Document Info

2018/2019 COMP1037 Coursework 1 – Search Techniques

Personal Info

Name : Tianyi GAO (高天怡)
Student ID : 20028309
Email Address : scytg1@nottingham.edu.cn

Task (a) - Analysis code of Maze-Generator

The main structure is mainly one Matrix (2D Array, n*n) named *maze* which is the noumenon of Maze contains 5 type of number (0->wall, 1->path, 3->start_point, 4->end_point) and another Matrix (2D Array, 2*n) named *nodes* which contains the expanded points.
The core logic of the implementation is using Depth-First Search to explore routes. After exploring the routes, call function *adjustEnd* to make sure *endpoint* is linked to routes found by DFS

Following are detailed steps for DFS algorithm in Maze-Generator
Lines of Code : **#main.m** 34-37; **#move.m** all; **#adjustEnd.m** all

1. Starting from start cell, the program then selects a random neighbouring cell that has not yet been visited.
2. Repeat selecting a random neighbouring cell until to a dead-end (a cell has no direction to expand). When reaching to a dead-end, current node backtracks through the path until it reaches a parent cell which has any directions to expand.

(Main DFS Algorithm)

3. Repeat (2) until every cell is visited, check whether *endpoint* is linked to routes, if not, link it to routes.

Task (b) - Logic problems in Maze-Generator

I thought out three possible logic problems, first two problems are caused by search algorithm of maze-generator provided for this coursework, which is Depth-First Search. The third problem is caused by incorrect weight assignment for moving to next neighbor.

Following are three problems I thought would be:

1. The route created is always **long path**.
   This is because DFS algorithm will continuing search for next one until a dead-end.
2. Right path (solution) for generated maze is only one.
   As I thought, maze can have multiple solutions sometimes, but in this case, solution is unique, it is also due to the DFS algorithm.
3. This problem is caused by code in **#move.m 63**, after correcting the sum(locations), the value is still not 100, it should be corrected as *locations(k) = locations(k) + locations(k)/sum(locations)*(100 - sum(locations));* .Origin code will cause the maze have more oblique path.

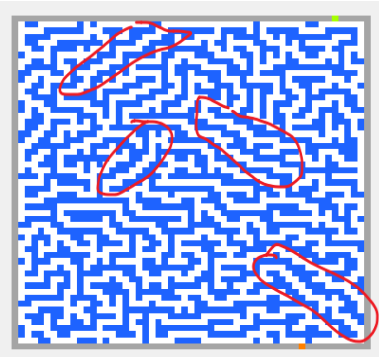


Figure 1

Task (c) – Develop log for AStarMazeSolver

To implement AStarMazeSolver, I mainly used code in folder "**Astar**" and take the file "**dispMaze.m**" to display maze dynamically.

I made changes in these lines of code (**star means vital change**) :

1. **A_Star.m** -> **AStarMazeSolver.m**
   - ● Delete code for old input (*problem* function) and add new code for getting a maze
   - ● Add code for get Start and Target
   - ★ Change the if-case for adding obstacles into list – value for obstacle is different in maze
   - ★ Add code for changing value of expanded point in *maze_output* for displaying all the routes that A* has processed with RED color as required.
   - ● Add if-case for *result* function
2. **dispMaze.m**
   - ● Define new color in *cmap* for node been visited (value == 5) and node on optimal (6)
3. **result.m**
   - ★ Add code for getting 'total path cost', 'number of nodes discovered' and 'number of nodes expanded' from QUEUE and print them
   - ★ Add code for changing value of optimal point (when finding the optimal route) in *maze_output* for displaying the final solution with BLACK color as required.
4. **Expand.m**
   - ★ Change if-case condition for finding surrounding nodes, no more node at corner

Task (d) – Modify AStarMazeSolver to DFSMazeSolver

To modify my DFSMazeSolver from AStarMazeSolver, the difference is mainly choosing next *(xNode,yNode)*. So how dose DFSMazeSolver choose a new *(xNode,yNode)*?

There are two main cases:

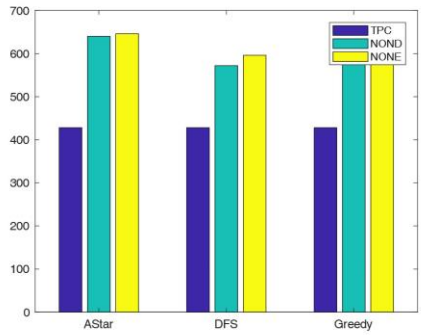
1. **When current node have child** : DFSMazeSolver doesn't need to evaluate nodes in the *QUEUE* and find the best one in the *QUEUE* but choose a random child which is very easy.
2. **When current node have no child (dead path) :** Unlike GreedyMazeSolver and AStarMazeSolver, DFSMazeSolver need to trace back to parentNodes when meets a dead path, which is the *elseif (exp_count == 0)* part in my main function of DFSMazeSolver, following is how it works:
   a) No child node found
   b) Set current node to current node's parent
   c) Find a unprocessed sibling whose parent is current node
   d) Loop a) b) c) until a available node is found, set the node to current node

Task (e) – Modify AStarMazeSolver to GreedyMazeSolver

To modify my GreedyMazeSolver from AStarMazeSolver, there is one change in total, which is transfer the evaluate function from f(n) to h(n), what did I do is just delete code for f(n) and set the evaluate function in main function to h(n), very easy !.

Task (f)

Comparison



Explanation

1. **total path cost**

Definition : the number of nodes in the optimal solution
How to extract : the last inserted node in *QUEUE* is the last node on optimal path, and it have g(n) which is the **path cost** from start point to this point, so it is the **total path cost**.

2. **number of nodes discovered**

Definition : the number of nodes which have been returned by *expand* function
How to extract : once a node is returned by expand function, main function will add it into *QUEUE* unless the node is duplicate, therefore, the **number of nodes discovered** is the length of *QUEUE* at last.

3. **number of nodes expanded**

Definition : the number of nodes which have been discovered and treated as current node in loop (added to *OBSTACLE* and marked status as 0)
How to extract : because every node expanded is marked status as 0, so **number of nodes expanded** is the number of nodes whose status (first value of array) is 0.

Code

```
%% Group data from different algorithm
dataCategory = categorical({'AStar','DFS','Greedy'});
data = [AStar; DFS; Greedy];

%% Generate the bar chart
bar(dataCategory, data);

%% Set legend and title of graph
legend('TPC', 'NOND', 'NONE');
title(sprintf('SIZE: %d     DIFFICULTY: %d;', size, difficulty));
```