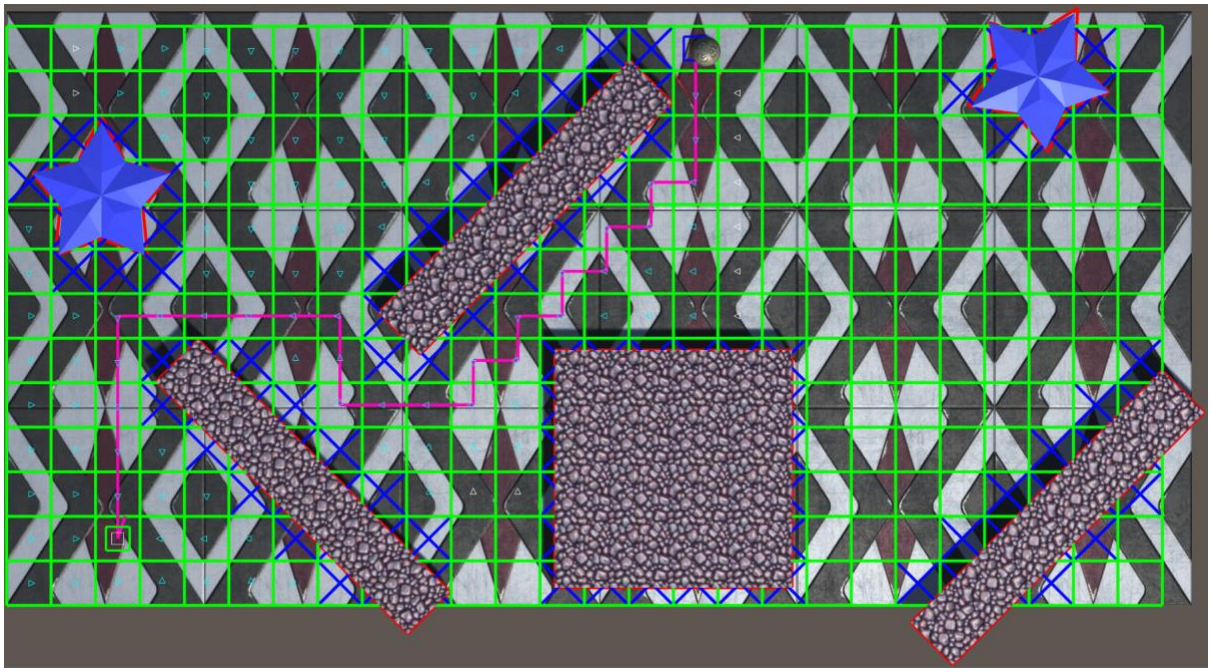
Homework 3: A* Pathfinding

One of the main uses of artificial intelligence in games is to perform *path planning*, the search for a sequence of movements through the virtual environment that gets an agent from one location to another without running into any obstacles.

One of the most commonly used pathfinding algorithms for computer games is A*. With an admissible heuristic, A* provides nice guarantees about optimality—it will find the shortest path—and keep the amount of unnecessary search to a minimum.

In lecture, we have seen the pseudocode for the A* algorithm. However, in video games it is often desirable to break up the computation of A* so as not to impact game performance (e.g. maintain frame rate with minimal impact). Threads or multiprocessing may be utilized in some approaches, however in this assignment you will be implementing the A* algorithm with a modification to support incremental progress on the path planning task. This allows the algorithm to be spread across many frames in the main process/thread. Additionally, it's often useful for a path search algorithm to return the closest node to the goal if the goal cannot be reached. A* will also be modified to do so in your implementation.

To find the closest node to goal, use A*'s estimated cost to goal node found in the search node record metadata of closed set nodes. If more than one node is tied for closest, select the one with the smallest cost from the start node up to that point. If A* is used with a null heuristic (e.g. Dijkstra) then nodes will have an estimated cost to goal of zero. In that case, fall back to a manually computed Euclidean distance.



What you need to know

Please consult earlier assignments for background on the Unity project. In addition to the information about the game engine provided there, the following are new elements you need to know about.

AStarPathSearchImpl

This class will provide the implementation for incremental path planning using the A* algorithm. Other algorithms in the Unity project depend on this implementation. Once you have successfully implemented the A* algorithm, GreedyBestFirstSearch and DijkstrasSearch will also become functional. They work by providing alternative G()/H() functions.

float CostNull(Vector2 nodeA, Vector2 nodeB)
This function provides the null cost (weighted graph edge weight) between two pathNodes. It is already implemented for you and should not be modified. Used for Greedy Best First Search.

float HeuristicNull(Vector2 nodeA, Vector2 nodeB)
This function provides the default null heuristic estimate of the remaining distance between two pathNodes. It is already implemented for you and should not be modified. Used for Dijkstra's Algorithm.

float Cost(Vector2 nodeA, Vector2 nodeB)
This function provides the default cost (weighted graph edge weight) between two pathNodes. You must implement this function. Return must be non-negative. Measure the Euclidean distance between nodes.

float HeuristicManhattan(Vector2 nodeA, Vector2 nodeB)
This function provides the heuristic estimate of the remaining distance between two pathNodes using Manhattan distance. You must implement this function. Return must be non-negative.

float HeuristicEuclidean(Vector2 nodeA, Vector2 nodeB)
This function provides the heuristic estimate of the remaining distance between two pathNodes using Euclidean distance. You must implement this function. Return must be non-negative.

public static PathSearchResultType FindPathIncremental()
This method calculates an incremental update of the A* algorithm. It may take several calls before the goal is found (if possible). You must implement this function.

Arguments:
GetNodeCount getNodeCount – A callback that returns the integer number of graph nodes.
E.g. int count = getNodeCount().

GetNode getNode – A callback that returns the node at position *i*. E.g. Vector2 v = getNode(i)

GetNodeAdjacencies getAdjacencies – A callback that returns the adjacency list for node at position *i*. E.g. List<int> adj = getAdjacencies(i)

CostCallback G – Cost function that return a non-negative float, taking two Vector2 arguments. If you aren't familiar with C#, G can be called like any other function. E.g., G(nodeA, nodeB)

CostCallback H – Heuristic estimate function that returns a non-negative float, taking two Vector2 arguments. If you aren't familiar with C#, H can be called line any other function. E.g., H(nodeA, nodeB)

int startNodeIndex – index of the PathNode from which the search begins

int goalNodeIndex – index of the PathNode from which the search ends

int maxNumNodesToExplore – the maximum number of nodes to visit during a call to FindPathIncremental(). The method may return early if openNodes are exhausted or the goal is found. One node explored equates to one run of the outer loop from the A* implementation described in lecture. When a node is pulled from the open set and you process its edges that counts as one node processed.

bool doInitialization – true if the method should be initialized as starting a new search from the startNodeIndex, with new currentNodeIndex, searchNodeRecords, openNodes, closedNodes, and returnPath. Future incremental calls will be made with false. Note that initialization can be requested on any call

ref int currentNodeIndex – The current node the A* algorithm is working from. Note that this is only used for visualization purposes in the A* search (especially during incremental search). The current node is determined by the open set priority queue. However other search algorithms might use this as an in/out param to track state.

ref Dictionary<int, PathSearchNodeRecord> searchNodeRecords – a dictionary that associates a PathNode index with metadata pertaining to the PathNode in question. A Dictionary is built into the .NET/mono framework and provides quick mapping between key/value pairs.

PathSearchNodeRecord (struct)

The metadata necessary to facilitate A* search (and other search types). See: /Assets/Scripts/Framework/PathSearch/PathSearchNodeRecord.cs

Struct Members:

NodeIndex – the PathNode index of the node that the PathSearchNodeRecord pertains to

FromNodeIndex – the PathNode index that lead to the NodeIndex being added to the openNodes set.

CostSoFar – The sum of edge weights (G function) that lead to this node

EstimatedTotalCost – The CostSoFar plus the Heuristic estimate to the goal from this node

Constructor() – Note that there are constructors for instantiating this struct

ref SimplePriorityQueue<int, float> openNodes – A priority queue associating an index to a PathNode (nodes list) to an assigned priority. These nodes are considered to be open to further consideration in the A* algorithm. Refer to lecture materials for the class. The EstimatedTotalCost is appropriate for the priority. The implementation (documented code and license) for SimplePriorityQueue is provided in Assets/3rdParty/FastPriorityQueue/

ref HashSet<int> closedNodes – A HashSet of PathNode indexes. These nodes are considered to be (mostly) closed to further consideration in the A* algorithm. A HashSet is built into the .NET/mono framework and provides quick determination of membership in the set.

ref List<int> returnPath – A list of indexes of PathNodes. Upon completion of the A* algorithm. This list will be a path from the startNode to the goalNode. If the goalNode was impossible to reach, the returnPath will lead from the startNode to the node closest to the goalNode. Note that this requires modification to the A* algorithm. If return value is successful and has run to completion (returning Complete, Partial) then the returnPath should not be null and contain the path from startNode to goalNode and include those two nodes. If the path is partial, then the final node is instead the node-closest-to-goal. If the return value is InitializationError or InProgress, then returnPath is considered undefined (it can be anything).

Return value:

PathSearchResultType – This is an enumeration that details the status of the AStar Incremental Search. The enumeration values are as follows.

Complete – The A* algorithm has completed calculation and found the goal.
Partial – The A* algorithm has completed calculation and only found a partial path.
InProgress – The A* algorithm has not yet found a solution. Further calls must be made to complete the search.
InitializationError – The A* algorithm has been called with invalid arguments. This can be returned even if *doInitialization* is not set to true.

Instructions

To complete this assignment, you must implement the A* algorithm.

Download this project from github. Open the project in Unity. Utilize your previous grid and path network solutions, and open either scene (do eventually test both).

If you don't have a valid solution, then you can use the hard coded solutions by enabling "Use Hard Coded Cases" in the Unity Inspector when viewing the NavigationArea game object of both the GridNavigation and PathNetwork scenes. The option will be in the Game Grid or Path Network component sections.

Try left-clicking somewhere on the map that is not an obstacle or waypoint. You should see the agent find a path using BreadthFirstSearch. Hit spacebar to cycle through search algorithms (text in top right of HUD). Note that A*, GreedyBestFirst, and Dijkstra won't work until you implement A*. Try right-clicking to initiate an incremental search. Observe the metadata visualization that draws little arrows showing open/closed sets and FromNode direction (best observed in the grid scene). You can use Shift Right-Click to manually step with "N" for next step. You can clear the metadata visualization with "C".

In the Unity Inspector view of the AgentSphere, you can change IncrSearchMaxNode to different values for testing step sizes other than 1.

Once you are familiar with the interface, implement your A* algorithm in Assets/Scripts/GameAIStudentWork/AStar/AstarPathSearchImpl.cs

Make sure you change the name string and also implement the three methods. Keep in mind that A* must:

1.) Solve incrementally
2.) Return a path to the closest node to the goal, if the goal cannot be reached
3.) Work correctly when the caller passes different G() and H() functions. This is demonstrated with Dijkstra and GreedyBestFirstSearch methods, which are implemented as variants of the base AStar algorithm.

Grading

Assessment of this assignment will be based on determining whether your algorithm can successfully find the goal, if reachable, can do so incrementally, only search nodes as necessary for A*, etc.

Special considerations:

- You are expected to implement the Millington version of A* search (pseudocode shown in lecture)
    - This especially applies to testing if a node is the goal when pulling from the open set and not when first encountered.
- You must not skip putting the goal node in the searchNodeRecords with correct meta data. If the goal node is not reachable, then instead the node closest to goal must be in the searchNodeRecords with correct meta data. This is for grading purposes.
- You must always include the start node, the goal node (or node closest to goal), and all the nodes in-between and in order, in your returned path

Please remove all print statements before submitting. The autograder will only provide a few hundred lines of feedback and you might overflow the buffer so that the informative part doesn't show up. Also, print statements can cause significant slowdown such that you might fail a test due to timeout (see below). When you remove print statements, please test your

code again! Quite often we receive assignments that don't compile due to a hanging *if-statement* where a print() was the consequent.

Your code will be allowed at least 10 seconds to complete each test.

Hints

See the lecture notes for a corrected version of the AStar pseudocode from the Millington book.

Start with the basic A* and ignore *maxNumNodesToExplore.* Just return *PathSearchResultType.Complete/Partial* instead of *InProgress*.

Similarly, you can ignore finding the closest node to goal (if goal cannot be reached) until you get the basics working.

Don't forget to later test with AgentSphere's IncrSearchMaxNode Inspector property.

Check out BasicPathSearchImpl.cs and GreedySimplePathSearchImpl.cs for other working search algorithm code.

Make new entries in CustomPresetConfig.cs for further testing (do not submit this file).

Create Unit/Integration tests with AStarTest.cs. It's highly recommended that you create several tests! Passing the provided example test(s) in this file does NOT indicate that you have a correct solution!

Submission

To submit your solution, upload your modified AStarPathSearchImpl.cs **with updated name string**

You should not modify any other files in the game engine. (You can modify CustomPresetConfig.cs but don't turn that in.)
DO NOT upload the entire game engine.

See Canvas assignment description for specific submission instructions (most like GradeScope).