# Academic Reference

We are proposing to implement the A* algorithm, which, broadly, is an algorithm used for graph traversal and path search in order to find the path of minimum length from a start node to a goal node. For more information regarding a more comprehensive definition of the algorithm, please refer to A Formal Basis for the Heuristic Determination of Minimum Cost Paths.

# Algorithm Summary

The A* search algorithm is an algorithm used to find the shortest path from a specified start to a specified goal in a graph. It works by considering possible paths and making decisions at each step based on a combination of two values: the actual cost (known as "g" cost) to reach a particular point from the start and an estimated cost (known as the "h" cost) to reach the goal from that point. A* combines these costs to calculate an overall score ("f" cost) for each point, and it explores paths in order of increasing f-cost. This way, A* searches for the optimal path while considering the actual distance traveled and an estimate of the remaining distance to the goal. It terminates when it explores the goal point, and the shortest path can be reconstructed by tracing back through the explored points.

# Function I/O

1. vector<vector<Point>> toArray(std::string file)

@param file -- the absolute path to the preprocessed .txt file of 1s and 0s
@return – a 2D vector of ints that represents the grid, where 1s represent nodes and 0s represent obstacles

This function will create the grid, represented as a 2D vector of Points, which we will be traversing from a txt file. We will have a class called Graph, which stores the 2D array, as well as its height and width, as private member variables.

2. Graph::Graph(vector<vector<Point>> graph, int width, int height)

@param graph – a 2D array of Points used as the graph
@param int, height – integer numbers representing the height and width of the graph.

This function is the constructor for the Graph that will be traversed by our algorithm

3. Point::Point()

Default constructor for the Point class, which has private member variables bool value_, representing whether or not the Point is an obstacle or node, h_ (the estimated cost), g_ (actual cost), and f_ (the sum of the g and h costs).

4. Point::Point(bool value, int g, int h)

@param g – the actual cost to reach a particular point from the start and an estimated cost
@param h - the estimated cost to reach the goal from that point
@param value - the boolean that represents whether or not the point is an obstacle

Point constructor that sets a points' value as either an obstacle or a node. Also sets the g and h values of the node with respect to the start and goal point, if applicable.

5. int getHeuristic(Point endPoint, Point testPoint)

@param endPoint – our goal point that we are attempting our algorithm to reach
@param testPoint -- the point whose heuristic value we are returning
@return – the h value of testPoint

This function returns the h value of testPoint given the start and end points of the traversal. The heuristic value will be calculated using the Euclidean distance from the test point to the endPoint. Test will make sure that found heuristic value will equals the actual heuristic value of the point.

6. vector<Point> findPath(Graph graph, Point startPoint, Point endPoint)
@param graph – the graph that we will be performing the traversal on
@param startPoint – the starting point of the traversal
@param endPoint – the goal point of the traversal
@return – a vector of Points that represents the shortest path from the startPoint to the endPoint on our graph

Function that finds the shortest path on the Graph graph from Point startPoint to Point endPoint. Returns an array of points that represents the shortest path, starting from the startPoint and ending at the endPoint. Tested by comparing the return value of the function on given dataset, and comparing to the actual shortest path vector of that dataset.

# Data Description

The smaller test datasets (3x3, 6x3, 6x5, 10x10) were written manually in order to guarantee accuracy. However, in order to create the large datasets, we used this website in order to randomly generate matrices with specified dimensions, which we then copied and pasted into the .txt files in /datasets. I then manually inserted 1s in place of certain 0s in order to make sure that certain nodes were connected to others throughout the entire graph.