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

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

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

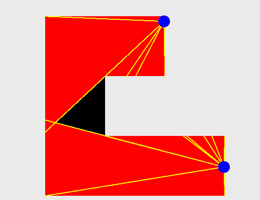
Our team has prior experience to several languages, including C, Java and Python. However, we ended up using Java to implement our solution as we are more familiarised with the language and it’s suitability for representing a graphical interface. This includes using Java for parsing in the text files and putting the data into variables (with the aid of regex in Java). No external libraries were used; we only imported Java libraries.

**Part 1 algorithms**

Initially, our team planned the first step to plot the polygon given the set of vertices. This was relatively easy to implement, as it involved reading in a text file of inputs and putting said inputs into a set of vectors. These vertices are then plotted sequentially with lines being drawn accordingly, with the polygon being filled after completion.

**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. 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):

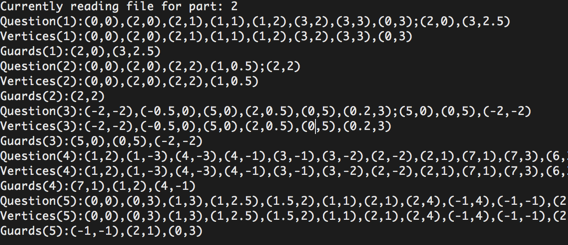


**Algorithm complexities**

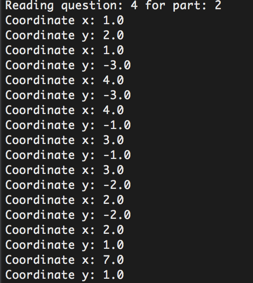
**Algorithm for testing guard sets**

**File processing**

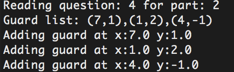
Our visualisation application has two separate options for selecting a gallery to display (one for part one, which specifies only the polygon, and one for part two, which specifies a polygon and guard locations). These use a very similar algorithm, except that part two requires the **string to be split 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:



**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 identifying galleries by human intuition while 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