Process Letter

Creating the Directed Weighted Graph

Directed Weighted Graph:

```
class DirectedWeightedGraph {
public:
    // Use an adjacency list to represent the graph
    std::unordered_map<int, std::vector<DirectedWeightedEdge>> adjacencyList;

    std::unordered_map<int, Node> nodes;
```

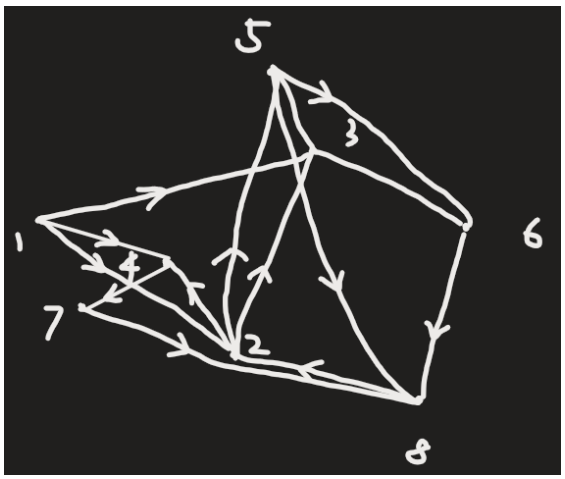
Directed Weighted Edge:

```
struct DirectedWeightedEdge {
    int source;
    int destination;
    float weight;

    DirectedWeightedEdge(int source, int destination, float weight)
        : source(source), destination(destination), weight(weight) {}
};
```

Node that stores coordinates to provide meaningful information for the A* algorithm:

```
struct Node {
    // Coordinates
    float x;
    float y;

    // Constructor
    Node() : x(0), y(0) {}
    Node(float x, float y) : x(x), y(y) {}
};
```

Small Graph:



The small graph represents a few cities, but the roads are singly directed.

I made the small graph hard coded so that the content remains the same during each run. This way, I was able to debug my Dijkstra and A* algorithms' implementations.

The weights of the edges are the Euclidean distances.

```
DirectedWeightedGraph smallGraph;

// Add nodes (cities)
std::vector<Node> cities = {
    Node(10, 75), Node(45, 20), Node(70, 85), Node(30, 50),
    Node(55, 95), Node(90, 60), Node(20, 35), Node(80, 10)
};

for (int i = 0; i < cities.size(); ++i) {
    smallGraph.addNode(i, cities[i]);
}

// Use Euclidean distance as edge weight
smallGraph.addEdge(DirectedWeightedEdge(0, 1, 70.71f));
smallGraph.addEdge(DirectedWeightedEdge(0, 2, 60.83f));
smallGraph.addEdge(DirectedWeightedEdge(0, 3, 32.02f));
smallGraph.addEdge(DirectedWeightedEdge(1, 2, 67.18f));
smallGraph.addEdge(DirectedWeightedEdge(1, 3, 36.06f));
smallGraph.addEdge(DirectedWeightedEdge(1, 4, 75.33f));
smallGraph.addEdge(DirectedWeightedEdge(2, 4, 17.49f));
smallGraph.addEdge(DirectedWeightedEdge(2, 5, 29.68f));
smallGraph.addEdge(DirectedWeightedEdge(3, 6, 15.13f));
smallGraph.addEdge(DirectedWeightedEdge(4, 5, 47.17f));
smallGraph.addEdge(DirectedWeightedEdge(4, 7, 87.80f));
smallGraph.addEdge(DirectedWeightedEdge(5, 7, 50.00f));
smallGraph.addEdge(DirectedWeightedEdge(6, 7, 66.94f));
smallGraph.addEdge(DirectedWeightedEdge(7, 1, 37.42f));
```

Large Graph:

I had the large graph randomly generated with 10000 edges.

```
DirectedWeightedGraph largeGraph;

// Number of edges
int numEdges = 10000;

// Start with a certain number of nodes (e.g., square root of number of edges)
int numNodes = static_cast<int>(sqrt(numEdges));

// Add nodes
for (int i = 0; i < numNodes; ++i) {
    float x = static_cast<float>(rand() % 1000);  // Random x coordinate between 0 and 1000
    float y = static_cast<float>(rand() % 1000);  // Random y coordinate between 0 and 1000
    largeGraph.addNode(i, Node(x, y));
}

// Add edges
for (int i = 0; i < numEdges; ++i) {
    int source = rand() % numNodes;  // Random source node
    int destination = rand() % numNodes;  // Random destination node
    while (source == destination) {  // Ensure source and destination are not the same
        destination = rand() % numNodes;
    }
    // Calculate Euclidean distance between source and destination nodes
    float dx = largeGraph.getNode(source).x - largeGraph.getNode(destination).x;
    float dy = largeGraph.getNode(source).y - largeGraph.getNode(destination).y;
    float weight = sqrt(dx * dx + dy * dy);  // Euclidean distance
    largeGraph.addEdge(DirectedWeightedEdge(source, destination, weight));
}
```

Performance of Dijkstra and A*:

After implementing the Dijkstra and A* algorithms, I had both run the same graph and find the same shortest path, record their time consumed, and print them out.

The heuristic function I implemented for A* was an Euclidean distance between the coordinates of two cities:

```
float heuristic(const DirectedWeightedGraph& graph, int startNode, int endNode)
{
    //return ConstantCost(graph, startNode, endNode);
    return EuclideanDistance(graph, startNode, endNode);
}
```

```
float EuclideanDistance(const DirectedWeightedGraph& graph, int startNode, int endNode)
{
    float dx = graph.getNode(endNode).x - graph.getNode(startNode).x;
    float dy = graph.getNode(endNode).y - graph.getNode(startNode).y;
    return sqrt(dx * dx + dy * dy);
}
```

However, I found out soon that the order of the two algorithms affects their times recorded. Hence, I had both the algorithms be called once before calling them again to time them. Below are the results I got:

Small Graph:

Path: 3 -> 6 -> 7 -> 1 -> 2 -> 5 Dijkstra Time: 0.000028 A* Time: 0.000025	Path: 3 -> 6 -> 7 -> 1 -> 2 -> 5 Dijkstra Time: 0.000028 A* Time: 0.000025	Path: 3 -> 6 -> 7 -> 1 -> 2 -> 5 Dijkstra Time: 0.000030 A* Time: 0.000033
Path: 3 -> 6 -> 7 -> 1 -> 2 -> 5 Dijkstra Time: 0.000029 A* Time: 0.000026	Path: 3 -> 6 -> 7 -> 1 -> 2 -> 5 Dijkstra Time: 0.000051 A* Time: 0.000047	Path: 3 -> 6 -> 7 -> 1 -> 2 -> 5 Dijkstra Time: 0.000048 A* Time: 0.000041

Large Graph:

Path: 3 -> 95 -> 6 Dijkstra Time: 0.002728 A* Time: 0.000211	Path: 3 -> 6 Dijkstra Time: 0.002456 A* Time: 0.000162	Path: 3 -> 6 Dijkstra Time: 0.002599 A* Time: 0.000174
Path: 3 -> 6 Dijkstra Time: 0.002336 A* Time: 0.000175	Path: 3 -> 87 -> 10 Dijkstra Time: 0.004072 A* Time: 0.000484	Path: 3 -> 94 -> 50 Dijkstra Time: 0.004116 A* Time: 0.000220

For the small graph, the time difference between the two algorithms was very small, with A* being slightly faster than Dijkstra at most of the times.

For the large graph, the time difference between the two algorithms was very large, with A* being consistently 10 – 20 times faster than Dijkstra.

The main reason why A* is faster than Dijkstra is that it uses a heuristic function to guide its search, which often results in fewer nodes being explored, making it faster, especially on large graphs.

The reason why the time difference was not obvious on the small graph is that while doing pathfinding on the small graph, the Dijkstra algorithm did not have many more nodes to explore than A* had.

Heuristics:

I implemented two heuristics functions. One is a constant guess; another is a Euclidean distance:

```
float heuristic(const DirectedWeightedGraph& graph, int startNode, int endNode)
{
    //return ConstantCost(graph, startNode, endNode);
    return EuclideanDistance(graph, startNode, endNode);
}
```

```
float ConstantCost(const DirectedWeightedGraph& graph, int startNode, int endNode)
{
    return 0.0f;
}
```

```
float EuclideanDistance(const DirectedWeightedGraph& graph, int startNode, int endNode)
{
    float dx = graph.getNode(endNode).x - graph.getNode(startNode).x;
    float dy = graph.getNode(endNode).y - graph.getNode(startNode).y;
    return sqrt(dx * dx + dy * dy);
}
```

Admissibility: A heuristic is admissible if it never overestimates the cost to reach the goal.
Consistency: A heuristic is consistent (or monotone) if for every node N and each successor P of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost of reaching the goal from P.

For the constant guess:
It is admissible because it always returning 0. It is consistent because of the same reason.

For the Euclidean distance:
It is admissible because its return value is never greater than the actual cost (it is exactly the cost). It is consistent because the calculated cost decreases when getting closer to the goal.

If using the constant guess which always returns 0, it effectively converts the A* algorithm into a Dijkstra algorithm, which leads to its performance being the same as the Dijkstra algorithm.

If using the Euclidean distance as the heuristics function, the A* algorithm's performance becomes much better due to exploring less nodes.

Implementing It on the Boid:

I created a tile map data structure which is a two dimensional array of Booleans, 0 represents empty, 1 represents obstacle:

```
class TileMap {
public:
    TileMap(int width, int height);
    bool isObstacle(int x, int y) const;
    void SetObstacle(int x, int y, bool obstacle);
    int GetWidth() const { return width; }
    int GetHeight() const { return height; }
    float GetTileSize() const { return tileSize; }
    float SetTileSize(float size) { tileSize = size; }

    float GetxPosition(int x) const { return x * tileSize; }
    float GetyPosition(int y) const { return y * tileSize; }
    int GetxIndex(float x) const { return x / tileSize; }
    int GetyIndex(float y) const { return y / tileSize; }

private:
    int width, height;
    float tileSize = 10;
    std::vector<std::vector<bool>> map;
};
```

Then I made a pathfinding boid that derives from my common Boid class:

```
class PathfindingBoid : public Boid
{
public:
    PathfindingBoid(TileMap* map, const int i_xCoordinate, co

    void Update(const float i_deltaTime) override;

    void Draw() override;

    // Input functions
    void mousePressed(int x, int y, int button) override;
    void mouseDragged(int x, int y, int button) override;

private:
    TileMap* map;
    Cell pathfindingTarget;
    std::vector<Cell> path;

    const float maxSpeed = 50.0f;

};
```

In the new class, the mousePressed() and mouseDragged() does not set the seek target anymore, instead, it sets the pathfinding target and computes a path using A*:

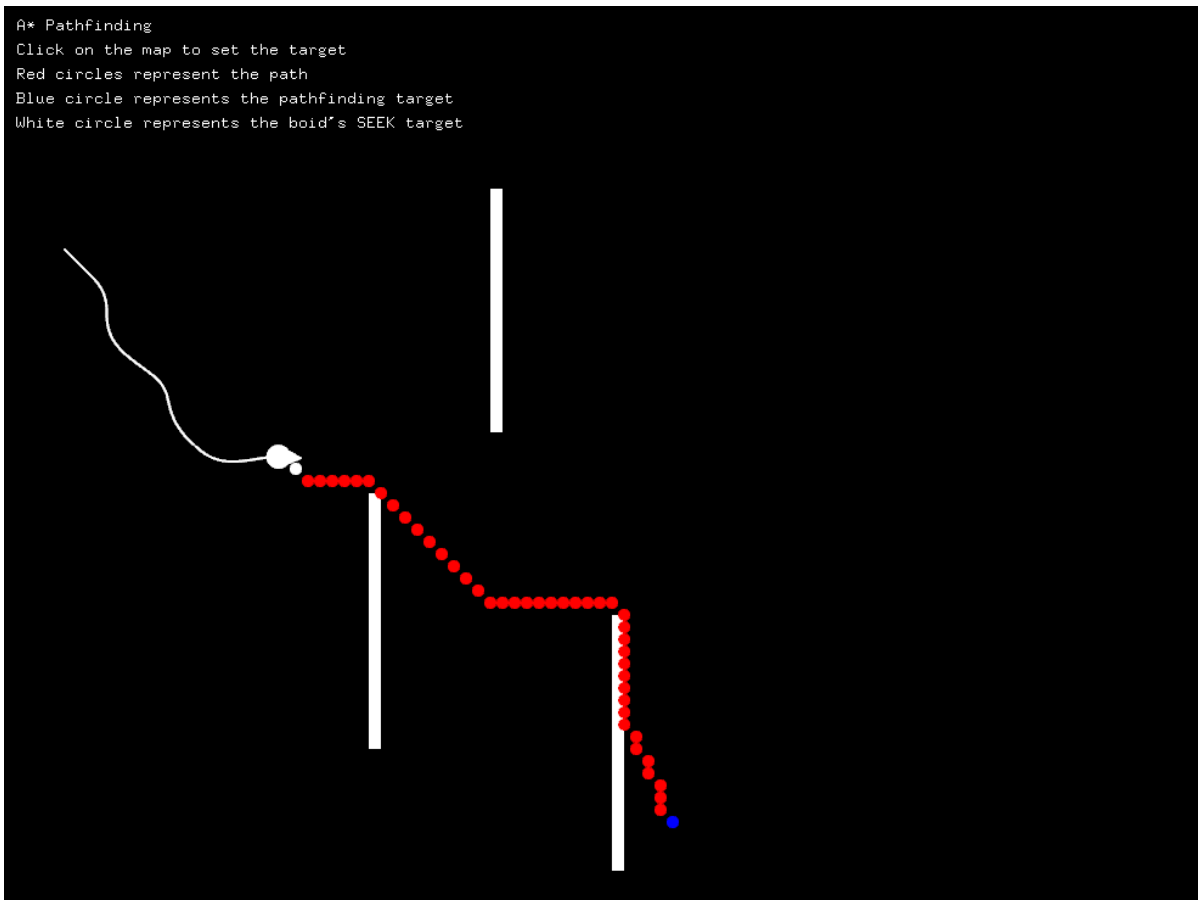
```
void PathfindingBoid::mousePressed(int x, int y, int button)
{
    this->pathfindingTarget = Cell(map->GetxIndex(x), map->GetyIndex(y));
    path = AStar(*map, Cell(map->GetxIndex(this->GetPosition().x), map->GetyIndex(this->GetPosition().y)), p
    target = eae6320::Math::sVector(map->GetxPosition(path[0].x), map->GetyPosition(path[0].y), 0.0f);
}
```

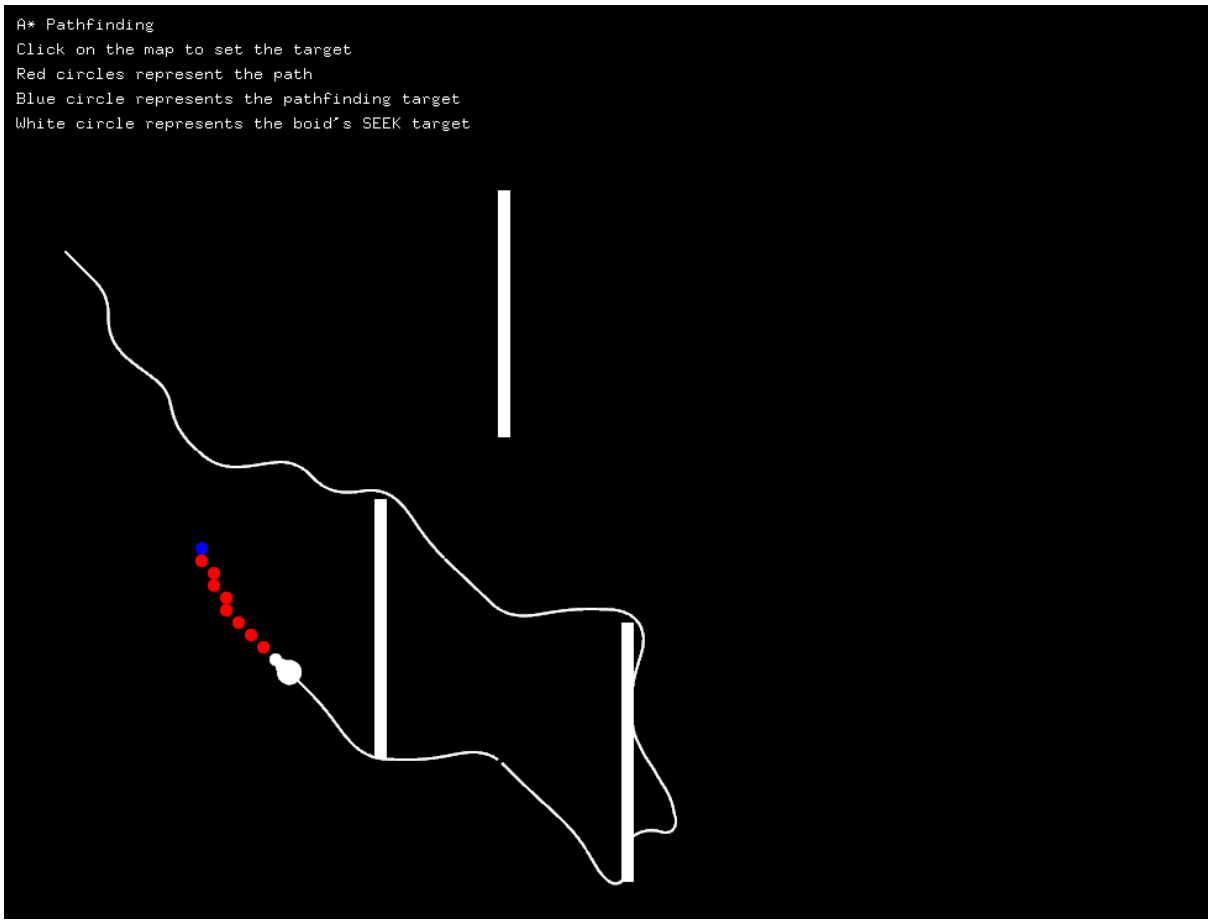
Then, I update the next target for SEEK behavior in the Update() function:

```
// Update the target based on the pathfinding path
if (!path.empty())
{
    // If the current position is close enough to the current target
    if ((target - this->GetPosition()).GetLength() < map->GetTileSize())
    {
        // Remove the current target from the path
        path.erase(path.begin());

        // If there are still cells left in the path
        if (!path.empty())
        {
            // Update the target with the next cell in the path
            target = eae6320::Math::sVector(map->GetxPosition(path[0].x), map->GetyPosition(path[0].y), 0.0f);
        }
    }
}

this->m_rigidBody->AddForce(Seek(target));
```

Result:


A* Pathfinding
Click on the map to set the target
Red circles represent the path
Blue circle represents the pathfinding target
White circle represents the boid's SEEK target


A* Pathfinding
Click on the map to set the target
Red circles represent the path
Blue circle represents the pathfinding target
White circle represents the boid's SEEK target


A* Pathfinding
Click on the map to set the target
Red circles represent the path
Blue circle represents the pathfinding target
White circle represents the boid's SEEK target