GRAPHS, TREES, AND SEARCHING

Relearning what I have forgotten

Solve a Maze using a Depth First Search algorithm written in C with Explanations

Clyde Hoadley
Arvada, Colorado

July 2021

Version 1.0a

Abstract

The sole purpose of this project is to play with the **LaTeX**[1] typesetting software, practice programming, and relearn some theory.

Using non-technical language this paper will introduce just enough graphs and algorithms for the non-technical reader to gain an understanding of what's going on. After the brief review the paper examines the Depth First Search algorithm. Finally the paper discusses the implementation of a maze solving computer program and explain it section by section.

Caveat emptor: this paper is offered without guarantee of it's completeness, correctness, having been proof read or spell checked.

[1]**LaTeX** is a document preparation system used for the formatting and publication of technical documents in many different fields.[7]

Acknowledgments

Kudos go to Mr. Richard West. His YouTube video demonstrating the depth-first maze solution proved to be invaluable for me. His YouTube channel can be found here:
`https://www.youtube.com/watch?v=XP94WC_XnZc`

Contents

List of Figures

1 Why?

I wrote my first computer program in 1976 and worked in the information processing field for 30 years, but I haven't studied algorithms in nearly 40 years and have forgotten most of what I knew.

I have more hobbies than I can keep track of, I cycle through them about every two years. I'm in my programming phase now and have started studying algorithms and practicing programming.

While brushing up on algorithms I came across the LaTeX text formatting software and started digging into it. I've been having fun playing with LaTeX. I thought it would be fun to try combining algorithms, programming and LaTeX into a single project. I decided to develop a program to solve a maze then use LaTex to write a paper about what I learned.

While doing a quick review of algorithms and graphs I decided that a depth first search solution would be a good choice. I didn't know at the time that this subject has been done to death on the Web and YouTube. I started having second thoughts about the project, but decided just because everyone else has done it was no reason I couldn't do it too.

This paper is not intended to be a real research paper but as a form of geeky play. I get to relearn some things I've forgotten, practice programming, and play with the LaTeX software.

2 Algorithms

An algorithm is a set of instructions that when followed, step by step, solves a problem or does something useful. In computer science, algorithms are further required to be:

- ◇ Finite: An algorithm must always terminate after some number of steps.
- ◇ Unambiguous: Every step of an algorithm must be well defined, clear, and not subject to interpretation.
- ◇ Input: An algorithm must receive some input before it starts (ingredients, numbers, a set of objects, etc...)
- ◇ Output: An algorithm must return some kind of result that is related to its input.
- ◇ Effectiveness: The steps in an algorithm must be simple enough that they could be done by hand in a finite amount of time.[3](pages 4-5)

Algorithms are often written in a pseudocode language. *"Psuedocode is a notation resembling a programming language but not intended for actual compilation. It usually combines some of the structure of a programming language with an informal natural-language description of the computations to be carried out."*[2]

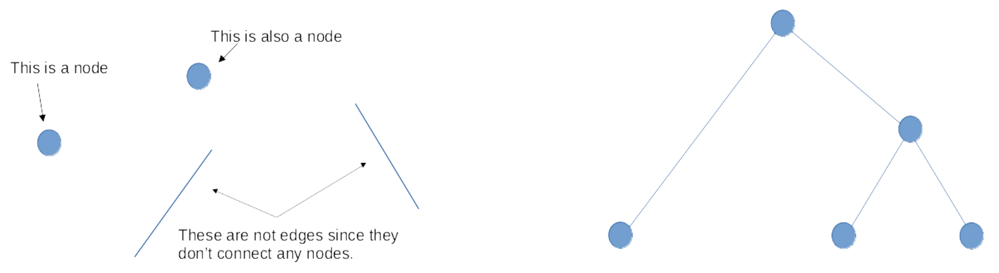
For example:

Bubble sort:

```
repeat
  set a flag to False
  for each pair of keys
     if the keys are in the wrong order then
        swap the keys
        set the flag to True
     end if
  next pair
until flag is not set.
```

3 Graphs

A graph is a collection of one or more points and lines that connect some of the points together. The points on a graph are called **vertices**. The lines that connect vertices together are called **edges**. A vertex can have zero or more edges. An edge can not exist unless there are at least two vertices but a vertex can exist without having any edges.



(a) Two vertices but no edges (b) Five vertices and four edges

Figure 1: Vertices and Edges

A *path* in a graph is a sequence of vertices connected by edges. A *simple path* is one with no repeated vertices.[5](page 519)

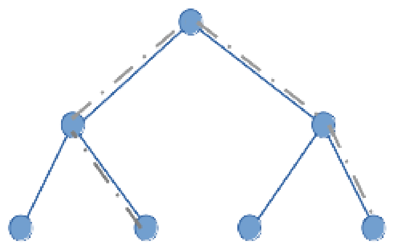
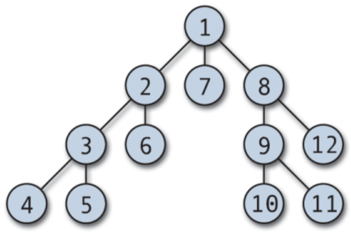


Figure 2: Dashed line represents a simple path.

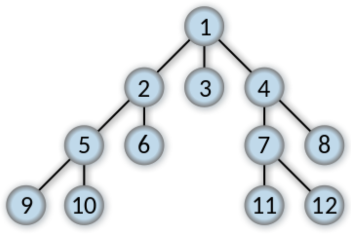
Graphs can be used to model many different kinds of relationships. One of the most common problems that arises when working with graphs is to find and visit every edge and vertex of the graph. The two most common ways to solve this problem is to use either the *breadth-first search* algorithm or the *depth-first search* algorithm.[6]

4 Depth First Search

The depth-first search algorithm uses a recursive[2] method to visit every vertex and edge of a graph. When given a starting vertex the algorithm travels as far as possible down each edge and vertex until it must backup to an un-visited edge. The algorithm continues this process until all vertices and edges have been visited.



(a) Depth first order of vertices traversal.



(b) Breadth first order of vertices traversal.

Figure 3: Depth first vs Breadth first search.

Depth-first search:[5](page 531)

Given a vertex, mark the vertex as having been visited
recursively visit all neighboring vertices that have not been marked.

[2]In computer programming recursive means a procedure such as the *Depth-first search* above invokes (or calls) itself to solve the next step of the problem. It's a divide and conquer technique.

5 Making a maze

The definition of a maze that I like the best comes from The American Heritage Dictionary where they define a maze as *"A graphic puzzle, the solution of which is an uninterrupted path through an intricate pattern of line segments from a starting point to a goal."*[4]

A maze can be thought of as a grid of cells where each cell has four walls (North, East, South and West). When given a starting cell, a computer program can use the rules below to carve out a path.[1]

- Examine each neighboring cell. If there are neighboring cells that have never been visited, choose one at random and move to it then remove the wall between the cells.
- If all of the neighboring cells have been visited the program has reached a dead end and must backtrack to the last cell that has an unvisited neighbor.

For this paper the program being reviewed will only solve a maze. The maze to be solved will have already been created. by hand as a comma separated values (csv) data file. The perimeter and internal walls are represented with ones and pathways are represented with zeros. The finish, represented with the letter “F”can be anywhere in the maze. For simplicity this program will have a fixed starting position at row 2 column 1 (array index 1,0.)

A simple matrix csv file is shown below.

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
S,0,0,0,0,0,0,0,1,0,0,0,1,0,0,1
1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,1
1,1,1,1,1,0,0,0,1,0,0,0,1,0,0,1
1,0,0,0,0,0,0,0,1,0,1,1,1,0,0,1
1,1,0,0,1,0,1,1,1,0,0,0,0,0,0,1
1,0,0,0,0,0,1,0,1,0,1,1,1,1,0,1
1,0,0,1,1,0,1,0,1,0,1,0,0,0,0,1
1,0,0,0,1,0,1,0,0,0,1,1,1,1,1,1
1,1,1,1,1,0,0,0,1,0,1,0,0,0,0,1
1,0,0,0,1,0,0,1,1,0,1,0,0,0,0,1
1,0,0,0,1,0,0,1,1,0,1,0,1,0,0,1
1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1
1,1,1,1,1,1,1,1,1,0,0,0,1,0,0,1
1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1
1,1,1,1,1,1,1,1,1,1,1,1,1,1,F,1
```

6 Solve a maze

For the depth-first maze solving procedure, when given a maze and a starting coordinate. If the coordinate is not on a wall, recursively call this procedure for every neighboring coordinate that has not already been visited. The procedure will return to the previous call (i.e. backtrack) if it reaches a dead end.

I won't go over every line of the program code but will describe what's taking place in the maze solving sections. A complete and well documented copy of the rmaze.c source code can be downloaded from:
`https://github.com/ClydeHoadley/Simple-samples-of-C-with-explanations`

6.1 main()

The main() function of the program does a good job of laying out what will take place. The *main()* function is where C programs start, here it creates and initializes a few variables and arrays, loads the data file, establishes the starting coordinates, then sets *findTheWay()* to work solving the maze.

aMaze is a 2 dimensional array of characters that will hold the maze.

```
char aMaze[MAX_MAZE_SIZE][MAX_MAZE_SIZE];
```

path is a 1 dimensional array of row/column coordinates. The path array will function as a first in last out stack.

```
typePoint path[MAX_MAZE_SIZE*MAX_MAZE_SIZE];
```

The get_maze() function opens and reads-in the csv data file that describes the maze. The file name was passed in from the command line via argv[1]. The function also filters out any characters that shouldn't be there. When done it returns a count of the number of rows in the maze.

```
number_of_rows = get_maze(argv[1],aMaze);
```

The addToPath() functions "pushes" a row/column coordinate pair onto the top of the path stack. Here the starting coordinates of the maze are pushed onto the stack. The variable pathSize keeps a count of how many items are on the stack.

```
pathSize=addToPath(path, 1,0, pathSize);
```

The findTheWay() function is the heart of the program. It is a recursive implementation of a depth-first search algorithm.

```
findTheWay(aMaze,path,pathSize,number_of_rows);
```

6.2 display_maze()

The function display_maze() displays the maze and current progress of the findTheWay() function on the screen. The display_maze() function would destroy the maze if it didn't make a temporary copy of it first.

```
char mzCopy[MAX_MAZE_SIZE][MAX_MAZE_SIZE];
copy_maze(mzCopy,maze,number_of_rows);
```

If pathSize is greater than zero it means findTheWay() has started down a path. The display function needs to show the current path and where findTheWay() is working at the moment. The display_maze() function cycles through the stack and places a "." for each coordinate on the stack. After cycling through the stack it places an "*" to show where findTheWay() is at the present.

```
    if (pathSize) {
        for(int i=1; i<(pathSize+1); i++)
            mzCopy[path[i].x][path[i].y] = (char)'.';
        mzCopy[ path[pathSize].x ][ path[pathSize].y ] =(char)'*';
    };
```

After the path has been displayed the function cycles through the maze and prints a blank space for open pathways and a colored square for walls. Remember the maze is a 2 dimensional array filled with the 1's and 0's from the csv data file. A 2 in the array indicates a point that has been fully inspected and does not need to be visited again. It will also be displayed as a space. The ANSI terminal hexadecimal escape codes E2 96 88 print a small color square.

```
    for(x=0; x<number_of_rows; x++) {
        for(y=0; y<MAX_MAZE_SIZE; y++) {
            if (mzCopy[x][y]==(char)NULL) { printf("\n"); break; }
            if (mzCopy[x][y]=='2') { printf(" "); continue;}
            if (mzCopy[x][y]=='1') { printf("\xE2\x96\x88"); continue;}
            if (mzCopy[x][y]=='0') { printf(" "); continue;}
            printf("%c",mzCopy[x][y]);
         }
    }
```

After the maze has been put onto the screen, it's necessary to pause the program briefly to give the user time enough to see it. The code below pauses the program for about 1/8 of a second which is just long enough for our eye/brain to register it but fast enough for the screen to appear animated.

```
    struct timespec t1, t2;
    t1.tv_sec = 0;
    t1.tv_nsec = 125000000L;
    nanosleep(&t1 , &t2);
```

6.3 findTheWay()

The findTheWay() function is an implementation of a depth first search (DFS) algorithm to find a path (any path) through a maze. It looks big and complex but looks can be deceiving, it's actually a straight forward implementation of the algorithm. Most of the function is testing to see if the coordinate it has is a new one or an old one. If it's a new coordinate does it have any unvisited paths or does the function need to backtrack.

The coordinate the function is working with was passed into it from a previous function call. The first thing it does is create a list of east, west, south, north choices of potential next moves. Randomizing or juggling the choices isn't a requirement of the DFS algorithm, it's done here to give the program the ability to choose a different path each time it's run.

```
    choices[0].x = path[pathSize].x;
    choices[0].y = path[pathSize].y+1;
    choices[1].x = path[pathSize].x;
    choices[1].y = path[pathSize].y-1;
    choices[2].x = path[pathSize].x+1;
    choices[2].y = path[pathSize].y;
    choices[3].x = path[pathSize].x-1;
    choices[3].y = path[pathSize].y;
    juggleChoices(choices);
```

The function next cycles through the four choices and checks:

1. Does the next coordinate lie within the maze?
   ```
   if ( (x < 0) ||
      (y < 0) ||
      (x > number_of_rows) ||
      (y > number_of_columns) ) continue;
   ```
2. Is the next coordinate a wall or a coordinate that has already been fully checked?
   ```
     else if( (maze[x][y] ==(char)'1' ) || (maze[x][y] ==(char)'2') )
           continue;
   ```
3. Is the next coordinate already in the current path? *The findInPath() function is a simple linear search of the path stack.*
   ```
   else if (findInPath(x,y,path,pathSize) > -1 ) continue;
   ```
4. Does the next coordinate finish the path and yield a solution to the maze?
   ```
   else if ( maze[x][y] ==(char)'F' ) {
        pathSize = addToPath(path, x,y ,pathSize);
        display_maze(maze,path,pathSize,number_of_rows);
        endTheProgram(EXIT_SUCCESS,number_of_rows,number_of_columns);
    }
   ```

Else, move to this next coordinate and call the findTheWay() function again passing it the new coordinate. This is where the recursion takes place. Backtracking takes place when, or if, the findTheWay() function returns to its previous call.

```
    else {
        pathSize = addToPath(path, x,y ,pathSize);
        findTheWay(maze, path, pathSize, number_of_rows);
        maze[x][y] = (char)'2';
        display_maze(maze,path,pathSize,number_of_rows);
     }//end-if-else
```

If the program is not able to solve the maze it eventually backtracks all the way back to the starting point before returning to the main() function where it reports that it was unable to solve the maze.

7 Coda

I'm a little surprised that I did this project.! Reading and writing are not my favorite pastimes. Yet, the fact is I had a lot of fun doing this project. LaTeX is so old school and technical that it has a strong appeal to me. I have **MiKTeX**[3] and **TexMaker**[4] installed on my computer. MiKTex provides the LaTeX software, packages and utilities. TexMaker provides a GUI interface for LaTeX. One can use a simple text editor like **vi** or **notepad** and compile your document from the command line but having a GUI editor that will also invoke the compiler for you is handy to have.

Developing the C program was easy; I had it running in a day then spent a couple of days refactoring the program, simplifying some data structures, and documenting it. The source code and this document are now on my GitHub.[5]

I studied algorithms and data structures 40 years ago. I can still see Robert Bruce, Ph.D. in *"Data Structures"* class, and Earl Haze, Ph.D. in *"Structured Programming"* class drawing graphs and trees on the chalk board, but aside from some of the terminology, I don't remember any of it. I certainly remember recursion and how cool I thought it was. I and many others failed to provide a stopping condition during our first recursion assignment. We all had tried comparing a floating point value for equality - which doesn't work. We learned what not to do.

I've used recursion many times since then. A month ago I implemented the Merge Sort algorithm in C using recursion. I had a lot of trouble getting that program working. Maybe that's why the depth-first maze solution was so much easier.

I have relearned some forgotten theory, impressed myself that I can still write a program, and had fun doing it. Time well spent!

[3]`https://miktex.org/`

[4]`http://www.xm1math.net/texmaker/`

[5]`https://github.com/ClydeHoadley/Simple-samples-of-C-with-explanations`

References

[1] Christian Hill. Making a maze. https://scipython.com/blog/making-a-maze/, 2017. [Online; accessed 17-July-2021].

[2] Denis Howe. ”psuedocode” from the the free on-line dictionary of computing. https://encyclopedia2.thefreedictionary.com/Psuedocode, 2003.

[3] Donald E Knuth. *The Art of Computer Programming, Volume 1/Fundamental Algorithms*, volume 1. Addison-Wesley Publishing Company., 3rd edition, 1997, 1973, 1968.

[4] Editors of the American Heritage Dictionaries, editor. *The American Heritage Dictionary Of The English Language*. Houghton Mifflin Harcourt Publishing Company, 5th edition, 2016, 2011.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley/Pearson Educationy, 4th edition, 2011.

[6] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 3rd edition, 2020.

[7] Wikipedia contributors. Latex — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=LaTeX&oldid=1023081178, 2021. [Online; accessed 18-July-2021].

Sample output

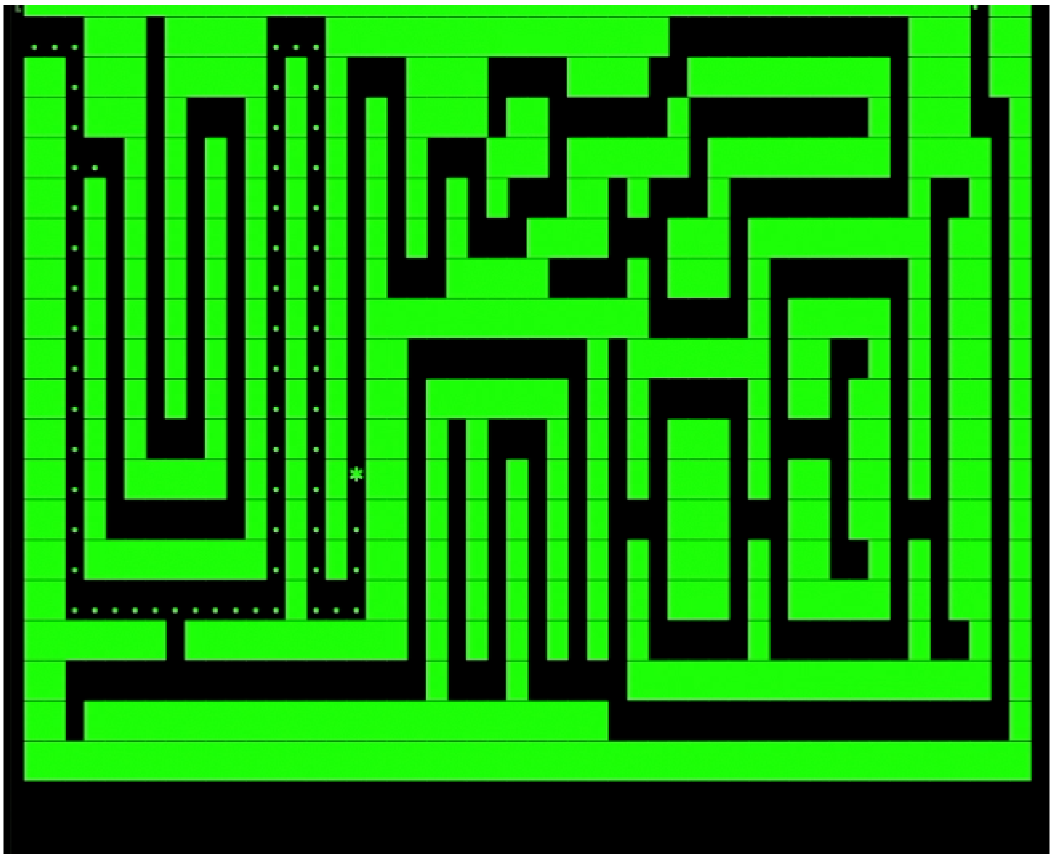
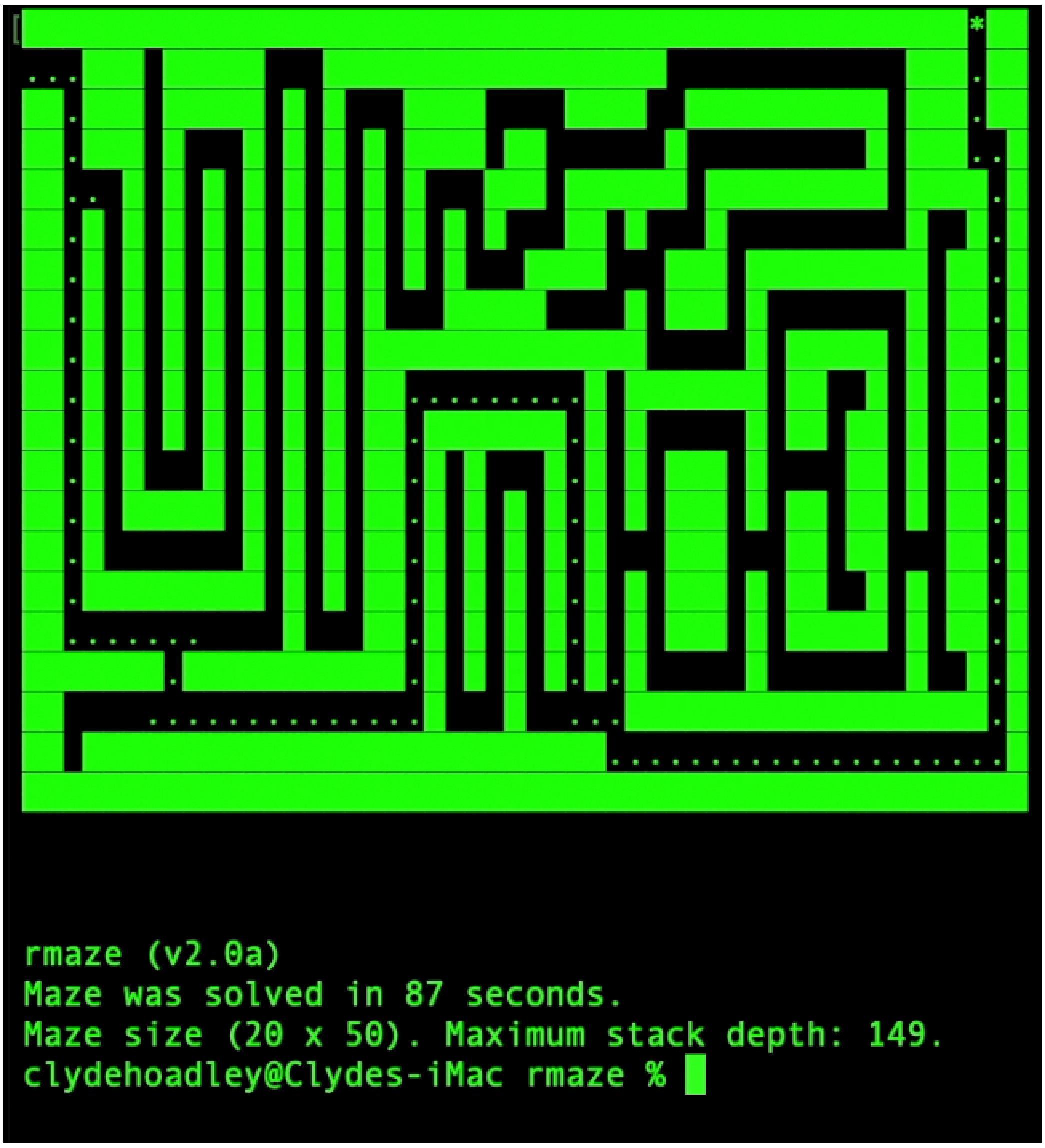


Figure 4: Attempting to solve the maze.
Note path and current location.



```
rmaze (v2.0a)
Maze was solved in 87 seconds.
Maze size (20 x 50). Maximum stack depth: 149.
clydehoadley@Clydes-iMac rmaze %
```

Figure 5: A fully solved maze.
Note the stats at the bottom.

Index