Lab 10 - Path Finding

*Continuing with AStar algorithm - GridBased implementation**
COMP396-Game Programming 2

Purpose: To **study** and **implement** Path Finding via:

- ~~BFS Algorithm (Breadth First Search)~~ — DONE
- ~~DFS Algorithm (Depth First Search)~~ — DONE
- ~~Dijstra Algorithm~~ — DONE
- ~~A* Algorithm (not grid-based)~~ — DONE
- A* Algorithm (grid-based)

Contents

1 Introduction
2 Implementing the A* Algorithm for PathFinding - Grid-Based
2.1 Node
2.2 PriorityQueue
2.3 The GridManager class
2.4 The AStar class
3 Setting up the scene
3.1 Testing the pathfinder
4 Summary
4.1 Tasks:

1. Introduction

In Labs 8 and 9 we implemented the following:

- A Graph data structure. We needed to select one of the ways to represent the graphs:
  - Set (List) of Vertices and Set (List) of Edges.
    - Vertices can be represented via:
      - `int`s,
      - `char`s (for very small graphs) **we used this**
      - `string`s,
      - a (templatized) `structure`,
      - a (templatized) `class`,
      - ...
    - Edges can be represented via:
      - pairs of vertices,
      - pairs of `id`s of vertices,
      - Adjacency Matrix
      - Linked List of `id`s of vertices neighbour to a vertex
      - a (templatized) `structure`,
      - a (templatized) `class`,
      - ...
  - An implied graph structure - a **grid** (**today we'll use this method**). We'll have a **GridManager** class (subsuming the role of the graph) with:
    - an implied node set (the grid), even though when we'll need a concreate node we'll use a Node class
    - an implied edge set (for each node, the edges are to *neighbor* either 4 nodes (N,E,S,W) or 8 nodes (N, NE, E, SE, S, SW, W, NW).
- Dictionary with KeyType the NodeType and ValueType the type of the collection of a=edges that are adjacent to the correspondin vertex (**we used this method**)

> The above is enough to represent an abstract graph, and therefor to be able to implement algorithms such as **BFS** and **DFS**. However **weighted graph algorithms** (such as **Dijkstra's** and/or **ASTar** need some extra information per edge: **weights** (or lengths/costs/times etc). Additionally, some algorithms (like A*), may require us to “pay” more with some more information, with the promise to give us in “return” better efficiency (A* reqires us to “pay” by giving the **heuristic function**, that is an estimate of the distance from each node to the target.)

- A few simple algorithms in graphs

> The complexity (or lack thereof) of these algorithms depends of the graph representation we have chosen.

- find all descendants of a given vertex
- find all parents (antescendants) of a given vertex
- find all neighbors (descendants and parents) of a given vertex
- find a spanning tree (Prim's algorithm)
- find an Hamilton Circuit (The TSP - The Traveling Salesman Problem)
- find a path from a vertex to another vertex (BFS, DFS, Dijkstra, A*)
- find a shortest path from a vertex to another vertex (Dijkstra, A*)
- find a shortest path from a vertex to every other vertex (Dijkstra)
- find a shortest path from every vertex to every other vertex (Dijkstra, A* running multiple times)

- Implemented these algorithms:

- BFS Algorithm (Breadth First Search)
- DFS Algorithm (Depth First Search)
- Dijstra Algorithm
- A* Algorithm - not grid-based

We will deal hereby:

- implementation of the A* Algorithm - (Grid Based).

2. Implementing the A* Algorithm for PathFinding - Grid-Based

- Open the last project
- Create a folder under Scenes folder named **Lab10_PathFinding_WithGrid_{YourInitials}**

First, we implement the basic classes that we introduced before, such as the Node class, the GridManager class, and the PriorityQueue class. Then, we use them in the main AStar class.

2.1. Node

The Node class represents each tile object in the 2D grid. Its code is shown in the Node.cs file:

```
using UnityEngine;
using System;
public class Node {
  public float costSoFar;
  public float fScore;
  public bool isObstacle;
  public Node parent;
  public Vector3 position;
  public Node(Vector3 pos) {
    fScore = 0.0f;
    costSoFar = 0.0f;
    isObstacle = false;
    parent = null;
    position = pos;
  }
  public void MarkAsObstacle() {
    isObstacle = true;
  }
```

> The Node class stores every valuable property we need for finding a path. We are talking about properties such as the cost from the starting point (costSoFar), the total estimated cost from start to end (fScore), a flag to mark whether it is an obstacle, its positions, and its parent node. costSoFar is G, which is the movement cost value from the starting node to this node so far, and fScore is obviously F, which is the total estimated cost from the start to the target node. We also have two simple constructor methods and a wrapper method to set, depending on whether this node is an obstacle or not.

Let's implement the **Equals** and **GetHashCode** methods, as shown in the following code:

```
  public override bool Equals(object obj) {
    return obj is Node node && position.Equals(node.position);
  }
  public override int GetHashCode() {
    return HashCode.Combine(position);
  }
}
```

These methods are important. In fact, even if the Node class has multiple attributes, two nodes that represent the same position should be considered equal as far as the search algorithm is concerned. The way to do that is to override the default Equals and GetHashCode methods, as in the preceding example.

2.2. PriorityQueue

A priority queue is an ordered data structure designed so that the first element (the head) of the list is always the smallest or largest element (depending on the implementation). This data structure is the most efficient way to handle the nodes in the open list because, as we will see later, we need to quickly retrieve the node with the lowest F value (in general, the highest priority value — it just happens with priority quest the highest priority is represented with the smallest values, so you can say the highest priority is the lowest cost/path length/time etc). Unfortunately, there is no easy out-of-the-box way to have a suitable priority queue (for earlier than .NET 6 versions) so we'll use for this the code shown in the following NodePriorityQueue.cs class (which still relies in `sorting after modification' method we have used before):

```
using System.Collections.Generic;
using System.Linq;
public class NodePriorityQueue {
  private readonly List< Node> nodes = new();
  public int Length {
    get { return nodes.Count; }
  }
  public bool Contains(Node node) {
    return nodes.Contains(node);
  }
  public Node Dequeue() {
    if (nodes.Count > 0) {
      var result = nodes[0];
      nodes.RemoveAt(0);
      return result;
    }
    return null;
  }
  public void Enqueue(Node node) {
    if (nodes.Contains(node)) {
      var oldNode = nodes.First(n => n.Equals(node));
      if (oldNode.fScore ≤ node.fScore) {
        return;
      } else {
        nodes.Remove(oldNode);
      }
    }
    nodes.Add(node);
    nodes.Sort((n1, n2) => n1.fScore < n2.fScore ? -1 : 1); // here is sorting
  }
}
```

This implementation is not particularly efficient because it relies on the Sort method to reorder the internal list of nodes after each insertion. This means that inserting a node becomes increasingly costly the more nodes we have in the queue. If you need better performance, you can find many priority queue implementations designed for A* and search algorithms. For now, though, our small **NodePriorityQueue** class will do its job nicely. The class is self-explanatory. The only thing you need to pay attention to is the **Enqueue** method. Before adding a new node, we need to check whether there is already a node with the same position but a lower **F-score**. If there is, we do nothing (we already have a better node in the queue). If not, this means that the new node we are adding is better than the old one. Therefore, we can remove the old one to ensure that we only have the best node possible for each position. Can you figure out what portion of the A* algorithm we've already seen, is this supposed to supplant?

2.3. The GridManager class

The **GridManager** class handles the 2D grid representation for the world map. We keep it as a **singleton** instance of the GridManager class, as we only need one object to represent the map. Revcall that a singleton is a programming pattern that restricts the instantiation of a class to one object and, therefore, it makes the instance easily accessible from any point of the application. The code for setting up GridManager is shown in the **GridManager.cs** file. 1. The first part of the class implements the singleton pattern. We look for the GridManager object in the scene and, if we find it, we store it in the staticInstance static variable:

```
using UnityEngine;
using System.Collections.Generic;
public class GridManager : MonoBehaviour {
  private static GridManager staticInstance = null;
  public static GridManager instance {
    get {
      if (staticInstance == null) {
        staticInstance = FindObjectOfType(typeof(GridManager)) as GridManager;
        if (staticInstance == null)
          Debug.Log("Could not locate an GridManager object. \n
                    You have to have exactly one GridManager in the scene.");
      }
      return staticInstance;
    }
  }
  // Ensure that the instance is destroyed when the game is stopped in the editor.
  void OnApplicationQuit() {
    staticInstance = null;
  }
```

2. Then, we declare all the variables that we need to represent our map. **numOfRows** and **numOfColumns** store the number of rows and columns of the grid. **gridCellSize** represents the size of each grid. **obstacleEpsilon** is the margin for the system we will use to detect obstacles (more on that later).

3. Then we have two **Boolean variables** to enable or disable the debug visualization of the grid and obstacles. Finally, we have a grid of nodes representing the map itself. We also add two properties to get the grid's origin in world coordinates (**Origin**) and the cost of moving from one tile to the other (**StepCost**). The final product is shown in the following code:

```
public int numOfRows;
public int numOfColumns;
public float gridCellSize;
public float obstacleEpsilon = 0.2f;
public bool showGrid = true;
public bool showObstacleBlocks = true;
public Node[,] nodes { get; set; }
public Vector3 Origin {
  get { return transform.position; }
}
public float StepCost {
  get { return gridCellSize; }
}
```

4. Now we need to build the grid. For this, we use the **ComputeGrid** method that we call on **Awake**. The code is shown here:

```
void Awake() {
  ComputeGrid();
}
void ComputeGrid() {
  //Initialise the nodes
  nodes = new Node[numOfColumns, numOfRows];
  for (int i = 0; i < numOfColumns; i++) {
    for (int j = 0; j < numOfRows; j++) {
      Vector3 cellPos = GetGridCellCenter(i,j);
      Node node = new(cellPos);
      var collisions = Physics.OverlapSphere(cellPos, gridCellSize / 2 - obstacleEpsilon,
       1 << LayerMask.NameToLayer("Obstacles"));
      if (collisions.Length != 0) {
        node.MarkAsObstacle();
      }
      nodes[i, j] = node;
    }
  }
}
```

5. The ComputeGrid function follows a simple algorithm. First, we just initialize the nodes grid. Then we start iterating over each square of the grid (represented by the coordinates i and j). For each square, we do as follows:
    1. First, we create a new **node** positioned at the center of the square (in world coordinates).
    2. Then, we check whether that square is occupied by an **obstacle**. We do this by using the **OverlapSphere** function. This Physics function returns all the colliders inside or intersecting the sphere defined in the parameters. In our case, we center the sphere at the center of the grid's cell (**cellPos**) and we define the sphere's **radius** as a bit less than the grid cell size. Note that we are only interested in colliders in the **Obstacles** layer, therefore we need to add the appropriate **layer mask**.
    3. If the **OverlapSphere** function returns anything, this means that we have an **obstacle** inside the cell and, therefore, we define the entire cell as an obstacle.

GridManager also has several helper methods to **traverse** the grid and **get** the grid cell data. We show some of them in the following list, with a brief description of what they do. The implementation is simple:

1. The **GetGridCellCenter** method returns the position of the grid cell in world coordinates from the cell coordinates, as shown in the following code:

```
public Vector3 GetGridCellCenter(int col, int row)
{
    Vector3 cellPosition = GetGridCellPosition(col, row);
    cellPosition.x += gridCellSize / 2.0f;
    cellPosition.z += gridCellSize / 2.0f;
    return cellPosition;
}
public Vector3 GetGridCellPosition(int col, int row) {
    float xPosInGrid = col * gridCellSize;
    float zPosInGrid = row * gridCellSize;
    return Origin + new Vector3(xPosInGrid, 0.0f, zPosInGrid);
}
```

2. The **GetGridCoordinates** method get the grid cell indices in the Astar grids with the position given:

```
public (int,int) GetGridCoordinates(Vector3 pos) {
    if (!IsInBounds(pos)) {
```

```
        return (-1,-1);
    }

    int col = (int)Mathf.Floor((pos.x-Origin.x) / gridCellSize);
    int row = (int)Mathf.Floor((pos.z-Origin.z) / gridCellSize);

    return (col, row);
}
```

3. The **IsInBounds** method checks whether a certain position in the game falls inside the grid:

```
public bool IsInBounds(Vector3 pos) {
  float width = numOfColumns * gridCellSize;
  float height = numOfRows * gridCellSize;
  return (pos.x >= Origin.x && pos.x ≤ Origin.x + width
          && pos.z ≤ Origin.z + height && pos.z >= Origin.z);
}
```

4. The **IsTraversable** method checks whether a grid coordinate is traversable (that is, it is not an obstacle):

```
public bool IsTraversable(int col, int row) {
    return col >= 0 && row >= 0 && col < numOfColumns && row < numOfRows && !nodes[col, row].isObstacle;
}
```

5. Another important method is **GetNeighbours**, which is used by the AStar class to retrieve the neighboring nodes of a particular node. This is done by obtaining the grid coordinate of the node and then checking whether the four neighbors' coordinates (up, down, left, and right) are traversable:

```
public List< Node> GetNeighbours(Node node) {
    List< Node> result = new();
    var (column, row) = GetGridCoordinates(node.position);
    if (IsTraversable(column - 1, row)) {
        result.Add(nodes[column - 1, row]);
    }
    if (IsTraversable(column + 1, row)) {
        result.Add(nodes[column + 1, row]);
    }
    if (IsTraversable(column, row - 1)) {
        result.Add(nodes[column, row - 1]);
    }
    if (IsTraversable(column, row + 1)) {
        result.Add(nodes[column, row + 1]);
    }
    return result;
}
```

6. Finally, we have **debug aid** methods used to visualize the grid and obstacle blocks:

```
void OnDrawGizmos() {
  if (showGrid) {
    DebugDrawGrid(Color.blue);
  }
  //Grid Start Position
  Gizmos.DrawSphere(Origin, 0.5f);
  if (nodes == null) return;
  //Draw Obstacle obstruction
  if (showObstacleBlocks) {
    Vector3 cellSize = new Vector3(gridCellSize, 1.0f, gridCellSize);
    Gizmos.color = Color.red;
    for (int i = 0; i < numOfColumns; i++) {
      for (int j = 0; j < numOfRows; j++) {
        if (nodes != null && nodes[i, j].isObstacle) {
          Gizmos.DrawCube(GetGridCellCenter(i,j), cellSize);
        }
      }
    }
  }
}
public void DebugDrawGrid(Color color) {
  float width = (numOfColumns * gridCellSize);
  float height = (numOfRows * gridCellSize);
  // Draw the horizontal grid lines
  for (int i = 0; i < numOfRows + 1; i++) {
    Vector3 startPos = Origin + i * gridCellSize * new Vector3(0.0f, 0.0f, 1.0f);
    Vector3 endPos = startPos + width * new Vector3(1.0f, 0.0f, 0.0f);
    Debug.DrawLine(startPos, endPos, color);
  }
```

```
    // Draw the vertial grid lines
    for (int i = 0; i < numOfColumns + 1; i++) {
      Vector3 startPos = Origin + i * gridCellSize * new Vector3(1.0f, 0.0f, 0.0f);
      Vector3 endPos = startPos + height * new Vector3(0.0f, 0.0f, 1.0f);
      Debug.DrawLine(startPos, endPos, color);
    }
  }
```

Gizmos can be used to draw visual debugging and setup aids inside the editor scene view. OnDrawGizmos is called every frame by the engine. So, if the debug flags, showGrid and showObstacleBlocks, are checked, we just draw the grid with lines and the obstacle cube objects with cubes. We won't go through the DebugDrawGrid method, as it's pretty simple.

INFO

You can learn more about gizmos in the following Unity3D reference documentation: https://docs.unity3d.com/ScriptReference/Gizmos.html.

2.4. The AStar class

The AStar class implements the pathfinding algorithm using the classes we have implemented so far. If you want a quick review of the A* algorithm, see the previous lab. The steps for the implementation of AStar are as follows:

1. We start by implementing a method called **HeuristicEstimateCost** to calculate the cost between the two nodes. The calculation is simple. We just find the direction vector between the two by subtracting one position vector from another. The magnitude of this resultant vector gives the straight-line distance from the current node to the target node:

```
using UnityEngine;
using System.Collections.Generic;
public class AStar {
  private float HeuristicEstimateCost(Node curNode, Node goalNode) {
    return (curNode.position – goalNode.position).magnitude;
  }
```

In theory, you can replace this function with any function, returning the distance between curNode and goalNode. However, for A* to return the shortest possible path, this function must be **admissible**. In short, an admissible heuristic function is a function that never overestimates the actual “real world” cost between **curNode** and **goalNode**. As an exercise, you can easily verify that the function we use in this demo is admissible. For more information on the math behind heuristic functions, you can visit https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

2. Then, we have the main A* algorithm in the **FindPath** method. In the following snippet, we initialize the open and closed lists. Starting with the start node, we put it in our open list. Then, we start processing our open list:

```
  public List< Node> FindPath(Node start, Node goal) {
    //Start Finding the path
    NodePriorityQueue openList = new NodePriorityQueue();
    openList.Enqueue(start);
    start.costSoFar = 0.0f;
    start.fScore = HeuristicEstimateCost(start, goal);
    HashSet<node> closedList = new();
    Node node = null;
```

3. Then, we proceed with the main algorithm loop:

```
    while (openList.Length != 0) {
      node = openList.Dequeue();
      if (node.position == goal.position) {
        return CalculatePath(node);
      }
      var neighbours = GridManager.instance.GetNeighbours(node);
      foreach (Node neighbourNode in neighbours)
      {
        if (!closedList.Contains(neighbourNode)) {
          float totalCost = node.costSoFar + GridManager.instance.StepCost;
          float heuristicValue = HeuristicEstimateCost(neighbourNode, goal);
          //Assign neighbour node properties
          neighbourNode.costSoFar = totalCost;
          neighbourNode.parent = node;
          neighbourNode.fScore = totalCost + heuristicValue;
          //Add the neighbour node to the queue
          if (!closedList.Contains(neighbourNode)) {
```

```
                openList.Enqueue(neighbourNode);
            }

        }
    }
    closedList.Add(node);
  }
```

1. The preceding code implementation strictly follows the algorithm that we have discussed previously, so you can refer back to it if something is not clear:
2. Get the first node from our openList. Remember, openList is always sorted in increasing order. Therefore, the first node is always the node with the lowest F value.
3. Check whether the current node is already at the target node. If so, exit the while loop and build the path array.
4. Create an array list to store the neighboring nodes of the current node being processed. Then, use the GetNeighbours method to retrieve the neighbors from the grid.
5. For every node in the array of neighbors, we check whether it's already in closedList. If not, we calculate the cost values, update the node properties with the new cost values and the parent node data, and put it in openList.
6. Push the current node to closedList and remove it from openList.
7. Go back to step I.
8. If there are no more nodes in openList, the current node should be at the target node if there's a valid path available:

```
    //If finished looping and cannot find the goal then return null
    if (node.position != goal.position) {
      Debug.LogError("Goal Not Found");
      return null;
    }
    //Calculate the path based on the final node
    return CalculatePath(node);
```

5. Finally, we call the CalculatePath method with the current node parameter:

```
  private List< Node> CalculatePath(Node node) {
    List< Node> list = new();
    while (node != null) {
      list.Add(node);
      node = node.parent;
    }
    list.Reverse();
    return list;
  }
}
```

6. The CalculatePath method traces through each node's parent node object and builds an array list. Since we want a path array from the start node to the target node, we just call the Reverse method.

Now, we'll write a test script to test this and set up a demo scene. The TestCode class The TestCode class uses the AStar class to find the path from the start node to the target node, as shown in the following code from the TestCode.cs file:

```
using UnityEngine;
using System.Collections;
public class TestCode : MonoBehaviour {
  private Transform startPos, endPos;
  public Node startNode { get; set; }
  public Node goalNode { get; set; }
  public List< Node> pathArray;
  GameObject objStartCube, objEndCube;
  private float elapsedTime = 0.0f;
  //Interval time between pathfinding
  public float intervalTime = 1.0f;
  //
  // TODO
  //
}
```

In the preceding snippet, we first set up the variables that we need to reference. The pathArray variable stores the nodes array that's returned from the AStar FindPath method. In the following code block, we use the Start method to look for objects with the tags Start and End and initialize pathArray. We are trying to find a new path at every interval, specified by the intervalTime property, in case the positions of the start and end nodes have changed. Finally, we call the FindPath method:

```
  void Start () {
    objStartCube = GameObject.FindGameObjectWithTag("Start");
    objEndCube = GameObject.FindGameObjectWithTag("End");
    pathArray = new List< Node>();
```

```
    FindPath();
  }
  void Update () {
    elapsedTime += Time.deltaTime;
    if (elapsedTime >= intervalTime) {
      elapsedTime = 0.0f;
      FindPath();
    }
  }
```

Since we implemented our pathfinding algorithm in the AStar class, finding a path is much simpler. In the following snippet, we first take the positions of the start and end game objects. Then, we create new Node objects using the GetGridIndex helper methods in GridManager to calculate their respective row and column index positions inside the grid. After that, we call the AStar.FindPath method with the start node and target node, storing the returned array list in the local pathArray property. Finally, we implement the OnDrawGizmos method to draw and visualize the resulting path:

```
void FindPath() {
  startPos = objStartCube.transform;
  endPos = objEndCube.transform;
  //Assign StartNode and Goal Node
  var (startColumn, startRow) = GridManager.instance.GetGridCoordinates(startPos.position);
  var (goalColumn, goalRow) = GridManager.instance.GetGridCoordinates(endPos.position);
  startNode = new Node(GridManager.instance.GetGridCellCenter(startColumn, startRow));
  goalNode = new Node(GridManager.instance.GetGridCellCenter(goalColumn, goalRow));
  pathArray = new AStar().FindPath(startNode, goalNode);
}
```

We look through our pathArray and use the Debug.DrawLine method to draw the lines, connecting the nodes in pathArray:

```
void OnDrawGizmos() {
  if (pathArray == null)
    return;
  if (pathArray.Count > 0) {
    int index = 1;
    foreach (Node node in pathArray) {
      if (index < pathArray.Count) {
        Node nextNode = pathArray[index];
        Debug.DrawLine(node.position, nextNode.position, Color.green);
        index++;
      }
    };
  }
}
```

When we run and test our program, we should see a green line connecting the nodes from start to end.

3. Setting up the scene

We are going to set up a scene.

- Set up a demo scene, named **Lab10_PathFinding_WithGrid_{YourInitials}**, with the following game objects:

Game Object	Type	Tag	Parent	P(x,y,z)	R(x,y,z)	S(x,y,z)	Color/Textur
Floor	Plane	Floor	Root	(0,0,0)	(0,0,0)	(10,1,10)	Gray
Start	Cube	Start	Root	Random	Random	Random	Green
End	Cube	End	Root	Random	Random	Random	Red
Obstacles	Empty	-	Root	(0,0,0)	(0,0,0)	(1,1,1)	-
Obstacle1-9	Cube	Obstacle	Root	Random	Random	Random	Gray/Wall
Scripts	Empty	-	Root	(0,0,0)	(0,0,0)	(1,1,1)	-
GridManager	Empty	-	Scripts	(0,0,0)	(0,0,0)	(1,1,1)	-
TestAStar	Empty	-	Scripts	(0,0,0)	(0,0,0)	(1,1,1)	-

- Attach the **GridManager.cs** script to **GridManager** object.
  - Set up:
    - the number of rows - 25
    - the number of columns - 25
    - the grid cell size - 2.
- $Take\ Snapshot$

3.1. Testing the pathfinder

- Play
- Notice that A* pathfinding algorithm in action in the Scene View (for Gizmos Visualization).
- $Take\ Snapshot$
- Try to move the start or end node around in the scene using the editor's movement gizmo (not in the Game view, but the Scene view)
- $Take\ Snapshot$
- Notice that the path is updated dynamically in real time. On the other hand, if there is no available path, you get an error message in the console window instead.
- Design obstacles to not have a path from start to end
- $Take\ Snapshot$
- Play. What do you see ?
- Notice the error message in console.
- $Take\ Snapshot$

Zip into a file named Lab10_{YourInitials}.zip with the following content:

- the solution (you may create a unitypackage but make sure you have all is needed first).
- the snapshots
- a short (~1 min) video with playtesting of the solution

Upload in eCentennial.

4. Summary

We learned how to implement the A* pathfinding algorithm in Unity3D via grid-based method.

First, we implemented our own:

- A* pathfinding class,
- grid representation class,
- priority queue class, and
- node class.

Finally, we used debug draw functionalities to visualize the grid and path information.

In the next lab, we'll use Unity to implement **Behaviour Trees**.

4.1. Tasks:

- Task 1: Compare and contrast the two implementations of A*: the not grid-based and the grid based ones.
- Bonus Task (continuing from last lab): Pick a simple graph and run the algorithm manually in a piece of paper, as we do in class (any of them actually not just A* — BFS, DFS, Dijkstra). If anything I can give you graph samples. You can strat with very simple ones (3-4 nodes) to get the feeling of the algorithm. Show me your work. I may ask you to go over one or two steps. You can have the pseudocode open.. **NB: This might be very important in your job interviews. If we don't have time in class, I will happily consider this outside of the class time - any time.**