Assignment 2 Write up

Student ID: 200110436

Unity ID: ychen74

Name: Yi-Chun Chen

Outline

Brief Description about Program Logic
Analysis and Implementation Description
Part 1: Small and Large Graph
Part 2: Dijkatra's and A*
Part 3: Comparison of Heuristics
Part 4: Integration
Appendix

Brief Description about Program Logic

This program was developed according to following components:

- Basic class for kinematic data and behaviors (Only related are listed)
  - KinematicData and SteeringData: The two classes are the data structures which contains kinematic and steering variables—position, velocity, accelerations and so on.
  - KinematicOperations: This class provides basic operations of kinematic variables, such as computing velocity from acceleration, compute next position, etc.
  - Seek_Steering_New: This class implements steering seek with arriving.
  - TimeControler: This class is used to compute elapsed time to control decision rate.
- Graph Data and Graph Algorithms (for each part of assignment)
  - GraphData and GraphGenerator will automatically generate the link edge between input nodes. The logic of graph generation will be explained in afterward sections.
  - Dijkstra: This class implements the Dijkstra algorithm, and the inside method computeDijkatra will return the result of a single step.
  - AStar: This class implements the AStar algorithm. It takes heuristic function as input to alternate different heuristics.

The program logic is simple. It can be divided by 3 parts: graph generating, algorithm computing, and the integration of path finding and path following. In the first part, the program loads the coordinates of obstacles and the position of nodes from some files, and then generates the edges between nodes to form a graph. The detail logic of graph generating will be described in Analysis section. The second part of this program is the implementation of the two graph algorithms. Both required Dijkstra and different heuristics A-Stars are in here. The final part integrated all things together. It shows that the character will find a nearest point on path and follow the computed route to reach goal.

Analysis and Implementation Description

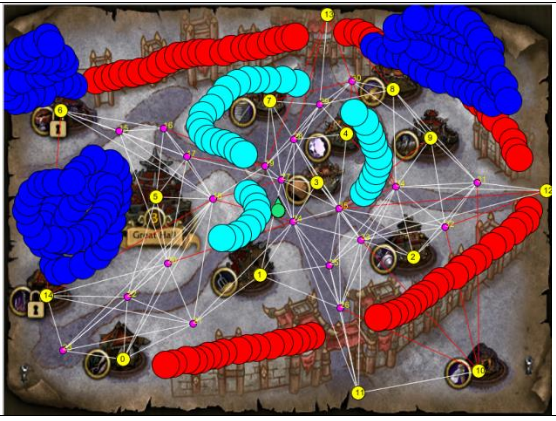
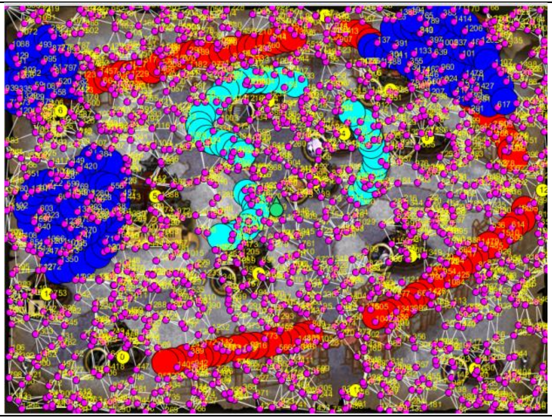
Part 1: Small and Large Graph

The content is this part is the strategy that how the small and large graphs are created, and what are the meanings behind them.

Strategy:

Figure 1 and **Figure 2** represent small and large graph, the reference picture is the Garrison map of Horde side in World of Warcraft (http://www.wow-garrisons.com/garrison-gold-making-and-farming-guide-6-2):

The red, cyan, and blue points represent different types of obstacles on the map. The yellow points are work stations, the magenta points are reachable points on the road. The white lines are edges between nodes and the read lines are edges which overlap with obstacles.

Figure 1: the small graph, with 36 nodes. Link to 5 nearest nodes.	**Figure 2**: the large graph with 1516 nodes. Link to 5 nearest nodes.
	

In this program, the generation of these graphs includes three steps:

1. Set the positions of nodes and obstacles.
   - **Small graph:** The nodes and obstacle positions in small graph are generated manually. According to the height level of obstacles on the map, they are marked with different colors. The walls are marked in red, the small hills are marked in cyan, and the higher hills are marked in blue. Then, each working station (the cabins) in the Garrison is marked a target node, labeled from 1 to 16 (yellow). The road nodes (magenta) are reachable position on the map.
   - **Large graph:** The large graph inherits the obstacles and target work stations from small map; the reachable positions on the map are also marked by magenta and are generated randomly.
2. Create edges according to distance between nodes.
   The edges are created by linking the nearest m nodes for each node. In other words, a node will look up the node list and find the nearest m nodes to mark them as neighbors. By this strategy, the map will have $\frac{\text{number of nodes}*\text{m (find nearest m nodes)}}{2} \leq \text{number of edges} \leq \text{number of nodes} *\text{m edges}$.
   - **Small graph:** The Figure 1 is the graph with 36 nodes (yellow target nodes + magenta road nodes), and linking to 5 nearest nodes. So the number of edges is in-between 90 to 180.

    - **Large graph:** The Figure 2 is the graph with 1516 nodes (target nodes + road nodes), and linking to 5 nearest nodes. So the number of edges is in-between 3790 to 7580.
3. Decide weights of edges by the length of edges and by checking the overlap with obstacles.
   To prevent zero weight, the strategy of creating edge weight is simple. The weights on edges are basically decided by the distances between nodes, but difference types of obstacles will add different weight on edges. Therefore, if an edge is overlap with an obstacle the weight will become:
   $\text{Edge Weight} = \text{length of the edge} + \sum weight\ of\ overlaped\ obstacles.$
   By assigning large value on obstacles, this method is able to prevent character to choose the path which is blocked by obstacles.
4. Check overlap of edges and obstacles.
   The method used in here is to check the shortest distance between an edge and obstacles. If the shortest distance is smaller than the radius of obstacle, then check whether the intersect angles are acute angles: Let the two side of an edge is P1, P2. And the Obstacle point is P0.
   The cosθ of {(P0 - P1), (P2 -P1)} and {(P0 - P2), (P1-P2)} should be positive real number.

*Part 2: Dijkstra's and A**

In this part, the program implement Dijkstra's algorithm and the A* algorithm with simple heuristic.

Strategy:

- **Dijkstra:** The implementation of Dijkstra references to the pseudo-code in textbook section 4.2.3.
- **A*:** The implementation of Dijkstra references to the pseudo-code in textbook section 4.3.3. And the heuristic which used in here is Euclidean distance between goal and current nodes.

Observation:

The test cases used in here are from target node 6 to target node 10 (the target nodes (yellow) remain unchanged in graph, only road nodes (magenta) changes).

- Performance (the cases are generated by same method with large graph; they are in Test Case folder):
    - Fill: The figure 3 in Appendix is the comparison of Fills about the two algorithm. We can see that as the toal number of nodes increase (In experiment, #nodes are 100, 200, 500, 1000, and 1500), although both of Dijkstra and A* has higher number of visited nodes, the growthing rate of A*'s is much lower. Since the number of visited nodes are depends on the graph of test case, This is the the average of 3 experiments with different test cases. The Fill (number of visited node) also influence the computation time. As a result, we can see that A* can not only find the goal more quickly and also reduce more time cost.

- Correctness:
  The figure 4 and 5 shows the result paths computed by Dijkstra and A*. From the result we can see that A* choose a path which has slightly higher weights than Dijkstra (Dijkstra: 651 weight, A*: 656 weight). That is because the heuristic here in A* is the Euclidean distance between goal and current visited node. The Euclidean distance will not consider the weight of obstacles and will lead A* to search the nodes which are closer to the direction of the goal. For example (as the two path shown in

Figure 5), if the goal is in left side, A* will tend to visit the nodes in left side, because the heuristic miss to take the real weight of links between nodes and hence to overestimate the cost of right side nodes. As a result, it will miss some possibilities.

Part 3: Comparison of Heuristics

In this section, the results of A* with different heuristics will be discussed.

Strategy:

- Heuristic 1: Euclidean distance between goal and current nodes.
- Heuristic 2: Mahanttan distance between foal and current nodes.
- Heuristic 3: Random choose a number between 0 to the distance from start points to goal.

Observation:

The figure 6 shows the experiment results which are the Fills of 4 different algorithms, one is Dijkstra, the other three are A* with different heuristics. From the results, we can see that the A* with random heuristics is close to the result of Dijkstra, that might because the expected value of binomial random is 0, so the average result of multiple experiments will lead to similar result to Dijkstra (with zero heuristic value). The second phenomenon appears in the result table is that the growing rates of Fills of A* with Euclidean distance and Manhattan distance are both much lower than Dijkstra and random heuristic. Although the results of the two are lower, they are different. From the result, we can find that the performance of using Manhattan distance is better than Euclidean distance. That might because the Manhattan distance can describe the relation between current node and goal more precision is the situation shown in Figure 7. We can see that although the two blue points both have Euclidean distance R, one of them is actually need more cost to reach the goal. The situation will be ignored when using Euclidean distance, but when using Manhattan distance the farer node will have larger weight. As the discussion in previous section, the Euclidean distance is admissible in some situations, but will lead to bias that it tend to search the nodes which are closer to the goal side and will miss some the possibility, and will overestimate when the possibilities are related to nodes which are farer. As the example shown in figure 8, in that case both Euclidean distance version and Manhattan distance version will tend to choose blue path, since they consider the physical relation of nodes rather than the real relations between nodes and edges.

Part 4: Integration

In this section, the program integrates the path finding algorithm and path following.

Strategy:

- Path finding: Same as part 3. By changing the parameter, this part can use different heuristics.

- Path following: this part is implemented through the seeking in assignment 1.

Once a target is set (mouse pressed), the character will find a nearest node on the graph as its start point, then set the node which is closest to the goal as target node. As a result, the path finding algorithm will generate a sequence of nodes which is the shortest path as seeking targets.

Observation:

The demo video with small graph:

https://youtu.be/GPowpawPm_g

The demo video with 200 nodes graph:

https://youtu.be/5826fets_X8

The demo video shows that the character will compute a path which avoid obstacle and reach the goal.

Appendix

Figure 3. The comparison of Fills about Dijkstra and A* with Euclidean distance as heuristic.

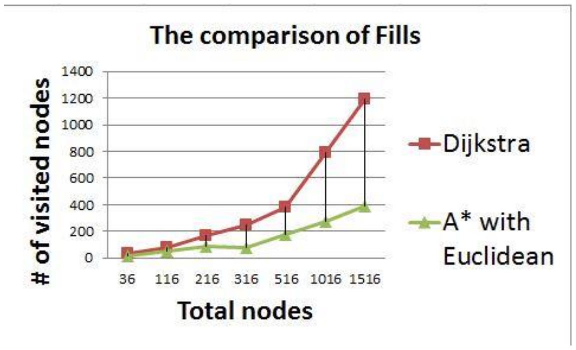
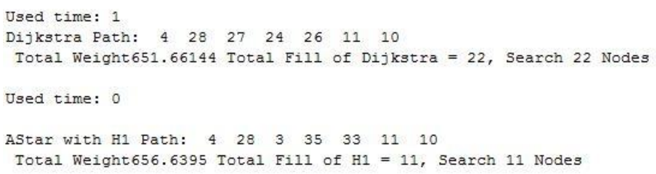


Figure 4. The result paths computed by Dijkstra and A*



```
Used time: 1
Dijkstra Path:  4  28  27  24  26  11  10
 Total Weight651.66144 Total Fill of Dijkstra = 22, Search 22 Nodes

Used time: 0

AStar with H1 Path:  4  28  3  35  33  11  10
 Total Weight656.6395 Total Fill of H1 = 11, Search 11 Nodes
```

Figure 5. The result paths computed by Dijkstra (orange line) and A* (green line).

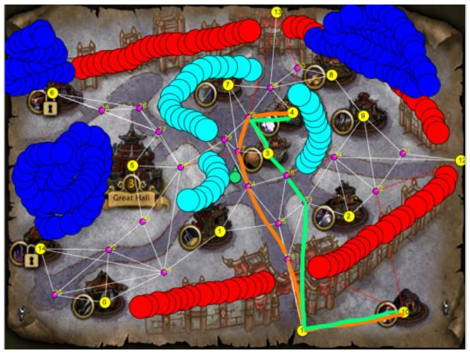


Figure 6. The Fills of different algorithms, Dijkstra and A* with different heuristics.

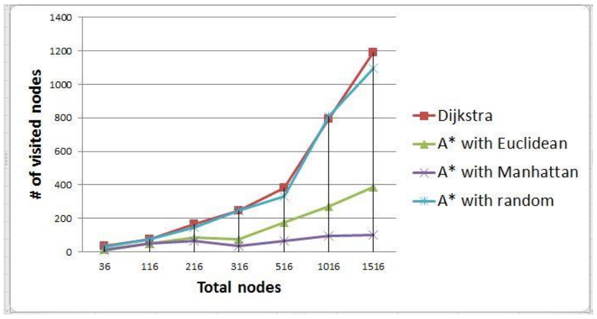


Figure 7. the example to explain why Manhattan distance can be better to present the relation between goal and current nodes. Two blue points will have same Euclidean distance, but different Manhattan distance. The later describes the relation more precisely.

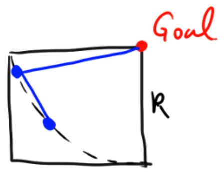


Figure 8. the example to show when Euclidean distance and Manhattan distance will lead to bias of search. Because in each iteration A* will choose an edge with smallest heuristic value, in this case both Euclidean distance version and Manhattan distance version will tend to choose blue path rather than the black one, and might result in find a wrong solution.