Maze Generator and Solver

**Author:** Wenhao Li

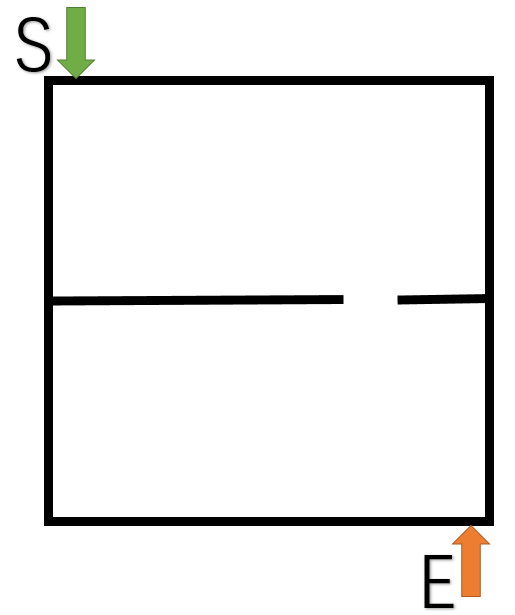
Contact: [wl85v@mail.missouri.edu](mailto:wl85v@mail.missouri.edu)

**Abstract:**

Maze solving has always been a good testing case to for various searching algorithms. In this graduate project of Artificial Intelligence course, I decided to develop a maze solver using the searching methods we have learned, breadth first search, depth first search, uniform cost search and A star respectively. Maze can have loops, so all these methods I mentioned are all graph-based.

**Introduction:**

A maze is a map that has paths and walls in it. There can either be one or more than one solution called solvable maze or no solution, called a dead maze. In my project, in order to simplify concepts, assume the maze always starts at the top left corner and exits at the bottom right corner, indicated as below.

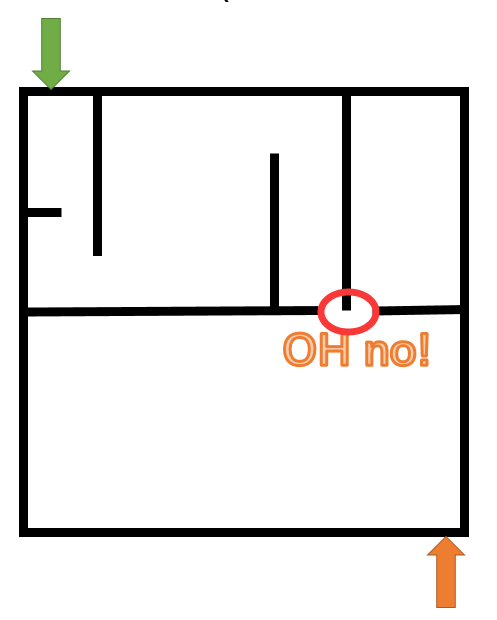


In order to have a solution, the maze needs to have at least a path to go from the upper left to bottom right. By using this principle, we can generate a maze by adding lines and cut off a opening to it. Here, the lines are called walls and the openings are called holes.

Solving a maze is to find a path to go from the starting point to the exit point. There may exist more than one solution, but our goal is to find only one.

Algorithms:

For generating a maze, as I described above, we randomly add walls (which can be either horizontal or vertical) and cut holes in an empty space. One wall and one hole divide an empty maze into two smaller spaces. And then we recursively repeat this process to each of the two spaces until the smallest space size is reached. But there will be a problem. What if one newly added wall is right blocking the hole of a previous wall so that there is no path going to the new spaces?



In order solve this, we can record all the locations of the holes we previously created. So that whenever a new wall is placed, it checks the two ends of that wall to see whether it is blocking a previously placed hole. If that is the case, then the new hole on that wall must be placed side by side at the location. By this means, we ensure that there will be a solution to the maze.

A pseudocode of the generation process is as below:

Recursively\_Generate(upperLeft, bottomRight, holes[])

{

rows= bottomRight.row-upperLeft.role

columns= bottomRight.column-upperLeft.column

if Size(upperLeft, bottomRight) is smaller than a threshold

return;

if(rows>=columns)

Randomly place a horizontal wall in Range(rows)

else

Randomly place a vertical wall in Range(columns)

for end in wall.end[2]

if end is in holes

holes += end

Randomly select a new hole

holes += new hole

Recursively\_Generate(upperLeft, wall.end, holes)

Recursively\_Generate(wall.end, bottomRight, holes)

}

Generate()

{

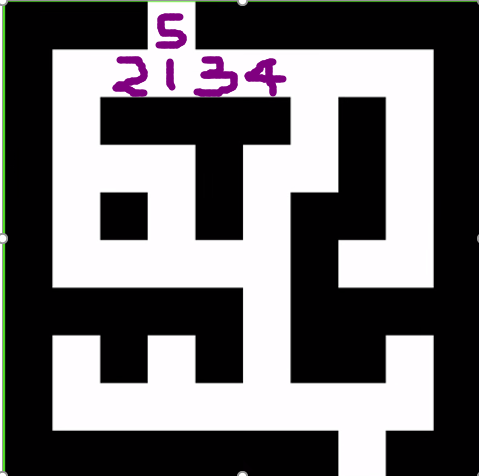
Recursively\_Generate();

}

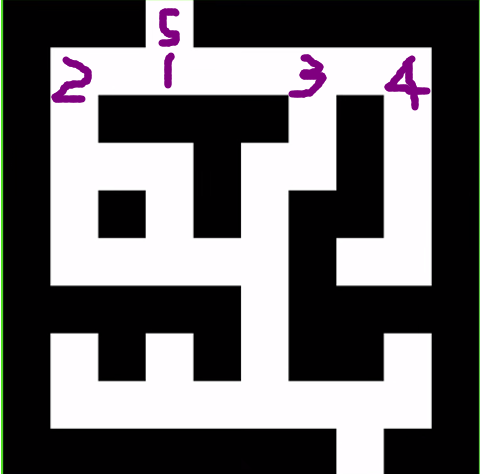
For maze solving, I implemented a list of four algorithms, which are all complete searching algorithms given enough time and memory.

* Breadth first search
* Depth first search
* Uniform cost graph search, using path length as the cost
* A\* search search, using Manhattan Distance as the heuristic

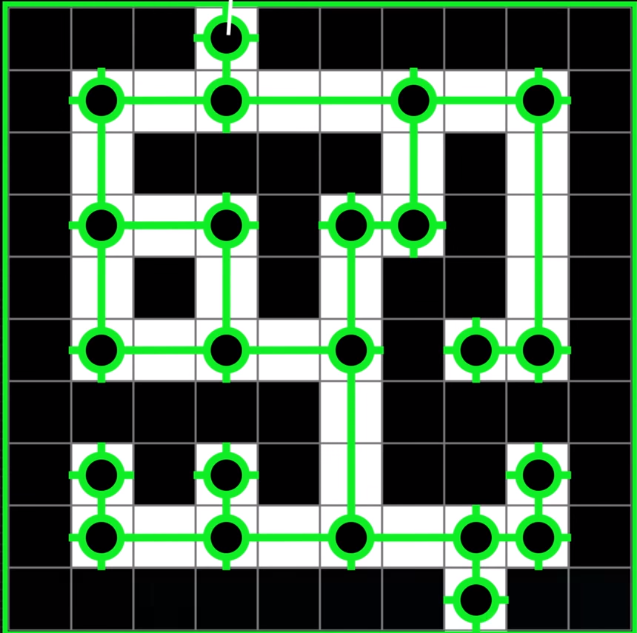
However, I further optimize the search methods after some analysis. For example, given a maze like this one, starting from point S, we expand this starting node S into node 1. Expand node 1 into node 2 and 3. But in node 3, there is no decision to be made. The only choice for node 3 is to go along the path to node 4. As a result, node 3 doesn’t need to be generated and put into memory. The same principle applies to node 2 and node 4.



After this quick analysis, the actual position of node 2, 3 and 4 can be placed as the figure below.



And the final representation of the whole maze can be reduced as below:



Comparatively, by this approach, much fewer node can be generated instead of generating a node at each single step. Intuitively, we can conclude that if the maze has more long corridor (meaning a long path without branches), the execution time of these search algorithms will be drastically reduced.

**Implementation:**

The programming languages I used is pure C++. For GUI, I am using QT framework with C++ to generate it. QT framework is a cross platform GUI framework. As a result, my program will be able to run on all major operating systems. I used some C++ 17 features to implement the algorithms, which may require a newer C++ compiler (MSVC 2017 or later on Windows, GCC 7.3 or later, Clang 5 or later on Mac or Linux) to compile.

In my implementation, I choose to follow the modulus development approach to divide up the maze and the solving part. For maze generation, it creates a maze defined by two parameters: rows and columns. For maze solving, the only goal is to expand the starting node to the end node.

Maze is represented as a 2D arrays of Boolean values, making it much easier to draw on the screen.

class Maze

{

rows;

columns;

maze[rows][columns];

};

Struct Node

{

CurrentPosition;

};

Class solver

{

Maze\* maze\_pointer;

Node\* currentNode\_pointer;

Void BreadthFirstSearch();

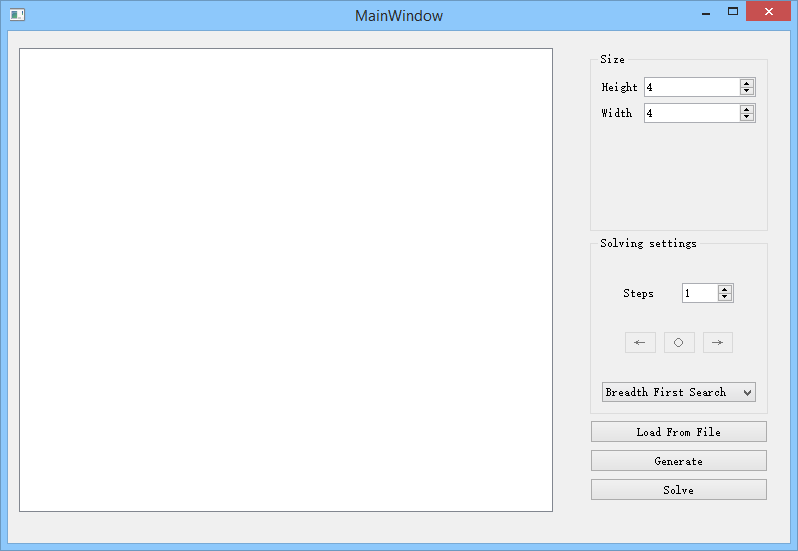
Void DepthFirstSearch();

Void UniformCostSearch();

Void AStarSearch();

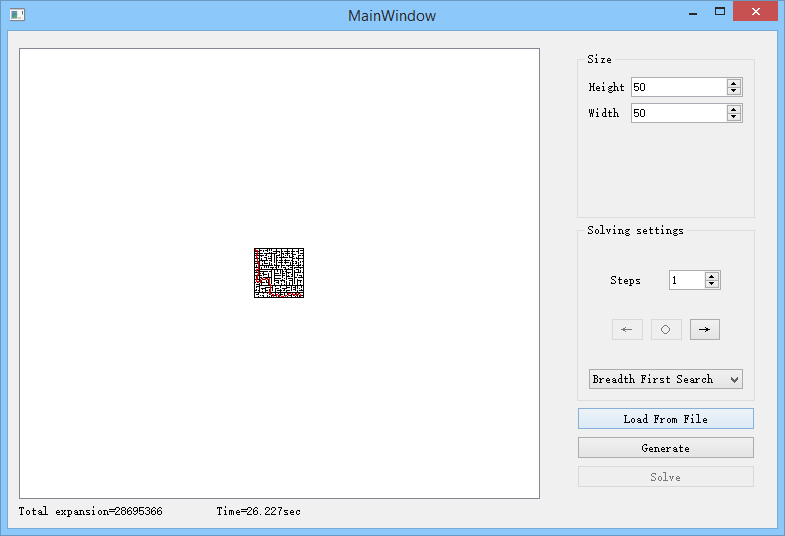
}

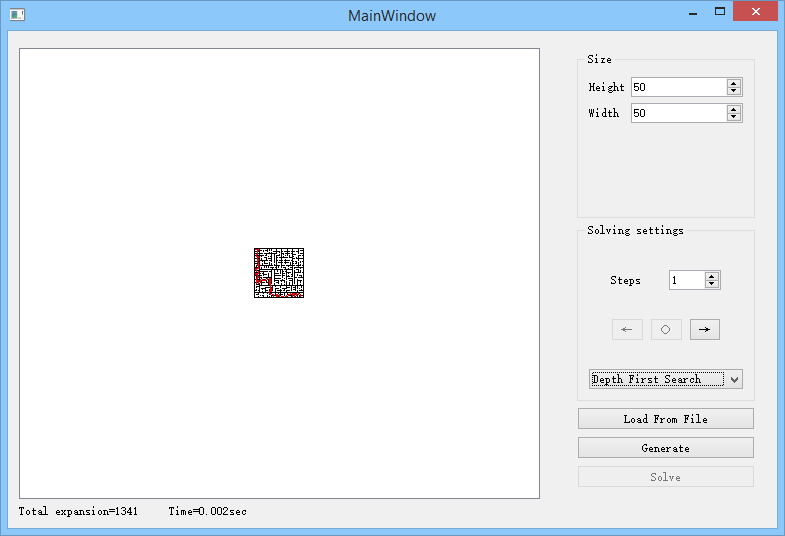
**Result:**

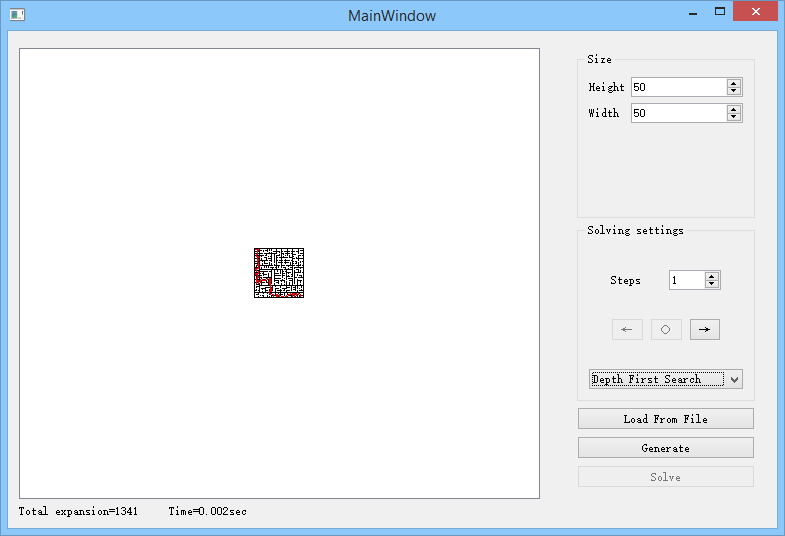


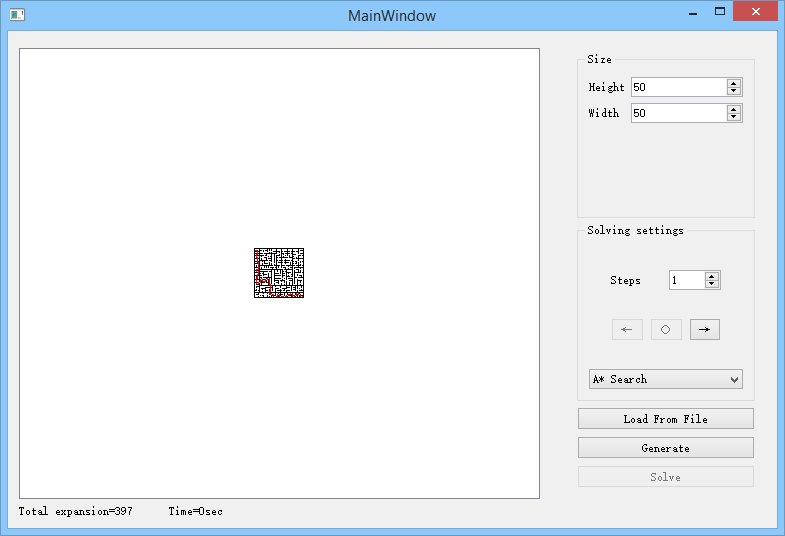
This is what my program is like when it is first starts up. The main blank area is for drawing the maze which has been generated. My program can generate arbitrarily large mazes and solve it. The right hand side are all the parameters and buttons with clear meanings. Information of node expansions and time consumptions will appear in the bottom as we will see.

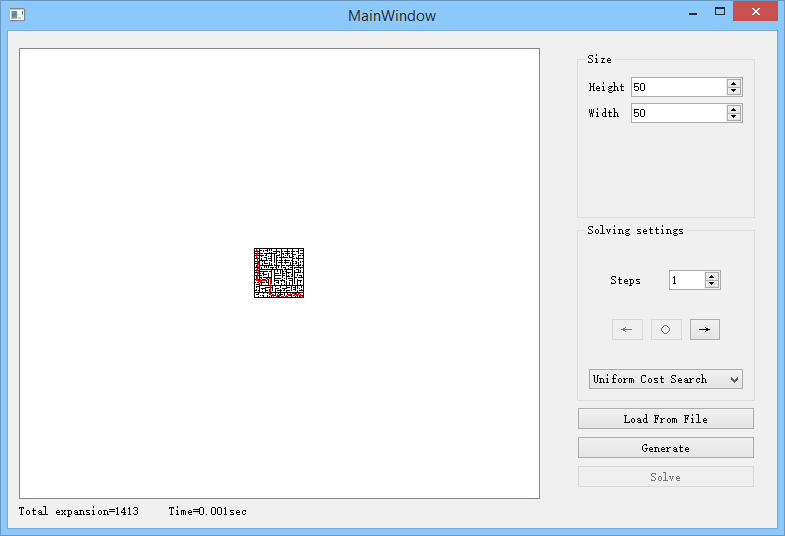
**50\*50**







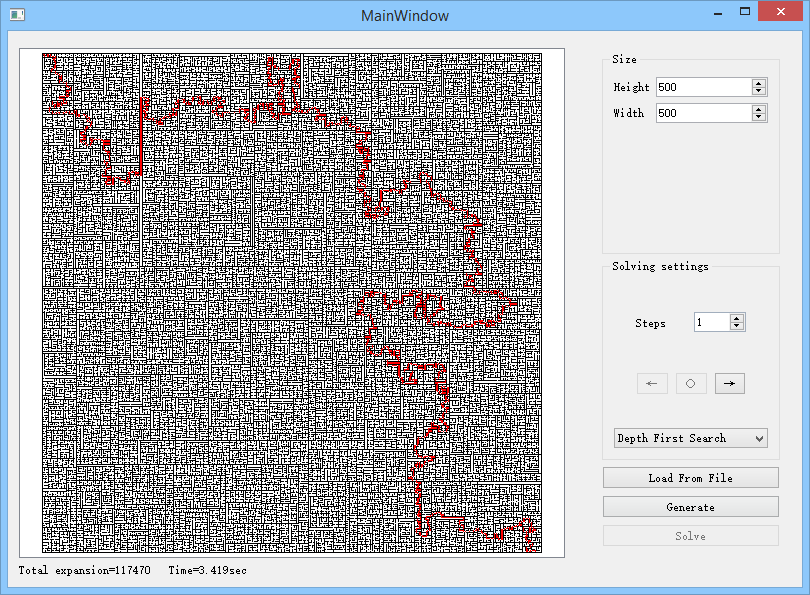


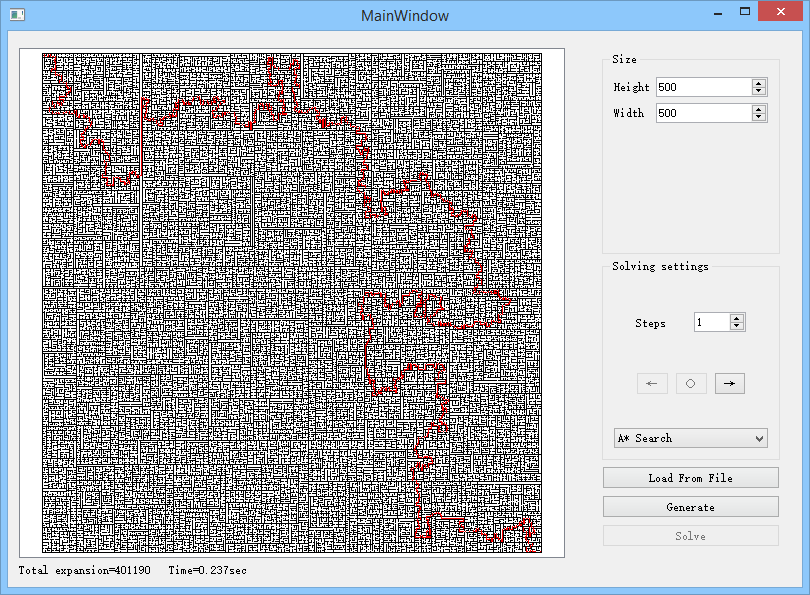


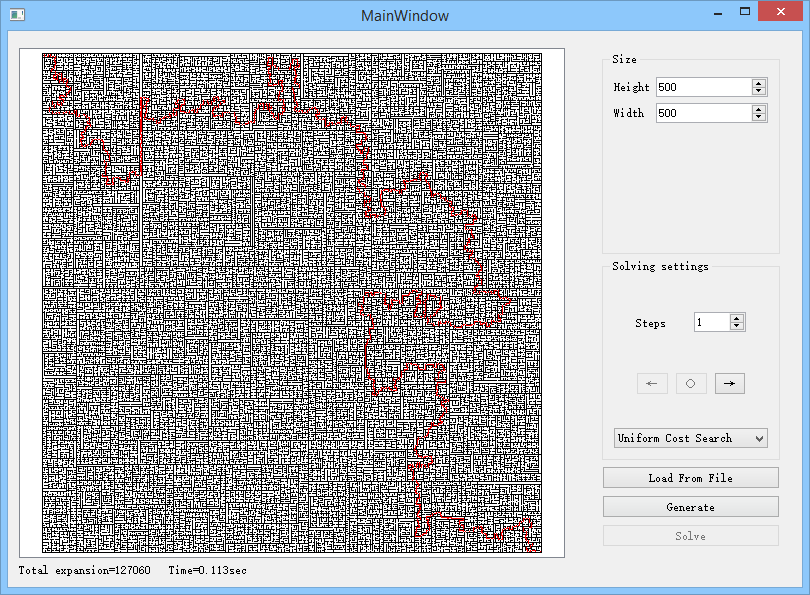
As we can see, solving a 50\*50 maze with breadth first search is already very challenging with normal computers, so it is not suitable for any larger mazes.

Let’s go for something harder. This time we omit breadth first search.

**500\*500**

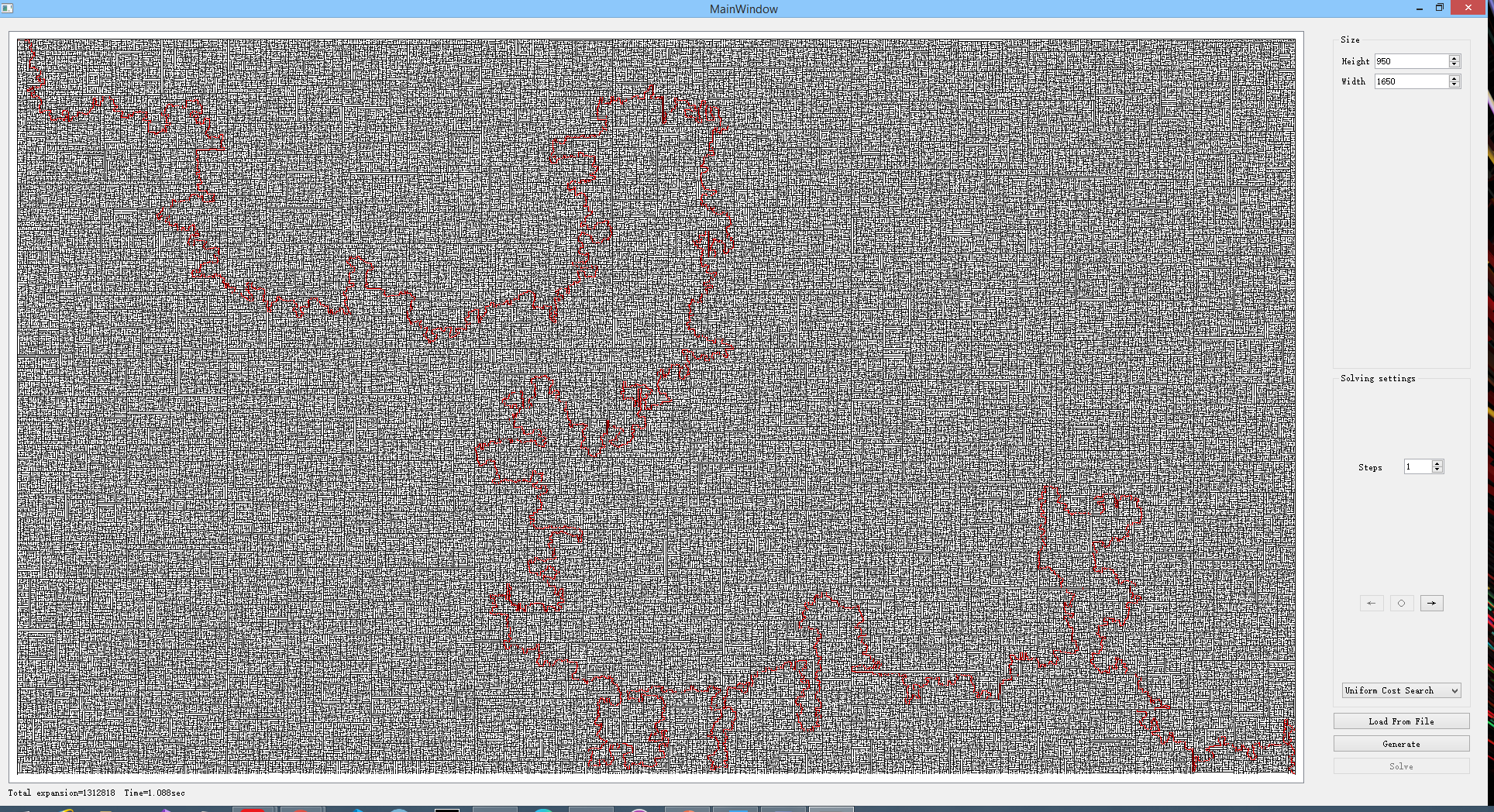


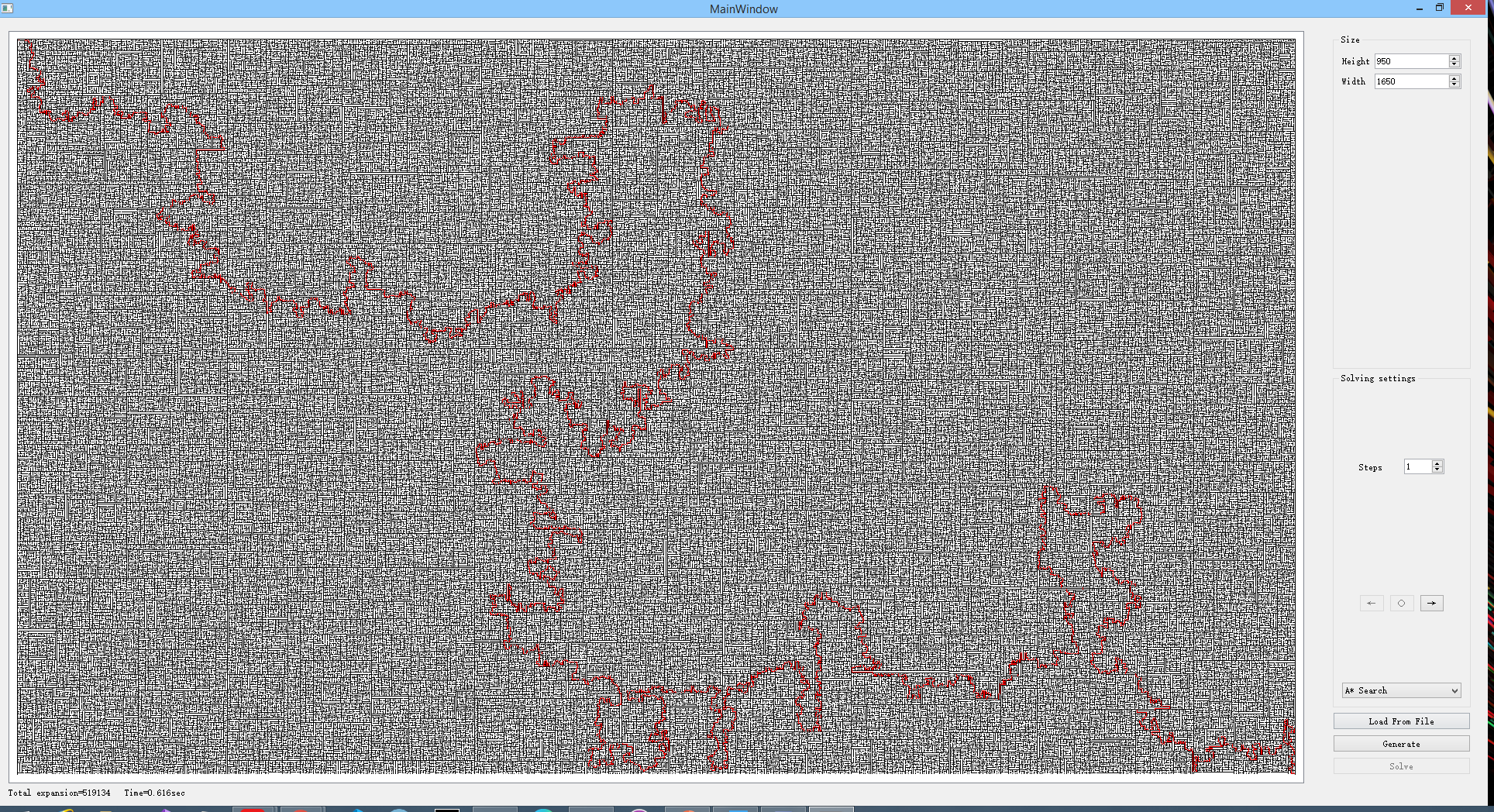




Let’s go for extreme cases because why not. This time, we omit breadth first search and depth first search, because these two search methods are close to brute force. The computation grows exponentially.

**950\*1650**





Below is a comprehensive graph of all the testing results.

**Summary and conclusion:**

As we can see intuitively from the graph, A\* search is very efficient when the problem size scales up. It always expands the least amount of nodes (in average case). And we may also notice that, depth first search is actually performing pretty well when the solution is dense, meaning it can quickly find a solution when the maze contains multiple possible paths.

**References:**

None.