厦門大學信息学院
School of Informatics Xiamen University
(特色化示范性软件学院)
(National Characteristic Demonstration Software School)

《离散数学》

Chapter 11：Trees（II）

王晓黎

2025年12月19日

Outline

- Introduction to Trees
- Tree Traversal
- Spanning Trees (生成树)
- Minimum Spanning Trees (最小生成树)

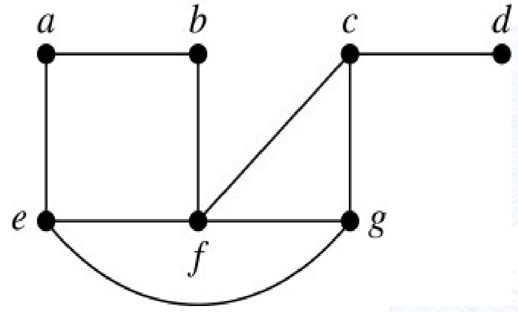
Spanning Trees

- Spanning Trees
- Depth-First Search
- Breadth-First Search
- Depth-First Search in Directed Graphs

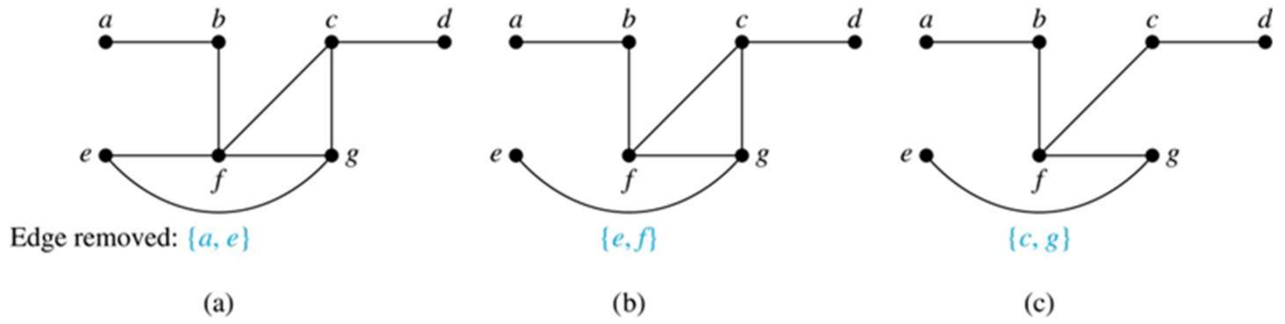
Spanning Trees

Definition: 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.

Example: Find the spanning tree of this simple graph:



Solution: The graph is connected, but is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. Now one simple circuit is gone, but the remaining subgraph still has a simple circuit. Remove the edge $\{e, f\}$ and then the edge $\{c, g\}$ to produce a simple graph with no simple circuits. It is a spanning tree, because it contains every vertex of the original graph.



Spanning Trees (continued)

Theorem: A simple graph is connected if and only if it has a spanning tree.

Proof: Suppose that a simple graph *G* has a spanning tree *T*. *T* contains every vertex of *G* and there is a path in *T* between any two of its vertices. Because *T* is a subgraph of *G*, there is a path in *G* between any two of its vertices. Hence, *G* is connected.

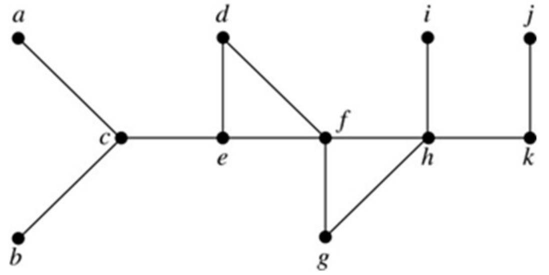
Now suppose that *G* is connected. If *G* is not a tree, it contains a simple circuit. Remove an edge from one of the simple circuits. The resulting subgraph is still connected because any vertices connected via a path containing the removed edge are still connected via a path with the remaining part of the simple circuit. Continue in this fashion until there are no more simple circuits. A tree is produced because the graph remains connected as edges are removed. The resulting tree is a spanning tree because it contains every vertex of *G*.

Depth-First Search

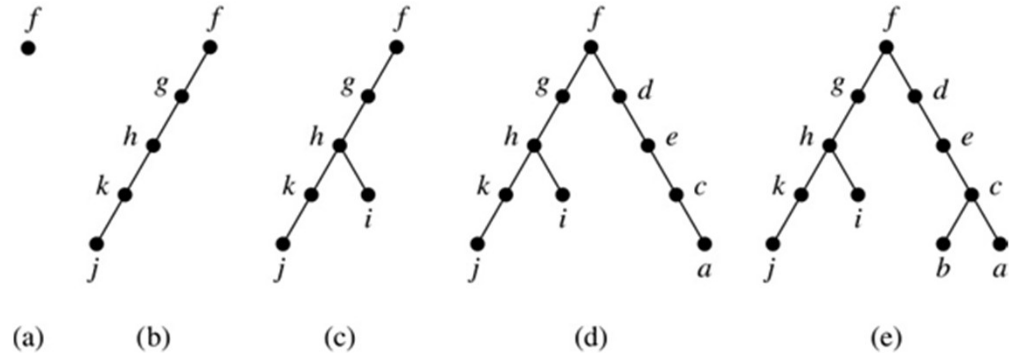
- To use *depth-first search* to build a spanning tree for a connected simple graph first arbitrarily choose a vertex of the graph as the root
    - Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible
    - If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree
    - Otherwise, move back to the next to the last vertex in the path, and if possible, form a new path starting at this vertex and passing through vertices not already visited. If this cannot be done, move back another vertex in the path
    - Repeat this procedure until all vertices are included in the spanning tree

Depth-First Search (continued)

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

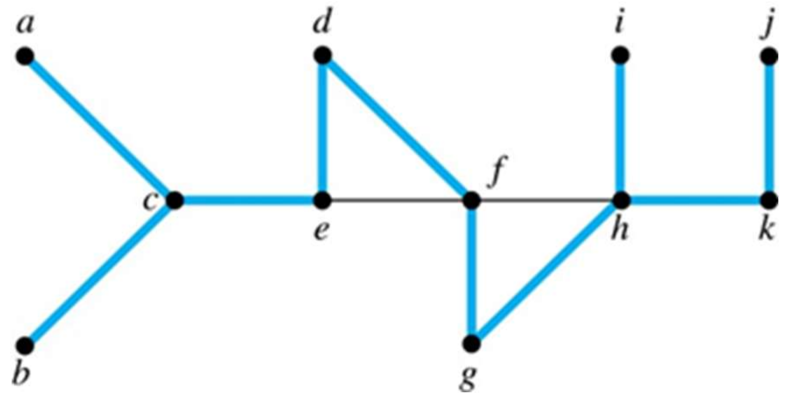


Solution: We start arbitrarily with vertex *f*. We build a path by successively adding an edge that connects the last vertex added to the path and a vertex not already in the path, as long as this is possible. The result is a path that connects *f*, *g*, *h*, *k*, and *j*. Next, we return to *k*, but find no new vertices to add. So, we return to *h* and add the path with one edge that connects *h* and *i*. We next return to *f*, and add the path connecting *f*, *d*, *e*, *c*, and *a*. Finally, we return to *c* and add the path connecting *c* and *b*. We now stop because all vertices have been added.



Depth-First Search (continued)

- 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*

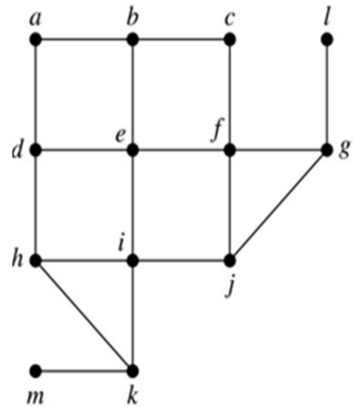
```
procedure DFS(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex $v_1$
visit($v_1$)

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)
```

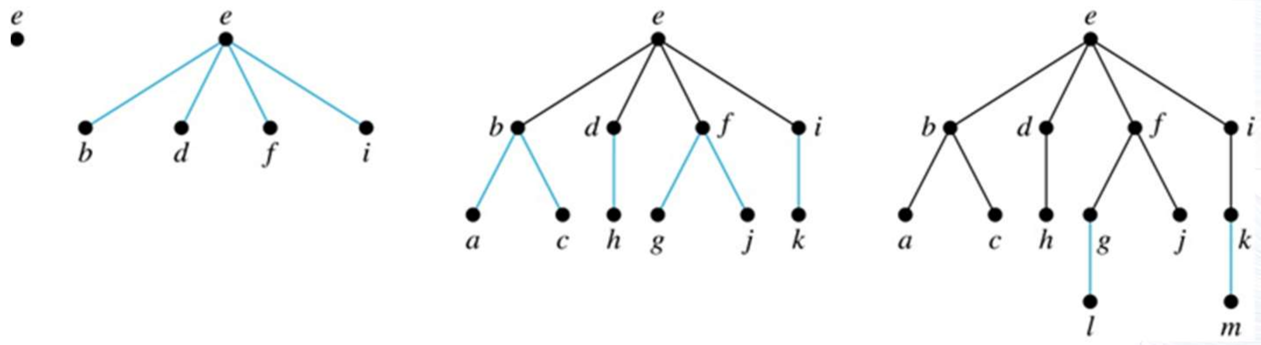
Breadth-First Search

- We can construct a spanning tree using *breadth-first search*. We first arbitrarily choose a root from the vertices of the graph
  - Then we add all of the edges incident to this vertex and the other endpoint of each of these edges. We say that these are the vertices at level 1
  - For each vertex added at the previous level, we add each edge incident to this vertex, as long as it does not produce a simple circuit. The new vertices we find are the vertices at the next level
  - We continue in this manner until all the vertices have been added and we have a spanning tree

Breadth-First Search (continued)

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





Solution: We arbitrarily choose vertex *e* as the root. We then add the edges from *e* to *b*, *d*, *f*, and *i*. These four vertices make up level 1 in the tree. Next, we add the edges from *b* to *a* and *c*, the edges from *d* to *h*, the edges from *f* to *j* and *g*, and the edge from *i* to *k*. The endpoints of these edges not at level 1 are at level 2. Next, add edges from these vertices to adjacent vertices not already in the graph. So, we add edges from *g* to *l* and from *k* to *m*. We see that level 3 is made up of the vertices *l* and *m*. This is the last level because there are no new vertices to find.

Breadth-First Search Algorithm

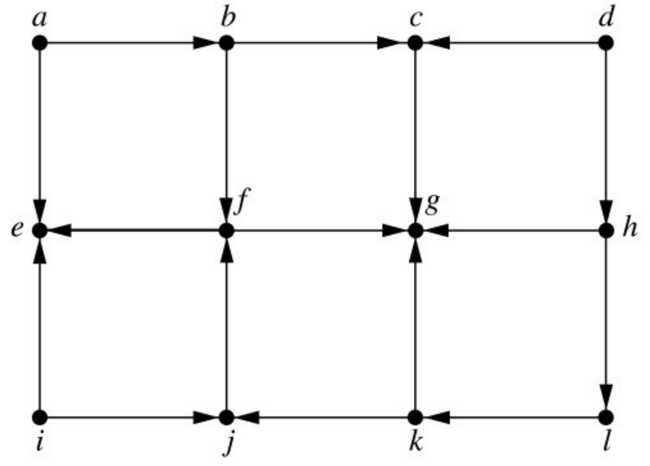
- We now use pseudocode to describe breadth-first search

```
**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 $T$
```

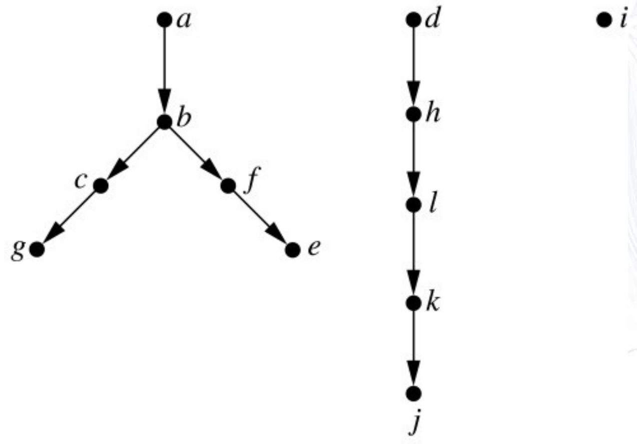
Depth-First Search in Directed Graphs

- 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

Example: For the graph in (a), if we begin at vertex *a*, depth-first search adds the path connecting *a*, *b*, *c*, and *g*. At *g*, we are blocked, so we return to *c*. Next, we add the path connecting *f* to *e*. Next, we return to *a* and find that we cannot add a new path. So, we begin another tree with *d* as its root. We find that this new tree consists of the path connecting the vertices *d*, *h*, *l*, *k*, and *j*. Finally, we add a new tree, which only contains *i*, its root.



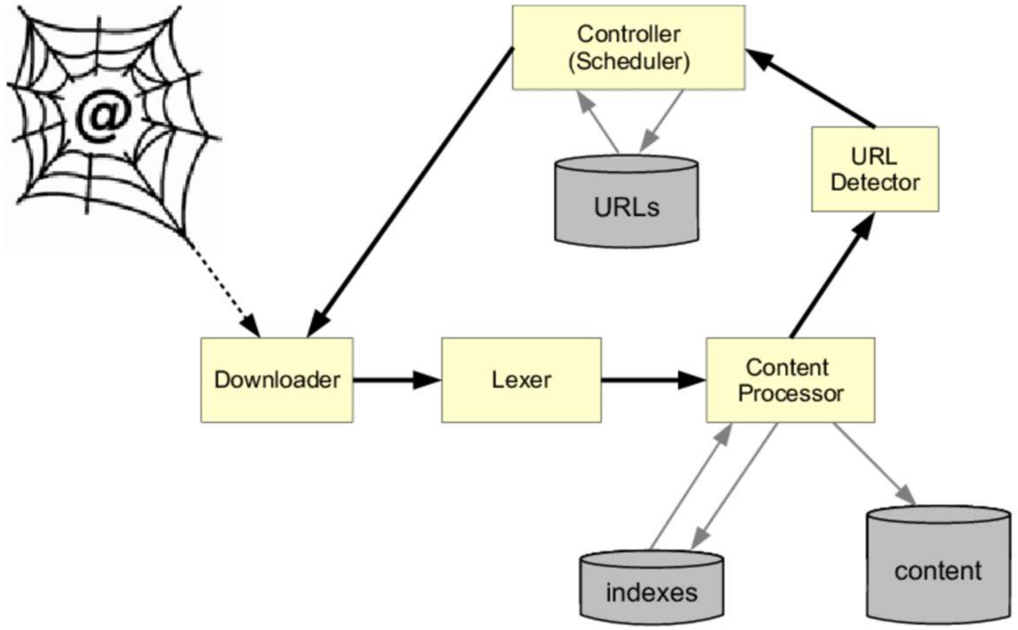
(a)



(b)

Applications: Google Search Engines

- To index websites, search engines such as Google systematically explore the web starting at known sites. The programs that do this exploration are known as *Web spiders*
  - A web spider is a computer program that's used to search and automatically index website content



Applications: Google Search Engines

- How does a web spider search and automatically index website content?
  - Start from a seed set: a list of URLs to visit, denoted by *U*
    - Visit hyperlinks in the pages and add them to crawl frontier
    - Depth first search or breadth first search

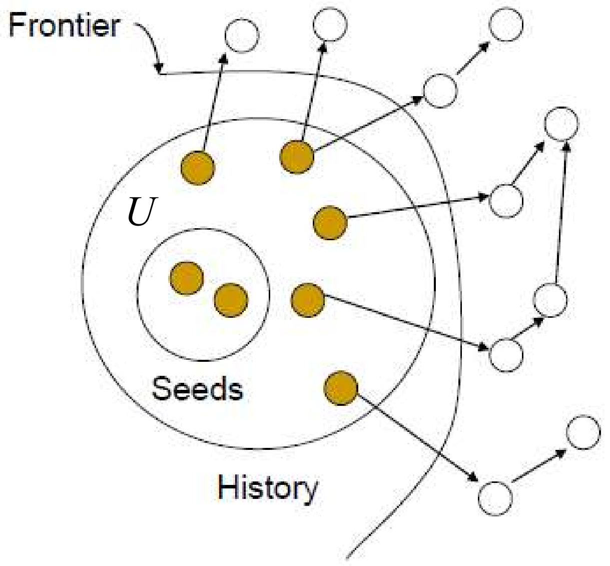
Crawling approach:

(1) Get next URL *u* from *U*

(2) Download the webpage of *u*

URL Frontier: contains URLs yet to be fetched

– **URLs from the frontier are recursively visited according to a set of policies**



Applications: Google Search Engines (Continued)

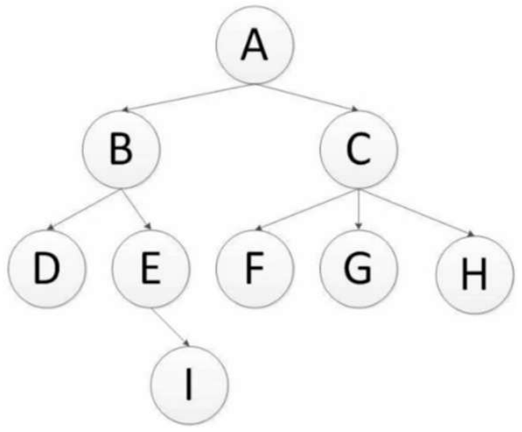
- Depth-first search
  - Start from the initial page, follow one link after another until the path ends, then move on to another starting page and continue tracking the links
- Breadth-first search
  - Directly insert each first-level link found on the downloaded web page at the end of the URL queue to be crawled

Depth-first crawling order:

A-B-D-E-I-C-F-G-H

Breadth-first crawling order:

A-B-C-D-E-F-G-H-I



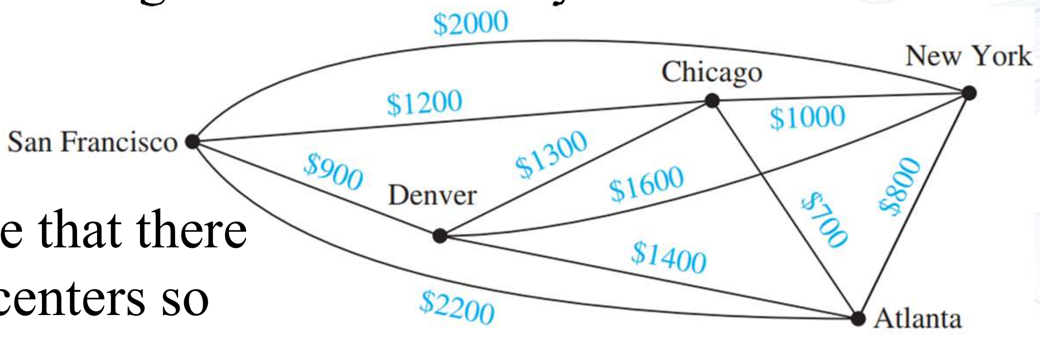
Minimum Spanning Trees

- Minimum Spanning Trees
- Algorithms for Minimum Spanning Trees

Minimum Spanning Trees

Definition: Graphs that have a number assigned to each edge are called ***weighted graphs***.

Example: A company plans to build a communications network connecting its five computer centers. Vertices represent computer centers, edges represent possible leased lines, and the weights on edges are the monthly lease rates of the lines represented by the edges.



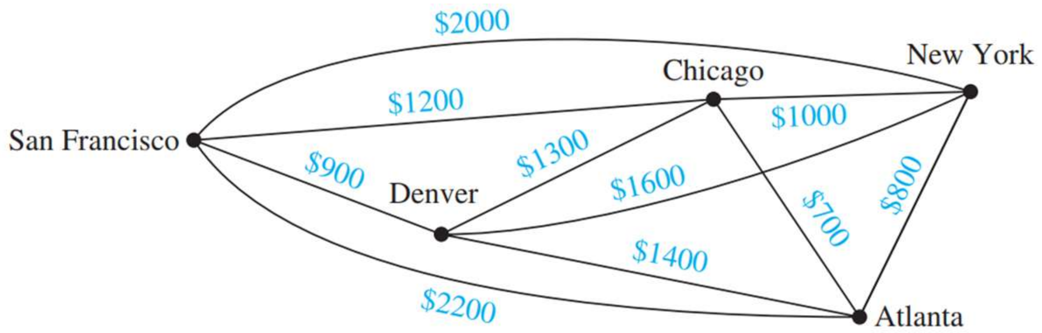
Which links should be made to ensure that there is a path between any two computer centers so that the total cost of the network is minimized?

Solution: We can solve this problem by finding a spanning tree so that the sum of the weights of the edges of the tree is minimized. Such a spanning tree is called a ***minimum spanning tree***.

Minimum Spanning Trees

Definition: A ***minimum spanning tree*** in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

Example: Design a minimum-cost communications network connecting all the computers represented by the graph?



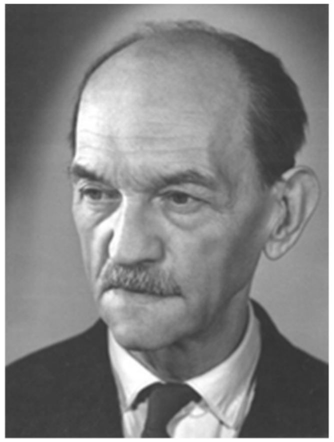
Solution: How about using brute force?

Algorithms for Finding Minimum Spanning Trees

Recall: In Section 3.1 a ***greedy algorithm*** is a procedure that makes an optimal choice at each of its steps. Optimizing at each step does not guarantee that the optimal overall solution is produced. However, the two algorithms presented in this section for constructing minimum spanning trees are greedy algorithms that do produce optimal solutions.

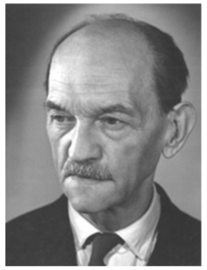
The first algorithm was discovered by Vojtěch Jarník in 1930

The second algorithm was discovered by Joseph Kruskal in 1956


Vojtěch Jarník
(1897-1970)

Joseph Kruskal
(1928-2010)

Prim’s algorithm


Vojtěch Jarník
(1897-1970)

Suppose the graph has *n* edges,

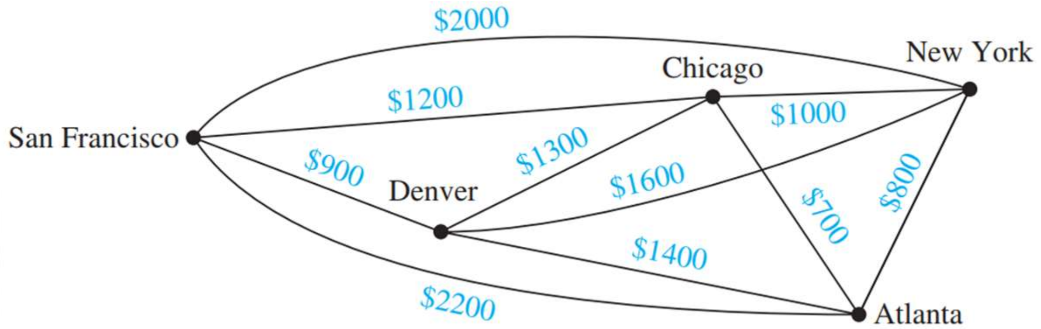
- Begin by choosing any edge with smallest weight, putting it into the spanning tree
- Successively add to the tree edges of minimum weight that are incident to a vertex already in the tree, never forming a simple circuit with those edges already in the tree
- Stop when *n*-1 edges have been added

ALGORITHM 1 Prim’s Algorithm.

```
procedure Prim(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for i := 1 to n − 2
    e := an edge of minimum weight incident to a vertex in T and not forming a
        simple circuit in T if added to T
    T := T with e added
return T {T is a minimum spanning tree of G}
```

Prim's algorithm

- **Example:** Use *Prim's algorithm* to design a minimum-cost communications network connecting all the computers represented by the graph



Choice	Edge	Cost
1	{Chicago, Atlanta}	$ 700
2	{Atlanta, New York}	$ 800
3	{Chicago, San Francisco}	$1200
4	{San Francisco, Denver}	$ 900
	Total:	$3600

- **Solution:** We solve this problem by finding a minimum spanning tree in the graph. Prim's algorithm is carried out by choosing an initial edge of minimum weight and successively adding edges of minimum weight that are incident to a vertex in the tree and that do not form simple circuits. The edges in color show a minimum spanning tree produced by Prim's algorithm, with the choice made at each step displayed.

Kruskal’s algorithm

Joseph Kruskal
(1928-2010)

Suppose the graph has *n* edges,

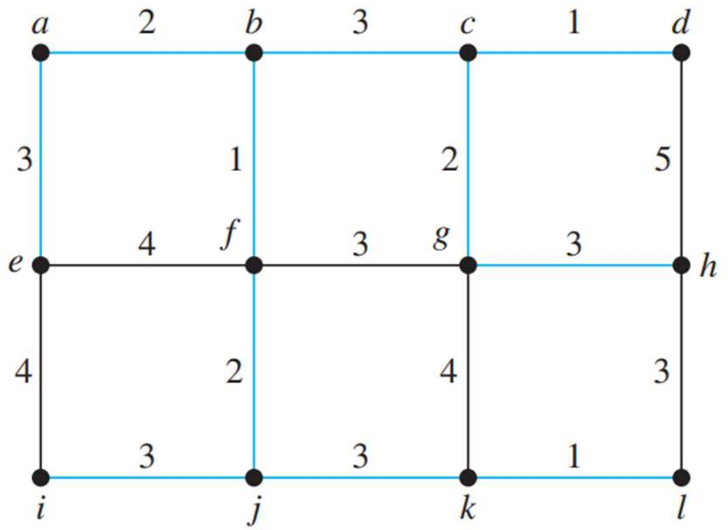
- Begin by choosing an edge in the graph with minimum weight, putting it into the spanning tree
- Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen
- Stop after *n*-1 edges have been selected

ALGORITHM 2 Kruskal’s Algorithm.

```
procedure Kruskal(G: weighted connected undirected graph with n vertices)
T := empty graph
for i := 1 to n − 1
    e := any edge in G with smallest weight that does not form a simple circuit
        when added to T
    T := T with e added
return T {T is a minimum spanning tree of G}
```

Kruskal’s algorithm

- **Example:** Use *Kruskal’s algorithm* to find a minimum spanning tree in the weighted graph



Choice	Edge	Weight
1	$\{c, d\}$	1
2	$\{k, l\}$	1
3	$\{b, f\}$	1
4	$\{c, g\}$	2
5	$\{a, b\}$	2
6	$\{f, j\}$	2
7	$\{b, c\}$	3
8	$\{j, k\}$	3
9	$\{g, h\}$	3
10	$\{i, j\}$	3
11	$\{a, e\}$	3
	Total:	24

Proof for Prim's algorithm

Proof: Let G be a connected weighted graph. Suppose that the successive edges chosen by Prim's algorithm are $e_1, e_2, \ldots, e_{n-1}$. Let S be the tree with $e_1, e_2, \ldots, e_{n-1}$ as its edges, and let S_k be the tree with $e_1, e_2, \ldots, e_k$ as its edges. Let T be a minimum spanning tree of G containing the edges $e_1, e_2, \ldots, e_k$, where k is the maximum integer with the property that a minimum spanning tree exists containing the first k edges chosen by Prim's algorithm. The theorem follows if we can show that $S = T$.

Suppose that $S \neq T$, so that $k < n - 1$. Consequently, T contains $e_1, e_2, \ldots, e_k$, but not e_{k+1}. Consider the graph made up of T together with e_{k+1}. Because this graph is connected and has n edges, too many edges to be a tree, it must contain a simple circuit. This simple circuit must contain e_{k+1} because there was no simple circuit in T. Furthermore, there must be an edge in the simple circuit that does not belong to S_{k+1} because S_{k+1} is a tree. By starting at an endpoint of e_{k+1} that is also an endpoint of one of the edges $e_1, \ldots, e_k$, and following the circuit until it reaches an edge not in S_{k+1}, we can find an edge e not in S_{k+1} that has an endpoint that is also an endpoint of one of the edges $e_1, e_2, \ldots, e_k$.

By deleting e from T and adding e_{k+1}, we obtain a tree T' with $n - 1$ edges (it is a tree because it has no simple circuits). Note that the tree T' contains $e_1, e_2, \ldots, e_k, e_{k+1}$. Furthermore, because e_{k+1} was chosen by Prim's algorithm at the $(k + 1)$st step, and e was also available at that step, the weight of e_{k+1} is less than or equal to the weight of e. From this observation, it follows that T' is also a minimum spanning tree, because the sum of the weights of its edges does not exceed the sum of the weights of the edges of T. This contradicts the choice of k as the maximum integer such that a minimum spanning tree exists containing $e_1, \ldots, e_k$. Hence, $k = n - 1$, and $S = T$. It follows that Prim's algorithm produces a minimum spanning tree. $\triangleleft$

厦門大學信息学院
School of Informatics Xiamen University
(特色化示范性软件学院)
(National Characteristic Demonstration Software School)

Q&A