MAZE PUZZLE GAME BASED ON

DFS

END TERM REPORT

*by*

**NAME OF THE CANDIDATES**

**Section: K18RD**

**Roll no: 05,06,15,16**



Department of Intelligent Systems

Lovely professional University, Jalandhar

04-2020

Student Declaration

This is to declare that this report has been written by us. No part of the report is copied from other sources. All information included from other sources have been duly acknowledged. We ever that if any part of the report is found to be copied, we are shall take full responsibility for it.

Name: Ayush Kumar

Reg no: 11807506

Roll no: 05

Name: Sahil Verma

Reg no: 11808204

Roll no: 06

Name: Jalluri V Koushik

Reg no: 11810544

Roll no: 15

Name: Ammad UI Ashraf

Reg no:11815482

Roll no: 16

Place: Lovely professional university.

Date: 08-april-2020.

TABLE OF CONTENTS

1. Background.
2. Depth First Search.
3. Objectives.
4. Description.

1. Background:-

Maze is a path or number of routes generally from an entrance to a goal. The term labyrinth is used as synonymous with maze. In this game pathways and walls in a maze are typically fixed. Maze generation is the act of designing the layout of passages and walls within a maze. There are many different approaches to generate mazes, with various maze generation algorithms for building them either by hand or automatically by computer. Maze solving is the act of finding a route through the maze from the start to finish. Some maze solving methods are designed to be used inside the maze by a traveler with no prior knowledge of the maze, whereas others are designed to be used by a person or computer program that can see the whole maze at once.

The mazes containing no loops are known as “Standard” or “perfect” mazes and are equivalent to a tree in graph theory. Thus, many maze solving algorithms are closely related to graph theory. Intuitively, if one pulled and stretched out the paths in the maze in the proper way, the result could be made to resemble the tree. There are many different types of mazes some of the examples are as follow Dexterity puzzles which involve navigating a ball through a maze or labyrinth. A maze in which the player must complete or clear the maze pathway by positioning blocks. Blocks may slide into place or be added. These are like standard mazes except they use rules other than “don not cross the line” to restrict the motion. A standard maze that forms a picture when solved. The maze that features one-way doors. The door can lead to the correct path or create trap that diverts you from the correct the correct path and lead you to the starting point.

Properties

The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time {\displaystyle O(|V|+|E|)}O(|V| + |E|),[4] linear in the size of the graph. In these applications it also uses space {\displaystyle O(|V|)}O(|V|) in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the time and space bounds are the same as for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

For applications of DFS in relation to specific domains, such as searching for solutions in artificial intelligence or web-crawling, the graph to be traversed is often either too large to visit in its entirety or infinite (DFS may suffer from non-termination). In such cases, search is only performed to a limited depth; due to limited resources, such as memory or disk space, one typically does not use data structures to keep track of the set of all previously visited vertices. When search is performed to a limited depth, the time is still linear in terms of the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, and as a result, is much smaller than the space needed for searching to the same depth using breadth-first search. For such applications, DFS also lends itself much better to heuristic methods for choosing a likely-looking branch. When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits. In the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

DFS may also be used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree.

Example

For the following graph:

Graph.traversal.example.svg

a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory. Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening is one technique to avoid this infinite loop and would reach all nodes.

Output of a depth-first search

The four types of edges defined by a spanning tree

A convenient description of a depth-first search of a graph is in terms of a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: forward edges, which point from a node of the tree to one of its descendants, back edges, which point from a node to one of its ancestors, and cross edges, which do neither. Sometimes tree edges, edges which belong to the spanning tree itself, are classified separately from forward edges. If the original graph is undirected then all of its edges are tree edges or back edges.

DFS ordering

An enumeration of the vertices of a graph is said to be a DFS ordering if it is the possible output of the application of DFS to this graph.

Let {\displaystyle G=(V,E)}G=(V,E) be a graph with {\displaystyle n}n vertices. For {\displaystyle \sigma =(v\_{1},\dots ,v\_{m})}{\displaystyle \sigma =(v\_{1},\dots ,v\_{m})} be a list of distinct elements of {\displaystyle V}V, for {\displaystyle v\in V\setminus \{v\_{1},\dots ,v\_{m}\}}{\displaystyle v\in V\setminus \{v\_{1},\dots ,v\_{m}\}}, let {\displaystyle \nu \_{\sigma }(v)}{\displaystyle \nu \_{\sigma }(v)} be the greatest {\displaystyle i}i such that {\displaystyle v\_{i}}v\_{i} is a neighbor of {\displaystyle v}v, if such a {\displaystyle i}i exists, and be {\displaystyle 0}{\displaystyle 0} otherwise.

Let {\displaystyle \sigma =(v\_{1},\dots ,v\_{n})}{\displaystyle \sigma =(v\_{1},\dots ,v\_{n})} be an enumeration of the vertices of {\displaystyle V}V. The enumeration {\displaystyle \sigma }\sigma is said to be a DFS ordering (with source {\displaystyle v\_{1}}v\_{1}) if, for all {\displaystyle 1<i\leq n}{\displaystyle 1<i\leq n}, {\displaystyle v\_{i}}v\_{i} is the vertex {\displaystyle w\in V\setminus \{v\_{1},\dots ,v\_{i-1}\}}{\displaystyle w\in V\setminus \{v\_{1},\dots ,v\_{i-1}\}} such that {\displaystyle \nu \_{(v\_{1},\dots ,v\_{i-1})}(w)}{\displaystyle \nu \_{(v\_{1},\dots ,v\_{i-1})}(w)} is maximal. Recall that {\displaystyle N(v)}N(v) is the set of neighbors of {\displaystyle v}v. Equivalently, {\displaystyle \sigma }\sigma is a DFS ordering if, for all {\displaystyle 1\leq i<j<k\leq n}{\displaystyle 1\leq i<j<k\leq n} with {\displaystyle v\_{i}\in N(v\_{k})\setminus N(v\_{j})}{\displaystyle v\_{i}\in N(v\_{k})\setminus N(v\_{j})}, there exists a neighbor {\displaystyle v\_{m}}v\_{m} of {\displaystyle v\_{j}}v\_{j} such that {\displaystyle i<m<j}{\displaystyle i<m<j}.

Vertex orderings

It is also possible to use depth-first search to linearly order the vertices of a graph or tree. There are four possible ways of doing this:

A preordering is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation.

A postordering is a list of the vertices in the order that they were last visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.

A reverse preordering is the reverse of a preordering, i.e. a list of the vertices in the opposite order of their first visit. Reverse preordering is not the same as postordering.

A reverse postordering is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. Reverse postordering is not the same as preordering.

For binary trees there is additionally in-ordering and reverse in-ordering.

For example, when searching the directed graph below beginning at node A, the sequence of traversals is either A B D B A C A or A C D C A B A (choosing to first visit B or C from A is up to the algorithm). Note that repeat visits in the form of backtracking to a node, to check if it has still unvisited neighbors, are included here (even if it is found to have none). Thus the possible preorderings are A B D C and A C D B, while the possible postorderings are D B C A and D C B A, and the possible reverse postorderings are A C B D and A B C D.

If-then-else-control-flow-graph.svg

Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flows. The graph above might represent the flow of control in the code fragment below, and it is natural to consider this code in the order A B C D or A C B D but not natural to use the order A B D C or A C D B.

if (A) then {

B

} else {

C

}

D

Pseudocode

Input: A graph G and a vertex v of G

Output: All vertices reachable from v labeled as discovered

A recursive implementation of DFS:[5]

procedure DFS(G, v) is

label v as discovered

for all directed edges from v to w that are in G.adjacentEdges(v) do

if vertex w is not labeled as discovered then

recursively call DFS(G, w)

The order in which the vertices are discovered by this algorithm is called the lexicographic order.

A non-recursive implementation of DFS with worst-case space complexity {\displaystyle O(|E|)}O(|E|):[6]

procedure DFS-iterative(G, v) is

let S be a stack

S.push(v)

while S is not empty do

v = S.pop()

if v is not labeled as discovered then

label v as discovered

for all edges from v to w in G.adjacentEdges(v) do

S.push(w)

These two variations of DFS visit the neighbors of each vertex in the opposite order from each other: the first neighbor of v visited by the recursive variation is the first one in the list of adjacent edges, while in the iterative variation the first visited neighbor is the last one in the list of adjacent edges. The recursive implementation will visit the nodes from the example graph in the following order: A, B, D, F, E, C, G. The non-recursive implementation will visit the nodes as: A, E, F, B, D, C, G.

The non-recursive implementation is similar to breadth-first search but differs from it in two ways:

it uses a stack instead of a queue, and

it delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before adding the vertex.

Applications

File:MAZE 30x20 DFS.ogv

Randomized algorithm similar to depth-first search used in generating a maze.

Algorithms that use depth-first search as a building block include:

Finding connected components.

Topological sorting.

Finding 2-(edge or vertex)-connected components.

Finding 3-(edge or vertex)-connected components.

Finding the bridges of a graph.

Generating words in order to plot the limit set of a group.

Finding strongly connected components.

Planarity testing.[7][8]

Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Maze generation may use a randomized depth-first search.

Finding biconnectivity in graphs.

Complexity

The computational complexity of DFS was investigated by John Reif. More precisely, given a graph {\displaystyle G}G, let {\displaystyle O=(v\_{1},\dots ,v\_{n})}{\displaystyle O=(v\_{1},\dots ,v\_{n})} be the ordering computed by the standard recursive DFS algorithm. This ordering is called the lexicographic depth-first search ordering. John Reif considered the complexity of computing the lexicographic depth-first search ordering, given a graph and a source. A decision version of the problem (testing whether some vertex u occurs before some vertex v in this order) is P-complete,[9] meaning that it is "a nightmare for parallel processing".[10]:189

A depth-first search ordering (not necessarily the lexicographic one), can be computed by a randomized parallel algorithm in the complexity class RNC.[11] As of 1997, it remained unknown whether a depth-first traversal could be constructed by a deterministic parallel algorithm, in the complexity class NC.

# Depth-first search

**Depth-first search** (**DFS**) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

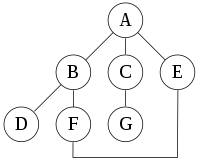
The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time{\displaystyle O(|V|+|E|)}, linear in the size of the graph. In these applications it also uses space{\displaystyle O(|V|)} in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the time and space bounds are the same as for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

For applications of DFS in relation to specific domains, such as searching for solutions in artificial intelligence or web-crawling, the graph to be traversed is often either too large to visit in its entirety or infinite (DFS may suffer from non-termination). In such cases, search is only performed to a limited depth; due to limited resources, such as memory or disk space, one typically does not use data structures to keep track of the set of all previously visited vertices. When search is performed to a limited depth, the time is still linear in terms of the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, and as a result, is much smaller than the space needed for searching to the same depth using breadth-first search. For such applications, DFS also lends itself much better to heuristic methods for choosing a likely-looking branch. When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits. In the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

DFS may also be used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree.

## **Example**

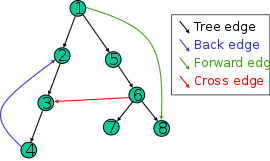
For the following graph:

[](https://en.wikipedia.org/wiki/File:Graph.traversal.example.svg)

a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory. Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening is one technique to avoid this infinite loop and would reach all nodes.

## **Output of a depth-first search**

[](https://en.wikipedia.org/wiki/File:Tree_edges.svg)

The four types of edges defined by a spanning tree

A convenient description of a depth-first search of a graph is in terms of a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: **forward edges**, which point from a node of the tree to one of its descendants, **back edges**, which point from a node to one of its ancestors, and **cross edges**, which do neither. Sometimes **tree edges**, edges which belong to the spanning tree itself, are classified separately from forward edges. If the original graph is undirected then all of its edges are tree edges or back edges.

### DFS ordering

An enumeration of the vertices of a graph is said to be a DFS ordering if it is the possible output of the application of DFS to this graph.

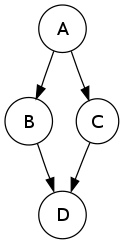
### Vertex orderings

It is also possible to use depth-first search to linearly order the vertices of a graph or tree. There are four possible ways of doing this:

* A **preordering** is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation.
* A **postordering** is a list of the vertices in the order that they were *last* visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.
* A **reverse preordering** is the reverse of a preordering, i.e. a list of the vertices in the opposite order of their first visit. Reverse preordering is not the same as postordering.
* A **reverse postordering** is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. Reverse postordering is not the same as preordering.

For binary trees there is additionally **in-ordering** and **reverse in-ordering**.

For example, when searching the directed graph below beginning at node A, the sequence of traversals is either A B D B A C A or A C D C A B A (choosing to first visit B or C from A is up to the algorithm). Note that repeat visits in the form of backtracking to a node, to check if it has still unvisited neighbors, are included here (even if it is found to have none). Thus the possible preorderings are A B D C and A C D B, while the possible postorderings are D B C A and D C B A, and the possible reverse postorderings are A C B D and A B C D.

[](https://en.wikipedia.org/wiki/File:If-then-else-control-flow-graph.svg)

Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flows. The graph above might represent the flow of control in the code fragment below, and it is natural to consider this code in the order A B C D or A C B D but not natural to use the order A B D C or A C D B.

if (**A**) then {

**B**

} else {

**C**

}

**D**

## **Pseudocode**

**Input**: A graph *G* and a vertex *v* of G

**Output**: All vertices reachable from *v* labeled as discovered

A recursive implementation of DFS:

**procedure** DFS(*G*, *v*) **is**

label *v* as discovered

**for all** directed edges from *v* to *w that are* **in** *G*.adjacentEdges(*v*) **do**

**if** vertex *w* is not labeled as discovered **then**

recursively call DFS(*G*, *w*)

The order in which the vertices are discovered by this algorithm is called the lexicographic order.

A non-recursive implementation of DFS with worst-case space complexity O(|E|)  
{\displaystyle O(|E|)}  
{\displaystyle O(|E|)}{\displaystyle O(|E|)}:

**procedure** DFS-iterative(*G*, *v*) **is**

let *S* be a stack

*S*.push(*v*)

**while** *S* is not empty **do**

*v* = *S*.pop()

**if** *v* is not labeled as discovered **then**

label *v* as discovered

**for all** edges from *v* to *w* **in** *G*.adjacentEdges(*v*) **do**

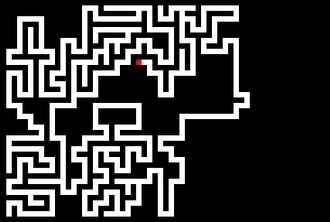
*S*.push(*w*)

These two variations of DFS visit the neighbors of each vertex in the opposite order from each other: the first neighbor of *v* visited by the recursive variation is the first one in the list of adjacent edges, while in the iterative variation the first visited neighbor is the last one in the list of adjacent edges. The recursive implementation will visit the nodes from the example graph in the following order: A, B, D, F, E, C, G. The non-recursive implementation will visit the nodes as: A, E, F, B, D, C, G.

The non-recursive implementation is similar to breadth-first search but differs from it in two ways:

1. it uses a stack instead of a queue, and
2. it delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before adding the vertex.

## **Applications**



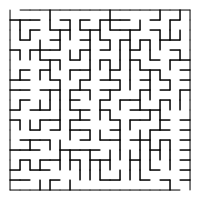
Randomized algorithm similar to depth-first search used in generating a maze.

Algorithms that use depth-first search as a building block include:

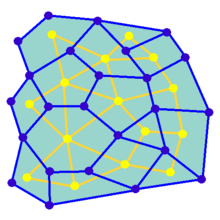
* Finding connected components.
* Topological sorting.
* Finding 2-(edge or vertex)-connected components.
* Finding 3-(edge or vertex)-connected components.
* Finding the bridges of a graph.
* Generating words in order to plot the limit set of a group.
* Finding strongly connected components.
* Planarity testing.[7][8]
* Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
* Maze generation may use a randomized depth-first search.
* Finding biconnectivity in graphs.

# Maze generation algorithm

**Maze generation algorithms** are automated methods for the creation of mazes.

[](https://en.wikipedia.org/wiki/File:Prim_Maze.svg)

## **Graph theory based methods**

[](https://en.wikipedia.org/wiki/File:Graph_based_maze_animation.gif)

Animation of Graph theory based method

A maze can be generated by starting with a predetermined arrangement of cells (most commonly a rectangular grid but other arrangements are possible) with wall sites between them. This predetermined arrangement can be considered as a 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.

If the subgraph is not connected, then there are regions of the graph that are wasted because they do not contribute to the search space. If the graph contains loops, then there may be multiple paths between the chosen nodes. Because of this, maze generation is often approached as generating a random spanning tree. Loops, which can confound naive maze solvers, may be introduced by adding random edges to the result during the course of the algorithm.

The animation shows the maze generation steps for a graph that is not on a rectangular grid. First, the computer creates a random planar graph G shown in blue, and its dual F shown in yellow. Second, computer traverses F using a chosen algorithm, such as a depth-first search, coloring the path red. During the traversal, whenever a red edge crosses over a blue edge, the blue edge is removed. Finally, when all vertices of F have been visited, F is erased and two edges from G, one for the entrance and one for the exit, are removed.

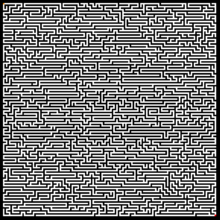
### Depth-first search



Animation of generator's thinking process using Depth-First Search

This algorithm is a randomized version of the depth-first search algorithm. Frequently implemented with a stack, this approach is one of the simplest ways to generate a maze using a computer. Consider the space for a maze being a large grid of cells (like a large chess board), each cell starting with four walls. Starting from a random cell, the computer then selects a random neighbouring cell that has not yet been visited. The computer removes the wall between the two cells and marks the new cell as visited, and adds it to the stack to facilitate backtracking. The computer continues this process, with a cell that has no unvisited neighbours being considered a dead-end. When at a dead-end it backtracks through the path until it reaches a cell with an unvisited neighbour, continuing the path generation by visiting this new, unvisited cell (creating a new junction). This process continues until every cell has been visited, causing the computer to backtrack all the way back to the beginning cell. We can be sure every cell is visited.

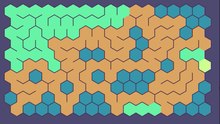
As given above this algorithm involves deep recursion which may cause stack overflow issues on some computer architectures. The algorithm can be rearranged into a loop by storing backtracking information in the maze itself. This also provides a quick way to display a solution, by starting at any given point and backtracking to the beginning.

[](https://en.wikipedia.org/wiki/File:Horizontally_Influenced_Depth-First_Search_Generated_Maze.png)

Horizontal Passage Bias

Mazes generated with a depth-first search have a low branching factor and contain many long corridors, because the algorithm explores as far as possible along each branch before backtracking.

### Recursive backtracker



Recursive backtracker on a hexagonal grid

The depth-first search algorithm of maze generation is frequently implemented using backtracking. This can be described with a following recursive routine:

1. Given a current cell as a parameter,
2. Mark the current cell as visited
3. While the current cell has any unvisited neighbour cells
   1. Choose one of the unvisited neighbours
   2. Remove the wall between the current cell and the chosen cell
   3. Invoke the routine recursively for a chosen cell

which is invoked once for any initial cell in the area.

A disadvantage of this approach is a large depth of recursion – in the worst case, the routine may need to recur on every cell of the area being processed, which may exceed the maximum recursion stack depth in many environments. As a solution, the same bactracking method can be implemented with an explicit stack, which is usually allowed to grow much bigger with no harm.

1. Choose the initial cell, mark it as visited and push it to the stack
2. While the stack is not empty
   1. Pop a cell from the stack and make it a current cell
   2. If the current cell has any neighbours which have not been visited
      1. Push the current cell to the stack
      2. Choose one of the unvisited neighbours
      3. Remove the wall between the current cell and the chosen cell
      4. Mark the chosen cell as visited and push it to the stack

### Randomized Kruskal's algorithm



An animation of generating a 30 by 20 maze using Kruskal's algorithm.

This algorithm is a randomized version of Kruskal's algorithm.

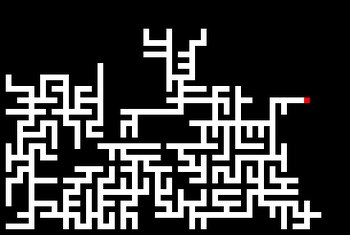
1. Create a list of all walls, and create a set for each cell, each containing just that one cell.
2. For each wall, in some random order:
   1. If the cells divided by this wall belong to distinct sets:
      1. Remove the current wall.
      2. Join the sets of the formerly divided cells.

There are several data structures that can be used to model the sets of cells. An efficient implementation using a disjoint-set data structure can perform each union and find operation on two sets in nearly constant amortized time (specifically, {\displaystyle O(\alpha (V))}O(α (V)) time; α (x)<5 for any plausible value of x x , so the running time of this algorithm is essentially proportional to the number of walls available to the maze.

It matters little whether the list of walls is initially randomized or if a wall is randomly chosen from a nonrandom list, either way is just as easy to code.

Because the effect of this algorithm is to produce a minimal spanning tree from a graph with equally weighted edges, it tends to produce regular patterns which are fairly easy to solve.

### Randomized Prim's algorithm



An animation of generating a 30 by 20 maze using Prim's algorithm.

This algorithm is a randomized version of Prim's algorithm.

1. Start with a grid full of walls.
2. Pick a cell, mark it as part of the maze. Add the walls of the cell to the wall list.
3. While there are walls in the list:
   1. Pick a random wall from the list. If only one of the two cells that the wall divides is visited, then:
      1. Make the wall a passage and mark the unvisited cell as part of the maze.
      2. Add the neighboring walls of the cell to the wall list.
   2. Remove the wall from the list.

It will usually be relatively easy to find the way to the starting cell, but hard to find the way anywhere else.

Note that simply running classical Prim's on a graph with random edge weights would create mazes stylistically identical to Kruskal's, because they are both minimal spanning tree algorithms. Instead, this algorithm introduces stylistic variation because the edges closer to the starting point have a lower effective weight.

#### Modified version

Although the classical Prim's algorithm keeps a list of edges, for maze generation we could instead maintain a list of adjacent cells. If the randomly chosen cell has multiple edges that connect it to the existing maze, select one of these edges at random. This will tend to branch slightly more than the edge-based version above.

### Wilson's algorithm

All the above algorithms have biases of various sorts: depth-first search is biased toward long corridors, while Kruskal's/Prim's algorithms are biased toward many short dead ends. Wilson's algorithm, on the other hand, generates an *unbiased* sample from the uniform distribution over all mazes, using loop-erased random walks.

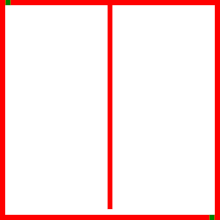
We begin the algorithm by initializing the maze with one cell chosen arbitrarily. Then we start at a new cell chosen arbitrarily and perform a random walk until we reach a cell already in the maze—however, if at any point the random walk reaches its own path, forming a loop, we erase the loop from the path before proceeding. When the path reaches the maze, we add it to the maze. Then we perform another loop-erased random walk from another arbitrary starting cell, repeating until all cells have been filled.

This procedure remains unbiased no matter which method we use to arbitrarily choose starting cells. So we could always choose the first unfilled cell in (say) left-to-right, top-to-bottom order for simplicity.

## **Recursive division method**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Illustration of Recursive Division** | | | | |
| ***original chamber*** | ***division by two walls*** | ***holes in walls*** | ***continue subdividing...*** | ***completed*** |
| step 1 | step 2 | step 3 | step 4 | step 5 |

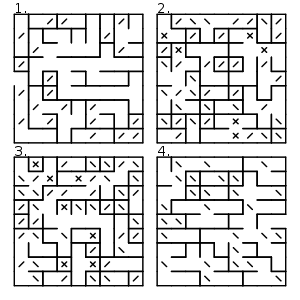
Mazes can be created with *recursive division*, an algorithm which works as follows: Begin with the maze's space with no walls. Call this a chamber. Divide the chamber with a randomly positioned wall (or multiple walls) where each wall contains a randomly positioned passage opening within it. Then recursively repeat the process on the sub chambers until all chambers are minimum sized. This method results in mazes with long straight walls crossing their space, making it easier to see which areas to avoid.

[](https://en.wikipedia.org/wiki/File:Recursive_maze.gif)

Recursive Maze generation

For example, in a rectangular maze, build at random points two walls that are perpendicular to each other. These two walls divide the large chamber into four smaller chambers separated by four walls. Choose three of the four walls at random, and open a one cell-wide hole at a random point in each of the three. Continue in this manner recursively, until every chamber has a width of one cell in either of the two directions.

## **Simple algorithms**

[](https://en.wikipedia.org/wiki/File:Prim_Maze_3D.svg)

3D version of Prim's algorithm. Vertical layers are labeled 1 through 4 from bottom to top. Stairs up are indicated with "/"; stairs down with "\", and stairs up-and-down with "x". Source code is included with the image description.

Other algorithms exist that require only enough memory to store one line of a 2D maze or one plane of a 3D maze. Eller's algorithm prevents loops by storing which cells in the current line are connected through cells in the previous lines, and never removes walls between any two cells already connected. The Sidewinder algorithm starts with an open passage along the entire the top row, and subsequent rows consist of shorter horizontal passages with one connection to the passage above. The Sidewinder algorithm is trivial to solve from the bottom up because it has no upward dead ends. Given a starting width, both algorithm create perfect mazes of unlimited height.

Most maze generation algorithms require maintaining relationships between cells within it, to ensure the end result will be solvable. Valid simply connected mazes can however be generated by focusing on each cell independently. A binary tree maze is a standard orthogonal maze where each cell always has a passage leading up or leading left, but never both. To create a binary tree maze, for each cell flip a coin to decide whether to add a passage leading up or left. Always pick the same direction for cells on the boundary, and the end result will be a valid simply connected maze that looks like a binary tree, with the upper left corner its root. As with Sidewinder, the binary tree maze has no dead ends in the directions of bias.

A related form of flipping a coin for each cell is to create an image using a random mix of forward slash and backslash characters. This doesn't generate a valid simply connected maze, but rather a selection of closed loops and unicursal passages. (The manual for the Commodore 64 presents a BASIC program using this algorithm, but using PETSCII diagonal line graphic characters instead for a smoother graphic appearance.)

## **Cellular automaton algorithms**

Certain types of cellular automata can be used to generate mazes. Two well-known such cellular automata, Maze and Mazectric, have rulestrings B3/S12345 and B3/S1234. In the former, this means that cells survive from one generation to the next if they have at least one and at most five neighbours. In the latter, this means that cells survive if they have one to four neighbours. If a cell has exactly three neighbours, it is born. It is similar to Conway's Game of Life in that patterns that do not have a living cell adjacent to 1, 4, or 5 other living cells in any generation will behave identically to it. However, for large patterns, it behaves very differently from Life.

For a random starting pattern, these maze-generating cellular automata will evolve into complex mazes with well-defined walls outlining corridors. Mazecetric, which has the rule B3/S1234 has a tendency to generate longer and straighter corridors compared with Maze, with the rule B3/S12345. Since these cellular automaton rules are deterministic, each maze generated is uniquely determined by its random starting pattern. This is a significant drawback since the mazes tend to be relatively predictable.

Like some of the graph-theory based methods described above, these cellular automata typically generate mazes from a single starting pattern; hence it will usually be relatively easy to find the way to the starting cell, but harder to find the way anywhere else.

4. Description Depth-First Search (DFS)

**Depth-first search (DFS)** is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored. Many problems in computer science can be thought of in terms of graphs. For example, analyzing networks, mapping routes, scheduling, and finding spanning trees are graph problems. To analyze these problems, graph-search algorithms like depth-first search are useful.

Depth-first searches are often used as subroutines in other more complex algorithms. For example, the matching algorithm, Hopcroft–Karp, uses a DFS as part of its algorithm to help to find a matching in a graph. DFS is also used in tree-traversal algorithms, also known as tree searches, which have applications in the traveling-salesman problem and the Ford-Fulkerson algorithm.

BONAFIDE CERTIFICATE

Certified that this project report “MAZE PUZZLE GAME BASED ON DFS” is the bonafide work of “ Names: Ayush Kumar, Sahil Verma, Jalluri V Koushik, Ammad UI Ashraf“ who carried out the project work under my supervision.

<<Signature of the Supervisor>>

<<Name of supervisor>>

<<Academic Designation>>

<<ID of Supervisor>>

<<Department of Supervisor>>