**Scenario Week 4: Team Sloth Implementation Report**

*Alexander Xu, Jasmine Lu, Yee Chong Tan, Jamie Law*

**Languages (and libraries) used for implementation**

Our team has prior experience to several languages, including C, Java and Python. However, we ended up using Java to implement our solution (for the algorithms and the visualisation) as we are more familiarised with the language and it’s suitability for representing a graphical interface via Swing. This includes using Java for parsing in the text files and putting the data into variables (regex in Java to filter the input). We used an external library as a basis for checking if a line segment crosses a polygon (to determine whether a point is within the polygon) - *http://www.codeproject.com/Articles/371959/A-brute-force-approach-to-check-if-a-line-segment*.

**Part 1 algorithms**

For part 1, we implemented a method of storing each vertex as a node. Due to our file input processing, our application automatically reads in the question set for part two and identifies the list of polygon vertices. Upon specifying a question to load, then the data for is loaded into the list of nodes and a set of temporary lines connecting the polygon is also added, assuming a connection can be made (explained later).

After this, each vertex is checked again to see if it can be connected to all of the other vertices of the polygon. If a connection exists (see paragraph below) between a pair of vertices, then a connection is added to another node array list (connectedNodes).

As explained previously, we used an external library that uses the idea of line segments to identify whether a point is within the polygon. A segment is considered to be inside a polygon if **every** point of the segment is inside the polygon (end points of the segment can be on the edge of the polygon). The library originally used only the midpoint of the line, but we adapted it to checking 10 segments for each line, therefore improving accuracy.

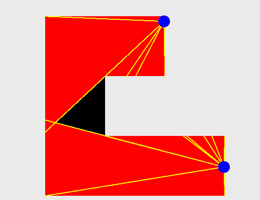
Upon completing this, the application iterates through the nodes and their respective connectedNodes list to identify the biggest connected size – add more here.

**Part 2 algorithms**

For part 2, we used our visualisation to identify areas not covered by guard vision. Due to our file input processing, our application automatically reads in the question set for part two and identifies the list of polygon vertices and list of guard locations.

Upon specifying a question to load (via a menu button for part 2), the application uses the visualisation to draw the respective polygon and adds the guards, which in turn adds the line of sights (shaded areas of vision) for them. The lines of sight are calculated by drawing 3 lines to each vertex from the guard (with each line being minimally different to improve accuracy/reliability) and joining the intersection points together. If it comes across an edge, then it counts as an intersection point to which it stops the line of sight from going further. From this, we are able to see areas which are not covered by the guards in each question set.

The image below shows what happens when question 1 is loaded (blue dot indicates guard, red area represents guard vision, black area represents unguarded vision):

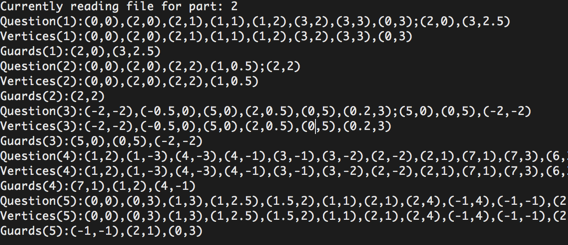


**Algorithm complexities**

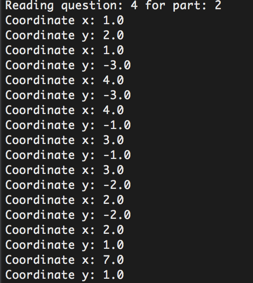
**Algorithm for testing guard sets**

**File processing**

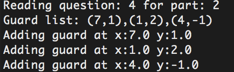
Our visualisation application has two separate options for selecting a gallery to display (one for part one, which specifies only the polygon, and one for part two, which specifies a polygon and guard locations). These use a very similar algorithm, except that part two requires the **string to be split where there is a semicolon** to identify the list of polygon vertices and the list of guard locations, whereas part one only has a list of vertices. These two substrings are then parsed into the following algorithm.

Upon starting either application (part one algorithm or visualisation), our implementation reads in the plaintext file that was provided and eliminates unnecessary data (such as whitespace and the question number) to identify a list of coordinates (points) for each question by filtering all non-digit characters (excluding decimal numbers) and the question number itself. The image below shows a snippet of the application reading the question file for part two (vertices and guards) and the lists being split into two.

Upon calling a question to open, the question data is **split via regex** and stored in a coordinate list (vertex or guard) that is called upon to display the graphics (our program allows the user to specify which question/gallery to show). The image below is an example for when the user requests to open question 4 from part 2, which shows the coordinates of the vertices list after regex splitting:



As the requested question is in part 2, then it will also do the same for the guard list:



File exporting is dealt with by using the data structures and a FileWriter to output a file containing the answer for the respective question. This is easily done by initially writing the team name and password, and then iterating through the data structure to write each location (in the correct format), while ensuring that the coordinates have been scaled down accordingly (due to magnification for visualisation).

**Workload split**

Our workload was split fairly, with roles being allocated accordingly:

* Alex dealt with implementing the algorithms into Java code and the visualisation of the galleries.
* Jamie dealt with file processing (input for part one/two and output), as well as composing the report.
* Jasmine dealt with analysing algorithms that could be used to generate the optimal answer.
* Yee Chong dealt with finding optimal algorithms for identifying the least amount of guards required.

**Github repository link**

https://github.com/lujasmine/teamsloth