Maze Solver

Description:

Given a file that contains a maze represented by '.' as an open path, '#' closed path; 'S' the start of the path and 'E' as the end, we need a tie to solve this maze 'find a way from the start point to the end '.

Two Methods are used to find this solution:

First: DFS

DFS, depth first search, a type of search that depends on visiting the deepest point first as it takes the first point then the second all in the same branch until it reaches the end after that it goes back to the nearest parent to go throw its other branches.

Algorithm:

- Using a stack, first we push the start point in the stack then we pop it to push all the adjacent point with visited value true then pop again and push the adjacent of the popped point as then we only go in one direction until it's finished.
- To find the path, I used a queue to enqueue the point I reached with every step so in the end, we have a queue contains the path which we can get by dequeuing the queue.
- The ordered pushed in the stack is taken as an assumption. First I push the right, left, up and down. so, the way the DFS takes in going around any point is down, up left and right.

```
➢ S = new Stack;
P = new point;
Q = new Queue;
   S.push(start);
Found = false;
While (S.size >0 )
{
p = S.pop();
Q.enqueue(p);
if(P equals end)
{
Found = true ;
break;
}
for all the adjacents to P
S.push(new point adjacent );
}
If(found){
```

```
Print Q ;
}
```

Second BFS :

BFS, Breath First search, a type of search that goes through all the points that are one step away then two steps away. As it moves layer by layer, level by level. As a result, it finds the shortest way because in every level the length of the path between the current point and the start is equal. As it reaches the answer it will stop cause no necessary to get all branches.

Algorithms:

- Using a Queue, first we enqueue the start point then we dequeue to enqueue all the adjacent of the point marked visited. In this way, we are sure that we visit all the points in the save level before going to the next one.
- To find path. I used a class 'NodeMaze' that represents the maze point that has two attributes:
    1. point position: the x and y for the point which is the row and the column of the cell.
    2. Point parent: the xx and y for the parent of the current point.
- When getting the parents I use a stack to push all the points then in the end I have a stack that contains the path.

```
➢ S = new Queue;
P = new NodeMaze;
    S.push(new NodeMaze (start, null));
Found = false;
While (S.size >0)
{
p = S.dequeue();
if(P.position equals end)
{
Found = true;
break;
}
for all the adjacent to P
S.enqueue(new NodeMaze (new point adjacent, P));
}
If(found) {
Q = new Stack;
While (P.parent ! =null) {
Q.push (point p);
P = p.parent;
}
Print Q;
}
```

The Maze solver design:

Two classes:

NodeMaze:

Class has two attributes and a constructor.

1. Point position: the position of the point (x, y)
2. Point parent: the parent of the position point (x, y).
3. Constructor: takes two parameters (position, parent).

MazeSolver:

Class has 3 attributes and 6 methods (functions) :

1. Point start: the start point position.
2. Point end: the end point position.
3. Char 2D array: has the maze as a single char per cell .
4. Method SolveBFS: Takes a file a return the solution of the given maze in 2D integer array using BFS so the solution is the shortest way .
5. Method SolveDFS : Takes a file a return the solution of the given maze in 2D integer array using DFS so the solution is not the shortest way .
6. Method Checkvalid: Used in the SolveBFS , SolveDFS methods to check the bounds of the array while getting the adjacents points and checks if it's visited or not and if it's an open cell.
   So, it only return true when the cell is in the bounds , not visited and not equal '#' (available cell).

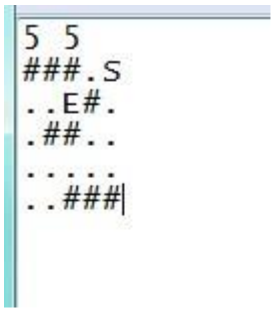
7. Method ReadFile : takes a file read all the data return a 2D char array of the cells , while reading , I get the place of the start and end points in order to store them in the attributes and checks if the maze is not correct (has more than one start and one end or none).Also Checks if the input row and column matches the input array of Strings .

8. Method getPath (Stack):takes a stack which has the path and return an 2D array of integers and this in the BFS method.

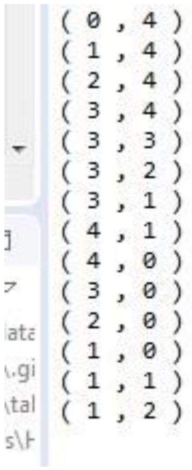
9. Method getPath (Queue):takes a Queue which has the path and return an 2D array of integers and this in the DFS method.

Sample Runs :

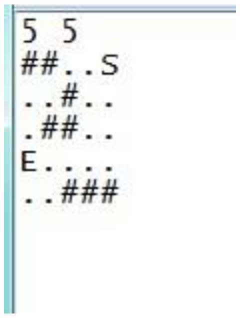
DFS :



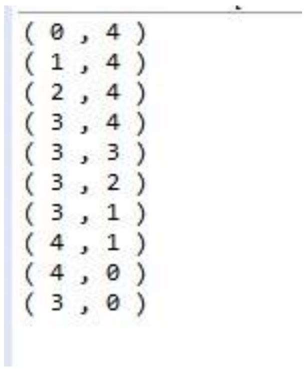
```
5 5
###.S
..E#.
.##..
.....
..###
```



```
( 0 , 4 )
( 1 , 4 )
( 2 , 4 )
( 3 , 4 )
( 3 , 3 )
( 3 , 2 )
( 3 , 1 )
( 4 , 1 )
( 4 , 0 )
( 3 , 0 )
( 2 , 0 )
( 1 , 0 )
( 1 , 1 )
( 1 , 2 )
```

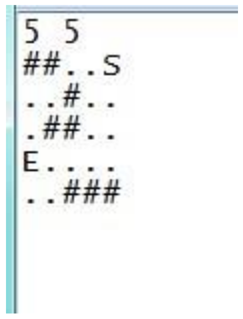


```
5 5
##..S
..#..
.##..
E....
..###
```

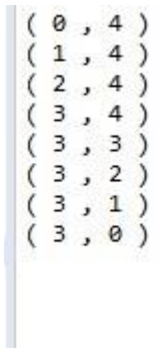


```
( 0 , 4 )
( 1 , 4 )
( 2 , 4 )
( 3 , 4 )
( 3 , 3 )
( 3 , 2 )
( 3 , 1 )
( 4 , 1 )
( 4 , 0 )
( 3 , 0 )
```

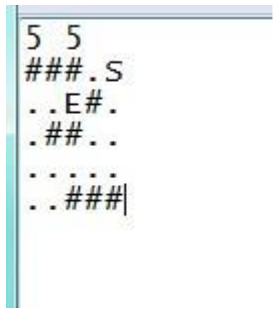
BFS :



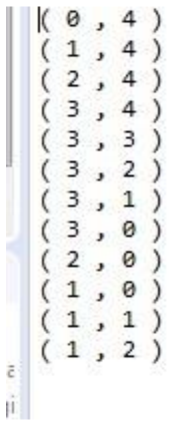
```
5 5
##..S
..#..
.##..
E....
..###
```



```
( 0 , 4 )
( 1 , 4 )
( 2 , 4 )
( 3 , 4 )
( 3 , 3 )
( 3 , 2 )
( 3 , 1 )
( 3 , 0 )
```



```
5 5
###.S
..E#.
.##..
.....
..###
```



```
( 0 , 4 )
( 1 , 4 )
( 2 , 4 )
( 3 , 4 )
( 3 , 3 )
( 3 , 2 )
( 3 , 1 )
( 3 , 0 )
( 2 , 0 )
( 1 , 0 )
( 1 , 1 )
( 1 , 2 )
```