Chapter 9 – Graph algorithms

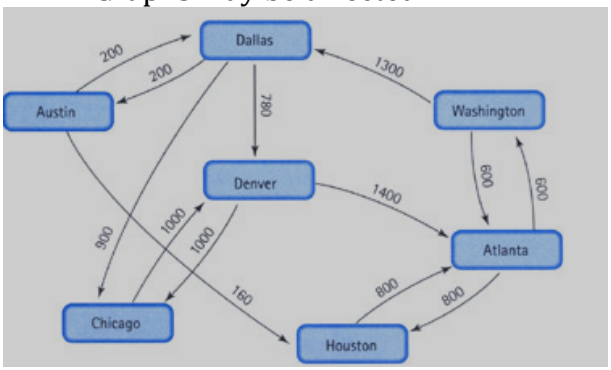
A graph represents relationships between pairs of objects – think graphs from mathematics (like Math 207 and 307). Graphs do not have a well-formed look/structure like trees and lists; for example, there is no obvious "start" to a graph; no fixed node to start at (no head as in a linked list, no root as in a tree).

Graphs are not search structures.

A graph is a collection of vertices (nodes) and edges that connect the vertices, often denoted G =(V, E), where

- V is the set of vertices (could be called nodes/objects, that hold data); V = {v_1, v_2, …, v_n}
- E is the set of edges (edges connect vertices that have some relationship); edges are represented as a pair of vertices, E = {e_1, e_2, …, e_k) where e_i = (v_j, v_k). Edges are also called arcs.
- Note – a tree is a graph, but a graph is not necessarily a tree

- Graphs may be directed



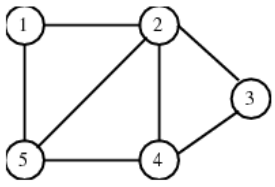
Directed graphs are often called digraphs.

The graph on the left is **directed** and **weighted** (in this example each edge carries a weight – possibly the distance between cities with direct flights between them. It says that there is a direct flight from Denver to Atlanta, but not from Atlanta to Denver.)

Image found at:
http://younginc.site11.com/source/5895/fos0094.ht

A directed graph with edges that carry values is called a **weighted graph** or **network**. (See text Figure 9.1, page 360).

- Graphs may be undirected – node direction goes both way



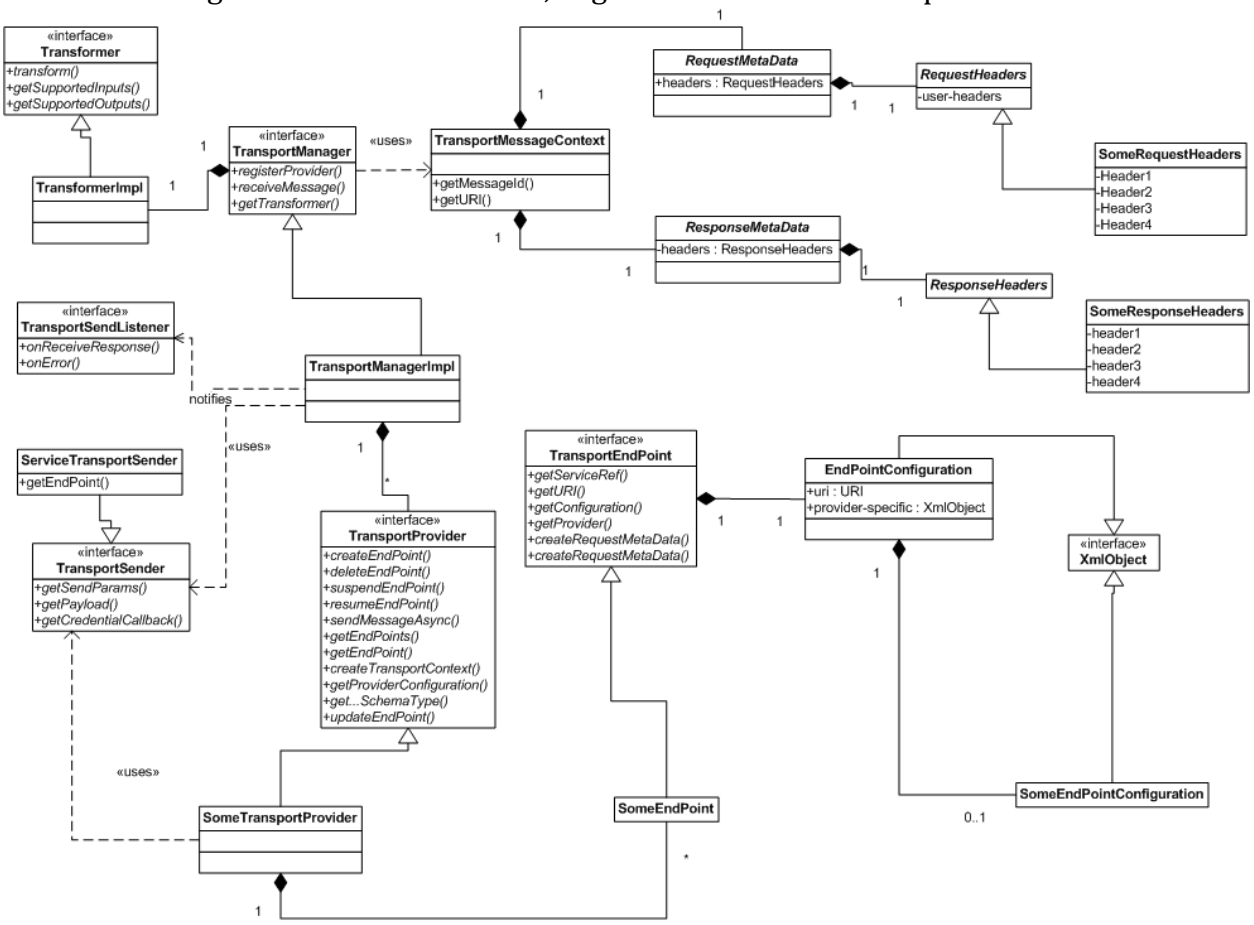
The edges in **undirected** graphs go both ways – for example, in the graph on the left includes an edge from 1 to 2 and from 2 to 1.

Image found at:
http://www.cs.sunysb.edu/~algorith/files/graph-data-structures-L.gif

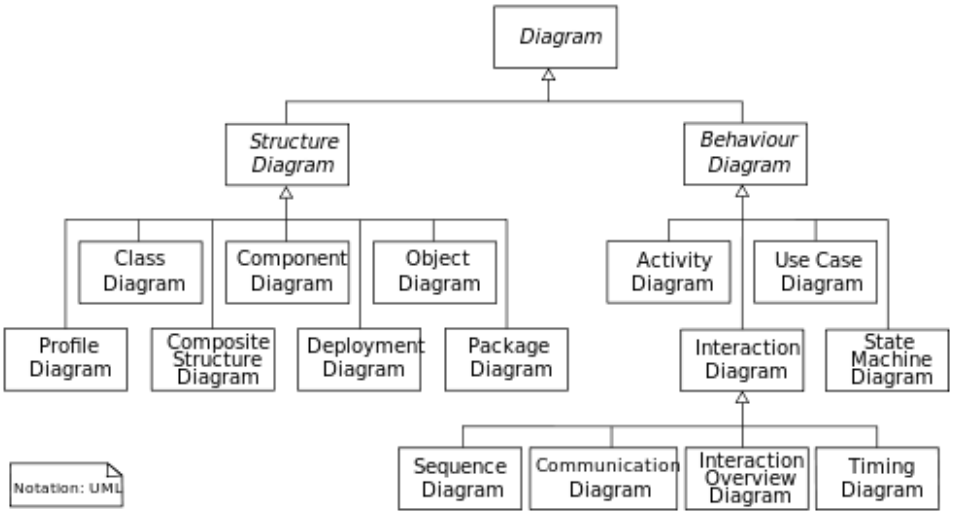
An undirected graph could be weighted.

Graph applications that come immediately to mind:

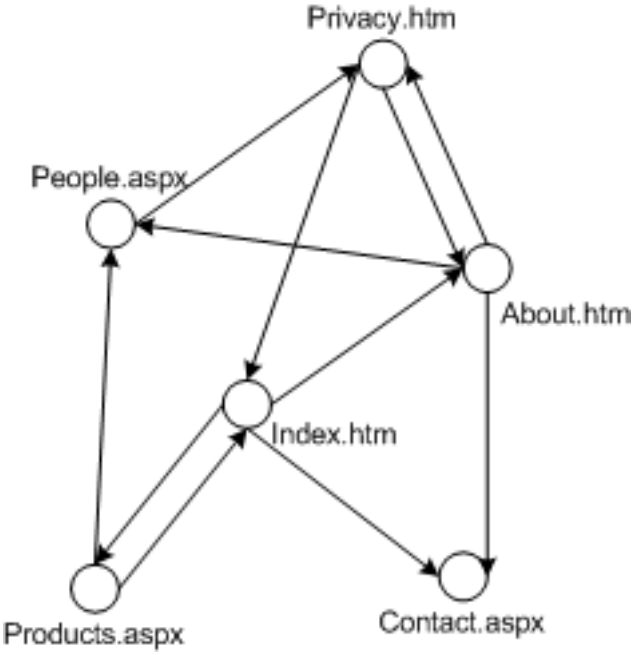
- UML diagram – classes are nodes, edges show the relationships between classes



- Special type of graph, a tree, that shows the many different types of UML diagrams



- Connected web pages – pages are nodes, edges show links between pages



- Facebook friends – vertices are people on facebook; edges connect friends

- Methods in a program that call each other

- Google maps / any map – nodes are cities, edges connections between cities (first page)
- Family trees – vertices are people; edges are connections
- CSCI course prerequisites – courses are nodes, edges are dependencies (see Figure 9.3, text)
- The internet itself – nodes are computers, edges are connections

Regardless of the application, the same algorithms can be used to manipulate the graphs.

Traveling salesperson problem (TSP) – (probably the most famous graph problem)

- Given a set of cities (vertices) and the distances between them (edges) find the tour (visit of all cities) that allows a salesperson to visit all cities and return home most efficiently (covering the least distance)
    - Hamiltonian tour – a tour that visits all cities exactly once
    - Hamiltonian circuit (HC) – a tour that starts at any node, visits all other nodes and returns to the start
    - TSP looks for an optimal (minimal in this case) HC
    - If a there are n cities, there are as many as **(n-1)!** possible tours; if n = 15 and we assume that we can travel between any two cities, there are 87,178,291,200 possible tours. Which is the optimal one?

Traveling salesperson problem

- Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

An optimal TSP tour through Germany's 15 largest cities. It is the shortest among 43,589,145,600 possible tours visiting each city exactly once.

Data and picture from Wikipedia, extracted 10-30-09.

8

Other relevant terminology:

- Nodes connected by an edge are called **adjacent**
- The **degree** of a vertex is the number of edges that connect to it. (For a directed graphs, we can also distinguish *in-degree* from *out-degree*.)
- **Path** – sequence of one or more vertices $v_1, v_2, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$. The length of the path is the number of edges on the path.
- **Cycle** –a path from a node to itself
- **Simple path** – all nodes, except possibly the first and last, are unique
- **Acyclic graph** – contains no cycles
- **Connected graph** – there is a path from every vertex to every other vertex
- **Complete** – every node is directly connected to every other node
- *u* ***reaches*** *v* if there is a directed path from *u* to *v*.
- **Strongly connected** – applies to directed graphs; for any two vertices *u* and *v*, *u* reaches *v* and *v* reaches *u*.

Graph representations: two typical representations, but there are variants

1 - **Adjacency matrix** – Basic representations of edges, vertices, V and E.

- A 2-dimensional *n* x **n** array, A, such that A[i, j] used to indicate edges that exist
  - Rows and columns indentify vertices –
    - Vertices typically stored as a distinct integer index, in the range 0, 1, …, *n*-1 (below, a mapped to 0, etc)
  - A true value in location A[v,w] indicates that edge (v,w), from v to w exists
- Other options: Weights could be stored instead of Booleans if a weighted graph

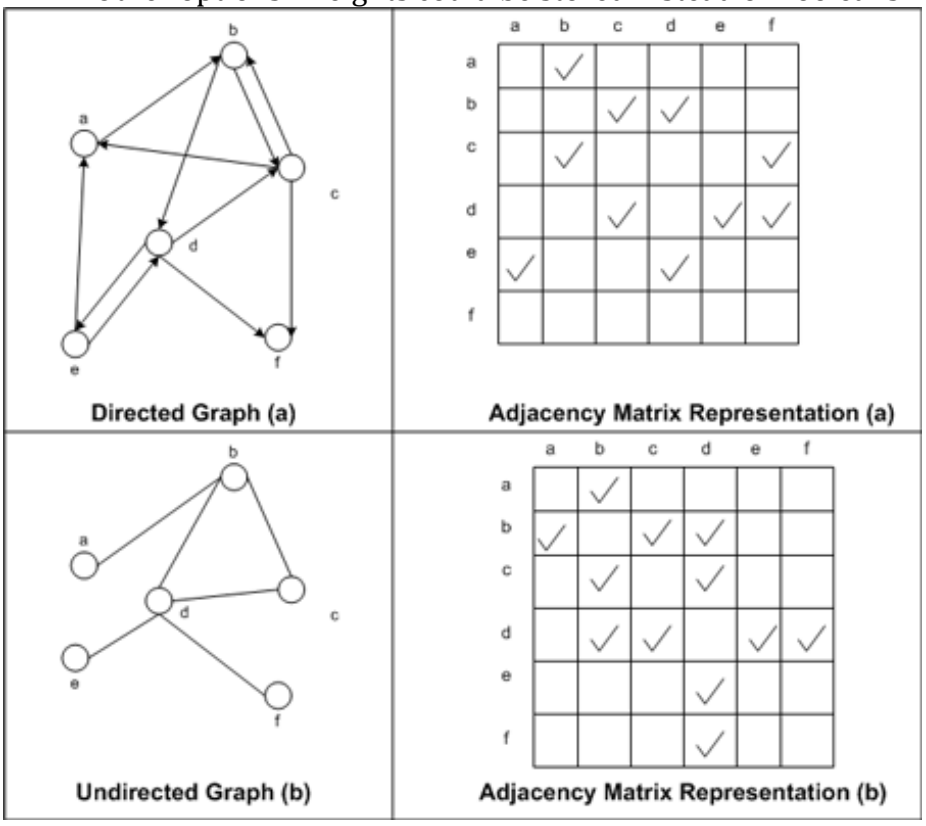


Figure from http://msdn.microsoft.com/en-us/library/ms379574(v=vs.80).aspx

Advantages of this implementation?

Disadvantages of this implementation?

1. Adjacency list

- Vertices stored as a list objects – along with additional data
- Every vertex stores s list of adjacent vertices
    i. For a weighted graph, weights can be stored with adjacent vertices in the appropriate listlist

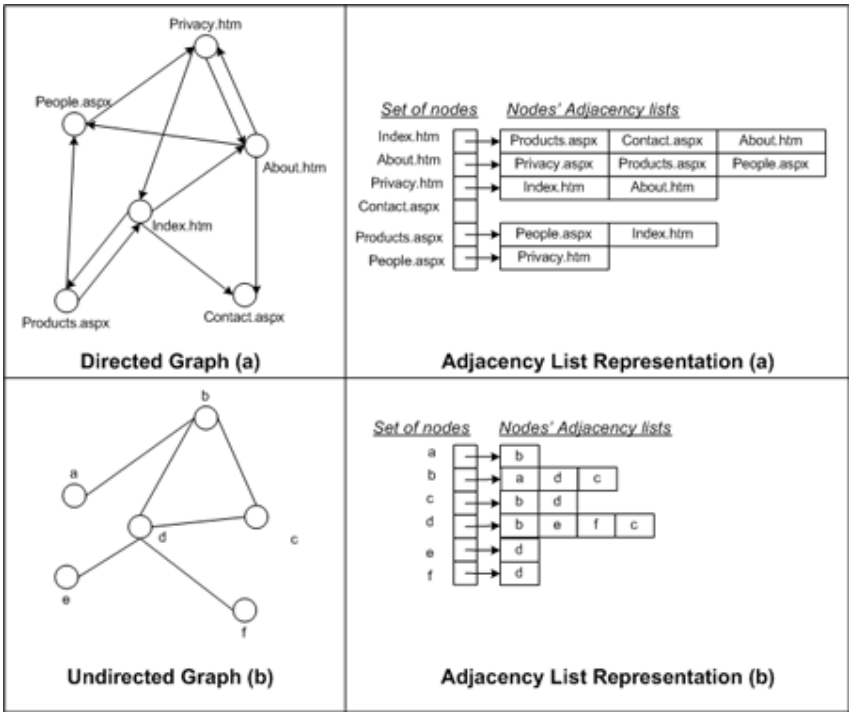


Figure from http://msdn.microsoft.com/en-us/library/ms379574(v=vs.80).aspx

For another picture, see Figure 9.2, text.

Actual implementation:

- Array or array list which references vertex objects
- Vertex objects store vertex label, other data associated with vertex, and reference to the list of nodes adjacent to it.
- For undirected graphs, edges both ways recorded; if (a,b) is an edge, a is in b's adjacency list and b is in a's adjacency list
- Map can be used to store vertices as keys, and the values they reference are the adjacency lists

Advantages of this implementation?

Disadvantages of this implementation?

Other storage option: **incidence list**

Each vertex and edge is an object.

- Each vertex stores the edges that it is incident with (those it touches)
- Each edge stores the vertices that it is incident with (those it connects)
- Use of an object provides room for additional information, for example, the weight of the edge can be stored in the edge's object

Common graph problems/algorithms and their application:

Problem/algorithm	**What**	**Practical applications**
topological sort	List vertices of a DAG in an order in which they can be visited, based on paths	Scheduling a sequence of jobs based on their dependencies, such as list course prerequisites
single source shortest path	Weighted or unweighted graphs of N vertices, find shortest path from one vertex v to every other vertex (N-1 paths between v and other nodes)	Communication costs between networked computers; cheapest air fare from city 1 to any other city
all-pairs shortest path	determination of the shortest graph distances between every pair of vertices in a given graph	Transitive closure – if there is a path from a to b and b to c, reveals a path from a to c.
network flow problems	Detemination of the maximum capacity from a vertex to another vertex;	Flow of water through pipe system, flow of people through transportation system, flow of electricity through electrical system, etc
minimum spanning tree	Undirected graph – find a set of edges, with minimum total weight, that connect all vertices	Design of computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids (which they were first invented for, according to wikipedia)
depth-first search	Start at some node v, and recursively visit all nodes adjacent to v.	Determine connectivity

9.2

Situation: Suppose a directed graphs models course prerequisites – how can we get a computer to list the order one must take the courses? (see page 363).

Topological sort – an **ordering** of vertices in a **directed acyclic graph** such that if there is a path from a to b in the graph, then b appears after a in the ordering.

Directed?
Acyclic?

Assume an adjacency list that stores the vertices in a DAG (like figure 9.4 stored as in Figure 9.2).

An algorithm (algorithm 1):

- Assume all nodes marked with "indegree"
- Repeat
    - Find a vertex with indegree 0, add it to the list, and remove it from the graph, updating the indegrees of nodes affected.
    - Until graph is empty

Slightly more efficient algorithm uses a queue to hold vertices with indegree 0 (algorithm 2):

- Assume all nodes marked with "indegree"
- enqueue all vertices of indegree 0
- while queue not empty
    - dequeue a vertex x
    - for all vertices y adjacent to x
        - decrement indegree of y
        - if y's indegree is now 0, enqueue y

Why is algorithm 2 better than algorithm 1?

How could a stack be used to design an alternative algorithm?

- A traversal through the graph is made and all nodes with incounters of 0 are pushed onto a stack S – this is the collection of possible "next" nodes in the sort

```
while !S.isEmpty( )
     u = S.pop( ) // mark u as i-th node visited
     for each node directly reachable from u (by a single
edge),
     decrement the  node's indegree
     if any indegree reaches 0, push that node onto the
stack
```

Trace above on graph in text, page 363, figure 9.4 or 9.3.

9.3 Shortest-path algorithms

Single source shortest path problem

For a designated node s – find the shortest path from s to every other node

a. unweighted version of problem – simply based on number of edges between vertices (on paths from start node)

Identify the source vertex, s

- Mark s with a 0, for path of length 0
- Find vertices adjacent to s, mark them with 1
- Find vertices adjacent to vertices marked 1, mark them with 2,
- Continue this process until all vertices are marked.

The above process is "breadth-first".

Figure 9.16 provides pseudocode for this algorithm.

- For each vertex, maintain
  - Distance from source, all except source started at infinity
  - Known indicates if its been visted
  - Path - vertex immediately before it in the path
- Searches through all vertices on every pass.

Figure 9.18 provides a more efficient version that uses a queue; visited vertices added to queue

- For each vertex, maintain
  - Distance from source, all except source started at infinity
  - Path - vertex immediately before it in the path
- Known variable not necessary, visited vertices not reconsidered; vertices to visit on queue

2. weighted version of problem – based on weights on edges

Dijkstra's algorithm –

Dijkstra – (from Wikipedia) was a Dutch computer scientist. He received the 1972 Turing Award for fundamental contributions to developing programming languages.
Among his contributions to computer science are the shortest path algorithm, also known as **Dijkstra's algorithm**; **Reverse Polish Notation** and related Shunting yard algorithm; the THE multiprogramming system, an important early example of structuring a system as a set of layers; **Banker's algorithm** (operating systems) and the **semaphore** (operating systems) construct for coordinating multiple processors and programs. Another concept due to Dijkstra in the field of distributed computing is that of self-stabilization – an alternative way to ensure the reliability of the system. Dijkstra's algorithm is used in SPF, Shortest Path First, which is used in the routing protocols OSPF and IS-IS.

Similar to the algorithms for unweighted graphs, but when visiting new nodes, overall distances to nodes adjacent to them are updated if the new distance is smaller than the known distance.

Dijkstra's algorithm called **greedy** because at each decision point, the best current solution is chosen.

"Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage. "

Coin changing problem

- Assuming an infinite number of coins on all denominations, make change for an amount A using the fewest number of coins.
- Greedy rule: select the largest denomination available

What is greedy solution for coin set with denominations 10, 5, 1 and the need to give $0.12 in change? Is it optimal?

What is greedy solution for coin set with denominations 10, 6, 1 and the need to give $0.12 in change? Is it optimal?

How do greedy algorithms work:

- Builds solution to problem in steps
- At each step (iteration), it adds a part to the solution
  Ex. A greedy coin-changing algorithm might choose a coin at each step
  Ex2. Single-source, shortest path problem chooses one new edge at each step
- Always choose the current best - a local optimum
  Ex. Greedy coin-changing algorithm that seeks to return fewest coins, might return the coin of largest denomination
  Ex2. Single-source shortest path problem chooses the edge of smallest weight at each step

Do local optima guarantee an optimal solution at the end (for the big problem)? Not for all problems

Look at another couple of greedy algorithms before moving on to "all-pairs shortest path problem" which is based on a different algorithmic technique.

Minimum spanning tree algorithms – section (section XX Prim & Kruskal both "greedy" but in different ways) - a minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost. A minimum spanning tree exists if and only if G is connected.

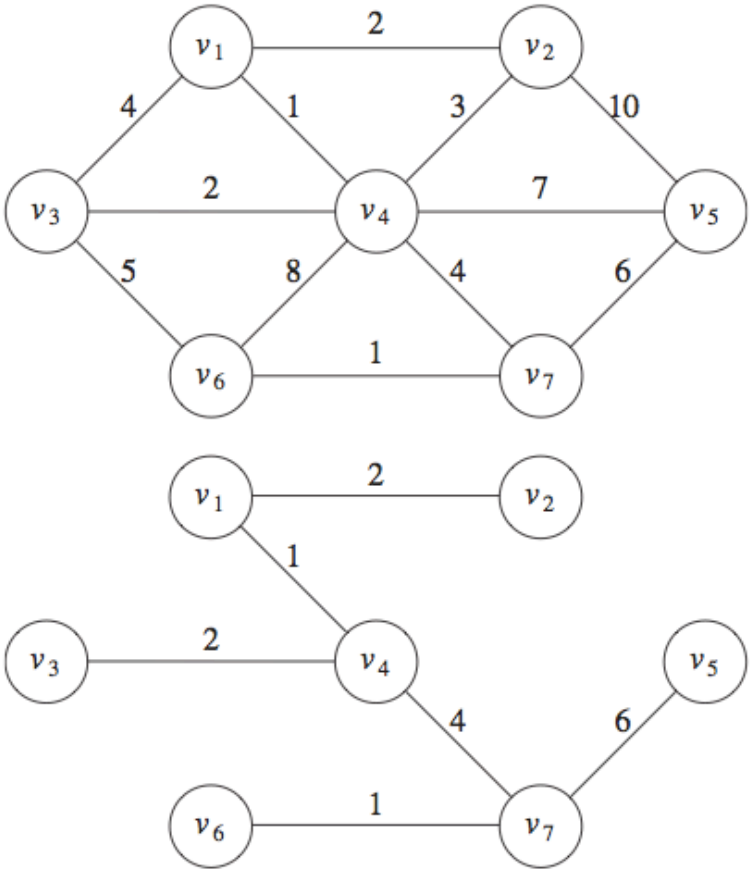


Figure 9.50 A graph G and its minimum spanning tree

Prim's algorithm – first chooses minimum weight edge, marks its vertices as visited, then chooses minimum weight edges (greedy) incident to one other already visited vertex as long as no cycle is generated, until all vertices visited, results in a **mst**

Note: the graph being built is always a single tree (not true in Kruskal's algorithm)

One solution via Prim:

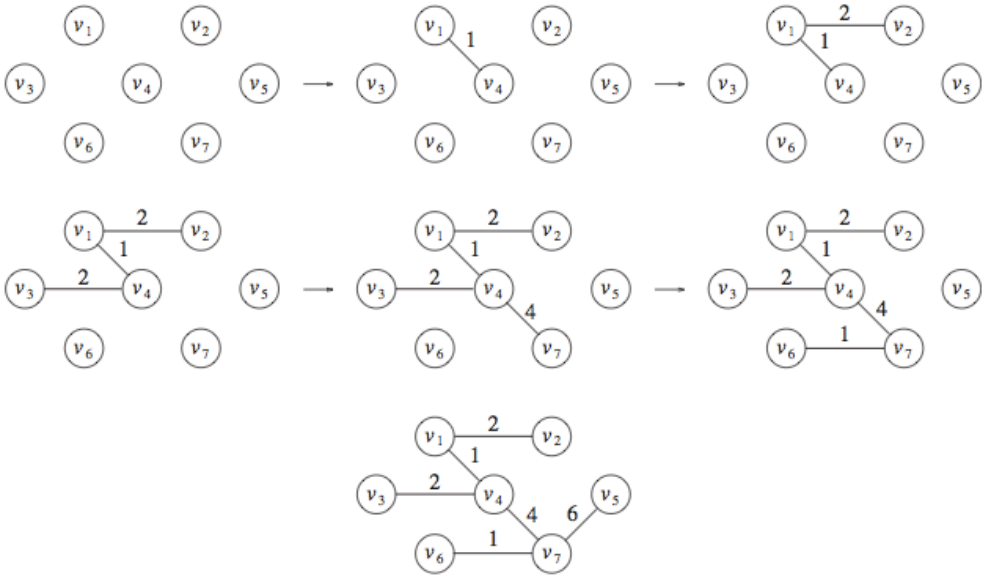


Figure 9.51 Prim's algorithm after each stage

Kruskal's algorithm: chooses lowest weight edges, that don't form a cycle, until all vertices are visited.

Note: structure being built may not be a single tree – may be unconnected edges or multiple trees.

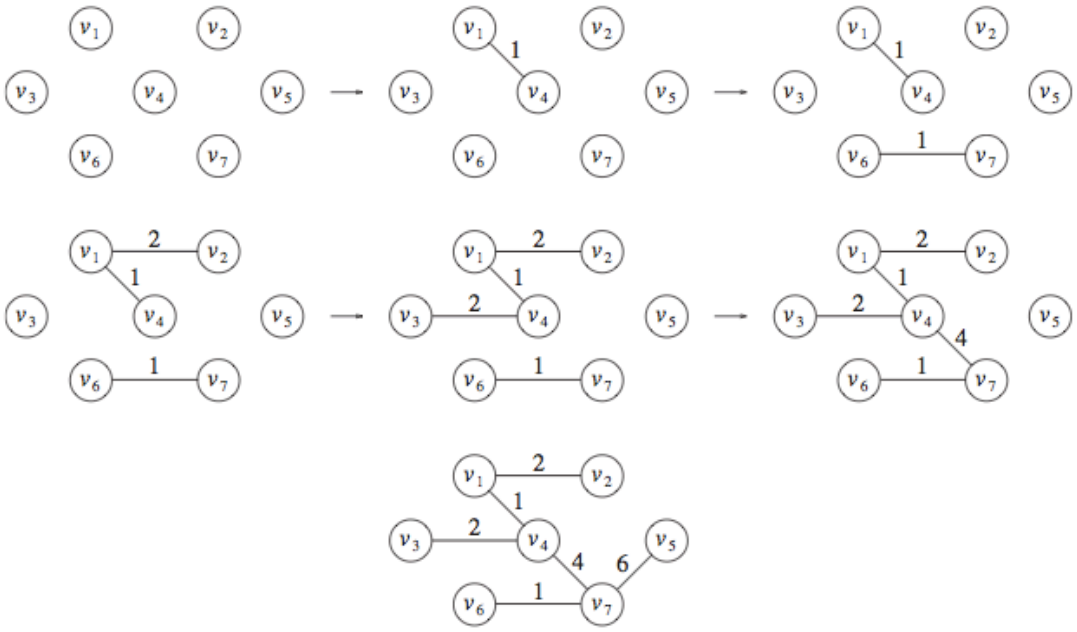


Figure 9.59 Kruskal's algorithm after each stage

Back to **all-pairs, shortest path problem** – could do Dijkstra's algorithm |V| times. Another solution (Floyd-Warshall algorithm) takes a different approach. This approach is called dynamic programming.

Dynamic programming:
- Solves simplest subproblems first
- Uses solutions to subproblems to solve bigger subproblems
- Works up to solution of original problem

Divide and conquer also breaks problem into subproblems, but then solves then from the top down often recursively. Think binary search, quicksort (haven't seen in detail yet), Fibonacci, computing 2^n

```
ALGORITHM: fib(n) {
        if (n == 0) return 0
        if (n == 1) return 1
        return fib(n - 2) + fib(n - 1)
}
```

```
ALGORITHM twoPower( int n ) {
    if (n == 0)  return 1;
    else
        return twoPower(n-1) +
                twoPower(n-1);
}
```

Dynamic programming solves subproblems, from the bottom up

In case of Fibonacci problem, computes and stores numbers in sequence so that they can be recalled when needed:

Divide & conquer

```
ALGORITHM: fib(n) {
        if (n == 0) return 0
        if (n == 1) return 1
        return fib(n - 2) + fib(n - 1)
}
```

Dynamic programming

```
ALGORITHM: fib(n){
  Fib[0] = 0;
  Fib[1] = 1;
   for i ← 2 to n
      Fib[i] = Fib[i-1] + Fib[i-2];
```

In case of 2^n problem:

Divide & conquer

```
ALGORITHM: pow2(n) {
   if (n == 0) return 1
  else
       return twoPower(n-1) +
                twoPower(n-1);
}
```

Dynamic programming

```
ALGORITHM: pow2(n){
  Pow2[0] = 0;
  for i ← 1 to n
      Pow[i] = Pow[i-1] + Pow[i-1];
```

All-pairs, shortest path problem (algorithm on page 473):

Stores edges costs – version 0 of table
Determines if any two vertices "closer" if it takes a path through vertex 1; updates table
Determines if any two vertices "closer" if it takes a path through vertex 2; updates table
…

9.6 Depth-first Search

We saw depth-first and breadth-first searches in the chapter on trees. Earlier in this chapter we did a breadth-first search of an unweighted graph to compute the single source shortest path for an unweighted graph.

In some instances, a depth-first search may be appropriate, so let's consider what a depth-first search of a graph would look like: see picture on pages 410.

9.7 Introduction to NP-Completeness

Revisiting the **traveling salesperson** problem – given a complete, weighted graph, is there a path from a given (start) vertex that visits all other vertices exactly once, and returns to the start vertex and has a total cost ≤ K?

If there are **n** nodes, how many possible cycles are there? N!

TSP is a combinatorial optimization problem.

- The number of possible solutions is "combinatorial" grows as the number of combinations grows (related to n!)
  - Combination – number of ways of selecting k items out of n items, order does not matter
  - Permutations – number of ways of selecting k items out of n items, order does matter; many problems have this many possible solutions
- Consider the possible paths the travelling salesman might take, in a 4 city tour:

A B C D	B A C D	C A B D	D A B C
A B D C	B A D C	C A D B	D A C B
A C D B	B D A C	C B A D	D B C A
A C B D	B D C A	C B D A	D B A C
A D B C	B C A D	C D A B	D C A B
A D C B	B C D A	C D B A	D C B A

- N cities, n = 4, 4! possible paths enumerated
- salesperson can start at any city, but it is OK to fix the start city since s/he likely starts from home anyway.
- Fixing the start city, brings the list down to 3! Or (n-1)!
- Divide by 2? Since ABCD and ADCB would give same result
  - For large n, (n-1)!/2 still too big for arbitrarily large n
  - For example that n = 15; 14!/2 = 43589145600

Exhaustive search (brute force) not reasonable for any large n. Need other ways to tackle this. Can't check this many paths in any reasonable amount of time.

Chapter 9 notes –

What is reasonable? Polynomial time is usually considered reasonable:

c, log n, n, n log n, n^2, n^3, n^k where k is a constant

TSP is on the order of n! – this is not polynomial.
Other algorithms are on the order 2^n – inefficient method of computing Fibonacci – not polynomial

But, given a candidate solution for the TSP, could you verify it in polynomial time? That is, could you verify that (1) it is a circuit visiting every node and (2) that its cost is less than some k, in polynomial time? Sure!

These are the characteristics of an NP problem.

NP – non-determinististic polynomial

Non-deterministic part: guess a solution
Polynomial part: verify it with polynomial-time algorithm

NP problems called intractable, not feasible, not practical.

There is a class of NP problems that have been shown to be NP-complete – that is, any NP problem can be reduced to an NP-complete problem. That means we can restate the NP problem so that a solution to the NP-complete problems, solves it.

Facts about NP-complete problems: (from my bulletin board)

P is a subset of NP.

Are all problems in NP?

No, some are worse than NP.

Is there no Hamiltonian circuit in a given graph? Would have to try up to n! possible paths to verify that none form a circuit. Can't just guess one tour and verify it, must check every possible tour and make sure that none exists. Can't do this in P time.

Some problems not decidable at all: the Halting Problem

- Given an algorithm and an input, does the algorithm halt on the input? (or does it loop forever?)
    - If this were solvable, we could build compilers or algorithms to run programs that could detect an infinite loop and stop it.

Chapter 9 notes –

Techniques for dealing with hard problems, like NP problems: this is what Chapter 10 is about: greedy algorithms, divide-and-conquer, dynamic programming, randomized algorithms etc.