Scenario Week 4

Move-and-Tag Competition

COMP205P – Software Engineering

Team Manticore

Kazuma Hochin	zcabkho@ucl.ac.uk
Sam Pham	zcabsph@ucl.ac.uk
William Lam	zcabwhy@ucl.ac.uk
Zi Sim	zcabzjs@ucl.ac.uk

Table of Contents

1 – Move-and-Tag Problem

1.1 - Introduction

The match and tag problem has two smaller sub-problems. The first sub-problem is the path finding problem where the robots need to find a path from a starting point to an end goal and also navigating through obstacles. The second problem is the freeze tag problem where the robots try to awaken a swarm of robots in the environment. So we have to combine solutions to these two problems to solve the match and tag problem.

1.2 - Finding the Solution

The parser for the mat (input) file was created using C#. The parser recognises the coordinates of robots and obstacles/polygons which are converted into valid format that can be processed by the Python program. Thirty text files with the coordinates of each robot and polygons for each configuration were generated.

The greedy passive time algorithm was also implementing in C#. This is because, C# and other languages like Java offer faster processing and more efficiency compared to Python. Also, Sam was comfortable in that language.

We used Python to implement the rapidly exploring random tree algorithm (RRT). We found a simplified implementation of the RRT algorithm using Python on online resources [1] and we extended this code for our case, such as: implementing our own version of the line intersection detection, smoothing the initial RRT path and dynamically setting the closet goal. Python's external library, Shapely[2], was used to implement the testing function to determine whether the generated new branch was intersecting the polygons.

1.3 - Visualisation

We also used Python to implement the visualiser as our RRT algorithm was implemented using same the language. Matplotlib [3] (Python's external library) was used, which provides easy and quick methods to visually represent our data. Since the RRT algorithm and the visualiser were both implemented using Python, the testing of the RRT algorithm was conducted conveniently. The path created by the RRT algorithm was directly fed to the visualiser to get instant feedback of the algorithm at each step.

Other Python libraries: random, math and ast were also used to carry out some mathematical calculation.

2 – Algorithms and Complexity

The algorithms used through this project can be sorted into two camps. One for the freeze problem and another for the pathfinder.

2.1– Path Finder

We used the RRT algorithm to find the robot paths. The rapidly exploring tree algorithm uses a random generator to try to randomly expand a tree's branches where a robot's position acts as a root. The starting robot first awakens another robot and the path of that is added to the list of paths. Then if there are two robots at one point, the robots are processed in turn using a queue and paths are continuously added to the path list. There is a list of unawaken robots which contains robots except the starting one. When this list reaches zero or the queue is empty, the program terminates and the path is returned.

We have also implemented an intersection detection algorithm using shapely to check if any paths created intersects with any polygons. Furthermore, we have implemented an algorithm that decreases the number of notes along the initial RRT path so straight paths are produced.

We have also implemented an intersection detection algorithm using shapely to check if any paths created intersects with any polygons. Furthermore, we have implemented an algorithm that smoothes the initial RRT path so straight paths are produced.

RT Algorithm Pseudocode

```
RRT(start,goal,obstacleList, rangeXcoord)
{
      While true
      {
            node, edge = CreateRandomNodeAndEdge(goal,rangeXcoord)  //O(1)
            boolCollide = CheckCollision(node,node,obstacleList)    //O(BV)
            if boolCollide and goal < sample distance
            {
                  return node, edge
                }
            else {
                  continue
            }
      }
      AppendPath(Node,Edge)  //O(1)
}

CheckCollsion(node,node,obstacleList)
{
      for each polygon in obstacleList
            return line(node,node).intersects(polygon) //O(BV)
}
```

The above pseudocode describes the RRT algorithm where the inputs are the start robot, the end robot, the list of vertices of the obstacles as obstacleList and the range of x coordinates. The algorithm first generates the random node and edge and checks for collision. If there is no collision, then the path is returned.

The time complexity of the algorithm is O(BV) because the checkcollision function contains code that loops through each polygon and then finds out if the line segments generated intersects any polygon edges. B is the number of polygons and V is the edges of the polygons and they both vary depending on the input.

2.2 - Freeze Tag

The current implementation for the freeze tag problem is a greedy algorithm with jobs being spread out between robots to produce the shortest possible time to awaken the entire swarm. This is a greedy algorithm has it will always take the shortest possible overall time/distance with no way of going back if a better decision comes along. As a result, yes it may bring better results compared to other algorithms but is still in some cases not the best.

The decisions are based on the time it takes for a robot to reach another specific robot and selecting the robot that will produce the lowest overall or no increase on the entire system as a whole. So the algorithm keeps a list of awakened robots and unawaken robots. Each robot has a total distance travelled stored and every time a robot moves, it picks the path that gives the smallest total distance travelled by the robot. The input to the algorithm is a list of all robots currently unawaken and the time complexity is O(n^2) because there are two loops that loops through the unawaken robots to calculate the shortest distance.

GreedyPassiveTime Algorithm

```
GreedyPassiveTime(unawakedRobots)
{
      while(number of unawakenRobots != 0)  //O(n)
      {
            for each awaken robot
            {
                  for 1 to len(unawakenRobots) //O(n)
                  {
                        distance = distance(awakenRobot, unawakenRobot)
                        if (distance < shortestDistance)
                        {
                              shortestDistance = distance
                        }
                        addDistance(shortDistance)
                  }
            }

            findShortestDistance()
            createPath(unawakenRobot,awakenRobot)
      }
}
```

2.3 - Other Alternatives

There were a number of alternative algorithms what we had considered when thinking about how to approach the problem. Initially, we did try using the Pyvisgraph[4] library in Python to create a visibility graph to construct robot paths. It also uses Dikjstra's algorithm to find the shortest path between two robots. We used an in-order and greedy claim algorithm to try to find the best paths for multiple robots so all robots are awakened. But unfortunately, the external module has a number of issues which we were unable to rectify.

We have also initially implemented a dynamic goal selector so that when a robot is found to be closer whilst randomly sampling, the robot is chosen as the goal instead. However, the code did not work as desired and as the deadline approach we decided it was best not to end up using this algorithm. This would solve both the freeze tag and the path finding problem.

The greedy passive time optimised algorithm came about after looking at available methods and also to understand what type of algorithm was needed to achieve the desired result. The initial algorithm was the naïve just connect the robot up in list order. Yes this did awake up the entire swarm, but was incredibility inefficient. The next step up was the greedy active robot algorithm where we iterated through the active robots and find the closest inactive robot and add it to the path and list of active robots. Another was to iterate through the passive robots. But this was only optimised for the shortest total distance travelled and had no care for shortest time.

Then we came up with the greedy passive time optimised algorithm as explained earlier. There were other alternatives such as using Dikjstra's or A* pathfinder.

On the next page shows what a different a different algorithm can make. On the left was one of the first algorithms we implemented, and compared to the right which is the greedy time optimised, not only does it have a shorter overall length, it is also short in overall time duration.

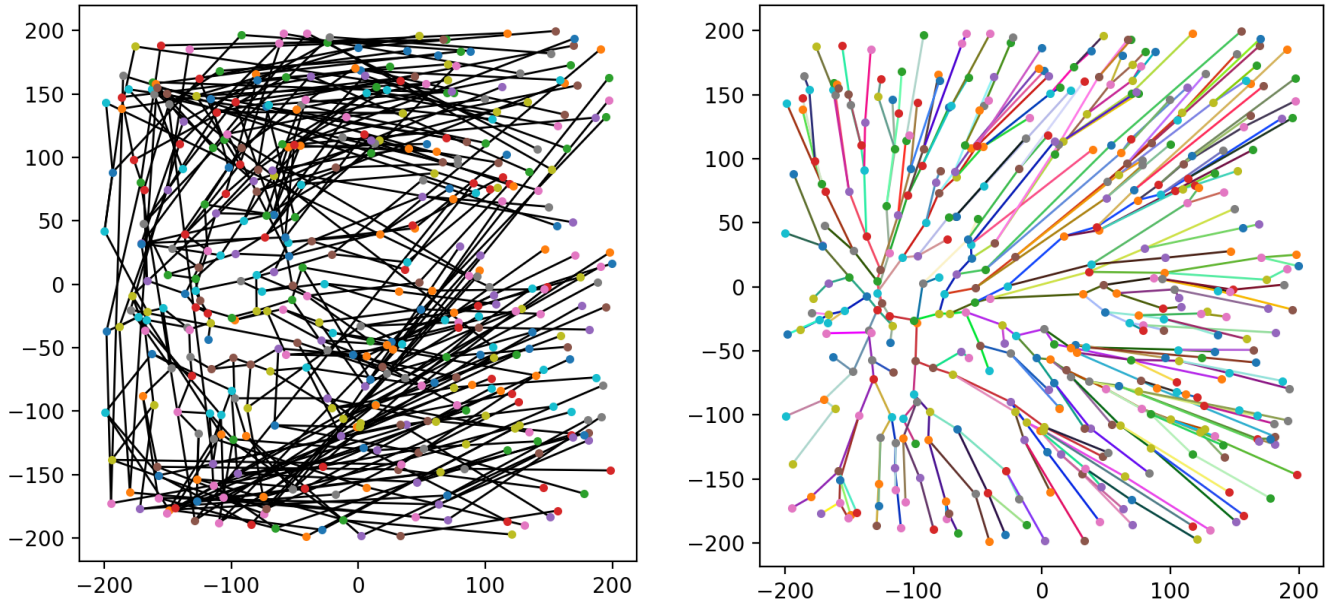

Figure 1 – Difference in algorithm for the same

3 – Testing Environment

The algorithms were tested using a variety of methods. We created some simple test cases with different robot positions and different polygons. We also used the visualisation library Matplotlib to visualise the different maps and see if any of the robot paths intersect with any of the polygons. Also, we used the library to display the tree of the RRT algorithm and to make sure the code is working correctly. We also continuously submitted robot paths to the server to make sure our algorithms work even with large number of robots and polygons. Furthermore, we used the IDE debugger to debug our algorithms and to check if the variables are storing the correct values. Moreover, we manually tested the algorithms by drawing out the algorithms using small test cases through unit testing.

4 – Processing Input and Output Data

The parser (C#) was used to generate a text file for each configuration with the coordinates of robots and polygons converted into a valid format which can be used by the Python visualiser and RRT.

The python program was used to apply RRT to find a collision free path between two robots selected by the greedy passive time algorithm. The RRT program will produce a solution text file with the paths for each robot. This text file was used by the python visualiser to display the path each individual robot took.

5 – Distribution of Work

We split the design and implementation workload so that we can try out different algorithms to solve the problem. Sam and Sim both worked on trying to implement the visibility graph using Pyvisgraph and also the in-order and greedy claim and greedy passive timed algorithm. William and Kazuma both worked on implementing the rapidly exploring random tree algorithm to generate the path between the robots and also to avoid the obstacles. Sam also focused on processing the processing data, automation at a number of stages and finding what two robots should be connected. Kazuma also implemented the visualiser using Matplotlib.

6 - Repository

All our work during this scenario week can be found on in a GitHub repository linked below. It has the different algorithms that we implemented and the results of each algorithm that we implemented.

https://github.com/kiriphorito/MoveAndTag-Manticore

7 - Reference

[1] - "RRT [Online]. Available: https://github.com/AtsushiSakai/PythonRobotics/tree/master/scripts/PathPlanning/RRT [Accessed: 23-Feb-17]

[2] – "The Shapely User Manual". Available: http://toblerity.org/shapely/manual.html [Accessed: 23-Feb-17]

[3] – "matploblib" [Online]. Available: http://matplotlib.org/ [Accessed: 23-Feb-17]

[4] – "PyVisGraph" [Online]. Available: https://github.com/TaipanRex/pyvisgraph [Accessed: 23-Feb-17]