Fundamental Algorithms
CSCI-GA.1170-001/Summer 2016

Solution to Homework 8

Problem 1 (CLRS 23.1-6). (1 point) Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

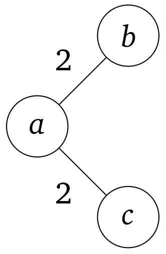
Solution:

Let us show that any two minimum spanning trees T_1 and T_2 of G are the same tree. Let (u, v) be an arbitrary edge of T_1. By Exercise 23.1-3, any edge in an MST is a light edge crossing some cut of the graph. Let $(S, V - S)$ be a cut for which (u, v) is a light edge.

Consider the edge $(x, y) \in T_2$ crossing $(S, V - S)$. (x, y) must exist, as otherwise T_2 would not be a *spanning* tree. (x, y) must also be a light edge, as otherwise T_2 would not be a *minimum* spanning tree.

By problem statement, there is an unique light edge crossing any cut of G. Thus, $(u, v) \in T_1$ and $(x, y) \in T_2$ must be the same edge. As (u, v) is an arbitrary edge of T_1, every edge in T_1 is also in T_2 and thus T_1 and T_2 are the same tree.

The converse – if a graph has a unique MST, then light edges for all cuts are unique – is not true, as demonstrated by a counterexample:



Here, the graph is its own (unique) MST, but the cut $(\{a\}, \{b, c\})$ has two light edges – (a, b) and (a, c).

Problem 2 (CLRS 23-1). (3 points) Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

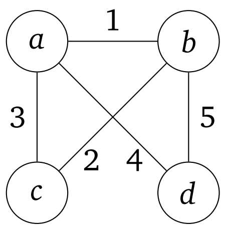
We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a *second-best minimum spanning tree* is a spanning tree T such that $w(T) = min_{T'' \in \mathcal{T} - \{T'\}}\{w(T'')\}$.

Solution:

(a) Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

In a connected graph with distinct edge weights, there is a unique light edge for any cut of the graph. By Problem 1, the graph has a unique MST.

We show that the second-best MST need not be unique with the following example:



Here, the MST of weight 7 is $\{(a,b),(a,d),(b,c)\}$, and there are two second-best MSTs: $\{(a,b),(a,c),(a,d)\}$ and $\{(a,b),(b,c),(b,d)\}$ (both of weight 8).

(b) Let T be the minimum spanning tree of G. Prove that G contains edges $(u,v) \in T$ and $(x,y) \notin T$ such that $T - \{(u,v)\} \cup \{(x,y)\}$ is a second-best minimum spanning tree of G.

Since any spanning tree has exactly $|V| - 1$ edges, any second-best minimum spanning tree must have at least one edge that is not in the (best) minimum spanning tree. If a second-best minimum spanning tree has exactly one edge, say (x,y), that is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree, except that (x,y) replaces some edge, say (u,v), of the minimum spanning tree. In this case, $T' = T - \{(u,v)\} \cup \{(x,y)\}$, as we wished to show.

Thus, all we need to show is that by replacing two or more edges of the minimum spanning tree, we cannot obtain a second-best minimum spanning tree. Let T be the minimum spanning tree of G, and suppose that there exists a second-best minimum spanning tree T' that differs from T by two or more edges. There are at least two edges in $T - T'$ and let (u,v) be the edge in $T - T'$ with minimum weight. If we were to add (u,v) to T', we would get a cycle c. This cycle contains some edge (x,y) in $T' - T$ (since otherwise, T would contain a cycle).

We claim that $w(x,y) > w(u,v)$. We prove this claim by contradiction, so let us assume that $w(x,y) < w(u,v)$. (Recall the assumption that edge weights are distinct, so that we do not have to concern ourselves with $w(x,y) = w(u,v)$.) If we add (x,y) to T, we get a cycle c', which contains some edge (u',v') in $T - T'$ (since otherwise, T' would contain a cycle). Therefore, the set of edges $T'' = T - \{(u',v')\} \cup \{(x,y)\}$ forms a spanning tree, and we must also have $w(u',v') < w(x,y)$, since otherwise T'' would be a spanning tree with weight less than $w(T)$. Thus, $w(u',v') < w(x,y) < w(u,v)$, which contradicts our choice of (u,v) as the edge in $T - T'$ of minimum weight.

Since the edges (u,v) and (x,y) would be on a common cycle c if we were to add (u,v) to T', the set of edges $T' - \{(x,y)\} \cup \{(u,v)\}$ is a spanning tree, and its weight is less than $w(T')$. Moreover, it differs from T (because it differs from T' by only one edge). Thus, we have formed a spanning tree whose weight is less than $w(T')$ but is not T. Hence, T' was not a second-best minimum spanning tree.

(c) Let T be a spanning tree of G and, for any two vertices $u, v \in V$ let $max[u,v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

Being allowed quadratic time, we can simply run one of the graph search algorithms from each vertex, and maintain the maximum edge weight seen so far. COMPUTE-MAX below is a modification of breadth-first search, using table max itself to keep track of visited vertices instead of coloring: $max[u, v] = \text{NIL}$ if and only if $u = v$ or we have not yet visited vertex v in a search from vertex u.

```
COMPUTE-MAX($T, w$)
for each vertex $u \in T.V$
    for each vertex $v \in T.V$
        $max[u, v] = \text{NIL}$
    $Q = \emptyset$
    ENQUEUE($Q, u$)
    while $Q \neq \emptyset$
        $x = $ DEQUEUE($Q$)
        for each vertex $v \in T.Adj[x]$
            if $max[u, v] = \text{NIL}$ and $u \neq v$
                if $x = u$ or $w(x, v) > max[u, x]$
                    $max[u, v] = (x, v)$
                else
                    $max[u, v] = max[u, x]$
                ENQUEUE($Q, v$)
return $max$
```

The algorithm runs a BFS from each of $|V|$ vertices, and thus takes $O(V(V + E))$ time. Observing that a spanning tree has exactly $|V| - 1$ edges, we can restate the running time of COMPUTE-MAX as $O(V^2)$.

(d) Give an efficient algorithm to compute the second-best minimum spanning tree of G.

By (b), we can obtain a second-best minimum spanning tree by replacing exactly one edge of the minimum spanning tree T by some edge $(u, v) \notin T$. If we create spanning tree T' by replacing edge $(x, y) \in T$ with edge $(u, v) \notin T$, then $w(T') = w(T) - w(x, y) + w(u, v)$.

For a given edge (u, v), the edge $(x, y) \in T$ that minimizes $w(T')$ is the edge of maximum weight on the unique path between u and v in T. By (c), that edge is $max[u, v]$.

Thus, our algorithm needs to determine an edge $(u, v) \notin T$ for which $w(max[u, v]) - w(u, v)$ is minimum:

```
SECOND-BEST-MST($G, w$)
 1  $T = \text{MST-PRIM}(G, w)$
 2  $max = \text{COMPUTE-MAX}(T, w)$
 3  $minEdge = \text{NIL}$
 4  $minValue = \infty$
 5  for each edge $(u, v) \in G.E$ and $(u, v) \notin T.E$
 6      $value = w(max[u, v]) - w(u, v)$
 7      if $value < minValue$
 8          $minEdge = (u, v)$
 9          $minValue = value$
10  return $T - \{max[minEdge]\} \cup \{minEdge\}$
```

MST-PRIM takes $O(E \lg V)$ time, COMPUTE-MAX – $O(V^2)$, lines 5-9 – $O(E)$. For a dense graph with $|E|$ close to $|V|^2$, SECOND-BEST-MST takes $O(V^2 \lg V)$ time.

As described in section 23.2 of CLRS, the running time of MST-PRIM can be improved to $O(E + V \lg V)$ by using Fibonacci heaps. This brings the running time of SECOND-BEST-MST down to $O(V^2)$.

Problem 3 (CLRS 24.3-6). (2 points) We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \le r(u, v) \le 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Solution: Observing that the probabilities are independent, we can view the reliability of a path as the product of the reliabilities of its edges, and thus reduce the original problem to a variation of the shortest-paths problem.

All weights $w(u, v) = r(u, v)$ are non-negative, allowing us to use Dijkstra's algorithm.

We modify the algorithm to maximize the product of weights instead of minimizing the sum of weights. Note that the proof of correctness for Dijkstra's algorithm still holds, because multiplying by $0 \le r(u, v) \le 1$ cannot make the product larger.

This requires the following changes:

- Initialization procedure needs to set initial keys to $-\infty$ instead of ∞:
- Relaxation procedure now maximizes the product of weights:

```
RELAX($u, v, w$)
1  if $v.d < u.d \cdot w(u, v)$
2      $v.d = u.d \cdot w(u, v)$
3      $v.\pi = u$
```

- Initial keys are set to $-\infty$ instead of ∞:

```
INIT($G, s$)
    for each vertex $v \in G.V$
        $v.d = -\infty$
        $v.\pi =$ NIL
    $s.d = 0$
```

- Given updated INIT and RELAX, the main procedure is largely unchanged, but uses EXTRACT-MAX instead of EXTRACT-MIN:

```
DIJKSTRA($G, w, s$)
    INIT($G, s$)
    $S = \emptyset$
    $Q = G.V$
    while $Q \neq \emptyset$
        $u =$ EXTRACT-MAX($Q$)
        $S = S \cup \{u\}$
        for each vertex $v \in G.Adj[u]$
            RELAX($u, v, w$)
```

Problem 4 (CLRS 24-2). (2 points) A d-dimensional box with dimensions $(x_1, x_2, ..., x_d)$ *nests* within another box with dimensions $(y_1, y_2, ..., y_d)$ if there exists a permutation π on $\{1, 2, ..., d\}$ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, ..., x_{\pi(d)} < y_d$.

Solution:

(a) Argue that the nesting relation is transitive.

Consider boxes:

$$\begin{aligned} x &= (x_1, ..., x_d), \\ y &= (y_1, ..., y_d), \\ z &= (z_1, ..., z_d). \end{aligned}$$

If x nests inside y and y nests inside z, there exist permutations π_1 and π_2 such that:

$$\begin{aligned} x_{\pi_1(i)} &< y_i \quad \text{for} \quad i = 1, ..., d, \\ y_{\pi_2(i)} &< z_i \quad \text{for} \quad i = 1, ..., d. \end{aligned}$$

Then, for permutation $\pi_3(i) = \pi_2(\pi_1(i))$:

$$x_{\pi_3(i)} < y_{\pi_2(i)} < z_i \quad \text{for} \quad i = 1, ..., d.$$

(b) Describe an efficient method to determine whether or not one d-dimensional box nests inside another.

Box $x = (x_1, ..., x_d)$ nests inside box $y = (y_1, ..., y_d)$ if and only if $x'_i < y'_i$ for $i = 1, ..., d$, where x' and y' are sorted dimensions of x and y respectively. (Straightforward to show in both directions by induction on the number of dimensions.)

This allows us to write the following $O(d \lg d)$ algorithm:

```
NESTS(x, y, d)
1  SORT(x)
2  SORT(y)
3  for i = 1 to d
4      if $x_i \geq y_i$
5          return FALSE
6  return TRUE
```

(c) Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Give an efficient algorithm to find the longest sequence $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k-1$. Express the running time of your algorithm in terms of n and d.

We construct a weighted directed graph $G = (V, E)$, where $(v_i, v_j) \in E$ if and only if box B_i nests inside box B_j. The problem is equivalent to finding the longest path in G.

To work around the case where the longest path is not reachable from the source vertex, we extend G with a supersource vertex s and zero-weight edges (s, v_i) for all vertices $v_i \in V$ with in-degree 0.

To avoid examining all computed distances once the algorithm completes, we extend G with a supersink vertex t and zero-weight edges (v_j, t) for all vertices $v_j \in V$ with out-degree 0. Thus, the longest path between s and t will be longest path in G.

We observe that G is a DAG. Formally, relation *nests inside* is irreflexive (a box does not nest inside itself), transitive (shown in (a)) and asymmetric (if X nests inside Y, then Y does not nest inside X) and thus is a strict partial order, and every strict partial order is a DAG.

G being a DAG allows us to use DAG-SHORTEST-PATHS from section 24.2 in CLRS. To find the *longest* path, we assign every edge in G (other than those incident to s and t) the weight of -1. This way, DAG-SHORTEST-PATHS can be used without modifications.

We reason about the running time of the algorithm as follows:

- Sorting the dimensions of each of the n boxes, taking $O(nd \lg d)$ time.
- Comparing each of the $\binom{n}{2}$ pairs of boxes for nestedness, taking $O(n^2 d)$ time.
- DAG-SHORTEST-PATHS takes $\Theta(V + E)$ time. G has one vertex for each box, plus two additional vertices s and t, for a total of $|V| = n + 2$. G has one edge for each of the $\binom{n}{2}$ pairs of boxes, plus additional edges incident to s and t, for a total of $|E| = O(n^2)$. Thus, DAG-SHORTEST-PATHS takes $O(n^2)$ time.
- Adding up the times above gives the overall running time of $O(nd \lg d + n^2 d)$ or $O(nd(n + \lg d))$.

Problem 5 (CLRS 24-3). (3 points) *Arbitrage* is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \times 2 \times 0.0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent.

Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i,j]$ units of currency c_j.

Solution:

(a) Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

$R[i_1,i_2] \cdot R[i_2,i_3] \cdot \ldots \cdot R[i_{k-1},i_k] \cdot R[i_k,i_1] > 1.$

Analyze the running time of your algorithm.

We construct a weighted directed graph $G = (V, E)$ with one vertex for each currency and edges (v_i, v_j) and (v_j, v_i) for each pair of currencies.

We now transform the product of $R[i,j]$ in the problem statement into a sum:

$$\begin{gathered} R[i_1,i_2] \cdot R[i_2,i_3] \cdot \ldots \cdot R[i_{k-1},i_k] \cdot R[i_k,i_1] > 1 \\ \frac{1}{R[i_1,i_2]} \cdot \frac{1}{R[i_2,i_3]} \cdot \ldots \cdot \frac{1}{R[i_{k-1},i_k]} \cdot \frac{1}{R[i_k,i_1]} < 1 \\ \lg \frac{1}{R[i_1,i_2]} + \lg \frac{1}{R[i_2,i_3]} + \ldots + \lg \frac{1}{R[i_{k-1},i_k]} + \lg \frac{1}{R[i_k,i_1]} < 0. \end{gathered}$$

Thus, if we define the weights as:

$$w(v_i, v_j) = \lg \frac{1}{R[i,j]} = -\lg R[i,j],$$

the problem is reduced to finding a negative-weight cycle in G.

To work around the case where a negative-weight cycle is not reachable from the source vertex, we extend G with a supersource vertex s and zero-weight edges (s, v_i) for all vertices $v_i \in V$. Unlike in Problem 4, it is not enough to connect s to vertices with 0 in-degree only, as G might have cycles, in which any vertex is reachable from any other vertex in the cycle (and thus has a non-zero in-degree), but not from outside the cycle.

We now recall that BELLMAN-FORD can detect the presence of negative-weight cycles and returns FALSE if one is found. Thus, we can write the following algorithm to determine whether or not there exists a sequence of currencies presenting an arbitrage opportunity:

```
ARBITRAGE-POSSIBLE($R$)
1  $G, w, s$ = CONSTRUCT-GRAPH($R$)
2  if BELLMAN-FORD(G, w, s) = FALSE
3      return TRUE
4  return FALSE
```

Here, CONSTRUCT-GRAPH needs to examine each of the $\binom{n}{2}$ pairs of currencies and thus takes $O(n^2)$ time. G has one vertex for each currency and one edge for each pair of currencies, so $|V| = n$ and $|E| = O(n^2)$. We know that BELLMAN-FORD takes $O(VE)$ time. Thus, the overall running time of ARBITRAGE-POSSIBLE is $O(n^3)$.

(b) Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

BELLMAN-FORD returns FALSE when for some edge (u, v), $v.d > u.d + w(u, v)$, indicating a negative-weight cycle that includes u. Thus, we can follow the predecessor pointers from u until we encounter u again; one way to do that is procedure PRINT-PATH in section 22.2 of CLRS. The following algorithm prints out a sequence of currencies presenting an arbitrage opportunity if one exists:

```
PRINT-ARBITRAGE($R$)
1  $G, w, s$ = CONSTRUCT-GRAPH($R$)
2  if BELLMAN-FORD(G, W, S) = TRUE
3      print "No arbitrage possible"
4      return
5  for each edge $(u, v) \in G.E$
6      if $v.d > u.d + w(u, v)$
7          PRINT-PATH($G, u, u.\pi$)
8          return
```

Lines 5-8 take $O(n^2)$ time (linear in the number of edges), and the overall running time is dominated by running time of BELLMAN-FORD, which we already shown to be $O(n^3)$.