# Building a State Space Maze Search in Python

## Problem:

Consider the problem of the Maze state space search where the search starts with a given start state while eating one or more dots or "food pellets." The maze layout will is given in a simple text format, where '%' stands for walls, 'P' for the starting position, and '.' forthe dot(s). All step costs are equal to one.

Here we implement two state space search strategies: Depth-First Search (DFS) Breadth-first Search (BFS), given three input samples of different sizes of mazes to test the program.

## Proposed Solution:

The maze has walls, a start position and an end destination or goal. Our objective is to find a path from the start to the goal. For this, we interpret our maze as a graph instead, where all the available steps are nodes. Our chief aim remains to traverse from our starting node to the end/goal node using:

i. DFS (Depth First Search).
ii. BFS (Breadth First Search).

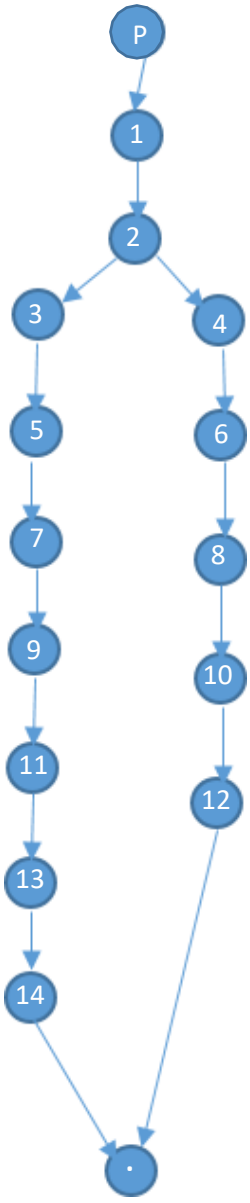
The original input is a text file that looks like (a). In class Maze, we read the file and store it as a list in variable maze[]. From this, we get the height and width of the maze (how many rows and columns are there), and the position of the starting point “P”, the goal “.” and the walls “%”. The “ “ in the maze is what we traverse to reach the end result. (b) being the initial state of the maze, each step we take further results in the formation of an intermediate states, the final state being our required result where a path has been discovered.

```
%%%%%%%
%     P%
% %%% %
% % %
%% %%
%. %%%%
%%%%%%%
```

(a)

| | | | | | | |
|---|---|---|---|---|---|---|
| % | % | % | % | % | % | % |
| % | | | | | P | % |
| % | | % | % | % | | % |
| % | | | % | | | % |
| % | % | | | | % | % |
| % | . | | % | % | % | % |
| % | % | % | % | % | % | % |

(b)




The path available is perceived like this:

Each node contains its state (its position), its parent node (from where have we reached the node) and the possible actions (what nodes can be traversed next) it can take.

A temporary list called check is used to keep track of the node we are to traverse as we move forward.

Initially, the list has just one node, the “start” point.

From here, in iteration we check:

Weather the node at hand is the “goal” we are looking for? If so, we trace back our steps (as we have the parent node for each cell stored) to the start, hence achieving the solution.

Where else can we traverse from here (is there any node up, down, left or right from here) which we have not been to yet (We keep track of all the traversed nodes in a set so that we don’t end up traversing the same track more than once).

All the possible nodes that we can go to next are added to the check list.

(i) In depth first search, a stack data structure is used to store this check list of nodes. It follows a Last In First Out approach, exhausting a path completely down to the very end before checking its counter path.

For e.g., when node 2 is checked, the possible actions, node 4 and 3 are added to the list (since our program checks left and then right). Since, the last node in is checked first, node 3 is removed from the check list and evaluated. From node 3, we can go to node 5 so node 5 is added to the list and since it is the last node added, it will be removed and checked and so on until we reach the solution, ultimately finding a path.

(ii) In breadth first search, we find the best path by using a queue data structure instead. It follows the First In First Out approach, checking parallel nodes in each path simultaneously.
For e.g., when node 2 is checked, the possible actions, node 4 and 3 are added to the list (since our program checks left and then right). Since, the first node in is checked first, node 4 is removed from the check list and evaluated. From node 4, we can go to node 6 so node 6 is added to the list. However, since node 3 was added first, so it will be removed and checked. Then node 5 will be added but as node 6 was added before 5, it will be checked next and so on until we reach the solution. Since we are checking all paths simultaneously, the one reaching the destination first will be the shortest path possible ultimately finding the most optimal solution.

Class Node is a data-structure to store all the traversable steps as nodes. It will store a node’s state, its parent and the action (traversing up, down, left or right).

In class Maze, we read the maze in the __init_() function by taking the filename as input, and evaluate the start, the goal and keep track of the walls(for printing the maze later using pygame).

Class DFS_Stack and BFS_Queue are the two data structures mainly responsible for adding nodes into the check list and popping them out for further evaluation when called by the solve() function.

The solve() function takes input from the user, weather to perform DFS or BFS and deploys one accordingly.
It initializes the stack/queue with the start node.
It accepts a node from the stack/queue calling the pop() function and:

- Runs a goal check. If the node in hand is the goal, it will retrieve the solution by back tracking each node's parent one after the other until it reaches the start position and saves the route. It calls the graphic_rep() function which uses pygame to graphically display the maze. The walls in red, the path /solution in yellow, the start point in green and goal in blue.
- If it is not the goal, it calls the neighbors(state) function which will return the next bunch of nodes that can be traversed from a given point i.e. the current state and passes it to the push() function of the DFS_Stack or BFS_Queue class to be added to the check list.
- It also keeps track of the traversed nodes and only passes the node if it has not been traversed before.

If the check list turns empty before a solution is found, an exception is raised and hence the solution does not exist.
Hence, for execution, we call the solve() function of the Maze class.