Name(s): Stephen Allegri and Zhiwei Zheng

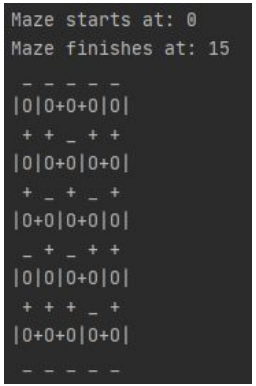
Structure and Development: Two main functions and one helper file were used to generate this program, both to generate and solve a maze. Loops were used when necessary, but recursive functions seemed (to us, especially for maze generation) a good way of emulating the solving/backtracking requirements for this assignment. Many different functions were used (like generate_path() and backtrack()), and the main() function acted primarily as a glue to sequence these functions and make the program work overall. As a final point, this program takes main() function parameters in the form of row size, column size, and .txt file to write to. Some notable programming “techniques” used include structs, functions, recursive functions, loops, header files, random number generation, and conditionals.

```c
typedef struct Node {

    int N;
    int E;
    int S;
    int W;
    int previous;
    int next;
    int visited;

} Node;
```

```c
int backtrack (Node *maze_ptr, int nRows, int nCols, int end) {

    if (maze_ptr[end - nCols].visited == -1 && (end - nCols) >= 0
        || maze_ptr[end + 1].visited == -1 && ((end % nCols) + 1) < nCols
        || maze_ptr[end + nCols].visited == -1 && (end + nCols) < (nRows * nCols)
        || maze_ptr[end - 1].visited == -1 && ((end % nCols) - 1) >= 0) {

        return end;

    } else {

        backtrack(maze_ptr, nRows, nCols, maze_ptr[end].previous);

    }
}
```

```c
int start (int nRows, int nCols) {

    return rand() % (nRows * nCols);

}
```

Maze Generation (Stephen): My approach to creating the maze was to treat each node as a struct containing information like storing locations of adjacent nodes, what the “next” node in the path is, and whether or not the node has been visited. The number of these nodes is determined by the row/column sizes. Once nodes are generated, all connections are set to -1, indicating that no change has happened (and this is for all variables within the struct N, E, S, W, next, previous, and visited). After the “matrix” or, more specifically, array of nodes, is initialized, a random start is determined. Generate_path() begins at this random start and visits nodes, connecting them, until a node is reached that is surrounded by visited neighbors. Once this happens, backtrack() is called and traces the steps followed by generate_path() until a node with at least one unvisited neighbor is reached. Generate_path() is then called again on this node. Repeat until the entire matrix/array is filled in. Once this happens, connection data and specifics are written to the maze_data.txt file and a rough sketch of the maze is printed (see below- “|” and “_” indicate walls, “+” indicates a connection).



Maze Solving (Zhiwei):

I started with how to read the adjacency list data in the txt file, the difficulty is that the amount of data in each row is not certain, I can't simply use fscanf to read. What's more, I need to read each row of data and distinguish their functions, and I divided the data in the adjacency list into four functions, East, West, north, South, and store them into a matrix. I use the while loops to achieve it, each column in the matrix represent one direction.

Then I use the information from this matrix to do the search. I use the Breadth First method to traverse all possible points around from one point radially, the result also be stored in a matrix and the number represent the step it go. It is worth to mention that in order to keep the read number from returning to its original path, after each direction instruction is read, I replace it with -1 from 1. This operation also helped me backtrack.

```
while (1)
{
    while (1)
    {
        if (a[k] == n + 1) // EAST
        {
            matrix[n][0] = 1;
            k++;
            continue;
        }
        if (a[k] == n - 1) // WEST
        {
            matrix[n][1] = 1;
            k++;
            continue;
        }
        if (a[k] == n + nCols) // SOUTH
        {
            matrix[n][2] = 1;
            k++;
            continue;
        }
        if (a[k] == n - nCols) // NORTH
        {
            matrix[n][3] = 1;
            k++;
            continue;
        }
        else
        {
            break;
        }
    }
    n++;
    if (n == size)
    {
        break;
    }
}
```

Test data

```
1    4 4
2    10 15
3    4
4    2 5
5    1 3
6    2 7
7    0 5 8
8    1 4
9    7 10
10   3 6 11
11   4 9 12
12   8 13
13   6
14   7
15   8
16   9 14
17   13 15
18   14
```

```
The map of step to finish:
8 5 4 3
7 6 1 2
8 9 0 3
9 10 11 12
```

Maze Solving (Zhiwei):

Finally, in order to print out the path from the starting point to the end point, I used a backtrack approach. I record all the passed points by the counting down in another matrix, and judging the signal -1 which I left in the previous direction reading, to help me judge whether the next point is related to the present point or not.

Our final results will be presented in the following format. If there are multiple possible results, our backtrack matrix can also record the situation of another route in the next row (this case is only one solution).

In summary, through the use of matrices to record the relative information, to help me with breadth-first-method search. Many loops like while and for loops are used to traverse all the points.

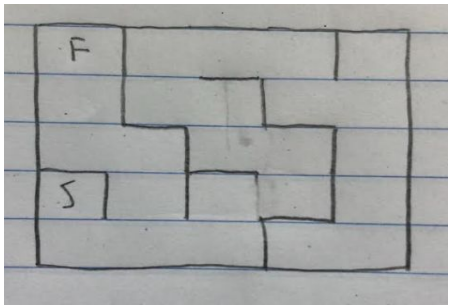
```
lawn-128-61-47-243:C zhengzhiwei$ ./a.out data.txt
Start point: (2,2);
Finish point: (3,3);
The number of step to finish: 12

The map of step to finish:
8 5 4 3
7 6 1 2
8 9 0 3
9 10 11 12

Backtrack of all possible path (From left to right is finish to start):
15 14 13 9 8 4 5 1 2 3 7 6 10

The track of the point:
(2,2)->(1,2)->(1,3)->(0,3)->(0,2)->(0,1)->(1,1)->(1,0)->(2,0)->(2,1)->(3,1)->(3,2)->(3,3)
lawn-128-61-47-243:C zhengzhiwei$
```

```
for (int j = 1; j < q; j++)
{
    for (int k = 0; k < 4; k++) //  Check where does the point come from, mak
    {
        check = 0;
        if (matrix[backtrack[i][count - 1]][k] == -1)
        {
            if (k == 0)
            {
                backtrack[check][count] = backtrack[i][count - 1] + 1;
                check++;
                backtrack[check][count] = 0;
            }
            if (k == 1)
            {
                backtrack[check][count] = backtrack[i][count - 1] - 1;
                check++;
                backtrack[check][count] = 0;
            }
            if (k == 2)
            {
                backtrack[check][count] = backtrack[i][count - 1] + nCols;
                check++;
                backtrack[check][count] = 0;
            }
            if (k == 3)
            {
                backtrack[check][count] = backtrack[i][count - 1] - nCols;
                check++;
                backtrack[check][count] = 0;
            }
```

Evidence of working maze generation (Stephen):

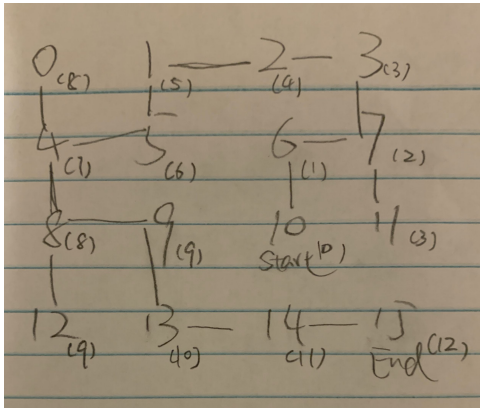
```
Maze starts at: 0
Maze finishes at: 15
 _ _ _ _ _
|0|0+0+0|0|
 + + _ + +
|0|0+0|0+0|
 + _ + _ +
|0+0|0+0|0|
 _ + _ + +
|0|0|0+0|0|
 + + + _ +
|0+0+0|0+0|
 _ _ _ _ _
```

~



Based on the program output key (“|” & “_” are walls, “+” is a connection), an equivalent, correct, maze can be generated (as seen on the right). This is a small 5X5 example, but it works on scales of 100X100 and above. Side note: the F and S of the drawn picture are mixed up...

Evidence of working maze solution (Zhiwei):



The data I used to test have a tricky part.
When backtrack from the step 10th to 9th, you can see in my case there are two possible value 9 and 12, but number 12 have not relative to number 13. So I in my code add a loop to judge the relative between current point to the previous one, which by using the -1 (The passed signal) I left.

Division of Labor:

Stephen - Wrote main_generate.c and generate.h, consolidated codes, README, and half of the slides.

Zhiwei - Do the part about solving maze and reading the adjacency list data.