CSCE 411
Shortest Paths and MSTs

---

1. Suppose that we are given a graph G and want to compute a *maximum* spanning tree–i.e. a spanning tree T which maximizes $\sum_{e \in T} w(e)$.

   (a) Suppose that we modify the Prim–Jarník algorithm so that at each step it adds the *heaviest* edge connecting its current subgraph to the rest of G. Does this algorithm solve the maximum spanning tree problem? Justify your answer.

   (b) Suppose we modify Kruskal's algorithm so that it sorts the edges from heaviest to lightest and then repeatedly adds the first edge which does not create a cycle. Does this algorithm solve the maximum spanning tree problem?

   Solution: Yes, both of these algorithms work. We could prove this by showing a "heavy edge property" for maximum spanning trees analogous to the light edge property for minimum spanning trees and then following the correctness proofs from the course notes.

   Alternatively, we can directly apply the known correctness of Prim–Jarník and Kruskal's algorithms via the following trick:

   Let G' be a copy of G with all the edge weights negated (i.e. the weight of e in G' is $-w(e)$). Then for any spanning tree T in G of weight w_T, the weight of T in G' is $-w_T$. This implies that every maximum spanning tree in G is a minimum spanning tree in G' (since maximizing w_T is the same as minimizing $-w_T$) and vice-versa. Furthermore, we note that when Prim–Jarník or Kruskal's algorithm selects a maximum weight edge satisfying a criterion (e.g. maximum weight edge that does not create a cycle), this will be a minimum weight edge satisfying the same criterion in G'. Thus, running our modified version of Prim–Jarník or Kruskal's on G is equivalent to running regular Prim–Jarník or Kruskal's on G'. Since we know that both of these algorithms will find a minimum spanning tree in G', we conclude that the maximizing versions will both find a maximum spanning tree in G.

2. The hit new role playing game Adventures in Algorithmica features n different zones $z_1, \ldots, z_n$. Each zone z_i has a minimum required level to enter, ℓ_i, which is an integer with $1 \leq \ell_i \leq n$. Fortunately, gaining experience in this game is very easy. You gain 1 level every time you enter a zone. Unfortunately, zones take a long time to load. Curiously, the time it takes a zone to load depends on which other zone you are entering it from. You are given an array A of loading times, where $A[i, j]$ is the time that zone j takes to load if you enter it from zone i. You are writing a guide for the game and want to figure out the fastest routes to all of the high level zones.

   Give an $O(n^4)$ algorithm which takes as input, the list of zones, list of level requirements, an array of loading times, and outputs a 2D array T where $T[i, j]$ is the minimum time to get to z_j, assuming that you start in z_i at level ℓ_i.

Solution: Let $Z = \{z_1, \ldots, z_n\}$ be the set of zones in the game. We construct a layered directed weighted graph $G = (V, E)$ to represent this problem as follows:

- $V := Z \times \{1, \ldots, n\}$
- E contains the following edges: For each distinct i, j between 1 and n, E contains an edge $((z_i, n), (z_j, n))$ with weight $A[i, j]$. Additionally, for each integer k with $\ell_j \leq k < n$, E contains an edge $((z_i, k), (z_j, k + 1))$ also with weight $A[i, j]$.

To find the minimum time to reach all other zones from zone z_i starting at level ℓ_i, we run Dijkstra's algorithm starting from the vertex (z_i, ℓ_i). Using the returned distance array D, we set $T[i, j]$ equal to $\min_{1 \leq k \leq n} D[(z_j, k)]$ (i.e. since we don't care what level we are when we reach z_j we take the minimum over all possibilities).

Correctness: Our layered graph is designed so that the second part of each vertex represents your current level. Our edges guarantee that every time you enter a zone your level increases (because every edge goes to the next layer) unless you are already at level n, in which case further level gains don't affect what zones you can access, so the graph does not track them. Edges to z_i only start appearing at layer ℓ_i, so every path in G satisfies the minimum level requirements.

Runtime: Our graph has $O(n^2)$ vertices (n layers of n vertices each) and $O(n^3)$ edges (n^2 edges for each of n layers). Each vertex and edge can be constructed in constant time so we spend $O(n^3)$ time constructing G. Since the time it takes for a zone to load cannot be negative, all of our edges have non-negative weight, so Dijkstra's algorithm on G runs in time $O(n^3 + n^2 \log n) = O(n^3)$. Since we run Dijkstra's once for each zone to get our final output, the overall runtime is $O(n \cdot n^3) = O(n^4)$. Note that this is substantially faster than running Floyd-Warshall to compute all-pairs shortest paths in this graph (which would have taken time $O((n^2)^3) = O(n^6)$).

3. Consider the following algorithm:

```
EdgeRemover(G)
  A ← an array of G's edges, sorted in order of decreasing weight
  T ← G
  For i from 1 to m:
      If removing A[i] from T would not disconnect T:
          Remove A[i] from T
  Return T
```

Prove that for every connected graph G with distinct edge weights, EdgeRemover(G) returns a minimum spanning tree for G.

Solution: Let n be an arbitrary positive integer. We will prove by induction on m that EdgeRemover$G$ returns an MST for every connected graph G with n vertices, m edges, and distinct edge weights.

- Base Case: Suppose $m = n - 1$ (the minimum possible number of edges for a connected graph). Then G is already a tree, so removing any edge would disconnect T. EdgeRemover(G) returns G, which is the minimum spanning tree (in fact the only spanning tree) for itself.
- Induction Hypothesis: EdgeRemover correctly computes an MST for every connected graph on n vertices with distinct edge weights and fewer than m edges.
- Induction Step: Let $m > n - 1$. Then G must contain a cycle (since every connected acyclic graph is a tree). Removing any edge from this cycle would not disconnect G. Therefore the loop in EdgeRemover(G) will remove at least one edge. Let $A[j] = (u, v)$ be the first edge removed from T. Since the edges in A were sorted in order of decreasing weight, this means that $A[j]$ is the heaviest edge whose removal would not disconnect T. Note that since removing $A[j]$ would not disconnect G, there must be a $u \to v$ path P in G which does not use $A[j]$.

  We claim that $A[j]$ is not part of the MST for G. Let T^* be the MST for G. AFSOC that $T*$ includes $A[j]$. Since T^* is a tree, removing $A[j]$ from T^* splits T^* into two connected components. Let S be the connected component of $T^* - A[j]$ containing u. Call the vertices of P $v_1, v_2, \ldots, v_k$, where $v_1 = u$ and $v_k = v$. Then, there must be at least one place where the path $v_1, v_2, \ldots, v_k$ transitions from S to $V \setminus S$ (since it starts in S and ends in $V \setminus S$). Let $1 \leq i < k$ be an integer such that $v_i \in S$ and $v_{i+1} \notin S$. Call the edge $\{v_i, v_{i+1}\}$ e. Since e's endpoints are disconnected in $T^* - A[j]$, $e \notin E(T^*)$.

  We note that e cannot be an edge whose removal disconnects G, because T^* does not include e and T^* is connected. Therefore, we must have $w(e) < w(A[j])$ (because EdgeRemover removed $A[j]$ first). Now consider the graph $T' := T^* - A[j] + e$. We note that by our choice of e, e connects the two components, S and $V \setminus S$, of $T^* - A[j]$, so T' is connected graph with $n - 1$ edges (since we added and removed one edge from T^*), so T' is a spanning tree. Furthermore, $w(T') = w(T^*) - w(A[j]) + w(e) < w(T^*)$, contradicting the fact that T^* was an MST. Thus, $A[j]$ is not in the MST.

  Since $A[j]$ is not part of the MST, the MST for G is the same as the MST for $G - A[j]$. Since $A[j]$ was the first edge removed, the remaining edges the algorithm removes will be the same ones it would've removed if we had run it on $G - A[j]$ from the start. By our IH, EdgeRemover finds the MST for $G - A[j]$, so therefore it finds the MST for G as well.

4. Suppose that we are given a list of n variables $x_1, \ldots, x_n$ and a list of m constraints of the form: $x_i - x_j \leq c_k$, where $c_1, \ldots, c_m$ are integer constants. Give an $O(mn)$ time algorithm to determine whether or not there exist integer values for the variables which satisfy all of the constraints.

Solution: We assume that each variable appears in at least one constraint. If this is not the case, we can begin by discarding any variable which appears in no constraints, since these variables will not affect whether the constraints are satisfiable. Thus, we can assume that $m \geq n/2$.

We create a directed weighted graph $G = (V, E)$ to represent this problem as follows:

- Our vertex set is the variables plus a new source vertex s: $V = \{x_1, \ldots, x_n\} \cup \{s\}$
- Our edge set E includes the following kinds of edges:
  - For each constraint $x_i - x_j \leq c_k$, we have an edge (x_j, x_i) of weight c_k (If there are multiple constraints with the same ordered pair of variables (e.g. $a - b < 2$ and $a - b < 3$) then we create a single edge whose weight is the smallest of the c_k values for this ordered pair).
  - For each variable x_i, we have an edge (s, x_i) of weight 0.

We run Bellman-Ford on this graph, starting from s, to determine whether there is a negative-weight cycle reachable from s. If so, we report that there is no assignment of integer values to the variables which satisfies all of the constraints. If there is no negative cycle, we report that it is possible to satisfy all of the constraints.

Correctness: We need to show two directions: that if there is a negative weight cycle then there is no satisfying assignment and that if there is no negative-weight cycle then there is a satisfying assignment (we note that because s has edges to all other vertices, any negative-weight cycle in G would be reachable from s).

- Suppose that G contains a negative-weight cycle. Without loss of generality, assume that the variables in the negative-weight cycle are $x_1, \ldots, x_k$ for some $k \leq n$, and that the constraints inducing the edges of the cycle take the form $x_i - x_{i+1} \leq c_i$ for each $1 \leq i < k$, plus the constraint $x_k - x_1 \leq c_k$. Since the cycle has negative weight, we have that $\sum_{j=1}^{k} c_j < 0$. Note that if we add all k of these constraints together, the left sides all cancel out (each variable appears once as a positive value and once as a negative value), and so we obtain $0 \leq \sum_{j=1}^{k} c_j < 0$, which is impossible, so there is no way to satisfy all of these constraints.
- Suppose that there is no negative-weight cycle in G. Then Bellman-Ford computes the length of a shortest walk from s to each of the other vertices. We claim that if we assign each variable x_i the value $dist(s, x_i)$, all of the constraints will be satisfied. Note that:

  $$x_i - x_j \leq c_k \Leftrightarrow x_i \leq x_j + c_k.$$

  Since our graph contains an edge (x_j, x_i) of weight c_k, the distances to these vertices must satisfy $dist(s, x_i) \leq dist(s, x_j) + c_k$, since we could reach x_i by following the path to x_j and then following the (x_j, x_i) edge.

Runtime: Our graph has $O(n)$ vertices and $O(m + n)$ edges. Each vertex and edge can be constructed in constant time, so creating the graph takes $O(m+n)$ time. Running Bellman-Ford on this graph takes $O((m + n)n)$ time, which is $O(mn)$ (since we discarded unused variables at the start to guarantee $m \geq n/2$) so our overall runtime is $O(mn)$.