Castle Escape

Can we create a video game-inspired visualization of many of the classic artificial intelligence pathfinding algorithms?

To answer this question, we first need to ask…

Which Pathfinding Algorithms Do We Hope to Model?

Breadth First Search

is an algorithm that explores out from a starting point level-by-level. In other words, tiles at a depth of one will be explored first, then those of depth two, then three, and so on until the destination tile is found.

Depth First Search

is an algorithm that explores out from a starting point as deep as possible before backtracking and trying a different direction.

Iterative Deepening Depth First Search

is an algorithm that runs Depth First Search repeatedly from the starting point, exploring down one direction until a depth limit is reached. With each iteration, the depth limit increases until the destination is found.

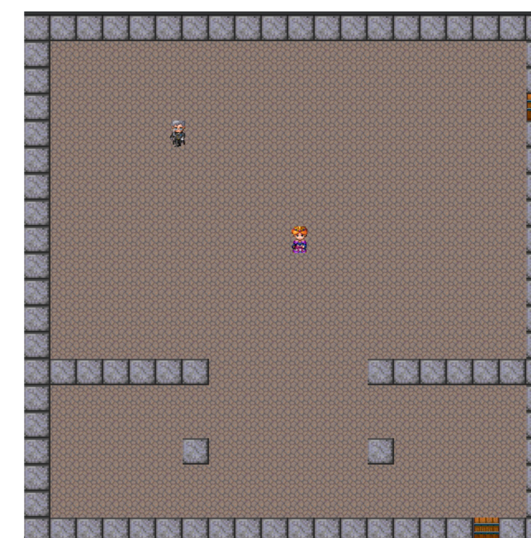
A*

is an algorithm that uses a heuristic to approximate the shortest path. For each cell, the algorithm calculates the distance from the initial tile to the current tile and adds the estimated distance between the current tile and the destination tile. It uses this value to then decide which direction to move in next.

Knowing which algorithms that we hoped to model, we needed to create an engaging platform to implement them onto…

How Can We Implement These Algorithms?

The next steps involved create the skeleton of a video game. Choosing a castle theme, we then created a 20x20 tile grid to represent a game board, on which we created obstacles, walls, and various characters. The main character-- the queen- is controlled by the user and moves along the paths returned by these various algorithms each time the user clicked on the screen.



Implementing the Algorithm

Once the user clicks on the screen, the position of the tile is sent through the code and to the appropriate algorithm. The algorithm then matches the clicked tile to a node. This node represents the destination that the algorithm finds a path to from the starting node.

Selecting the Algorithm

The user must simply press the key representing the algorithm before clicking on the screen in order to switch to using that algorithm to move.

```
def bfs(self, x, y, endX, endY, showPath):
    visited = []
    queue = []
    parents = {}

    curr = self.nodes[x][y]
    queue.append(curr)
    visited.append(curr)

    while queue:
        curr = queue[0]
        queue.pop(0)
        if showPath:
            self.colorSearch(curr, 'red')

        for neighbor in curr.adj_list:
            if neighbor not in visited:
                if showPath:
                    self.colorSearch(neighbor, 'blue')
                visited.append(neighbor)
                queue.append(neighbor)
                parents[neighbor] = curr

            if neighbor == self.nodes[endX][endY]:
                return self.getPath(x, y, endX, endY, parents)
    return[]
```

We Can Visualize the Algorithms… So What?

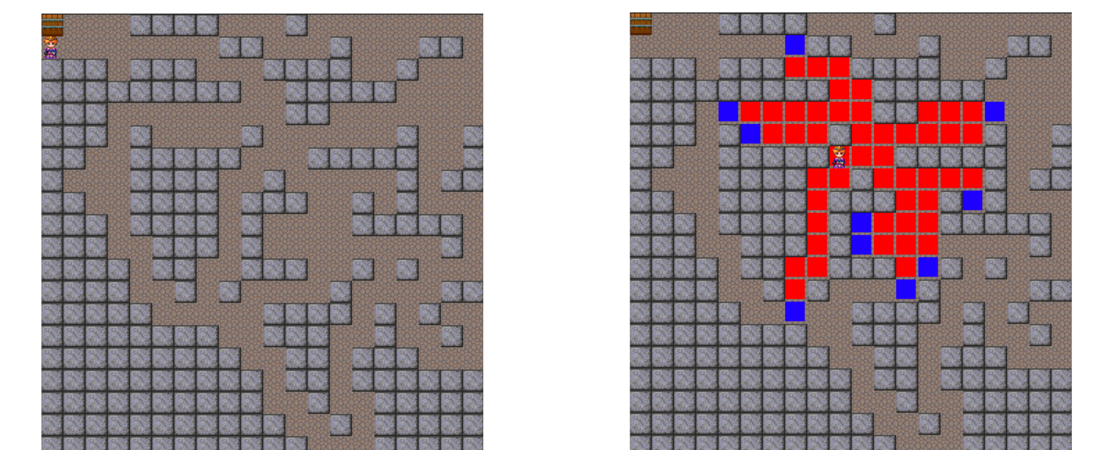
Each algorithm is implemented such that every time a tile is explored (as in… could this tile be along the path?), it is colored blue. For Breadth First Search, we added a red tile to represent the current tile being examined, allowing us to understand where the algorithm would then branch out from. In Iterative Deepening Depth First Search, we added colors to represent each depth level being explored. We then see that each iteration, or call to DFS, is colored differently. Lastly, we denote the final returned path by smaller black squares.

By coloring the tiles in this way, it is clear to see when an algorithm has not been implemented correctly, making debugging code much easier. For example, if tiles to the right of the starting position are continually being colored blue on Breadth First Search, there is likely an error in the code. Additionally, the difference in the behavior between these algorithms appears much more obvious, allowing us to gain a better understanding of which algorithm may be better suited to achieve various goals.

Are there any extensions?

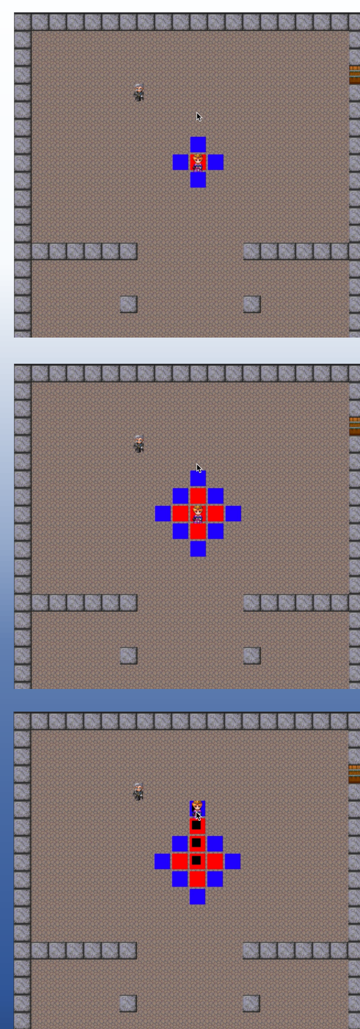
Beyond determining if there is an efficient way to travel from point A to point B, we can also use Depth First Search to randomly generate a room configuration, allowing us to simulate a maze. That is, we can use Depth First Search to find a path from the entrance to the exit, and randomly choose the tiles not along this path to become walls- much like a maze.

Within this maze, we can test out various algorithms to see how they would respond and better understand them. Here, we see how Breadth First Search can find every possible path outwards from a starting position:



But…

How Does This Help Us Visualize the Algorithms?

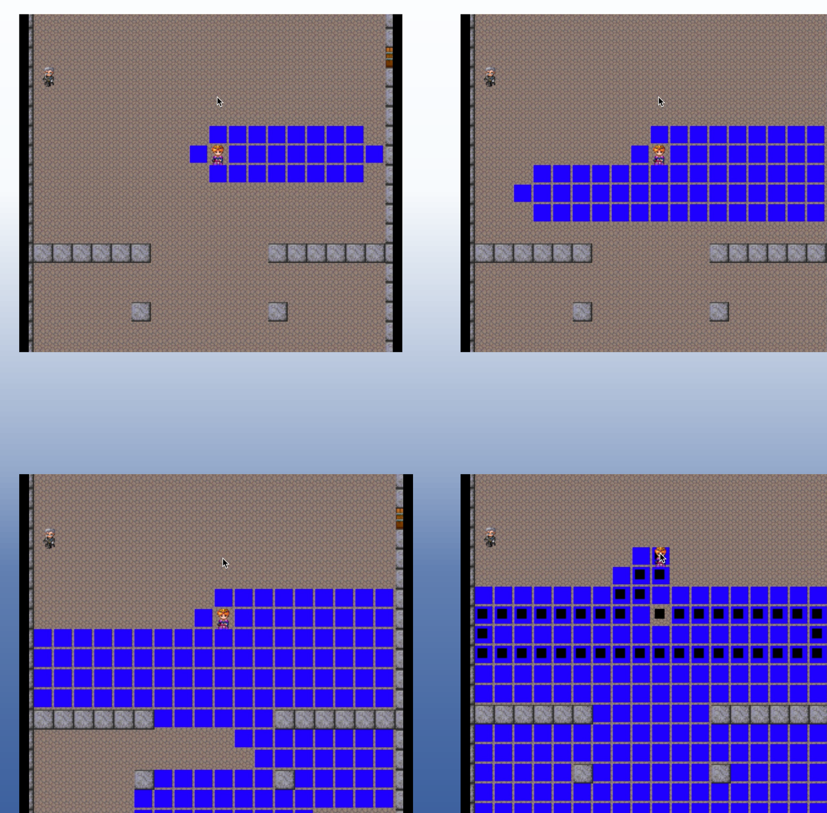


Breadth First Search

We can now clearly see the each direction being explored equally, creating a diamond pattern out from the start position.

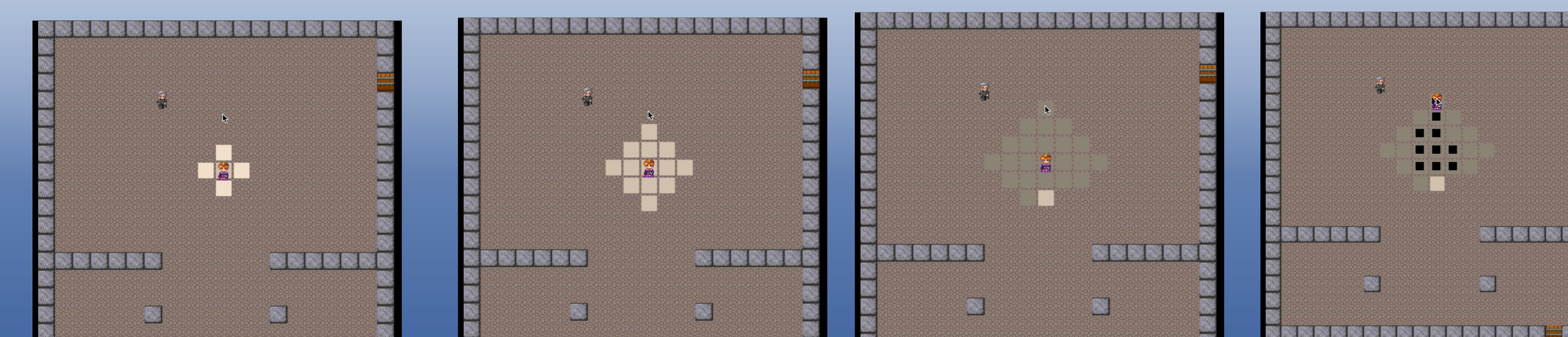
Depth First Search

It is now easier to see the difference between Breadth and Depth first search. Instead of the diamond-like pattern, this algorithm tries to continue in one direction for as far as possible. While Breadth First Search can sometimes find the shortest path, Depth First Search can find a much longer one.



Iterative Deepening Depth First Search

Here, we see how the algorithm behaves as a hybrid between Breadth and Depth First Search, trying to continue in one direction, but until a depth limit is reached. While we see a diamond-like exploration, the path is more like DFS.



A*

Reliable in finding the shortest path, A* explores the smallest amount, continually choosing the direction with the shortest distance from the current tile to the end.