CS 270 - Homework 2

Walid Krichene (23265217)

(1) In both routing problems, we seek to send one unit of flow between each pair (s_i, t_i), and to minimize the maximum edge flow. If $c(e)$ is the total flow on edge e, then the problem is to minimize $\max_e c(e)$.

In the path routing problem, we add the constraint that flows should be integers, therefore a single path will connect each pair of source-sink (s_i, t_i). The fractional routing problem is thus a relaxation of the path routing problem.

(1.a) Consider the simple network in figure 1, where the only source-sink pair is (s, t). In the fractional routing problem, the optimal value is $1/2$, and corresponds to routing half of the flow on each edge. In the path routing problem, the optimal value is 1.

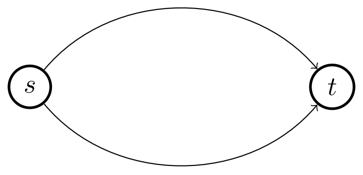

Figure 1: Example network for which the solution of fractional routing is strictly better than the solution of the path routing.

(1.b) Consider the toll problem, in which we seek to maximize $\sum_i w(p_i^*)$ where p_i^* is the shortest path for the weight function w. Thus the problem is

$$\begin{aligned} \text{maximize}_w \quad & \sum_i w(p_i^*) \\ \text{subject to} \quad & \sum_e w(e) = 1 \\ & \forall e,\ w(e) \geq 0 \end{aligned}$$

here $w(p)$ is the total weight on a given path p, and p_i^* is the shortest path connecting (s_i, t_i).

For all i, let P_i be the set of paths that connect source-sink pair (s_i, t_i), and let $P = \cup_i P_i$ (disjoint union). A feasible point for the fractional routing problem is given by a function (or a vector of flows) f

$$\begin{aligned} f : P &\to [0, 1] \\ p &\mapsto f(p) \end{aligned}$$

where $f(p)$ is the amount of flow along path p. This function also has to satisfy the following conditions

- for all i, $\sum_{p \in P_i} f(p) = 1$

We denote by $\mathcal{F}$ the feasible set. The problem is

$$\begin{aligned} &\text{minimize}_{f\in\mathcal{F}} && \max_{e\in E} c(e) \\ &\text{subject to} && c(e) = \sum_i \sum_{p\in P_i : e\in p} f(p) \end{aligned}$$

here $c(e)$ is the total flow on edge e. (this is by far not the most efficient way of encoding the problem if we wanted to solve it is suitable for this proof)

Now let f be any feasible point for the fractional routing problem, and let $\bar{c} = \max_e c(e)$ be the corresponding objective value. And let w^* be a maximizer of the toll problem, with corresponding shortest paths p_i^*. Then we have

$$\begin{aligned} \bar{c} &= \sum_e w^*(e)\bar{c} && \text{since } w^* \text{ is feasible for the toll problem} \\ &\geq \sum_e w^*(e)c(e) \\ &= \sum_e w^*(e) \sum_i \sum_{p\in P_i : e\in p} f(p) && \text{using the definition of } c(e) \\ &= \sum_i \left(\sum_{p\in P_i} f(p) \left(\sum_{e:e\in p} w^*(e) \right) \right) && \text{rearranging the sum} \\ &= \sum_i \left(\sum_{p\in P_i} f(p)w(p) \right) \\ &\geq \sum_i \left(\sum_{p\in P_i} f(p) \right) w(p_i^*) && \text{using } w(p_i^*) = \min_{p\in P_i} w(p) \\ &= \sum_i w(p_i^*) && \text{using } \sum_{p\in P_i} f(p) = 1 \end{aligned}$$

which proves that the optimal value of the toll game is a lower bound on the fractional routing game

(3) Let $G = (U \cup V, E)$ be a bipartite graph, with $|U| = |V| = n$. To simplify notation, let $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$. Let m be the number of edges of positive weight.

(3.a) Given a weight function $w : E \to \mathbb{R}$, let (M_w, p_w) be a solution pair for the max weight matching problem and the vertex cover problem with weights w. In other words,

- $M_w \subset E$ is a solution to the maximum wight matching problem.
- $p_w : U \cup V \to \mathbb{R}$ is a solution to the vertex cover problem. In particular,

$$p_w(u) + p_w(v) \geq w(e), \ \forall e = (u, v)$$

  with equality for $e \in M_w$ (matched edge $\Rightarrow$ tight edge).

Fix an edge $\bar{e} \in E$, and consider the max weight matching problem on the new weight function $w' : E \to \mathbb{R}$, obtained by increasing the weight of edge $\bar{e}$:

$$w'(\bar{e}) = w(\bar{e}) + c$$

We seek an efficient algorithm to compute a solution pair $(M_{w'}, p_{w'})$

We first observe that after increasing the weight of the edge $\bar{e} = (\bar{u}, \bar{v})$, the price p_w may become infeasible, if $p_w(\bar{e}) > p_w(\bar{v}) + p_w(\bar{u})$. Therefore we first increase the price of $\bar{u}$

$$p(\bar{u}) = p_w(\bar{u}) + c$$

Tight edges incident to $\bar{u}$ may become loose, however all other tight edges remain tight. Therefore by increasing $p(\bar{u})$, we drop at most one edge from the matching, thus we have a matching of size $n - 1$ and it suffices to increase the size of the matching by 1.

We look for an augmenting alternating path on the tight edges, that connects the two unmatched nodes (as in the original version of the algorithm presented in class, we orient matched tight edges from U to V and tight unmatched edges from V to U). This can be done by running BFS, starting from the unmatched node in U, and updating the reachable set $R = U_R \cup V_R$. When the BFS reaches a fixed point (i.e. no tight edges going out of the reachable set), then

- either we found an augmenting alternating path, and we are done, since this increases the size of the matching and we obtain a matching of size n.
- or no augmenting alternating path is found, and we have a cut between the set of reachable nodes $R = U_R \cup V_R$, and the set of non reachable nodes $N = U_N \cup V_N$ (so we have $U = U_R \cup U_N$ and $V = V_R \cup V_N$, the unions being disjoint). Then all edges from U_R to V_N are non-tight, and we can update the prices as follows

$$\forall u \in U_R, \ p(u) := p(u) - \delta$$
$$\forall v \in V_R, \ p(v) := p(v) + \delta$$

  where

$$\delta = \min_{e=(u,v) \in (U_R \times V_N)} p(u) + p(v) - w(e)$$

  the edges between U_R and V_R are still tight, and we created a new tight edge from U_R to V_N, thus increasing the reachable set. We continue the BFS until the next fixed point is reached.

The BFS requires exploring $O(m)$ edges. However, there are two issues here:

- if no additional data structure is used, computing each δ requires $O(n)$ time. However this can be improved by using a priority queue (with updates $O(\log n)$) that stores non-tight edges going out of the reachable set, where the priority of an edge $e = (u, v)$ is its "non-tightness" $p(u) + p(v) - w(e) > 0$. This requires a minor modification of the algorithm: instead of running the BFS on the graph of tight edges, we run it on the entire graph, and when a non-tight edge is explored we update the priority queue but do not add the node to the reachable set.

- each price update requires $O(n)$ time. We can work around this by storing, at the k-th fixed point

  1. the nodes that we need to update
  2. the value of δ

  then we only update the prices at the end of the BFS. More precisely, let $U_{R_k} \cup V_{R_k}$ be the reachable set at fixed point k, and δ_k be the corresponding price update. Call

  $$U_{B_k} = U_{R_k} \setminus U_{R_{k-1}}$$
  $$V_{B_k} = V_{R_k} \setminus V_{R_{k-1}}$$

  such that

  $$U_{R_k} = \cup_{i \leq k} U_{B_k}$$
  $$V_{R_k} = \cup_{i \leq k} V_{B_k}$$

  B_k stands for "the kth bucket". Let

  $$k(u) = \{k : u \in U_{B_k}\}$$
  $$k(v) = \{k : v \in V_{B_k}\}$$

  This is well defined since the U_{B_k}'s form a partition of U (and similarly for V). Then we have u is in the reachable set U_{R_k} for all $k \geq k(u)$, and the price of u is changed at each $k \geq i$. Therefore the prices at stage k are given by

  $$\forall u,\ p(u) = p_0(u) - \sum_{k \geq k(u)} \delta_k \tag{1}$$
  $$\forall v,\ p(v) = p_0(v) - \sum_{k \geq k(v)} \delta_k \tag{2}$$

  where p_0 are the initial prices (before running BFS). Therefore we only need to keep track of the δ_k, and we can compute the prices efficiently at the end of the BFS.

  Updating the priorities: we need to ensure that the priorities remain consistent, without having to update all the priorities at each stage k. At stage k, for an edge $e = (u, v)$ in the priority queue, we know that u is reachable and v is not, i.e. therefore the non-tightness of edge e is decreased by δ_k. This does not change the order in the queue, and instead of updating the priorities of nodes currently in the queue, we add δ_k to the priority of all nodes in future stages. Thus we work with these offset priorities.

To summarize: we run a BFS (on the graph of tight and non-tight edges, oriented as explained above) that maintains lists of reachable nodes $R_k = U_{R_k} \cup V_{R_k}$, a list of price updates δ_k, and a priority queue of non-tight edges going out of the reachable set. When a new edge is explored, if the edge is non-tight, add it to the priority queue (using the offset priority as explained above). If the edge is tight, add the new node to the reachable set U_{R_k} or V_{R_k}, and to the bucket U_{B_k} or V_{B_k}. When the k-th fixed-point is reached

- delete the top element e_k from the priority queue, add the new node to the list of nodes to explore.

- store the value δ_k
- increment k

Once BFS is done, compute the prices using (1), (2) and the buckets that were computed during the BFS.

Complexity

- The BFS explores m edges, and each explored edge requires up to $O(\log n)$ time (if the priority queue needs to be updated). Thus the complexity of the BFS run is $O(m \log n)$
- computing the prices $p_{w'}$ is done in $O(\max(n, m))$. (in more details: for each $i \in \{1, \ldots, k_{\max}\}$, compute $\bar{\delta}_i = \sum_{k \geq i} \delta_k$, this is done in $O(k_{\max}) = O(m)$, then for each bucket U_{B_k}, the price of nodes $u \in U_{B_k}$ is $p_0(u) + \bar{\delta}_k$. Computing these prices is $O(1)$ per node, thus the total complexity is $O(n)$)

The total complexity is $O(m \log n) + O(\max(n, m)) = O(m \log n)$ time.

(3.b) We can use the algorithm in (3.b) to solve the max weight matching problem (and the vertex cover problem) in $O(m^2 \log n)$ time:

- start with the weight function $w \equiv 0$, (identically zero, i.e. $w(e) = 0 \forall e \in E$). A solution pair is given by (M, v) where $v \equiv 0$ is identically zero, and $M = \{(u_i, v_i)\}_{i=1\ldots n}$ for example.
- for all $e \in E$, update the solution by increasing the weight of edge e from 0 to $w(e)$, and applying the algorithm from $(3.a)$.

Each iteration is $O(m \log n)$, and we have m iterations (one per edge) thus the total complexity is $O(m^2 \log n)$