

Graph

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

→ Graph Algorithms

Searching 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 objects

↳ Collection of V nodes and E edges, which joins the nodes

$$v, v \in V$$

$$e \in E$$

$$e = \{v, v\}$$

↳ an EDGES connects two other nodes! node is vertex

↳ Indicates a symmetric relationship between nodes

• DIRECTED Graph (G^1)

↳ When we want to encode asymmetric relations

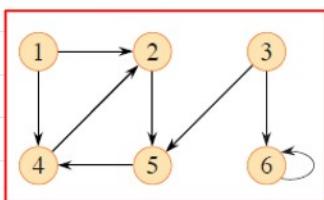
↳ Consist in a set of nodes (V) and a set of directed edges (E^1) based on an ordered pair (v, v)

$\begin{cases} v \rightarrow \text{tail} \\ v \rightarrow \text{head} \end{cases}$

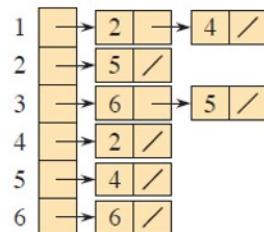
• edge e^1 leaves node v and enter node v !

↳ has a direction going from one node to the next

Direction does matter



(a)



(b)

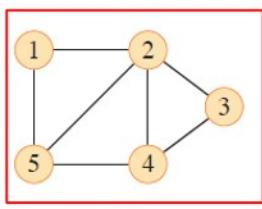
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

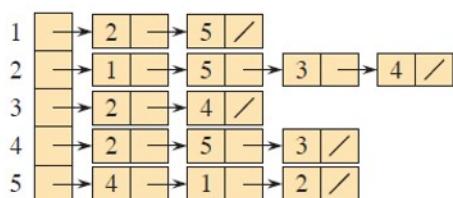
Figure 20.2 Two representations of a directed graph. (a) A directed graph G with 6 vertices and 8 edges. (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G .

• 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)

↳ operation on graphs

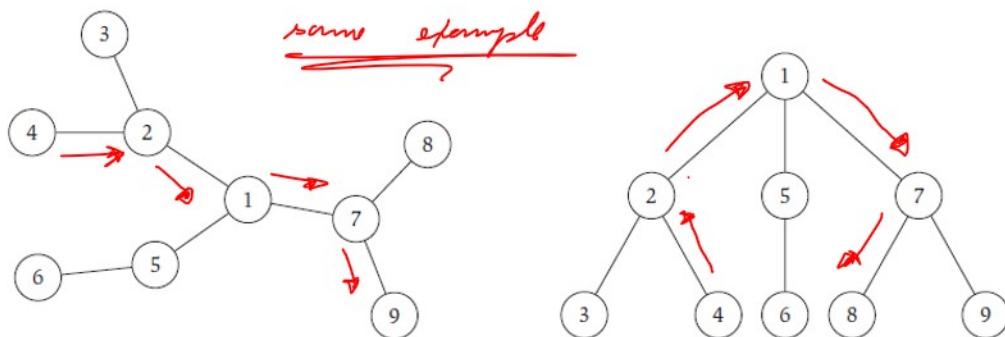


Figure 3.1 Two drawings of the same tree. On the right, the tree is rooted at node 1.

• 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 sequence

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

↳ Strongly connected unless the
↳ 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

$G = (V, E) \rightarrow$ Collection of adjacent lists or adj matrices

↳ Apply to direct and indirect graphs

prefer when the graph is dense

$G = (V, E)$

- $E \ll V^2 \rightarrow$ Sparse Graphs → Adjacent - List
- $E \approx V^2 \rightarrow$ Dense Graphs → Adjacent - Matrix

• Adjacent - list representation

$G = (V, E) \rightarrow$ array with adj $|V|$ list → One for each vertex

↳ To adj list contains all the vertices \vee 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 an edge (v, w) is present in the graph ↳ Then to search (w) in the $Adj[v]$ list

• you may run up a loop edge (v, v) present in the graph → then to remove (v) in the $\text{Adj}[v]$ list

• Adjacency - 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 $\Theta(|V|^2)$
→ Independent of the edges

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

↳ 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.

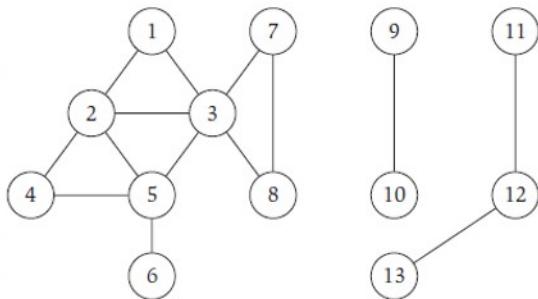
Implementing vertex and edge attributes in real programs can be another story entirely. There is no one best way to store and access vertex and edge attributes. For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how 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 $\text{Adj}[u]$, 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 reached (by edges)

→ Start in node S and include all nodes connected to S by one edge → then, include all additional nodes connected to the nodes from the first layer → Generating the second layer



$\bullet S = \text{node } 1$ Layer 0 (L_0)

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 TRS) (Gr)

→ Discovering vertices in waves / layers

→ Uses a queue (FIFO) containing 48 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

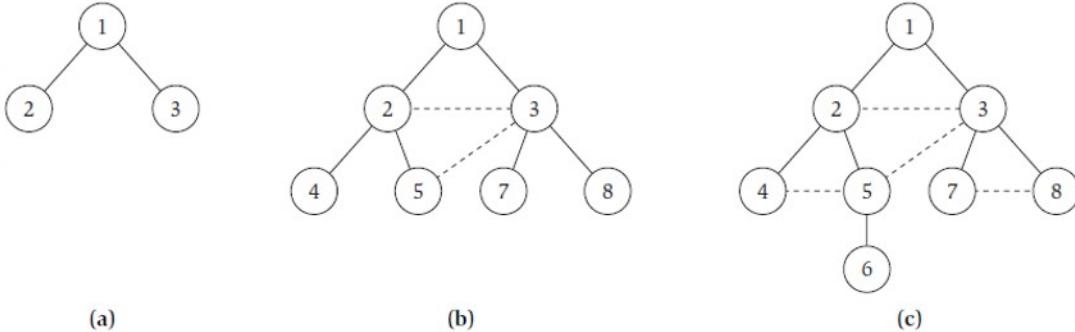


Figure 3.3 The construction of a breadth-first search tree T for the graph in Figure 3.2, with (a), (b), and (c) depicting the successive layers that are added. The solid edges are the edges of T ; the dotted edges are in the connected component of G containing node 1, but do not belong to T .

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 by at most 1.

Each vertex is discovered once (can be referenced later on) but it has one 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  $u \in G.V - \{s\}$ 
2    $u.\text{color} = \text{WHITE}$ 
3    $u.d = \infty$ 
4    $u.\pi = \text{NIL}$ 
```

```
5  $s.\text{color} = \text{GRAY}$ 
6  $s.d = 0$ 
7  $s.\pi = \text{NIL}$ 
8  $Q = \emptyset$ 
9 ENQUEUE( $Q, s$ )
```

```
10 while  $Q \neq \emptyset$ 
11    $u = \text{DEQUEUE}(Q)$  → consider an only-list
12   for each vertex  $v$  in  $G.\text{Adj}[u]$  // search the neighbors of  $u$ 
13     if  $v.\text{color} == \text{WHITE}$  // is  $v$  being discovered now?
14       discover  $v$  such vertex  $\left\{ \begin{array}{l} v.\text{color} = \text{GRAY} \\ v.d = u.d + 1 \\ v.\pi = u \end{array} \right.$  not yet discovered
15       connected to  $(u)$  ENQUEUE( $Q, v$ ) //  $v$  is now on the frontier
16      $u.\text{color} = \text{BLACK}$  //  $u$  is now behind the frontier
```

→ Enqueuing vertices → all vertices with
inf distance from the source (s) & nil parent

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

→ Creates the queue with just the source vertex

→ Generate all the
remaining vertices
discovering vertices
not yet examined

↳ Grid → for a square $n \times n$ steps always at most $\frac{n(n+1)}{2}$ nodes

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

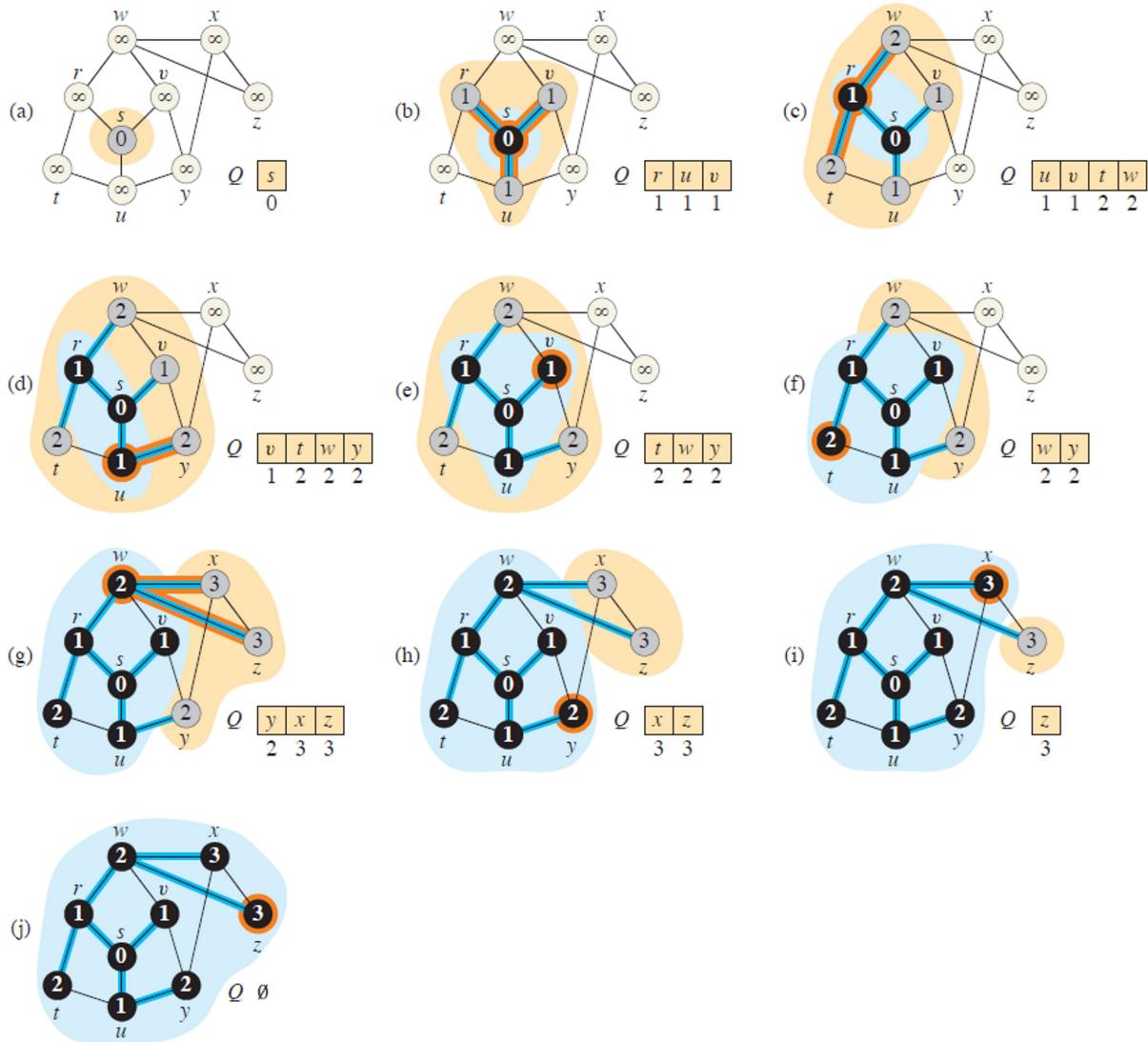


Figure 20.3 The operation of BFS on an undirected graph. Each part shows the graph and the queue Q at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear within each vertex and below vertices in the queue. The tan region surrounds the frontier of the search, 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($\underline{\underline{G}}, \underline{\underline{s}}, \underline{\underline{v}}$)

```
1 if  $v == s$  → source vertex
2   print  $s$ 
3 elseif  $v.\pi == \text{NIL}$  } → It was not covered by the BFS
4   print "no path from"  $s$  "to"  $v$  "exists" } so there is no connection
5 else PRINT-PATH( $\underline{\underline{G}}, \underline{\underline{s}}, \underline{\underline{v.\pi}}$ ) } → Call in a recursive manner → to keep
6   print  $v$  printing the value.
```