

# Graph

quarta-feira, 16 de outubro de 2024 13:50

## → Graphs Algorithms

Solving in graph means systematically following the edges of the graph to visit all the vertices

Graph → Way of encoding pairwise relationship among a set of objectives

↳ Collection of  $\boxed{V}$  nodes and  $\boxed{E}$  edges, which joins the nodes

$$\left\{ \begin{array}{l} u, v \in V \\ e \in E \end{array} \right. \Rightarrow e = \{u, v\}$$

on  $\boxed{\text{EDGES}}$  connects two other nodes! node is vertex

↳ Evaluates a symmetric relationship between nodes

### • DIRECTED Graph ( $G^1$ )

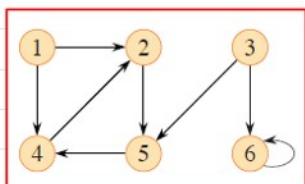
↳ When we want to encode asymmetric relations

↳ Consist in a set of nodes ( $V$ ) and a set of directed edges ( $E^1$ ) based on an ordered pair  $(v, w)$

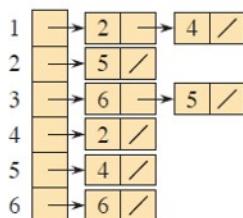
$\left\{ \begin{array}{l} v \rightarrow \text{tail} \\ w \rightarrow \text{head} \end{array} \right.$  edge  $e^1$  leaves node  $v$  and enter node  $w$ !

↳ Has a direction going from one node to the next

Direction does matter



(a)



(b)

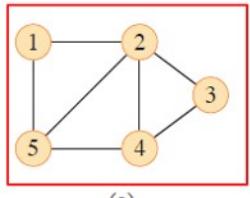
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

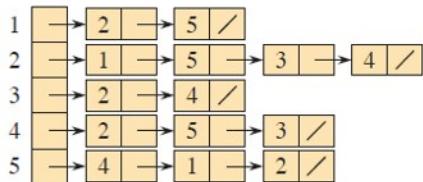
Figure 20.2 Two representations of a directed graph. (a) A directed graph  $G$  with 6 vertices and 8 edges. (b) An adjacency-list representation of  $G$ . (c) The adjacency-matrix representation of  $G$ .

### • UNDIRECTED Graph ( $G^1$ )

↳ By default → when called a graph → It means that it's undirected  
↳ It can go both ways ( $A \rightarrow B$ ) ( $B \rightarrow A$ )



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

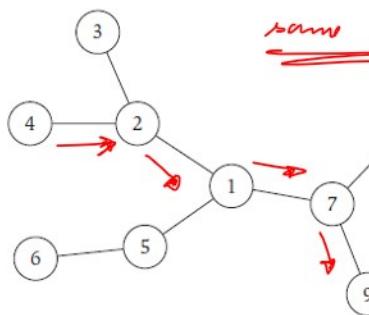
Figure 20.1 Two representations of an **undirected graph**. (a) An undirected graph  $G$  with 5 vertices and 7 edges. (b) An adjacency-list representation of  $G$ . (c) The adjacency-matrix representation of  $G$ .

→ Note that the edges can flow both ways

• **PATH**

↳ Used to represent a traversing sequence of edges (not paths)

→ One operation on graphs



some examples

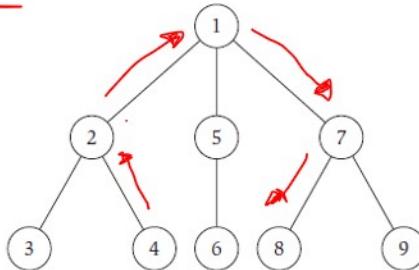


Figure 3.1 Two drawings of the same tree. On the right, the tree is rooted at node 1.

• **Simple Path** → if all vertices are distinct from one another

• **Cycle** → Path that comes back to the start point

a path  $v_1, v_2, \dots, v_{k-1}, v_k$  in which  $k > 2$ , and  $v_1 = v_k$ —in other words, the sequ

• for the direct graph, it must respect the edge connectivity

↳ Strongly connected when able to move both ways  $(A \rightarrow B)$   
 $(B \rightarrow A)$

• **DISTANCE** → Between 2 nodes → shortest distance

↳ minimum number of edges in the path

→ **Trees** We say that an undirected graph is a *tree* if it is connected and does not contain a cycle. For example, the two graphs pictured in Figure 3.1 are trees. In a strong sense, trees are the simplest kind of connected graph: deleting any edge from a tree will disconnect it.

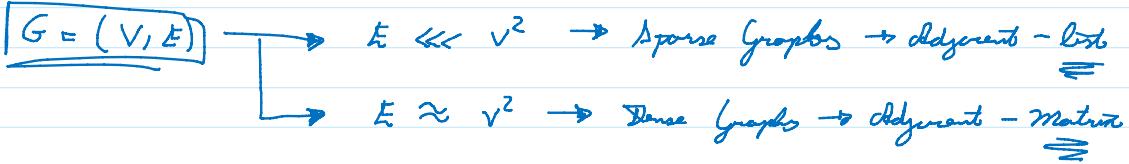
→ Every  $(N)$  node tree has exactly  $(N-1)$  edges

## Representation of graphs

$G = (V, E)$  → Collection of adjacent lists or adjacency matrix

Apply to direct and indirect graphs

(prefer when the graph is dense)



### • Adjacent - list representation

$G = (V, E)$  → array with adj [V] list → one for each vertex

↳ The adj list contains all the vertices  $v$  connected by edges from the original vertex

an undirected graph, the sum of the lengths of all the adjacency lists is  $2|E|$ , since if  $(u, v)$  is an undirected edge, then  $u$  appears in  $v$ 's adjacency list and vice versa. For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is  $\Theta(V + E)$ . Finding each edge in the graph also takes  $\Theta(V + E)$  time, rather than just  $\Theta(E)$ , since each of the  $|V|$  adjacency lists must be examined. Of course, if  $|E| = \Omega(|V|)$  — such as in a connected, undirected graph or a strongly connected, directed graph — we can say that finding each edge takes  $\Theta(E)$  time.

Adjacency lists can also represent **weighted graphs** that is, graphs for which each edge has an associated **weight** given by a **weight function**  $w : E \rightarrow \mathbb{R}$ . For example, let  $G = (V, E)$  be a weighted graph with weight function  $w$ . Then you can simply store the weight  $w(u, v)$  of the edge  $(u, v) \in E$  with vertex  $v$  in  $u$ 's adjacency list. The adjacency-list representation is quite robust in that you can modify it to support many other graph variants.

### DISADVANTAGE

↳ No queries may to determine if a given edge  $(v, w)$  is present in the graph

↳ Have to search ( $w$ ) in the  $\text{Adj}[v]$  list

### • Adjacent - Matrix representation

$G = (V, E)$  → Vertices are numbered  $\{1, 2, \dots, |V|\}$  so that the representation consists on a  $|V| \times |V|$  matrix

$$A_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Disadvantages → Requires more memory to store it  $\Theta(|V|^2)$

But it's simple to verify if edge  $(v, w)$  exists!

↳ Independent of the edges

↳ Disadvantages from the List representation

Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph. For example, if  $G = (V, E)$  is a weighted graph with edge-weight function  $w$ , you can store the weight  $w(u, v)$  of the edge  $(u, v) \in E$

Although the adjacency-list representation is asymptotically at least as space-efficient as the adjacency-matrix representation, adjacency matrices are simpler, and so you might prefer them when graphs are reasonably small. Moreover, adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

→ Representations of vertices to store information

### Representing attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. We indicate these attributes using our usual notation, such as  $v.d$  for an attribute  $d$  of a vertex  $v$ . When we indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have an attribute  $f$ , then we denote this attribute for edge  $(u, v)$  by  $(u, v).f$ . For the purpose of presenting and understanding algorithms, our attribute notation suffices.

Implementing vertex and edge attributes in real programs can be another story entirely. There is no one best way to store and access vertex and edge attributes. For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph. If you represent a graph using adjacency lists, one design choice is to represent vertex attributes in additional arrays, such as an array  $d[1 : |V|]$  that parallels the  $Adj$  array. If the vertices adjacent to  $u$  belong to  $Adj[u]$ , then the attribute  $u.d$  can actually be stored in the array entry  $d[u]$ . Many other ways of implementing attributes are possible. For example, in an object-oriented programming language, vertex attributes might be represented as instance variables within a subclass of a `Vertex` class.

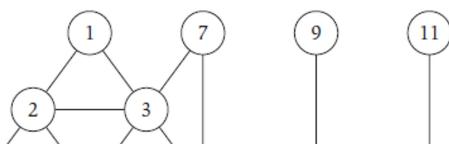
### BREADTH-FIRST SEARCH

→ Works for direct & indirect graphs

Explore outward from  $S$  in all possible directions, adding nodes one layer at a time, that can be reachable (by edges)

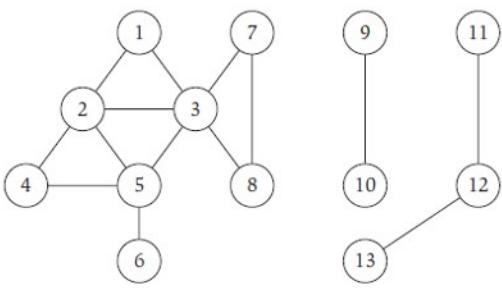
↳ Start in node  $S$  and include all nodes connected to  $S$  by an edge

↳ Then, include all additional nodes connected to the nodes from the first layer → Generating the second layer



\*  $S = \text{node } 1$  Layer 0 (1)

Layer 1: 2 3 Neighbors of  $S$



$s = \text{node } 1$       Layer 0 (L<sub>0</sub>)  
 Layer 1: 2 3      Neighbors of s  
 Layer 2: 4 5 7 8  
 Layer 3: 6

As this example reinforces, there is a natural physical interpretation to the algorithm. Essentially, we start at  $s$  and “flood” the graph with an expanding wave that grows to visit all nodes that it can reach. The layer containing a node represents the point in time at which the node is reached.

We can define the layers  $L_1, L_2, L_3, \dots$  constructed by the BFS algorithm more precisely as follows.

Recalling our definition of the distance between two nodes as the minimum number of edges on a path joining them, we see that layer  $L_1$  is the set of all nodes at distance 1 from  $s$ , and more generally layer  $L_j$  is the set of all nodes at distance exactly  $j$  from  $s$ . A node fails to appear in any of the layers if and only if there is no path to it. Thus, BFS is not only determining the nodes that  $s$  can reach, it is also computing shortest paths to them. We sum this up in the following fact.

(3.3) For each  $j \geq 1$ , layer  $L_j$  produced by BFS consists of all nodes at distance exactly  $j$  from  $s$ . There is a path from  $s$  to  $t$  if and only if  $t$  appears in some layer.

→ If the node doesn't appear in the layers, it can't be reached!

→ BFS → Produces a rooted tree at  $s$ , showing the nodes reachable from  $s$ . (BFS search + BFS) (Gr)

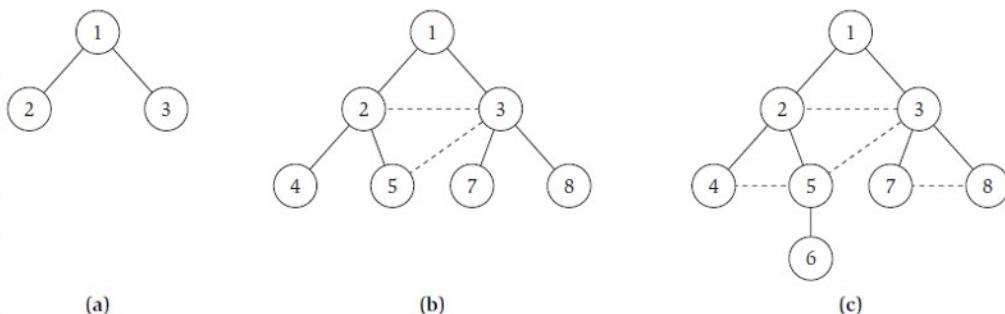


Figure 3.3 The construction of a breadth-first search tree  $T$  for the graph in Figure 3.2, with (a), (b), and (c) depicting the successive layers that are added. The solid edges are the edges of  $T$ ; the dotted edges are in the connected component of  $G$  containing node 1, but do not belong to  $T$ .

We notice that as we ran BFS on this graph, the nontree edges all either connected nodes in the same layer, or connected nodes in adjacent layers. We now prove that this is a property of BFS trees in general.

(3.4) Let  $T$  be a breadth-first search tree, let  $x$  and  $y$  be nodes in  $T$  belonging to layers  $L_i$  and  $L_j$  respectively, and let  $(x, y)$  be an edge of  $G$ . Then  $i$  and  $j$  differ

→ Discovering vertices in waves / layers  
 → Uses a queue (FIFO)  
 containing 46 vertices  
 → Also need to flag the vertex to see:  
 { \* if it's reachable or not  
 { when the V was found  
 → Define the frontier of the discovered vertices

→ Each vertex is discovered once (can be referenced later on) but it has one parent only  
 → Only  $s$  has no parent

(3.4) Let  $T$  be a breadth-first search tree, let  $x$  and  $y$  be nodes in  $T$  belonging to layers  $L_i$  and  $L_j$  respectively, and let  $(x, y)$  be an edge of  $G$ . Then  $i$  and  $j$  differ by at most 1.

parent only  
↳ Only  $s$  has no parent

Now, if one thinks about it, it's clear that BFS is just one possible way to produce this component. At a more general level, we can build the component  $R$  by "exploring"  $G$  in any order, starting from  $s$ . To start off, we define  $R = \{s\}$ . Then at any point in time, if we find an edge  $(u, v)$  where  $u \in R$  and  $v \notin R$ , we can add  $v$  to  $R$ . Indeed, if there is a path  $P$  from  $s$  to  $u$ , then there is a path from  $s$  to  $v$  obtained by first following  $P$  and then following the edge  $(u, v)$ . Figure 3.4 illustrates this basic step in growing the component  $R$ .

Suppose we continue growing the set  $R$  until there are no more edges leading out of  $R$ ; in other words, we run the following algorithm.

```

R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
    Add v to R
Endwhile

```

Here is the key property of this algorithm.

(3.5) The set  $R$  produced at the end of the algorithm is precisely the connected component of  $G$  containing  $s$ .

```

BFS(G, s)
1 for each vertex u ∈ G.V - {s}
2   u.color = WHITE
3   u.d = ∞
4   u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10 while Q ≠ ∅
11   u = DEQUEUE(Q)
12   for each vertex v in G.Adj[u]           // consider an adj-list
13     if v.color == WHITE                 // search the neighbors of u
14       v.color = GRAY                  // is v being discovered now?
15       v.d = u.d + 1                  // not yet discovered
16       v.π = u
17       ENQUEUE(Q, v)                // v is now on the frontier
18   u.color = BLACK                   // u is now behind the frontier

```

→ Enqueue vertices → all white with min distance from the source ( $s$ ) & no parent

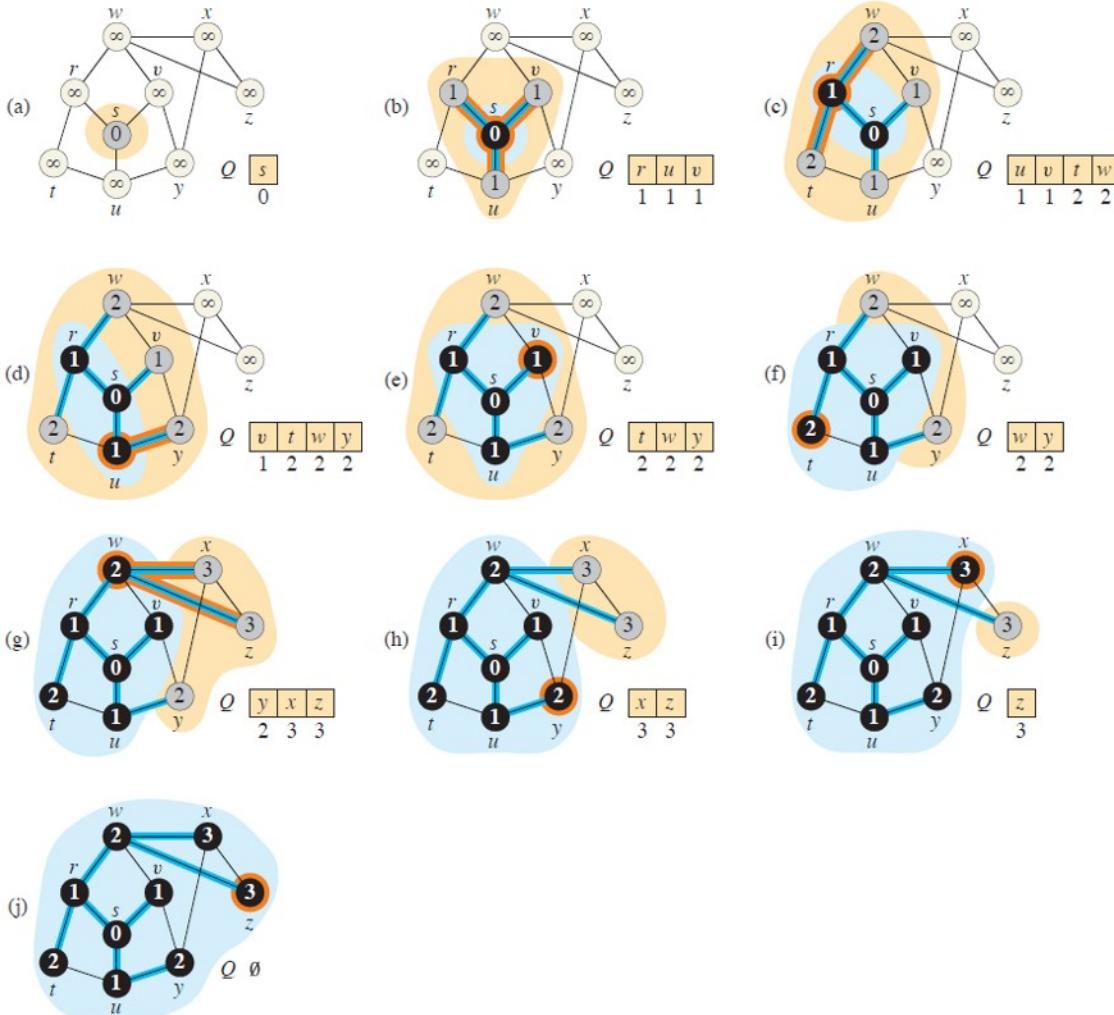
→ Source vertex → gray (evaluated), distance = 0  
and has no parent

→ Creates the queue with just the source vertex

→ Handles all the remaining vertices  
discovering vertices  
not yet examined

↳ General → for a graph with  $n$  nodes at most  $n-1$  edges

→ The resulting tree may differ depending on how it's processed + the adj-list → But the distance doesn't change



**Figure 20.3** The operation of BFS on an undirected graph. Each part shows the graph and the queue  $Q$  at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear within each vertex and below vertices in the queue. The tan region surrounds the frontier of the search, consisting of the vertices in the queue. The light blue region surrounds the vertices behind the frontier, which have been dequeued. Each part highlights in orange the vertex dequeued and the breadth-first tree edges added, if any, in the previous iteration. Blue edges belong to the breadth-first tree constructed so far.

## Analysis

Before proving the various properties of breadth-first search, let's take on the easier job of analyzing its running time on an input graph  $G = (V, E)$ . We use aggregate analysis, as we saw in Section 16.1. After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueueing and dequeuing take  $O(1)$  time, and so the total time devoted to queue operations is  $O(V)$ . Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all  $|V|$  adjacency lists is  $\Theta(E)$ , the total time spent in scanning adjacency lists is  $O(V + E)$ . The overhead for initialization is  $O(V)$ , and thus the total running time of the BFS procedure is  $O(V + E)$ . Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of  $G$ .

Depends linearly around the (vertices) and (edges)

→ Shortest Path distance  $\delta(s, v)$  ( $s \rightarrow v$ )

↳ Minimal number of edges between  $s$  and  $v$

↳ If no path ( $\delta(s, v) = \infty$ )

### Lemma 20.1

Let  $G = (V, E)$  be a directed or undirected graph, and let  $s \in V$  be an arbitrary vertex. Then, for any edge  $(u, v) \in E$ ,

$$\delta(s, v) \leq \delta(s, u) + 1.$$

### Lemma 20.2

Let  $G = (V, E)$  be a directed or undirected graph, and suppose that BFS is run on  $G$  from a given source vertex  $s \in V$ . Then, for each vertex  $v \in V$ , the value  $v.d$  computed by BFS satisfies  $v.d \geq \delta(s, v)$  at all times, including at termination.

### Lemma 20.3

Suppose that during the execution of BFS on a graph  $G = (V, E)$ , the queue  $Q$  contains the vertices  $\{v_1, v_2, \dots, v_r\}$ , where  $v_1$  is the head of  $Q$  and  $v_r$  is the tail. Then,  $v_r.d \leq v_1.d + 1$  and  $v_i.d \leq v_{i+1}.d$  for  $i = 1, 2, \dots, r - 1$ .

*d value  
always increase*

### Corollary 20.4

Suppose that vertices  $v_i$  and  $v_j$  are enqueued during the execution of BFS, and that  $v_i$  is enqueued before  $v_j$ . Then  $v_i.d \leq v_j.d$  at the time that  $v_j$  is enqueued.

### Theorem 20.5 (Correctness of breadth-first search)

Let  $G = (V, E)$  be a directed or undirected graph, and suppose that BFS is run on  $G$  from a given source vertex  $s \in V$ . Then, during its execution, BFS discovers every vertex  $v \in V$  that is reachable from the source  $s$ , and upon termination,  $v.d = \delta(s, v)$  for all  $v \in V$ . Moreover, for any vertex  $v \neq s$  that is reachable from  $s$ , one of the shortest paths from  $s$  to  $v$  is a shortest path from  $s$  to  $v.\pi$  followed by the edge  $(v.\pi, v)$ .

→ Based on the resulting tree

The PRINT-PATH procedure prints out the vertices on a shortest path from  $s$  to  $v$ , assuming that BFS has already computed a breadth-first tree. This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.

PRINT-PATH( $G, s, v$ )

```
1 if  $v == s$  → source vertex
2   print  $s$ 
3 else if  $v.\pi == \text{NIL}$ 
4   print "no path from"  $s$  "to"  $v$  "exists"
5 else PRINT-PATH( $G, s, v.\pi$ )
6   print  $v$ 
```

→ It was not covered by the BFS  
so there is no connection

→ Call in a recursive manner → to keep printing the value.

Breadth-First Search

As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex  $v$  that still has unexplored edges leaving it. Once all of  $v$ 's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which  $v$  was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search

Find the nodes reachable from  $S$

→ This approach when the graph is a maze of interconnected rooms and you are walking around it

→ Start from  $S$  and select the first edge. Now, do the same with the first edge from the second node until reaching the end node (with no more children)

→ Explore the graph by going as deep as possible and go back when necessary

DFS( $u$ ):

```
Mark  $u$  as "Explored" and add  $u$  to  $R$ 
For each edge  $(u, v)$  incident to  $u$ 
    If  $v$  is not marked "Explored" then
        Recursively invoke DFS( $v$ )
    Endif
Endfor
```

→ Both DFS & BFS build 4 connected components  $\in S$

→ Similar level of effort

→ Visit nodes in a diff order

→ It also returns a tree as an output of the component  $R$

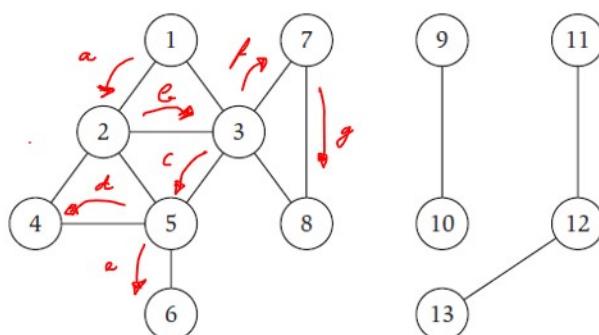
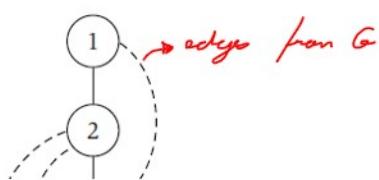
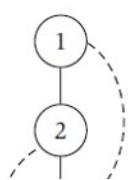
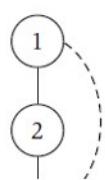
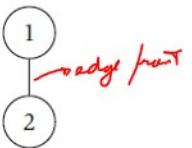


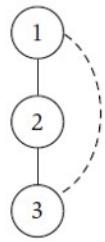
Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

Construction of a DFS tree rooted at node 1

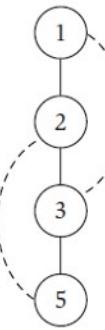




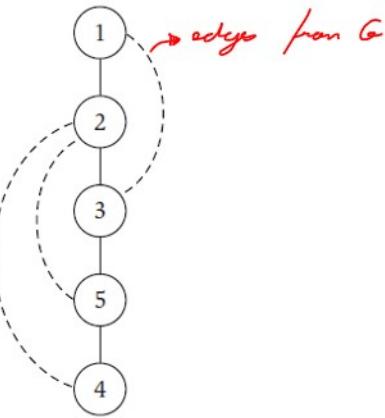
(a)



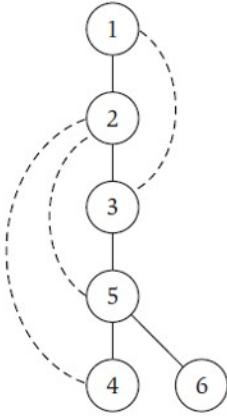
(b)



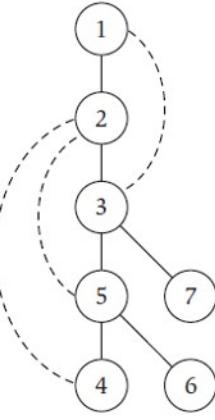
(c)



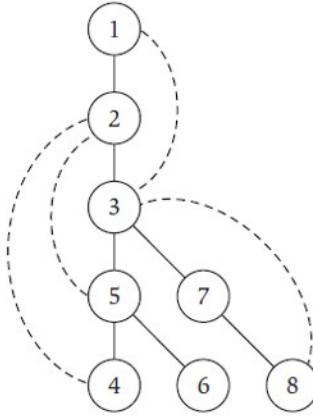
(d)



(e)



(f)



(g)

**Figure 3.5** The construction of a depth-first search tree  $T$  for the graph in Figure 3.2, with (a) through (g) depicting the nodes as they are discovered in sequence. The solid edges are the edges of  $T$ ; the dotted edges are edges of  $G$  that do not belong to  $T$ .

Figure 3.5 depicts the construction of a DFS tree rooted at node 1 for the graph in Figure 3.2. The solid edges are the edges of  $T$ ; the dotted edges are edges of  $G$  that do not belong to  $T$ . The execution of DFS begins by building a path on nodes 1, 2, 3, 5, 4. The execution reaches a dead end at 4, since there are no new nodes to find, and so it “backs up” to 5, finds node 6, backs up again to 3, and finds nodes 7 and 8. At this point there are no new nodes to find in the connected component, so all the pending recursive DFS calls terminate, one by one, and the execution comes to an end. The full DFS tree is depicted in Figure 3.5(g).

Like breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is *discovered* in the search, and is blackened when it is *finished*, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

→ saves a time stamp v-d

\* second timestamp v-f

*timestamps* → provides important information about the structure of the graph → also the behavior of the search integers between  $[1 \dots 2V]$  ( $v.d < v.f$ )

DFS( $G$ )

```

1 for each vertex  $u \in G.V$ 
2    $u.\text{color} = \text{WHITE}$ 
3    $u.\pi = \text{NIL}$ 
4    $\text{time} = 0$ 
5 for each vertex  $u \in G.V$ 
6   if  $u.\text{color} == \text{WHITE}$ 
7     DFS-VISIT( $G, u$ )

```

→ Marks each vertex as not explored yet

→ Resets the global time counter

→ Chooses each vertex and calls the DFS-visits with vertex  $v$  as the root of a new tree in the forest → explores all edges from this vertex, returning times

DFS-VISIT( $G, u$ )

```

1    $\text{time} = \text{time} + 1$            ↗ unless the time each time this function is called
2    $u.d = \text{time}$              // white vertex  $u$  has just been discovered
3    $u.\text{color} = \text{GRAY}$        } → Discovers time of vertex  $v$ 
4   for each vertex  $v$  in  $G.\text{Adj}[u]$  // explore each edge  $(u, v)$  → explores the adj[u] list
5     if  $v.\text{color} == \text{WHITE}$       } → Explores all the edges from that vertex
6        $v.\pi = u$ 
7       DFS-VISIT( $G, v$ )
8    $\text{time} = \text{time} + 1$            ↗ after every edge explored, now the finish time + 1
9    $u.f = \text{time}$                can paint it black saying it's done until the
10   $u.\text{color} = \text{BLACK}$           // blacken  $u$ ; it is finished

```

↗ unless the time each time this function is called

// white vertex  $u$  has just been discovered

} → Discovers time of vertex  $v$

// explores the adj[u] list

} → Explores all the edges from that vertex

↗ after every edge explored, now the finish time + 1

can paint it black saying it's done until the

vertices

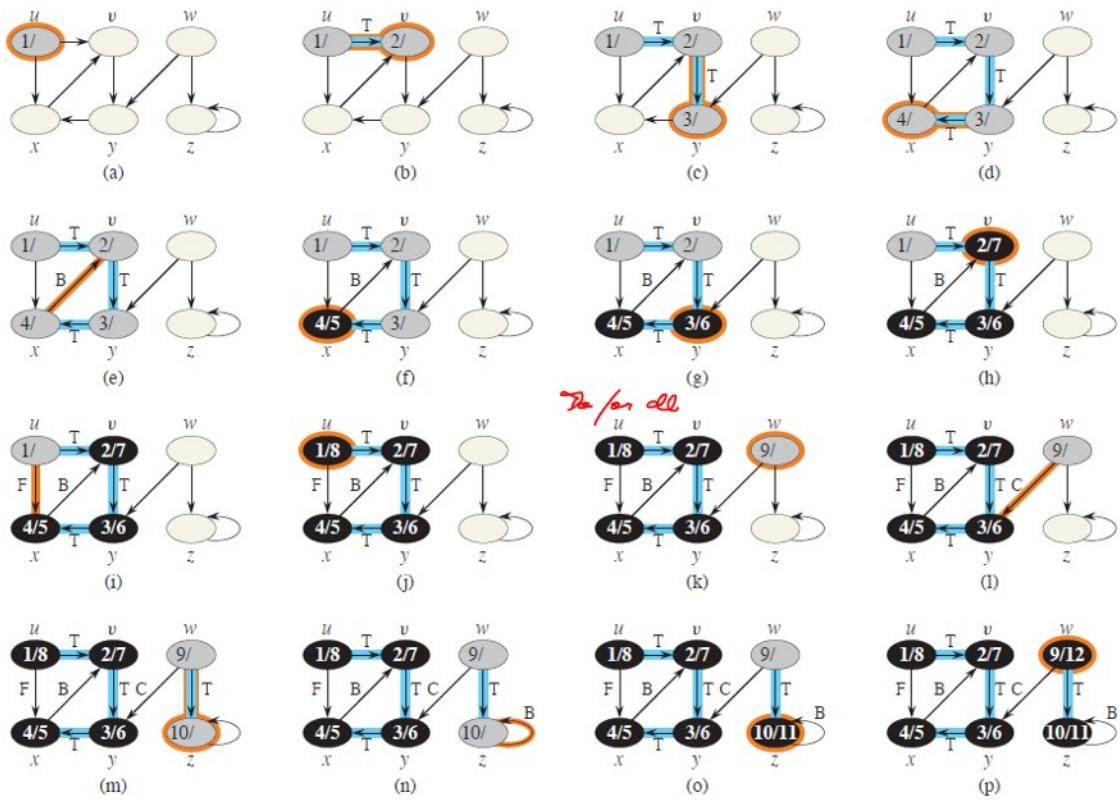
→ But causes  
no problems  
because any  
result can be used

The results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex. These different visitation orders tend not to cause

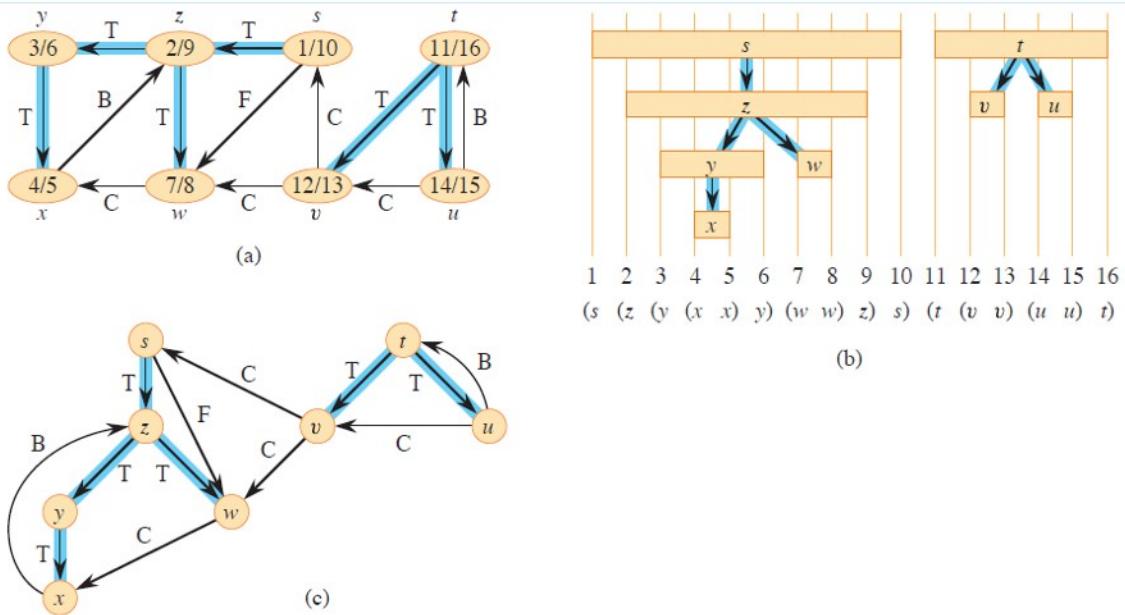
result from any depth-first search.

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take  $\Theta(V)$  time, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex  $v \in V$ , since the vertex  $u$  on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex  $u$  gray. During an execution of DFS-VISIT( $G, v$ ), the loop in lines 4–7 executes  $|Adj[v]|$  times. Since  $\sum_{v \in V} |Adj[v]| = \Theta(E)$  and DFS-VISIT is called once per vertex, the

total cost of executing lines 4–7 of DFS-VISIT is  $\Theta(V + E)$ . The running time of DFS is therefore  $\Theta(V + E)$ .



**Figure 20.4** The progress of the depth-first-search algorithm DFS on a directed graph. Edges are classified as they are explored: tree edges are labeled T, back edges B, forward edges F, and cross edges C. Timestamps within vertices indicate discovery time/finish times. Tree edges are highlighted in blue. Orange highlights indicate vertices whose discovery or finish times change and edges that are explored in each step.



**Figure 20.5** Properties of depth-first search. (a) The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 20.4. (b) Intervals for the discovery time and finish time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finish times of the corresponding vertex. Only tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. (c) The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

## Classification of edges

You can obtain important information about a graph by classifying its edges during a depth-first search. For example, Section 20.4 will show that a directed graph is acyclic if and only if a depth-first search yields no “back” edges (Lemma 20.11).

The depth-first forest  $G_\pi$  produced by a depth-first search on graph  $G$  can contain four types of edges:

1. **Tree edges** are edges in the depth-first forest  $G_\pi$ . Edge  $(u, v)$  is a tree edge if  $v$  was first discovered by exploring edge  $(u, v)$ .
2. **Back edges** are those edges  $(u, v)$  connecting a vertex  $u$  to an ancestor  $v$  in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. **Forward edges** are those nontree edges  $(u, v)$  connecting a vertex  $u$  to a proper descendant  $v$  in a depth-first tree.
4. **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

## → 3.3 Implementing Graph Traversal Using Queues and Stacks

$$\hookrightarrow \left\{ \begin{array}{l} \text{BFS} \rightarrow \text{Queue} \\ \text{DFS} \rightarrow \text{Stack} \end{array} \right\}$$

→ Represent graphs by adjacent MATRIX or LIST

$$G = (V, E) \rightarrow \left\{ \begin{array}{l} V \rightarrow \text{Number of nodes } (n) \\ E \rightarrow \text{Number of edges } (m) \end{array} \right.$$

$\left\{ \begin{array}{l} \cdot \text{Connected graphs} \rightarrow \text{at least } (m > n - 1) \\ (\text{edges} > \text{nodes} - 1) \end{array} \right.$  linear time  
→ Run running time based on both this parameters  $O(m+n)$

for Connected graphs  $\rightarrow O(n) \rightarrow (m > n - 1)$  Discard the less relevant

$$G = (V, E) \quad V = \{1, 2, 3, \dots, n\}$$

→ Simplest way to represent is a matrix

matrix, which is an  $n \times n$  matrix  $A$  where  $A[u, v]$  is equal to 1 if the graph contains the edge  $(u, v)$  and 0 otherwise. If the graph is undirected, the matrix  $A$  is symmetric, with  $A[u, v] = A[v, u]$  for all nodes  $u, v \in V$ . The adjacency matrix representation allows us to check in  $O(1)$  time if a given edge  $(u, v)$  is present in the graph. However, the representation has two basic disadvantages.

- The representation takes  $\Theta(n^2)$  space. When the graph has many fewer edges than  $n^2$ , more compact representations are possible.
- Many graph algorithms need to examine all edges incident to a given node  $v$ . In the adjacency matrix representation, doing this involves considering all other nodes  $w$ , and checking the matrix entry  $A[v, w]$  to see whether the edge  $(v, w)$  is present—and this takes  $\Theta(n)$  time. In the worst case,  $v$  may have  $\Theta(n)$  incident edges, in which case checking all these edges will take  $\Theta(n)$  time regardless of the representation. But many graphs in practice have significantly fewer edges incident to most nodes, and so it would be good to be able to find all these incident edges more efficiently.



### ADJACENT LIST

- There is a record for each node  $(v)$
- Contains a list of nodes which  $(v)$  has edges

array  $adj \rightarrow adj[v] \rightsquigarrow$  Contains a list of nodes adjacent to node  $(v)$

Undirected graph  $\rightarrow$  each edge  $e = (v, w) \in E$  occurs on two adjacent lists
 

- node  $w$  appears on the list for  $v$
- node  $v$  appears on the list for  $w$

**(3.10)** The adjacency matrix representation of a graph requires  $O(n^2)$  space, while the adjacency list representation requires only  $O(m + n)$  space.

In view of this, the adjacency list is a natural representation for exploring graphs. If the algorithm is currently looking at a node  $u$ , it can read this list of neighbors in constant time per neighbor; move to a neighbor  $v$  once it encounters it on this list in constant time; and then be ready to read the list associated with node  $v$ . The list representation thus corresponds to a physical notion of “exploring” the graph, in which you learn the neighbors of a node  $u$  once you arrive at  $u$ , and can read them off in constant time per neighbor.

Two of the simplest and most natural options are to maintain a set of elements as either a queue or a stack. A *queue* is a set from which we extract elements in *first-in, first-out* (FIFO) order: we select elements in the same order in which they were added. A *stack* is a set from which we extract elements in *last-in, first-out* (LIFO) order: each time we select an element, we choose the one that was added most recently. Both queues and stacks can be easily implemented via a doubly linked list. In both cases, we always select the first element on our list; the difference is in where we insert a new element. In a queue a new element is added to the end of the list as the last element, while in a stack a new element is placed in the first position on the list. Recall that a doubly linked list has explicit *First* and *Last* pointers to the beginning and end, respectively, so each of these insertions can be done in constant time.

## Implementing Breadth-First Search

→ Exam edges leaving from a given node ( $\rightarrow$  by  $\perp$ )

(1) →  $v$  → Leaving  $v$  and come to edge ( $v \rightarrow v'$ ) → To  $v'$  already discovered? → Maintain a discovered array for fast

(2) → To make this simple, we maintain an array Discovered of length  $n$  and set  $\text{Discovered}[v] = \text{true}$  as soon as our search first sees  $v$ . The algorithm, as

(2) described in the previous section, constructs layers of nodes  $L_1, L_2, \dots$ , where  $L_i$  is the set of nodes at distance  $i$  from the source  $s$ . To maintain the nodes in a layer  $L_i$ , we have a list  $L[i]$  for each  $i = 0, 1, 2, \dots$

→ Queue or Stack

Keep a control of which nodes where on which layers

BFS( $s$ ):

```

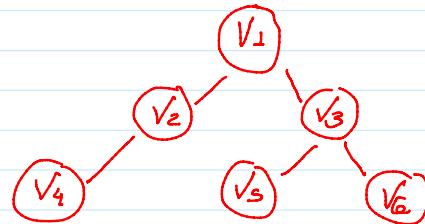
Set Discovered[ $s$ ] = true and Discovered[ $v$ ] = false for all other  $v$  → Not discovered
Initialize  $L[0]$  to consist of the single element  $s$  → Queue
Set the layer counter  $i = 0$ 
Set the current BFS tree  $T = \emptyset$ 
While  $L[i]$  is not empty
    Initialize an empty list  $L[i+1]$  → new element list
    For each node  $u \in L[i]$ 
        Consider each edge  $(u, v)$  incident to  $u$ 
        If Discovered[ $v$ ] = false then
            Set Discovered[ $v$ ] = true
            Add edge  $(u, v)$  to the tree  $T$ 
            Add  $v$  to the list  $L[i+1]$ 
        Endif
    Endfor
    Increment the layer counter  $i$  by one
Endwhile

```

→ Represent the graph as an adjacency list

{ → Array of lists with size  $\frac{V}{\text{number of nodes}}$

V <sub>1</sub>	V <sub>2</sub> V <sub>3</sub>
V <sub>2</sub>	V <sub>1</sub> V <sub>4</sub>
V <sub>3</sub>	V <sub>1</sub> V <sub>5</sub> V <sub>6</sub>
V <sub>4</sub>	V <sub>2</sub>
V <sub>5</sub>	V <sub>3</sub>
V <sub>6</sub>	V <sub>3</sub>



↪ Each vertex as a class, + let can store data + methods

```
private:  
    vector< pair<shared_ptr<Node>, vector<int>> > graphDefinition;
```

V <sub>1</sub>	V <sub>2</sub> V <sub>3</sub>
V <sub>2</sub>	V <sub>1</sub> V <sub>4</sub>
V <sub>3</sub>	V <sub>1</sub> V <sub>5</sub> V <sub>6</sub>
V <sub>4</sub>	V <sub>2</sub>
V <sub>5</sub>	V <sub>3</sub>
V <sub>6</sub>	V <sub>3</sub>

↪ shared pointer to the node class.

↪ It has an identifier that we need to loop in order to find the right node

↪ Vector contains stores just the identifiers of the nodes

```
for (  
    vector< pair<shared_ptr<Node>, vector<int>> >::iterator it=graphDefinition.begin();  
    it < graphDefinition.end(); it++)  
{  
    if(it->first->getId() == fromNode){  
        it->first->includeEdge(toNode);  
        it->second.push_back(toNode);  
        break;  
    }  
}
```

↪ I can define + graph → now need to start the BFS

```

76 vector<shared_ptr<Node>> GraphsClass::BFS_Search(){
77     vector<shared_ptr<Node>> BFS_result;
78     queue<shared_ptr<Node>> BFS_queue; → main difference
79     vector< pair<shared_ptr<Node>, vector<int>> > graphToApplyBFS = graphDefinition;
80
81     if(graphDefinition.empty()){
82         return BFS_result;
83     }
84
85     initializeSourceNodeBFS(graphToApplyBFS, BFS_queue);
86
87     while (!BFS_queue.empty())
88     {
89         shared_ptr<Node> nodeToEvaluate = BFS_queue.front();
90         BFS_queue.pop();
91
92         cout<< "[GraphsClass::BFS_Search] nodeToEvaluate ID: " << nodeToEvaluate->getId() << endl;
93
94         vector<int> childNodes = nodeToEvaluate->getEdges();
95
96         //for(vector<int>::iterator it=childNodes.begin(); it != childNodes.end(); ++it){
97         for(int it : childNodes){
98             cout<< "[GraphsClass::BFS_Search] Node ID: " << it << endl;
99             shared_ptr<Node> node = retrieveNodePointerFromGraph(it, graphToApplyBFS);
100            if(node && node->color == color::white){
101                node->color = color::gray;
102                node->distance = nodeToEvaluate->distance + 1;
103                node->parent = nodeToEvaluate;
104                BFS_queue.push(node); → populates the containers based on the child!
105            }
106        }
107        nodeToEvaluate->color = color::black;
108        BFS_result.push_back(nodeToEvaluate);
109    }
110
111    return BFS_result;
112 }
113
114 vector<shared_ptr<Node>> GraphsClass::DFS_Search(){
115
116     vector<shared_ptr<Node>> BFS_result;
117     stack<shared_ptr<Node>> BFS_stack; [ ]
118     vector< pair<shared_ptr<Node>, vector<int>> > graphToApplyBFS = graphDefinition;
119
120     if(graphDefinition.empty()){
121         return BFS_result;
122     }
123
124     initializeSourceNodeDFS(graphToApplyBFS, BFS_stack);
125
126     while (!BFS_stack.empty())
127     {
128         shared_ptr<Node> nodeToEvaluate = BFS_stack.top();
129         BFS_stack.pop();
130
131         cout<< "[GraphsClass::BFS_Search] nodeToEvaluate ID: " << nodeToEvaluate->getId() << endl;
132
133         vector<int> childNodes = nodeToEvaluate->getEdges();
134
135         //for(vector<int>::iterator it=childNodes.begin(); it != childNodes.end(); ++it){
136         for(int it : childNodes){
137             cout<< "[GraphsClass::BFS_Search] Node ID: " << it << endl;
138             shared_ptr<Node> node = retrieveNodePointerFromGraph(it, graphToApplyBFS);
139             if(node && node->color == color::white){
140                 node->color = color::gray;
141                 node->distance = nodeToEvaluate->distance + 1;
142                 node->parent = nodeToEvaluate;
143                 BFS_stack.push(node);
144             }
145         }
146     }
147     nodeToEvaluate->color = color::black;
148     BFS_result.push_back(nodeToEvaluate);
149 }
150
151     return BFS_result;
152 }
153

```

- Paths

(3.11) The above implementation of the BFS algorithm runs in time  $O(m + n)$  (i.e., linear in the input size), if the graph is given by the adjacency list representation.

**(3.13)** The above implementation of the DFS algorithm runs in time  $O(m + n)$  (i.e., linear in the input size), if the graph is given by the adjacency list representation.

{ → DFS  
→ Shortest