COSC 302/307: Mazesolve - Recursive and Iterative Tips

Clara Nguyễn

November 6, 2019

1 First off...

Consider this a continuation of that sketch'd up document about Mazemake. This time, though, instead of me writing on literal glass, I'm using LaTeX. Cool, huh? As an actual note, all coordinate notation in this document is assumed to be in (x, y) unless otherwise specified.

2 Reading in the Maze

So Mazesolve has a few ways to be implemented. In this assignment, you are doing **DFS** (Depth-first search), which can be implemented with recursion or iteration if you know what you're doing. You have solution executables to compare your answers to (namely `smazesolve`). The executable will take the following parameters:

```
UNIX> ./smazesolve maze.txt path.txt
```

This application will generate `path.txt` using data from `maze.txt`.

2.1 Format of `maze.txt`

Before we even try to solve a maze, we need to read it in properly. Let's recap on the format of a valid `maze.txt` file. It follows this format:

```
MAZE Nrows Ncols
c1_row c1_col c2_row c2_col
c1_row c1_col c2_row c2_col
...
c1_row c1_col c2_row c2_col
```

If you are comfortable with (x, y) notation, it is as follows:

```
MAZE Nrows Ncols
y1 x1 y2 x2
y1 x1 y2 x2
...
y1 x1 y2 x2
```

Example: Assume we have already run `smazemake` to generate the following valid `maze.txt` of size 2×2:

```
MAZE 2 2
  0   0   1   0
```

The choice to do a 2×2 maze is due to it being guaranteed to only have a single wall. This maze has a horizontal wall between the two cells in the first column. Assume this generated maze is M, and assume it visually looks like this:

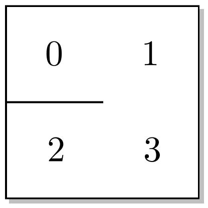


Figure 1: M from `maze.txt`

We will cover the format of `path.txt` when the algorithms are covered.

2.2 Reading in `maze.txt`

You can't do anything with data if you don't read it into your program. Recall that the first line of `maze.txt` is the following:

```
MAZE 2 2
```

This has the `MAZE` header, as well as row and column (height and width) specifications for the entire maze. Since you're restricted to doing C-style input, we can read this in with `fscanf`:

```c
FILE *fp = fopen("maze.txt", "r");
fscanf(fp, "MAZE %d %d", &Nrow, &Ncol);
```

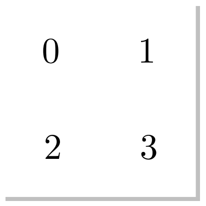
Simple enough. Now let's initialise our maze M. Assume M is a three-dimensional array of bools of size M[Ncol][Nrow][4] (Writeup says *row* should be first, but I flip mine. Deal with it). Set all values in $M[][][]$ to false, meaning no walls exist.

You may have noticed that the furthest inward array is a fixed size... 4. This is for whether the cell at $M[X][Y]$ has a wall in a specific direction. You may define these directions in any way you wish. For reference, I define mine as the following enum:

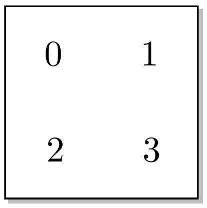
```c
typedef enum direction {
    DIR_LEFT,  /* 0 */
    DIR_UP,    /* 1 */
    DIR_RIGHT, /* 2 */
    DIR_DOWN   /* 3 */
} DIR_T;
```

Using this enum as reference, we can say `DIR_LEFT` is 0, `DIR_UP` is 1, and so on. You may set yours in whatever way you wish. It won't matter because, assuming your maze has no cycles and is all in one set, only one solution exists, and DFS will *eventually* find it.

Anyways, so we read in the header and have $M[Ncol][Nrow][4]$ set up with everything set to false. This is what M looks like: Let's start setting some to true. Before we read in lines, set all boundary walls to true. M should look like this:



(a) M with all walls false



(b) M with boundary walls true

Now we can read in data. Use `fscanf` to read in each line into (y_1, x_1, y_2, x_2) (Just read in `"%d %d %d %d"` until `fscanf` doesn't return 4). Then use that to set the walls... but then we run into another problem. How do we know which walls to break down? If you're given 2 points, you need to know the *direction* of one point facing another to know to put a wall between them.

Let's make up a function that takes 2 points and returns the direction from the first point to the second. We'll name it `get_dir`. This function is incredibly simple assuming the following 2 properties hold:

- With any 2 given points (x_1, y_1) and (x_2, y_2), either $x_1 = x_2$ or $y_1 = y_2$. The points will always be aligned on one of the 2 axes.
- There is never a case where both $x_1 = x_2$ and $y_1 = y_2$. It isn't a pair of different points otherwise.

With these properties, we can write "`DIR_T get_dir(x1, y2, x2, y2)`" like so:

- If $x_1 = x_2$... it must be either up or down
  - If $y_1 < y_2$... return `DIR_DOWN`.
  - If $y_1 > y_2$... return `DIR_UP`.
- If $y_1 = y_2$... it must be either left or right
  - If $x_1 < x_2$... return `DIR_RIGHT`.
  - If $x_1 > x_2$... return `DIR_LEFT`.

There are some examples on the next page to visualise this.

Example 1: Assume $(x_1, y_1) = (2, 2)$ and $(x_2, y_2) = (4, 2)$. The direction is `DIR_RIGHT` because...

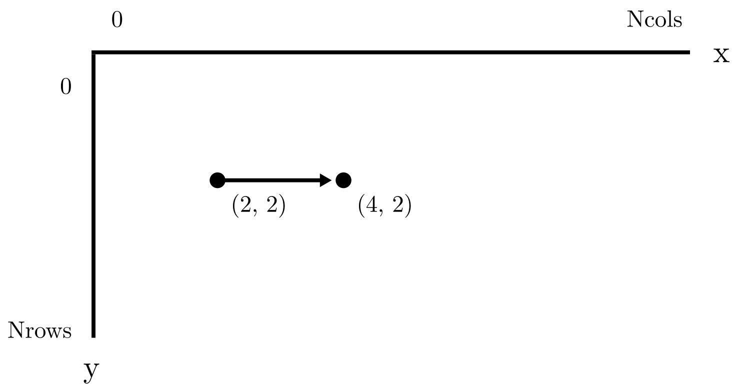


Figure 3: Plot of $(2, 2)$ pointing to $(4, 2)$.

Example 2: Assume $(x_1, y_1) = (3, 1)$ and $(x_2, y_2) = (3, 3)$. The direction is `DIR_DOWN` because...

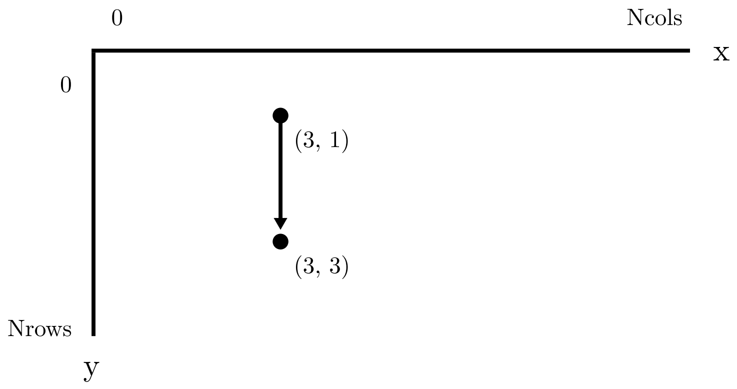


Figure 4: Plot of $(3, 1)$ pointing to $(3, 3)$.

Understand now? Okay, great. Now we can use this function to help us construct the walls. Remember, because of symmetry, we have to construct the opposite wall on the second cell of each pair as well. We can get the opposite direction by simply flipping the arguments of `get_dir`. Finally, behold the following (very short and elegant) pseudocode:

```
While fscanf(...) returns 4
    M[x1][y1][ get_dir(x1, y1, x2, y2) ] = true
    M[x2][y2][ get_dir(x2, y2, x1, y1) ] = true
```

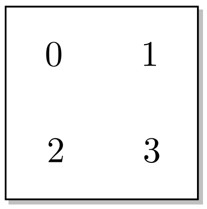
Let's go back to M. The second line is "`0 0 1 0`". So $(x_1, y_1) = (0, 0)$ and $(x_2, y_2) = (0, 1)$. Let's run the pseudocode from above on this input:

```
    M[0][0][ get_dir(0, 0, 0, 1) ] = true
    M[0][1][ get_dir(0, 1, 0, 0) ] = true
```

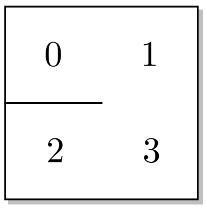
That resolves to this...

```
    M[0][0][ DIR_DOWN ] = true
    M[0][1][ DIR_UP   ] = true
```

And this will perform the following change...



(a) M before the pseudocode above.



(b) M after the pseudocode above.

Repeat this procedure for every 4 numbers read in and your maze will be constructed. Congrats, you're done reading in `maze.txt`. Now let's get to computing a valid path from one point to another...

3 Theoretical maze traversal via Depth-first search

Before we discuss a programmatical approach to solving this problem, it's good to understand how to theoretically apply DFS to a graph, as this algorithm has variations that apply to all sorts of problems in Computer Science. Don't like theory? Skip to Section 4 (pg. 8). Anyways, because the maze, M, is generated by Randomised Kruskal's Algorithm, it has the following properties:

- There exists exactly *one* unique path from *any* source S to *any* sink T.
- There are no cycles (a consequence of the first point).

Because of these properties, we can interpret M as a graph or tree where the source S is the root node and T exists somewhere in the graph that is approachable from S. If we interpret it as a graph, we can then use Depth-first search to traverse the graph until we hit T. Once we do, we have the only valid path from S to T!

General Algorithm: Given a maze M, a source S, and a sink T... Keep a stack P (for "path") of coordinates while we are traversing the graph. Assume P_n is the nth coordinate in the stack P where n is the number of elements in the stack and P_{n-1} is the top element (current point). Start with pushing S into P (thus $n = 1$). Then,

1. At P_{n-1}, try enumerate all possible valid directions. A valid direction in the maze would be one where there's no wall blocking the way, and the destination cell hasn't already been visited. You may start from any direction since we know there only exists one unique solution.

2. If a valid direction is found, push the coordinate we would be at if we took that direction to P (e.g. if $P_{n-1} = (0, 0)$ and we went down, push $(0, 1)$ to P). Also mark that cell as *visited*. Check if $P_{n-1} = T$. If this is the case, we have reached our destination and P contains the points in a path from S to T.

3. Once all directions have been checked at P_{n-1}, unless $T \in P$, we need to pop the top element off and go to the previous cell. When popping from P, you must also mark that cell as *unvisited*. Repeat this as many times as this condition holds. If $n = 0$, there is no possible solution from S to T that exists in M.

Example 1: Assume the following maze M is as shown:

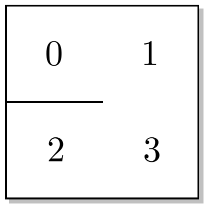


Figure 6: M from `maze.txt`

Given that the source $S = (0, 0)$ and the sink $T = (1, 1)$, find a path P from S to T using DFS.

Solution:

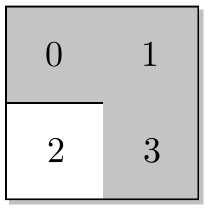
1. Start at S and push it to P. Thus, $n = 1$ and $P_{n-1} = S = (0, 0)$. This is our starting point for the algorithm listed above. Set $(0, 0)$ to *visited*.

2. Look through all possible valid directions in P_{n-1}. The only valid direction to go is to the right, which is $(1, 0)$. Push that to P. Thus, $n = 2$ and $P_{n-1} = (1, 0)$. Set $(1, 0)$ to *visited*.

3. Since we pushed a new coordinate to P, $P_{n-1} = (1, 0)$. Look at all possible valid directions we can go to from here. There is no wall between $(0, 0)$ and $(1, 0)$. However, $(0, 0)$ has been *visited*, so we can't go to it or else we will get into a cycle. The only valid path is to go down to $(1, 1)$. Push that to P. Thus, $n = 3$ and $P_{n-1} = (1, 1)$. Set $(1, 1)$ to *visited*.

4. $P_{n-1} = T$, thus a path from S to T has been found.

The result is that P has the following elements: $(0, 0) \rightarrow (1, 0) \rightarrow (1, 1)$. This is our path from S to T. The path (coloured in grey) applied to the maze is as shown:



Example 2: Assume the following maze M is as shown:

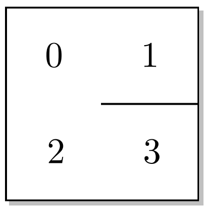


Figure 7: M from `maze_backtrack.txt`

Given that the source $S = (0, 0)$ and the sink $T = (1, 1)$, find a path P from S to T using DFS. Assume the order of directions to try is `DIR_LEFT`, `DIR_UP`, `DIR_RIGHT`, `DIR_DOWN`.

Notice something different? A specific order of directions was given so let's follow that. From S, we can't go to the left or up. But, we can go to the right and be at $(1, 0)$. However, we can't go anywhere from there since there's a wall down below and $(0, 0)$ has been visited. So we have to pop off of P, go back to $(0, 0)$ and try the next direction (`DIR_DOWN`) to reach our solution. Let's try it.

Solution:

1. Start at S and push it to P. Thus, $n = 1$ and $P_{n-1} = S = (0, 0)$. Set $(0, 0)$ to *visited.*

2. Look through all possible valid directions in P_{n-1}. We were given an order of directions to follow so we try left and up. They have walls. Right is a valid direction so go there. Push $(1, 0)$ to P. Thus, $n = 2$ and $P_{n-1} = (1, 0)$. Set $(1, 0)$ to *visited.*

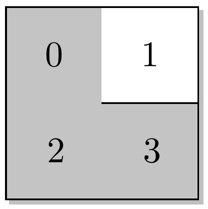
3. Look through all possible valid directions in P_{n-1}. There is no wall to the left of $(1, 0)$, but $(0, 0)$ has been visited. Therefore, it's an invalid direction. Going up, right, and down is impossible since there are walls. All directions have been checked at P_{n-1} and $T \notin P$. Pop $(1, 0)$ off of P and go back to the previous cell in the stack. Thus, $n = 1$ and $P_{n-1} = (0, 0)$. Set $(1, 0)$ to *unvisited.*

4. Back at $(0, 0)$, the last direction we tried was right. We know that direction led to a bad result, so try the next direction, being down at $(0, 1)$. This is a valid direction, so push it to P. Thus, $n = 2$ and $P_{n-1} = (0, 1)$. Set $(0, 1)$ to *visited.*

5. Now we're at $(0, 1)$. Look at all possible valid directions we can go to from here. Left has a wall. Up has no wall, but $(0, 0)$ is *visited* so it's invalid. Right has no wall and $(1, 1)$ is *unvisited.* Push that to P. Thus, $n = 3$ and $P_{n-1} = (1, 1)$. Set $(1, 1)$ to *visited.*

6. $P_{n-1} = T$, thus a path from S to T has been found.

The result is that P has the following elements: $(0, 0) \to (0, 1) \to (1, 1)$. This is our path from S to T. The path applied to the maze is as shown:



4 A recursive approach to finding a path

This kind of problem has a very short and elegant recursive solution, even without the use of C++ data structures.

Setup: Do the following:

1. Make a `cell` struct that stores an x and y coordinate. Both of them can be integers.
2. Assume the source S is at $(0, 0)$ and the sink T is at $(\text{Ncols} - 1, \text{Nrows} - 1)$. For instance, considering maze M above, $S = (0, 0)$ and $T = (1, 1)$.
3. Make an array to mimic a stack-like data structure. Let's name it "stack". To do this, either `malloc` or `new` an array of `cell` sized Nrows $\times$ Ncols (the max possible size of a valid maze path). Then, make an integer to keep track of the current position you're at in the stack. Call that "stack_size". If you push an element, this "position" is incremented by 1. If you pop an element, it's decremented.
4. Make a 2D array of integers to keep track of whether a `cell` has been visited or not. We'll name it V[Ncols][Nrows] for "visited" (lab writeup uses `iswhite`). By default, make sure all elements are set to 0. If $V[0][0] = 0$, then the cell at $(0, 0)$ hasn't been visited. If it's 1, the cell has been visited.

Now that we're set up, let's make a recursive function, named `solve`, to find a path between S and T. We need this function to push a cell to the stack. Then, for every possible direction, call itself again if there's no wall and the target cell hasn't already been visited. The function returns a `bool`. If the recursive call returns `true`, then return `true` as a solution has been found. Otherwise... after all directions have been attempted, assume failure, pop off the stack, and return `false`. Behold the following pseudocode:

```
bool solve(...) {
    cell point, to // Stores current position and cell to go to in the next step

    set V[point.x][point.y] to "true". //We are at this cell, so we have "visited" it.
    push (point.x, point.y) to stack and increment stack_size by 1

    if point.x = T.x and point.y = T.y
        return true //we hit the sink

    for all possible directions (DIR_LEFT, DIR_UP, DIR_RIGHT, DIR_DOWN) as "d" {
        if there is a wall at M[point.x][point.y][d]
            continue

        compute to.x and to.y. Use the direction to determine this
        //(e.g. if d = DIR_RIGHT, to.x = point.x + 1; to.y = point.y)

        if (to.x, to.y) has been visited already
            continue

        if (solve(...) returns true)
            return true, because we have found a solution.
    }

    set V[point.x][point.y] to "false"
    pop the stack and decrement stack_size by 1 //assume failure and go back

    return false
}
```

You should pass in the following:

- Your three-dimensional array that holds the walls (M).
- Your two-dimensional array that holds whether the cells have been visited or not (V).
- Your one-dimensional array "stack" (stack), as well as an integer of the size of the stack (stack_size).
- The current point as a `cell` struct (one argument) or as x, y (two arguments). When making the initial call to `solve`, this should be S (the source point).
- The sink point (T) as a `cell` struct (one argument) or as x, y (two arguments).

When this function completes (and returns `true`), `stack` (from 0 to `stack_size`) will hold the pairs of coordinates from the source S to the sink T. You can just loop through it and print the coordinates (in row, column order). If this function ever returns `false`, a solution doesn't exist.

5 An iterative approach to finding a path

This problem also has an iterative variation that can be done in main() with little variance from its recursive counterpart. You need the same data from the recursive variation, with the addition of a direction stack-like array variable (see #4 below) to keep track of your direction, since we lose the benefits of backtracking.

Setup: Do the following:

1. Make a `cell` struct that stores an x and y coordinate. Both of them can be integers. I don't care if signed or unsigned.
2. Assume the source S is at $(0, 0)$ and the sink T is at (Ncols $-$ 1, Nrows $-$ 1). For instance, considering maze M above, $S = (0, 0)$ and $T = (1, 1)$.
3. Make an array to mimic a stack-like data structure. Let's name it "stack". To do this, either `malloc` or `new` an array of `cell` sized Nrows $\times$ Ncols (the max possible size of a valid maze path). Then, make an integer to keep track of the current position you're at in the stack. Call that "stack_size". If you push an element, this "position" is incremented by 1. If you pop an element, it's decremented.
4. Make another array to mimic a stack-like data structure. Let's name it D[Ncols $\times$ Nrows]. This time, store an integer (or a `DIR_T` if you did an enum like me) in it rather than a `cell`. This stack is for the current direction of a particular point in the path. You do not need to have a "size" integer for this one, since we can use the one in the previous stack-like data structure here (they both grow and shrink at the same time).
5. Make a 2D array of integers to keep track of whether a `cell` has been visited or not. We'll name it V[Ncols][Nrows] for "visited" (lab writeup uses `iswhite`). By default, make sure all elements are set to 0. If $V[0][0] = 0$, then the cell at $(0, 0)$ hasn't been visited. If it's 1, the cell has been visited.

With recursion, we had each function call keep track of the current direction so we wouldn't have to push it to any kind of stack. Now that we are able to keep track of a point's current attempted direction via our own stack variable, we can act without recursion (essentially mimicking it). It's a bit more work compared to recursion, but it can be done:

```
main() {
    /* read in data and set up walls */

    /* begin DFS */
    cell point, to;
    stack[0]'s x and y = S //First iteration's point starts at Source
    D[0] = DIR_LEFT        //First iteration's direction starts at left
    V[S.x][S.y] = true     //Source has been visited
    stack_size = 1         //Stack has Source, that's 1 element

    while (stack_size > 0) {
        while D[stack_size - 1] == 4 (1 past directions 0, 1, 2, 3)
            set visited for point at top of the stack to "false"
            decrement stack_size by 1

            if (stack_size == 0)
                break //no solution is possible

        point = stack[stack_size - 1] //Get the current point

        if point.x = T.x and point.y = T.y
            break; //we hit the sink

        compute to.x and to.y for the direction in D[stack_size - 1]
        increment D[stack_size - 1] by 1

        if M[point.x][point.y][D[stack_size - 1] - 1]
            continue

        if V(to.x, to.y) has been visited already
            continue

        //push "to" to the stack
        stack[stack_size]'s x and y = to's x and y
        D[stack_size] = DIR_LEFT
        V[to.x][to.y] = true
        increment stack_size by 1
    }

    /* end DFS */
    /* print out coordinates in row, column format to file */
}
```

Just like with recursion, the results will be stored in "stack" from 0 to "stack_size". Loop through and print out the points to get the path from source S to sink T.