Pathfinding for Evolving Terrain

CSCI 4101 Final-presentation
Group 4 11/08/2023

Eddie Andres Castro | Shijun Jiang | Lenaoria Guerin

Division of Labor

Eddie Andres Castro

- Map creation
- Adding obstacles to the map

Shijun Jiang

- Player Behavior
- Enemy Pathfinding

Lenaoria Guerin

- Timer for shortest path and Obstacles
- Run and Record Time

Research on the A* algorithm & Graph and Analyze results and Runtime

Problem Statement

- Finding the most efficient / shortest path to reach a player as the amount of obstacles change/ increase
- Useful to solve to create more intelligent enemy AI in a game environment that changes

Tools

Pathfinding-for-evolving-terrain / Game /

texturedcookie Added graphing functionality

Name	Last commit message
..	
__pycache__	Grid added
Game.py	Added timer dictionary
Game_reference.py	Add files via upload
Maze.py	Maze positioning update
game2.0.py	reference 2.0
game3.0.py	Grid added
game3.1.py	Added graphing functionality

Used in AI and Games industry
We all agreed on it

ame

- To visually represent the Algorithm

The Algorithm we chose and why

- For our solution we are using the A* algorithm
  - Used in Unity Game Engine for Pathfinding
  - Combines Breadth First Search and Dijkstra's Algorithm
    - Breadth First Search
      - estimates the distance to the goal point
    - Dijkstra's Algorithm
      - the distance from the start point

The A* Algorithm Input

- In the A* algorithm, a graph is taken as input. In the context of a pathfinding algorithm, a graph is a series of locations and connections between them.
- In our implementation of the A* algorithm, the group will be using a grid map, which results in a lot more nodes but is easier to work with and understand.
- In algorithm analysis, we want to see how our algorithm reacts from small inputs to very large inputs, but what defines large and small inputs in a video game?

Map and Obstacles in the A* algorithm

- **In our game, a grid map is used as our input for the A* algorithm.**
  - The map contains...
    - A horizontal row of obstacle blocks set in the center of the map, used to separate the player and enemy
    - The player
    - The enemy
    - White space, which is space that the enemy or player can traversal
- **Rendering**
  - After the under-the-hood implementation of the Maze is finished, we use PyGame to render the Maze graphics into a window.

```
Eddie Andres Castro
def __init__(self, rows, cols):
    self.rows = rows
    self.cols = cols
    self.maze = np.zeros( shape: (rows, cols), dtype=int)
    self.obstacle_length = 1
    self.setup_obstacle((9, 16))

1 usage  Eddie Andres Castro
def setup_obstacle(self, center_pos):
    self.obstacle_center = center_pos
    self.obstacle_left = center_pos[1]
    self.obstacle_right = center_pos[1]
    self.maze[center_pos[0]][center_pos[1]] = 1

# length added to the obstacle
1 usage  Eddie Andres Castro
def add_to_obstacle_length(self):
    self.obstacle_left = self.obstacle_left - 1
    self.obstacle_right = self.obstacle_right + 1
    self.maze[self.obstacle_center[0]][self.obstacle_left] = 1
    self.maze[self.obstacle_center[0]][self.obstacle_right] = 1
    self.obstacle_length += 2


1 usage  Eddie Andres Castro
def get_maze(self):
    return self.maze
```

Contributions

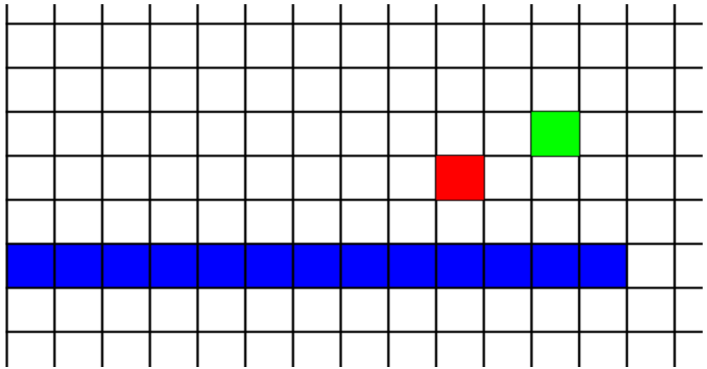
- Separate Maze/Map class for the MazeGame runner class to use.
- Map Grid implementation using an array.
- Sets up the obstacle and adds to the obstacle length.
- Code refinement
  - Improved functionality
    - Bug fixing, program speed, etc...
  - Improved readability
    - Adhering to SWE principles throughout the program.
- Game Loop
  - Properly set up the Game Loop for everything to come together.

A* Algorithm

- A* is an algorithm for finding the shortest path from a start point to an end point in a graph or grid.
- It uses **heuristics** to help prioritize directions that are most likely to be the shortest paths

f(*score*)

g(*score*)

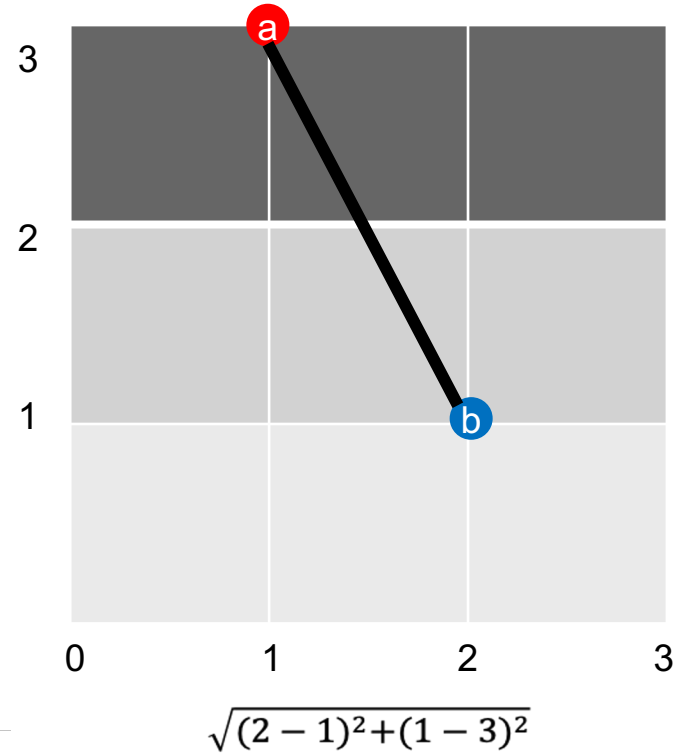


Euclidean distance

We can use **Euclidean distance** to estimate the distance from the current location to the end point.

```
def heuristic(a, b):
    return np.sqrt((b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2)
```

$$d(a,b) = \sqrt{(b_0 - a_0) + (b_1 - a_1)^2}$$



The core of the algorithm of the entire game is to start from the starting point, consider all surrounding nodes, and select the node with the smallest f(n). This process is repeated until the end point is reached or all possible nodes are considered. Implementing the A* algorithm can provide enemies with a clear path guide to the player.

A* algorithm Code Definition

open_set contains all nodes to be evaluated, that is, nodes that have been discovered but have not yet been evaluated.

```
def astar(maze, start, goal):
    closed_set = set()  # Initialize an empty set to store the nodes that have been evaluated
    open_set = [(0, start)]  # Initialize the open set with the starting node
    came_from = {}  # Initialize an empty dictionary to store the most efficient previous step
    g_score = {tuple(position): float('inf') for position in np.ndindex(maze.shape)}
    # Initialize g_score with infinity for each position in the maze
    g_score[start] = 0  # Set the g_score for the start position to 0
    f_score = {tuple(position): float('inf') for position in np.ndindex(maze.shape)}
    # Initialize f_score with infinity for each position in the maze
    f_score[start] = heuristic(start, goal)
    # Set the f_score for the start position using the heuristic function
```

closed_set contains all nodes that have been evaluated. Once a node is taken out of the open_set and processed, it is added to the closed_set

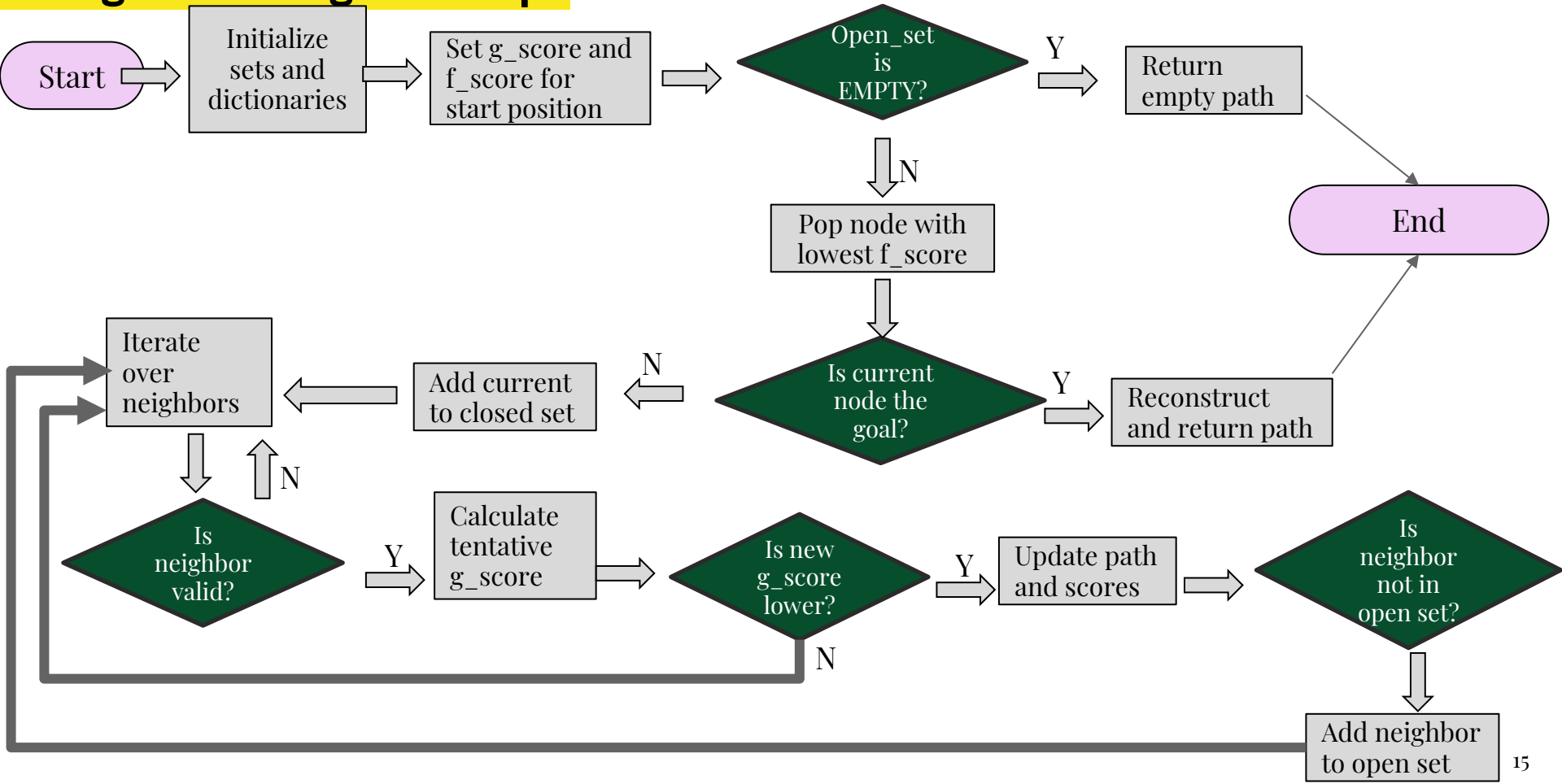
A* algorithm Code

```
while open_set:  # Start the main loop of the A* algorithm
    current_f_score, current = heappop(
        open_set)  # Remove and return the node with the lowest f_score from the priority queue
    if current == goal:  # If the current node is the goal, we've found the path
        return reconstruct_path(came_from, current)  # Return the path from start to goal

    closed_set.add(current)  # Add the current node to the set of nodes already evaluated

    # Iterate over the direct neighbors of the current node (up, down, left, right)
```

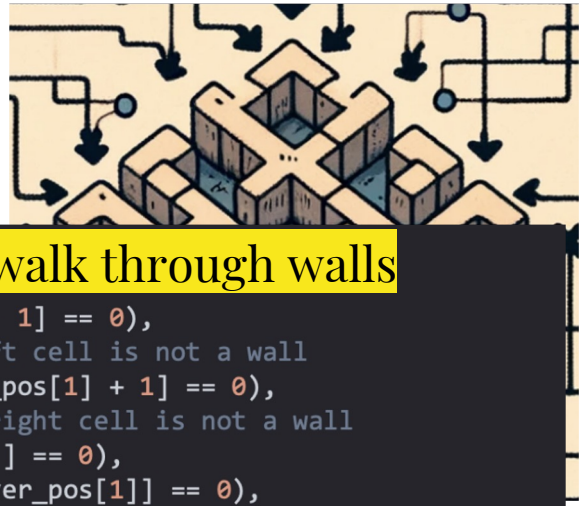
```
def reconstruct_path(came_from, current):
    total_path = [current]  # Initialize the path with the goal node
    while current in came_from:
        # As long as the current node is in the came_from map, which records where we came from for each node...
        current = came_from[current]  # update the current node to the one we came from
        total_path.append(current)  # and add it to the path

        # Once we've reached the start node, it won't be in the came_from map, and the loop will end.
        # We then return the reversed path, starting from the start node and ending at the goal node.
    return total_path[::-1]  # Reverse the path to get the correct order from start to goal
```

```
                f_score[neighbor] = tentative_g_score + heuristic(neighbor, goal)
                # Update the f_score for the neighbor (g_score plus heuristic)

                if neighbor not in open_set:  # If the neighbor is not in the open set, add it
                    heappush(open_set, (f_score[neighbor], neighbor))

    return []  # If the goal cannot be reached, return an empty list
```

A* algorithm Logical Graph



Player Behavior Code



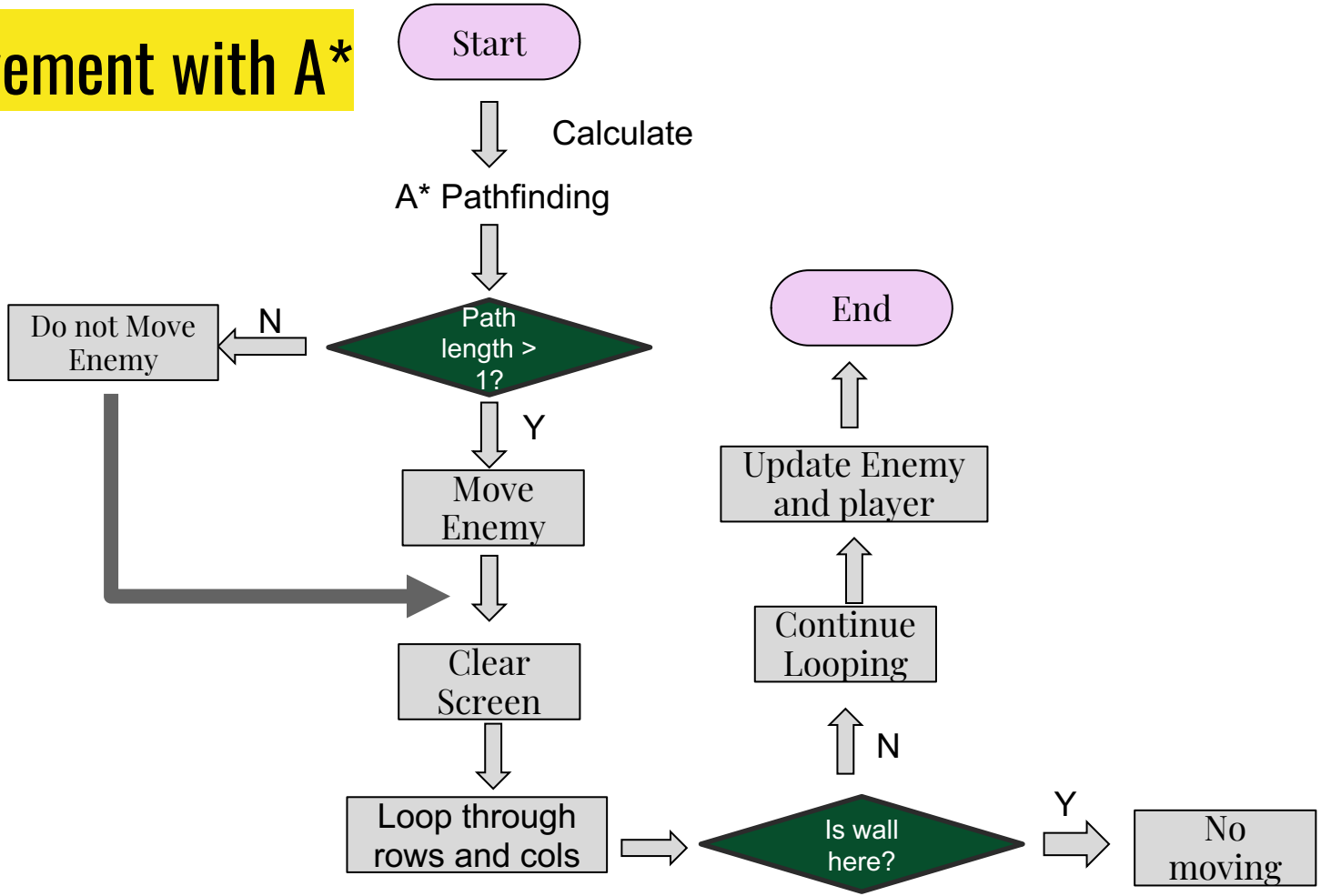
Make sure players can't walk through walls

```
# Random player movement logic
directions = [
    ('left', lambda: player_pos[1] > 0 and maze[player_pos[0]][player_pos[1] - 1] == 0),
    # 'left' move is valid if player is not on the leftmost column and the left cell is not a wall
    ('right', lambda: player_pos[1] < cols - 1 and maze[player_pos[0]][player_pos[1] + 1] == 0),
    # 'right' move is valid if player is not on the rightmost column and the right cell is not a wall
    ('up', lambda: player_pos[0] > 0 and maze[player_pos[0] - 1][player_pos[1]] == 0),
    ('down', lambda: player_pos[0] < rows - 1 and maze[player_pos[0] + 1][player_pos[1]] == 0),
]
random_direction, condition_fn = random.choice(directions)
if condition_fn():

    if random_direction == 'left':  # Move the player left if the random direction is 'left' and the move is valid
        player_pos[1] -= 1
    elif random_direction == 'right':
        player_pos[1] += 1
    elif random_direction == 'up':  # Move the player up if the random direction is 'up' and the move is valid
        player_pos[0] -= 1
    elif random_direction == 'down':
        player_pos[0] += 1
```

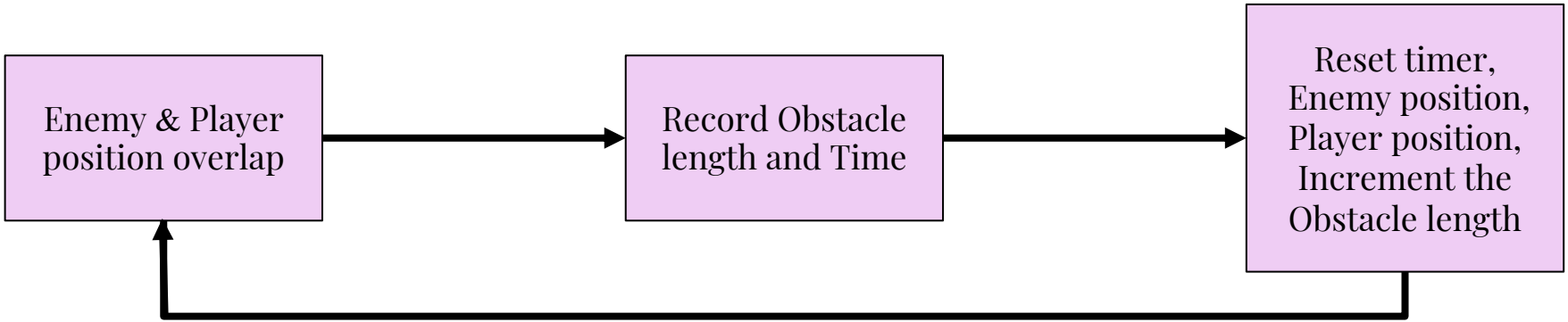
Enemy Movement with A* Code

```
124 def enemy_movement_logic():
125     # Enemy A* movement logic
126     path = astar(maze, tuple(enemy_pos),
127                  tuple(player_pos))  # Calculate the path from the enemy to the player using A*
128     if len(path) > 1:
129         enemy_pos[0], enemy_pos[1] = path[1]  # Move the enemy to the next step in the path
130
131
132 def render():
133     screen.fill(WHITE)  # Clear the screen with a white background? There have some bugs here...
134     for row in range(rows):  # Loop through each row
135         for col in range(cols):  # Loop through each cols
136             if maze[row][col] == 1:  # If there's a wall at this position
137                 pygame.draw.rect(screen, BLUE,
138                                  (col * tile_size[0], row * tile_size[1], tile_size[0], tile_size[1]))
139                 # Draw a blue rectangle (wall) on the screen at the corresponding position
140
141     draw_grid()
142
143     pygame.draw.rect(screen, GREEN, (player_pos[1] * tile_size[0], player_pos[0] * tile_size[1], tile_size[0],
144                                      tile_size[1]))  # green means player
145     pygame.draw.rect(screen, RED, (
146         enemy_pos[1] * tile_size[0], enemy_pos[0] * tile_size[1], tile_size[0],
147         tile_size[1]))  # red means enemy
148     pygame.display.flip()  # Update the full display surface to the screen
```

Enemy Movement with A*



Recording input and times



Loops until it reaches the max input size/ Obstacle length

Demo- Console Output

```
pygame 2.5.2 (SDL 2.28.3, Python 3.11.2)
Hello from the pygame community. https://www.pygame.org/contribute.html
2023-11-06 20:24:25.492 Python[12026:6689384] WARNING: Secure coding is not enabled for restorable state! Enable secure coding by implementing
    NSApplicationDelegate.applicationSupportsSecureRestorableState: and returning YES.
{1: 8333}
{1: 8333, 3: 1167}
{1: 8333, 3: 1167, 5: 750}
{1: 8333, 3: 1167, 5: 750, 7: 958}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125, 19: 750}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125, 19: 750, 21: 500}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125, 19: 750, 21: 500, 23: 833}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125, 19: 750, 21: 500, 23: 833, 25: 583}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125, 19: 750, 21: 500, 23: 833, 25: 583, 27: 1000}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125, 19: 750, 21: 500, 23: 833, 25: 583, 27: 1000, 29: 708}
{1: 8333, 3: 1167, 5: 750, 7: 958, 9: 625, 11: 875, 13: 541, 15: 750, 17: 1125, 19: 750, 21: 500, 23: 833, 25: 583, 27: 1000, 29: 708, 31: 958}
An error occurred: index 32 is out of bounds for axis 0 with size 32
```

Factors to consider

Obstacle

- Obstacle size range from 1 to 31 and increments by 2 (N)

Random Player Movement

- O(1) because they don't depend on the maze size or the length of the path.

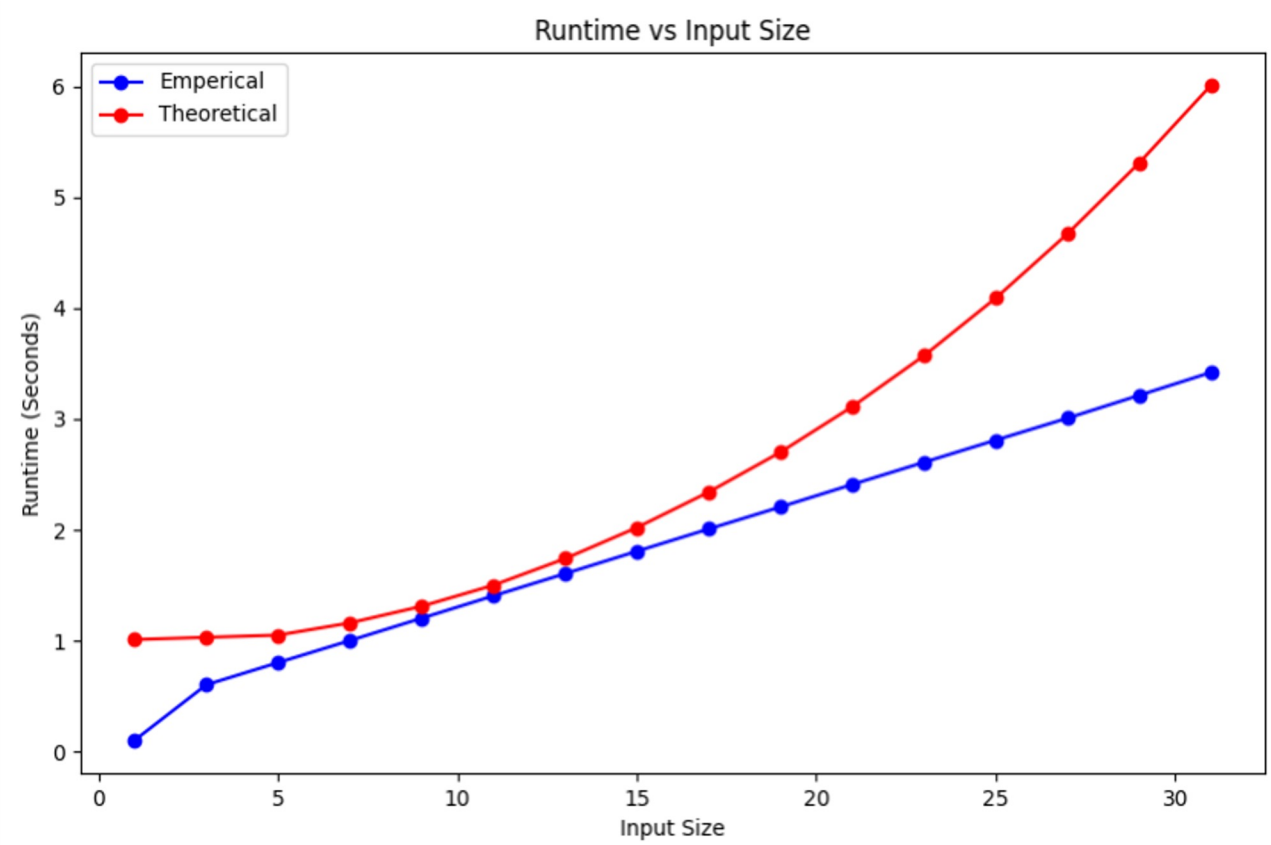
Theoretical runtime

In the worst case, A* may expand all possible states, resulting in a time complexity of $\mathbf{O(b^d)}$, where:

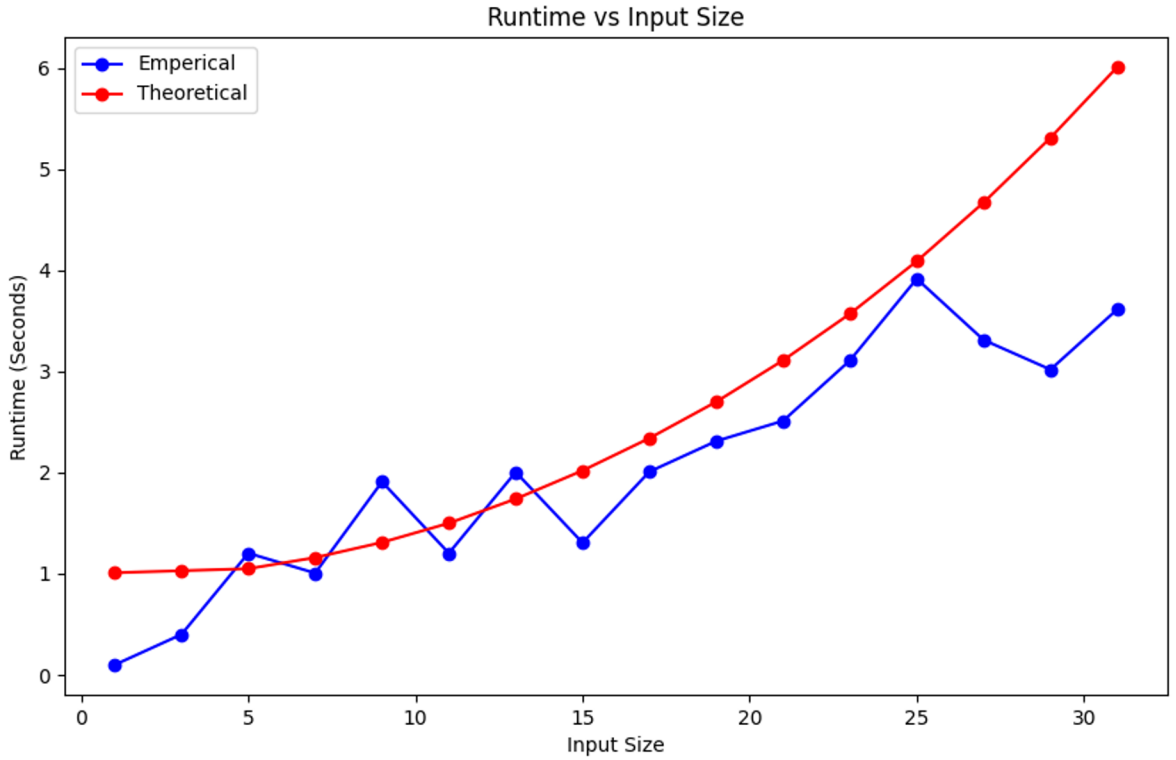
- **b** is the branching factor, representing the maximum number of neighbors each node can have (in this case, **4** for up, down, left, and right moves).

- **d** is the depth of the search, which can be thought of as the maximum number of cells in the path from the enemy to the player

No Player Movement Graph



Random Player Movement Graph



Dead ends, limitations...

- Max input size dependent on screen resolution
- Performance Considerations
- Limited Obstacle Evolution
- Limited Enemy Movement (cannot travel diagonally)

Demo