Get To Know Your Trees

David Pritchard

Canadian Computing Competition, 2006 Stage 2

Outline

1. Preliminaries
2. Spanning Trees of Graphs
3. A General Framework
   - Depth-First Search
   - Breath-First Search
   - Minimum Spanning Tree
   - Dijkstra's Shortest Paths Algorithm
4. Advanced Tactics
   - A-Star, Meet in the Middle
   - Preorder, Postorder, Topological Sort
   - Biconnectivity, Strong Connectivity

What Is a Tree?

What Is a Tree?

What Is a Tree?

- A *forest* is a collection of trees.

What Is a Tree?

- Trees have lots of interesting characterizations as *graphs* …

What Is a Tree?

- Trees have lots of interesting characterizations as *graphs* ...
  - *A connected graph with no cycles*
  - *A graph where there is each pair of vertices is joined by a single path*

What Is a Tree?

- Trees have lots of interesting characterizations as *graphs* . . .
  - *A connected graph with no cycles*
  - *A graph where there is each pair of vertices is joined by a single path*
- . . . but we won't talk about this here.

Basic Botany

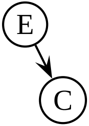
- In this talk we mainly deal with *rooted*, *labelled* trees.

Basic Botany

- In this talk we mainly deal with *rooted, labelled* trees.
- There is a *root vertex*.

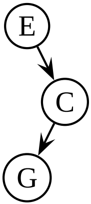
Basic Botany

- In this talk we mainly deal with *rooted, labelled* trees.
- There is a *root vertex*.
- Each other node that we add to the tree is the *child* of an existing node.



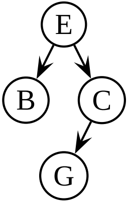
Basic Botany

- In this talk we mainly deal with *rooted, labelled* trees.
- There is a *root vertex*.
- Each other node that we add to the tree is the *child* of an existing node.



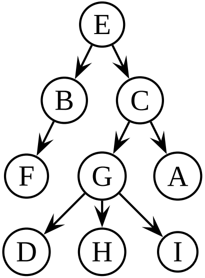
Basic Botany

- In this talk we mainly deal with *rooted, labelled* trees.
- There is a *root vertex*.
- Each other node that we add to the tree is the *child* of an existing node.



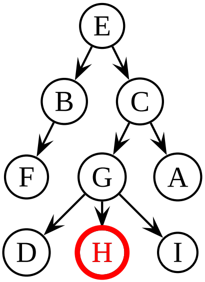
Basic Botany

- In this talk we mainly deal with *rooted, labelled* trees.
- There is a *root vertex*.
- Each other node that we add to the tree is the *child* of an existing node.
- If node x is a child of node y then we say that y is the parent of x. Each non-root node has exactly one parent.



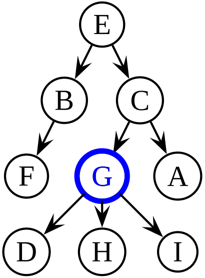
Basic Botany

- In this talk we mainly deal with *rooted*, *labelled* trees.
- There is a *root vertex*.
- Each other node that we add to the tree is the *child* of an existing node.
- If node x is a child of node y then we say that y is the parent of x. Each non-root node has exactly one parent.
- For example, since H



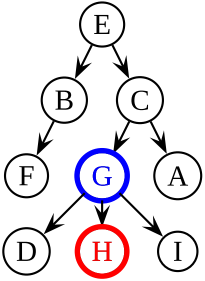
Basic Botany

- In this talk we mainly deal with *rooted, labelled* trees.
- There is a *root vertex*.
- Each other node that we add to the tree is the *child* of an existing node.
- If node x is a child of node y then we say that y is the parent of x. Each non-root node has exactly one parent.
- For example, since H is a child of G,



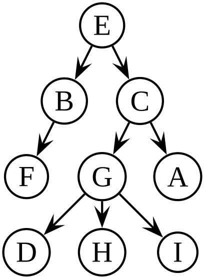
Basic Botany

- In this talk we mainly deal with *rooted, labelled* trees.
- There is a *root vertex*.
- Each other node that we add to the tree is the *child* of an existing node.
- If node x is a child of node y then we say that y is the parent of x. Each non-root node has exactly one parent.
- For example, since H is a child of G, node G is the parent of H.



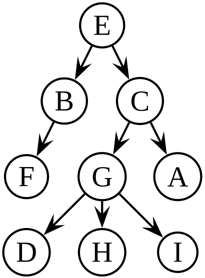
Basic Botany

- A node with no children is called a *leaf*.
- A node that is not a leaf is called an *internal node*.
- In this tree the leaves are F, D, H, I, A.



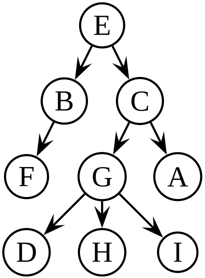
Basic Botany

- A node with no children is called a *leaf*.
- A node that is not a leaf is called an *internal node*.
- In this tree the leaves are F, D, H, I, A.
- In contest problems, explicitly given trees often model:



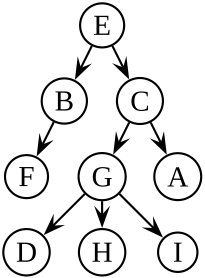
Basic Botany

- A node with no children is called a *leaf*.
- A node that is not a leaf is called an *internal node*.
- In this tree the leaves are F, D, H, I, A.
- In contest problems, explicitly given trees often model:
- a work hierarchy (nodes = people; parent = boss, child = subordinate)



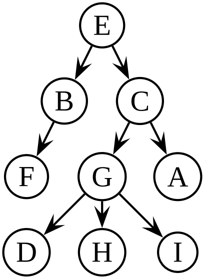
Basic Botany

- A node with no children is called a *leaf*.
- A node that is not a leaf is called an *internal node*.
- In this tree the leaves are F, D, H, I, A.
- In contest problems, explicitly given trees often model:
- a work hierarchy (nodes = people; parent = boss, child = subordinate)
- an expression (leaves = values, internal nodes = functions)



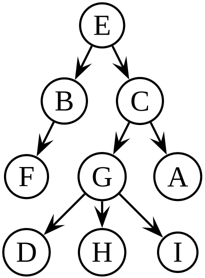
Basic Botany

- A node with no children is called a *leaf*.
- A node that is not a leaf is called an *internal node*.
- In this tree the leaves are F, D, H, I, A.
- In contest problems, explicitly given trees often model:
- a work hierarchy (nodes = people; parent = boss, child = subordinate)
- an expression (leaves = values, internal nodes = functions)
- states of a game (nodes = board positions, root = initial board, edges = valid moves, leaves = ending positions)



Basic Botany

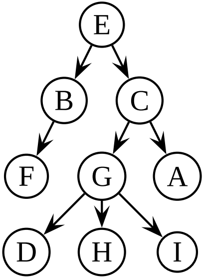
- A node with no children is called a *leaf*.
- A node that is not a leaf is called an *internal node*.
- In this tree the leaves are F, D, H, I, A.
- In contest problems, explicitly given trees often model:
- a work hierarchy (nodes = people; parent = boss, child = subordinate)
- an expression (leaves = values, internal nodes = functions)
- states of a game (nodes = board positions, root = initial board, edges = valid moves, leaves = ending positions)
- occasionally, a tree (leaves = leaves, root = root)



Basic Botany

- Straightforward representation: keep an array P of the nodes' parents and an array C of child-lists.

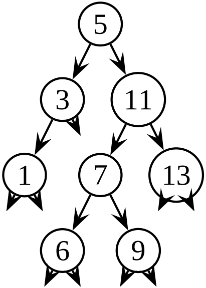
x	$P[x]$	$C[x]$
A	C	()
B	E	(F)
C	E	(G,A)
D	G	()
E	`nil`	(B,C)
F	B	()
G	C	(D,H,I)
H	G	()
I	G	()



- If we don't care about (or don't know) the order of each node's children then we may only need to keep track of P.
- Alternatively, we can just keep track of C.

Aside: Binary Trees

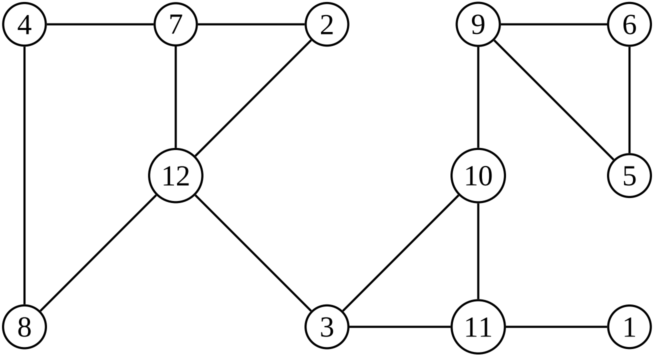
- Another form of tree is a *binary tree*.
- Each node may or may not have a left child, and may or may not have a right child.
- Each node is a record with fields (value, left, right), where left and right are pointers to nodes. A null pointer means that that child doesn't exist.
- If we stick values in the nodes the right way, we can make a *binary search tree* which is useful for some applications.
- Different generalization: k child positions is a *k-ary tree*.



Outline

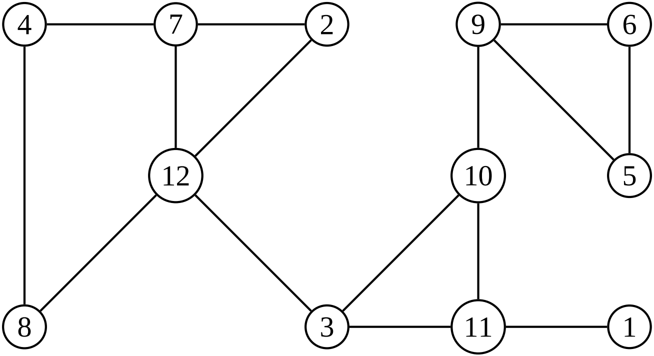
Definitions

- A *graph* $G = (V, E)$ is a set V of n nodes (which we call $1, \ldots, n$) together with a collection E of *edges*. Each edge is just a pair of nodes.
- E.g., nodes/edges = cities/roads or computers/links.



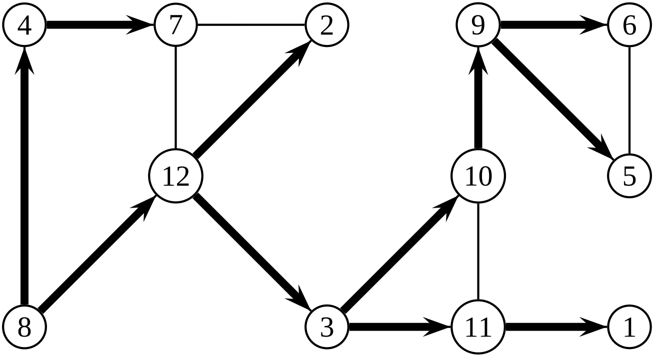
Definitions

- A *graph* $G = (V, E)$ is a set V of n nodes (which we call $1, \ldots, n$) together with a collection E of *edges*. Each edge is just a pair of nodes.
- E.g., nodes/edges = cities/roads or computers/links.
- A *spanning tree* is a tree that contains every node.
- Here's a spanning tree with root 8:



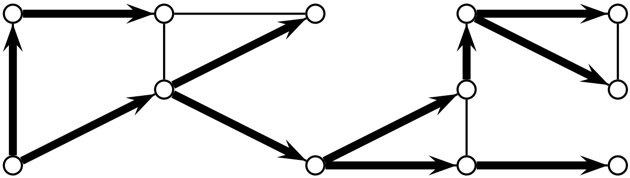
Definitions

- A *graph* $G = (V, E)$ is a set V of n nodes (which we call $1, \ldots, n$) together with a collection E of *edges*. Each edge is just a pair of nodes.
- E.g., nodes/edges = cities/roads or computers/links.
- A *spanning tree* is a tree that contains every node.
- Here's a spanning tree with root 8:



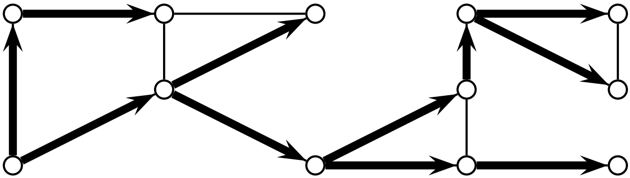
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.



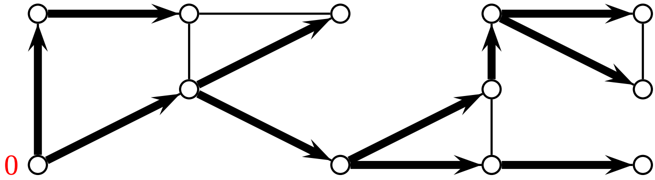
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:



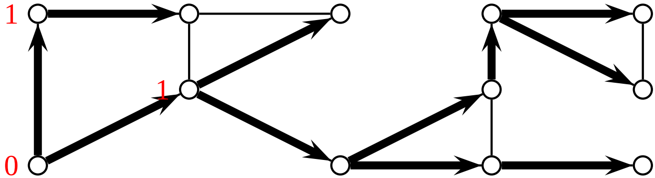
Spanning Tree $\Rightarrow$ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:



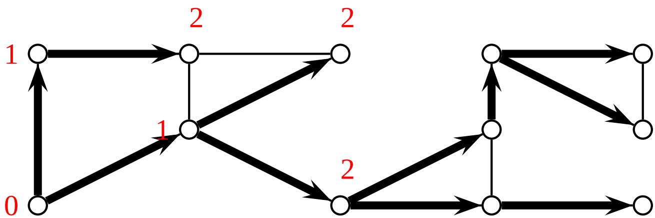
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:



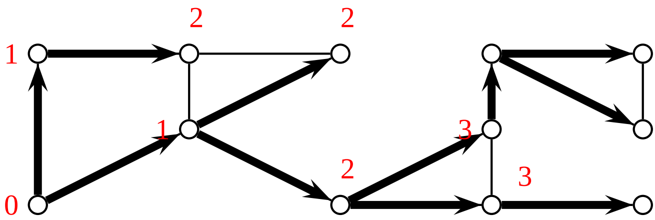
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:



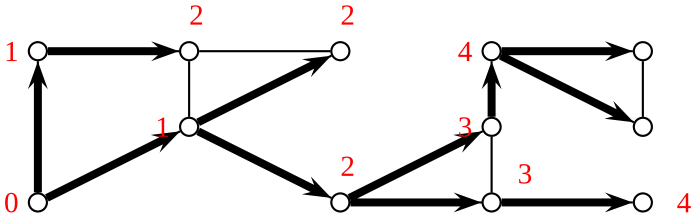
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:



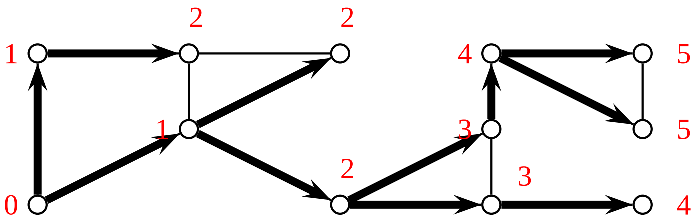
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:



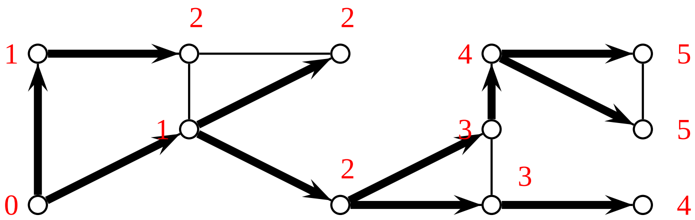
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:
- Definition: a graph is *bipartite* if the nodes can be colored green and blue so that each there are no green-green or blue-blue edges.



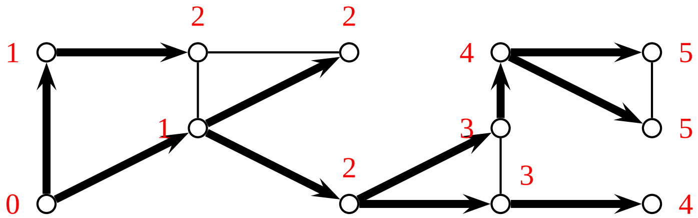
Spanning Tree ⇒ Check Bipartite

- The *level* of a node in any tree is the number of tree edges between that node and the root. I.e., level(root)=0 and level(x)=level($P[x]$)+1.
- The spanning tree pictured has these levels:
- Definition: a graph is *bipartite* if the nodes can be colored green and blue so that each there are no green-green or blue-blue edges.
- You can show that a graph is bipartite if and only if for each non-tree edge $\{u, v\}$ we have $\text{level}(u) \not\equiv \text{level}(v) \pmod 2$.



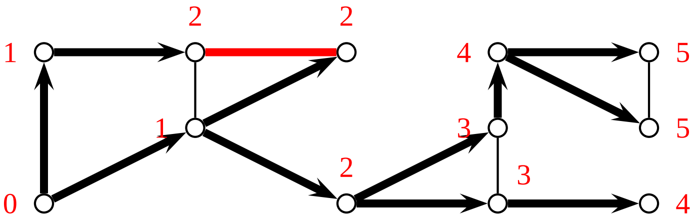
Spanning Tree ⇒ Check Bipartite

- Last slide: "You can show that a graph is bipartite if and only if for each non-tree edge $\{u, v\}$ we have $\text{level}(u) \not\equiv \text{level}(v) \pmod 2$".



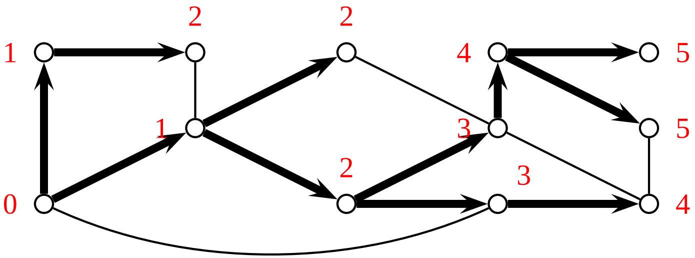
Spanning Tree ⇒ Check Bipartite

- Last slide: "You can show that a graph is bipartite if and only if for each non-tree edge $\{u, v\}$ we have $\text{level}(u) \not\equiv \text{level}(v) \pmod 2$".
- Because of the edge pictured (among others) we know G is not bipartite.



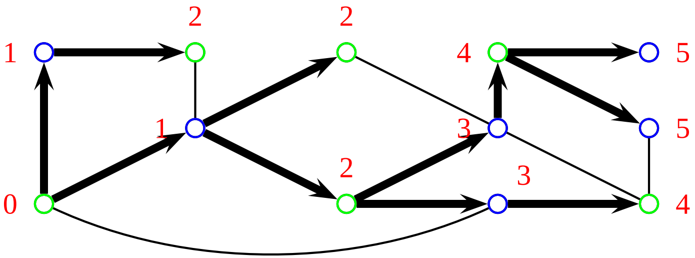
Spanning Tree ⇒ Check Bipartite

- Last slide: "You can show that a graph is bipartite if and only if for each non-tree edge $\{u, v\}$ we have $\text{level}(u) \not\equiv \text{level}(v) \pmod 2$".
- Because of the edge pictured (among others) we know G is not bipartite.
- But this other graph (with the same spanning tree) *is* bipartite.



Spanning Tree ⇒ Check Bipartite

- Last slide: "You can show that a graph is bipartite if and only if for each non-tree edge $\{u, v\}$ we have $\text{level}(u) \not\equiv \text{level}(v) \pmod 2$".
- Because of the edge pictured (among others) we know *G* is not bipartite.
- But this other graph (with the same spanning tree) *is* bipartite.
- We color the even-level nodes green and the odd-level nodes blue.



Outline

Three Important Data Structures

- Data structures allow you to *push* (insert) and *pop* (remove) items.

Three Important Data Structures

- Data structures allow you to *push* (insert) and *pop* (remove) items.
- A *stack* is a LIFO (last-in, first-out) data structure.
- When we *pop*, the newest item in the stack is returned & removed.
- E.g. push A, then push B. Then a pop returns B. If we push C and then pop again we get C, and another pop finally retrieves A.

Three Important Data Structures

- Data structures allow you to *push* (insert) and *pop* (remove) items.
- A *stack* is a LIFO (last-in, first-out) data structure.
- When we *pop*, the newest item in the stack is returned & removed.
- E.g. push A, then push B. Then a pop returns B. If we push C and then pop again we get C, and another pop finally retrieves A.
- A *queue* is a FIFO (first-in, first-out) data structure.
- Popping removes & returns the oldest remaining item.
- E.g. push A, then push B. Then a pop returns A. If we push C and then pop again we get B, and another pop will return C.
- Note: for a queue, order of removal = order of insertion.

Three Important Data Structures

- Data structures allow you to *push* (insert) and *pop* (remove) items.
- A *stack* is a LIFO (last-in, first-out) data structure.
- When we *pop*, the newest item in the stack is returned & removed.
- E.g. push A, then push B. Then a pop returns B. If we push C and then pop again we get C, and another pop finally retrieves A.
- A *queue* is a FIFO (first-in, first-out) data structure.
- Popping removes & returns the oldest remaining item.
- E.g. push A, then push B. Then a pop returns A. If we push C and then pop again we get B, and another pop will return C.
- Note: for a queue, order of removal = order of insertion.
- A *priority queue* is a cheapest-out structure.
- Each item is inserted with a fixed numerical priority.
- Popping returns & removes the least-priority remaining item.
- E.g. push(A, 2) then push(B, 3). Pop returns A. If we push(C, 1) and then pop again we get C, and another pop will return B.

Three Important Data Structures

- Data structures allow you to *push* (insert) and *pop* (remove) items.
- A *stack* is a LIFO (last-in, first-out) data structure.
- When we *pop*, the newest item in the stack is returned & removed.
- E.g. push A, then push B. Then a pop returns B. If we push C and then pop again we get C, and another pop finally retrieves A.
- A *queue* is a FIFO (first-in, first-out) data structure.
- Popping removes & returns the oldest remaining item.
- E.g. push A, then push B. Then a pop returns A. If we push C and then pop again we get B, and another pop will return C.
- Note: for a queue, order of removal = order of insertion.
- A *priority queue* is a cheapest-out structure.
- Each item is inserted with a fixed numerical priority.
- Popping returns & removes the least-priority remaining item.
- E.g. push(A, 2) then push(B, 3). Pop returns A. If we push(C, 1) and then pop again we get C, and another pop will return B.
- Implement priority queue with a *heap* or *balanced binary tree*.

Exploring a Graph

- Here is an abstract algorithm for exploring a graph.

```
procedure SEARCH-GRAPH(G, root)
    isExplored := boolean[vertices of G], initially false
    waitingEdges := struct⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that !isExplored[w] do
                waitingEdges.add((v, w))
```

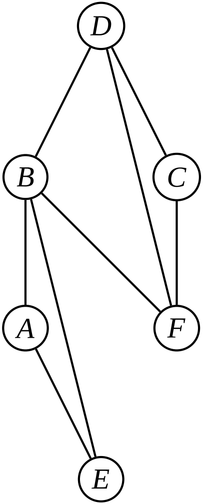
- Basically we try to explore every edge that we learn about.
- No matter what order edges are removed from waitingEdges, we get a spanning tree.

Outline

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

procedure DEPTH-FIRST-SEARCH(G, root)
  isExplored := boolean[vertices of G]
  waitingEdges := stack$\langle$ pair$\langle$ vertex $\rangle\rangle$
  waitingEdges.add((`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p
      for all neighbours w of v such that
        !isExplored[w] **do**
        waitingEdges.add((v, w))

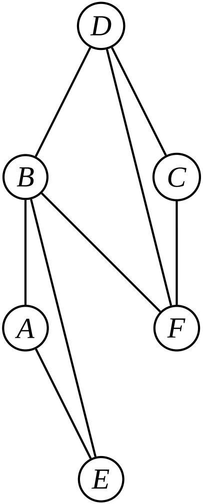


[Top] [Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((`nil`, root))
    while waitingEdges is not empty **do**
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) **then**
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] **do**
                waitingEdges.add((v, w))

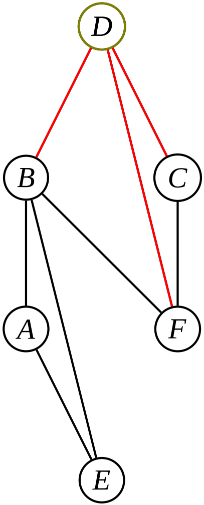


[Top] (`nil`, D)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```

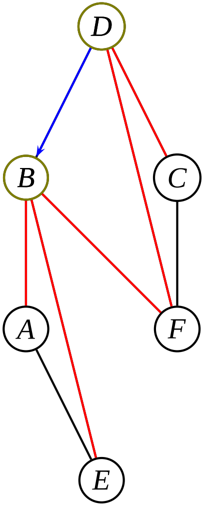


[Top] (D, B)(D, F)(D, C)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search*.
- Stack is LIFO (last in, first out).

```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```

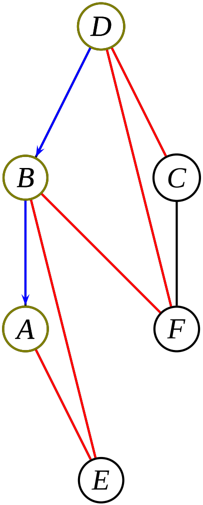


[Top] (B, A)(B, E)(B, F)(D, F)(D, C)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add((v, w))
```

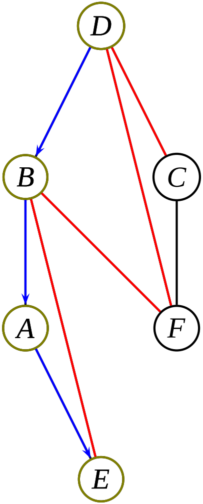


[Top] (A, E)(B, E)(B, F)(D, F)(D, C)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search*.
- Stack is LIFO (last in, first out).

```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```

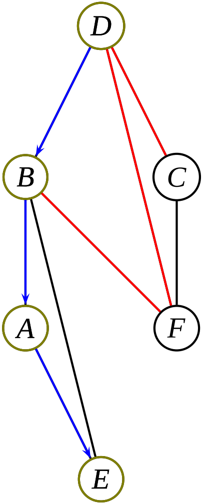


[Top] (B, E)(B, F)(D, F)(D, C)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while  waitingEdges is not empty  do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```

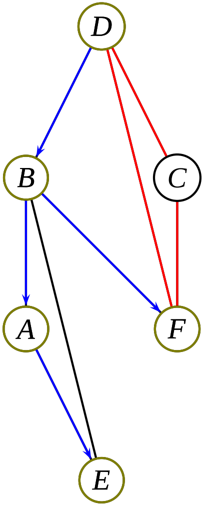


[Top] (B, F)(D, F)(D, C)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                  !isExplored[w] do
                waitingEdges.add((v, w))
```

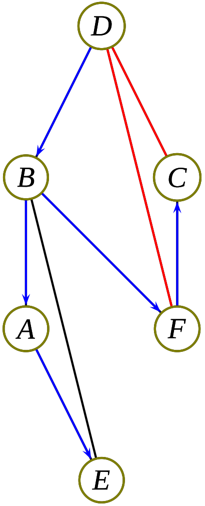


[Top] (F, C)(D, F)(D, C)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add((v, w))
```

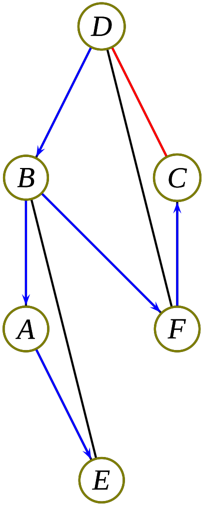


[Top] (D, F)(D, C)[Bot]

Depth-First Search

- Make waitingEdges a stack: *depth-first search.*
- Stack is LIFO (last in, first out).

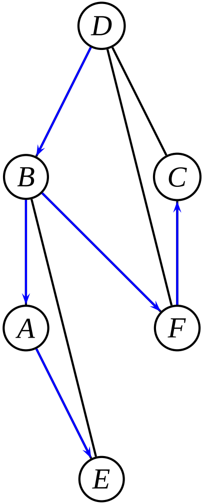
```
procedure DEPTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := stack⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```



[Top] (D, C)[Bot]

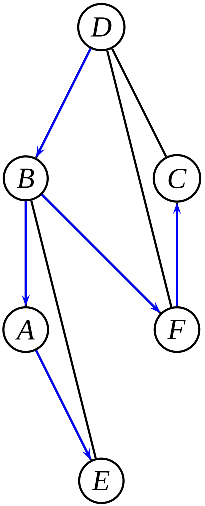
Properties of Depth-First Search

- *Complexity*
- Each edge enters and leaves the stack exactly once.
- So $O(m+n)$ time complexity where $m = |E|, n = |V|$



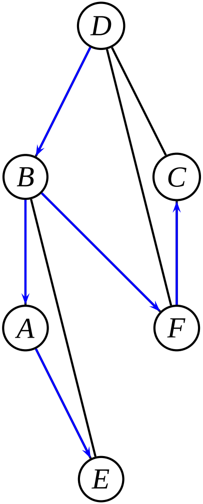
Properties of Depth-First Search

- *Complexity*
- Each edge enters and leaves the stack exactly once.
- So $O(m+n)$ time complexity where $m = |E|, n = |V|$
- *Properties*
- For each non-tree edge uv, either u is a descendant of v in the DFS tree or vice-versa. (*No cross edges*)



Properties of Depth-First Search

- *Complexity*
- Each edge enters and leaves the stack exactly once.
- So $O(m+n)$ time complexity where $m = |E|, n = |V|$
- *Properties*
- For each non-tree edge uv, either u is a descendant of v in the DFS tree or vice-versa. (*No cross edges*)
- *Applications*
- We will see later that using DFS and some other ideas (preorder, postorder) we can get efficient algorithms for biconnectivity and strong connectivity.

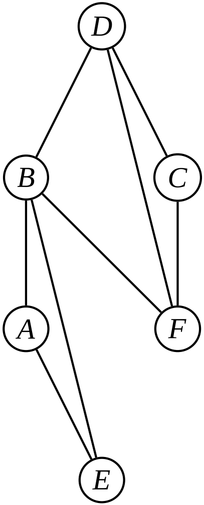


Outline

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add((v, w))
```

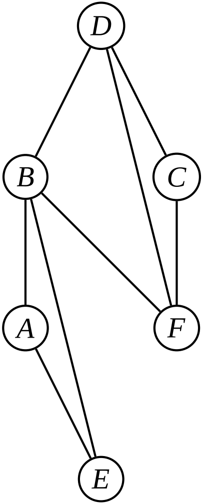


[Head] [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

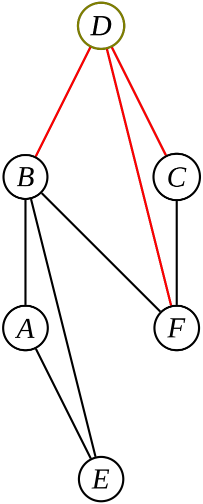
```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```



[Head] (nil, D) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

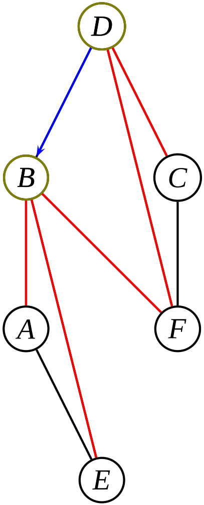



```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```

[Head] (D, B)(D, F)(D, C) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).



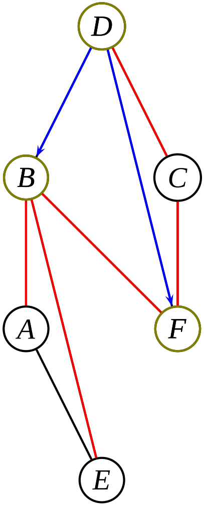

```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while  waitingEdges is not empty  do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add((v, w))
```

[Head] (D, F)(D, C)(B, A)(B, E)(B, F) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```

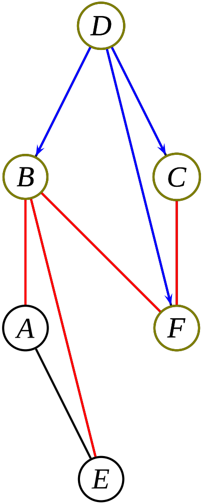


[Head] (D, C)(B, A)(B, E)(B, F)(F, C) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add((v, w))
```

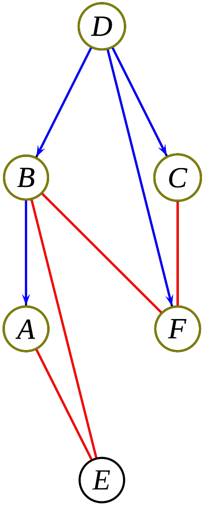


[Head] (B, A)(B, E)(B, F)(F, C) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add((v, w))
```

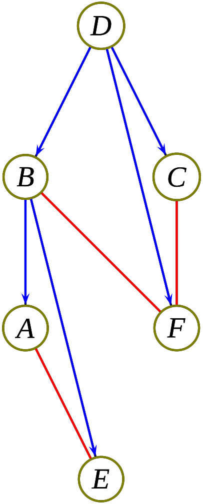


[Head] (B, E)(B, F)(F, C)(A, E) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

procedure BREADTH-FIRST-SEARCH(G, root)
  isExplored := boolean[vertices of G]
  waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
  waitingEdges.add((`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p
      for all neighbours w of v such that !isExplored[w] **do**
        waitingEdges.add((v, w))

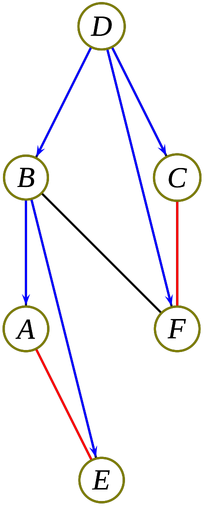


[Head] (B, F)(F, C)(A, E) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add((v, w))
```

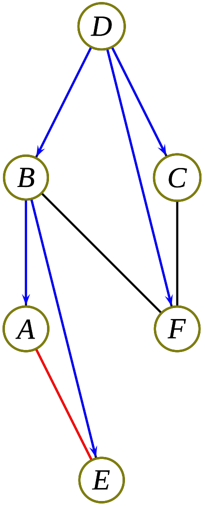


[Head] (F, C)(A, E) [Tail]

Breadth-First Search

- Make waitingEdges a queue: *breath-first search.*
- Queue is FIFO (first in, first out).

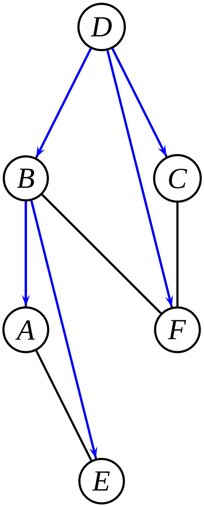
```
procedure BREADTH-FIRST-SEARCH(G, root)
    isExplored := boolean[vertices of G]
    waitingEdges := queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add((nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add((v, w))
```



[Head] (A, E) [Tail]

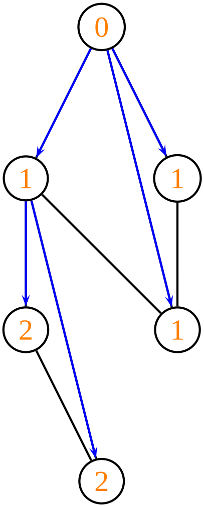
Properties of Breadth-First Search

- *Complexity*: $O(m+n)$ time.



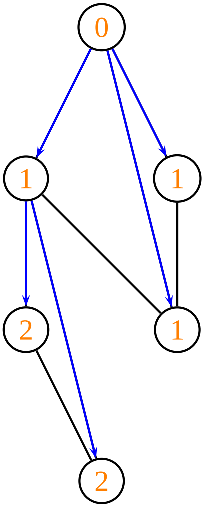
Properties of Breadth-First Search

- *Complexity*: $O(m+n)$ time.
- *Properties*
- level$[v]$ = dist(root, v) (shortest paths!)
- Edge uv not in tree $\Rightarrow$ $|$level$[u]$-level$[v]| \leq 1$



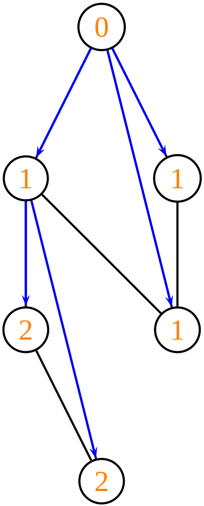
Properties of Breadth-First Search

- *Complexity*: $O(m+n)$ time.
- *Properties*
- level[v] = dist(root, v) (shortest paths!)
- Edge uv not in tree $\Rightarrow$ $|$level[u]-level[v]$| \leq 1$
- *An Application*
- *Girth* g : length of the shortest cycle.
- There is a length-g cycle through the root if and only if some non-tree edge uv satisfies level[u]+level[v]+1=g.
- So to compute g : do a BFS from each vertex and return the minimum value of level[u]+level[v]+1 over all non-tree edges uv in all trees.



Properties of Breadth-First Search

- *Complexity*: $O(m+n)$ time.
- *Properties*
- level[v] = dist(root, v) (shortest paths!)
- Edge uv not in tree $\Rightarrow$ $|$level[u]-level[v]$| \leq 1$
- *An Application*
- *Girth* g : length of the shortest cycle.
- There is a length-g cycle through the root if and only if some non-tree edge uv satisfies level[u]+level[v]+1=g.
- So to compute g : do a BFS from each vertex and return the minimum value of level[u]+level[v]+1 over all non-tree edges uv in all trees.
- No known fast algorithm for determining the *longest* cycle.



Interlude

- What if the graph we are given is not connected?

Interlude

- What if the graph we are given is not connected?
- Then Search(G, root) will hit only those nodes that have some path to root. We call these nodes *connected* to root.
- The set of all nodes reachable from root is a *connected component*. The vertices of every graph are naturally partitioned into connected components.

Interlude

- What if the graph we are given is not connected?
- Then Search(G, root) will hit only those nodes that have some path to root. We call these nodes *connected* to root.
- The set of all nodes reachable from root is a *connected component*. The vertices of every graph are naturally partitioned into connected components.
- Here's pseudocode for connected components. Search-and-Label(G, root) is any kind of search routine, but when it explores a node w it sets connected-component-label[w] := root.

```
procedure CONNECTED-COMPONENTS(G)
    isExplored := boolean[v]        ▷ Assume vertices are 0, ..., v − 1
    for i := 1 to v − 1 do
        if !isExplored[i] then
            Search-and-Label(G, i)
```

Interlude

- What if the graph we are given is not connected?
- Then Search(G, root) will hit only those nodes that have some path to root. We call these nodes *connected* to root.
- The set of all nodes reachable from root is a *connected component*. The vertices of every graph are naturally partitioned into connected components.
- Here's pseudocode for connected components. Search-and-Label(G, root) is any kind of search routine, but when it explores a node w it sets connected-component-label[w] := root.

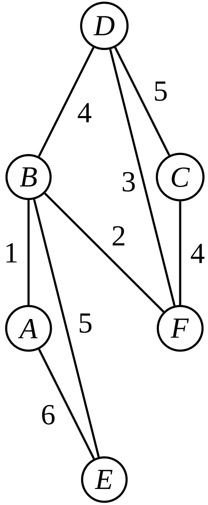
```
procedure CONNECTED-COMPONENTS(G)
    isExplored := boolean[v]          ▷ Assume vertices are 0, . . . , v − 1
    for i := 1 to v − 1 do
        if !isExplored[i] then
            Search-and-Label(G, i)
```

- Computes a *spanning forest* of G.

Outline

Minimum Spanning Tree

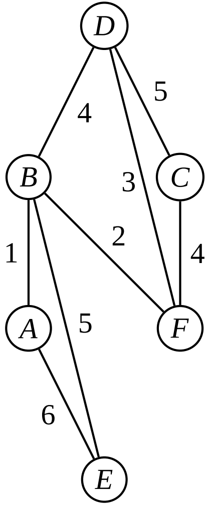
- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

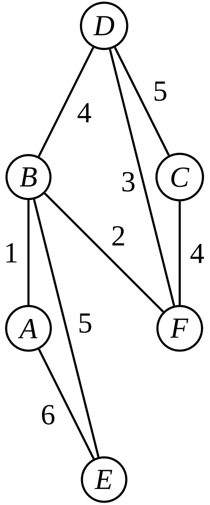
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (`nil`, root))
    while waitingEdges is not empty **do**
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) **then**
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that !isExplored[w] **do**
                waitingEdges.add($c[v, w]$, (v, w))



Minimum Spanning Tree

- Each edge uv is given a cost $c[u,v] = c[v,u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u,v]$.

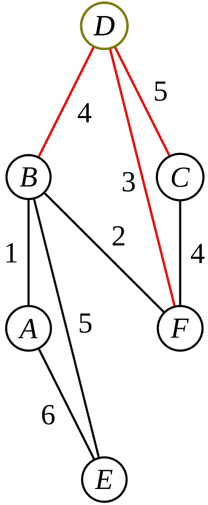
```
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add(c[v, w], (v, w))
```



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

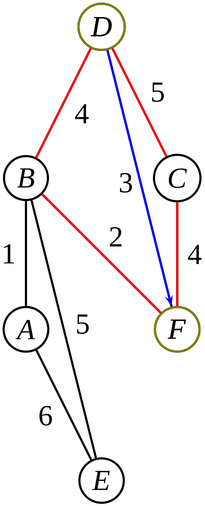
```
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add(c[v, w], (v, w))
```



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

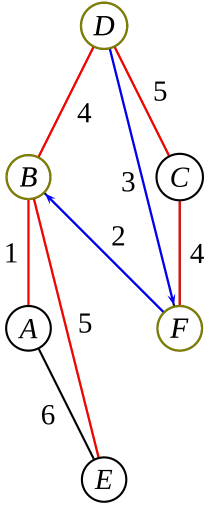
```
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                    waitingEdges.add(c[v, w], (v, w))
```



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

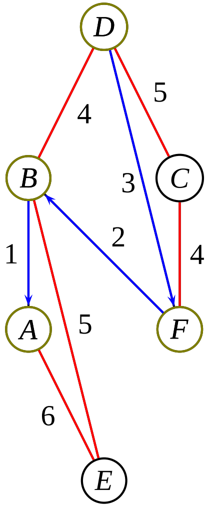
procedure MINIMUM-SPANNING-TREE(G, root)
  waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
  waitingEdges.add(0, (`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p
      for all neighbours w of v such that !isExplored[w] **do**
        waitingEdges.add($c[v, w]$, (v, w))



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

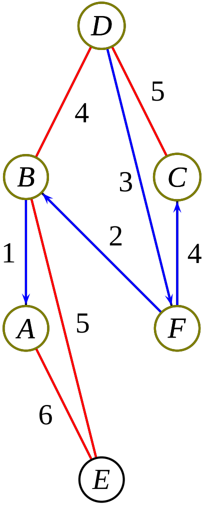
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (`nil`, root))
    while waitingEdges is not empty **do**
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) **then**
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that !isExplored[w] **do**
                waitingEdges.add($c[v, w]$, (v, w))



Minimum Spanning Tree

- Each edge uv is given a cost $c[u,v] = c[v,u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u,v]$.

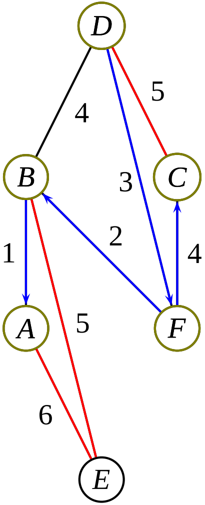
```
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                    waitingEdges.add(c[v, w], (v, w))
```



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

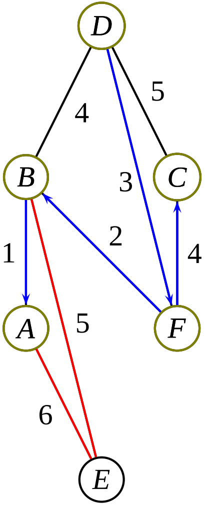
```
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add(c[v, w], (v, w))
```



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

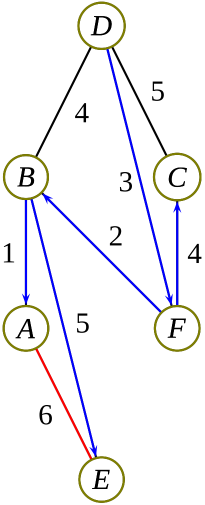
```
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                waitingEdges.add(c[v, w], (v, w))
```



Minimum Spanning Tree

- Each edge uv is given a cost $c[u, v] = c[v, u]$.
- What spanning tree has minimal sum of edge costs?
- Use a priority queue, priority(uv) = $c[u, v]$.

```
procedure MINIMUM-SPANNING-TREE(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p
            for all neighbours w of v such that
                !isExplored[w] do
                    waitingEdges.add(c[v, w], (v, w))
```



Minimum Spanning Tree

- Intuitively, we are “growing” a spanning tree starting from the specified root.
- The priority queue always contains all edges that go *from* the current tree *to* some non-tree vertex.
- (In the priority queue there will additionally be some edges that go between 2 tree vertices, but they will be skipped)
- We always grow the tree in the cheapest way possible!

Minimum Spanning Tree

- Intuitively, we are "growing" a spanning tree starting from the specified root.
- The priority queue always contains all edges that go *from* the current tree *to* some non-tree vertex.
- (In the priority queue there will additionally be some edges that go between 2 tree vertices, but they will be skipped)
- We always grow the tree in the cheapest way possible!
- *Complexity*
- Each edge enters and leaves the priority queue exactly once.
- Priority queues generally have $O(\log n)$ time complexity per access so total time complexity is $O(m \log n)$

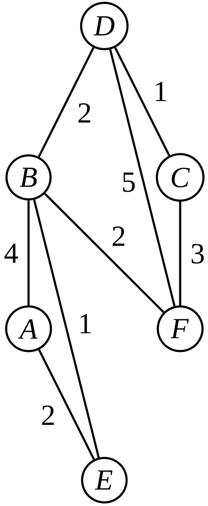
Minimum Spanning Tree

- Intuitively, we are "growing" a spanning tree starting from the specified root.
- The priority queue always contains all edges that go *from* the current tree *to* some non-tree vertex.
- (In the priority queue there will additionally be some edges that go between 2 tree vertices, but they will be skipped)
- We always grow the tree in the cheapest way possible!
- *Complexity*
- Each edge enters and leaves the priority queue exactly once.
- Priority queues generally have $O(\log n)$ time complexity per access so total time complexity is $O(m \log n)$
- Also known as Prim's algorithm.
- Can be implemented somewhat faster, in $O(m + n \log n)$ time.

Outline

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.

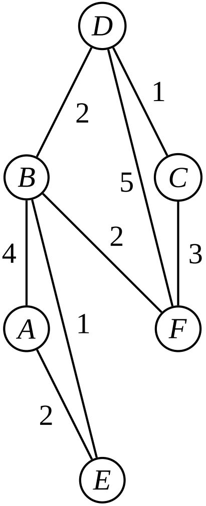


[Top]

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] $+ c[u, v]$.

```
procedure SHORTEST-PATHS(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p, d[v] := d[p] + c[p, v]
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add(d[v] + c[v, w], (v, w))
```

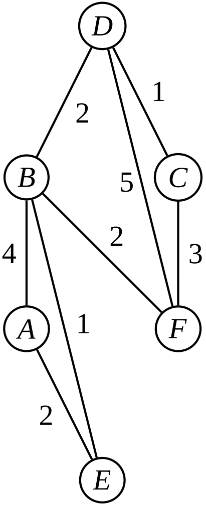


[Top]

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] $+ c[u, v]$.

```
procedure SHORTEST-PATHS(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p, d[v] := d[p] + c[p, v]
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add(d[v] + c[v, w], (v, w))
```

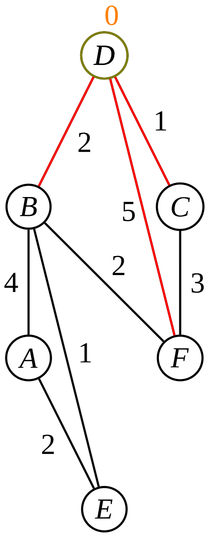


[Top] (nil, D)^0

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] $+\, c[u, v]$.

```
procedure SHORTEST-PATHS(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while  waitingEdges is not empty  do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p, d[v] := d[p] + c[p, v]
            for all neighbours w of v such that
                    !isExplored[w] do
              waitingEdges.add(d[v] + c[v, w], (v, w))
```

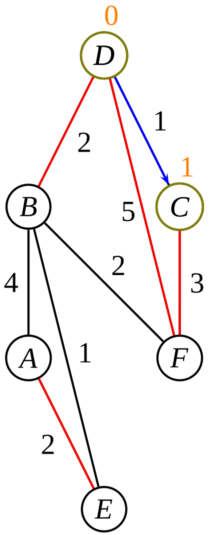


[Top] (D, C)^1(D, B)^2(D, F)^5

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] + $c[u, v]$.

```
procedure SHORTEST-PATHS(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p, d[v] := d[p] + c[p, v]
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add(d[v] + c[v, w], (v, w))
```

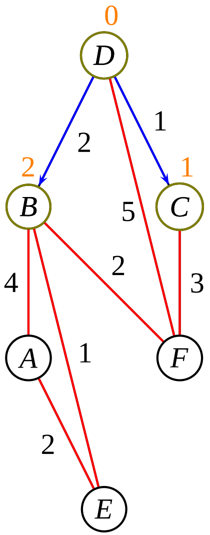


[Top] (D, B)^2(C, F)^4(D, F)^5(E, A)^5

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] + $c[u, v]$.

procedure SHORTEST-PATHS(G, root)
  waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
  waitingEdges.add(0, (`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p, d[v] := d[p] + $c[p, v]$
      for all neighbours w of v such that !isExplored[w] **do**
        waitingEdges.add($d[v] + c[v, w]$, (v, w))

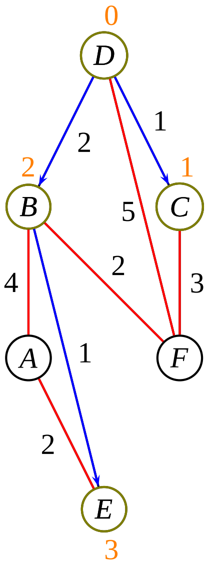


[Top] (B, E)^3(C, F)^4(B, F)^4(D, F)^5(E, A)^5(B, A)^6

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] $+ c[u, v]$.

procedure SHORTEST-PATHS(G, root)
  waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
  waitingEdges.add(0, (`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p, d[v] := d[p] + $c[p, v]$
      for all neighbours w of v such that !isExplored[w] **do**
        waitingEdges.add($d[v] + c[v, w]$, (v, w))

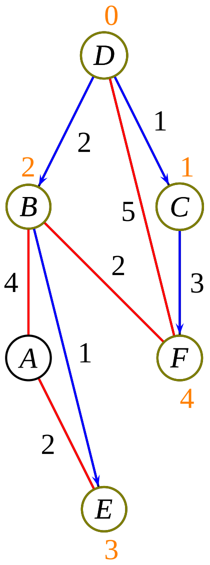


[Top] (C, F)^4(B, F)^4(D, F)^5(E, A)^5(B, A)^6

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] + $c[u, v]$.

procedure SHORTEST-PATHS(G, root)
  waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
  waitingEdges.add(0, (`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p, d[v] := d[p] + $c[p, v]$
      for all neighbours w of v such that !isExplored[w] **do**
        waitingEdges.add($d[v] + c[v, w]$, (v, w))

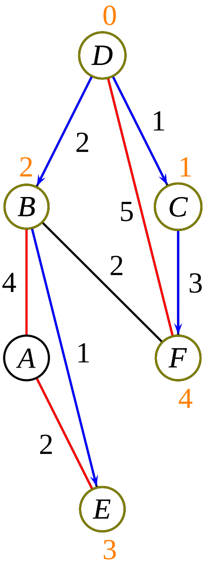


[Top] (B, F)^{4}(D, F)^{5}(E, A)^{5}(B, A)^{6}

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] + $c[u, v]$.

procedure SHORTEST-PATHS(G, root)
  waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
  waitingEdges.add(0, (`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p, d[v] := d[p] + $c[p, v]$
      for all neighbours w of v such that !isExplored[w] **do**
        waitingEdges.add($d[v] + c[v, w]$, (v, w))

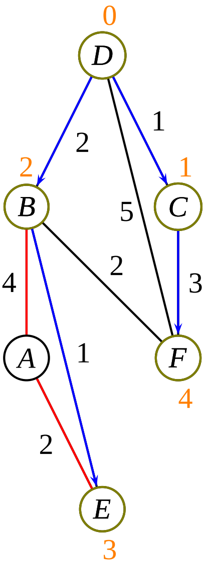


[Top] (D, F)^5(E, A)^5(B, A)^6

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] + $c[u, v]$.

procedure SHORTEST-PATHS(G, root)
  waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
  waitingEdges.add(0, (`nil`, root))
  while waitingEdges is not empty **do**
    (p, v) := waitingEdges.remove()
    if (!isExplored[v]) **then**
      isExplored[v] := true
      parent[v] := p, d[v] := d[p] + $c[p, v]$
      for all neighbours w of v such that !isExplored[w] **do**
        waitingEdges.add($d[v] + c[v, w]$, (v, w))

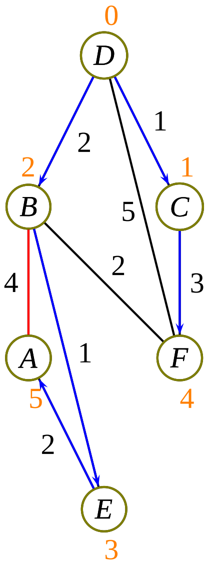


[Top] (E, A)^{5}(B, A)^{6}

Single-Source Weighted Shortest Paths

- Find the shortest distance d[x] from root to each vertex x in a *weighted* graph.
- Use a priority queue, priority(uv) = d[u] $+ c[u, v]$.

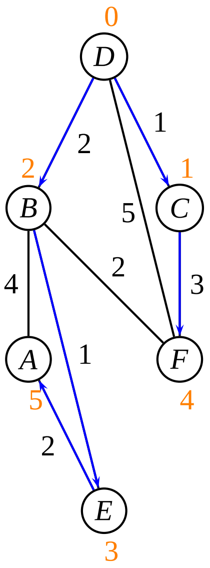
```
procedure SHORTEST-PATHS(G, root)
    waitingEdges := pri-queue⟨ pair⟨ vertex ⟩⟩
    waitingEdges.add(0, (nil, root))
    while waitingEdges is not empty do
        (p, v) := waitingEdges.remove()
        if (!isExplored[v]) then
            isExplored[v] := true
            parent[v] := p, d[v] := d[p] + c[p, v]
            for all neighbours w of v such that
                    !isExplored[w] do
                waitingEdges.add(d[v] + c[v, w], (v, w))
```



[Top] (B, A)[6]

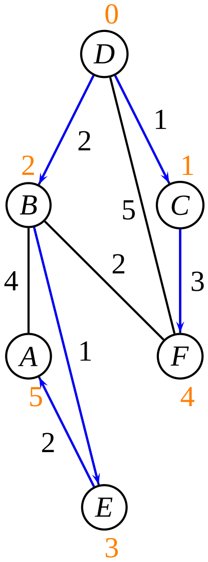
Single-Source Weighted Shortest Paths

- Is it somewhat surprising that the shortest paths can be even be *represented* by a tree?



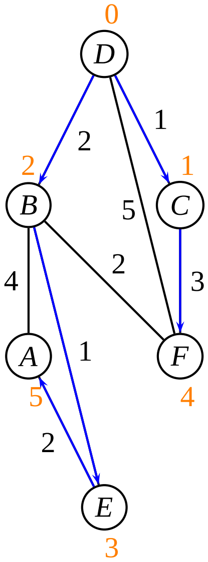
Single-Source Weighted Shortest Paths

- Is it somewhat surprising that the shortest paths can be even be *represented* by a tree?
- Basic reason: if the shortest path from root to x needs to go through y first, then it needs to actually take a shortest path to y.
- Nodes are discovered in increasing order of distance from root.



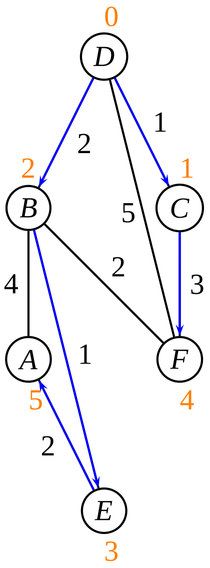
Single-Source Weighted Shortest Paths

- Is it somewhat surprising that the shortest paths can be even be *represented* by a tree?
- Basic reason: if the shortest path from root to x needs to go through y first, then it needs to actually take a shortest path to y.
- Nodes are discovered in increasing order of distance from root.
- Side note: $\mathrm{d}[u] \leq \mathrm{d}[v] + c[u, v]$ for all edges uv.



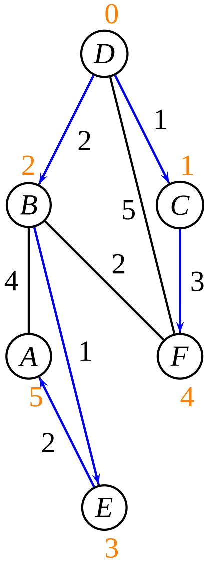
Single-Source Weighted Shortest Paths

- Is it somewhat surprising that the shortest paths can be even be *represented* by a tree?
- Basic reason: if the shortest path from root to x needs to go through y first, then it needs to actually take a shortest path to y.
- Nodes are discovered in increasing order of distance from root.
- Side note: $\mathrm{d}[u] \leq \mathrm{d}[v] + c[u,v]$ for all edges uv.
- *Complexity*
- Takes $O(m \log n)$ time, like Prim's MST algorithm.



Single-Source Weighted Shortest Paths

- Is it somewhat surprising that the shortest paths can be even be *represented* by a tree?
- Basic reason: if the shortest path from root to x needs to go through y first, then it needs to actually take a shortest path to y.
- Nodes are discovered in increasing order of distance from root.
- Side note: $\mathrm{d}[u] \leq \mathrm{d}[v] + c[u, v]$ for all edges uv.
- *Complexity*
- Takes $O(m \log n)$ time, like Prim's MST algorithm.
- Works for directed graphs. *Doesn't work* if there are negative edge weights.



Outline

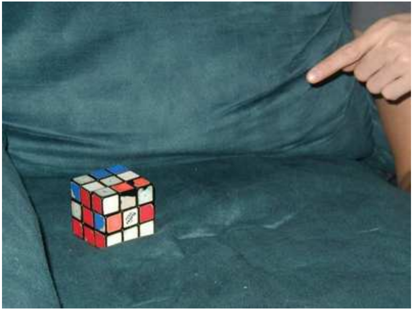
Outline

Generalized Searching

- What if we have a graph that is given *implicitly*?

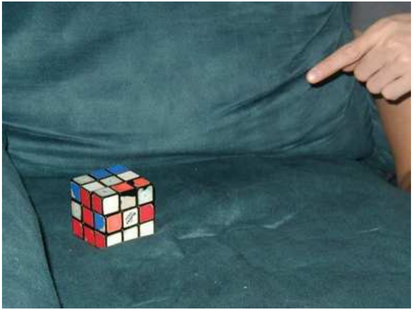
Generalized Searching

- What if we have a graph that is given *implicitly*?
- Example: Nodes are *states* of Rubik's cube, edges are valid moves.
- Want to solve the cube quickly $\Leftrightarrow$ shortest path.



Generalized Searching

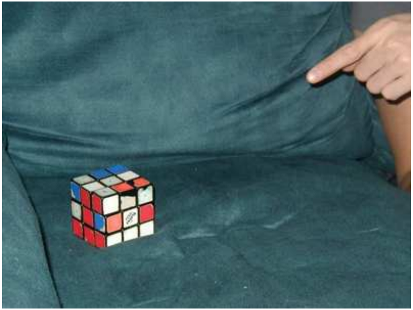
- What if we have a graph that is given *implicitly*?
- Example: Nodes are *states* of Rubik's cube, edges are valid moves.
- Want to solve the cube quickly $\Leftrightarrow$ shortest path.



- Can we use BFS without actually constructing the whole graph?

Generalized Searching

- What if we have a graph that is given *implicitly*?
- Example: Nodes are *states* of Rubik's cube, edges are valid moves.
- Want to solve the cube quickly $\Leftrightarrow$ shortest path.



- Can we use BFS without actually constructing the whole graph?
- Sure. It is trickier to keep track of isExplored; you can use a Set class like a *hash set* or a *sorted set / balanced binary tree*.

Generalized Searching

- Summary of last slide: to find path from x to y do a BFS from x, stopping when we hit y.
- The "implicit graph" idea is used a lot in AI: planning driving routes, automatic theorem proving, operations research.

Generalized Searching

- Summary of last slide: to find path from x to y do a BFS from x, stopping when we hit y.
- The “implicit graph” idea is used a lot in AI: planning driving routes, automatic theorem proving, operations research.
- A caveat. If there is no path from x to y, then our BFS will explore the whole graph anyway, which is inefficient!
- 43,252,003,274,489,856,000 positions for a Rubik’s cube.

Generalized Searching

- Summary of last slide: to find path from x to y do a BFS from x, stopping when we hit y.
- The "implicit graph" idea is used a lot in AI: planning driving routes, automatic theorem proving, operations research.
- A caveat. If there is no path from x to y, then our BFS will explore the whole graph anyway, which is inefficient!
- 43,252,003,274,489,856,000 positions for a Rubik's cube.
- Actually, if x and y are diametrically opposite then the given strategy still would explore (nearly) the entire graph just to find the x-y path.
- But now I will explain 2 ways to improve performance even in this "worst" case: Meet-in-the-Middle and A* ("A-star") search.

Meet in the Middle

- Suppose that each node of our graph has k neighbours, and we are applying the previous BFS technique to find a shortest path between x and y who are at distance d.
- Roughly speaking, each level of the search will expand the universe of "explored" nodes by a factor of d, so about d^k total time is needed.
- What's a simple way to improve? (Hint: look at the title)

Meet in the Middle

- Suppose that each node of our graph has k neighbours, and we are applying the previous BFS technique to find a shortest path between x and y who are at distance d.
- Roughly speaking, each level of the search will expand the universe of "explored" nodes by a factor of d, so about d^k total time is needed.
- What's a simple way to improve? (Hint: look at the title)
- Conduct a BFS simultaneously from x and y.
- Think of the BFS from y as going backwards.
- Do a level of x's BFS, then y's BFS, then x's, etc.
- Let P be a shortest path between x and y, and z be a middle point of that path; hence it is distance $k/2$ from both x and y.
- We can detect that two trees will hit after $k/2$ rounds — total time complexity $O(d^{k/2})$.

A-Star Search

- Again, we want to search from x to y in a huge graph.
- Basic idea: We can improve Dijsktra's shortest path algorithm by taking in to account an *estimate* of how far each node is from the target.

A-Star Search

- Again, we want to search from x to y in a huge graph.
- Basic idea: We can improve Dijsktra's shortest path algorithm by taking in to account an *estimate* of how far each node is from the target.
- Let h be a nonnegative underestimating function:

$$\text{for all } v : dist(y, v) \geq h(v).$$

A-Star Search

- Again, we want to search from x to y in a huge graph.
- Basic idea: We can improve Dijsktra's shortest path algorithm by taking in to account an *estimate* of how far each node is from the target.
- Let h be a nonnegative underestimating function:

$$\text{for all } v : dist(y, v) \geq h(v).$$

- Intuition: if $h(v_1) \gg h(v_2)$ then we should explore v_2 first.

A-Star Search

- Again, we want to search from x to y in a huge graph.
- Basic idea: We can improve Dijsktra's shortest path algorithm by taking in to account an *estimate* of how far each node is from the target.
- Let h be a nonnegative underestimating function:

$$\text{for all } v : dist(y, v) \geq h(v).$$

- Intuition: if $h(v_1) \gg h(v_2)$ then we should explore v_2 first.
- Example: 15-square. Each step we can slide a square into the hole.

13	10	11	6
5	7	4	8
1	12	14	9
3	15	2	

$\Rightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

A-Star Search

- Again, we want to search from x to y in a huge graph.
- Basic idea: We can improve Dijsktra's shortest path algorithm by taking in to account an *estimate* of how far each node is from the target.
- Let h be a nonnegative underestimating function:

$$\text{for all } v : dist(y, v) \geq h(v).$$

- Intuition: if $h(v_1) \gg h(v_2)$ then we should explore v_2 first.
- Example: 15-square. Each step we can slide a square into the hole.

13	10	11	6
5	7	4	8
1	12	14	9
3	15	2	

$\Rightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- If y is the unscrambled state, then we may take

$$h(v) = \text{number of out-of-position elements in } v.$$

- In route planning, h can be the Euclidean distance to y.

A-Star Search

- Implementing the idea: in Dijkstra's version of the generic search algorithm, we gave the edge (v, w) priority $\mathrm{d}[v]+c[v, w]$.

A-Star Search

- Implementing the idea: in Dijkstra's version of the generic search algorithm, we gave the edge (v, w) priority $\mathrm{d}[v]{+}c[v, w]$.
- Instead, give it priority $\mathrm{d}[v]{+}c[v, w]{+}h(w)$.
- Penalizes the search away from nodes believed to be far off-target.

A-Star Search

- Implementing the idea: in Dijkstra's version of the generic search algorithm, we gave the edge (v, w) priority $\mathrm{d}[v]+c[v, w]$.
- Instead, give it priority $\mathrm{d}[v]+c[v, w]+h(w)$.
- Penalizes the search away from nodes believed to be far off-target.
- Unfortunately, this doesn't exactly work as we had hoped. In order to get the right answer we may have to explore some nodes many times.
- Essentially, inconsistent local overestimates can deter us from short paths.
- We must add the following two *consistency* conditions to h :
  - $h(y) = 0$,
  - $h(p) - h(q) \leq c[p, q]$ whenever pq is an edge of the graph.

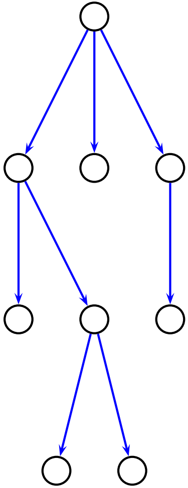
A-Star Search

- Implementing the idea: in Dijkstra's version of the generic search algorithm, we gave the edge (v,w) priority $\mathrm{d}[v]{+}c[v,w]$.
- Instead, give it priority $\mathrm{d}[v]{+}c[v,w]{+}h(w)$.
- Penalizes the search away from nodes believed to be far off-target.
- Unfortunately, this doesn't exactly work as we had hoped. In order to get the right answer we may have to explore some nodes many times.
- Essentially, inconsistent local overestimates can deter us from short paths.
- We must add the following two *consistency* conditions to h :
  - $h(y) = 0$,
  - $h(p) - h(q) \leq c[p,q]$ whenever pq is an edge of the graph.
- This *always* performs at least as quickly as Dijkstra's algorithm.
- As $h(v)$ increases towards a better underapproximation of $dist[v,y]$, the number of iterations required by A^* search decreases.

Outline

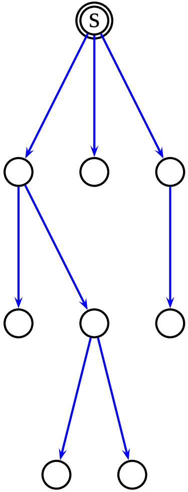
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.



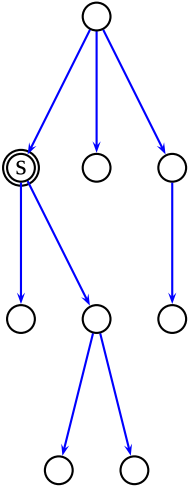
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.



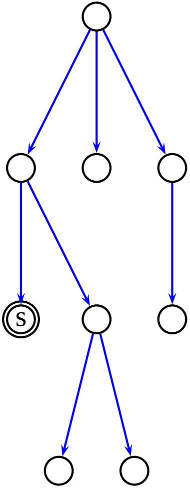
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.



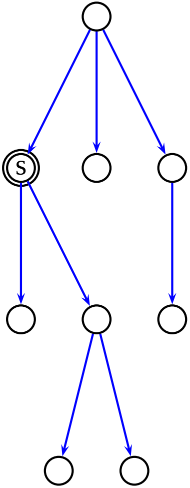
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.



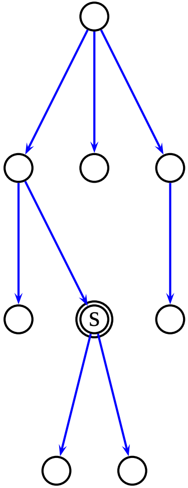
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



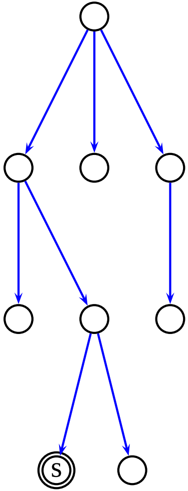
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



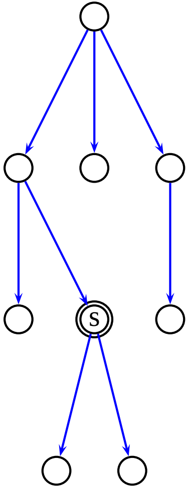
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



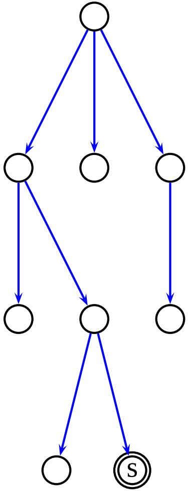
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



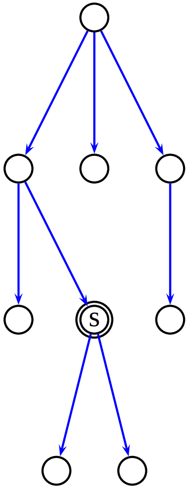
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



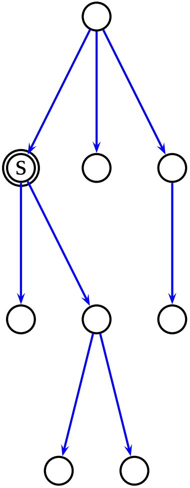
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



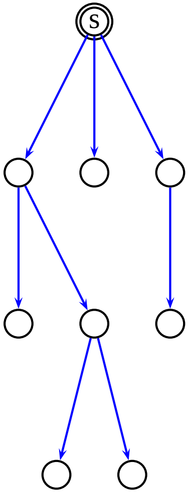
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



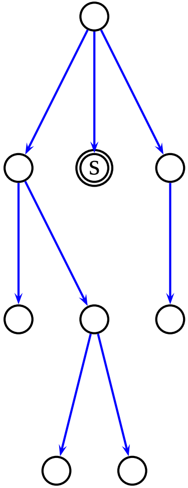
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



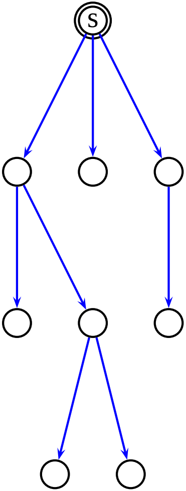
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



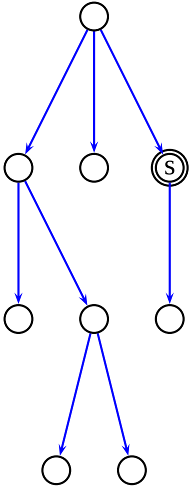
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



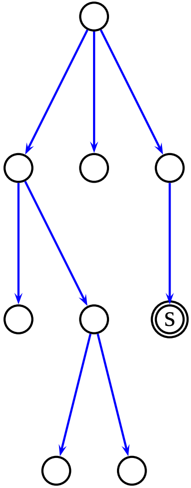
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



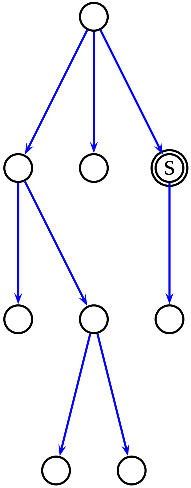
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



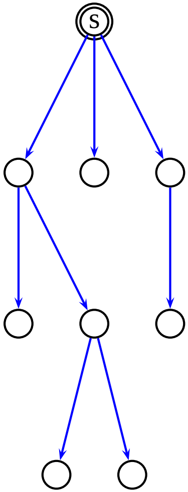
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



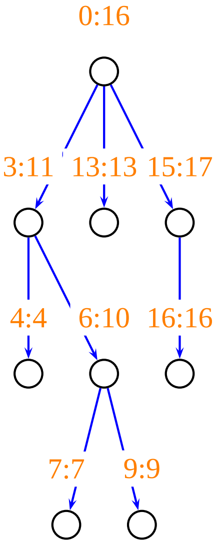
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.



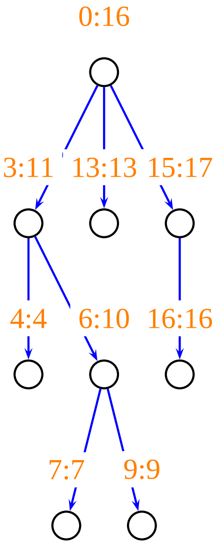
Walking on Trees

- Given: a tree, with the children of each node in some order (here, left-to-right).
- Imagine a squirrel walking along the edges of the tree, starting with the root.
- The squirrel always goes to the leftmost unvisited child of his current position.
- If the squirrel cannot move to any child (because he has visited them all, or none exist) he instead goes to the parent of that node.
- For each node record the first and last time the squirrel visited the node.



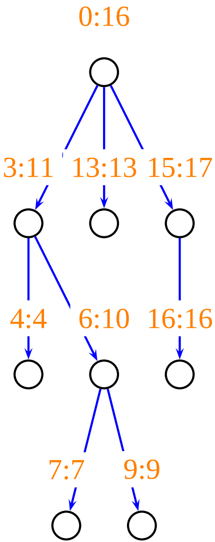
Walking on Trees

- If we order the nodes according to their first-visited times, we get a *preorder* on the nodes.
- Each vertex has preorder label less than its children.
- Conceptually: visit children of x *after* visiting x.



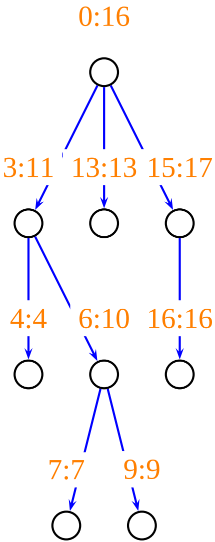
Walking on Trees

- If we order the nodes according to their first-visited times, we get a *preorder* on the nodes.
- Each vertex has preorder label less than its children.
- Conceptually: visit children of x *after* visiting x.
- Similarly the last-visited times define a *postorder*.
- Each vertex has postorder label greater than its children.
- Conceptually: visit children of x *before* visiting x.



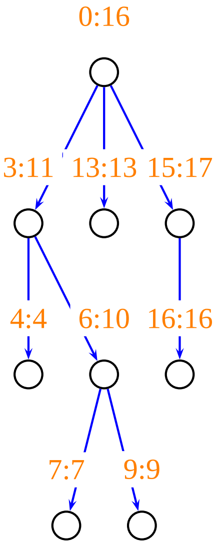
Walking on Trees

- If we order the nodes according to their first-visited times, we get a *preorder* on the nodes.
- Each vertex has preorder label less than its children.
- Conceptually: visit children of x *after* visiting x.
- Similarly the last-visited times define a *postorder*.
- Each vertex has postorder label greater than its children.
- Conceptually: visit children of x *before* visiting x.
- Forget the names? In *pre*order, x *pre*cedes its children.



Walking on Trees

- If we order the nodes according to their first-visited times, we get a *preorder* on the nodes.
- Each vertex has preorder label less than its children.
- Conceptually: visit children of x *after* visiting x.
- Similarly the last-visited times define a *postorder*.
- Each vertex has postorder label greater than its children.
- Conceptually: visit children of x *before* visiting x.
- Forget the names? In *pre*order, x *pre*cedes its children.
- *Aside:* for binary trees there is also *inorder* where you first visit the left child, then the root, then the right child.



Topological Sort

- *Topological sort* models the following problem.

[1] Usually DFS as it leads to efficient postorder computation.

Topological Sort

- *Topological sort* models the following problem.
- It is early in the morning and we are getting dressed.
- We have shoes, a hat, underwear, socks, jacket, pants, etc.
- But if we are too tired to figure out the correct order: disaster!

[1] Usually DFS as it leads to efficient postorder computation.

Topological Sort

- *Topological sort* models the following problem.
- It is early in the morning and we are getting dressed.
- We have shoes, a hat, underwear, socks, jacket, pants, etc.
- But if we are too tired to figure out the correct order: disaster!
- Formally, we have some vertices, and directed edges between the vertices. Edge $\overrightarrow{uv}$ means v must be put on before u.
- Assume there are no cycles (or else getting dressed is impossible). *In other words this is a directed acyclic graph (DAG)*.
- How can we determine an order to get dressed?

[1] Usually DFS as it leads to efficient postorder computation.

Topological Sort

- *Topological sort* models the following problem.
- It is early in the morning and we are getting dressed.
- We have shoes, a hat, underwear, socks, jacket, pants, etc.
- But if we are too tired to figure out the correct order: disaster!
- Formally, we have some vertices, and directed edges between the vertices. Edge $\overrightarrow{uv}$ means v must be put on before u.
- Assume there are no cycles (or else getting dressed is impossible). *In other words this is a directed acyclic graph (DAG).*
- How can we determine an order to get dressed?
- Basic idea: make any[1] (directed) search tree and use postorder.
- Complication: may need to pick multiple trees.

[1] Usually DFS as it leads to efficient postorder computation.

Topological Sort

- *Topological sort* models the following problem.
- It is early in the morning and we are getting dressed.
- We have shoes, a hat, underwear, socks, jacket, pants, etc.
- But if we are too tired to figure out the correct order: disaster!
- Formally, we have some vertices, and directed edges between the vertices. Edge $\overrightarrow{uv}$ means v must be put on before u.
- Assume there are no cycles (or else getting dressed is impossible). *In other words this is a directed acyclic graph (DAG).*
- How can we determine an order to get dressed?
- Basic idea: make any[1] (directed) search tree and use postorder.
- Complication: may need to pick multiple trees.
- Same idea gives *cycle detection.*

[1] Usually DFS as it leads to efficient postorder computation.

DFS Lite & Topological Sort

- DFS is often implemented without an *explicit* stack.

DFS Lite & Topological Sort

- DFS is often implemented without an *explicit* stack.
- Here's a short implementation of topological sort:

```
isExplored := boolean[v]                          ▷ Initialized to false.
postList := list⟨ int ⟩                            ▷ Initially empty.
procedure DFS-ORDER(G, v)
    isExplored[v] := true
    //preList.add(v)
    for all outneighbours w of v do
        if (!isExplored[w]) then DFS-Order(G, w)
    postList.add(v)
procedure TOPOLOGICALSORT(G, v)
    for i := 0 to v − 1 do
        DFS-Order(G, i)
    return postList
```

Outline

Bridge-Finding

- An edge of a connected graph is a *bridge* if, when it is deleted, the graph is no longer connected.
- Equivalently uv is a bridge if every path from u to v uses the edge uv.
- How can we determine the bridges of a graph?

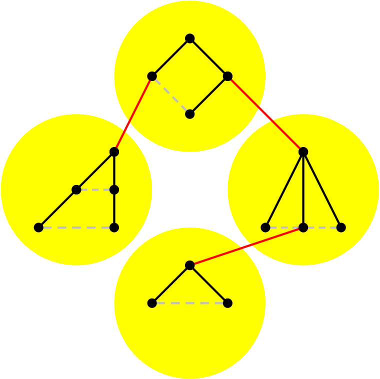
Bridge-Finding

- An edge of a connected graph is a *bridge* if, when it is deleted, the graph is no longer connected.
- Equivalently uv is a bridge if every path from u to v uses the edge uv.
- How can we determine the bridges of a graph?
- It is clear that any spanning tree contains all bridges.
- Furthermore we can argue that the tree edge $(P[v], v)$ is a bridge exactly when there are no edges “out of” the subtree rooted at v.

Bridge-Finding

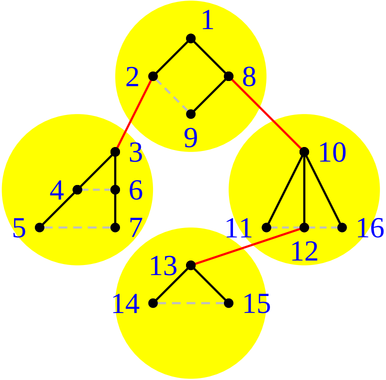
- An edge of a connected graph is a *bridge* if, when it is deleted, the graph is no longer connected.
- Equivalently uv is a bridge if every path from u to v uses the edge uv.
- How can we determine the bridges of a graph?
- It is clear that any spanning tree contains all bridges.
- Furthermore we can argue that the tree edge $(P[v], v)$ is a bridge exactly when there are no edges “out of” the subtree rooted at v.
- How can we compute this “out of” property precisely? Use the squirrel.

Bridge-Finding



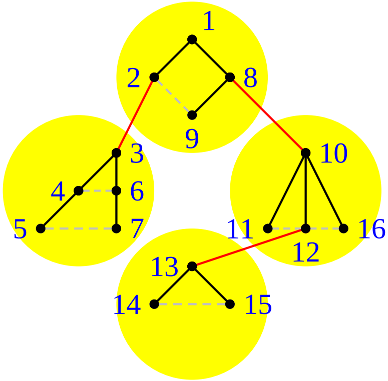
- What does “out of the subtree” mean?

Bridge-Finding



- What does "out of the subtree" mean?

Bridge-Finding



- What does "out of the subtree" mean?
- For each node let $low(v)$ be the minimum of its prelabel, its non-tree neighbours' prelabels, and its children's *low* values.
- For each node let $high(v)$ be the maximum of its prelabel, its non-tree neighbours' prelabels, and its children's *high* values.
- Can show that $(P[v], v)$ is a bridge if and only if $low(v) = pre(v)$ and $high(v) = pre(v) + subtreesize(v) - 1$.

Bridge-Finding

- An *articulation point*, analogous to a bridge, is a *vertex* whose deletion causes a graph to be disconnected.
- By refining the ideas above we can get a $O(n + m)$ time algorithm for articulation points. The formulation is cleanest using DFS because then there are no *cross edges* (edges uv such that neither u nor v is an ancestor of the other).
- Note that the naive algorithm for articulation points — delete each point in turn and see if the graph is connected — takes $O(n(m + n))$ time.
- You can also compute some other things called *biconnected components* and *blocks*. Roughly speaking, you can cut the graph into parts such that each part can tolerate any single node or vertex failure.

Strong Connectivity

- Consider a directed graph. Write $x \leftrightarrow y$ if there is a path from x to y and also from y to x.

Strong Connectivity

- Consider a directed graph. Write $x \leftrightarrow y$ if there is a path from x to y and also from y to x.
- Note: if $x \leftrightarrow y$ and $y \leftrightarrow z$ then $x \leftrightarrow z$. Thus $\leftrightarrow$ is an *equivalence relation.*

Strong Connectivity

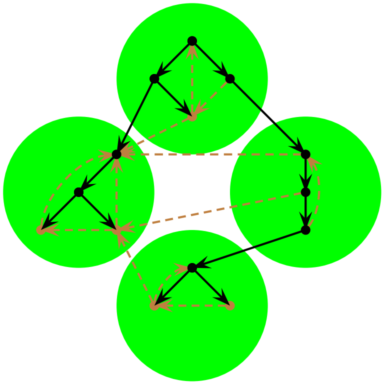
- Consider a directed graph. Write $x \leftrightarrow y$ if there is a path from x to y and also from y to x.
- Note: if $x \leftrightarrow y$ and $y \leftrightarrow z$ then $x \leftrightarrow z$. Thus $\leftrightarrow$ is an *equivalence relation*.
- In English: the vertices can be partitioned into *strong components* so that $x \leftrightarrow y$ if and only if x and y are in the same component.

Strong Connectivity

- Consider a directed graph. Write $x \leftrightarrow y$ if there is a path from x to y and also from y to x.
- Note: if $x \leftrightarrow y$ and $y \leftrightarrow z$ then $x \leftrightarrow z$. Thus $\leftrightarrow$ is an *equivalence relation*.
- In English: the vertices can be partitioned into *strong components* so that $x \leftrightarrow y$ if and only if x and y are in the same component.

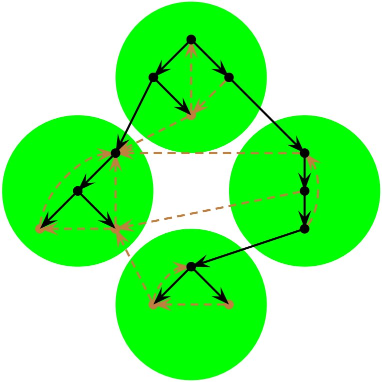
```
isExplored := boolean[v]                          ▷ Initialized to false.
postList := list⟨ int ⟩                           ▷ Initially empty.
procedure StrongComponents(G, v)
    for i := 0 to v − 1 do
        DFS-Order(G, i)
    newOrder := postList.copy().reverse()
    fill(isExplore, false)
    for i in newOrder do
        if !isExplored[i] then
            DFS-Label(G^T, i)                     ▷ When j is explored, label[j] := i.
    return labels
```

Strong Components



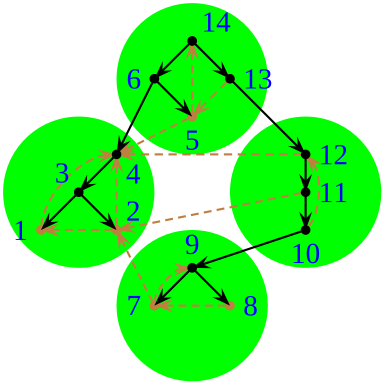
- Why does this DFS witchcraft work?

Strong Components



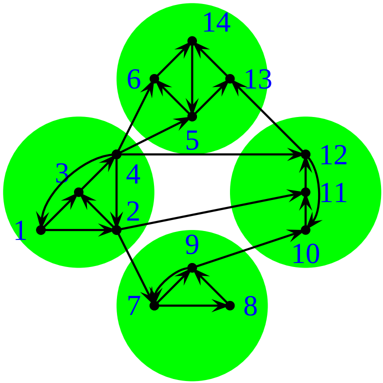
- Why does this DFS witchcraft work?
- The strong component blobs form a DAG (directed acyclic graph).

Strong Components



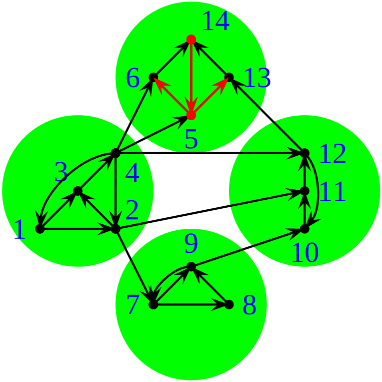
- Why does this DFS witchcraft work?
- The strong component blobs form a DAG (directed acyclic graph).
- Compute postorder,

Strong Components



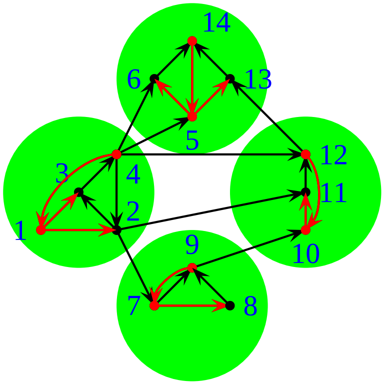
- Why does this DFS witchcraft work?
- The strong component blobs form a DAG (directed acyclic graph).
- Compute postorder, reverse G.

Strong Components



- Why does this DFS witchcraft work?
- The strong component blobs form a DAG (directed acyclic graph).
- Compute postorder, reverse G.
- Now starting from the highest-numbered vertex, the DFS gets "stuck" in that blob.

Strong Components



- Why does this DFS witchcraft work?
- The strong component blobs form a DAG (directed acyclic graph).
- Compute postorder, reverse G.
- Now starting from the highest-numbered vertex, the DFS gets "stuck" in that blob.
- Explore other blobs in turn.

Summary

- BFS is most useful for finding shortest paths.
- DFS can be coded very quickly. Gives many $O(m+n)$ time algorithms: topological sort, biconnectivity, strong connectivity, *planarity*, *triconnectivity*, …

Summary

- BFS is most useful for finding shortest paths.
- DFS can be coded very quickly. Gives many $O(m+n)$ time algorithms: topological sort, biconnectivity, strong connectivity, *planarity*, triconnectivity, ...
- Minimum Spanning Tree (Prim) and Single-Source Nonnegative Weighted Paths (Dijkstra) can be solved in the same framework.
- (Implementing heaps efficiently is left as a homework exercise)

Summary

- BFS is most useful for finding shortest paths.
- DFS can be coded very quickly. Gives many $O(m+n)$ time algorithms: topological sort, biconnectivity, strong connectivity, *planarity*, triconnectivity, …
- Minimum Spanning Tree (Prim) and Single-Source Nonnegative Weighted Paths (Dijkstra) can be solved in the same framework.
- (Implementing heaps efficiently is left as a homework exercise)
- Other useful ideas: preorder, postorder, bipartite.
- Can also search *implicit graphs;* then Meet-in-the-Middle and A^* are useful.
- A^* heuristic function must be an underestimate and must also be *consistent*.

Summary

- BFS is most useful for finding shortest paths.
- DFS can be coded very quickly. Gives many $O(m+n)$ time algorithms: topological sort, biconnectivity, strong connectivity, *planarity*, *triconnectivity, . . .*
- Minimum Spanning Tree (Prim) and Single-Source Nonnegative Weighted Paths (Dijkstra) can be solved in the same framework.
- (Implementing heaps efficiently is left as a homework exercise)
- Other useful ideas: preorder, postorder, bipartite.
- Can also search *implicit graphs;* then Meet-in-the-Middle and A^* are useful.
- A^* heuristic function must be an underestimate and must also be *consistent*.
- `return 0`