Table of Contents

[Graph Traversal 1](#_Toc115205015)

[Graph Traversal Algorithms 2](#_Toc115205016)

[Depth-First Search 3](#_Toc115205017)

[**Algorithm Overview** 3](#_Toc115205018)

[**Mark Visited Nodes** 3](#_Toc115205019)

[**Traversal Order** 3](#_Toc115205020)

[**Example** 4](#_Toc115205021)

[**DFS Recursive Algorithm** 4](#_Toc115205022)

[**DFS Observations** 4](#_Toc115205023)

[**DFS Iterative Algorithm** 5](#_Toc115205024)

[**Example** 5](#_Toc115205025)

[**Runtime** 7](#_Toc115205026)

[Breadth-First Search 7](#_Toc115205027)

[**BFS Overview** 7](#_Toc115205028)

[**Algorithm Overview** 7](#_Toc115205029)

[**Implementing BFS** 11](#_Toc115205030)

[**BFS Pseudocode** 13](#_Toc115205031)

[**DFS and BFS Runtime** 14](#_Toc115205032)

# **Graph Traversal**

**Graph Traversal**

* A **graph traversal** visits all vertices that it can reach from a starting vertex ***v***.
* In other words, a graph traversal that starts at vertex ***v***will visit all vertices ***u***for which there is a path between ***v***and ***u***.
* The ***order*** vertices are traversed in depends on the ***type***of graph you are traversing.
  + **Connected Graph**
    - A connected graph traversal can visit every vertex in the graph.
    - The starting vertex does not matter.
    - Thus, you can use a graph traversal to determine whether a graph is connected.
  + **Not Connected**
    - If a graph is not connected, a graph traversal that begins at vertex v will visit only a subset of the graph’s vertices.
    - This subset is called the *connected component* containing *v*.
    - You can determine all of the connected components of a graph by repeatedly starting a traversal at an unvisited vertex.
  + **Cycle**
    - If a graph contains a cycle, a graph-traversal algorithm can loop indefinitely.
    - To prevent such a misfortune, the algorithm must mark each vertex during a visit and must never visit a vertex more than once.
* A **tree traversal** always visits all nodes in a tree.
  + There are different ways to traverse a tree, such as inorder, preorder, and postorder.
* A graph traversal may or may not visit all nodes in a graph, depending on the type of graph and the problem constraints.

# **Graph Traversal Algorithms**

**Traversal Algorithms**

* There are two basic graph-traversal algorithms.

1. **Depth-First Traversal (DFS)**
2. **Breadth-First Traversal (BFS)**

* These algorithms apply to both directed and undirected graphs.
* These algorithms visit vertices in **different orders**, but ***if they both start at the same vertex***, they will ***visit the same set of vertices***.

**Example**

* The following figure shows the traversal order of DFS vs. BFS on an undirected graph.
* Both algorithms begin at vertex *v*.

A picture containing diagram

Description automatically generated

# **Depth-First Search**

## **Algorithm Overview**

* **Depth-First Search (DFS)** is a graph traversal strategy.
* **DFS** finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  + Often implemented ***recursively*** (but can be implemented iteratively).
  + Many graph algorithms involve ***visiting*** or ***marking*** vertices.

Diagram, schematic

Description automatically generated

* Depth-first paths from ***a*** to all other vertices (assuming ABC edge order):
  + to b: {a,b}
  + to c: {a,b,e,f,c}
  + to d: {a,d}
  + to e: {a,b,e}
  + to f: {a,b,e,f}
  + to g: {a,d,g}
  + to h: {a,d,g,h}

### **Mark Visited Nodes**

* If this process is performed on a **tree**
  + All tree vertices are systematically visited in a total of *O*(|*E*|) time, since |*E*| = *O*(|*V*|).
* If we perform this process on an **arbitrary graph**
  + We need to be careful to ***avoid cycles***.
  + To do this, when we visit a vertex *v*, we ***mark it visited***, since now we have been there, and we recursively call depth-first search on all adjacent vertices that are not already marked.
* We implicitly assume that for undirected graphs every edge (*v*, *w*) appears twice in the adjacency lists: once as (*v*, *w*) and once as (*w*, *v*).

### **Traversal Order**

* How do we determine **which unvisited adjacent vertex to visit**?
  + DFS does not have a standard for the order in which vertices adjacent to *v* are visited.
  + One possibility is to visit the vertices adjacent to *v* in natural sorted order (e.g., alphabetic, or numerically increasing).
  + This possibility is natural either when an adjacency matrix represents the graph or when the nodes in each linked chain of an adjacency list are linked in sorted order.

### **Example**

* The DFS order the nodes in Figure 20-10a (above) are visited in: *v*, *u*, *q*, and *r* until it reaches a vertex—such as *r*—that has **no unvisited adjacent vertices**.
* When this happens, it backs up to the previously visited node, and if possible, visits another unvisited adjacent vertex.
* Thus, the traversal backs up from *r* to *q* and then visits *s*.
* Continuing in this manner, the traversal visits vertices in the order given in the figure.

## **DFS Recursive Algorithm**

// *Main function that calls helper DFS function*

public void **dfs**(Vertex v1, Vertex v2)

path = dfs(v1, v2, {});

// *Traverses a graph beginning at vertex* v *using recursive DFS*

private void **dfs**(Vertex v1, Vertex v2, Path path)

path += v1;

v.visited = true;

if (v1 is v2)

a path is found!

**for** (each unvisited Vertex u adjacent to Vertex v)

if (!u.visited)

dfs(u, v2, path);

* The ***path*** parameter above is used if you want to have the path available as a list once you are done.
* By recursively calling the procedures only on nodes that have not been visited, we guarantee that we do not loop indefinitely (in the case of cyclic graphs).
* We then search for an unmarked node, apply a depth-first traversal there, and continue this process until there are no unmarked nodes.

## **DFS Observations**

* **Discovery:**
  + DFS is guaranteed to find ***a*** path if one exists.
  + This does not mean it finds the most optimal (i.e, shortest) path.

Example: dfs(a,f) returns {a,d,c,f} rather than {a,d,f}.

* **Retrieval:**
  + It is easy to retrieve exactly what the path taken (sequence of edges) is if we find it.
* **Optimality:**
  + If the graph is undirected and not connected, or directed and not strongly connected, this strategy might fail to visit some nodes.

## **DFS Iterative Algorithm**

//  *Traverses a graph beginning at vertex* v *by using a*

//  *depth-first search: Iterative version.* dfs(v: Vertex)

public void **dfs**(Vertex v)

s = *a new empty stack*

// *Push* v *onto the stack and mark it*

s.push(v)

*Mark* v *as visited*

//  *Loop invariant: there is a path from vertex* v *at the bottom of the stack* s *to the vertex at the top of* s

**while** (!s.isEmpty())

{

// pop a vertex from the stack

v = s.pop();

**if** (*no unvisited vertices are adjacent to the vertex on the top of the stack*)

{

s.pop() // *Backtrack*

}

**else**

{

*Select an unvisited vertex* u *adjacent to the vertex on the top of the stack* s.push(u)

*Mark* u *as visited*

}

}

## **Example**

* For another example of a DFS traversal, consider the graph in Figure 20-11.
* Figure 20-12 shows the contents of the stack as the previous function dfs visits vertices in this graph, beginning at vertex *a.*
* Because the graph is connected, a DFS traversal will visit every vertex.
* In fact, the traversal visits the vertices in this order: *a, b, c, d, g, e, f, h, i*.
* The vertex from which a depth-first traversal embarks is the vertex that it visited most recently.
* This *last visited, first explored* strategy is reflected both in the explicit stack of vertices that the iterative dfs uses and in the implicit stack of vertices that the recursive dfs generates with its recursive calls.

Chart

Description automatically generated

Table

Description automatically generated

## **Runtime**

* Because this strategy guarantees that each edge is encountered only once, the total time to perform the traversal is *O*(|*E*| + |*V*|), as long as adjacency lists are used.

# **Breadth-First Search**

## **BFS Overview**

* **Breadth-First Search (BFS)** is a graph traversal strategy.
* Breadth-First Search finds a path between two nodes by taking **one step down all paths** (visiting all **adjacent** vertexes) and then immediately **backtracking**.
* It is often implemented by maintaining a **queue** of **vertices to visit**.
* BFS always returns the **shortest path** (the one with the fewest edges) between the start and the end vertices.

Diagram, schematic

Description automatically generated

* Depth-first paths from ***a*** to all other vertices (assuming ABC edge order):
  + to b:{a,b}
  + to c: {a,e,f,c}
  + to d:{a,d}
  + to e:{a,e}
  + to f: {a,e,f}
  + to g:{a,d,g}
  + to h:{a,d,h}

## **Algorithm Overview**

* Figure 9.10 shows an unweighted graph, *G*.

A picture containing watch, old

Description automatically generated

* **Input:**
  + We are given a starting vertex, *s* as an input parameter.
* **Goal:**
  + We would like to find the shortest path from *s* to all other vertices.
* **Return:**
  + The **length** of the shortest path from *s* to all other vertices.
* **Constraints:**
  + We are only interested in the number of edges contained on the path, so there are no weights on the edges.
  + This is clearly a special case of the weighted shortest-path problem, since we could assign all edges a weight of 1.
* For now, suppose we are interested only in the length of the shortest paths, not in the actual paths themselves. Keeping track of the actual paths will turn out to be a matter of simple bookkeeping.

**Example Walkthrough**

1. Suppose we choose *s* to be *v*3. Immediately, we can tell that the shortest path from *s* to *v*3 is then a path of length 0.

We can mark this information, obtaining the graph in Figure 9.11.

A picture containing watch, old

Description automatically generated

1. Now we can start looking for all vertices that are a **distance 1 away** **from** ***s***.

These can be found by looking at the **vertices that are** **adjacent** to ***s***.

If we do this, we see that *v*1 and *v*6 are one edge from *s*.

This is shown in Figure 9.12.

A picture containing watch, old

Description automatically generated

1. We can now find vertices whose **shortest path from *s* is exactly 2**.

We find all the vertices adjacent to *v*1 and *v*6 (the vertices at distance 1), whose shortest paths are not already known.

This search tells us that the shortest path to *v*2 and *v*4 is 2.

Figure 9.13 shows the progress that has been made so far.

A picture containing watch, old

Description automatically generated

1. Finally, examine the vertices adjacent to the recently evaluated *v*2 and *v*4.

This search tells us that the shortest path to *v*5 and *v*7 is 3.

All vertices have now been calculated, and so Figure 9.14 shows the final result of the algorithm.

A picture containing watch, old, fashioned

Description automatically generated

* This strategy for searching a graph is known as **breadth-first search**.
* It operates by processing vertices in layers:

1. The vertices closest to the start are evaluated first

and

1. The most distant vertices are evaluated last

* This is much the same as a **level-order traversal for trees**.

## **Implementing BFS**

**Variables**

* Given this strategy, we must translate it into code.
* Figure 9.15 shows the initial configuration of the table that our algorithm will use to keep track of its progress.

Table

Description automatically generated

* For each vertex, we will keep track of three pieces of information.

1. First, we will keep its distance from s in the entry **dv**.
   1. Initially all vertices are unreachable except for s, whose path length is 0.
2. The entry in **pv** is the bookkeeping variable, which will allow us to print the actual paths.
3. The entry **known** is set to **true** after a vertex is processed.
   1. Initially, all entries are **not known**, including the start vertex.
   2. When a vertex is marked known, we have a guarantee that no cheaper path will ever be found, and so processing for that vertex is essentially complete.

**Algorithm**

* The basic algorithm can be described in Figure 9.16.
* The algorithm in Figure 9.16 mimics the diagrams by declaring as **known** the vertices at distance d = 0, then d = 1, then d = 2, and so on, and setting all the adjacent vertices w that still have dw = ∞ to a distance dw = d + 1.

Text

Description automatically generated

* By tracing back through the pv variable, the actual path can be printed.
* We will see how when we discuss the weighted case.
* The running time of the algorithm is O(|V|2), because of the doubly nested for loops.
* An obvious inefficiency is that the outside loop continues until NUM\_VERTICES − 1, even if all the vertices become known much earlier.
* Although an extra test could be made to avoid this, it does not affect the worst-case running time, as can be seen by generalizing what happens when the input is the graph in Figure 9.17 with start vertex v9.
* We can remove the inefficiency in much the same way as was done for topological sort.
* At any point in time, there are only two types of unknown vertices that have dv ̸= ∞.
* Some have dv = currDist and the rest have dv = currDist + 1.
* Because of this extra structure, it is very wasteful to search through the entire table to find a proper vertex.
* A very simple but abstract solution is to keep two boxes.
* Box #1 will have the unknown vertices with dv = currDist, and box #2 will have dv = currDist + 1. The test to find an appropriate vertex v can be replaced by finding any vertex in box #1. After updating w (inside the innermost if block), we can add w to box #2. After the outermost for loop terminates, box #1 is empty, and box #2 can be transferred to box #1 for the next pass of the for loop.
* We can refine this idea even further by using just one queue. At the start of the pass, the queue contains only vertices of distance currDist. When we add adjacent vertices of distance currDist + 1, since they enqueue at the rear, we are guaranteed that they will not be processed until after all the vertices of distance currDist have been processed. After the
* We know that we can immediately access all vertices adjacent to the source vertex with a path of length one.
* We then search the neighbors of all of these vertices to find paths of length two.
* … and, so on, until all edges have been considered.
* This is much the same as a level-order traversal for trees.
* We can extend the idea to a graph
* Instead of “child” in tree, use "neighbor" in graph (adjacent)
* Keep track of paths
* Find BFS ordering in those graphs
* For now, suppose we are interested only in the length of the shortest paths, not in the actual paths themselves.
* Therefore, we need to keep track of each path we traverse down.
* This is a matter of simple bookkeeping.

## **BFS Shortest Path Pseudocode**

public void **bfs**(Vertex v1, Vertex v2)

// *Initialize queue and add* v1 (head) *onto the queue and mark it visited*

queue = *a new empty queue*

add v1 to queue

*Mark* v1 *as visited*

**while**(*queue* is not empty)

v = queue.removeFirst( ); // remove current node

if (v1 is v2) // use for shortest path from v1 to v2

a path is found!

**for**(each unvisited neighbor *n* of *v*)

mark *n* as visited // avoid cycles by marking n as visited

queue.addLast(*n*); // add n to queue

// path is not found

## **BFS Path Pseudocode**

public void **bfs**(Vertex v1, Vertex v2)

// *Initialize queue and add* v1 *onto the queue and mark it visited*

queue = *a new empty queue*

add v1 to queue

*Mark* v1 *as visited*

**while**(*queue* is not empty)

v = queue.removeFirst( );

if (v1 is v2)

a path is found! (reconstruct it by following .prev back to v)

**for**(each unvisited neighbor *n* of *v*)

mark *n* as visited (set n.prev = v)

queue.addLast(*n*);

// path is not found

## **BFS Traversal Pseudocode**

If there are multiple vertices you must traverse at each BFS interval, traverse over every node that is currently in the queue.

public void **bfs**(Graph *g*)

// *Initialize queue and add* {vs} *onto the queue and mark them as visited*

queue = *a new empty queue*

add all starter vertices {vs} from *g* to queue

mark vertices in {vs} as visited

**while**(*queue* is not empty)

n = *queue* size

**for**(every node *currently* in queue)

v = queue.removeFirst( ); // remove current node

**for**(each adjacent neighbor *n* of *v*)

if(*n* is unvisited)

mark *n* as visited // avoid cycles by marking n as visited

queue.addLast(*n*); // add n to queue

**Optimality**

* Always finds the **shortest path** (fewest edges).
* In **unweighted graphs**, finds **optimal cost path**.
* In **weighted graphs**, **not always optimal cost**

**Retrieval**

* Harder to reconstruct the actual sequence of vertices or edges in the path once you find it.
* Conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress.
* Solution: We can keep track of the path by storing predecessors for not easy to store a path array/list in progress each vertex (each vertex can store a reference to a previous vertex).

**DFS vs BFS**

* DFS uses less memory than BFS
* DFS is easier to reconstruct the path when found
* DFS does not always find the shortest path (for unweighted graphs). BFS does.

Thus, we have solved the **Single-Source Unweighted Shortest-Path Problem**.

## **DFS and BFS Runtime**

**Adjacency List**

* What is the expected runtime of DFS and BFS, in terms of the number of vertices V and the number of edges E?
* Answer: O(|V| + |E|)
  + where |V| = number of vertices, |E| = number of edges
  + Must potentially visit every node and/or examine every edge once.
* Assuming that the graph is connected, each vertex is placed on the queue/stack exactly once.
  + (We mark the vertex as visited immediately after we put it on the queue/stack and this prevents it from every being placed on the queue/stack again.)
* When we examine each vertex, we consider each of its edges exactly once.
* This implies that the algorithm requires O(V + E) time for an adjacency list, where
  + V is the number of vertices

and

* + E is the number of edges
* This is O(E) if the graph is connected, since there must be at least O(V) edges in that case.

**Adjacency Matrix**

* An adjacency matrix holds O(V2) indices. We must visit every index, even if it does not hold a vertex. Therefore, DFS/BFS for an adjacency matrix requires O(V2) time.

Table

Description automatically generated