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 Problem3

1.1 Introduction3

1.2 Finding the Solution3

1.3 Visualisation

2. Algorithms & Complexity

3. Testing Environment

3.1 Penetration Testing Overview

3.2 Web Crawling

3.3 Resource Discovery

3.4 HTTP Proxy

3.5 Password Brute Forcer

3.6 SQL Injector

4. Processing Input and Output Data

5. Distribution of Work

6. Repository

7. Reference

1 – Move-and-Tag Problem

* 1. – Introduction

Snippets! is a web application that allows users to post snippets and store files on the web server that is based on the Google Gruyere application. Snippets! meets almost all the requirements that was given. So it has a login and registration system and users can create snippets which can be seen by all. Each user has their own profile and can change a number of information such as profile picture, password and even the username. Some users can be administrators and have the power to give or take away permission of a user to post and turn other users to administrators. And lastly, there is a cloud based storage area.

We did have a link on the user’s homepage that the user can give out but after going through the vulnerabilities and began hashing user’s passwords, this was no longer possible. The only other specification that we missed was the ability to change your profile homepage colour.

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, 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 (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.

1. – 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.

* 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 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.

* 1. - Freeze Tag

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 smoothes the initial RRT path so straight paths are produced.

* 1. - Other Alternatives

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*](https://github.com/kiriphorito/MoveAndTag-Manticore)

7 - Reference

* “Web Application Exploits and Defenses.” [Online]. Available: http://googlegruyere.appspot.com/. [Accessed: 9-Jan-2017].
* C. Martorella, “Learning Python Web Penetration Testing,” Lynda.com - from LinkedIn. [Online]. Available: https://www.lynda.com/Python-tutorials/Learning-Python-Web-PenetrationTesting/521198-2.html. [Accessed: 10-Jan-2017].
* OWASP, “OWASP Top 10 2013-Top 10” [Online]. Available: https://www.owasp.org/index.php/Top\_10\_2013-Top\_10. [Accessed: 9-Jan-2017]
* PHP, “Safe password hashing” [Online]. Available: http://php.net/manual/en/faq.passwords.php. [Accessed: 14-Jan-2017]
* Veracode, “OWASP Top 10 Vulnerabilities” [Online]. Available: <https://www.veracode.com/directory/owasp-top-10>. [Accessed: 12-Jan-2017]