;
        end
    end
    % Store Costs and Paths
    costs(k,:) = min_cost(FID);
    paths(k,:) = path(FID);
    if iswaitbar, waitbar(k/L,wbh); end
end
if iswaitbar, close(wbh); end

if isreversed
    costs = costs';
    paths = paths';
end
```

```
if L == 1 && M == 1
    paths = paths{1};
end

% ------------------------------------------------------------------
    function [E,C] = processInputs(AorV,xyCorE)
        C = sparse(n,n);
        if n == nc
            if m == n
                if m == mc % Inputs: A,cost
                    A = AorV;
                    A = A - diag(diag(A));
                    C = xyCorE;
                    all_positive = all(C(logical(A)) > 0);
                    E = a2e(A);
                else % Inputs: A,xy
                    A = AorV;
                    A = A - diag(diag(A));
                    xy = xyCorE;
                    E = a2e(A);
                    D = ve2d(xy,E);
                    all_positive = all(D > 0);
                    for row = 1:length(D)
                        C(E(row,1),E(row,2)) = D(row);
                    end
                end
            else
                eval(['help ' mfilename]);
                error('Invalid [A,xy] or [A,cost] inputs. See help
notes above.');
            end
        else
            if mc == 2 % Inputs: V,E
                V = AorV;
                E = xyCorE;
                D = ve2d(V,E);
                all_positive = all(D > 0);
                for row = 1:m
                    C(E(row,1),E(row,2)) = D(row);
                end
            elseif mc == 3 % Inputs: V,E3
                E3 = xyCorE;
                all_positive = all(E3 > 0);
                E = E3(:,1:2);
                for row = 1:m
                    C(E3(row,1),E3(row,2)) = E3(row,3);
                end
            else
                eval(['help ' mfilename]);
                error('Invalid [V,E] inputs. See help notes above.');
            end
        end
    end

    % Convert Adjacency Matrix to Edge List
    function E = a2e(A)
        [I,J] = find(A);
```

```
        E = [I J];
    end

    % Compute Euclidean Distance for Edges
    function D = ve2d(V,E)
        VI = V(E(:,1),:);
        VJ = V(E(:,2),:);
        D = sqrt(sum((VI - VJ).^2,2));
    end
end
```