CS4234: Optimization Algorithms Sept. 17, 2016

Tutorial Week 6

Problem 1. Scaling it Down.

Recall that Ford-Fulkerson takes time $O(m^2U)$ to find a maximum flow on a graph with m edges in which the maximum capacity is U (as long as all the capacities are integers, and the augmenting path is found via depth-first-search). When U is large, the algorithm can be very slow, while when U is small, it should be quite efficient.

In this problem, we look at a simple way to improve the performance: scale all the capacities to some smaller value, hence improving the running time. Then scale the values back up.

Problem 1.a. Assume you are given a graph $G = (V, E)$ with n nodes and m edges, source s, target t, and capacities c. For some value k, consider the following algorithm:

- For all edges $e \in E$, let $c'(e) = \lfloor c(e)/k \rfloor$. (That is, scale all the capacities down by k, and take the floor.)
- Run Ford-Fulkerson on graph G with capacities c'. Let f' be the resulting flow.
- For each edge e, set $f(e) = k \cdot f'(e)$. (That is, scale all the flows up by k.)
- Run Ford-Fulkerson on graph G with capacities c starting with flow f as the initial flow.

Prove that this algorithm is correct (i.e., it returns a *feasible* flow, and the result is the maximum flow). Analyze the running time of the algorithm. What is a good value for k? What is the final running time based on that value?

Problem 1.b. Assume that each integer value consists of at most w bits. (Here, the letter w stands for word size.) That is, we can think of $w = \log U$. Now consider the following algorithm, for graph $G = (V, E)$ with capacities c:

1. Flow $f = 0$.
2. For $k = w$ down to 0 do:
   (a) $f = 2f$.
   (b) $c' = \lfloor c/2^k \rfloor$.
   (c) Run Ford-Fulkerson on G with capacities c' starting with flow f.
3. Return flow f.

Prove that this algorithm is correct (i.e., it returns a *feasible* flow, and the result is the maximum flow). Analyze the running time of the algorithm.

Problem 2. Bipartite Matching

In this problem, we look at the problem of *bipartite* matching. Imagine you are given a bipartite graph $G = (A, B, E)$ where sets A and B are vertices and E are edges. The graph is undirected, and there are *no weights*. The goal is to find a maximum sized matching. (Again: notice there are no weight!) Remember, a matching is a set of edges M that are disjoint, i.e., no two edges e_1 and e_2 in the matching share a vertex.

Problem 2.a. Show how to use Ford-Fulkerson to find a maximum sized matching. Prove that the result is a matching, and that it is the maximum-sized matching. Analyze the running time of your algorithm.

Problem 2.b. Imagine we have an interstellar party on Mars. There are n Plutonians visiting Mars, and we have arrange for n Martians to be their hosts. Each Martian is friends (on Facebook) with exactly k Plutonians. Each Plutonian is friends (on Facebook) with exactly k Martians. Your job is to assign each Plutonian a guide on Mars, i.e., match each Plutonian with a Martian host. Is it always possible?

Problem 3. Disjoint Paths

Assume you have an undirected graph $G = (V, E)$ with a source s and a target t (and *no weights*).

Problem 3.a. Give an algorithm that finds the maximum number of edge-disjoint paths from s to t. Two paths P_1 and P_2 are edge disjoint if they do not share any edges—they may share a node. (Hint: use Ford-Fulkerson.) What is the running time of your algorithm?

Problem 3.b. What if you want to find the maximum number of node-disjoint paths from s to t? (Two paths are node-disjoint if they do not share any nodes.)

Problem 3.c. **(Extra hard)** Now consider the case where you have many source-target pairs:

$$(s_1, t_2), (s_2, t_2), \ldots, (s_k, t_k).$$

The goal is to find the maximum number of pairs that you can connect by edge disjoint paths. (That is, a path P connects a pair (s_i, t_i) if it runs from s_i to t_i and does not intersect any other path.) Notice that the algorithms from the previous part *do not* work here. (This is not a MaxFlow problem!) The Steiner-tree style algorithms also do not work. (Why?) In fact, your goal here is to find a $2\sqrt{m}$-approximation of optimal.

Hint: Think about the greedy algorithm that adds shortest paths first. Look at the optimal solution, and divide them into short paths of length $\leq \sqrt{m}$ and long paths of length $> \sqrt{m}$. If some path P is in the optimal solution, why didn't greedy find that one?