Watch the last Video – this covers Backtracking using Stack

Coding challenge #4: Select Part 4 video
https://thecodingtrain.com/challenges/10-dfs-maze-generator

Video: time	Main.cpp Lines
	Source split into 4 files: wxDFS3App.h, wxDFS3App.cpp, wxDFS3Main.h, wxDFS3Main.cpp
00:00 Summary Cell moving around removing walls – need backtracking	
00:50 Backtracking - algorithm review	Illustration 1 Algorithm Steps
01:00 choose randomly unvisited neighbor	3e Case 4
01:12 Push current cell to stack	3e Case 4
01:37 What is stack? - data structure	
02:40 push – pop LIFO	
03:50 implement stack – JS array?	std::stack <Cell> MStack;
04:27 stack.push(current);	3e Case 4
05:20 when need to use stack? No available neighbors stuck	void Maze::NewCurrentFromStack () Line 308
06:05 stuck, so use stack if not empty	Line 325
06:30 pop new current cell from stack	
07:20 runs now without getting stuck – visits every single spot	
07:37 stops when stack is empty – maze map complete	Line 314
08:00 see backtrack when got stuck	Backtrack not shown, just proceed with exploring next available neighbor

Algorithm review

Algorithm outline from the wxDFS3Main.cpp code:

Algorithm Steps from wxDFS3Main.cpp code

1) TInit: Initialize the state machine

2) TNextCell: Get the next cell from stack (backtrack)
3) TTryEdges (neighbors)
   a) Case0 → TryEdge up
   b) Case1 → TryEdge right
   c) Case2 → TryEdge bottom
   d) Case3 → TryEdge left
   e) Case4 → randomize available edges onto stack

4) TDoNothing: stack empty, stop timer

Illustration 1: Algorithm Steps

for 2) TNextCell:

```
case TNextCell:
    // get new current cell from stack
    // current(squirrel) = stack.pop()
    MyPath.NewCurrentFromStack();<<<<<<< Backtrack though the Stack contents
    edgeCounter = 0;
    break;
```

Setup and run:

Either use original code that was unzipped or go back and use code from Lesson 3 and re-enable the backtracking function

In Case 4, line 225 remove the code added in Lesson Two and switch back to the original code.

```
  convert back to original code like below
225        for (int i=0; i<numEdges; i++)
226        { // int i=0; // only one neighbor, not more
```

This original code for State 4: randomizes available neighbor edges and pushes them onto stack. This way in Illustration 1, Step 2, TNextCell: if Current cell has no

available neighbors, then Step 2) is repeated and another cell is pulled from the stack. This keeps popping prior available neighbors from the stack until one is found still not visited and available or stops looping if the stack is empty. This decision is made by looping and repeating TNextCell in Illustration 1 Algorithm Steps.

Assignment:

1. Compile and run the code modified as instructed above – use a different origin by entering coordinates for the ‘Start’ step. Illustration 2

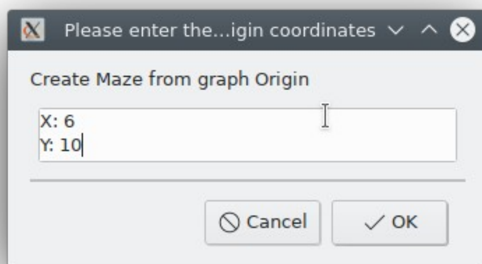

Illustration 2: Set the Maze origin

2. The program will run to examine every cell in the Maze. Notice the backtracking when there is no free neighbor available when exploring.
3. The ‘Step’ button can be pressed and will place the exploration into a single step mode where a single ‘current’ cell is explored.
4. Pressing the ‘Start’ button after ‘Step’ will continue the solution at full speed. The coordinates given there are ignored and the solution continues.

Created Maze

5. When every cell has been explored, you will see all the remaining cells still on the stack being rejected because they are already visited.

6. When the Stack is empty, the Timer and program are stopped automatically.

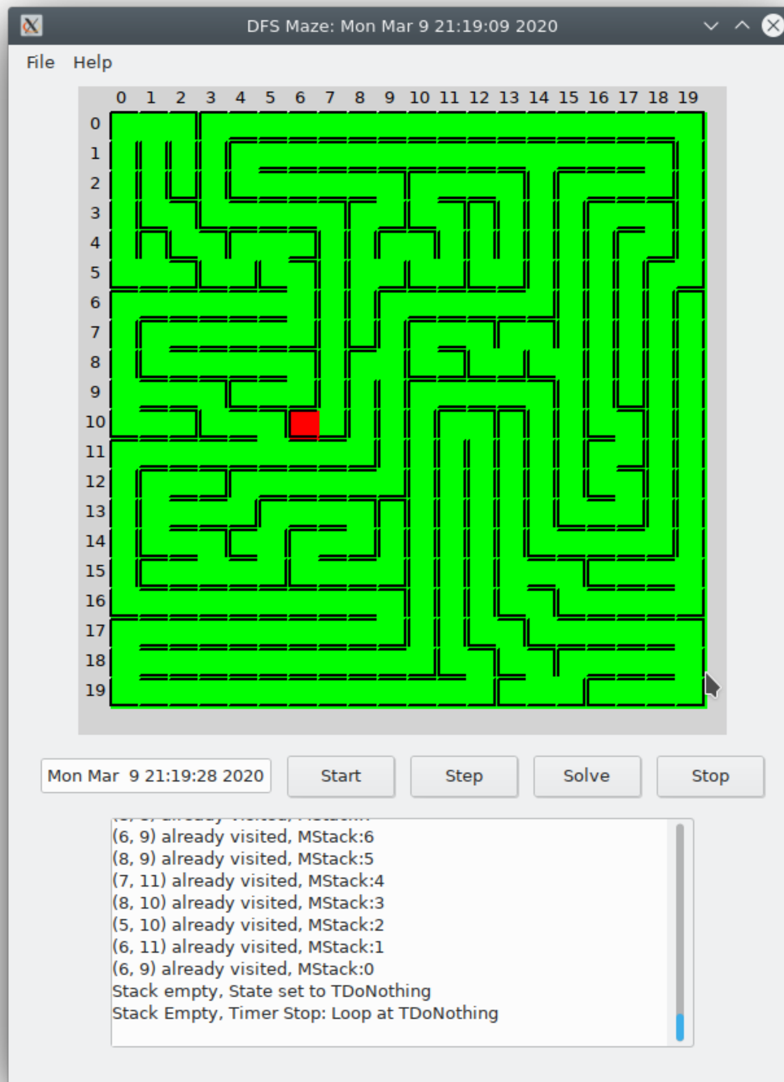

Illustration 3: Maze, completed DFS discovery

7. Turn in a copy of this screen shot when it is complete. Screen Shot #1

Your Maze will be different because the randomization of available cells makes the path unique and different each time.

Now, have some fun with the Maze!

Solved Maze

The ‘Solve’ button allows tracing back through the Maze to the origin from any selected end point cell in the Maze.

Press the ‘Solve’ button and the path back to the origin will be solved. The linked list ‘from’ variable in each cell provides a numeric value for the edge which entered the Cell. This provides a reverse linked list allowing tracing back to the orgin to be done.

Since a temporary copy of the solution is created to ‘Solve’, multiple path ‘Solve’s can be done on the same Maze solution.

Press the ‘Solve’ button and give the coordinates of a destination cell. Illustration 4

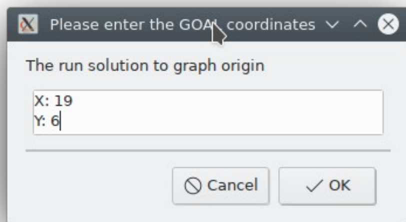

Illustration 4: 'Solve' coordinate dialog

8. Pick the destination cell coordinates and turn in a copy of it's screen shot when solved. Error: Reference source not found

The path from the specified destination is traced back to the origin. This can be repeated multiple times from different destinations.

Illustration 5: Solved Maze

9. Turn this in as Screen Shot #2.