Spanning Trees

Introduction

Consider the system of roads in Maine represented by the simple graph shown bellow.

The only way the roads can be kept open in the winter is by frequently plowing them.

The highway department wants to plow the fewest roads so that there will always be cleared roads connecting any two towns.

How can this be done?

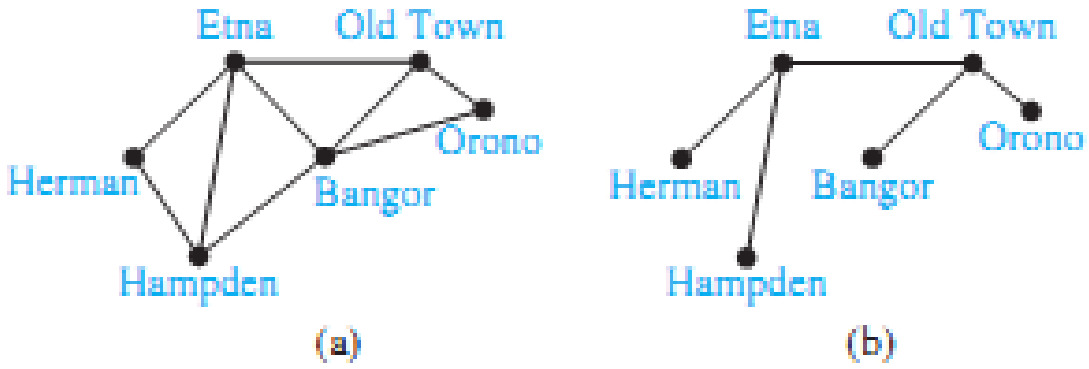
At least 5 roads must be plowed to ensure that there is a path between any two towns.

Figure 1(b) shows one such set of roads.

Note that the subgraph representing these roads is a tree, because it is connected and contains 6 vertices and 5 edges.

This problem was solved with a connected subgraph with the minimum number of edges containing all vertices of the original simple graph.

Such a graph must be a tree.



(a) A Road System

(b) a Set of Roads to Plow.

DEFINITION 1

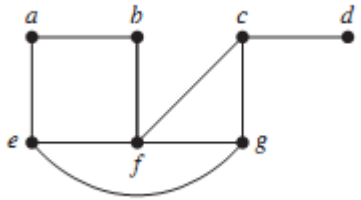
Let *G* be a simple graph.

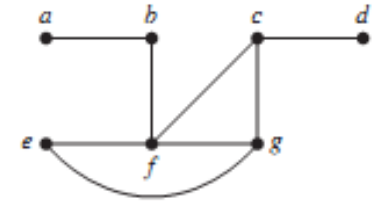
A spanning tree (остовное дерево) of *G* is a subgraph of *G* that is a tree containing every vertex of *G*.

A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any 2 vertices.

The converse is also true; that is, every connected simple graph has a spanning tree.

EXAMPLE





Edge removed: $\{a, e\}$

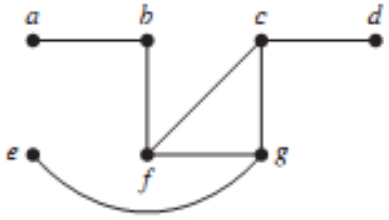
(a)

Find a spanning tree of the simple graph *G*.

Solution: The graph *G* is connected, but it is not a tree because it contains simple circuits.

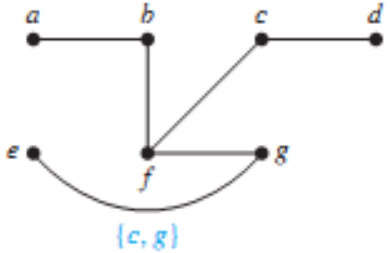
Remove the edge $\{a, e\}$.

This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of *G*.



(b)

Next remove the edge $\{e, f\}$ to eliminate a second simple circuit.



(c)

Finally, remove edge $\{c, g\}$ to produce a simple graph with no simple circuits.

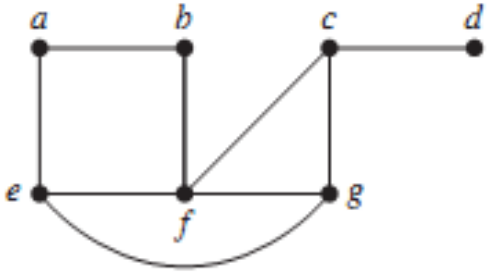
This subgraph is a spanning tree, because it is a tree that contains every vertex of *G*.

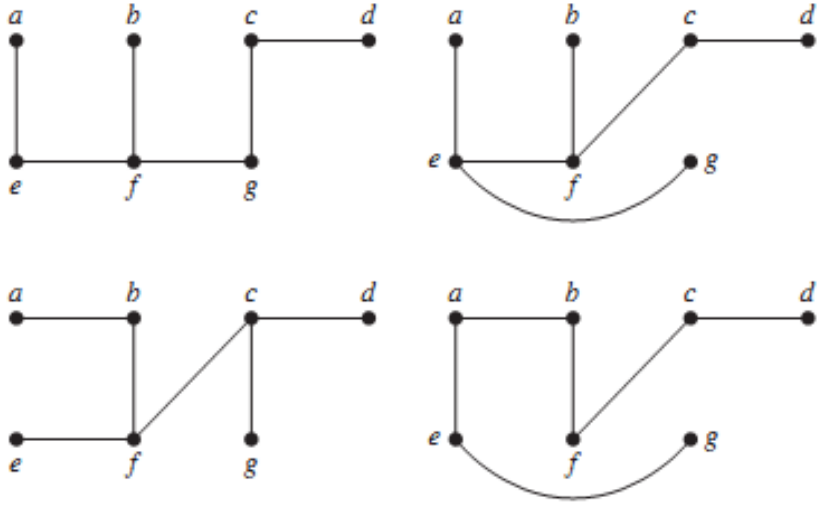
.

Other spanning trees of *G*

The tree shown in Example 1 is not the only spanning tree of *G*.

For instance, each of the trees bellow is a spanning tree of *G*.





THEOREM 1

THEOREM 1 A simple graph is connected ⇔ it has a spanning tree.

Proof:

<= First, suppose that a simple graph *G* has a spanning tree *T* .

T contains every vertex of *G*.

Furthermore, there is a path in *T* between any two of its vertices.

Because *T* is a subgraph of *G*, there is a path in *G* between any two of its vertices.

Hence, *G* is connected.

=> Now suppose that *G* is connected.

If *G* is not a tree, it must contain a simple circuit.

Remove an edge from one of these simple circuits.

The resulting subgraph has one fewer edge but still contains all the vertices of *G* and is connected.

This subgraph is still connected because when two vertices are connected by a path containing the removed edge, they are connected by a path not containing this edge.

We can construct such a path by inserting into the original path, at the point where the removed edge once was, the simple circuit with this edge removed.

If this subgraph is not a tree, it has a simple circuit; so as before, remove an edge that is in a simple circuit.

Repeat this process until no simple circuits remain.

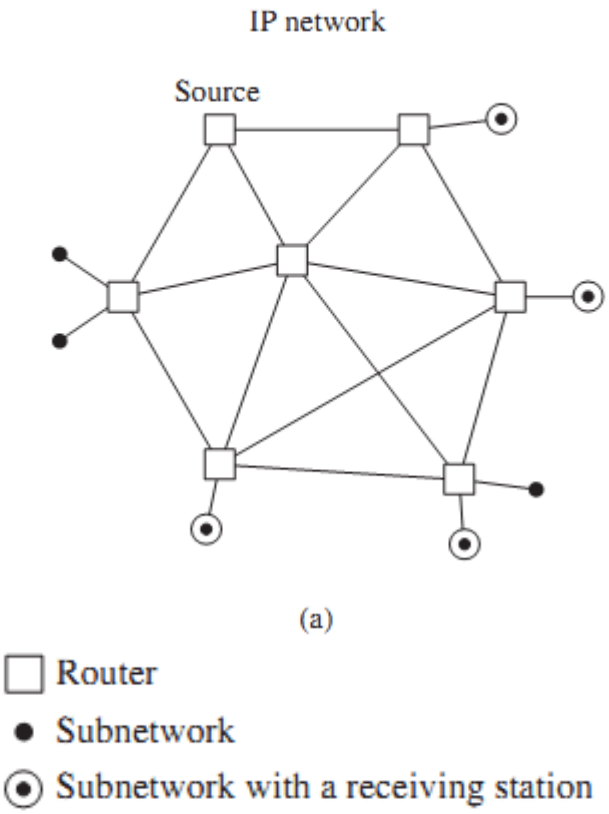
This is possible because there are only a finite number of edges in the graph.

The process terminates when no simple circuits remain.

A tree is produced because the graph stays connected as edges are removed.

This tree is a spanning tree because it contains every vertex of *G*.

EXAMPLE IP Multicasting



Spanning trees play an important role in multicasting (многоадресная передача) over Internet Protocol (IP) networks.

To send data from a source computer to multiple receiving computers, each of which is a subnetwork, data could be sent separately to each computer.

This type of networking, called unicasting (одноадресная передача), is inefficient, because many copies of the same data are transmitted over the network.

To make the transmission of data to multiple receiving computers more efficient, IP multicasting is used.

With IP multicasting, a computer sends a single copy of data over the network, and as data reaches intermediate routers (маршрутизаторы), the data are forwarded to one or more other routers so that ultimately all receiving computers in their various subnetworks receive these data.

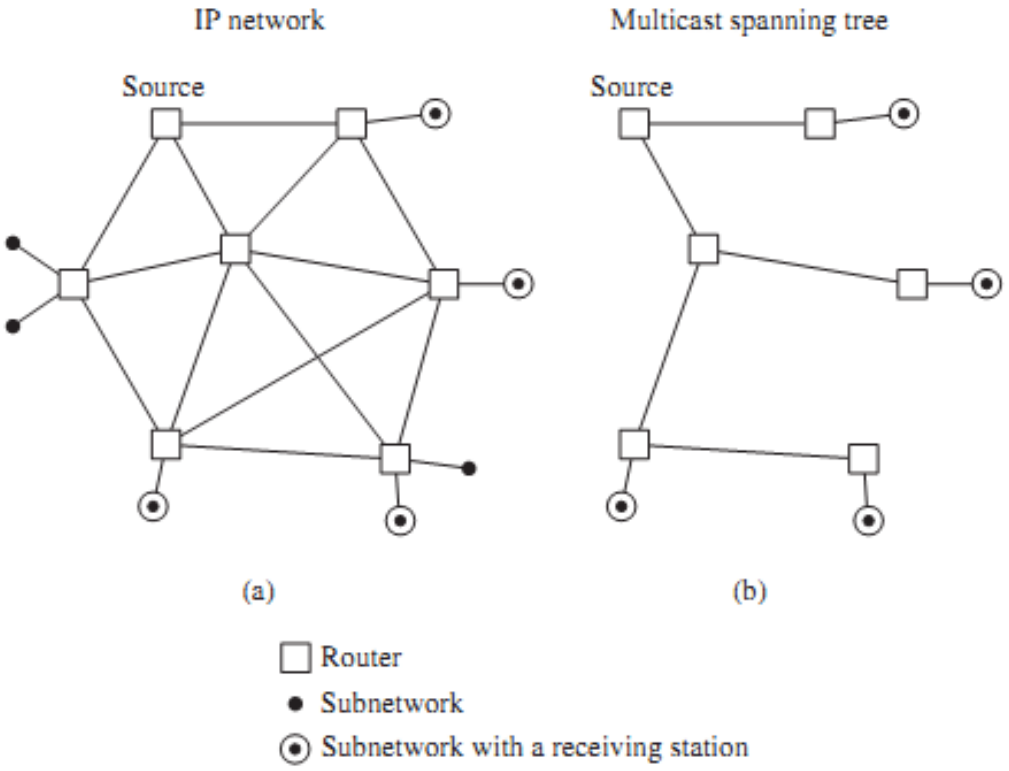
(Routers are computers that are dedicated to forwarding IP datagrams between subnetworks in a network).

For data to reach receiving computers as quickly as possible, there should be no loops (which in graph theory terminology are circuits or cycles) in the path that data take through the network.

That is, once data have reached a particular router, data should never return to this router.

To avoid loops, the multicast routers use network algorithms to construct a spanning tree in the graph that has the multicast source, the routers, and the subnetworks containing receiving computers as vertices, with edges representing the links between computers and/or routers.

Example: A Multicast Spanning Tree.



The root of this spanning tree is the multicast source.
The subnetworks containing receiving computers are leaves of the tree.
(Note that subnetworks not containing receiving stations are not included in the graph.)
This is shown in Figure above.

The proof of Theorem 1 gives an algorithm for finding spanning trees by removing edges from

simple circuits.

This algorithm is inefficient, because it requires that simple circuits be identified.

Instead of constructing spanning trees by removing edges, spanning trees can be built up by successively adding edges.

2 algorithms based on this principle will be considered:

- Breadth-first search
- Depth-first search

Breadth-First Search (BFS)

We can produce a spanning tree of a simple graph by the use of breadth-first search.

Again, a rooted tree will be constructed, and the underlying undirected graph of this rooted tree forms the spanning tree.

Arbitrarily choose a root from the vertices of the graph.

Then add all edges incident to this vertex.

The new vertices added at this stage become the vertices at level 1 in the spanning tree.

Arbitrarily order them.

Next, for each vertex at level 1, visited in order, add each edge incident to this vertex to the tree if it does not produce a simple circuit.

Arbitrarily order the children of each vertex at level 1.

This produces the vertices at level 2 in the tree.

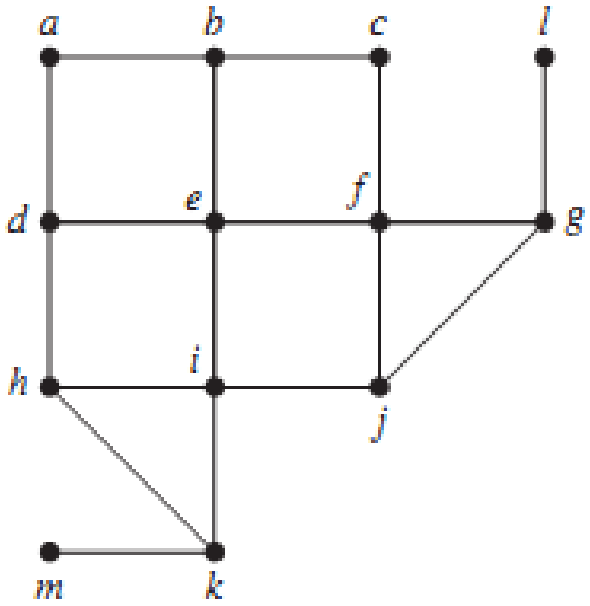
Follow the same procedure until all the vertices in the tree have been added.

The procedure ends because there are only a finite number of edges in the graph.

A spanning tree is produced because we have produced a tree containing every vertex of the graph.

EXAMPLE

Use breadth-first search to find a spanning tree for the graph shown bellow.



A Graph *G.*

Solution: The steps of the breadth-first search procedure are shown bellow.

We choose the vertex *e* to be the root.

Then we add edges incident with all vertices adjacent to *e*, so edges from *e* to *b, d, f* , and *i* are added.

These 4 vertices are at level 1 in the tree.

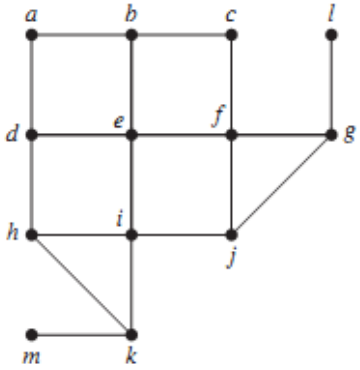
Next, add the edges from these vertices at level 1 to adjacent vertices not already in the tree.

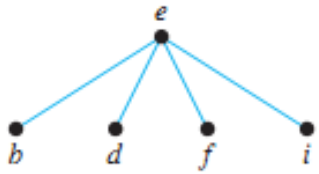
Hence, the edges from *b* to *a* and *c* are added, as are edges from *d* to *h*, from *f* to *j* and *g*, and from *i* to *k*.

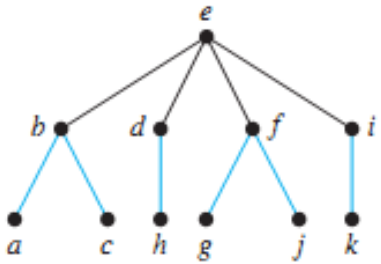
The new vertices *a, c, h, j , g*, and *k* are at level 2.

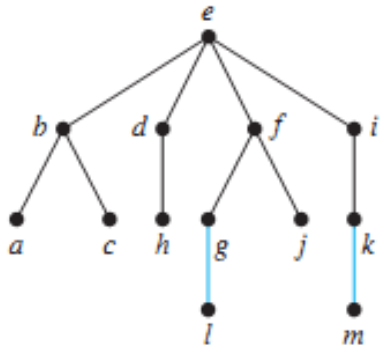
Next, add edges from these vertices to adjacent vertices not already in the graph.

This adds edges from *g* to *l* and from *k* to *m*.









ALGORITHM 1 Breadth-First Search (BFS1).

ALGORITHM 1 Breadth-First Search.

procedure BFS1 (*G*: connected graph with vertices v_1, v_2, ..., v_n)

T := tree consisting only of vertex v_1

L := empty list

put v_1 in the list *L* of unprocessed vertices

while *L* is not empty

    remove the first vertex, *v*, from *L*

    for each neighbor *w* of *v*

        if *w* is not in *L* and not in *T* then

            add *w* to the end of the list *L*

            add *w* and edge {*v, w*} to *T*

Extended breadth-first-search (BFS2)

The breadth-first-search procedure BFS2 below assumes that the input graph $G = (V, E)$ is represented using adjacency lists.

It attaches several additional attributes to each vertex in the graph.

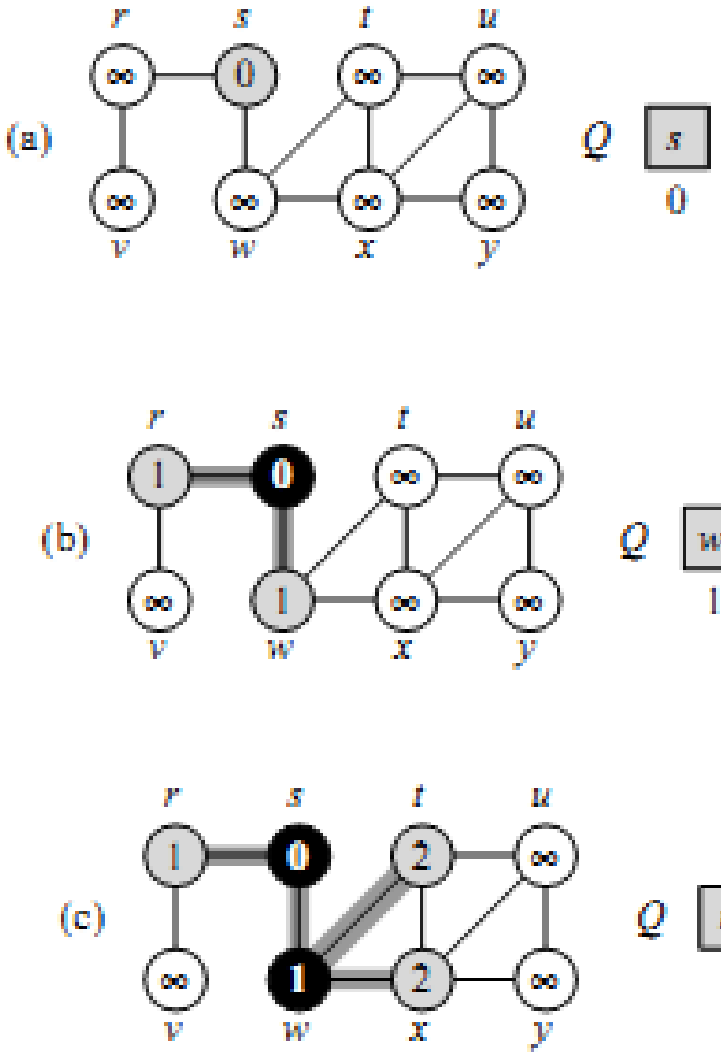
We store the color of each vertex $u \in V$ in the attribute $u.color$ and the predecessor of u in the attribute $u.\pi$

If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $u.\pi$ = NIL.

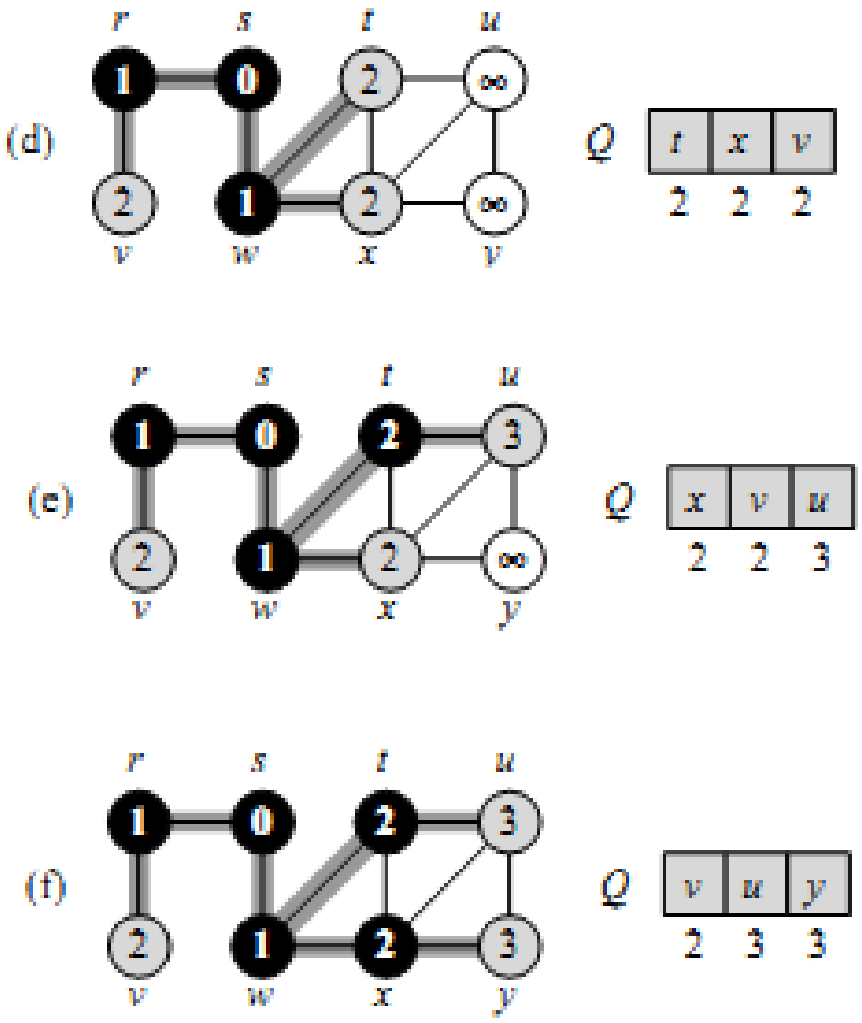
The attribute $u.d$ holds the distance from the source s to vertex u computed by the algorithm.

The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices.

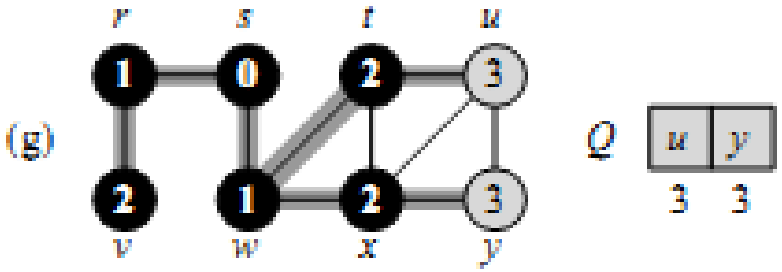
```
BFS2(G, s)
1 for each vertex u ∈ G.V - {s}
2       u.color = WHITE
3 .     u.d = ∞
4       u.π = NIL
5 s.color = GRAY
6 s.d = 0
7 s.π = NIL
8 Q =∅ ;
9 ENQUEUE(Q, s)
10 while Q ≠∅ ;
11        u = DEQUEUE(Q)
12        for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                v.color = GRAY
15                v.d = u.d + 1
16                v.π = u
17                ENQUEUE(Q, v)
18        u.color = BLACK
```

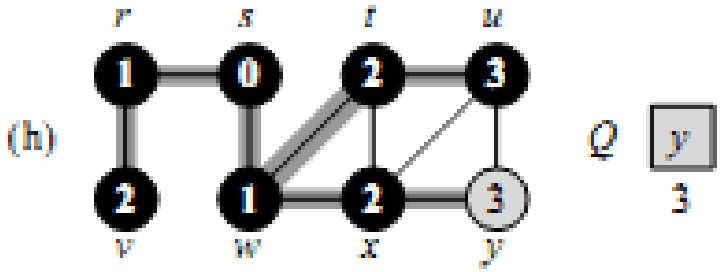


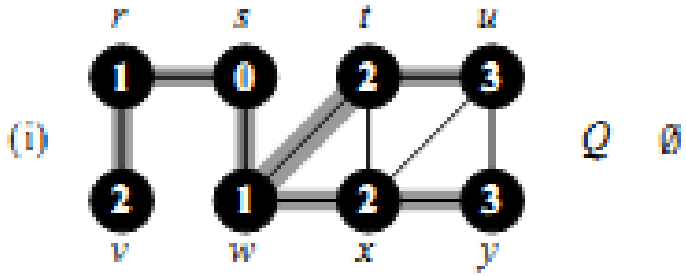

```
BFS2(G, s)
1 for each vertex u ∈ G.V - {s}
2       u.color = WHITE
3 .     u.d = ∞
4       u.π = NIL
5 s.color = GRAY
6 s.d = 0
7 s.π = NIL
8 Q =∅ ;
9 ENQUEUE(Q, s)
10 while Q ≠∅ ;
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```




```
BFS2(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 v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```








```
BFS2(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 v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16               v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Invariant:

At the test in line 10, the queue Q consists of the set of gray vertices.

The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12:

the breadth-first tree may vary, but the distances d computed by the algorithm will not.

Analysis

Before proving the various properties of breadth-first search, let us analyze its running time on an input graph $G = (V, E)$.

After initialization, BFS 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 the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$.

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.

Shortest paths

BFS2 finds the distance to each reachable vertex in a graph $G = (V, E)$ from a given source vertex $s \in V$.

Define the shortest-path distance $\delta(s, v)$ (кратчайшее расстояние) from s to v as the minimum number of edges in any path from vertex s to vertex v;

if there is no path from s to v, then $\delta(s, v) = \infty$.

We call a path of length $\delta(s, v)$ from s to v a shortest path (кратчайший путь) from s to v.

Lemma 2

Lemma 2 Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex.

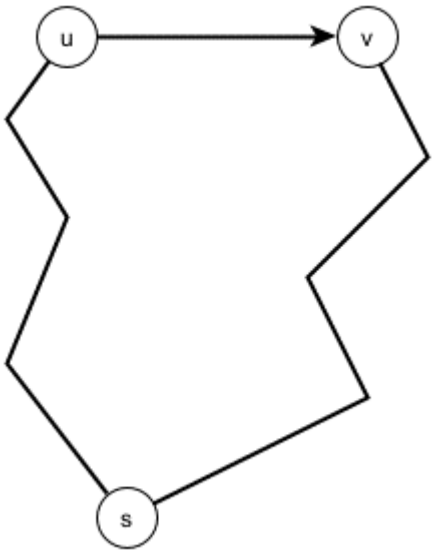
Then, for any edge $(u, v) \in E$,

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

Proof If u is reachable from s, then so is v.

In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v), and thus the inequality holds.

If u is not reachable from s, then $\delta(s, u) = \infty$, and the inequality holds.



Lemma 3

We want to show that BFS properly computes $v.d = \delta(s, v)$ for each vertex $v \in V$.

We first show that $v.d$ bounds $\delta(s, v)$ from above.

Lemma 3 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 upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies

$v.d \geq \delta(s, v)$.

Proof We use induction on the number of ENQUEUE operations.

Our inductive hypothesis is that $v.d \geq \delta(s, v)$ for all $v \in V$.

1) The basis of the induction is the situation immediately after enqueuing s in line 9 of BFS.

The inductive hypothesis holds here, because $s.d = 0 = \delta(s, s)$ and

$v.d = \infty \geq \delta(s) \quad \forall v \in V - \{s\}$.

For the inductive step, consider a white vertex v that is discovered during the search from a vertex u.

The inductive hypothesis implies that $u.d \geq \delta(s, u)$.

From the assignment performed by line 15 of the BFS2 and from Lemma 2, we obtain

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) . \end{aligned}$$

Vertex v is then enqueued, and it is never enqueued again because it is also grayed and the **then** clause of lines 14–17 is executed only for white vertices.

Thus, the value of $v.d$ never changes again, and the inductive hypothesis is maintained.

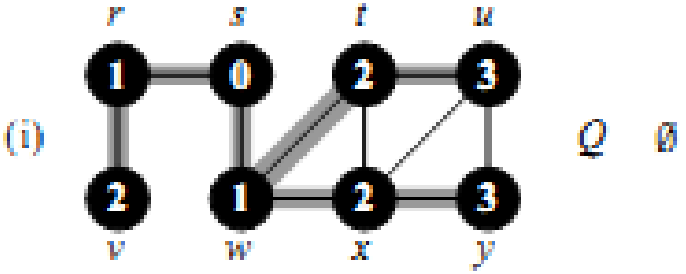
To prove that $v.d = \delta(s, v)$, we must consider more precisely how the queue Q operates during the course of BFS (colloquium).

Theorem 4 (Correctness of BFS-2)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS2 is run on G from a given source vertex $s \in V$.

Then, during its execution, BFS2 discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ $\forall$ $v \in V$.

Breadth-first trees



The procedure BFS2 builds a breadth-first tree as it searches the graph.

The tree corresponds to the π attributes.

More formally, for a graph $G = (V, E)$ with source s, we define the predecessor subgraph of G as $G_\pi = (V_\pi, E_\pi)$,

where $V_\pi = \{ v \in V : v.\pi \neq \text{NIL} \} \cup \{s\}$

And $E_\pi = \{ (v.\pi, v) : v \in V_\pi - \{s\} \}$.

The predecessor subgraph G_π is a breadth-first tree if V consists of the vertices reachable from s and, for all $v \in V$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G.

A breadth-first tree is in fact a tree, since it is connected and $|E| = |V| - 1$.

We call the edges in E *tree edges*.

The following lemma shows that the predecessor subgraph produced by the BFS2 procedure is a breadth-first tree.

Lemma 5 When applied to a directed or undirected graph $G = (V, E)$, procedure BFS2 constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Proof Line 16 of BFS2 sets $v.\pi = u \Leftrightarrow (u, v) \in E$ and $\delta(s, v) < \infty$— that is, if v is reachable from s

— thus V_π consists of the vertices in V reachable from s.

Since G_π forms a tree, it contains a unique simple path from s to each vertex in V_π.

By applying Theorem 5 inductively, we conclude that every such path is a shortest path in G.

Depth-first search

We can build a spanning tree for a connected simple graph using depth-first search (DFS).

We will form a rooted tree, and the spanning tree will be the underlying undirected graph of this rooted tree.

Arbitrarily choose a vertex of the graph as the root.

Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path.

Continue adding vertices and edges to this path as long as possible.

If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree.

However, if the path does not go through all vertices, more vertices and edges must be added.

Move back to the next to last vertex in the path, and, if possible, form a new path starting at this vertex passing through vertices that were not already visited.

If this cannot be done, move back another vertex in the path, that is, 2 vertices back in the path, and try again.

Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added.

Because the graph has a finite number of edges and is connected, this process ends with the production of a spanning tree.

Each vertex that ends a path at a stage of the algorithm will be a leaf in the rooted tree, and

each vertex where a path is constructed starting at this vertex will be an internal vertex.

a)Note the recursive nature of this procedure.

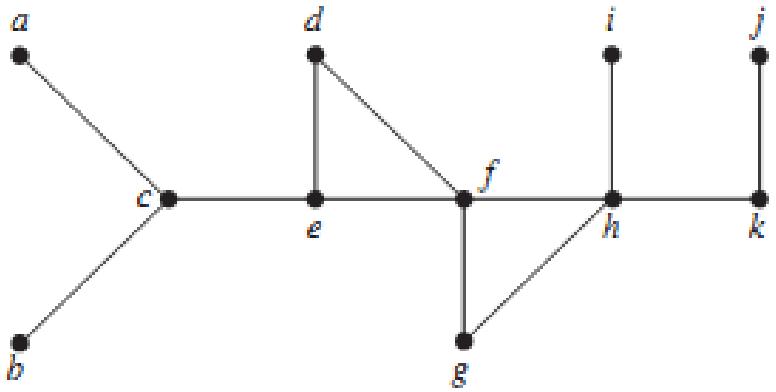
b) Also, note that if the vertices in the graph are ordered, the choices of edges at each stage of the procedure are all determined when we always choose the first vertex in the ordering that is available.

However, we will not always explicitly order the vertices of a graph.

Depth-first search is also called backtracking (откат), because the algorithm returns to vertices previously visited to add paths.

Example bellow illustrates backtracking.

EXAMPLE Use depth-first search to find a spanning tree for the graph *G*.



Solution: The steps used by depth-first search to produce a spanning tree of *G* are shown bellow.

We arbitrarily start with the vertex *f* .

A path is built by successively adding edges incident with vertices not already in the path, as long as this is possible.

This produces a path *f, g, h, k, j* (note that other paths could have been built).

Next, backtrack to *k*.

There is no path beginning at *k* containing vertices not already visited.

So we backtrack to *h*.

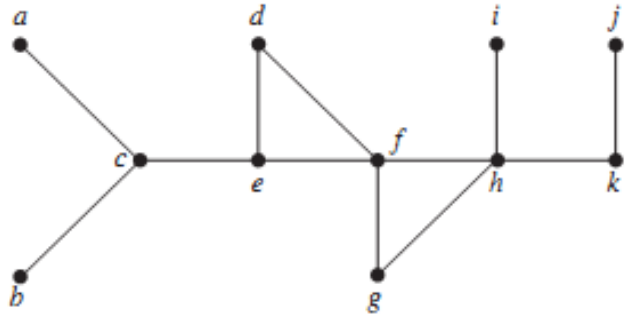
Form the path *h, i*.

Then backtrack to *h*, and then to *f* .

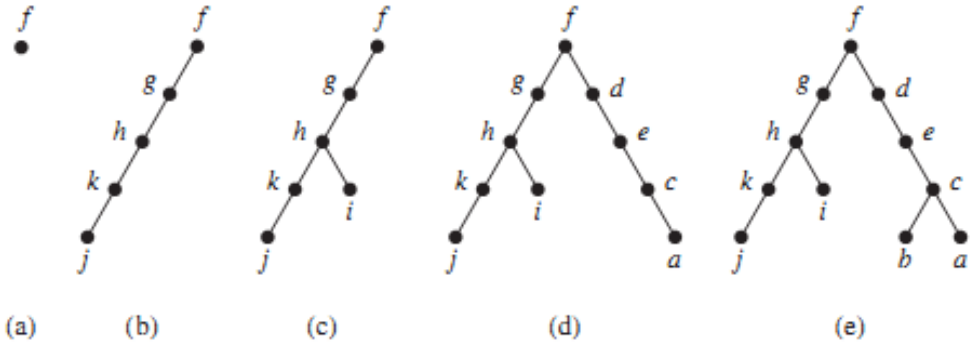
From f build the path *f, d, e, c, a*.

Then backtrack to *c* and form the *path c, b*.

This produces the spanning tree.



The Graph G.



Depth-First Search of G.

We have explained how to find a spanning tree of a graph using depth-first search.

However, our discussion so far has not brought out the recursive nature of depth-first search.

To help make the recursive nature of the algorithm clear, we need a little terminology.

We say that we explore (осуществляем поиск) from a vertex *v* when we carry out the steps of depth-first search beginning when *v* is added to the tree and ending when we have backtracked back to *v* for the last time.

The key observation needed to understand the recursive nature of the algorithm is that when we add an edge connecting a vertex *v* to a vertex *w*, we start exploring from *w.*

And we finish exploring from *w* before we return to *v* to complete exploring from *v*.

In Algorithm DFS1 we construct the spanning tree of a graph G with vertices $v_1, ..., v_n$ by first selecting the vertex v_1 to be the root.

We initially set T to be the tree with just v_1.

At each step we add a new vertex to the tree T together with an edge from a vertex already in T to this new vertex and we explore from this new vertex.

Note that at the completion of the algorithm, T contains no simple circuits because no edge is ever added that connects 2 vertices in the tree.

Moreover, T remains connected as it is built.

(These last 2 observations can be easily proved via mathematical induction.)

Because G is connected, every vertex in G is visited by the algorithm and is added to the tree.

It follows that T is a spanning tree of G.

ALGORITHM 3 Depth-First Search (DFS1)

ALGORITHM 3 DFS1.

procedure DFS1(*G*: connected graph with vertices $v_1, v_2, \ldots, v_n$)

T := tree consisting only of the vertex v_1

visit(v_1)

procedure *visit*(*v*: vertex of *G*)

for each vertex *w* adjacent to *v* and not yet in *T*

add vertex *w* and edge {*v*, *w*} to *T*

visit(*w*)

Depth-first search can be used as the basis for algorithms that solve many different problems.

For example, it can be used:

- to find paths and circuits in a graph,
- to determine the connected components of a graph,
- to find the cut vertices of a connected graph.

As we will see, depth-first search is the basis of backtracking techniques used to search for solutions of computationally difficult problems.

Extended depth-first search (DFS2)

As in BFS2, whenever DFS2 discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute $v.\pi$ to u.

Unlike BFS2, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a DFS may be composed of several trees, because the search may repeat from multiple sources.

Therefore, we define the predecessor subgraph of a DFS slightly differently from that of a BFS:

$G_\pi = (V, E_\pi)$,

where $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$.

The predecessor subgraph of a DFS forms a depth-first forest comprising several depth-first trees.

The edges in E_π are tree edges.

DFS2

As in BFS2, DFS2 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 1 depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, DFS2 also timestamps each vertex.

Each vertex *v* has 2 timestamps ((метки времени):

the first timestamp *v.d* records when *v* is first discovered (and grayed),

the second timestamp *v.f* records when the search finishes examining *v*'s adjacency list (and blackens *v*).

These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of DFS2.

DFS2

The procedure DFS2 below records when it discovers vertex u in the attribute $u.d$ and when it finishes vertex u in the attribute $u.f$.

These timestamps are integers between 1 and $2|V|$, since there is 1 discovery event and 1 finishing event for each of the $|V|$ vertices.

For every vertex u,

$u.d < u.f$. (1)

Vertex u is

WHITE before time $u.d$,

GRAY between time $u.d$ and time $u.f$, and

BLACK thereafter.

The following pseudocode is the basic DFS2 algorithm.

The input graph G may be undirected or directed.

The variable *time* is a global variable that we use for timestamping.

Algorithm 4 (DFS2)

```
DFS2(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)

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  v ∈ G.Adj[u]      // explore edge (u, v) /
5       if v.color == WHITE
6             v.π = u
7     DFS-VISIT(G, v)
8 u.color =BLACK              // blacken u; it is finished
9 time = time + 1
10 u. f = time
```

Procedure DFS2works as follows.

Lines 1–3 paint all vertices white and initialize their π attributes to NIL.

Line 4 resets the global time counter.

Lines 5–7 check each vertex in *V* in turn and, when a white vertex is found, visit it using DFS-VISIT.

Every time DFS-VISIT(*G, u*) is called in line 7, vertex *u* becomes the root of a new tree in the depth-first forest.

When DFS2 returns, every vertex *u* has been assigned a discovery time *u.d* and a finishing time *u.f* .

In each call DFS-VISIT(*G, u*), vertex *u* is initially white.

Line 1 increments the global variable *time*,

line 2 records the new value of *time* as the discovery time *u.d*,

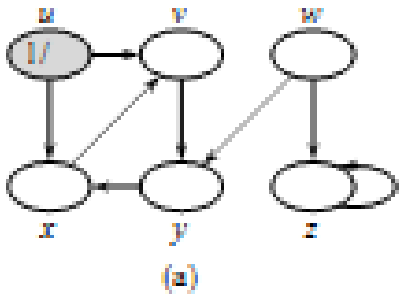
and line 3 paints *u* gray.

Lines 4–7 examine each *vertex* *v* adjacent to *u* and recursively visit *v* if it is white.

As each vertex $v \in$ Adj[*u*] is considered in line 4, we say that edge (*u, v*) is explored by the DFS2.

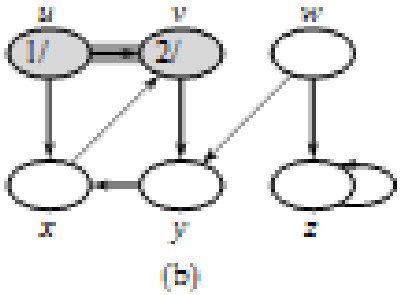
Finally, after every edge leaving *u* has been explored, lines 8–10 paint *u* black, increment , and record the finishing time in *u.f* .

EXAMLE:DFS2



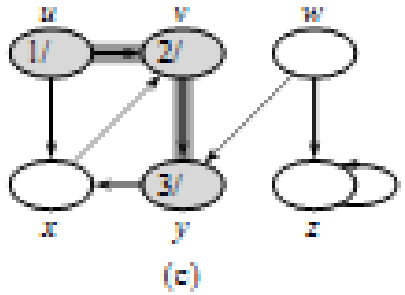
(a)

time = 1



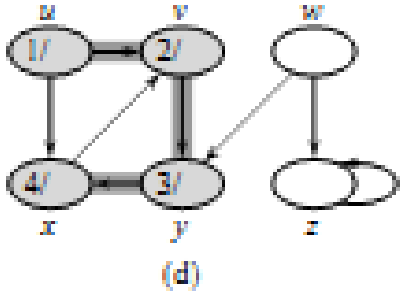
(b)

time = 2



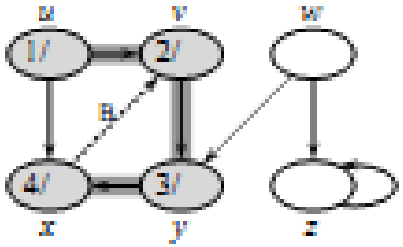
(c)

time = 3



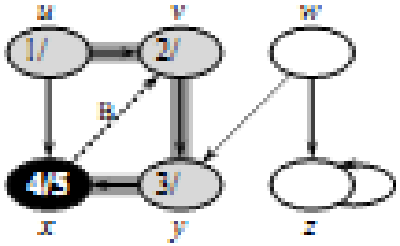
(d)

time = 4



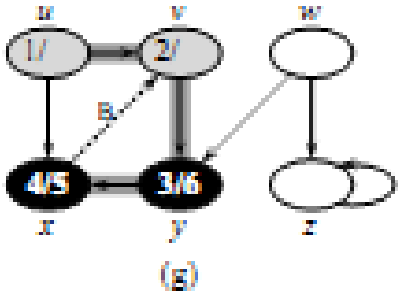
(e)

time = 4



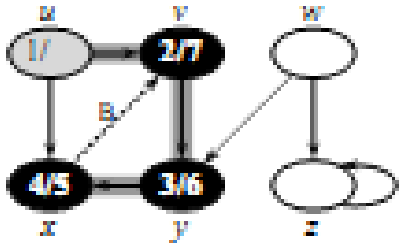
(f)

time = 5



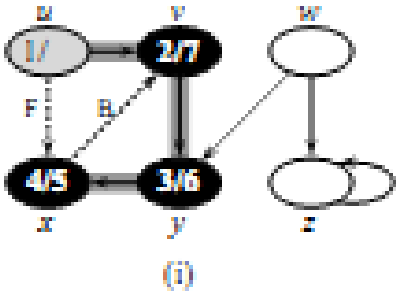
(g)

time = 6



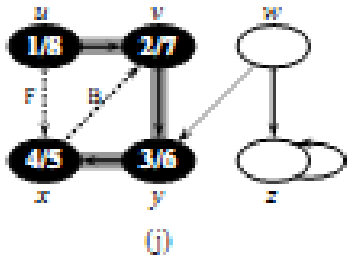
(h)

time = 7



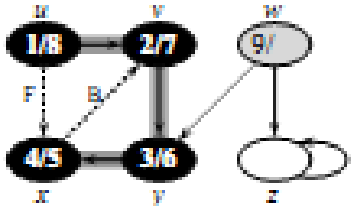
(i)

time = 7



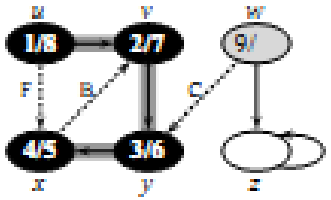
(j)

time = 8



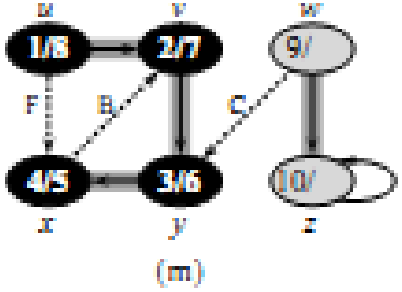
(k)

time = 9



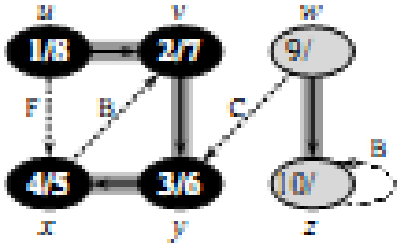
(l)

time = 9



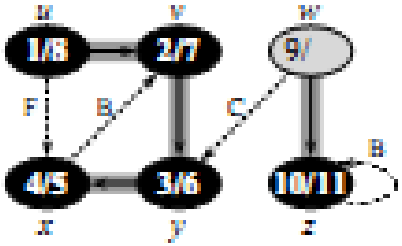
(m)

time = 10



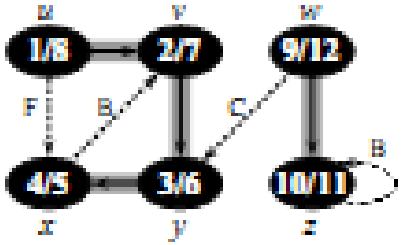
(n)

time = 10



(o)

time = 11



(p)

time = 12

Note that the results of DFS may depend upon the order in which line 5 of DFS2 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 problems in practice, as we can usually use any DFS result effectively, with essentially equivalent results.

The running time of DFS2

The loops on lines 1–3 and lines 5–7 of DFS2 take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT.

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 on lines 4–7 executes $|Adj[v]|$ times.

Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

the total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$.

The running time of DFS2 is therefore $\Theta(V + E)$.

Properties of DFS2

DFS2 yields valuable information about the structure of a graph.

Perhaps the most basic property of DFS is that the predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT.

That is, $u = v.\pi$ *(*vertex *u* is the parent of vertex *v)*: ⇔ DFS-VISIT(*G, v*) was called during a search of *u*'s adjacency list.

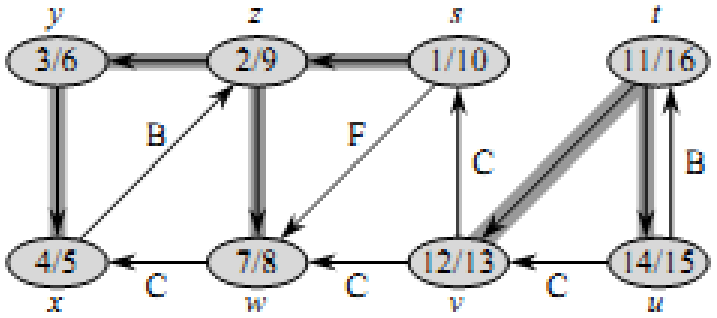
Additionally, vertex *v* is a descendant of vertex *u* in the depth-first forest ⇔ *v* is discovered during the time in which *u* is gray.

Another important property of DFS2 is that discovery and finishing times have parenthesis structure.

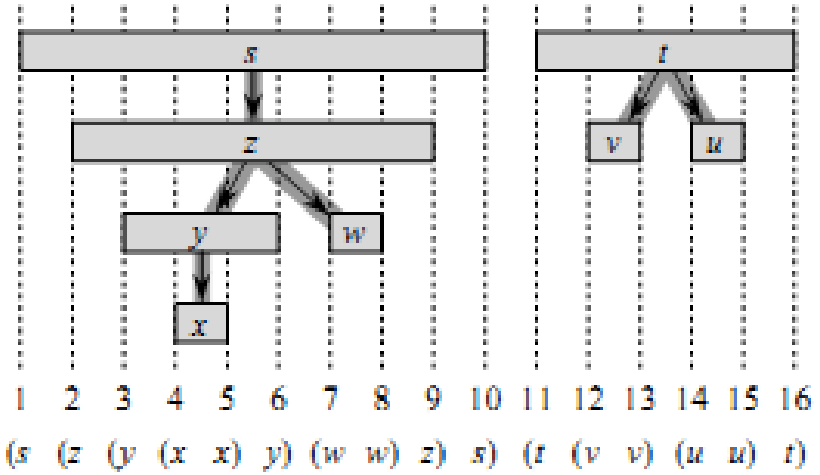
If we represent the discovery of vertex *u* with a left parenthesis "(*u*" and represent its finishing by a right parenthesis "*u*) ", then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested.

For example, the DFS2 of Figure (a) corresponds to the parenthesization shown in Figure (b).

(a)



(b)



(b) Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex.

Only tree edges are shown.

If 2 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.

The following theorem provides another way to characterize the parenthesis structure.

Theorem 6 (Parenthesis theorem)

In any DFS2 of a (directed or undirected) graph $G = (V, E)$ for any 2 vertices u and v, exactly 1 of the following 3 conditions holds:

the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,

the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree,

or

the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

Proof We begin with the case in which $u.d < v.d$.

We consider 2 subcases,

according to whether $v.d < u.f$ or not.

1) The first subcase occurs when $v.d < u.f$,

so v was discovered while u was still gray,

which implies that v is a descendant of u.

Moreover, since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u.

In this case, therefore, the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$.

2) In the other subcase, $u.f < v.d$, and by inequality (1),

$u.d < u.f < v.d < v.f$;

thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint.

Because the intervals are disjoint,

neither vertex was discovered while the other was gray,

and so neither vertex is a descendant of the other.

The case in which $v.d < u.d$ is similar, with the roles of u and v reversed in the above argument.

Corollary 7 (Nesting of descendants' intervals)

Vertex *v* is a proper descendant of vertex *u* in the depth-first forest for a (directed or undirected) graph *G* $\Leftrightarrow u.d < v.d < v.f < u.f$.

Proof Immediate from Theorem 6.

Classification of edges

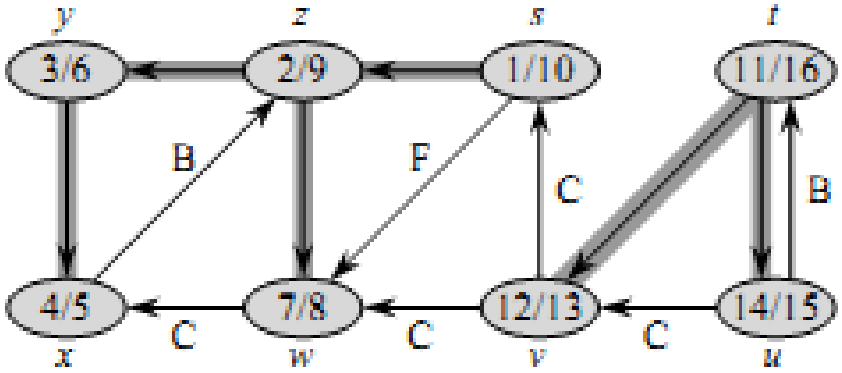
Another interesting property of DFS is that the search can be used to classify the edges of the input graph $G = (V, E)$.

The type of each edge can provide important information about a graph.

For example, a directed graph is acyclic ⇔ a DFS yields no "back" edges.

We can define 4 edge types in terms of the depth-first forest G_π produced by a DFS2 on directed graph G:

1. Tree edges are edges in the depth-first forest $G_{\pi..}$ Edge (u, v) is a tree edge if was first discovered by exploring edge (u, v) .
2. Back edges are those edges (u, v) connecting a vertex u to an ancestor 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 descendant 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.



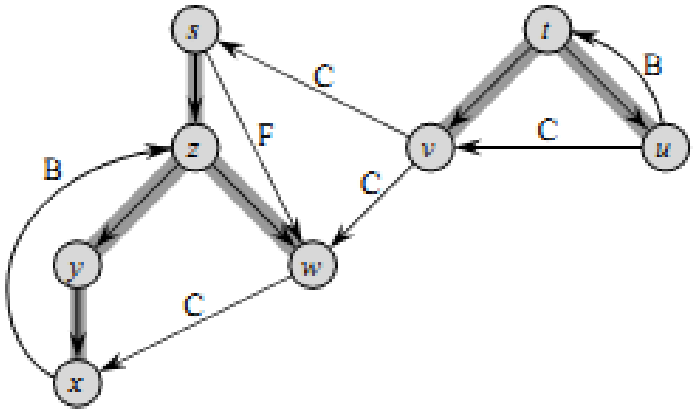


Figure (c) also shows how to redraw the graph of Figure (a) so that all tree and forward edges head downward in a depth-first tree and all back edges go up.

The DFS2 algorithm has enough information to classify some edges as it encounters them.

The key idea is that when we first explore an edge (*u, v*), the color of vertex *v* tells us something about the edge:

1. WHITE indicates a tree edge,
2. GRAY indicates a back edge, and
3. BLACK indicates a forward or cross edge.

We now show that forward and cross edges never occur in a DFS of an undirected graph.

Theorem 8

In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

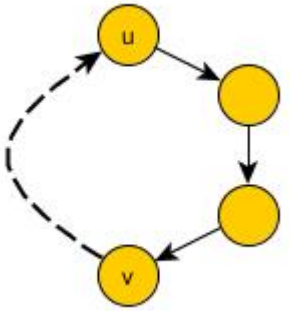
Proof Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $u.d < v.d$.

Then the search must discover and finish v before it finishes u (while u is gray), since v is on u's adjacency list.

1)If the first time that the search explores edge (u, v), it is in the direction from u to v , then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u.

Thus, (u, v) becomes a tree edge.

2) If the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge, since u is still gray at the time the edge is first explored.



Backtracking Applications

There are problems that can be solved only by performing an exhaustive search of all possible solutions.

One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution.

To find a solution via backtracking, first make a sequence of decisions in an attempt to reach a solution as long as this is possible.

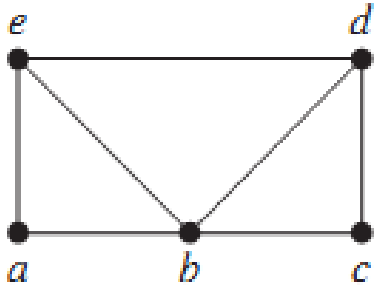
The sequence of decisions can be represented by a path in the decision tree.

Once it is known that no solution can result from any further sequence of decisions**, backtrack to the parent of the current vertex** and work toward a solution with another series of decisions, if this is possible.

The procedure continues until a solution is found, or it is established that no solution exists.

EXAMPLE Graph Colorings

How can backtracking be used to decide whether a graph can be colored using *n* colors?



Solution: We can solve this problem using backtracking in the following way.

First pick some vertex *a* and assign it color 1.

Then pick a second vertex *b*, and

if *b* is not adjacent to *a*, assign it color 1.

Otherwise, assign color 2 to *b*.

Then go on to a third vertex *c*.

Use color 1, if possible, for *c*.

Otherwise use color 2, if this is possible.

Only if neither color 1 nor color 2 can be used should color 3 be used.

Continue this process as long as it is possible to assign one of the *n* colors to each additional vertex, always using the first allowable color in the list.

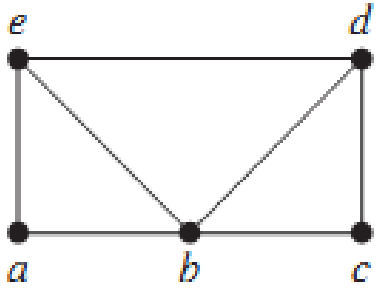
If a vertex is reached that cannot be colored by any of the *n* colors, backtrack to the last assignment made and change the coloring of the last vertex colored, if possible, using the next allowable color in the list.

If it is not possible to change this coloring, backtrack farther to previous assignments, one step back at a time, until it is possible to change a coloring of a vertex.

Then continue assigning colors of additional vertices as long as possible.

If a coloring using *n* colors exists, backtracking will produce it.

(Unfortunately this procedure can be extremely inefficient.)



In particular, consider the problem of coloring the graph shown bellow with 3 colors.

The tree shown bellow illustrates how backtracking can be used to construct a 3-coloring.

In this procedure, red is used first, then blue, and finally green.

This simple example can obviously be done without backtracking, but it is a good illustration of the technique.

In this tree, the initial path from the root, which represents the assignment of red to *a*, leads to a coloring with *a* red, *b* blue, *c* red, and *d* green.

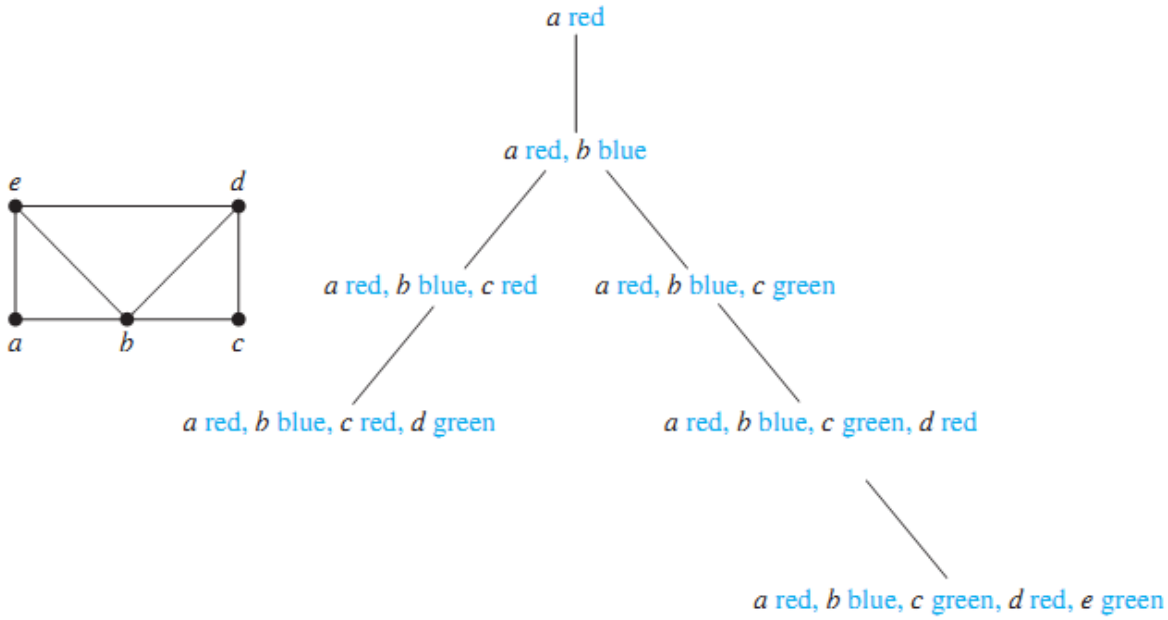
It is impossible to color *e* using any of the 3 colors when *a, b, c,* and *d* are colored in this way.

So, backtrack to the parent of the vertex representing this coloring.

Because no other color can be used for *d*, backtrack one more level.

Then change the color of *c* to green.

We obtain a coloring of the graph by then assigning red to *d* and green to *e.*



EXAMPLE The *n*-Queens Problem

The *n*-queens problem asks how *n* queens can be placed on an $n \times n$ chessboard so that no 2 queens can attack one another.

How can backtracking be used to solve the *n*-queens problem?

Solution: To solve this problem we must find *n* positions on an $n \times n$ chessboard so that no 2 of these positions are in the same row, same column, or in the same diagonal [a diagonal consists of all positions (i, j) with $i + j = m$ for some m, or $i - j = m$ for some m].

We will use backtracking to solve the *n*-queens problem.

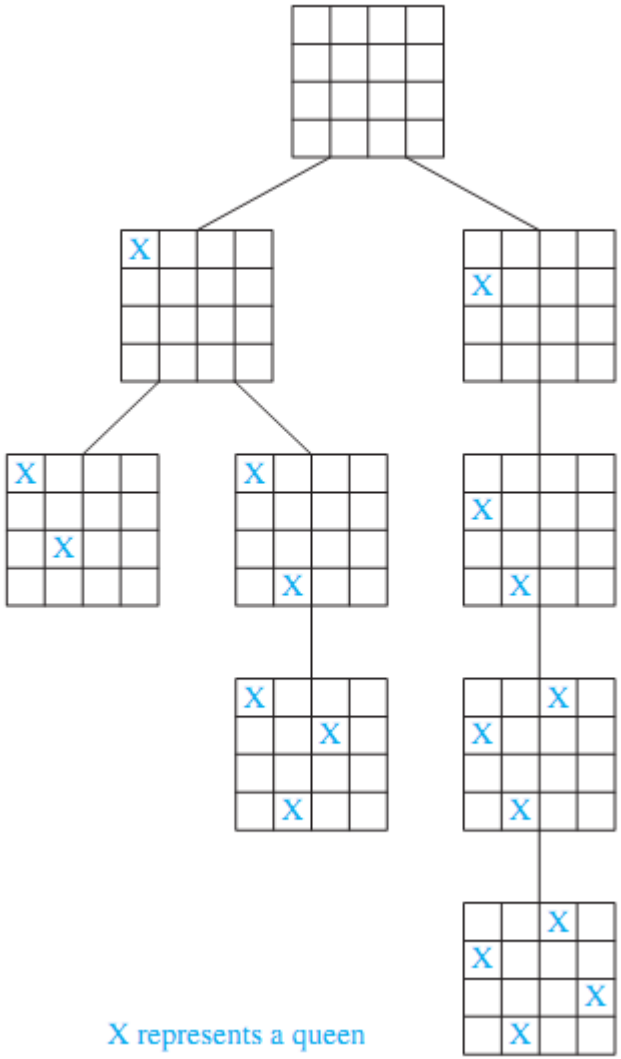
We start with an empty chessboard.

At stage $k + 1$ we attempt putting an additional queen on the board in the $(k + 1)$st column, where there are already queens in the first k columns.

We examine squares in the $(k + 1)$st column starting with the square in the first row, looking for a position to place this queen so that it is not in the same row or on the same diagonal as a queen already on the board.

(We already know it is not in the same column.)

A Backtracking Solution of the 4-Queens Problem.



If it is impossible to find a position to place the queen in the $(k + 1)$st column, backtrack to the placement of the queen in the kth column, and place this queen in the next allowable row in this column, if such a row exists.

If no such row exists, backtrack further.

In particular, Figure displays a backtracking solution to the 4-queens problem.

In this solution, we place a queen in the 1rst row and column.

Then we put a queen in the 3rd row of the second column.

However, this makes it impossible to place a queen in the 3'rd column.

So we backtrack and put a queen in the 4'th row of the second column.

When we do this, we can place a queen in the second row of the 3'rd column.

But there is no way to add a queen to the fourth column.

This shows that no solution results when a queen is placed in the first row and column.

We backtrack to the empty chessboard, and place a queen in the second row of the first column.

Example Sums of Subsets

Given a set of positive integers $x_1, x_2, ..., x_n$, find a subset of this set of integers that has M as its sum.

How can backtracking be used to solve this problem?

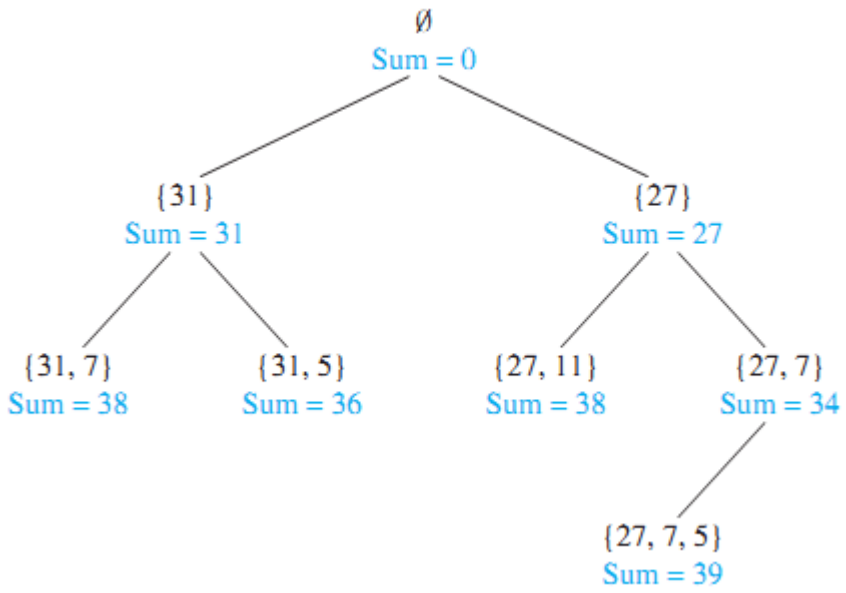
Solution: We start with a sum with no terms.

We build up the sum by successively adding terms.

An integer in the sequence is included if the sum remains $\leq M$ when this integer is added to the sum.

If a sum is reached such that the addition of any term is $> M$, backtrack by dropping the last term of the sum.

A backtracking solution to the problem of finding a subset of {31, 27, 15, 11, 7, 5} with the sum = 39.



Depth-first search in directed graphs is the basis of many algorithms.

It can be used:

- to determine whether a directed graph has a circuit,
- to carry out a topological sort of a graph,
- to find the biconnected components of a undirected graph,
- to find the strongly connected components of a directed graph.