Building Game AI

Rishabh Sinha(rsinha2)

Homework2

CSC 584

Graphs:

Graphs are data structures consisting of Nodes or Vertices's, which are connected to each other using, edges of a certain weight. A graph is usually represented as G = <V,E,W> where V are the Vertexes, E is the edges and W represents the Weights.

Each Node in a graph usually consists of information like node id, which helps identify the particular node, as well as Other information which pertains to the specific implementation.

For example a graph representing a social network may contain nodes that information regarding an individual profile.

Small Graph:

As a part of the Homework, I have created a small Graph. The small graph that has been generated represents the Room created for the last part of the Homework Assignment. The Graph Consists of tiles, which are square in shape. Each tile is 100 pixel, by 100 pixel. And the graph consists of 100 tiles.

In the Tile Graph, Each Node represents a tile and each edge represents a side that is shared between two tiles. Two tiles are connected with an edge, if they share a side, and there is no obstacle present in the given tile.

So basically all the tiles which are not covered by obstacles, are connected with their neighboring tiles, if the neighbor doesn't have an obstacle.

To implement the small as well as the large graph, I have created a Node class. The Node class keeps track of the the x,y coordinate information of each node. The x,y value for each coordinate node is the coordinate value at the top left corner of the given tile. Thus the center of the tile is obtained by adding a value of (50,50) to the (x,y) of the tile. The Node also contains the Node id and the Node name. The Node id is used in implementation of A* and Dijkstra Algorithm, And the Node Name is just used to make the debugging easier. The class also contains getters for all of the above attributes as well as a to_string function.

Along with the Node class. An Edge class has been defined, Each Edge has been defined as a pair of Nodes along with the cost of traversing the Edge. For the given Tile Graph, I have used 100, that is the number of pixels between adjoining nodes as the edge cost joining neighboring tiles. The Edge Class also contains getters as well as a to_string function to help in debugging the code.

Finally There is also a Graph Interface that has been implemented, the graph Interface consists of two functions which are getNodes() and getEdges(). The two Graphs which are the small and the large graph Classes

implement this Interface.

Thus the SmallGraph class is where the SmallGraph has been created. The graph has been created as a part of the constructor of the class.
And the Class also implements the functions of the interface. The Nodes containing the obstacles are stored in a list of Nodes, and while connecting Nodes, with edges, We check if both the Node are not a part of the obstacle list then we connect the adjoining nodes. We also take care of the boundary conditions to ensure that the tiles on the first and last rows and columns are dealt with correctly.

The Small Graph has been visually depicted in Figure 1.

Large Graph:
Just Like the small graph, The Large Graph also implements the graph Interface and is a tile graph. However in the case of the LargeGraph, The number of Nodes for the graph is 10000. Further a random variable has been created to get the probability if two neighboring nodes will be connected or not. For this a probability value of 0.7 has been used to connect neighboring edages of the large graph. If a smaller values is used, then it leads to situations where there are 2 or more connected components in a graph rather than a large single connected graph. Thus if we use a smaller value, there can be cases where a path may not exist between two nodes. We can always use a higher probability, but in that case the graph becomes highly predictable because of the standard implementation of tile graphs.
Further the large graph also takes care of the boundary conditions just like the SmallGraph.
The Implementation of the large graph is just to test the Scalability of the Dijkstra as well as the A* algorithm, in terms of the fill that is the number of Nodes searched, the memory footprint which is the max size of the openList and also the execution time of the algorithm.

Dijkstra:
Dijkstra's Algorithm is one of the most popularly used PathFinding Algorithms. Dijkstra however in case of Game AIs does not return shortest path of all points from the source point but just just the shortest path between the source and goal node. In the homework, the Dijkstra's algorithm has been used on both the Small as well as the LargeGraph.

Astar:
Astar Algorithm is also one of the most popularly used PathFinding Algorithms. Astar, like the Dijkstra Algorithm gives the shortest path between the source and the goal node, But unlike Dijkstra it does not progress by minimizing, the actual distance, but progresses further, by minimizing the total value of Distance and the Heuristic function. The Heuristic value is a thumb rule which gives the best guess between a node to the goal node.
As a part of the Homework I have also implemented a working A* algorithm which is also used in the final part of the assignment to navigate the character through the room.

Heuristic:
As a part of A star Algorithm, I have implemented two heuristics, which are the Manhattan Distance as well as the Euclidean Distance,

ManHattan Distance is the sum of the difference between the X coordinates of the goal and the source Node as well as the sum of Y coordinates of source and the goal Node. As a part of the Homework, I have implemented a tile graph, For the tile graph, we know that Manhattan Distance is a consistent as well as an under estimating heuristic, thus it is an admissible heuristic that will always give the best path for an A* search algorithm.

The Second Heuristic implemented as a part of the homework is the Euclidean distance. Euclidean distance is the shortest distance between any two points that is the straight line distance. Once again the Euclidean distance when used as a heuristic for the Astar Algorithm is a underestimating consistent and thus an admissible heuristic which guarantees the shortest path for an Astar Algorithm.

The third heuristic, which, here for the experiments performs the fastest is the difference between the node ids, In the graph, the value of the node ids grows, row wise, thus a node which is diagonally distant will have the max value for the heuristic, and which is at a higher column, by a row is same, will have very low value for the heuristic. The Heuristic is an over estimating heuristic, as for large graphs, the difference between the node ids, can be larger than actual distance. The heuristic is also not consistent as it follows nothing like the triangle inequality. Thus it is an inadmissible heuristic. (However for the small graph, this heuristic will remain underestimating because of lower possible difference between the node ids.)

Comparitive Analyis:
This section of of the write up analyzes the difference between the Astar and the Dijkstra Algorithm as well as the difference between the The performance of the three heuristics used here as a part of the Astar Algorithm. Since Large Graph is a Random tile graph, an new graph of the given size is generated on every run, with a probability of 2 nodes adjacent being connected as 0.7

	Djikstra	A* (manhattan)	A*(euclidean)	A*(index diff)
Input Start and End Are at two ends of Row of a large random tile graph				
No. of Nodes Explored	27059.2	10239.4	14404.2	3648.0
Size of OpenList	263.4	181.0	181.8	168.2
Time(s)	7.741	2.103	2.891	0.650

The Table contains values averaged over 5 runs for each input set

Table 1

	Djikstra	A* (manhattan)	A*(euclidean)	A*(index diff)
Input Start and End Are at two ends of Column of a large random tile graph				
No. of Nodes Explored	27089.0	10650.6	13934.0	12862.6
Size of OpenList	248.8	180.4	185.2	201.2
Time(s)	7.455	2.861	3.237	3.597

The Table contains values averaged over 5 runs for each input set

Table 2

	Djikstra	A* (manhattan)	A*(euclidean)	A*(index diff)
Input Start and End Are at two ends of Diagonal of large random tile graph				
No. of Nodes Explored	39495.4	39425.2	39411.6	17826.2
Size of OpenList	265.4	265.0	283.2	250.8
Time(s)	11.748	15.016	14.653	6.279

The Table contains values averaged over 5 runs for each input set

Table 3

From the above data it is evident that the A star Algorithm has generally both a lower time as well as a lower space complexity than Dijkstra's Algorithm. We can see that When, input and goal are in columns or rows, which are close, The Astar with heuristic as Manhattan and Euclidean distance performs well, This is because the heuristic is able to guide the search algorithm along the correct direction.
However, in cases where the source and goal are at diagonally opposite ends, the Astar performs badly using Manhattan and Euclidean distance. This may be due to nature of tile graph, which allows only nodes which are adjacent in terms of sharing a side be connected, This may be the nature of poorer performance of Astar in this case.

However for the Heuristic, difference in node id. The heuristic performs better than Astar in all cases, However this difference is very stark in case of horizontal distances, that is nodes on the same row, as in the graph, the node ids increase row wise. Thus the very superior performance of this heuristic over the others

Putting it together:

In this part of the Assignment, I combined the Underlying Path finding Algorithm which is the Astar Algorithm, with the movement Algorithm Arrive, to create a scenario, where the character is able to navigate from start to finish, by avoiding obstacles. Each obstacle is stored in a list of Nodes, these nodes are not connected to the nodes of the graphs, thus the

character will never be able to enter a tile which contains an obstacle as it is not a part of the graph.

The Quantization is done by using mod on the coordinates value to generate the node id as the node ids increase in a row major format. Localization is done by using the x,y values stored at the nodes plus an offset to bring the character the center of the tile.

For this, In the draw function of the code, I have called the Astar Algorithm, which, builds the path of Nodes from the current Node to the goal Node, and the character follows the path to the goal node, if another point is clicked, then that point gets quantized causing the path to be updated, and once the path has been calculated using the Astar Algorithm, the path once again localized in form of coordinate points which is delegated to the arrive function to achieve the path following characteristic.

Appendix:

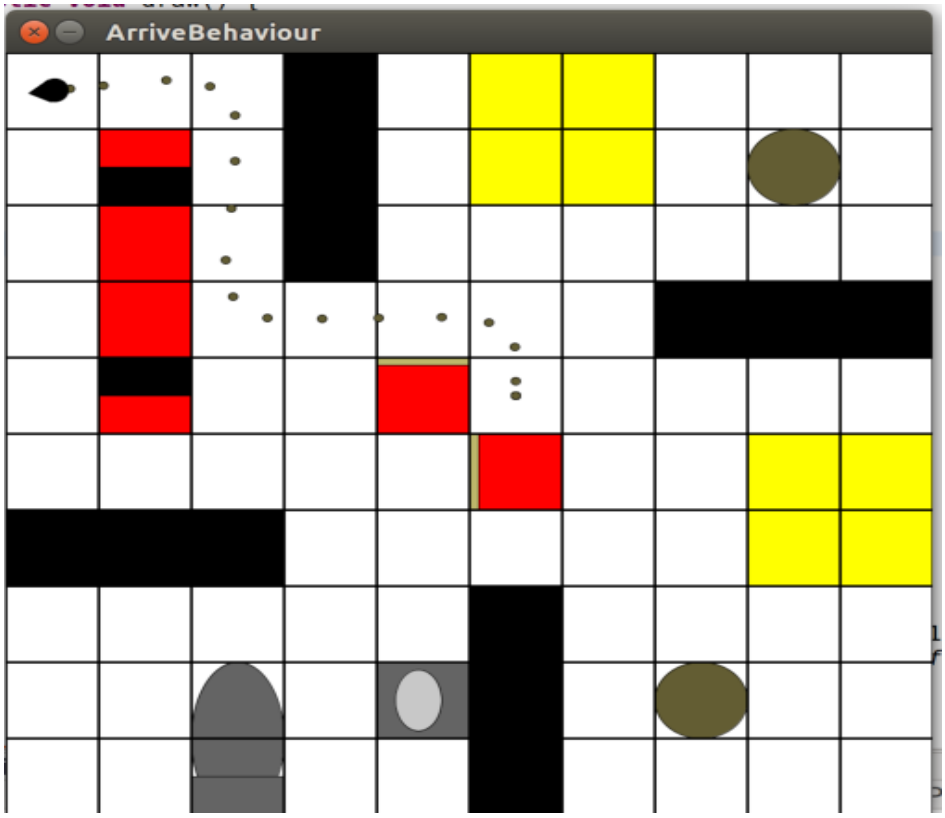

Figure 1:
Figure depicts path finding in action, where the path is depicted by the breadcrumb.

The Figure also depicts the Tile graph, where all white tiles are nodes, two tiles are connected if both Tiles are white and share a side.

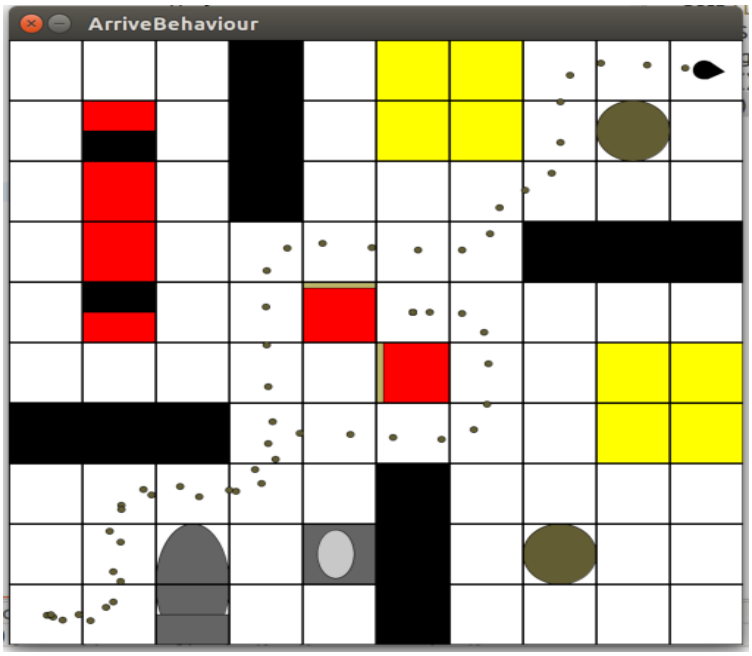

Figure 2
The Figure depicts the path taken by the character from middle to bottom and then from bottom to the top.

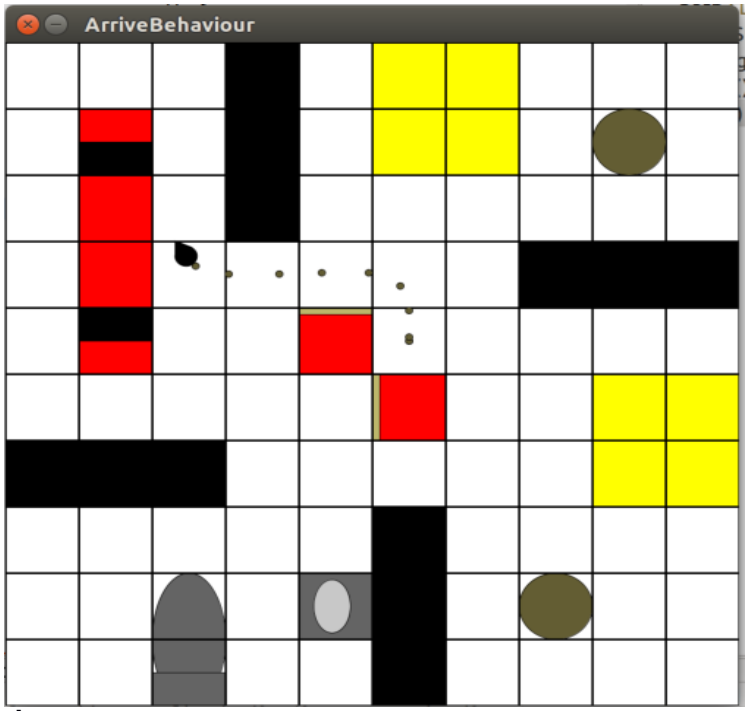

Figure 3
Figure depicts the character taking a turn on its way to the top corner

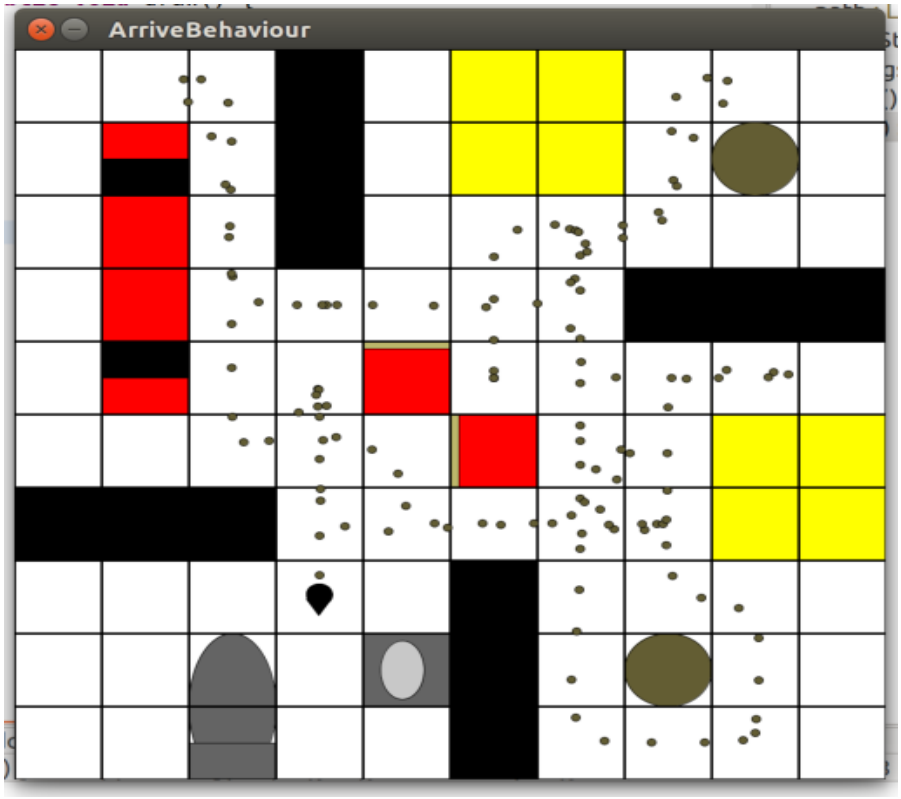

Figure 4
Depicting multiple iterations, passing different values of inputs using mouse click

References:

1. http://www.vogella.com/tutorials/JavaAlgorithmsDijkstra/article.html
2. http://www.processing.org