Department of Computer Science and Engineering

(IIT Indore)

PROJECT REPORT

ANALYSIS OF MAZE SOLVING ALGORITHMS

June 3, 2020

Guided by :

- **Kapil Ahuja Sir**

Submitted by :

- **Ayush Agrawal** cse180001011
- **Harsh Chaurasia** cse180001018

CONTENTS :

- Optimization
    - Improving A* algorithm
    - Improving BFS
- Implementation and Results
- Conclusion
- References

Introduction :

- *Solving a maze using computers is a complex though enticing task as one needs to come up with an algorithm that is effective in terms of time as well as space for finding the shortest path.*
- *There are ample number of graph algorithms to solve this problem, each one surpasses the other in performance aspects.*
- *This project reviews different graph algorithms for maze solving along with their performance.*
- *It is generally a two step task.*
    - *Conversion of Image to graph data Structure.*
    - *Finding Solutions using naive or Standard Graph Search Algorithms (with or without heuristic).*
- *These Algorithms are of two types :*
    - *Uninformed Search Algorithms.*
    - *Informed Search Algorithms.*

Algorithm Design and Analysis :

Algorithms can be widely classified into two categories depending on the solution approach we follow:-

A) Naive Algorithms

Algorithms that do not follow any specialized approach or do not have a fixed structure for solving problems. We will not discuss them in detail here.

1. Random Mouse

Approach: The most primitive algorithm to solve mazes that can be implemented by any mouse or unintelligent robot. At every encountered node, it chooses a random direction. Eventually it will solve the maze but it can take a lot of time.

Complexity: Its complexities cannot be calculated because of no specific structure of the algorithm.

2. Wall Follower

Approach: As the name suggests, this algorithm always follows the path adjacent to the walls. This algorithm follows either left hand rule or right hand rule in calculating the final path. But its drawback is that this algorithm won't be useful when the destination cell is situated in the centre of maze i.e. having no contact with the outer boundary walls directly or indirectly.

Complexity: This algorithm has a very high space and time complexity because its approach is very naive and if maze is not simply-connected then it has no solution.

B) Graph Search Algorithms

These Algorithms are used by the computer to find the solution of a maze in a structured and computationally efficient manner. These algorithms are the matter of interest of this project.

1. DFS Algorithm

Approach: The Principal behind DFS is to analyse a path until we get to the goal or a dead end. In the latter case, we backtrack until we arrive at an un-analysed node and repeat the same. It is a sure shot way to find a path from source to destination given that such a path exists. However, it may not be the shortest.DFS is an optimal approach when it is known that the maze has atmost one correct path. (eg. Labyrinths) Complexity.

Algorithm: Pseudocode.

- Push start node into a stack. (and mark it as visited)
- While(Stack is not empty)
    - Remove the top of stack as curr node.
    - if(cur node is goal node)
        - Set Flag done = 1 and break the while loop.
    - Push all the un-visited neighbours into the stack.
- If (Done flag is true)
    - The remaining elements in the stack give the path.
- Else, The maze has no solution.

2. BFS Algorithm

Approach: The Breadth First Search Algorithm explores the maze in such a way that at every node, it queues every possible direction, traversing them in that order. If the search queue becomes empty before getting out, the maze is declared to have no solution. BFS provides the most optimal solution to the maze but its drawback is that it consumes a lot of memory and explores un-necessary nodes.

Algorithm: Pseudocode

- Push start node into a queue and mark it as visited.
- while(queue is not empty)
    - Pop out first element from queue and mark it as current node
    - if(current node is destination node)
        - set flag done=true and break the loop
    - Push all the adjacent non-visited nodes in the queue
- if(done is true)
    - We get the most optimum solution of the maze.
- Else, No solution for the maze exists.

3. Dijkstra Algorithm

Approach: Dijkstra's Algorithm is an algorithm for computing the shortest path between two vertices of a graph where all edges have nonnegative weight. It is based on repeatedly expanding the closest vertex which has not yet been reached.

Algorithm: Pseudocode

- Create two sets of nodes, **chosen** and **noble**. Chosen contains the selected nodes while noble will contain the remaining node.
- Assign distance of all vertices of noble set to infinite. Set dist of start as 0.
- While(there are vertices in nobel set)
    - Pick the node which has the least distance in the nobel set and add it to the chosen set (let it be current).
    - if(current is goal node)
        - Flag done = true.
        - Break while.
    - Else
        - Update the minimum distance of all visitable vertices in the nobel set from the chosen set.
- If done is true, the chosen set is the required path
- Else the maze has no solution.

4. FloodFill Algorithm

Approach: Flood Fill algorithm is one of the best maze solving algorithms. The flood fill algorithm involves assigning values to each of the cells in a maze where these values represent the distance from any cell on a maze to the destination cell. The flood fill algorithm mainly contains four parts: update walls, flood maze, turn determination and move to the next cell.

Algorithm: PseudoCode

- Assign value to each cell in the maze where this value represent the distance from any cell to the destination cell (destination cell is assigned the value 0).
- Initialize starting node to current node
- while(destination node is reached)
    - Detect the walls around each node and save them.
    - If (dead end is reached)
        - Update the values of the cell
    - else
        - Compare the cell's value with its neighbours and determine whether it is necessary to make the cell's value plus 1 and if so do it.
    - Now determine which movement should be taken.

5. A* Algo

Approach: A* unlike any previous algorithm is Informed search Algorithm. A* is a variant of Dijkstra's algorithm commonly used in games. A* assigns a weight to each open node equal to the weight of the edge to that node plus the approximate distance between that node and the finish. This approximate distance is found by the heuristic, and represents a minimum possible distance between that node and the end.

Heuristic Distance:

- It is calculated by taking the end maze into picture, and for a node **'n',** is given by :
    - F(n) = G(n) + H(n)
    - G(n) is the exact cost of a path from initial node to current node.
    - H(n) is the heuristic function that approximates the distance between the final node and the current node.

Algorithm: PseudoCode

- Create two sets of nodes, **chosen** and **noble**. Chosen contains the selected nodes while noble will contain the remaining node.
- Assign the distance of all vertices of the nobel set to infinite. Set dist of start as 0.
- While(there are vertices in noble set)
    - Pick the node which has the least "heuristic distance" in the nobel set and add it to the chosen set (let it be current).
    - if(current is goal node)
        - Flag done = true.
        - Break while.
    - Else

        - Update the minimum distance of all visitable vertices in the noble set from the chosen set.
    - If done is true, the chosen set is the required path
    - Else the maze has no solution.

Time Complexity Analysis

Standard symbols used : -

b -> Branching Factor (It means how many branches arise from each node in maze, We will consider that diagonal movement is not possible)

d -> Depth (It refers to the depth of the final path obtained from any algorithm)

V -> denotes the number of vertices in the graph

E -> denotes the number of edges in the graph

N -> length of grid

M -> width of grid

Wall Follower Algorithm

The worst case time and space complexity of this algorithm can be O(the length of maze).

• DFS Algorithm

The best case time complexity of DFS is O(d) when the DFS algorithm gets the final path in one go.

In case, when a dead end is encountered the algorithm needs to come back to the previous junction to take an alternative path. Its worst case Time Complexity can reach O(b^d).

The space complexity of DFS is all the possible nodes that can be occupied i.e. O(N*M).

This algorithm works extremely well for small and medium sizes mazes. It may not provide the most optimum path but it is a very fast algorithm. In the case of labyrinth, it can be the most optimum approach to find a solution. As the size of maze increases its performance deteriorates in comparison to heuristic algorithms.

• BFS Algorithm

In BFS Algorithm at each node we push all the neighbouring nodes in the queue, so it's complexities can be best defined in terms of branching of nodes.

Average case Time Complexity = O(b^d)

Average case Space Complexity = O(b^d)

BFS algorithm provides the shortest distance solution to the maze. But its space complexity is quite high. It will work well for small and medium sized mazes as space won't be an issue. But for large size mazes it is not a good option to use BFS as it would consume a lot of memory. In large size mazes heuristic approach should be preferred instead whose performance improves with size of maze.

Dijkstra Algorithm

On some analysis, we can reduce the time complexity of the first by updating the distance of only those vertices that are connected to the newly added node.

And since in a maze, each node can utmost be connected to 4 other nodes, we have

T = n * (4 + n)

Further, we see that by maintaining a minHeap, we can reduce the time complexity of the latter part to Log(n).

Hence, we Have

T = 4*n + nlog(n)

T = O(n *log(n)). { Taking into consideration that m <= 4*n }

FloodFill Algorithm

In worst cases this algorithm can fill the entire maze and entire maze has to undergo value changes so :-

Time Complexity = O(N*M)

Space Complexity = O(N*M)

A* Algorithm

The time complexity of A star algorithm is difficult to generalize, but an upper bound can always be obtained.

Worst case Time complexity = M + N log(N) (Similar to dijkstra)

Optimization

• Optimizing A*

1. Following are the changes that could be done to optimise the A* Search Algorithm:

● A simple onList flag can be set for each node thereby removing the need to search the Closed Set repeatedly, thus making the algorithm more time efficient.

● Open Set can be made a priority queue instead of an array, mainly because the size of the Open Set continues to grow over time and so the cost of finding and removing the smallest entry keeps increasing. Priority queue will help do the operations in O(logn) which is better than finding the smallest element in array in O(n).

● Adding neighbors with lesser H-distance to a different set from Open Set

2. **Selection Of Heuristics**
    - Selection of heuristic function is an important part of ensuring the best A* performance.
    - Ideally H is equal to the cost necessary to reach the goal node. In this case A* would always follow a perfect path, and would not waste time traversing unnecessary nodes.

- If overestimated value of H is chosen, the goal node is found faster, but at a cost of optimality.
- In some cases that may lead to situations where the algorithm fails to find a path at all, despite the fact, that path exists.
- If underestimated value of H is chosen, A* will always find the best possible path.
- The smaller H is chosen, the longer it will take for the algorithm to find a path.
- In the worst-case scenario, H = 0, A* provides the same performance as Dijkstra's algorithm
- Estimated heuristic cost is considered admissible, if it does not overestimate the cost to reach the goal

The different type of Heuristics used are:

- **Manhattan Distance**:

  h = abs(current_cell.x – goal.x) + abs(current_cell.y – goal.y)

- **Euclidean Distance:**

  h = sqrt((current_cell.x – goal.x)^2 + (current_cell.y – goal.y)^2)

- **Diagonal Distance**

  h = max(abs(current_cell.x – goal.x), abs(current_cell.y – goal.y))

Greedy BFS by Heuristic Techniques.

The regular BFS can somewhat be improved as follows

- Using a Priority queue instead of a regular queue.
- The priority will be assigned on the basis on greedy heuristic (distance from the goal node)
- At each step, proceed to the node closest to the goal.

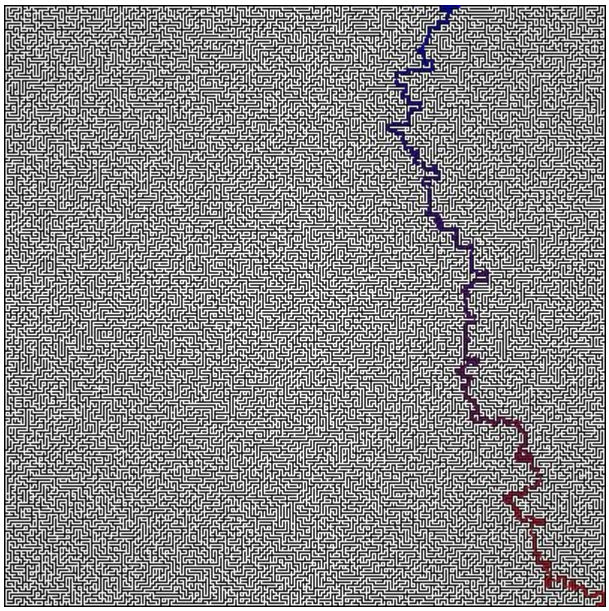
Optimizing Flood Fill Algorithm

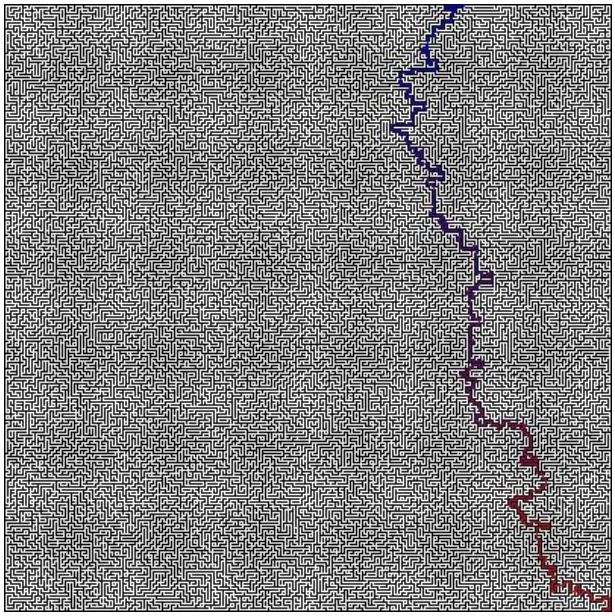
When maze solving is implemented by real robots, turning also becomes a very heavy operation. In the Flood Fill Algorithm the step in which the robot changes its cell has a constant priority for choosing the next cell if their values are the same. We can convert it into variable priority and always consider moving in direction such that the robot does not need to change its direction in case it has value equal to other cells. It can save time and computational costs.

Implementation and Results

- Solving 400*400 maze using various algorithms

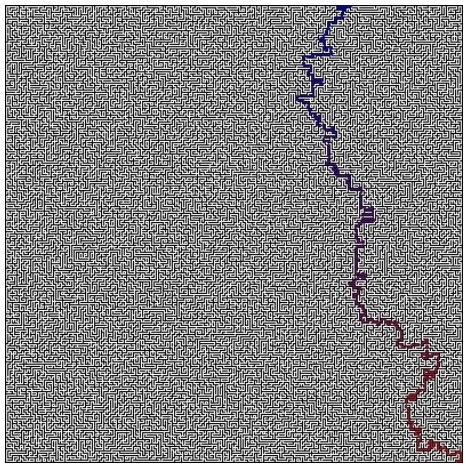
A* Algorithm ->

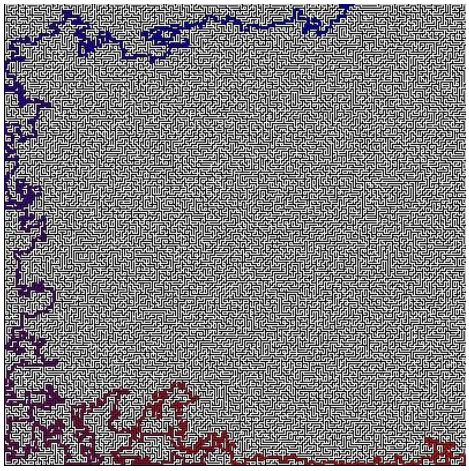




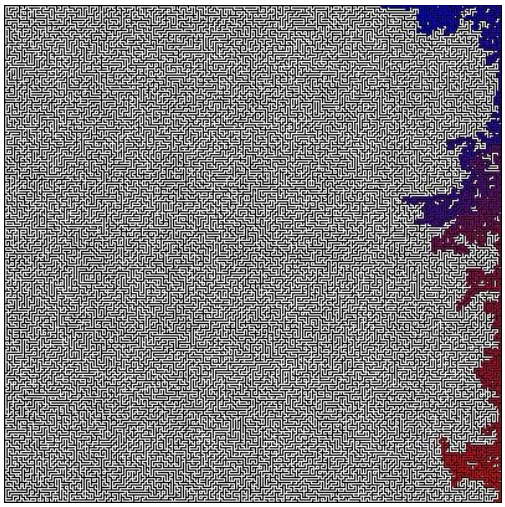
<- Dijkstra Algorithm

BFS ->



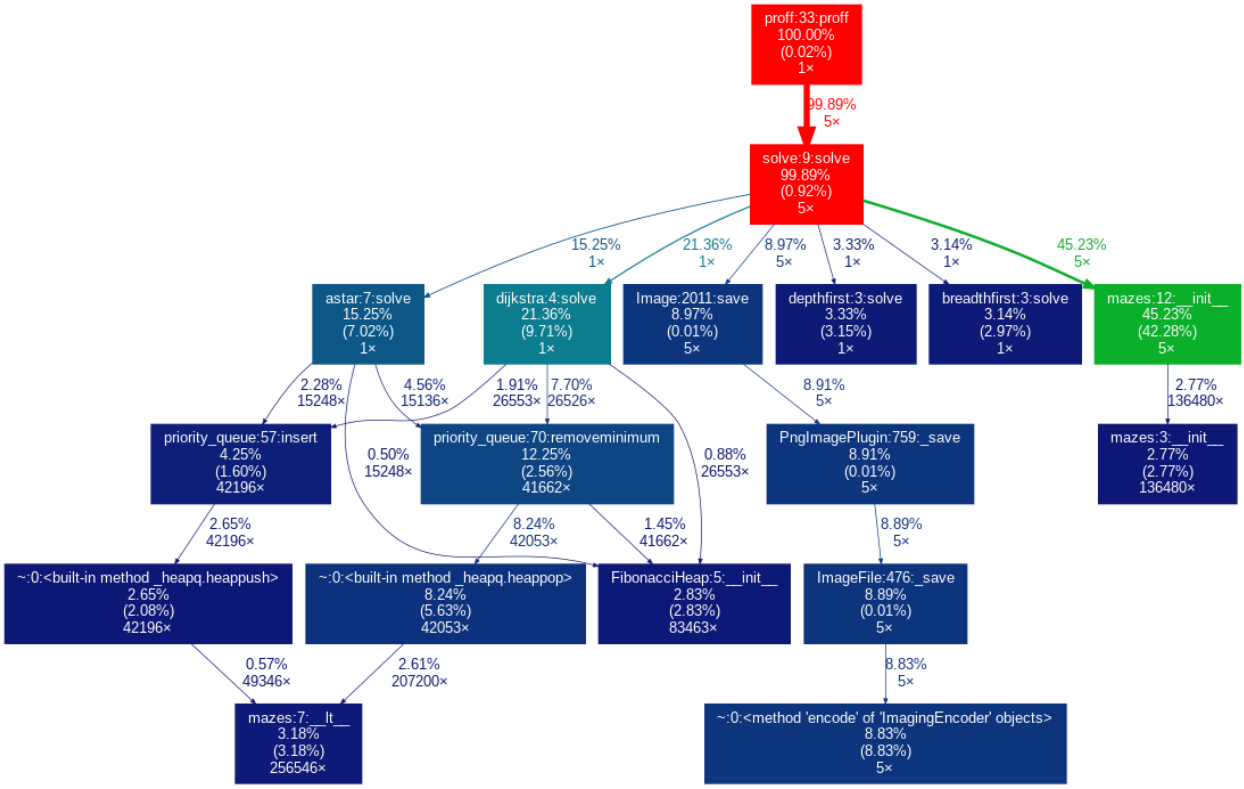


<- DFS



Wall Follower ->

Runtime Visualizer Chart



The above image shows what percentage of time is taken by each process when the program is run.

	DFS	BFS	Dijkstra	A*
braid_x400	0.009	0.042	0.267	0.285
perfext_x400	0.011	0.046	0.261	0.284
braid_x2000	0.149	1.177	9.123	7.028
perfect_x2000	0.374	0.915	8.036	6.751

Conclusion

It can be Concluded that -

- Each graph algorithm can be seen as an upgrade of the previous algorithm, in the order DFS -> BFS -> Dijkstra -> A star.
- It can be seen that BFS comes out to be the fastest maze solving algorithm that finds the shortest path.
- However, this only works with small mazes.
- Several papers have proved the actual superiority of A* algorithm, over any other algorithm, when it comes to large mazes (with 10k*10k size).
- We were unable to test with such mazes due to lack of computation power.
- Also, in the original paper in which it was declared, it was proved that a star algorithm needs to analyse the minimum number of nodes. Therefore in real life, where cost to explore a node is significant, A star algorithm has an upper hand.

References

- *P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136.*
- *Huijuan Wang, Yuan Yu and Quan Bo Yuan, "Application of Dijkstra algorithm in robot path-planning," 2011 Second International Conference on Mechanic Automation and Control Engineering, Hohhot, 2011, pp. 1067-1069.*
- *A. M. J. Sadik, M. A. Dhali, H. M. A. B. Farid, T. U. Rashid and A. Syeed, "A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory," 2010 International Conference on Artificial Intelligence and Computational Intelligence, Sanya, 2010, pp. 52-56.*
- *T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, "Introduction to Algorithms", Third edition, Prentice Hall of India, pp. 587-748, 2009.*
- *M. Pond, (2017, February 24). "Maze Solving - Computerphile, Youtube".*
- *Leo Willyanto Santoso, Alexander Setiawan, Andre K. Prajogo "Performance Analysis of Dijkstra, A* and Ant Algorithm for Finding Optimal Path Case Study: Surabaya CityMap".*