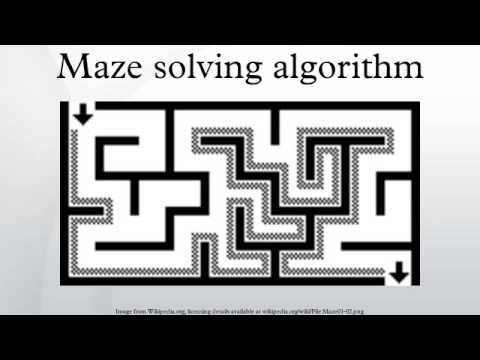
**MAZE SOLVING**

A MAZE is a puzzle consisting of a complicated network of paths or passages, the aim of which is to find one’s way. There are a number of MAZE SOLVING algorithms, that is , automated method for solving of mazes.

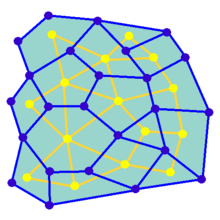
Maze Solving Algorithms are designed to be used inside the maze by a traveller with no prior knowledge of the maze where as the dead end filling and short path algorithms are designed to be used by a person or computer program that can see the whole maze at ones.



* Graph Maze Solving
* Breadth First Search
* Depth First Search
* Random Mouse Algorithm
* Wall Follower
* Pledge Algorithm
* Tremaux’s Algorithm
* Dead-end Filling
* Recursive Algorithm
* Maze-routing Algorithm
* Shortest Path Algorithm

1. **Graph Maze Solving**

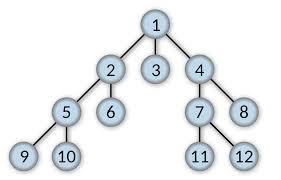
* Maze containing no loops are known as “simply connected “or “perfect “mazes and are equivalent to a tree in graph theory. Thus the maze solving algorithm are closely related to graph theory.



* A maze can be generated by starting with a predetermined arrangement of cells with wall sites between them.
* This predetermined arrangement can be considered as a [connected graph](https://en.wikipedia.org/wiki/Connected_graph) with the edges representing possible wall sites and the nodes representing cells.
* The purpose of the maze generation algorithm can then be considered to be making a subgraph in which it is challenging to find a route between two particular nodes.

1. **Breadth First Search**

* Breadth-first search (**BFS**) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures.



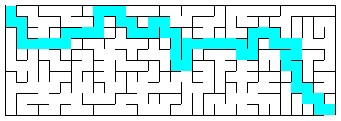
* It starts at the [tree root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.
* It uses the opposite strategy as [depth-first search](https://en.wikipedia.org/wiki/Depth-first_search), which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.
* This non-recursive implementation is similar to the non-recursive implementation of [depth-first search](https://en.wikipedia.org/wiki/Depth-first_search), but differs from it in two way :-

it uses a [queue](https://en.wikipedia.org/wiki/Queue_(abstract_data_type)) (First In First Out) instead of a [stack](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)) and

it checks whether a vertex has been discovered before enqueueing the vertex rather than delaying this check until the vertex is dequeued from the queue.

1. **Depth First Search**

* Depth-first Search (DFS) is a graph / tree traversal algorithm that follows a path as far as it can until it either reaches the goal or has nowhere else to go.
* It’s almost running as far as it can in one direction until it hits a wall.
* There are many approaches and styles to implement a depth-first search algorithm and a lot of those implementation choices will depend on the problem we are trying to solve.
* Maze generated with maze first search have low branching factor and contain many long corridors because the algorithm explores as far as possible along each branch before backtracking.



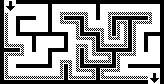
* There are certain assumptions in depth-first search :-

1. Each maze has a starting point and a ending point
2. Mazes are composed of walls
3. We only care about finding the first available path through maze and not the shortest one
4. **Random Mouse Algorithm**

* This is a trivial method that can be implemented by a very unintelligent [robot](https://en.wikipedia.org/wiki/Robot) or perhaps a mouse as it works on hit and trial method.
* It is simply to proceed following the current passage until a junction is reached, and then to make a random decision about the next direction to follow. This process continues until it reaches it’s destination.
* Although such a method would always [eventually find the right solution](https://en.wikipedia.org/wiki/Las_Vegas_algorithm), this algorithm can be extremely slow.

1. **Wall Follower**

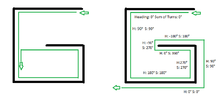
* The best-known rule for traversing mazes is the *wall follower*, also known as either the *left-hand rule* or the *right-hand rule*.
* Another perspective into why wall following works is topological. If the walls are connected, then they may be deformed into a loop or circle.



* If the maze is not simply-connected (if the start or endpoints are in the center of the structure surrounded by passage loops, or the pathways cross over ), this method will not reach the goal.

1. **Pledge Algorithm**

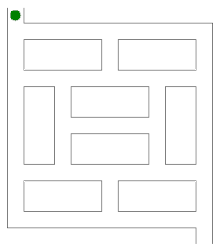
* The Pledge algorithm, designed to circumvent obstacles, requires an arbitrarily chosen direction to go toward, which will be preferential.
* When an obstacle is met, one hand (say the right hand) is kept along the obstacle while the angles turned are counted (clockwise turn is positive, counter-clockwise turn is negative).
* When the solver is facing the original preferential direction again, and the angular sum of the turns made is 0, the solver leaves the obstacle and continues moving in its original direction.



* This algorithm allows a person with a compass to find their way from any point inside to an outer exit of any finite two-dimensional maze, regardless of the initial position of the solver.
* However, this algorithm will not work in doing the reverse, namely finding the way from an entrance on the outside of a maze to some end goal within it.

1. **Tremaux’s Algorithm**

* Trémaux's algorithm is an efficient method to find the way out of a maze that requires drawing lines on the floor to mark a path, and is guaranteed to work for all mazes that have well-defined passages ,  but it is not guaranteed to find the shortest route.



* The "turn around and return" rule effectively transforms any maze with loops into a simply connected one; whenever we find a path that would close a loop, we treat it as a dead end and return. Without this rule, it is possible to cut off one's access to still-unexplored parts of a maze if, instead of turning back, we arbitrarily follow another path.
* When you finally reach the solution, paths marked exactly once will indicate a way back to the start. If there is no exit, this method will take you back to the start where all paths are marked twice.

1. **Dead-end Filling**

* Dead-end filling is an algorithm for solving mazes that fills all dead ends, leaving only the correct ways unfilled. It can be used for solving mazes on paper or with a computer program.
* But it is not useful to a person inside an unknown maze since this method looks at the entire maze at once.
* The method is to

1) find all of the dead-ends in the maze, and then

2) "fill in" the path from each dead-end until the first junction is met. Note that some passages won't become parts of dead end passages until other dead ends are removed first.

* Dead-end filling cannot accidentally "cut off" the start from the finish since each step of the process preserves the topology of the maze.

1. **Recursive Algorithm**

* If given an omniscient view of the maze, a simple recursive algorithm can tell one how to get to the end.
* The algorithm will be given a starting X and Y value.
* If the X and Y values are not on a wall, the method will call itself with all adjacent X and Y values, making sure that it did not already use those X and Y values before.
* If the X and Y values are those of the end location, it will save all the previous instances of the method as the correct path.

1. **Maze-routing Algorithm**

* The maze-routing algorithm is a low overhead method to find the way between any two locations of the maze.
* The algorithm is initially proposed for [chip multiprocessors](https://en.wikipedia.org/wiki/Chip_multiprocessor) (CMPs) domain and guarantees to work for any grid-based maze.
* In addition to finding paths between two location of the grid (maze), the algorithm can detect when there is no path between the source and destination.
* The algorithm is to be used by an inside traveler with no prior knowledge of the maze with fixed memory complexity regardless of the maze size; requiring 4 variables in total for finding the path and detecting the unreachable locations.
* Nevertheless, the algorithm is not to find the shortest path.

1. **Shortest Path Algorithm**

* When a maze has multiple solutions, the solver may want to find the shortest path from start to finish.
* One such algorithm finds the shortest path by implementing a [breadth-first search](https://en.wikipedia.org/wiki/Breadth-first_search).



* The breadth-first search algorithm uses a [queue](https://en.wikipedia.org/wiki/Queue_(data_structure)) to visit cells in increasing distance order from the start until the finish is reached.
* Each visited cell needs to keep track of its distance from the start or which adjacent cell nearer to the start caused it to be added to the queue.
* When the finish location is found, follow the path of cells backwards to the start, which is the shortest path.
* The breadth-first search in its simplest form has its limitations, like finding the shortest path in weighted graphs.

<https://en.wikipedia.org/wiki/Maze_solving_algorithm>

<https://medium.com/swlh/solving-mazes-with-depth-first-search-e315771317ae>

<https://en.wikipedia.org/wiki/Maze_generation_algorithm>

<https://en.wikipedia.org/wiki/Breadth-first_search>