1 Introduction

Graphs are quite common data structure used in almost every computation task. Naturally, there are many graph algorithms developed to facilitate interactions with this data structure. This article provides a method of visualising some of these algorithms.
For this purpose we use a maze that can be modelled as a graph. We put a small tank on the top which moves on the maze tracing out the path of the algorithm. We have implemented the visualisation of the DFS algorithm.
The DFS algorithm forms a basis of a large number of graph algorithms. Thus out visualisation tool can be suitably extended to visualise many other algorithms. We provide a theoretical layout for one such algorithm - Bipartite Checking.
The DFS simulator can be found here: Simulator (Private repository as of now)

2 Depth-First-Search Visualisation

2.1 Data Structures

The easiest way to implement a DFS algorithm is to use an adjacency list. The adjacency list contains all the nodes adjacent to a particular node. The adjacency list can be directly generated from the maze, by iterating through all the cells and adding an undirected edge between the connected adjacent cells.
For the purpose of visualisation, we used a queue to store the movements of the tank. This storage is required because the algorithm will be computed too fast for the movement to be visible.

2.2 Algorithm

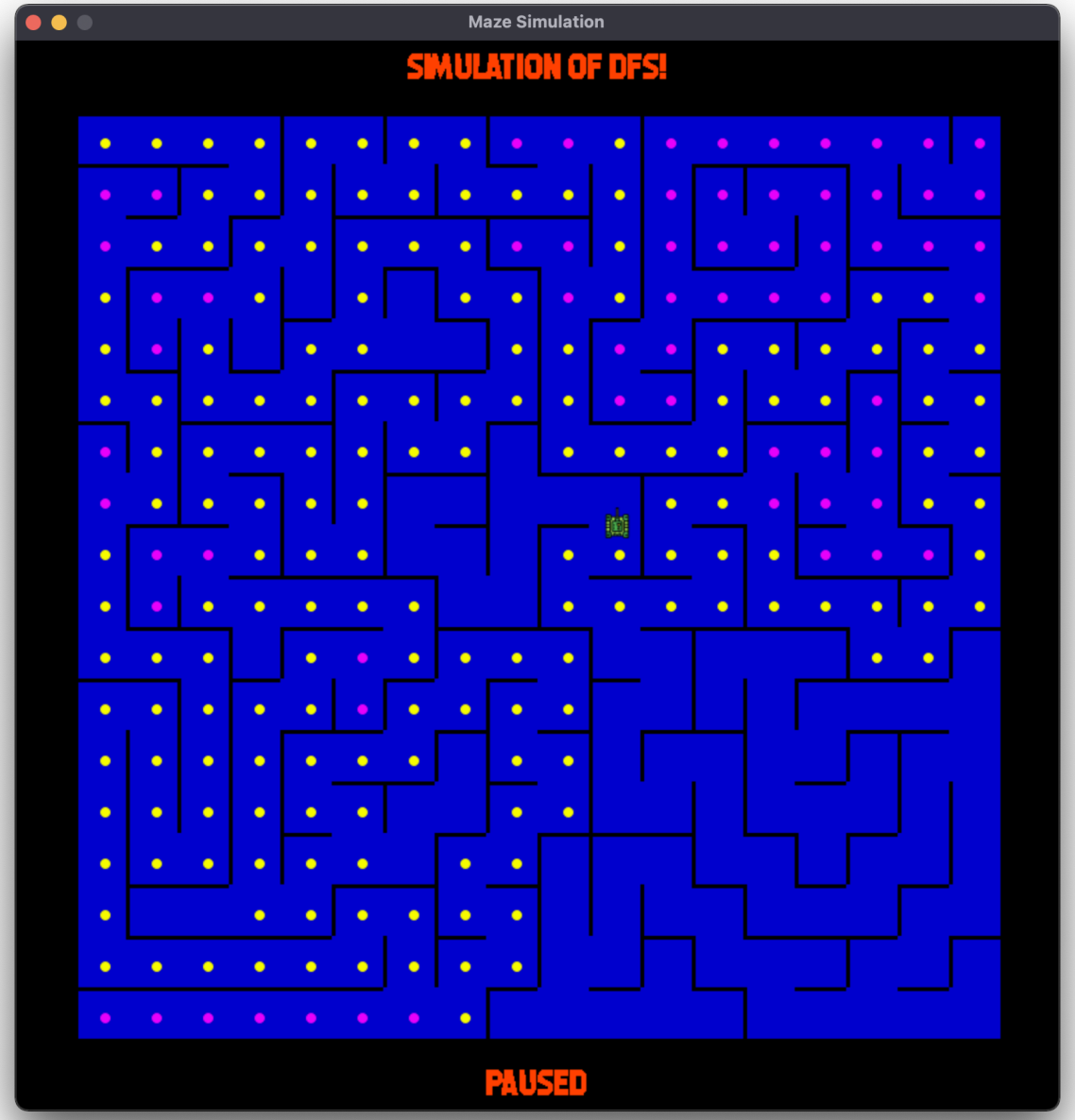
$DFS(i, j)$ {
    color (x, y) $yellow$
    for(all pairs (x, y) in $adj[i][j]$) {
        if color (x, y) is $blue$ then {
        DFS(x,y)
    }
    color (x, y) $pink$
}

Note: All the nodes are coloured blue initially.

2.3 Visualisation

The visualisation is done by placing a tank on the graph and moving it according the algorithm. The position of the tank represents the node which the DFS algorithm is currently visiting.
The main challenge is to move the tank slow enough that the movement is visible. This done by using a queue. We store the movements of the DFS algorithm in the queue. Then the visualiser will pop each element from the queue. Each element will consist of an integer denoting the directions to move (UP/DOWN/LEFT/RIGHT). The tank will start moving in the appropriate direction at a suitable velocity. The next element will be pop-ed only after the movement has completed.
Thus the overall visualisation occurs in two steps, first the DFS dumps the movement into the queue, and then the queue is executed at a viewing speed.

We provide buttons for increasing and decreasing the speed of the tank. For this, we define a move function which moves the the tank be one pixel in appropriate direction. The move function is called the speed number of times in a single loop iteration. We also make arrangements to keep the speed fractional.
Lastly we colour the nodes we visited once with yellow, and the ones visited twice with purple. So at the end, all the nodes are purple.



2.4 Execution

Follow the instructions in Readme.md to setup and start the visualising tool for DFS algorithm.

2.5 An Example Application

DFS algorithm can be used to solve a maze. We can start from the source node and apply DFS. The path from the source to the current node can be stored in a stack. A node is pushed to the stack when the DFS is called on that node, and it is poped when the DFS completes on that node. If we reach the target node during the whole process, the content of the stack at that point gives a path from the source to the target.

3 Bipartite Checking Algorithm

A graph is bipartite if it can be divided into two groups with only cross edges, or equivalently, we should be able two colour all the nodes using two colours with no two adjacent nodes having the same colour. On a maze, this translates to having different colour in all the adjacent positions which can be reached. Bipartite can be easily checked using DFS.

Note: This is not implemented in the repository as of yet. This is a design report.

3.1 Data Structures

This algorithm does not require any additional data structures other than DFS.

3.2 Algorithm

$starting\_cell <- yellow$
$DFS(i,j)$\{
    mark (i,j) yellow
    for(all pairs (x,y) in $adj[i][j]$) {
        if color (x,y) is $blue$ then {
        if $i,j <- yellow$ and $x,y <- yellow$ then $return\ false$
        if $i,j <- pink$ and $x,y <- pink$ then $return\ false$
        $continue$
    }
    if $i,j <- yellow$ then mark x,y as $pink$
    else mark x,y as $yellow$
    DFS(x,y)
    }
}

Note: All the nodes are coloured blue initially. This is a generic pseudo-code of Bipartite checking algorithm and is not meant to be used directly in the simulator.

3.3 Visualisation

The visualiser can be easily built on top of the DFS visualiser. We can store the colour of the last node. Now colour the current node with opposite colour. Finally we check the color of all the adjacent nodes. If there is a node with the same colour, then the graph is not bipartite and we stop. If we can colour all the nodes, then the graph is bipartite.

3.4 An Example Application

Consider two droids and a maze. Droids can capture cells of the maze.Both droids take moves one by one. The motive of first droid is to capture nearest non-captured cell. The motive of second droid is to capture all the adjacent cells of the cell currently captured by first droid. We have to find if the first droid can prevent second droid from having two adjacent captured cells. This problem can be solved using bipartite graphs algorithm. We check if our maze can be made bipartite. The first droid will capture all the yellow marks cells, the second will capture all the pink marked cells. In the end, if the maze if bipartite then first droid wins else the second droid wins.

4 Conclusion

We have implemented a generic DFS visualiser using SDL libraries. This visualiser shows the path traced by DFS algorithm via movement of a tank on a maze. The speed of the tank can be adjusted and the simulation can be paused. We also colour code the nodes that are visited.
The DFS visualiser can be easily extended to simulate a multitude of DFS based graph algorithms like cycle detection, path finding, finding connected components etc. We provide a layout to extend the DFS visualiser to bipartite checking algorithm.