Maze Solving Algorithms

(Category: Computer Science)

Paintbrush Algorithm, Breadth First Search Algorithm, Depth First Search Algorithm, Trimming Algorithm & Compass Algorithm

Manish Chand
Semester – IV
Department of Computer Science
Maharaja Agrasen Institute of Technology
G.G.S.I.P.U., New Delhi.

Mayank Goel
Semester – IV
Department of Computer Science
Maharaja Agrasen Institute of Technology
G.G.S.I.P.U., New Delhi

Shantur Rathore
Semester – IV
Department of Electronics & Communication
Maharaja Agrasen Institute of Technology
G.G.S.I.P.U., New Delhi

Abstract

Maze solving - a seemingly minor challenge for the analytical minds of humans – has generated enough curiosity and challenges for A.I. experts to make their machines (robots) solve any given maze. Although, there are a lot of algorithms but we thought that we might not be having algorithm which we can easily say is the *fastest*. Here, we won't go on the record and say that, "Yes!! We have found the quickest way of solving a maze", but yes here are some methods which we feel will take the quest to find a perfect algorithm at least a step further. The question that automatically comes into mind is that why do we require our robots to solve maze? Is it really required? The answer is "Yes". Robotics is a field of wide reaching applications. From bomb sniffing robots to devices for finding humans in wreckage to home automation, we require our robots to have a certain degree of analytical mind and if our robots can solve mazes, as complex as it can get, then we think that their analyticity will be improved significantly. The existing algorithms have the problem that all of them concentrated on finding the shortest possible way present in the maze and this made these algorithms a bit slow. Here, consider the situation where the maze has only one solution. In this case, most of the existing algorithms failed to keep up the pace as they tried to find alternative routes at every junction to find a shorter path, and this made the algorithms slow. We tried to improve on this, and we deviced these algorithms keeping in mind that the time taken by them in finding the way out of the maze should be as less as possible. The algorithms that we will be discussing in full detail are Paintbrush Algorithm, Compass Algorithm, Trimming Algorithm, Depth First Search Algorithm and Breadth First Search Algorithm.

1. What is the problem with the existing algorithms?

The problem, as discussed in abstract, is that most of the existing algorithms try to find the shortest possible way and ignoring the time taken by these algorithms. The question is that why do all these algorithms concentrate on providing the shortest way? And, the answer is the concept of robotics competitions in which the robots compete in a race to complete the given maze. One of the competitions with the richest history is MicroMouse Competition.

MicroMouse Competition *(Ref. # 6)*

The MicroMouse Competition has existed for almost 20 years in United States, and even longer in Japan. Micromice are small, autonomous devices designed to solve maze quickly, and efficiently. The goal of the contest is simple: the robot must navigate from the corner of a maze to the target as quickly as possible. The actual final score for a robot is primarily is a function of the total time in the maze and the time of the fastest run. The most widely used algorithms in these competitions are: Random Algorithm, Left / Right Algorithm and Bellman Flooding Algorithm. We will be discussing all these algorithms and others.

2. How to define the complexity of a maze?

2.1 For Human Brain

The characteristic of human brain does not provide the strong senses of direction and position. The brain is easily confused by similar environments. Therefore, if a maze comprises of many junctions, dead ends and loops, the maze is considered to be a complex one for human.

2.2 For Robotic Brain

The robot or computer is able to remember the orientation and position correctly. This is due to the ability of the software algorithm and retrieval of data from memory storage. Therefore, a maze with large number of junctions, dead ends and loops is no longer complex for robot.

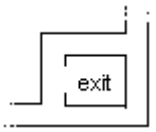
3. Existing Algorithms *(Ref. #1 & 2)*

3.1 Random Algorithm

This is by far the simplest way of solving a maze. Keep in mind that we are saying that it is the simplest algorithm and in **no way be the fastest** or most efficient. You simply have your robot running around making a random decision to turn or not when it encounters an opening to the left or right. The only problem with this, as mentioned above, is that it will be slow, and there is a good possibility that the robot will not find the exit in the time allotted i.e. your robot could wander for hours always taking the wrong turns. Needless to say, it is probably well worth it to invest some programming time into a better algorithm if your looking for speed or accuracy.

3.2 Left / Right Algorithm

This algorithm is the amateur programmers' favorite. The whole principle behind this algorithm is that you can solve any continuous, i.e. no "islands", maze by following either the right hand or left hand wall. **This will always get you out, unless the finish is an "island,"** like the picture below.

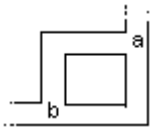


In the above maze, a robot using the left/right algorithm would never reach the exit. This algorithm is just slightly more complex to code, but its benefits over the random algorithm are large. Simply have your robot turn to the left (or right) whenever it encounters an obstruction. Again, the downside of this

algorithm is speed. One wall may continue for a long way before reaching the end.

3.3 Branch & Return Algorithm

The principle behind this algorithm is that by exploring each branch of the maze you will eventually find the exit. This algorithm requires that you "remember" when you come to a branch, and begin to record your steps from that branch. After exploring that branch and you come to a dead end, you simple follow your path back to the original branch and take the next turn. This algorithm requires much more coding, and some way of knowing your distance and direction**. The problem with this algorithm is that the robot could fall into an endless loop.** For instance suppose we had a maze that looked like this:



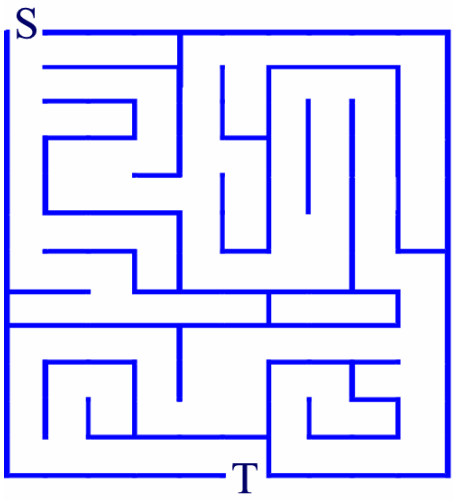
If the robot is heading from the top of the maze toward 'a' then it may decide to take a right and follow the corridor until it reaches 'b', it then might turn left and reach 'a' again, and then follow back to 'b', and never realize that it is going in a circle. One possible way to combat this is to have the robot take a random corridor when coming to an intersection. Giving the robot a degree of "forgetfulness", i.e. having it forget intersections encountered long ago could prevent it from being caught in a very long loop.

4. Our New Algorithms

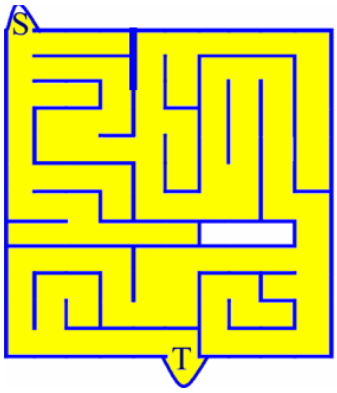
4.1 Paintbrush Algorithm

This is easily the highlight of the whole paper. This algorithm makes use of the concept of floodfill in paintbrush – that if there is an opening in a seemingly closed figure and if you fill it with color than whole of the painting gets filled i.e. the paint floods from that opening.

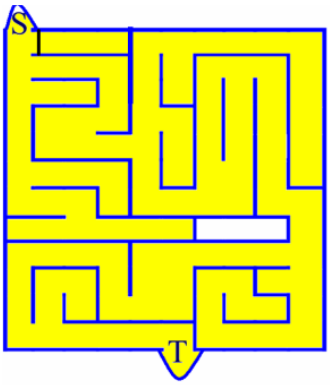
Now, here consider a regular maze as shown in fig.



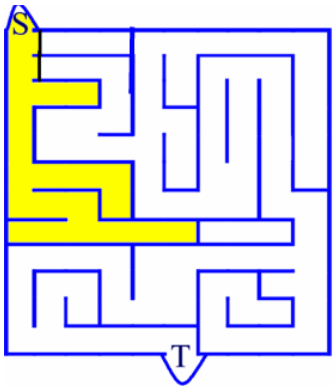
Here, S is the starting point and you have to reach T. Now, consider that this is a bitmap that you have opened in paintbrush. Now if you fill the starting cell (S) with any colour, the whole (or most part of the maze) will be colored as you can reach any point of the maze from one way or the other. And, if there exists a path from S to T then T will also be filled with that color. Now, at cell S, you have two options for where to go. Either the robot will go in right direction or downwards. Here, according to our Paintbrush Algorithm, if you block one of these ways and now fill the cell S with any colour and now also if the T cell is painted then this means that you can reach T through the path which you haven't blocked. Thus, you can directly neglect the whole of the maze that you can access through that blocked way at cell S. And, if the cell T is not painted, then this means that in order to reach T you have to take the path which you blocked. The following illustration will fully explain the concept to you.



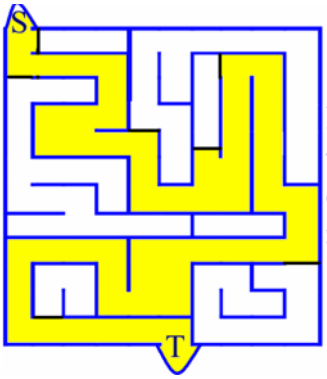
Here, we have just filled the original maze with yellow colour, and you can see that most of the maze has been painted.



Here, we have blocked the right way from block S and you can see that the target cell T has still been painted, so this means that the blocked way is more or less useless for us. So, we will simply ignore it and move ahead.



Here, you can see that on moving one step downwards to block S, we again blocked the right route and here you can see that the target cell, T is not painted. This means that the blocked way was the key to target, so here we can without any doubt ignore the downward route, this way our work has become really simple.



Now, we just kept on blocking and releasing the junctions and after blocking just six optional routes we got our solution which is highlighted in the figure by yellow colour.

So, here you saw that using this algorithm, we can easily solve any given maze, no matter what it contains i.e. junctions, loops, islands or anything for that matter. And, mind you it is also very fast. A systematic algorithm for this process is as follows:

1. Start scanning from first cell and scan for a cell with three openings. (Because if a cell has two openings then one will be used for coming and one for exit, so there won't be any problem).
2. As soon as you get a cell with three openings, select one of the openings and block it.
3. Flood fill the cell.

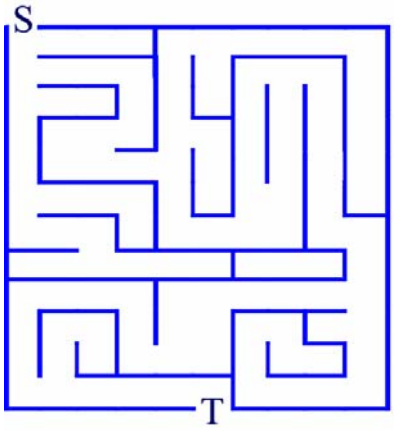
4. If the target is filled, then move ahead i.e. into the opening you haven't blocked.
5. Else if the target is not filled, this means that the blocked opening was the key. So release this opening and block the other opening and move into the released opening.
6. You will soon reach the target.

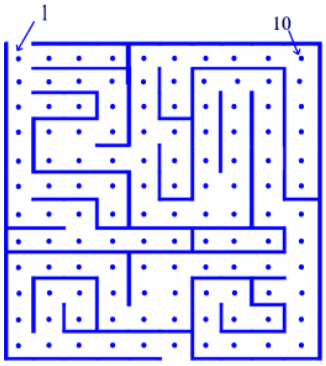
4.2 Breadth First Search Algorithm (Ref. # 5)

Breadth first search is a graph traversing technique. According to this technique, we select a root node and add it into a queue. Then, we will add its neighbours into the queue (from rear) and will remove one more element from queue (from front) and will add its neighbours into the queue. This way, we traverse the whole graph.

Now, we consider that this technique of graph traversal can be easily used to solve a maze. What you have to do is that, you have to convert the whole maze into a graph. And, this is how you have to do it:

1. Consider the following maze:

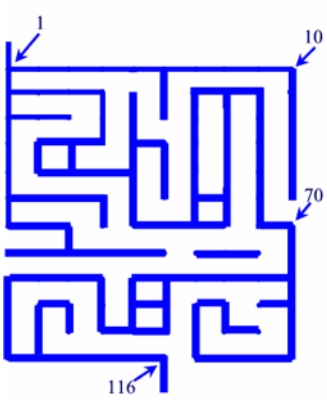




Now, just identify all the cells within the maze and mark them numerically.

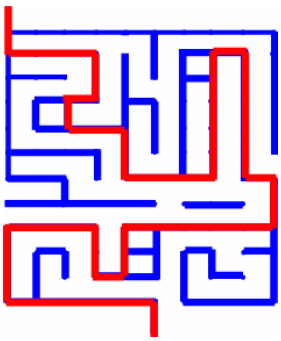
2. Now, consider the cell 1 and you can see that it has three openings: top, right and bottom. So, in the graph, instead of a cell, we will represent 1 by a dot and join this dot with the top, right and bottom cell (dot in case of

graph). The following illustration will explain the concept to you in full detail.



Here, see point 10, in maze 10th cell had an opening in left and bottom. So, you can see that in graph we have point 10 joined with its right and bottom neighbour.
In case of point 70, in maze it had opening in left and bottom direction and thats what is satisfied in the graph. And, in case of 116 (target), it had an opening in left and bottom direction and once again the graph is illustrating that only. So, this way we can make whole of the graph.

3. So, we have made our graph. Now, as we do in regular breadth first search, we will keep on adding elements into the queue and removing them and at the same time adding their neighbours into the queue. After all the vertices of the graph will be traversed, we will backtrack the traversed vertices, in accordance to their origin vertices and we will get different paths (if more than one path exist) to the target and the one with lesser number of traversed vertices will be shorter one.
4. The above example has only two ways to the target (116) and the shortest path will be:



The advantage of this algorithm above all algorithms is that it gives you the shortest path from any number of possible paths in a maze.

4.3 Depth First Search Algorithm (Ref # 5)

Depth first search technique is used to traverse the graph. The difference between breadth first search and depth first search is that in depth first search technique we make use of stack instead of queue, which is used in breadth first search.

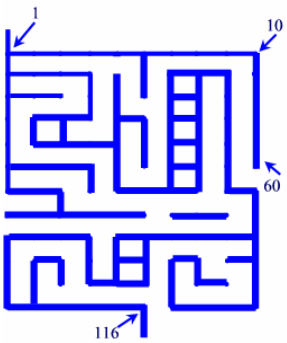
So, even depth first search technique can be used to solve any maze. But, the difference is that this algorithm merely gives you a solution of the maze, not necessarily the shortest one. But, depth first search gives you the solution much earlier than breadth first search algorithm, because in breadth first search you have to traverse all the vertices of the graph, but in depth first search, the process terminates as soon as you reach the target vertex.

Here also you have to make use of the same way of getting the graph from the given maze, as we did in the case of breadth first search. And, then we have to carry out normal depth first search on this graph with the use of stacks. And, as soon as you traverse the target vertex, you will have to backtrack the traversed vertices with the help of their recorded origins. Now, the question that might be troubling you will be that how will we get the origin of a vertex? The answer is simple: When you remove (in case of queue) or pop (in case of stack) a particular vertex (say A) and add its neighbours into the respective data structure then you will have to record somewhere along with these added vertices that they are added as neighbours of A and hence, their origin is A.

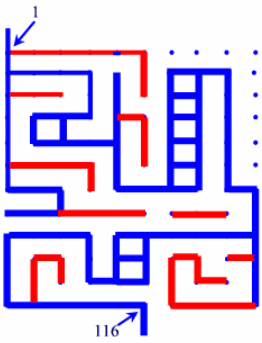
So, this way the maze can be solved using depth first search also. And, the answer in this case will be same as shown in breadth first search. And, the only practical difference between both the algorithms is that in breadth first search algorithm we get the shortest path but the time taken in computing this path might be a bit larger than what depth first search takes.

4.4 Trimming Algorithm

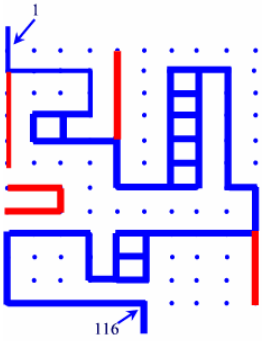
In this algorithm, we have to make use of the corresponding graph of the given maze. The way of obtaining the graph is exactly same as in breadth first search or depth first search algorithms. Here, we trim all the vertices that have degree 1 (except starting and ending vertices). For example, consider the following graph:



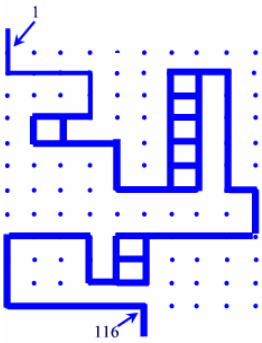
Here, consider the vertex # 60. Its degree is 1. So, we will trim it. And, as soon as we trim this vertex the degree of vertex above it becomes 1 so we will trim it also. This way quite a bit of graph will be trimmed.



Now, that line which we discussed eariler is trimmed. And, here, all the lines that are painted red will also be trimmed as its end vertices will have degree 1.



Here, we have trimmed all the lines that were painted red in the previous illustration. And now, we have painted all those edges red which now has degree 1.



After trimming the red edges of the previous diagram, we are now left with no edges with vertices of degree 1 (except starting and end points). So, we are left with the solution of the maze and it is clearly visible that more than one solution exists in the given maze.

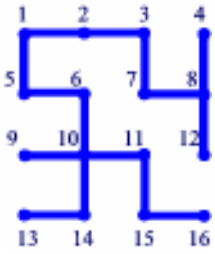
4.5 *Compass Algorithm*

This is probably most complex algorithm in this whole paper. And, this is also the most useful one. All the algorithms that we have discussed till now had

one condition, that the entire maze configuration is known *a priori*, and that we have random access to the whole maze configuration. But, this algorithm has no such limitation and it will work in any circumstances.

There are some algorithms that are designed in such a way that they will work if the maze configuration is not known initially. But, there are some problems that most of them face. The problems which we discussed in the Section – 3 (Existing Algorithms), such as islands problem and loop problems constantly plague most of the existing algorithms. But, this compass algorithm has been designed in such a way that it can counter any of these problems. The algorithm is called Compass Algorithm because the machine has to constantly deal with directions and it has to virtually remember which direction it has traversed and which direction is pending and also that its parent cell is in which direction. This algorithm is based on recursion and the machine will change direction recursively, so that it can conveniently come back to the parent cell after full traversal of the respective children cells. This algorithm is inspired by the preorder, inorder and postorder traversal of a binary tree.

To use this algorithm, we once again have to convert the given maze into its respective graph, as we did in Breadth First Search Algorithm and Depth First Search Algorithm. But, here the work is not over after you achieve the graph. Instead, now you have to make a matrix of (nX4) where n is the number of vertices of the graph. Suppose the graph obtained by the given maze is:



Matrix:

x	E	W	N	S
1	1	0	0	1
2	1	1	0	0
3	0	1	0	1
4	0	0	0	1
5	1	0	1	0
6	0	1	0	1
7	1	0	1	0
8	0	1	1	1
9	1	0	0	0
10	1	1	1	1
11	0	1	0	1
12	0	0	1	0
13	1	0	0	0
14	0	1	1	0
15	1	0	1	0
16	0	1	0	0

Now, how did we get this matrix? The answer is write all the vertices in the y-axis, now consider vertex – 1, it is connected only to its eastern and southern vertices, so we have 1 under E & S in front of 1 in the matrix. This way we can easily make the whole matrix. Although, this matrix won't be required once you implement this algorithm in physical conditions, here we require it only for simulation purposes.

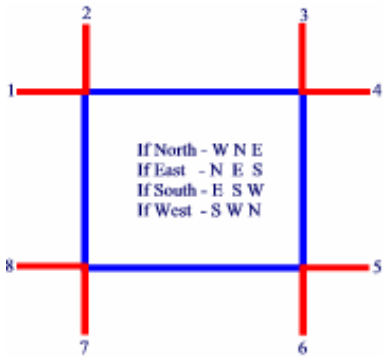
Under this algorithm, the machine knows, what is the smallest unit of distance within the given maze i.e. the distance between the vertices 1 & 2

(under the given example)? Now, the complete explanation of compass algorithm is as follows:

The current cell number of machine is saved in x, number of vertices in horizontal direction is m (we don't require it in physical implementation, but is necessary for mathematical simulation of the algorithm). And, the target vertex is saved in y.

Now, the algorithm will start from cell 1 and will first try to move in south direction. It will be checked whether there is an opening in southern direction or not. It there is no opening out there, then the computer will try western, then northern & at the end eastern direction. And, whenever the machine will get an opening, the current cell no (x) will be suitably updated. For example, take a look at the above graph. When the machine is at vertex – 1, it will check in southern direction and will move to 5th vertex (and hence, the value of x will be updated to 5, i.e. x+m). x is always updated according to this rule: x+1 for east, x-1 for west, x-m for north & x+m for south. For example, if you move in east direction from cell 5 then you will reach 6(x+1), and so on.

Now, whenever a machine moves into a particular direction, it looks for new directions i.e. if we have entered a new cell from southern direction, then we will look into west, north and east directions (strictly in that order) i.e. whichever way you have moved you always have to first look towards your left, then front and then right. This kind of direction rule never lets the robot to endlessly move into a loop. Here is an illustration on how this happens:



Here, you can see that there is a blue square in the middle of the figure and red extensions of the edges of this square. The blue portion signifies the loop (in which our machine might just fall) and the red lines signify the way into and out of the loop. And, in the middle of the square you can see that we have some kind of code written. One of them says, If North – W N E. By this we mean that if our robot is facing northern direction then it will first check W (West) then North and at last East.

So, now consider that the machine is coming from 1, as the robot is facing east, so firstly it will check north and then east. At this point, if the robot has to fall into the loop then this would mean that path 2 does not exist. Okay,

let's assume this thing and move further. Now, on reaching the top – right corner of Blue Square, the robot is facing east. Here it will again check north, then east and at last south. At this point if the robot still has to end up in the loop then again this would mean that paths 3 & 4 can't exist. So, let's assume this also. Now we are at the bottom – right corner of the square. Here the robot is facing south. Now, it will first check east, then south and at the end west. So, paths 5 & 6 also can't exist. Now, at bottom – left corner of the square, the machine will first check south, then west and at last north. So, the paths 7 & 8 can't exist. So, if the robot has to fall into the loop then none of the red edges can exist and if we remove all these red edges, then the blue square itself will become inaccessible. So, here we infer that if the machine is running on Compass Algorithm, then it can never fall into a loop.

And, at the end, as soon as value of x (current cell number of the machine) is equal to y (target cell number), the algorithm is completed.

Conclusion

Now that we have gone through all these algorithms, one thing is clear that, some algorithms do existed, but they definitely had some problems, like unable to counter islands and loops. But, all the four algorithms that we have devised (although Breadth First Search and Depth First Search were probably present even earlier, but only thing that everyone has to offer is that you can't use them directly, you have to modify them. Here, we have provided those modifications) can easily counter all these problems. Our Paintbrush Algorithm is perhaps, one of the fastest ways of solving a maze. Breadth First Search Algorithm can provide you all the possible ways that can exist in a maze and also give you the shortest of them all. Also, for the MicroMouse competitors, we would like to say that with minor adjustments you can also find the fastest way. We are differentiating between fastest and shortest way because in some cases the shortest way might be having a large number of turns whereas its nearest competitor, it terms of distance, might be having far lesser number of turns. And, as far as the Depth First Search Algorithm goes, it is practically a cousin of Breadth First Algorithm only and it can employed when only one path exists in the maze and you have serious time constraints. And, at last our Compass Algorithm. Most of the existing algorithms considered that they know the complete maze configuration by default and those which didn't were constantly plagued by a lot of problems (as discussed earlier). But, our Compass Algorithm efficiently takes care of all these problems. So, we think that we are successful in devising some simple and efficient algorithms for a problem that can be definitely used in robots for achieving certain degree of analyticity.

References

1. Building A Maze Solving Robot – My Experiences. www.gorobotics.net/articles/article.php?name=mazebot
2. C. Scott Ananian & Greg Humphreys, Theseus : A Maze Solving Robot (1997)
3. Kevin Lam & Gabriel Wainer, Modelling of Maze Solving Systems using Cell-DEVS
4. Maze Solving Algorithm. www.lboro.ac.uk/departments/el/robotics/maze_solver.html
5. Swapan Kumar Sarkar, A Textbook on Discrete Mathematics. S. Chand Publications.
6. The MicroMouse Competition. World Wide Web. http://www.nis.lanl.gov/projects/robot/html/rules/micromouse.html
7. Think Labyrinth: Maze Algorithms. www.astrolog.org/labyrnth/algrithm.htm