King Fahd University of Petroleum and Minerals

Robotics and Autonomous Systems Concentration (CX)

CISE 483

AI and ML for Robotics

Simulating Path-Planning Algorithms used in RoboCup SSL Competitions

December 2020

Ahmed Alharbi	201636500	Computer Engineering
Hamza Alsharif	201642320	Computer Engineering
Abdulrahman Javaid	201678920	Electrical Engineering

Dr Khaled Alshehri

Contents

List of Figures

1 Outline

In our project we have focused on implementing path planning algorithms for the RoboCup Small-Sized League (SSL) competition [4] so that we can observe their behaviour within grSim, the simulator that was designed for the competition. These algorithms were the Fast Path Planner (STOx' Planner) [8] and the Rapidly-exploring Random Tree (RRT) [5]. The report first gives context to our project through a brief overview of the RoboCup SSL competition. It then dives into the algorithms that we have implemented for our project, then goes into the structure of our code-base, and finally provides the results of our work. We then provide a section that talks about the issues we have faced during development as well as the improvements we can make to our project in the future. The code-base for the project can be found at this link: https://github.com/hsfalsharif/CISE483-Robot-Football

2 Competition Overview

The RoboCup SSL competition is a worldwide robotics competition which aims at improving the field of robotics and research within it by having participants develop teams of robots that compete against each other in a small-sized version of football [4]. A team consists of eight robots each moving with four omni-wheels and tracked using colour-coded labels that are detected using a global vision system. The vision system sends information about the current state of the game to each of the participants' systems, which use this information to send commands to the robots so that they may be able to execute the optimal winning strategy. The field is 12 m long and 9 m wide

and an orange golf ball is used in the game. RoboCup competitions have been held consistently since the RoboCup federation's establishment in 1997, with the goal of eventually reaching a level in which robots can compete against humans and possibly even win against them by 2050.

3 Algorithms Implemented

3.1 Rapidly-exploring Random Tree (RRT)

The RRT algorithm [5] is considered to be a standard in terms of path-planning in dynamic environments with obstacles and is also the basis for many similar algorithms. It works by building the RRT tree by adding random branches to the tree starting from the initial node towards the goal node. Its effectiveness is in the fact that it is not biased towards a certain direction but rather expands outwards in the general direction of the goal rather than towards it directly. This is useful in avoiding obtrusive obstacles which could prevent the algorithm from finding the optimal path. Many iterations of RRT have also been developed such as RRT* [3] and real-time RRT* (RT-RRT*) [6], but for our case we decided to implement the base RRT algorithm. The algorithm is as follows:

1. First read the start and goal positions as arguments and initialise the tree with the start position as the first node.

2. While the the goal has not been reached: find the nearest node in the tree to the goal and extend it with a new node that is of a fixed distance away and is in the direction of a random vector.

3. If the goal has been reached, traverse the tree to find the computed direct path from the start to the goal node. Otherwise continue tree extension.

The pseudo-code algorithm given in the original paper is shown below:

```
BUILD_RRT(x_init)
 1  T.init(x_init);
 2  for k = 1 to K do
 3      x_rand ← RANDOM_STATE();
 4      EXTEND(T, x_rand);
 5  Return T
```

```
EXTEND(T, x)
 1  x_near ← NEAREST_NEIGHBOR(x, T);
 2  if NEW_STATE(x, x_near, x_new, u_new) then
 3      T.add_vertex(x_new);
 4      T.add_edge(x_near, x_new, u_new);
 5      if x_new = x then
 6          Return Reached;
 7      else
 8          Return Advanced;
 9  Return Trapped;
```

Figure 1: Here we observe the original pseudo-code algorithm given in the RRT paper [5].

3.2 Fast Path Planner (STOx' Planner)

This algorithm will generate straight line segments between two points while avoiding obstacles. It was developed by STOx team to compete against the RRT algorithm and was designed to be fast and finish in microseconds, unlike RRT which requires time in the range of milliseconds to find a path. Both STOx' planner and RRT are designed to work in dynamic environments

where the obstacles change positions every 16.67ms (60 frame per second). The algorithm works as follow :

1. Generate a straight line between the origin to the target.
2. If there is an obstacle then create a sub-goal above or below the obstacle with a certain distance. This sub-goal must be perpendicular to the obstacle.
3. Generate straight lines that connect the origin to the sub-goal and connect the sub-goal to the target position.
4. Check for obstacles again and if there are any, generate a sub-goal above it.

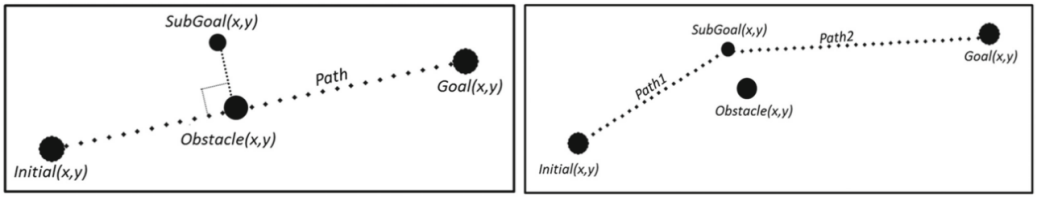

Figure 2: The figure shows how the algorithm deals with an obstacle in the path in real time, by generating a nearby subgoal that it can travel to [8].

The respective algorithm pseudo code is as follows:

```
def FastPathPlanning(environment, trajectory, depth)
    obstacle = trajectory.Collides(environment)
    if obstacleFaced and depth < max_recursive:
        subgoal = SearchPoint(trajectory, obstacle, environment)
```

```
        trajectory1 = GenerateSegment(trajectory.start, subgoal)
        trajectory1 = FastPathPlanning(environment, trajectory1, depth+1)
        trajectory2 = GenerateSegment(subgoal, trajectory.goal)
        trajectory2 = FastPathPlanning(environment, trajectory2, depth+1)
        trajectory = JoinSegments(trajectory1, trajectory2)
        return trajectory
```

3.3 A* Algorithm

A* is an informed search algorithm [2]. It starts from a specific starting node of a graph and aims to find a path to the given goal node having the smallest cost (least distance travelled in shortest time). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

A* algorithm has 3 parameters:
g : The cost of moving from the initial point to the current point
h : The estimated cost of moving from the current point to the final point.In our case we used Manhattan distance as the h function.
f : It is the sum of g and h. So, f = g + h

The algorithm works as follows

1. First, it saves the location of the origin point and the goal point
2. It expands the children of the origin point and calculate f for each child
3. The child with least f value would now become the new state

4. The new state would now expand its children.

5. The previous steps will be repeated until the goal is reached

The pseudo code for A* algorithm is as follows

1- Add the starting node to the open list
2- Repeat the following steps:
    a. Look for the node that has the lowest f on the open list. Refer to this node as the current node.
    b. Switch it to the closed list.
    c. For each reachable node from the current node:
        i. If it is on the closed list, ignore it.
        ii. If it isn't on the open list, add it to the open list. Make the current node the parent of this node. Record the f, g, and h value of this node.
        iii. If it is on the open list already, check to see if this is a better path. If so, change its parent to the current node, and recalculate the f and g value.
    d. Stop when
        i. Add the target node to the closed list.
        ii. Fail to find the target node, and the open list is empty.
3- Trace backwards from the target node to the starting node. That is your path

Figure 3: Pseudo code for A* algorithm [7]

4 Code Structure

Network.py This class is responsible for the communication between our code and the simulator over a UDP socket. It ensures that we always receive the latest packet from the simulator.

Playground.py This class will keep our local representation of what happens inside the simulator. It will also update each component in our system

when we receive a new frame from the simulator through the Network class. It also contains our custom GUI to show the path planned by our algorithms as a visual representation.

robot.py This is the main functioning class in our project and is the most relevant in the context of CISE 483. It contains our implementation of both aforementioned algorithms as well as our simple kinematic controller, which is relevant to what we studied in CISE 480. The implementation of the STOx planner is distributed throughout the following functions:

- **Planner:** This is the main STOx planner implementation. It contains the recursive function that will be invoked by the robot instance with the given parameters.
- **get_obstacles:** This function will check a line against a group of obstacles using the overlap function.
- **overlap** This function will check if a line intersects with a circle by doing two checks. Firstly it check if the circle center inside the rectangle formed by the line vertices and if it is inside it performs the second check. The second check is done by calculating the normal between the circle's center and the line. If the distance is less than the circle's radius then there is a collision. These calculations assume that the radius of the circle is less than the height or width of the formed rectangle which is mostly true in the context of the robocup where the robot radius is usually around 80cm.
- **get_sub_goal** this function calculate the position of the sub-goal. first

it finds the point of intersection then finds a perpendicular vector from the point of intersection . finally using this vector we calculate the sub-goal position by adding it to the point of intersection.

$$X = X_{intersection} + v_x * d \tag{1}$$

$$Y = Y_{intersection} + v_y * d \tag{2}$$

where d is the distance between the sub-goal to the line.

- **move_to and move_to_RRT** These are the main functions that are called to move the robot to a specific position. They are practically identical, except that move_to uses the STOx planner and move_to_RRT uses the RRT algorithm. One other difference is that we pass the playground directly in the case of move_to_RRT which is not how it is supposed to be done but we did this for debugging purposes. Robots within the simulator accept commands in the form of tangential and normal velocities, or V_x and V_y, so after a path is returned from the respective algorithms a conversion is performed from the individual path positions into velocity commands that can be accepted by the robot. These velocities are with respect to the robot frame while the planned path is with respect to the world's frame. To turn a path into a velocity commands first split the path into line segments. Then find out which segment of the path the robot is currently in. After that create a unit vector parallel with the incoming line segment and finally transform the vector with the specified magnitude into the robot frame using the rotation matrix.

- **RRT** This is the main function that performs the RRT algorithm. It takes the start and goal positions as parameters, as well as the Playground object for GUI representation as well as for debugging. We start by initializing the tree with the start node. Then while the distance to the goal is greater than a defined threshold we call the extend function which adds a new node to the tree and returns the distance to that new node to the goal. After building the tree we need to return the planned path. This is done by iterating from the goal node to the start node by accessing the parent nodes. We finally reverse the path (since we have started from the goal and have stopped at the start node), and finally return that path.

- **extend** The extend function is the main definitive function for the RRT algorithm. It extends the current version of the tree by one new node based on a random position in the general direction of the goal. First a random position in the field is generated, and the nearest node in the tree to the goal is obtained. A unit vector from the obtained nearest node and in the direction of the random position is then computed and the tree is extended with a new node in the direction of this unit vector and with a fixed specified magnitude. The current nearest node is assigned as a parent to this new node and the new node is added to the tree, extending from the assigned nearest node parent. The distance of this new node to the goal is finally returned. In retrospect and after already completing this function's implementation, we believe that finding a unit vector in a random direction might be a more efficient approach in terms of performance but the approach we

have used works also.

Node.py This class defines a general structure for a node to be used as a building block for the tree in the RRT algorithm. It consists of a position and the parent node. The position is used to indicate the coordinate of the node within the field, and the parent denotes the previous node from which this current one extends from. The reason for including the parent is that we obtain the planned path from the tree by iterating backwards through the parents starting from the goal node until we reach the start node. The returned planned path is the reversal of this computed path.

Astar.py In the code of A* star, we defined many functions. In this section, we will explain these functions and the use of each function.

- *h(p1, p2) :* This function is used to find the heuristic distance between current point and the target point. Its inputs are *p1* (current state position) and p2 (target state position)

- *reconstruct_path(came_from, current, draw)*: This function is defined to construct the optimal path as we got to the goal. Its inputs are *came_from* (an array that saves all the states that have been visited to reach current state), *current* (it has the location of current state (goal usually) and *draw* (to draw the path between came_from and current))

- *algorithm(draw, grid, start, end)* : This is the most important function is the program. It contains the algorithm of A*. It calculates the f function of A* and decides next states we should go to.

- *make_grid(rows, width)* : it divides the screen of *pygame* with certain rows and width.
- *draw_grid(win, rows, width)* : it draws the grid line that are divided by *make_grid* in pygame screen
- *draw(win, grid, rows, width)* : it draw the paths which are made by algorithm
- *get_clicked_pos(pos, rows, width)* : this is used to get the position from the mouse (clicks) of the obstacles, start and end points
- *main(win, width)* : this the main function of the program and it combines all of the above functions

5 Results

The final results of our project is that we have completed the RRT and STOx planner algorithms and have successfully run both of them on the GUI interface we developed. Additionally we have successfully run the STOx planner algorithm on the simulator. In the following sections we have outlined the results for each of the algorithms separately. More animations and videos that show demos of the different components of our project are available in the README of the github repo.

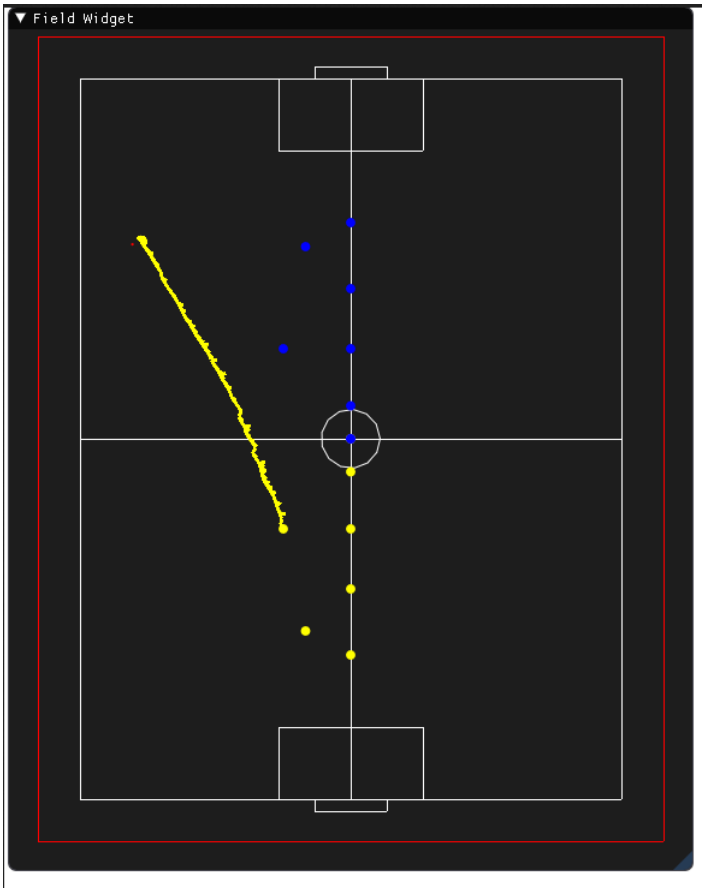

Figure 4: The figure shows the operation of RRT within the field using our GUI interface. The yellow path shows the final path that the robot follows to get to the ball which is represented using a red dot. Notice that the robot stops within a threshold of the ball, which can be set by the user as necessary.

5.1 RRT

As mentioned earlier, we were able to run the RRT algorithm on our GUI interface. Figure 4 shows one of these runs. Figure 5 shows multiple runs of the algorithm on the GUI with the same start and end positions. There are

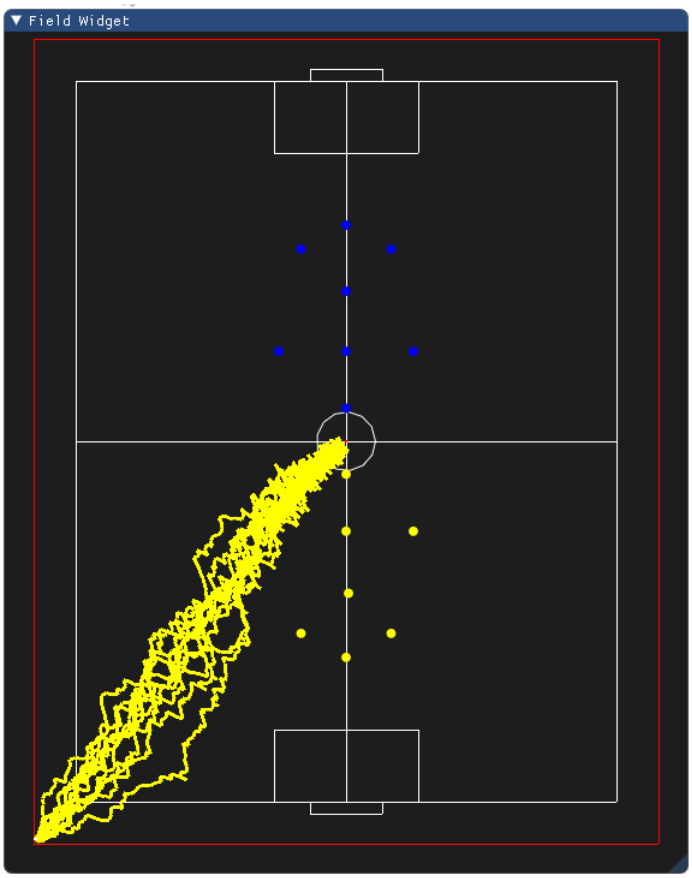

Figure 5: Here we observe the algorithm's operation with the same start and end positions but across different runs. We have different paths because there is an element of randomness to the path computation so the path might not be the same each time

different paths computed due to the element of randomness in the algorithm. We were unfortunately unable to run the RRT algorithm on grSim due to some issues with it processing in real-time. This is mainly due to the fact that the RRT algorithm is a lot slower than the STOx algorithm in computation and produces constantly changing velocity vectors. This is difficult to

simulate on grSim since the path is not being computed in real-time while the simulator needs to accept the new velocity vectors as soon as they arrive.

5.2 STOx Planner

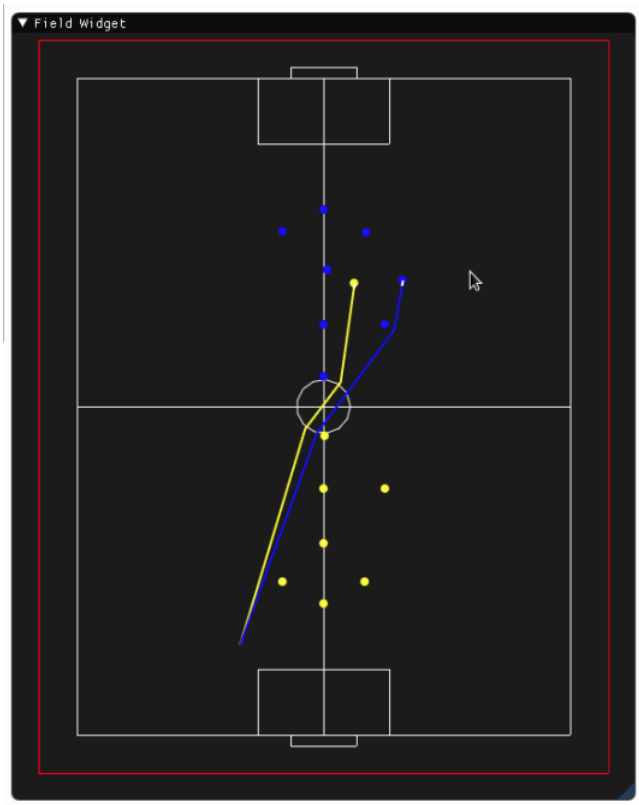

Figure 6: The figure shows a run of the STOx planner for two robots on opposing teams. The robots start at different positions and reach the same goal position in the end.

In contrast to RRT, we were able to run the STOx planner algorithm on both the GUI and the grSim simulator. The main difference here is that

it was designed to be able to run in real-time and in an environment with small-sized obstacles (in this case other robots) so running it on the simulator was more feasible than it was for RRT. For the STOx planner, the path is computed and corrected during each frame which gives a sense of a closed feedback loop. Therefore running STOx planner in the simulator was much easier than RRT which takes more than one frame to generate the path which makes it lag behind the changes in the environment. The result here is that the STOx planner computation is in the range of microseconds while RRT is in the range of milliseconds. Figure 6 shows a run of the STOx planner on the GUI for two robots on opposing teams. Figure 7 shows the STOx planner algorithm run on a more open environment with a lot more obstacles.

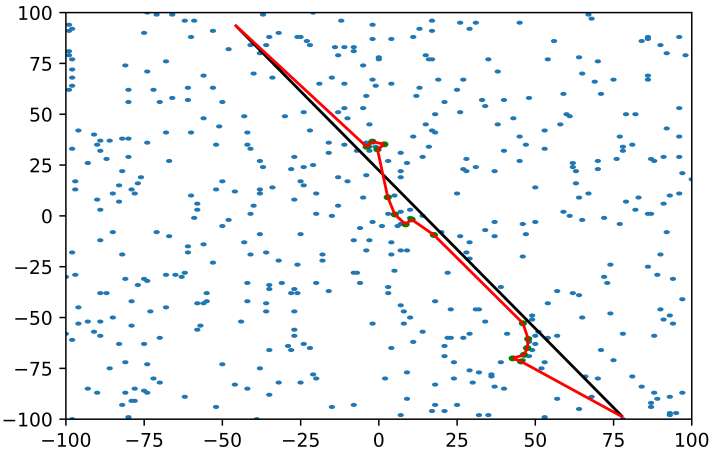

Figure 7: STOx' planner in random generated environment with 100 obstacles , green circles are the sub-goals (generated using matplotlib).

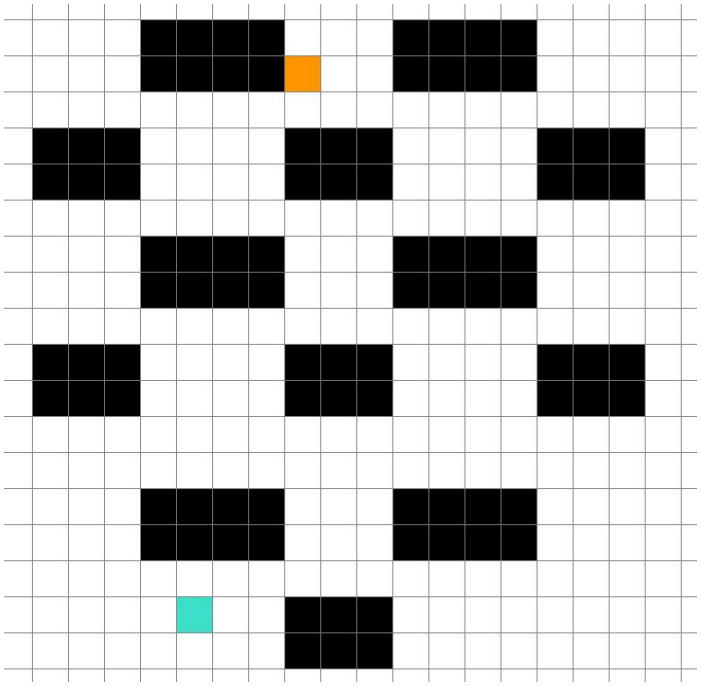

Figure 8: Initial arrangement of the robots and ball

5.3 A* Algorithm

To best test A* algorithm in short time, we chose built in library of python *pygame*. We select the obstacles (player) by click the right button of the mouse and determine the initial and end points. Figure 8 shows the robot_player (black), initial robot position (Turquoise) and ball position (Orange). Figure 9 shows the algorithm while it is running. Figure 10 shows the algorithm while it is running.

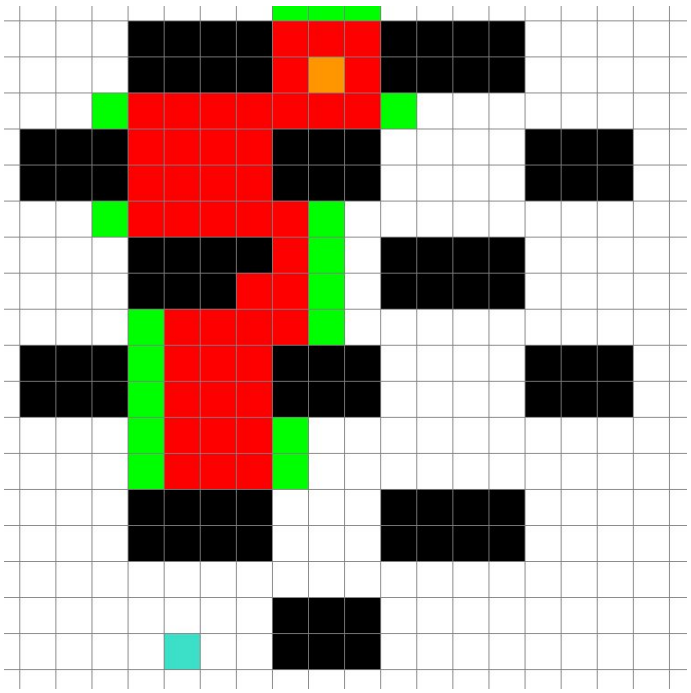

Figure 9: During the search.

6 Issues Faced

During development we faced a number of issues that caused us to fall short of the level that we wanted to reach with this project. The first and most prominent of these is figuring out how to actually get the robot to maintain itself on the path after it has been computed. This is because the path is something that is computed beforehand and is subject to change while the robot is moving due to the dynamic nature of the environment. This issue is even more prevalent when it comes to the RRT algorithm which takes more time to compute and also follows a changing vector so the real-time aspect must be more closely regulated. This has prevented us from being

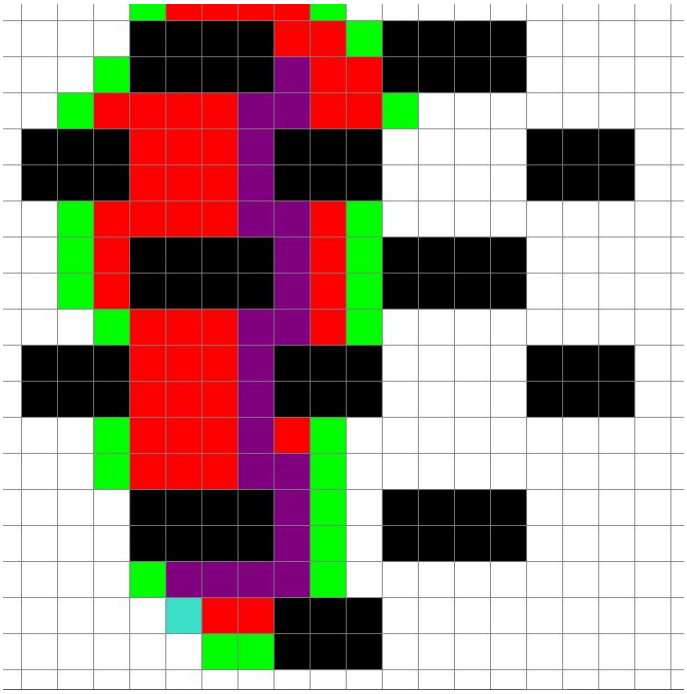

Figure 10: The final computed path.

able to run the RRT algorithm in the simulator due to the slower nature of its computations, so it instead only works on the GUI we had developed. The STOx planner algorithm straight line paths using a constant vector, so is less susceptible to this problem, but still suffers from it slightly. We believe that with a better understanding of the real-time aspect of the path-planning as well as the kinematic physics involved during the motion we would be able to better tackle this issue. The second major issue we had faced is finding a way to observe the computed path in real-time so that we actually know what vectors are being sent to the robot to direct it. We fortunately did find a solution to this by developing a GUI that shows the real-time motion of the robots as dots on the field with the path as a line that is drawn as the robots

move. The development of this GUI helped us significantly during debugging and refinement of our code. There are two issues faced us in A*. first the A* algorithm we developed works for discrete environments. However, the environment of robocup is continuous. The second issue is that A* algorithm takes longer computational time in Comparison to RRT and STOx, which make A* less competitive in the robocup competition.

7 Future Work

While our project does achieve many of the objectives that we initially placed for it, improvements can definitely be made going forward. One aspect that can be accounted for in our algorithm computations is the gradual change of the robot's heading so that it can better adjust to sudden changes in the vectors given by the path. We can also further improve our algorithms by taking more physical factors into consideration such as friction, inertia, acceleration, as well as other computational errors such as in the vision system, latency in receiving the commands from the central computer, etc. Additionally, path planning alone is not sufficient to participate in the RoboCup. For that we would need to implement a fully-functional artificial intelligence that is actually able to command the team of robots synchronously to be able to beat the opposing robot team. For this we found a potential option that is used by many teams called Skills, Tactics, and Plays (STP) [1]. The idea behind this is that skills are the basic manoeuvres executed by single robots such as moving to specific positions, passing the ball, scoring, etc. Tactics are defined sequences of skills that are also executed by a single robot to

achieve an objective that is dependant on the current state of the game. Plays are strategies that are performed by the whole robot team and consist of tactics that are given to each of the individual robot members. We plan to experiment with the implementation of this approach, possibly when we have acquired more knowledge in the field of multi-agent control. We could also consider designing our own path planning and control algorithms to see how they perform on this system so that we may better understand how the best robot team performance plays can be implemented in practise.

8 Work Distribution

Ahmed Alharbi designed the GUI interface as well as the implementation of the STOx planner algorithm. Hamza Alsharif implemented the RRT algorithm. Abdulrahman Javaid implemented the A* algorithm. Code debugging and report writing was a joint effort.

References

[1] B Browning, J Bruce, M Bowling, and M Veloso. Stp: Skills, tactics, and plays for multi-robot control in adversarial environments. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 219(1):33–52, 2005.

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

[3] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[4] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: the robot world cup initiative. *Proceedings of the International Conference on Autonomous Agents*, 04 1998.

[5] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, volume 1, pages 473–479 vol.1, 1999.

[6] Kourosh Naderi, Joose Rajamäki, and Perttu Hämäläinen. Rt-rrt*: a real-time path planning algorithm based on rrt*. In *The 8th ACM SIGGRAPH Conference*, pages 113–118, 11 2015.

[7] M. Parth, S. Hetasha, S. Soumya, and V. Saurav. A review on algorithms for pathfinding in computer games. *researchgate*, pages 1–6, 2015.

[8] Saith Rodríguez, Eyberth Rojas, Katherín Pérez, Jorge López, Carlos Quintero, and Juan Calderón. Fast path planning algorithm for the robocup small size league. In Reinaldo A. C. Bianchi, H. Levent Akin, Subramanian Ramamoorthy, and Komei Sugiura, editors, *RoboCup 2014: Robot World Cup XVIII*, pages 407–418, Cham, 2015. Springer International Publishing.