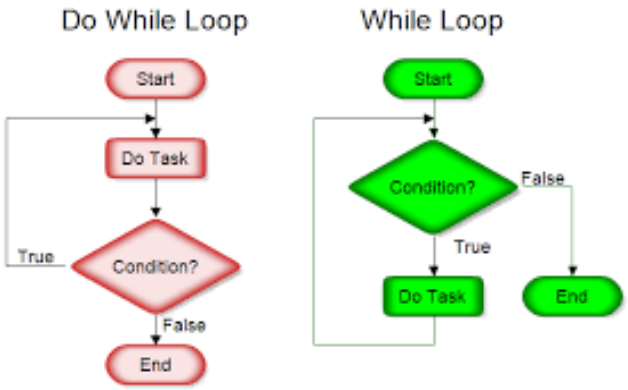
CHAPTER 4 Graphs and Trees

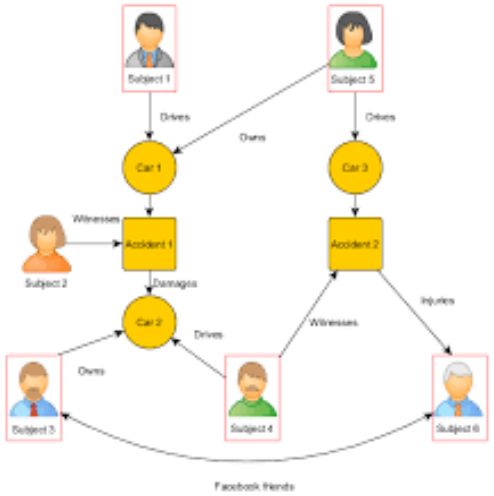
- *Introduction*
- *Terminology and basic properties*
- *Shortest path problems*
- *Trees and binary trees*
- *Spanning tree*

1. Introduction

Many problems can be expressed in the form of graphs. In general terms, a graph is a set of points (called *vertices*) plus a set of lines (called *edges*). They are useful in representing relations between objects.

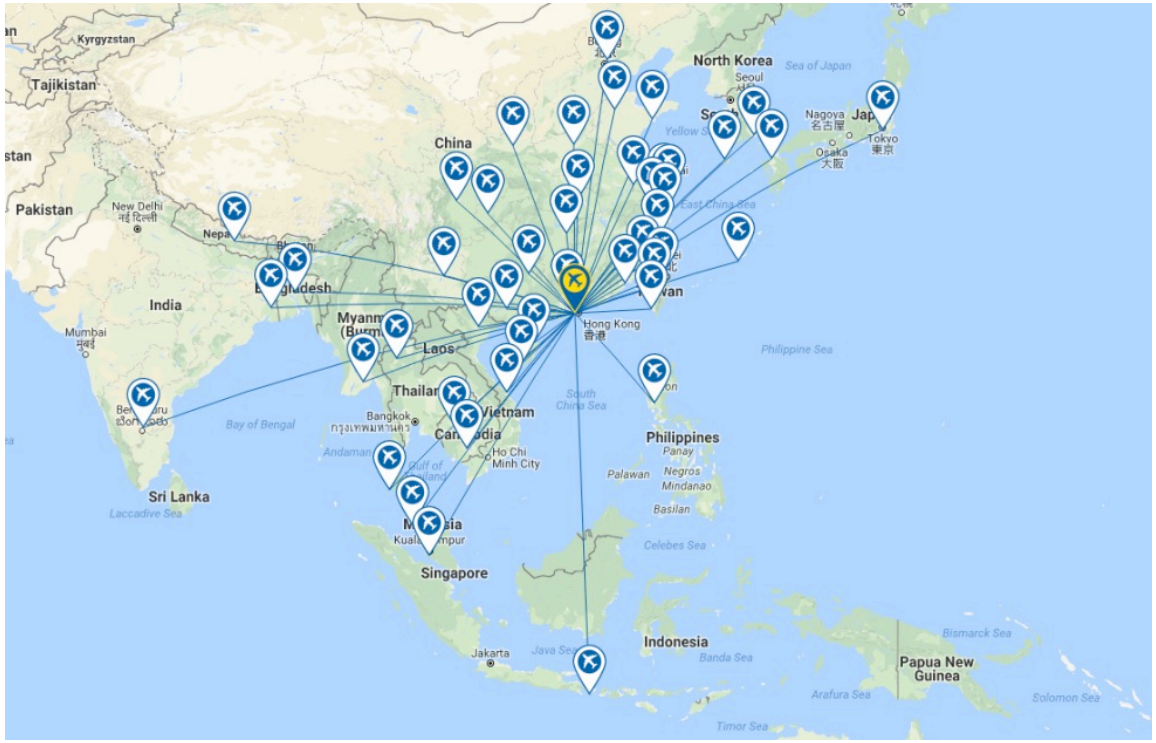


Flow chart



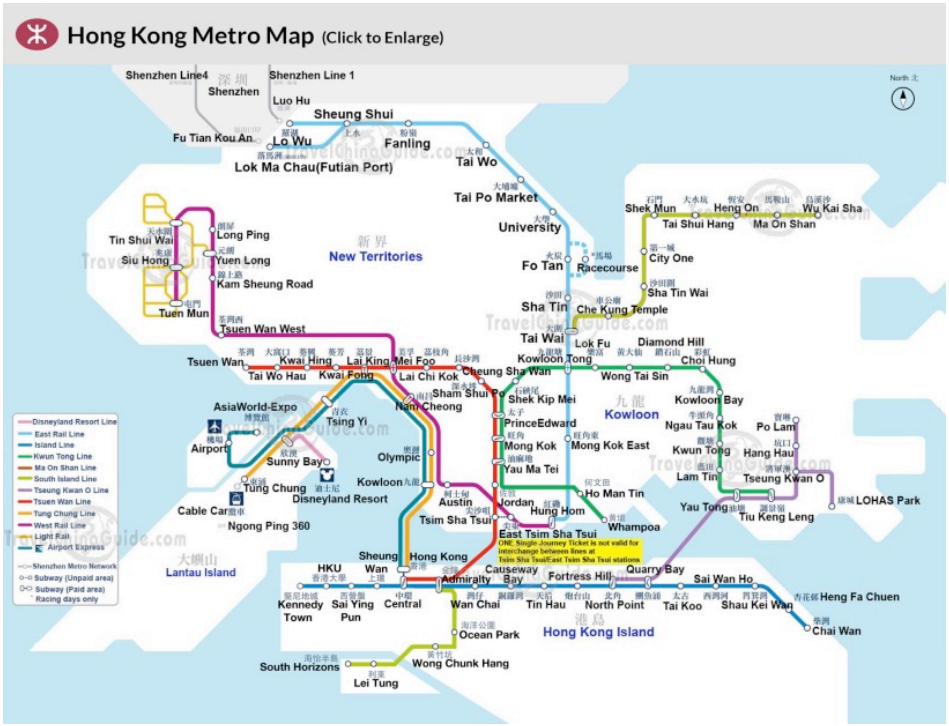
Insurance fraud detection

1. Introduction



Cathay-Dragon Asia air flight route

1. Introduction



Hong Kong metro map

Essential features:

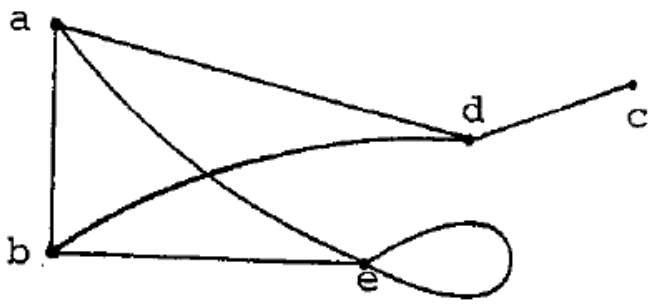
1. Objects: boxes, circles or dots.
2. Connecting lines: directed or undirected.

2. Terminology and basic properties

A (undirected) graph consists of

- a set V of *vertices*,
- a set E of unordered pairs of elements in V called *edges*.

We write $G = (V, E)$.



In the above graph,

$$V = \{a, b, c, d, e\}, \quad E = \{ab, ae, ad, bd, be, cd, ee\}.$$

- A *multigraph* G may have multiple edges between a single pair of vertices.

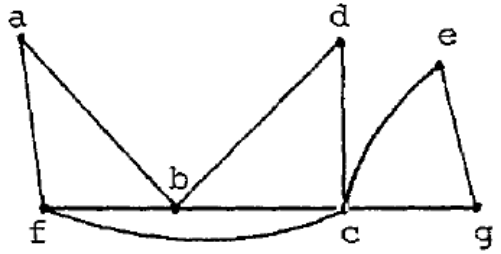
- A *loop* is an edge which starts and ends at the same vertex.

- A graph is *simple* if it has

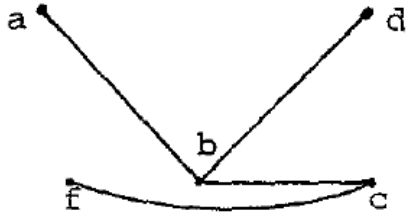
  (i) no multiple edges, and
  (ii) no loops.

Let G be a graph with vertex set V and edge set E.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$.



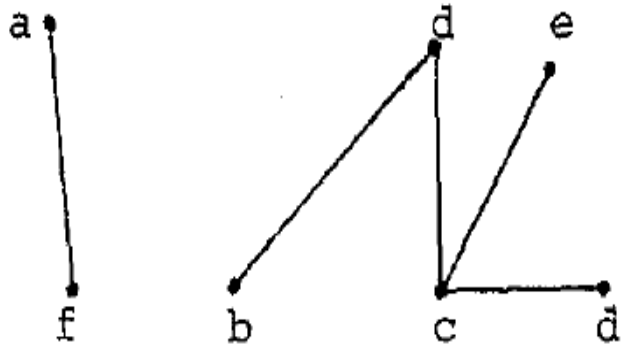
Graph G



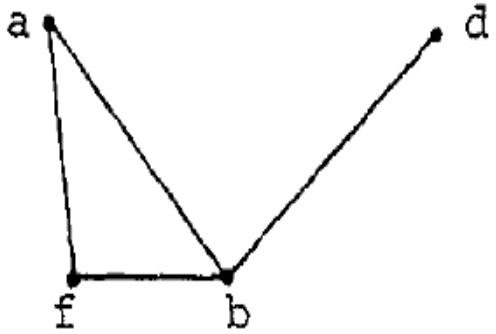
A subgraph of G

Let $V'' \subseteq V$. A graph $G'' = (V'', E'')$ is the subgraph of G induced by V'' if E'' consists of all the edges of G joining vertices in V''.

A subgraph H of G with the same vertex set V is called a *spanning* subgraph of G.



A spanning subgraph of G

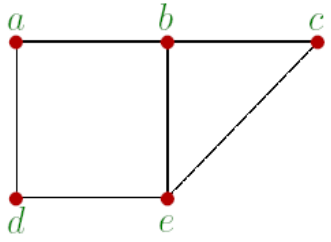


Subgraph of G induced by $\{a, b, d, f\}$.

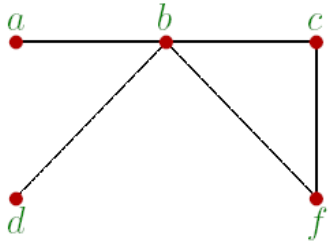
The union of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ denoted by $G_1 \cup G_2$ is the simple graph with the vertex set $V_1 \cup V_2$ and the edge set $E_1 \cup E_2$.

Example 2.1

Find the union of the graphs G_1 and G_2.

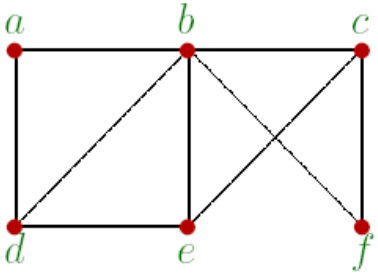


G_1



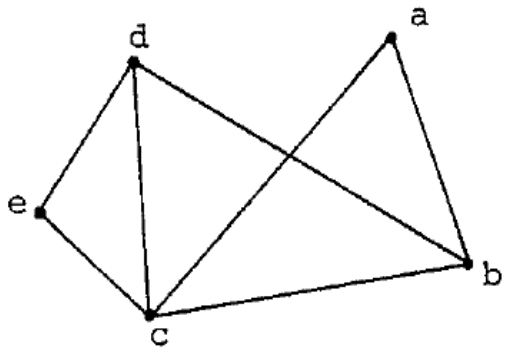
G_2

Solution:



$G_1 \cup G_2$

The *degree* of a vertex v is the number of edges joining v to other vertices, denoted by $d(v)$.



Degree: $d(a) = 2$, $d(b) = 3$, $d(c) = 4$, $d(d) = 3$, $d(e) = 2$.

Theorem 2.1

Sum of vertex degrees = 2 × Number of edges.

In particular, the sum of vertex degrees is even.

Example 2.2
How many edges are there in a graph with 10 vertices each of degree 6?

Example 2.3
Is it possible to construct a network with nine computers so that each computer is connected to exactly five other computers?

A *path of length n from u to v* in an undirected graph is a sequence of edges

$$\gamma = (e_1, e_2, \dots , e_n),$$

and vertices

$$(v_0, v_1, v_2, \dots , v_n), \quad v_0 = u, \quad v_n = v,$$

such that v_{i-1}, v_i are the endpoints of e_i for $i = 1, 2, \dots , n$.

- When the graph is simple, this path is denoted by its vertex sequence:

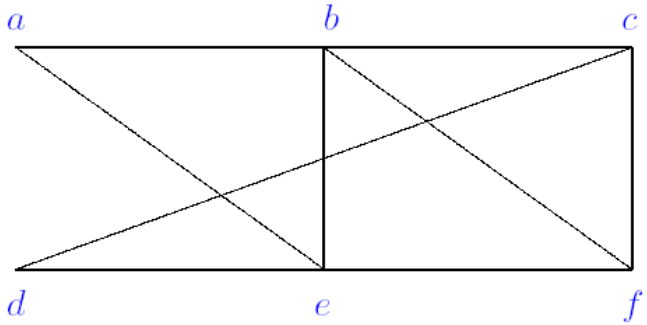
  $$(v_0, v_1, v_2, \dots , v_n).$$

- The path is a circuit if it begins and ends at the same vertex, i.e. $u = v$.

- A path or circuit is simple (resp. loop-free) if it does not contain the same edge (resp. vertex) more than once.

Example 2.4

In the following simple graph

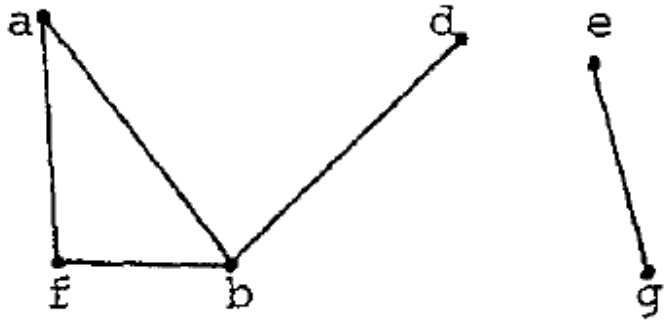


a, b, c, f, e is a simple path of length 4.
b, c, f, e, b is a simple circuit of length 4.
a, b, f, e, b, c is a simple path of length 5.

- A graph is *connected* if every pair of distinct vertices is joined by a path. Otherwise, the graph is *disconnected*.

- The disjoint connected subgraphs of a graph are called *components* of *G*.

For example, the graph *H* shown below has two components.



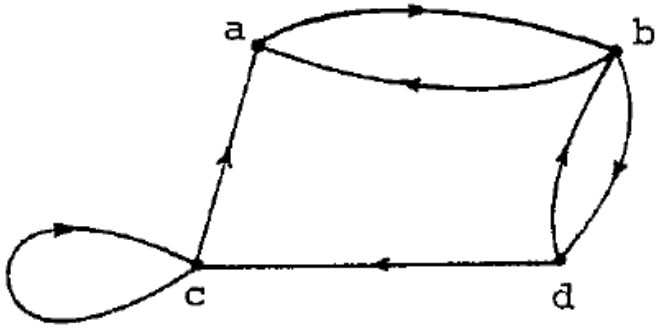
Theorem 2.2

A simple and connected graph $G = (V, E)$ with n vertices has at least $n - 1$ edges.

A *directed* graph (a *digraph* for short) is a graph model in which the edges have arrows. More formally, a digraph consists of

(1) a vertex set V, plus
(2) an edge set E whose elements are ordered pairs of vertices.

The following is a digraph with vertex set $V = \{a, b, c, d\}$ and edge set $E = \{ab, ba, bd, da, ca, dc, cc\}$. Note that here edges ab and ba are different because they are directed.



- The *out-degree* of a vertex v, denoted by $d^-(v)$, is the number of edges directed away from v.

- The *in-degree* of v, denoted by $d^+(v)$, is the number of edges directed into v.

Theorem 2.3

$$\sum \text{in-degrees} = \sum \text{out-degrees}.$$

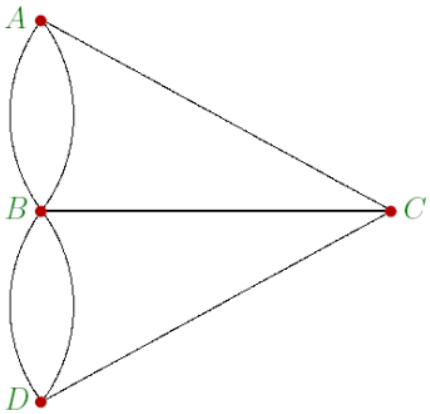
Euler Path

The Königsberg bridges

Question: Is it possible to start at some location in the town and travel across all the bridges without crossing any bridge twice, and return to the starting point?

This is equivalent to the following question.

Question: Is there a simple circuit in this multigraph that contains every edge?

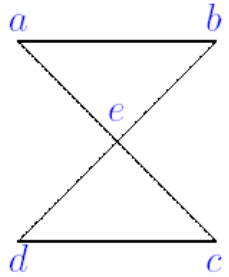


Definition:

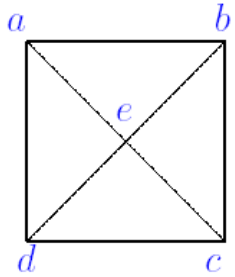
- An *Euler circuit* in a graph G is a simple circuit containing every edge of G.
- An *Euler path* in G is a simple path containing every edge of G.

Example 2.5

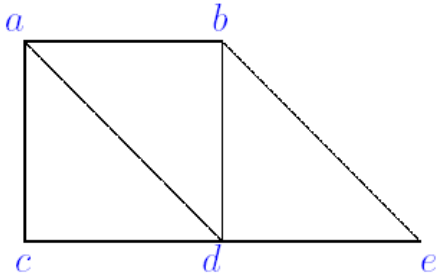
Which of the following undirected graphs have an Euler circuit? Of those that do not, which have an Euler path?



G_1



G_2



G_3

Theorem 2.4

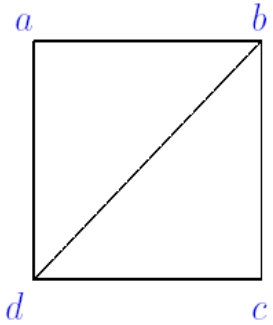
A connected graph has an Euler circuit if and only if each of its vertices has even degree.

Theorem 2.5

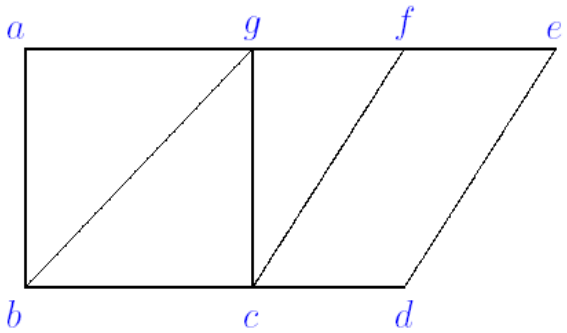
A connected graph has an Euler path but not an Euler circuit if and only if it has exactly two vertices of odd degree.

Example 2.6

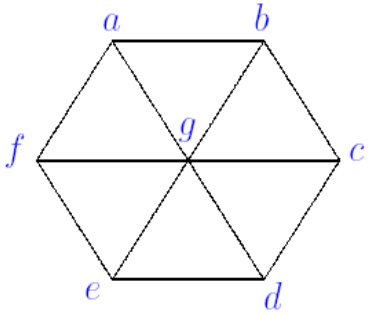
Which of the following graphs have an Euler path or circuit?



$G1$



$G2$



$G3$

Hamilton Path

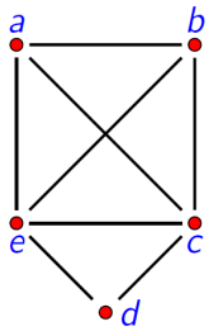
A path $v_0, v_1, \ldots, v_{n-1}, v_n$ in the graph $G = (V, E)$ is called a *Hamilton path* if $V = \{v_0, v_1, \ldots, v_{n-1}, v_n\}$ and $v_i \neq v_j$ for $0 \leq i < j \leq n$.

In other words, a Hamilton path visits every vertex of the graph exactly once.

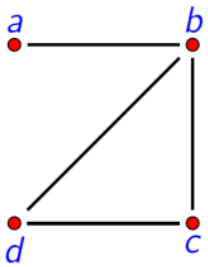
A circuit $v_0, v_1, \ldots, v_{n-1}, v_n, v_0$ (with $n > 1$) in a graph $G = (V, E)$ is called a *Hamilton circuit* if $v_0, v_1, \ldots, v_{n-1}, v_n$ is a Hamilton path.

Remark: There are no known efficient algorithms to detect existence of Hamilton paths or circuits. It is believed such algorithms do not exist.

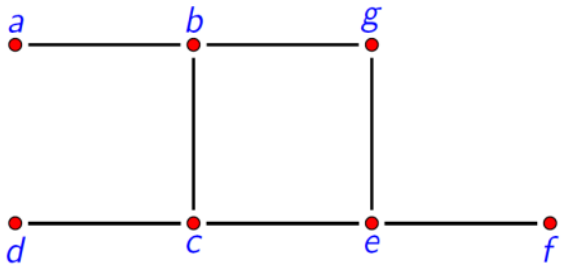
Example: Which of the simple graphs below have a Hamilton circuit or, if not, a Hamilton path?



G_1



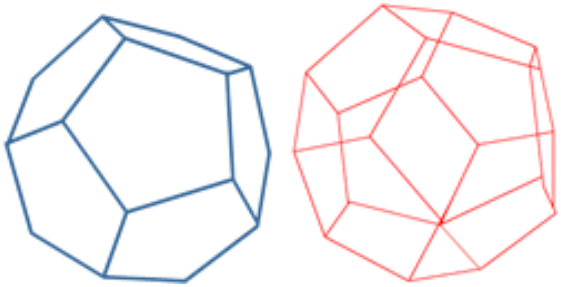
G_2

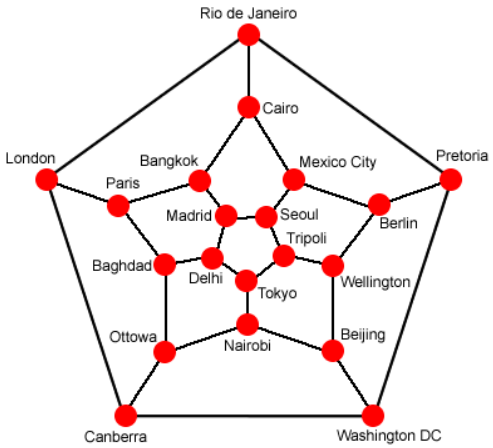


G_3

Hamilton's “Round the world” Puzzle

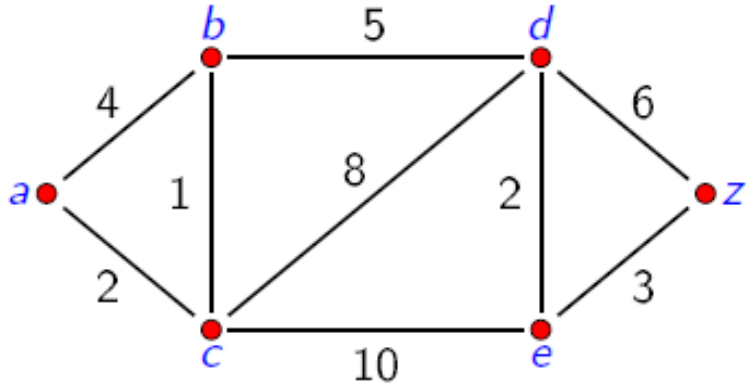
In 1859, the Irish mathematician Sir William Rowan Hamilton devised a puzzle with a regular dodecahedron which has 20 vertices, 30 edges and 12 pentagonal faces. He labelled each of the vertices with the name of an important city. The challenge was to find a route along the edges of the dodecahedron which visited every city exactly once and returned to the start.





3. Shortest Path Problems

A *weighted graph* is a graph that has a number (the *weight*) assigned to each edge. We will always assume nonnegative weights.
The *length* length(γ) of a path γ in a weighted graph is the sum of the weights of the edges of this path.



- Given $u, v \in V$, we denote any shortest path between u and v by SP(u, v).
- The *distance* between u and v is dist(u, v) = length(SP(u, v)).

Problem: Find the distance (or a shortest path) between two given vertices in a weighted graph.

Let $S = \{v: \text{SP}(u_0, v) \text{ is known}\}$. Observe that $u_0 \in S$ and $\text{dis}(u_0, u_0) = 0$, so $S \neq \emptyset$.

Dijkstra's Algorithm

INPUT (G, u_0)

$s := u_0$

$S := \{s\}$

$\overline{S} := V - \{s\}$

$d(s) := 0$ % for all $s \in S$, $d(s) = dist(u_0, s)$ %

for all $v \neq u_0$ **do** $d(v) := \infty$

while $\overline{S} \neq \phi$ **do**

  for all $v \in N(s) \cap \overline{S}$ **do**

    if $d(v) > d(s) + w(s,v)$ **then**

      % since $s \in S$ and $v \in \overline{S}, d(s) + w(s,v) = l(s,v)$ %

      $d(v) := d(s) + w(s,v)$

      PARENT$(v) := s$

  end for all

  $m := \min_{v \in \overline{S}} d(v)$

  $s := \arg\min_{v \in \overline{S}} d(v)$

  OUTPUT "$dist(u_0, s) = m$"

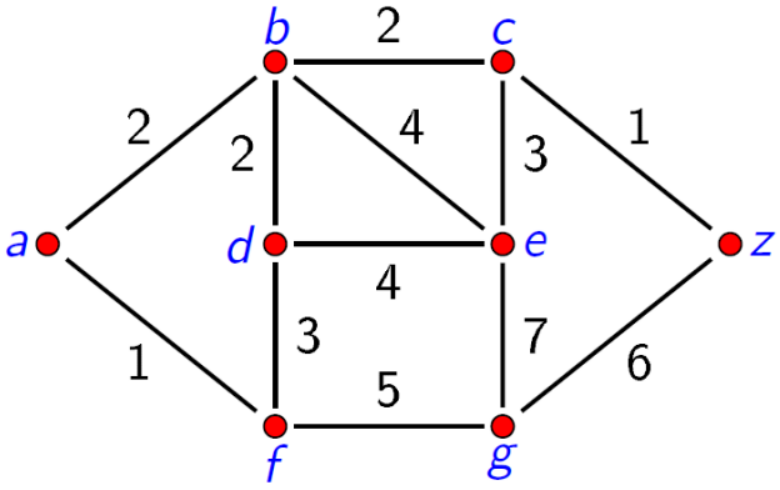
  $S := S \cup \{s\}$

  $\overline{S} := \overline{S} - \{s\}$

end while

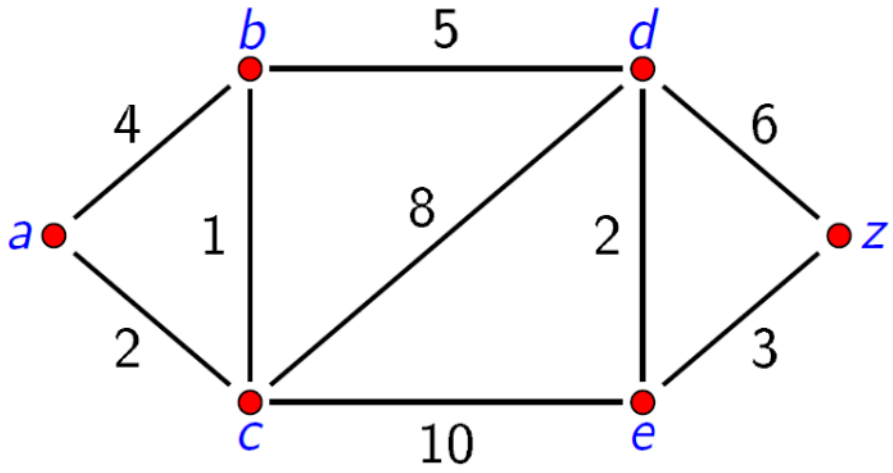
Example 3.1

Find the shortest path from *a* to every other vertex in the following graph.



Example 3.2

Find the length of the shortest path between the vertices *a* and *z* in the following weighted graph.

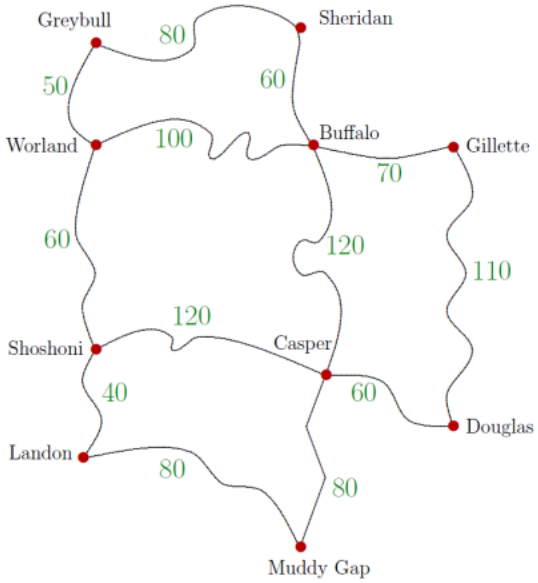


Theorem 3.1

Dijkstra's algorithm use $O(n^2)$ operations (additions and comparisons) to find the length of the shortest path between a vertex u and any other vertex in a connected simple undirected weighted graph with n vertices.

Travelling salesman problem

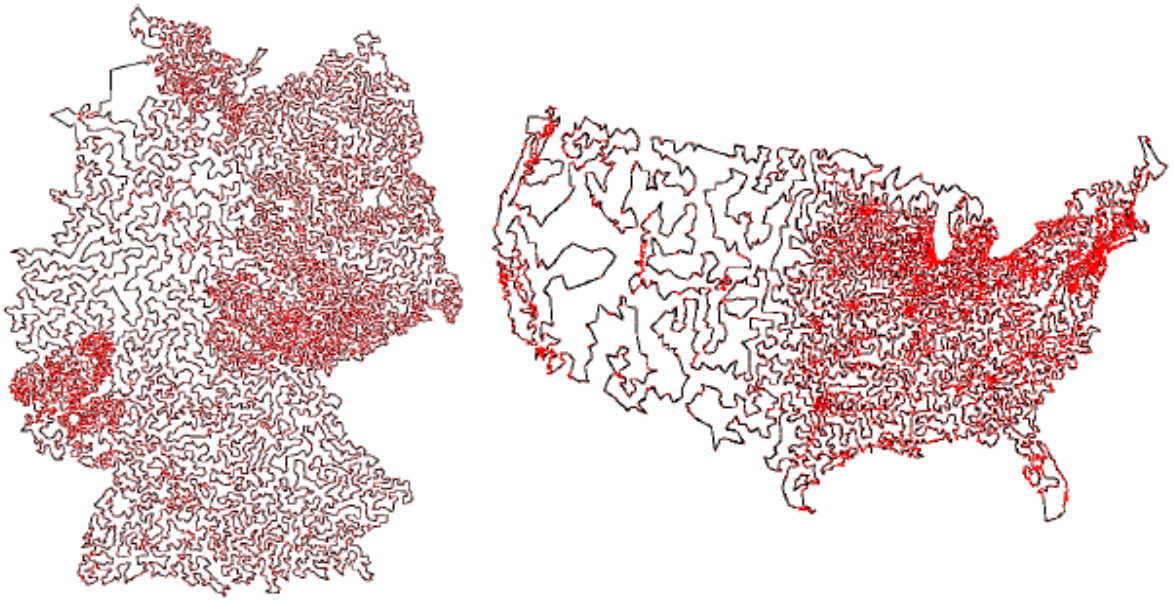
Given a number of cities and a road network, find a shortest route in which the salesman can visit each city one time, starting and ending at the same city.



Remark: As with the problem of deciding the existence of a Hamilton circuit, there are no known efficient algorithms solving the Travelling Salesman Problem and it is believed that such algorithms do not exist.

- Nevertheless, people still try to solve TSP problems, or subclasses of TSP, or try to get close to the optimal solution.
- One such program, which holds a few world records is Concorde by Applegat, Bixby, Chvátal, Cook (also available as iPhone app).

Some pictures (from www.math.uwaterloo.ca/tsp):

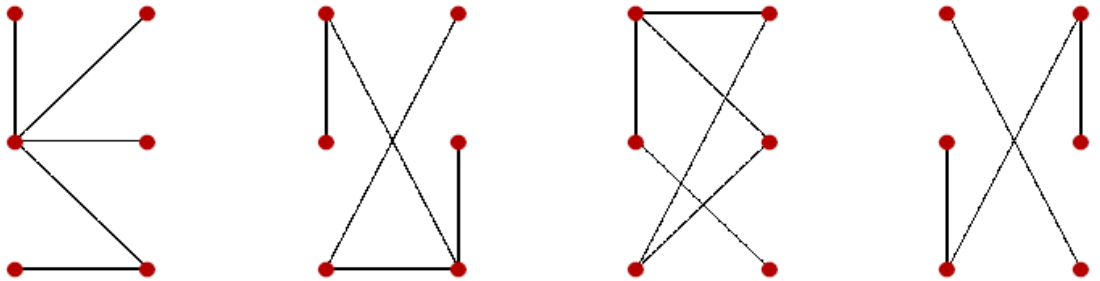


4. *Trees and binary trees*

A *tree* is a connected undirected graph with no simple circuits (a loop is regarded as a simple circuit).

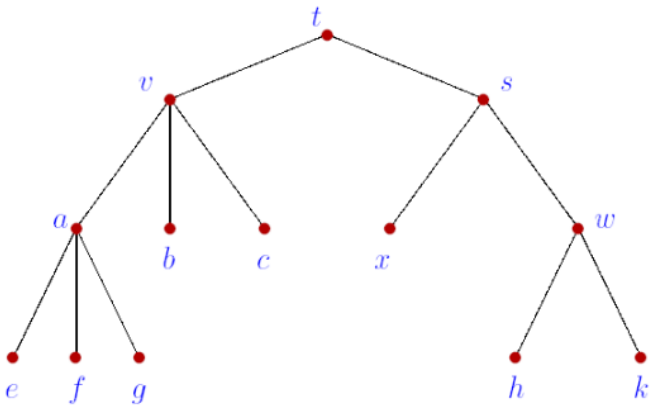
Remark: Any tree must be a simple graph.

Example 4.1
Which of the following graphs are trees?



It follows from the definition of *tree* that there is a unique simple path between any two of its vertices.

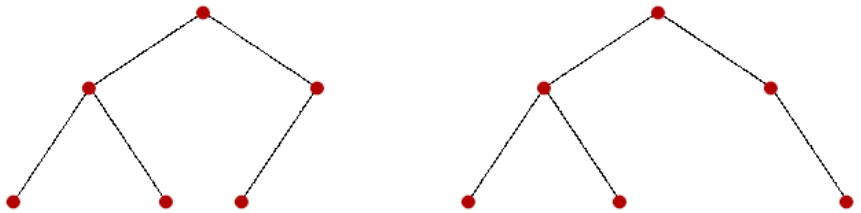
- A *rooted tree* is a tree T in which a vertex is distinguished from the rest. Such a vertex is called the *root* of the tree and (in graphical representations of T) is always placed at the top.



- The vertices which are k edges away from the root are said to be at *level* k. The root is at level 0. In the tree above, h is at level 3.

- For a vertex u, the vertices connected to it and at a level immediately below are called *children* of u. Vertex u is called the *parent* of those vertices.

- A vertex is called a *leaf* if it has no children. Vertices that have children are called *internal vertices*.
- A *binary tree* is a rooted tree in which every vertex has at most two children. In a binary tree, a left child is distinguished from a right child.



The above binary trees are different.

Theorem 4.1

Let G be a simple connected graph with n vertices and e edges. Then, G is a tree if and only if

$$e = n - 1.$$

Example (a) What is the number of trees with four vertices labelled from v_1 to v_4?

(b) What is the number of trees in (a) if v_1v_2 is an edge?

m-ary trees

- The *height* of a rooted tree is the maximum of the levels of its vertices.
- A rooted tree is called an *m-ary tree* if every internal vertex has no more than m children. The tree is called a *full m-ary tree* if every internal vertex has *exactly* m children.
- An *m-ary* tree with $m = 2$ is a binary tree.

Theorem 4.2

A full *m-ary* tree with i internal vertices contains $n = mi + 1$ vertices.

Theorem 4.3

A full *m-ary* tree with

(i) n vertices has $i = \dfrac{n-1}{m}$ and $l = \dfrac{(m-1)n+1}{m}$,

(ii) i internal vertices has $n = mi+1$ and $l = (m-1)i+1$,

(iii) l leaves has $n = \dfrac{ml-1}{m-1}$ and $i = \dfrac{l-1}{m-1}$.

Theorem 4.4

If a binary tree of height h has l leaves, then $\log_2 l \le h$.

An *m-ary* tree is *balanced* when all its vertices with at most $m-1$ children are at level at least $h-1$ (where h is the height of the tree).

Theorem 4.5

If a balanced *m-ary* tree T has height h then any *m-ary* tree with height less than h has less nodes than T.

Theorem 4.6

If a binary tree T with l leaves is balanced and full then its height is $\lceil \log_2 l \rceil$ where $\lceil n \rceil$ is the smallest integer which is larger than or equal to n.

Question: Given a large amount of items in a list. How should the items be stored so that an item can be easily located? Assume that the items belong to a totally ordered set.

A *binary search tree* is a binary tree T in which data are associated with the vertices.

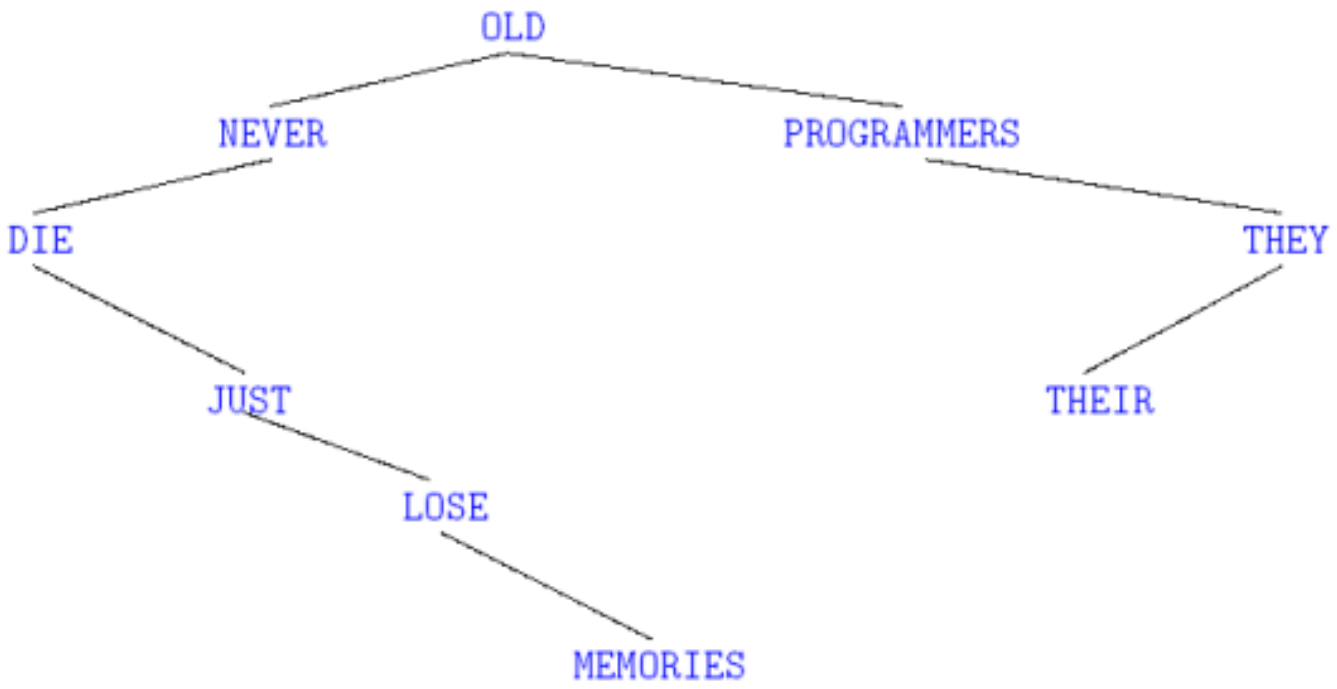
The data are arranged so that, for each vertex v in T,

- each data item in the *left subtree* of v is less than the data item in v,
- each data item in the *right subtree* of v is greater than the data item in v.

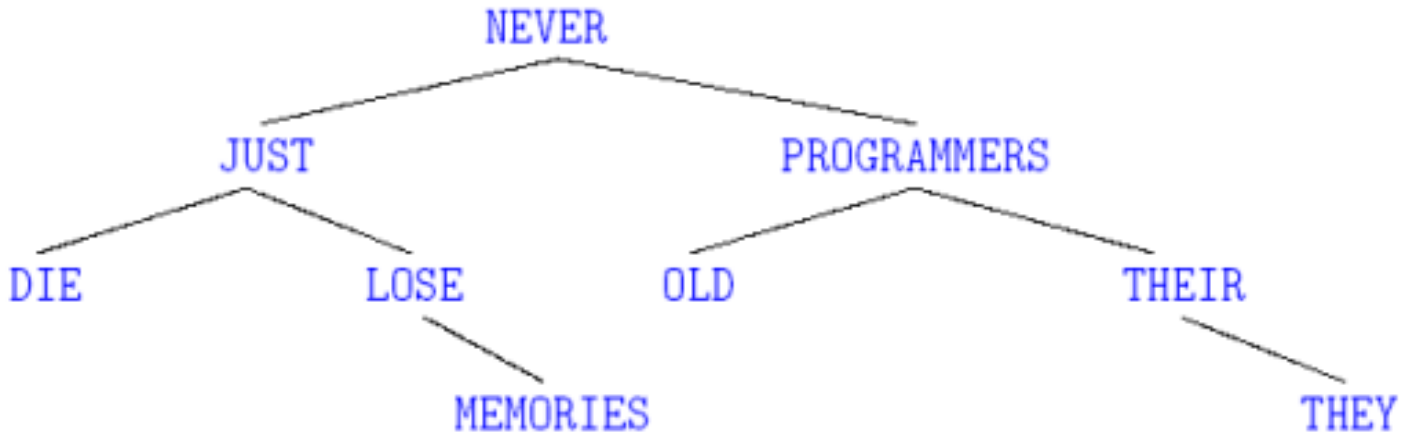
Example 4.2

Form a binary search tree for the words OLD PROGRAMMERS NEVER DIE THEY JUST LOSE THEIR MEMORIES using alphabetical order.

Solution:



In general, there will be many ways to place data into a binary search tree. The following shows another binary search tree:



Remark: If the height of a binary search tree is small, searching the tree is always fast.

Binary Search Tree Algorithm

The following algorithm finds out whether a given item is present in a given binary search tree, and if not inserts the item into the tree.

procedure *insertion* (*T:* binary search tree, *x*: item)
$v :=$ root of T.
{a vertex not present in T has the value null}
while $v \neq$ null and label$(v) \neq x$ **do**
    if $x <$ label(v) **then**
        if left_child$(v) \neq$ null **then** $v :=$ left_child(v)
        else add new vertex as a left child of v and set $v :=$ null
    else
        if right_child$(v) \neq$ null **then** $v :=$ righ_child(v)
        else add new vertex as a right child of v to T and set $v :=$ null
end while
if label $(v) \neq x$ **then** label new vertex with x
{$v =$ location of x} % if x was not present in T then $v =$ null.

Example 4.3

If *n* items are stored in a balanced binary search tree *T,* how many comparisons are needed to locate an item in the worst case?

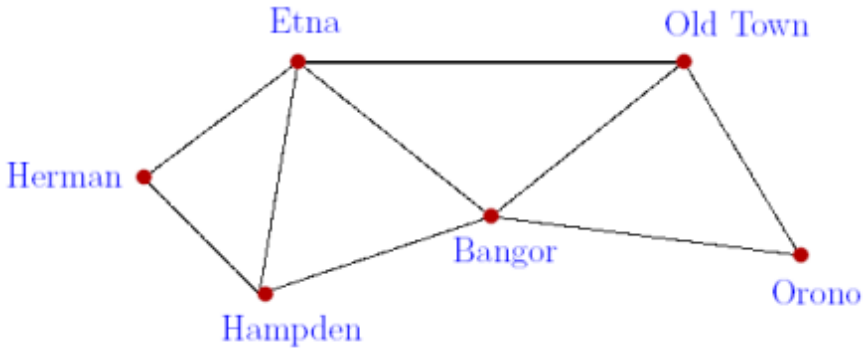
For example, if n =200 million, the number of comparisons is 20!

Summary:

- Binary search trees provide an efficient way to store large (totally ordered) data in a database.
- Key to the efficiency of this method is that the search tree is kept balanced.

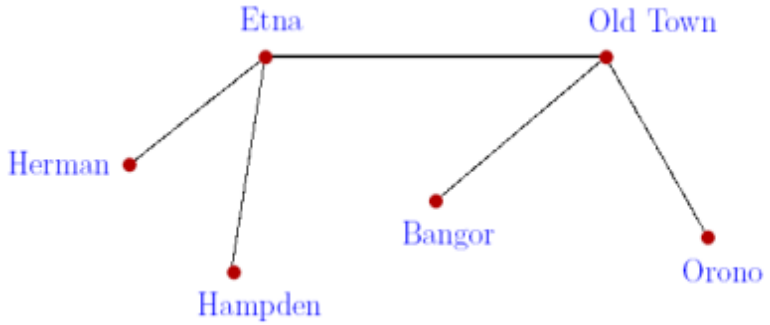
5 Spanning Trees

Consider the system of roads represented by the simple graph below:



The highway department wants to plow the fewest roads in winter so that there will always be cleared roads connecting any two towns. How can this be done?

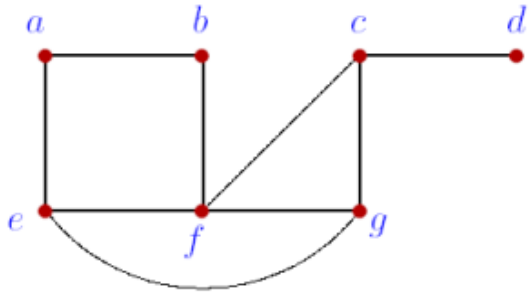
SPANNING TREE



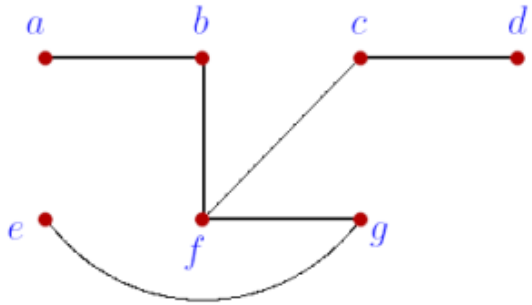
Let G be a simple graph. A *spanning tree* of G is a subgraph of G that is a tree containing every vertex of G.

Example 5.1

Find a spanning tree of the simple graph G below:



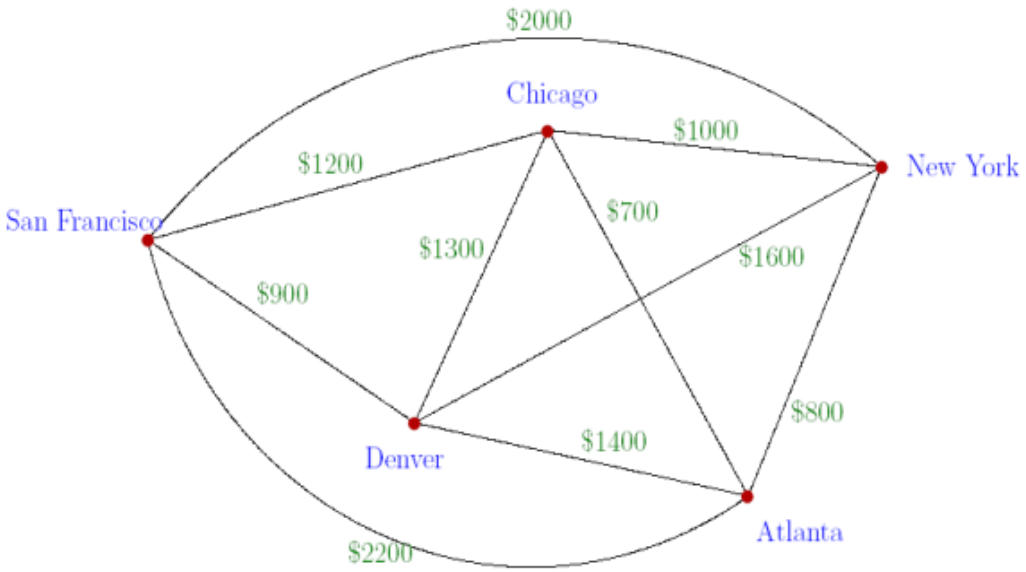
Solution: We can produce a spanning tree for G by removing the following edges that form simple circuits: ae, ef, cg.



Note that there are many spanning trees for a graph. A simple connected graph always contains a spanning tree.

Minimal Spanning Trees

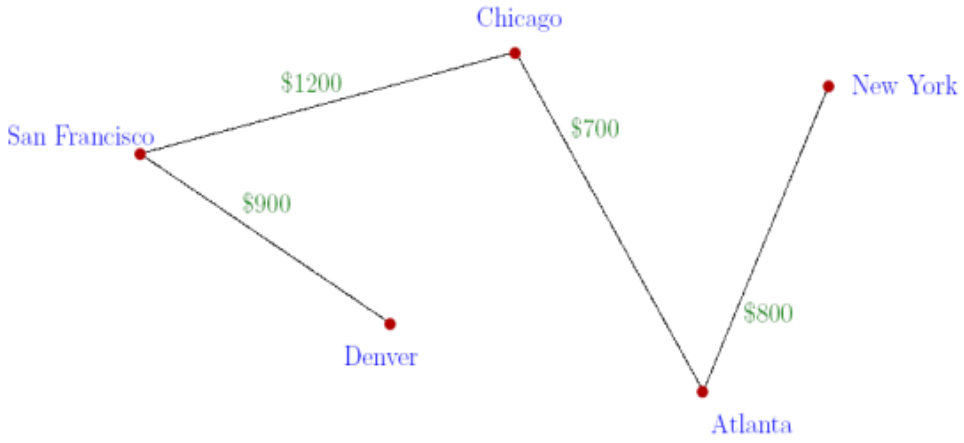
A company plans to build a communications network connecting the following five computer centers:



Monthly lease Costs for computer lines

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:



Problem: Find a spanning tree in a weighted graph such that the sum of the weights of the edges in the tree is minimal.

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

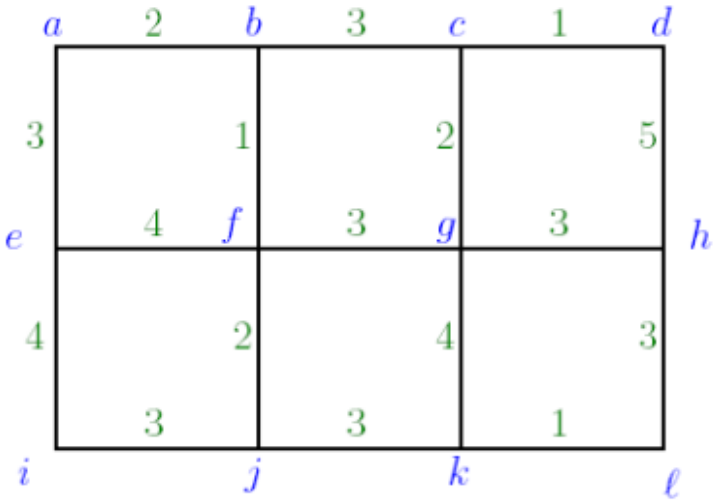
There are two standard algorithms to solve this problem: Kruskal's algorithm and Prim's algorithm.

Kruskal's algorithm

procedure Kruskal (G : weighted connected undirected graph with n vertices)
$T := \phi$
for $i = 1$ to $n - 1$ **do**
    $e :=$ any edge in G with smallest weight that does not form a simple circuit when added to T
    $T := T \cup \{e\}$
end for
{ T is a minimal spanning tree of G }

Example 5.2

Use Kruskal's algorithm to find a minimal spanning tree in the following graph.

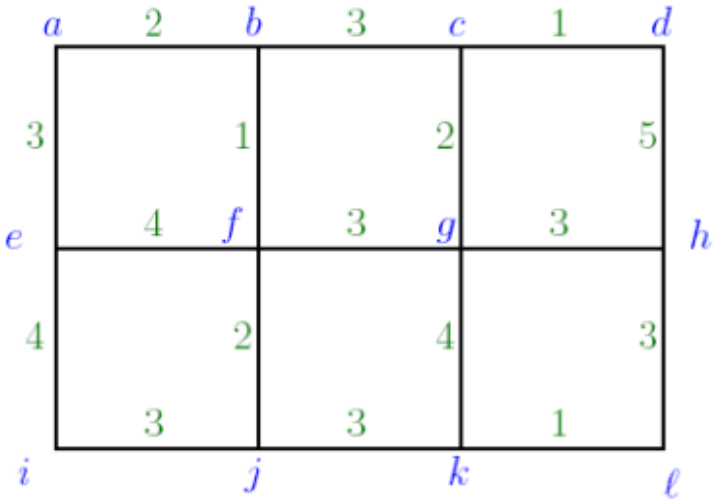


Prim's algorithm

procedure Prim (G : weighted connected undirected graph with n vertices)
e := any edge with minimal weight
$T := \{e\}$
for $i = 1$ to $n - 2$ **do**
    e := any edge in G with smallest weight joining a vertex in T with a vertex not in T
    $T := T \cup \{e\}$
end for
{ T is a minimal spanning tree of G }

Example 5.2

Use Prim's algorithm to find a minimal spanning tree in the previous example.



Kruskal vs. Prim

- Both algorithms produce minimal spanning trees.
- In Kruskal's algorithm one obtains a (single) tree only after the very last step, while Prim's algorithm maintains a minimal spanning tree for a subgraph of G the entire time; this subgraph is the whole graph after the last step.
- Both algorithms only take $n - 1$ "steps", but these steps involve several operations.
- It can be shown that Kruskal's algorithm can be implemented that it uses $O(e \log e)$ operations, where e denotes the number of edges of the graph, while Prim's algorithm can be implemented that it uses $O(e \log n)$ operations.
- Both algorithms are usually very fast; Kruskal's algorithm is preferred for sparse graphs (graphs with few edges), while Prim's algorithm is better for dense graphs (graphs with lots of edges).

Example

Suppose 1000 people enter a chess tournament. Use a rooted tree model of the tournament to determine how many games must be played to determine a champion, if a player is eliminated after one loss and games are played until only one entrant has not lost. (Assume there are no ties.)