page1image45511024

Complexity & Algorithm

Assignment 3: Backtracking

page1image45507776

Group 47:

Linh Hoang- 467424

Nam Trinh – 464560

Table of Contents

[1. System design 3](#_Toc27422158)

[1.1 Node 3](#_Toc27422159)

[1.2 Edge 3](#_Toc27422160)

[1.3 Maze State 3](#_Toc27422161)

[1.4 Maze puzzle class diagram: 4](#_Toc27422162)

[1.5 Flow chart: 5](#_Toc27422163)

[2. Code explanations: 7](#_Toc27422164)

[2.1 Function find solutions: 7](#_Toc27422165)

[2.2 Function dfs backtrack: 7](#_Toc27422166)

# **1. System design**

Based on the assignment description, the maze puzzle is made of:

+ 23 nodes

+ Edges attached to these nodes

+ Colors

+ 2 Pawn

+ Maze states (current positions of 2 pawn)

## 1.1 Node



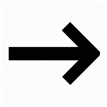
Visual appearance of a node:

A node contains the information of a position in a maze. A node has its color (ex: red, …) and a position number (ex: 1). Pawns can traverse through these nodes.

There is also a goal node, which was implemented as node 23.

* These nodes will be implemented as 23 objects of the nodes class

## 1.2 Edge



Visual appearance of an edge:

An edge contains the information of the ability to travel from 1 node to another. An edge has its color (ex: black, …) and a destination node (ex: Node 1). In short, edge provide the direction.

* These edges will be implemented as list of objects owned by a node.

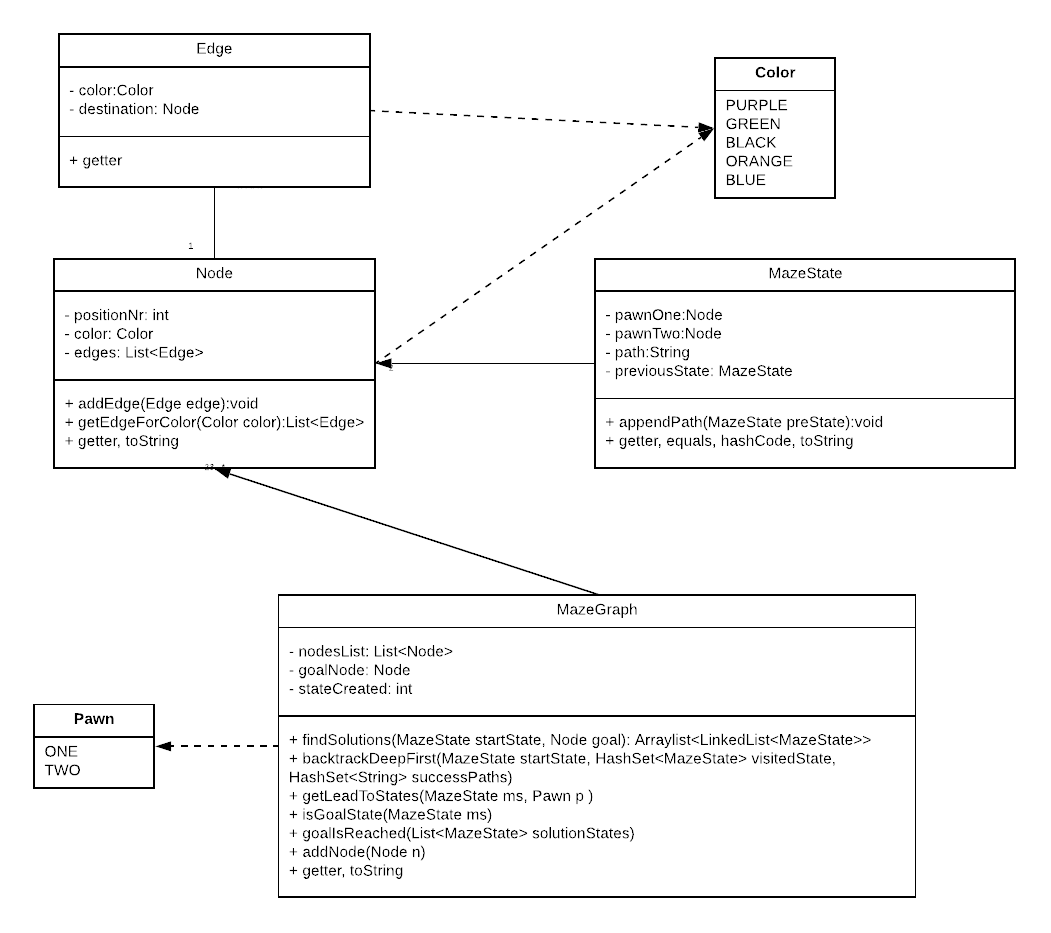
## 1.3 Maze State

A state represents a position of pawn one and pawn two at the same time. For example, when pawn one is at node 1 and pawn two is at node 2, the state would be [1,2].

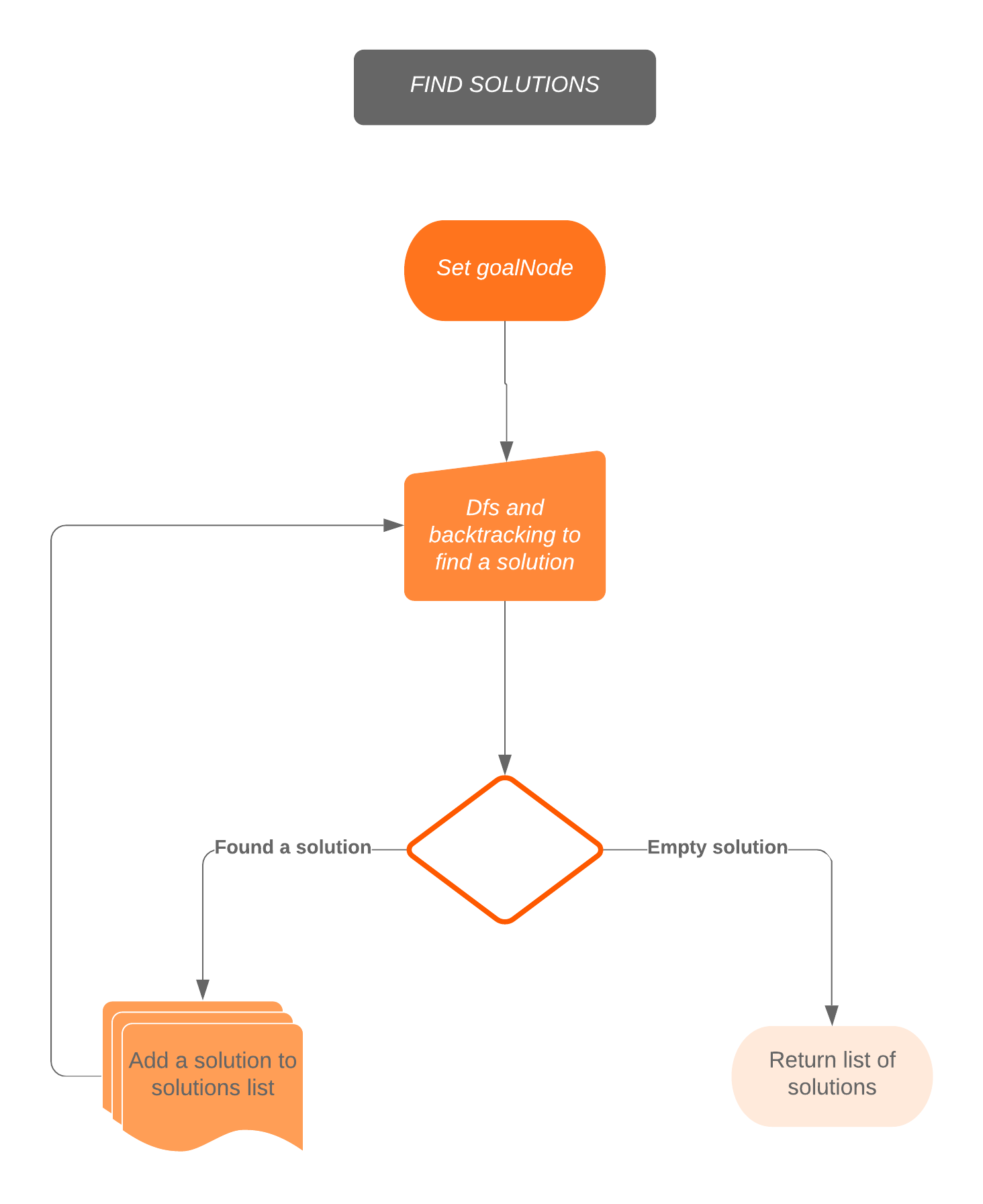
A maze state contains its previous state in order to get the paths to the current state.

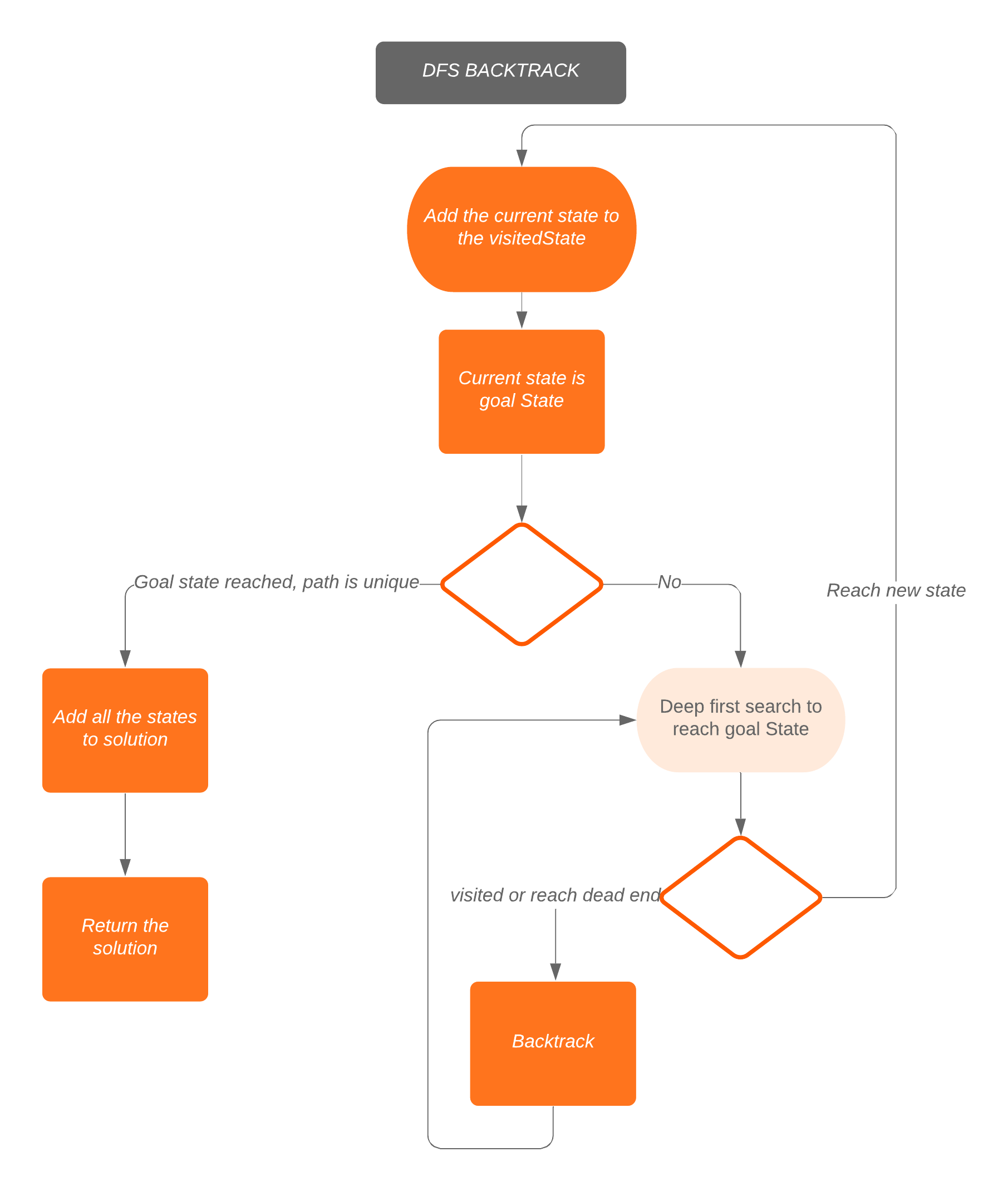
* The states will be implemented as objects contains 1 node for pawn one and 1 node for pawn two and a previous state to fetch the previous path.

## 1.4 Maze puzzle class diagram:



## 1.5 Flow chart:





# **2. Code explanations:**

All the magic happens in two function: foundSolutions which return a list of valid solutions and dfsBacktrack which implement deep first search and back track algorithms to progress through the graph, back to previous states and finally return a solution contains the states and the path to goalNode when one of two pawns reach the goalNode or an empty solution when all solutions is found.

## 2.1 Function find solutions:

This function requires a start state and the goal node in order to find all the solutions for the maze graph.

This function keeps track of a list of solutions, a set of success paths (the paths to goal node of solutions), a set of visited states (the states that has already been visited), a linkedList of states as a solution. Every solution will contain a unique list of visited states to avoid cycles traverse and ensure that different solutions with part of states duplicates with each other is checked. When a new solution (paths is unique compared to success paths) is found using dfsBacktrack it is added to the list of solutions and a new path is added to success paths. When an empty solution is found, it means that other unique solutions don’t exist, the code get out of the while loop and return the list of solutions.

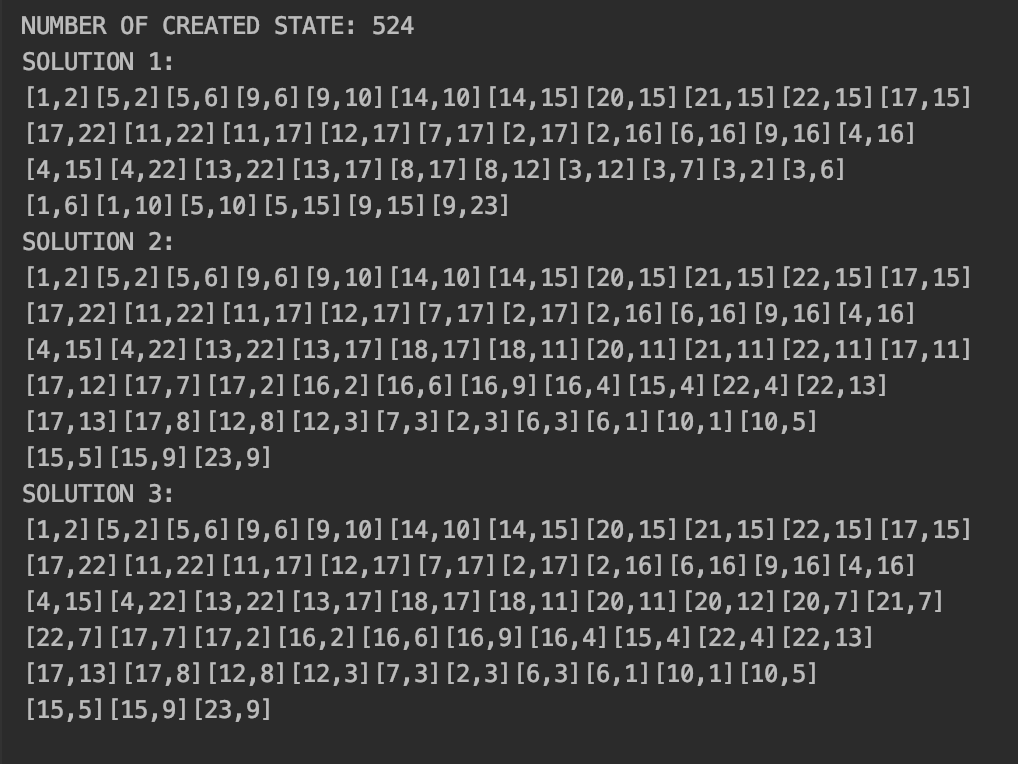
## 2.2 Function dfs backtrack:

This function requires a start state, list of visited state, list of success paths from the find solutions function. It is called inside the find solutions function and will return a solution which will later been added to the solutions list or help getting out of the loops.

Dfs backtrack is a recursive function with a base case which provokes when the startState is the goalState (the goal node is reached by one or two pawns) and the path lead to this state is unique (this is a must checked to avoid repeatedly returning the same solution). If the goal state is not reached, we keep on going deeper by recursively move to the neighbor of current states. There always be points where we have to make a decision to move pawnOne or pawnTwo, we start with pawnOne, move it as far as possible then pawnTwo if pawnOne is stuck. If we reached a dead end (pawnOne and pawnTwo cannot move forward) we come back to the time we have to make the decision and move pawn two instead of pawn one. In the code this is done by remove the current state in visited state and return an empty linkedList of solution (prevent the recursion to dive deeper, go back instead). This is repeated until the solution is reached which will take us straight to inside the base case which will step by step add the current state to the solutionStates, return it to the previous recursive level … Finally, the solution is return to the findSolutions function.

After the first solution is found, finding the second one is a bit trickier. Luckily, with the help of successPaths, the base case will not except the first solution as an answer this time. The algorithm will get to the state of first answer because each solution owns a unique visited path, then try another part with will result in the incensement of states created to find second solutions. Therefore, the number of states created for solution 2 = the number of states created for solution 1 + states created for another unique path. The same with solution 3, which will go through the path of both solution1 and solution2. This means that the algorithm can be optimized for a better performance. In our result, the states created to find 3 solutions is 524, this can be greatly reduced if the dfsBacktrack don’t have to traverse through old solutions, traverse back then try another path to find a new solution.

However, the algorithm covers all the solutions in spite of its weakness in performance. Below is the output of the program:



All is valid solutions to the puzzle.