**COMP205P: Move-and-Tag Competition Scenario Week Report**

**Group Y**

by:

**Matt Policane**

**Yll Kelani**

**Fasbeer Eskander**

**Phoom Yenbamroong**

# The UCL Department of Computer Science

# February 24, 2017

[**1 Introduction**](#_axaori7axkk1) **3**

[**2 Algorithms**](#_l1j6zrl8awq0) **3**

[2.1 Geometric algorithms](#_86o3wl893yfz) 3

[2.2 Language and libraries](#_gowa64tmn1f5) 3

[2.3 How were the input files processed, and output files produced?](#_v51txpt0vecs) 4

[**3 Visualisation**](#_cm29fet1noru) **4**

[**4 Complexity and observed runtime**](#_l18a557cb76) **4**

[4.1 Complexity](#_vdw6cp38jwrx) 4

[4.2 Evaluation](#_bliw15w6dmmv) 5

[**5 Team structure**](#_wrutk4bhv4c7) **5**

[**6 Conclusion**](#_kuwuax199qii) **5**

[**7 Our Git Repo**](#_js3y8752hxbn) **6**

# 

# 

# 1 Introduction

As part of the COMP205P module, all students, in groups, were required to design and implement algorithms capable of navigating a starting robot to wake up the other robots whilst at the same time avoiding obstacles. We were required to create a visualiser in order to represent those paths taken by the robots during the process by indicating the starting robot, the paths taken by the robots and how they avoided the obstacles.

# 2 Algorithms

### 2.1 Geometric algorithms

Our first working program found a path between multiple robots with no obstacles. These were the conditions for the question in the set. What the program did was find the distance between all the robots and choose a path by picking the shortest distance between an awake robot and the asleep robots. We did not implement parallelisation here so only one robot “moves” at a time.

Our next and main solution worked for multiple robots navigating around obstacles. We represented the space the robots move in as graph with the nodes in the graph being the locations of the robots at the start and the vertices of the obstacles. We form edges between the vertices of the obstacles to represent the shape of a polygon. This representation is called visibility graphs. To join the nodes with edges we used a naive algorithm. This algorithm compared every pair of nodes in the graph and checked if an edge formed between them would cross the wall of an obstacle or occur completely inside an obstacle.

To find paths within the graph we used breadth-first search. It would have been simple to extend our implementation to Dijkstra or even A\* search but we had trouble with our implementation of the graph.

### 2.2 Language and libraries

We decided to use Python in order to solve the problem we have been given as we wanted to learn Python better. Our research also showed that there were libraries available that would help us solve the problem, and hopefully save us time.

The libraries we used were SymPy, NumPy, Shapely and matplotlib. We used NumPy for convenient functions like arange which allow iterating with float size steps. Matplotlib was used for the visualisation of the program. SymPy and Shapely were used for geometric representation and calculation. SymPy was more inefficient than Shapely.

### 2.3 How were the input files processed, and output files produced?

The input files were processed by reading each line of the .Txt file that was supplied to us. We created a function that could output a specific line from the file so that we can get the data required for each question. Each line was then parsed in order to turn what originally was a string into two lists of tuples - one containing coordinates of the robots and the other containing coordinates of obstacles. We therefore created a further two functions in order to achieve this. We were able to use many of the built in python features for this such as list comprehension. In order to output the list of obstacle coordinates some extra work was required due to the fact that one obstacle has many coordinates separated by a semicolon. Therefore we had to process all of the coordinates before each semicolon as a separate obstacle which we would add to a final list, creating a list of lists of tuples.

The output files were produced by simply writing our solutions to a text file in the format required by the server. This was achieved using built-in python functionality.

# 3 Visualisation

For developing the visualization application we looked into different utilities to be used, including d3.js, processing, and matplotlib.py. After we settled on using Python for our geometric algorithm, we decided that using matplotlib.py would be the wisest option. The plyplot framework was used to plot the coordinates into the graph and also used the artist module to draw the obstacle polygons,robots and different path lines. Algorithms were developed to interpret the robot.mat text file and store the coordinates of robots and obstacles into tuples within lists of lists. These were then passed onto functions to draw the polygons and robots. The solutions were then outputted into a different text file with the data being kept in arraylists which were then passed into the function to draw the paths starting from the 1st robot and assigning different colours for each path. To be able to view the graphs without the paths, another application was created to be able to see how the obstacles and robots were organized. The application prompts the user for the question number to be visualised on the command line and takes the integer value and parses it with the information received from the text file to visualize the graph.

# 4 Complexity and observed runtime

### 4.1 Complexity

The complexity of our program was suspected to be O(n^3) because of the algorithm that connects all the nodes in the graph with edges. It uses nested for loops in order to enumerate the subsets of the set of nodes. For every node in the graph, it checks every other node in the graph to see if an edge can be formed according to the conditions of the problem.

### 4.2 Evaluation

The algorithm ran very slowly when there were a lot of nodes to graph. This led to us finding other solutions for the problems that contained the most robots and obstacle vertices. An example of one of these solutions were to contain every obstacle within a 4 vertex square as to reduce the amount of nodes that we had to graph.

# 5 Team structure

We divided the workload into four main categories: algorithms design, algorithms implementation, algorithms enhancement and visualiser. We started working as a team on deciding what algorithm to be used before modifying/designing it to be suitable for the project. While half of the team was working on implementing the algorithm, one person attempted to enhanced the algorithm in order to speed up the runtime and the visualiser was responsible by the other team member. Despite a clear split in the workload, we all lent each other a helping hand when it was required.

# 6 Conclusion

D3.js is a very interactive and colorful visualization library but the choice of using python and matplotlib made it easier for us to synchronize our work and coordinate with each other on the algorithms used and help each other when it was required.We learned how to utilise matplotlib visualization tools. Matplotlib has the basic tools to be able to visualise the graph, robots, paths and obstacles and can also include animation to make interactive and colourful figures.

One of the first problems we had was the time taken to run the program. Using the implementation already described, we were using SymPy to represent the obstacles as polygons and checking if potential edges cross the walls of a polygon or are entirely contained by an obstacle to decide if it was contained within an obstacle. Initially we achieved both with one test by simply iterating over the point in a line and checking if any of those lines occur in an obstacle. The problem with this was that for certain questions the interval with which we tested point wasn’t small enough and when we did make it small enough the program would take way too long to run.

We then implemented those conditions separately with SymPy and it ran a lot faster but not fast enough to complete all the questions. We then did the same implementation with another library called Shapely. This was a lot more efficient than SymPy and sped up the program a lot but was still not enough. As we could not see another way to iterate through the nodes and find edges we thought of an idea to reduce the number of nodes that we iterate over. We would represent every obstacle with a 4 vertex square. The problem we ran into was then there were situations when a robot would be encased within an obstacle. We set up a conditional as to leave an obstacle as normal if it contains a robot.

In conclusion, this week we have seen how to view an abstract problem and attempt to solve it in an efficient way. We take from the project the fact that in order to achieve this it is important to start of solving a simpler sub problem and keep building up until the problem is solved. We hope to use these skills in future projects.

# 7 Our Git Repo

<https://github.com/officialmatt/move_and_tag>