The
Maze
Game

Harshil Solanki

Contents

1 Lorem **2**

2 Modules Used **2**

3 Directory Structure **2**

4 Running Instructions **3**

5 Features **3**

6 Project Journey **3**

7 Maze Generation Algorithms **4**

7.1 Krsukal's Algorithm 4
7.2 Prism's Algorithm 4
7.3 Recursive Backtracing 4
7.4 Aldous-Broder Algorithm 4
7.5 Wilson's Algorithm 5
7.6 Hunt and Kill Algorithm 5
7.7 Growing Tree Algorithm 5
7.8 Eller's Algorithm 5

1 Lorem

See [2] And See [1]

2 Modules Used

- pygame: The main module in creating this game, it provided all methods required to setup the display and the objects and their attributes which we can use to modify the object as per our choice
- numpy: This modules gives base to this game by allowing it to use an ndarray object that store all the maze information in matrix form
- math: Used in settings module, to compute the score as an exponential function linearly scaled to give us output on given input, that is the distance between the player and the end point
- os: os.path function used in settings module, to access directory at a lower level than the directory given file is in
- sys: used in game_functions.py and end_screen.py, to exit the game
- random: used in settings.py, end_screen.py, builder.py and hunt_and_kill.py to select a random element out of a given sequence
- datetime: used in game_screen.py, to save the score with a timestamp of when the game ended

3 Directory Structure

```
notes.md # Store TODOs and progress and literally anything I wanted to note down for time being
game.py # Running this file starts the game! Contains Initializations and Steps of what to do next based
    on previous output
path.txt # File containing the Directions to be taken Sequentially to Reach to the end and complete the
    level
last_maze.txt # File to store the matrix of the maze that was created in the last game
|
\-modules/
    |--__init__.py # Formality
    |--player.py # Class
    |--settings.py # Class
    |--button.py # Class
    |--camera.py # Class
    |--sprites.py # Class
    |--timer.py # Class
    |--game_functions.py # Functions
    |--menu.py # Functions
    |--end_screen.py # Functions
    |--game_screen.py # Functions
    |
    |--maze_logic/
        |---maze.py # Class
        |---builder.py # Functions
        |---hunt_and_kill.py # Function
        |---random_walk.py # Function
    |
    \-images/
        |--player.bmp # Pegion as our Player :)
        |--blocks[i].jpeg # Flexibility to choose
        |--sky.jpg # Sky is not the limit
        |--nest.png # The Nest is the Goal
    |
    \-the_latex_project/
        |--report.tex # Parent of the pdf you're reading
        |--references.bib # All the references I used, noted
```

4 Running Instructions

Before proceeding to the run this game, you must be aware of the frustated situation of our *dear pegion* who has been caught into a MAZE laid by the *Evil magician* who *conspires to take the home of this little pegion away from it*, essentially leviating it in open air in midst of a vast calm sea (See the paradox, calm sea and evil villian, duh). The Evil has constructed a maze around nest *articulating in front of the world (you mean the world to him ;)) his immense intelligence* and **challenging you in front of the world (yourself, hehe)** to break the his code, solve the maze, reach to your nest and rest in peace. By watching you in peace, the evil will *burn to ashes* and so burns away the maze with it (pegion and nest are fire resistant by nature) and you prove your intelligence against the world (now it means something).
Run the file `game.py` with `python3` and help the pegion (**by controlling it with up, down, right and left arrows**) reach it's nest!
Beware, you must first be considerate of your own intelligence, choose the level wisely. Otherwise you'll have to face depression as you'll be graded realtivelty to the players who've played if before! but if you're my type, and grades don't matter much, enjoy the pegion, the flight and the music!
In case the game is struck and not loading (a random walk error maybe due to overconsumption of memory) interrupt the terminal and re-run the game.

5 Features

Not much of a feature, but you can exit the game anytime you want by pressing ESC :)

Menu Screen

- The game has three levels to select from

Game Screen

- A reverse timer is show on the top of screen showing how much time is left with you to complete the game
- Current score is shown below the timer, essentially based on the distance between the player and the end point, hence, as a hint, giving you an idea of where to move
- Background Music is being played to motivate the player to complete the game
- Collision detection is applied producing ducky-toy squash sound effect on hitting a wall (we have a pegion obsessed with ducks)
- Fading music when Game ends

End Screen

- Interesting music is being played in each outcome of game
- Option of returning to Menu is given

6 Project Journey

–Pending–

7 Maze Generation Algorithms

7.1 Krsukal's Algorithm

[Source]
Kruskal's algorithm is a method for producing a minimal spanning tree from a weighted graph. It works something like this:

1. Throw all of the edges in the graph into a big burlap sack. (Or, you know, a set or something.)
2. Pull out the edge with the lowest weight. If the edge connects two disjoint trees, join the trees. Otherwise, throw that edge away.
3. Repeat until there are no more edges left.

The *Randomized Kruskal's algorithm* just changes the second step, so that instead of pulling out the edge with the lowest weight, you remove an edge from the bag at random. Making that change, the algorithm now produces a fairly convincing maze.

7.2 Prism's Algorithm

The standard version of the algorithm works something like this:

1. Choose an arbitrary vertex from G (the graph), and add it to some (initially empty) set V.
2. Choose the edge with the smallest weight from G, that connects a vertex in V with another vertex not in V.
3. Add that edge to the minimal spanning tree, and the edge's other vertex to V.
4. Repeat steps 2 and 3 until V includes every vertex in G.

And the result is a minimal spanning tree of G. For maze generation, in the second step, we select a random edge instead of edge with lowest weight.

7.3 Recursive Backtracing

Here's the mile-high view of recursive backtracking:

1. Choose a starting point in the field.
2. Randomly choose a wall at that point and carve a passage through to the adjacent cell, but only if the adjacent cell has not been visited yet. This becomes the new current cell.
3. If all adjacent cells have been visited, back up to the last cell that has uncarved walls and repeat.
4. The algorithm ends when the process has backed all the way up to the starting point.

Seems simple enough.

7.4 Aldous-Broder Algorithm

Aldous and Broder were researching these uniform spanning trees, and independently arrived at the following algorithm:

1. Choose a vertex. Any vertex.
2. Choose a connected neighbor of the vertex and travel to it. If the neighbor has not yet been visited, add the traveled edge to the spanning tree.
3. Repeat step 2 until all vertexes have been visited.

Note: this algorithm is notable in that it selects from all possible spanning trees (i.e. mazes) of a given graph (i.e. field) with equal probability. The other algorithms shown don't have this property.

7.5 Wilson's Algorithm

Note: a spanning tree is a tree that connects all the vertices of a graph. A uniform spanning tree (UST) is any one of the possible spanning trees of a graph, selected randomly and with equal probability. The algorithm goes something like this:

1. Choose any vertex at random and add it to the UST.
2. Select any vertex that is not already in the UST and perform a random walk until you encounter a vertex that is in the UST.
3. Add the vertices and edges touched in the random walk to the UST.
4. Repeat 2 and 3 until all vertices have been added to the UST.

So, it's still doing the random walk, but this algorithm converges much more rapidly than Aldous-Broder.

7.6 Hunt and Kill Algorithm

Toda's algorithm is the "hunt-and-kill algorithm". Sounds violent, doesn't it? It's actually quite tame. In a nutshell, it works like this:

1. Choose a starting location.
2. Perform a random walk, carving passages to unvisited neighbors, until the current cell has no unvisited neighbors.
3. Enter "hunt" mode, where you scan the grid looking for an unvisited cell that is adjacent to a visited cell. If found, carve a passage between the two and let the formerly unvisited cell be the new starting location.
4. Repeat steps 2 and 3 until the hunt mode scans the entire grid and finds no unvisited cells.

7.7 Growing Tree Algorithm

A slick algorithm. Here's how it works:

1. Let C be a list of cells, initially empty. Add one cell to C, at random.
2. Choose a cell from C, and carve a passage to any unvisited neighbor of that cell, adding that neighbor to C as well. If there are no unvisited neighbors, remove the cell from C.
3. Repeat step 2 until C is empty.

Pretty straight-forward, really. But the fun lies in how you choose the cells from C, in step 2. If you always choose the newest cell (the one most recently added), you'll get the recursive backtracker. If you always choose a cell at random, you get Prim's. It's remarkably fun to experiment with other ways to choose cells from C.

7.8 Eller's Algorithm

It does this by building the maze one row at a time, using sets to keep track of which columns are ultimately connected. But it never needs to look at more than a single row, and when it finishes, it always produces a perfect maze.

Like the recursive backtracking algorithm, here's the "mile-high" overview of Eller's algorithm:

1. Initialize the cells of the first row to each exist in their own set.
2. Now, randomly join adjacent cells, but only if they are not in the same set. When joining adjacent cells, merge the cells of both sets into a single set, indicating that all cells in both sets are now connected (there is a path that connects any two cells in the set).
3. For each set, randomly create vertical connections downward to the next row. Each remaining set must have at least one vertical connection. The cells in the next row thus connected must share the set of the cell above them.

4. Flesh out the next row by putting any remaining cells into their own sets.
5. Repeat until the last row is reached.
6. For the last row, join all adjacent cells that do not share a set, and omit the vertical connections, and you're done!

My Algorithm

Taking inspiration from the previous algorithms, I'm going to implement my algorithm in this way:

1. Make a random path from start-point to the end-point.
2. Choose two arbitrary points in this path and block the initial path we made between them and generate another random path.
3. Repeat the previous step according to the complexity of maze.
4. Use Hunt and Kill algorithm to verify no cell is univisited.

References

[1] *geeksforgeeks guide for creating start menu in pygame.* URL: https://www.geeksforgeeks.org/creating-start-menu-in-pygame/.

[2] Eric Matthes. *Python Crash Course.* William Pollock, 2016.