Team R: Ali Asim, Collins Ryan, Wu Mengyang, Xia Hai

Implementation Report

Language and Libraries

Language: Java

Reason: This decision was made partly due to our familiarity with the language but more importantly because Java's capability of dealing with each stage of the project. These stages include parsing the input, algorithmic processing, output to text as well as visualisation.

Libraries: We did not use any external libraries to aid our project for both the algorithm, input + output processing, and visualisation. Albeit, our algorithm was influenced by the greedy delayed algorithm made by Mdoescher[1] to solve the freeze-tag problem.

Algorithms

Delaunay Triangulation: We used this algorithm to generate a graph from all the vertices present including all of the robots and the edges of the objects. Where the paths present in the this graph are only added if there is no intersection with the obstacles.

Dijkstra's Algorithm: We use this algorithm to find the shortest paths from a source vertex to a target vertex, where these vertices could again be both robots, and obstacle vertices. Dijkstra's whilst not the most efficient, is unlike A* guaranteed to provide the optimal path (shortest).

Greedy Delayed Algorithm: The greedy delayed algorithm involves looking for the closest sleeping bot for a given awake robot and then verifying that the awake bot is the closest awake bot. This takes more time than just simply iterating through a robot list and each robot claiming its closest sleep bot automatically. However, it leads to a more optimal solution as we know that a sleeping robot is claimed by the robot nearest to them.

Complexity and Time

n - the number of nodes in the graph, including robots and vertices of obstacles

Input - O(n)

Construct the connected graph - O(n^3)

- enum all the possible node to node straight path - O(n^2)

    - check intersection with exist edges of obstacles - O(n)

Dijkstra for the shortest path between all the robot nodes - O(n^3)

- From each node of robots - O(n)

    - Shortest path from one source - O(n^2)

Greedy delay algorithm for freeze-tag problem - O(n^3)

- Fill in the target list - O(n)

    - Calculate distance to nearest robot for all awake robots - O(n^2)

    - Find which robot will reach it's target first - O(n)

    - Update acquired distance for all awake robots - O(n)

    - Reset robot that made the jump - O(1)

Output - O(n)

Total time complexity - O(n^3)

Testing

In order to test our solution we followed the strategy listed below:

- We wrote the auxiliary functions that we thought that we needed, and ran them with several easy to check test cases that we designed based on the early solutions i.e. they could easily be confirmed manually.
- We then in effect climbed further up the algorithm hierarchy, using these auxiliary functions with their respective higher level algorithm, and again ran them with test cases generated from the problems sets that could easily be confirmed from manual checks.
- Once we got all of the main functions mentioned above working, we implement the visualisation methods, and ran the problem sets individually with all the algorithms collaboratively running, allowing us to identify collisions and other issues.

Processing output and input

Input

The input “robot.mat” file was parsed using regular expressions. We used several expressions which replied upon this core expression: (-?[.0-9E]*) which allowed us to identify floating points numbers in the format provided, and simply call Double.parseDouble(string) in order to retrieve the values which would eventually be stored in our ArrayLists.

Output

The output was automatically generated by looping through every path within our solution arraylist and printing these out to a text file. While this would output for a single question, we looped our main algorithm to output all solutions to a single file.

Workload

We had first researched into the problem together and discussed possible solutions. Then we collaborated to design a valid version of the algorithm while considering possible flaws. Finally, we each worked on different parts of the algorithm including inputs, outputs, visualisation, implementation of the solution. Some of us also worked on an alternative version of the solution when our initial solution was insufficient to provide valid answers to all problems.

Tools

Development:

For actual code development, we have opted for text editors or IDEs to our liking. Here is a list of the tools we used.

IntelliJ IDEA: IntelliJ is a popular IDE used by everyone on our team for developing, running and debugging the code.

Notepad++: Some of us also had simple text editors such as Notepad++ open for tampering with the input file and checking output

Communication:

Discord: Discord is a online chatting platform similar to Slack, although Discord’s voice communication capability is what appealed the most to us. It also offers a selection of other valuable features such as image, file and highlighted code sharing. All of our team members were active on this platform throughout this scenario week.

Sharing:

Cosketch: Cosketch is an online sketch sharing tool, we occasionally used this to collaborate on visual ideas.

Codeshare: Codeshare is an online code sharing service that we employed when two or more members of the team were working on the same portion of the code implementation

Google Drive: We used Google Drive for general file sharing and organisation.

Github: Github was our choice for repository hosting.

Surprises

Input

We used regular expressions to parse the input “robot.mat” file. However we did not anticipate the occurrence of the character “E”(used to denote ^-10) in the input file which appeared twice. This caused problems for two of our solutions. We simply adjusted our input function to account for “E”s to fix this problem.

Algorithm

Our algorithm for determining whether a path is crossing an obstacle thus making it invalid depends on making the assumption that all vertices of any obstacle is given in anticlockwise order. This allows us to determine the area that the obstacle encloses. However, in fact, some obstacles' vertices are given in clockwise order. This caused invalid solutions as some invalid paths were deemed valid by the programme. To fix this issue, we implemented a maths algorithm[2] to determine whether the vertices of the obstacle was given in clockwise or anticlockwise order. If clockwise, then the order of the vertices are reversed to mimic an anticlockwise order.

Precision

When using the same precision for all problems, the server rejected some solutions stating that there were intersections although the solution seemed feasible on the visualisation. We solved these by adapting the algorithm to select a suitable precision for each question. Also in our final submission, the result for problem set 27 was from an archived result that we were unable to recreate (as it was dependant on a particular precision which we manipulated further on). The result for problem set 1 was done manually because we could not manipulate our algorithm to produce an equivalent answer due its greedy nature..

Repository

https://github.com/fenixisonfire/MoveAndTag

References

"Github - Mdoescher/Freezetag: Visualization Of Several Algorithms For The Freeze Tag Problem Aka Robot Swarm Awakening". Github.com. N.p., 2017. Web. 22 Feb. 2017.

"How To Determine If A List Of Polygon Points Are In Clockwise Order?". Stackoverflow.com. N.p., 2017. Web. 23 Feb. 2017.