Assignment 2 Report

Che Chang
The University of Utah
che.chang@utah.edu

ABSTRACT
This is the write-up for assignment 2, which covers the implementation of Dijkstra shortest path algorithm and A* pathfinding algorithm

KEYWORDS
Graph theories, Dijkstra, A*, pathfinding

INTRODUCTION
In this write-up, we will demonstrate the result of the two pathfinding implementation (Dijkstra, A*), and also discuss the performance difference between them. We will first apply the two algorithms to directed weighted graphs (one graph authored by hand, the other one downloaded from DIMACS shortest path challenge website), and then quantize the game map we created from the previous assignment into our tile graph, and finally use A* to compute a path from the character boid to our target. We will also apply different resolutions to our tile graph, and observe how that impacts our pathfinding result.

FIRST STEPS
Our First Graph
This is a directed weighted graph representing where I was born: Hsinchu City, Taiwan. To better demonstrate the relative locations of the nodes in this graph, I plotted the graph using an online graph editor: Graph Online (https://graphonline.ru/en/). This graph contains 12 nodes and 22 one-directional edges, formatted according to the DIMACS shortest path challenge.

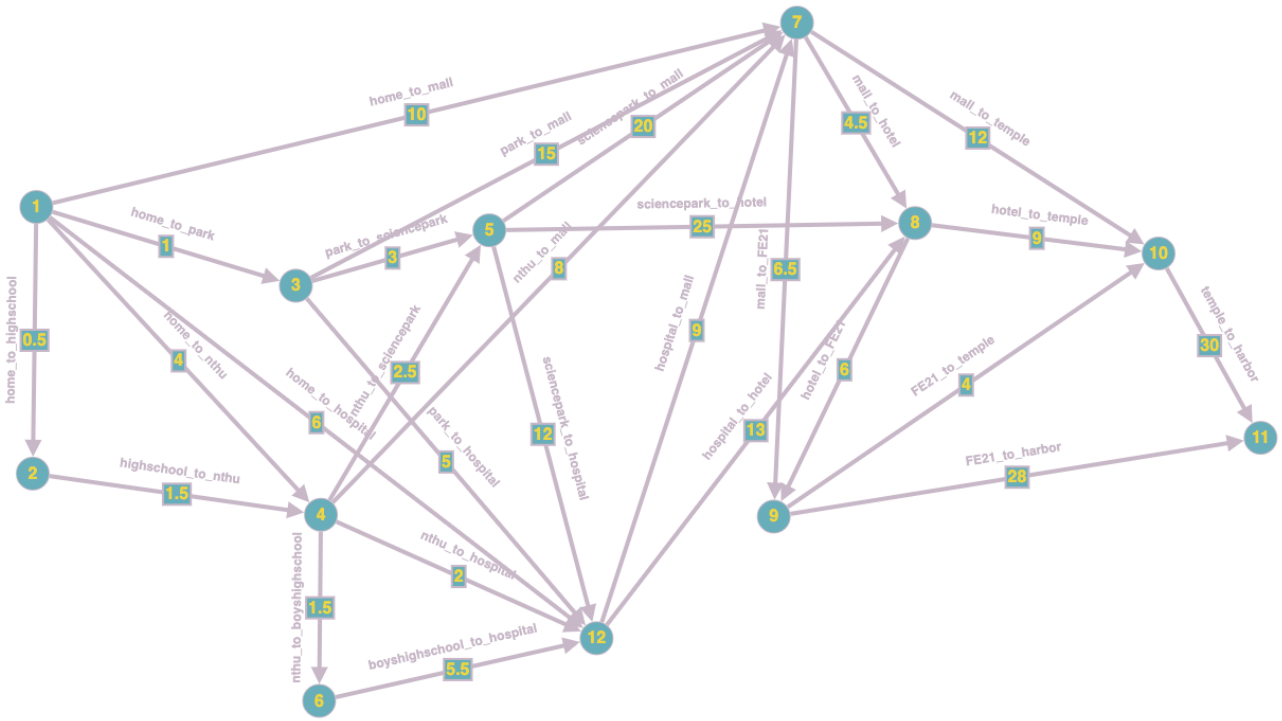

Figure 1: Small directed weighted graph representing Hsinchu City, Taiwan.

Finding a Path with Dijkstra's Algorithm

Now we have a relatively small graph, we can proceed to pathfinding with Dijkstra's algorithm. Let's consider the following input: node 1 as the start node, node 10 as the goal node. The pathfinding result is as follows: `1 => 7 => 9 => 10`.

However, how was the performance when we applied Dijkstra? With the help of `std::chrono` library, we obtained a result of < 1 ms. (using `std::chrono::duration_cast<std::chrono::milliseconds>`, the runtime displays as 0 ms, meaning it's smaller than 1 ms)

Also by stepping into the debugger, we found out that Dijkstra's algorithm visited 10 nodes before completing the path computation.

Finding a Path with A Algorithm*

Now we shall proceed to path searching with the same graph. Note that at this point we don't have real world representation of the graph yet, so nodes in the graph have no world positions, which means we can't compute Euclidean or Manhattan distance as our heuristic functions for now.

However, we can still use random heuristics. There are numerous ways of constructing random heuristics, but here we'll be simply using this: `(ofRandomf() + 1.0f) * 2)`, which outputs a floating point number between 0.0 and 4.0. After setting up the heuristics, the rest is the same, we have node 1 as our start node, and node 10 as our goal node. The pathfinding result is as follows: `1 => 7 => 10`.

The performance did not differ much from Dijkstra's algorithm, we also obtained a run time of < 1 ms from running A* on the same graph. Yet, we obtained a different path when we applied A* to path searching, and it's not the optimal result as Dijkstra's computed. This difference is probably due to the usage of random heuristics.

The random heuristic function generates a number between 0.0 and 4.0, which is quite big compared to the overall edge costs in the graph, and therefore causing overestimation of the total costs. Overestimation would likely lead to A* returning a path with less nodes but not actually the optimal path.

Again by stepping into the debugger, we know that A* algorithm visited 10 nodes before completion.

Our Second Graph

Our second graph is downloaded from the DIMACS shortest path challenge website (http://www.diag.uniroma1.it//~challenge9/download.shtml).
This large graph represents New York City, containing 264346 nodes and 733846 edges. I had to reduce the edge numbers down to 3000 edges (and approximately 1300+ nodes),

because with the original number of edges, our current algorithm implementation won't be able to finish processing.

Finding a Path with Dijkstra's Algorithm

Consider the following inputs: node 1 as the start node, node 934 as the goal node. We apply Dijkstra's algorithm first, and the pathfinding result is: `1 => 1363 => 1358 => 1357 => 1356 => …… => 936 => 933 => 934`. (Total 17 edges in the path) Again using the `std::chrono` library, the run time is 9 ms.

Also by stepping into the debugger, we found out that Dijkstra's algorithm visited 185 nodes before completing the path computation.

Finding a Path with A Algorithm*

Applying A* algorithm, we obtained the pathfinding result as follows: `1 => 1363 => 1358 => 1357 => 1356 => …… => 936 => 933 => 934`. (Total 17 edges in the path).

Note that the pathfinding result is identical to the one Dijkstra's algorithm produced. The overestimation issue mentioned in the previous section is likely not affecting much in this scenario, because the overall costs of the edges in this New York City graph are quite large (1000+), so the random heuristic isn't capable of influencing the final outcome. Finally the run time is 9 ms, which is pretty much the same as Dijkstra's algorithm.

By stepping into the debugger, we know that A* algorithm visited 185 nodes before completing the path computation.

Furthermore, in terms of memory usage, Dijkstra and A* both maintained a list with the same amount of nodes, so the memory usage is identical.

Which Path Search Is Better?

In terms of performance, there's not much difference between these two.
Implementation-wise, Dijkstra and A* are both pretty easy to construct. However, A* provides more flexibility as we can apply different heuristic functions for different scenarios, so if we're talking about applying pathfinding in games, A* is more favorable than Dijkstra.

HEURISTICS

When we implement the A* algorithm, it's not like implementing Dijkstra's algorithm when there's only one way to define cost from the current searching node to the goal node. In my implementation of A* algorithm, I implemented 3 kinds of heuristic functions: random, euclidean distance, and manhattan distance.

Heuristic Functions Implemented In This Assignment

Random

1. **Is it admissible?** There's no good answer to this, sometimes it is, sometimes it is not. Just like what we mentioned in the "First Steps" section, we talked about due to not having a meaningful world representation of our graph, we can only apply

random heuristics to our A* algorithm. In the smaller graph, where we defined smaller costs, random heuristics is not admissible, because it's pretty much always overestimating; in the bigger graph, where the costs are much larger, random heuristics seem to be always underestimating. (or even omittable since it's too small compared to the edge costs, making A* almost identical to Dijkstra's)

2. **Is it consistent?** No, since it's a random number added to actual costs, it is not guaranteed that we're always making forward progress with random heuristics.
3. **Is it overestimating?** Same as our answer in question 1, it depends on the overall defined costs of our graph structure.

Euclidean Distance

1. **Is it admissible?** Yes, because euclidean distance is measured directly in world space, ignoring all walls and obstacles, so it's guaranteed to be an underestimate of the total cost.
2. **Is it consistent?** Yes, because the way we're measuring "forward progress" is by checking if this heuristic function obeys the triangle inequality in euclidean geometry, and current cost plus euclidean distance to target is always bigger than current euclidean distance to target.
3. **Is it overestimating?** No, because euclidean distance is always smaller than the actual distance to travel to target.

Manhattan Distance

1. **Is it admissible?** It depends, because manhattan distance is measured as follows: |x1- x2| + |y1- y2|, which is underestimating when there's obstacles between our current position and target, but overestimating when there's no blockage in between our current position and target.
2. **Is it consistent?** Yes, if we recall what we discussed in the euclidean distance section, if euclidean distance obeys the triangle inequality, then manhattan distance will too, since euclidean distance represents the hypotenuse of an triangle, and manhattan distance represents the sum of the lengths of the two arms of an triangle.
3. **Is it overestimating?** It depends, just like what we discussed in question 1.

Heuristic Functions Performance Measurement

To support euclidean distance and manhattan distance, I manually assigned some world positions to the smaller graph we mentioned in the "First Steps" section.
I also changed the std::chrono usage a little bit to reflect more precise run time measurement results. Now we can actually see how many milliseconds did the algorithm run.

The test case is: start node = 1, end node = 12

Heuristics: Random

1. run time: 0.072 ms
2. node visited: 10

Heuristics: Euclidean Distance

1. run time: 0.049 ms

2. node visited: 7

Heuristics: Manhattan Distance

1. run time: 0.057 ms
2. node visited: 7

PUTTING IT ALL TOGETHER

The way I designed this application to be used is: there are 3 buttons, the “Draw Obstacles” button, the “Path Find” button, and the “Reset” button.

The “Draw Obstacles” button allows the user to click on tiles on screen and mark them as “unwalkable” to represent obstacles (they will appear blue after clicking). After finishing drawing all the obstacles, the user can then click the “Path Find” button to search the path and the boid would follow along the path. After the boid has arrived at its target location, click the “Reset” button to reset the obstacles and clear the path list.

Drawing Obstacles

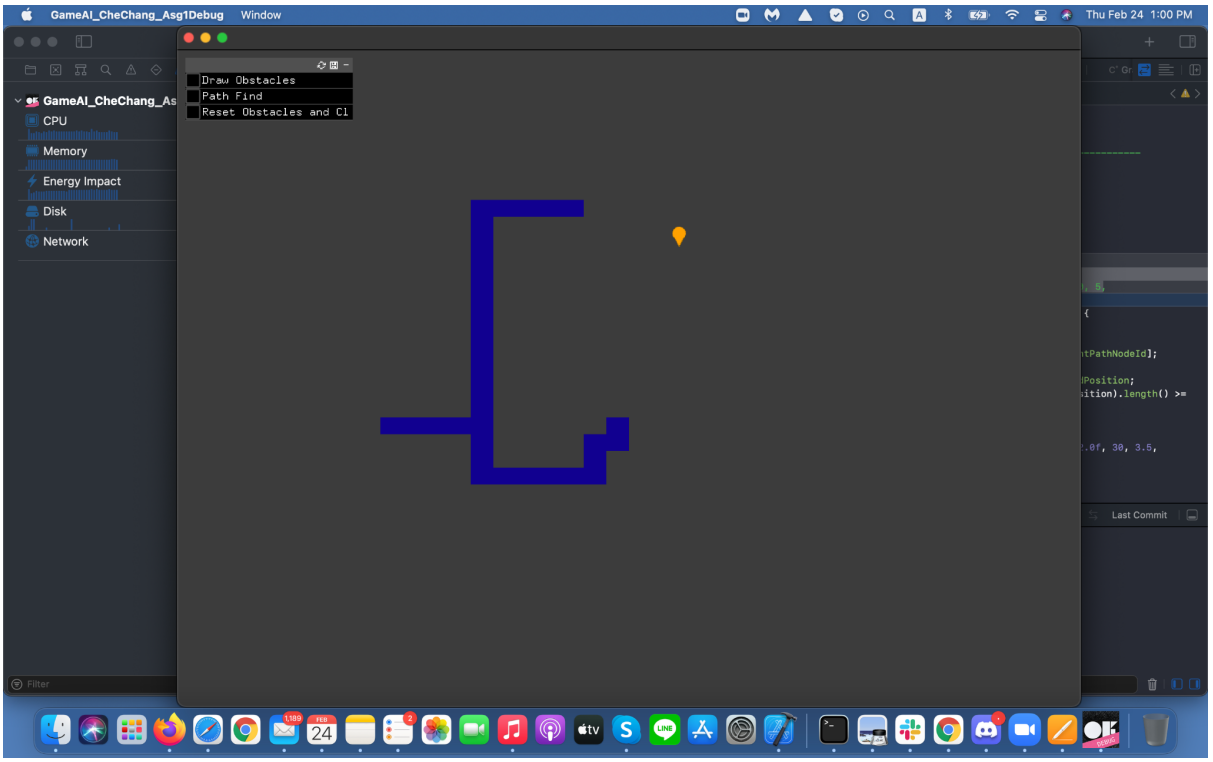

Figure 2: Screenshot of the application in “Draw Obstacle” mode

PathFinding

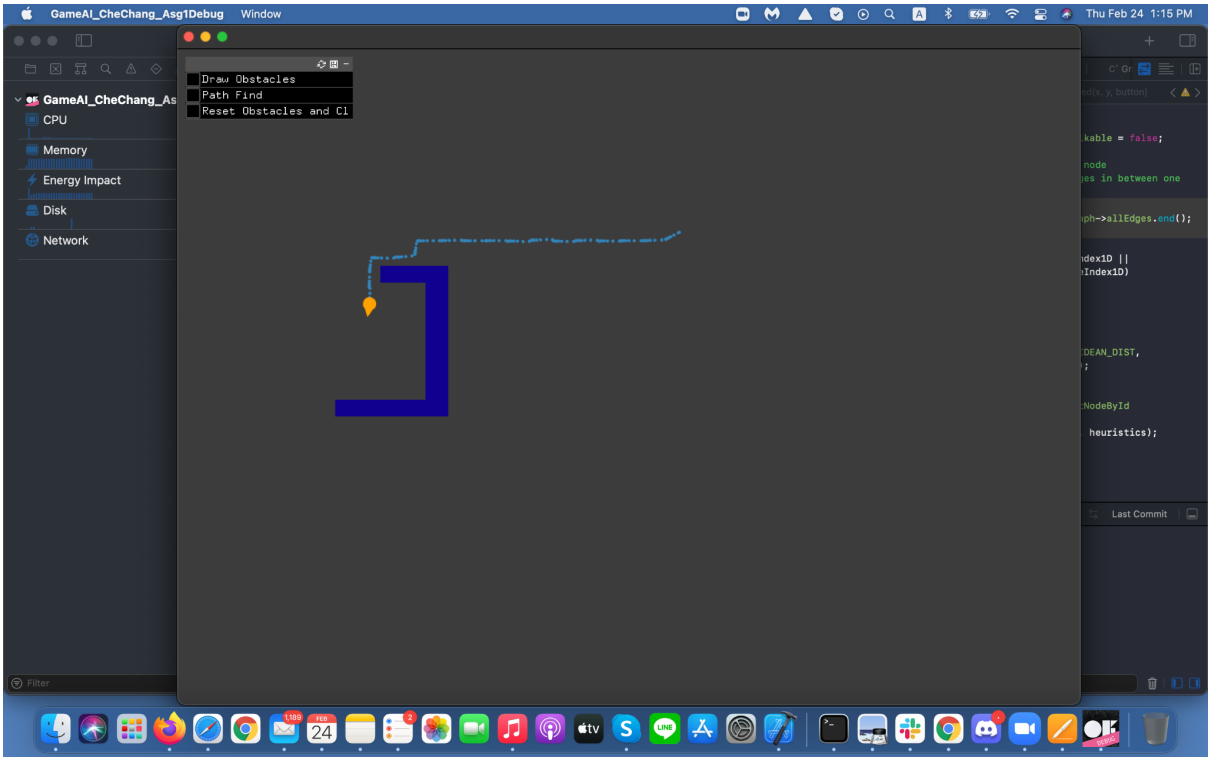

Figure 2: Screenshot of the application in "Path Find" mode

FUTURE WORKS

As mentioned in the AIFG (Artificial Intelligence for Games) book, pathfinding algorithms that we implemented in this assignment are only useful for non-changing game levels. If the level is constantly changing (not necessarily obstacles changing, in some war games, a path becomes invalid if enemies appear in that path), then dynamic pathfinding should be applied in this kind of scenario. We could improve the pathfinding algorithms by adding dynamic checks, so it allows us to create more obstacles during runtime and the boid would still be able to compute a path to its target.

Another possible optimization is related to frame rates, for now our graphs are small enough that the process time doesn't even come close to a sixtieth of a second, so the application frame rate does not get affected. However in some games, the pathfinding process takes up so much time because they have enormous tile graphs or navigation meshes to search on. This would be the time interruptible pathfinding come into play, we could optimize the pathfinding algorithms by pausing them just before its run time exceeds a certain threshold, update other time sensitive properties (physics, animations, etc.), and continue searching in the next frame.