Assignment 2 – Algorithms

Christopher LaChance

1. Game Engine - LibGDX

I chose to use LibGDX for this assignment. The main reason that I chose to use this game engine was because it seemed to have the smallest learning curve. I do not have very much experience programming video games, and I wanted to spend the most time focusing on the algorithms and not on the game engine. Not only does it have the simplest game loop, but it also uses Java as its code base. I am very familiar with Java and that gave me the opportunity to focus on the actual programming of the game. This game engine gave us access to keyboard inputs for controls, an Intersector class to handle collisions, and render function to display all of the game components. I also used the Intersector class to check if the navigation nodes on the navigation graph are valid and can be travelled to.

2. Code Specifics and Explanations

When reading the instructions, I noticed that it could be very easily broken down into 3 main sub problems. The first problem was to implement a simple seek function. The next problem was running the A* algorithm given a source and destination. The final problem was to use the seek function to navigate along the A* shortest path. To accomplish these tasks, I broke the application up into 4 states (Seek, Display A*, Navigate A*, and Evaluate Sensors). The "Evaluate Sensors" state is used to perform the sensor evaluations from the first assignment. The user can easily switch states using keyboard input. This was accomplished by using a singleton called the ApplicationModeSingleton to keep track of the application mode. Using these states, each component of the assignment can be easily tested independently, and the performance of these algorithms can be seen in real time.

I created an interface called RenderedObjects that all drawn objects must extend. The interface contains a function "draw()" that specifies how that object should be drawn. Then, in my main GameAI object, all of these rendered objects are added to a list. This way, in order to draw the game objects in the render() function, I just go through the list and call draw() for each one.

The other main data structure in the game is a list of game agents. In my main GameAI object, all of the game agents (and walls too) are added to this list. This list is used when we detect collisions and for checking if a navigation node can be travelled to.

The navigation graph is implemented by a 2-dimensional array of NavigationNodes. The reason I chose to use a 2D array to represent the graph was because I wanted to use a very dense mesh of nodes, that way the user can navigate to most of the map. The navigation nodes contain a location, the cost to that node, the heuristic, and the guess of the total distance to the goal (cost to that node + heuristic). These nodes are used to implement the A* algorithm. The navigation nodes are drawn in the Background object (as they are not particularly a game component themselves, but could be considered part of the map). Only the navigation nodes that can be reached are drawn. Note that a node is considered reachable if the character can move its center to that node position and not be colliding with any game objects.

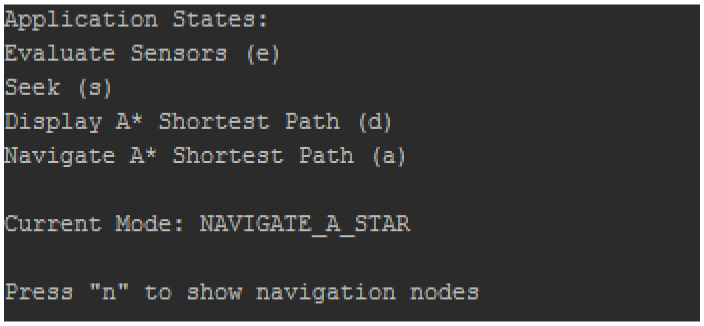
The seek method was implemented very similarly to the book. The only changes I made were to limit the speed of the character to less than the max speed and to update the characters heading to face the direction it is seeking. The reason why I chose to limit the speed of the character in the seek method was to make the movement seem a bit more realistic since we are not using an acceleration function. The reason why I wanted the character to always face in the direction of the seek position was to make the navigation along the A* path look better.

The primary component of this assignment was to implement an A* algorithm. The procedure that I came up with to implement A* was to use a priority queue, called the search queue, to choose which node to search. When we start the algorithm, we look at all of the nodes adjacent to the start. When we “look” at those nodes, we calculate the cost to that node (just one plus the previous node), the heuristic (Manhattan distance), total, and previous node. Then, we add these nodes to the search queue. The search queue is a priority queue that is sorted based on the lowest total comes first. To break ties, we want to pick the nodes that we most recently added to the search queue. This will essentially keep the algorithm on the same path until it hits an unreachable node, instead of bouncing around between all of the possible shortest paths, where the totals are all tied. We continuously choose the best node by removing the first node from the search queue (and since it is a sorted priority queue, the first node is the best node). We have reached the destination node when our current node’s heuristic is 0. Then, we can get the shortest path by tracing the back to the start node from the current node (which will be the destination node when we start tracing). Each node has a link to the previous node that searched it. Since we update each nodes previous node when searching adjacent nodes in A*, it is easy to trace back to the start node.

To implement the Display A* functionality, all we have to do is compute the A* shortest path. Then, from within Background, we draw the path nodes from the start to the destination. To implement the Navigate A* functionality, we just do the A* shortest path, and seek to each node. As we seek to a node, we remove it from the shortest path list when we get to it. That takes care of the dynamic path display as we navigate along it. Removing the node from the shortest path list as we arrive at it also makes it easy to seek to the next node, since we are always just seeking to the first node in the shortest path list.

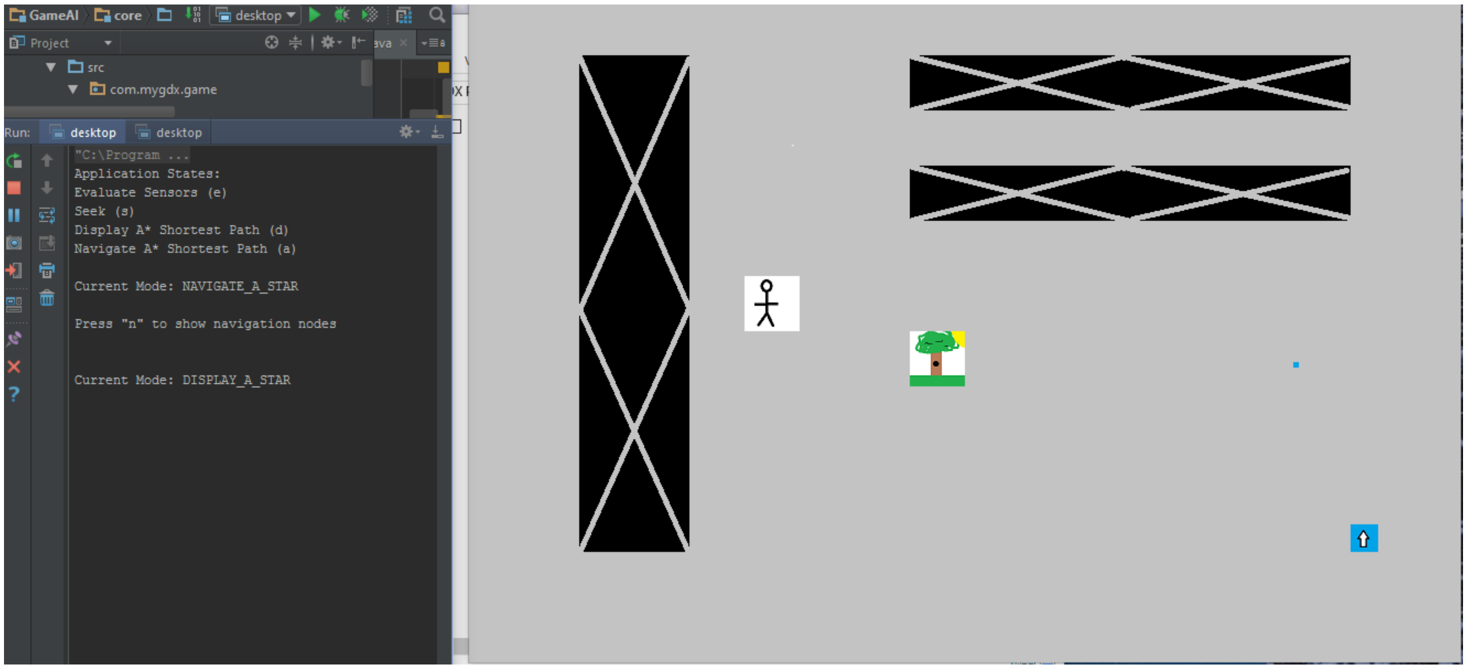
3. Screenshots

Application Header (displayed at the start of the application):



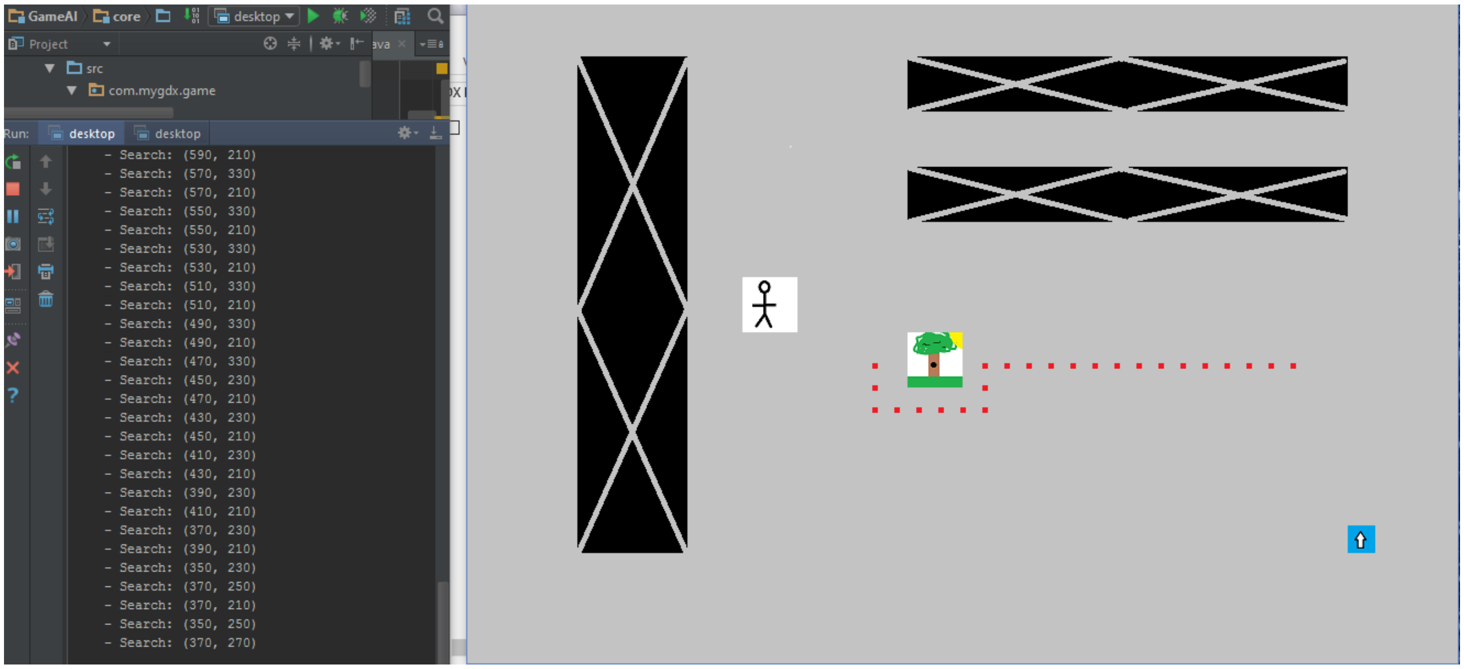
```
Application States:
Evaluate Sensors (e)
Seek (s)
Display A* Shortest Path (d)
Navigate A* Shortest Path (a)

Current Mode: NAVIGATE_A_STAR

Press "n" to show navigation nodes
```

Display A* shortest Path after one click (note the blue navigation node above the character)



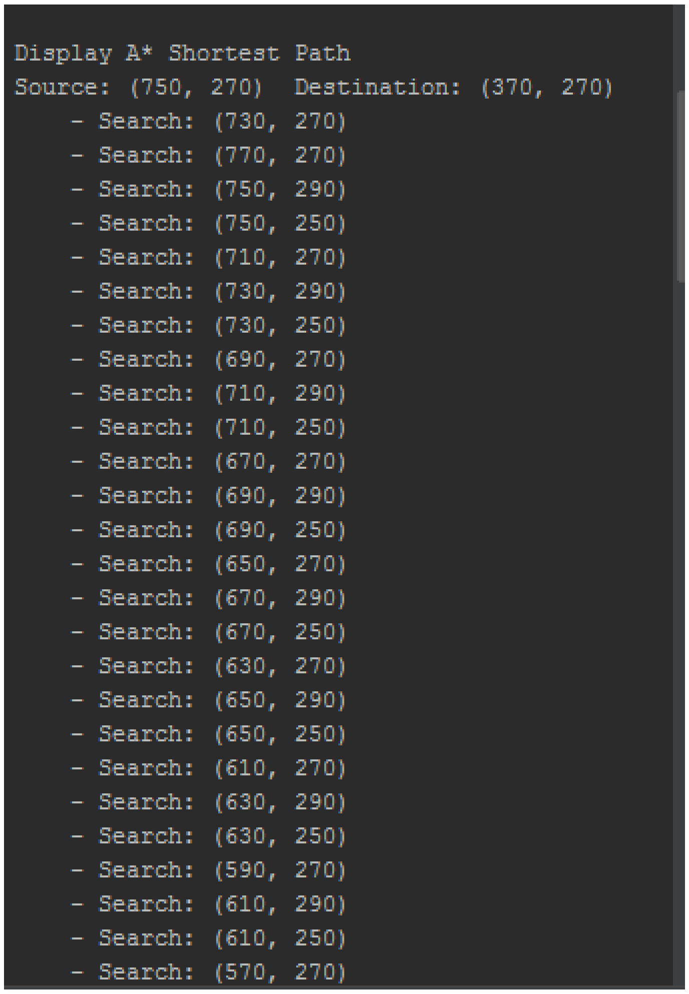
```
"C:\Program ...
Application States:
Evaluate Sensors (e)
Seek (s)
Display A* Shortest Path (d)
Navigate A* Shortest Path (a)

Current Mode: NAVIGATE_A_STAR

Press "n" to show navigation nodes


Current Mode: DISPLAY_A_STAR
```

Display A* shortest path after a second click



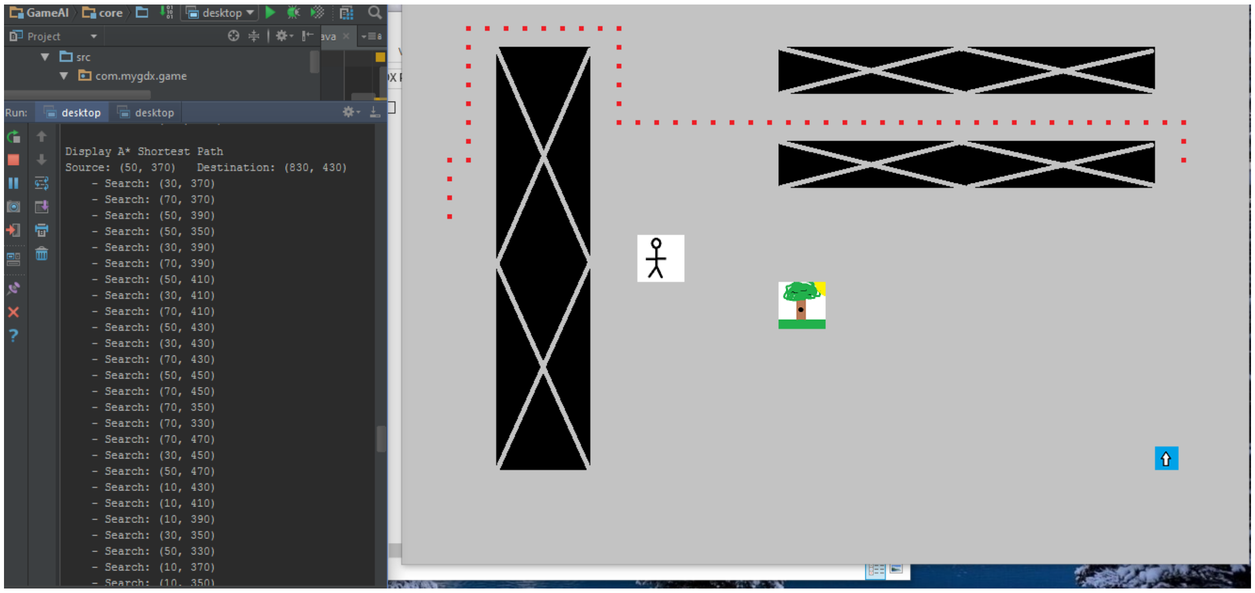
```
    - Search: (590, 210)
    - Search: (570, 330)
    - Search: (570, 210)
    - Search: (550, 330)
    - Search: (550, 210)
    - Search: (530, 330)
    - Search: (530, 210)
    - Search: (510, 330)
    - Search: (510, 210)
    - Search: (490, 330)
    - Search: (490, 210)
    - Search: (470, 330)
    - Search: (450, 230)
    - Search: (470, 210)
    - Search: (430, 230)
    - Search: (450, 210)
    - Search: (410, 230)
    - Search: (430, 210)
    - Search: (390, 230)
    - Search: (410, 210)
    - Search: (370, 230)
    - Search: (390, 210)
    - Search: (350, 230)
    - Search: (370, 250)
    - Search: (370, 210)
    - Search: (350, 250)
    - Search: (370, 270)
```

This is the console after the second click of the display A*
It will show the user the Source Node and Destination Node location
It also the location of each node that is searched in the A* algorithm
Note – this is not a full list (as the full list did not fit on the screen)



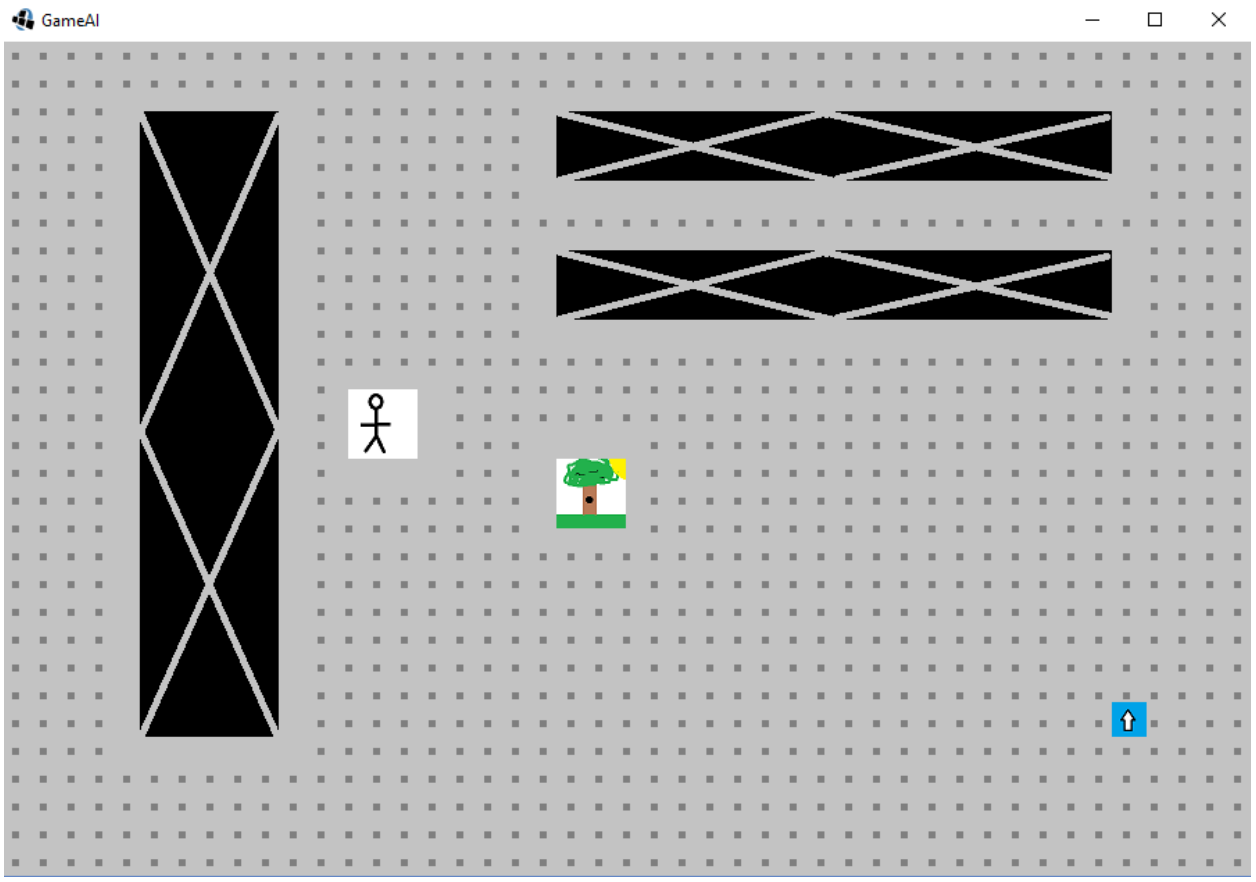
```
Display A* Shortest Path
Source: (750, 270)  Destination: (370, 270)
    - Search: (730, 270)
    - Search: (770, 270)
    - Search: (750, 290)
    - Search: (750, 250)
    - Search: (710, 270)
    - Search: (730, 290)
    - Search: (730, 250)
    - Search: (690, 270)
    - Search: (710, 290)
    - Search: (710, 250)
    - Search: (670, 270)
    - Search: (690, 290)
    - Search: (690, 250)
    - Search: (650, 270)
    - Search: (670, 290)
    - Search: (670, 250)
    - Search: (630, 270)
    - Search: (650, 290)
    - Search: (650, 250)
    - Search: (610, 270)
    - Search: (630, 290)
    - Search: (630, 250)
    - Search: (590, 270)
    - Search: (610, 290)
    - Search: (610, 250)
    - Search: (570, 270)
```

Here is another example of the display A*

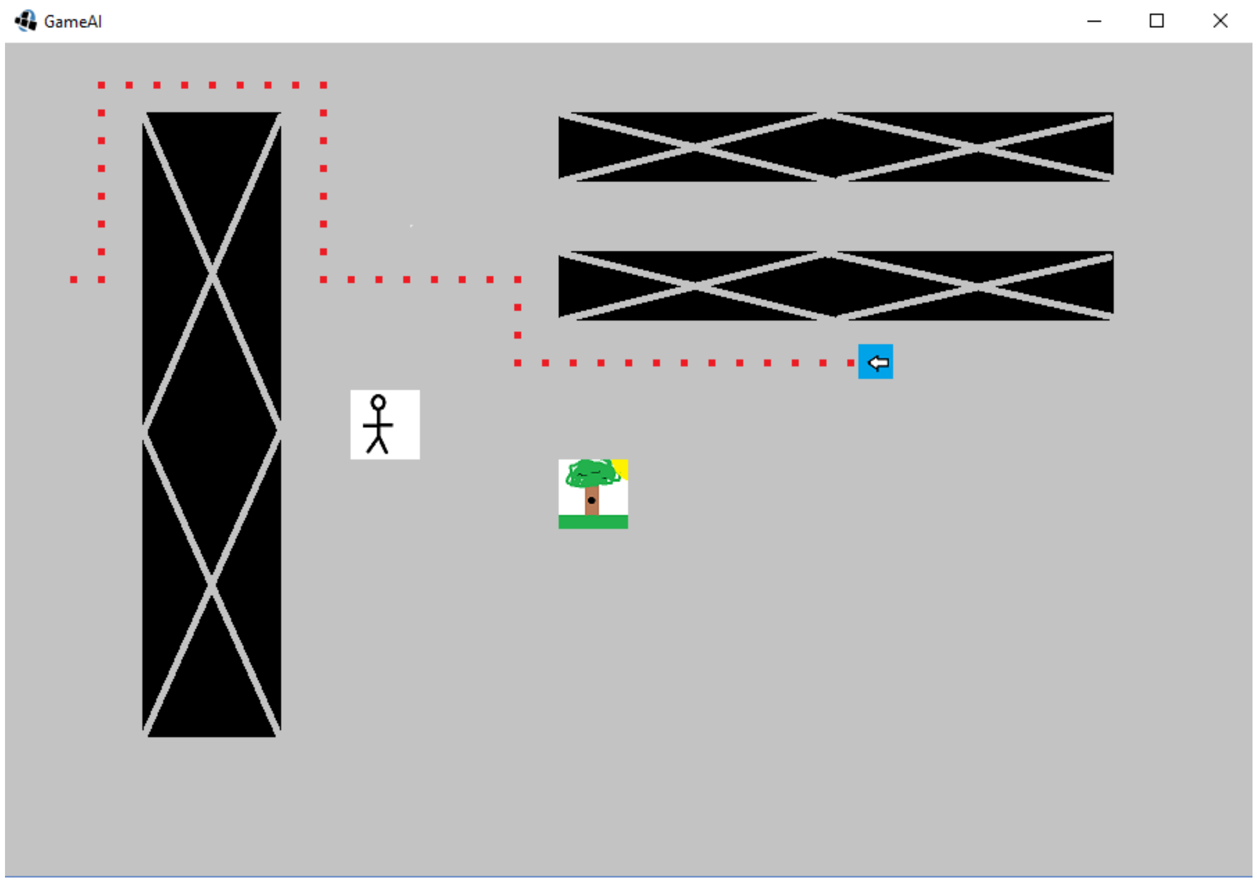


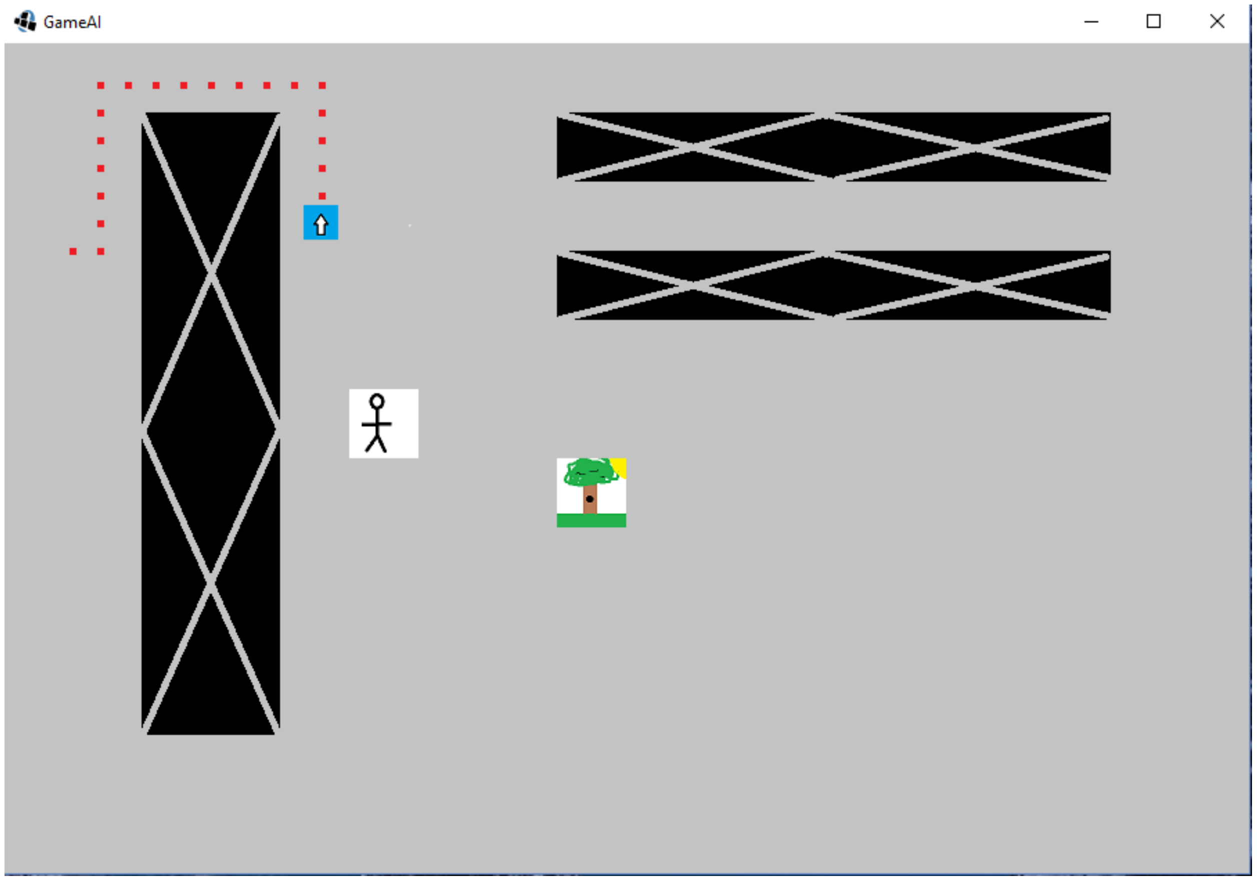
```
Display A* Shortest Path
Source: (50, 370)   Destination: (830, 430)
     - Search: (30, 370)
     - Search: (70, 370)
     - Search: (50, 390)
     - Search: (50, 350)
     - Search: (30, 390)
     - Search: (70, 390)
     - Search: (50, 410)
     - Search: (30, 410)
     - Search: (70, 410)
     - Search: (50, 430)
     - Search: (30, 430)
     - Search: (70, 430)
     - Search: (50, 450)
     - Search: (70, 450)
     - Search: (70, 350)
     - Search: (70, 330)
     - Search: (70, 470)
     - Search: (30, 450)
     - Search: (50, 470)
     - Search: (10, 430)
     - Search: (10, 410)
     - Search: (10, 390)
     - Search: (30, 350)
     - Search: (50, 330)
     - Search: (10, 370)
     - Search: (10, 350)
```

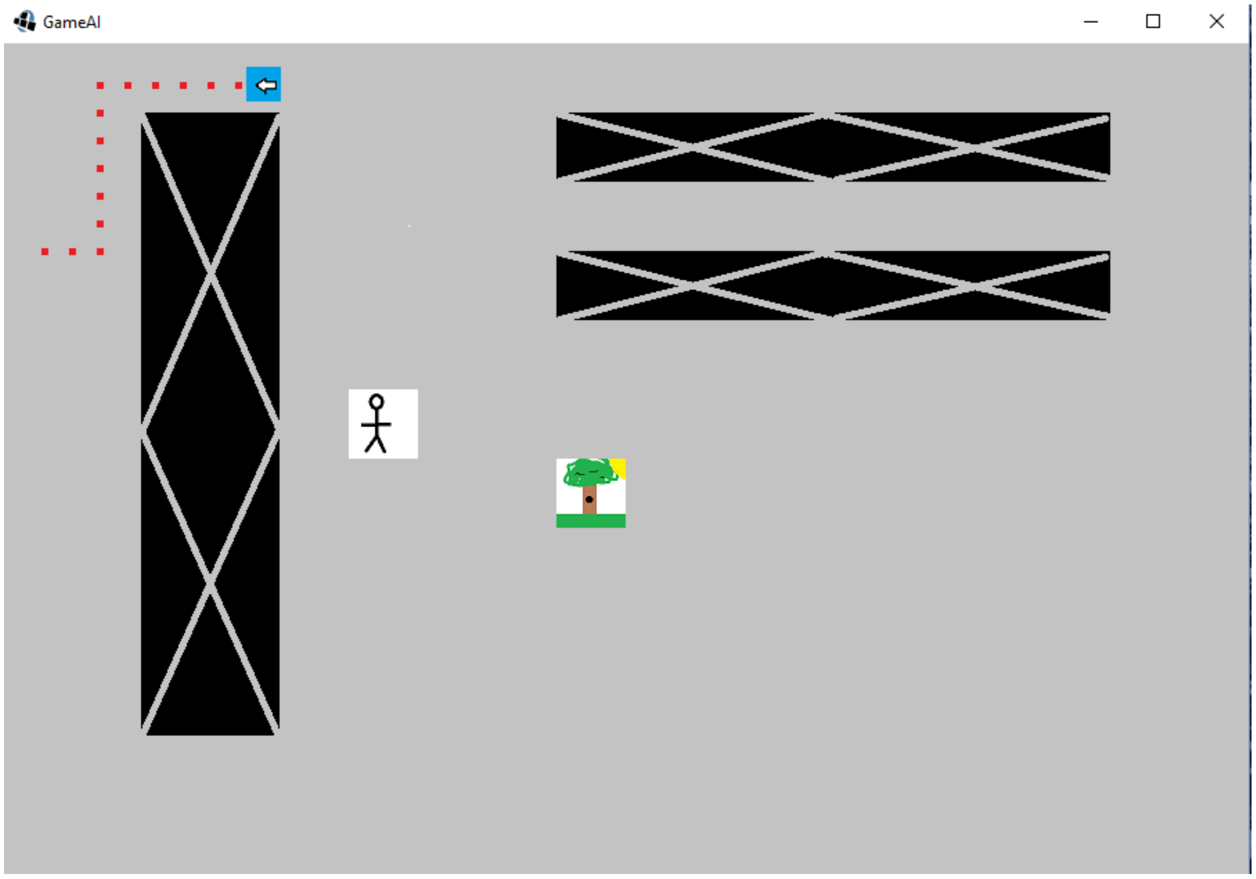
This is the entire navigation graph (displayed when the user clicks "n")



Here is the character navigating the A* shortest path






GameAI