SHARATH SREENIVASAN
STUDENT ID - 200109355

CSC 584 BUILDING GAME AI HOMEWORK 2 – REPORT

The main task of the assignment was to implement the Dijkstra's and A-star algorithm on a small and a big graph followed by the usage of the A-star algorithm in an indoor environment containing a number of obstacles. The character follows a computed path along the graph before seeking towards the mouse click. Processing 3 was used to implement the algorithms, each of which have been described in detail below.

Obtaining Graphs

The smaller graph is a road map of the interstate highways connecting some of the major cities in the United States of America. Since it consists of roads, every edge is bi directional which means, for every pair of vertices (x1, y1) and (x2, y2) there exists an edge u from (x1, y1) to (x2, y2) and vice versa. The edges in the graph indicate the distance between the cities by road. Every point in the graph is connected to every other point in it. The road distance was taken by plugging in the city names on Google Maps. The small graph consists of 27 vertices. A snapshot of the small graph is as shown below.

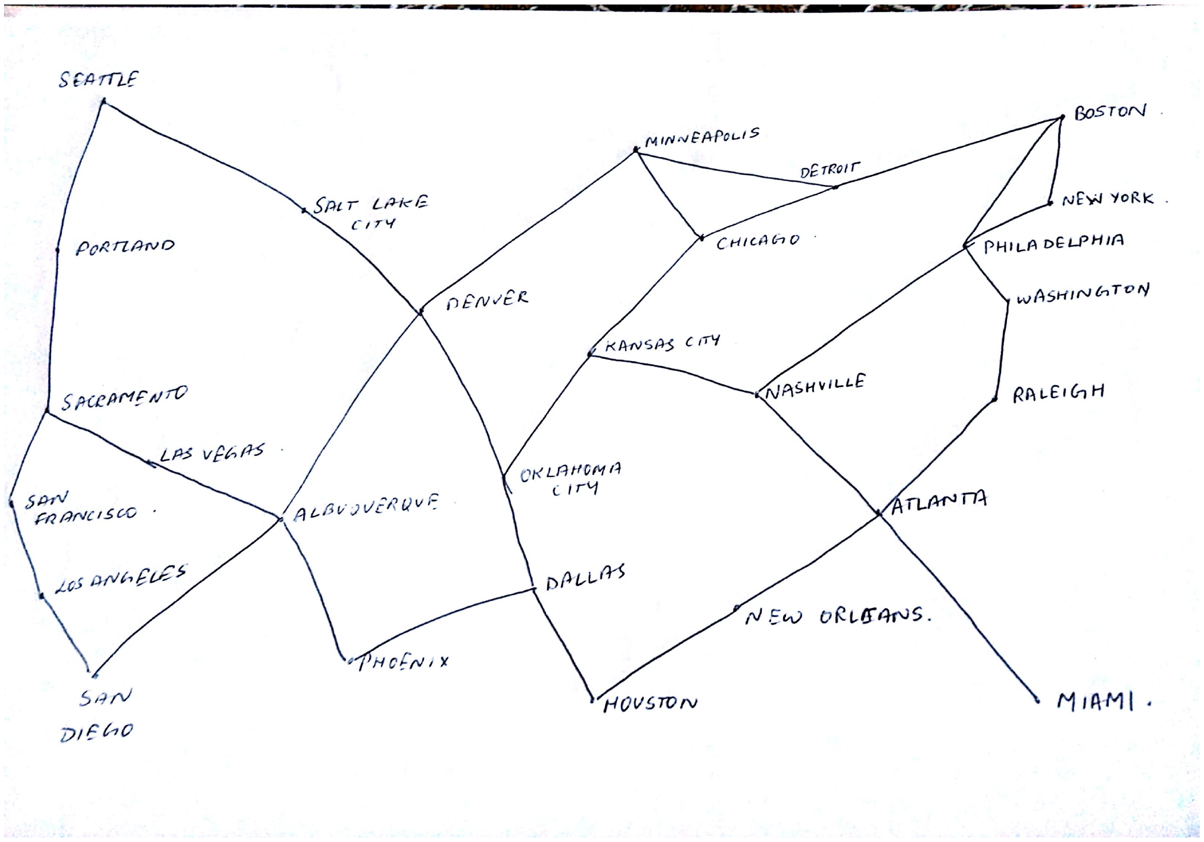


Fig 1 – Small Graph

I obtained the larger graph by using a dataset taken from an earlier course on Social Computing (CSC 555) which was used for processing and analysis. It consists of a network facebook.com nodes. The file was processed using Python. The spaces were removed, commas were inserted at the right place and edge length between two vertices to allow smooth usage in the codes. The resulting file consisted of a series of lines which had the source vertex, the destination vertex and the edge length between the two. The edge length chosen was a random integer. The large graph consisted of 1967 vertices.

A* and Dijkstra's Algorithm

The nodes.txt text file contains the index, name, latitude and longitude of a given city. Each line is split individually and every city name is mapped to an integer with the use of a hash map. The edges file determines the two vertices that are connected and the distance between the said vertices. The algorithm goes through the file and with the help of the hash map, maps the city name to its corresponding indices and adds the distance value in the adjacency matrix. Since the roads are bi directional, it results in a symmetric adjacency matrix which is used in both Dijkstra's and A* algorithms. Additionally, in A*, the values of the latitude and longitude are stored in individual hash maps.

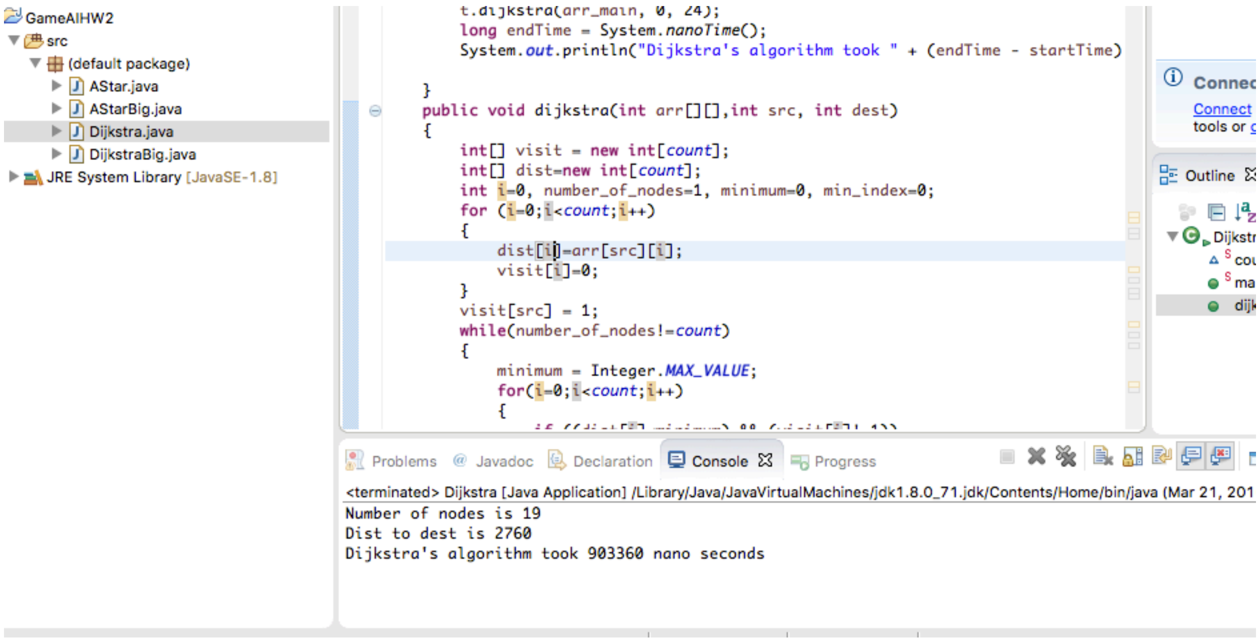
In the Dijkstra's algorithm implementation, a visit array is used to keep track of all the nodes that are visited. Each time the distance is lesser than the minimum distance, the minimum distance is set to the lower value and the visit of that node is updated to one. The distance to the next point in the path is calculated and the lowest value is stored in the distance matrix. When the minimum index is the destination, the loop exits and the number of 1's in the visited array is counted since each time a node is visited, the corresponding value in the visit array is updated to 1.

The A* algorithm implementation is an extension of the Dijjkstra's algorithm. The heuristics used is the Euclidean distance between the intermediate node and destination. As previously mentioned, the latitude and longitude values are stored in two hash maps and the distance is calculated between them using a mathematical formula. Care should be taken to ensure the right path is entered for the file input in both cases.

On performance comparison, it can be seen that the number of nodes visited in the A* algorithm for the same source and destination is lesser than the number of nodes visited in the Dijkstra's algorithm for the smaller graph. However, it should be mentioned that since the graph is relatively small, number is not significantly lesser. As the size of the graph increases, the difference in the number of visited nodes increases as well. In the bigger graph, there Is no way to check this because we cannot be sure if the graph is admissible or not since it has been constructed using random edge lengths.

In my implementation of the Dijkstra's and A* algorithm, the memory used is almost the same because both use the concept of adjacency matrix. In the A*, hash maps for latitudes and longitudes and the array for the estimated cost contributes to the extra memory usage which is not present in Dijkstra's. This is negligible compared to the 2D adjacency matrix which is an nxn matrix where n is the number of vertices in the graph.

In the case of runtime, the A* algorithm takes significantly longer than the Dijkstra's algorithm. This is because of the extra computation of the heuristics and the hash maps usage for the co-ordinates of the vertex. Screenshots to accompany the facts mentioned above are provided below.

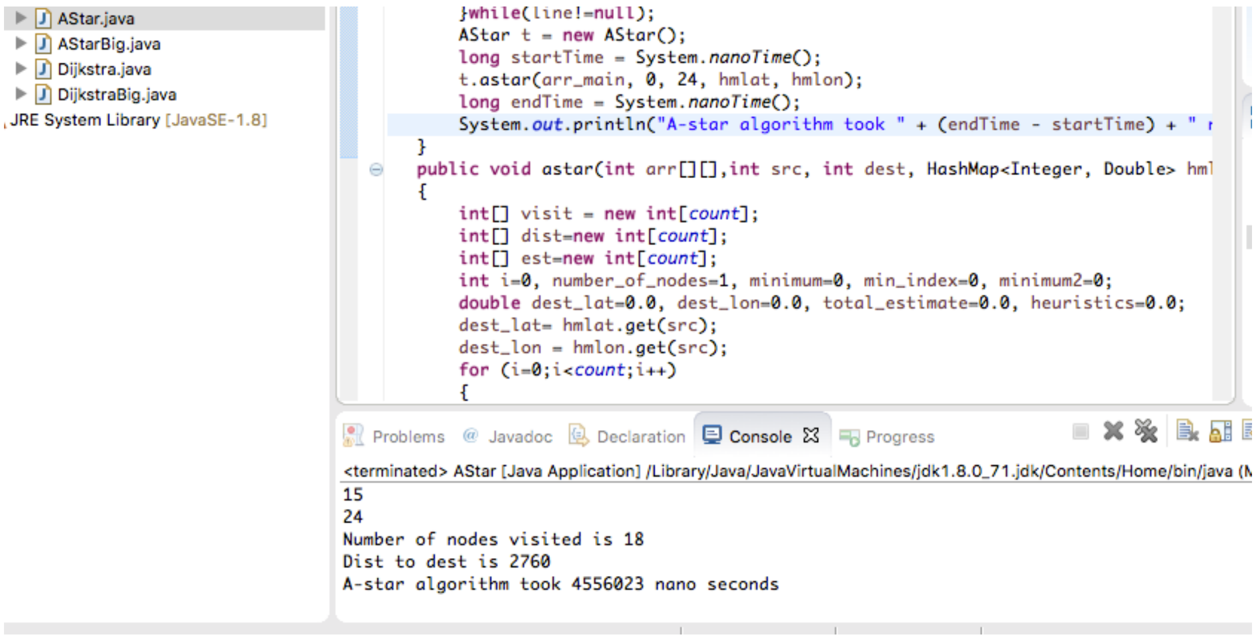


```
        t.dijkstra(arr_main, 0, 24);
        long endTime = System.nanoTime();
        System.out.println("Dijkstra's algorithm took " + (endTime - startTime)

    }
    public void dijkstra(int arr[][],int src, int dest)
    {
        int[] visit = new int[count];
        int[] dist=new int[count];
        int i=0, number_of_nodes=1, minimum=0, min_index=0;
        for (i=0;i<count;i++)
        {
            dist[i]=arr[src][i];
            visit[i]=0;
        }
        visit[src] = 1;
        while(number_of_nodes!=count)
        {
            minimum = Integer.MAX_VALUE;
            for(i=0;i<count;i++)
            {
```

```
<terminated> Dijkstra [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_71.jdk/Contents/Home/bin/java (Mar 21, 201
Number of nodes is 19
Dist to dest is 2760
Dijkstra's algorithm took 903360 nano seconds
```

Fig 2 – Dijkstra's Algorithm

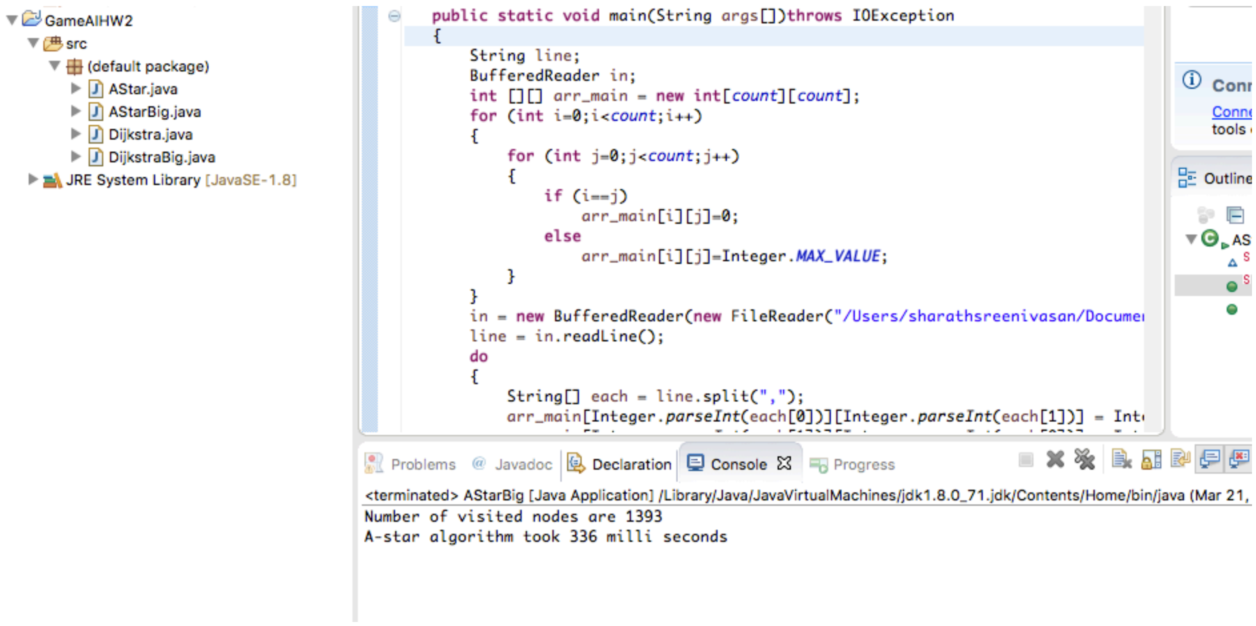


```
        }while(line!=null);
        AStar t = new AStar();
        long startTime = System.nanoTime();
        t.astar(arr_main, 0, 24, hmlat, hmlon);
        long endTime = System.nanoTime();
        System.out.println("A-star algorithm took " + (endTime - startTime) + " 
    }
    public void astar(int arr[][],int src, int dest, HashMap<Integer, Double> hml
    {
        int[] visit = new int[count];
        int[] dist=new int[count];
        int[] est=new int[count];
        int i=0, number_of_nodes=1, minimum=0, min_index=0, minimum2=0;
        double dest_lat=0.0, dest_lon=0.0, total_estimate=0.0, heuristics=0.0;
        dest_lat= hmlat.get(src);
        dest_lon = hmlon.get(src);
        for (i=0;i<count;i++)
        {
```

```
<terminated> AStar [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_71.jdk/Contents/Home/bin/java (
15
24
Number of nodes visited is 18
Dist to dest is 2760
A-star algorithm took 4556023 nano seconds
```

Fig 3 – A-star Algorithm

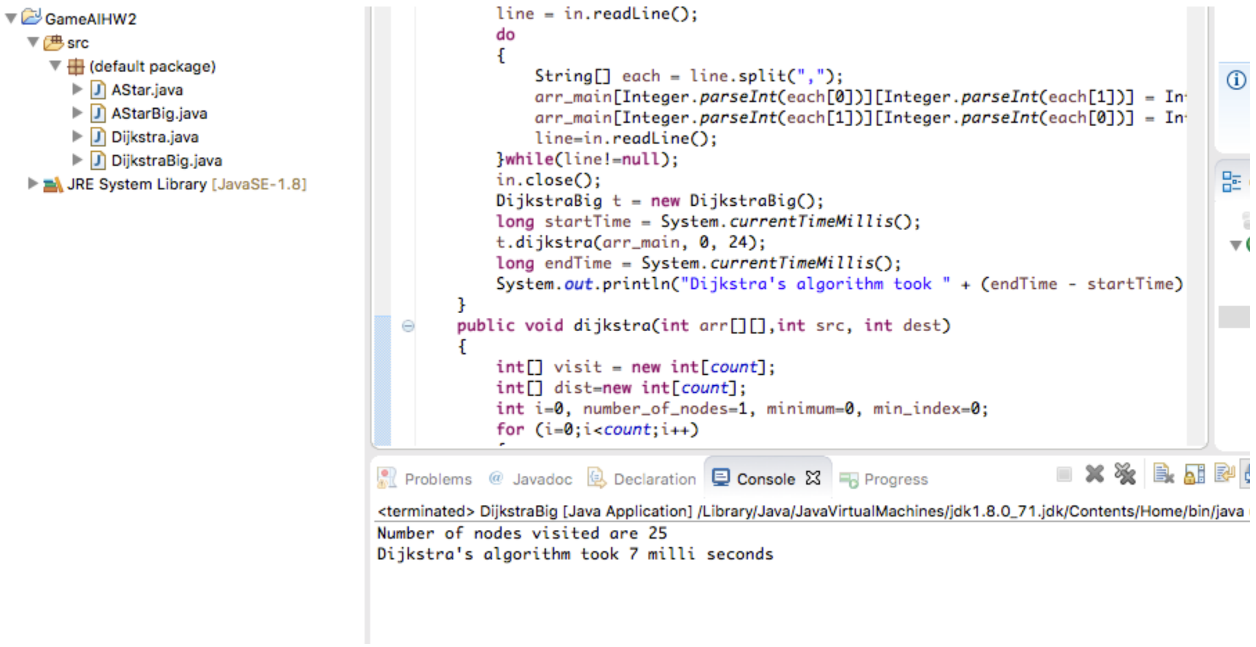
Larger the graph, the longer the Dijkstra's and A* algorithm. It can be seen that the rise in the execution is significantly larger in the case of A* because of the heuristics computation. The number of nodes visited in A* is also higher which is probably due to the fact the edges are of random length. The graph in itself might be impossible to construct. The gap in the memory usage also increases by a large margin because of the usage of the extra arrays for computational purposes. The number of nodes visited for A* is also much higher than Dijkstra's which is impossible thereby proving the graph is flawed. Screenshots to accompany are shown below.



```
    public static void main(String args[])throws IOException
    {
        String line;
        BufferedReader in;
        int [][] arr_main = new int[count][count];
        for (int i=0;i<count;i++)
        {
            for (int j=0;j<count;j++)
            {
                if (i==j)
                    arr_main[i][j]=0;
                else
                    arr_main[i][j]=Integer.MAX_VALUE;
            }
        }
        in = new BufferedReader(new FileReader("/Users/sharathsreenivasan/Docume
        line = in.readLine();
        do
        {
            String[] each = line.split(",");
            arr_main[Integer.parseInt(each[0])][Integer.parseInt(each[1])] = Int
```

```
<terminated> AStarBig [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_71.jdk/Contents/Home/bin/java (Mar 21,
Number of visited nodes are 1393
A-star algorithm took 336 milli seconds
```

Fig 4 – A* for big graphs.



```
        line = in.readLine();
        do
        {
            String[] each = line.split(",");
            arr_main[Integer.parseInt(each[0])][Integer.parseInt(each[1])] = In
            arr_main[Integer.parseInt(each[1])][Integer.parseInt(each[0])] = In
            line=in.readLine();
        }while(line!=null);
        in.close();
        DijkstraBig t = new DijkstraBig();
        long startTime = System.currentTimeMillis();
        t.dijkstra(arr_main, 0, 24);
        long endTime = System.currentTimeMillis();
        System.out.println("Dijkstra's algorithm took " + (endTime - startTime)
    }
    public void dijkstra(int arr[][],int src, int dest)
    {
        int[] visit = new int[count];
        int[] dist=new int[count];
        int i=0, number_of_nodes=1, minimum=0, min_index=0;
        for (i=0;i<count;i++)
```

```
<terminated> DijkstraBig [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_71.jdk/Contents/Home/bin/java
Number of nodes visited are 25
Dijkstra's algorithm took 7 milli seconds
```

Fig 5 – Dijkstra's for big graphs

Heuristics

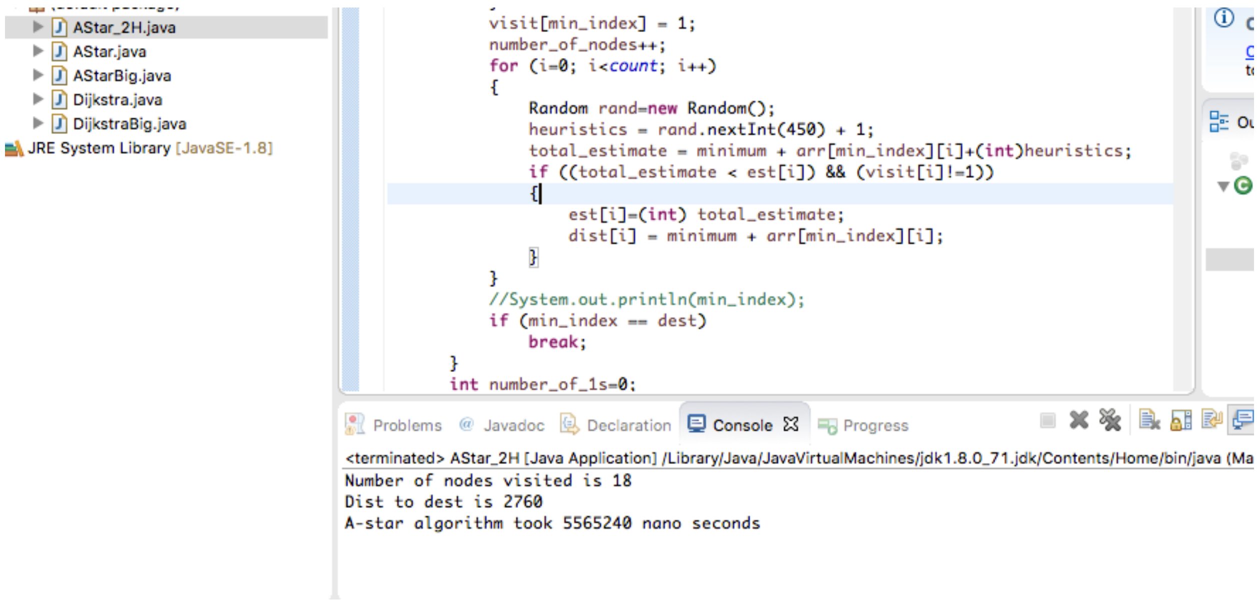
The two heuristics that I chose to use are

1. Euclidean Distance
2. Random Integer.

The Euclidean distance basically uses a function which converts the difference in the latitude and longitude values between the intermediate vertex and the destination into a distance with the help of a formula. The heuristic chosen is an admissible underestimating heuristic as the distance computed is a straight line path from the source to the destination whereas in the graph it is the distance by road. The straight line path is always lesser than or equal to the distance travelled from the intermediated source to the destination by road. An admissible heuristic results in an optimal solution and increases the accuracy of the A* algorithm.

The second heuristic, a random integer, is chosen between 1 and 450. A random integer results in an inadmissible heuristic as the heuristic values are assigned at random. At times, the value could be more than the actual number and at times it could be lesser. There is no way to determine this. Consequently, the random number heuristic is inadmissible.

Screenshots for the random heuristic is as shown below: -



```
        visit[min_index] = 1;
        number_of_nodes++;
        for (i=0; i<count; i++)
        {
            Random rand=new Random();
            heuristics = rand.nextInt(450) + 1;
            total_estimate = minimum + arr[min_index][i]+(int)heuristics;
            if ((total_estimate < est[i]) && (visit[i]!=1))
            {
                est[i]=(int) total_estimate;
                dist[i] = minimum + arr[min_index][i];
            }
        }
        //System.out.println(min_index);
        if (min_index == dest)
            break;
    }
    int number_of_1s=0;
```

```
<terminated> AStar_2H [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_71.jdk/Contents/Home/bin/java (Ma
Number of nodes visited is 18
Dist to dest is 2760
A-star algorithm took 5565240 nano seconds
```

Fig 6 – A* with random heuristic

Combining

The indoor environment that I chose to represent is the ground floor of my house. The environment is as shown in the screenshot below.

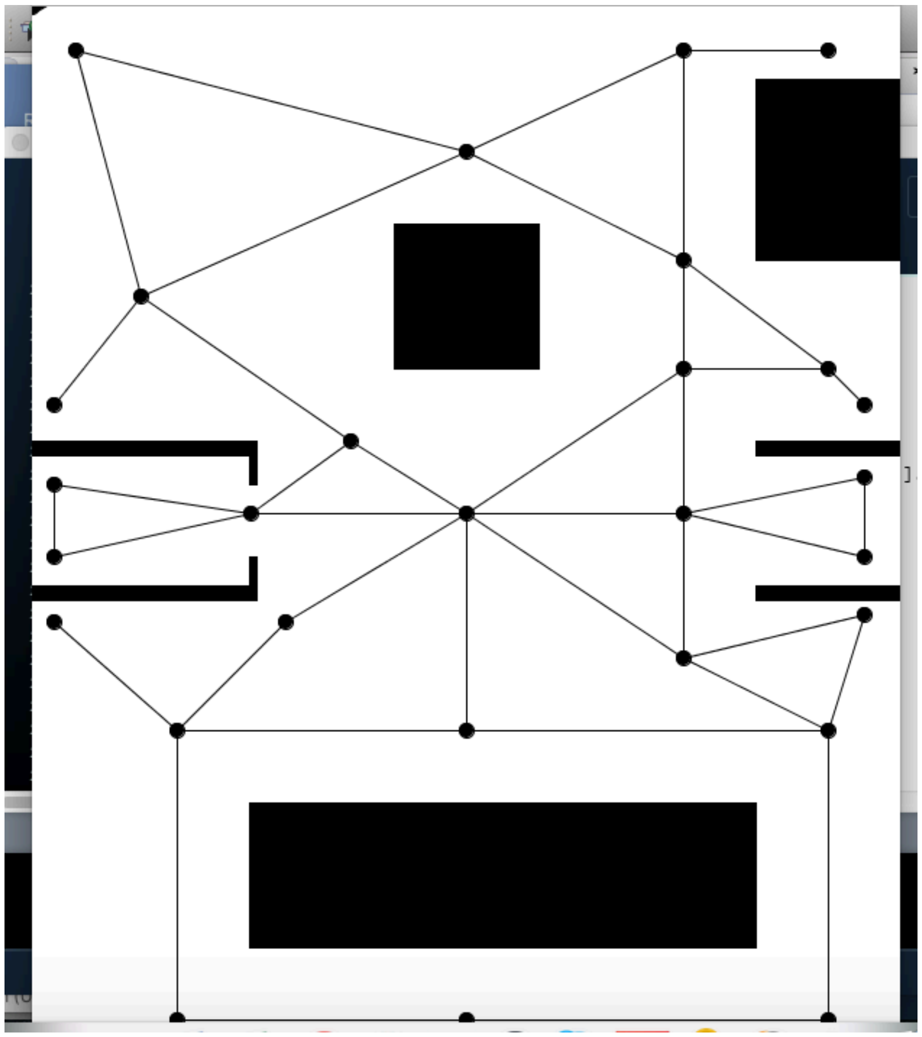


Fig 7 – Indoor environment

The graph was constructed by representing each point in the environment as co-ordinates. Each pair is mapped to an integer which represents the node. The graph was carefully constructed to avoid the obstacles. The graph was built in such a way so as to avoid the implementation of obstacle avoidance. The adjacency matrix was calculated for the graph with the distance between the two vertices being the Euclidean distance. The A* algorithm for a given source and destination was implemented on the graph. Both the source and the destination of the character was localized to the nearest point on the graph. This was done by calculating the least distance between the position of the character and all the nodes followed by seeking to the nearest node on the graph.

The A* algorithm uses the Euclidean distance between the intermediate vertices and the destination as its heuristic. The astar function returns the path from the source to the destination. The character seeks the points on the path. For instance, let the function return [1, 5, 7, 8] the character seeks 1, followed by 5 then 8 and so on. At the end, after traversing the entire path array, the character then seeks the position of the mouse click. The entire path is lit up in a different color to show the passage of the character along the lines of the graph. Another way of implementing the seek is to keep checking the distance between the character and the mouse click and when the distance to the mouse click is lesser than the next point on the graph the character seeks to the mouse click. However, the issue with this implementation is that the character could pass through walls and tables.

Screenshots that show the path-following are as shown below

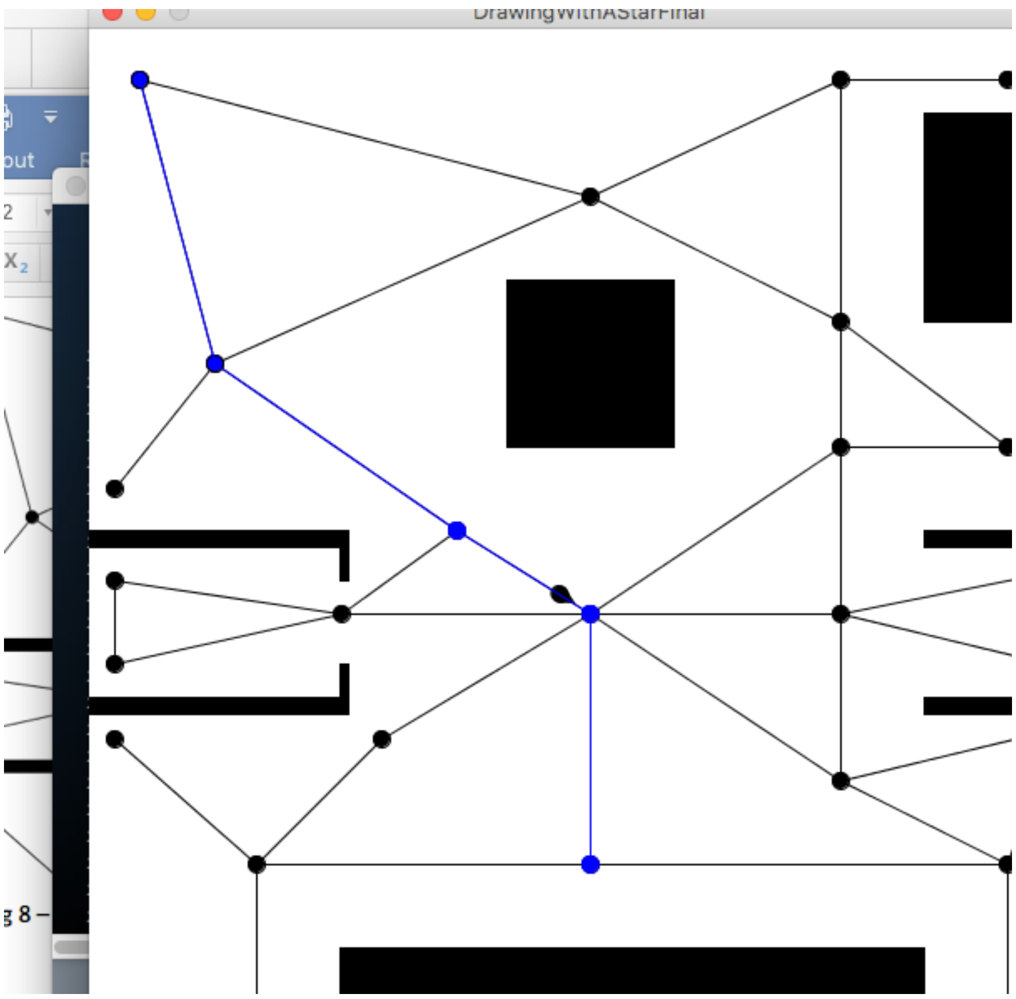


Fig 8 – Character traversing through the nodes in the path array

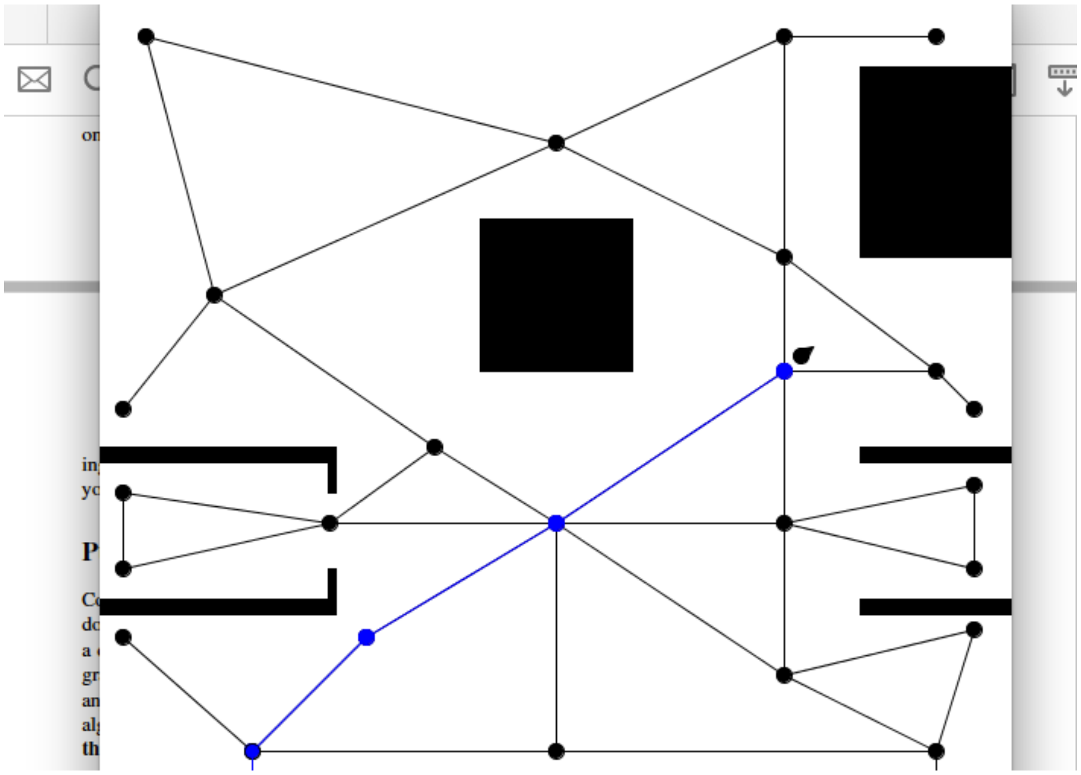


Fig 9 – Seeking target after passing all the nodes in the path