GRAPHS.

1. Graphs are useful models for reasoning about relations among objects and combinatorial problems. Many real-life problems can be solved by converting them to graphs. Proper application of graph theory ideas can drastically reduce the solution time for some important problems.

2. DEFINITIONS.

   A graph has a set vertices V, often labeled v1, v2, etc.. and a set of edges E, labeled e1, e2, ... Each edge is a pair (u, v) of vertices. We write G = (V, E) for the graph with vertex set V and edge set E. In applications, where pair (u, v) is distinct from pair (v, u), the graph is "directed". Otherwise, the graph is undirected. We can convert an undirected graph to a directed one by duplicating edges, and orienting them both ways.

   When (u, v) is an edge, we say "v is adjacent (neighbor) to u". A loop is an edge with both endpoints being the same.

   The out-degree of v = the number of neighbors of v
   The in-degree of v = how many vertices have v as a neighbor.

   Some times, the edges can be associated with weights or costs.

   Paths.

   A path is sequence of vertices w1, w2,.., wn, such that each pair (wi, wi+1) is an edge.
   The length of a path is the number of edges in it, or total weight if there are weights.
   A simple path has no repeated vertex, except first and last can be the same; in that case, the path is a cycle.

   Connectivity.

   An undirected graph is connected if there is a path between any two vertices. A directed graph with this property is "strongly connected."
   A weakly connected graph---underlying graph connected but the directed graph not strongly connected.

   Examples of Graphs.

   1. airport system:
      nodes = airports; edges = pairs of airports with non-stop flights.
      (weight/cost = airfare; distance; capacity)

   2. Internet:
      nodes = routers; edges = links.

   3. social graphs: (6 degrees of separation)
      nodes = people; edges = friends/acquaintance/co-authors

4. academic graphs:
    nodes = courses; edges = prereqs;

3. REPRESENTATION

Adjacency MATRIX: a 2-dim array V x V. For each edge (u,v), set A[u,v] true; equal to cost, etc. Use infty or 0 for non-edges.

Pros: easy to check if (u,v) an edge in G.
Cons: Takes V^2 space if even graph has very few edges; e.g. street map
     A steet map is O(V) edges. Imagine V = 10^6.

Adjacency LIST. An array of (header cells for) adjaceny lists. The ith cell points to a linked list of all vertices adjacent to vertex vi.

Example

```
1:      2, 4, 3
2:      4, 5
3:      6
4:      6, 7, 3
5:      4, 7
6:
7:      6
```

Space is O(E); each directed edge stored just once. Thus, if G is undirected (u,v) appears in lists of both u and v.

Pros. Linear space. Easy to list out all vertices adjacent to u.

4. TOPOLOGICAL SORT

An application: You have a set of tasks. You are also told a set of precedence relations; some jobs cannot be done before others. How shall you schedule the jobs without violating any prec constraint?

Job -> nodes; precedeance relations -> edges.

Clearly, if there is a cycle in the graph, no feasible schedule.

When there is no cycle, *topological sorting* is an ordering of vertices such if there is a path from vi to vj, then vi appears BEFORE vj in the schedule.

Algorithm:

```
Find a vertex v with zero in-degree   (must exist!)
Print v, delete v, and its outgoing edges;
Repeat;
```

Take O(V^2) time.

Improved Topological Sort

```
Compute all vertices' indegrees
Enqueue all those with zero indegree
Pick a vertex w from the queue;
        put w next in schedule
        for each vertex v adj to w
                decrement v's indegree
                add v to queue if its indeg = 0
```

This code only looks at each edge once, so O(E) time.

EXAMPLE.

5. SHORTEST PATHS.

Assume c(u,v) is the cost of traversing the edge (u,v).
Cost of a path v1, v2, ..., vk is sum_{i=1}^{k-1} c(vi, vi+1).

Single-Source SP Problem:
Given a weighted directed graph G=(V,E), and a start node s,
find shortest weighted paths from s to every other node.

Examples.

Fig. 9.8 Fig. 9.9

6. Finding Unweighted Shortest Paths.

All edges cost the same.
E.g. Min Hop count routing. Quickest path to a diploma.

Strategy:
distance to s is zero.
Next, distances of all neighbors of s can be set to 1.
Inductively, if a (new) vertex v can be reached in 1 hop from a vertex whose distance is j, then v's distance is j+1.

Example.

Program:

```
enqueue(s); s.dist = 0;
while queue not empty
     v = dequeue();
     set v.known = true;

     for each w adjacent to v
         if w.dist == infinity {
             w.dist = v.dist + 1
             enqueue (w)
         }
```

Using the same analysis as topological sort, the complexity of this algorithm is O(V + E).

7. Weighted Shortest Paths. Dijkstra's Algorithm

Each vertex marked as known or unknown. Each vertex keeps a tentative distance d_v, which turns out to be the "shortest distance" from s to v "using only the known vertices" as intermediates. By keeping track of p_v (the last vertex to cause change to d_v), we can also recover the shortest paths.

The best-known method for weighted graph shortest paths is Dijkstra's, published in 1959. It's a classical greedy scheme: do what seems best at each step. (Greedy methods don't always work, so be careful and "prove correctness".)

At each stage, Dijkstra selects the "unknown" vertex v with the smallest d_v, and declares it "known". It then "updates" the values of d_w for all neighbors of v.

```
In unweighted case, we did:  d_w = d_v + 1,     if d_w = infty.

In Dijkstra's case, we do:   d_w = d_v + c(v,w) if d_w > d_v + c(v,w)
```

That is, we decide if it's good idea to go reach w through v.

8. Dijkstra's Algorithm:

```
s.dist = 0
for (;;)
    v = smallest unknown distance vertex
    if (v == not_a_vertex) break;
    v.known = true;

    for each w adjacent to v
        if ( ! w.known )
```

```
            if ( v.dist + c(v,w) < w.dist)
                { decrease (w.dist to  v.dist + c(v,w)
                  w.path = v;
                }
        }
```

9. Running time.

Depends on data structures. Naive method is O(V^2 + E).
Scan vertex list to find min each time, for total of V^2;
Weight updates happen once per edge, so O(E).

Can be improved to O((V+E) log V) by organizing the distances
in a heap.
Selection of v is a deleteMin operation--- V of them;
The update is a decreaseKey operation--- E of them.

10. Graph with negative edge weights.

Dijkstra's algorithm can fail if some edges have neg weight.
Problem: even after v is declared known, it's possible that
there is path to v from some "unknown vertex" using a VERY
negative edge that compensates for visiting other positive
weight unknown vertices.

One tempting solution is to just scale things up: add a large
constant Delta to each edge, so all edges become positive.
In the end, we can subtract those additions out.
Does not work because a cheaper path with MORE edges gets
penalized more and won't be found as shortest.

11. Minimum Spanning Trees

A communications company wants to build a network connecting N sites.
You are given a cost matrix: c(u,v) is the cost to build the link
between two sites u and v. What is the cheapest way to interconnect
all N sites?

This is the MST problem. (We assume links are bidirections, meaning
G is undirected. The problem makes sense for directed graphs too,
but more complicated.)

The cheapest interconnection graph must be a tree---if a cycle,
deleting an edge reduces cost. It connects all the nodes, which
is why it is called "spanning".

All algorithms are based on the following greedy idea. Adding
an edge to a tree creates a cycle. In the tree is an MST, then
we must throw away the costliest edge from the cycle.
Two ways to use this greedy idea: Prim and Kruskal.

12. Prim's Algorithm.

Start with a tree consisting of one node; grow it by one node in each step. The step growing the tree by the cheapest edge (u,v) such that u is in the tree, and v is not.

Example.

The algorithm is nearly identical to Dijkstra's. Each vertex is classified as known (already in the tree) or unknown (not yet), and maintains two values: d_v and p_v. The first is the minimum cost edge from v to some node in the tree,and p_v is that node in the tree.

The rest of the algorithm is the same as Dijkstra, except the update step, where we use $d_w = \min(d_w, c(v, w))$.

Running time $O(V^2)$ naively, and $O((V + E) \log V)$ with heaps.

13. Kruskal's Algorithm.
Scan edges in ascending order of cost, and accept an edge if it doesn't create a cycle with already chosen edges.

Example.

The algorithm terminates when V-1 edges accepted. How to determine whether to accept (u,v). Use Union-Find data structure.
The definition of a set is a "connected component/subtree".
Accept (u,v) iff u and v are in different sets. Then also do the union of those trees.

Kruskal's algorithm takes $O(E \log E) = O(E \log V)$ time.