Maze Report

الاسم : عمر أحمد فؤاد حسن وصفى .

رقم الكشف : 46

From Wiki

**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)(or some arbitrary node of a graph, sometimes referred to as a 'search key'[[1]](https://en.wikipedia.org/wiki/Breadth-first_search#cite_note-1)) and explores the neighbor nodes first, before moving to the next level neighbors.

BFS was invented in the late 1950s by [E. F. Moore](https://en.wikipedia.org/wiki/Edward_F._Moore), who used it to find the shortest path out of a maze,[[2]](https://en.wikipedia.org/wiki/Breadth-first_search" \l "cite_note-skiena-2) and [discovered independently](https://en.wikipedia.org/wiki/Multiple_discovery) by C. Y. Lee as a **Input**: A graph *Graph* and a *starting vertex* *root* of *Graph*

**Output**: All vertices reachable from *root* labeled as explored.

A non-recursive implementation of breadth-first search:

1 Breadth-First-Search(Graph, root):

2

3 **for** each node n in Graph:

4 n.distance = INFINITY

5 n.parent = NIL

6

7 create empty queue Q

8

9 root.distance = 0

10 Q.enqueue(root)

11

12 **while** Q is not empty:

13

14 current = Q.dequeue()

15

16 **for** each node n that is adjacent to current:

17 **if** n.distance == INFINITY:

18 n.distance = current.distance + 1

19 n.parent = current

20 Q.enqueue(n)

**More details**[[edit](https://en.wikipedia.org/w/index.php?title=Breadth-first_search&action=edit&section=2" \o "Edit section: More details)]

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 ways:

1. it uses a [queue](https://en.wikipedia.org/wiki/Queue_(abstract_data_type)) instead of a [stack](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)) and
2. 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.

The *distance* attribute of each vertex (or node) is needed for example when searching for the shortest path between nodes in a graph. At the beginning of the algorithm, the distance of each vertex is set to *INFINITY*, which is just a word that represents the fact that a node has not been reached yet, and therefore it has no distance from the starting vertex. We could have used other symbols, such as *-1*, to represent this concept.

The *parent* attribute of each vertex can also be useful to access the nodes in a shortest path, for example by backtracking from the destination node up to the starting node, once the BFS has been run, and the predecessors nodes have been set.

The *NIL* is just a symbol that represents the absence of something, in this case it represents the absence of a parent (or predecessor) node; sometimes instead of the word *NIL*, words such as *null*, *none* or *nothing* can also be used.

Note that the word *node* is usually interchangeable with the word *vertex*.

Breadth-first search produces a so-called *breadth first tree*. You can see how a *breadth first tree* looks in the following example.

[wire routing](https://en.wikipedia.org/wiki/Routing_(electronic_design_automation)) algorithm (published 1961).[[3]](https://en.wikipedia.org/wiki/Breadth-first_search#cite_note-3)[[4]](https://en.wikipedia.org/wiki/Breadth-first_search#cite_note-4)

**Depth-first search** (**DFS**) 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. One starts at the [root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology)(selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before[backtracking](https://en.wikipedia.org/wiki/Backtracking).

A version of depth-first search was investigated in the 19th century by French mathematician [Charles Pierre Trémaux](https://en.wikipedia.org/wiki/Charles_Pierre_Tr%C3%A9maux)[[1]](https://en.wikipedia.org/wiki/Depth-first_search#cite_note-1) as a strategy for [solving mazes](https://en.wikipedia.org/wiki/Maze_solving_algorithm).[[2]](https://en.wikipedia.org/wiki/Depth-first_search#cite_note-2)[[3]](https://en.wikipedia.org/wiki/Depth-first_search#cite_note-3)

**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]](https://en.wikipedia.org/wiki/Depth-first_search" \l "cite_note-5)

1 **procedure** DFS(*G*,*v*):

2 label *v* as discovered

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

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

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

A non-recursive implementation of DFS:[[6]](https://en.wikipedia.org/wiki/Depth-first_search" \l "cite_note-6)

1 **procedure** DFS-iterative(*G*,*v*):

2 let *S* be a stack

3 *S*.push(*v*)

4 **while** *S* is not empty

5 *v* = *S*.pop()

6 **if** *v* is not labeled as discovered:

7 label *v* as discovered

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

9 *S*.push(*w*)

From programmerintervew.com

## Differences between DFS and BFS

|  |
| --- |
|  |

Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS, because it’s not necessary to store all of the child pointers at each level. Depending on the data and what you are looking for, either DFS or BFS could be advantageous.

For example, given a family tree if one were looking for someone on the tree who’s still alive, then it would be safe to assume that person would be on the bottom of the tree. This means that a BFS would take a very long time to reach that last level. A DFS, however, would find the goal faster. But, if one were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree. Then, a BFS would usually be faster than a DFS. So, the advantages of either vary depending on the data and what you’re looking for.

Data structure used

1. stack for dfs implementantion
2. queue for bfs implementation
3. array for both implementation

the priority for North ,east ,south,west

dfs doesn’t give the best solution but

bfs does because it visit all nodes at level distance from the source together

we can’t tell what is the better algorithm here . if you want just a path

dfs is better or if you want the shortest unweighted path you should use bfs

shortly dfs does

1. visit the node
2. mark as completed
3. push it onto the stack
4. go to the next node until you can’t reach
5. then go backword to the last node you can reach through it
6. repeat until get your target

while : bfs

1-visit the node

2-mark as completed

3-Visit all the neighbors put them in queue

1. dequeue
2. do the same event until you reach your target

important

I used scanner method to read from file .

{'#','.','.','.','S'},

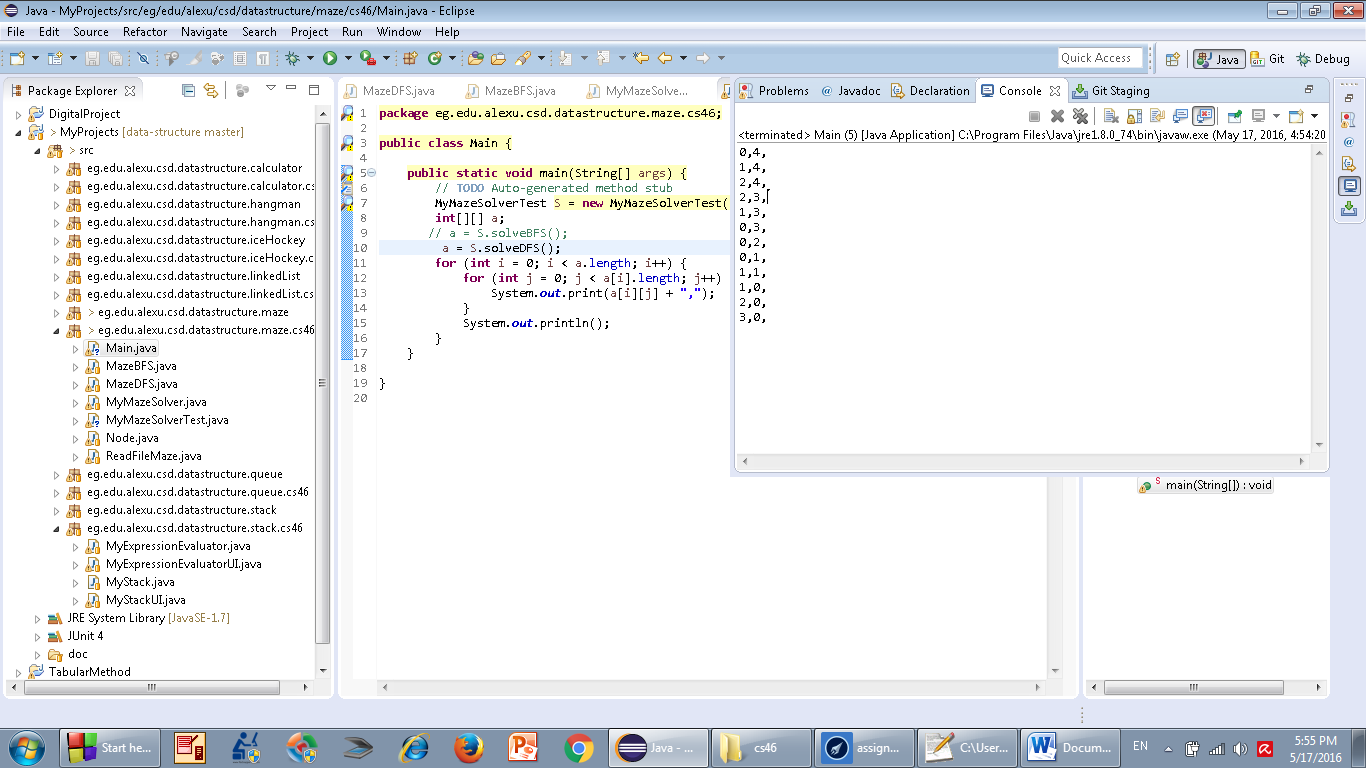
{'.','.','#','.','.'},

{'.','#','#','.','.'},

{'E','#','#','.','#'},

{'.','#','.','.','#'}

DFS



BFS

