

Dungeon Generator

Dhruv Daiya

Introduction

Idea: To create a program that can use procedural generation to create dungeon maps, such that every dungeon generated is unique, and usable in the context of board games(such as D&D, Pathfinder, etc.) or video games.

Inspiration: Some of my favorite video games utilize this technique to generate levels and worlds, making each player's experience unique. Specifically, my inspiration is from Hades, Hades 2 and Minecraft.



Project Timeline

1. Setup 2D Tiled Environment
2. Drunken Walker Randomness
3. Path Finder
4. Cellular Automata
5. Drunken Miner
6. Guided Tipsy Miner
7. Flood Fill Miner
8. Smoothing

The Environment

Wrote a Python Script using Pygame to render a 2D integer array as a blank grid.

A screenshot of a code editor showing a Python script named `main.py`. The script uses Pygame to initialize a screen and draw a grid. The code editor also shows a terminal window below it displaying the command prompt and the output of running the script.

```
PCG PROJECT
> __pycache__
> venv
generator.py
main.py 1
requirements.txt

main.py > ...
20 pygame.init()
21 screen = pygame.display.set_mode((1000, 500))
22 pygame.display.set_caption("AI Dungeon Master - Milestone 1")
23 clock = pygame.time.Clock()
24
25 # ---#
26 # Gen#
27 # We#
28
29 #grid#
30
31 grid_#
32
33 # ---#
34 def d(#
35   "#
36   f#
37
38
39
40
41
42
43

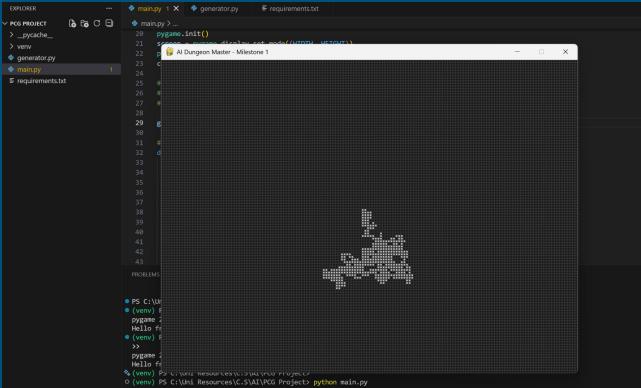
PROBLEMS 1

● PS C:\Uni Re...
● (venv) PS C:
pygame 2.6.1
Hello from t
❶ (venv) PS C:
>>
pygame 2.6.1
Hello from t
```

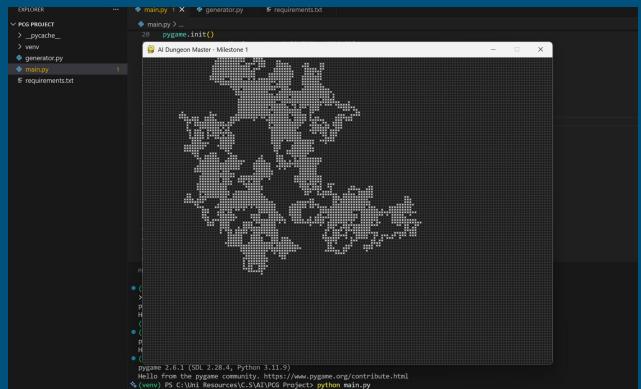
The Drunken Walker

Implemented a random walk algorithm to generate a playable dungeon map. The Drunken Walker starts in the center of the grid and moves in a random direction for a set number of iterations, this creates a playable dungeon that is close to being random.

1000 iterations



10000 iterations



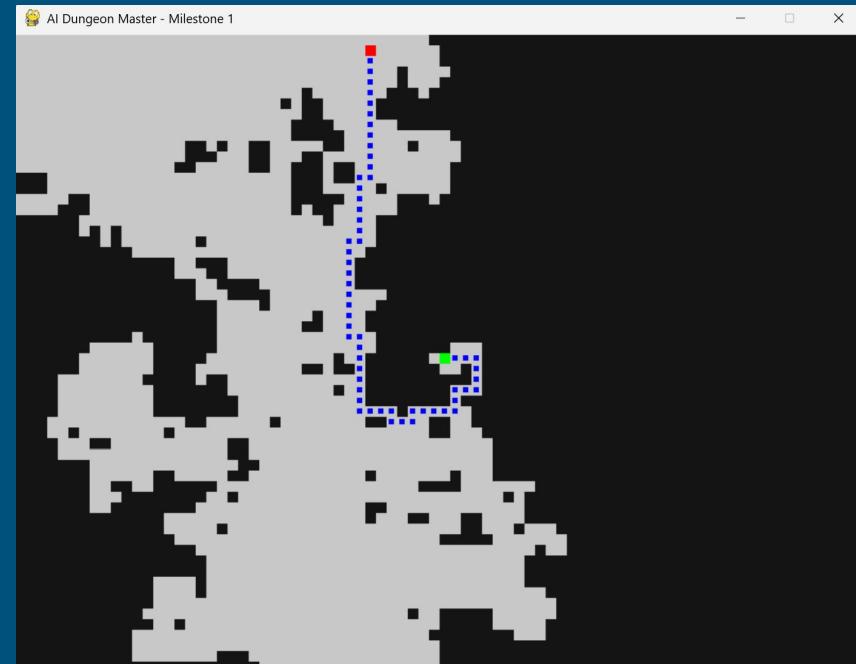
Why it's kinda bad

The quality of maps created by the drunken walker is inconsistent and the randomness can't be enforced by many constraints. It is also inefficient since it can and does visit the same tiles multiple times. This leads to wide fields or narrow corridors at random without giving us much control over it.

```
generator.py > ...
1 import random
2
3 def generate_drunkender...
4     """
5         Generates a map
6         0 = Wall
7         1 = Floor
8         ...
9     # 1. Initialize
10    grid = [[0 for ...
11
12    # 2. Start the
13    r, c = rows // ...
14    grid[r][c] = 1
15
16    # 3. The Walk Lo...
17    for i in range(...
18        # Pick a ran...
19        # format: (<...
20        directions...
21        dr, dc = ran...
22
23        # Calculate
24        new_r, new...
25
26    PROBLEMS 1 OUTPUT DEBUG
27
28  (venv) PS C:\Users\Resh...
29  info: please complete auth...
30  Enumerating objects: 6, done.
31  Counting objects: (6/6)
32  Delta compression using ...
33  Compressing objects: 100% (
34  Writing objects: 100% (6/6)
35  Total 6 (delta 0), reused 6
36  To https://github.com/Dhruv...
```

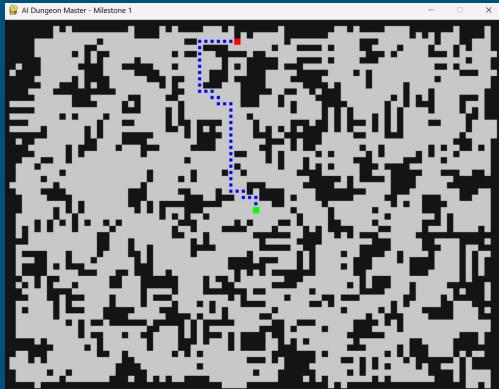
Path Finder

We need to make sure that whatever dungeon we create is solvable, and we need an appropriate search algorithm to determine whether a dungeon is solvable. I decided to use the A* algorithm since it is relatively efficient and it not only answers whether a path exists, but it also provides the shortest path. The Start point is chosen as the center of the grid and the end point is generated randomly from among the traversed tiles.



The Better Way - Game of Life

Cellular Automata, or **Game of Life**, is a much better way to generate dungeons since it is a probabilistic method that works across the entire map without have a specific starting point, it is still random generation, and the parameters can be tweaked to control the density of the dungeon.



Still A Problem

The astute observers would have noticed that when we try to make a sparse cave system with fewer rooms, it isn't always guaranteed that there will always be a path from the start to the end, as is visible in the image (here is where the search algorithm comes in handy since in some cases it may not be so obvious).



The Drunken Miner

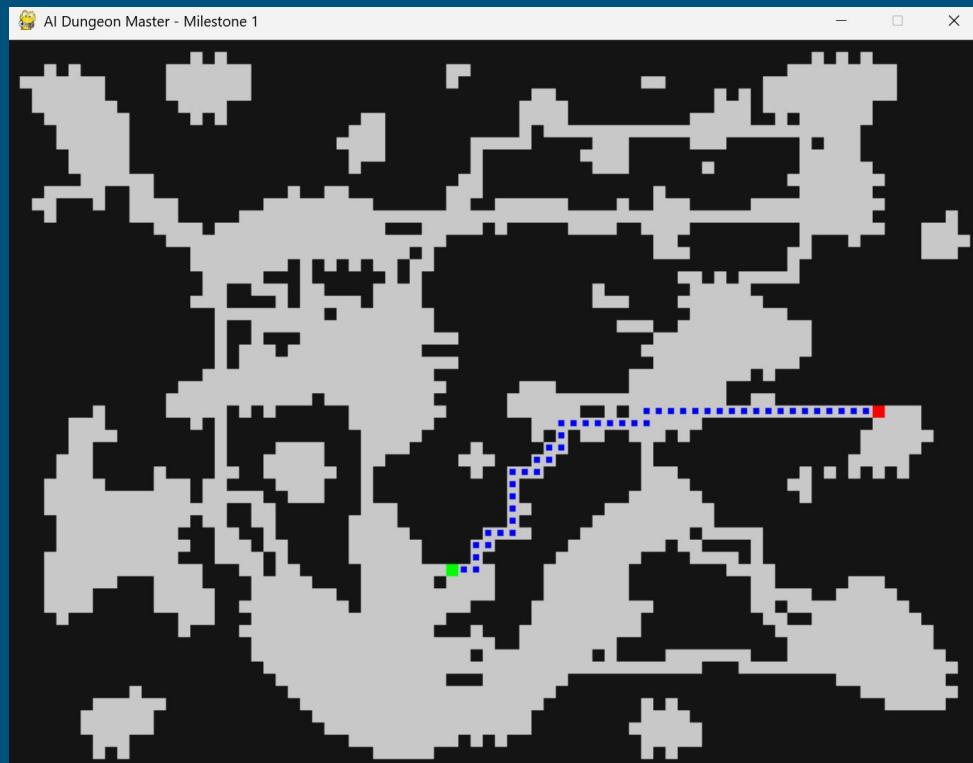
By combining the cellular automata method with our earlier drunken walker method, we can connect the nice looking rooms and ensure that we have a path to the end. We will also be modifying the drunken walker to be not quite as random and we'll also prevent the backtracking.

This is still not the best, because not all rooms are connected, and the mined tunnels look too straight.



The Guided Tipsy Miner

Much better! This maintains the miner's drunkenness but instead of going in random straight lines, the miner targets other unvisited floor tiles in the room and drunkenly makes their way towards them. I also added a check to ensure that there was always a way from the start to the end. There's still some unvisited tiles but it's much better now.



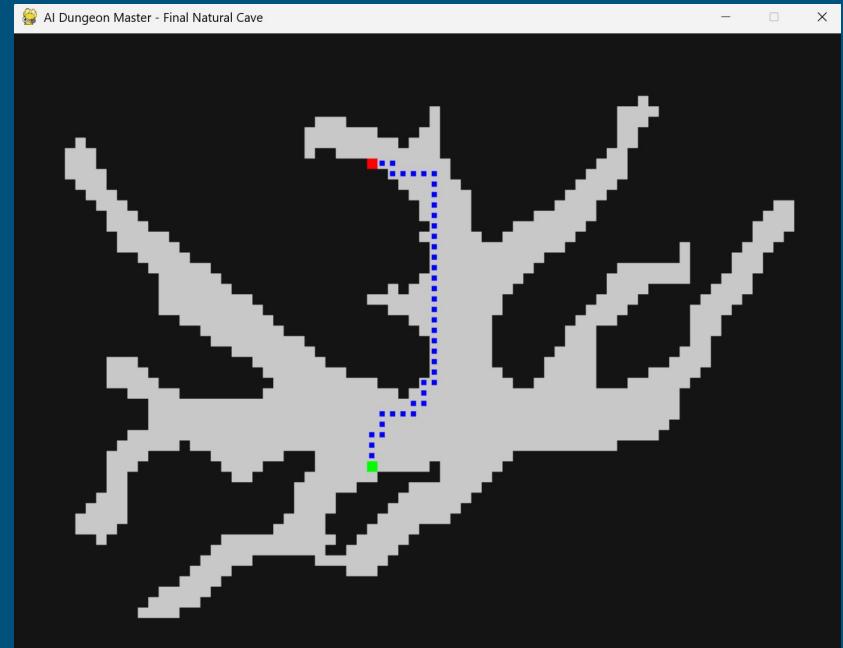
Flood Fill

Changing from the miner algorithm to the Flood fill connected components algorithm, this ensures that every room is connected. However, there's some weird geometric shapes happening now.

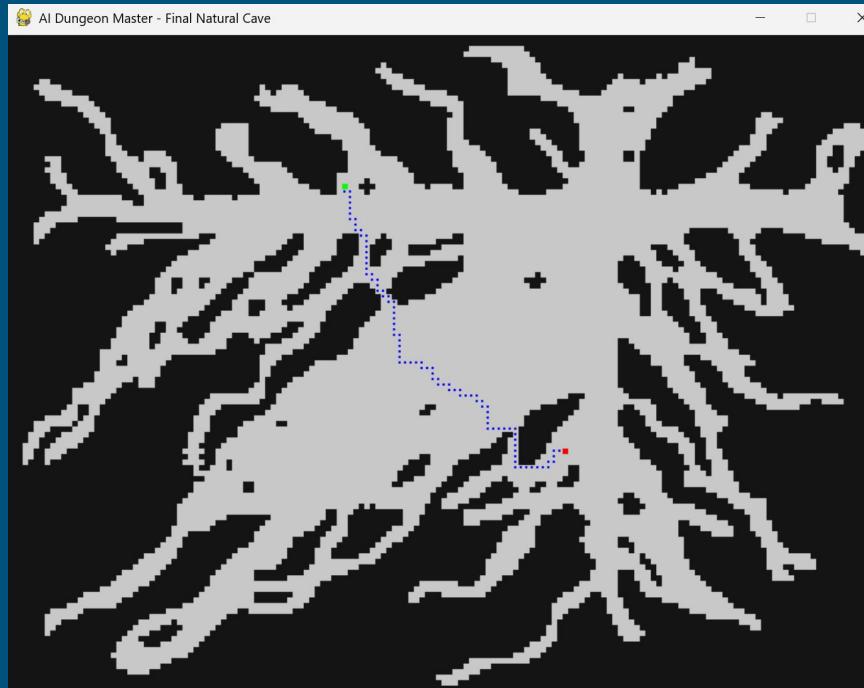


Smooth with The Game of Life!

We can apply the cellular automata smoothing logic once again after the miner's flood fill. This makes those weird geometric shapes go away. Now this was interacting with the miner's tunnels in a weird way and closing off some of the rooms, so I added a marker to the miner's tunnels that prevents cellular automata from interacting with those tiles.



Pretty cool result with more tiles



Just a cool generation with double the dimensions

Thank You!