Controls

**W,A,S,D** – Basic player agent movement if not seeking or following an A\* path

**F**  – Toggle player agent movement behavior to seek (default = do not seek)

**Space**  – Toggle showing the waypoint nodes (default = do not show)

**Tab** – Toggle showing the edges between neighboring waypoint nodes (default = do not show)  
 – Make sure Gizmos are enabled when playing (when playing, click the word Gizmos in the top-right of the Game tab to toggle)

**E** – Execute A\* pathfinding algorithm;  
 – Do not press again until nodes are reset (see **R**), and do not press again while player agent is navigating the path or Unity will crash  
 – Player agent will follow the calculated path if only the destination is specified; – Path is listed in the Debug console (nodes are numbered from 0-44, top-left to bottom-right, source and destination id’s are in parentheses)

**R** – Reset the waypoint node graph such that no node is the source, destination, or intermediate node;  
 – Must be pressed before starting a new A\* search or Unity will crash

**Left-click** – Finds the closest node to the mouse position and makes it the source node  
 – If closest node is already specified as the destination node, override

**Right-click** – Finds the closest node to the mouse position and makes it the destination node  
 – If closest node is already specified as the source node, override

Node Color Codes

**Gray** – Basic, default node

**Yellow** – Source node

**Green**  – Destination node

**White** – Intermediate waypoint node

Summary of Code

For this individual assignment, I wrote only three C# scripts. I would have broken some of them up into several, shorter scripts, but it was becoming a pain to call methods from another script. Also, some methods were just small helper functions that have simple uses.

NodeController.cs is the controller script for each node. It mostly manages the animations which determine which color it should be based on if it is the source, destination, or intermediate node. This script also reacts to when Space Bar is pressed by showing or hiding all nodes. Each node has its own unique id and list of ids of its neighbors, which I set up manually by making the array public and using Unity’s Inspector. Changing the layout of the map requires manually changing the neighbor list of each affected node.

PlayerAgentController.cs is a modified version of the same file name from Assignment1, and it is the controller that is attached to the player agent. Normal movement is determined by WASD controls, but these are disabled when the user switches to seek to mouse position (by pressing F) or when the player agent is navigating an A\* path. The method findClosestNode() acts as the path node sensor for this Assignment; it finds all nodes within some radius from the player agent and determines the id of the closest one. The method moveto() is used when the player agent must follow an A\* path, and it will turn the player towards some position (with a 10 degree margin) and then have it seek towards it. At the end of this method, the else clause updates the waypoint to go to, and if this was the last waypoint then the user gains control again with WASD. The method mySeek() is similar, but it is more basic in that the player must just move and turn towards some specified target position, which is used when in seek mode.

DebugListener.cs is where everything else is, and it is attached to the main camera. Firstly, this script contains a pathNode class, which contains the information needed about each checked node during the A\* search algorithm. The update() function simply checks for user input and updates the target waypoints the game is in the middle of pathfinding by moving the player agent. OnDrawGizmos() simply draws lines between all neighbor nodes, which if visible gives you the entire node-edge graph of the map. The method checkForInputs() is a bunch of if statements that check for valid user inputs, which are described on the first page of this report. findClosestNodeToMouse() looks for the closest node from a given position, which is assumed to be the mouse position. The method findIndexOf() takes in a unique node id and gives its index in the nodesList, which is just an array of GameObjects that are nodes. This is because the id of each node is different from the index in this array, which consequentially created some complicated statements (such as line 67). The method aStar() is where the A\* pathfinding search algorithm is performed; starting from some source node (ns), search along neighbors of nodes until the destination node (nd) is found. The queue of nodes is sorted by lowest cost before the next node is selected, and nodes are only checked once. Once the destination is found, the breadcrumbs represent the shortest path of ids from the source to the destination. I know Dijkstra’s algorithm only enqueues the lowest-cost neighbor, but I had a feeling that doing that would lead to dead-ends at local minima, so I enqueued all checked neighbors. The method sortByCost() takes in an ArrayList, which is assumed to be the queue for the A\* search algorithm, and sorts its pathNodes by lowest cost (specifically, using Selection Sort).

Lessons Learned

Half of my group initially wanted to do artificial neural network (ANN) for this Assignment, but because of my bad experience trying to code the Perceptron learning algorithm in CAP 4630: Artificial Intelligence (the class average was ~60% before the re-grade!), I decided not to do it. The other group member also mentioned that A\* is heavily used in video games today, and since I am pursuing game programming as my career, I would rather learn it now than code up another learning algorithm.

Most of the work I did on this Assignment was over Spring Break since I had no other pressing homeworks from other classes. This was the first time that I got Unity to crash, first with the Animator’s finite state machine, and then again with infinite while loops (which don’t display any Debug.Log() statements inside them, by the way). The A\* search algorithm itself was not much trouble, but overall this Assignment took much longer than I expected.

One regrets I had about the way I did this assignment was that I made each node’s neighbor list non-dynamic; I had to enter in each of the 45 nodes’ neighbor ids manually, and each time I added a wall I had to change the nearby nodes’ neighbor lists. If this way didn’t work, I would have done a raycast around each node and dynamically added each node the rays hit into its neighbor lists. Furthermore, I did not want to make the map much bigger, because that would require more tedious work of creating the nodes and their neighbors manually.

Another minor annoyance is that when the player agent is automatically following a path in a straight line, it tends to wobble a bit. This can be fixed by lowering the initial value for turnSpeed in Start() of PlayerAgentController.cs. However, doing so makes the turning speed slower, which in turn means it takes more time to travel a path automatically.