Maze generation and solving comparison

A simple program that generates a maze using Kruskal's algorism then solves it with three different algorithms. BFS, DFS and A\*. The main purpose is to compare them together, especially the speed. There are two versions of the code, a GUI version and a CLI version.

## Table of content:

[Table of content: 1](#_Toc567744273)

[1.Maze generation 1](#_Toc196654523)

[Maze diplay 3](#_Toc1382381509)

[Class Tree: 3](#_Toc712938518)

[Edge list 4](#_Toc1972592370)

[Carveing the enrance and exit 6](#_Toc69080949)

[While edges: Kruskal’s loop 7](#_Toc1404697308)

[2.Maze Solutions: 7](#_Toc443149595)

[BFS 7](#_Toc659793711)

[How it works? 7](#_Toc1602975503)

[Code implimentation 8](#_Toc2029479914)

[DFS 8](#_Toc557244097)

[How it works? 8](#_Toc575741829)

[Code implimentation 8](#_Toc1113494963)

[A\* 9](#_Toc253612311)

[How it works? 9](#_Toc1611019068)

[Code implimentation 9](#_Toc1647294236)

[Visualization: 10](#_Toc620849502)

[Terminal: 10](#_Toc2088335696)

[GUI: 11](#_Toc1325683054)

[3.GUI 11](#_Toc538482647)

[4.Conclusion 12](#_Toc1882809754)

**1.Maze generation**

In the maze generation we used kruskal’s algorism (slightly modified). It is the best choice for generating perfect mazes. Which made it more suitaple for our project.

**Notes: (fundmental concepts)**  
  
The **main rule** of the algorism (kruskal’s) is “Only remove a wall if it connects two *different* sets.”

**-Edges are the same as walls.** If a wall between two cells exist then the edge is blocked**.**  
  
**-Perfect maze means:** There is exactly one path between any two points in the maze. Meaning there are no cycles   
  
**-Cycle** means there’s more than one way to get from one point to another — which we don’t want in a perfect maze.  
**Kruskal’s alorism avoids cycles creating one path maze. (There are no loops (no cycles) – you can’t go in circles)**

**-How bit flags work?**

Bit flags are defined as decimal values N, S, E, W = 1, 2, 4, 8.

Their binary values are:

N = 1 # 0001 → North

S = 2 # 0010 → South

E = 4 # 0100 → East

W = 8 # 1000 → West.

Through out the code they are used multiple times both to create the logical maze grid and to visualize it. They are also used in visualizeing the solutions. The way they work is N north means the top wall, south for bottom east for right and west for left. The bitwise operator or “ | “ is used to add them together “ N | E “ means “ 0101 “ also means that this cell is open from top and right. The bitwise operator and ” & ” is used to check if it is open or not so ” cell & W “ if cell is 1000 then the boolean value will be true meaning this cell is open from left. To help understand it better, 0000 means a closed cell and a ” 1111 “ means an open cell. (this are not real cases). The binary values are expreced in their decimal form for ease of writing and understanding the code.

**-ANSI escape codes** are special sequences of characters used to control the appearance of text in the terminal. They can change:

* Text color
* Background color
* Text style (bold, underline, etc.)
* Cursor movement
* Screen clearing

They start with the ESC (escape) character, followed by [ and a command. They are used to display the mazes and solutions in the treminal  
  
Input: -terminal version GUI version explainel later-

the user inputs the width and height which results in the number of cells the user also inputs the seed which uses the built in function random.seed(seed) to ensure the seed will give the same maze eveerytime it is used.   
  
how the built in random.seed(seed) works? it tells Python's random generator to start from a specific position in its sequence, making the output repeatable and predictable.

**So i t works in collaporation with:**

random.**randint**() #entrance and exit

random.**shuffle**() #edges and connecting cells

Constants:  
  
**N S W E values** They are called **bit flags**

The resaon for choosing this values is to ensure that they will not cause any conflict when added together using the or operator

**DX dict**

if you're moving East, then: x increases by 1 (right), y stays the same.

**DY dict**

If you're moving North, then: y decreases by 1 (go up), x stays the same.

Both are used to  
  
**Opposite dict**

Is used to knock the wall in the neighbour cell. The path needs to be opened (the walll needs to be removed from the other side)

Maze diplay  
**terminal**  
def display\_maze(grid): -Terminal version-  
The maze takes the logical grid as a paramiter. This \033[H is an ANSI code is used to tell the terminal "go home" it is used so that everytime a change is made the terminal overwrites the old maze. Then, the top border is printed making sure the entrance is open. the top bordere is prionted as a single string. Enumrate is used to determine the y index which is the number of rows and same is done for x and columns. The if condtion is used to visualize the maze creation highlighting the unconnected cells with white background. Then the S and E walls are printed. if (cell == 0) #this check is used so that we stop highlighting outside these specfic cells. Note that all cells have the 0 value in the logical grid when it is first intalized and changes their color when they are connected.

**GUI**

GUI version explainel later in GUI section.

## Class Tree:

**This class is one of the main parts in the algorism as it connects the cells moreover it is responsible for avoiding cycles (core part of kroskal’s alogrism).**

**def init(self):**  
This is used #when the obj is first initalized it sets the parents to mone

**def root(self):**   
This is used to trace the root of the cell untill it reaches the paent if no parent it is the parent. **It is mainly used to find the set/tree the cell belongs to.**

**def connected(self, tree):**   
Check if two cells/nodes are in the same tree or not.it returns a bolean value. **Used to avoid cycles.**

**def connect(self, tree)**:  
This functions joins the two cells/nodes. It modifies the tree structure by assigning parent pointers. **Union operation**   
  
Grid and Sets initiliaztion:

grid = [[0 for \_ in range(width)] for \_ in range(height)]

This creats a 2d array of 0s. This is used to visually display and modify the maze

sets = [[Tree() for \_ in range(width)] for \_ in range(height)]

This is used to logically modify and create the maze

## Edge list

## Carving the entrance and exit

This function is called right after the maze generation starts

It chooses a random cell then opens it’s N wall at the top same at the bottom for the exit.

## While edges: Kruskal’s loop

The loop runs as long as there are still edge tuples in the edge list.

**set1.connect(set2)**

Before connect(): two trees (set1, set2) represent separate maze areas.

After connect(): they are part of the same maze area — a wall was removed to connect them.

**grid[y][x] |= direction**

This is a bitwise operator that adds the values of n w e s together if a cell have the value of 6 it means the E and S values 2 and 4 are added which tells the display function to remove the wall between these two cells visually

**grid[ny][nx] |= OPPOSITE[direction]** #this removes corresponding wall in the neighbour cell because if the cell on the west side of cell 1 is removed the east wall one cell 2 (one its left) should be removed

# **2.Maze Solutions:**

In **search algorithms** like **Breadth-First Search (BFS)**, the **frontier** refers to the **set of nodes that have been discovered but not yet explored**. In BFS When a neighbor is added to the queue, it becomes part of the **frontier**.

As mentioned at the start, the algorism used to generate the maze is kurskal’s algorism which creates perfect algorisms meaning they have one solution it is implimented this way to make it simpler to compare the speed it takes each to find the same path but in different mazes that may have multible solutions the algorithms might behave differently.

## BFS

### How it works?

Breadth First Search BFS works by searching the graph layer by layer. In the maze case it starts from one cell and marks all its neighbours (there is a path to them) as visited then it explores them and so on. Exploer means to check if it is the end\_cell or not. So the layers are the neighbours of the cell. More percisely each layer consists of all reachable cells that are the same distance from the start.

### Code implimentation

The function initalizes a queue more specifically a deque “double-ended queue” this is done to ensuer operation speen of O(1) so it doesn’t affect the comparison.

The advanage it provides is that it allows for enqueue and dequque operations to be done from both sides. It is used to store the cells that should be explored next

The visited set is used to avoid repeation. Parent dict’s purpose is to keep track of the path. The key is neighbor and the value is the current cell.

Then a timer is started.

After that, we add the start cell to both the queue and the visited

The loop starts untill no more is left in the queue it is followed by an if condition to check if the goal is found or not then a for loop that itirates over the loop if the neighbor cell is not yet visited add it to the queue, the visited set and the parent dict.

The algorism returns the path , true if it is found, the lenght of the path the number of total steps taken to find the path and the time it took to find it.

## DFS

### How it works?

Depth First Search (DFS) dives into a graph until it reaches a dead end. In this code, it works by exploring a path until it reaches a wall. When it reaches the wall, it returns to the last cell that still has unexplored neighbors, then it repeats the process. This cell can be called the most recent decision point.

### Code implimentation

Same as the bfs in general but it differes only as it uses a stack insted of the queue/ deque therefor it uses pop() indstead of popleft() minor changes.

--------->Does the stack equalls the list datastructure in python? #####

The algorism returns the path , true if it is found, the lenght of the path the number of total steps taken to find the path and the time it took to find it.

## A\*

### How it works?

A\* algorism takes desion to visit a cell based of the sum of g(n) which is the distance already traveled from the start and h(n) which is a smart estimate of the remaining distance to the goal. The h(n) is knowen as “**Heuristics”.** The sum is called f(n), which is the main factor by which the alorism decides which cell to explore. This approach helps the A\* to find the solution faster. Because it literaly searches for the shortes path (in this case best solution) unlike BFS and DFS that visits and explores every possible cell untill they reach the end cell.

The h(n) / Heuristics can be calculated in different ways each have its use and situation. The one used here is manhattan\_distance. It's suitable for grids where movement is allowed only in 4 directions (up, down, left, right). The manhattan distance is the shortest path from point a to point b (a straight line).

Note that h(n) is different for each cell same as g(n), Therefor each cell has its own f(n) claculated whil the algorism is searching for the shortest path.

The algorism uses two main structeurs the “open list” and the “closed list”.

The open list contains cells that need to be explored and they are sorted (lowest first ) by f(n). New nodes are added as they are discovered. So in simple words it contains the visited (not explored) cells arranged.

The open list is implimented using a heapq “priority queue” it is a sorted queue that allows access to the smallest (min-heap) or largest (max-heap) element. The diference between it and the normal queue is that it orders the elelments Based on **priority** (smallest first). Not based on arrival order FIFO.

The closed list contains already explored cells. Using it, we can avoid re-evaluating cells and mainly it is how we reconstruct the final path.

### Code implimentation

The open list is implimented as heapq (priority q). The heapq is vert simple heapq.heappush(heap, item) the item in a\* is typically a tuple. the heapq sorts by the first item in the tuple .

The closed list is implimented as a set. Due to their dependeancy over hash tables sets are faster the list o(1). It is named explored.

g(n) is stored in a dict it starts with one key start\_cell and the value is 0.

Core part of the algorsim: while the open list is not empty get the cell, then check if this is the end cell or not. If not add it to explored set then, start exploring. For each neighbouring cell skip if it is already explored if not check is this the this is the first time to calculate f(n) to this neighbour OR check if the g\_score in the dict for the neighbour is shorter than the temporary. If neither are true just skip this iteration. If any is true Assign the value of the temporary variable to the neighbour (g\_score value). Calcualte f(n) and h(n) then store the neighbour and its f(n) in the heapq. Finally store the current cell in a dict. The key is the previous chosen cell.

The path is stored in a dict where each cell points to the previous node in the path. Then the path cells are stored in a list. The list is reversed becaue the dict will pass them from the end to teh start.

The algorism returns the path , true if it is found, the lenght of the path the number of total steps taken to find the path and the time it took to find it.

## Visualization:

Note that dfs and a\_star uses same logic for visualization

### Terminal:

A total of four functions are used in the display part.

“**display\_path()”** is used to show the final path only it must run after the bfs function runs because it takes the path returned by it as an argument. It is created this way to avoid repeating code when displaying the DFS and the A star.

“**bfs\_visualize()**” and “display\_search” works together to display the maze while the algorism is searching for the solution. The bfs\_visualize is a copy of the orginal bfs with minor additions. **Which are:**

1. new neighbours list used to display the frontier.

2. two calls for the “display\_search” function display\_search one to show each step of the search the secound call is to show the final maze with the solution. Note that this function is also created in order to work with the other two search algorisms.

The 4th function is “**get\_neighbors()**” it is used to get all connected cells.

**Note**

The “**get\_neighbors()**”, “**display\_search(**)” and “**display\_path()**” are all in a seperate file because they are written to be used by all algorsims.

**Display functions (“display\_path()”, “display\_search ()”):**

These two were used in the CLI version. They use ASCI and ANSI codes to visualize the maze and the search steps in the terminal.

**display\_path()**

It is a copy of the display maze with slight modification. The main difference is the   
“ if (x, y) in path: ” as it only highlights the cells returnded by the search function and lapeled as path. Meanwhile the display maze highlights the cells with 0 value in the logical grid (all cells at first) and changes their color when they are connected.

**display\_search()**

The display search is essentioally the same as display\_path() but it has more paramiters and conditions with different colors. It displays the visited cells with blue, teh current cell with red, the frontier or cells thar are discovered but not yet to be explored with green and finally the path is displayed in yellow at the end.

### GUI:

Only three are used here display path was removed.

**bfs/dfs/a\_star\_visualize\_gui()** impliments the same logic as bfs\_visualize() but it works with **draw\_visualization()** to print/diplay in the main GUI screen. It was used insted of bfs\_visualize(). This is implimented the same way for all three algorisms.

**draw\_visualization()** was used instead of display\_search(). #####

**show\_ bfs/dfs/a\_star \_results\_window()** in the terminal the code prints the search results normally in the GUI version the results are printed on a seperate tkinter window. This is implimented the same way for all three algorisms.

# **3.GUI**

The code uses both tkinter and pygame. Tkinter is used to show the search information and to take input from user. The main screen, the buttons, the maze generation,the search visualization are all implemented using pygame library. Each library was used to do what it can do easier and better.

When the code starts the user is prompted to enter the height, width, seed and delay time in a small screen. The input window is created by **get\_maze\_parameters()** which is a function that creats the window, the text fields and labels along with the submit button. The submit button reads the valuses or uses the defaults. Then, it closes the input window and starts the main code. The values are stored in a dict by submit button and returned normally by the function.

The main window shows up with the maze in the middle and options at the bottom. The maze is drawn by the function **draw\_maze()**. It works by #####

Most of the functions are explained in the visualization part of the algorisms exept for the **show\_all\_results\_window().** It is basically the same but it diplays all at one in case teh user wants to see all the results at the same time.

**draw\_visualization()** diplays on the main screen but it is also explained in the visulization part.

# **4.Conclusion**

The A\* algorism is the smartest and generally it is the fastest and the best solution. The DFS sometimes gives better results in terms of number of steps, but still the a\* will be the better solution most of the time.