Analysis of maze generation and path finding algorithms

Hanna Britt Parman & Kristiina Keps
Institute of Computer Science, University of Tartu

UNIVERSITY OF TARTU 1632

Abstract

This project focuses on implementing and comparing different maze generation, maze solver and shortest path algorithms.

Introduction

In this project we implemented two maze generation algorithms: depth-first search with flooding and randomized Prim's algorithm. We also compared five maze path finding algorithms: random mouse, wall follower, Pledge, recursive and Trémaux algorithm and four shortest path algorithms: Dijkstra, A*, depth-first search and breadth-first search. We give some examples and the results of testing the speed and number of steps of these algorithms.

Graph-based pathfinding algorithms

Breadth-first search (BFS) - iterates through each neighbour node before moving to the next level

Depth-first search (DFS) - explores as far as possible before backtracking (reverse strategy to BFS)

Dijkstra's algorithm - first the distance to the root vertex is set as 0 and the distance to other vertices is set as ∞. All of those vertices are added to the set Q. First vertex is then set to be processed. From there onwards the vertex v with the smallest distance is pulled. For the neighbouring vertices of v in set Q, alternative distance is calculated. The distance for a neighbouring vertex t is the sum of the distance of v and the distance from v to t. If that distance happens to be smaller than the current distance, it is set as the new distance. The algorithm terminates when the set Q is empty.

A* search algorithm - it is a goal-oriented variant of Dijkstra's algorithm, where instead of using the smallest distance when considering which node to process next, we use an evaluation function f(n) = g(n) + h(n), where g(n) is the path covered so far from root to n and h(n) the estimated distance from n to goal.

Implementation of graph-based algorithms for maze solving

A vertex class object is created for each m x n pixel in the maze and the vertices are added to m x n array. The vertex class has following parameters: height and width coordinate, distance, value (RGB), previously visited vertex and processed truth value.

The heap queue for Dijkstra and A* is implemented using the module heapq. The stored values for A* and Dijkstra are tuples containing the newly calculated distance and the vertex itself. When pulling an element, the vertex processed truth value is checked to see whether to process it.

For DFS, list is used as a LIFO structure with .pop() (retrieves and removes last element of the list) called when retrieving the next element to be processed. Contrarily, for BFS, list is used as a FIFO structure, where every element to be processed is called with pop(0) (retrieves and removes first element of the list).

Maze path finding algorithms

Random mouse - at every junction choose the next path randomly.

Wall follower - one hand is placed on the wall and then at every junction we follow that wall.

Pledge - go in one preferred direction until an obstacle, then place hand on the wall and keep count of the turns (positive degrees for clockwise, negative for counterclockwise). Once the sum of turns is 0 and we're facing the preferred direction again, remove hand from wall and continue.

Recursive - recursively check connected cells.

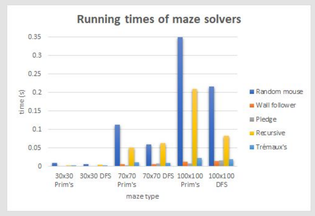
Trémaux - mark the path the first and second time you traverse it. Never take a path that's been marked twice.

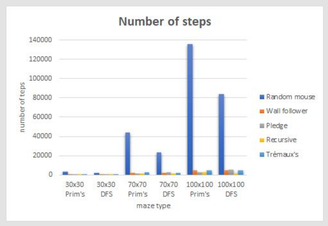
Comparison of maze path finders

We conducted 100 tests with 3 different sizes of mazes. The average of the tests is used in the comparison.

* The recursive algorithm exceeded the recursion depth limit in many cases with 70x70 and 100x100 DFS generated mazes and the results from those tests were not obtained, so the data presented is not accurate in these cases.

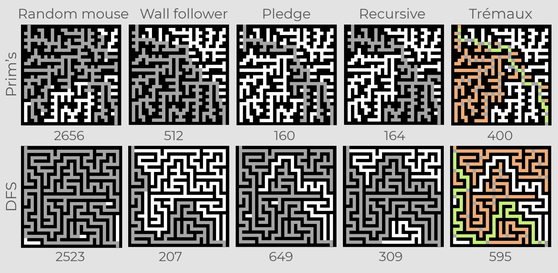
As expected, the random mouse performed the worst in terms of both time and number of steps. The recursive algorithm was the second most time consuming one. Not to mention with larger DFS mazes the maximum recursion depth got exceeded in many cases.





The Pledge algorithm was a clear winner in terms of both time and number of steps in the mazes generated by the Prim's algorithm. In case of the DFS mazes, the wall follower performed best.

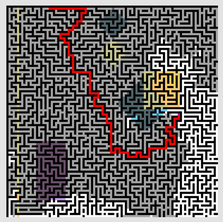
Examples of mazes, paths and the number of steps



Dijkstra and A* comparisons in a maze with and without obstacles

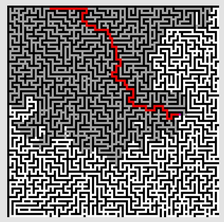
Shown on 100x100 maze, generated by the DFS algorithm with 5% of initial walls deleted, example:

Dijkstra



with obstacles

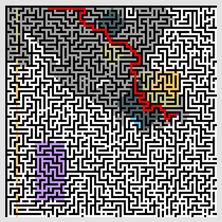
Coverage: 4136 Path length: 200



without obstacles

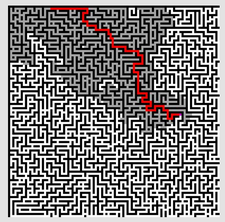
Coverage: 2572 Path length: 136

A*



with obstacles

Coverage: 1569 Path length: 140



without obstacles

Coverage: 1535 Path length: 140

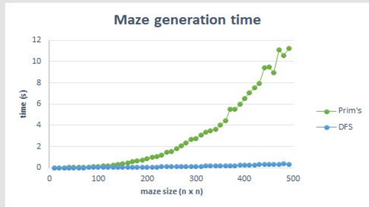
Maze generation

Randomized Prim's algorithm - First we start with a grid full of walls. Then we select a random point and make it a cell. We add the walls around the cell to a walls list. Then we start picking walls from the list until there are none left. For each wall we pick, we make it a cell instead and add the walls around it to the walls list.

Depth first search (DFS) with flooding - we start at a random point and append its' 2-step neighbours (2 tiles in a vertical or horizontal direction) to a stack of tiles to visit. The initial tile will be a cell in the maze. Next we remove a tile from the stack and repeat the same process with this tile. Only this time, the 1-step neighbour and the tile taken from the stack will both become cells. It's important to note that a tile can be added to the stack only if the tile isn't a cell and the 1-step neighbour tile isn't a cell. So under these circumstances there will be a moment when there are no available neighbours to add. The remaining tiles become walls.

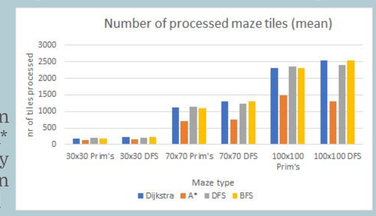
In our implementation we also added an extra feature of special tiles that have a lesser or greater cost than a normal tile to see how these obstructions change the coverage and shortest path. The special tiles were sea, desert and ice with costs 4, 2 and 0.5. The second addition was removing 5% of the walls for the mazes we created for graph-based algorithms

To compare the running times of the maze generation algorithms we generated mazes with sizes ranging from 10x10 to 500x500 (always square). An average of 10 tests was taken. As seen on the graph, in our implementations, the running time of the DFS algorithm grows much more slowly than the Prim's implementation.



Comparison of graph-based algorithms

Alike the tests of maze path finders, 100 tests were conducted with 3 different sizes of mazes. The same mazes were created in a same shape with and without obstacles (henceforth special and normal maze) with Prim's and DFS methods.
Dijkstra and A* were ran on both type of maze and the results when comparing the coverage between the normal and special maze are quite interesting. For example, for 100x100 mazes the average coverage for Dijkstra on a Prim's algorithm generated graph was ~2300 (for both special and normal mazes). But interestingly, there were cases where the coverage for the normal maze was over a 1000 larger than the special one and vice versa. On average, both algorithms had a larger coverage on the special maze and there were no big differences when compared from the maze generation algorithm point of view.
When compared on normal mazes, A* shows significantly better results in coverage compared to Dijkstra, BFS and DFS. The coverage on the same sized differently generated graphs doesn't seem to differ much



Conclusion and acknowledgements

Out of the two maze generation algorithms: Prim's and DFS, the DFS was faster and the running time grew more slowly.

The Pledge algorithm achieved best results with mazes generated by the Prim's algorithm, in case of DFS mazes, the wall follower was best.

From the graph-based algorithm, A* showed us the best results on coverage. The coverage on a maze was very similar for all algorithms when comparing the two generating methods.

Link to the code repository: https://github.com/kristiinakeps/pathfinding

This project was funded by the University of Tartu.