Graph

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

→ Graphs 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 objectives

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

$u, v \in V$ $e \in E$ $> e = \{u, v\}$

↳ an EDGES connects two other nodes!

node ≈ vertex

↳ Indicates a symmetric relationship between nodes

• DIRECTED GRAPH (G^1)

↳ When we want to encode asymmetric relations

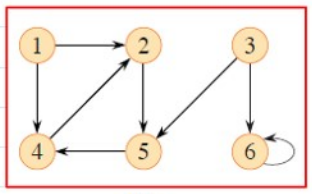
↳ Consist in a set of nodes (V) and a set of directed edges (E^1) based on an ordered pair (u, v)

$u \rightarrow$ tail
$v \rightarrow$ head

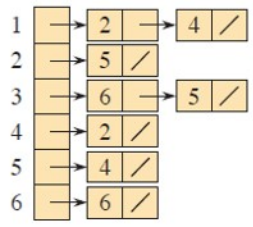
• edge e^1 leaves node u and enter node v!

↳ 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 → B) (B → A)

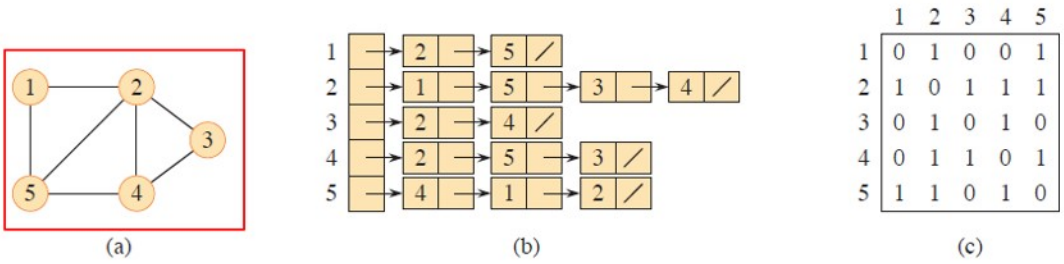


Figure 20.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Note that the edges can flow both ways

- PATH
  - Are operation on graphs
  - Used to represent a traversing sequence of edges (find path)

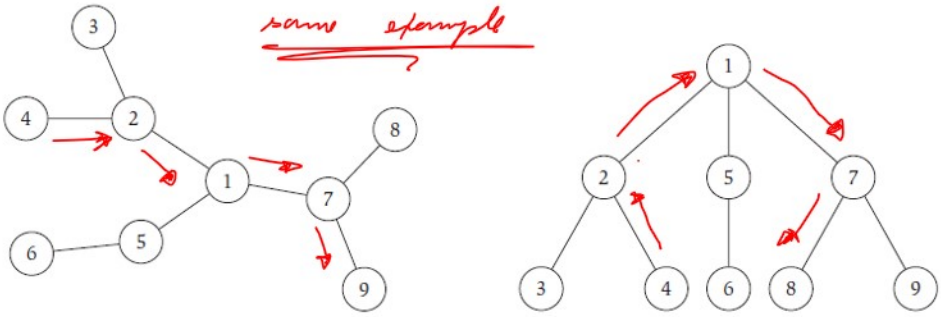


Figure 3.1 Two drawings of the same tree. On the right, the tree is rooted at node 1.

- SIMPLE PATH → if all vertices are distinct from one another
- Cycle → Path that comes back to the start point

a path $v_1, v_2, \ldots, v_{k-1}, v_k$ 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 ($4 \to 5$, $5 \to 4$)
- 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 adj matrix

↳ Apply to direct and indirect graphs

↳ pref. when the graph is dense

$G = (V, E)$

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

• Adjacent-list representation

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

↳ 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*** $w : E \to \mathbb{R}$. 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 guarantee way to determine if a given edge (u, v) is present in the graph ↳ Have to search (v) in the $Adj[u]$ list

• Adjacent-Matrix representation

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

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

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

↳ Independent of the edge

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

↳ Disventage 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

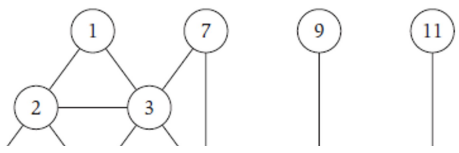
Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. We indicate these attributes using our usual notation, such as $v.d$ for an attribute d of a vertex v. When we indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have 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

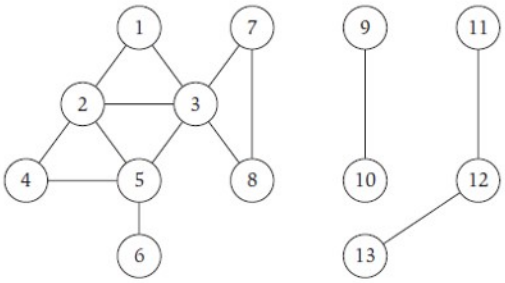
Explore outward from S in all possible directions, adding nodes one layer at a time, but can be revisited (by edges)

↳ 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 = node 1 Layer 0 (L_0)

Layer 1 : 2 3 Neighbours of S



- S = node 1 Layer 0 (L_0)

Layer 1 : 2 3 Neighbors of S

Layer 2 : 4 5 7 8

Layer 3 : 6

→ Discovering vertices in waves / layers
↳ Uses a queue (FIFO) containing the vertices
↳ Also need to flag the vertex to see:
- if it's reachable or not
- when the V was found

↳ Defines the frontier of the discovered vertices

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 $L_1, L_2, L_3, \ldots$ 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 Trooted tree at S, showing the nodes reachable from S. (BFS search TREE) (G_π)

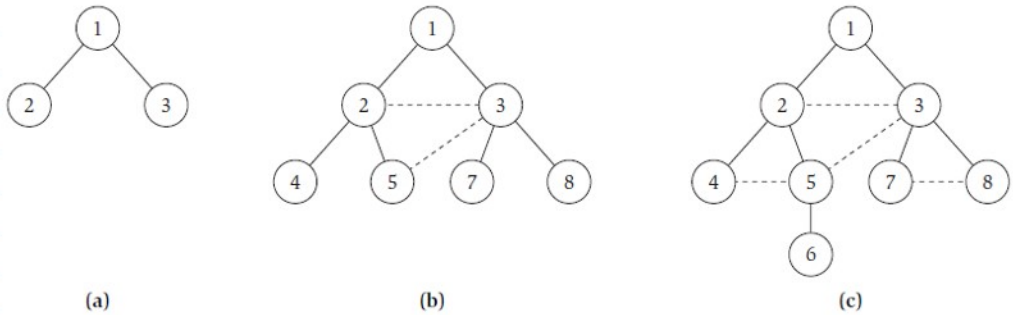


Figure 3.3 The construction of a breadth-first search tree T for the graph in Figure 3.2, with (a), (b), and (c) depicting the successive layers that are added. The solid edges are the edges of T; the dotted edges are in the connected component of G containing node 1, but do not belong to T.

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.

↳ 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*

(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]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```

Initializing vertices → all colors until 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

Iterates all the remaining vertices discovering vertices not yet examined

consider adj-list

consider each vertex connected to (u)

not yet discovered

Guiné → faz a enqueue e depois loope até não ter mais nada

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

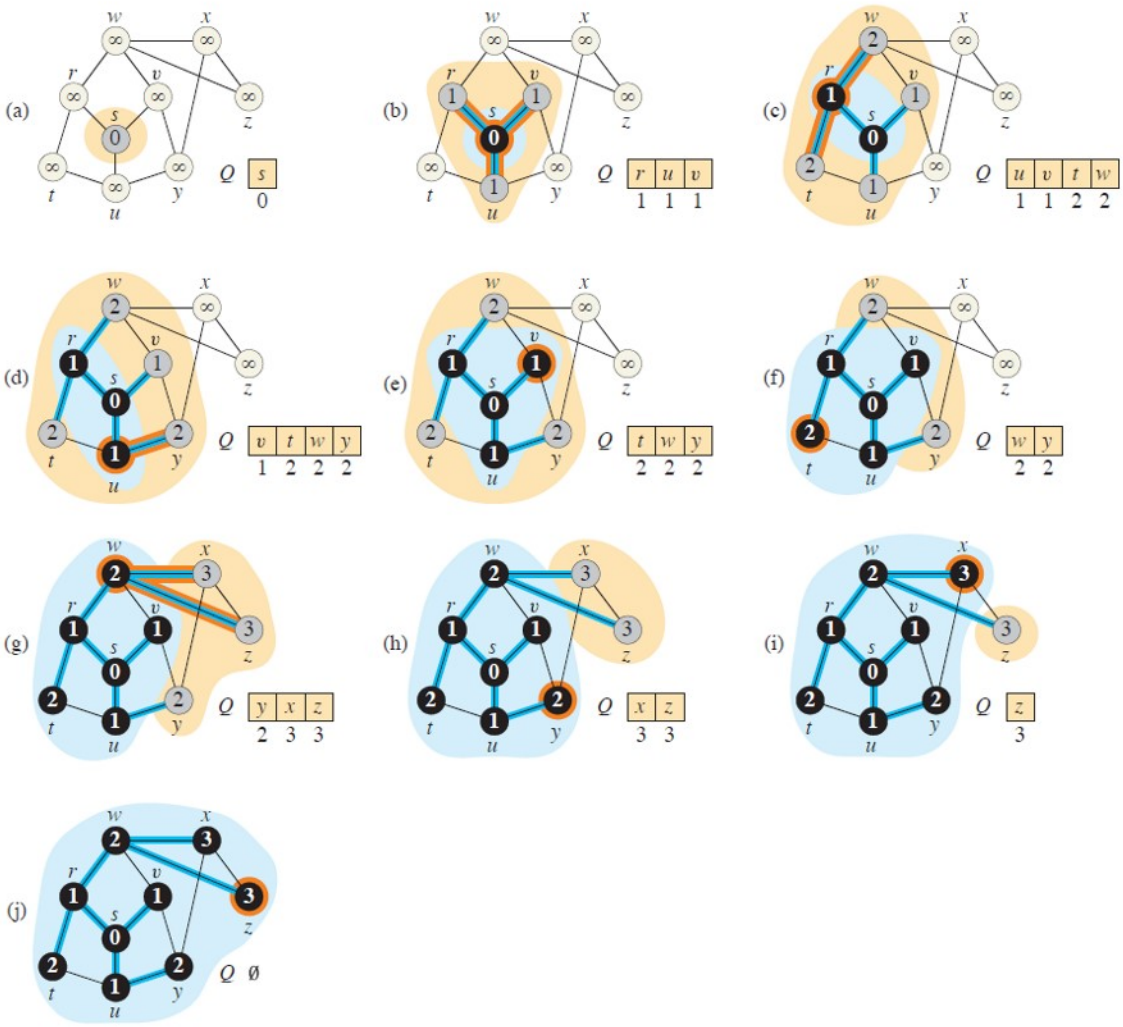


Figure 20.3 The operation of BFS on an undirected graph. Each part shows the graph and the queue Q at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear within each vertex and below vertices in the queue. The tan region surrounds the frontier of the search,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

↳ 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 $\langle v_1, v_2, \ldots, v_r \rangle$, 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 $i = 1, 2, \ldots, r - 1$.

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  elseif v.π == 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(G, s, v.π)         → Call in a recursive manner → to keep
6      print v                          printing the value.
```

DEPTH-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

→ Take 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 the connected component ∈ S

↳ Similar level of eff

↳ Visit nodes in a diff order

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

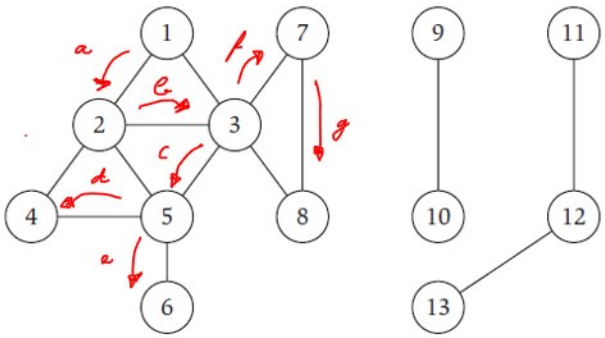
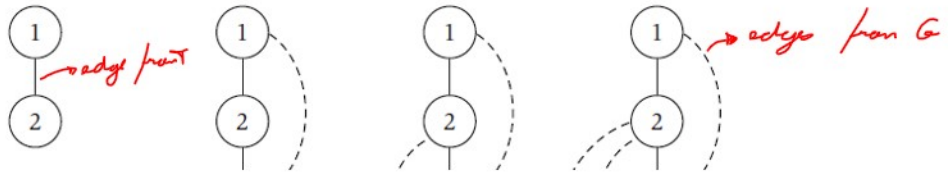


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

Construction of a DFS tree (T) rooted at node 1



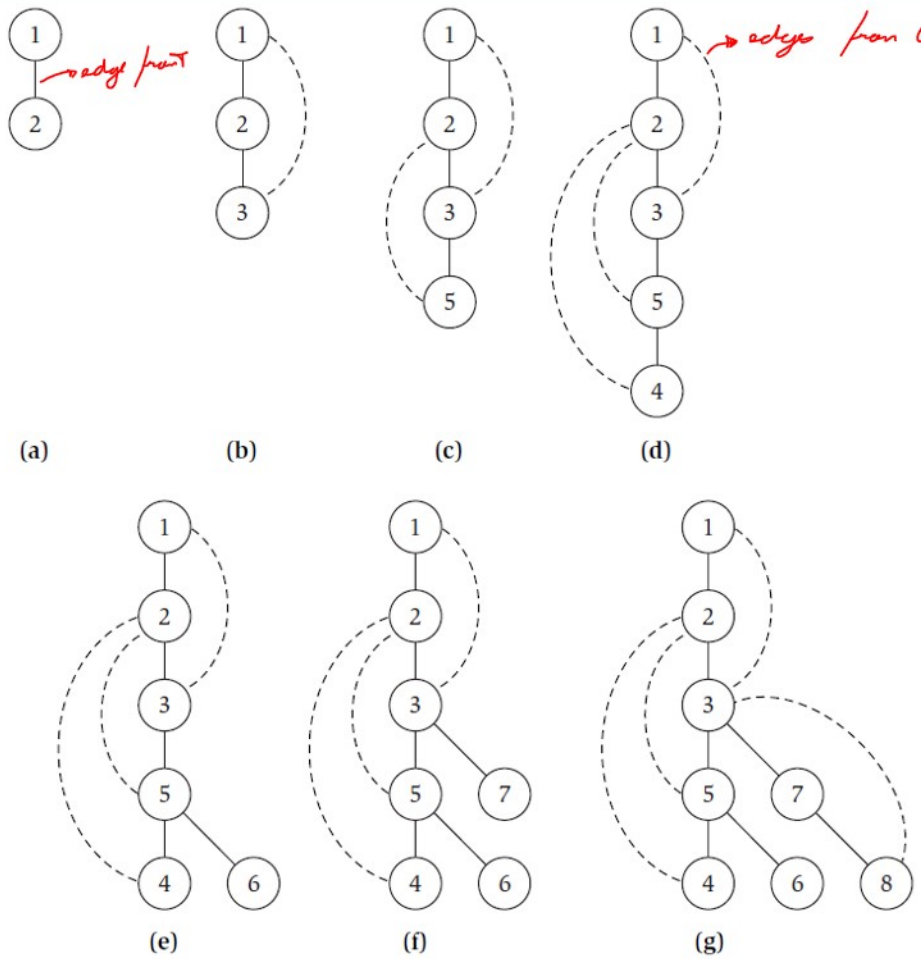


Figure 3.5 The construction of a depth-first search tree T for the graph in Figure 3.2, with (a) through (g) depicting the nodes as they are discovered in sequence. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T.

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.

timestamps → Provide important information about the structure of the graph → show the behavior of the search
integers between [1 & 2V] (u.d < u.f)

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

→ Paints each vertex as not explored yet
→ Resets the global time counter
→ Check each vertex and call the DFS-visit with vertex u as the root of a new tree in the forest → Explore all edges from this vertex passing time

```
DFS-VISIT(G, u)
1  time = time + 1          // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each vertex v in G.Adj[u]   // explore each edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  time = time + 1
9  u.f = time
10 u.color = BLACK          // blacken u; it is finished
```

Saves the time each time this function is called
→ Discovered time of vertex u
→ explores the adj[] list
→ Explore all the edges from that vertex
→ After every edge explored, save the finish time + 1 on u and it blacks saying it's done with the vertex

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

→ But cause no problems because any results can be used

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 $\Theta(V + E)$.

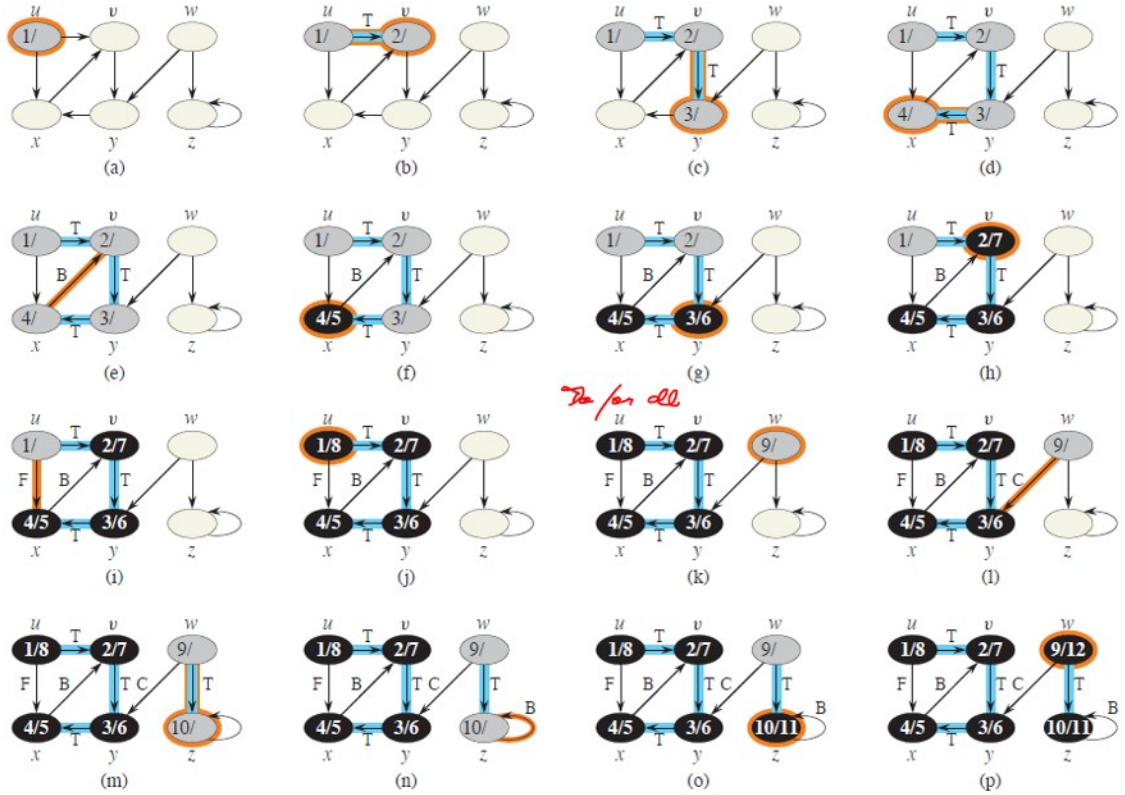


Figure 20.4 The progress of the depth-first-search algorithm DFS on a directed graph. Edges are classified as they are explored: tree edges are labeled T, back edges B, forward edges F, and cross edges C. Timestamps within vertices indicate discovery time/finish times. Tree edges are highlighted in blue.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

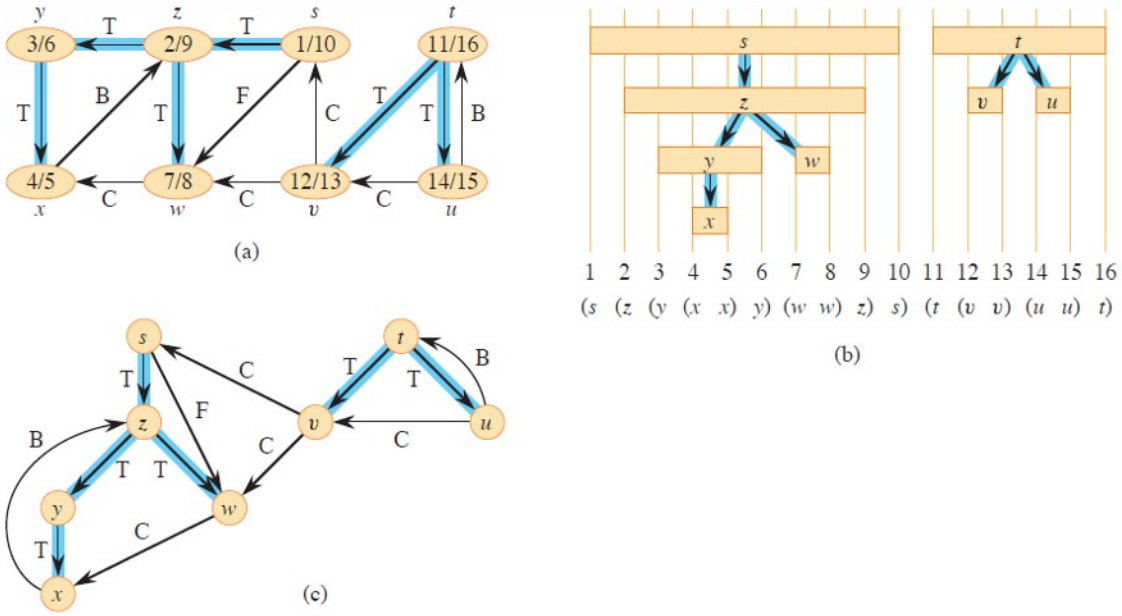


Figure 20.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 20.4. **(b)** Intervals for the discovery time and finish time of each vertex correspond to the parenthesization shown. Each 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_π 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_π. 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

↳ BFS → Queue
DFS → Stack

→ Represent graphs by adjacent MATRIX or LIST

$G = (V, E)$ → V → Number of nodes (n)
E → Number of edges (m)

• Connected graph → at least $(m \geq n - 1)$
(edges ≥ nodes − 1)

→ Run running time based on both this parameters $O(m+n)$ linear time

for Connected graph → $O(m)$ → $(m \geq n-1)$ Discard the less relevant

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

→ 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 ⟶ adj[v] ⟶ Contains a list of nodes adjacents to node (v)

Undirected graph ⟶ 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

→ Come edges leaving from a given node (↓ by ↓)

(u) → Leaving u and come to edge (u,v) → (v)

→ Is v already discovered ?

↳ Maintain a discovered array for that

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

(2) described in the previous section, constructs layers of nodes $L_1, L_2, \ldots$, 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 $i = 0, 1, 2, \ldots$.

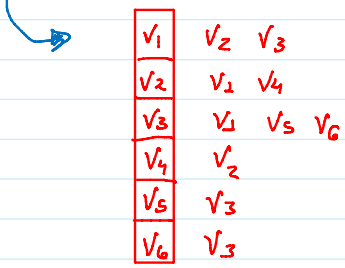
↳ Queue or Stack

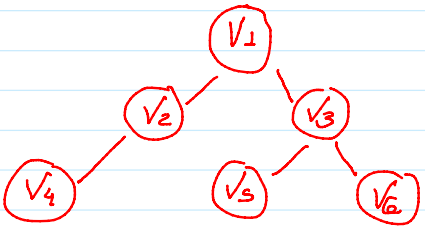
↳ Keep a control of which nodes where on each layer

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v → Not is discovered
  Initialize L[0] to consist of the single element s → Queue
  Set the layer counter i = 0
  Set the current BFS tree T = ∅
  While L[i] is not empty
    Initialize an empty list L[i+1] → New element list
    For each node u ∈ 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 adj list

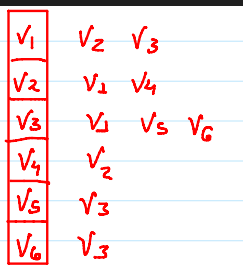
↳ Array of lists with size V
↳ number of nodes





↳ Each vertex as a class, that can store data + methods

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



↳ shared pointer to the node class.

↳ vector<int> which stores just the identifiers of the node

↳ to be able to identify that we need to loop in order to find the right node

```
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 the graph → now need to start the BFS

```cpp
vector<shared_ptr<Node>> GraphsClass::BFS_Search(){

    vector<shared_ptr<Node>> BFS_result;
    queue<shared_ptr<Node>> BFS_queue;
    vector< pair<shared_ptr<Node>, vector<int>> > graphToApplyBFS = graphDefinition;

    if(graphDefinition.empty()){
        return BFS_result;
    }

    initializeSourceNodeBFS(graphToApplyBFS, BFS_queue);

    while (!BFS_queue.empty())
    {
        shared_ptr<Node> nodeToEvaluate = BFS_queue.front();
        BFS_queue.pop();

        cout<< "[GraphsClass::BFS_Search] nodeToEvaluate ID: " << nodeToEvaluate->getId() << endl;

        vector<int> childNodes = nodeToEvaluate->getEdges();

        //for(vector<int>::iterator it=childNodes.begin(); it != childNodes.end(); ++it){
        for(int it : childNodes){
            cout<< "[GraphsClass::BFS_Search] Node ID: " << it << endl;
            shared_ptr<Node> node = retrieveNodePonterFromGraph(it, graphToApplyBFS);
            if(node && node->color == color::white){
                node->color = color::gray;
                node->distance = nodeToEvaluate->distance + 1;
                node->parent = nodeToEvaluate;
                BFS_queue.push(node);
            }
        }
        nodeToEvaluate->color = color::black;
        BFS_result.push_back(nodeToEvaluate);
    }

    return BFS_result;
}

vector<shared_ptr<Node>> GraphsClass::DFS_Search(){

    vector<shared_ptr<Node>> BFS_result;
    stack<shared_ptr<Node>> BFS_stack;
    vector< pair<shared_ptr<Node>, vector<int>> > graphToApplyBFS = graphDefinition;

    if(graphDefinition.empty()){
        return BFS_result;
    }

    initializeSourceNodeDFS(graphToApplyBFS, BFS_stack);

    while (!BFS_stack.empty())
    {
        shared_ptr<Node> nodeToEvaluate = BFS_stack.top();
        BFS_stack.pop();

        cout<< "[GraphsClass::BFS_Search] nodeToEvaluate ID: " << nodeToEvaluate->getId() << endl;

        vector<int> childNodes = nodeToEvaluate->getEdges();

        //for(vector<int>::iterator it=childNodes.begin(); it != childNodes.end(); ++it){
        for(int it : childNodes){
            cout<< "[GraphsClass::BFS_Search] Node ID: " << it << endl;
            shared_ptr<Node> node = retrieveNodePonterFromGraph(it, graphToApplyBFS);
            if(node && node->color == color::white){
                node->color = color::gray;
                node->distance = nodeToEvaluate->distance + 1;
                node->parent = nodeToEvaluate;
                BFS_stack.push(node);
            }
        }
        nodeToEvaluate->color = color::black;
        BFS_result.push_back(nodeToEvaluate);
    }

    return BFS_result;
}
```

main difference

populate the container based on the child!

- Path

(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.*

4.4 Shortest Paths in a Graph

↳ Some times it can covers a lot of possibilities that doesn't worth to use the brute force

↳ Input to a shortest-path problem → weighted directed graph

↳ $G = (V, E)$ $w : E \to \mathbb{R}$ → Map edges to real time values

↳ The weight of a path is $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v\,, \\ \infty & \text{otherwise}\,. \end{cases}$$

A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that you want to minimize.

Dijkstra's → Greedy algorithm

Floyd-Warshall → Dynamic-Programming algorithm

The Greedy Algorithm

↳ Identify one element (among a group) and reject all the rest then, select only the next and so on

↳ Big problem here is to define the simple rule to use for the selection

Key Principles of Greedy Algorithms

1. **Greedy Choice Property**: Greedy algorithms make decisions that seem the best at the moment without considering future consequences. The goal is to pick the "local optimum" at each step, hoping that this will lead to the "global optimum" for the problem.
2. **Optimal Substructure**: Problems suitable for greedy algorithms must have an optimal substructure, meaning that the solution to the overall problem can be constructed from solutions to smaller subproblems. This allows the greedy algorithm to work through the problem step-by-step.
3. **Irrevocable Choices**: Once a choice is made, it cannot be undone. Greedy algorithms do not backtrack, making them efficient but sometimes limiting their effectiveness for certain problem types.

Example :

5. **Dijkstra's Algorithm for Shortest Path**
   - Goal: Find the shortest path from a source node to all other nodes in a graph.
   - Greedy Choice: Select the node with the minimum known distance, update its neighbors' distances, and repeat until all nodes are visited.

Distances over different nodes to find a path

Limitations of Greedy Algorithms

Greedy algorithms are not always optimal, particularly if:

- **Global Optimum Isn't Achieved**: Greedy choices might yield suboptimal results for problems where future choices are affected by earlier decisions.
- **Complex Constraints**: For problems with interdependent choices (e.g., traveling salesperson problem), greedy algorithms often fail to find the best solution.

Summary

Greedy algorithms can be a powerful and efficient solution to certain optimization problems, but they require a specific problem structure to guarantee an optimal solution.

Representing shortest paths

It is usually not enough to compute only shortest-path weights. Most applications of shortest paths need to know the vertices on shortest paths as well. For example, if your GPS told you the distance to your destination but not how to get there, it would not be terribly useful. We represent shortest paths similarly to how we represented breadth-first trees in Section 20.2. Given a graph $G = (V, E)$, maintain for each vertex $v \in V$ a ***predecessor*** $v.\pi$ that is either another vertex or NIL. The shortest-paths algorithms in this chapter set the π attributes so that the chain of predecessors originating at a vertex v runs backward along a shortest path from s to v. Thus, given a vertex v for which $v.\pi \neq \text{NIL}$, the procedure PRINT-PATH(G, s, v) from Section 20.2 prints a shortest path from s to v.

Both known the path to the predecessor until finding

file shortest path

defined. A ***shortest-paths tree*** rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

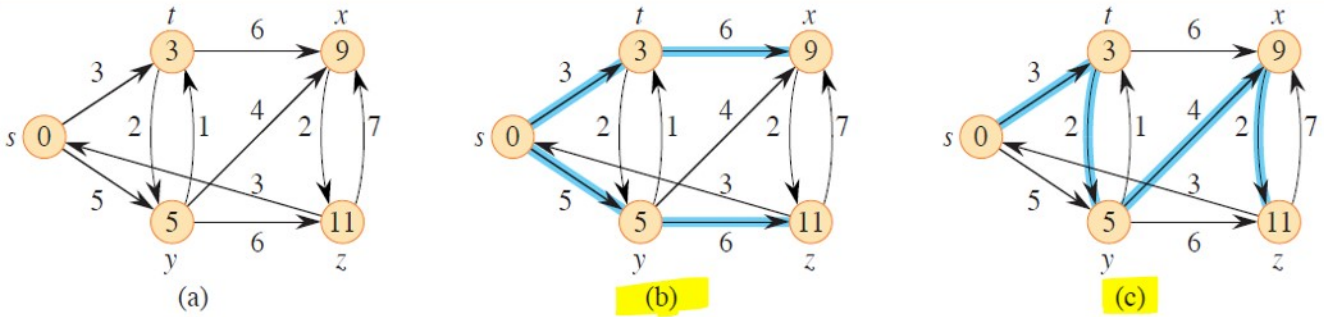


Figure 22.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The blue edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

Shortest paths are not necessarily unique, and neither are shortest-paths trees. For example, Figure 22.2 shows a weighted, directed graph and two shortest-paths trees with the same root.

Relaxation

The algorithms in this chapter use the technique of ***relaxation***. For each vertex $v \in V$, the single-source shortest paths algorithms maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v. We call $v.d$ a ***shortest-path estimate***. To initialize the shortest-path estimates and predecessors, call the $\Theta(V)$-time procedure INITIALIZE-SINGLE-SOURCE. After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$ and $v.d = \infty$ for $v \in V - \{s\}$.

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

The process of *relaxing* an edge (u, v) consists of testing whether going through vertex u improves the shortest path to vertex v found so far and, if so, updating $v.d$ and $v.\pi$. A relaxation step might decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute $v.\pi$. The RELAX procedure on the following page performs a relaxation step on edge (u, v) in $O(1)$ time. Figure 22.3 shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

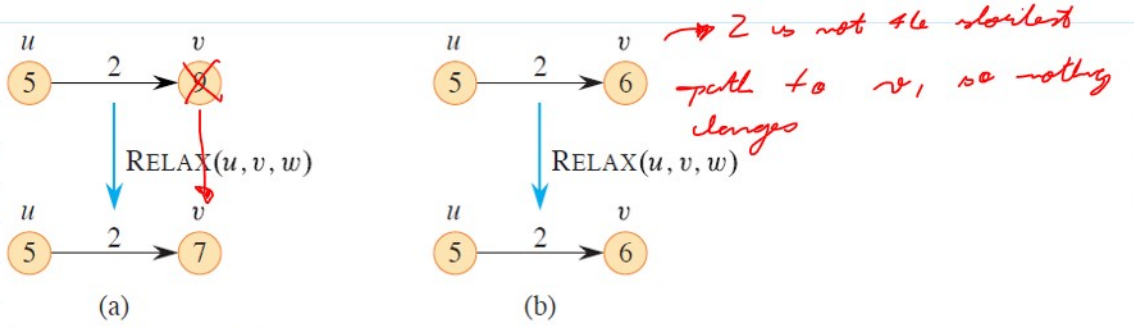


Figure 22.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Since we have $v.d \leq u.d + w(u, v)$ before relaxing the edge, the relaxation step leaves $v.d$ unchanged.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Triangle inequality (Lemma 22.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 22.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 22.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 22.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 22.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 22.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

DIJKSTRA

→ Creates the tree of shortest path from the start path to all other vertice

↳ Proposed a greedy algorithm to solve the shortest path problem!

In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the single-source shortest-paths problem. We begin by describing an algorithm that just determines the *length* of the shortest path from s to each other node in the graph; it is then easy to produce the paths as well. The algorithm maintains a

↳ Solves this problem on weighted directed graphs but requires non-negative weights

the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford

↳ Generalized BFS for weighted graphs → Not BFS or DFS
↳ Moves in waves from a vertice ↳ Priority first !!
↳ Every time it arrives in a vertice, a new wave is created from there
↳ edge weights give the time for the waves

Simple queue won't work !!

↳ It maintains a set of a set S of vertices whose final shortest path weights is already defined

adds u into S, and relaxes all edges leaving u. The procedure DIJKSTRA replaces the first-in, first-out queue of breadth-first search by a min-priority queue Q of vertices, keyed by their d values.!!!

↳ Requires positive weights

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

→ Initializes d & π values for all $v \in V$

→ Initializes the set to empty set

→ Initializes the min priority queue to contain all the vertices in V

→ Extracts a vertex from queue and add to the set

↳ To each edge from u, evaluates using the rules, the weight to the next vertex of shortest path

↳ If smaller → update the min-priority queue (Q)

↳ Never inserts values into Q after line (5) → while loop exactly $|V|$ times

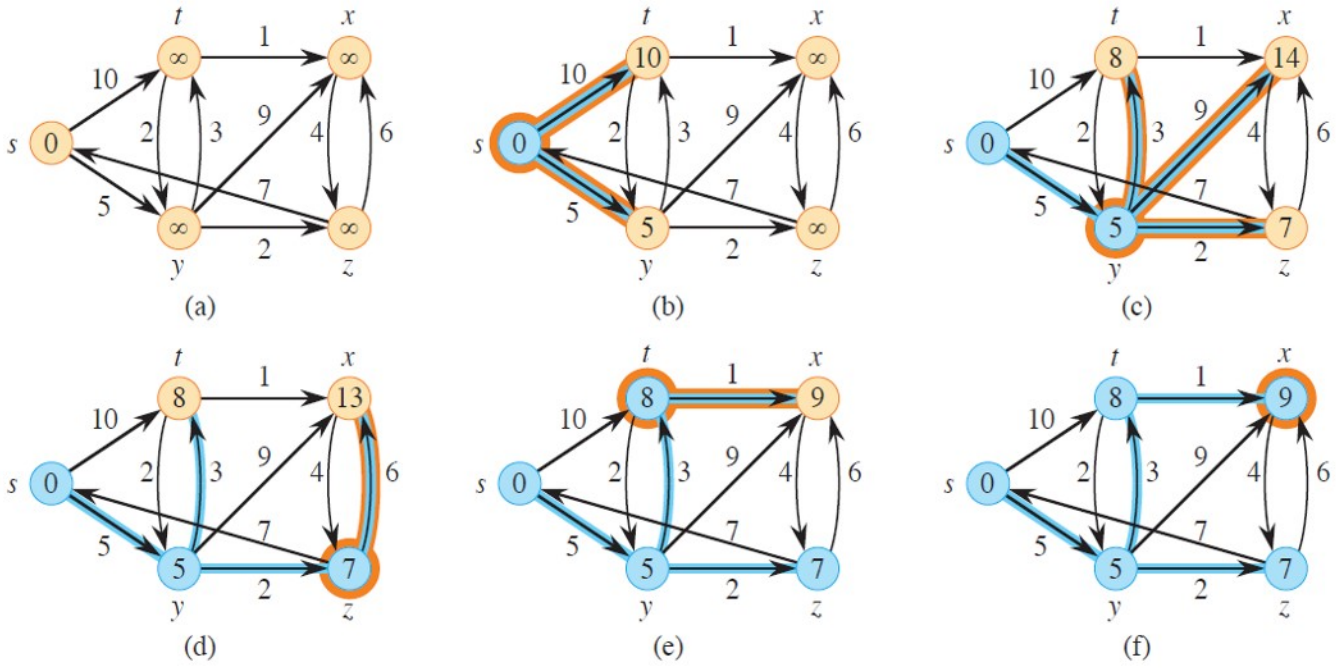


Figure 22.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the set S, and tan vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 6–12. **(b)–(f)** The situation after each successive iteration of the **while** loop. In each part, the vertex highlighted in orange was chosen as vertex u in line 7, and each edge highlighted in orange caused a d value and a predecessor to change when the edge was relaxed. The d values and predecessors shown in part (f) are the final values.

Theorem 22.6 (Correctness of Dijkstra's algorithm)
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source vertex s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Algorithm for Dijkstra's in C++

To know how Dijkstra's algorithm works behind the scene, look at the below steps to understand it in detail:

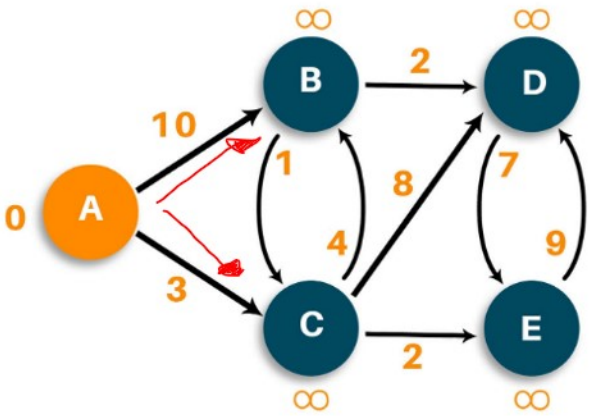
1. First of all, we will mark all vertex as unvisited vertex
2. Then, we will mark the source vertex as 0 and all other vertices as infinity
3. Consider source vertex as current vertex
4. Calculate the path length of all the neighboring vertex from the current vertex by adding the weight of the edge in the current vertex
5. Now, if the new path length is smaller than the previous path length then replace it otherwise ignore it
6. Mark the current vertex as visited after visiting the neighbor vertex of the current vertex
7. Select the vertex with the smallest path length as the new current vertex and go back to step 4.
8. Repeat this process until all the vertex are marked as visited.

Once we go through the algorithm, we can backtrack the source vertex and find our shortest path.

→ Generate the shortest path from the source to every other V

For now, the list of unvisited nodes will be: {B, C, D, E}

Fa✓tutor



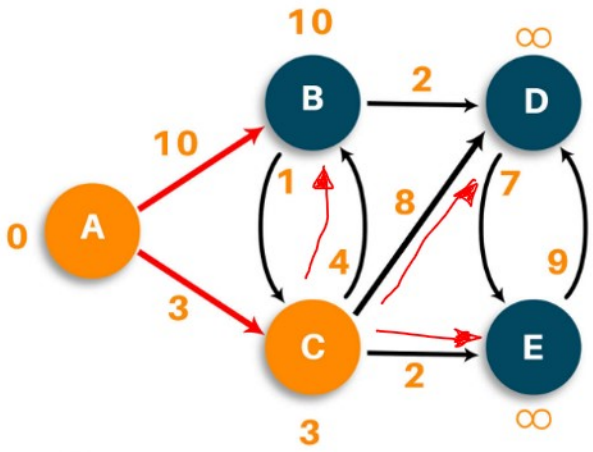
Q: | A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| | 10 | 3 | | |

S : {A} → Visited Path

} → Select the closest node to move forward

Now select the vertex with the smaller path length as visited vertex and put it in the answer. Therefore, the list of unvisited nodes is {B, D, E}

Fa✓tutor

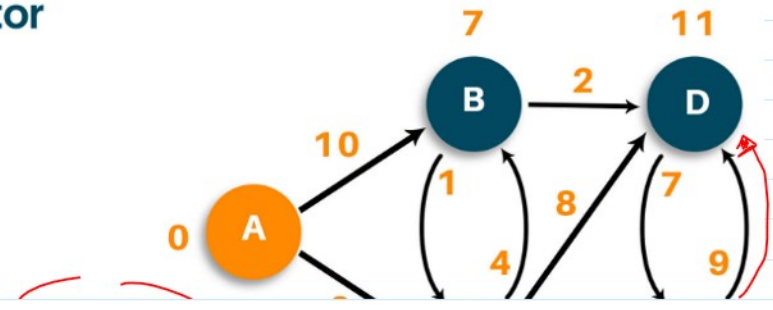


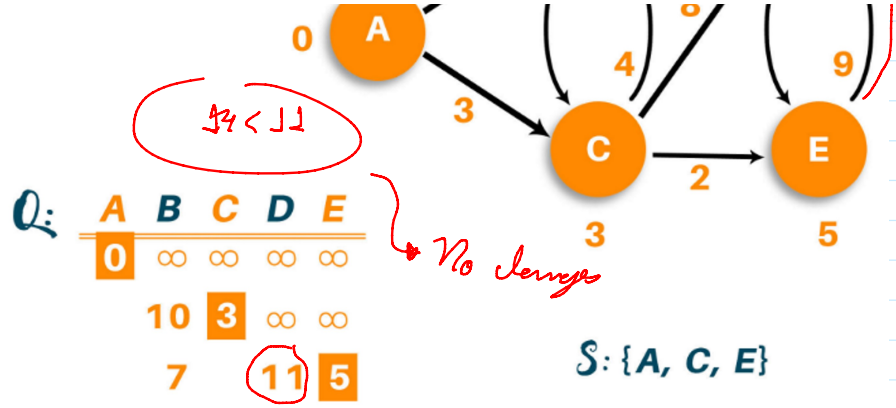
Q: | A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| | 10 | 3 | ∞ | ∞ |
| | 7 | | 11 | 5 |

S: {A, C}

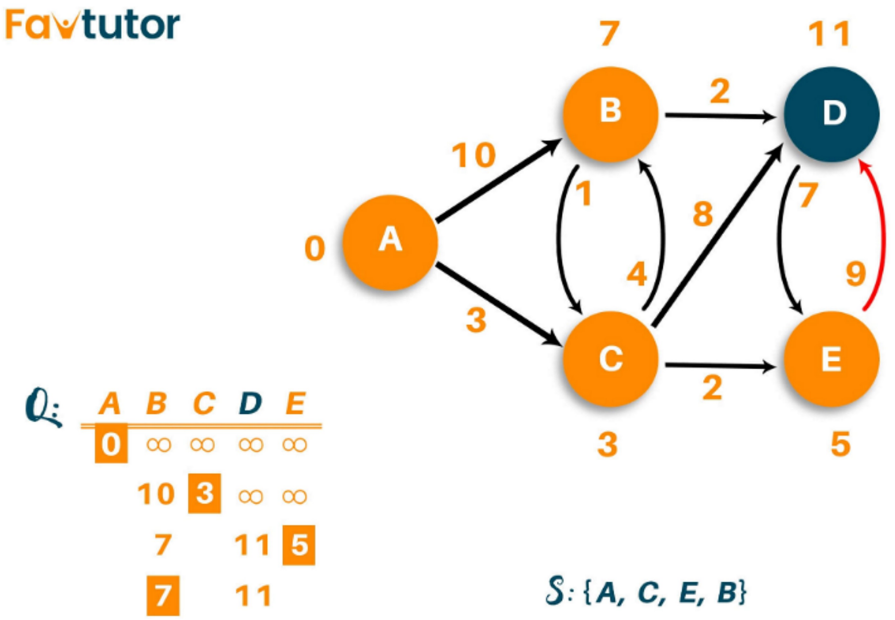
Now, choosing the shortest path from the above table results in choosing the node 'E' with the shortest distance of 5 from the source vertex. And hence the list of unvisited nodes is {B, D}

Fa✓tutor

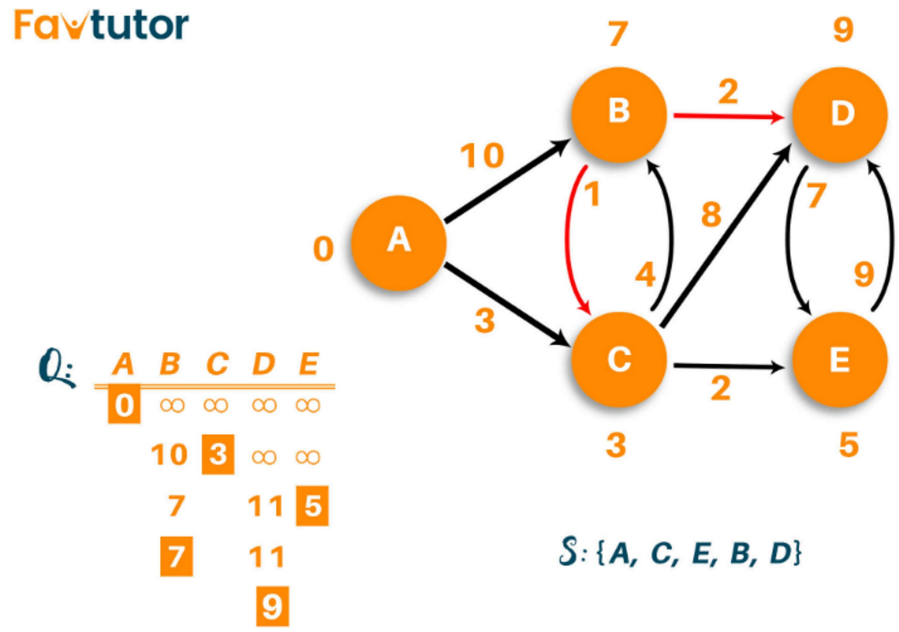




Therefore, the weight of vertex 'B' is minimum compared to vertex 'D,' so we will mark it as a visited node and add it to the path. The list of the unvisited nodes will be {D}



Therefore, the final output of the algorithm will be {A, C, E, B, D}



22.1 The Bellman-Ford algorithm

→ Similar to DIJKSTRA but allows negative values

↳ Can have negative weights

↳ Returns bool indicating whether there is a negative weight cycle that is reachable from the source

↳ if yes, it indicates that no solution exists

↳ if no, it produces the shortest weight and their weights

↳ if negative weight for a node → it fails the algorithm

The procedure BELLMAN-FORD relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

O(V) — weights to ∞ for all nodes besides source → Parents to null

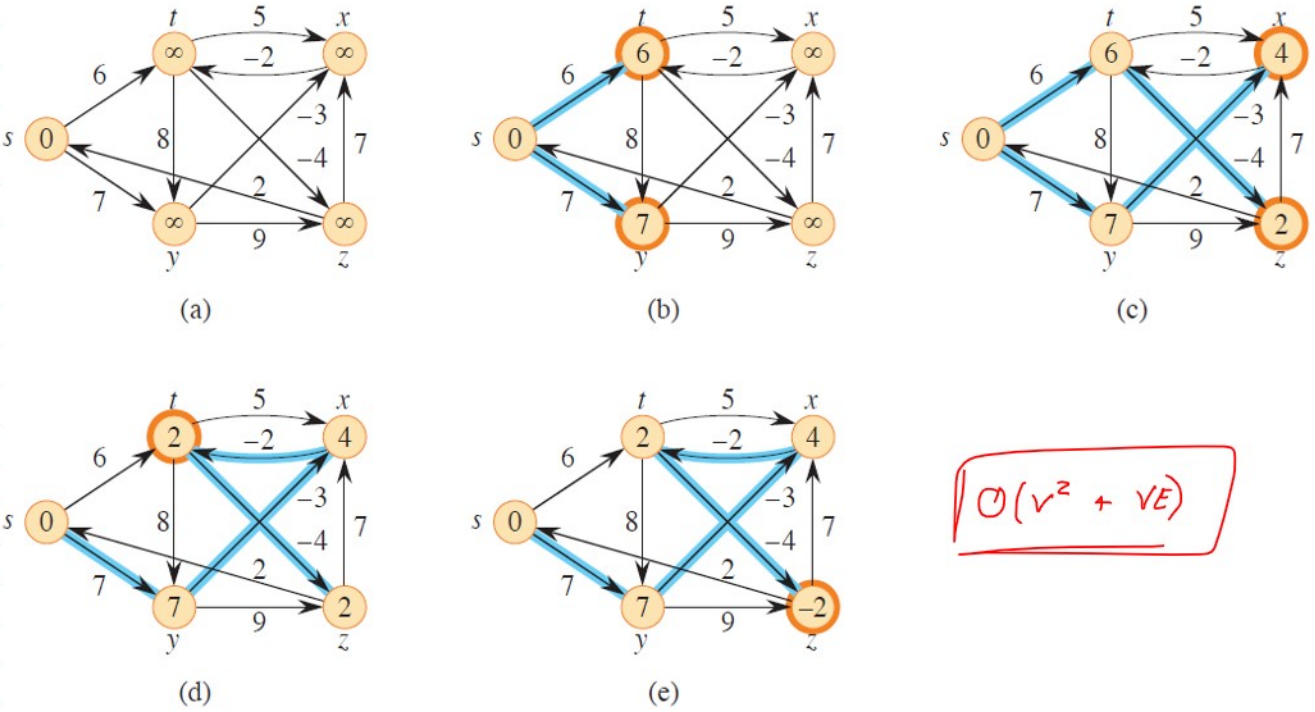
O(V·E) — Loop through all nodes

Loop through all adj edges

to cover the combinations of edges and calculate the weight

O(V+E) — Return the solution otherwise

No solution if weight from destination → from source + path weight

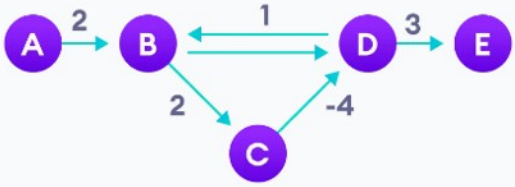


$O(V^2 + VE)$

Figure 22.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and blue edges indicate predecessor values: if edge (u, v) is blue, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. Vertices whose shortest-path estimates and predecessors have changed due to a pass are highlighted in orange. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Why do we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.



Negative weight cycles can give an incorrect result when trying to find out the shortest path

Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

How Bellman Ford's algorithm works

1 → Over estimate the length of the path from the start vertex to all the others

2 → Relax those estimations by finding new paths shorter than the previous

Bellman Ford vs Dijkstra

↳ (slower) !!

Bellman Ford's algorithm and Dijkstra's algorithm are very similar in structure. While Dijkstra looks only to the immediate neighbors of a vertex, Bellman goes through each edge in every iteration.

```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL

    distance[S] <- 0

    for each vertex V in G

        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V]
            Error: Negative Cycle Exists

    return distance[], previous[]
```

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q

    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    return distance[], previous[]
```

Bellman Ford's Algorithm vs Dijkstra's Algorithm

The idea behind Bellman Ford Algorithm:

*The Bellman-Ford algorithm's primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on "**Principle of Relaxation**".*

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

.....**a)** Do following for each edge u-v

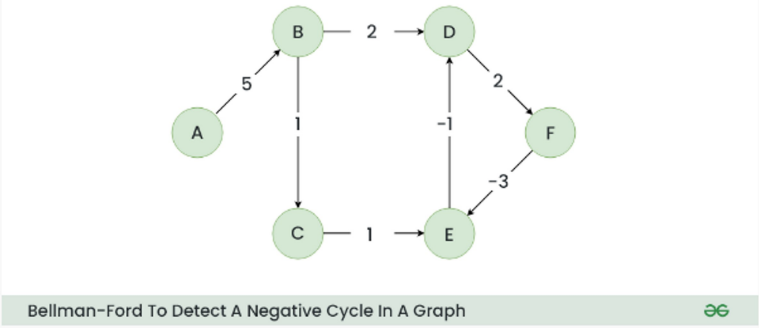
.....If dist[v] > dist[u] + weight of edge uv, then update dist[v]

.....dist[v] = dist[u] + weight of edge uv

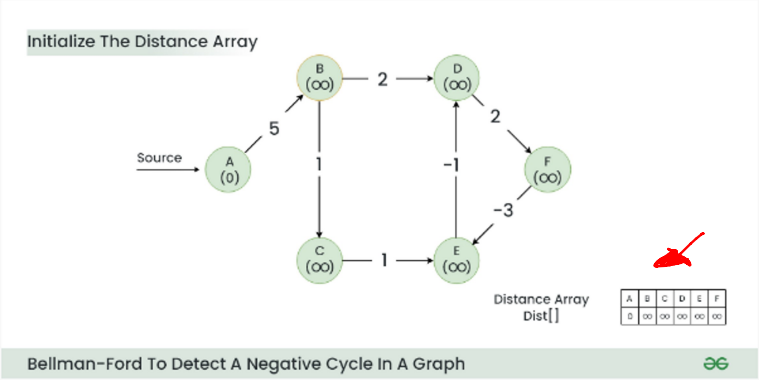
3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

.....If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

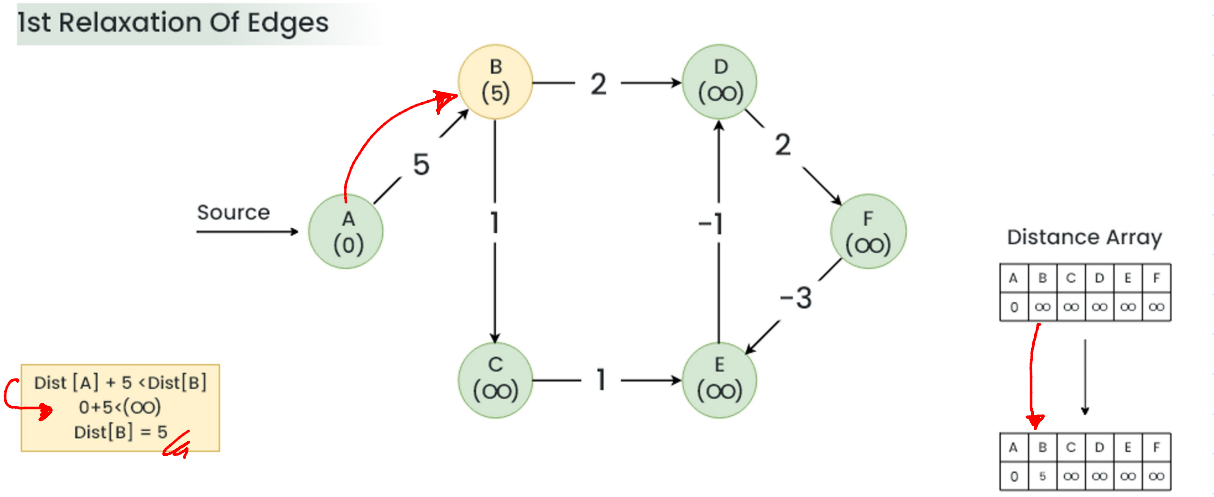
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle



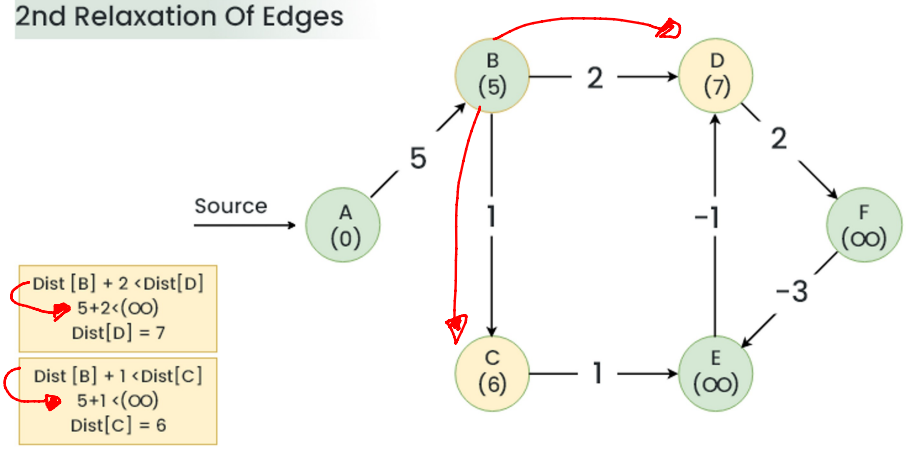
Step 1: *Initialize a distance array Dist[] to store the shortest distance for each vertex from the source vertex. Initially distance of source will be 0 and Distance of other vertices will be INFINITY.*



Initialize a distance array



2nd Relaxation Of Edges

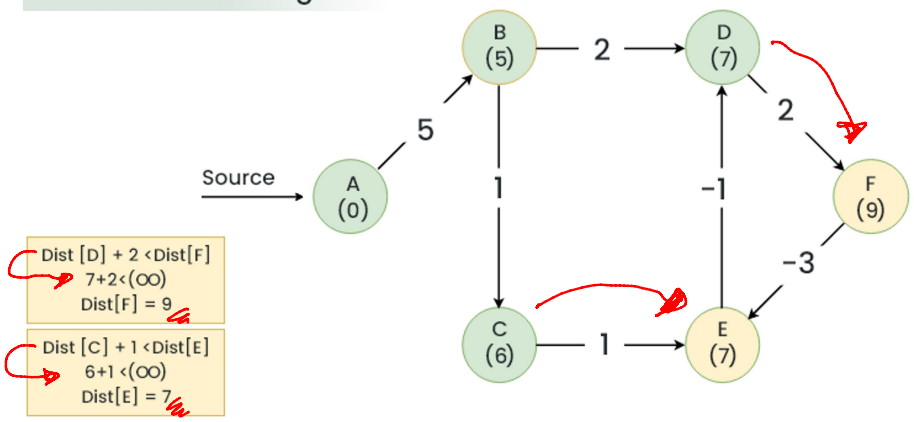


Distance Array

A	B	C	D	E	F
0	5	∞	∞	∞	∞

A	B	C	D	E	F
0	5	6	7	∞	∞

3rd Relaxation Of Edges

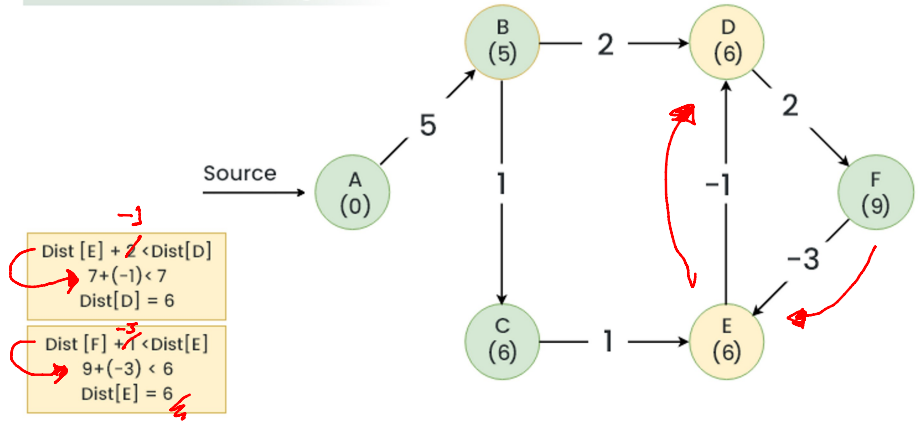


Distance Array

A	B	C	D	E	F
0	5	6	7	∞	∞

A	B	C	D	E	F
0	5	6	7	7	9

4th Relaxation Of Edges



Distance Array

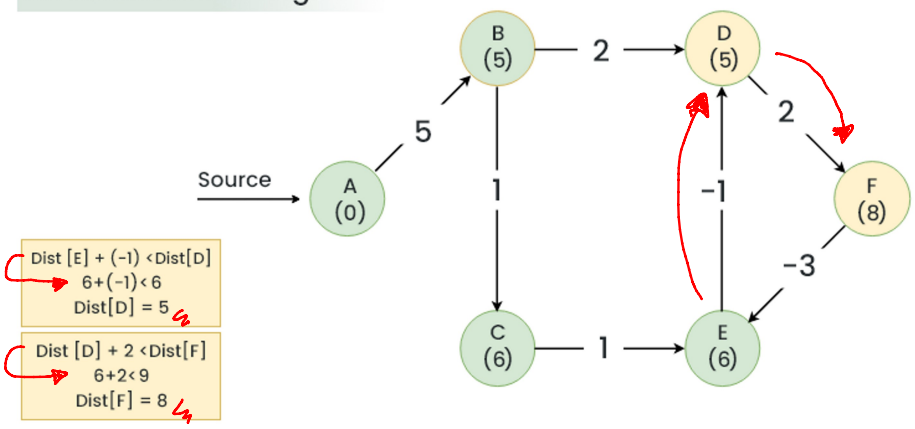
A	B	C	D	E	F
0	5	6	7	7	9

A	B	C	D	E	F
0	5	6	6	6	9

6V → V-1 Relaxations (supposed to stop here)

6V → V-1 Relaxations (supposed to stop here)

5th Relaxation Of Edges



Distance Array

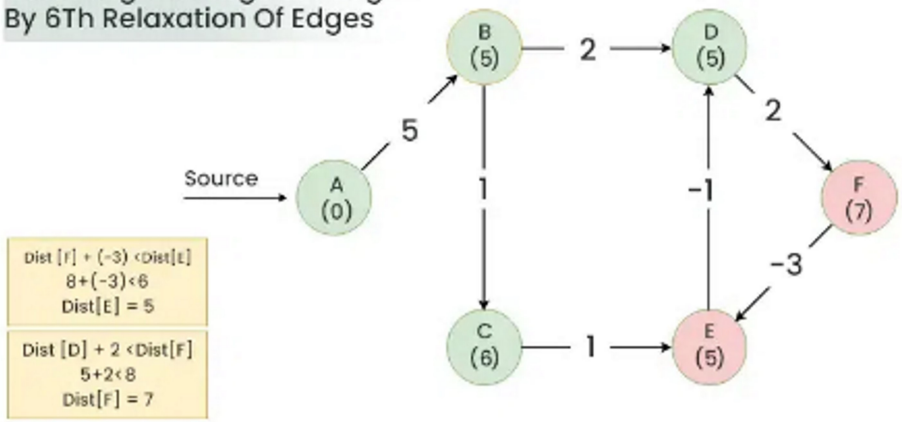
A	B	C	D	E	F
0	5	6	6	6	9
0	5	6	5	6	8

Bellman-Ford To Detect A Negative Cycle In A Graph

Step 7: *Now the final relaxation i.e. the 6th relaxation should indicate the presence of negative cycle if there is any changes in the distance array of 5th relaxation.*

During the 6th relaxation, following changes can be seen:

Detecting The Negative Edge By 6Th Relaxation Of Edges



Distance Array

A	B	C	D	E	F
0	5	6	5	6	8
0	5	6	5	4	7

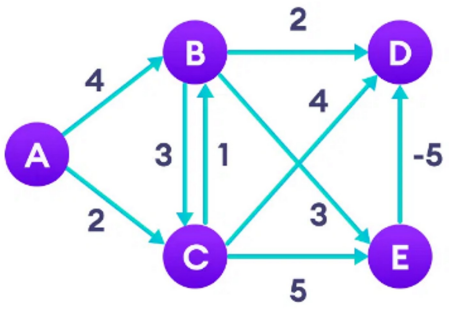
Bellman-Ford To Detect A Negative Cycle In A Graph

Algorithm to Find Negative Cycle in a Directed Weighted Graph Using Bellman-Ford:

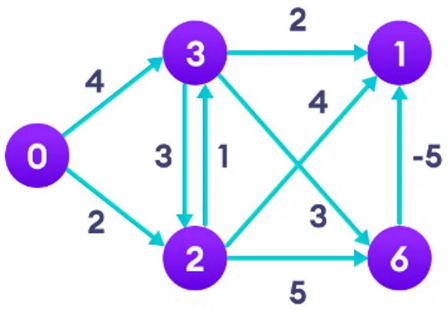
- Initialize distance array dist[] for each vertex 'v' as **dist[v] = INFINITY**.
- Assume any vertex (let's say '0') as source and assign **dist = 0**.
- Relax all the **edges(u,v,weight) N-1** times as per the below condition:
    - **dist[v] = minimum(dist[v], distance[u] + weight)**

  (included all edge into a vector to loop it later)
- Now, Relax all the edges one more time i.e. the **Nth** time and based on the below two cases we can detect the negative cycle:
    - Case 1 (Negative cycle exists): For any **edge(u, v, weight), if dist[u] + weight < dist[v]**
    - Case 2 (No Negative cycle) : case 1 fails for all the edges.

Step 1: Start with the weighted graph



Step 5: Notice how the vertex at the top right corner had its patha length adjusted



→ The order matters

↳ If just follow the adj list here E will receive 7 instead of 6

↳ It would process (A → B → C)

↳ following this logic will lead B = 4 instead of 3

SOLUTION → Include the adj node into a vector and Relax the edges from there

↳ It'll cover all edges and not just the adj one for current node