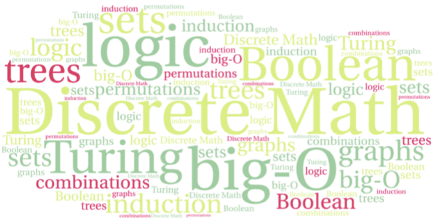
CS 220: Discrete Structures and their Applications

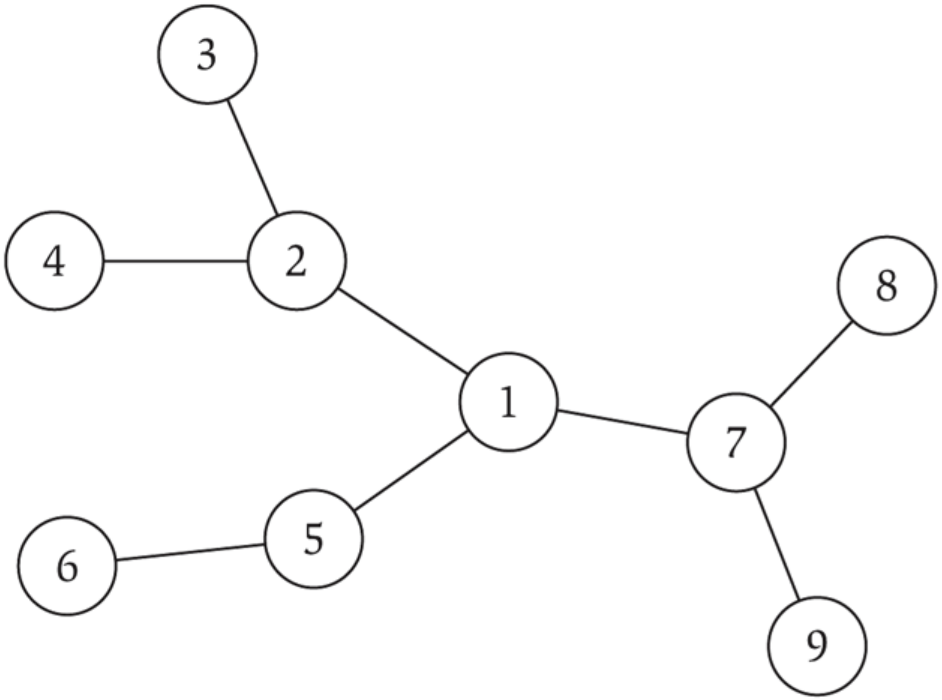
Trees

Chapter 11 in zybooks



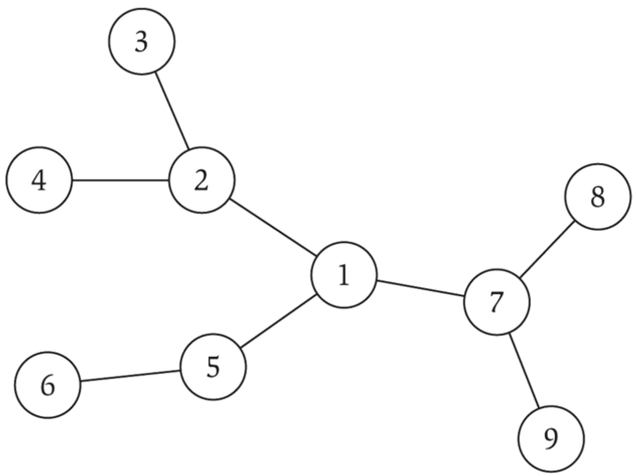
trees

A tree is an undirected graph that is connected and has no cycles.

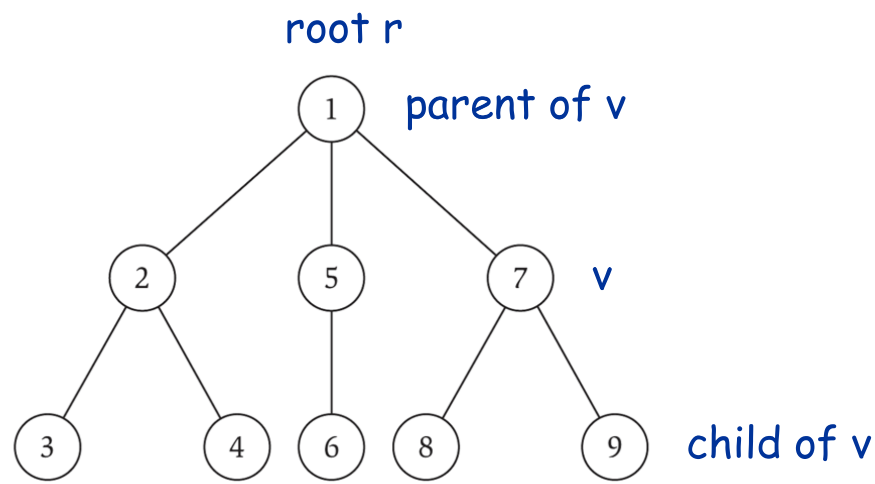


rooted trees

Rooted trees. Given a tree T, choose a root node r and orient each edge away (down) from r.



a tree

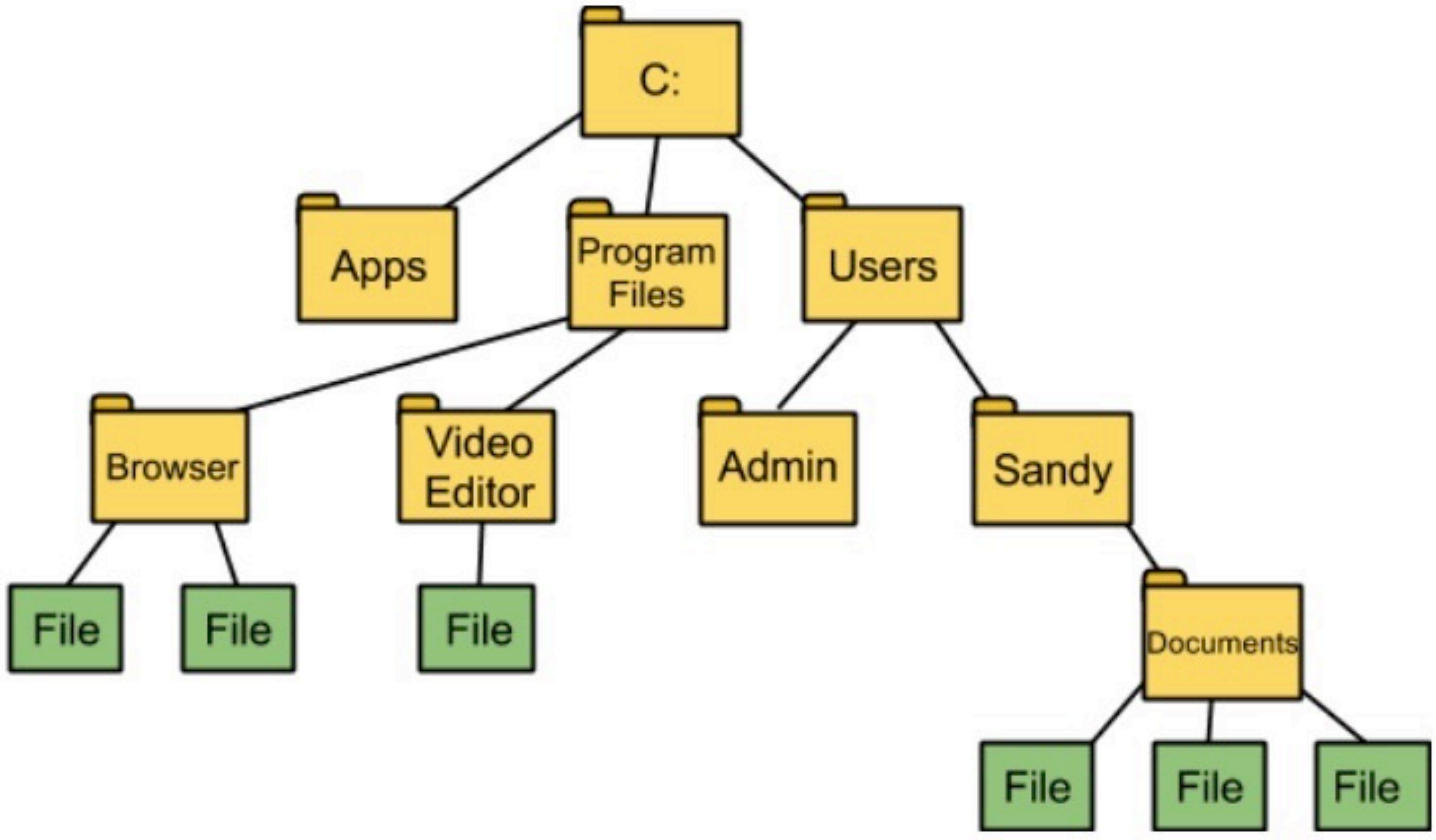


the same tree, rooted at 1

rooted trees

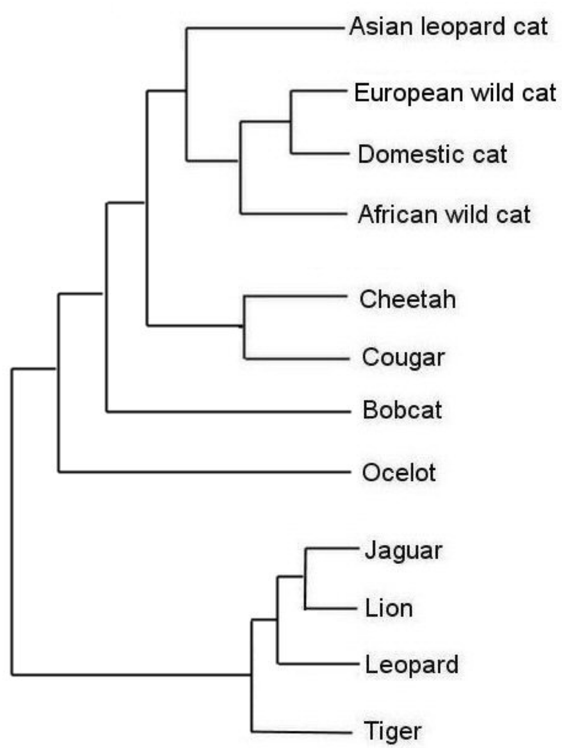
Rooted trees model hierarchical structure.

The file system as a rooted tree:



phylogenetic trees

Phylogeny. Describe the evolutionary history of species.

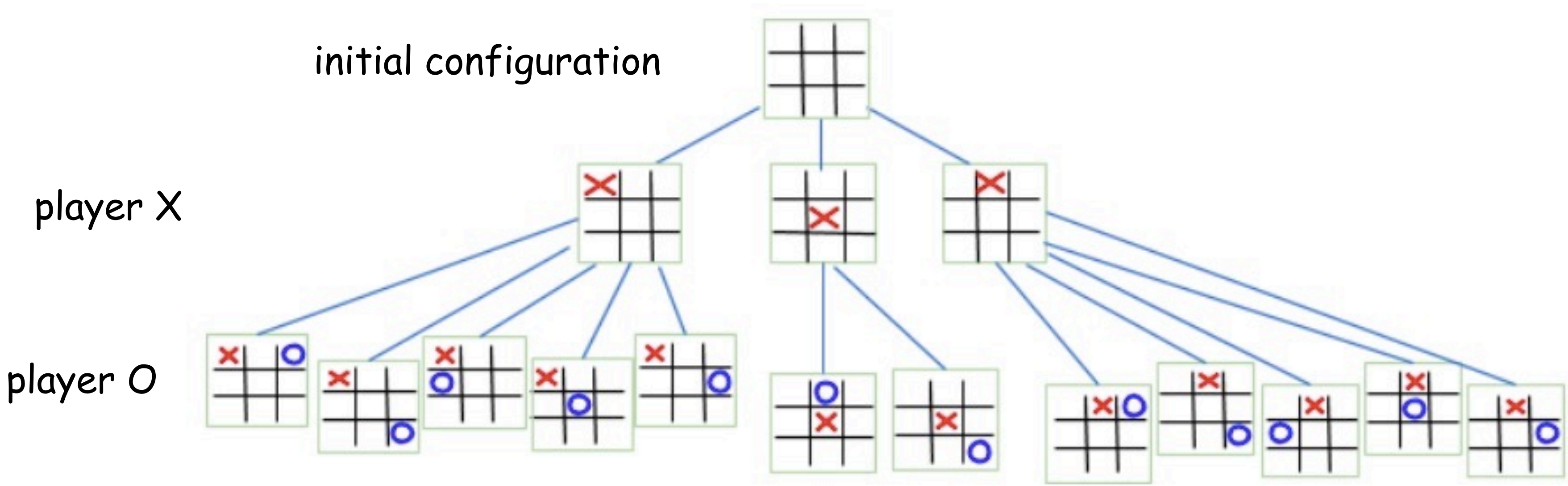


(Redrawn after Johnson, et al, 2006)

http://www.whozoo.org/mammals/Carnivores/Cat_Phylogeny.htm

game trees

games can be represented by trees:



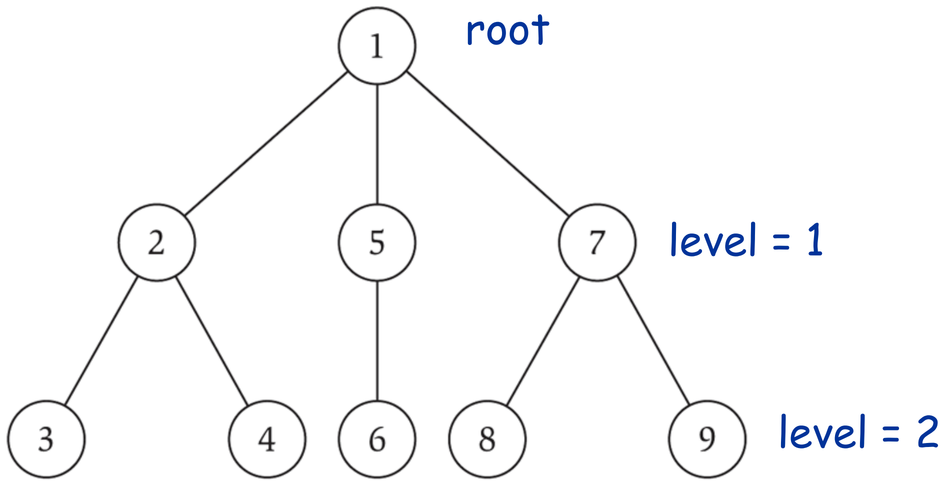
The root is the initial configuration.

The children of a state c are all the configurations that can be reached from c by a single move.

A configuration is a leaf in the tree if the game is over.

rooted trees

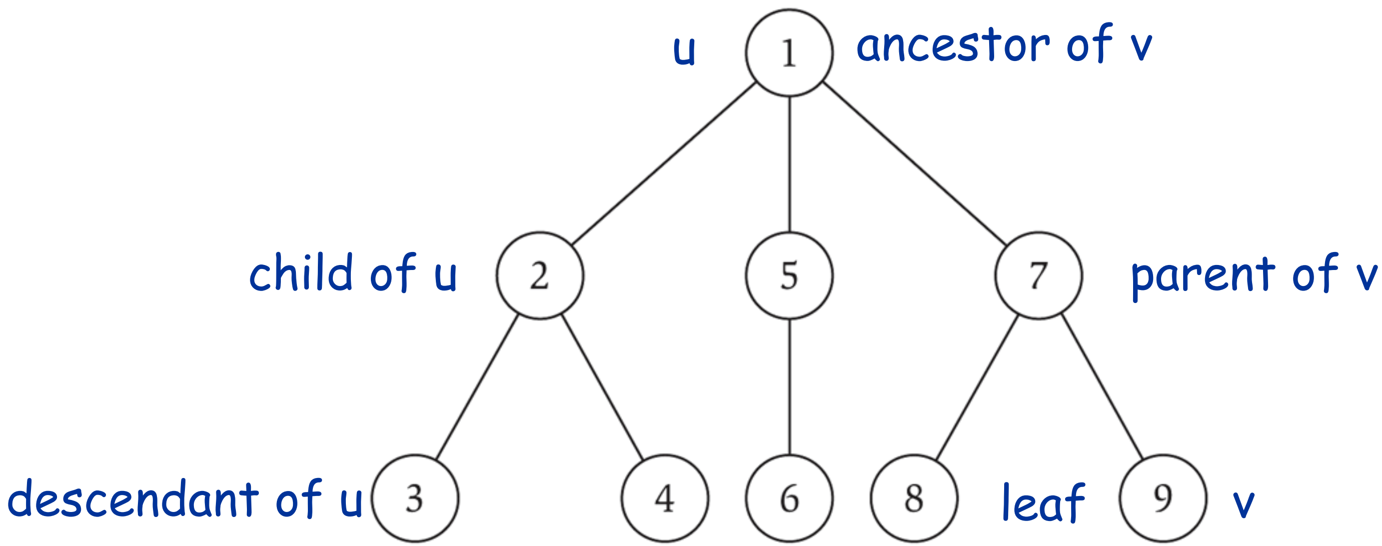
Rooted trees. Given a tree T, choose a root node r and orient each edge away from r.



The level of a node is its distance from the root

The height of a tree is the highest level of any vertex.

rooted trees

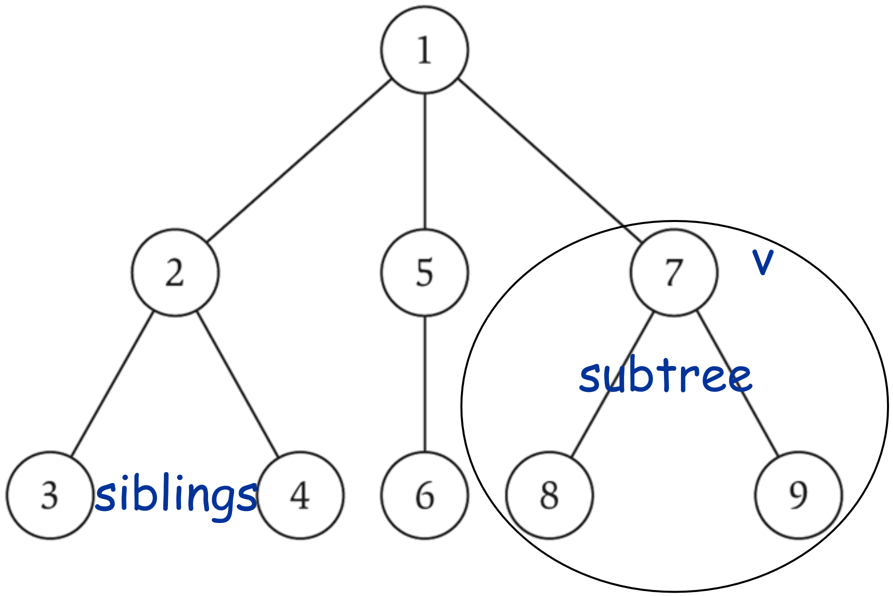


Every vertex in a rooted tree has a unique parent, except for the root which does not have a parent.

Every vertex along the path from v to the root (except for the vertex v itself) is an ancestor of v.

A leaf is a vertex which has no children.

rooted trees



Two vertices are siblings if they have the same parent.

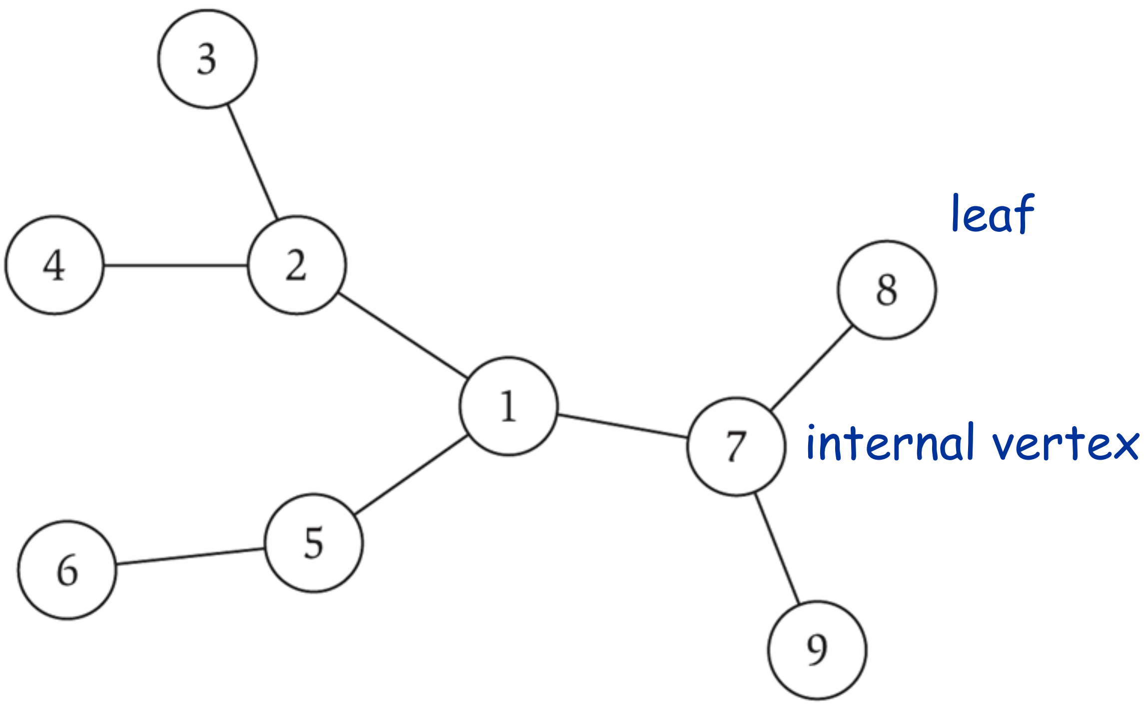
A subtree rooted at vertex v is the tree consisting of v and all v's descendants.

properties of trees

A leaf of an unrooted tree is a vertex of degree 1.

If a tree has only one vertex, then that vertex is a leaf.

A vertex is an internal vertex if the vertex has degree at least two.



properties of trees

A leaf of an **unrooted** tree is a vertex of degree 1.

Theorem: Any **unrooted** tree with at least two vertices has at least two leaves.

Proof.

Consider the longest path in the tree.

Its end vertices are both leaves.

But: what about a rooted tree?

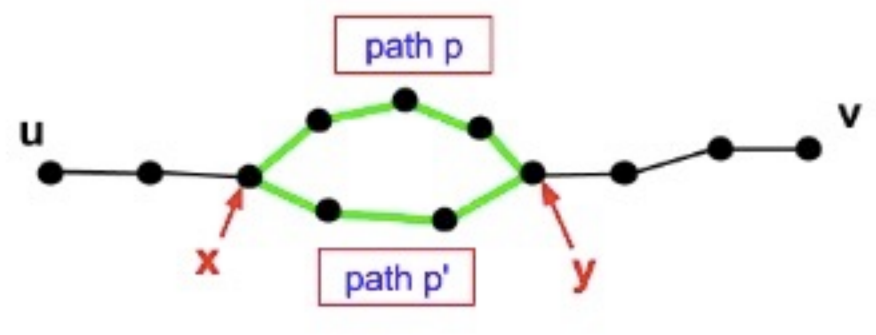
properties of trees

Theorem: There is a unique path between every pair of vertices in a tree.

Proof.

There is a path between every pair of vertices because a tree is connected. It remains to be seen that the path is unique.

Let's assume that the path is not unique:

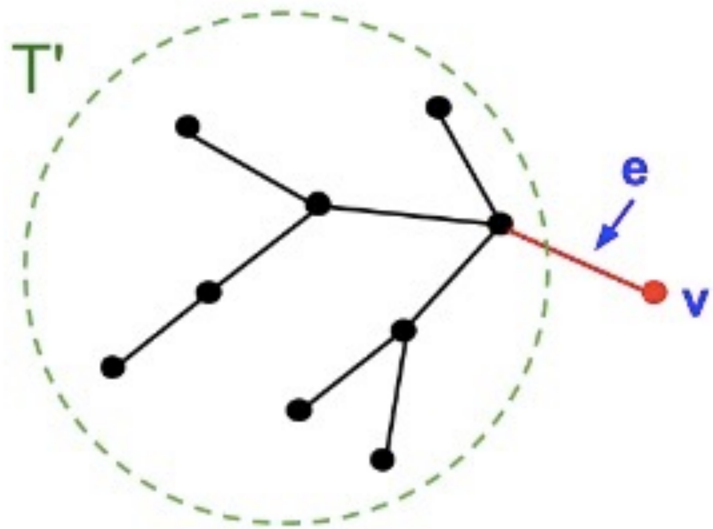


properties of trees

Theorem: Let T be a tree with n vertices and m edges, then m = n - 1.

Proof. By induction on the number of vertices.

Base case: is where n = 1. If T has one vertex, then it is has no edges, i.e. m = 0 = n - 1.

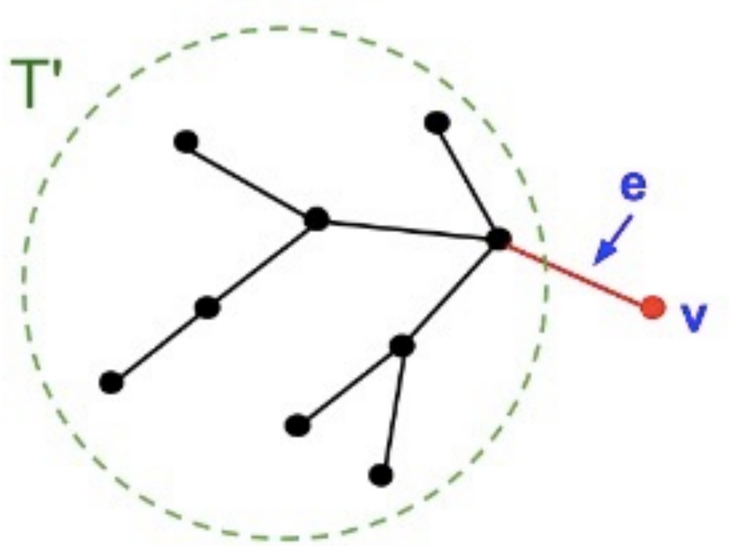


properties of trees

Theorem: Let T be a tree with n vertices and m edges, then m = n – 1.

Inductive step: assume the theorem holds for trees with n-1 vertices and prove that it holds for trees with n vertices.

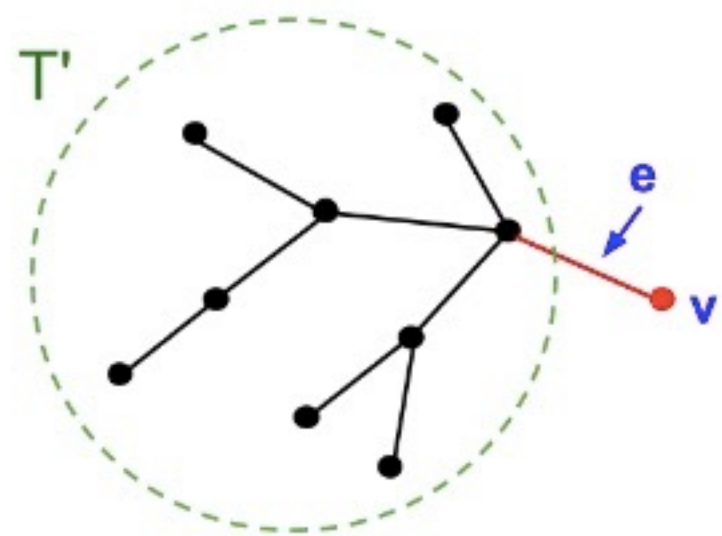
Consider an arbitrary tree T with n vertices. Let v be one of the leaves. Remove v from T along with the edge e incident to v. The resulting graph (call it T') is also a tree and has n-1 vertices.



properties of trees

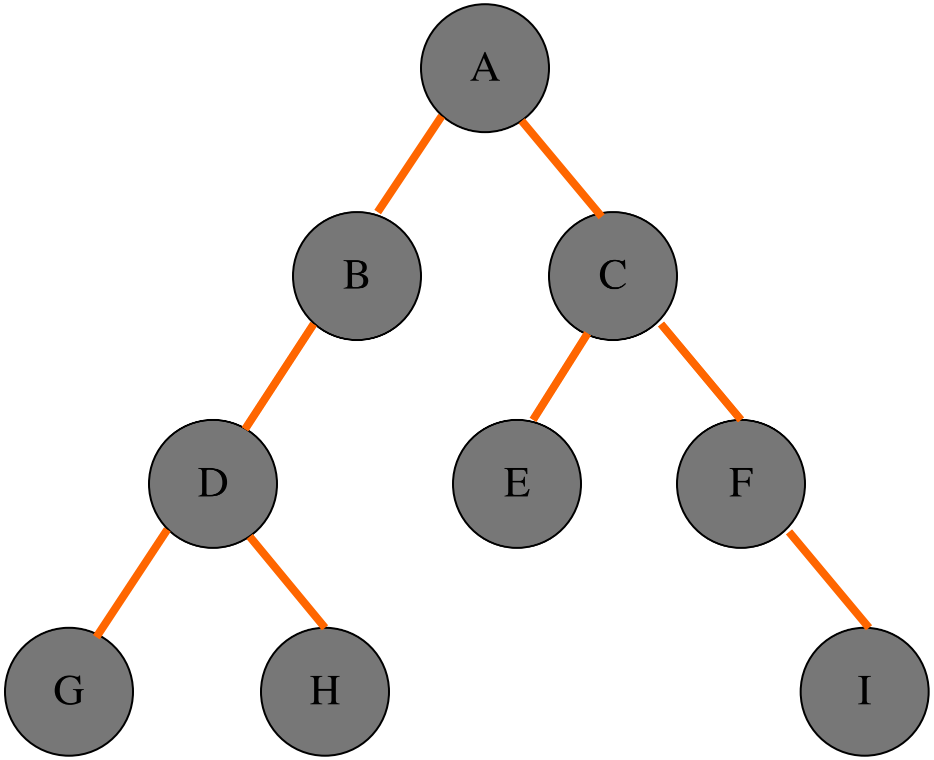
Theorem: Let T be a tree with n vertices and m edges, then m = n – 1.

By the induction hypothesis, The number of edges in T' is (n - 1) - 1 = n - 2. T has exactly one more edge than T', because only edge e was removed from T to get T'. Therefore the number of edges in T is n - 2 + 1 = n - 1. ■



Think of it as a rooted tree: every node except the root has 1 edge to its parent

traversal of a rooted tree



Pre order

Process the node
Visit its children

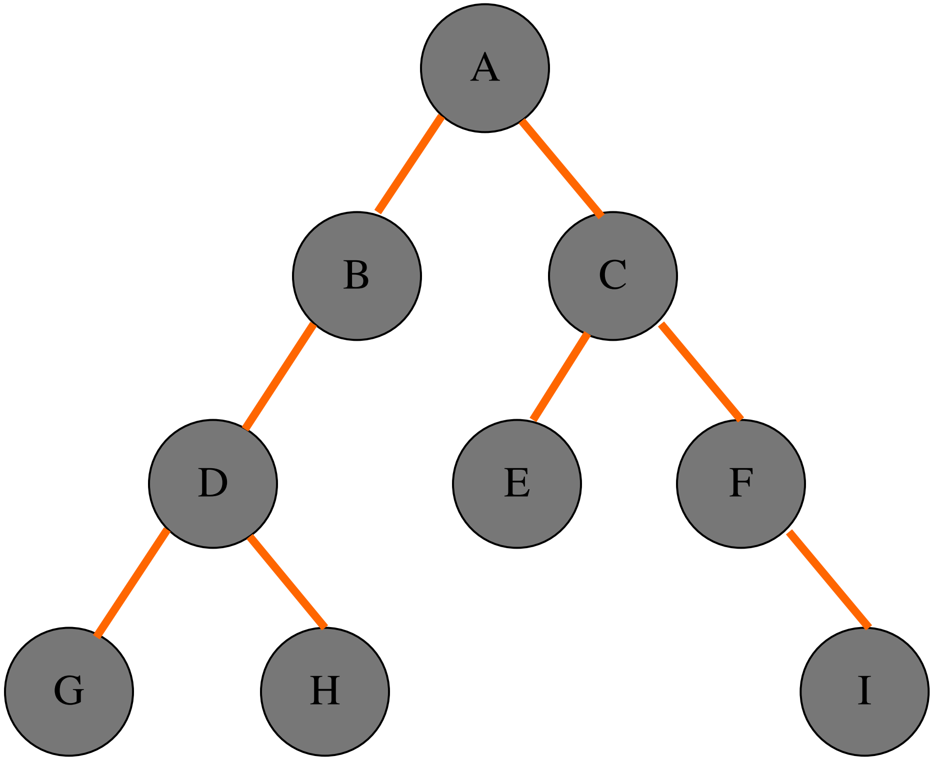
A B D G H C E F I

Post order

Visit the children
Process the node

G H D B E I F C A

traversal of a rooted tree



Pre order

Process the node
Visit its children

A B D G H C E F I

Post order

Visit the children
Process the node

G H D B E I F C A

which node gets processed first/last in each of these traversals?

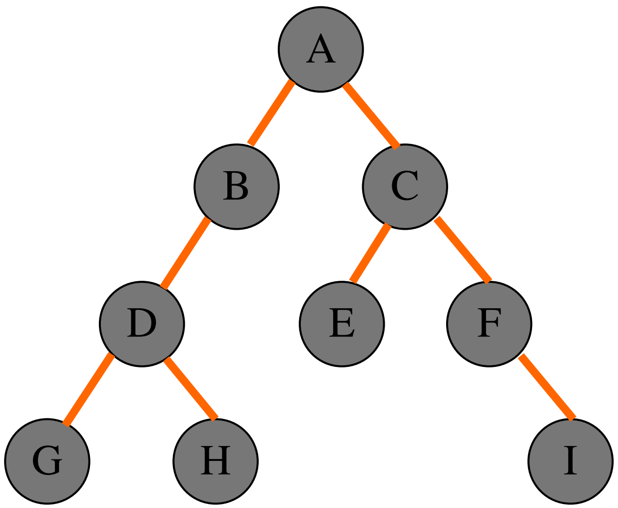
traversal of a rooted tree

pre-order(v)

```
process(v)
for every child w of v:
    pre-order(w)
```

post-order(v)

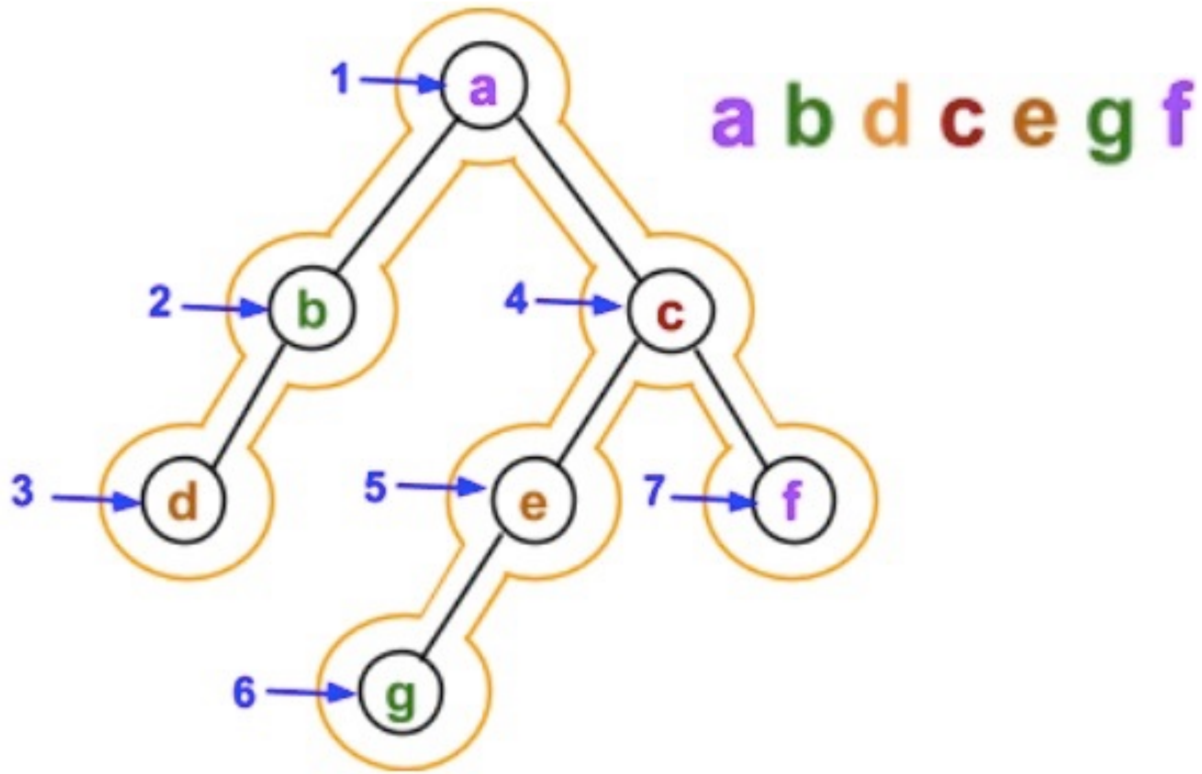
```
For every child w of v:
    post-order(w)
process(v)
```



a trick for pre-order traversal

To determine the order in which nodes are traversed in pre-order:

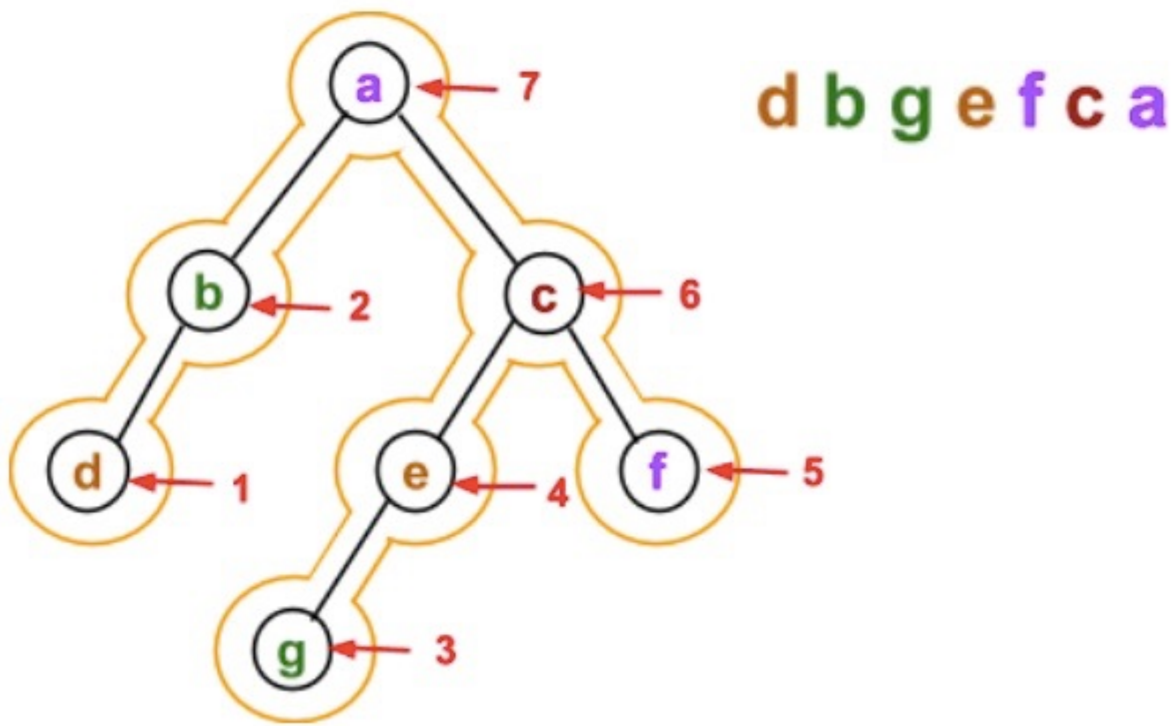
Follow the contour starting at the root; visit a vertex when passing to its left.



a trick for post-order traversal

To determine the order in which nodes are traversed in post-order:

Follow the contour starting at the root; visit a vertex when passing to its right.



counting leaves with post-order traversal

```
post-order-leaf-count(v)
  for every child w of v:
        post-order-leaf-count(w)
  if v is a leaf:
          leaf-count(v) = 1
  else :
          leaf-count(v) = sum of leaf counts of children
```

computing properties of trees using post-order

post-order-leaf-count(v)

```
for every child w of v:
    post-order-leaf-count(w)
if v is a leaf:
    leaf-count(v) = 1
else :
    leaf-count(v) = sum of leaf counts of children
```

Other properties that can be computed similarly:

- ✓ the total number of vertices in the tree.
- ✓ the height

traversal of a rooted binary tree

pre-order

- process the vertex
- go left
- go right

in-order

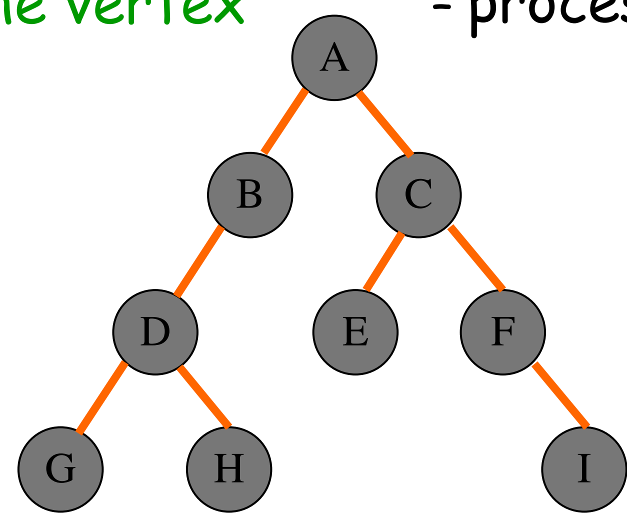
- go left
- process the vertex
- go right

post-order

- go left
- go right
- process the vertex

level order / breadth first

- for d = 0 to height
  - process vertices at level d



graph traversal

What makes it different from rooted tree traversal:

- graphs have cycles

What to do about it?

graph traversal

What makes it different from rooted tree traversal:

- graphs have cycles

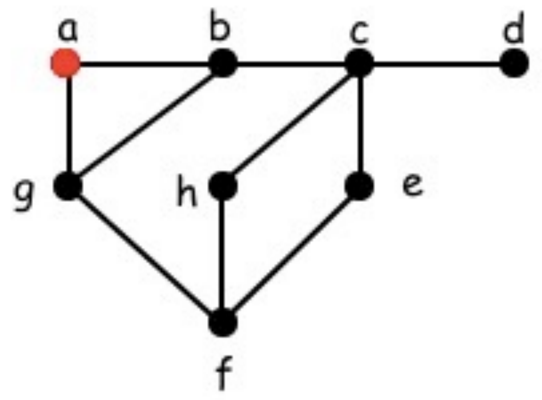
What to do about it?

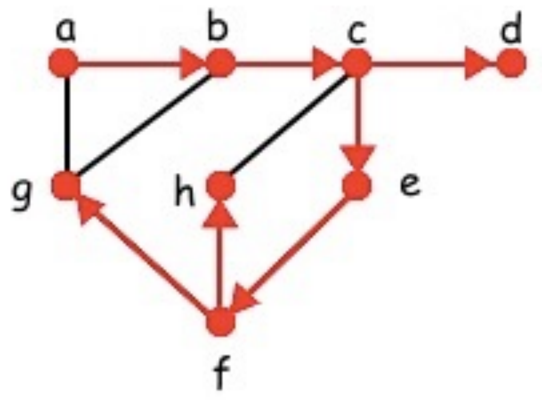
mark the vertices

depth-first search

Idea:

Go as deep as you can; backtrack when you get stuck





depth-first search

Pseudo-code:

```
dfs(v):
  mark v as explored
  for every neighbor w of v :
     if w is not explored :
        dfs(w)
```

dfs - nonrecursively

```
dfs(v):
   mark v as explored
   for every neighbor w of v :
      if w is not explored :
         dfs(w)
```

```
dfs(v) :
  s – stack of vertices to be processed
  mark v as explored
  s.push(v)
  while(s is non empty) :
     u = s.pop()
     for (each vertex v adjacent to u) :
         if v is not explored :
            mark v as explored
            s.push(v)
```

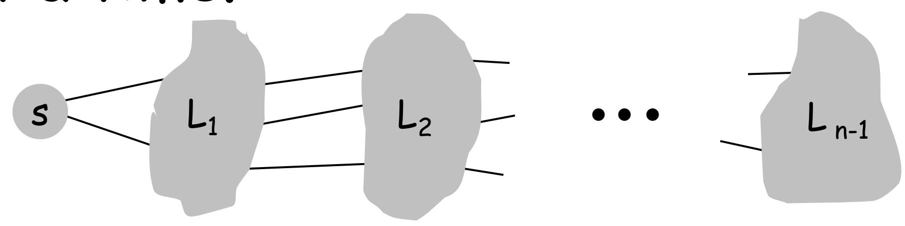
dfs vs bfs

DFS: Explores from the most recently discovered vertex; backtracks when reaching a dead-end.

BFS: Explores in order of distance from starting point

breadth first search

BFS intuition. Explore outward from s, adding vertices one "layer" at a time.



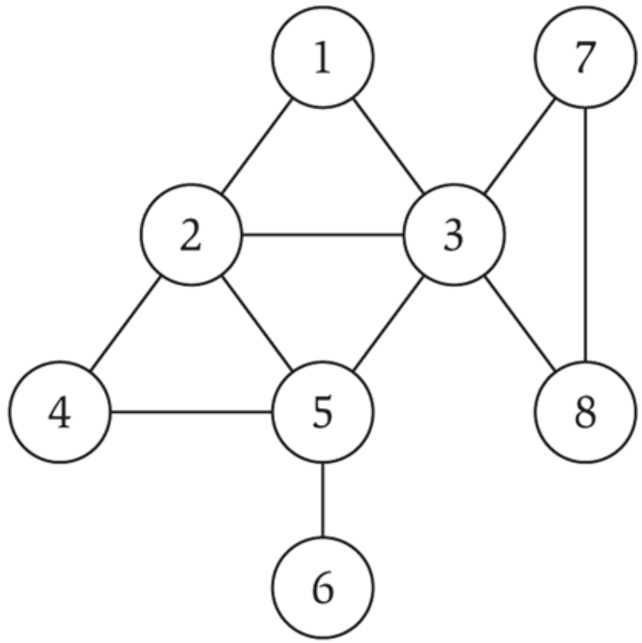
BFS algorithm.

- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all vertices that do not belong to L_0 or L_1, and that have an edge to a vertex in L_1.
- L_{i+1} = all vertices that do not belong to an earlier layer, and that have an edge to a vertex in L_i.

BFS - implementation

```
bfs(v) :
  q - queue of vertices to be processed
  mark v as explored
  q.enque(v)
  while(q is not empty) :
      u = q.dequeue()
      for (each vertex v adjacent to u) :
          if v is not explored :
              mark v as explored
              q.enqueue(v)
```

BFS - example



breadth first search: analysis

```
bfs(v) :
  q - queue of vertices to be processed
  mark v as explored
  q.enque(v)
  while(q is not empty) :
     u = q.dequeue()
     for (each vertex v adjacent to u) :
        if v is not explored :
           mark v as explored
           q.enqueue(v)
```

Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

Proof:

- when we consider vertex u, there are deg(u) incident edges (u, v)
- total time processing edges is $\Sigma_{u \in V} deg(u) = 2m$ ▪

each edge (u, v) is counted exactly twice in sum: once in deg(u) and once in deg(v)

DFS - Analysis

```
DFS(v) :
  s - stack of vertices to be processed
  s.push(v)
  mark v as Explored
  while(s is non empty) :
      u = s.pop()
      for (each vertex v adjacent to u) :
          if v is not Explored :
              mark v as Explored
              s.push(v)
```

Theorem. The above implementation of DFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

Proof:

Same as in BFS ▪

detecting cycles with dfs

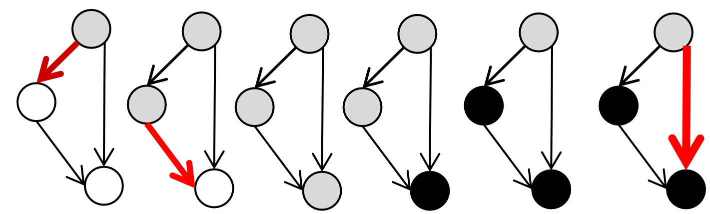
How would you modify DFS to detect cycles?

```
dfs(v):
   mark v as explored
   for every neighbor w of v :
      if w is not explored :
         dfs(w)
```

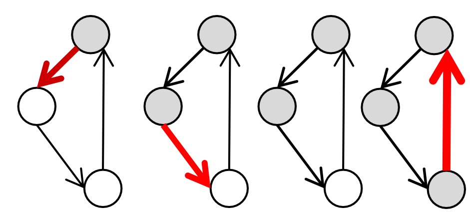
DFS and cyclic graphs

There are two ways DFS can **revisit** a node:

1. DFS has already fully explored the node. **What color does it have then? Is there a cycle then?**
No, the node is revisited from outside.

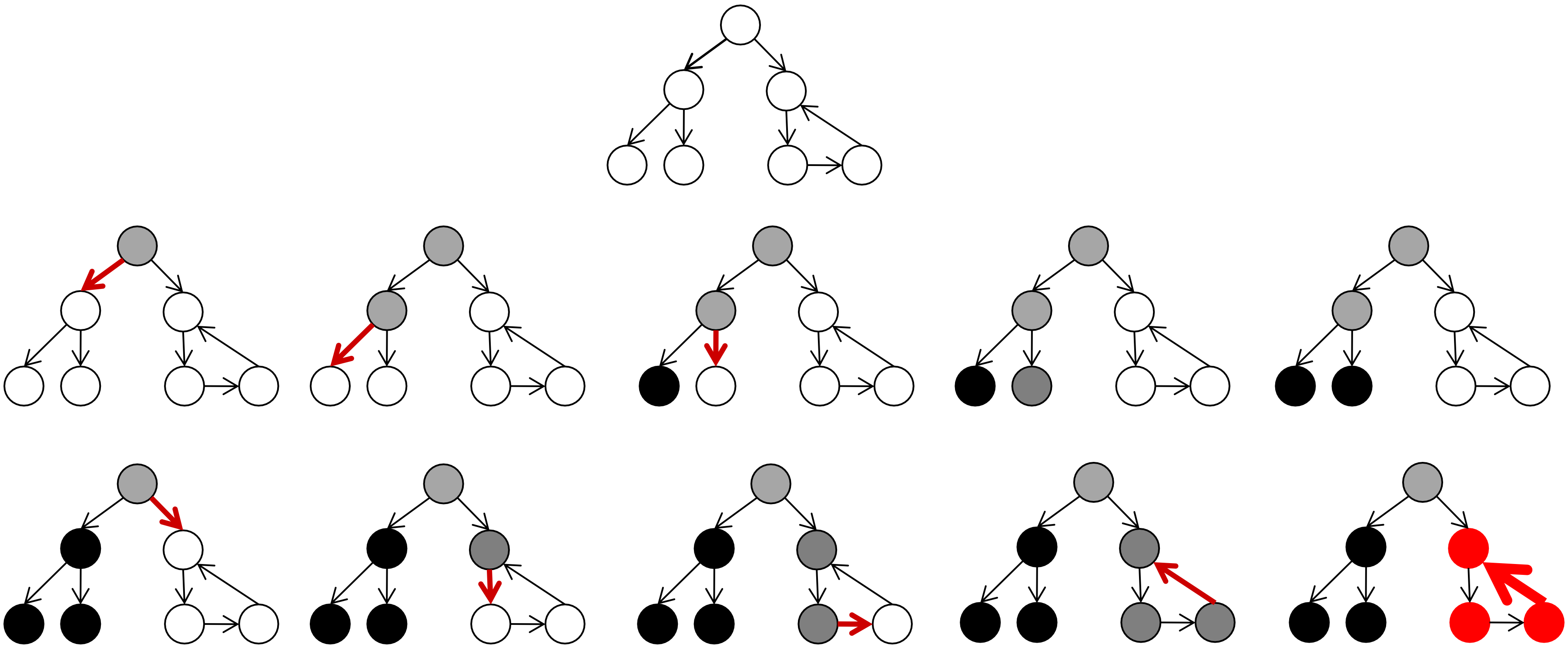


2. DFS is still exploring this node. **What color does it have in this case? Is there a cycle then?**
Yes, the node is revisited on a path containing the node itself.



So DFS with the white, grey, black coloring scheme detects a cycle when a GREY node is visited.

Cycle detection: DFS + coloring



When a grey (frontier) node is visited, a cycle is detected.

Recursive / node coloring version

```
DFS(u):
    #c: color,  p: parent
    c[u]=grey
    forall  v in Adj(u):
        if c[v]==white:
            p[v]=u
            DFS(v)
    c[u]=black
```

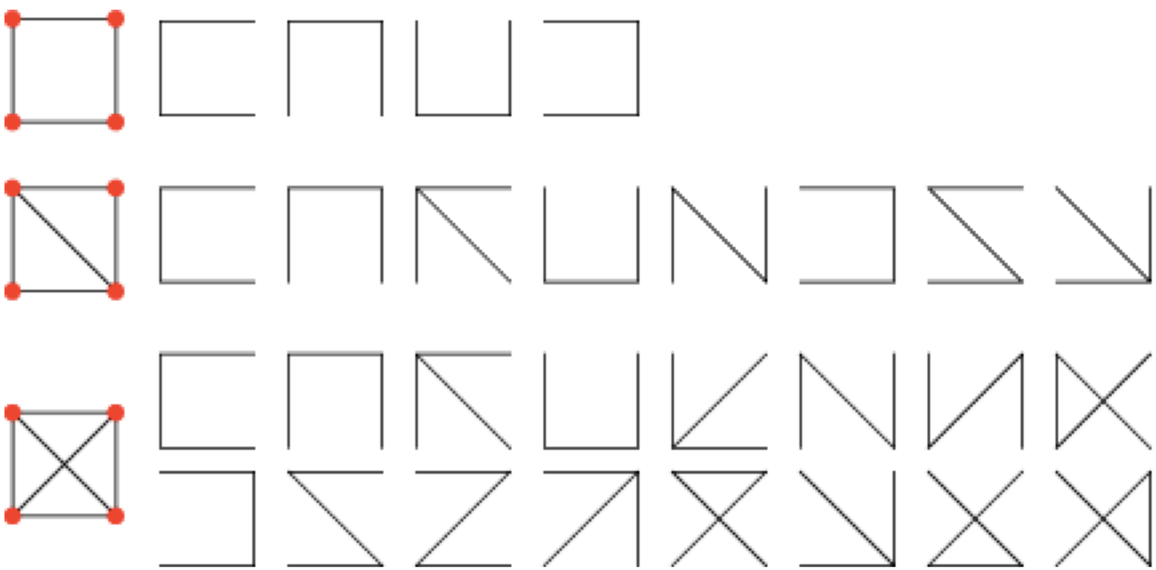
The above implementation of DFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

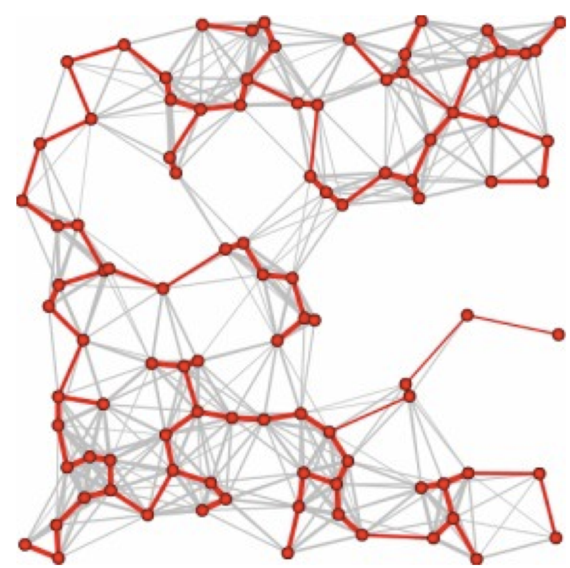
Proof:

Same as in BFS ▪

spanning trees

A spanning tree of a connected graph G is a subgraph of G which contains all the vertices in G and is a tree.





http://mathworld.wolfram.com/SpanningTree.html

computing spanning trees using graph traversal

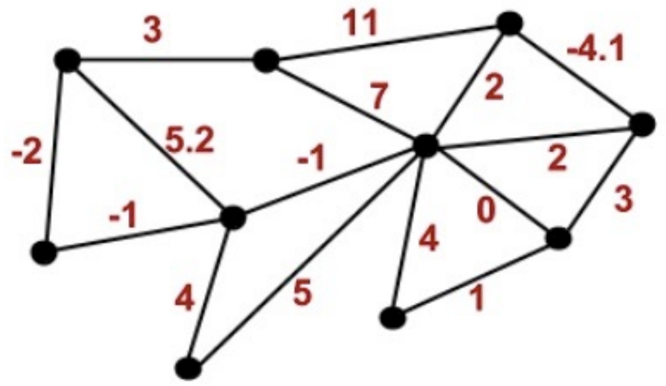
A spanning tree can be computed by a variation on DFS:

```
dfs-spanning-tree() :
   T is an empty tree
   add v to T
   visit(v)

visit(v):
   for every neighbor w of v :
      if w is not in T :
         add w and {v, w} to T
         visit(w)
```

can also be computed using BFS.

weighted graphs

A weighted graph is a graph G = (V ,E), along with a function w: E → R. The function w assigns a real number to every edge.



minimum spanning trees

Motivating example: each house in the neighborhood needs to be connected to cable

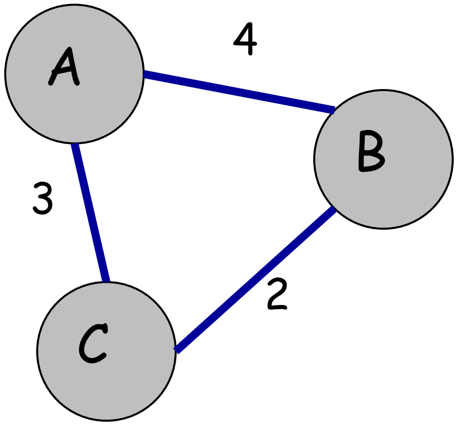
- Graph where each house is a vertex.
- Need the graph to be connected, and minimize the cost of laying the cables.

Model the problem with weighted graphs

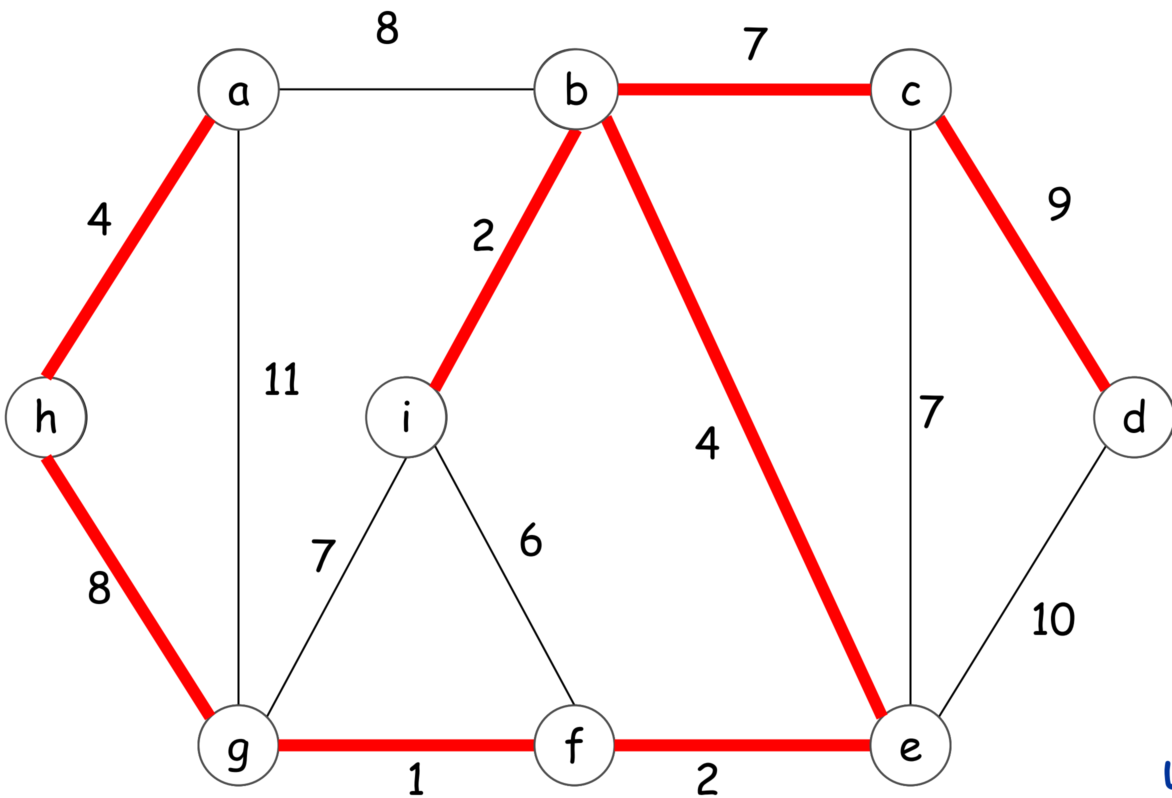
Minimum spanning tree

- Spanning tree minimizing the sum of edge weights

Incrementally build spanning tree by adding the least-cost edge to the tree



Prim's algorithm



unique?

{(d,c),(c,b), (b,i), (b,e), (e,f), (f,g), (g,h), (h,a) }

Prim's algorithm

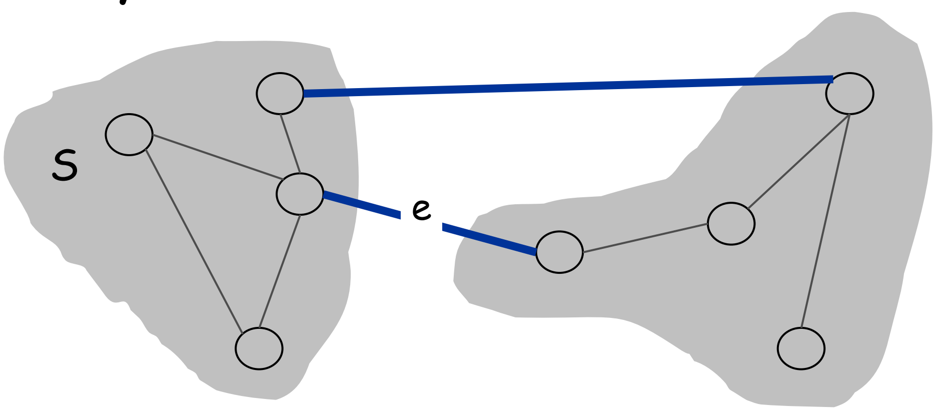
```
prims(G):
   Input: An undirected, connected, weighted graph G
   Output: T, a minimum spanning tree for G.

   T = Ø
   pick any vertex in G and add it to T.

   for j = 1 to n-1 :
      let C be the set of edges with one endpoint
           in T and one endpoint outside T
      let e be a minimum weight edge in C
      add e to T.
      add the endpoint of e not already in T to T
```

The cut property

Simplifying assumption. All edge costs are distinct.

Cut property. Let S be a subset of nodes, S neither empty nor equal V, and let e be the minimum cost edge with exactly one endpoint in S.
Then the MST contains e.

The cut property establishes the correctness of Prim's algorithm.



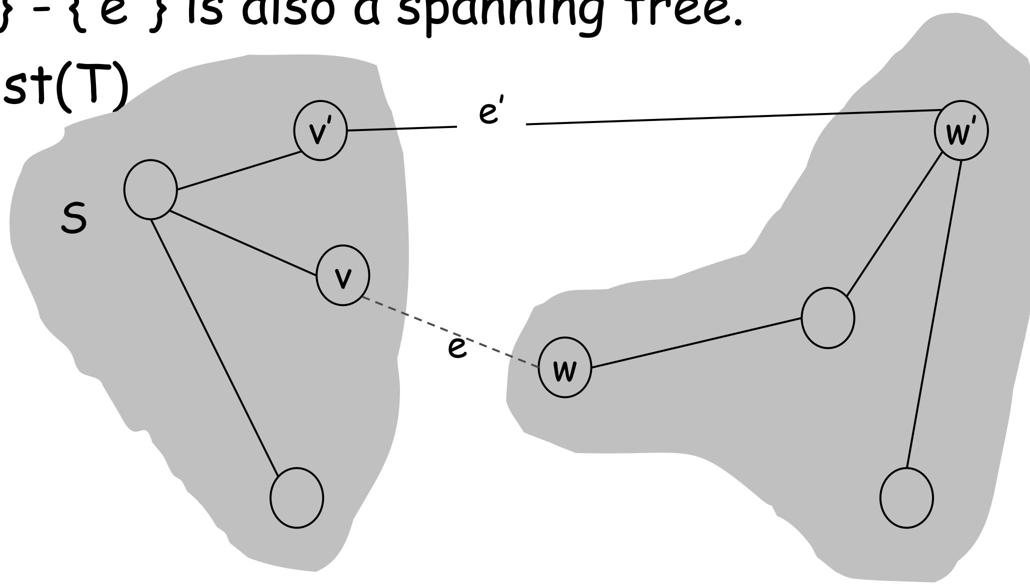
e is in the MST

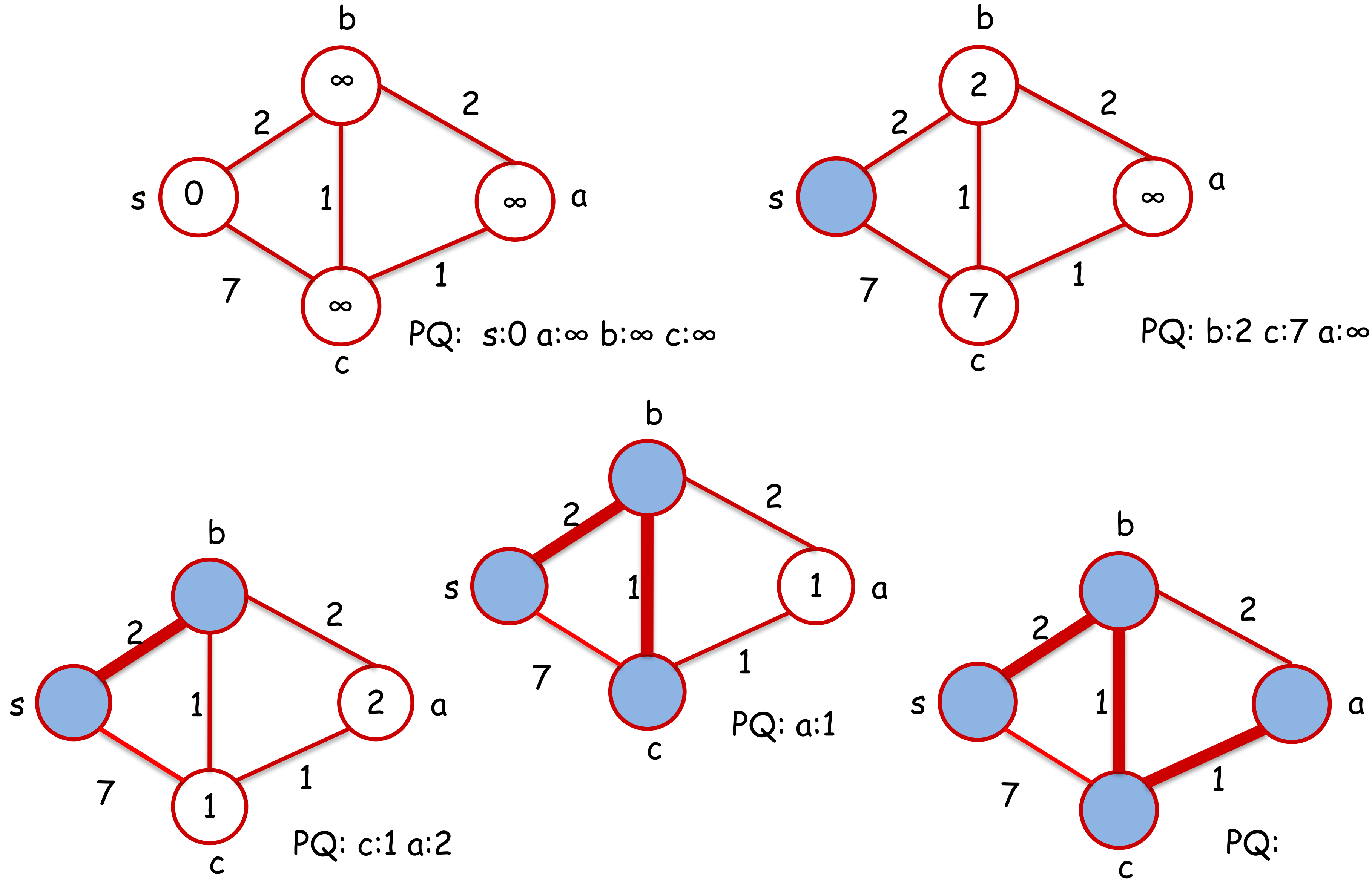
The cut property

Cut property. Let S be a subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T contains e.

Proof. (exchange argument)

- If e =(v,w) is the only edge connecting S and V-S it must be in T, else e is on a cycle in the graph (not the MST). Now suppose e does not belong to T.
- Let e'= (v',w') be the first edge between S and V-S on the path from v'. T' = T ∪ { e } - { e' } is also a spanning tree.
- Since $c_e < c_{e'}$, cost(T') < cost(T)
- This is a contradiction. ▪




b
∞
2
2
s
0
1
∞
a
7
∞
1
c
PQ: s:0 a:∞ b:∞ c:∞
b
2
2
2
s
1
∞
a
7
7
1
c
PQ: b:2 c:7 a:∞
b
2
2
s
1
2
a
7
1
1
c
PQ: c:1 a:2
b
2
2
s
1
1
a
7
1
c
PQ: a:1
b
2
2
s
1
a
7
1
c
PQ:

Shortest Paths Problems

Given a **weighted directed** graph G=(V,E)

find the shortest path

- path length is the sum of its edge weights.

The shortest path from u to v is ∞ if there is no path from u to v.

Variations:

1) **SSSP** (Single source SP): find the SP from some node s to all nodes in the graph.

2) **SPSP** (single pair SP): find the SP from some u to some v.

We can use 1) to solve 2), also there is no asymptotically faster algorithm for 2) than that for 1).

3) **SDSP** (single destination SP) can use 1) by reversing its edges.

4) ***APSP*** (all pair SPs) could be solved by |V| applications of 1), but can be solved faster (cs420).

Dijkstra SSSP

Dijkstra's (Greedy) SSSP algorithm only works for graphs with only positive edge weights.

S is the set of explored nodes

For each u in S, d[u] is a distance

Init: S = {s} the source, and d[s]=0

while S≠V:

    select a node v in V-S with at least one edge

    from S, for which $d'[v]=\min_{e=(u,v),u \text{ in } S} d[u]+w_e$

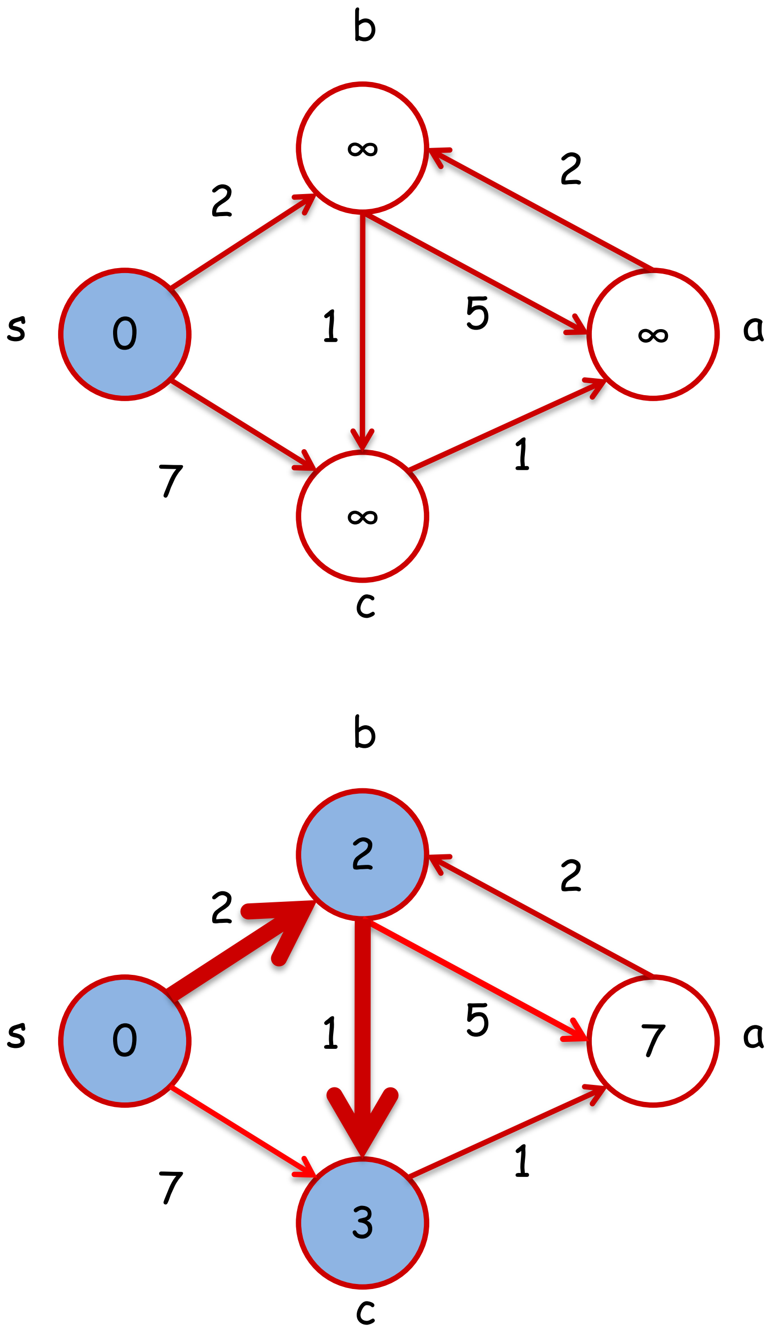
    add v to S (S=S+v)

    d[v]=d'[v]

the minimum **path** extending out of S

To compute the actual minimum paths, maintain an array p[v] of predecessors.

Notice: Dijkstra is very similar to Primm's MST algorithm


b
∞
2
2
s
0
1
5
∞
a
7
1
∞
c
b
2
2
2
s
0
1
5
∞
a
7
1
7
c
b
2
2
2
s
0
1
5
7
a
7
1
3
c
b
2
2
2
s
0
1
5
4
a
7
1
3
c

Dijkstra works

For each u in S, the path $P_{s,u}$ is the shortest (s,u) path

Proof by induction on the size of S

Base: |S| = 1 d[s]=0 OK

Step: Suppose it holds for |S|=k>=1, then grow S by 1 adding node v using edge (u,v) (u already in S) to create the next S.
Then path $P_{s,u,v}$ is path $P_{s,u}$+(u,v), and is the shortest path to v

WHY? What are the "ingredients" of an exchange argument?
What are the inequalities?

Greedy exchange argument

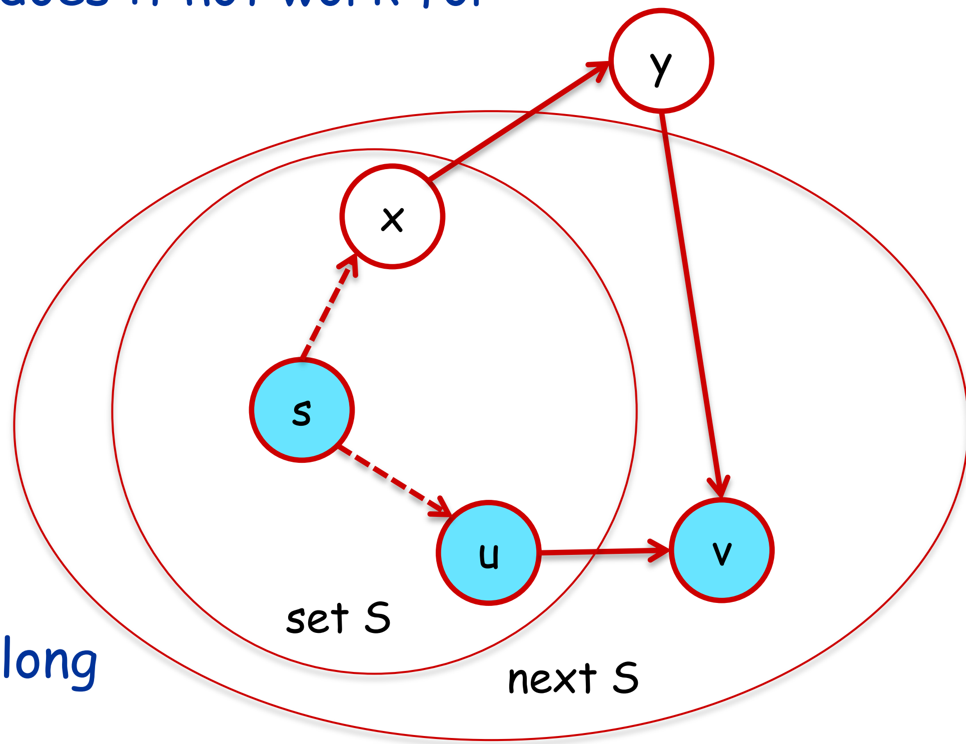
Assume there is **another path P from s to v**.

P leaves S somewhere with edge (x,y).

Then the path P goes from s to x to y to v.

What can you say about P: s →* x → y compared to $P_{s,u,v}$? How does the algorithm pick $P_{s,u,v}$? Why does it not work for negative edges?

P from s to y is at least as long as $P_{s,u,v}$ because the algorithm picks the shortest extension out of S.



Hence the path

P: s →* x → y →* v is at least as long as

$P_{s,u,v}$: s →* u → v

Would not work if w(y,v) <0