Game AI Pathfinding Algorithm

Wei Zeng
University of Utah
u1330357@utah.edu

ABSTRACT

In this report I would mainly discuss my implementation of a 2D path-finder simulation via dijkstra's and A* algorithms. The simulation is built with openFramework.

Keywords

Game AI, Dijkstra's Algorithm, A* Algorithm, Pathfinding, Graph Theory

INTRODUCTION

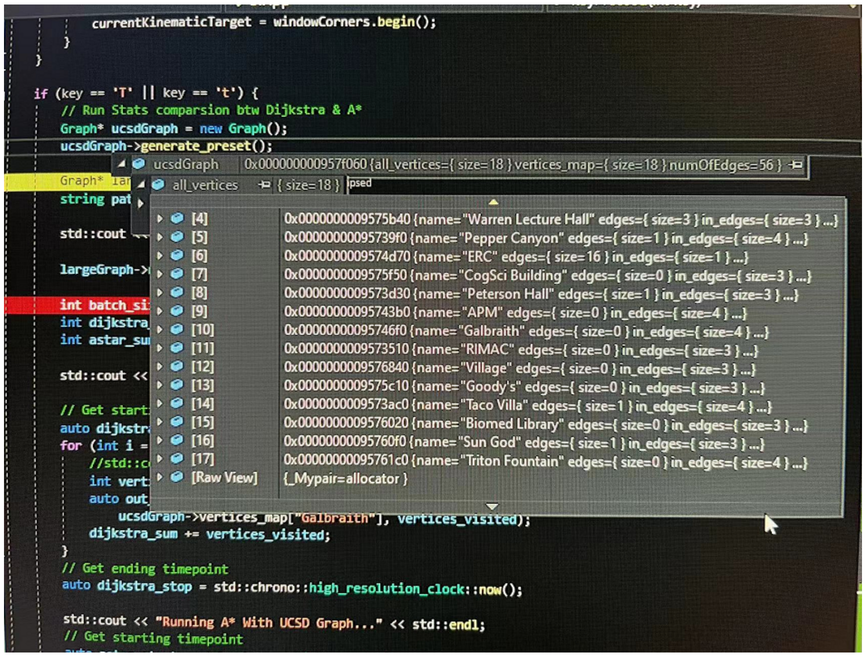
Pathfinding algorithms, without a doubt, are very important topics in the field of Game AI. In this simulation, I will explore some of the findings from my implementation and testing and also I will try to identify the advantages and disadvantages of some of my implementations and algorithms.

GRAPHS

In order to perform any sorts of pathfinding problems, we often want to construct a virtual representation of our problems, usually in the form of Graph. Graphs could be represented in various kinds of ways, for example, a very popular choice would be an adjacency list. In my implementation however, I decided to go into a different route in which I created a custom class to represent my Graph.

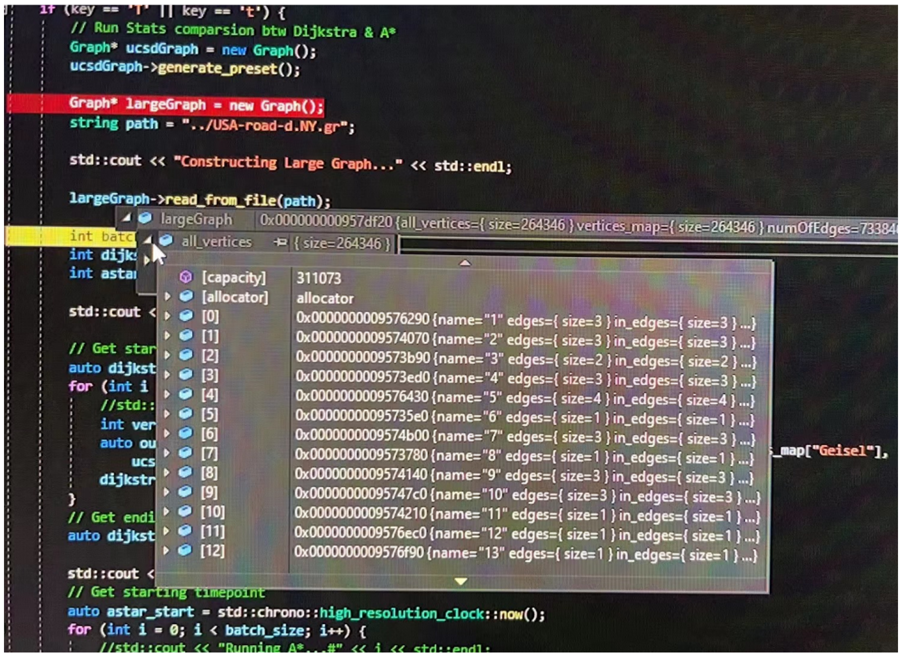
In my implementation, I defined a graph class with two most important class variables: vector of all vertices and a map of vertices from which I could perform O(1) searches via the name of the vertex. I also defined a struct called vertex to represent individual vertices in the graph. Within the struct, I stored the vertex's ingoing and outgoing edges in two vectors and also its index in the graph's all vertices vector for fast query.

In order to have some test cases for the later algorithms, with my custom graph class, I created two graphs for future usage. The first of them was a representation of the main campus of University of California, San Diego. This graph consists of 18 vertices each representing a popular destination at UCSD. Though most of the edges and their corresponding weights were based on actual distance in kilometers from Google Map, I intentionally created a few "cheat paths" with small weights to test the functionality of the algorithms. The Second Graph was downloaded from DIMACS and I wrote a text file parser for the data file and parsed it into my custom graph structure. This graph is really big with more than 20000 vertices.



```
            currentKinematicTarget = windowCorners.begin();
        }
    }

    if (key == 'T' || key == 't') {
        // Run Stats comparsion btw Dijkstra & A*
        Graph* ucsdGraph = new Graph();
        ucsdGraph->generate_preset();
            ◢ ucsdGraph    0x000000000957f060 {all_vertices={ size=18 } vertices_map={ size=18 } numOfEdges=56 }
        Graph* lar ◢ all_vertices    { size=18 }   psed
        string pat
                [4]    0x0000000009575b40 {name="Warren Lecture Hall" edges={ size=3 } in_edges={ size=3 } ...}
        std::cout   [5]    0x00000000095739f0 {name="Pepper Canyon" edges={ size=1 } in_edges={ size=4 } ...}
                [6]    0x0000000009574d70 {name="ERC" edges={ size=16 } in_edges={ size=1 } ...}
        largeGraph->  [7]    0x0000000009575f50 {name="CogSci Building" edges={ size=0 } in_edges={ size=3 } ...}
                [8]    0x0000000009573d30 {name="Peterson Hall" edges={ size=1 } in_edges={ size=3 } ...}
        int batch_si  [9]    0x00000000095743b0 {name="APM" edges={ size=0 } in_edges={ size=4 } ...}
        int dijkstra  [10]   0x00000000095746f0 {name="Galbraith" edges={ size=0 } in_edges={ size=4 } ...}
        int astar_su  [11]   0x0000000009573510 {name="RIMAC" edges={ size=0 } in_edges={ size=3 } ...}
                [12]   0x0000000009576840 {name="Village" edges={ size=0 } in_edges={ size=3 } ...}
        std::cout <<  [13]   0x0000000009575c10 {name="Goody's" edges={ size=0 } in_edges={ size=3 } ...}
                [14]   0x0000000009573ac0 {name="Taco Villa" edges={ size=1 } in_edges={ size=4 } ...}
        // Get start  [15]   0x0000000009576020 {name="Biomed Library" edges={ size=0 } in_edges={ size=3 } ...}
        auto dijkstr  [16]   0x00000000095760f0 {name="Sun God" edges={ size=1 } in_edges={ size=3 } ...}
        for (int i =  [17]   0x00000000095761c0 {name="Triton Fountain" edges={ size=0 } in_edges={ size=4 } ...}
            //std::c  [Raw View]   {_Mypair=allocator }
            int vert
            auto out
                ucsdGraph->vertices_map["Galbraith"], vertices_visited);
            dijkstra_sum += vertices_visited;
        }
        // Get ending timepoint
        auto dijkstra_stop = std::chrono::high_resolution_clock::now();

        std::cout << "Running A* With UCSD Graph..." << std::endl;
        // Get starting timepoint
```

Figure 1: This is the captions showing the custom graph loaded successfully.



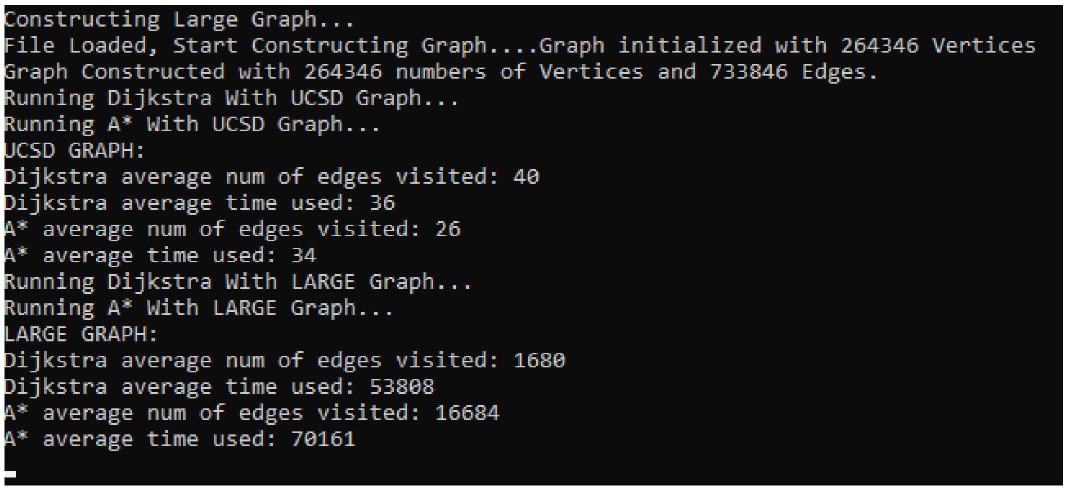
```
    if (key == 'T' || key == 't') {
        // Run Stats comparsion btw Dijkstra & A*
        Graph* ucsdGraph = new Graph();
        ucsdGraph->generate_preset();

        Graph* largeGraph = new Graph();
        string path = "../USA-road-d.NY.gr";

        std::cout << "Constructing Large Graph..." << std::endl;

        largeGraph->read_from_file(path);
            ◢ largeGraph    0x000000000957df20 {all_vertices={ size=264346 } vertices_map={ size=264346 } numOfEdges=73384
        int batc ◢ all_vertices    { size=264346 }
        int dijk
        int astar   [capacity]   311073
                [allocator]  allocator
        std::cout <  [0]    0x0000000009576290 {name="1" edges={ size=3 } in_edges={ size=3 } ...}
                [1]    0x0000000009574070 {name="2" edges={ size=3 } in_edges={ size=3 } ...}
        // Get star  [2]    0x0000000009573b90 {name="3" edges={ size=2 } in_edges={ size=2 } ...}
        auto dijkst  [3]    0x0000000009573ed0 {name="4" edges={ size=3 } in_edges={ size=3 } ...}
        for (int i   [4]    0x0000000009576430 {name="5" edges={ size=4 } in_edges={ size=4 } ...}
            //std::  [5]    0x00000000095735e0 {name="6" edges={ size=1 } in_edges={ size=1 } ...}
            int ver  [6]    0x0000000009574b00 {name="7" edges={ size=3 } in_edges={ size=3 } ...}
            auto ou  [7]    0x0000000009573780 {name="8" edges={ size=1 } in_edges={ size=1 } ...}  _map["Geisel"],
                ucs  [8]    0x0000000009574140 {name="9" edges={ size=3 } in_edges={ size=3 } ...}
            dijkstr  [9]    0x00000000095747c0 {name="10" edges={ size=3 } in_edges={ size=3 } ...}
        }        [10]   0x0000000009574210 {name="11" edges={ size=1 } in_edges={ size=1 } ...}
        // Get endi  [11]   0x0000000009576ec0 {name="12" edges={ size=1 } in_edges={ size=1 } ...}
        auto dijkst  [12]   0x0000000009576f90 {name="13" edges={ size=1 } in_edges={ size=1 } ...}

        std::cout <
        // Get starting timepoint
        auto astar_start = std::chrono::high_resolution_clock::now();
        for (int i = 0; i < batch_size; i++) {
            //std::cout << "Running A*...#" << i << std::endl;
```

Figure : This is the captions showing the large graph loaded successfully.

DIJKSTRA’S ALGORITHM & A*

After implementing both algorithms, I tested them with both my custom small graph and with the large graph. The differences between the two algorithms essentially is the heuristic function. In the following test, I used a random guess heuristic function for the A* algorithm which outputs a random estimate from 100 to 1300. The range was designed based on the minimum and maximum weights from the custom map. And from the statistics below we could see that A* perform better in the custom graph than the large graph. This could be due to the fact that the random guess is not matching with the actual cost range of edges in the large graph.

One big hindering factor for runtime could be the design of our graph structure. Since both algorithms need to perform some sorts of searching through edges of vertices and searching through previous vertices (outputting the path), I found it really useful to store the edges information together with vertices and also record both current vertex’s index and its previous vertex’s index such that we could perform O(1) searching when needed. However, this created a larger burden on the side of memory cost and also more importantly this became an issue when trying to remove some vertices as you would need to update all other vertices after the ones you are deleting, which cost O(n) time bringing down the performance again, not to mention that it was a disaster to manage/implement.



```
Constructing Large Graph...
File Loaded, Start Constructing Graph....Graph initialized with 264346 Vertices
Graph Constructed with 264346 numbers of Vertices and 733846 Edges.
Running Dijkstra With UCSD Graph...
Running A* With UCSD Graph...
UCSD GRAPH:
Dijkstra average num of edges visited: 40
Dijkstra average time used: 36
A* average num of edges visited: 26
A* average time used: 34
Running Dijkstra With LARGE Graph...
Running A* With LARGE Graph...
LARGE GRAPH:
Dijkstra average num of edges visited: 1680
Dijkstra average time used: 53808
A* average num of edges visited: 16684
A* average time used: 70161
```

Figure 1: This is the captions for stats returned from batching both algorithms on ucsd graph and large graph.

HEURISTICS

For the A* algorithm, I designed two simple heuristic functions to test its performance. First one being a constant guess with h(x) = 1. In my custom graph, this heuristic function is both admissible and consistent. Since the weights of all edges in my graph are significantly larger than 1, this is definitely not an overestimation. As for consistency, it satisfies the triangle inequality since weights are much larger than 1 which means 1 + a, where a >= 1, is always larger than 1.

The second heuristic is a random guess function. Its output was capped to be larger than 100 and smaller than 1300. For this heuristic function, it is hard to say whether it is admissible or consistent because it really depends on the output of the random function. It could produce admissible and consistent results for some certain edges but most of the time it should violate at least one of the rules.

SIMULATION (ALL TOGETHER)

In order to simulate everything more visually, I created a tile graph abstraction scheme to impose on the 2D world and implemented a method for users to create obstacles in real time. From the figure below, yellow dots represent each tiles' center position, yellow bars represent obstacles created by users, the purple trails represent the path taken by the path finder and finally the blue dots represent the target. The path finder is also implemented so that it evades from the obstacles under a safe radius. As shown in the highlighted section in the figure below, the path finder got pushed away from the walls.

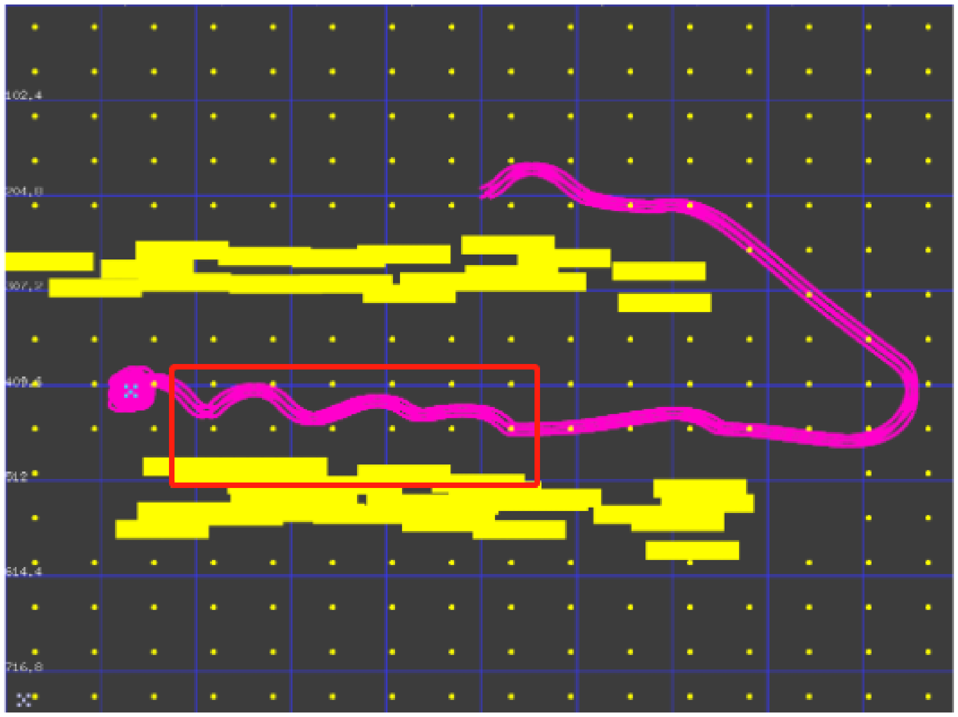


Figure 3: This is the caption for A* with a constant guess heuristic of h(x) = 1.

Here is another figure showing the path using a different heuristic function (random guess). This heuristic function was actually designed for my small graph outputting significantly larger cost estimation than actual. An overestimating heuristics means we might be getting suboptimal results which was shown in the figure below.

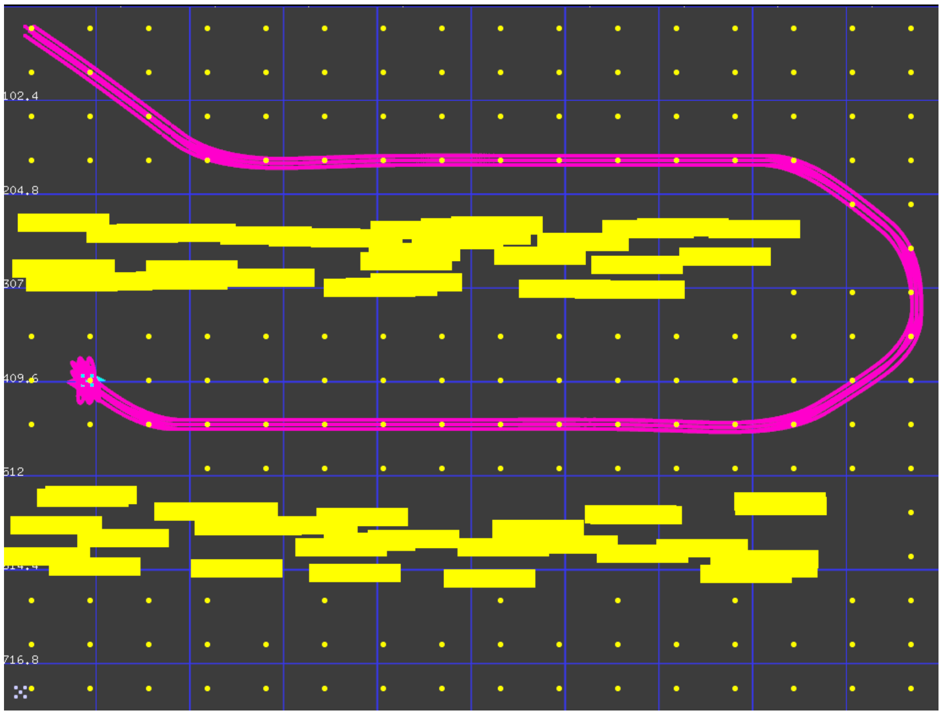


Figure 4: This is the caption for A* with a random guess heuristic of h(x) = rand() % 1200 + 100, which is an overestimation.

Conclusion

After all the simulations, in conclusion, I found it most challenging to create a user-friendly yet at the same time runtime efficient data structure for graphs. In our simulation the usage of graphs is so ubiquitous that the impact of its performance, in terms of searching for specific vertices, removing one, adding one etc, become significantly large over the simulation. During implementation and designing my structure, I oftentimes try to get O(1) via using more memory to store extra data for searching but at the end got to a point where some other system completely ruins the time saved due to some extra operations needed to manage that data.

BIBLIOGRAPHY

"New York City." 9th DIMACS Implementation Challenge: Shortest paths. Accessed February 25, 2022. http://www.diag.uniroma1.it/challenge9/download.shtml.