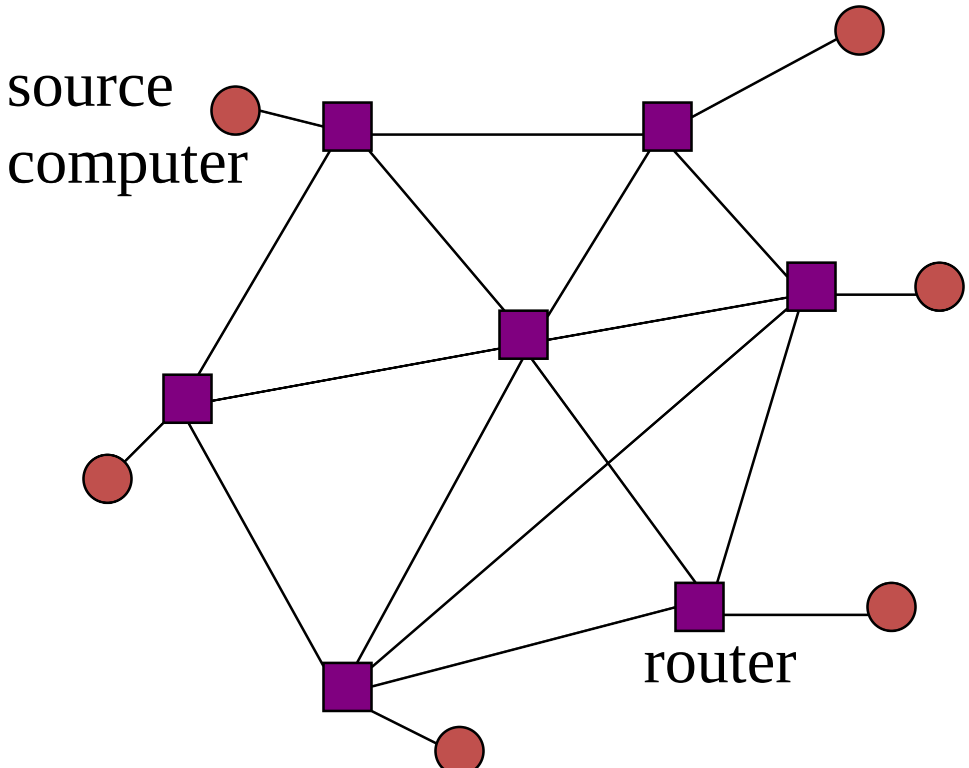
Ch 11.4 Spanning Trees

- A simple graph is connected
  if and only if it has a spanning tree.
- Applied in IP multitasking.

Example: IP Multicasting

- A network of computers and routers:



Example: IP Multicasting

- How can a packet (message) be sent from the source computer to every other computer?

- The *inefficient* way is to use *broadcasting*
  – send a copy along every link, and have each router do the same
  – each router and computer will receive many copies of the same packet
  – loops may mean the packet never disappears!

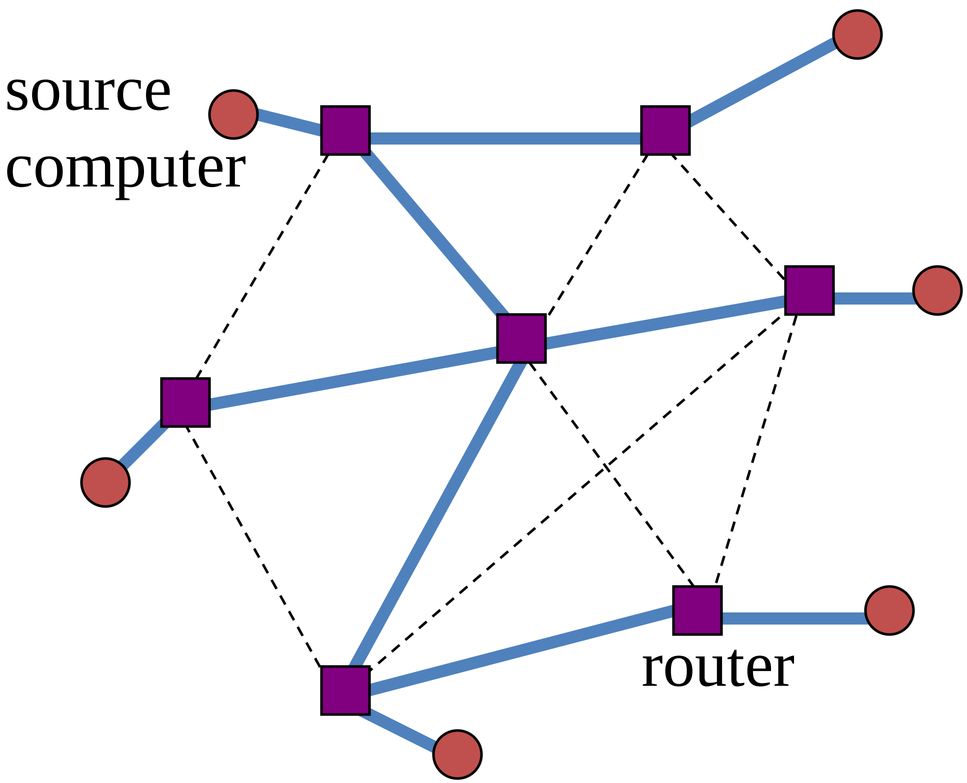
Example: IP Multicasting

- IP multicasting is an *efficient* solution
  - send a single packet to one router
  - have the router send it to 1 or more routers in such a way that a computer never receives the packet more than once

- This behaviour can be represented by a spanning tree.

Example: IP Multicasting

- The spanning tree for the network:



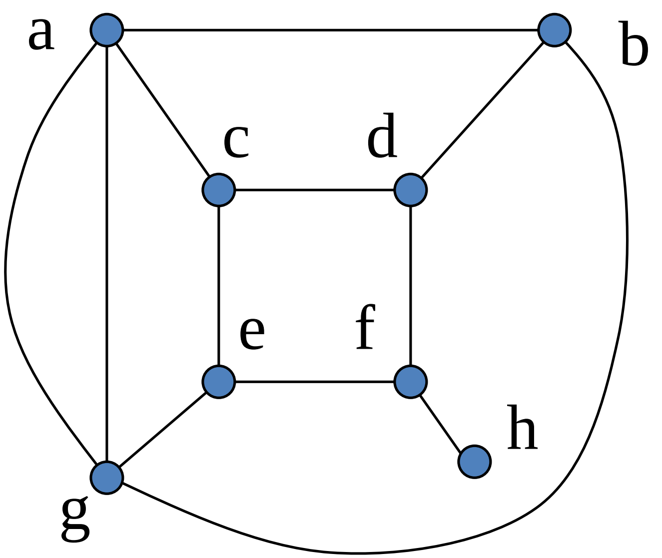
the tree is drawn with thick lines

Spanning Trees

- Let G be a simple graph.
- A *spanning tree* of G is a subgraph of G
  that is a tree
  containing every vertex of G.

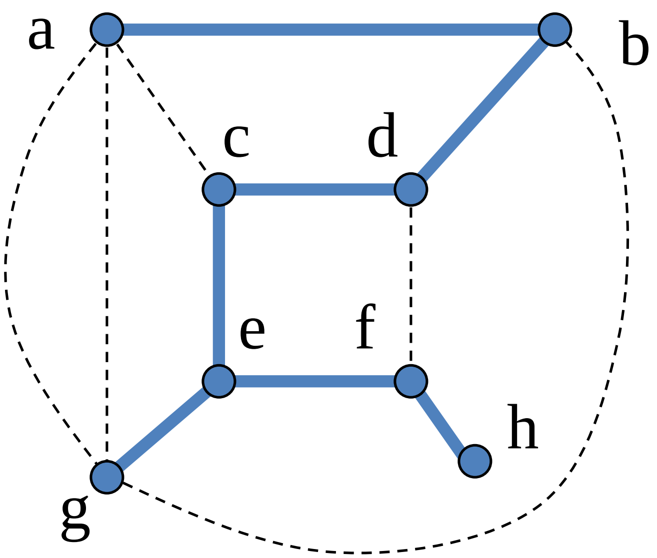
Spanning Trees

- Example graph G:



Spanning Trees

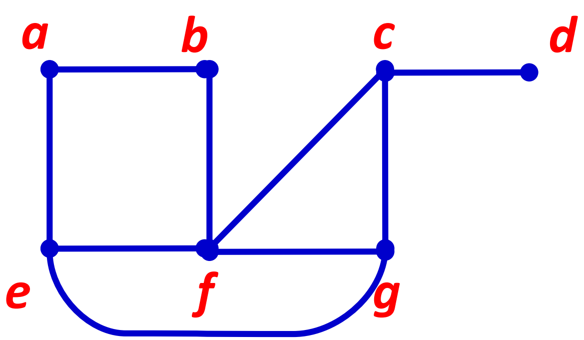
- One possible spanning tree:



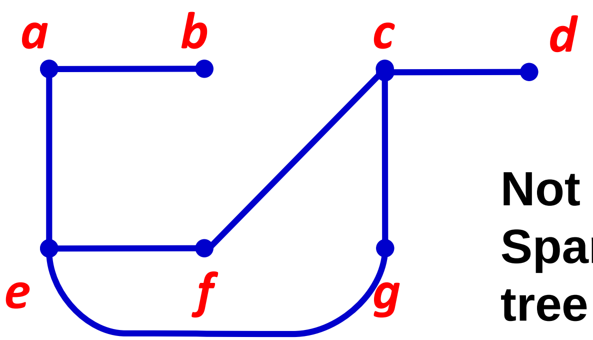
the tree is drawn with thick lines

Spanning Trees

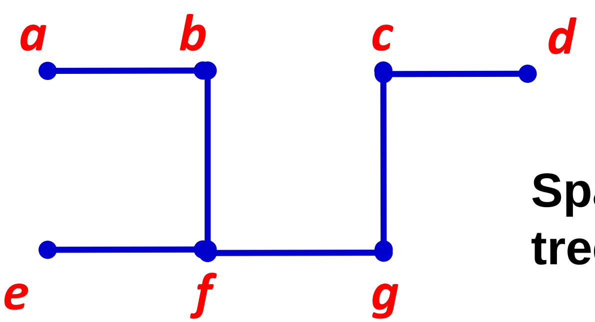
- Example Spanning Tree



A simple graph



Not Spanning tree



Spanning tree

Spanning Trees

Find a spanning tree for the following graphs.

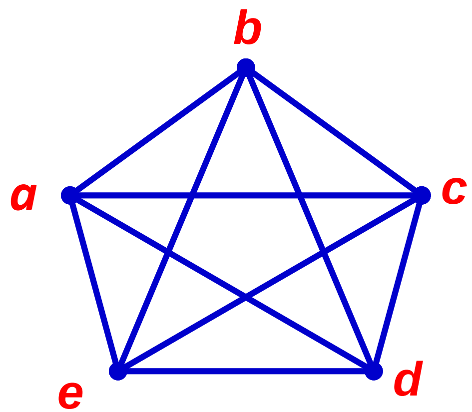


FIGURE 1

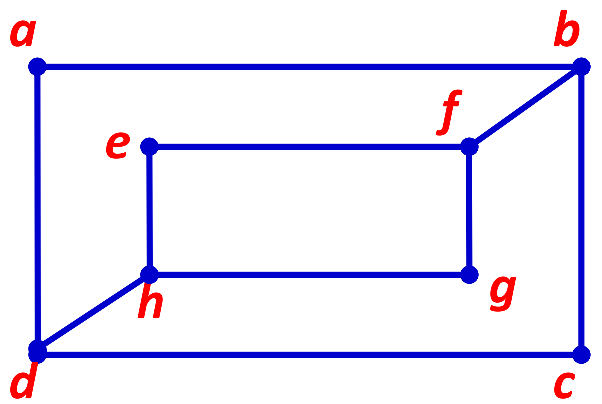


FIGURE 2

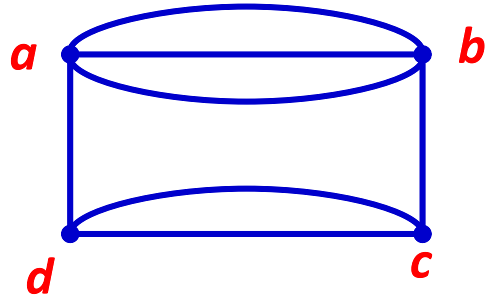


FIGURE 3

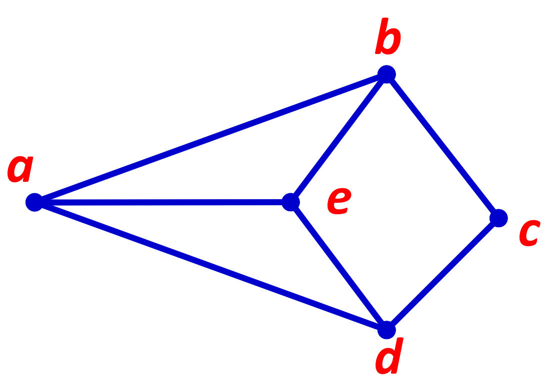


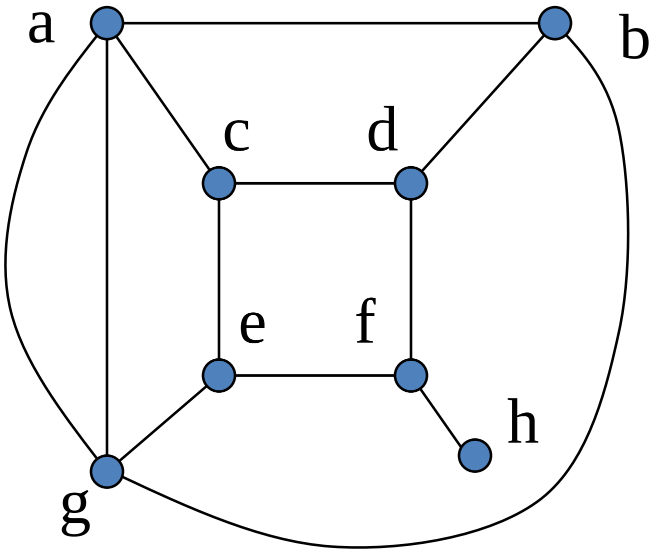
FIGURE 4

Finding a Spanning Tree

- There are two main types of algorithms:
  - breadth-first search
  - depth-first search

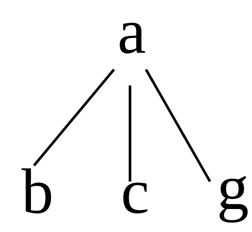
Breadth-first Search

- Process all the verticies at a given level before moving to the next level.
- Example graph G (again):



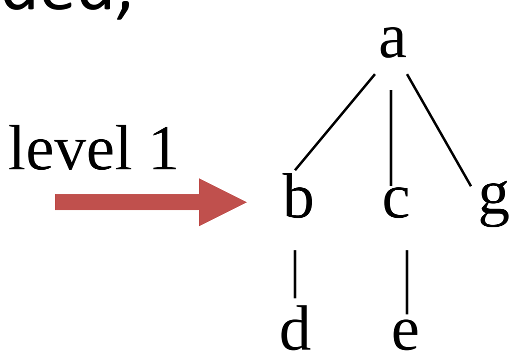
Informal Algorithm

- 1) Put the verticies into an ordering
  - e.g. {a, b, c, d, e, f, g, h}
- 2) Select a vertex, add it to the spanning tree T: e.g. a
- 3) Add to T all edges (a,X) and X verticies that do not create a cycle in T
  - i.e. (a,b), (a,c), (a,g)
    T = {a, b, c, g}

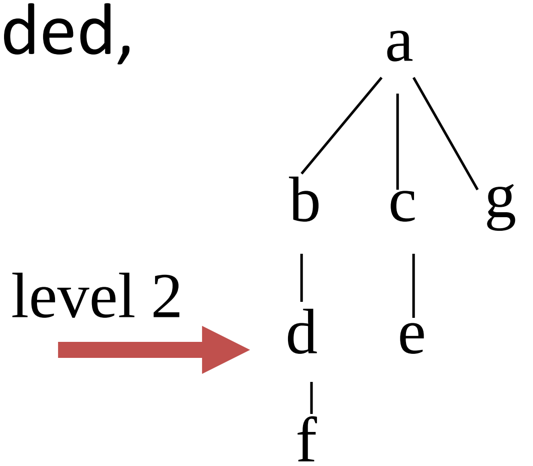


Informal Algorithm

- Repeat step 3 on the verticies just added, these are on level 1
  - i.e. b: add (b,d)
    c: add (c,e)
    g: nothing
    T = {a,b,c,d,e}

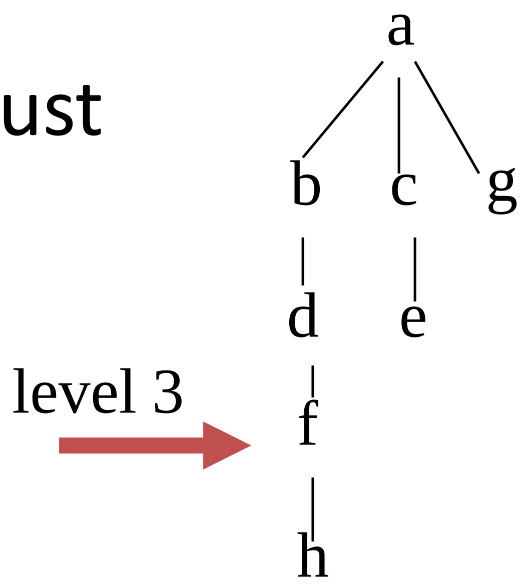


- Repeat step 3 on the verticies just added, these are on level 2
  - i.e. d: add (d,f)
    e: nothing
    T = {a,b,c,d,e,f}



Informal Algorithm

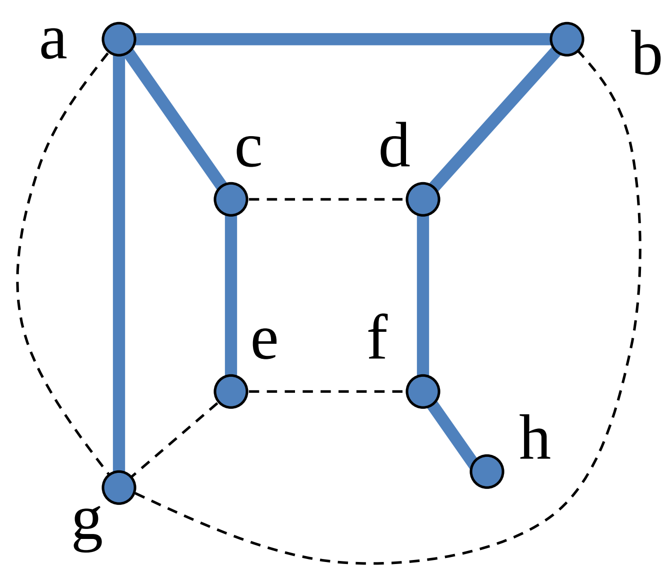
- Repeat step 3 on the verticies just added, these are on level 3
  - i.e. f: add (f,h)
    T = {a,b,c,d,e,f,h}



- Repeat step 3 on the verticies just added, these are on level 4
  - i.e. h: nothing, so stop

Breadth-first Search

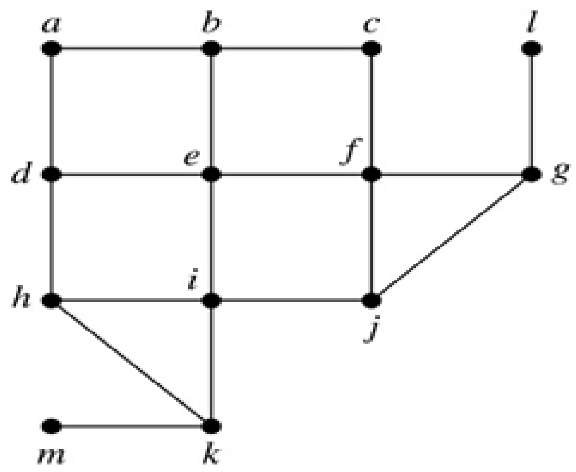
- Resulting spanning tree:



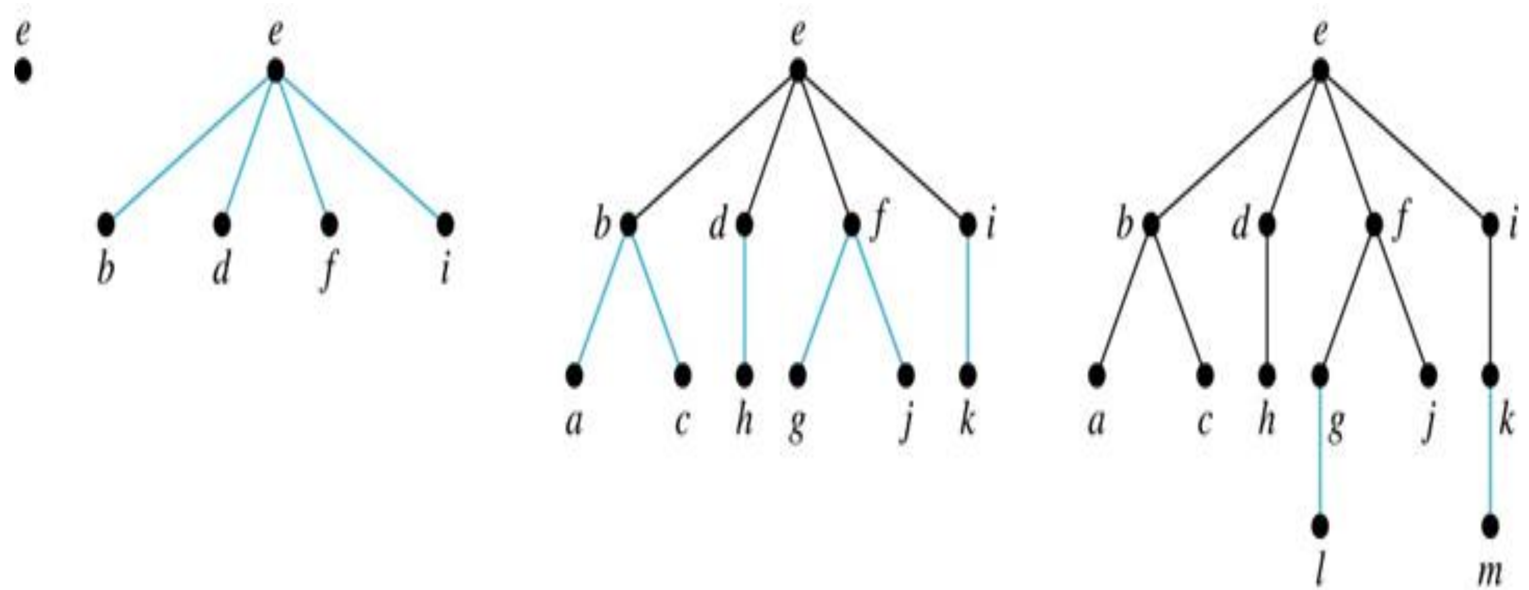
a different spanning tree from the earlier solution

Breadth-First Search

Example: Use breadth-first search to find a spanning tree for this graph.



Breadth-First Search



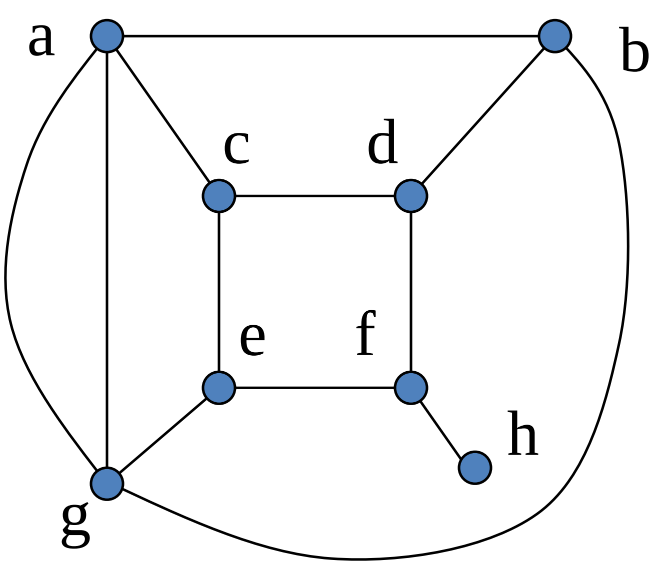
Breadth-First Search Algorithm

- We now use pseudocode to describe breadth-first search.

procedure *BFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
L := empty list *visit*(v_1)
put v_1 in the list L of unprocessed vertices
while L is not empty
    remove the first vertex, v, from L
    for each neighbor w of v
        if w is not in L and not in T **then**
            add w to the end of the list L
            add w and edge $\{v,w\}$ to T

Depth-first Search

- Process all the verticies down one path, then *backtrack* (go back) to verticies along other paths.
- Example graph G (again):

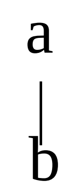


Informal Algorithm

- 1) Put the verticies into an ordering
  - e.g. {a, b, c, d, e, f, g, h}

- 2) Select a vertex, add it to the spanning tree T: e.g. a

- 3) Add the edge (a,X) where X is the smallest vertex in the ordering, and does not make a cycle in T
  - i.e. (a,b), T = {a, b}



Informal Algorithm

- 4) Repeat step 3 with the new vertex, until there is no possible new vertex
  - i.e. add the edges (b,d) (d,c) (c,e) (e,f) (f,h)

    T = {a,b,d,c,e,f,h}

- 5) At this point, there is no (h,X), so backtrack to a vertex that does have another edge:
  - parent of h == f

    but there is no new (f,X) to add, so backtrack

```
a
|
b
|
d
|
c
|
e
|
f
|
h
```

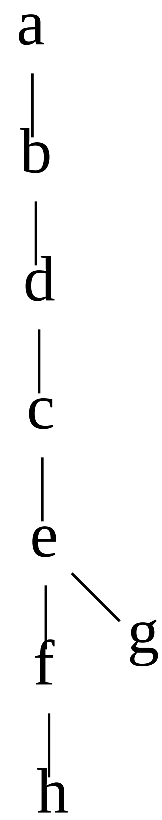
UTD

Informal Algorithm

– parent of f == e

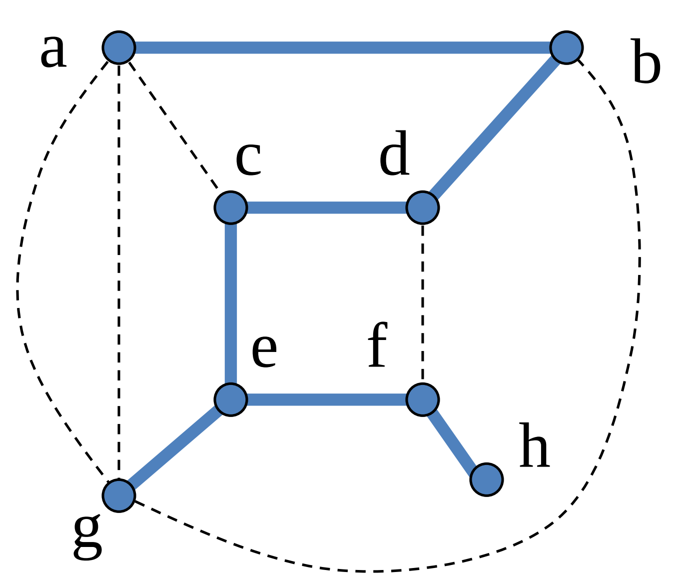
– there is an (e,g) to add, so repeat step 3 with e

- 6) After g is added, there are no further verticies to add, so stop.



Informal Algorithm

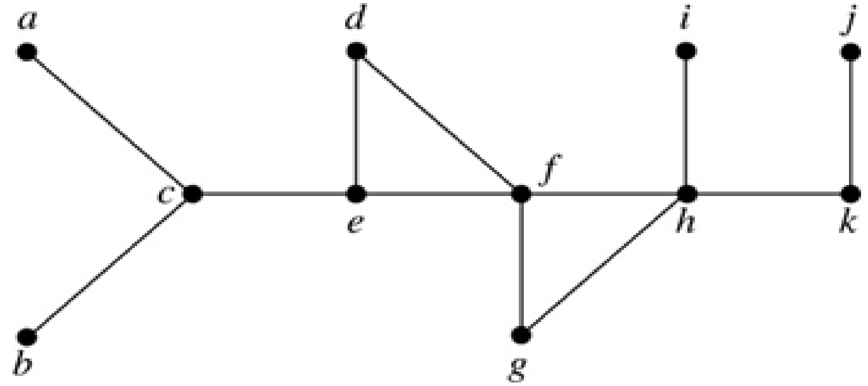
- Resulting spanning tree:



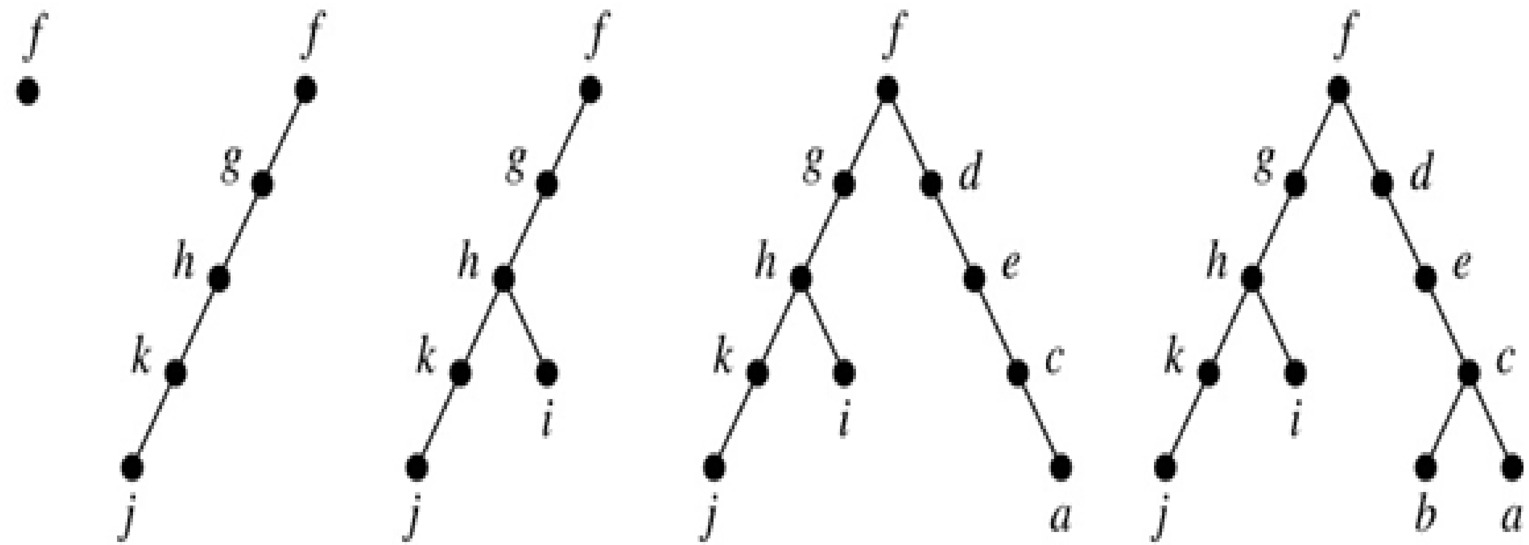
a different spanning tree from the breadth-first solution

Depth-First Search (*continued*)

Example: Use depth-first search to find a spanning tree of this graph.



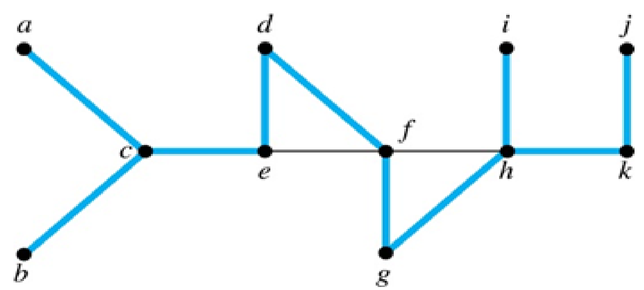
Depth-First Search



(a) (b) (c) (d) (e)

Depth-First Search

- The edges selected by depth-first search of a graph are called *tree edges*. All other edges of the graph must connect a vertex to an ancestor or descendant of the vertex in the graph. These are called *back edges*.
- In this figure, the tree edges are shown with heavy blue lines. The two thin black edges are back edges.



Depth-First Search Algorithm

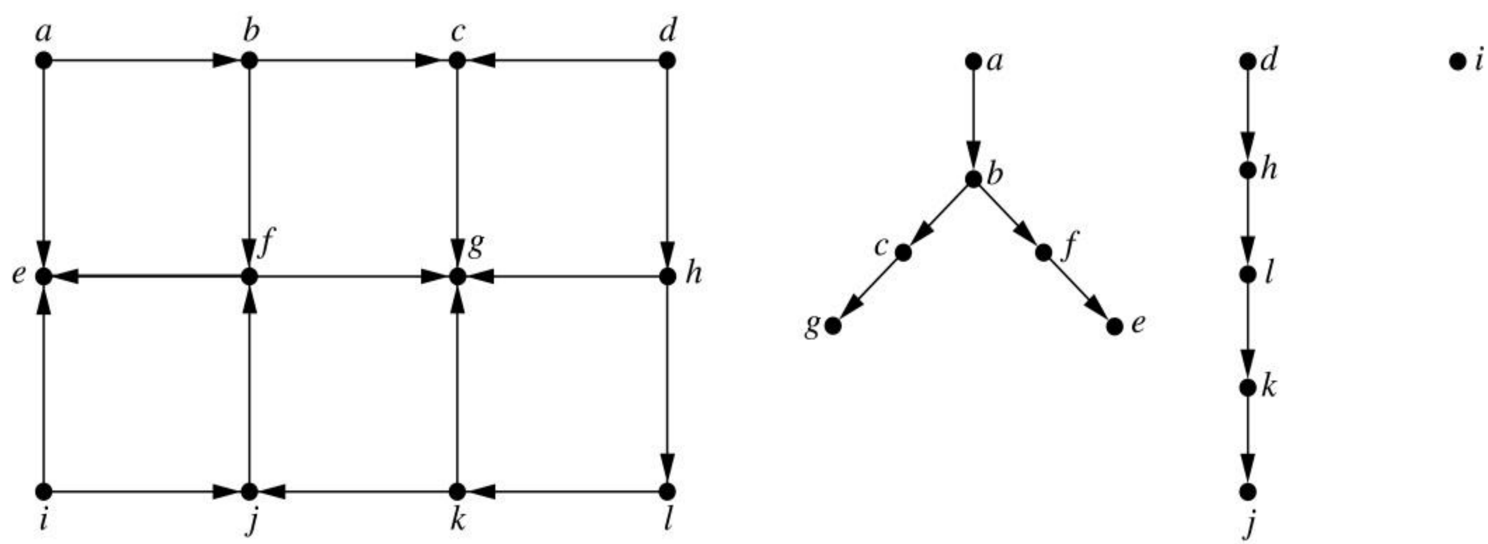
- We now use pseudocode to specify depth-first search. In this recursive algorithm, after adding an edge connecting a vertex *v* to the vertex *w*, we finish exploring *w* before we return to *v* to continue exploring from *v*.

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
visit(v_1)

procedure *visit*(v: vertex of G)
for each vertex w adjacent to v and not yet in T
    add vertex w and edge $\{v,w\}$ to T
    visit(w)

Depth-First Search in Directed Graphs

- Both depth-first search and breadth-first search can be easily modified to run on a directed graph. But the result is not necessarily a spanning tree, but rather a spanning forest.



(a) (b)

Practical Applications: Web Spiders

- Also called crawlers, bots
- Search engines such as Google, Yahoo index websites
- BFS, DFS both used
- Start with an initial web page
- Stop until a page with no new links are found
- Web Graph – web pages are vertices, links are directed edges

BFS & DFS

- BFS uses a queue to store information during tree traversal - queue uses more memory to store pointers
- BFS more memory intensive than DFS
- DFS uses a stack to push nodes onto, stack is LIFO
- DFS less memory intensive than BFS
- BFS finds shortest paths, in the sense of fewest edges
- Edges in the original graph not in BFS tree are cross edges, not in DFS tree are back edges, useful for problems such as special colorings

Spanning Trees

Find a spanning tree for the following graphs using BFS and DFS.

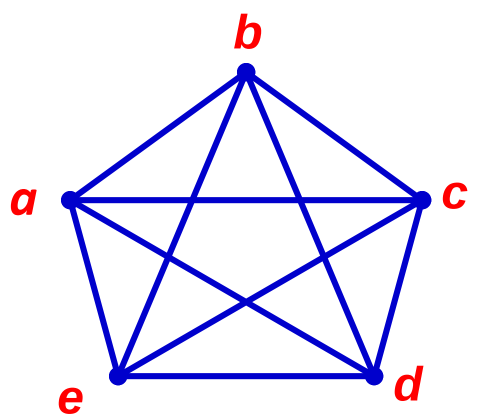


FIGURE 1

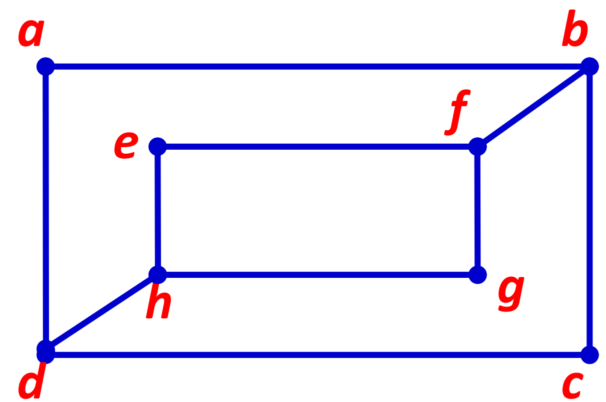


FIGURE 2

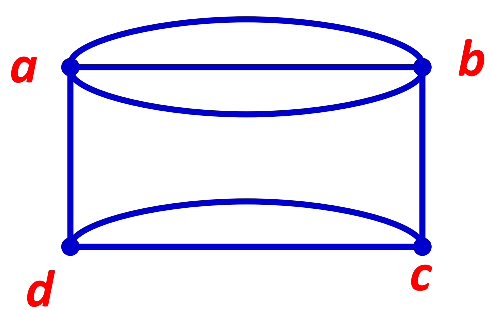


FIGURE 3

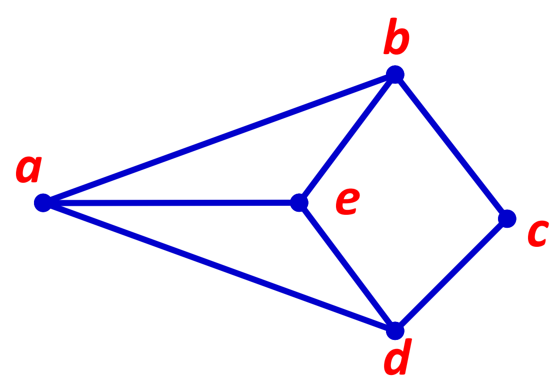


FIGURE 4