# Part -2: Reflection Analysis

## Analysis story -1

At this point you may have created a general-purpose solver. If not, try identify some types of mazes that your algorithm would not solve correctly.

Though my solver is not super-general-purpose, the DFS algorithm does a decent job in exploring all the branches in the maze or in other words, the algorithm I have implemented is a **DFS on an undirected graph.**

### Why did I combine DFS and A\* in the same order (DFS and A\*)?

The real-world problems that contain a deterministic grid (like the maze matrix) are akin to robot path planning problem except,

1. In Maze, we know only the start cell(s). End cells have to be found by search algorithm
2. In robot path planning, we know start and end cells or configurations but not the path.

So,

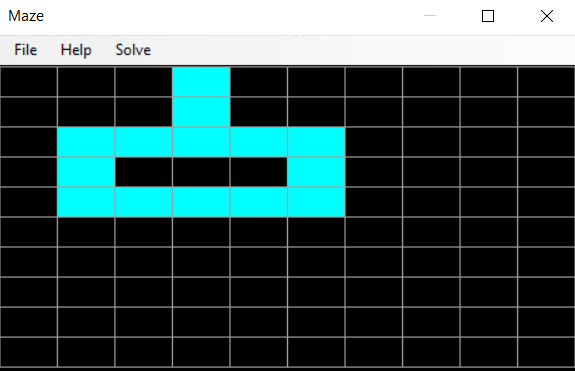
1. I used DFS to find all the exits from the maze.
2. I use Breadth First Traversal to set the H (see below)
3. Use the combination of entry and exit to find the optimal path using A\*.
   1. An evaluated value of a cell = shortest path weight( sum of all edge weights of the shortest path from the given cell to the start of the cell, with each edge weight = 1 (G) AND sum of all the weights from the given cell to the end cell, with each edge weight = 1 again (H)). Eval\_Value at each cell = G + H.
   2. I have implemented non-greedy version of A\*.
   3. To implement the greedy version of A\*, instead of selecting “any” cell from list of neighboring cells that all have the same evaluated value, each of the cell should be considered independently.

### Why undirected graph?

1. Though the maze challenge provided is a tree, I have generalized it to be a graph, because, a typical maze might just be graph. A tree is a special kind of a graph.
2. As any system, that tries to solve, will not know the path from start in a graph, when starting to search, DFS is best suitable. In the challenge provided, the end of the maze can be assumed to the cells that either exist at the last row and/or last column, but I have not considered the existence of the end cells, to actually simulate the unknown-ness of the end of the maze.
3. It is for this reason that I run DFS on the graph, get all the paths that are explored and select only those which ends. My source code returns all the paths and all the solution paths.

### Some special cases of the given challenge

1. A tree can not best represent the following maze.



1. If the graph contains more than one connected component



Though it is not a failure case, the graph is not explored for all possible solutions. The graph can be analyzed for N-connectedness. For smaller graphs, DFS shall be used to find the connectedness. For larger graphs, the multiplicity of “0” eigen value of the Graph Laplacian matrix (Degree matrix – adjacency matrix) gives the N-connected components to know the total n-connected components to start with. The DFS shall still be used to find all the paths.

So, in general, to make the best use of DFS, any un-visited node on the first row (top row), shall be used as a start node and DFS should be run to span all the connected components.

‘To conclude, my algorithm will work for a single connected component. With extra changes to loop through un-visited start nodes (on the first row), the algorithm will give out all the paths.

## Cases that are not considered

* If the maze (graph) contains cycles, it is detected but only one path of the multiple paths that contains the cycle is displayed.
* The A\* is non greedy. It computes only one path which has the least possible sum of each edge weight (each edge weight is assumed **1**). If more than 1 A\* paths are found, only one is returned.
* The graph is modeled to the requirements of a maze solver. Though it could be used as a configuration space for the robot, it is not considered in this version. So, modeling of the obstacles are not considered. Please refer to the last section on this.

## Analysis Story-2:

If you kept things simple, it is likely that your algorithm may not be as efficient as possible. Describe the solution’s complexity and approaches that could be used to optimize it further.

* The complexity for the algorithm that I have implemented is
  + the complexity of running DFS,
  + complexity of running BFS AND
  + complexity of running A\* on a graph.
* For the DFS
  + Time complexity, O (|V| + 2\*|avg no of neighbors of each V|) => avg no of neighbors of each V is the avg no of edges (branching factor) = O(|V| + |E| ). Or O(b^m) where b is the branching factor and m is the maximum depth.
  + I used one stack and two vector of vectors. The space complexity of the stack is the maximum no of nodes it will store. Assuming the graph is a linear list tree, the worst-case space complexity is no of edges or the sum of all the neighbors from start node. The vector stores all the paths and the solution paths. (it’s a vector of vector of nodes).
* For the BFS, the time complexity is O(|V| + |E| ) where O(|E|) may vary between O(1) to O(|V|^2) and the space complexity is O(L) where L is the maximum no off nodes in the single level. ( L is the depth itself if the graph is a linear list).
* I do not store edges explicitly as I assumed the edge weight as 1. The graph is a vertex graph and |V| = no of white cells in the maze.
* Each node has its adjacent nodes and is deemed to be an edge. So, the BFS is used to implicitly traverse all the edges and set the weights (H).
* For A\* algorithm:
  + I used BFS to set heuristic H from target to each node.
  + The BFS to evaluate distance metric from source (G)
  + I use two priority queues namely open and closed
    - The priority queues contain a standard priority queue of C++ and a vector to store all the elements
    - Adding a new element – Log(N).
    - Finding an element – N (worst case)
    - Removing element – N (All the above can improved by using Hash Tables for really larger values of N).

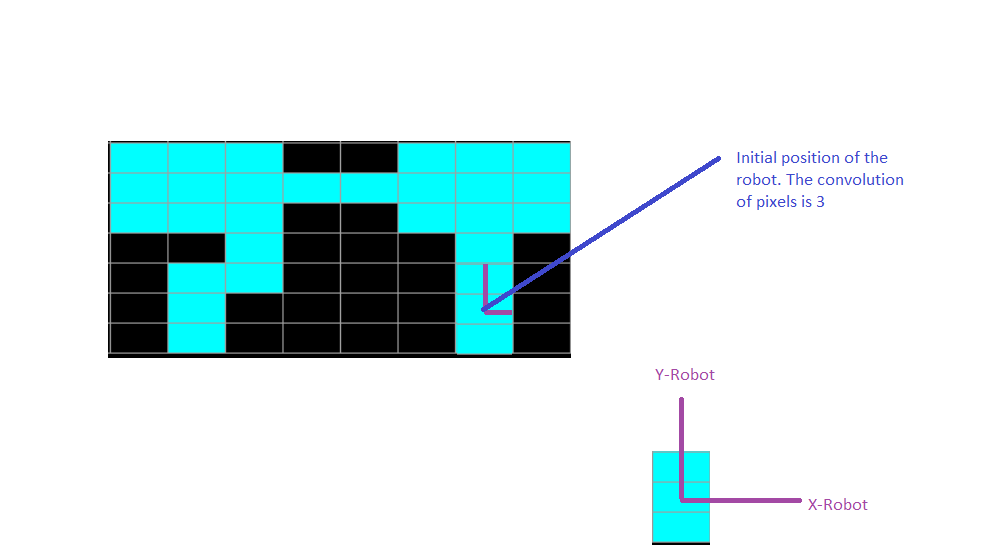
## Analysis Story-3

The story -3 makes us differentiate between the task space where the entire robot movement is viewed and the configuration space or the C-Space where the robot is represented as a point or a cell, with each cell representing a configuration of the robot.

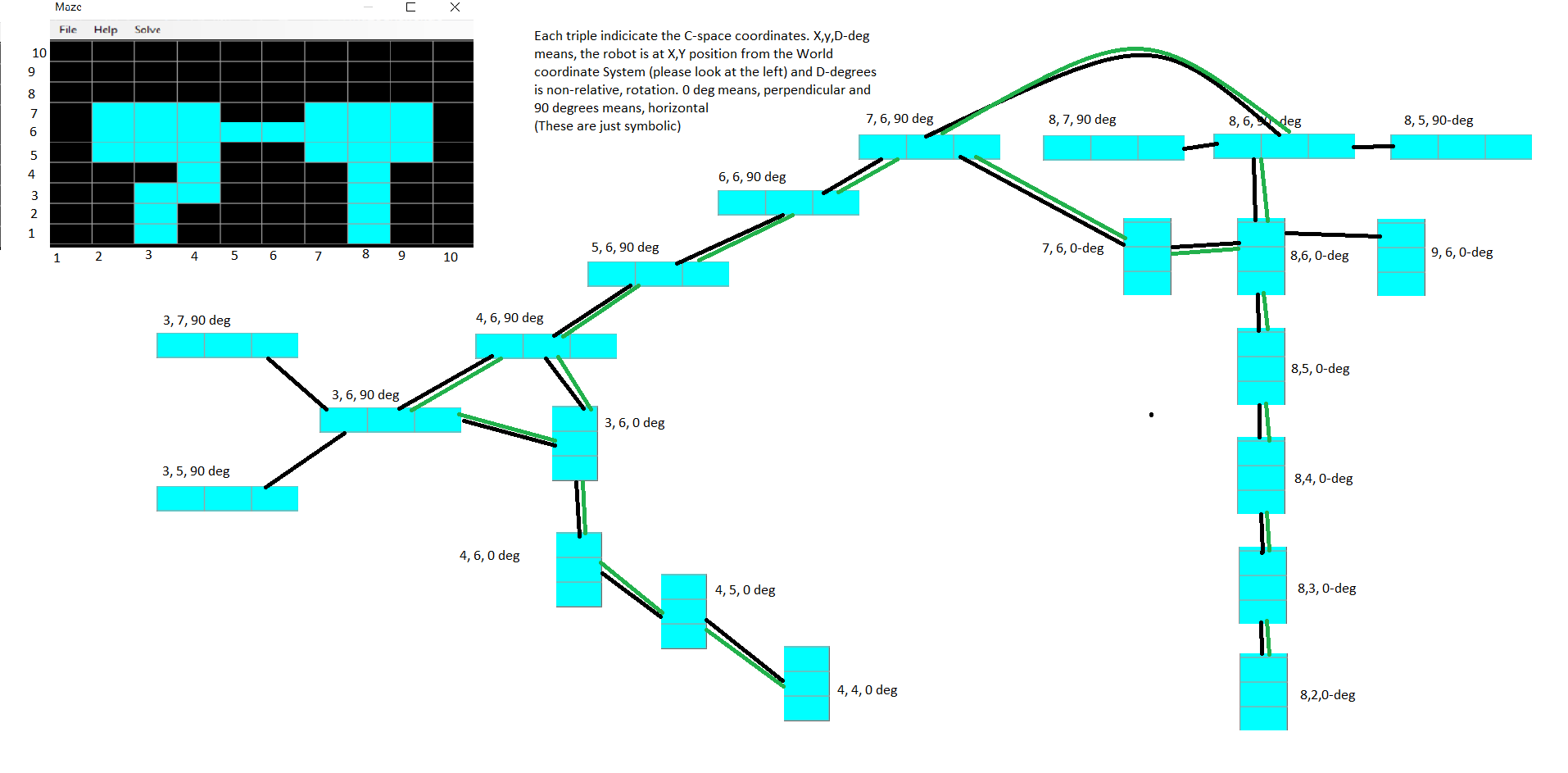
A configuration of a robot is a complete definition of the geometry of the robot with its position and orientation. IN this configuration it may or may not collide with an obstacle. The configuration space just contains cells and not the obstacles themselves rather each configuration is labelled as feasible or infeasibile. A path is the curve that connects two feasible configurations and always pass through a set of feasible points.

In the case, provided, the first step is to convert the robot space or the cartesian space to C-Space as follows. This is generally done using Minkowski sum computation (in theory). In the given case, a convolution of the robot of 3X1 pixels can be made with the cells of the maze. All the 3’s are the cells that mark each configuration of the robot.

The same A\* algorithm implemented can be used as path planning algorithm for a robot. In that case, the cells of the maze represent a configuration of the robot.



I have computed that below. It looks like the graph.



The robot can move from 8,2 to 4,4 in more than 1 paths but **can not reach 3,2. The above graph is not yet a path but a graph of possible configurations, or the C-Space.**

Please refer to the **Robot-C-Space.png** for full information. The orientation in degrees is not a continuous measure but to just inform that the robot being either vertical or horizontal.

## Obstacle avoidance:

The classical problem of robot path planning (not including the entire motion planning that includes trajectory, redundancy and time) is to minimize the length of the path in the C-Space while minimizing the (sum of) shortest distance from the nearest obstacles at each configuration.

IN the given case for the above graph, the same A\* algorithm can be used by additionally calculating the measure of closeness to an obstacle.

For each node in the above graph, one of the measure of closeness from each obstacle M = W\* (Sum\_of\_Edge-weights-from-obstacle-node-to-any-node), which should be added to F+H. In effect

Evaluated cell value = F+H +M.

The weight W controls how serious this infeasible configuration because of obstacle is to be considered. The subtraction of the Evaluated Value makes each cell in the proximity of the in-feasible or obstacle hitting configuration node, FELT.

The A\* algorithm can then plan the path by keeping the in-feasible configuration away from the PATH in the C-Space.

The actual task space path is then the continuous set of the feasible configurations.

In the file attached, it is displayed as black/green edges from one configuration to the other.