**Implementing Depth-first search**

A pseudocode implementation of depth-first search is given below. The variable **state** represents the current state at any given point in the algorithm, and **queue** is a data structure that stores a number of states, in a form that allows insertion and removal from either end. In this algorithm, we always insert at the front and remove from the front, which as we will see later on means that depth-first search can be easily implemented using a stack. In this implementation, we have used the function **successors (state)**, which simply returns all successors of a given state.

Function depth () {

queue = []; // initialize an empty queue

state = root\_node; // initialize the start state

while (true) {

if is\_goal (state)

then return SUCCESS

else add\_to\_front\_of\_queue (successors (state));

if queue == []

then report FAILURE;

state = queue [0]; // state = first item in queue

remove\_first\_item\_from (queue);

}

}

**Table 4.2** shows the states that the variables **queue** and **state** take on when running the depth-first search algorithm over a simple search tree, as shown in Figure 4.3. In fact, depth-first search can be readily implemented on most computer systems using a **stack**, which is simply a “last in first out” queue (sometimes called a LIFO). In this way, a recursive version of the algorithm given above can be used, as follows. Because this function is recursive, it needs to be called with an argument:

recursive\_depth (root\_node);

The function is defined as follows:

Function **recursive\_depth** (state) {

if is\_goal (state)

then return SUCCESS

else {

remove\_from\_stack (state);

add\_to\_stack (successors (state))

}

while (stack != []) {

if recursive\_depth (stack [0]) == SUCCESS

then return SUCCESS;

remove\_first\_item\_from (stack);

}

return FAILURE;

}

If you run through this algorithm on paper (or in a programming language such as C++ or LISP), you will find that it follows the tree in the same way as the previous algorithm, **depth**.

**Table 4.2 Analysis of depth-first search of tree shown in Figure 4.5**

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **State** | **Queue** | **Notes** |
| 1 | A | (empty) | The queue starts out empty, and the initial state is the root node, which is A. |
| 2 | A | B,C | The successors of A are added to the queue. |
| 3 | B | C |  |
| 4 | B | D,E,C | The successors of the current state,B, are added to the front of the queue. |
| 5 | D | E,C |  |
| 6 | D | H,I,E,C |  |
| 7 | H | I,E,C | H has no successors, so no new nodes are added to the queue. |
| 8 | I | E,C | Similarly, I has no successors. |
| 9 | E | C |  |
| 10 | E | J,K,C |  |
| 11 | J | K,C | Again, J has no successors. |
| 12 | K | C | K has no successors. Now we have explored the entire branch below B, which means we backtrack up to C. |
| 13 | C | (empty) | The queue is empty, but we are not at the point in the algorithm where this would mean failing because we are about to add successors of C to the queue. |
| 14 | C | F.G |  |
| 15 | F | G |  |
| 16 | F | I,M,G |  |
| 17 | L | M,G | SUCCESS: the algorithm ends because a goal node has been located.In this case, it is the only goal node, but the algorithm does not know that and does not know how many nodes were left to explore. |

The queue starts out empty, and the initial state is the root node, which is A. The successors of A are added to the queue. The successors of the current state, B, are added to the front of the queue. H has no successors, so no new nodes are added

to the queue. Similarly, I have no successors. Again, J has no successors. K has no successors. Now we have explored the entire branch below B, which means we backtrack up to C. The queue is empty, but we are not at the point in the algorithm where this would mean failing because we are about to add successors of C to the queue. SUCCESS: the algorithm ends because a goal node has been located. In this case, it is the only goal node, but the algorithm does not know that and does not know how many nodes were left to explore.

**Implementing Breadth-first search**

As was mentioned previously, depth-first search and breadth-first search can be implemented very similarly. The following is a pseudocode of a non-recursive implementation of breadth-first search, which should be compared with the implementation above of depth-first search:

Function breadth () {

queue = []; // initialize an empty queue

state = root\_node; // initialize the start state

while (true) {

if is\_goal (state)

then return SUCCESS

else add\_to\_back\_of\_queue (successors (state));

if queue == []

then report FAILURE;

state = queue [0]; // state = first item in queue

remove\_first\_item\_from (queue);

}

}

Notice that the only difference between depth and breadth is that where depth adds successor states to the front of the queue, breadth adds them to the back of the queue. So when applied to the search tree in Figure 4.4, breadth will follow a rather different path from depth, as is shown in Table 4.3. You will notice that in this particular case, depth-first search found the goal in two fewer steps than breadth-first search. As has been suggested, depth-first search will often find the goal quicker than breadth-first search if all leaf nodes are the same depth below the root node. However, in search trees

where there is a very large subtree that does not contain a goal, breadth-first search will nearly always perform better than depth-first search. Another important factor to note is that the queue gets much longer when using breadth-first search. For large trees, and in particular for trees with high branching factors, this can make a significant difference because the depth-first search algorithm will never require a queue longer than the maximum depth of the tree, whereas breadth-first search in the worst case will need a queue equal to the number of nodes at the level of the tree with the most nodes (eight in a tree of depth three with branching factor of two, as in Figure 4.3). Hence, we say that depth-first search is usually more memory efficient than breadth-first search.

**Table 4.3 Analysis of breadth-first search of tree shown in Figure 4.4**

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **State** | **Queue** | **Notes** |
| 1 | A | (empty) | The queue starts out empty, and the initial state is the root node, which is A. |
| 2 | A | B,C | The two descendants of A are added to the  queue. |
| 3 | B | C |  |
| 4 | B | C,D,E | The two descendants of the current state,B, are added to the back of the queue. |
| 5 | C | D,E |  |
| 6 | C | D,E,F,G |  |
| 7 | D | E,F,G |  |
| 8 | D | E,F,G,H,I |  |
| 9 | E | F,G,H,I |  |
| 10 | E | F,G,H,I,J,K |  |
| 11 | F | G,H,I,J,K |  |
| 12 | F | G,H,I,J,K,L,M |  |
| 13 | G | H,I,J,K,L,M |  |
| 14 | G | H,I,J,K,L,M,N,O |  |
| 15 | H | I,J,K,L,M,N,O | H has no successors, so we have nothing to add to the queue in this state, or in fact for any subsequent states. |
| 16 | I | J,K,L,M,N,O |  |
| 17 | J | K,L,M,N,O |  |
| 18 | K | L,M,N,O |  |
| 19 | L | M,N,O | SUCCESS: A goal state has been reached |

As we have seen, however, depth-first search is neither optimal nor complete,

whereas breadth-first search is both. This means that depth-first search may not find the best solution and, in fact, may not ever find a solution at all. In contrast, breadth-first search will always find the best solution.

**Implementing Depth-first Iterative Deepening search**

**IDDFS** combines depth-first search’s space-efficiency and breadth-first search’s fast search (for nodes closer to root).

**How does IDDFS work?**  
IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

**Algorithm:**

// Returns true if target is reachable from src within max\_depth

**bool** IDDFS (src, target, max\_depth)

**for** limit **from** 0 **to** max\_depth

**if** DLS (src, target, limit) == **true**

**return** true

**return** **false**

**bool** DLS (src, target, limit)

**if** (src == target)

**return** **true**;

// If reached the maximum depth, stop recursing.

**if** (limit <= 0)

**return** **false**;

**for each** adjacent i of src

**if** DLS (i, target, limit?1)

**return** **true**

**return** **false**

**Best First Search (Informed Search)**

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

**Best-First-Search**(Graph g, Node start)

1) Create an empty PriorityQueue

PriorityQueue **pq**;

2) Insert "start" in pq.

pq.insert(start)

3) Until PriorityQueue is empty

u = PriorityQueue.DeleteMin

If u is the goal

Exit

Else

Foreach neighbor v of u

If v "Unvisited"

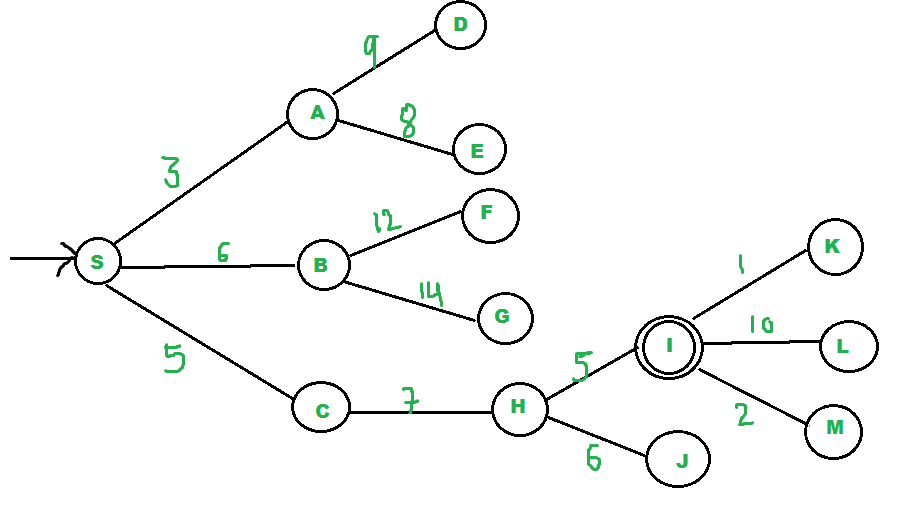
Mark v "Visited"

pq.insert(v)

Mark u "Examined"

End procedure

Let us consider the below example.

[](https://media.geeksforgeeks.org/wp-content/uploads/BFS2.png)