Link to Lab Drive

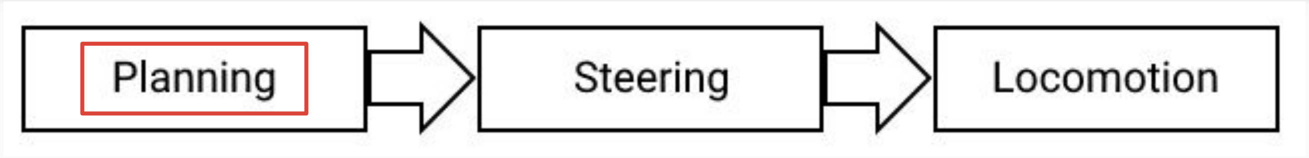
https://drive.google.com/drive/folders/1jgFTWCJ5JD-JnIY7n_L7s2n2vA1F9nNX?usp=sharing

OR

https://tinyurl.com/ta-comp476-daniel

Pathfinding

Planning



When moving characters within a game there's 3 phases : planning, steering, and then locomotion, in that order.

At the moment, we are more interested in non-human (AI) character movement as human character movement is a more trivial process. We've already seen Locomotion (animation) and Steering, but before these can be applied we need to know where we need to move to which we refer to as the **Planning** phase.

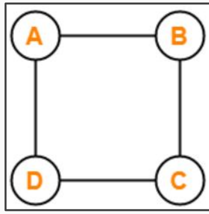
So how does one plan a movement? Well, this can be broken down into two problems:

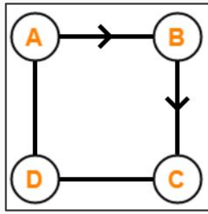
- **How to reason about the environment** : This problem is more **global** and for the most part **static**, in that it takes into account the whole scene and shares this info publically which usually does not change much. The information is stored in one or more data structures (usually some sort of **graph**).
- **How to get there (calculating the path)** : This problem is more **local** and **dynamic**, it only considers where to move which is specific to the character you are moving and as such, will vary between characters/agents. This problem requires as input, at the very least: the **graph**, a **starting point**, and an **end point**. The **result is a path** (sequence of graph nodes) and it is usually **calculated using a pathfinding algorithm** like Dijkstra or A*.

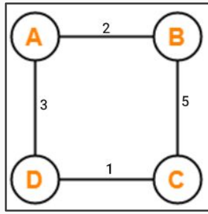
Graphs

There are multiple ways to build a graph and there is no perfect generic solution. The type of graph you need is going to be specific to your use case. There are many types of graphs in the wild but for now, let's look at just a few different types of graphs:

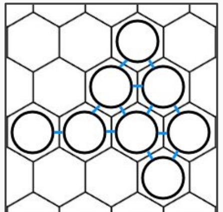
- **Basic graph** : The most basic graph will have a set of **nodes** and **edges**. The nodes (also called vertices) are connected together by the edges. The edges are bidirectional (they go both ways).
- **Directed graph** : Also known as **digraph**. A graph is called a digraph if there are unidirectional (one way) edges present between any nodes of the graph.
- **Weighted graph** : A graph is referred to as a weighted graph if all edges in a graph have a set weight (a.k.a cost). At least one of the edges in the graph must have a different weight than any other edge or else it just degrades to a non-weighted graph. The weight of an edge can be increased/decreased to signify that taking a certain path is more costly. Example: walking over concrete vs walking over mud.
- **Multi graph** : A graph is referred to as a multi graph if the graph doesn't consist of any self-loops, but parallel edges are present in the graph. If there is more than one edge present between two vertices, then that pair of vertices is said to be having parallel edges. Example: using a boat vs swimming to cross a river.
- **Grid graph** : Commonly seen in games that use grid based movement, a grid graph is a graph in which the edges only traverse cardinal directions (N,S,W,E). There can be more than 4 directions depending on the type of grid graph being used as well. A square grid can be 4-way or 8-way whereas a hex grid can be 6-way or 12-way.

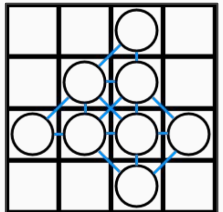


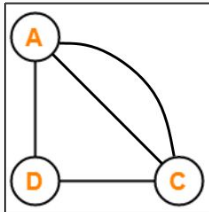




A graph does not need to be exclusively one of the types listed above. You can have a graph that is directed, weighted, multi, and grid-based. The type of graph you choose will depend on your use case and the geography of the scene.







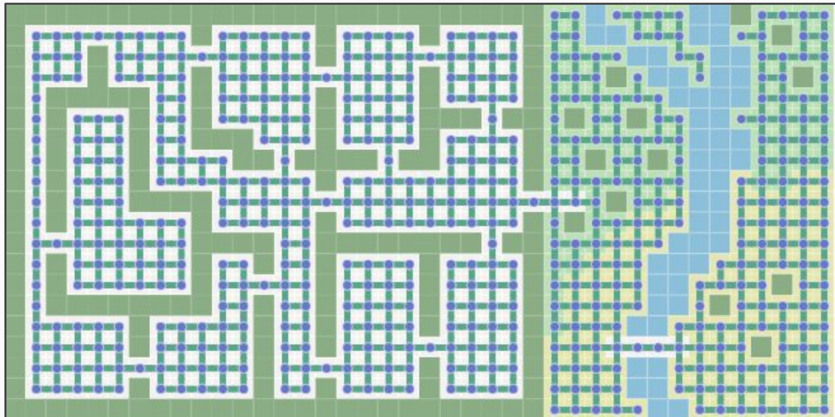
Graphs

In most games we usually don’t navigate in a void, instead there are **obstacles** in the scene that we want our AI agents to avoid. An obstacle can either be **static** or **dynamic** meaning that it either **does not move** or can **possibly move**.

How do we represent obstacles in graph form? There are three general strategies :

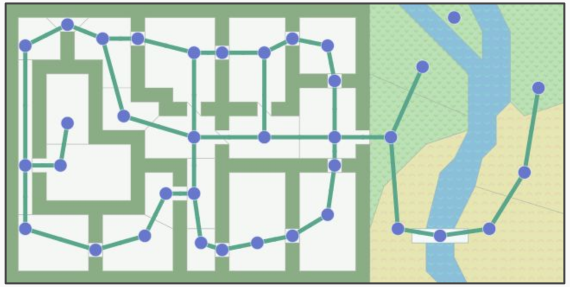
- **Remove nodes**. If an obstacle occupies the location of a node, you can remove those nodes from the graph. You also need to remove the corresponding edges, which is usually automatic if the graph is designed properly.
- **Remove edges**. If obstacles occupy the space between nodes (intersect the edges), you can remove those edges from the graph. If obstacles only occupy the location of the nodes, you can also just remove the edges that lead to those nodes but leave the nodes there.
- **Infinite weight edges**. If obstacles occupy the space between nodes, you can also mark those edges with Infinity as the weight. If obstacles occupy the location of a node, you can mark edges leading to those nodes as having Infinity as the weight. Altering the weights of the edges may be easier if you want to change the obstacles after the graph is constructed.

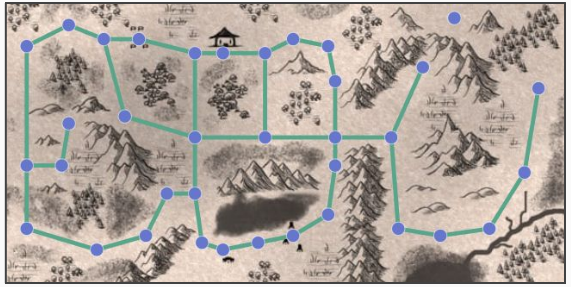
For example, the graph on the right removed the nodes (and the edges leading to those nodes) that overlap walls and untraversable areas (river). However, let’s say that in your game the player activates a switch that drains the river allowing characters to cross over it. Instead of rebuilding or swapping graphs which is expensive, one could simply keep the nodes and edges connecting the river to the surrounding land and when the switch is activated change the weights for edges leading to river nodes from Infinity to an actual weight value. This would save both time and space.



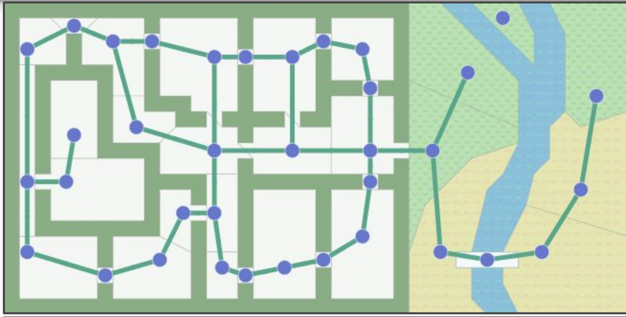
Graphs

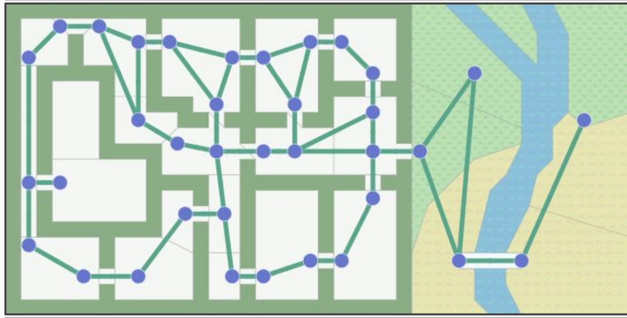
The geographical information of the scene is only important when constructing the graph. The **pathfinding algorithms**, which we will look at soon, do not know anything about the scene itself; they **only know about the graph** data structure! They don't know whether a node is indoors or outdoors, in a room or is a doorway. They don't know the difference between these two maps:

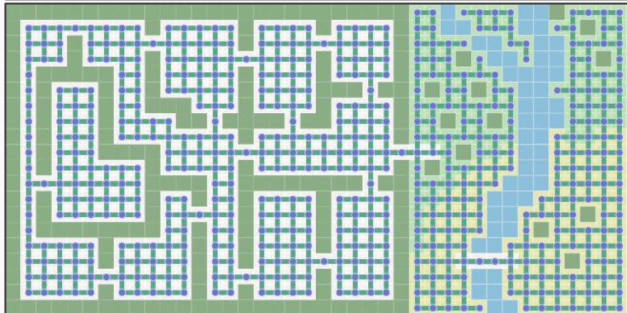




For any given scene, there are many different ways of making a graph. Either you will have to manually configure the graph or you can automatically generate it through code. The example graphs on the right showcase a graph with nodes placed at doorways, a graph with edges going through doorways, and a grid graph respectively.







Pathfinding

In games we often want to find paths from one location to another, this is referred to as **pathfinding**. We're not only trying to find the path but we also want to take into account travel time (costs).
As such **pathfinding algorithms** take the following as input and output the following.

Input:

- A graph
- A starting node
- One or more goal/destination nodes
- A heuristic function (optional)

Output:

- A sequence of nodes (a path)

There are lots of algorithms that run on graphs, however we will cover the following three:

Breadth First Search : explores equally in all directions. This is an incredibly useful algorithm, not only for regular path finding, but also for procedural map generation, flow field pathfinding, distance maps, and other types of map analysis.

Dijkstra's Algorithm (also called Uniform Cost Search) : lets us prioritize which paths to explore. Instead of exploring all possible paths equally, it favors lower cost paths. We can assign lower costs to encourage moving on roads and higher costs to avoid forests, discourage going near enemies, etc. When movement costs vary, we use this instead of Breadth First Search.

A* : a modification of Dijkstra's Algorithm that is optimized for finding the goal. Dijkstra's Algorithm can find paths to multiple locations whereas A* finds paths to one location, or the closest of several locations. It prioritizes paths that seem to be leading closer to a goal (uses a **heuristic function** to determine this).

Pathfinding

The key idea for all of these algorithms is that we keep track of an expanding ring called the **frontier** (also known as **open list**), a list of processed nodes called the **closed list**, and a table to keep track of which node we came from (let's call this the **came_from table**).

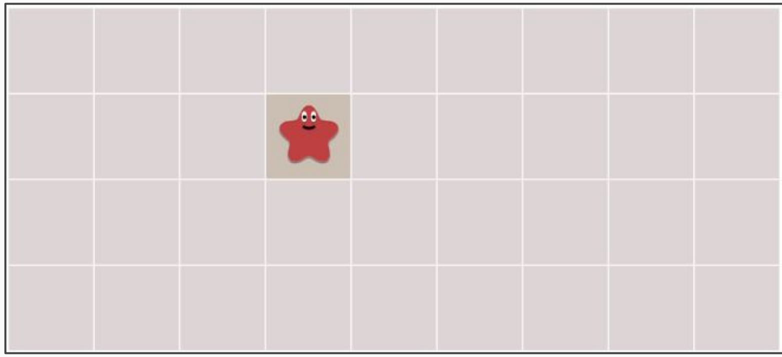
How do we implement this? Repeat these steps until the frontier is empty:

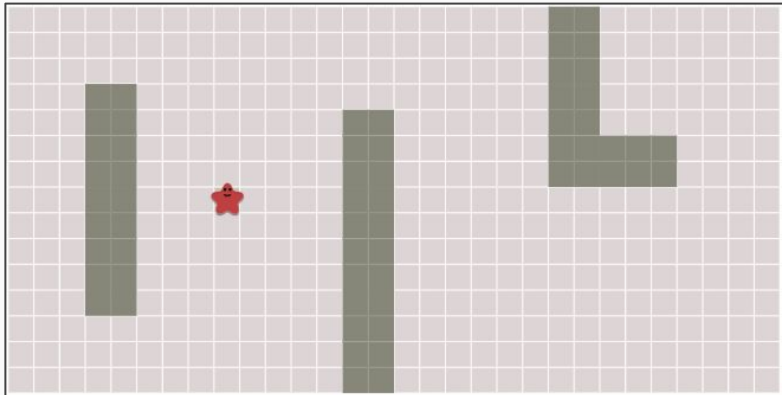
1. Pick and remove a node from the open list.
2. Expand/update the open list by looking at the picked node's neighbors.
3. Update the came_from table

The following pseudo code is the essence of our pathfinding algorithms and is also the code for our **breadth first search** algorithm:

```
open_list = Queue()
open_list.Push(start_node)
closed_list = Set()
closed_list.Add(start_node)
came_from = Dictionary()
came_from[start_node] = null

while not open_list.Empty():
   current_node = open_list.Pop()
   foreach next_node in graph.GetNeighbors(current_node):
      if next_node not in closed_list:
         open_list.Push(next_node)
         came_from[next_node] = current_node
         closed_list.Add(next_node)
```

For each node, **came_from** points to the place where we came from. These are like "breadcrumbs". It's enough to reconstruct the entire path.

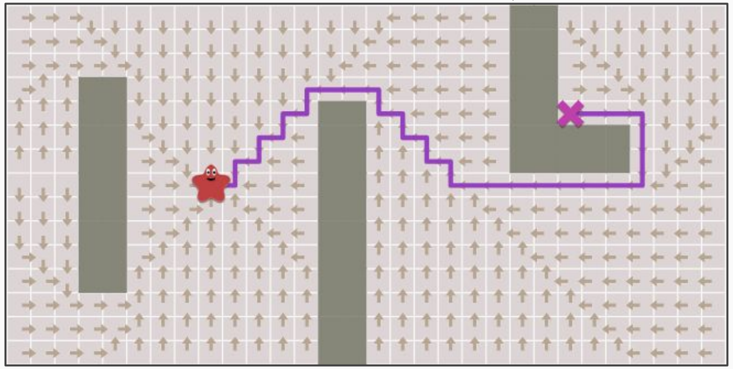




Pathfinding

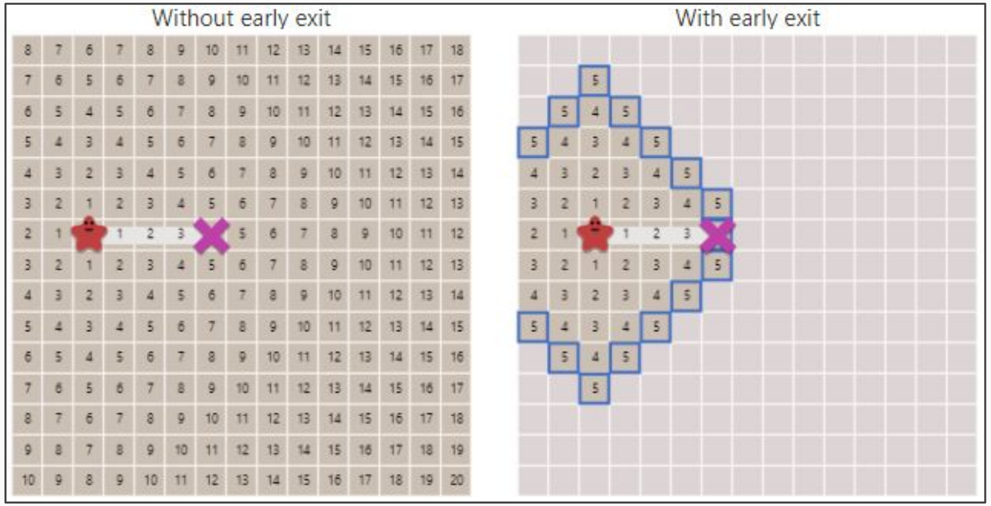
The code to reconstruct paths is simple, just follow the arrows backwards from the goal to the start (use the **came_from** table).

```
current = goal
path = []
while current != start_node:
   path.Append(current)
   current = came_from[current]

path.Reverse() // optional
```



More often than not we don't need all the paths, we only need a path from one location to a single end location. We can stop expanding the open_list as soon as we've found our goal. This is known as "**early exit**". The code is straightforward:

```
while not open_list.Empty():
   current = open_list.Pop()

   if current == goal:
      break

   foreach next in graph.GetNeighbors(current):
      if next not in closed_list:
         open_list.Push(next)
         came_from[next] = current
         closed_list.Add(next)
```

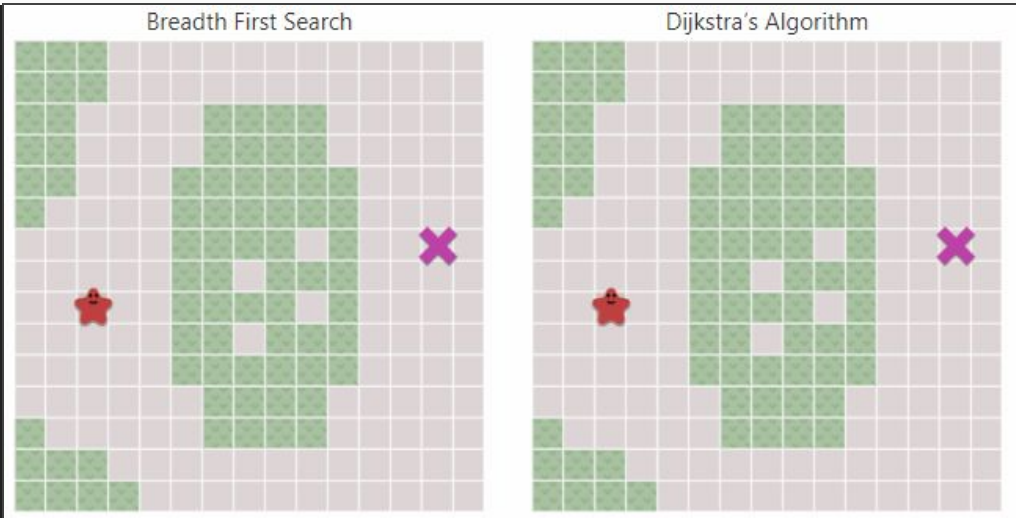


Pathfinding

So far we've assumed that our edges have the same cost. In some cases there are different costs for traversing different edges in a graph. For example, in a video game moving through plains or desert might cost 1 move point but moving through forest or hills might cost 5 move points. We'd like the pathfinding algorithm to take these costs into account so that the path returned to us will actually be the most efficient path. To do this, we use **Dijkstra's algorithm**.

How does it differ from Breadth First Search? We now need to **track movement costs** (also known as **g cost**), so let's add a new table called **gn**, to keep track of the total movement cost from the start node to a given node n. Also, we now we want to always **pick the node with the lowest gn cost from the open list**. We can easily achieve this by turning our **open list** into a **priority queue** that's sorted by lowest g costs first. Lastly, we also need to alter the logic a little bit. Instead of adding a node to the open list if the node is not in the closed list, we add it if the new movement cost to the node is better than the best previous g cost.

```
open_list = PriorityQueue()
open_list.Push(start_node, 0)
// initialize the closed_list and came_from same as before
gn = Dictionary()
gn[start_node] = 0
while not open_list.Empty():
   current = open_list.Pop()
   closed_list.Add(current)
   foreach next in graph.GetNeighbors(current):
      g_next = gn[current] + graph.EdgeWeight(current, next)
      if next not in gn or g_next < gn[next]:
         gn[next] = g_next
         open_list.Push(next, g_next)
         came_from[next] = current
```



Pathfinding

With Breadth First Search and Dijkstra's Algorithm, the open list expands in all directions. This is good if you're trying to find a path to more than one location. However, more often than not, you only have a single goal. This is where the **A* algorithm** comes in and makes its optimization. Instead of expanding in all directions, A* **tries to expand towards the goal** more than it expands in other directions. To do this A* takes an additional input, a **heuristic function** (**h(n)**) that tells us how close a given node n is to the goal. It then uses the result of our heuristic + the movement cost as the priority value for the priority queue. Doing this will cause the algorithm to expand in the general direction of the goal and will also **guarantee the best path as long as the heuristic we choose is admissible**.

What exactly is a heuristic function and **what does it mean to be admissible**?
A heuristic function is basically an "educated guess" of how close a node is to the goal. How you implement the heuristic function is up to you and depends on the problem you are trying to solve. Here are three popular basic heuristic methods for 2D graphs:

- Manhattan Distance : h(n) = D * (|current.x - goal.x| + |current.y - goal.y|)
  - Good for square grid graphs (4-way movement). D is the cost of an orthogonal movement.
- **Diagonal Distance** : h(n) = D * (|current.x - goal.x| + |current.y - goal.y|) + (D_2 - 2 * D) * min(|current.x - goal.x|, |current.y - goal.y|)
  - If your grid allows diagonal movement as well then you should use this instead. D_2 is the cost of a diagonal movement.
- **Euclidean Distance** : h(n) = D * sqrt((current.x - goal.x) * (current.x - goal.x) + (current.y - goal.y) * (current.y - goal.y))
  - If your graph allows multi directional movement (not a grid) then you should probably use euclidean distance.

Note that when calculating these heuristics we can disregard things like obstacles, dead ends, etc. As long as we do not overestimate the actual cost then its ok, which is why the heuristic is often also called a "**guess**".

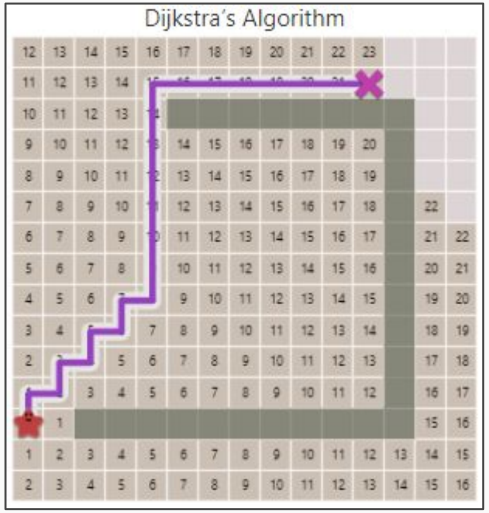
Finally, a heuristic is said to be **admissible if and only if the heuristic function does not overestimate the actual cost** to get to the goal. If your heuristic is not admissible, then A* does not guarantee the best path to the goal.

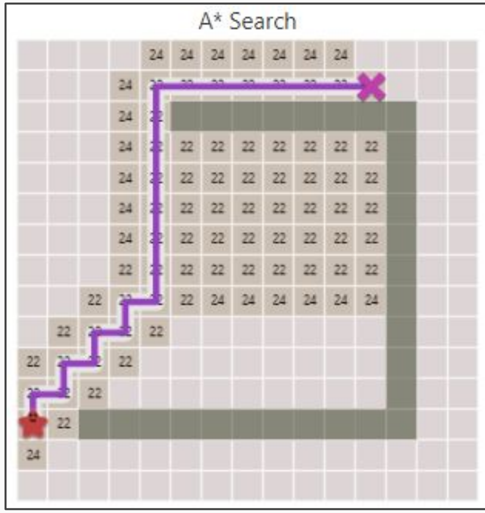
Pathfinding

The final pseudo code for A* is very similar to Dijkstra's Algorithm but with very minor changes. The only difference is that we use both g(n) and h(n) together to determine which node in the open list to expand first:

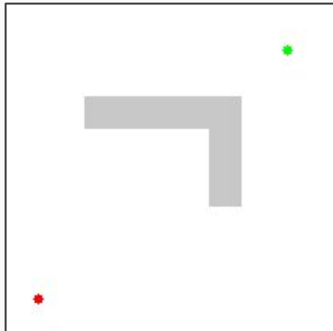
```
open_list = PriorityQueue()
open_list.Push(start_node, 0)
closed_list = Set()
closed_list.Add(start_node)
came_from = Dictionary()
came_from[start_node] = null
gn = Dictionary()
gn[start_node] = 0
fn = Dictionary()
fn[start_node] = gn[start_node] + Heuristic(start_node, goal)

while not open_list.Empty():
    current = open_list.Pop()
    closed_list.Add(current)
    foreach next in graph.GetNeighbors(current):
        g_next = gn[current] + graph.EdgeWeight(current, next)
        if next not in gn or g_next < gn[next]:
            gn[next] = g_next
            fn[next] = g_next + Heuristic(next, goal)
            open_list.Push(next, fn[next])
            came_from[next] = current
```

Note: Dijkstra is just A* with a heuristic of 0.





As long as the heuristic is admissible, A* finds an optimal path and does it faster than Dijkstra's Algorithm.



Hierarchical Pathfinding

Hierarchical Pathfinding (HP), is an optimization that can be applied to our pathfinding algorithms.

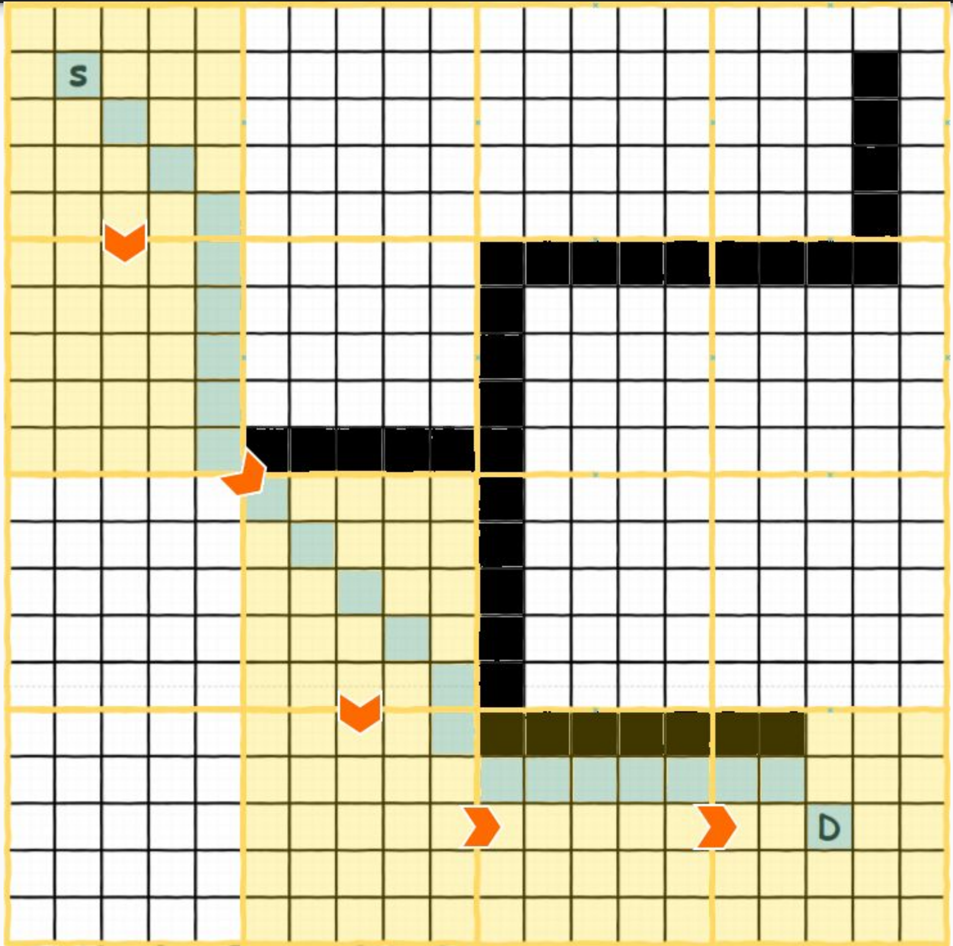
By separating the graph into clusters, we can create a “higher level” graph which is even more simple than the original.

We can then perform a pathfinding algorithm on the higher level graph to determine which clusters will take us to our goal (cluster path - shown in yellow in the example on the right).

With this knowledge, we can optimize the pathfinding algorithm at the “lower level” (shown in blue in the example on the right).

The example on the right is using **Hierarchical Pathfinding A* (HPA*)** in order to calculate the path.

When calculating the path at the lower level, since we know which clusters the path should be part of, then when expanding the open list we can ignore nodes that are not part of the cluster path returned by the higher level pathfinding. This results in an optimization as less nodes are being explored.



Tasks

Reasoning about the environment

- In Unity, we will look at generating a very basic grid graph.
- We will also make it so that nodes and edges that intersect obstacles will not be generated.

How to get there (calculating the path)

- Implement Dijkstra's algorithm in Unity. Most of the code is already provided, you just need to fill in the missing code.
- Construct the path from the pathDict variable
- Implement A* algorithm using an admissible heuristic
- We will color code the nodes using a debug property to visually show our open list, our closed list, and the final path.

Thank You

Link to Lab Drive

https://drive.google.com/drive/folders/1jgFTWCJ5JD-JnIY7n_L7s2n2vA1F9nNX?usp=sharing

OR

https://tinyurl.com/ta-comp476-daniel

Other Links

https://docs.unity3d.com/2020.3/Documentation/Manual/index.html

https://docs.unity3d.com/2020.3/Documentation/Manual/ExecutionOrder.html

https://www.redblobgames.com/pathfinding/a-star/introduction.html