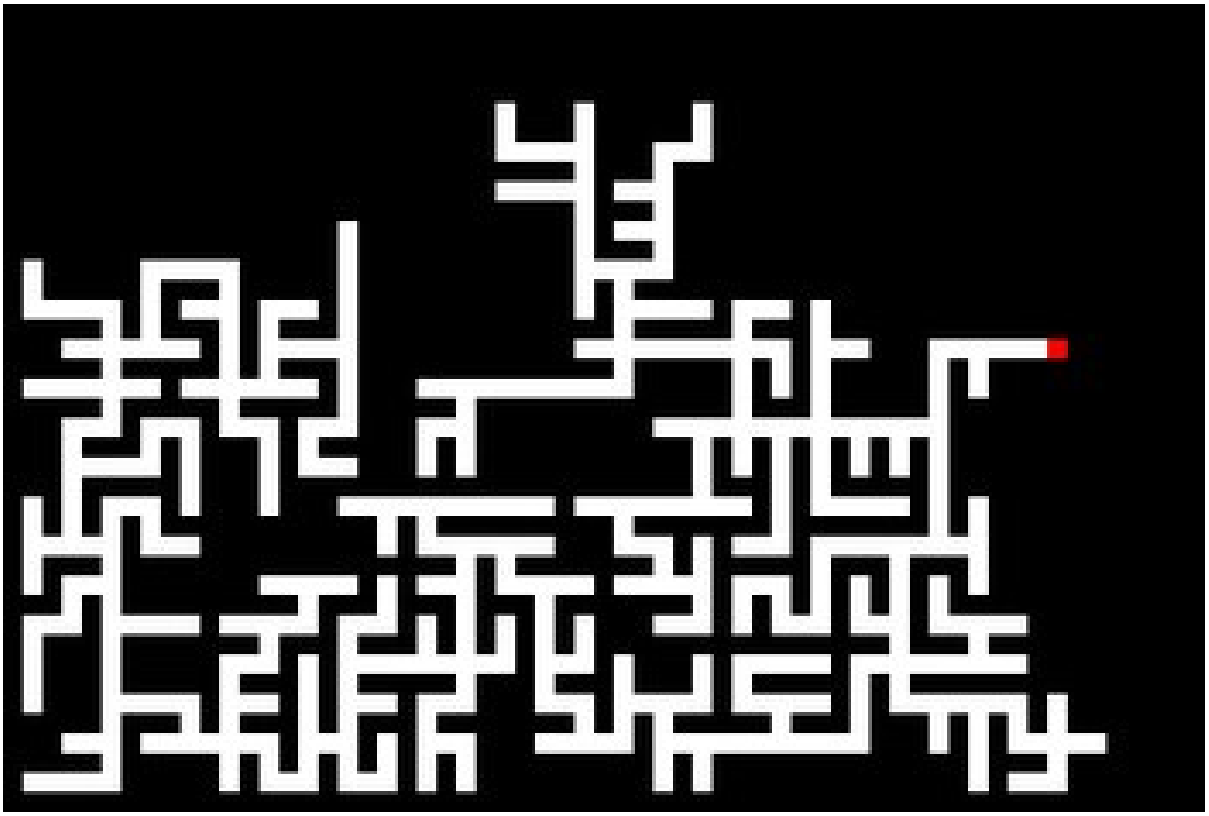
Fundamentals of Maze Generation



1. **Why use maze generation?**

   Maze generation is a great source of complexity in term projects. Many types of projects can find a use for a maze generation algorithm, and when used they make the path to MVP a lot clearer. Once you have one maze algorithm, you can always add more later on for even more points.

   Besides maze games, maze generation has all sorts of applications, including but not limited to:

   - Creating terrain with obstacles
   - Creating the layout of rooms in a dungeon crawler
   - Generating puzzles
   - Generating cool visuals
   - Teaching graph algorithms

2. **Case Study: Pacman**

Pacman is a relatively common type of term project that often includes maze generation. The combination of maze generation for creating the board and/or pathfinding for the ghosts are critical components of MVP.

Normally mazes aren't symmetric, and have only one path between any two points in the maze (the technical term for this is that they have no *cycles*). However, you can modify a maze algorithm to work for Pacman like so:

- Randomly add in extra edges to create cycles
- Only generate one quadrant of the grid with the maze algorithm, then create mirrored versions of it for the other 3 quadrants.

The lesson here is that all sorts of games/applications can use maze generation algorithms, even if they don't technically use mazes, by modifying the algorithms to suit your purposes. You can do something similar to generate twisted pipes games, Hashi/Bridges puzzles, Kami puzzles, Sokoban, Mini Motorways, Minesweeper, Jigsaw puzzles, Flow Mania, and so on.

3. **Case Study: Pokemon Gym**

Another common category of term project are games like Pokemon gym games, or Cuphead, where there is a side scrolling world that you navigate with NPCs that, when touched, cause you to enter a battle mode. These types of term projects often get their complexity from making the outer world be a maze, with the different enemies randomly dispersed throughout.

4. **Algorithms**

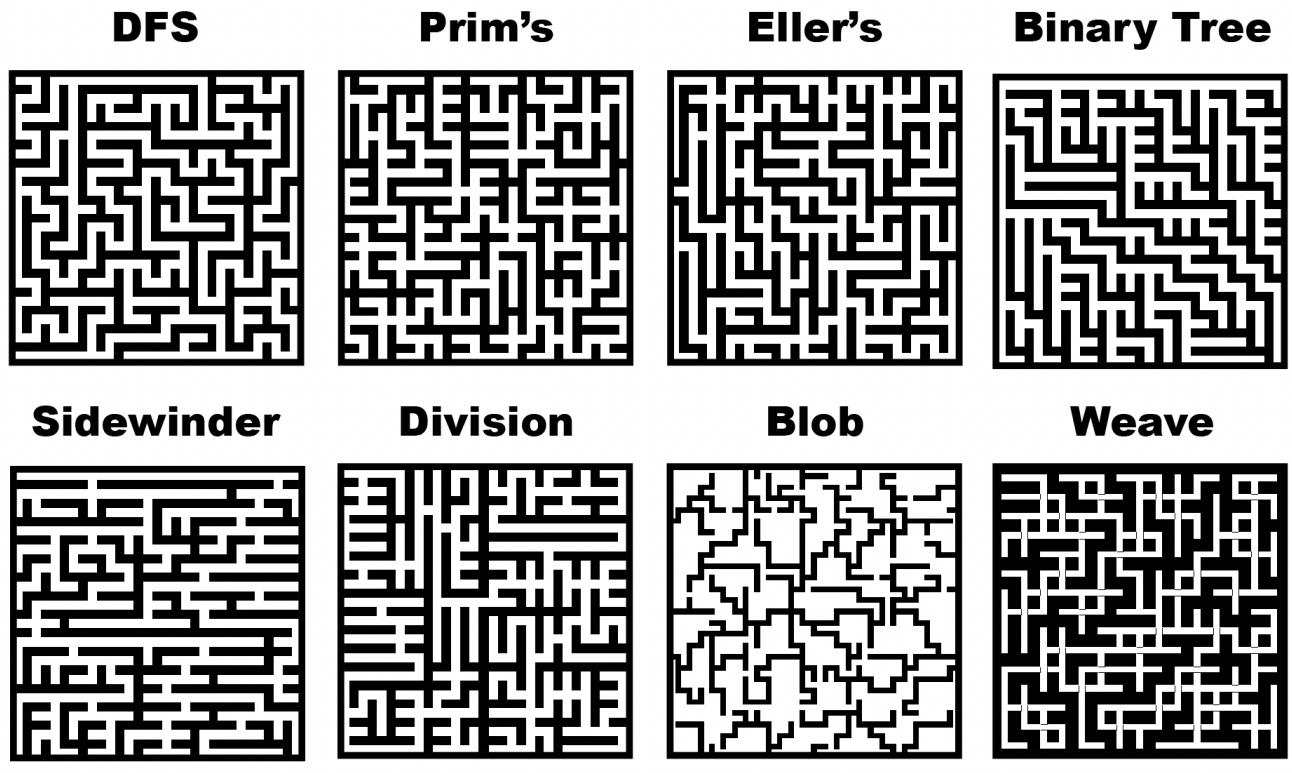
There are MANY maze generation algorithms (you can use more than one if you want). Below are the four options that are recommended to start out with (in increasing order of difficulty). The next few pages have more detailed descriptions of some of these.

- Depth First Search (a.k.a. DFS)
- Prim's Algorithm
- Hunt and Kill
- Kruskal's Algorithm

Once you have MVP, you can add more additional features by using several of the algorithms above (i.e. different modes that use different algorithms). You can also use the algorithms below, which are less recommended:

- Binary Tree Algorithm (almost zero complexity)
- Sidewinder Algorithm (low complexity)
- Aldous-Broder Algorithm (less complex cousin of DFS, and it has a random runtime)
- Wilson's Algorithm (similar to Aldous-Broder)
- Eller's Algorithm (extremely high complexity)
- Recursive Division (**use at your own peril**)
- Blob Division (a bit less nasty than Recursive Division; produces mazes with rooms)
- Weave Mazes (DFS and Kruskal can be modified to create "overpasses")

Some of these algorithms produce different looking mazes. Below are visual examples of how they can vary (Kruskal looks the same as Prim's, and Hunt Kill, Aldous-Broder, and Wilson's usually look like Eller's):

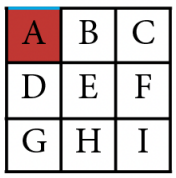


5. Example: DFS

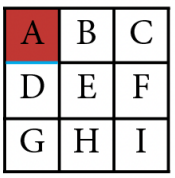
The DFS algorithm starts with no passages. It uses recursive backtracking to tunnel through the cells until every cell has been visited. An initially empty set is used to keep track of which cells have already been visited, thus ensuring no repeats. The algorithm proceeds as follows:

```
define backtracker(location, visited):
    Add the current location to the visited set
    Loop over all four neighbors of the current location in random order
        Skip if the neighbor has already been visited
        Add a passage from the current location to the neighbor
        Call backtracker(neighbor, visited)
```

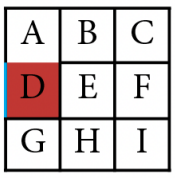
Below is an example of DFS creating a 3x3 maze. For the purposes of this demonstration, the 9 cells are labeled with letters A-I. The current cell (initially chosen to be cell A) is highlighted in red.



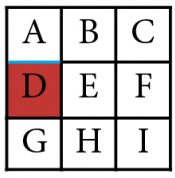
Current: A
Neighbor: Up
Can visit? No (out of bounds)



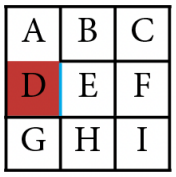
Current: A
Neighbor: Down
Can visit? Yes
--> Carve passage A-D



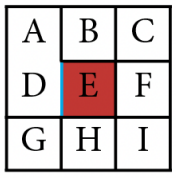
Current: D
Neighbor: Left
Can visit? No (out of bounds)



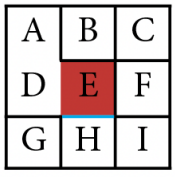
Current: D
Neighbor: Up
Can visit? No (already visited)



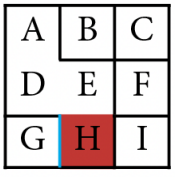
Current: A
Neighbor: Right
Can visit? Yes
--> Carve passage D-E



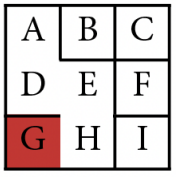
Current: E
Neighbor: Left
Can visit? No (already visited)



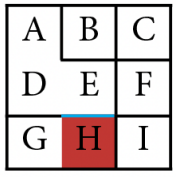
Current: E
Neighbor: Down
Can visit? Yes
--> Carve passage E-H



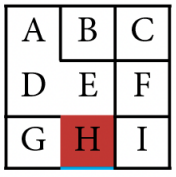
Current: H
Neighbor: Left
Can visit? Yes
Carve passage H-G



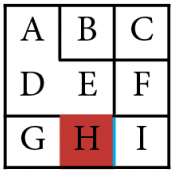
Current: G
All four neighbors are visited
or out of bounds
--> Backtrack



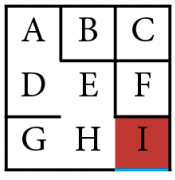
Current: H
Neighbor: Up
Can visit? No (already visited)



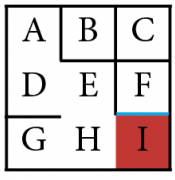
Current: H
Neighbor: Down
Can visit? No (out of bounds)



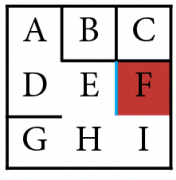
Current: H
Neighbor: Right
Can visit? Yes
Carve passage H-I



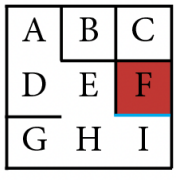
Current: I
Neighbor: Down
Can visit? No (out of bounds)



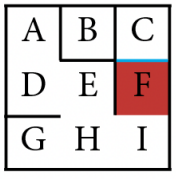
Current: I
Neighbor: Up
Can visit? Yes
--> Carve passage I-F



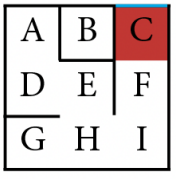
Current: F
Neighbor: Left
Can visit? No (already visited)



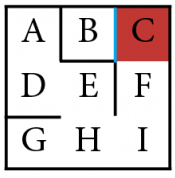
Current: F
Neighbor: Down
Can visit? No (already visited)



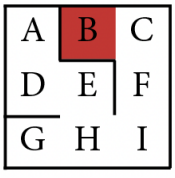
Current: F
Neighbor: Up
Can visit? Yes
--> Carve passage F-C



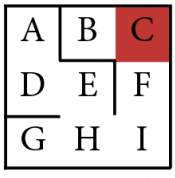
Current: C
Neighbor: Up
Can visit? No (out of bounds)



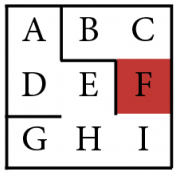
Current: C
Neighbor: Left
Can visit? Yes
--> Carve passage C-B



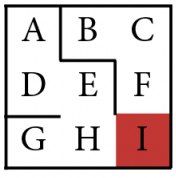
Current: B
All four neighbors are visited
or out of bounds
--> Backtrack



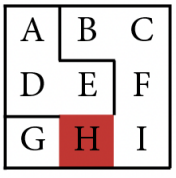
Current: C
All four neighbors are visited
or out of bounds
--> Backtrack



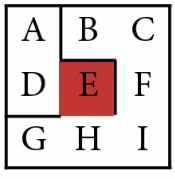
Current: F
All four neighbors are visited
or out of bounds
--> Backtrack



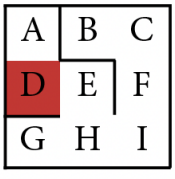
Current: I
All four neighbors are visited
or out of bounds
--> Backtrack



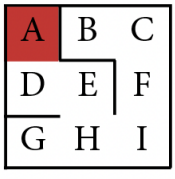
Current: H
All four neighbors are visited
or out of bounds
--> Backtrack



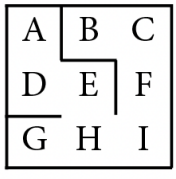
Current: E
All four neighbors are visited
or out of bounds
--> Backtrack



Current: D
All four neighbors are visited
or out of bounds
--> Backtrack



Current: A
All four neighbors are visited
or out of bounds
--> Backtrack



All done :)

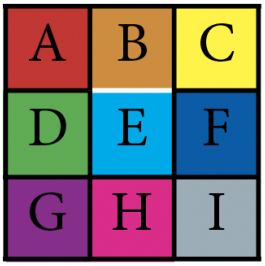
6. **Example: Kruskal**

Kruskal's algorithm loops over every possible passage in random order. It checks if the two cells that would be bridged by the passage are already connected, if not it adds the passage to the maze. The difficulty comes in keeping track of which cells are already connected.

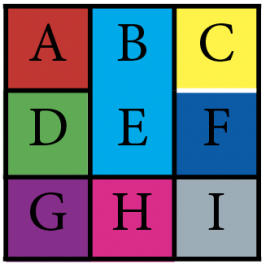
Below is an example of Kruskal creating a 3x3 maze. For the purposes of this demonstration, the 9 cells are labeled with letters A-I. The various connected components (initially each cell is only connected to itself) are highlighted in different colors.

Passages: A-B, A-D, B-C, B-E, C-F, D-E, D-G, E-F, E-H, F-I, G-H, H-I

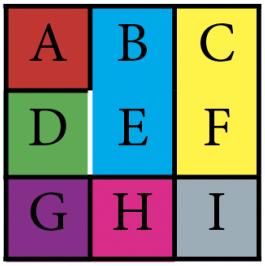
Randomized: B-E, C-F, D-E, A-D, H-I, A-B, E-H, G-H, D-G, E-F, F-I, B-C



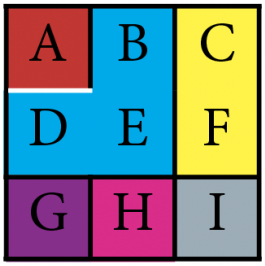
Passage: B-E
Already connected: No
--> Carve passage B-E



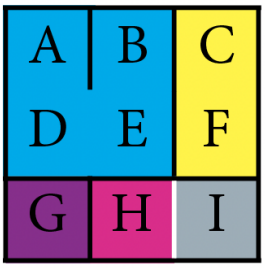
Passage: C-F
Already connected: No
--> Carve passage C-F



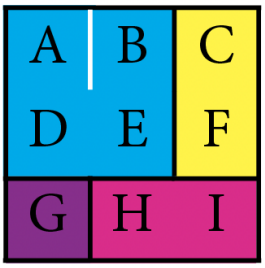
Passage: D-E
Already connected: No
--> Carve passage D-E



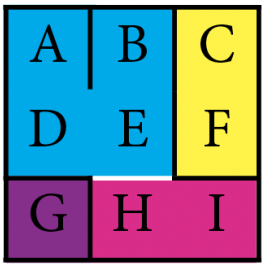
Passage: A-D
Already connected: No
--> Carve passage A-D



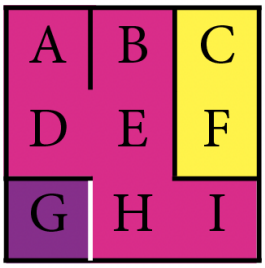
Passage: H-I
Already connected: No
--> Carve passage H-I



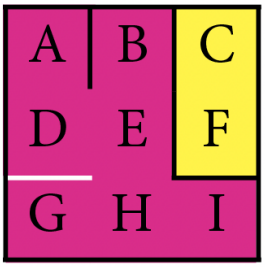
Passage: A-B
Already connected: Yes
--> Do nothing



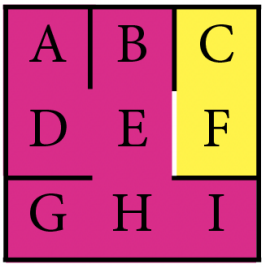
Passage: E-H
Already connected: No
--> Carve passage E-H



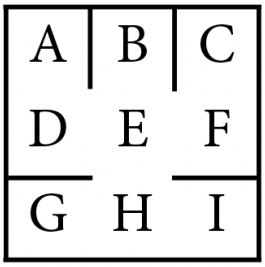
Passage: G-H
Already connected: No
--> Carve passage G-H



Passage: D-G
Already connected: Yes
--> Do nothing



Passage: E-F
Already connected: No
--> Carve passage E-F



Every cell is connected
All done :)

7. **Where to start looking**

Some of these algorithms are well known, others less so. The first two sources mentioned below are some of the best you will find for most of the algorithms. Also, be careful to avoid any sources with code! If there is code, try to avoid looking at it (ideally it is in another language so you won't be tempted)

You must cite all sources, even the ones below!

- The Wikipedia page for maze generation algorithms [**Click Here**] is a FANTASTIC source. It has descriptive pseudocode for most of the algorithms mentioned above, without any actual code (which you should not look at). It also has a bunch of animations showing how each algorithm generates a maze (the ones for DFS, Prim's, and Kruskal's are particularly useful).

- A website called Buckblog [**Click Here**] is an extremely common source for 112 students learning about maze algorithms. It has very detailed descriptions and animations for all of the algorithms mentioned above (plus a bunch of others if you dig further into the website). There is code, but it is in a completely different language and should look absolutely nothing like your code, so don't bother looking at it. For many of the algorithms, this is one of the best sources out there.

- 112 typically has a TA-led mini lecture on graphs and graph algorithms (a category which includes maze generation). Graphs are a datastructure that are very useful for writing maze generation algorithms, so attending that lecture / watching the recording is a good idea. PS: representing your maze as a graph is also wise if you want to use any pathfinding algorithms, since those algorithms also involve graphs.

- Several universities (including CMU) have course notes you can find online that explain several of these algorithms (mostly Prim and Kruskal). These course notes usually cover those algorithms in the context they were intended: finding minimum spanning trees.

- Some articles on maze generation: [**Baeldung**] [**Medium**] [**Neocomputer**]
  And some videos: [**8 Algorithms**] [**DFS in Scratch**]