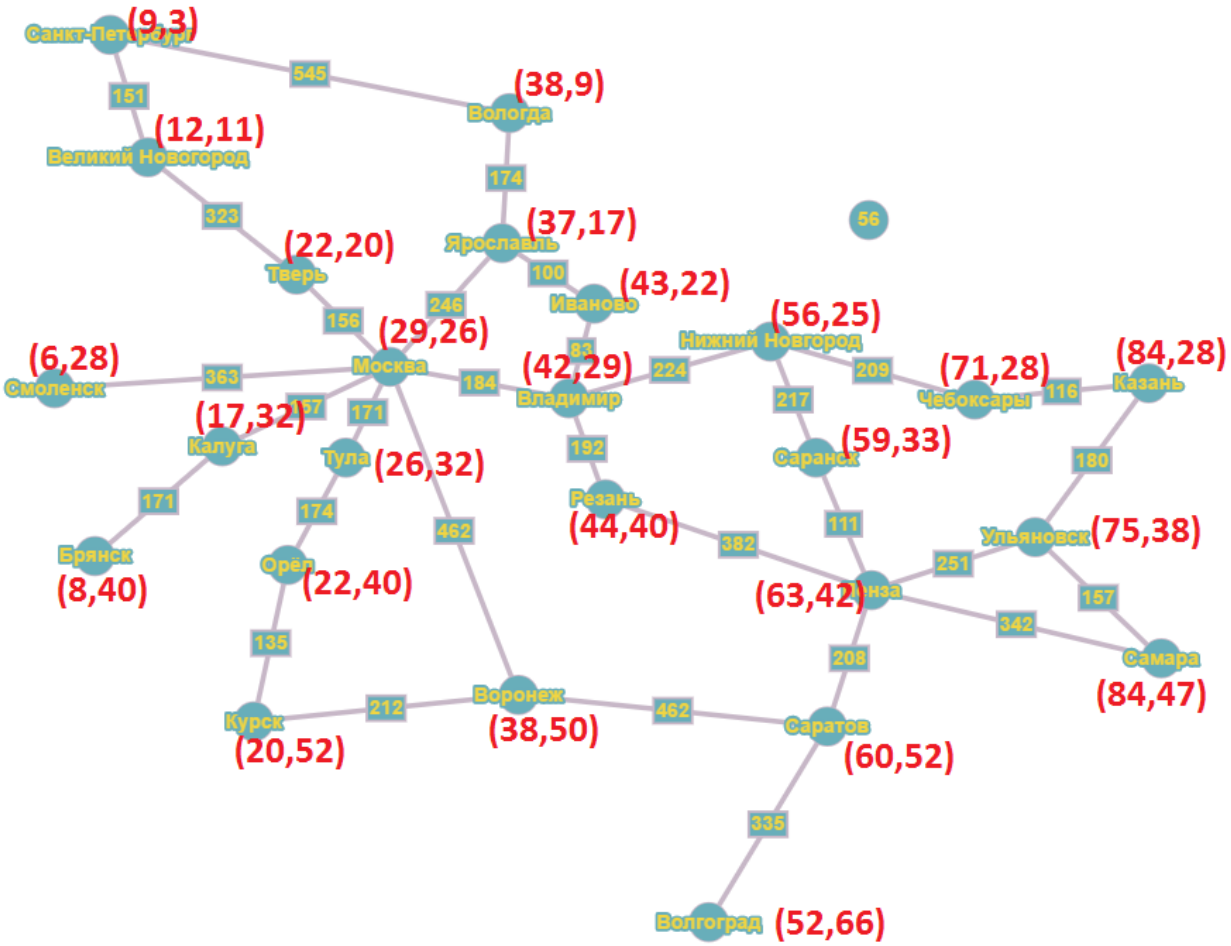
CSC 584/484 Spring 17 Homework 2: Pathfinding and Path Following

Due: 3/27/17 by the start of class

First Steps (10pts)

In this part of the homework we are asked to create two graphs. First which is small and meaningful and second one which is a large graph with tens-of-thousands of vertices and edges.

My first graph is a graph of Center European cities of Russian (Click on Russian for the link).



Each node has an idea corresponding to its position on a ruled graph of 100x100. The way the graphs are defined with its hierarchy is as follows.

- **Graph**
  - List of **Graph-Nodes**
    - Node-Id (an integer value)
    - List of **Connections** (edges with positive weights)
      - Origin of connection (node-Id)
      - Destination of connection (node-Id)

- Cost of connection (i.e. the edge weight)

In the above small graph the node-Id are nothing but the coordinates of the vertices. If the node has the coordinate (52,66) then its node-Id is 5266. This enables us to calculate Manhattan distance, Euclidean distance and node-Id distance. Thus, making us available three heuristics for A* algorithm.

For my second graph, I have written a program to create it randomly by taking input as number of vertices you want for your graph, and constant value between 0-1. This constant value is the measure of number of edges a graph can have. Higher the value, the denser will be the graph. It is a probability that there exists an edge between any given pair of vertices. The values that I prefer (found out by experimenting and playing around it) is 0.5 when number of vertices less than a thousand. As you increase the number of nodes say about ten thousand, the parameter value of about 0.3 is sufficient to generate a dense connected graph. For small graph if vertices less than hundred this value needs to be at least greater than 0.5.

Dijkstra's Algorithm and A* (15pts)

The first thing I noted when comparing these algorithms, was that A* algorithm is fast. Before discussing about the performance, let me first describe my heuristic to you.

I have used a simple heuristic and I called it as node-distance. It is the absolute difference between the node-Id of the graph nodes. For example, if the two vertices have their respective node-Id's as 31 and 87, then their node-distance will be $|31 - 87| = 56$.

Performance Report for small graph:

Note: all the time complexities mentioned are in milliseconds. Unless its mentioned. Path are represented as the set of edges i.e. they are represented in the format (FromNode, ToNode, Weight) which is same as (v1,v2,w).

Example output: After running each algorithm for 100 thousand time.

A* Working. Time taken: 300ms **********THE PATH IS******** 903 3809 545.0 3809 3717 174.0 3717 4322 108.0 4322 4229 83.0 4229 5625 224.0 5625 7128 209.0 7128 8428 116.0 8428 7538 180.0 7538 8447 157.0	Dijkstra Working. Time taken: 519ms **********THE PATH IS******** 903 1211 151.0 1211 2220 323.0 2220 2926 156.0 2926 4229 184.0 4229 5625 224.0 5625 7128 209.0 7128 8428 116.0 8428 7538 180.0 7538 8447 157.0

The tabulated result of running time for variable number of runs is given below (start:903, goal: 8447):

Number of runs	Time taken by Dijkstra	Time taken by A*
10	2	36
100	6	42

1000	19	53
10000	79	102
100000	518	337
1000000	3154	1426

The amount of memory usage for both the algorithms is:

A* Working. Max size of open list encountered: 6 Max size of close list encountered: 9 Time taken: 34	Dijkstra Working. Max size of open list encountered: 7 Max size of close list encountered: 23 Time taken: 2

Performance report for large Graph:

This graph has 10 thousand nodes. In this we get a clear distinction in the performance based on time and memory.

Example output: After running each algorithm (start: 9201 & goal: 1341).

A* Working. Max size of open list encountered: 6533 Max size of close list encountered: 3 Time taken: 298ms **********THE PATH IS******** 9201 1342 4.0 1342 1341 8.0	Dijkstra Working. Max size of open list encountered: 9974 Max size of close list encountered: 536 Time taken: 21145ms **********THE PATH IS******** 9201 8290 1.0 8290 1341 1.0

As seen above, although the path given by A* is not optimal, you can see the difference in time & memory complexities. If we add the heuristic in Dijkstra's output you'll find that A* output is better since the estimated total cost for A* is the least.

Following is the result in tabulated form for the time complexity (averaged over 10 runs) of the algorithms, as the number of vertices in the graph are increased. The graphs are dense if number of nodes < 1000 in a graph

Number of nodes	Avg. Time taken by Dijkstra	Avg. Time taken by A*
10	0	3
100	1	3
1000	237	5
10000	137008	71
50000 (avg. over 5 runs)	340545	2050

Heuristics (10pts)

I have implemented 3 heuristics. First is the node-distance, as discussed before. Second is the Manhattan distance and Euclidean distance. In large graph, I'll be considering a graph with 10000 nodes. This will enable me to follow the same heuristic calculation function for both the graphs, given the node-Id.

Once we know the position of the node, i.e., their x and y coordinates then it is easy to find the Manhattan and Euclidean distance. So, the way to get the coordinates given node-Id is as follows

x = node-Id/100 & y = node-Id%100. For example consider if node-Id is 3456. Then its coordinates will be (34,56). Obviously, the node-Id lies within 0 – 9999.

Small Graph (Start: 903, Goal: 8447):

Reference output using Dijkstra's algorithm:

Path:	903 1211 151.0 1211 2220 323.0 2220 2926 156.0 2926 4229 184.0 4229 5625 224.0 5625 7128 209.0 7128 8428 116.0 8428 7538 180.0 7538 8447 157.0

Heuristics		
Node-Distance	Manhattan-Distance	Euclidean Distance
903 3809 545.0 3809 3717 174.0 3717 4322 108.0 4322 4229 83.0 4229 5625 224.0 5625 7128 209.0 7128 8428 116.0 8428 7538 180.0 7538 8447 157.0	903 1211 151.0 1211 2220 323.0 2220 2926 156.0 2926 4229 184.0 4229 5625 224.0 5625 7128 209.0 7128 8428 116.0 8428 7538 180.0 7538 8447 157.0	903 1211 151.0 1211 2220 323.0 2220 2926 156.0 2926 4229 184.0 4229 5625 224.0 5625 7128 209.0 7128 8428 116.0 8428 7538 180.0 7538 8447 157.0

If we compared the actual total cost of each path, we get the following:

Heuristic	Path Cost
Node-Distance	1796
Manhattan-Distance	1700
Euclidean-Distance	1700
Actual Minimum cost (using Dijkstra)	1700

Thus, node-distance Heuristic appears to be over-estimating in this scenario.

Large Graph:

Large graph is created as described above. The parameters used to create the graph are: numberOfVertices = 10,000; ProbabilityFactor = 0.01 and edge weight lies between [1,19].

Reference output using Dijkstra's algorithm:

Path:	6562 179 1.0 179 7038 1.0 7038 3339 1.0 3339 3578 1.0

Heuristics		
Node-Distance	Manhattan-Distance	Euclidean Distance
6562 3605 5.0 3605 3504 5.0 3504 3657 8.0 3657 3622 4.0 3622 3522 8.0 3522 3501 8.0 3501 3578 11.0	6562 5068 2.0 5068 2571 2.0 2571 2976 1.0 2976 2583 4.0 2583 2178 1.0 2178 3578 17.0	6562 5068 2.0 5068 5879 1.0 5879 4776 1.0 4776 2089 3.0 2089 3269 5.0 3269 3578 14.0

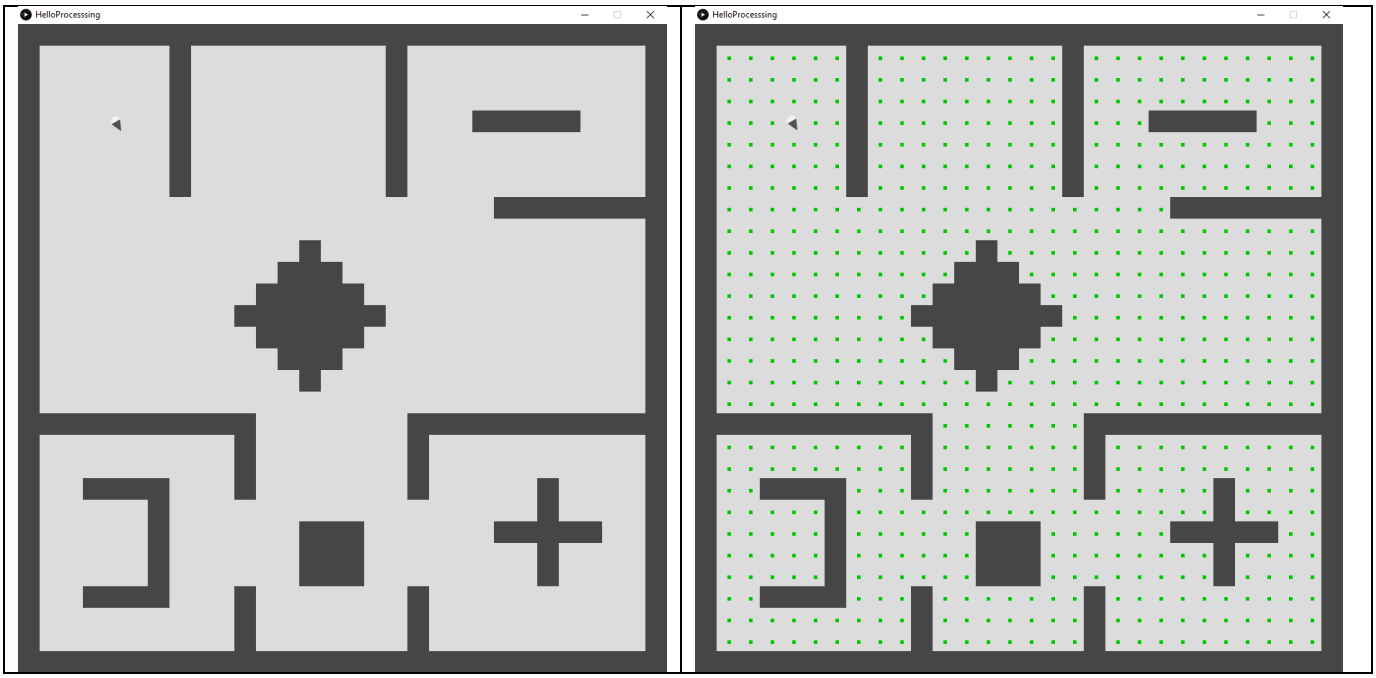
If we compared the actual total cost of each path, we get the following:

Heuristic	Path Cost
Node-Distance	49
Manhattan-Distance	27
Euclidean-Distance	26
Actual Minimum cost (using Dijkstra)	4

Although all the Heuristic appears to be over-estimating in this scenario, the time-complexity is less compared to that for the Dijkstra.

Putting it All Together (25pts):

Following is the game environment used for this part of the assignment. This layout represents the floor plan of my home on left.



The graph is created by dividing the screen into grids. The obstacles are all square shaped, since I found the character difficult to maneuver when obstacles contain sharp edges. Maneuvering over a circular obstacle is easy since we know the radius of the circle and we can restrict the position of character outside it, since it is easy to calculate the normal.

This layout, basically consists of 3 rooms each having a unique type of obstacle, a (fancy) dining table in the dining room, hall and an empty kitchen. The graph I chose to be less dense and incorporated obstacle avoidance. The layout with the graph node is above on right. For obstacle avoidance, I've implemented 2 methods. First is ray casting. In this method, I shoot 2 rays in the direction of motion, with minimum lookahead greater than the size of the object and the other ray is twice of this value. In second method, I treated each collision as a circle of radius 1.4 times the length of grid side.

Issues faced by the first method is that, sometimes object overlaps a little bit over the obstacle. In second method, the character avoids this. But only in one case, it faces overlap. When it must take a tight left U turn. During this event, this collision detection method fails. But overall it gives a smoother traversal of the character around the obstacles.

In Path Following, I'm doing a predictive path following by looking character's position into future. The path Offset value 2 seemed to be fit for my game environment. For path Offset = 1 the character's current position shoots its target position on the path, stuck in a loop and doesn't go anywhere. For path Offset value greater than 3, the character gets stuck into other side of the obstacles and keep approaching the target, which is not accessible directly.

For path finding algorithm I've used A* with Euclidean heuristic. The graph edges follows triangle in inequality. The edge to adjacent vertex is 10 and the edge to vertex along the diagonal is 14(or15). To see the character in action, watch the video file submitted along with this report.