

Pathfinding Algorithms

This report includes the implementation of A-Star and Dijkstra algorithms, and also some of the exciting findings I've discovered. I've used the OpenFrameworks for C++ with Visual Studio 2017 to implement all my algorithms. OpenFrameworks is an elementary yet powerful library that provides support for multiple systems such as Graphics, GUI, Sound, etc.

Before discussing the implementation of the algorithms, I would like to explain and define some of the essential data structures I used to implement this algorithm.

NodeRecord: A node is just a region of space. Each node has a unique id. NodeRecord is a struct that holds the data about a node.

```
struct NodeRecord {  
    int node;  
    DirectedWeightedEdge * incomingEdge;  
    float costSoFar;  
    float estimateToGoal;  
  
    friend bool operator < (const NodeRecord& i_lhs, const NodeRecord& i_rhs);  
    friend bool operator > (const NodeRecord& i_lhs, const NodeRecord& i_rhs);  
    friend bool operator == (const NodeRecord& i_lhs, const NodeRecord& i_rhs);  
    friend bool operator != (const NodeRecord& i_lhs, const NodeRecord& i_rhs);  
};
```

Edge: An edge is a directed and weighted connection between two nodes.

```
class DirectedWeightedEdge {  
public:  
    float GetCost();  
    int GetSource();  
    int GetSink();  
    DirectedWeightedEdge(int i_source, int i_sink, float i_cost) :  
        source(i_source), sink(i_sink), cost(i_cost){  
    }  
private:  
    float cost;  
    int source;  
    int sink;  
    DirectedWeightedEdge() = default;  
};
```

Graph: A graph is a collection of directed weighted edges.

```

class DirectedGraph {
public:
    std::vector<DirectedWeightedEdge *> GetOutgoingEdges(int i_node);
    int GetTotalNodes();
    DirectedGraph(std::vector<DirectedWeightedEdge *> i_allEdges);

private:
    std::map<int, std::vector<DirectedWeightedEdge *>> graph;
    DirectedGraph() = default;
};

```

First steps: I've acquired a very large graph with more than 25000 edges from the internet. This graph represents the US airport network. I've added the source of the graph in the references section. Below is a part of the file I've downloaded. The graph is represented as follows:

Syntax: Source Sink Cost

```

1 2 1
1 3 7
4 5 3
6 7 5
6 8 14137
9 10 8
11 2 8
12 5 413
13 14 2

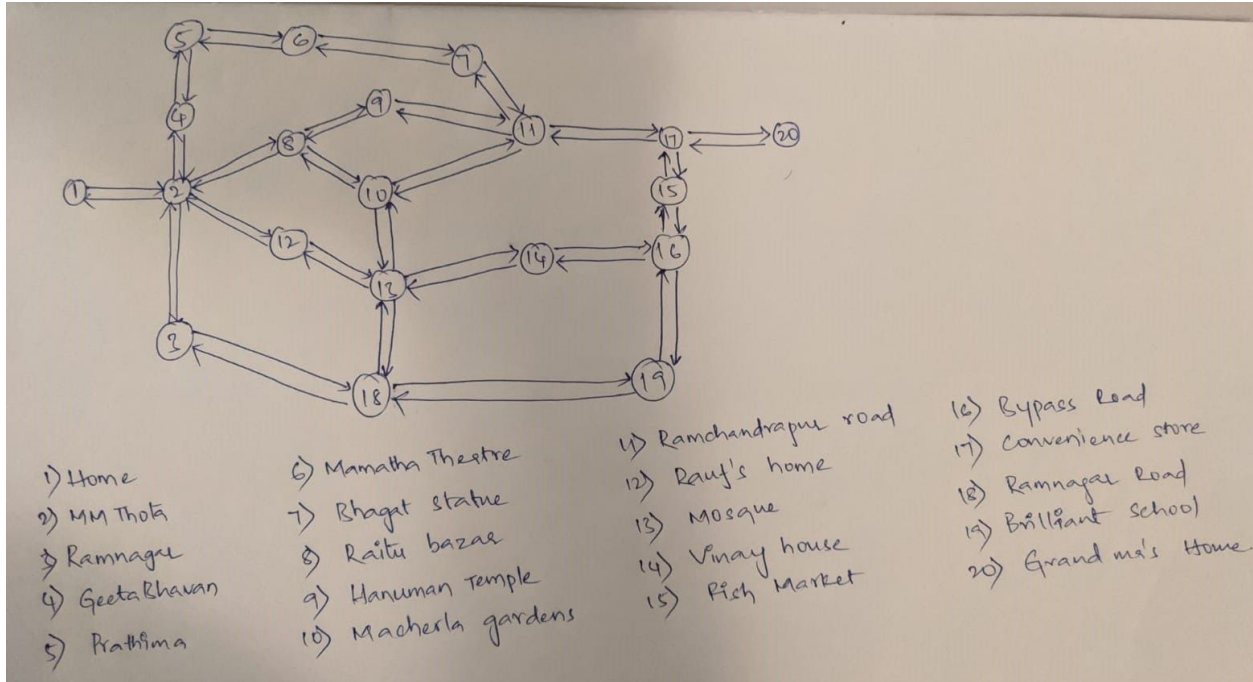
```

I've written a parser to convert this file into the graph structure I've used. I will use the term "**Large graph**" to refer to this graph throughout this report.

Following the same syntax as above, I've created a graph that represents an area of my hometown. Here is the graph

```
1 2 5
2 1 5
2 3 12
3 2 12
2 4 8
4 2 8
4 5 7
5 4 9
5 6 4
6 5 4
6 7 8
7 6 9
2 8 10
8 2 12
2 12 6
12 2 6
3 18 4
18 3 5
7 11 15
11 7 15
8 9 16
9 8 16
8 10 9
10 8 9
9 11 15
11 9 15
10 11 10
11 10 10
10 13 7
13 10 7
11 17 5
17 11 5
12 13 4
13 12 4
13 14 6
14 13 6
13 18 8
18 13 8
14 16 3
16 14 3
15 16 4
16 15 4
15 17 3
17 15 3
16 19 10
19 16 10
17 20 5
20 17 5
18 19 5
19 18 5
```

Here is a sketch of the graph



This graph consists of 20 nodes and 50 edges. I will use the term “**Small graph**” to refer to this graph throughout this report.

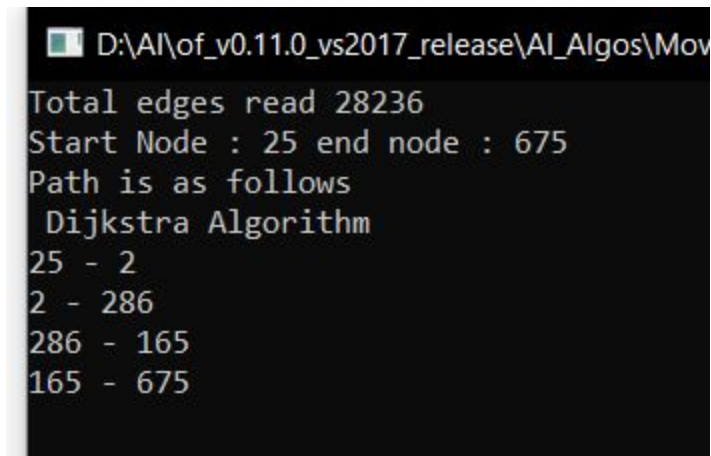
Now that we have everything to get going, let's start implementing the pathfinding algorithms.

Dijkstra Algorithm: Dijkstra algorithm computes the shortest path between two nodes, by calculating all the paths that connect the nodes and choosing the one with the smallest cost. Since we are calculating the path by considering all the nodes in the graph, Dijkstra is not very performance efficient.

Here is a screenshot of the Dijkstra algorithm returning the path from node 1 to node 20 in the small graph.

```
D:\AI\of_v0.11.0_vs2017_release\AI_Algos\Movement
Total edges read 50
Start Node : 1 end node : 20
Path is as follows
Dijkstra Algorithm
1 - 2
2 - 12
12 - 13
13 - 14
14 - 16
16 - 15
15 - 17
17 - 20
```

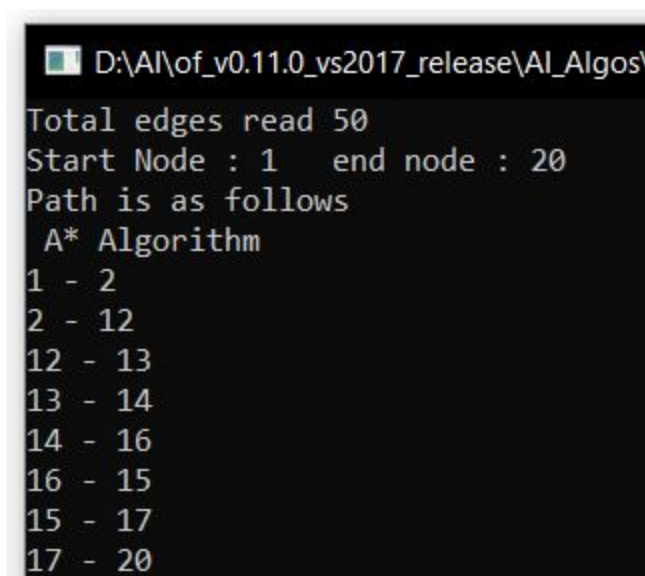
Here is a screenshot of the Dijkstra algorithm returning the path from node 25 to node 675 in the large graph.



```
D:\AI\of_v0.11.0_vs2017_release\AI_Algos\Mov
Total edges read 28236
Start Node : 25 end node : 675
Path is as follows
  Dijkstra Algorithm
25 - 2
2 - 286
286 - 165
165 - 675
```

A-Star Algorithm: A-Star is an optimal pathfinding algorithm. A* uses heuristics in addition to cost to compute the shortest path. A* has many variants and A* is only as good as its heuristics. I've used random numbers as the heuristics.

Here is a screenshot of the path returned by the A* algorithm for the small graph. The start node is 1 and the end node is 20.



```
D:\AI\of_v0.11.0_vs2017_release\AI_Algos\
Total edges read 50
Start Node : 1 end node : 20
Path is as follows
  A* Algorithm
1 - 2
2 - 12
12 - 13
13 - 14
14 - 16
16 - 15
15 - 17
17 - 20
```

Here is a screenshot of the path returned by the A* algorithm for the large graph. The start node is 25 and the end node is 675.

```
D:\AI\of_v0.11.0_vs2017_release\AI_Algos\
Total edges read 28236
Start Node : 25 end node : 675
Path is as follows
A* Algorithm
25 - 2
2 - 286
286 - 165
165 - 675
```

Performance Analysis:

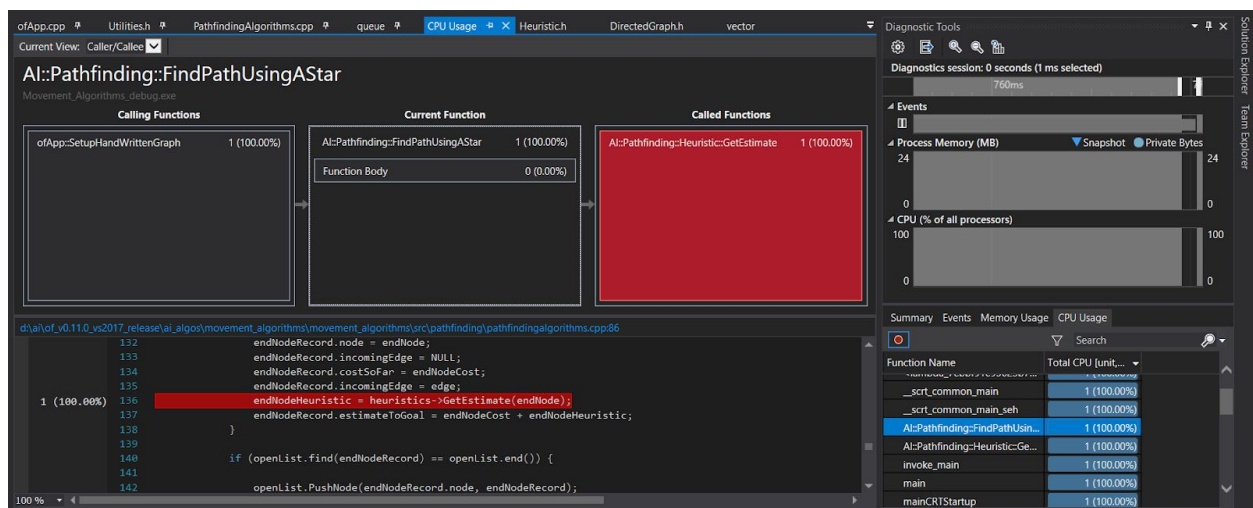
If $g(x)$ is the cost and $h(x)$ is the heuristic function, then

A-Star $f(x) = g(x) + h(x)$

Dijkstra $f(x) = g(x)$

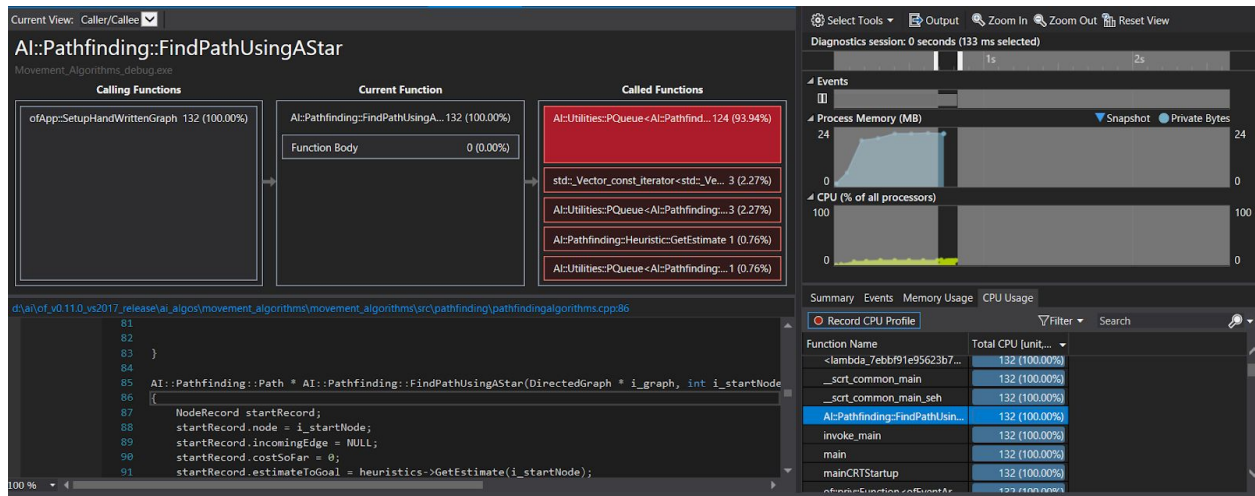
Because we are calling the heuristic function at every iteration of the open list, the performance of $h(x)$'s calculation will dominate the performance of A-Star. The other dominating factor of A-Star performance is the design of the fringe - priority queue.

Since the heuristics I'm using are randomly generated at the beginning of the algorithm. The calls to the `GetEstimate` function of heuristics isn't that bad because it is only returning a value from the vector. The average time the algorithm spent in this function is 5ms. Here is a screenshot.



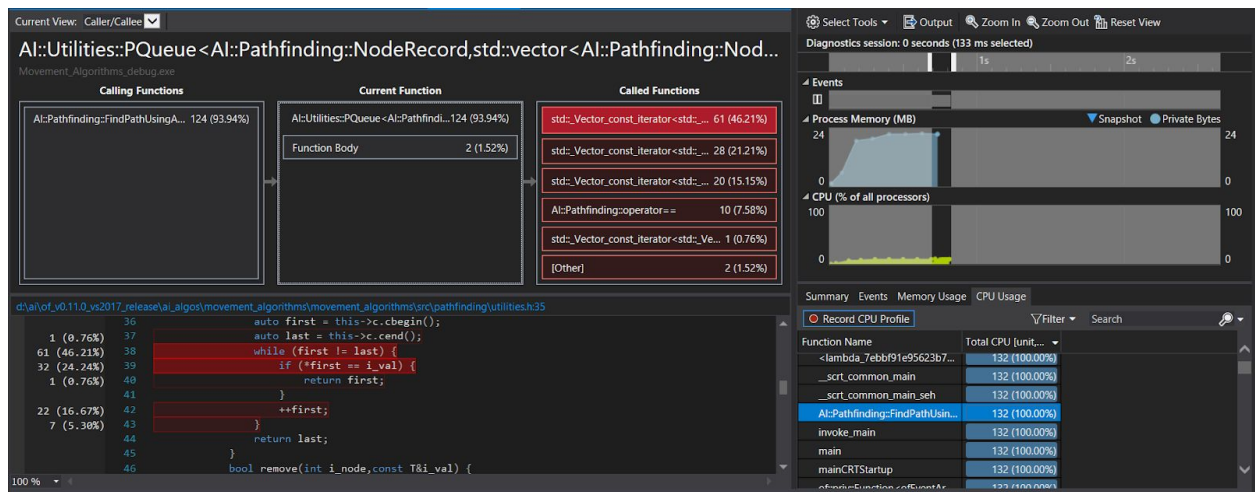
Since the retrieval is already at $O(1)$, I couldn't make it any better than that.

The below image is a screenshot showing the breakdown of the A-Star algorithm. As you can see, most of the time is taken by the find method of the priority queue. This check happens to see whether the current node is in the open list or closed list.



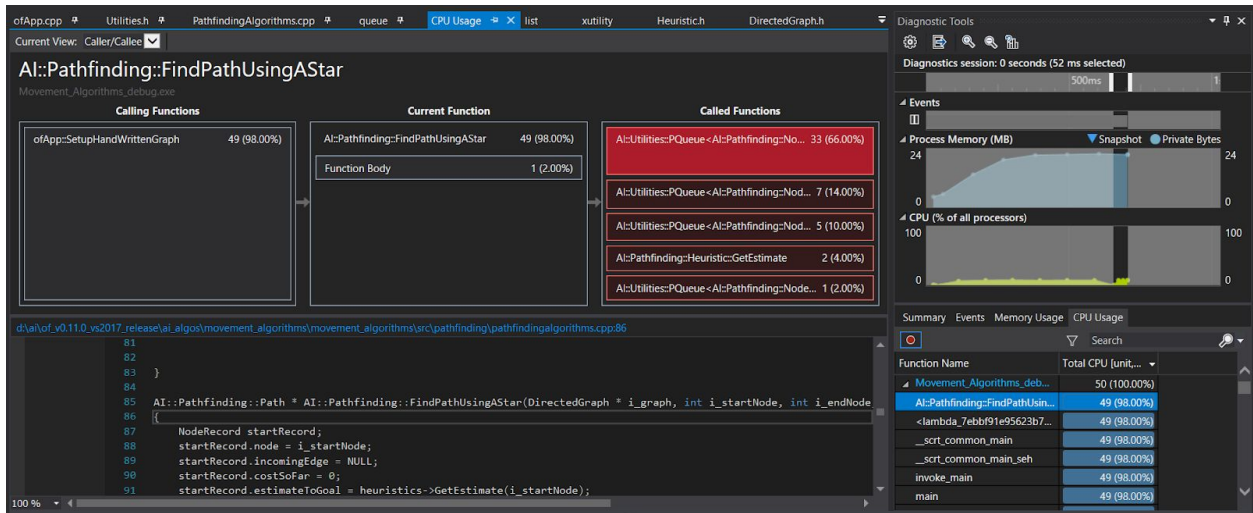
The calls to find take up 124 ms on average and almost 94% of the total time it takes to execute the A-Star.

The below screenshot helps you to see more clearly what is happening inside the find method.



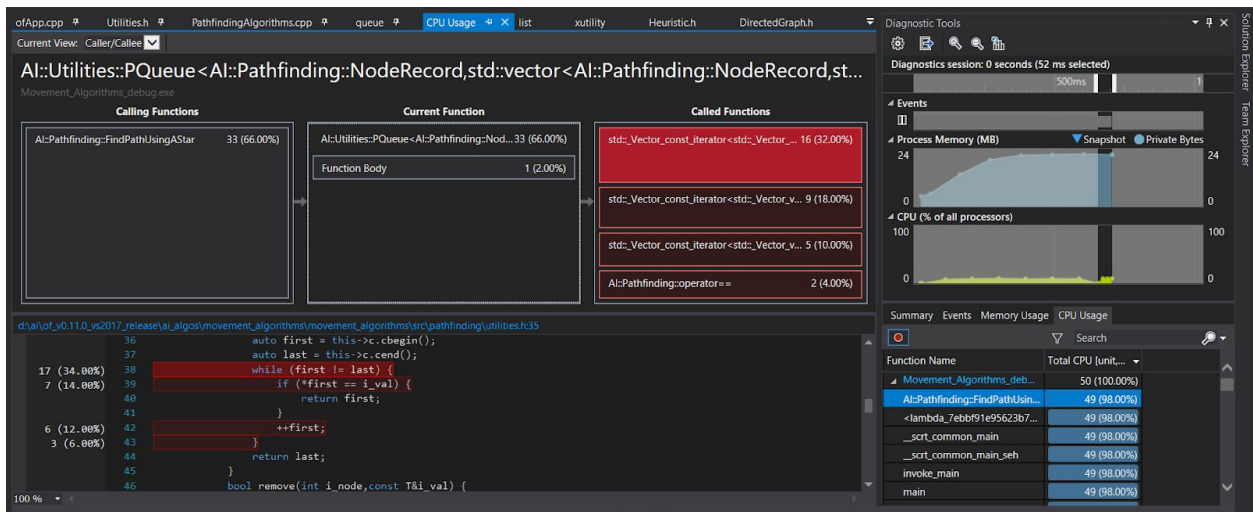
As you can see, the find method traverses through each object in the list and checks if it is equal to the one we're looking for. This can be very inefficient if our list has a large number of items. So, I needed to improvise the performance of the find method.

I decided to use an unordered set of the nodes in the priority queue, so when I'm trying to check if the corresponding node is in the open list or closed list, I can just check in the unordered set. The complexity of the unordered set is $O(1)$, so it will greatly reduce the runtime. Here is the screenshot showing the optimized A-Star algorithm breakdown



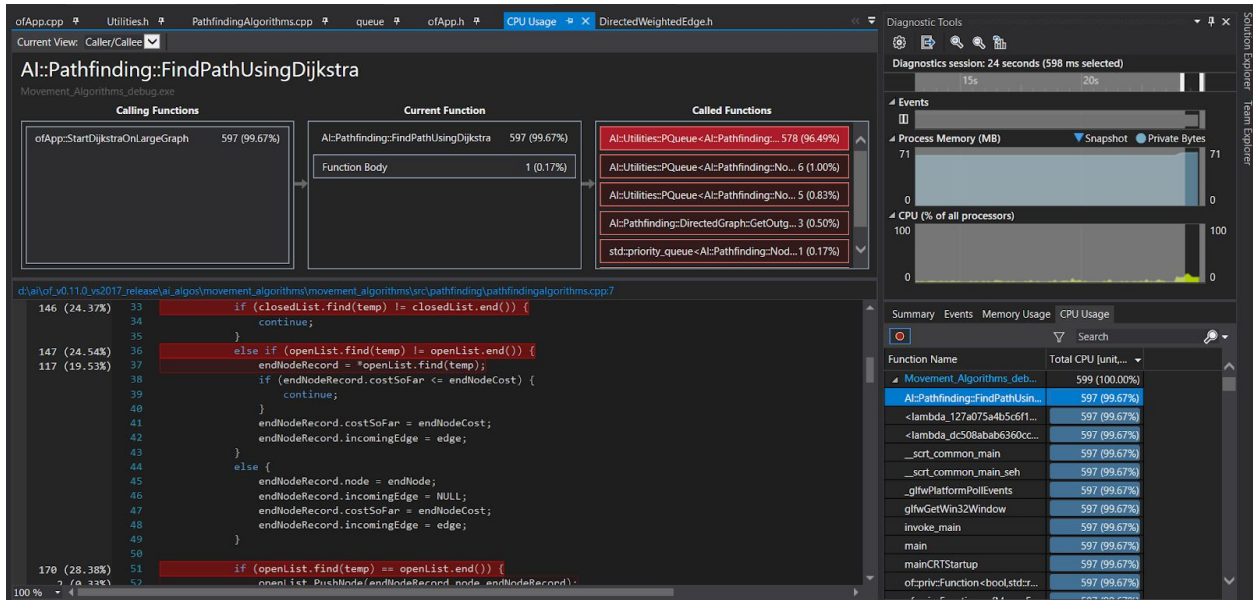
As you can see, the optimized version only took 49 ms to execute, whereas the unoptimized version took 132 ms to execute. I've also greatly reduced the execution time of the find method. It reduced from 128 ms to 33 ms. The new method contains, which checks if the given node is in the unordered set, has an execution time of 7ms. That is a great improvement to the performance.

Here is the screenshot of the optimized find method

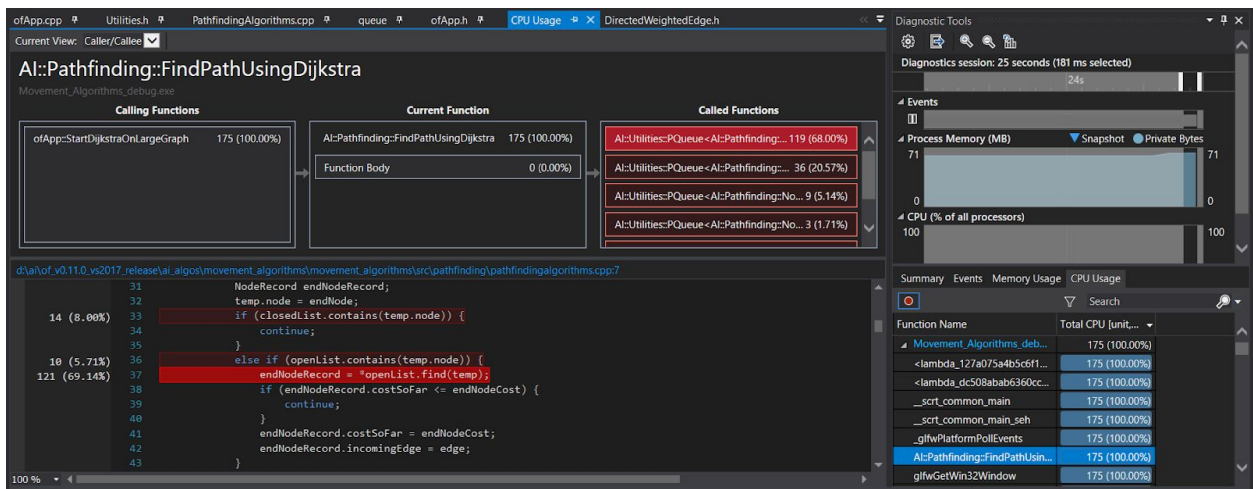


It still iterates through the list, but I will call the find method only if the list contains the item. One drawback to this approach is it adds an additional space complexity of $O(m)$ where m is the size of the listing. But I've decided speed over space.

Now coming to the performance of the Dijkstra algorithm, the unoptimized version (using find method to check whether a node is in the open list) takes 597 ms to execute the algorithm. And the execution time of the find method has a whopping 578 ms. This is almost 6 times the execution time of A-Star.



However, the optimized version takes only 175 ms to execute. The execution time of the find method is reduced to 119 ms from 578 ms. But even then, this is almost four times the execution time of the optimized A-Star, which has an execution time of 50ms.



All this data gives us an understanding of the runtime of both algorithms. Another interesting value we can use to understand performance is the fill pattern of the algorithm. Fill pattern is the ratio of open list to closed list nodes.

For the small graph using Dijkstra, the fill pattern looks like this.

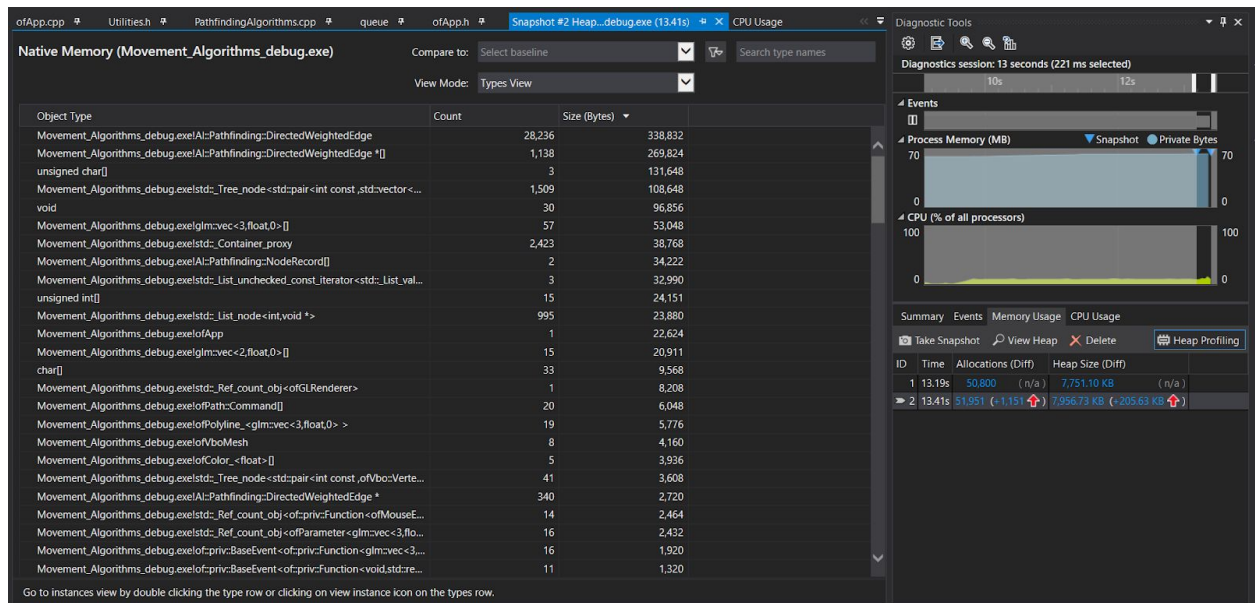
```
1
2
1.33333
1
1.2
1
0.714286
0.625
0.555556
0.5
0.454545
0.416667
0.384615
0.285714
0.266667
0.1875
0.176471
0.111111
0.0526316
```

Whereas for A-Star it looks like this

```
1
2
1.33333
1
1.2
1
0.714286
0.625
0.555556
0.5
0.454545
0.416667
0.384615
0.285714
0.266667
0.1875
0.176471
0.111111
0.0526316
```

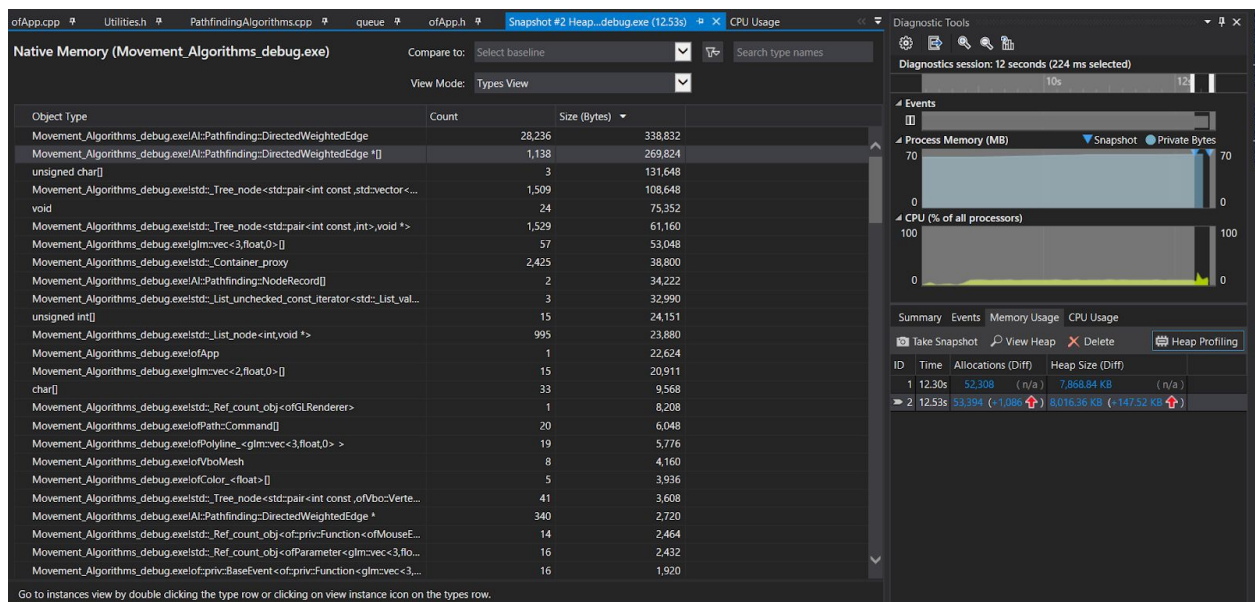
As you can see, both are quite similar. This is also true for the large graph, both the algorithms have a similar fill pattern.

Now calculating performance in terms of memory usage, the below screenshot shows the memory snapshot of the Dijkstra algorithm.



In the bottom right corner, you can see the memory usage window. It shows that the Dijkstra algorithm has a raise of 1151 allocations and 205Kb of data.

Whereas for A-Star, it is 1086 allocations and 148Kb of data. This helps us to understand that there is a significant difference between AStar and Dijkstra in terms of memory usage as well.



Heuristics: I wanted to see if the change in heuristics has any performance impact on the A-Star algorithm. The heuristic class looks like this.

```

class Heuristic {
public:
    int GetEstimate(int i_node);
    Heuristic(int i_goalNode, DirectedGraph * i_graph);
private:
    void CalculateAndStoreHeuristics(int i_goalNode, int i_totalNodes);
    Heuristic() = default;
    std::map<int, int> heuristicMap;
};

```

Previously I was using the randomly generated heuristics, but for this part, I've used a hand-authored heuristics. Since I know the layout of my hometown, I used my knowledge to design heuristics for the small graph. I came up with this heuristics for goal node 20

```

constantGuessMap.insert(std::make_pair(1, 2000));
constantGuessMap.insert(std::make_pair(2, 1900));
constantGuessMap.insert(std::make_pair(3, 2500));
constantGuessMap.insert(std::make_pair(4, 2600));
constantGuessMap.insert(std::make_pair(5, 2100));

constantGuessMap.insert(std::make_pair(6, 2000));
constantGuessMap.insert(std::make_pair(7, 1700));
constantGuessMap.insert(std::make_pair(8, 1900));
constantGuessMap.insert(std::make_pair(9, 1200));
constantGuessMap.insert(std::make_pair(10, 1500));

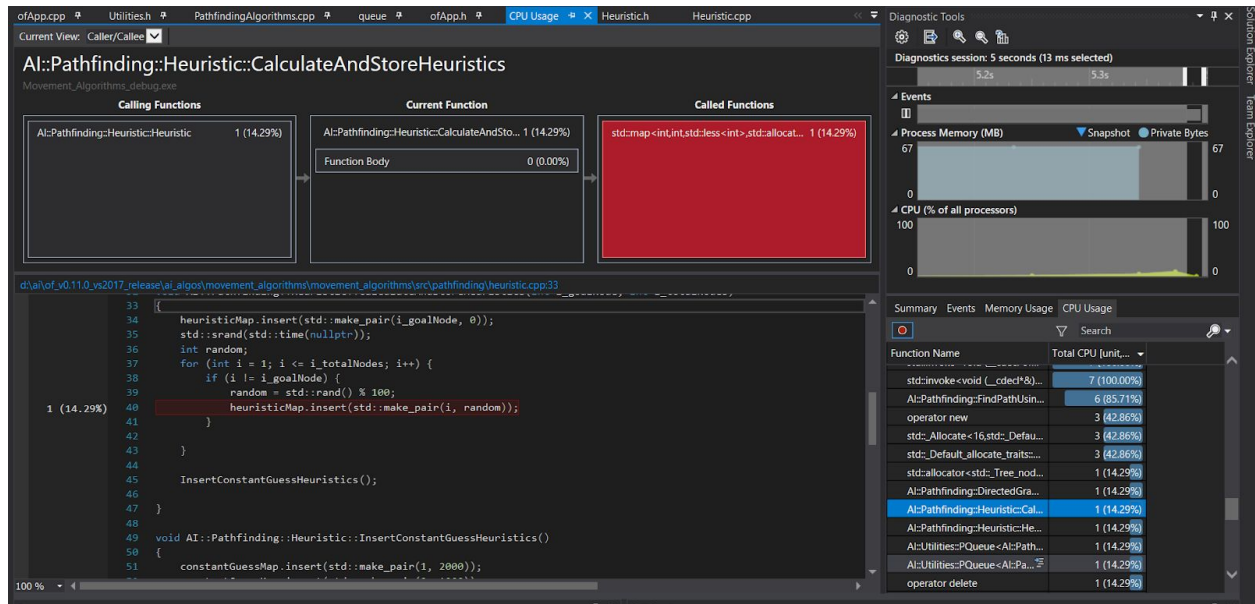
constantGuessMap.insert(std::make_pair(11, 1100));
constantGuessMap.insert(std::make_pair(12, 1800));
constantGuessMap.insert(std::make_pair(13, 1500));
constantGuessMap.insert(std::make_pair(14, 1000));
constantGuessMap.insert(std::make_pair(15, 500));

constantGuessMap.insert(std::make_pair(16, 800));
constantGuessMap.insert(std::make_pair(17, 200));
constantGuessMap.insert(std::make_pair(18, 2000));
constantGuessMap.insert(std::make_pair(19, 1500));
constantGuessMap.insert(std::make_pair(20, 0));

```

Even though they are hand authored, I derived those values from the actual distance between two points in the real world in meters. Since it is a direct distance between two points, I think this is admissible, consistent and not overestimating.

I'm not sure if the randomly generated heuristics are consistent or admissible.



This image shows the runtime of the heuristics calculation. The random number heuristics take up 1 ms time and the time taken by hand authored heuristics setup is very less than 1 ms.

PathFollow:

The final part of this project is to create the graph space dynamically from the existing world space. And then use this graph to find the path to a specified location, and use the steering behaviors to traverse through that path.

Division Scheme: The process of converting a point from world space to a point in graph space is called Quantization. And the process of converting a point in graph space to a point in world space is called Localization. The process of Quantization and Localization is called a division scheme.

There are many division schemes, some of the popular ones include Voronoi diagrams, Navmesh, and Tile-based.

I'm using the Tile-based division scheme because of its simplicity. In a tile-based division scheme, the entire world space is turned into an $m \times n$ grid where m is the number of rows and n is the number of columns. Each tile will represent a node in the graph space and all the neighboring nodes have an edge between them.

To quantize, any position, we find the tile we're in and return it.

To localize, we localize to the center of the tile (node) we're in.

The math included is pretty simple,

X and Y are the x, y components of the position vector. And i, j represent the tile in the grid, if the grid is a 2D array.

$$X = (i + 0.5) * \text{heightOfTile}$$

$$Y = (j + 0.5) * \text{widthOfTile}$$

For the path follow, I start by converting the world space into the graph space using a Tile-based division scheme. Then based on input from the player, the final position is quantized into a node (endNode). I will pass the startNode (the quantized node of player start position) and endNode to the A* algorithm. The A* algorithm will return a path. A path is generally a collection of nodes to reach. So, I will just iterate through each node and call seek the node's position, as soon as I reach a node, I will seek the next node until I reach the end of the path.

The obstacles are generated randomly. And when I'm creating graph space from world space, I'm also sending information about the obstacles. All the nodes with obstacles are considered as not accessible from the neighboring nodes. There is no need for checking obstacle avoidance because the A* algorithm will not generate any path through the obstacles because they aren't accessible from other nodes.

Here is a video that shows the path follow algorithm.

[Click on the link to open the video](#)

References:

A* Algorithm explanation:

<https://www.redblobgames.com/pathfinding/a-star/introduction.html>

Large graph source :

```
@MISC{konect:2016:opsahl-usairport,
  title = {US airports network dataset -- {KONECT}},
  month = sep,
  year = {2016},
  url = {http://konect.uni-koblenz.de/networks/opsahl-usairport}
}

@online{konect:opsahl11,
  title={Why Anchorage Is Not (that) Important: Binary ties and Sample Selection},
  author = {Tore Opsahl},
  year = {2011},
  url = {http://wp.me/poFcY-Vw}
}

@inproceedings{konect,
  title = {{KONECT} -- {The} {Koblenz} {Network} {Collection}},
  author = {Jérôme Kunegis},
  year = {2013},
  booktitle = {Proc. Int. Conf. on World Wide Web Companion},
  pages = {1343--1350},
```

```
url =
{http://userpages.uni-koblenz.de/~kunegis/paper/kunegis-koblenz-network-collection.pdf},
url_presentation =
{http://userpages.uni-koblenz.de/~kunegis/paper/kunegis-koblenz-network-collection.presentation.pdf},
}
```