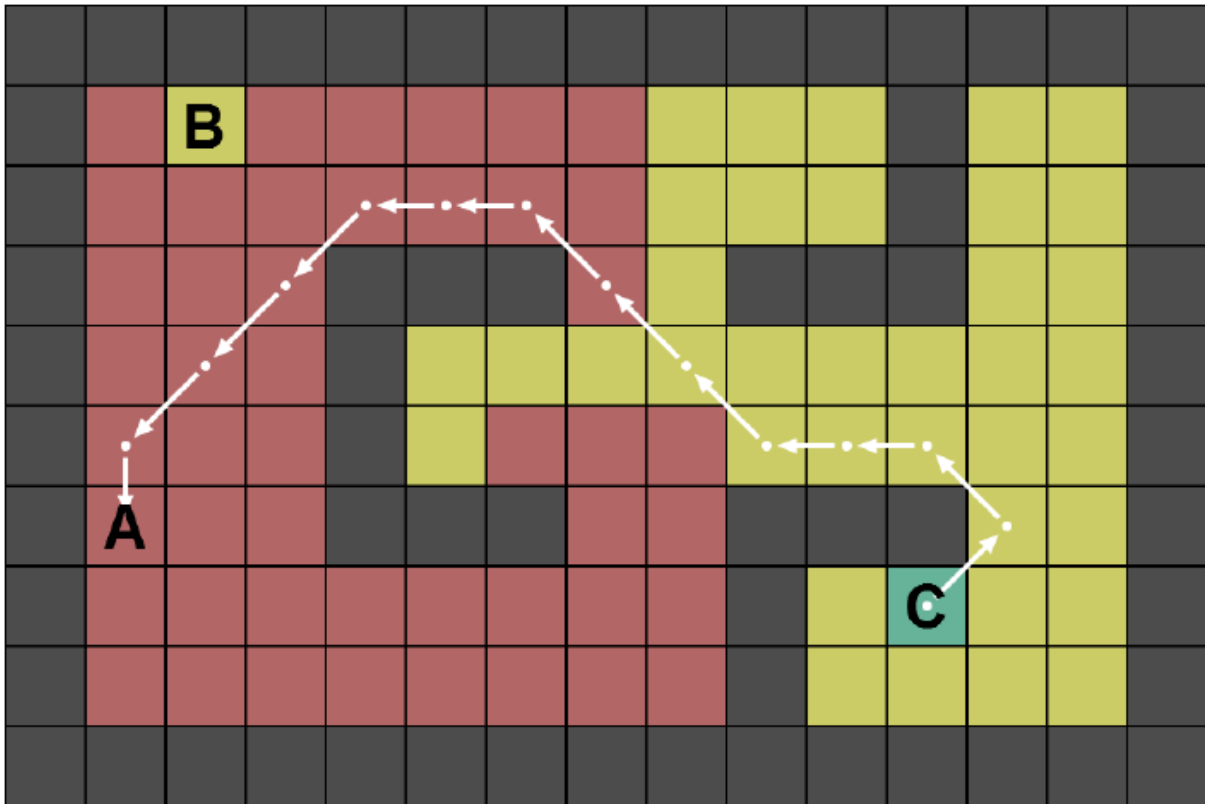
Fundamentals of Pathfinding



1. **Why use pathfinding?**

   Pathfinding is one of the most common sources of complexity in term projects. Many types of games have enemies that need to move intelligently, or mazes that need to be solved, and these types of interactions can be done with pathfinding. They can also be used for things you wouldn't think of, Bubble Witch or the floodfill button in paint applications. These algorithms are often paired up with a maze/terrain generation algorithm as part of MVP. Later on more pathfinding algorithms can be added for additional complexity.

   There are several different kinds of pathfinding that are useful for different things:

   - Finding out whether two locations connected
   - Finding all points that are connected to a location
   - Finding any path between two locations
   - Finding the path between two locations with the fewest steps
   - Finding the path between two locations with the lowest cost

2. **Case Study: Pacman**

Pacman is a relatively common type of term project that almost always includes pathfinding. Most MVP definitions for Pacman involve the ghosts using one or more complex pathfinding algorithms to chase Pacman. Other examples are creepers in Minecraft, a bot playing Agar.io, or how troops in Civ V plan routes between different cities.

3. **Case Study: Bubble Witch**

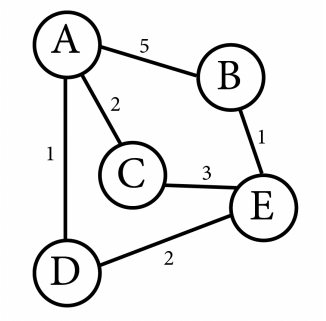
This game works by launching colored bubbles into the bubbles already in the air, and if it collides with bubbles of the same color, then every bubble of that color that is connected to the collision is destroyed. Also, if the bubbles in the air become disconnected from the ceiling, they are also destroyed. Finding out if a bubble is connected to the ceiling, or is destroyed by a collision, can be done with pathfinding. This is similar to how a the paint bucket tool in paint applications works, and it is often referred to as **floodfill.**

4. **Graphs**

Graphs are an important datastructure in computer science, especially for pathfinding. Graphs have two components: the nodes (i.e. locations) and edges (i.e. connections between adjacent locations). For example, if you wanted to represent a problem of how to travel between different American cities via train, you could represent it as a graph where the cities are the nodes, and train routes between cities are the edges.

There are several different types of graphs. Graphs can be **undirected** (meaning that each edge is 2-way) or **directed** (meaning that the edges can be 1-way). Graphs can also be **unweighted** (meaning that each edge has equal cost) or **weighted** (meaning that each edge has its own cost).

Below is a visual example of a undirected-weighted graph with 5 nodes labeled A-E. The edges are shown as lines connecting the nodes, and each edge has its weight next to it.



Back to the train example, that would likely be a undirected-weighted graph since train routes are usually 2-way, but each route is has its own length. Pacman would likely be an undirected-unweighted graph where each cell is a node, and any neighboring cells without a wall between them have an edge. Bubble Witch would also likely be an undirected-unweighted graph where each bubble is a node, and neighboring bubbles of the same color share an edge. If we had a city map, where intersections were the nodes and streets connecting intersections are edges, then it would likely be a directed-weighted graph since the distance between intersections could vary, and some of the streets could be 1-way.

5. **Algorithms**

There are four major pathfinding algorithms worth considering:

- Depth First Search (a.k.a. DFS)
- Breadth First Search (a.k.a. BFS)
- Dijkstra's Algorithm
- A* Algorithm

DFS works great for solving mazes, finding *any* path between two locations, or floodfill. It uses recursive backtracking (in fact, DFS and backtracking are almost interchangeable terms)

BFS works great for finding the shortest path between two locations in an unweighted graph, but also works for the floodfill. It only uses loops, but can be a bit harder than DFS since its less similar to what is done in 112 so far.

Dijkstra finds the shortest path between two location in a weighted graph. Like BFS it only uses loops, but the algorithm is more complex and requires more datastructures to keep track of the state of the algorithm.

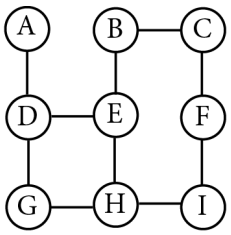
A* (pronounced A-Star) is a minor extension of Dijkstra that makes it run faster. It is usually a good idea to do Dijkstra first, then upgrade it to A* later.

6. **Example: BFS**

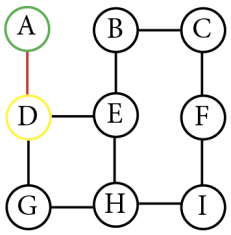
The idea of BFS is that you start at a single node, then visit all of its neighbors, then visit all of its neighbors' neighbors, then all of the neighbors' neighbors' neighbors, etc. until you find the target. This has the effect of radiating outward from the start node. Along the way, each time a node is visited for the first time, the previous node that it was visited from was recorded. This allows you to trace backwards and extract the path at the end. Below is the pseudoocode for BFS:

```
define BFS(start, target):
    Initialize a queue containing the start node
    Initialize a datastructure marking which nodes are visited
    Mark the start node as visited
    Initialize a datastructure mapping each node to its previous node
    Repeat until the queue is empty:
        Extract a node from the front of queue
        If the node is the target:
            Use the mapping to trace backward and extract the path
            Return the path
        For each neighbor N of the node:
            If N is unvisited:
                Mark N as visited
                Add N to the end of the queue
                Update the mapping so that N points to the node
    Return no path
```

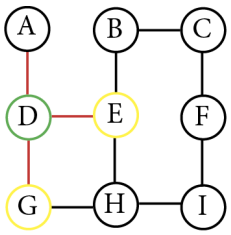
Below is an example of BFS finding a path in a graph. The goal is to find a path from node A to node C.



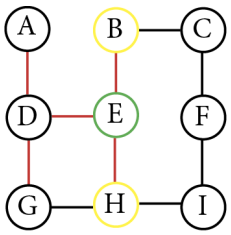
Q = [A]
V = {A}
P = {}
Starting state



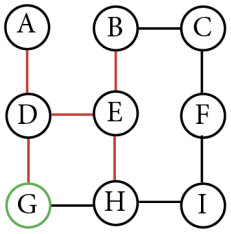
Node: A
Q = [D]
V = {A, D}
P = {D: A}
Visiting neighbor D



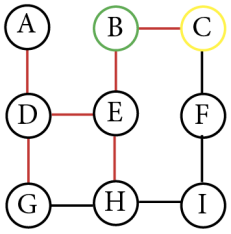
Node: D
Q = [E, G]
V = {A, D, E, G}
P = {D: A, E: D, G: D}
Visiting neighbors E and G



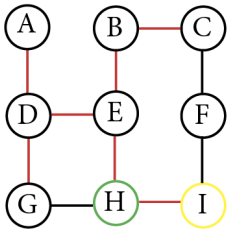
Node: E
Q = [G, B, H]
V = {A, D, E, G, B, H}
P = {D: A, E: D, G: D, B: E, H: E}
Visiting neighbors B and H



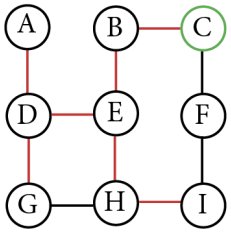
Node: G
Q = [B, H]
V = {A, D, E, G, B, H}
P = {D: A, E: D, G: D, B: E, H: E}
All neighbors already visited



Node: B
Q = [H, C]
V = {A, D, E, G, B, H, C}
P = {D: A, E: D, G: D, B: E, H: E, C: B}
Visiting neighbor C

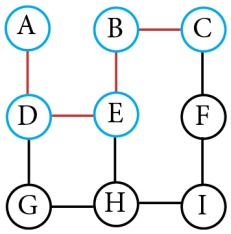


Node: H
Q = [C, I]
V = {A, D, E, G, B, H, C}
P = {D: A, E: D, G: D, B: E, H: E, C: B, I: H}
Visiting neighbor I



Node: C
Reached the target!!!

Using P to trace the path from A to C:
C maps to B
B maps to E
E maps to D
D maps to A
Therefore the path is [A, D, E, B, C]

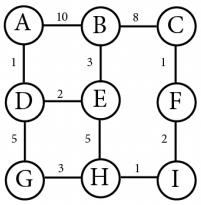


7. Example: Dijkstra

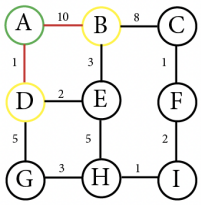
Dijkstra has some similarities to BFS, but the order it explores the nodes is more intelligent and it also has the ability to change the paths if it finds something better later down the line. Below is a pseudocode description of Dijkstra's algorithm:

```
define Dijkstra(start, target):
    Initialize a datastructure of unvisited nodes (initially all nodes)
    Initialize a mapping of each node to its distance from the start:
        ∞ for every node except start
        0 for the start
    Initialize a mapping of each node to its parent (initially empty)
    Repeat until every node is visited:
        Choose the unvisited node with the minimum distance
        If the node is target:
            Use the parents to trace backward and extract the path
            Return the path
        Remove the node from the univisted nodes
        Loop over each unvisited neighbor of the node
            D = distance to node + edge cost from node to neighbor
            If D is less than the distance to neighbor:
                Set neighbor's distance to D
                Set neighbor's parent to node
```

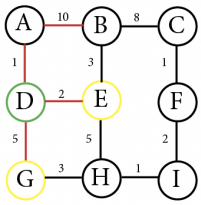
Below is an example of Dijkstra finding a path in a graph. The goal is to find the path of minimum cost from node A to node C. Each edge is labled with its cost.



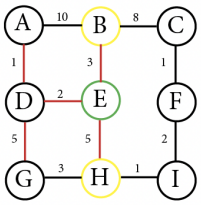
V = {}
D = {A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞, G: ∞, H: ∞, I: ∞}
P = {}
Starting state



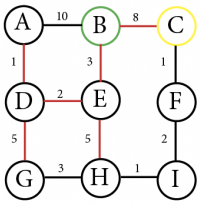
Node: A
V = {A}
D = {A: 0, B: 10, C: ∞, D: 1, E: ∞, F: ∞, G: ∞, H: ∞, I: ∞}
P = {B: A, D: A}
Relaxing edges AB, AD



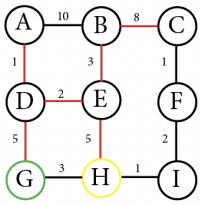
Node: D
V = {A, D}
D = {A: 0, B: 10, C: ∞, D: 1, E: 3, F: ∞, G: 6, H: ∞, I: ∞}
P = {B: A, D: A, E: D, G: D}
Relaxing edges DE, DG



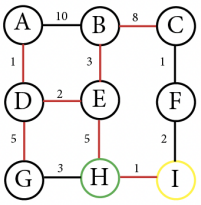
Node: E
V = {A, D, E}
D = {A: 0, B: 6, C: ∞, D: 1, E: 3, F: ∞, G: 6, H: 8, I: ∞}
P = {B: E, D: A, E: D, G: D, H: E}
Relaxing edges BE, EH



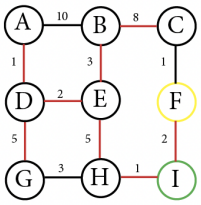
Node: B
V = {A, D, E, B}
D = {A: 0, B: 6, C: 14, D: 1, E: 3, F: ∞, G: 6, H: 8, I: ∞}
P = {B: E, D: A, C: B, E: D, G: D, H: E}
Relaxing edge BC



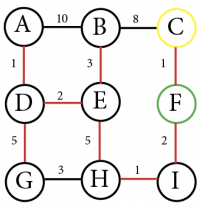
Node: G
V = {A, D, E, B, G}
D = {A: 0, B: 6, C: 14, D: 1, E: 3, F: ∞, G: 6, H: 8, I: ∞}
P = {B: E, D: A, C: B, E: D, G: D, H: E}
NOT relaxing edge GH



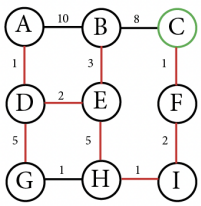
Node: H
V = {A, D, E, B, G, H}
D = {A: 0, B: 6, C: 14, D: 1, E: 3, F: ∞, G: 6, H: 8, I: 9}
P = {B: E, D: A, C: B, E: D, G: D, H: E, I: H}
Relaxing edge HI



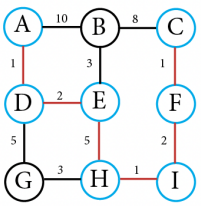
Node: I
V = {A, D, E, B, G, H, I}
D = {A: 0, B: 6, C: 14, D: 1, E: 3, F: 11, G: 6, H: 8, I: 9}
P = {B: E, D: A, C: B, E: D, F: I, G: D, H: E, I: H}
Relaxing edge IF



Node: F
V = {A, D, E, B, G, H, I, F}
D = {A: 0, B: 6, C: 12, D: 1, E: 3, F: 11, G: 6, H: 8, I: 9}
P = {B: E, D: A, C: F, E: D, F: I, G: D, H: E, I: H}
Relaxing edge FC



Node: C
Reached the target!



Using P to extract the path:
C -> F,
F -> I
I -> H
H -> E
E -> D
D -> A
Shortest path from A to C: [A, D, E, H, I, F, C]

8. **Where to start looking**

   All four of these algorithms are extremely is a well known, so there are MANY online sources for them. The rule of thumb is if you see code, **don't look at it**, but anything that looks like pseudo is fine.

   You must cite all sources, even the ones below!

   - The Wikipedia pages for these algorithms ([**DFS**], [**BFS**], [**Dijkstra**], [**A***]) should be one of your first starting points. They all contain detailed descriptions of the algorithms, descriptive pseudocode, and animations of the algorithms in action. Note that the A* pseudocode is a bit *too* descriptive, so try not to copy it directly.

   - The GeeksForGeeks articles on these algorithms do provide decend descriptions, but also include Python code implementations. Avoid looking at the latter.

   - Some articles on Dijkstra: [**Brilliant**] [**Baeldung**] [**Medium**].
     Also some decent video links: [**Video 1**] [**Computerphile**]

   - Some articles on A*: [**Brilliant**] [**Isaac Computer Science**]
     Also some decent video links: [**Computerphile**]

   - Plenty of universities (including CMU) have course notes you can find online explaining some or all four of these algorithms. Some of these courses at CMU are 15-210, 15-281 and 15-451.