```
Minimum Spanning Tree (Review from CSC263)
    Input: Connected undirected "weighted" graph G = (V,E), i.e., with
        non-negative integer weight/cost w(e) for each edge e (- E.
    Output: A spanning tree T (_ E such that cost(T) (sum of the costs of
        edges in T) is minimal.

  - Terminology:
      . "Spanning tree": acyclic connected subset of edges.
      . "Acyclic": does not contain any cycle.
      . "Connected": contains a path between any two vertices.

  - Properties:  For all spanning trees T of a graph G,
      . T contains exactly n-1 edges;
      . the removal of any edge from T disconnects it into two disjoint
        subtrees (otherwise T would contain a cycle);
      . the addition of any edge to T creates exactly one cycle (otherwise T
        would be disconnected).

 A. Brute force: consider each possible subset of edges.
    Runtime? Exponential, even if we limit search to spanning trees of G
    (instead of considering all possible subsets of edges).

 B. Prim's algorithm:
    Idea:  Start with some vertex r (- V (pick arbitrarily) and at each
        step, add lowest-cost edge that connects a new vertex to existing
        partial tree.
    Runtime? \Theta(m log n) using min-heap to implement priority queue of
        candidate edges (with priority = weight of smallest edge to tree).

 C. Kruskal's algorithm:
    Idea:  Repeatedly put in smallest-cost edge remaining, as long as it
        doesn't create a cycle.
    Runtime? \Theta(m log m) for sorting; main loop involves sequence of m
        Union and FindSet operations on n elements which is \Theta(m log n)
        -- faster using best heuristics. Total is \Theta(m log n) since
        log m is \Theta(log n).

 D. Reverse-delete algorithm:
    Idea:  Repeatedly delete largest-cost edge remaining, as long as it does
        not disconnect the graph.
    Runtime? For each edge, run BFS/DFS starting from one endpoint to figure
        out if other endpoint can still be reached. Requires \Omega(m^2).

Correctness of Kruskal's algorithm:

  - First, spell out algorithm in pseudo-code:
        Sort edges by non-decreasing weight: w(e_1) <= ... <= w(e_m).
        T = {}
        for j = 1,2,...,m:
            let (u,v) = e_j
            if T does _not_ contain a path between u and v:
                T = T u {e_j}

  - Algorithm generates subsets of edges T_0, T_1, ..., T_m.
    Say T_i is "promising" if it can be extended to some MST T* using only
    edges {e_{i+1},...,e_m}, i.e., T_i (_ T* (_ T_i u {e_{i+1},...,e_m}.
```

(Set notation again because solution is a subset of input again.)

- Loop invariant: T_i is promising.

  Base: T_0 = {} is promising: every MST T* is a subset of {e_1,...,e_m}.

  I.H.: Suppose i >= 0 and T_i can be extended to T*.

  Step: Either T_{i+1} = T_i or T_{i+1} = T_i u {e_{i+1}}.

    Case 1: If T_{i+1} = T_i, then T_i u {e_{i+1}} contains a cycle. Since T_i (_ T* and T* is a MST, e_{i+1} !(- T*, so T_{i+1} (_ T* (_ T_{i+1} u {e_{i+2},...,e_m}, i.e., T* extends T_{i+1}.

    Case 2: If T_{i+1} = T_i u {e_{i+1}}, then consider whether or not e_{i+1} (- T*.

      Subcase 2.1: If e_{i+1} (- T*, then T_{i+1} (_ T* (_ T_{i+1} u {e_{i+2},...,e_m}, i.e., T* extends T_{i+1}.

      Subcase 2.2: If e_{i+1} !(- T*, then T* does not extend T_{i+1}. Construct T** that does, as follows.
      Consider endpoints of e_{i+1} in T*: they are connected by a path. Fact: not all edges on this path belong to T_i -- else algorithm would not generate T_{i+1} = T_i u {e_{i+1}}. So T* contains some edge e_j on this path with j > i+1 (because T* agrees with T_i on all edges e_1,...,e_i and e_{i+1} is not in T*).
      Then w(e_j) >= w(e_{i+1}) and we let
      T** = T* u {e_{i+1}} - {e_j}.
      T** is a MST: it is connected, acyclic, and with total cost <= total cost of T* -- in fact, it must be that w(e_j) = w(e_{i+1}) since T* is also optimal.

    In every case, T_{i+1} is promising.

  Since every T_i is promising, T_m is promising: T_m = T* for some MST T*. So T_m is a MST.

- Correctness of other algorithms proved similarly.

Shortest Paths

Input: connected graph G = (V,E) with edge costs w(e) for all e (- E; vertices s,t (- V. IMPORTANT: costs are POSITIVE integers.
Output: a path from s to t with minimum total cost ("shortest" path).

- Brute-force: in general, exponentially many paths possible.

- Special case: if w(e) = 1 for all e: BFS!

- Dijkstra's algorithm: "modified" BFS: use _priority queue_ instead of queue to collect unvisited vertices; set priority = shortest distance so far. Note similarity to Prim's algorithm, but also important difference: for Prim's algorithm, priority = minimum cost of single edge to new vertex; for Dijkstra's algorithm, priority = minimum total distance from s to new vertex.
  Intuition: find shortest s-t path by finding shortest paths from s to every vertex.

- Algorithm:

```
# Initialization.
P = {}  # edges in shortest paths tree
initialize empty min-priority queue
for all v in V:
    pi[v] = NIL  # predecessor of v on shortest s-v path so far
    d[v] = oo    # priority of v = minimum distance s-v so far
    enqueue v    # with priority d[v] = oo
d[s] = 0
update queue order of s

# Main loop.
while queue not empty:
    v = dequeue element with minimum priority d[]
    P = P u {(pi[v],v)}  # problem when v = s: handled below
    for all edges (v,u):
        if u in the queue and d[v] + w(v,u) < d[u]:  # "relaxation"
            pi[u] = v
            d[u] = d[v] + w(v,u)
            update queue order of u
P = P - {(NIL,s)}  # clean up

return P
```

- Runtime:
  - O(n) for initialization.
  - Main loop iterates at most n times (each iteration removes one vertex from the queue).
  - Each iteration examines edges in one adjacency list and updates priorities. Over all iterations, each edge generates at most one queue update. And each priority update takes time O(log n).
  - Total: O(m log n).

- Optimal substructure in Shortest Path problem:
    A subpath of a shortest path is also a shortest path.

  Proof: Cut and paste technique! Assume that we have a shortest path (P) between two vertices u and v and consider two vertices x and y on this path. If the path between x and y along P is not a shortest path between x and y, we can simply replace this subpath by the shortest between x and y (and we are sure that there is one, why?). Replacing this subpath with another shorter path will make the shortest path between u and v even smaller which is in contradiction with the fact that path p is a shortest path! The reasoning applies even if the shortest x-y path contains nodes that appear elsewhere on P -- then, the contradiction is derived from using only the non-overlapping parts of the path.

- Triangle Inequality:
    for all vertices u, v, x in V we have
    delta(u,v) <= delta(u,x) + delta(x,v).

- Lemma:
    During Dijkstra's algorithm, d[v] >= delta(s,v) and this holds after taking any sequence of relaxations.

  Proof idea: by induction. The inequality holds after the initialization step (every vertex except s has d[v] = infinity and d[s] = 0). And each relaxation preserves the property (because d[v] is either oo or equal to the total weight of some path in G).

- Correctness (main idea):

  Show d[v] = delta(s,v) for all v in P_i, where delta(x,y) is minimum total weight of any x-y path.

  Consider one iteration of main loop: v with minimum d[v] removed from queue, P_{i+1} = P_i u {(pi[v],v)}.

  To show: d[v] = delta(s,v). For contradiction, suppose d[v] > delta(s,v) and let P = s-v path with total weight delta(s,v). Let (x,y) be first edge on P with x in P_i, y outside P_i. Either y = v or y != v.

```
Case 1: y = v.
    Then d[v] <= d[x] + w(x,v) [(x,v) one possible edge from P_i to v]
               = delta(s,x) + w(x,v) [by I.H. since x in P_i]
               = delta(s,v)
              <  d[v],
    contradiction: d[v] < d[v]!
Case 2: y != v.
    Then d[y] <= d[x] + w(x,y) [(x,y) one possible edge from P_i to y]
               = delta(s,x) + w(x,y) [by I.H. since x in P_i]
              <  delta(s,x) + w(x,y) + delta(y,v) [positive weights]
               = delta(s,v)
              <  d[v],
    contradiction: d[y] < d[v] but d[v] = minimum outside P_i.
So d[v] = delta(s,v).
```

  Rest of proof involves induction structure around main idea. Included below for reference but not covered during lecture.

- Correctness (details):

  Algorithm generates subsets of edges P_0, P_1, ..., P_{n-1}.
  Say P_k is "promising" if it can be "extended" to some collection of shortest paths P* (really, a shortest paths tree) using only edges that do not have *both* endpoints "in" P_k, i.e., edges with at least one endpoint still in the queue. (Technically, P_k contains edges, not vertices: when we speak of a vertex being "in" P_k, we mean that it is the endpoint of some edge in P_k.)

```
Loop invariant:
  - P_k is promising, and
  - for all u in P_k, all v outside P_k,
    d[u] = delta(s,u) <= delta(s,v) <= d[v]
where delta(s,v) is the minimum total weight of all paths from s to v.
(Extra clause is required for the proof to go through.)
```

  Proven by induction on k.

  Base Case:
      P_0 = {} is promising, trivially.

  Ind. Hyp.:
      For some arbitrary k, suppose P_k can be extended to some shortest paths tree P*, using only edges without both endpoints in P_k, and that d[u] = delta(s,u) <= delta(s,v) <= d[v] for all u in P_k and v outside P_k.

  Ind. Step:
      Consider P_{k+1} = P_k u {(u,v)}, with u in P_k and v outside P_k.

```
Either (u,v) (- P* or it does not.

If (u,v) (- P*, then P* extends P_{k+1}.
Also, delta(s,v) = delta(s,u) + w(u,v) (because (u,v) in P*) so d[v]
= d[u] + w(u,v) = delta(s,u) + w(u,v) = delta(s,v).
Moreover, because d[v] was the smallest of the d[] values for
vertices outside P_k, d[x] = delta(s,x) <= delta(s,w) <= d[w] for
all x in P_{k+1} and all w outside P_{k+1}.

If (u,v) !(- P*, then:
  - consider the path P in P* from s to v, and let (w,v) be the last
    edge on this path;
  - w must belong to P_k -- otherwise, let (x,y) be the first edge
    on P with x in P_k, y outside P_k
      . d[y] <= d[x] + w(x,y) (because d[y] is the smallest value of
                   d[t] + w(t,y) for all edges (t,y) with t in P_k))
              = delta(s,x) + w(x,y) (because x in P_k)
             <  delta(s,x) + w(x,y) + delta(y,v)
                   (since all edge weights strictly positive)
              = delta(s,v)
             <= d[v]
      . but this would contradict the fact that d[v] is the smallest
        value of d[t] for all vertices t outside P_k
    so w in P_k;
  - so delta(s,v) = delta(s,w) + w(w,v) = d[w] + w(w,v);
  - since d[v] is the minimum of d[x] + w(x,v) over vertices x, this
    means d[v] <= d[w] + w(w,v) = delta(s,v);
  - so d[v] = delta(s,v) (it cannot be smaller);
  - so we can let P** = P* - {(w,v)} u {(u,v)}, and after the update
    to d[w] for all edges (v,w) with w outside P_k, we still have
    that d[x] = delta(s,x) <= delta(s,y) <= d[y] for all x in P_k,
    y outside P_k.
```

Hence, the loop invariant holds. When the algorithm terminates, this means d[u] = delta(s,u) for all vertices u: we have found shortest paths to every vertex.

For Next Week

* Readings: Section 6.5.
* Self-Test: Trace matrix chain multiplication algo. on an example.