CSE643 Artificial Intelligence - Course Project
AI for solving mazes

Naveen Attri
Roll. No. 2015064

Abstract—**The project is focused on efficient algorithm to solve mazes. The project has two components: Maze generator and Maze solver. Maze generator is DFS based algorithm with few tweaks as mentioned in algorithms section. Maze solver itself has two parts: Maze optimizer and actual maze solver(shortest path finder) which is based on A* search algorithm. Both the algorithms, i.e. maze generator and maze solver, run in $O(M \times N)$, where M and N are dimensions of the maze grid. The plots for actual running time are shown in results section. One of the objectives was to be able to solve 100×100 size grid in real-time, which it is able to do with average time 0.49 seconds.**

I. Motivation

Maze solving algorithms are important because a lot of other problems can be reduced to maze solving problems. Some examples include path finding games or any other field that needs efficient real-time path finding. Path finding problems can be solved by this algorithm by tweaking the optimizer algorithm. Designing fast and efficient path finding algorithm looked like an interesting problem and somewhat challenging. So, this was the motivation behind taking this project.

II. Objectives

The project had a few objectives to be achieved. All of them have been achieved. Details are as follows:

- **Develop a maze generator:** Developed a maze generator based on DFS with little tweaks as mentioned in algorithms section.
- **Able to solve perfect/imperfect mazes:** Solving here refers to not just finding a path from start to goal, but find the shortest path from start to goal. The final code is able to solve both types of mazes. Solving perfect mazes is a trivial problem, but not imperfect mazes. To speed up the solving algorithm, designed a maze optimizer which reduces maze to a sparse graph while greatly reducing number of nodes. On an average, number of nodes in resulting graph is 5% of number of cells in whole maze grid, and 10% of the number of walkable cells in the maze grid.
- **Solve 100×100 sized grid in real time:** The solving algorithm is able to solve 100×100 sized grid in real-time, taking 0.49 seconds on an average. Plots are in results section.
- **Able to solve large grids**: The solver can solve large grids too. Tested on 1001×1001 grid, which has 10,02,001 cells in the grid. Solves it in 9sec. Initially it wasn't able to solve this big grid, reaching recursion limit of python(can be edited by programmer, but not a good idea), so I shifted code from recursive to iterative approach, allowing the algorithm to be able to solve mazes as big as 1001×1001.

III. Algorithms

The project has two parts: maze generator and maze solver. Maze generator is DFS based and maze solver has an optimizer algorithm and a solving algorithm, latter is based on A* search algorithm. Other than these algorithms, implemented a shell script to plot the results(shown in results section) which calculates time taken by each component and plots the results. Individuals parts are explained in next subsections.

The maze generator code is in ***generate.py*** and maze solving code is in ***solve.py***. The file ***graph.py*** contains implementation of graph used by solving algorithm. Some utilitiy methods required almost all files in the project folder are put separately in ***util.py***. The file ***main.py*** calls methods from all other files. To run the code, set the grid size in ***main.py*** and execute ***bash run.sh***(alternatively *python main.py*, but it doesn't clean the directory after execution). To perform only particular functions(like maze generation only or solving only), the file to use would be ***performance.py***(for details on how to use it, check ***evaluate_performance.sh***).

A. *Maze generator*

Maze generator is based on DFS. The key-points about the maze-generator algorithm are as follows, stressing more over parts that are different from standard DFS:

- Used matrix to represent the grid, with states(i.e. empty, wall, start, goal) represented by value at that cell
- Used DFS fashion for traversal followed by backtracking to generate paths in grid
- Didn't use stack for backtracking, instead saved the data in another matrix of dimensions same as grid - this part doesn't make much difference
- After DFS algorithm is done with making the paths, the generated is a perfect maze. Made it imperfect by dividing grid into boxes of uniform size and randomly deleting such a wall that removing it creates more paths in the maze(placed checks to make sure that removing a cell creates additional paths not in original grid). For each box, algorithm will keep trying until it finds a cell it can delete to create more paths. This method has additional advantage that the imperfection in maze is uniform because of dividing maze into boxes of same size and making sure extra path is created in each box.

- Time complexity of the algorithm is $O(M \times N)$, as evident by points above and further verified by plots in the results section.

B. Grid optimizer

The optimizer optimizes the grid to a sparse while greatly reducing the number of nodes. On an average, the number of nodes in graph returned by grid optimizer is 5% of the total number of cells in the grid, which is 10% of the number of walkable cells in the grid. One run of optimizer algorithm for 25×25 grid is shown in Fig. 1. Note that the dark blue cells are the cells that the optimizer algorithm will make a node in final graph. Note that a path starting from one blue cell terminates only at one other blue node. This means that we can consider that path as edge and that is basis of optimization of the algorithm. Further, the grey paths are path that will never be used(Removed multiple edges between two nodes, keeping only the shortest one). The algorithm works by traversing nodes in DFS fashion, keeping track of when a point comes where there is a fork/dead end in the path. At such points, create a new graph node, add the path from previous node to current node as edge, and then continue traversing with current node taken as previous node for next encounter of fork or a dead end, or stop if current node was dead end. The algorithm was initially recursive, but now changed to iterative because of reaching high recursion depths.

C. Maze solver

Grid solver takes as input the graph generated by the maze generated and runs A* algorithm on it. The plan was to first use standard A*, and if it has memory problems, then use SMA*. The optimizer algorithm greatly reduces the memory requirement by reducing the number of nodes. Now, if we use SMA*, it would unnecessarily make it slow(depending on the limit we put on its memory), while it already runs fast without memory problems for grids as big as 1001×1001. So, I implemented simple A*, which gave good results, so didn't implement SMA*. The time it takes for solving the maze(including the time taken by optimizer algorithm) is $O(M \times N)$, which is further verified by plots in results section. Solution of maze refered to in previous subsection is also present in Fig. 1.

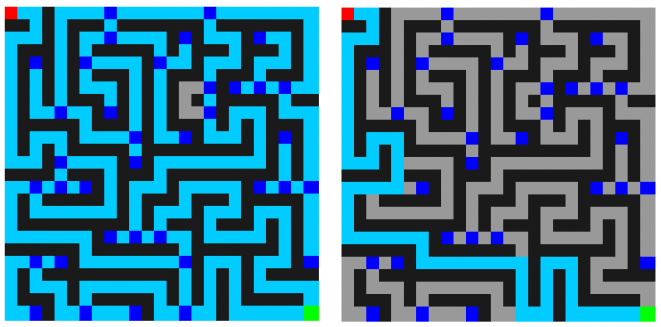

Fig. 1. Left: Optimized maze, Right: Solved maze

IV. Results

The key-points of the obtained results:

- Can find shortes path from start to goal in perfect and imperfect mazes.
- Can solve 100×100 grids in real time(avg. time 0.49sec)
- Can solve grids as big as 1000×1000 in 9sec.
- Performance of maze generator and solver(optimizer included) is $O(M \times N)$, which is same as dimension of grid.

A. Performance of Generator

Time complexity is $O(M \times N)$, which is evident from algorithm and below plots for square grids. Left one is ***time** vs **edge length*** and right one is ***time** vs **grid size i.e.*** $M \times N$. As expected, left curve is parabolic and right one is linear. The values on $x - axis$ show grid dimensions.

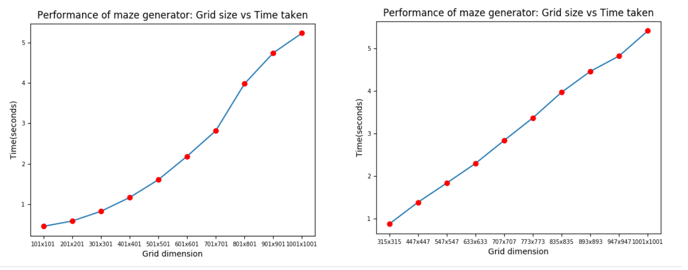

Fig. 2. Perfromance of maze generator

B. Performance of Solver

Time complexity is $O(M \times N)$, which is evident from algorithm and below plots for square grids. Plots same quantities as above plot, but for solver(including time taken by optimizer). All of these and above plots are also available in project folder.

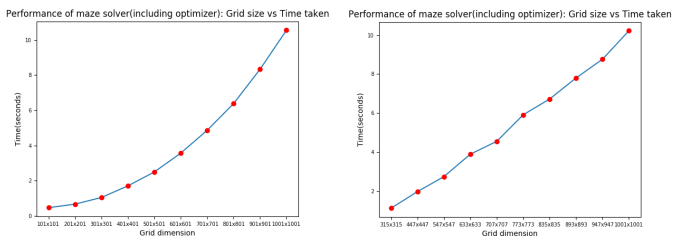

Fig. 3. Perfromance of maze generator