Solving a Maze-based Game with Graph Theory

Ricardo Frumento

University of South Florida
Tampa, FL
Fall 2021

Contents

1 Introduction

For this project it was required to implement graph theory and solve a maze-based game where each cell of the maze has a direction and the possible movements are jumps of three or four cells. The steps discussed below are creating the graph from the input provided and finding a possible path.

2 Discussion

The input file consists of a text file with $r+2$ lines. The first one has two integers indicating the number of rows and columns respectively, the second one has two other integers that indicate where the game starts. The remaining lines are the rows of the maze with special characters representing the directions (W, N, E, S), if the cell is unreachable (X), and the goal (J). To create the graph the first step was figuring out what kind of graph and an implementation that can make the next steps easier. As the requirement is to solve the game a directed, unweighted, and unlabelled graph is enough. The vertices will be the cells, the edges represent the paths to reachable cells. Cells marked with X in the input will be present in the graph but will not have outgoing edges. The chosen implementation was adjacency list, which consists of a list with all the nodes where every node has its neighborhood attached. In this case, the neighborhood can be formed of zero, one, or two vertices because if the cell is an X it will not have edges arriving in any other cell, if the movement goes out of bounds it might have one or zero edges outward, and finally if both movements are valid, the vertex will have two outgoing edges. After the decision was made one more preparation step is required, create a continuous vector where the index represents the cell and the content represents the direction of the cell. Now it is possible to come up with an algorithm to create the graph.

```
   Data: vector map
   Data: int r
   Data: int c
   Result: vector adjList
 1 graph(map, r, c:
 2 |   Vector adjList;
 3 |   for v in map do
 4 |   |   Check the string stored on v;
 5 |   |   if string is a movement and does not go out of bounds then
 6 |   |   |   Store new vertex at the same position in adjList that v is in
   |   |   |    map;
 7 |   |   if string is a unreachable cell then
 8 |   |   |   Go to next vertex;
 9 |   |   if string is goal then
10 |   |   |   Store position of string;
11 |   |   end
12 |   end
13 |   Append the position of goal cell at the end of adjList;
14 return adjList;
```

Once the graph representation is implemented the next algorithm required is one to find a path. Here the choice was BFS. BFS goes through the graph checking the neighbors of the current vertex and only exploring them once the parent vertex's level was completely explored. This is due to the fact that BFS uses a queue to store the discovered graphs. To store the path first it is necessary to store the parents of each node, in other words the vertex that discovered them. After that it is just a matter of, after starting at the goal, finding out the subsequent parents that lead to the starting node.

```
   Data: vector adjList
   Data: int start
   Data: int end
   Result: vector path
 1 bfs(adjList, start, end:
 2 |   Queue queue;
 3 |   Vector path;
 4 |   Enqueue start to queue;
 5 |   Mark start as visited;
 6 |   Distance to start is zero;
 7 |   while queue is not empty do
 8 |   |   dequeue queue and store in v;
 9 |   |   for u in neighborhood of v do
10 |   |   |   if u is not visited then
11 |   |   |   |   Mark u visited;
12 |   |   |   |   Enqueue u to queue;
13 |   |   |   |   Distance to u is distance to v + 1;
14 |   |   |   |   Store parent node of u as v;
15 |   |   |   end
16 |   |   end
17 |   end
18 |   for w = last; w ≠ start; w = parent of w do
19 |   |   Add w to path;
20 |   end
21 return path;
```