6.854J / 18.415J Advanced Algorithms
Fall 2008

18.415/6.854 Advanced Algorithms

Problem Set 2

If you have any doubt about the collaboration policy, please check the course webpage.

Problems:

1. Given a directed graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$ and a length function $l : E \to \mathbb{R}$, the fattest path problem is to find a simple path P from s to t which maximizes $\min_{(v,w)\in P} l(v, w)$.

   (a) Give a modification of Dijkstra's algorithm for the shortest path problem which solves the fattest path problem. Argue correctness of your algorithm.

   (b) Suppose that all arcs lengths are integer-valued between 1 and m where $m = |E|$. Can you provide an implementation of your algorithm that runs in $O(m)$ time? (Hint: Do not use any fancy priority queue.)

2. In this problem, you will show that the fattest augmenting path algorithm for the maximum flow problem can be implemented to run in $O(m)$ time per iteration after some basic preprocessing. Remember that in the fattest augmenting path algorithm, the augmenting path with largest minimum *residual* capacity is chosen at every iteration.

   (a) Show that if we have a total ordering of the residual capacities then the fattest augmenting path can be found in $O(m)$ time.

   (b) Show that, this total ordering of the residual capacities can be maintained in $O(m)$ time after pushing flow along one augmenting path (how do the residual capacities change)?

   (c) What is the running time ofthe resulting inplementation of the fattest augmenting path algorithm?

3. 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

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

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.)

(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 $O(n^2)$?)

4. In this problem, we will propose another way to solve the minimum mean cost cycle problem. The resulting algorithm will be quite slow, but the technique is widely applicable (and for other problems, this will give the fastest known approach). The problem of finding μ^* is equivalent to the problem of finding the largest value of μ such that the graph with lengths $l_\mu(u,v) = l(u,v) - \mu$ has no negative cycles.

(a) Argue that for a given value of μ, we can decide whether $\mu^* \geq \mu$ or $\mu^* < \mu$ by performing at most $O(A(m,n))$ additions of 2 numbers and $O(C(m,n))$ comparisons involving 2 numbers (and no other operations except control statements). Please state the values you can obtain for $A(m,n)$ and $C(m,n)$. Observe that, as we are performing only additions and comparisons, all the numbers involved are linear functions of the input lengths and μ.

(b) Now suppose we run the above algorithm with μ equal to the *unknown* value μ^*. We can easily perform the additions provided that we store all the numbers (including the inputs) as linear functions of μ^* (i.e. of the form $a + b\mu^*$). Explain how we can resolve the comparisons (even though we do not know μ^*). (It is normal if the solution requires a fair amount of time to resolve each comparison.) As a function of $A(m,n)$ and $C(m,n)$, what is the total running time of your algorithm to compute μ^*?

5. We argued in lecture that for the maximum flow problem, there always exists a maximum flow which is integer-valued if the capacities are integral. Prove that a corresponding statement for minimum cost circulations also holds, namely that if the capacities and the costs are integer-valued then (i) the minimum cost circulation can be chosen to be integer-valued and (ii) the vertex potentials proving optimality can also be chosen to be integer-valued.

6. In this problem, we will add a time dimension to network flows. Suppose we have a network $G = (V, E)$ in which each arc has unit capacity ($u(v, w) = 1$ for all arcs (v, w)), and we have two special vertices, a source s and a sink t. Our network for example could be a computer network and our unit of flow could be a packet. Each arc also has an integer-valued transit time $\tau(v, w) \in \mathbb{Z}_+$ which represents the time it takes (a unit of flow or packet) to travel through the arc. At every unit of time, say at time d, only one packet can enter the arc (there might be several packets already travelling through the arc since there could have been packets injected in it at times $d-1$, $d-2$, etc.). We can assume that vertices can instantaneously accept packets on its incoming arcs and also inject one packet (if available) on each of its outgoing arcs (and if there are remaining packets, they can be queued at the vertex).

   The first problem we consider is, given a deadline D, to find the maximum number $k(D)$ of packets that can enter the network at s at time 1 or later and leave the network at vertex t at time D or earlier. As an example, suppose that our graph has only 3 arcs (s, a), (a, t) and (s, t) each with a transit time of 2. Then, if $D = 5$, the answer should be $k(5) = 4$ packets. Indeed, we can send 3 packets along the arc (s, t), entering at times 1, 2 and 3 and leaving at time 3, 4 and $5 \leq D$. We can also send a 4th packet, along the path (s, a) and (a, t); it will enter the arc (s, a) at time 1, arrive at a at time 3 and arrive at t at time 5. (Observe by the way that in this example, no packet had to wait at intermediate vertices.)

   (a) Construct a maximum flow instance on a bigger network $G' = (V', E')$ such that the solution of this maximum flow instance allows you to find $k(D)$ and the scheduling (when they travel through each arc) of the packets in the original network G. $|V'|$ can be of the order of $D|V|$.

   (b) The solution above is not polynomial when D is part of the input (since the size of the network grows linearly in D). To find a polynomial time algorithm, consider the following circulation problem. Take the original graph $G = (V, E)$ with all arcs of capacity 1 and give arc $(v, w) \in E$ a cost $c(v, w) = \tau(v, w)$. Add one arc (t, s) of infinite capacity and cost equal to $-D$. Let $-C^*$ be the cost of the minimum cost circulation f^*. Prove the following claim: C^* is precisely $k(D)$. Also explain how one can find the scheduling of the packets from the minimum cost circulation f^*.

(It might be helpful to first see what happens on the simple example with 3 arcs given above.)

(c) Now, suppose that we want to solve the converse problem. We would like to send k packets from s to t so that the last packet arrives at t as early as possible. Propose a polynomial-time algorithm which given k finds $D(k)$, the minimum time at which all packets have arrived at t. What is the running time of your algorithm as a function of $n = |V|$, $m = |E|$, $T = \max \tau(v, w)$ and k?

(d) Your algorithm for (c) is probably not strongly polynomial, in the sense that its running time depends on $\log(T)$ and/or $\log(k)$. Can you propose a strongly polynomial-time algorithm? Just sketch it (a few lines are enough); do not give all the details. (Kind of hint: this solution will be much slower than (c) when T and k are small.)

(By the way, all the results above are still true if the capacities are integers possibly greater than 1; in such a case, at every time d, at most $u(v, w)$ packets can be injected in arc (v, w). Arguing about the validity of (b) is slightly more difficult.)

7. Which question did you like the most (excluded this one...)? Which question did you like the least?