**BFS and DFS Algo**

In both the Algorithms we follow two key steps:

1. Visiting a vertex.
2. Exploration of a vertex:-Visiting all vertex attached to a vertex.

|  |  |
| --- | --- |
| **BFS** | **DFS** |
| Is a vertex based technique for finding a shortest path in graph. | Is a edge based technique |
| Queue data structure is used here | It uses the [Stack data structure](http://www.geeksforgeeks.org/stack-data-structure/), performs two stages, first visited vertices are pushed into stack and second if there is no vertices then visited vertices are popped. |
| Slower than DFS | Faster than BFS |
| The Time complexity of BFS is O(V + E), where V stands for vertices and E stands for edges. | The Time complexity of DFS is also O(V + E), where V stands for vertices and E stands for edges. |
| BFS is more suitable for searching verteces which are closer to the given source. | DFS is more suitable when there are solutions away from source. |

Note: there can be more than one graph for DFS and BFS.

**BFS:**

Algo:

BFS (G, s) //Where G is the graph and s is the source node

let Q be queue.

Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

mark s as visited.

while ( Q is not empty)

//Removing that vertex from queue,whose neighbour will be visited now

v = Q.dequeue( )

//processing all the neighbours of v

for all neighbours w of v in Graph G

if w is not visited

Q.enqueue( w ) //Stores w in Q to further visit its neighbour

mark w as visited.

Applications of BFS –

* Copying garbage collection,Cheney’s algorithm
* Finding the shortest path between two nodes u and v, with path length measured by number of edges (an advantage over depth first search)
* Testing a graph for bipartiteness
* Minimum Spanning Tree for unweighted graph.
* Web crawler
* Finding nodes in any connected component of a graph
* Ford–Fulkerson method for computing the maximum flow in a flow network
* Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

**DFS**

**Pseudocode**

**DFS-iterative (G, s):** //Where G is graph and s is source vertex

let S be stack

S.push( s ) //Inserting s in stack

mark s as visited.

while ( S is not empty):

//Pop a vertex from stack to visit next

v = S.top( )

S.pop( )

//Push all the neighbours of v in stack that are not visited

for all neighbours w of v in Graph G:

if w is not visited :

S.push( w )

mark w as visited

**DFS-recursive(G, s):**

mark s as visited

for all neighbours w of s in Graph G:

if w is not visited:

DFS-recursive(G, w)

**CODE of DFS and DFS recursion**

**BFS:**

**public** **class** BFSRecurrsion {

**public** **boolean**[] visited;

**public** Queue<Integer> queue = **new** LinkedList<Integer>();

**public** **void** printBfsOrder(**int**[][] arr) {

**if**(queue.isEmpty()) {

**return**;

}

**int** vertex = queue.remove();

visited[vertex] = **true**;

System.***out***.println(" BFS vertex = "+vertex);

**for** (**int** i = 0; i < arr.length; i++) {

**if** (!visited[i] && arr[vertex][i] != 0) {

queue.add(i);

}

}

printBfsOrder(arr);

}

}

**DFS:**

**public** **class** DFSRecurrsion {

**public** **boolean**[] visited;

**public** **void** printDfsOrder(**int**[][] arr, **int** vertex) {

System.***out***.println(" DFS vertex = "+vertex);

visited[vertex]= **true**;

**for** (**int** i = 0; i < arr.length; i++) {

**if**(!visited[i] && arr[vertex][i]!=0) {

printDfsOrder(arr,i);

}

}

}

}