Robot Soccer Strategy

Harshvardhan Sukthankar
Steven Weinberg

A thesis submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF APPLIED SCIENCE

Supervisor: B. Benhabib

Department of Mechanical and Industrial Engineering
University of Toronto

March, 2008
Algorithm based decision making can be a challenging task in a system with dynamic interactions. An example of such a system is a robotic soccer match in which autonomous robots must navigate around and win against obstacles in the form of an opposing team that is itself being run by a program. This thesis focuses on developing a strategy that is adaptable to a changing environment on a soccer pitch.

The strategy is implemented by breaking down reactions from the robot team into two modes: offensive and defensive, based on which team is in possession of the ball. The pitch is divided into discrete cells with Cartesian coordinates. On determining the appropriate mode, an individual cost map is generated for each robot by assigning appropriate costs to each cell based on the proximity of the cell to obstacles to be avoided, or affinity to pitch zones that are strategically important. A destination cell is set for the robot, and a path is created and executed.
Acknowledgements

We would like to thank the members of UTRA in general, and Ramy Ghattas, Onome Igharo, Isaac Good, Kshitij Saxena, and Audy Chang specifically for help given throughout this project. We would also like to thank the members of UTAIR for first (to our knowledge) introducing robot soccer to U of T. Special thanks goes to Professor Benhabib for his supervision and advice during the course of the thesis.
List of Figures

Fig 1.1 A typical small sized robot game ................................................................. 39
Fig 1.2 Small sized competition pitch dimensions .................................................... 39
Fig 2.1 Potential field method problem .................................................................... 40
Fig 2.2 Tangent planning method ............................................................................. 40
Fig 3.1 AI Program Overview ................................................................................ 41
Fig 3.2 Sample cost map ......................................................................................... 42
Fig 3.3 Sample scenario for pathfinding algorithm .................................................. 42
Fig 5.1 Defensive situation ....................................................................................... 43
Fig 5.2 Converging behavior .................................................................................... 44
Fig 5.3 Converging behavior .................................................................................... 45
Fig 5.4 Diverging behavior continued ...................................................................... 46
List of Tables

Table 3.1a: Initial values of cost_from. .......................................................... 47
Table 3.1b: Initial values of connect_matrix .................................................. 47
Table 3.1c: Initial values of the checker matrix .............................................. 47
Table 3.2a: cost_from during step 1. ............................................................. 48
Table 3.2b: checker during step 1. ............................................................... 48
Table 3.3a: cost_from during step 2. ............................................................. 48
Table 3.3b: checker during step 2. ............................................................... 49
Table 3.4a: cost_from during step 3. ............................................................. 49
Table 3.4b: checker during step 3. ............................................................... 49
Table 3.5a: cost_from during step 4. ............................................................. 50
Table 3.5b: checker during step 4. ............................................................... 50
Table 3.6a: cost_from during step 5. ............................................................. 50
Table 3.6b: checker during step 5. ............................................................... 51
Table 3.7a: cost_from during step 6. ............................................................. 51
Table 3.7b: checker during step 6. ............................................................... 51
Chapter 1: Introduction

1.1 Summary

The task of guiding autonomous robots is complicated by operating in a dynamic and irregular environment. Human control may not always be suitable or desirable for an application, and may have to be fully or partially replaced by an artificial decision making procedure. Decisions made by the controlling program must be quick and the response to changes in variable states must be efficient. Algorithm based decision procedures are useable in a wide range of areas, from military applications to manufacturing. The focus of this thesis was on soccer playing robots. These robots have to compete autonomously in a dynamic environment against opponent robots operating under a competing set of algorithms. A reasonable program for winning such a soccer game, as well as a good base for future development in this area was developed.

1.2 Background

Founded in 1993, the Robocup is an international competition featuring soccer playing robots founded with the ambitious goal of creating, by the mid 21st century, a team of fully autonomous humanoid robots capable of beating the winners of the FIFA Soccer World Cup. Soccer is useful as a tool for developing the skills to work with dynamic systems.
The current competition features autonomous robots that come in different categories: small sized, medium sized, four legged and humanoid. The University of Toronto Robotics Association (UTRA) has a robot soccer division planning to build a robot team capable of eventually competing in the Robocup. UTRA is also planning to have a robot ready to compete in MiRosot (Micro Robot Soccer Tournament) in 2008.

The small sized robots (see fig 1.1) have a weight of less than 650g and have max dimensions of 7.5 x 7.5 x 7.5 cm and are placed on a pitch (see fig 1.2 for dimensions) with goals at each end.

A vision system is mounted over the pitch, sending the computer that controls the robots information on the positions of all objects on the field. UTRA will likely be using Miabot Classic mobile robots developed by Merlin Robotics, although other options are being looked into. The Miabots are equipped with Bluetooth communications and are wirelessly controllable with an ASCII serial protocol.

### 1.2 Objectives

UTRA was looking to build an AI and vision system for robot soccer, capable of competing effectively in MiRosot and using that as a platform to attempt to qualify and compete in the RoboCup.

This thesis, being conducted with the purpose of assisting UTRA’s goals, focused on developing the AI. The AI was required be able to navigate the robots in a dynamic environment and, ideally, anticipate reactions from opposing AIs. The AI was required to be capable of steering
the robots while avoiding collisions with other robots, passing the ball and scoring goals, while conceding as few goals as possible.

The primary objective of the thesis was to design and code working passing, movement, scoring, and defensive algorithms. Focus was put on completing an entire AI system, and testing it to prove its efficacy in time to compete in MiRosot. As such, research often had to be done alongside algorithm design and coding rather than strictly as a precursor. Optimization, while vitally important, was secondary to timely completion.

As a secondary objective, it was desired that this AI system be easily adapted and improved upon by members of UTRA in the future. As such, the code was to be well-commented and as simple and modular as possible. The functions were also required to be relatively generic, capable of being incorporated into a wide range of strategies for best optimization potential.

1.3 Motivation

The skills developed in a robot soccer program have wide-ranging applications with multi-agent collaboration involving getting robots to work together to accomplish something they could not do on their own. Future applications might take this to the microscopic level, where teams of robots could search out and destroy cancer cells or the like. Other applications that could benefit from robotic autonomous teamwork are construction, repair, and manufacturing. Other areas that are developed in robot soccer are design of autonomous agent, real-time reasoning, strategy acquisition, vision, and robotics. For robot soccer that utilizes onboard sensors rather than an external vision system, there is also sensor-fusion.¹
Chapter 2: Methodology

2.1 Resources

The majority of the coding was done in the C programming language, using the Microsoft Visual C++ IDE v. 6.0, and in the Engineering Computing Facilities at the University of Toronto.

2.2 Approach

Generalized soccer strategies had to be considered and applied to a robotic environment with limitations of the robots in mind. In a soccer match, the objective is to win by outscoring the opponent. It was noted that teams play differently based on whether they have possession or the opponents do. In case of opponent position, soccer teams drop back into their own half to defend their goal; with different players often covering opponents to intercept passes. The idea of creating different modes of play – based on which team possessed the ball – was developed and integrated into the thesis concept at the outset.

Limitations of the robots had to be considered. The Miabot Classic, the robots available for UTRA use, does not have kicking or trapping mechanisms; balls are passed or shot simply by moving into them. Algorithms for passing and dribbling had to be kept simple, and the decision procedure in the attacking mode reflects the simplicity. Instead of complicated passing maneuvers, each robot is assigned to simply push the ball forward when the ball is located within a zone designated to it.
In defensive mode, real soccer teams both mark opponents without the ball, and also designate one or more players to tackle the opponent with the ball. The difficulty in tackling is a major limitation in robot soccer; rules that robots do not crash into and damage each other, and the robots do not possess limbs to steal the ball from an opponent. Therefore it was decided to limit defensive play to marking and interceptions in order to regain possession. This plan was expected to be especially effective, as the rules state that a robot cannot travel more than 50mm with the ball without passing. Thus, many interception opportunities should arise, and hampered passing lanes are a severe handicap. A plan was formulated to have the robots block passes between opponent team mates, and also block shots on goal while playing defensively.

With the general overarching strategy developed, tactics at the smaller scale also had to be considered. The robots had to have an idea of the conditions on the field, giving them an idea of which areas are favorable to move towards and which should be avoided. Several concepts were considered:

1. Potential field: The basis behind this method is giving the robot that the path is being planned for a “negative charge,” and other objects “positive” or “negative charges,” based on what they are. Then, having like charges attract and opposite charges repel, the vectors of the effects of the various objects on the robot are calculated. Using weighting and some basic rules, the robot will head towards a desired location while avoiding obstacles. This method seems to have some advantages. One such is that it requires very little computation. Given coordinates of the various objects on the field, distances and direction are readily calculated. As well, the output is a direction for the
robot to go, one that probably won’t vary very much in short intervals of time, rather than waypoints from which a direction must be extracted, and which may change more in a small time interval.

However, it was decided not to use this method because it was predicted that it would be difficult to avoid bad decisions in certain situations. One example is when the target location for the robot with the ball lies along a line that passes directly between two closely spaced opponents (see fig 2.1).

While common sense might dictate to pass around one side or the other, this method will choose a path directly between the opponents, giving a good chance of being intercepted. While this may be avoidable with careful programming and addition of rules, the focus on having a fully operational system by the summer put a time constraint on such that it was felt that this method would not be optimal. As well, it doesn’t seem to be as open to further modification as other systems. Since one of the goals of this endeavor is to create a platform for constant improvement by UTRA members in the future, this is a serious flaw.

2. Tangent Planning: Similar to the final method chosen, this method involves representing each obstacle by a larger shape representing a “safe-zone” around the obstacle in question (see fig 2.2vii). In this method, the safe-zone is circular. Noting that the shortest path around any circle from a point is along the tangent, possible path sections are plotted from the robot’s location and the target position to the tangents of the closest obstacle safe-zones. Tangents are also plotted from these points at the
tangents of the safe-zone to the tangents of the next closest safe-zones. This process is continued until all possible path-sections are computed. A heuristic or directcalculation model is then used to choose which path sections to travel upon.

It was felt that this method was unnecessarily complex. In addition, it does not work well with the intended plan to warp the safety-zones from a circular formation based on object velocity.

3. Heuristic Approaches: The heuristic approach most closely considered was A*, which seems to be widely used in path finding applications. The advantages of this over another system, m Robust Tracing, are demonstrated in an applet available at http://www.luar.com.hk/demo/pathfinding.htm.

A* ranks the available nodes according to a heuristic and then visits the nodes in the order of their ranking. It maintains a set of partial paths, and backs up if it gets stuck and follows a new path. Eventually, a near-optimal path is found.

However, it was felt that given the small number of obstacles present on the robot soccer field, a direct-calculation approach would not only directly obtain an optimal path (within the constraints of the inputs) without relying on a heuristic, but might also be computationally quicker as well.

A cost map based approach was chosen. This approach is fully described in section 3.2.
Chapter 3: Strategy

3.1 Overview

An overview of the AI algorithm can be seen in fig 3.1. The program first takes in the data from the vision software. It then decides which pre-programmed situation fits the object positions on the field. On offence, the robot’s cost map, a virtual topography of the field, with higher costs being “high” areas to be avoided, is created straight away, whereas on defence, each robot is first assigned a role, and then the robot’s cost map is created. A decision is then made whether to pass/shoot (if it is possible) or to move. A passing/shooting or movement plan is formulated, and then executed by sending the appropriate commands to the robot. The next robot’s movement is then decided upon.

3.2 Conditions

To determine an effective strategy, the conditions on the field had to first be ascertained. To this end, the pitch was split into discrete cells in a matrix format. The Vision program then provided the positional data of all objects on the pitch, as well as their respective velocities and accelerations. A scenario was then decided upon. In the form taken at the time of this report, there were only two scenarios available, based on possession of the ball. Possession is determined by the ball position, after offsetting to compensate for motion and the robot in its closest proximity, also after proper compensation. If the coordinates of the ball were closer to the coordinates of an opponent robot, the opposing team would be identified as having possession and the program would switch to a defensive scenario. In the event of the ball coordinates being
closer to a team robot or equidistantly close to a team and enemy robot, the team would be identified as having possession and the program would switch to an offensive scenario.

Code-wise, possession is determined by finding the distance from the ball to every robot on the field. The shortest distances to a team robot and an enemy robot are respectively stored, and compared.

The scenario identified is used in the construction of the individual cost maps that determine each robot’s behavior. More scenarios, allowing more detailed situational control, could easily be implemented by simply creating new methods of modifying the cost map for each situation, thus accomplishing the stated goal of ease of improvement of the program.

### 3.3 Cost map generation

Each robot was assigned an individual cost map, generated per vision refresh. The cost map consisted of an integer cost being assigned to each cell on the pitch grid making up the field. High cost areas represented areas to be avoided, whereas low cost areas were areas the robot would tend to be attracted to. In general, all cells on the field started out with a relatively low cost. The costs in various cells would then be modified by their proximity to objects of note on the field. Objects to be avoided added a high cost modification to cells they occupied and cells in close proximity, with this modification becoming reduced in magnitude as the distance from the object increased. Conversely, cells occupied by objects the robot should tend towards induced a
negative cost, with the modification becoming mitigated with distance. It should be noted that “object” here is used loosely, as “objects” to be avoided might include edges of the field on offence, or, in future revisions, zones that decrease pass potential. Also included would be such areas as passing lanes between opponent robots when on defense.

Determining the values of the costs was based on the scenarios described in sections 3.3.1 and 3.3.2. A sample cost map is shown in fig 3.2.

3.3.1 Defensive scenario

The important considerations in a defensive scenario were 1) preventing the opposing team from scoring, and 2) regaining possession of the ball. Noting that Robocup rules require passing within 50mm of first contact with the ball, and in order to prevent collisions between robots, possession is regained not by moving directly towards the opposing robot with possession, but by restricting and intercepting potential passes between opposing team robots.

Noting that the shortest path from a point to a line is perpendicular to that line, a projection function was used to find this shortest distance from each defending robot to the line lying between the opponent robot with the ball and another attacking robot to which a potential pass may have been directed. This function worked by finding the coordinates on the line such that the dot product of the intercepting path and the passing path would equal zero.
Determining which robot is set to which task was set by using a priority list. The highest priority path to block was the one lying between the ball and the center of our team’s net, the “shooting lane”. The friendly robot closest to this path was assigned to block this lane, regardless of all considerations. The remaining friendly robots were each assigned a potential passing lane between the opponent in possession and another opponent robot. Which lane each robot was assigned was based on the distance between each robot and the path in question. The closest friendly robot, not previously assigned to another task, was assigned the task of blocking the pass.

For each robot thus assigned a path to block, the cells along this path were assigned a negative cost modification to encourage movement towards the line. Opponent robots were still given a positive cost modification to increase spacing between a blocking robot and the opponents to prevent an opponent from maneuvering around a blocking robot.

### 3.3.2 Offensive scenario

In the offensive scenario, the team is designed to split up and move upwards towards the opponent goal. The robot with the ball in possession heads directly towards the net; the cost map for this robot has its values around the opponent goal set to low numbers. In order to ensure that the robots spread out, the other robots on the same team are given high costs so they will tend to move away from each other, treating each other as obstacles en route towards the opponent end of the pitch.
As mentioned in section 3.3.1, the robot is not allowed to hold onto the ball for too long. The aim of each robot in possession on offence was to simply push the ball towards the enemy goal.

3.4 Path Finding

After the cost map is generated, a destination has to be found and a path has to be set. Before a movement was executed, a rough idea of the robot’s projected path to reach some goal was formulated. This was accomplished in two steps. First, a goal point where the robot would ultimately “like” to get to is determined. Then a path is computed that would result in achieving that position incurring the lowest accumulated cost, as determined by the distance traveled multiplied by the cost of each cell traveled through.

To determine the goal point, the program merely searches for the lowest cost in the cost map. Because the cost map is created such that target areas to tend towards incur a negative cost, and areas to be avoided incur a positive cost, all costs being cumulative, the cell with the lowest cost is the one that places the robot in the most advantageous situation for the scenario in question. In case of multiple cells with equally low costs, all cells sharing the lowest cost were selected as possible goal points.

The calculation of the best path to accomplish movement to this goal point started by selecting several cells as “nodes” likely to be found along the optimal path. Noting that the shortest path around any object is one that carries one close to that object, while also realizing that objects in
the robot soccer environment are not stationary, six nodes were selected at a set “safe” distance around each opposing robot. A node was also created at the robot’s current position, and another at each of its potential target positions.

The task was now to determine which combination of available nodes formed the path of lowest cumulative cost. To check all available paths would require checking $N!$ possible paths, a task which is intensely computationally expensive, even having limited the possible “waypoints” along the path to the nodes selected. Instead, an algorithm similar to Dijkstra’s Algorithm, with several adaptations, was implemented.

To save computation time, each combination of two nodes was checked for “connection”. Two nodes were considered to be connected if along the line between them there was no cell with a cost above a certain cut-off limit. If the cost of even one cell along this line was above this limit, no path utilizing a movement between these two nodes would be considered. The nodes that each node was connected to were stored in a matrix, called “connections”, for later use.

The next step was to assign each potential target node a cost of zero within a separate matrix keeping track of each nodes cost (not to be confused with the cost of the cell the node occupies. This will be explained shortly.) . This matrix was called “cost_from”, as it recorded the cost of each node, and which node it had most recently been moved from in the path check (more on this shortly). All other nodes were given an arbitrary high number. Within a third matrix, known as “checker”, and using the connections already established, all potential “moves” from one of these target nodes to a connected node were recorded.
The program then checked the first potential move in “checker”. The program used a method, borrowed from graphic programs, called “ray tracing” to evaluate the cost of this move. To accomplish this, an array with higher resolution (64x64 cells within each cell of the main array) was “overlaid” on the main array. The location within this fine array at which the line between the nodes in question crossed the boundaries of a cell within the main array were recorded. This allowed for an exact cost to be calculated where the cost was determined by both the costs of the cells along the path, and the length of the line within each cell.

The cost of the move was then added to the cost of the “source” of this move, in this case, the potential goal node in question, a cost of zero, and compared to the current cost of the node at the end of the move. If the cost of the move plus the source node, representing the total cost from this node to a potential goal node, is lower than its current cost, this means that from this node the path being checked from the goal node is shorter than the previous path that would have been followed from the node to a goal node. Therefore, the cost is changed to reflect this new, shortest path. The source node for this move is also recorded in “cost_from” for the target node. This way, a complete path can be traced by going from one node to the source of the shortest path to it, and from there to the source to the shortest path to that node, etc. Lastly, all connecting paths from this altered node are added to “checker” as potential extensions of this shorter path. Having served its purpose, the potential move is erased from checker.

If, however, the cost of the node is currently lower than the cost to move to it, the move is erased from checker with no effect. Either way, the next move in “checker” is then checked.
In this manner, the shortest paths from each of the target nodes branches outwards through the complex of nodes. Where two possible shortest paths collide, the shorter one continues, and the longer one is “killed out”. In this way, each node is generally not checked more than two or three times, reducing the number of paths checked considerably.

As the process continues, less and less movements will result in a cost change, as each node reaches its lowest possible cost. When all nodes are at their lowest values, “checker” empties out, and the loop ends.

At this point, the nodes used for the final path can be determined. Starting at the node where the robot is located, the next node on the path is taken from “cost_from” as the node from which movement to the starting node produced the lowest cost. The next point is then taken from “cost_from” from this node in the same manner. The process continues until a node with a cost of zero is reached, signifying that it is a goal point. It is also the goal point to which the lowest cost is incurred. Thus the function determines the path and the endpoint from among the available options at the same time. This allows for more generic strategizing. For example, when trying to block a shooting lane, it is not necessary to determine in advance which location the robot should move to. Rather, a uniform cost modification can be applied to the entire line. The number of possible goal points will first be reduced by influencing factors, such as the proximity of opponent robots, varying the costs on the path. After this, the program can quickly and efficiently determine the optimal path, even given multiple potential end points.
3.4.1 Example of the path finding algorithm

Fig 3.3 shows a simple scenario with 4 nodes with start node A, destination node D and intermediate nodes B and C. The circles represent nodes being considered as waypoints for the path. The lines are straight paths from one node to another. If there is a line, the two nodes are “connected”. If there is not, they are not “connected” due to a cost existing somewhere on the line between the two nodes that exceeds a set value. Paths between non-connected nodes are not considered. Each line has a cost associated with traveling along it, calculated by combining distances, angles, and the cost map grid spaces that the line passes through.

3 Matrices are involved in the algorithm.

1. “Cost_from” has two entries per list entry (i.e. it is sized nx2). In the first is the cost associated with traveling to that node from the last node. In the second is the node that was last traveled from to reach this node (if there was a cost modification).
2. “Checker” is also an nx2 matrix. It stores the x and y variables of the nodes still to be checked. The x and y coordinates of the last location on the path that you will be adding this node to.
3. “Connect_matrix” stores which nodes are “connected” to which other nodes.
   This matrix doesn’t change value during the path finding operation.

In our example in traveling from A to D, tables 3.1a, 3.1b and 3.1c respectively show the initial values of the cost_from, connect_matrix and checker matrices.
This indicates that the shortest known cost from D to D is 0. There are no known costs for any other path.

Node B is to be added to the path currently ending at D.

Node C is to be added to the path currently ending at D.

Step 1:

Tables 3.2a and 3.2b show the values of the cost_from and checker during step 1.

The first entry in checker has been checked. There is now a new path known. The new lowest path from D to B has a cost of 7. The checked entry is removed. All possible moves from the checked node are added to checker.

Removed: D->B

Added: B->A, B->C, B->D

Step 2:
Tables 3.3a and 3.3b show cost_from and checker during step 2.

The next move in checker has been checked. The new lowest cost to C is 5, moving from C from D.

Removed: D->C

Added: C->A, C->B, C->D

**Step 3:**

Tables 3.4a and 3.4b show cost_from and checker during step 3.

The new lowest cost to A is 16, coming from B. The cost is the current value of B (7) and the added cost of the move (9).

Removed: B->A

Added: A->b, A->C

**Step 4:**

Tables 3.5a and 3.5b show cost_from and checker during step 4.

The path B->C was checked. B had a value of 7. Added to the cost of the move, 2, the new value for C would be 9. But the current value, 5<9, so no change is incurred.
Step 5:

Tables 3.6a and 3.6b show cost_from and checker during step 5.

The cost was checked from B->D. The value was higher than the current value of 0 at D.

Step 6:

Tables 3.7a and 3.7 show cost_from and checker during step 6.

The value from C to A (5 + 6) is less than the previous value of A (16), so the value of A is changed to 11.

Checker:
Removed: C->A
Added: A->B, A->C

Steps 7, 8, 9, 10, 11, 12:

No other changes are made as all nodes are at their minimum value. The entries are systematically removed from checker, leaving it empty, and ending the loop.

The final path can now be read out from the “From” column of Cost_from:
A->C->D

3.5 Path Execution

After determining the nodes to be used as waypoints for the robot’s path, the actual motion of the robot had to be determined. The process for doing this was based off of an online paper. While the algorithm used in that paper was for a static situation, with a little modification, it was modified to suit the needs of a dynamic environment.

The robot’s projected position a few vision refreshes after the time in question, given no change to its velocity, is determined by simply adding several times its velocity to its current position. Using a projection function, as described earlier for use in blocking passes, the distance of this point from a line drawn from the robot’s current position to the first waypoint along its chosen
path is found. If this distance is sufficiently short, no change to the robot’s velocity is implemented. This is to recognize that the path selected will never be exactly optimal in a dynamic environment, and if the robot is sufficiently close to this path, there is no reason to adjust its course unnecessarily.

If, however, the robot’s current course diverges too widely from the selected path, corrective steering is applied. This is done by simply taking the point on the path that is the projection of the robot’s projected position, provided by the projection function, and steering towards it, using one of the available functions provided in the simulator.

If the projection onto the path is beyond the first waypoint, the projection is shifted to the waypoint. Since the environment is dynamic, and thus the nodes constantly changing, the robot should not actually ever reach this point unless it is in fact the goal point it is trying to reach. Thus, by shifting the point the robot aims at to this waypoint, the robot will arrive at its goal point at near-zero velocity, while maintaining smooth motion otherwise.
Chapter 4: Functions

4.1 Main:

As the simulator could not yet be used at the time this report was written, testing had been done by manually inputting values for the position, velocities, and accelerations of moving objects on the pitch. Being prior to the execution of the AI code in actual operation, this was the first task carried out in the main function. The positions of the objects were then offset, using the velocities and accelerations from their current position to simulate the dynamic environment by predicting their future behavior. This would all be carried out separately from the individual robot AI programs that come after this under normal operating conditions. As the last element universal to all robots, the function decides which robot, if any, is considered in possession of the ball.

The function then called the requisite functions to create the individual cost map for the robot in question. This involves deciding on which situation from the virtual playbook created is in effect, and applying the appropriate cost modifications for that situation.

Having completed the cost map generation, the main function calls on the general path finding algorithm, “get_path”.

Sub-functions called by “main”: 
4.1.1 whosepossession:

This function runs through the distances of each robot from the ball. It then declares that the robot closest to the ball is considered to be in possession. If robots from both teams are equally close, the ball is considered to be loose. If future testing warrants it, a loose ball may be declared if no robot is within a certain distance of the ball.

4.1.2 getscenario

“getscenario” runs through the available plays in the virtual playbook. It then decides on which play is currently in effect for the robot in question. In its current form, the function takes in only which robot is in possession of the ball, and chooses from four possible scenarios:

i) The team is on offence, and the robot in question is in possession of the ball
ii) The team is on offence, but a teammate is in possession of the ball
iii) The opposing team has possession of the ball
iv) The ball is loose, with neither team in possession

4.1.3 cost_modification

After a scenario is selected, this function updates the cost modification values for objects of interest on the field. Objects to be avoided are given a high cost modification value, a objects the robot should tend towards are given a low value.
4.1.4 assign_cost

The position and type of each object that incurs a cost modification is passed through this function. It updates the robot’s cost map by applying this modification to cells around the position of the object incurring the cost, with the magnitude of change decreasing with distance.

4.1.5 prioritysetup

This function is the base for the current defensive algorithm. The path between the opponent in possession of the ball and the team’s net is given highest priority, and the closest friendly robot to this shooting lane is assigned to block it, regardless of other considerations. Passing lanes from the robot with the ball to other opponent robots are then assigned to be blocked by the other robots on the team. This function will only be called once per vision refresh in actual operation.

4.1.6 lineblock

After the robot had been assigned a path to block, the cells along that path were given a negative cost modification to encourage travel to that path along an optimal path.

4.1.8 wall

In order to avoid collisions, both for the safety of the robots, and to adhere to the rules, a “wall” of high cost was placed around each robot on the field. This ensured that the robots would never be commanded to travel through, or too close to, another robot.
4.2 get_path

The primary role of this function was to create a set of nodes, or selected cells, from which waypoints along a final, near-optimal, path would be chosen. After inputting the robot’s current position as the starting node, the robot called a sub-function, “design_hex” to create a series of nodes around each “obstacle” on the field, most often opposing players. Nodes were also created at the midpoint between objects, and at each cell in the map sharing the lowest cost, as these were considered as goals for the robot to reach.

Sub-functions called by “get_path”:

4.2.1 Design_Hex

Design_hex takes in the coordinates of an object, and creates a series of 6 nodes, equidistantly spaced, ringing it. These nodes are added to the list of available waypoints to be used in the robot’s planned path.

4.3 Connections:

After the creation of the set of nodes to be considered, each pair of nodes was checked for connection. To be connected, two nodes could not have a single cell lying on the path between them containing a cost above a certain threshold value. While “connections” stored which nodes were connected to each node, another function, “connect” was called to ascertain connectivity. Before “connect” can be properly explained, though, first a function that it calls, “find_intercepts” should be explained.
Sub-functions called by “connections”:

4.3.1 find_intercepts:

Borrowing a technique from graphic programming, find_intercepts created a fine meshed grid, with 64x64 cells per each cell in the main grid, and overlaid it on the main grid. By evaluating which cell in this fine grid a line passed through when it encountered a cell border in the larger grid, the length of the line within the grid space could be computed. As well, the cells on the large grid passed through could also be easily identified.

The function worked by starting at one end of a line, and converting the coordinates in the main grid to a corresponding coordinate in the fine grid. It then moved to the first interface between cells on the main grid encountered along the path and recorded the position in the fine grid. Horizontal interfaces were kept separate from vertical interfaces simply because it was computationally simpler to do so. This process continued until the last grid space on the path was reached.

These intercepts could then be used for two purposes. Firstly, they could readily be converted back to corresponding locations in the main grid, and all cells traveled through on the path could be found (cells above and below a vertical interface were on the path, as were cells to the left and right of a horizontal interface.) As well, the distances between interface encounters could be determined for accurate scoring of a path’s cost.
4.3.2 connect:

After calling find_intercepts, this function merely checked all cells lying on a particular path to determine if any had a cost above a certain threshold. If any did, the path was invalid for path consideration, and a failure of connection was noted. If no failure occurred, the nodes at either end of the path were connected.

4.4 least_path:

Least path was the primary function involved in path selection. It was an adaptation of the well known Dijkstra’s algorithm, and was used to directly calculate the optimal path, restricted in that it could only choose paths created by moving from one selected node to another, without incurring overly much computation time.

The algorithm used three separate matrices to accomplish its task. One of these, “cost_from” kept track of both the cost of each node, calculated as the currently known least cost incurred by travel to any of the potential goal points, and the path taken to reach said end point (by tracking which node precedes each node along the path). At the outset of computation, each potential end point had a value of 0, as each was already at an end point, and all other points had arbitrary high values, representing that no known path to the end point existed from them. Another matrix used was the connect_matrix, formulated in “connections” that tracked what movements were available from each node. Lastly, there was “checker” that stored the moves the algorithm wishes to check next for potential low-cost paths.
The algorithm started by feeding in all the connections from each potential end node to “checker”. The algorithm then checked each of these possible moves, one at a time. If the movement incurred a lower cost in the node being moved to (computed as the total cost to reach an end node) than it contained previously, a new, shorter, path had been found from that node to an end point. The nodes values were changed in “cost_from” and potential movements from that node were fed into “checker” for future testing. The move executed was then removed from checker. If the cost contained in the node was lower than the new cost incurred by the potential path, the move was eliminated from “checker” with no effect.

Paths produced in this manner would branch out from the end points in all directions as paths to each node would be discovered. Eventually these paths would start colliding, and the costlier paths would be killed off, while the cheaper paths would continue to propagate. For this reason, multiple potential end points could be considered with little extra computation time required. In short order, all nodes would contain their lowest possible cost. When this occurred, checker would empty out, and the algorithm would cease.

The optimal path for the robot, chosen from all possible paths that could be formed by combining nodes, could then be extracted from “cost_from”. By starting the program from the potential end points, rather than the starting point, there was no need to search through the costs for a low cost path. Rather, the path could directly be traced from the start node, as it’s current value and path are that which ends at one of the potential end nodes along the cheapest path to any of the end nodes.
Sub-functions called by “least_path”:

4.4.1 get_score:

Get_score was used to determine the cost of moving from one node to another. To do this, it merely used the intercepts found from “find_intercepts” to compute how long a line segment for the path lied in each cell. The distance of this segment was then multiplied by the cost of the cell, and all such costs along the path were summed for a total cost of the move.

If a cost was negative, as negative cells do exist in this program, the cost was merely the distance traveled. This was done to avoid retracing a path, and in fact infinite recursion, and thus no solution that would result in a negatively weighted node movement.

4.5 path_follow

Having determined the waypoints for the final path, the steering behaviour of the robot had to be determined. This was done in the manner described in Section 3.4.
Chapter 5: Results

5.1 Effectiveness

The primary test of the program’s effectiveness, prior to competition, was to be a simulator created for the soccer program by members of UTRA. After sufficient testing in the simulator, real Miabots could be used. However, the member responsible for the simulator had been out of contact for some time, and there seemed to be some bugs in the simulator that could not be overcome without his expertise. Thus, the simulator was not used. Instead situations were created where certain responses were expected, and the program was run to see how it matched up to expectations.

An example of a defensive situation is illustrated in fig 5.1. The cost map shown is only for one of the robots, but serves to illustrate the layout of the field. As well, the size is far smaller than would be used in a competition situation, but was chosen for illustrative purposes, as larger fields would not fit on the page without wrapping around.

Red Circles indicate opponent robots, with the robot at 15,10 in control of the ball. The team’s net is centered behind 0,17. Velocities and accelerations were set to zero, to ease evaluation of the situation. Red lines indicate passing and shooting lanes for the robot with the ball, and blue lines indicate the paths generated for the defending robots. The robot paths take them quickly and efficiently into the passing and shooting lanes far enough away from the opponent robots that maneuvering around them would not be easy.
Similar tests were run with the robots in various positions, and the program’s defensive strategy was shown to appear to be sound.

A fully functional scheme for offense would require physical testing with the robots themselves to determine the limitations and momentum provided by robots pushing on the ball. However, some initial theoretical testing was carried out.

For example, given that the robots have no dribbling mechanism as yet, a scheme where the robots converge together towards the goal, bumping the ball in that direction, while simultaneously avoiding opponents, might be effective. The converging behavior shown in fig 5.2, with the robot at 0,18 in possession of the ball, was achieved by adjusting the cost modifications of objects on the field:

Some interesting behavior results, such as the robot at 13,20 looping back around to keep close to the robot with the ball before converging with the others. In this situation, this improves the chance for an effective pass to this robot, as well as likely drawing out the defender at 15,30, allowing a pass through to the robot at 20,30.

Alternately, it might be productive to have the robots spread out some to increase opportunities for passing, and force the defense to split up their players. The behavior displayed in fig 5.3 was modified from the earlier case by merely introducing a positive cost around team members not in possession of the ball. It should be noted that while the paths appear to converge, this is only due to the wide spacing of the robots, and their eventual trend towards the opponent net. If two
robots approach one another, they will tend to diverge once more. Fig 5.4 shows the same scenario as it might be shortly afterwards.

Note the continued wide spacing of the robots. Also of note is that while the path of the ball-carrying robot is very different than it was previously, as the defenders are moving to close the gap through the middle, most of the other robots are following similar paths as they were before, allowing easy path execution.

One more test concerning effectiveness involved the choice of nodal placements. The program’s generated paths were compared to a program that had a grid of evenly spaced nodes throughout the field added to the regular set of nodes. While this many nodes was infeasible in practice, it likely generated a very near-optimal path, so was useful for comparison.

Using the behavior for diverging behavior, the paths generated were identical to those generated by the original program. In other cases, the paths diverged slightly, but in all tests, the paths followed closely. This suggests that the node creation scheme was valid.

It should also be noted that adding nodes between objects, as we did, rather than just in proximity to them, while intuitively seeming to add value, did not produce different results than without them in almost every test conducted.

5.2 Time

The run-time of the program was computed by running a clock function in C.
It was found that the generation of each cost map, followed by the path finding and path execution, took 20 clocks. 

CLOCKS_PER_SEC returned a value of 1000 clocks per second, and this was confirmed by testing (the debugger was run between recordings of the clock functions, and 30 seconds took approximately 31 000 clocks). This means the function ran at a frequency of about 52 Hz or 19.52 ms. The cumulative cost for five robots would be approximately 96.15 ms, which is considerably greater than the vision refresh rate of 11 ms. This is far too slow, even given that certain parts of the code would not be repeated for each robot, so the time would be somewhat faster. However, this analysis was performed on the relatively slow ECF computers. Using a dedicated computer for the program might result in usable results. As well, much of the time taken could easily be shaved off by simple coding practices. For example, it is almost certain that a large portion of the time taken was due to a large number of square root functions computed for distances. Pre-computing the required distances, and storing them in a matrix for permanent use would probably cut the time down considerably. With proper coding, the AI system should run fast enough for effective use, although this will require further testing.
Chapter 6: Conclusion & Recommendations

6.1 Conclusion

After a few more or less failed starts by UTAIR and later UTRA to introduce a robot soccer program into U of T, this year, via collaborative efforts with UTRA, that goal may now be a reality. Through the course of the year, a nearly complete AI system has been developed to complement the vision software completed by UTRA last year.

The program has been shown to produce results commensurate with expectations. It is capable of taking in the vision software, choosing a scenario from a virtual playbook, and then carrying it out by choosing individual actions for each team member.

Individual behaviours are easily handled as is the slightly more advanced roled play, used in defensive situations. The full team dynamic required for efficient offensive play with passing has not yet been fully developed.

Some testing has been done, but actual simulations and real-life play has not yet been implemented. There are also issues with the processing time, as well as incompleteness of the offensive strategy, as mentioned.

Despite these issues, it is more than possible that the program will be ready to be competitive in the MiRosot soccer tournament this year. Regardless of the stage the program is in when on the
day of the tournament, the program can be used by future generations at UTRA to work with, study, and build upon.

**6.2 Recommendations**

As a stated goal of this thesis was to develop a platform for future development by members of the UTRA club, clearly not all possible avenues of this development have been thought of. So the discussion of future development will involve two parts: the features that were projected to ease future development, and some ideas about possible ideas for this development.

Perhaps the most obvious feature allowing for improvement is the cost map system being used. Using this format, a “playbook”, one which may well become quite involved, can be developed by recognizing various situations and modifying the map accordingly. Thus a programmer might identify a situation where a defending player should make a fast break towards the opponents net, predicting a turnover in the near future. The cost map would then reflect this by reducing the cost in the opponent zone and increasing the cost in the defensive zone. Or perhaps a screening maneuver will be called for, in which case an attacking robot can cut off a path towards the robot with the ball, much as defending robots currently cut off passing lanes.

The cost map also allows for some easy optimization. The relative cost modifications can be shifted, and the effects noted in various situations. The cost modifications could depend on certain factors, such as the distance between robots.
The cost map and node creation are also fairly obvious candidates for some more advanced programming techniques, such as evolutionary computing. Rather than figure out all the possibilities by trial and error or direct calculation, an evolutionary approach can be used.

The genome would simply be the combination of cost map and node placement schemes. Simple experiments might include the same scenarios already described, but the costs for each object might be modified, or the manner in which the cost modifications decrease with distance (perhaps weighting to one side, or merely making the incremental change more or less), or the number of nodes around each object changed. More advanced experiments could have the computer develop its own playbook, where various objects are given random cost modifications based on random criteria. This is made feasible by the limited number of types of objects on the field.

The survival criteria would also be simple. The programs would merely be pitted against each other in the simulator, perhaps for five minutes per game, and fitness evaluated by the difference in score. Given a number of computers, many runs could be computed in a short time.

Breeding would also be relatively straight-forward, as only selected sections of code would be interchanged. For example, the costs of various objects might be what is changed, or the number or placement of nodes. Or the contents of an “if” statement for when a function is called to apply certain costs. Each of these can be readily identified in the code, so that they can be interchanged and mutated without causing the program to crash.
References


Other sources:

Christopher Lampton, Gardens of Imagination, (The Waite Group, Inc. copyright 1994)

LaMothe, Ratcliff, Seminatore & Tyler, Tricks of the Game Programming Gurus (Sams Publishing, copyright 1994)

Andre Lamothe, Black art of 3D game programming, (Waite Group Inc. copyright 1995)


Chapter 7: Figures and Tables

7.1 Figures

Fig 1.1 A typical small sized robot game.

Fig 1.2 Small sized competition pitch dimensions
Fig 2.1 Potential field method problem

Fig 2.2 Tangent planning method
Fig 3.1 AI Program Overview
Fig 3.2 Sample cost map.
The first column of numbers on the left shows the y Cartesian coordinates. The first row of numbers on the top shows the x Cartesian coordinates. All the other numbers represent cells defined by these coordinates and display the costs of these cells.

Fig 3.3 Sample scenario for pathfinding algorithm
Fig 5.1 Defensive situation
Fig 5.2 Converging behavior
Fig 5.3 Converging behavior.
Fig 5.4 Diverging behavior continued
### 7.2 Tables

#### Table 3.1a: Initial values of cost_from.

<table>
<thead>
<tr>
<th>Node</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9999</td>
</tr>
<tr>
<td>B</td>
<td>9999</td>
</tr>
<tr>
<td>C</td>
<td>9999</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Table 3.1b: Initial values of connect_matrix

<table>
<thead>
<tr>
<th>Node</th>
<th>Connected to</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C</td>
</tr>
<tr>
<td>B</td>
<td>A, C, D</td>
</tr>
<tr>
<td>C</td>
<td>A, B, D</td>
</tr>
<tr>
<td>D</td>
<td>B, C</td>
</tr>
</tbody>
</table>

#### Table 3.1c: Initial values of the checker matrix.

<table>
<thead>
<tr>
<th></th>
<th>B_x</th>
<th>D_x</th>
<th>C_x</th>
<th>D_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>B_y</td>
<td>D_y</td>
<td>C_y</td>
<td>D_y</td>
</tr>
</tbody>
</table>

Table 3.1c: Initial values of the checker matrix.
<table>
<thead>
<tr>
<th>Node</th>
<th>Cost</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9999</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>9999</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2a: cost_from during step 1.

<table>
<thead>
<tr>
<th>Node</th>
<th>Cost</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9999</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3a: cost_from during step 2.
<table>
<thead>
<tr>
<th>X</th>
<th>A_x</th>
<th>B_x</th>
<th>C_x</th>
<th>B_x</th>
<th>D_x</th>
<th>B_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>A_y</td>
<td>B_y</td>
<td>C_y</td>
<td>B_y</td>
<td>D_y</td>
<td>B_y</td>
</tr>
</tbody>
</table>

Table 3.3b: checker during step 2.

<table>
<thead>
<tr>
<th>Node</th>
<th>Cost</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4a: cost_from during step 3.

<table>
<thead>
<tr>
<th>X</th>
<th>C_x</th>
<th>B_x</th>
<th>D_x</th>
<th>B_x</th>
<th>A_x</th>
<th>C_x</th>
<th>B_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>C_y</td>
<td>B_y</td>
<td>D_y</td>
<td>B_y</td>
<td>A_y</td>
<td>C_y</td>
<td>B_y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>C_x</th>
<th>D_x</th>
<th>C_x</th>
<th>C_x</th>
<th>A_x</th>
<th>B_x</th>
<th>A_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>C_y</td>
<td>D_y</td>
<td>C_y</td>
<td>C_y</td>
<td>A_y</td>
<td>B_y</td>
<td>A_y</td>
</tr>
</tbody>
</table>

Table 3.4b: checker during step 3.
<table>
<thead>
<tr>
<th>Node</th>
<th>Cost</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.5a: cost_from during step 4.**

<table>
<thead>
<tr>
<th>X</th>
<th>D_x</th>
<th>B_x</th>
<th>A_x</th>
<th>C_x</th>
<th>B_x</th>
<th>C_x</th>
<th>D_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>D_y</td>
<td>B_y</td>
<td>A_y</td>
<td>C_y</td>
<td>B_y</td>
<td>C_y</td>
<td>D_y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>C_x</th>
<th>C_x</th>
<th>A_x</th>
<th>B_x</th>
<th>A_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>C_y</td>
<td>C_y</td>
<td>A_y</td>
<td>B_y</td>
<td>A_y</td>
</tr>
</tbody>
</table>

**Table 3.5b: checker during step 4.**

<table>
<thead>
<tr>
<th>Node</th>
<th>Cost</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.6a: cost_from during step 5.**
<table>
<thead>
<tr>
<th>Node</th>
<th>Cost</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6b: checker during step 5.

<table>
<thead>
<tr>
<th>X</th>
<th>A_x</th>
<th>C_x</th>
<th>B_x</th>
<th>C_x</th>
<th>D_x</th>
<th>C_x</th>
<th>C_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>A_y</td>
<td>C_y</td>
<td>B_y</td>
<td>C_y</td>
<td>D_y</td>
<td>C_y</td>
<td>C_y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>A_x</th>
<th>B_x</th>
<th>A_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>A_y</td>
<td>B_y</td>
<td>A_y</td>
</tr>
</tbody>
</table>

Table 3.7a: cost_from during step 6.

<table>
<thead>
<tr>
<th>X</th>
<th>B_x</th>
<th>C_x</th>
<th>D_x</th>
<th>C_x</th>
<th>C_x</th>
<th>A_x</th>
<th>B_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>B_y</td>
<td>C_y</td>
<td>D_y</td>
<td>C_y</td>
<td>C_y</td>
<td>A_y</td>
<td>B_y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>A_x</th>
<th>C_x</th>
<th>A_x</th>
<th>B_x</th>
<th>A_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>A_y</td>
<td>C_y</td>
<td>A_y</td>
<td>B_y</td>
<td>A_y</td>
</tr>
</tbody>
</table>

Table 3.7b: checker during step 6.
Appendix A: Commented C Code

Libraries, Definitions, Global Matrices, and Declarations

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#define INTER 500 //size of intercept matrices
#define SIZE1 40 //220 //one axis of playing field
#define SIZE2 35 //180 //the other axis
#define ENEMIES 5 //number of opponent robots
#define CHECKSIZE 13000 //size of checker used in pathfinding algorithm
#define NODES 100//number of possible waypoints along the robot's path
#define RAD 5 //distance nodes are set from opponent robots
#define NOCOSTVALUE 6 //max allowable cost for any space on path
#define TARGETSIZE 1000 //number of highest possible end point target node
#define ROBOTS 5 //number of robots on our team
#define SIZE 1 //the radius of the wall around each robot's center to prevent collisions
#define LOOKAHEAD 3 //number of velocity cycles to look ahead to see if off path
#define VARIATION 2 // acceptable dist off path

int get_path (int start_x, int start_y, /*int end_x, int end_y,*/ int cost_map[][SIZE2], int enemy_positions[][2], int finalpath[][2]);
int design_hex(int nodes[32][2], int center_x, int center_y, int rad, int i);
float find_intercepts(int x_intercepts[][2], int y_intercepts[][2],int x1, int y1, int x2, int y2);
int connect (int cost_map[][SIZE2], int x_intercepts[][2], int y_intercepts[][2],int x1, int y1, int x2,int y2);
int connections (int nodes[][2], int connect_matrix[][NODES], int cost_map[][SIZE2], int x_intercepts[][2], int y_intercepts[][2],int max_nodes);
int leastpath (int finalpath[][2], int nodes[NODES][2], int cost_map[][SIZE2], int x_intercepts[][2], int y_intercepts[][2],int connect_matrix[][NODES],int targets, int target_nodes[TARGETSIZE]);
double get_score (float slope, int x1, int y1, int x2, int y2,int cost_map[][SIZE2], int x_intercepts[][2], int y_intercepts[][2]);
void lineblock (int xa, int ya, int xb, int yb, int cost_map[][SIZE2]);

int assign_cost (int value, int target_x, int target_y, int cost_map[][SIZE2]);

int simpleprojection(int, int,int,int,int);
void prioritysetup(void);

int getscenario(int robot);
int cost_modification(int scenario);
void whosepossession();
void wall (int friendly_positions[][2], int enemy_positions[][2], int cost_map[][SIZE2], int robot);

// global matrices for which robot is in possession of the ball and the various costs of objects on
// the field
int hasBall[(ROBOTS + ENEMIES)], robotcost[10];

// global position matrices for positions, velocities and accelerations
int enemy_positions[ENEMIES][2], friendly_positions[ROBOTS][2], ball_position[2];
int enemy_velocities[ENEMIES][2], friendly_velocities[ROBOTS][2], ball_velocities[2];
int enemy_accelerations[ENEMIES][2], friendly_accelerations[ROBOTS][2],
bball_accelerations[2];

// global priority matrix for use in defensive costmap creation
int prioritymatrix[ROBOTS][2];

clock_t begin;
clock_t finish;
double total;
Main Function

void main()
{
    int start_x, start_y, /*end_x, end_y,*/ i, j, robot, scenario;
    int cost_map[SIZE1][SIZE2], finalpath[NODES][2];
    //first fill in data that will be provided by vision in actual operation
    //set enemy positions
    enemy_positions[2][0]=18;
    enemy_positions[2][1]=15;

    enemy_positions[1][0]=15;
    enemy_positions[1][1]=10;

    enemy_positions[0][0]=18;
    enemy_positions[0][1]=24;

    enemy_positions[3][0]=30;
    enemy_positions[3][1]=28;

    enemy_positions[4][0]=31;
    enemy_positions[4][1]=14;
    // enemy_positions[5][0]=15;
    // enemy_positions[5][1]=5;

    //set enemy velocities (x,y)
    enemy_velocities[2][0]=0;
    enemy_velocities[2][1]=0;

    enemy_velocities[1][0]=0;
    enemy_velocities[1][1]=0;

    enemy_velocities[0][0]=0;
    enemy_velocities[0][1]=0;

    enemy_velocities[3][0]=0;
    enemy_velocities[3][1]=0;
enemy_velocities[4][0]=0;
enemy_velocities[4][1]=0;

// enemy_positions[5][0]=15;
// enemy_positions[5][1]=5;

//sets enemy accelerations (x,y)

    enemy.accelerations[2][0]=0;
    enemy.accelerations[2][1]=0;
    enemy.accelerations[1][0]=0;
    enemy.accelerations[1][1]=0;
    enemy.accelerations[0][0]=0;
    enemy.accelerations[0][1]=0;
    enemy.accelerations[3][0]=0;
    enemy.accelerations[3][1]=0;
    enemy.accelerations[4][0]=0;
    enemy.accelerations[4][1]=0;

//sets friendly positions

    friendly_positions[0][0] = 12;
    friendly_positions[0][1] = 19;
    friendly_positions[1][0] = 12;
    friendly_positions[1][1] = 10;
    friendly_positions[2][0] = 22;
    friendly_positions[2][1] = 10;
    friendly_positions[3][0] = 27;
    friendly_positions[3][1] = 33;
    friendly_positions[4][0] = 32;
    friendly_positions[4][1] = 19;

//set friendly velocities (x,y)
friendly_velocities[2][0]=0;
friendly_velocities[2][1]=0;

friendly_velocities[1][0]=0;
friendly_velocities[1][1]=0;

friendly_velocities[0][0]=0;
friendly_velocities[0][1]=0;

friendly_velocities[3][0]=0;
friendly_velocities[3][1]=0;

friendly_velocities[4][0]=0;
friendly_velocities[4][1]=0;

//friendly_positions[5][0]=15;
//friendly_positions[5][1]=5;

//sets friendly accelerations (x,y)

friendly_accelerations[2][0]=0;
friendly_accelerations[2][1]=0;

friendly_accelerations[1][0]=0;
friendly_accelerations[1][1]=0;

friendly_accelerations[0][0]=0;
friendly_accelerations[0][1]=0;

friendly_accelerations[3][0]=0;
friendly_accelerations[3][1]=0;

friendly_accelerations[4][0]=0;
friendly_accelerations[4][1]=0;

//sets ball position
ball_position[0] = 12;
ball_position[1] = 20;

//set ball velocity
ball_velocities[0] = 0;
ball_velocities[1] = 0;
//set ball acceleration
ball_accelerations[0] = 0;
ball_accelerations[1] = 0;

//offset various objects to simulate dynamic environment
//ensures the new positions remain on the playing field
for (i=0; i<(ROBOTS); i++)
{
    friendly_positions[i][0] += friendly_velocities[i][0] +
    friendly_accelerations[i][0];
    if (friendly_positions[i][0] <0) friendly_positions[i][0] =0;
    if (friendly_positions[i][0] >SIZE1) friendly_positions[i][0] =SIZE1;

    friendly_positions[i][1] += friendly_velocities[i][1] +
    friendly_accelerations[i][1];
    if (friendly_positions[i][1] <0) friendly_positions[i][1] =0;
    if (friendly_positions[i][1] >SIZE2) friendly_positions[i][1] =SIZE2;

    enemy_positions[i][0] += enemy_velocities[i][0] +
    enemy_accelerations[i][0];
    if (enemy_positions[i][0] <0) enemy_positions[i][0] =0;
    if (enemy_positions[i][0] >SIZE1) enemy_positions[i][0] =SIZE1;

    enemy_positions[i][1] += enemy_velocities[i][1] +
    enemy_accelerations[i][1];
    if (enemy_positions[i][1] <0) enemy_positions[i][1] =0;
    if (enemy_positions[i][1] >SIZE2) enemy_positions[i][1] =SIZE2;
}

ball_position[0] += ball_velocities[0] + ball_accelerations[0];
if (ball_position[0] <0) ball_position[0] =0;
if (ball_position[0] >SIZE1) ball_position[0] =SIZE1;

if (ball_position[1] <0) ball_position[1] =0;
if (ball_position[1] >SIZE2) ball_position[1] =SIZE2;

//robot is the robot we are creating the map for
robot= 3;

//testing time
begin = clock();

//switched 0 and 1
start_x=friendly_positions[robot][1];
start_y=friendly_positions[robot][0];

//   end_x=33;
//   end_y=35;

//gives one robot the ball for testing
/*
   hasBall[6] =1;
   hasBall[7] =hasBall[8] =hasBall[9] =0;
*/

whoseposession();

for (i=0; i<SIZE1; i++)
{
   for (j=0; j<SIZE2; j++)
   {
      cost_map[i][j]=2;
   }
}

scenario = getscenario(robot);

cost_modification(scenario);

if (scenario ==5)
{
   prioritysetup();

   if (prioritymatrix[robot][1]==100)
   {
      lineblock
      (enemy_positions[prioritymatrix[robot][0]][1],enemy_positions[prioritymatrix[robot][0]][0],
      SIZE2/2, 0, cost_map);
   }
}
for (i=0; i<ENEMIES; i++)
      */
      else {
        lineblock
        (enemy_positions[prioritymatrix[robot][0]][1],enemy_positions[prioritymatrix[robot][0]][0],
          enemy_positions[prioritymatrix[robot][1]][1], enemy_positions[prioritymatrix[robot][1]][0],
          cost_map);
      }
  }

  // feed opponent positions into cost map
  for (i=0; i<ENEMIES; i++)
  {
    assign_cost (robotcost[3]/15*, enemy_positions[i][0] , enemy_positions[i][1],
          cost_map);
  }

  //feed teammate positions into cost map
  for (i=0; i<ROBOTS; i++)
  {
    if (i==robot) i++;
    assign_cost (robotcost[2], friendly_positions[i][0] , friendly_positions[i][1], cost_map);
  }

  //for ball
  assign_cost (robotcost[0], ball_position[0], ball_position[1], cost_map);

  //for own net
  assign_cost (robotcost[4], 0, SIZE2 / 2, cost_map);

  //for opponent net
  assign_cost (robotcost[1], SIZE2 - 1, SIZE1 / 2, cost_map);

  wall (friendly_positions,enemy_positions, cost_map, robot);

  /*
  // display the cost map to check
  printf(" 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
    \n");
  */
  for (i=0;i<SIZE1;i++)
Sub functions called in Main:

Whosepossession

//decides which robot is in possession of the ball
void whosepossession ()
int ourdist = 100000, theirdist = 100000, dummy = 0;
int whichour, whichtheir;
int i;

for (i = 0; i < (ROBOTS + ENEMIES); i++) {
    hasBall[i] = 0;
}

for (i = 0; i < ROBOTS; i++) {
    dummy = ((ball_position[0] - friendly_positions[i][0])*(ball_position[0] -
    friendly_positions[i][0])) +
            (ball_position[1] - friendly_positions[i][1])*(ball_position[1] -
    friendly_positions[i][1]);
    if (dummy <= ourdist) {
        ourdist = dummy;
        whichour = i;
    }
}

for (i = 0; i < ENEMIES; i++) {
    dummy = ((ball_position[0] - enemy_positions[i][0])*(ball_position[0] -
    enemy_positions[i][0])) +
            (ball_position[1] - enemy_positions[i][1])*(ball_position[1] -
    enemy_positions[i][1]);
    if (dummy <= theirdist) {
        theirdist = dummy;
        whichtheir = i;
    }
}

if (ourdist < theirdist) hasBall[whichour] = 1;
else if (theirdist < ourdist) hasBall[whichtheir + ROBOTS] = 1;
Getscenario

//decides which scenario from a virtual playbook each robot is in
int getscenario(int robot){
    int i, a,l;
    int WehaveBall = 0, EnemyhasBall = 0, j, k, hasBallFlag = 0, scenario = 0;

    /*
     for (i=0; i<10; i++)
     {
     hasBall [i]=0;
     }
     
     // giving robot 1 the ball for testing purposes
     hasBall[7] = 1;
    */
    //hasBall should be a global matrix

    /* Figure out if we have the ball or enemies have the ball */

    //moved from last loop to first to avoid resetting if someone has the ball
    for (l = 0; l < 10; l++)
    {
        if (hasBall[l] == 0)
        {
            WehaveBall = 0;
            EnemyhasBall = 0;
        }
    }

    for (i = 0; ((i < 5) && (hasBallFlag == 0)); i++)
    {
        if (hasBall[i] == 1)
        {
            WehaveBall = 1;
            EnemyhasBall = 0;
            hasBallFlag = 1;
            break;
        }
    }
//changed program, runs equiv to check for our robots above to check if they have ball
for (i = 5; ((i < 10) && (hasBallFlag == 0)); i++)
{
    if (hasBall[i] == 1)
    {
        WehaveBall = 0;
        EnemyhasBall = 1;
        hasBallFlag = 1;
        break;
    }
}

/*
if (hasBallFlag == 0)
{
    WehaveBall = 0;
    EnemyhasBall = 1;
}
*/

/*/  
/*
printf("\n i = %d\n", i);
printf("\nWehaveBall = %d\n", WehaveBall);
printf("\nEnemyhasBall = %d\n", EnemyhasBall);
*/

if (WehaveBall == 1)
{
    /* Robot 0 to 4 = us, Robot 5 - 9 = enemy
        Robot 0 and 1 are attackers, 2 and 3 are defenders, 4 is goalie*/

    //removed defender and attacker roles
    /*
    if ((robot == 0) || (robot == 1)) {
        if (hasBall[robot]) scenario = 1;
        else scenario = 2;
    }
    */

    else { /*
        if (hasBall[robot]) scenario = 1;
        else scenario = 2;
        //}
    }
}
else if (EnemyhasBall == 1)
scenario = 5; /* scenario 5: all go to enemy or ball*/

else if ((WehaveBall == 0) && (EnemyhasBall == 0))
{
  for (k = 0; k < 5; k++)
  {
    if ((hasBall[k] == 0) && (k == 0) || (k == 1))
      scenario = 6; /* scenario 6: all go to ball */
  }
  scenario = 6;
}

return scenario;

Cost Modification

// determines the cost mods of various objects based on the scenario
int cost_modification(int scenario) {

  /* Attacker_1 and Attacker_2 are attacking, goalscoring robots. */
Defender_1 and Defender_2 are defensively oriented robots.

The 10 size arrays store cost modifications for particular objects.

```
// Local variables
int i;

// Resetting all cost modifications to 0
for (i = 0; i < 11; i++) {
    robotcost[i] = 0;
}
```

```
switch (scenario) {
    case 1:
        robotcost[1] = -15; // Very strong inclination for enemy goal
        robotcost[2] = -5; // Inclination from team mates
        robotcost[5] = +5; // Weak aversion to edges
        robotcost[3] = +10; // Very strong aversion to opponents
        robotcost[0] = cost modification for ball
        robotcost[1] = cost modification for enemy goal
        robotcost[2] = cost modification for team-mates
        robotcost[3] = cost modification for all opponents
        robotcost[4] = cost modification for own team goal
        robotcost[5] = cost modification for edges
        robotcost[6] = cost modification for midline
        area costs:
        Keep in mind that these are cost modifications, and not the costs themselves.
```
break;

case 2:

//For the attacker without the ball
robotcost[1] = -5; //Strong inclination for enemy goal
robotcost[0] = -5; //Relatively weak inclination for ball
robotcost[3] = +10; //Aversion to opponents
robotcost[5] = +10; //Aversion to teammates to induce spreading

break;

/* reduced complexity. Not needed with current setup

 case 3:

 //For the defender with the ball
 robotcost[1] = -15; //Strong inclination for enemy goal
 robotcost[5] = -12; //Strong inclination for team-mates
 robotcost[3] = +20; //Very strong aversion to opponents

 break;

 case 4:

 //For the defender without the ball
 robotcost[3] = -12; //Strong inclination for opponents

 break;

 */

//Opponents have the ball

 case 5:

 //Attackers and defenders

 // robotcost[0] = -20; //Very strong inclination for ball
 robotcost[3] = 5; //some aversion to opponents
 robotcost[4] = -5; //Relatively weak inclination for own goal

 break;

 //modified to reflect use of priority, lineblock etc. functions

 // Loose ball!
case 6:

    // Attackers
    robotcost[0] = -30; // Very very strong inclination for ball

    break;

}

return 0;

}

Assign Cost

// assigns cost modifications in decreasing magnitude
// with distance
int assign_cost (int value, int target_x, int target_y, int cost_map[][SIZE2])
{
    int i, j;
    double distance, system;

    for (i=0; i<SIZE1; i++)
    {
        for (j=0; j<SIZE2; j++)
        {
            // determines the distance from the cost_map space to the "target"
            distance = sqrt((target_x-j)*(target_x-j)+(target_y-i)*(target_y-i));
            // the system that relates the distance to the cost
            system = 1.5/(distance+1);
            // assigns a cost based on the target value and the distance
            cost_map[i][j] += system*value;
            if ((system *value) < 0) cost_map[i][j] += 1;
        }
    }

    return 0;
}
Priority Setup

// sets the target passing/shooting lane to block
// for each robot
void prioritysetup() {

    int i, j;
    int dist = 100000, z = SIZE2/2;
    int dummy = 0;
    int whichenemy, whichfriendly, passfriendly, passenemy;

    int usedenemy[ENEMIES];
    int usedfriendly[ROBOTS];

    for (i = 0; i < ENEMIES; i++) {
        usedenemy[i] = 0;
        usedfriendly[i] = 0;
    }

    // Finds the enemy closest to the ball
    for (i = 0; i < ENEMIES; i++) {
        dummy = (enemy_positions[i][0] - ball_position[0]) * (enemy_positions[i][0] - ball_position[0]) +
                (enemy_positions[i][1] - ball_position[1]) * (enemy_positions[i][1] - ball_position[1]);
        if (dummy <= dist) { whichenemy = i; dist = dummy; }
    }

    // Finds our team's robot which is closest to line between whichenemy and the goal
    dummy = 0;
    dist = 1000000;
    for (i = 0; i < ROBOTS; i++) {
        dummy = simpleprojection(enemy_positions[whichenemy][0],
                                  enemy_positions[whichenemy][1], 0, z,
                                  enemy_positions[i][0], enemy_positions[i][1], 0, z,
                                  0);
    }
}
friendly_positions[i][0], friendly_positions[i][1]);
    if (dummy <= dist) {whichfriendly = i; dist = dummy; }
}

prioritymatrix[whichfriendly][0]= whichenemy;
prioritymatrix[whichfriendly][1] = 100;
usedenemy[whichenemy] = 1;
usedfriendly[whichfriendly] = 1;

//Assigns the remaining robots to cover passes

dummy = 0;
dist = 100000;
for (i = 0; i < ROBOTS; i++) {
    if (!usedfriendly[i]) {
        for (j = 0; j < ENEMIES; j++) {
            if (!usedenemy[j]) {
                dummy =
simpleprojection(enemy_positions[whichenemy][0], enemy_positions[whichenemy][1],
enemy_positions[j][0],enemy_positions[j][1],
friendly_positions[i][0],
friendly_positions[i][1]);
                if (dummy <=dist) {
                    passfriendly = i;
passenemy = j;
dist = dummy;
                }
            }
        }
    }
}

prioritymatrix[passfriendly][0]=whichenemy;
prioritymatrix[passfriendly][1] =passenemy;
usedfriendly[passfriendly]=1;
usedenemy[passenemy]=1;
dummy = 0;
dist = 100000;

// Line blocking - assigns low costs along the line the robot wants to block.

// Lineblock recieves the x and y (xa,ya and xb,yb) coordinates of the two target points
// and also recieves the costmap array.
void lineblock (int xa, int ya, int xb, int yb,
   int cost_map[][SIZE2]) {

   // X and Y are the loop variables.
   // x1 is the point to the left of x2.

   int x,y, x1, x2, y1, y2, i, j;

   // Costmaptrue is an array set to ensure that one cost map
   // cell along the line is not modified twice.
   int costmaptrue[SIZE1][SIZE2];

   // m is the slope of the line.
   double m;

   double x1double, y1double, x2double, y2double;

   // Direction is the variable that shows if the line has a negative
   // or positive slope.
int direction = 1;

// costmaptrue is set to 0, all cells
for (i=0; i<SIZE1; i++)
{
    for (j=0; j<SIZE2; j++)
    {
        costmaptrue [i][j]=0;
    }
}

i=0;

j=0;

// The following lines checks out of xa and xb, which is lower.
// Then x1 is designated is the leftmost point and is given the corresponding
// y variable. For example, if the leftmost point is xa, then x1 = xa
// y1 = ya.
// Then the direction is checked. If the left point has the higher y value
// then the line slope is set to negative, otherwise it is left as positive.
if (xa <= xb) {
    x1 = xa;
    y1 = ya;
    x2 = xb;
    y2 = yb;

    if (y1 > y2) direction = 0;
}
else {
    x1 = xb;
    y1 = yb;
    x2 = xa;
    y2 = ya;

    if (y1 > y2) direction = 0;
}

// The slope value is set. Note that this is a double, whereas all the values
// it is based on are ints.

x1double = x1;
double y1double = y1;
ex2double = x2;
y2double = y2;
m = (y2double - y1double)/ ((x2double - x1double) + .01);

// Now we loop along the x-axis.
// If a cell is found to have the line in it, the value of that cell
// is modified. Note that since the slope is a double, the integer x value of
// that we find in the cell will be truncated.
// Some cells that contain the line will be skipped - this will be taken care
// of in the y-axis loop.
for (x = x1; x <= x2; x++) {
    y = m*(x-x1) + y1;
    cost_map[x][y] += -10;

    if (!costmaptrue[x][y]) cost_map[x][y] += -10;
}

// If we modify a cell along the line, the corresponding costmaptrue cell
// is set to 1.
costmaptrue[x][y]=1;

// Next we loop along the y axis.
// If the slope is positive we loop from y1 to y2.
if (direction) {
    for (y = y1; y <= y2; y++) {
        x = ((y-y1)/(m +0.01)) + x1;
        // The cost is not modified in the y-axis loop if costmaptrue[x][y] is 1.
        // This is to ensure that the cells that were modified in the x axis loop
        // do not get repeated.
        if (!costmaptrue[x][y]) cost_map[x][y] += -10;
    }
}

// If the slope is not positive we loop from y2 to y1.
else {
    for (y=y2; y<=y1; y++) {
        x = ((y-y1)/(m +0.01)) + x1;
        if (!costmaptrue[x][y]) cost_map[x][y] += -10;
    }
}
void wall (int friendly_positions[][2], int enemy_positions[][2], int cost_map[][SIZE2], int robot)
{
    int i, j, k;

    for (i=0; i< ROBOTS; i++)
    {
        //doesn't create a wall for the robot in question, as that would
        //result in no valid moves
        if (i == robot) i++;

        for (j=friendly_positions[i][0]-SIZE; j<=friendly_positions[i][0] + SIZE && j<SIZE1; j++)
        {
            if (j<0) j=0;
            for (k=friendly_positions[i][1]-SIZE; k<=friendly_positions[i][1] + SIZE && k<SIZE2; k++)
            {
                if (k<0) k=0;
                cost_map[k][j] =10;
            }
        }
    }

    for (i=0; i< ENEMIES; i++)
    {
        for (j=enemy_positions[i][0]-SIZE; j<=enemy_positions[i][0] + SIZE && j<SIZE1; j++)
        {
            if (j<0) j=0;
            for (k=enemy_positions[i][1]-SIZE; k<=enemy_positions[i][1] + SIZE && k<SIZE2; k++)
            {
                if (k<0) k=0;
                wall
            }
        }
    }
}
for (k=enemy_positions[i][1]-SIZE; k<=enemy_positions[i][1] + SIZE && k<SIZE2; k++)
{
  if (k<0) k=0;
  cost_map[k][j] =10;
}

Get_Path Function

//pathfinding switchboard function
int get_path (int start_x, int start_y, /*int end_x, int end_y, */int cost_map[][SIZE2], int enemy_positions[][2],int finalpath[NODES][2])
{
  int x_intercepts[INTER][2], y_intercepts[INTER][2];
  int nodes [NODES][2], connect_matrix[NODES][NODES];
  int counter, i=1, j, k, l, m, n,p, max_nodes, valid;
  float slope;
int target_nodes[TARGETSIZE];
int targets;
int check, end;

// clear nodes matrix
for (i=0; i<NODES; i++)
{
    for (j=0; j<2; j++)
    {
        nodes[i][j]=-1;
    }
}

// clear finalpath matrix
for (i=0; i<NODES; i++)
{
    for (j=0; j<2; j++)
    {
        finalpath[i][j]=-1;
    }
}

i=j=1;

// first point on path is the robot's position
nodes[0][0]=start_x;
nodes[0][1]=start_y;

for (counter=0;counter<ENEMIES;)
{
    // creates a spread of nodes around opponent robots for use in the path
    i=design_hex(nodes,enemy_positions[counter][0],enemy_positions[counter][1],RAD,i);
    // avoids a problem with overdoing the number of enemies by moving this here
    counter++;
}

// records the highest number node created around opponent
max_nodes=i;

// finds nodes with lowest cost, records number of these
/targets as "targets"
end=999999;
targets=0;

for (n=0; n<SIZE1; n++)
{
    for (k=0; k<SIZE2; k++)
    {
        check=cost_map[n][k];

        if (check < end)
        {
            for (l=0; l<targets;l++)
            {
                target_nodes[l]=-1;
            }

            for (p=max_nodes; p<NODES; p++)
            {
                nodes[p][0]=nodes[p][1]=-1;
            }

            targets=0;
            m=0;
            i=max_nodes;
        }

        if (check<=end)
        {
            target_nodes[m]=i;
            m++;
            targets++;
            nodes[i][0]=n;
            nodes[i][1]=k;
            i++;
            end=check;
        }
    }
}

//adding nodes midway between each pair of robots
//tests seem to indicate this does not make much difference

for (p=0; p<ENEMIES-1; p++)
{
// by starting off at k=p+1, it becomes a convolution rather than
// permutation, preventing copies of nodes which increase computation time
for (k=p+1; k<ENEMIES; k++)
{
// order of positional data x,y functional but inconsistent throughout
// should clean this up when there is time
nodes[i][1] = (enemy_positions[p][0]+enemy_positions[k][0])/2;
nodes[i][0] = (enemy_positions[p][1]+enemy_positions[k][1])/2;
i++;
}

// checks which nodes are connected to which other nodes
connections (nodes, connect_matrix, cost_map, x_intercepts, y_intercepts, max_nodes);
/*
   // for check
   for (i=0;i<NODES;i++)
   {
       printf("nodes %d: ", i);
       printf("%d,%d
", nodes [i][0],nodes [i][1]);
   }
   printf ("\n\n\n");
   for (i=0; i<NODES;i++)
   {
       printf("%d is connected to...", i);
       for (j=0; connect_matrix[i][j]>-1; j++)
       {printf ("%d ",connect_matrix[i][j]);
        printf ("\n");
    }
    printf ("\n\n\n");
    // until here
    */

// if invalid returns -1, else returns cost of move
valid = leastpath (finalpath, nodes, cost_map, x_intercepts, y_intercepts, connect_matrix, targets,
target_nodes);

// To check processing time
/* finish = clock();
total = finish-begin;

printf ("%f\n", total / CLOCKS_PER_SEC);
printf ("%d \n\n", valid);
/* // for check
for (i=0; i<NODES; i++)
{
    printf("nodes %d: ", i);
    printf("%d,%d\n", nodes[i][0], nodes[i][1]);
}
printf("\n\n\n");

for (i=0; i<NODES; i++)
{
    printf("%d is connected to...", i);
    for (j=0; connect_matrix[i][j] > -1; j++)
    {
        printf("%d ", connect_matrix[i][j]);
    }
    printf("\n");
}
printf("\n\n\n");
//until here
*/
return 0;
}

Sub functions called in Get Path

Design_Hex

// creates the nodes
int design_hex(int nodes[32][2], int center_x, int center_y, int rad, int i)
{
    int x, y;

    x = int(center_x + rad);
    y = center_y;
    if (x >= 0 && y >= 0 && x <= SIZE2 && y <= SIZE1) {
nodes[i][1]=x;
nodes[i][0]=y;
i++;
}

x=int(center_x +.5*rad);
y=int(center_y+0.866*rad);
if (x>=0 && y>=0 && x<=SIZE2 && y<=SIZE1) {
    nodes[i][1]=x;
nodes[i][0]=y;
i++;
}

x=int(center_x -.5*rad);
y=int(center_y+0.866*rad);
if (x>=0 && y>=0 && x<=SIZE2 && y<=SIZE1) {
    nodes[i][1]=x;
nodes[i][0]=y;
i++;
}

x=int(center_x -rad);
y=center_y;
if (x>=0 && y>=0 && x<=SIZE2 && y<=SIZE1) {
    nodes[i][1]=x;
nodes[i][0]=y;
i++;
}

x=int(center_x -.5*rad);
y=int(center_y-0.866*rad);
if (x>=0 && y>=0 && x<=SIZE2 && y<=SIZE1) {
    nodes[i][1]=x;
nodes[i][0]=y;
i++;
}

x=int(center_x +.5*rad);
y=int(center_y-0.866*rad);
if (x>=0 && y>=0 && x<=SIZE2 && y<=SIZE1) {
    nodes[i][1]=x;
nodes[i][0]=y;
i++;
}
return i;
}

Connections

//stores which nodes are connected to which other nodes
int connections (int nodes [][][2], int connect_matrix[][NODES], int cost_map[][SIZE2], int x_intercepts[][2], int y_intercepts[][2], int max_nodes)
{
    int i,j,k,a,n,m;

    //resets connect_matrix
    for (i=0; i<NODES;i++)
    {
        for (j=0; j<NODES; j++)
        {
            connect_matrix [i][j]=-1;
        }
    }
}
for (i=0;i<NODES;i++)
{
    // reduce computation time by stopping when hit an empty node
    if (nodes[i][0]==-1 || nodes[i][1]==-1)
        break;
    for (j=0,k=0;j<NODES;j++)
    {
        //to save computation time, targets aren't connected to eachother
        if (i > max_nodes && j>max_nodes) break;
        /* each node skips checking if it's connected to same spot
           not using i==j b/c that won't check for two nodes at same spot*/
        if (nodes[i][0]==nodes[j][0]&&nodes[i][1]==nodes[j][1]) j++;
        //breaks at empty node
        if (nodes[j][0]==-1 || nodes[j][1]==-1)
            break;
        a=connect (cost_map, x_intercepts, y_intercepts,
                   nodes[i][0],nodes[i][1],nodes[j][0],nodes[j][1]);
        //resets intercept matrices
        for (n=0;n<INTER;n++)
        {
            for (m=0;m<2;m++)
            {
                x_intercepts[n][m]=-1;
                y_intercepts[n][m]=-1;
            }
        }
        /*if the points are connected, we add the connected node to "connect"
          in the "row" of the node it's connected to*/
        if (a==0)
        {
            connect_matrix[i][k]=j;
            k++;
        }
    }
}
return 0;


**Sub functions called by Connections**

**Connect**

//checks if 2 nodes are connected
//ie there is no node on the path with a cost above a certain threshold
int connect (int cost_map[][SIZE2], int x_intercepts[][2], int y_intercepts[][2], int x1, int y1, int x2, int y2)
{
    int i, fail=0;

    // finds all x and y intercepts on the line
    find_intercepts (x_intercepts, y_intercepts, x1, y1, x2, y2);
    // now checks that all grid spaces the line passes through the max cost or less
    // changed limit of i from 20 to inter
    for (i=0; i<INTER; i++)
    {
        if (cost_map[((int)x_intercepts[i][0])/64][((int)x_intercepts[i][1])/64] > NOCOSTVALUE)
        {
            //added in check for -1
            if ( x_intercepts[i][0]!=-1)
            {
                fail=-1;
                break;
            }
        }
        if (cost_map[((int)y_intercepts[i][0])/64][((int)y_intercepts[i][1])/64] > NOCOSTVALUE)
        {
            //added in check for -1
            if (y_intercepts[i][0]!=-1)
            {
                fail=-1;
                break;
            }
        }

        if (cost_map[((int)x_intercepts[i][0])/64-1][((int)x_intercepts[i][1])/64] > NOCOSTVALUE)
        {
            //added in check for -1
        }
    }
}
if (x_intercepts[i][1]/64!=0 && y_intercepts[i][0]!=-1)
{
    fail=-1;
    break;
}
}
if (cost_map [(int)y_intercepts [i][0]/64] [(int)y_intercepts [i][1]/64-1] > NOCOSTVALUE)
{
    //added in check for -1
    if (y_intercepts[i][0]/64!=0 && y_intercepts[i][0]!=-1 )
    {
        fail=-1;
        break;
    }
}
//if there is any cost on the line, fail =-1, else it equals 0
return fail;

Find_Intercepts

// creates a "fine grid" overlayed over the main grid
// with a resolution of 64:1. By finding where in fine
//grid one is when crossing from one grid space
// to another in the main grid, one can compute the distance
// travelled through each grid space
float find_intercepts (int x_intercepts[][2], int y_intercepts[][2],int x1, int y1, int x2, int y2)
{
    // need arrays to put intercepts in (first x, then y), and start and end points
    int fine_x1, fine_y1, fine_x2, fine_y2, sign_x=1, sign_y=1, x, y, i=0,j;
    float slope;
// set values to -1
for (i=0; i<INTER; i++)
{
    for (j=0; j<2; j++)
    {
        x_intercepts[i][j]=y_intercepts[i][j]=-1;
    }
}

i=0;

//first find the slope
slope=(float)(y2-y1)/(float)((x2-x1)+.00000001);

//find which direction to move in
if (x2<x1) sign_x=-1;
if (y2<y1) sign_y=-1;

//now convert start in large grid to fine grid (center of grid space)
fine_x1=x1*64+32;
fine_y1=y1*64+32;
//goes until last interception
fine_x2=x2*64; if (sign_x==-1)fine_x2+=64;
fine_y2=y2*64; if (sign_y==-1)fine_y2+=64;

//find y-intercepts
y=fine_y1-sign_y*32;
while (y!=fine_y2)
{
    //move to next space in large grid
    y+=64*sign_y;
    // from y-y1=m(x-x1)
    x=(y-fine_y1)/slope+fine_x1;
    y_intercepts[i][0]=x;
    y_intercepts[i][1]=y;
    i++;
}

i=0;
x=fine_x1-sign_x*32;

//find x intercepts same way
while (x!=fine_x2)
Least Path Function

*/-starts at the endpoint.

-enters all connecting nodes into a matrix (checker) with checker[n][0] being the node the move originates from and checker[n][1] being the node moved to.

-assigns all connecting nodes a value equal to the cost of moving to them from the node being checked from+ the cost of that node(in this case the last node and it has a cost of 0). this value goes into cost_from as the first entry. the second entry in cost_from is the node the move is being made from (if the value is changed by the move).

-every time it checks a connection, if the score is changed, it puts all connections from the node being moved to into checker with the node being moved to as the new node being moved from.
-erases the move executed from checker

-since moves are added to checker only when change occurs, when all nodes have the lowest possible value (smallest cost until end) checker will empty out. then the loop ends.

-starting from the first node it inputs the coordinates of the nodes along the best path. since the second entry in cost_from is the next node in the path, the second node is the second number in cost_from [0] (first node) next node is second number in cost_from at whatever value corresponds to the second node in the path etc.

-finally the cost of the move is returned in case we want to use it later*/

int leastpath (int finalpath[][2], int nodes[NODES][2], int cost_map[][SIZE2], int x_intercepts[][2],int y_intercepts[][2],int connect_matrix[][NODES],int targets, int target_nodes[TARGETSIZE])
{
    int cost_from [NODES][2], checker [CHECKSIZE][2]={-1}, counter, i, j,k=0,n=0,m=0, p=0, r=0, final_cost;
    double movescore, slope;
    //assign all nodes high number so that all later costs are lower
    for (i=0; i<NODES;i++) cost_from [i][0]=999999;
    //to account for multiple targets
    for (i=0; i<targets; i++)
    {
        cost_from [target_nodes[i]][0]=0;
    }
    // sets checker entries to -1
    for (i=0; i<CHECKSIZE;i++)
    {
        for (j=0; j<2; j++)
        {
            checker [i][j]=-1;
        }
    }

    for (i=0; i<targets; i++)
{ 
    j=target_nodes[i];
    
    r=0;
    /* fill in beginning of checker with first move, so that it's not empty 
    and loop won't end (to avoid do loop)*/
    // changed to > instead of != for safety
    while (connect_matrix[j][r] > -1) 
    { 
        checker[k][0]=j;
        checker[k][1]=connect_matrix[j][r];
        k++;
        r++;
    }
}
/* checker connects two points. If cost from the absolute end is less than 
previous value at the end point of single move, that node is assigned that 
cost. If a change is made, all connections from the end node of the movement 
are added to connect. Checking stops when no more changing occurs and the 
matrix is emptied*/
/* changed to > from != for safety - have to set checker to all -1. old way 
doesn't work*/
while (checker[0][0] > -1)
{
    // Gets intercepts and slope to be used by get_score
    slope=find_intercepts ( x_intercepts, y_intercepts, nodes[checker[n][0]][0], 
    nodes[checker[n][0]][1], nodes[checker[n][1]][0], nodes[checker[n][1]][1]);
    // Gets the cost of single node-to-node move
    movescore=get_score (slope, nodes[checker[n][0]][0], nodes[checker[n][0]][1], 
    nodes[checker[n][1]][0], nodes[checker[n][1]][1], cost_map, x_intercepts, 
    y_intercepts);
    /* Checks if the distance from the end (ie the cost of the first node + the 
cost of the move < previous cost from end (cost of 2nd node). If so assigns 
a new cost to 2nd node, and adds connecting nodes to be checked later*/
    
    // To eliminate -ve cost for a move which causes an infinite loop

    /* Setting move to 1 gives weird path as any move through a -ve zone is considered 
good, no matter the length. Need to find a way that more -ve is good, but doesn't 
ignore distance*/
    // Trying cost = distance
    if (movescore <= 0)
    {
    
    }
movescore = sqrt ((nodes[checker[n][0]][0] - nodes[checker[n][1]][0]) *
(nodes[checker[n][0]][0] - nodes[checker[n][1]][0])
+ (nodes[checker[n][0]][1] - nodes[checker[n][1]][1]) *
(nodes[checker[n][0]][1] - nodes[checker[n][1]][1]));

if ((movescore+cost_from[checker[n][0]][0]<cost_from[checker[n][1]][0])
{
    //assigns second node the new, shorter path score
    cost_from[checker[n][1]][0]=movescore+cost_from[checker[n][0]][0];
    cost_from[checker[n][1]][1]= checker[n][0];
    /*prepares all connecting nodes to second node for future checking if change affects them
    by putting them at end of checker matrix*/
    j=checker[n][1];
    //adds at end of checker matrix (where k left off) and ignores node the move originated from
    // changed to > for safety
    while (connect_matrix [j][p]> -1)
    {
        //changed k++ to p++
        if (connect_matrix [j][p] == checker[n][0]) p++;
        if (connect_matrix[j][p] == -1) break;
    }
    checker [k][0]=j;
    checker [k][1]=connect_matrix[j][p];
    k++;
    p++;
    }
    p=0;
    //removes move just executed from the checker matrix
    m=0;
    //changed to > from != for safety
    while (checker[m][0]>-1)
    {
        checker[m][0]=checker[m+1][0];
        checker[m][1]=checker[m+1][1];
        m++;            
    }
    k=-1;
}
/*now traces the shortest path back. starts at first node (which now has a
value equal to the total cost of movement) and creates a path based on
cost_from's second value which is next node in path*/

counter=0;
i=0;
/*stops adding node, which has a cost from the end (ie cost_from's 1st
number) of 0*/
   // changed while to exclude null move
while (cost_from[i][0]!= 0 && i>=0 && i<=1000)
{
   //switched x and y - make sure to fix
finalpath[counter][1]=nodes [i][0];
finalpath[counter][0]=nodes [i][1];
   //changed from [i][0]
      i=cost_from[i][1];
      counter++;
}
if (i>=0 && i<=1000)
   //fills in final move
   {
finalpath[counter][1]=nodes[i][0];
finalpath[counter][0]=nodes [i][1];
//gets score of final move in case we'll use it later
final_cost=cost_from[0][0];
}
/*
   //for check
   printf ("checker");
for (i=0; i<CHECKSIZE; i++)
{
   for (j=0; j<2; j++)
   {
      printf (" %d ", checker[i][j]);
   }
   printf ("\n\n\n");
}  
*/
   //until here
else final_cost = -1;
return final_cost;
Sub Functions called by Least_Path

Get_Score

double get_score (float slope, int x1, int y1, int x2, int y2, int cost_map[][SIZE2], int x_intercepts[][2], int y_intercepts[][2])
{
    int i=0, j=0, x_y=1, point1_x, point1_y, point2_x, point2_y, sign_x=1, sign_y=1;
    //not optimal, switched x_y values to accomodate that this has x/y opposite of find_intercepts

double cost=0;
    // checks if first intercept is an x or y intercept
if (slope<-1 || slope>1) x_y=0;
    // converts start coord to fine grid
point1_x=(x1*64)+32;
point1_y=(y1*64)+32;

    //added in for backwards move
if (x2<=x1) sign_x=-1;
if (y2<=y1) sign_y=-1;

while ((point1_x+(32*sign_x))/64 != x2 || (point1_y+(32*sign_y))/64 != y2)
{
    //sets end of increment intercept
if (x_y==1)
    {
        point2_x=x_intercepts[i][0];
        point2_y=x_intercepts[i][1];
    }
else
    {
        point2_x=y_intercepts[j][0];
        point2_y=y_intercepts[j][1];
    }
    //gets distance*costmap value, adds to total cost
    //divides by 32 to reduce value, allow farther movement w/o overloading the int designation
    cost += sqrt (((point2_x-point1_x)*(point2_x-point1_x))+(point2_y-point1_y)*(point2_y-
        point1_y))*(cost_map[(point1_x+5*sign_x)/64][(point1_y+5*sign_y)/64]/32);
    //sets the 2nd point as first point of next increment
    point1_x=point2_x;
    point1_y=point2_y;
    /* goes to next intercept in matrix and checks if next intercept used should be x or y intercept by checking if next x/y intercept is in the same grid space*/
}
//changed from && to ||
if (x_intercepts[i+1][0]!=\-1|| y_intercepts[j+1][0]!=\-1)
//otherwise problems with 2nd last move
{
  if (x_y==0)
  {
    j++;
    // checks if intercept is corner, if so skips copy in other intercept grid
    if (point1_x%64==0&&point1_y%64==0) i++;
    if (point1_y/64==y_intercepts[j][1]/64) x_y=1;
  }
  else
  {
    i++;
    // checks if intercept is corner, if so skips copy in other intercept grid
    if (point1_x%64==0&&point1_y%64==0) j++;
    if (point1_x/64==x_intercepts[i][0]/64) x_y=0;
  }
}

// evaluates cost in the last grid space
point2_x=x2*64+32;
point2_y=y2*64+32;
//divides by 32 to reduce value, allow farther movement w/o overloading the int designation
cost += sqrt (((point2_x-point1_x)*(point2_x-point1_x))+((point2_y-point1_y)*(point2_y-point1_y)))*(cost_map[x2][y2]/32);
return cost;
**Path Follow Function**

```c
void path_follow (int velocity_x, int velocity_y, int robot, int friendly_positions[][], int finalpath[][]) {
    int predicted_x, predicted_y, target_x, target_y, distance;
    int target [ROBOTS][2]
    //may get rid of robot and make targets [1][2] b/c when call would be for specific robot

    predicted_x = friendly_positions [robot][0]+velocity_x*LOOKAHEAD;
    predicted_y = friendly_positions [robot][1]+velocity_y*LOOKAHEAD;

    distance=pathprojection (friendly_positions [robot][0],friendly_positions [robot][1],finalpath[1][0],finalpath[1][1],predicted_x,predicted_y, robot, target[i][2]);

    /*assume will never reach the first point on the desired path b/c it keeps on being updated
    unless this point is the final point. Thus can go to the projection and don't need to worry
    about
    turning corners*/

    //need a GOTO function, provided by simulator
    if (distance > variation)
    { 
        GOTO (targer[robot][0], target[robot][1]); //x and y coordinates of projection on path
    }
}
```