Assignment 8

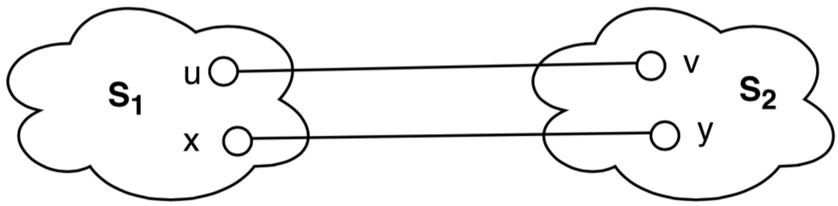
Benjamin Jakubowski

July 20, 2016

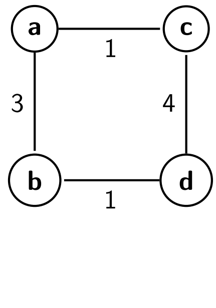
1. Show G has a unique MST

Let G be a graph where for every cut there is a unique light edge crossing the cut. We show the graph has a unique minimum spanning trees (MSTs).

Let T_1 and T_2 be two MSTs of G. Now if $T_1 \neq T_2$, there must be some $u \in V$ such that $(u, v) \in T_1$ but $(u, v) \notin T_2$. Note removing the edge (u, v) from T_1 produces two disconnected subtrees S_1 and S_2 and defines a cut of G, namely $(S_1.V, V - S_1.V = S_2.V)$. Now, since T_2 is a tree and thus connected, there must be vertices x and y in S_1 and S_2, respectively, such that $(x, y) \in T_2$.



Now, by CLRT 23.1-3, (u, v) and (x, y) must both be light edges since they are edges in MSTs of G. But then, since all light edges are unique, $(u, v) = (x, y)$. Hence $T_1 = T_2$, so G has a single unique MST.
Now for the counter example to disprove the converse: the graph shown below has a unique MST but does not have a unique light edge crossing each cut.

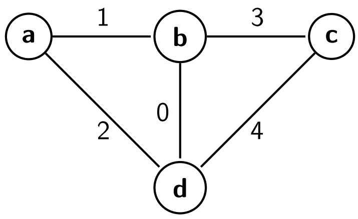


2. Second-best MSTs

Now let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, with $|E| \geq |V|$ and all edge weights distinct.

a. MST is unique but second-best MST not necessarily unique

Since all edge weights are distinct, for every cut of the graph G there is a unique light edge. Thus by problem 1, G has a unique MST.
We show the second-best MST is not necessarily unique by providing an example:



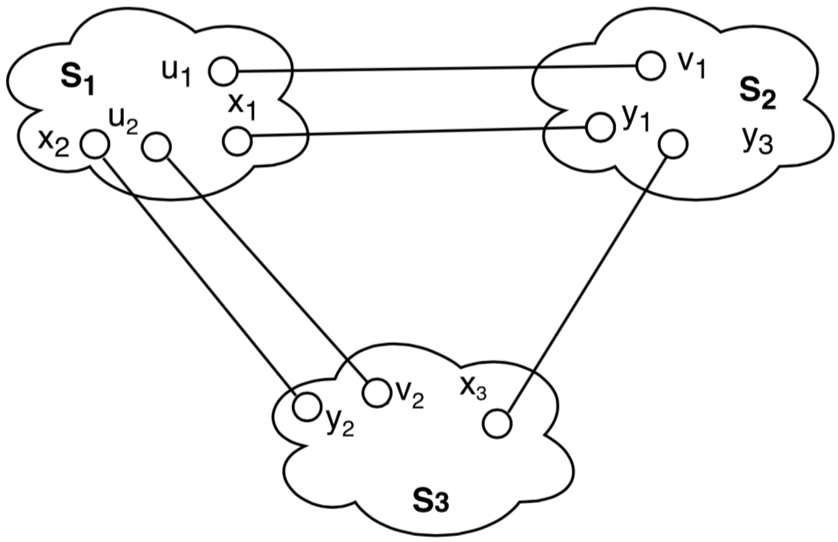
b. Constructing a second-best MST

We now show that G contains $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best MST of G.

First note we must replace at least one edge in T to obtain a second-best MST T''. If we didn't replace any edges, we'd have T, which is still our MST. Thus assume we replace at least one edge (noting this statement allows for replacement of all edges in T).

Now assume we can obtain a second-best MST T'' by replacing two or more edges. We show this yields a contradiction, and thus we must only replace a single edge to obtain our second-best MST.

Since we replace at least two edges, say we remove $(u_1, v_1), (u_2, v_2)$ from T. This produces three disconnected subtrees S_1, S_2, and S_3. Since we are constructing our second-best MST T'', we must connect these components- thus we need to replace $(u_1, v_1), (u_2, v_2)$ with two of $(x_1, y_1), (x_2, y_2)$, and (x_3, y_3).



In particular note that we must have either

- Cut (u_1, v_1) and pasted (x_1, y_1). In this case, note since T was our MST, $w(u_1, v_1) < w(x_1, y_1)$.
- Cut (u_2, v_2) and pasted (x_2, y_2). In this case, similarly note since T was our MST, $w(u_2, v_2) < w(x_2, y_2)$.

Without loss of generality, let's say we cut (u_1, v_1) and pasted (x_1, y_1) (the second case is identical, so we address just the first case to ease notation). Then note we can construct a new spanning tree T''' by undoing this cut-and-paste (replacing (x_1, y_1) in T'' with (u_1, v_1)). Now, since $w(u_1, v_1) < w(x_1, y_1)$, $w(T''') < w(T'')$. However, $T''' \neq T$. Thus we have

$$w(T) < w(T''') < w(T'')$$

But then we have our contradiction, namely that T'' is not a second-best MST. Instead T''' is our second-best MST. Thus we only replace a single edge from T to construct our second-best MST.

C. $O(V^2)$ ALGORITHM TO COMPUTE $\max[u, v]$

Now 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. We present an $O(V^2)$-time algorithm that, given T, computes $\max[u, v]$ for all $u, v \in V$.

```
function FIND_MAX(T)
    for v ∈ G.V do
        modified_BFS(T, v)
```

The bulk of the work is clearly in `modified_BFS`(T, v). This helper function runs a breadth first search from v, with the modification being the computation and storage of the maximum weight in the path from s to every other vertex in the tree.

```
1: function MODIFIED_BFS(T, s)
2:     for v ∈ G.V − {s} do
3:         v.color = white
4:         v.d = ∞
5:         v.π = NIL
6:         v.max[s] = −∞                ▷ v.max[s] stores max weight in path from v to s
7:     s.color = Gray
8:     s.d = 0
9:     s.π = NIL
10:    s.max[s] = −∞
11:    Q = new Queue
12:    Enqueue(Q, s)
13:    while Q is not empty do
14:        u = Dequeue(Q)
15:        for each v ∈ G.Adj[u] do
16:            if v.color == white then
17:                v.color = gray
18:                v.d = u.d + 1
19:                v.π = u
20:                if w(u,v) > u.max[s] then
21:                    v.max[s] = w(u,v)
22:                    v.max_edge[s] = (u,v)
23:                else
24:                    v.max[s] = u.max[s]
25:                    v.max_edge[s] = u.max_edge[s]
26:                Eunqueue(Q,v)
27:        u.color = black
```

Obviously the additional steps in `modified_BFS` (lines 6, 10, and 20-25) are constant time additions to BFS. Hence the time complexity of `modified_BFS` is still just $O(V + E)$. However, since we're restricting our search to trees, $|E| = |V| - 1$, so `modified_BFS` is just $O(V)$. Since we call this function $|V|$ times, our algorithm is $O(V^2)$ as desired.

D. Algorithm to compute the second-best MST of G

Finally, we present an algorithm to compute the second-best MST of G. The basic idea of the algorithm is to

- Compute the unique MST T
- Find the edge (u, v) not in T such that removing the maximum weight edge in the path between u and v in T and adding (u, v) increases the weight the least.
- Return this second best spanning tree.

```
 1: function FIND_SECOND_BEST_MST(T, s)
 2:     T = minimum_spanning_tree(G, w)                  ▷ Using Kruskal or Prim
 3:     find_max(T)              ▷ Annotates tree with max weight edge between each u, v
 4:     min_diff = ∞
 5:     for (u,v) ∈ G and ∉ T.E do
 6:         if w(u,v) - u.max[v] < min_diff then
 7:             edge_to_add = (u,v)
 8:             edge_to_delete = u.max_edge[v]
 9:     T” = T.E - edge_to_delete ∩ edge_to_add
10:     return T”
```

To analyze the runtime of this algorithm, note

- On line 2 we can compute the spanning tree in $O(E \lg V)$
- On line 3 running find_max on tree T is $O(V^2)$
- Assuming the lookups needed on lines 6-8 are constant time, the for loop on lines 5-8 run in $O(E - V)$ (since we check every edge not in the tree, and there are $V - 1$ edges in the tree).

3. Most reliable path

Let $G = (V, E)$ be a directed graph on which each edge is an associated value $r(u, v) \in [0, 1]$, which we interpret as the probability that the channel from u to v will not fail. Assume that the probabilities are independent. We present an efficient algorithm to find the most reliable path between two given vertices.

First, consider a path between $v_0 \sim v_k$, where $v_0 \sim v_k = \langle v_0, v_1, v_2, \cdots, v_k \rangle$. Then by independence

$$P(\text{Probability successful communication between } v_0 \text{ and } v_k) = \prod_{i=0}^{k-1} r(v_i, v_{i+1})$$

Now note we aim to find

$$\begin{aligned} v_0 \overset{*}{\sim} v_k &= \arg \max_{\{v_0 \sim v_k} \prod_{i=0}^{k-1} r(v_i, v_{i+1}) \\ &= \arg \min_{\text{All paths } v_0 \sim v_k} - \log[\prod_{i=0}^{k-1} r(v_i, v_{i+1})] \\ &= \arg \min_{\text{All paths } v_0 \sim v_k} - \sum_{i=0}^{k-1} \log\left(r(v_i, v_{i+1})\right) \end{aligned}$$

where we let k be the length of the path under consideration (i.e. a variable, not constant).

But now this is a problem amenable to our standard algorithms- since $-\log(x) \geq 0$ for all $x \in (0, 1]$, we can just use Dijkstra's. Thus the final algorithm is just

function MOST_RELIABLE_PATH(G, u, v)
  for (u', v') in G.E **do**
    if $r(u', v') = 0$ **then**
      delete (u', v') ▷ Edge not in any reliable path
    else
      $r(u', v') = -\log r(u', v')$
  `Dijkstra`$(G, -\log r, u)$
  return v.d

4. Nesting boxes

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

a. Nesting relation is transitive

We first show the nesting relation is transitive. Say x nests in y, and y nests in z. We show x nests in z.
Since y nests in z, there exists some π_{yz} such that $y_{\pi_{yz}(1)} < z_1, \cdots, y_{\pi_{yz}(d)} < z_d$.

Similarly, x nests in y, there exists some π_{xy} such that $x_{\pi_{xy}(1)} < y_1, \cdots, x_{\pi_{xy}(d)} < y_d$.

But then composing the permutation π_{yz} and π_{xy} yields a permutation such that

$$x_{\pi_{xy}(\pi_{yz}(1))} < z_1, \cdots, x_{\pi_{xy}(\pi_{yz}(d))} < z_d$$

Hence *nesting* is a transitive relation.

b. Efficient method to determine whether or not one box nests inside another

First, note that a box x nests inside a box y if there is relative ordering of the dimensions of x and y such that

$$x_{\pi(i)} < y_i \qquad \forall\ i \in \{1, \cdots, d\}$$

Clearly such a permutation exists if and only if

$$\text{sorted}(x)_i < \text{sorted}(y)_i \qquad \forall\ i \in \{1, \cdots, d\}$$

Thus, a linearithmic algorithm to see if x nests in y is

function BOXES_NEST(y, x)
  sort x ▷ using your favorite linearithmic sorting algorithm
  sort y
  for i in 1 to d **do**
    if $x_i \geq y_i$ **then**
      return false
  return true

C. Finding longest sequence of nesting boxes

Now suppose we are given a set of n d-dimensional boxes $\{B_1, \cdots, B_n\}$. We present an algorithm to find the longest sequence $\langle B_{i_1}, \cdots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \cdots, k-1$.

We first present a couple of helper function to sort the set of boxes. To briefly motivate `sort_boxes`, note this function (i) sorts the dimensions of each box in increasing order, then (ii) sorts the set of boxes in decreasing order of smallest dimension. Sort (ii) reveals the pairs of boxes B_i, B_j that do not nest; namely, if $i < j$, then B_i does not nest in B_j.

The second helper function `sorted_boxes_nest` just refactors `boxes_nest` to avoid duplicating the sorting step.

function SORT_BOXES$(\{B_1, \cdots, B_n\})$
  for i in 1 to n **do**
    sort B_i
  sort $\{\text{sorted}(B_1), \cdots, \text{sorted}(B_n)\}$ by smallest dimension in decreasing order

function SORTED_BOXES_NEST(x, y)
  for i in 1 to d **do**
    if $y_i \geq x_i$ **then**
      return false
  return true

Now that we've presented the helper functions, we present the actual algorithm. Note following the call to `sort_boxes`, we ease notation by letting B_i be the i^{th} box in sorted order. The algorithm proceeds by constructing a weighted DAG, where the edge (B_i, B_j) indicates B_j nests inside B_i. We assign each edge a weight of -1, since this will allow us to find the longest path through the DAG beginning at the super source using standard shortest-path algorithms (since the longest path with have the least weight).

function LONGEST_NESTED_SEQUENCE$(\{B_1, \cdots, B_n\})$
  sort_boxes$(\{B_1, \cdots, B_n\})$
  $E = \{super\}$
  $V = \{\}$
  $G = \{E, V\}$

6: **for** i in 1 to n **do**
7: $V = V \cup \{B_i\}$
8: $E = E \cup \{(super, B_i)\}$
9: $w((super, B_i)) = -1$
10: **for** j in $i+1$ to n **do** ▷ After sorting, no B_k with $k < i$ nests inside B_i
11: **if** sorted_boxes_nest(B_i, B_j) **then**
12: $E = E \cup \{(B_i, B_j)\}$
13: $w((B_i, B_j)) = -1$
14: DAG-SHORTEST-PATHS$(G, w, super)$
15: **return** - min$(B_i$.d) for i in 1 to n ▷ Longest path through DAG from *super*

Now we analyze the running time:

- First, on line 2 sorting all of the n boxes takes $O(d\ n \lg n)$ time, and sorting the n boxes in decreasing order of their smallest dimension takes $O(n \lg n)$ time.
- Lines 3-5 run in constant time
- The nested loops on lines 6-13 run in $O(d\ n^2)$ time, where n^2 is from the nested loop and d is from `boxes_nest`.
- `DAG-Shortest-Paths` takes $\Theta(V + E)$ time (by CLRS page 655). However, $V = n + 1$ and $E = O(n^2)$ (by construction). Thus `DAG-Shortest-Paths` takes $O(n + 1 + n^2) = O(n^2)$ time.
- Finding the minimum -B_i.d is $O(n)$

Thus, the running time is dominated by the $O(d\ n^2)$ nested loop.

5. Arbitrage

Suppose we are given n currencies $c_1, c_2, \cdots, c_n$ 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.

a. Existence of a profitable cycle in the currency graph

We present an efficient algorithm to determine whether or note there exists a sequence of currencies $\langle c_{i_1}, \cdots, c_{i_k} \rangle$ such that

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

First, note

$$
\begin{aligned}
& R[i_1, i_2] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 \\
\iff & \log\left(R[i_1, i_2] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1]\right) > \log 1 \\
\iff & \left[\sum_{j=1}^{k-1} \log\left(R[i_j, i_{j+1}]\right)\right] + \log\left(R[i_k, i_1]\right) > 0 \\
\iff & -\left[\sum_{j=1}^{k-1} \log\left(R[i_j, i_{j+1}]\right)\right] - \log\left(R[i_k, i_1]\right) < 0
\end{aligned}
$$

Hence, if we take the negative log of each element in R, the problem of determining whether there exists some sequence of currencies $\langle c_{i_1}, \cdots, c_{i_k}\rangle$ such that

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

is equivalent to the problem of detecting a negative-weight cycle in the $-\log R$ weighted graph.

1: **function** ARBITRAGE_POSSIBLE(R)
2:     Add *super* to $G.V$, with weight 1 on edge $(super, c_i)$ for all c_i.
3:     **return** not `Bellman-Ford`$(G, -\log R, super)$

Since `Bellman-Ford` returns `True` if and only if the graph contains no negative-weight cycles, all we need to do is apply the negative log transformation then return the logical inverse of the `Bellman-Ford` return value. This algorithm runs in $O(n^3)$ time, since

- Line 1 runs in $O(n)$ times, since we're simply adding $2n - 1$ entries to table R.
- `Bellman-Ford` runs in $O(VE)$ time, and in the currency problem $V = n$ and $E = O(n^2)$, so $O(VE) = O(n \cdot n^2) = O(n^3)$.

B. Printing profitable cycle from the currency graph

Now our objective is to print a profitable cycle from the currency graph if one exists. We again rely on (a modified version of) Bellman-Ford:

function MODIFIED_BELLMAN_FORD(R)
    Add *super* to $G.V$, with weight 1 on edge $(super, c_i)$ for all c_i.
    `Initialize_Single_Source`$(G, super)$
    for each vertex $v \in G.V$ **do**   ▷ Extra initialization step
        v.color = white
    for $i = 1$ to $|G.V| - 1$ **do**
        for each edge $(u, v) \in G.E$ **do**
            `Relax`$(u, v, -\log R)$ ▷ Relax using $-\log$ transformed R entry
    for each edge $(u, v) \in G.E$ **do**

```
        if v.d > u.d + (−log R(u, v)) then          ▷ If true u in negative-weight cycle
            cycle_found = True
            find_cycle(G, u)
    if not cycle_found then
        print "no negative cycle found"
```

To print the cycle, we call `find_cycle`. Note this function follows the parent chain back from u until it first detects a cycle; it prints out this cycle (which is necessarily a negative-weight cycle under the negative log transformation, and thus a profitable cycle for arbitrage).

```
function FIND_CYCLE(G, v)
    cycle_superset = []                                         ▷ Empty list
    while v.color == white do                  ▷ Exit first time v.color == black
        cycle_superset.insert(v)
        v.color = black
        v = v.π
    print v                                         ▷ v is first black vertex seen
    next_vertex = cycle_superset.head                  ▷ next_vertex is child of v
    while next_vertex != v do
        print next_vertex
        next_vertex = next_vertex.next
    print next_vertex          ▷ Exit with next_vertex == v; print to close cycle
```