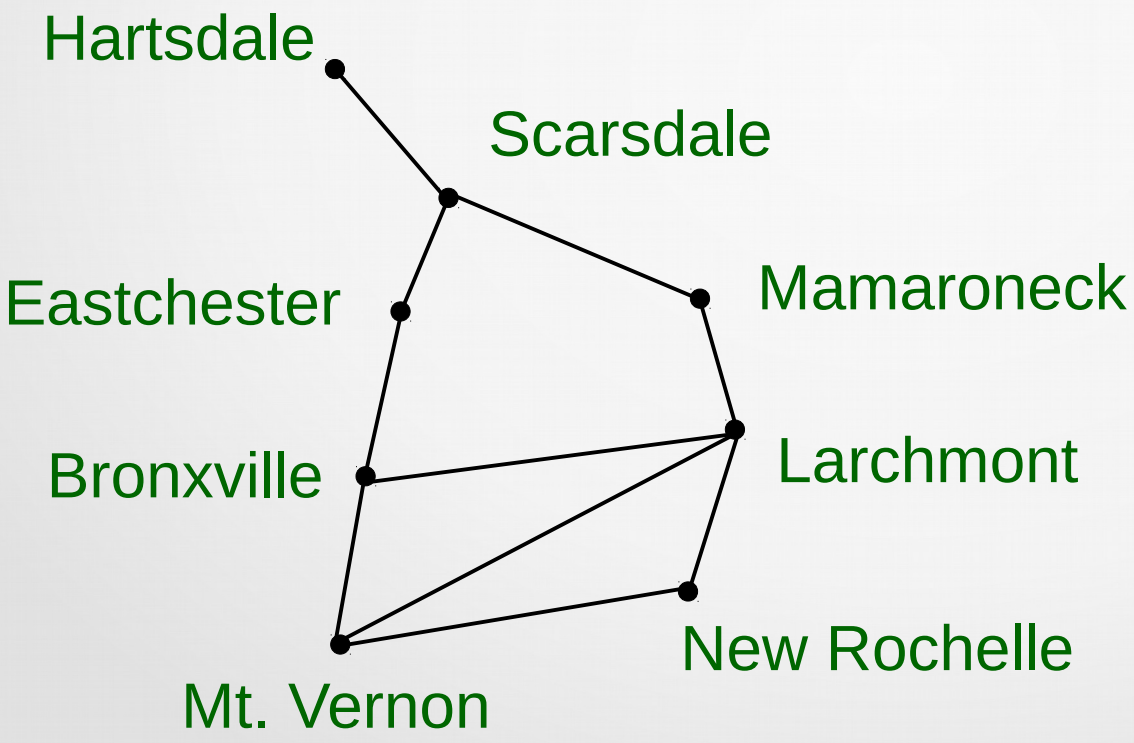
11.4 *Spanning Trees*

Plowing example

Consider a part of the system of roads in NY state, represented by the simple graph.

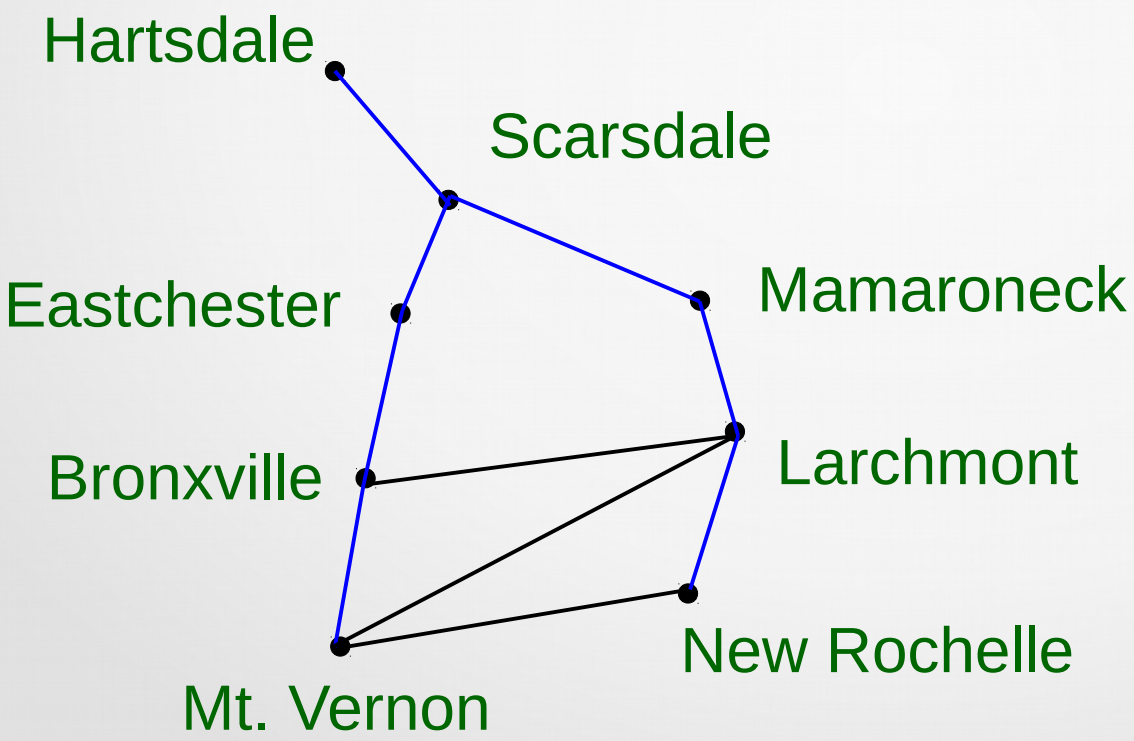


In winter the roads have to be plowed to be kept open. The highway department wants to plow the fewest roads so that there will always be cleared roads connecting any two towns.

11.4 *Spanning Trees*

Plowing example

Consider a part of the system of roads in NY state, represented by the simple graph.



In winter the roads have to be plowed to be kept open. The highway department wants to plow the fewest roads so that there will always be cleared roads connecting any two towns.

We can see that at least 7 roads must be plowed.

11.4 *Spanning Trees*

[Def] 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.

11.4 *Spanning Trees*

[Def] 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.

Note that a simple graph with a spanning tree must be connected. The converse is also true: every connected simple graph has a spanning tree.

11.4 *Spanning Trees*

[Def] 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.

Note that a simple graph with a spanning tree must be connected. The converse is also true: every connected simple graph has a spanning tree.

How to get a spanning tree from a connected simple graph?

11.4 *Spanning Trees*

[Def] 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.

Note that a simple graph with a spanning tree must be connected. The converse is also true: every connected simple graph has a spanning tree.

How to get a spanning tree from a connected simple graph? by removing edges

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.
If G is a tree, then G is also a spanning tree.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.
If G is a tree, then G is also a spanning tree.
If G is not a tree, then it must contain at least one simple circuit.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.
If G is a tree, then G is also a spanning tree.
If G is not a tree, then it must contain at least one simple circuit. Remove an edge from one of these simple circuits.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.
If G is a tree, then G is also a spanning tree.
If G is not a tree, then it must contain at least one simple circuit. Remove an edge from one of these simple circuits. The resulting graph is still connected and has all the vertices.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.
If G is a tree, then G is also a spanning tree.
If G is not a tree, then it must contain at least one simple circuit. Remove an edge from one of these simple circuits. The resulting graph is still connected and has all the vertices. If the resulting graph is not a tree, then there is at least one simple circuit; so as before, remove an edge from any of the circuits.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.
If G is a tree, then G is also a spanning tree.
If G is not a tree, then it must contain at least one simple circuit. Remove an edge from one of these simple circuits. The resulting graph is still connected and has all the vertices. If the resulting graph is not a tree, then there is at least one simple circuit; so as before, remove an edge from any of the circuits. The process terminates when no simple circuits are left.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(→) Assume a simple graph G is connected.
If G is a tree, then G is also a spanning tree.
If G is not a tree, then it must contain at least one simple circuit. Remove an edge from one of these simple circuits. The resulting graph is still connected and has all the vertices. If the resulting graph is not a tree, then there is at least one simple circuit; so as before, remove an edge from any of the circuits. The process terminates when no simple circuits are left. A tree is produced because the graph stays connected, and it contains every vertex of G.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(←) Assume a simple graph G has a spanning tree T.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(←) Assume a simple graph G has a spanning tree T. T contains all the vertices of G and there is a path between any two vertices in T.

11.4 *Spanning Trees*

[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(←) Assume a simple graph G has a spanning tree T. T contains all the vertices of G and there is a path between any two vertices in T. T is a subgraph of G, no new edges are added.

11.4 *Spanning Trees*

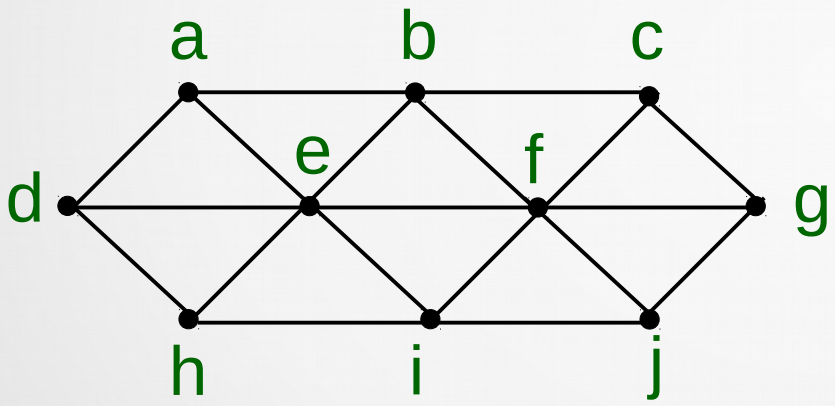
[Theorem 1] A simple graph is connected iff it has a *spanning tree*.

Proof:
(←) Assume a simple graph G has a spanning tree T. T contains all the vertices of G and there is a path between any two vertices in T. T is a subgraph of G, no new edges are added. Therefore there is a path between any two vertices in G as well.

q.e.d.

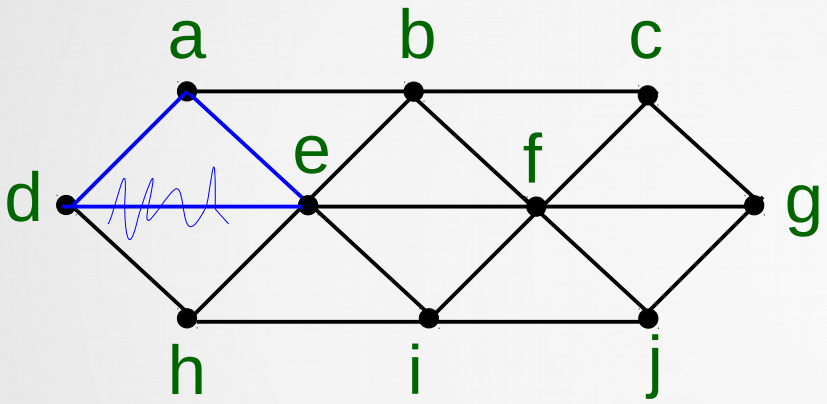
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



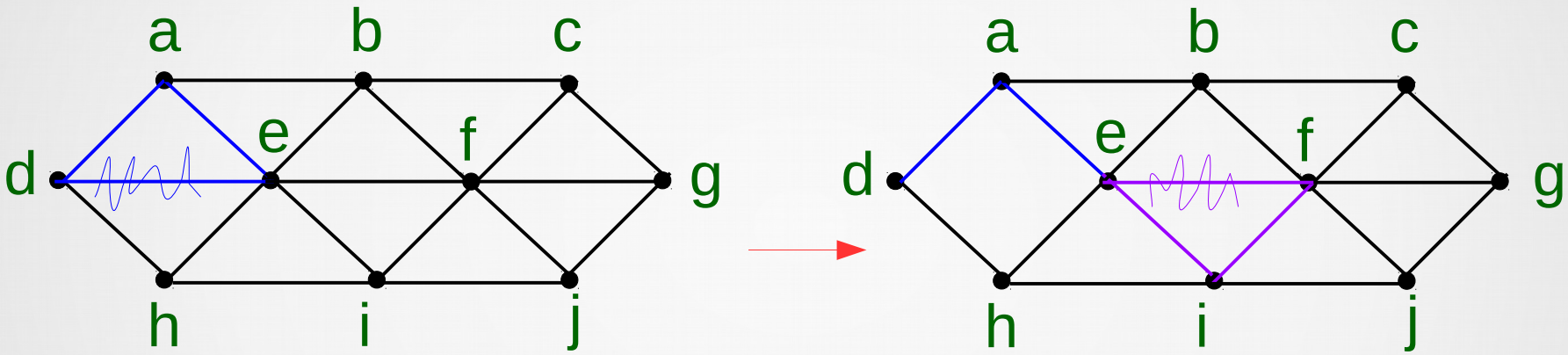
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



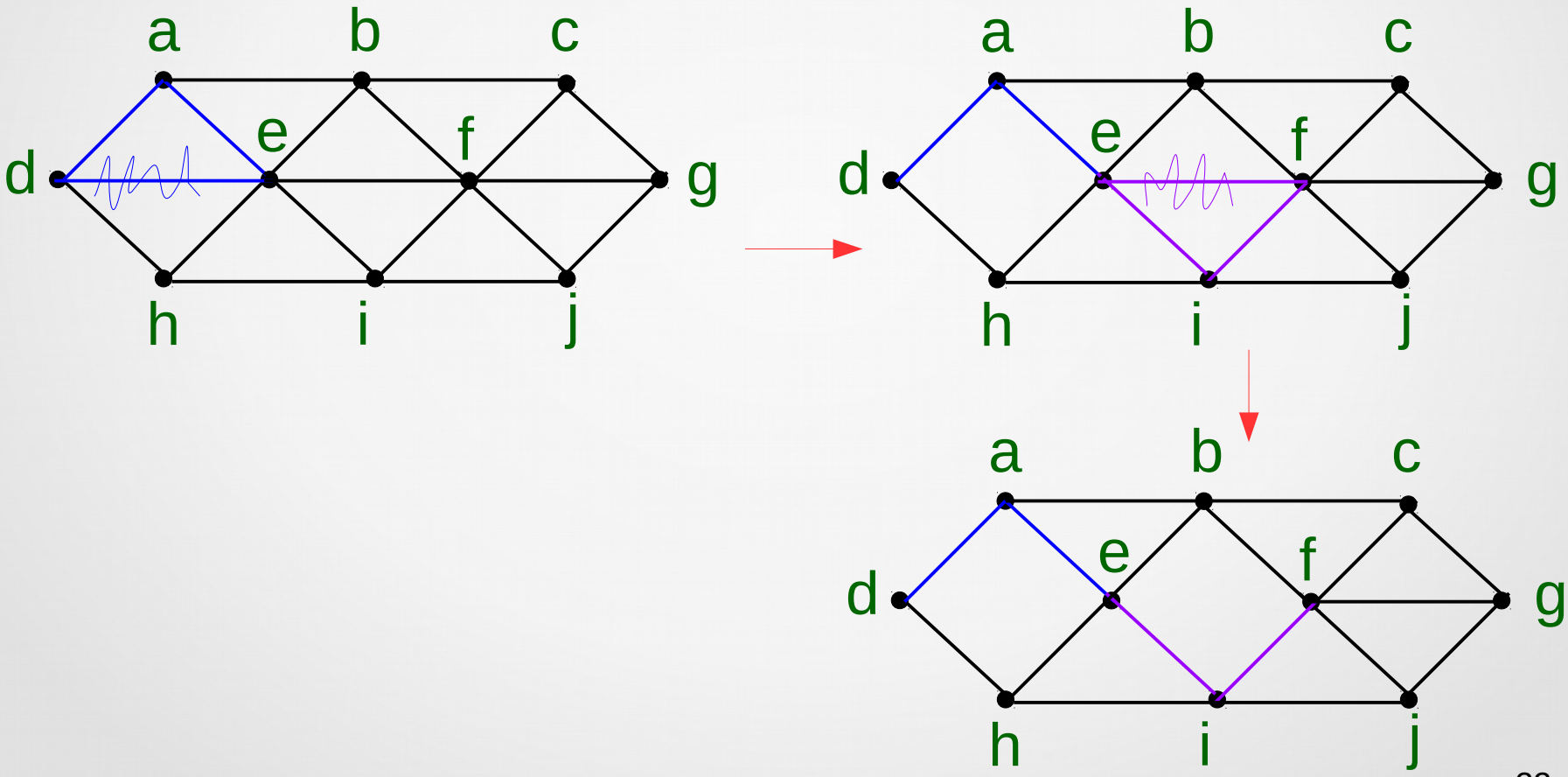
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



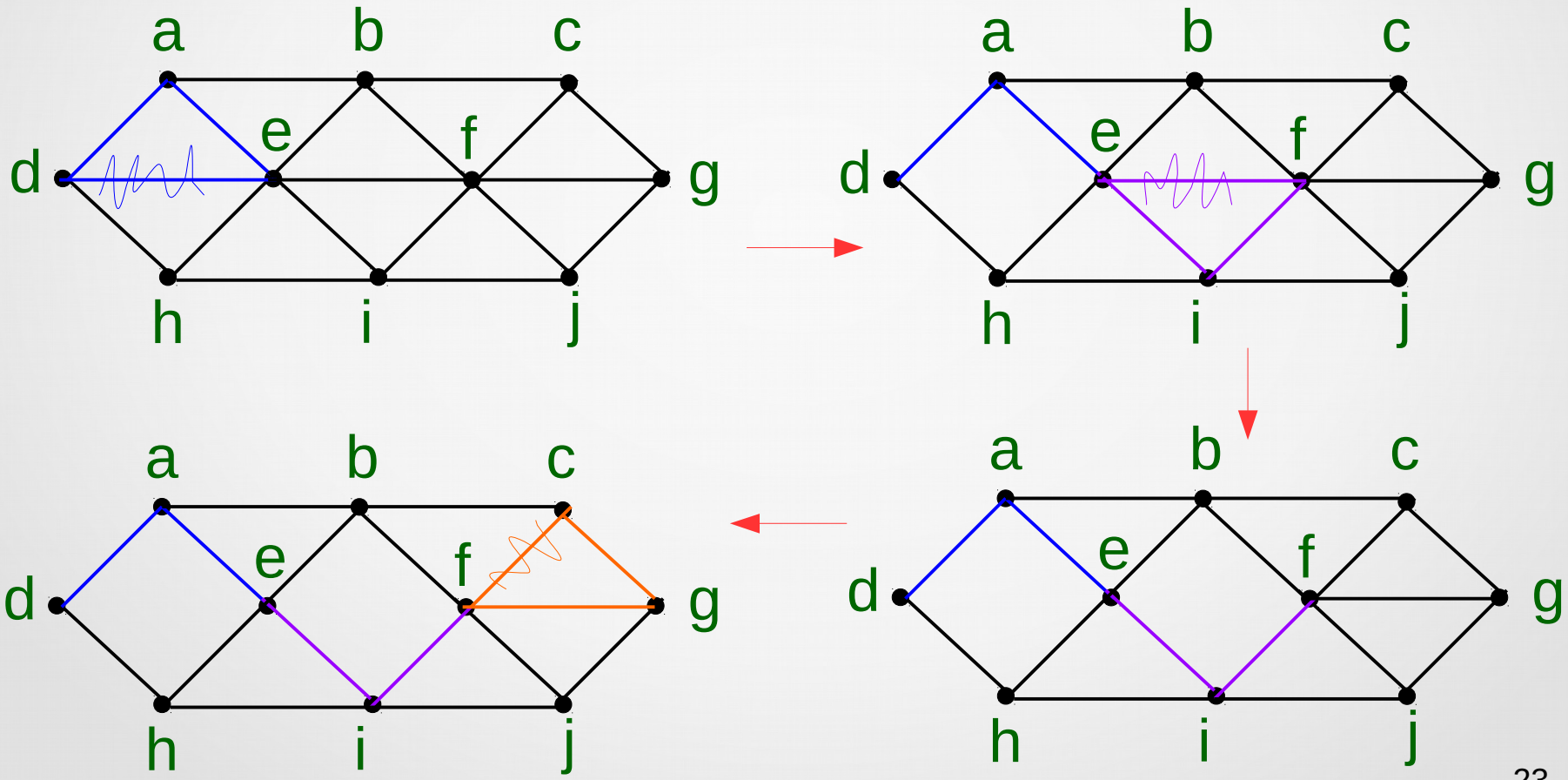
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



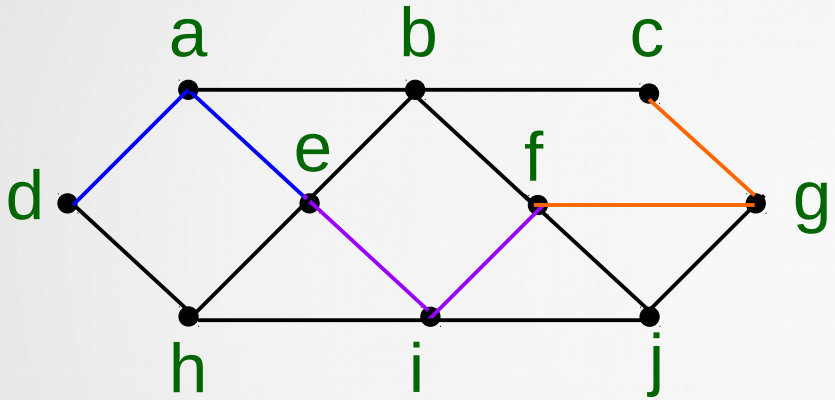
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



11.4 *Spanning Trees*

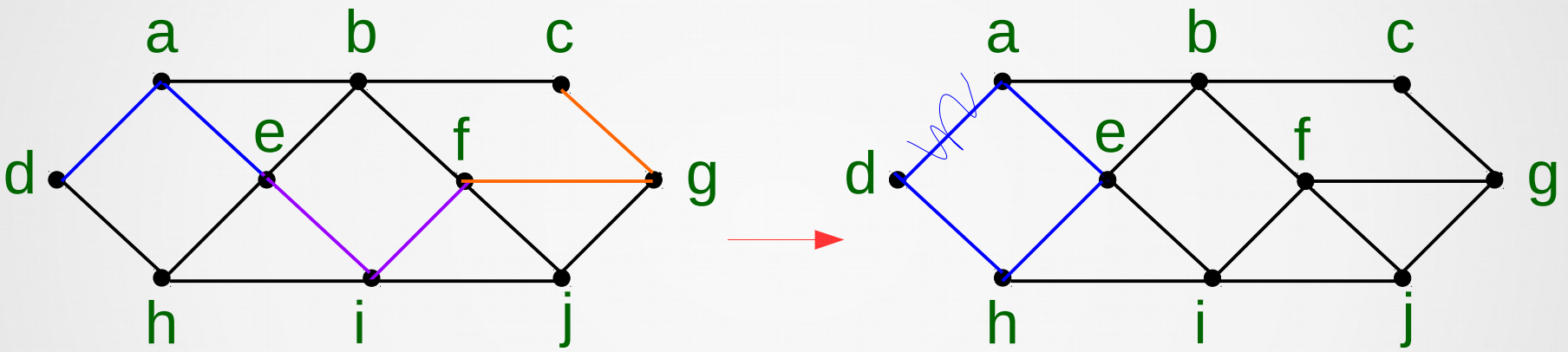
Example: Find a spanning tree of the following graph.



a b c

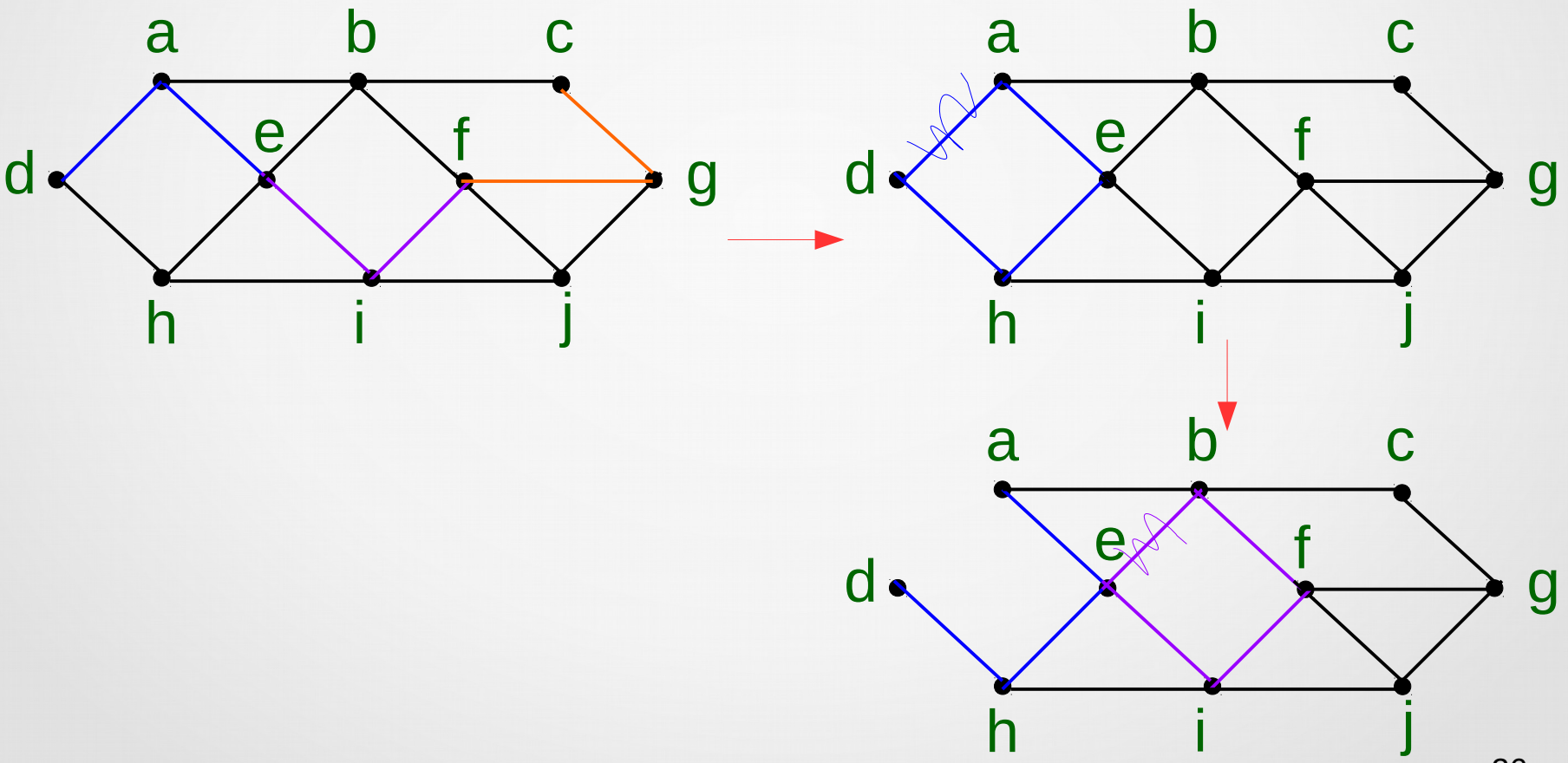
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



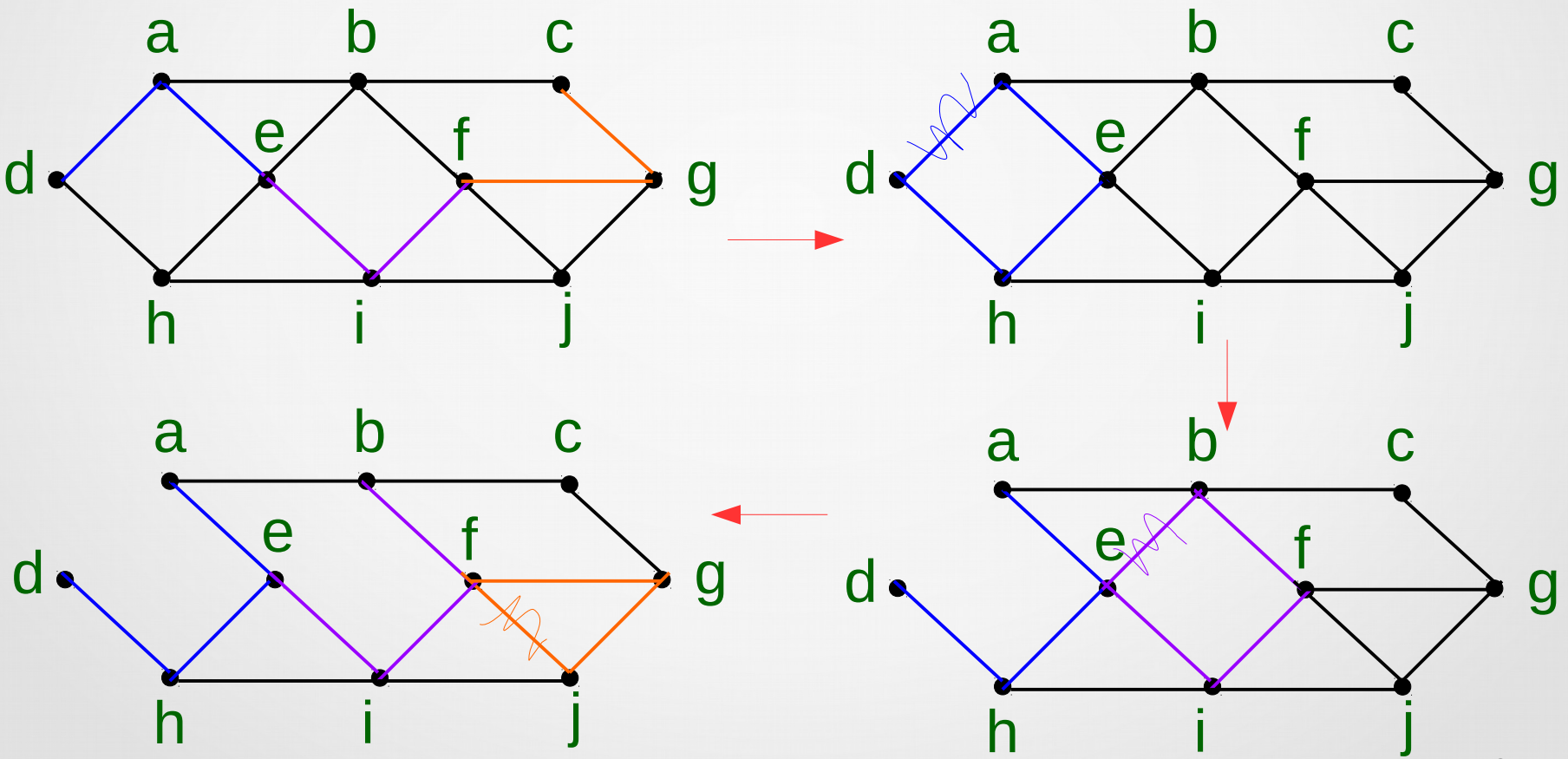
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



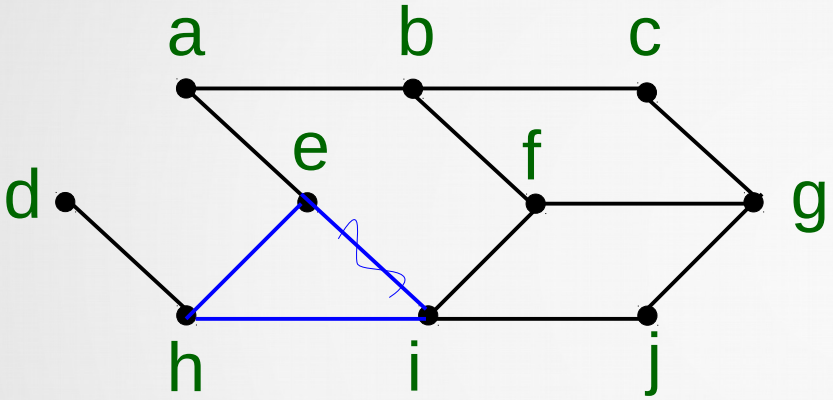
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



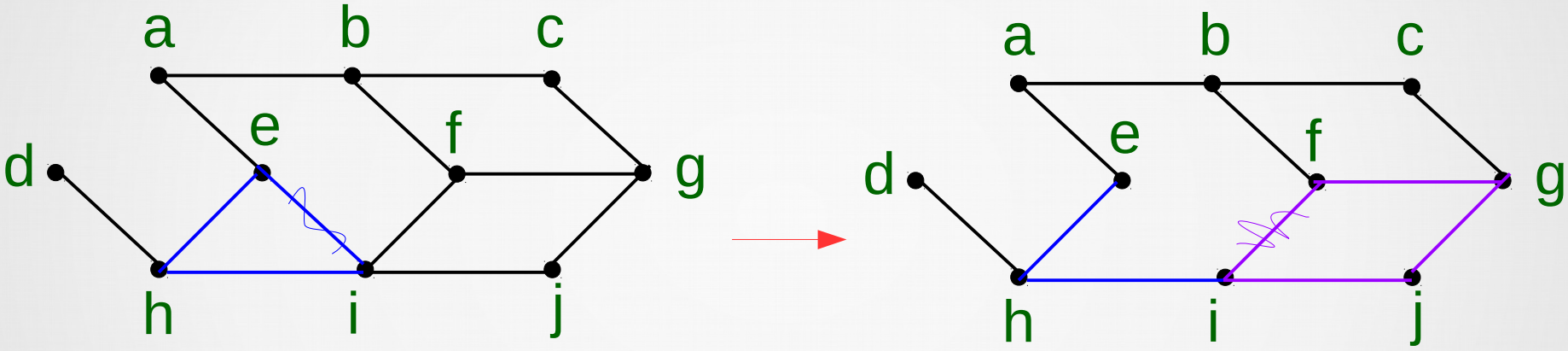
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



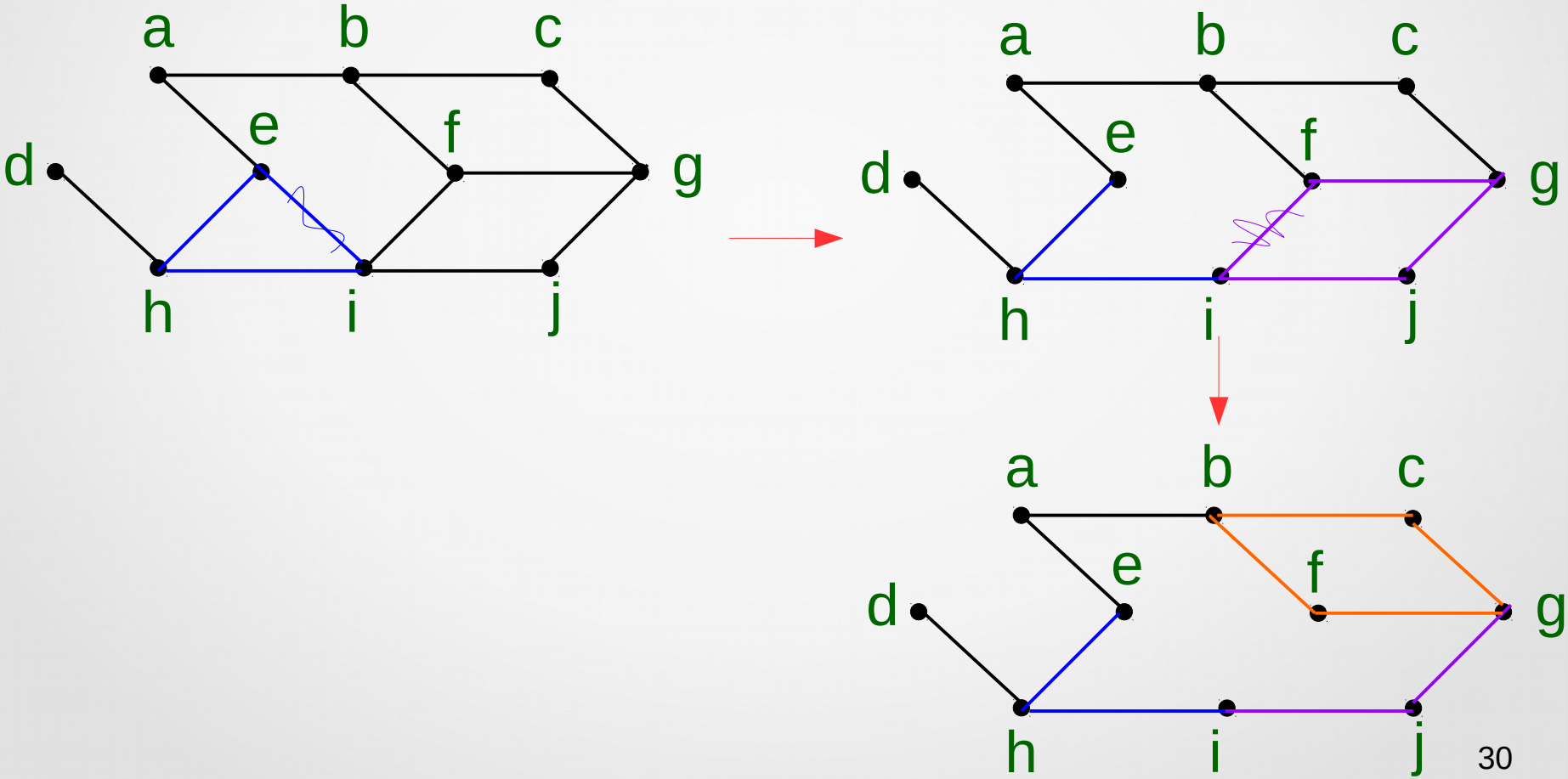
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



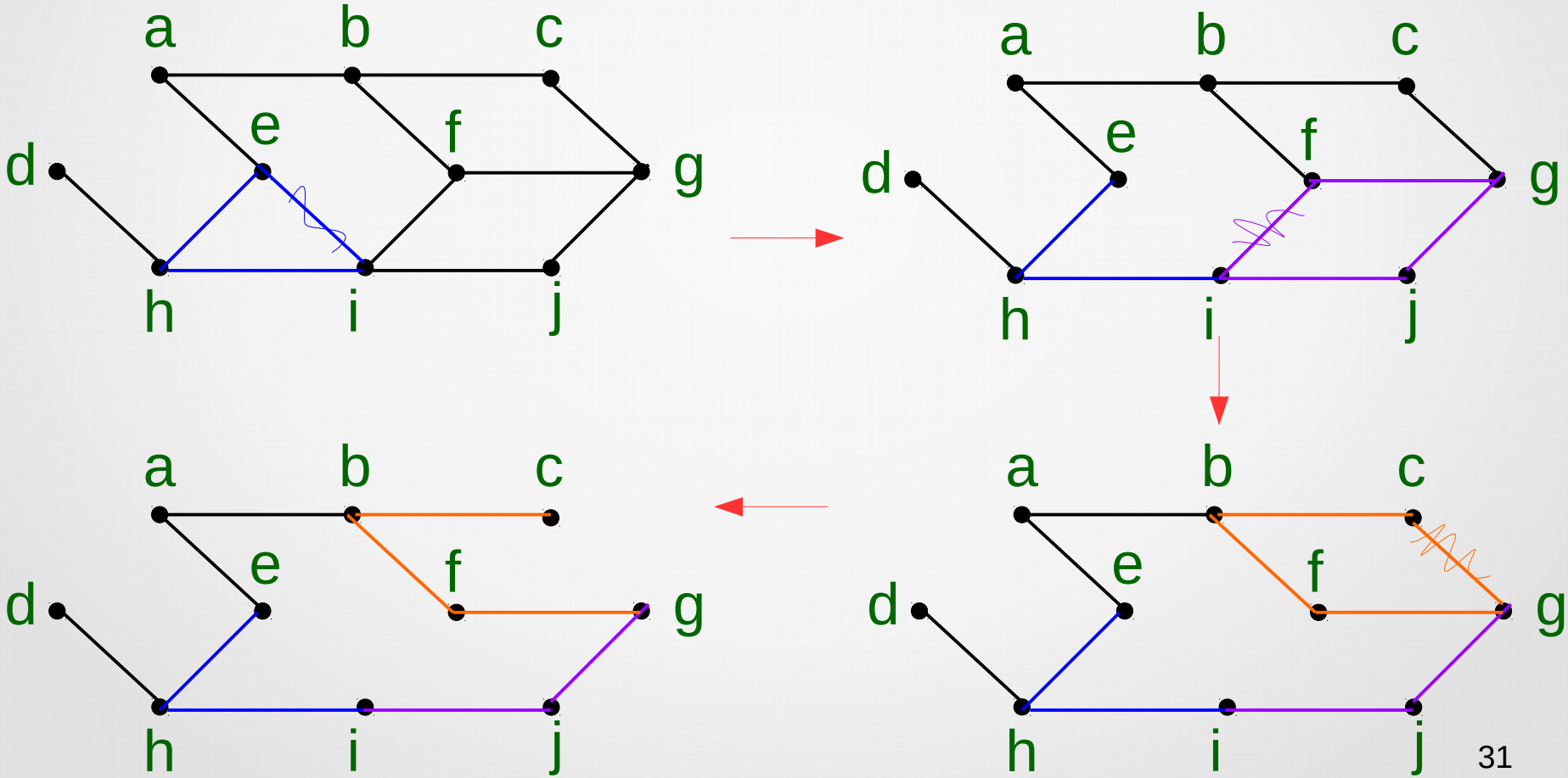
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



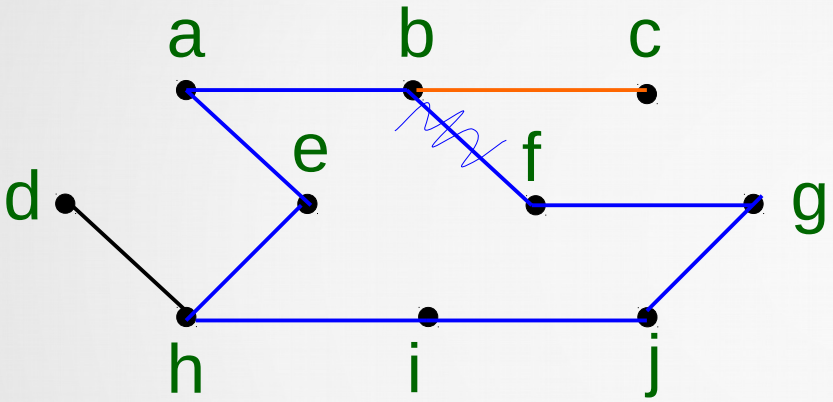
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



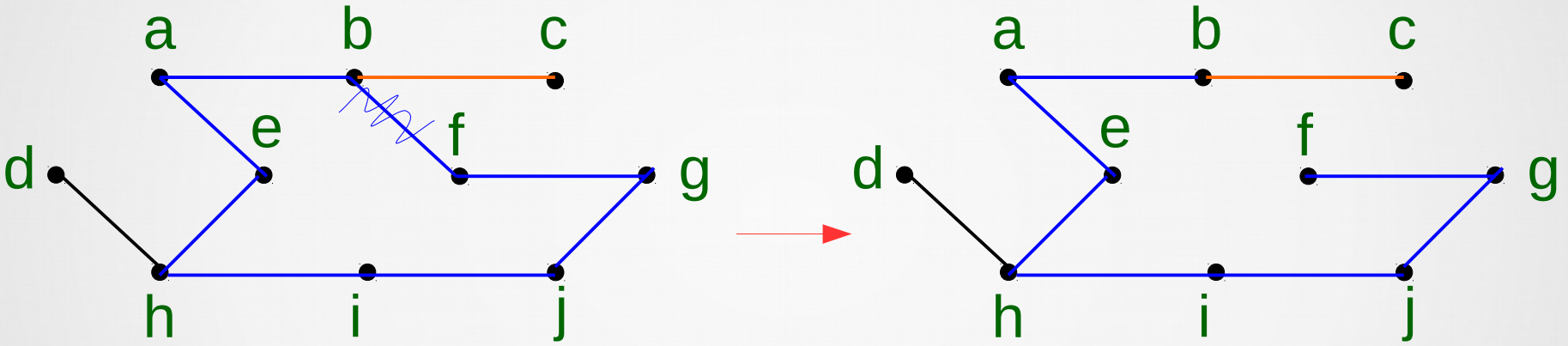
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



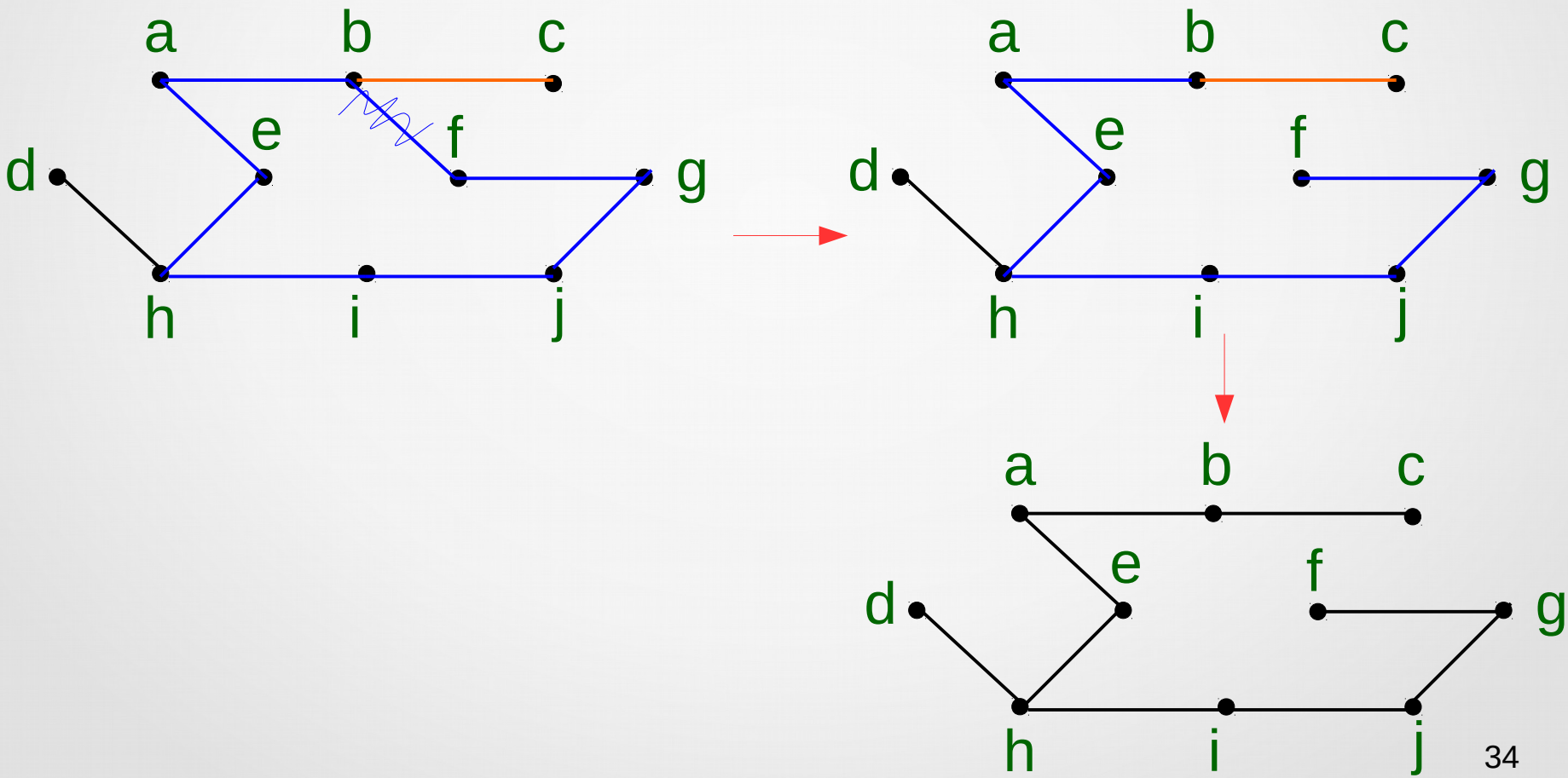
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



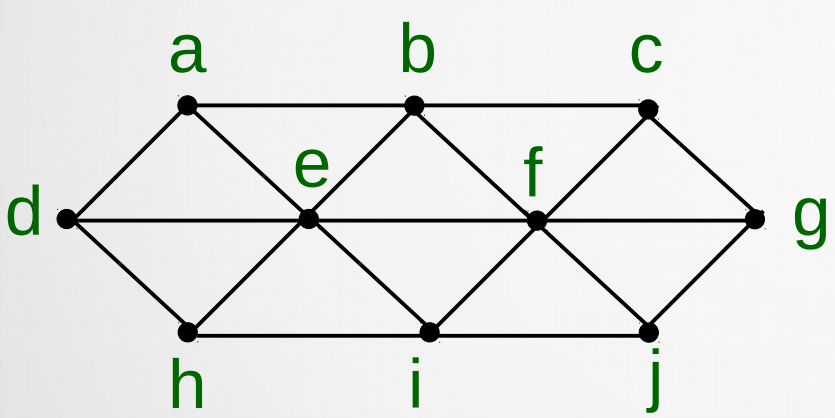
11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.

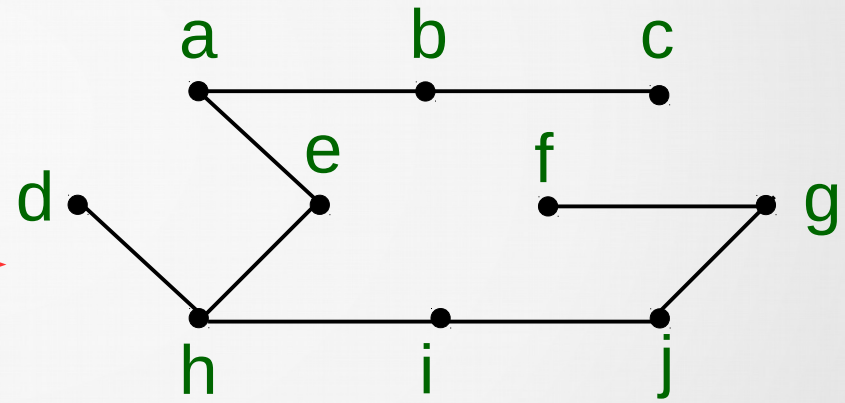


11.4 *Spanning Trees*

Example: Find a spanning tree of the following graph.



graph G

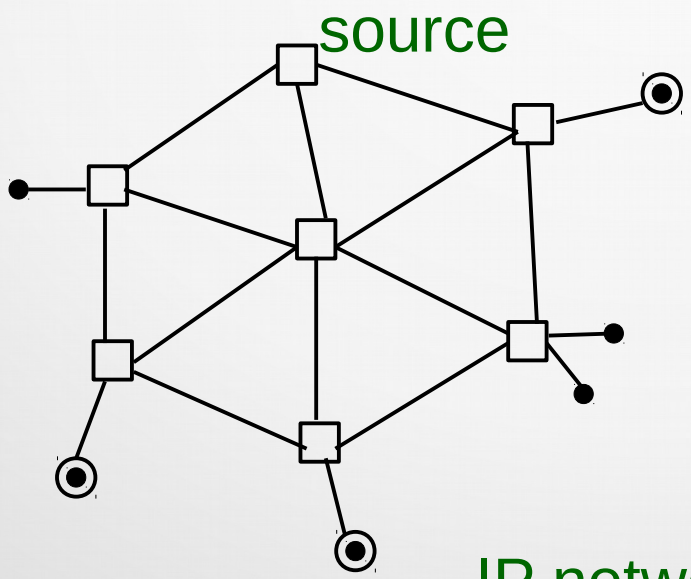


a spanning tree T of graph G

11.4 *Spanning Trees*

Example 2: IP Multicasting

Assume we want to send data from *source computer* to multiple *receiving computers* (each of which is a subnetwork).



router

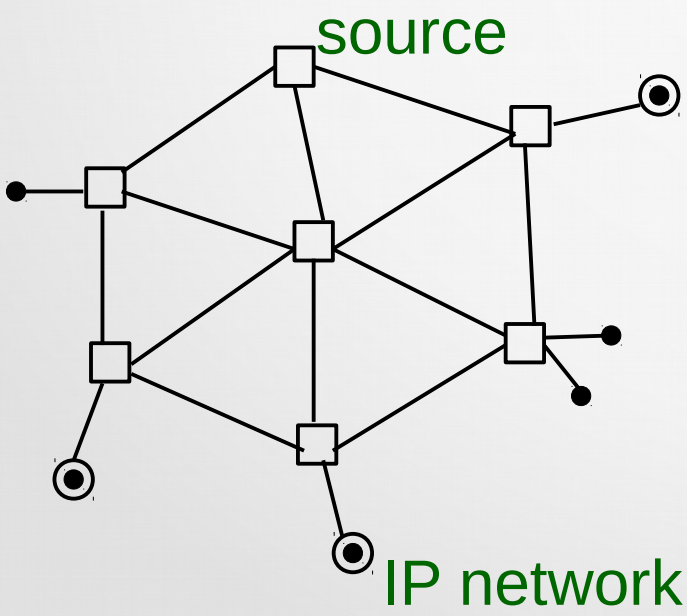
subnetwork

subnetwork with a receiving station

11.4 *Spanning Trees*

Example 2: IP Multicasting

Assume we want to send data from *source computer* to multiple *receiving computers* (each of which is a subnetwork).



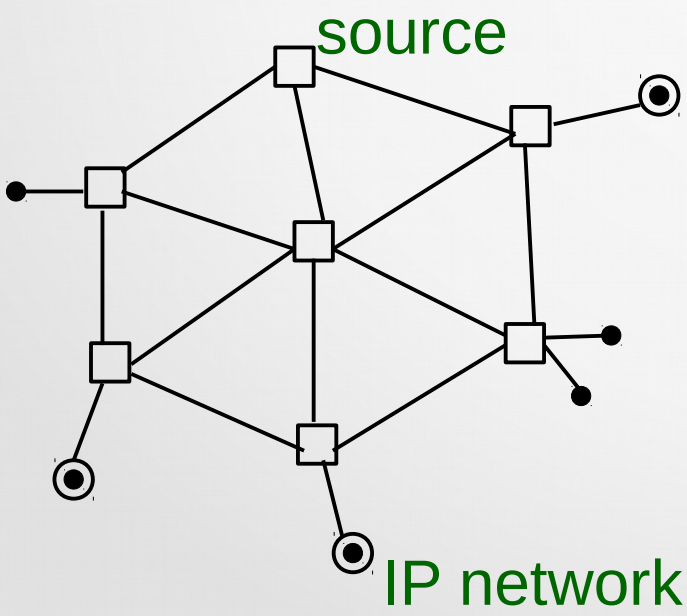
Data can be sent to each computer separately (unicasting) – many copies of the same data are transmitted over the network

- □ router
- • subnetwork
- ◉ subnetwork with a receiving station

11.4 *Spanning Trees*

Example 2: IP Multicasting

Assume we want to send data from *source computer* to multiple *receiving computers* (each of which is a subnetwork).



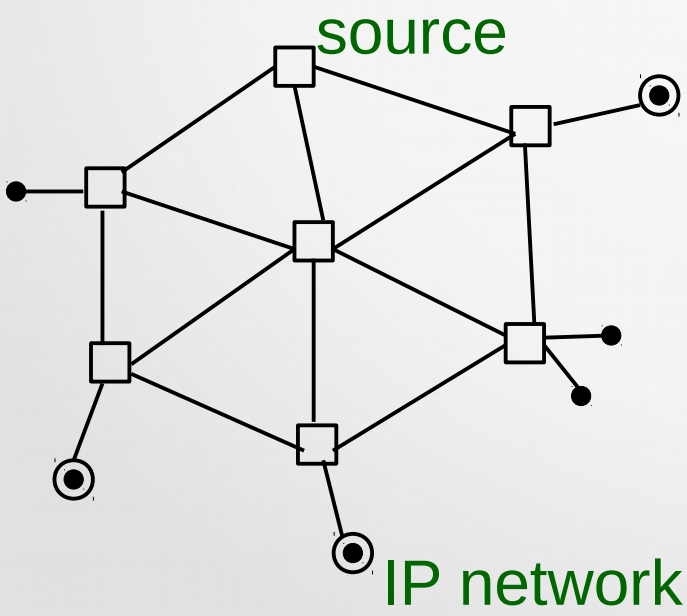
A computer sends one copy of data over the network, interme-diate routers forward data to more other routers. (multicasting)

- router
- subnetwork
- subnetwork with a receiving station

11.4 *Spanning Trees*

Example 2: IP Multicasting

Assume we want to send data from *source computer* to multiple *receiving computers* (each of which is a subnetwork).



A computer sends one copy of data over the network, interme-diate routers forward data to more other routers. (multicasting)

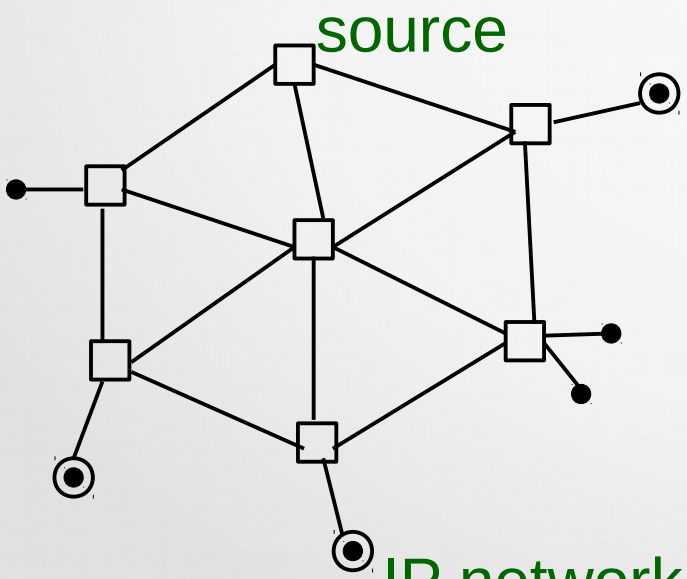
For fast delivery: no loops (circuits, cycles) in the path!

Multicast routers use network algorithms to construct a spanning tree (root is the source).

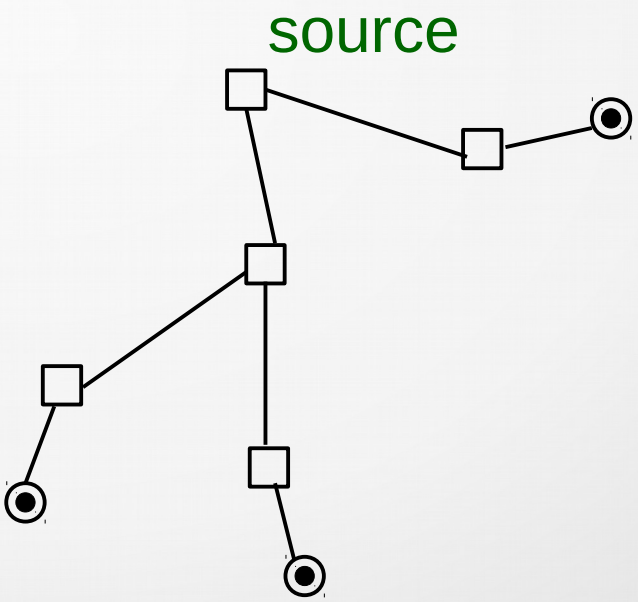
11.4 *Spanning Trees*

Example 2: IP Multicasting

Assume we want to send data from *source computer* to multiple *receiving computers* (each of which is a subnetwork).



IP network



Multicast spanning tree

11.4 *Spanning Trees*

Depth-First Search and Breadth-First Search

The proof of the **Theorem 1** gives an algorithm for finding spanning trees by removing edges from simple circuits.

It is inefficient because simple circuits should be identified first.

Alternative: let's build spanning trees by adding edges.

We will consider two algorithms:

- Depth-First Search, and
- Breadth-First Search

11.4 *Spanning Trees*

Depth-First Search

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

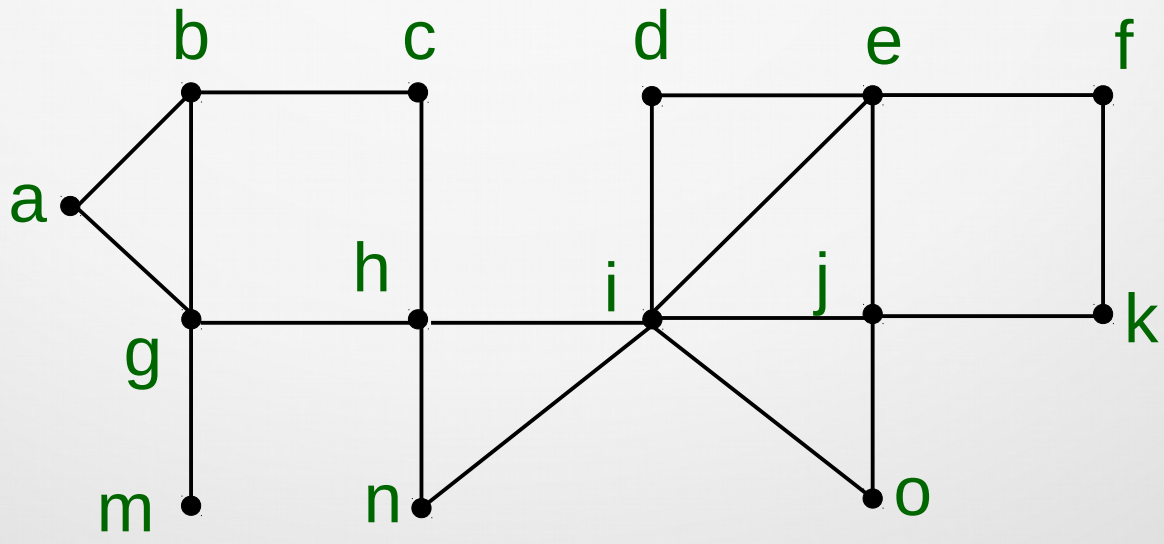
procedure *visit*(v: vertex of *G*)
for each vertex w adjacent to v and not yet in T,
    add vertex w and edge {v,w} to T
    visit(w)

11.4 *Spanning Trees*

Depth-First Search

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



11.4 *Spanning Trees*

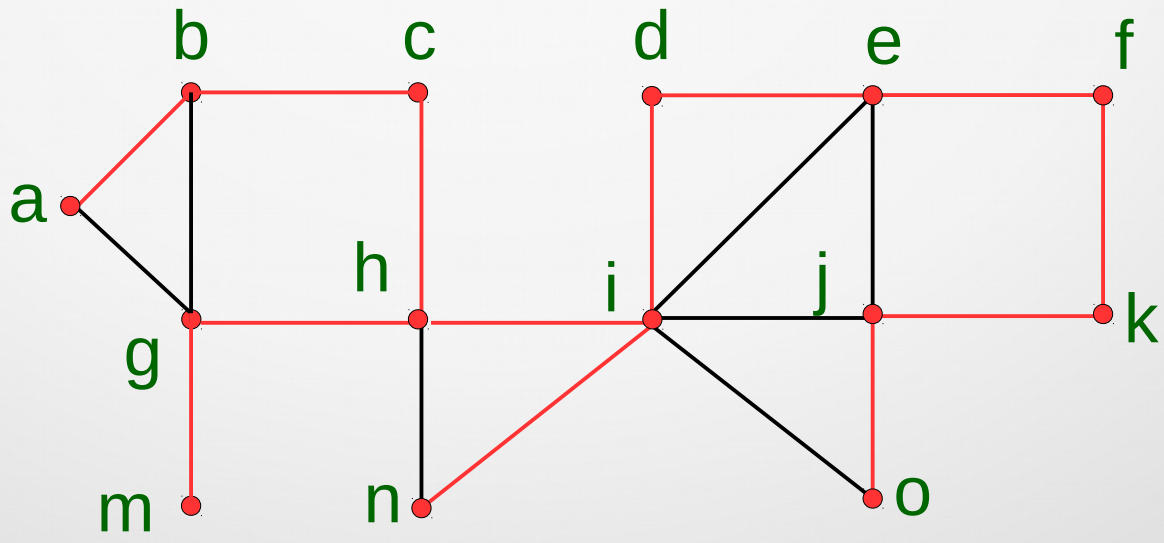
Depth-First Search

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

T := tree consisting only of 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)



11.4 *Spanning Trees*

Breadth-First Search

procedure *BFS*(*G*: connected graph with vertices v_1, v_2, …, v_n)

T := tree consisting only of vertex v_1

L := empty list

put v_1 in the list L of unprocessed vertices

while L is not empty

    remove the first vertex, v, from L

    for each neighbor w of v

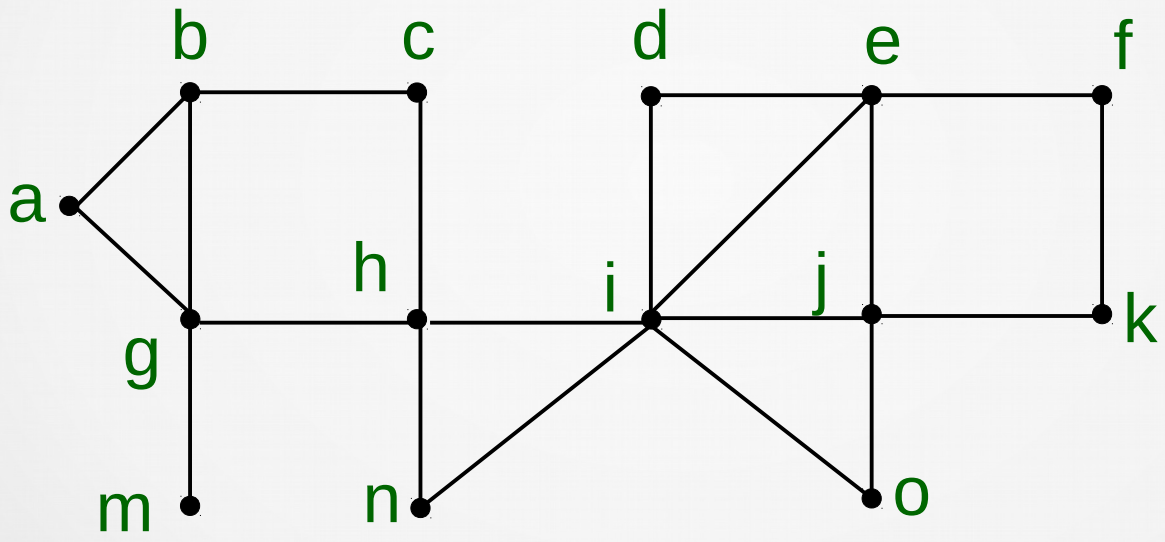
        if w is not in L and not in T **then**

            add w to the end of the list L

            add w and edge {v,w} to T

11.4 *Spanning Trees*

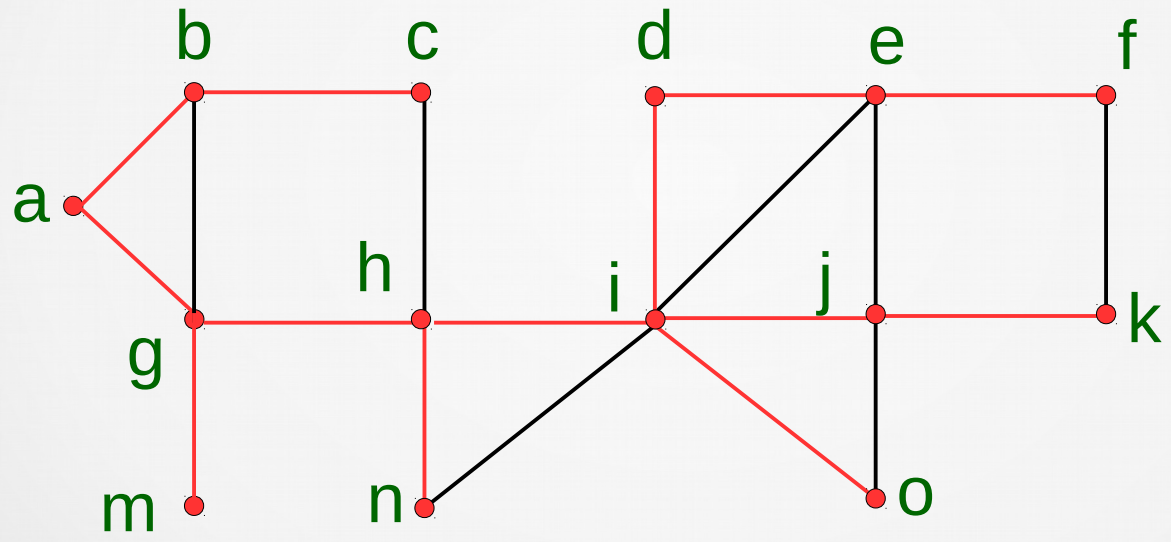
Breadth-First Search



L =

11.4 *Spanning Trees*

Breadth-First Search



L = { a, b, g, c, h, m, i, n, d, j, o, e, k, f }

11.4 *Spanning Trees*

Depth-First Search and Breadth-First Search

DFS algorithm can be used in algorithms that
- find *paths* and *circuits* in a graph
- determine the *connected components* of a given graph
- find the *cut vertices* of a connected graph

BFS algorithm can be used in algorithms that
- find the *connected components* of a given graph
- determine whether the *graph is bipartite*
- find the *path with the fewest edges* between the two vertices

11.4 *Spanning Trees*

DFS and BFS complexity

For both algorithms we assume that list adjacency for G is given.

Both DFS and BFS construct spanning tree using O(|E|) or $O(n^2)$ steps.

- each edge is explored at most twice
- for any simple graph $|E| \leq \frac{n(n-1)}{2}$

11.4 *Spanning Trees*

Backtracking applications

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

One way is to use a decision tree, with
internal vertices: decisions
leaves: possible solutions

To find a solution using backtracking:
1) make a sequence of decision in attempt to reach a solution (as long as possible) – a path in the decision tree
2) once no solution can result from any further sequence of decisions, backtrack to the parent of the current vertex, work toward a solution with another series of decisions (if possible)
3) The procedure continues until a solutio is found, or it is established that no solution exists.

11.4 *Spanning Trees*

Backtracking applications

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

11.4 *Spanning Trees*

Backtracking applications

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

One way is to use a decision tree, with
internal vertices: decisions
leaves: possible solutions

11.4 *Spanning Trees*

Backtracking applications

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

One way is to use a decision tree, with
internal vertices: decisions
leaves: possible solutions

To find a solution using backtracking:
1) make a sequence of decision in attempt to reach a solution (as long as possible) – a path in the decision tree
2) once no solution can result from any further sequence of decisions, backtrack to the parent of the current vertex, work toward a solution with another series of decisions (if possible)
3) The procedure continues until a solutio is found, or it is established that no solution exists.

11.4 *Spanning Trees*

Backtracking applications

Example Graph colorings

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

11.4 *Spanning Trees*

Backtracking applications

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

Algorithm:
1) pick some vertex *a*, assign it color 1
2) pick a second vertex *b*
If *b* is adjacent to *a*, assign color 2 to *b*
Otherwise, assign color 1 to *b*
3) Pick a third vertex *c*
Use color 1 for vertex *c* if possible
Otherwise, use color 2 is possible.
If neither colors can be used, color 3 should be used.

11.4 *Spanning Trees*

Backtracking applications

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

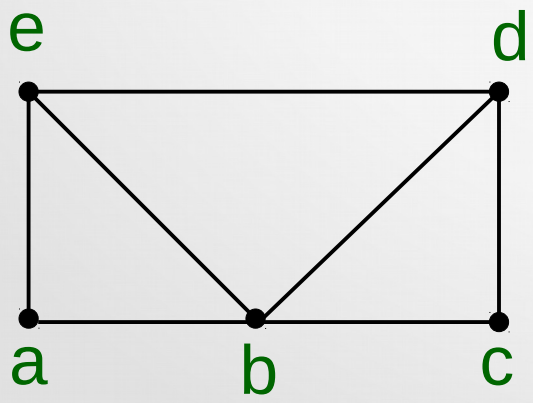
Algorithm:
4) Continue the process as long as it is possible
5) If a vertex is reached that cannot be colored by any of the *n* colors, backtrack to its closest ancestor where it is possible to change a coloring vertex. Continue assigning colors of additional vertices as long as possible.
6) If a coloring using *n* colors exists, backtracking will produce it.

11.4 *Spanning Trees*

Backtracking applications

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

Let's try to color the following graph with 3 colors: red, blue and green using the backtracking procedure.

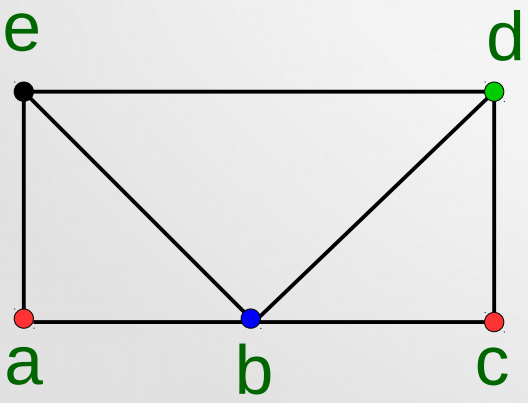


11.4 *Spanning Trees*

Backtracking applications

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

Let's try to color the following graph with 3 colors: red, blue and green using the backtracking procedure.



a red

a red, b blue

a red, b blue, c red

a red, b blue, c red, d green

we are stuck, *backtrack*

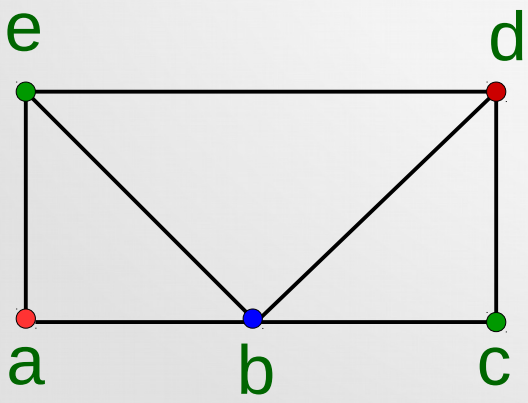
11.4 *Spanning Trees*

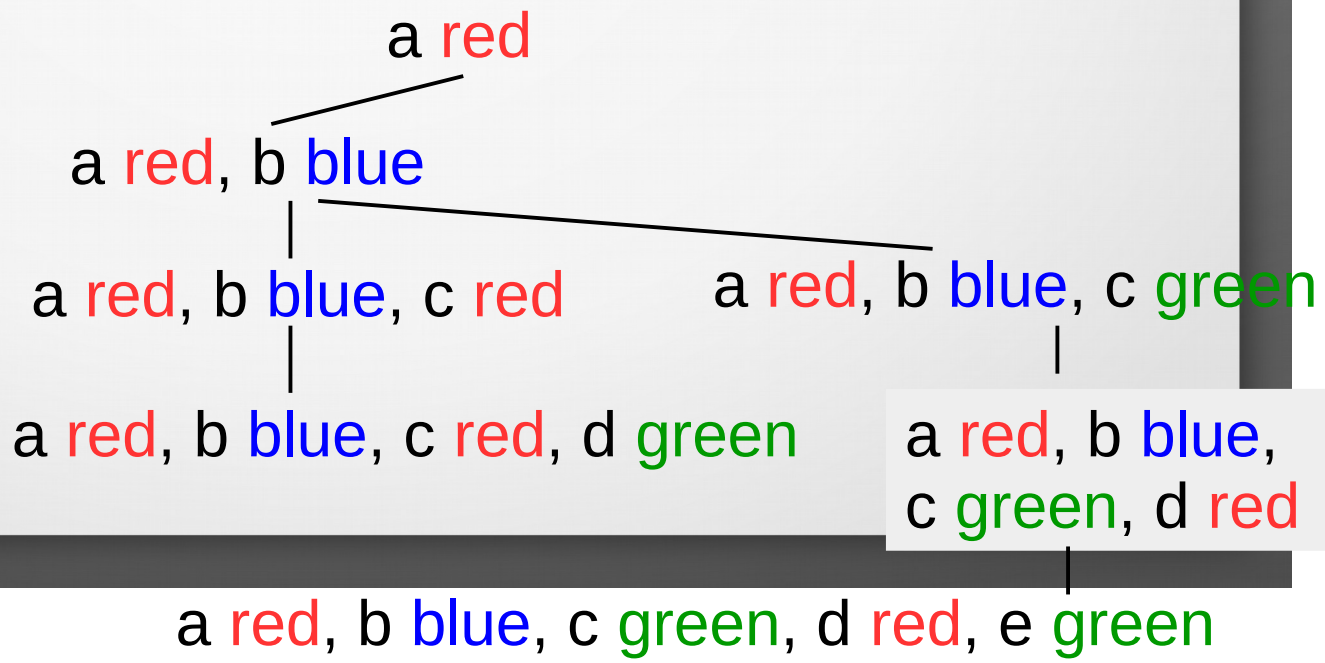
Backtracking applications

Example Graph colorings

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

Let's try to color the following graph with 3 colors: red, blue and green using the backtracking procedure.





11.4 *Spanning Trees*

Backtracking applications

Example Sums of subsets

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

How can backtracking be used to solve this problem?

11.4 *Spanning Trees*

Backtracking applications

Example Sums of subsets

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

How can backtracking be used to solve this problem?

Algorithm:

1) start with an empty sum

2) at every step:

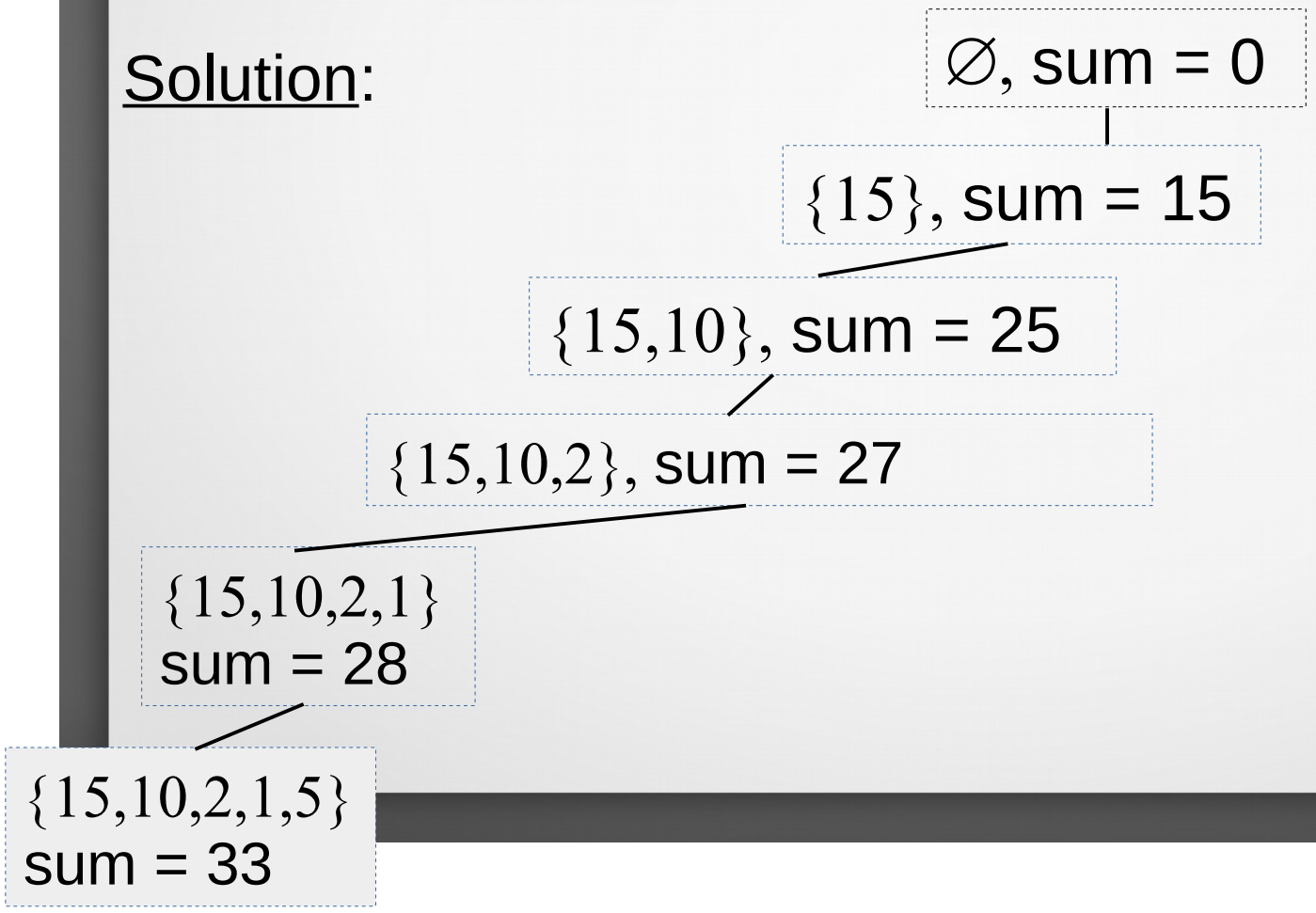
- add an integer if the sum remains < M
- if a sum is reached such that the addition of any term > M, backtrack by dropping the last term of the sum

11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

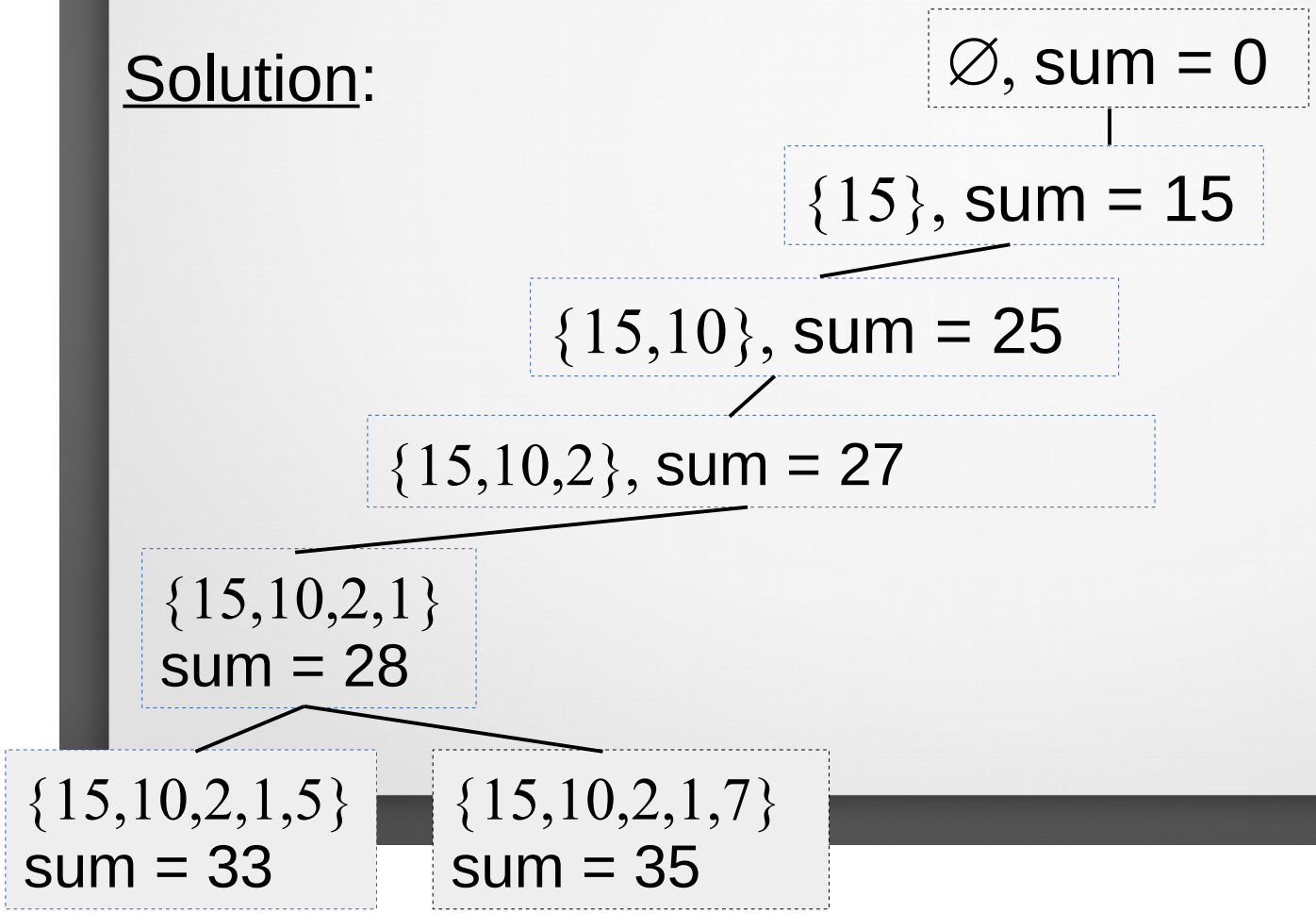


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

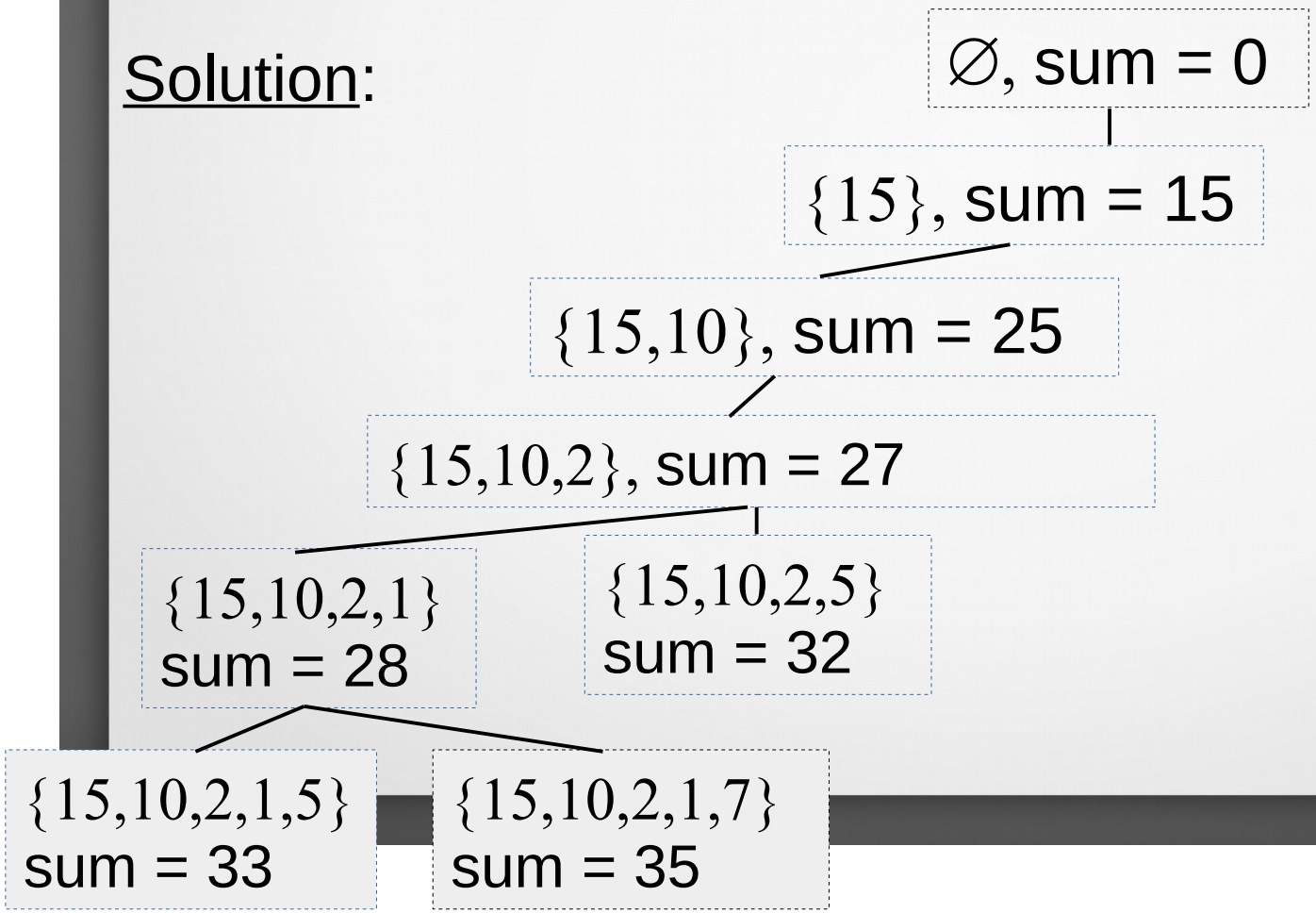


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

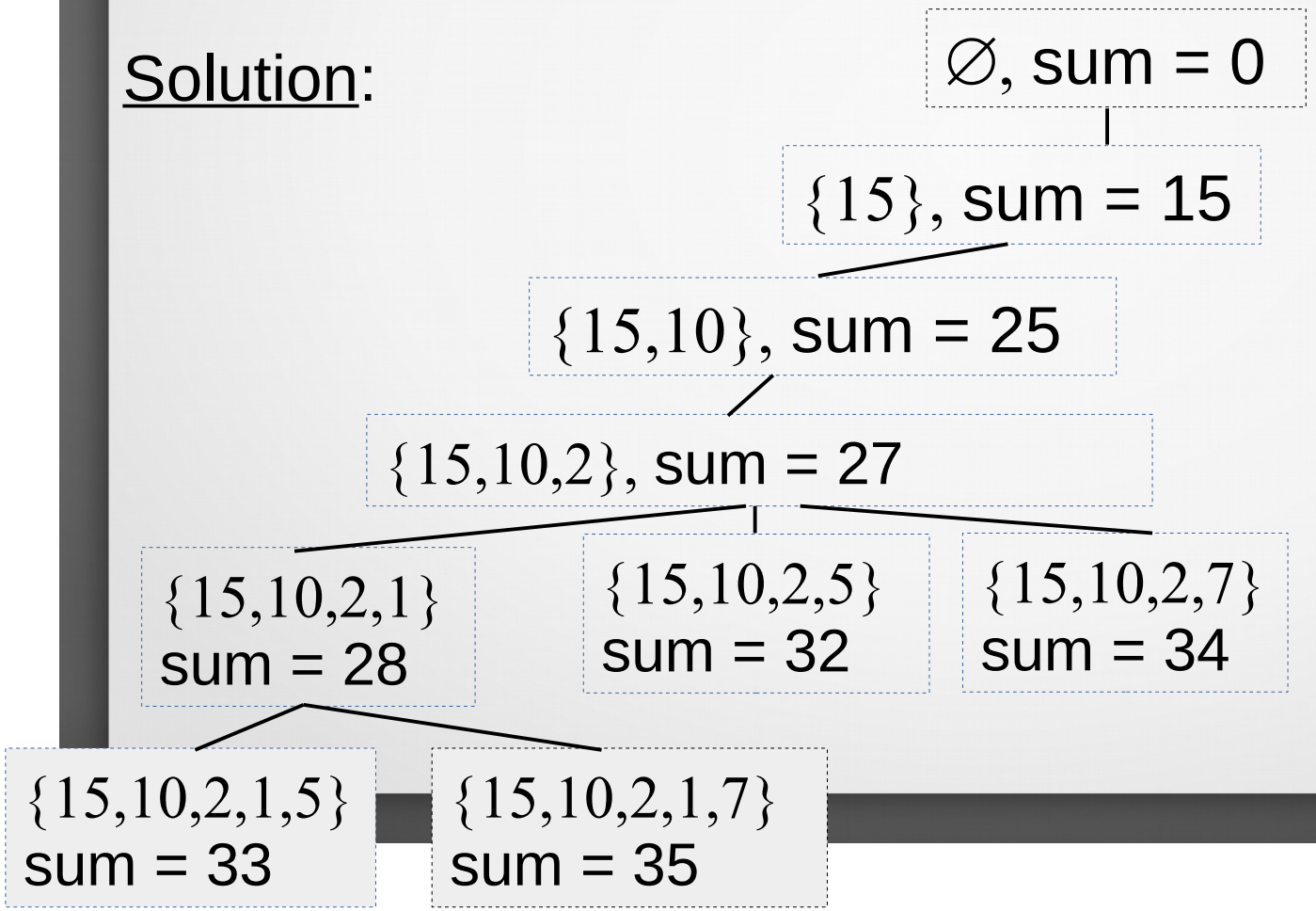


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

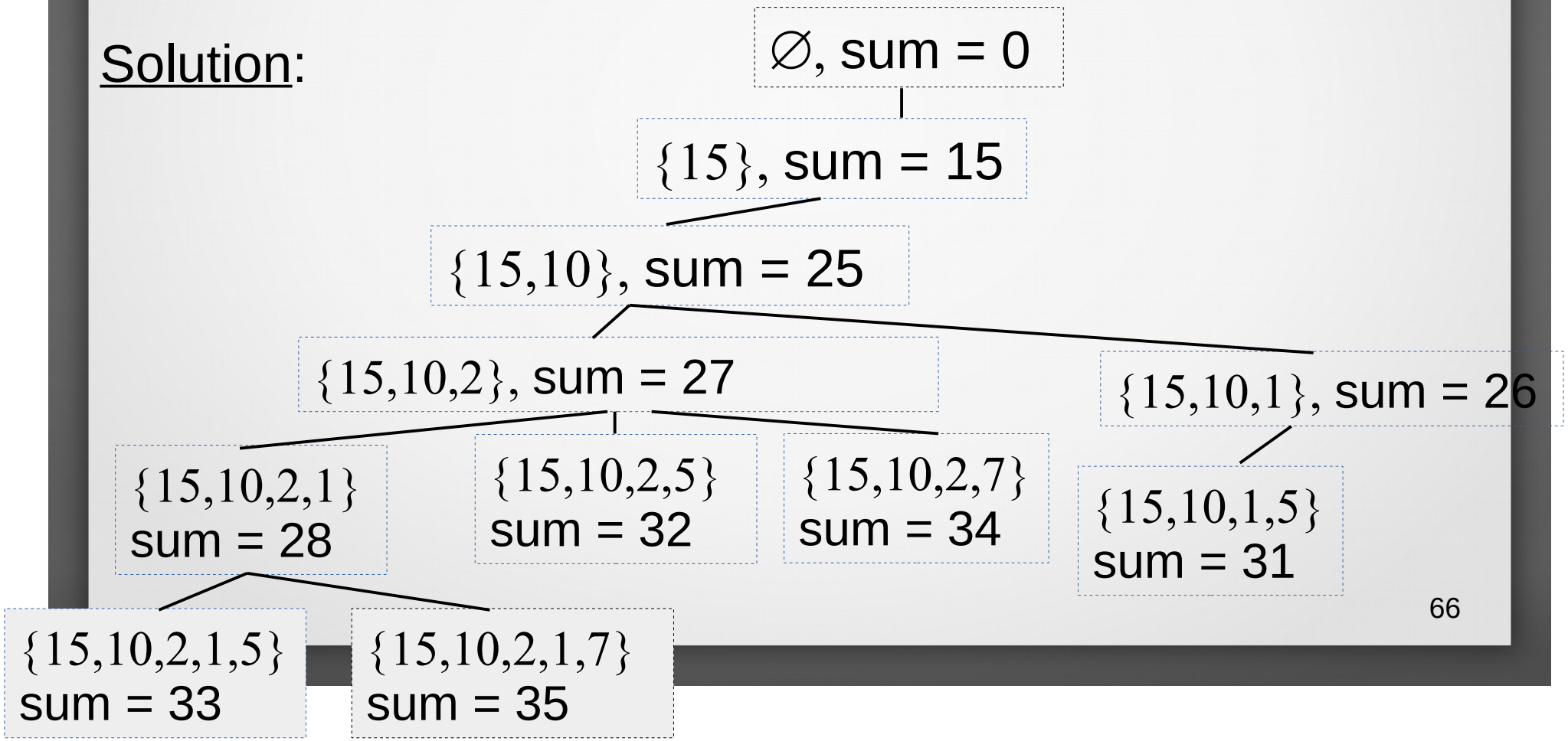


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

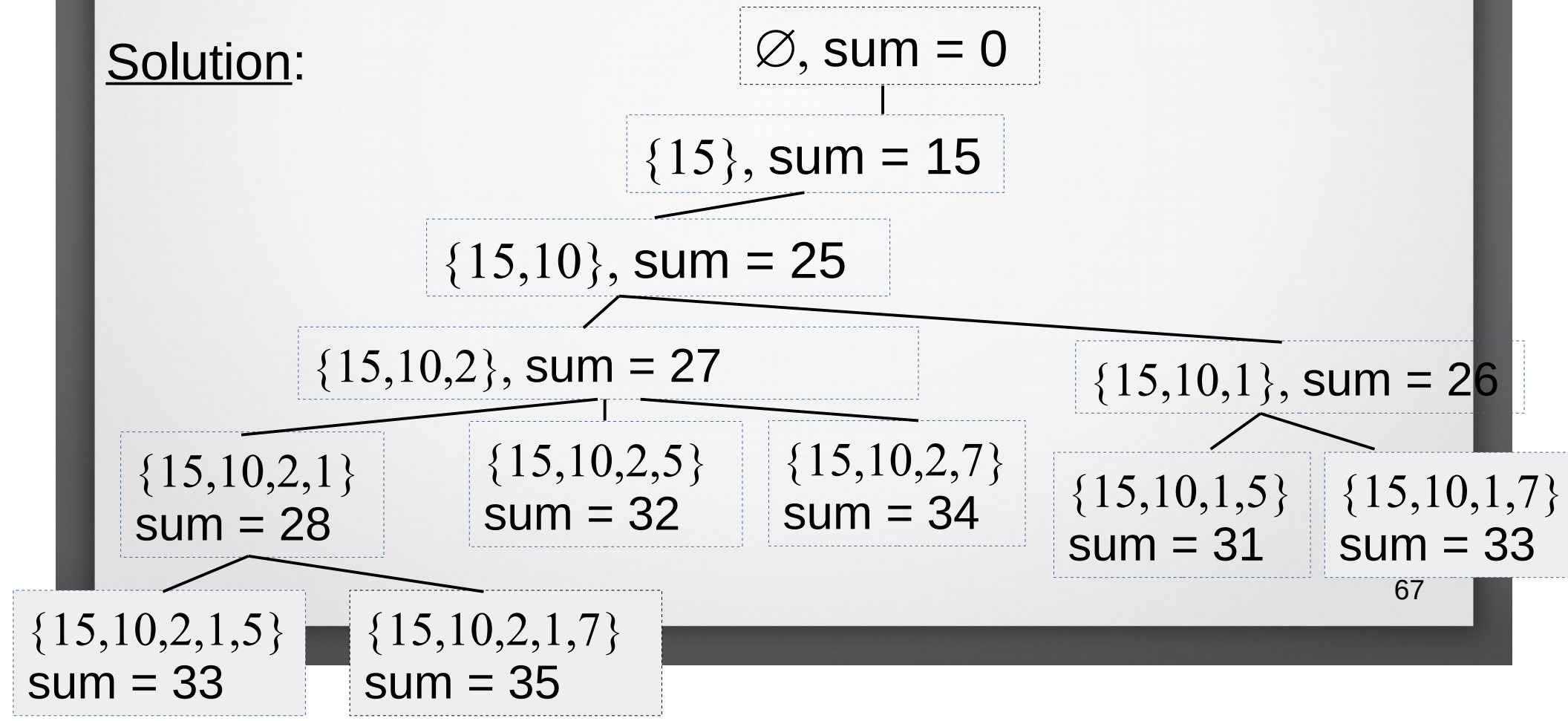


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

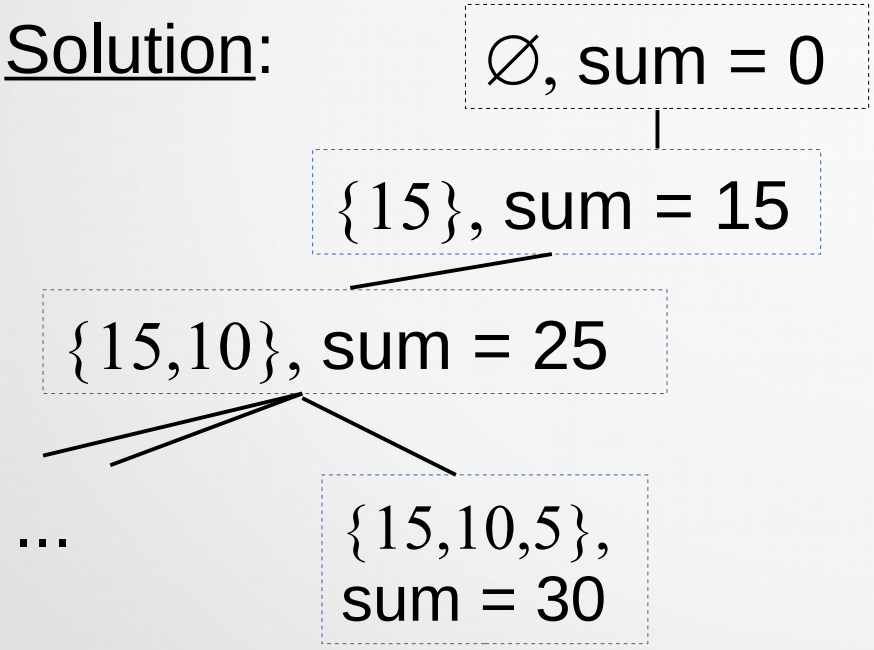


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

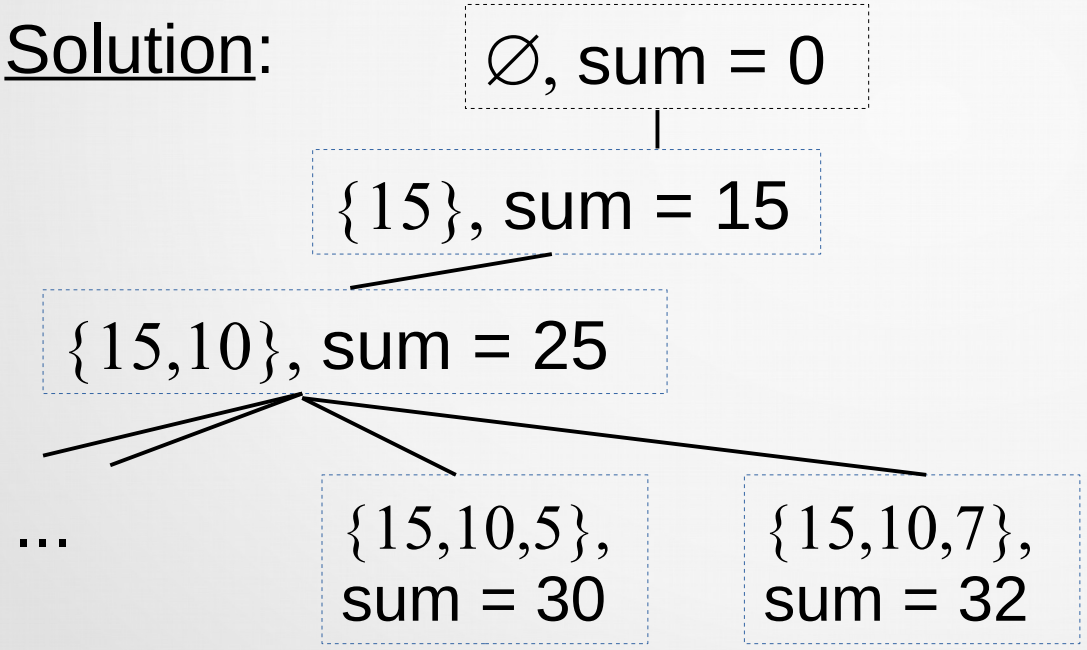


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:

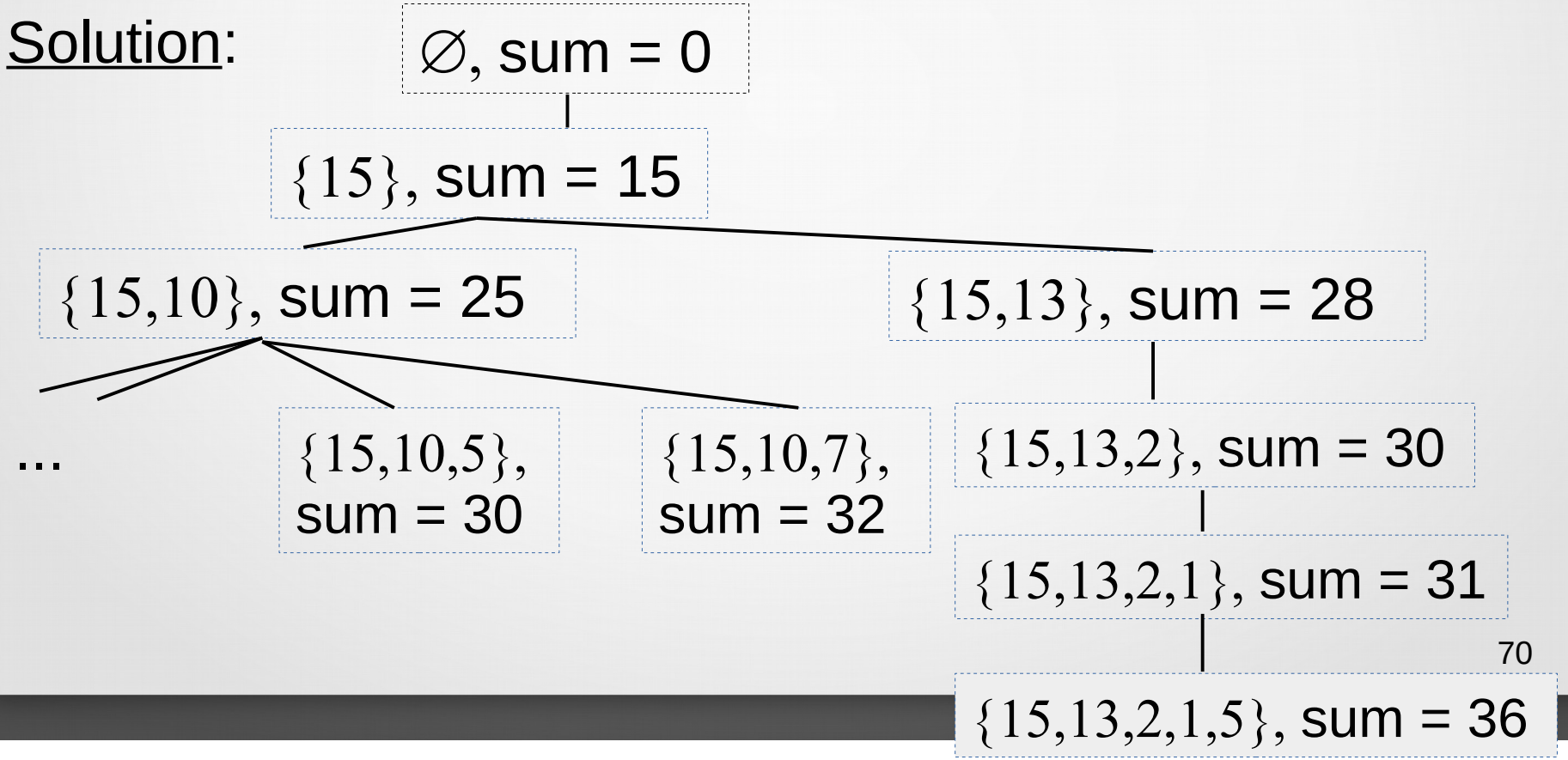


11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Solution:



11.4 *Spanning Trees*

Backtracking applications

Example: Sums of subsets Let S = {15,10,13,2,1,5,7}
Find a subset of S that gives 36 as its sum.

Answer: {15, 13, 2, 1, 5}

Note that this is not a smallest possible subset.
For example, {15,13,1,7} will be a smallest subset of S, such that the sum of its members is 36.

11.4 *Spanning Trees*

Depth-First Search and Breadth-First Search in directed graphs

Both algorithms can be easily modified to support directed graphs: we will be adding an edge when it is directed away from the vertex we are visiting, to a vertex not yet added.

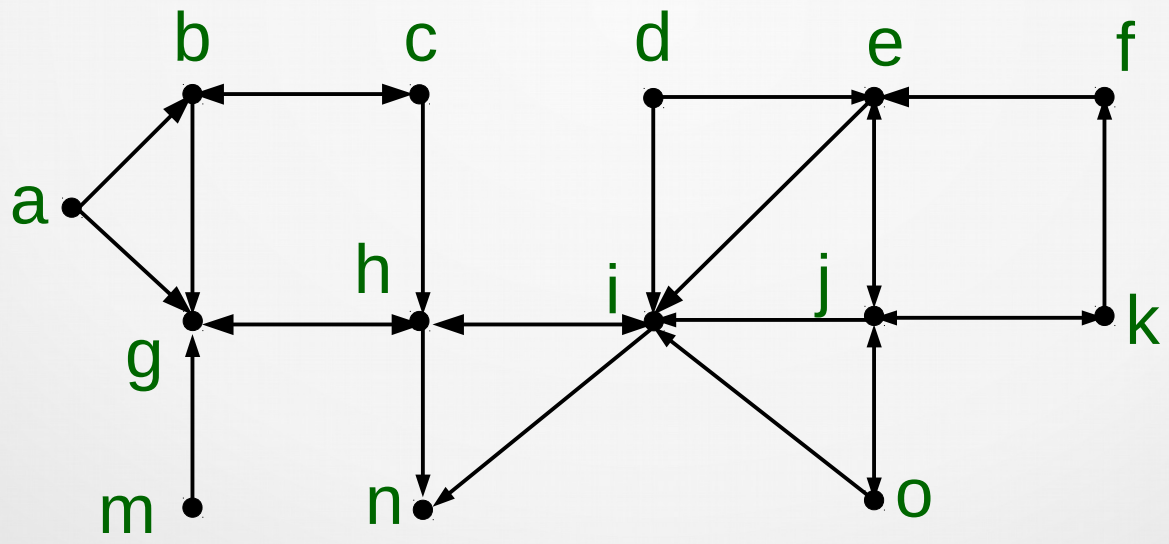
However, the output will not necessarily be a spanning tree. It could be a spanning forest.

DFS algorithm in directed graphs is basis of many algorithms. It can be used to determine whether a directed graph has a circuit, carry out topological sort of a graph, find strongly connected components of a directed graph.

11.4 *Spanning Trees*

Depth-First Search and Breadth-First Search in directed graphs

Example: use DFS and BFS on the graph below to find a spanning tree/forest)

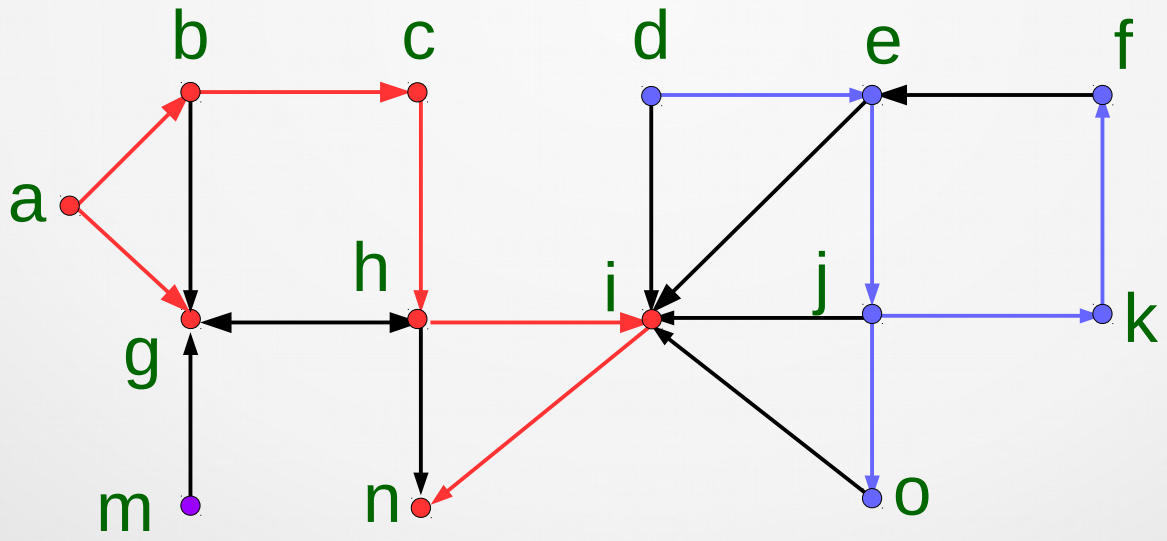


11.4 *Spanning Trees*

Depth-First Search and Breadth-First Search in directed graphs

Example: use DFS and BFS on the graph below to find a spanning tree/forest)

DFS:

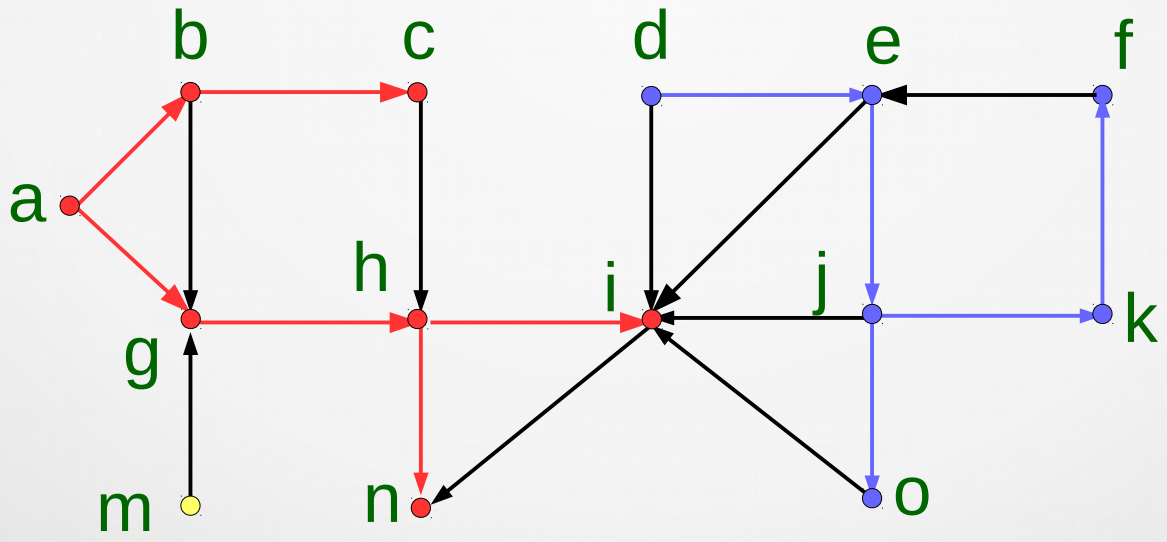


11.4 *Spanning Trees*

Depth-First Search and Breadth-First Search in directed graphs

Example: use DFS and BFS on the graph below to find a spanning tree/forest)

BFS:



L = {a,b,g,c,h,i,n} L'={d,e,k,o,f} L''={m}

11.4 *Spanning Trees*

Web Spiders

To index websites, search engines such as Google and Yahoo systematically explore Web starting at known sites.

The engines use programs called Web spiders (or crawlers or bots) to visit websites and analyze their contents.

Web spiders use both depth-first and breadth-first searching to create indices.

Recall Web graphs:
vertices: web pages
Edges (directed): links

11.4 *Spanning Trees*

Web Spiders

Using depth-first search, an initial Web page is selected,a link is followed to a second web page (if such a link exists), a link on the second web page is followed to a third web page (if it exists) and so forth, until a page with no links is found.

Backtracking is used to examine links at the previous level to look for new links, and so on.

Web spiders have limits to depth in the depth-first search.

11.4 *Spanning Trees*

Web Spiders

Using breadth-first search, an initial web page is selected and a link on that page is followed to a second web page, then a second link on the initial web page is followed (if it exists), and so forth, until all links of the initial page have been followed.

Then links on the page one level down are followed, page by page, and so on.