6.854J / 18.415J Advanced Algorithms
Fall 2008

18.415/6.854 Advanced Algorithms

Problem Set Solution 4

Lecturer: Michel X. Goemans

1. **In class, we have seen Klein's cycle cancelling algorithm for the Min Cost Circulation Problem (MCCP). This algorithm requires $O(mCU)$ iterations in the worst case, i.e., its running time is not polynomial in m, $\log C$ and $\log U$. In this problem, we will see how to apply the idea of *cost scaling* on this algorithm to obtain an algorithm whose running time is polynomial in m, $\log C$, and U.**

   (In fact, it is possible to apply the same idea on both costs and capacities to obtain an algorithm whose running time is polynomial in m, $\log U$ and $\log C$, but this is not required in this problem.)

   Recall that in MCCP, a bidirected graph $G = (V, E)$, an anti-symmetric cost function $c : E \mapsto \mathbb{Z}$, and a capacity function $u : E \mapsto \mathbb{Z}$ are given. Let n and m denote the number of vertices and edges in G, $U = \max_{(v,w)\in E} |u(v, w)|$, and $C = \max_{(v,w)\in E} |c(v, w)|$.

   (a) **For every integer i, define the cost function $c^{(i)} : E \mapsto \mathbb{Z}$ as follows:**

$$c^{(i)}(v, w) := \text{sgn}(c(v, w)) \left\lfloor \frac{|c(v, w)|}{2^i} \right\rfloor,$$

   where $\text{sgn}(x)$ is the sign of x. Notice that by the above definition, $c^{(0)}(v, w) = c(v, w)$ and $c^{(\lceil \log(C+1) \rceil)}(v, w) = 0$. Our objective is to find a way to solve MCCP for the cost function $c^{(i)}$, given its solution for $c^{(i+1)}$.
   Let f be an optimum circulation for G with the cost function $c^{(i+1)}$ and the capacity function u. Prove that if we apply Klein's cycle cancelling algorithm on G with the cost function $c^{(i)}$ and capacity function u, starting from the circulation f, then the number of iterations of this algorithm is $O(mU)$.

   Since f is an optimum circulation for G with the cost function $c^{(i+1)}$, by the theorem that was proved in the class, there must be a potential function p, such that for every edge (v, w) in the residual graph G_f, $c_p^{(i+1)}(v, w) = c^{(i+1)}(v, w) + p(v) - p(w) \geq 0$. By the definition of $c^{(i)}$, for every edge (v, w), $c^{(i)}(v, w)$ is equal to $2c^{(i+1)}(v, w)$ plus one of the numbers -1, 0, or 1. Therefore, for every edge (v, w) in E_f,

$$c_{2p}^{(i)}(v, w) = c^{(i)}(v, w) - 2c^{(i+1)}(v, w) + 2\left(c^{(i+1)}(v, w) + p(v) - p(w)\right) \geq -1. \quad (1)$$

   We know that for every cycle (flow, resp.), the cost of the cycle (flow, resp.) is the same with respect to $c^{(i)}$ and $c_{2p}^{(i)}$. Therefore, if we replace the cost function $c^{(i)}$ with the cost function $c_{2p}^{(i)}$, Klein's cycle cancelling algorithm works in exactly the same way that it works with $c^{(i)}$, and outputs the same result. Therefore, we consider the cost function $c_{2p}^{(i)}$ instead of $c^{(i)}$. Let f^* be the optimal circulation for $c_{2p}^{(i)}$ and consider the circulation $f' := f^* - f$. By the definition of the residual graph, f' is a circulation in E_f. Therefore, the cost of f' is at least

$$\sum_{(v,w)\in E_f} c_{2p}^{(i)}(v, w) f'(v, w) \geq -2mU,$$

where the last inequality is a consequence of (1) and the fact that $|f'(v,w)| \leq 2U$ for every edge (v,w). Now, notice that the cost of f' is equal to the cost of f^* minus the cost of f. Therefore, the cost of f^* is at most $2mU$ units less than the cost of f. This, together with the fact that in every iteration of Klein's algorithm the cost of the circulation is decreased by at least 1, implies that the number of iterations of Klein's algorithm is at most $2mU$.

(b) **Use part (a) to obtain an algorithm for MCCP that requires $O(mU \log C)$ iterations.**

Start with the cost function $c^{(\lceil \log(C+1) \rceil)}(v,w)$. For this function, the any feasible circulation is an optimal circulation. Run the algorithm in part (a) for $i = \lceil \log(C+1) \rceil - 1, \ldots, 0$, to compute the optimal circulation with respect to the cost function $c^{(i)}$ for every i, and in particular the optimal circulation with respect to $c^{(0)} = c$. By part (a), every step takes at most $O(mU)$ iterations, and therefore the total number of iterations is $O(mU \log C)$.

2. **Consider a directed graph $G = (V,E)$ with a length function $l : E \to \mathbb{Z}$ and a specified source vertex $s \in V$. The Bellman-Ford shortest path algorithm computes the shortest path lengths $d(v)$ between s and every vertex $v \in V$, assuming that G has no directed cycle of negative length (otherwise the problem is NP-hard). Here is a description of this algorithm:**

The Bellman-Ford algorithm computes $d(v)$ by computing $d_k(v)$ = the shortest walk[1] between s and v using exactly k edges. $d_k(v)$ can be computed by the recurrence

$$d_k(v) = \min_{(w,v) \in E} [d_{k-1}(w) + l(w,v)] .$$

Let $h_l(v) = \min_{k=1,\ldots,l} d_k(v)$. It can be shown that if the graph has no negative cycle then $h_{n-1}(v) = d(v)$ for all $v \in V$. Moreover, the graph has no negative cycle iff, for all v, $d_n(v) \geq h_{n-1}(v)$.

(You are not required to prove any of the above facts.)

(a) **Let μ^* be the minimum average length of a directed cycle C of G, i.e.,**

$$\mu^*(G) = \min_{\text{directed cycles } C} \mu(C) = \min_C \frac{\sum_{(u,v) \in C} l(u,v)}{|C|}.$$

Using the Bellman-Ford algorithm, show how to compute μ^* in $O(nm)$ time. (Hint: Use the fact that if we decrease the length of each edge by μ the average length of any cycle decreases by μ.)

Notice that if we decrease the length of each edge in the graph by μ, the mean length of each cycle in the graph is decreased by μ. Therefore, μ^* is the largest value of μ such that if we decrease the length of each edge by μ, there will be no negative length cycle in the graph. Also, by the definition of $d_l(v)$, if one decreases the length of each edge by μ, the effect on $d_l(v)$ is simply that it will be decreased by $l\mu$. Therefore, by the above facts about the Bellman-Ford algorithm, μ^* is the largest value of μ for which all the inequalities

$$d_n(v) - n\mu \geq \min_{k=1,\ldots,n-1} (d_k(v) - k\mu) \tag{2}$$

[1] A walk is like a path except that vertices might be repeated.

hold for every $v \in V$. Let's compute the largest value of μ that satisfies the above inequality for a fixed v. This value is equal to the maximum over k of the largest value of μ that satisfies the inequality $d_n(v) - n\mu \geq d_k(v) - k\mu$. Therefore, for a fixed v, the largest μ that satisfies inequality (2) is $\max_{k=1,\ldots,n-1} \frac{d_n(v)-d_k(v)}{n-k}$. Thus, the largest μ that satisfies inequality (2) for *every* vertex $v \in V$ is equal to

$$\mu^* = \min_{v \in V} \max_{k=1,\ldots,n-1} \frac{d_n(v) - d_k(v)}{n-k}.$$

It is obvious that given all the values $d_l(v)$ that are computed by the Bellman-Ford algorithm, the above value can be computed in $O(n^2)$.

(b) **Can you find the cycle C with $\mu(C) = \mu^*$ using only $O(n^2)$ additional time? (In other words, suppose you are given all the values that the Bellman-Ford algorithm computes. Can you find a minimum mean cost cycle using this information in $O(n^2)$?)**

We use the following algorithm:

1. Compute the value of μ^* using the algorithm in part (a).
2. Subtract μ^* from the length of each edge. From now on, when we talk about the length of an edge, we mean the updated length.
3. For every $l = 0, \ldots, n$ and every vertex $v \in V$, compute the following values:

$$\begin{aligned} d'_l(v) &:= d_l(v) - l\mu^* \\ h'_n(v) &:= \min_{k=1,\ldots,n-1} d'_k(v) \end{aligned}$$

4. Find a vertex v such that $h'_n(v) = d'_n(v)$.
5. Let $v_0, v_1, v_2, \ldots v_n$ be a sequence of vertices defined as follows:
   - $v_0 = v$.
   - For $k = 0, \ldots, n-1$, v_{k+1} is a vertex for which we have

$$d'_{n-k}(v_k) = d'_{n-(k+1)}(v_{k+1}) + l(v_{k+1}, v_k). \tag{3}$$

6. Find i and j $(i < j)$ such that $v_i = v_j$.
7. Output the cycle $C := v_j, v_{j-1}, \ldots, v_{i+1}, v_i = v_j$.

We prove that the above algorithm finds the minimum mean length cycle. After step 2, we know that a cycle of total length zero must exist in the resulting graph (and this cycle corresponds to a cycle with the minimum mean length in the original graph). Therefore, we only need to find such a cycle. Also, we know that there is no negative length cycle in the graph.

After step 3, by the argument in part (a), we know that $d'_l(v)$ is the length of the shortest walk with exactly l edges from s to v, and $h'_n(v)$ is the length of the shortest walk with at most n edges from s to v.

In step 4, such a vertex v must exist, since otherwise we would have $h'_n(v) < d'_n(v)$ for every vertex v, and therefore we can find a small positive ϵ such that if we subtract ϵ from the length of each edge, all the inequalities $h'_n(v) \leq d'_n(v)$ are still satisfied. Therefore, subtracting ϵ from the length of each edge does not create any negative length cycle. This is in contradiction with the fact that there is a zero length cycle in the graph.

The sequence $v_0, v_1, v_2, \ldots v_n$ defined in step 5 is well-defined, because by the definition of $d'_k(v)$ we know that for every v and k, there must be a vertex w such that $d_k(v) =$

$d_{k-1}(w) + l(w,v)$. Also, notice that by summing Equation (3) for $k = 0, \ldots, n-1$, we obtain $d'_n(v) = d'_0(v_n) + l(P)$, where $l(P)$ is the length of the walk $P := v_n v_{n-1} \ldots v_1 v_0$. Therefore, since $d'_0(w)$ is 0 if $w = s$ and infinity if $w \neq s$, and $d'_n(v)$ is not infinity (this is because of the assumption that the graph is strongly connected), the equality $d'_n(v) = d'_0(v_n) + l(P)$ implies that $v_n = s$ and P is a walk from s to v of length $d'_n(v)$.

Since there are only n vertices in the graph, there must be i and j ($i < j$) such that $v_i = v_j$. Consider all the equalities $d'_{n-k}(v_k) = d'_{n-(k+1)}(v_{k+1}) + l(v_{k+1}, v_k)$ for $k = i, \ldots, j-1$. By adding all these equalities together, we get $d'_{n-i}(v_i) = d'_{n-j}(v_j) + l(C)$, where $l(C)$ is the length of the cycle C that is output by the above algorithm. We claim that $l(C) = 0$. Suppose this claim is not true. Then we must have $l(C) > 0$ (Remember that there is no negative length cycle in the graph). Therefore, $v_n v_{n-1} \ldots v_{j+1} v_j v_{i-1} v_{i-2} \ldots v_0$ is a walk that uses less than n edges and its length is $d'_n(v) - l(C) < d'_n(v)$. This is a contradiction with the assumption $h'_n(v) = d'_n(v)$.

Therefore, the above algorithm finds the minimum mean length cycle correctly. It is also obvious from the description of the algorithm that the running time of the above algorithm is $O(n^2)$.

3. **Suppose we have n objects that we want to store in a data structure. After storing these objects in the data structure, we would like to perform m find operations on the data structure. Assume that the i'th object is accessed $k_i \geq 1$ times. Therefore, $\sum_{i=1}^{n} k_i = m$. We want to evaluate the performance of the data structure by computing the total running time of these m find queries (no other operation, such as delete or insert, is performed on the data structure).**

   (a) **Show that if we store the objects in a splay tree, then no matter what the initial configuration of the splay tree is, and no matter in which order we access the objects, the total running time of the m access operations is at most**

$$O\left(m + \sum_{i=1}^{n} k_i \log(\frac{m}{k_i})\right).$$

We use the theorem that is proved in the class with the weight function $w_i = k_i$. With this weight function, the theorem says that the amortized cost of accessing the i'th object is at most $1 + r(v) - r(i)$, where $r(v)$ is the credit of the root and $r(i)$ is the credit of the node of the tree that contains the i'th object. We know that the total weight of the tree is $\sum_{i=1}^{n} k_i = m$. Therefore, $r(v) = \log m$. Also, $r(i) = \log(s(i)) \geq \log(k_i)$. Therefore, the amortized cost of accessing the i'th object is at most $1 + \log(m/k_i)$. Thus, the total amortized cost of m accesses is at most $m + \sum_{i=1}^{n} k_i \log(m/k_i)$. By the definition of the amortized cost, this is equal to the total running time of m accesses, plus the total credit in the tree after the accesses, minus the total credit in the tree that we start with. Since in any tree the credit of the node that contains the i'th object is at least $\log k_i$ and at most $\log m$, the total credit in the tree at any time is at least $\sum_{i=1}^{n} \log k_i$ and at most $n \log m$. Therefore, the total running time of m accesses is at most

$$\begin{aligned} m + \sum_{i=1}^{n} k_i \log(\frac{m}{k_i}) + n \log m - \sum_{i=1}^{n} \log k_i &= m + \sum_{i=1}^{n} (k_i + 1) \log(\frac{m}{k_i}) \\ &= O\left(m + \sum_{i=1}^{n} k_i \log(\frac{m}{k_i})\right), \end{aligned}$$

where the last equality follows from the fact that $k_i \geq 1$ and therefore $k_i + 1 \leq 2k_i$.

(b) **Show that if we store the objects in a static binary search tree (i.e., a binary search tree that does not change by a find operation), then no matter in which order the objects are stored in the BST, and no matter in which order they are accessed, the total running time of the m access operations is at least**

$$\Omega\left(m + \sum_{i=1}^{n} k_i \log(\frac{m}{k_i})\right).$$

We use induction on n to prove that the total cost of m operations on any BST is at least $\frac{1}{3}(m + \sum_{i=1}^{n} k_i \log(\frac{m}{k_i}))$. The induction basis is trivial. We prove the induction step as follows: Consider an arbitrary BST and let r be the object that is in the root of the tree, and L and R be the sets of objects that are in the left and right subtrees, respectively. Also, let m_L and m_R denote $\sum_{i \in L} k_i$ and $\sum_{i \in R} k_i$, respectively. By the definition of BST, the left and right subtrees are BSTs. Therefore, by the induction hypothesis (and considering the fact that the cost of accessing each element in L in our tree is one plus the cost of accessing it in the left subtree), the total cost of the m_L operations on the elements of L is at least

$$m_L + \frac{1}{3}(m_L + \sum_{i \in L} k_i \log(\frac{m_L}{k_i})),$$

and similarly for the right subtree. Also, the cost of k_r accesses to the object r is precisely k_r. Therefore, the total cost of m accesses is at least

$$\frac{4}{3}m_L + \frac{1}{3}\sum_{i \in L} k_i \log(\frac{m_L}{k_i}) + \frac{4}{3}m_R + \frac{1}{3}\sum_{i \in R} k_i \log(\frac{m_R}{k_i}) + k_r$$

Considering the facts that $m_L + m_R + k_r = m$, $\sum_{i \in L} k_i = m_L$ and $\sum_{i \in R} k_i = m_R$, the above cost is lower bounded by

$$m + \frac{1}{3}(m_L \log m_L + m_R \log m_R + k_r \log k_r) - \frac{1}{3}\sum_{i=1}^{n} k_i \log k_i.$$

The function $f(x) := x \log(x)$ is convex, therefore, $\frac{1}{3}(f(x) + f(y) + f(z)) \geq f(\frac{x+y+z}{3})$ for every $x, y, z > 0$. Thus, the above cost is at least

$$\begin{aligned} m + \frac{m}{3}\log(\frac{m}{3}) - \frac{1}{3}\sum_{i=1}^{n} k_i \log k_i &\geq (1 - \frac{1}{3}\log 3)m + \frac{1}{3}\sum_{i=1}^{n} \log(\frac{m}{k_i}) \\ &\geq \frac{1}{3}(m + \sum_{i=1}^{n} k_i \log(\frac{m}{k_i})) \end{aligned}$$

This completes the proof of the induction step.