CSORE 4231 ANALYSIS OF ALGORITHMS I Homework 5

Francis Zhang xz3279

September 19, 2024

1. Minimum mean-weight cycle algorithm

Solution:

a. If $\mu^* = 0$, it implies the minimum of $\frac{1}{k}\sum_{i=1}^{k} w(e_i)$ is zero. Consequently, the smallest possible value for $\sum_{i=1}^{k} w(e_i)$ is 0, indicating that all cycles have a non-negative total weight. Furthermore, any path from s to v can be transformed into a simple path devoid of cycles, equating its weight to a path with at most $n-1$ edges. By considering the minimum across paths with varying edge counts, we effectively obtain the minimum across all potential paths.

b. Given that $\mu^* = 0$, we infer that negative weight cycles do not exist. Consequently, the inequality

$$\max_{0\le k<n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \ge 0$$

implies

$$\max_{0\le k<n-1} \delta_n(s,v) - \delta_k(s,v) \ge 0.$$

This suggests that as the number of permissible edges in a path increases past $n-1$, the minimum cost of the path does not decrease. It follows that there exists at least one path whose cost is as low as any path with fewer than n edges. While paths longer than $n-1$ edges may also be of minimal cost, due to the possibility of zero cost cycles, such paths are not guaranteed to be cheapest from s to v unless they incorporate a zero cost cycle that is also on a minimum cost path from s to v.

c. Since the total cost of the cycle is 0, and one segment of it incurs a cost of x, the weight of the remaining segment must be $-x$ to compensate. Consider a shortest path from s to u. Traversing from u to v along the cycle yields a path from s to v with length $\delta(s,u)+x$, establishing that $\delta(s,v) \le \delta(s,u)+x$. Conversely, starting with a shortest path from s to v and then moving through the cycle from v to u—a segment we've acknowledged costs $-x$—suggests $\delta(s,u) \le \delta(s,v) - x$. Rearranging gives us $\delta(s,u)+x \le \delta(s,v)$. The presence of both inequalities implies that they are in fact equalities, thus we have $\delta(s,u)+x = \delta(s,v)$.

d. To demonstrate this, we locate a vertex v and a natural number $k \le n-1$ for which $\delta_n(s,v)-\delta_k(s,v) = 0$. We commence by selecting any shortest path from s to a vertex on the cycle that utilizes the fewest edges. Continuing, we traverse the cycle until we have covered n edges. The vertex at which we arrive becomes our chosen v. Since the δ value of v remains unaltered after considering paths of length n, as indicated in part a, it follows that there exists a path length k possessing identical cost. Hence, we establish that $\delta_n(s,v) = \delta_k(s,v)$.

e. This is an immediate consequence of the previous problem and part b. Part b asserts that for all vertices v the following inequality is satisfied:

$$\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \ge 0 \tag{1}$$

The foregoing section demonstrates the existence of a vertex v within every minimum weight cycle for which:

$$\max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} = 0 \tag{2}$$

This implies:

$$\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} \le 0 \tag{3}$$

By synthesizing the two inequalities, we arrive at the desired equality.

f. If we add t to the weight of each edge, the mean weight of any cycle becomes

$$\mu(c) = \frac{1}{k}\left(\sum_{i=1}^{k}(w(e_i)+t)\right) = \frac{1}{k}\left(\sum_{i=1}^{k} w(e_i)\right) + t. \tag{4}$$

This is the original, unmodified mean weight cycle, plus t. Because the mean weight of every cycle is altered in this manner, the cycle with the lowest mean weight remains the one with the lowest mean weight. Consequently, μ^* will increase by t. Assuming that we initially calculate μ^* and then deduct μ^* from the weight of every edge, we set the new μ^* to zero. According to part e, this implies that

$$\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} = 0. \tag{5}$$

Since both are equal to zero, they are equal to one another.

g. By the previous part, it suffices to compute the expression on the previous line. We will begin by constructing a table that enumerates $\delta_k(s,v)$ for every $k \in \{1,\dots,n\}$ and each vertex $v \in V$. This operation can be performed in $O(|V|(|E|+|V|))$ time by generating a $|V|\times|V|$ table, where the entry at the kth row and vth column corresponds to $\delta_k(s,v)$. To calculate a specific entry, we inspect a number of entries in the preceding row equal to the in-degree of the vertex in question. Hence, aggregating the computations needed for each row necessitates $O(|E|+|V|)$ time. This overall runtime can be optimized to $O(|V||E|)$ if we omit any isolated vertices from the table, ensuring that $E \in \Omega(|V|)$. Consequently, $O(|V|(|E|+|V|))$ is reduced to $O(|V||E|)$. After this table is computed, we can simplify each row by subtracting it from the last row and dividing each entry by $n-k$. Then, we identify the minimum value in each column and select the maximum of these minimum values.

2. Arbitrage

Solution:

a. To do this, we take the negative of the natural log (or any other base will also work) of all the values c_i that are on the edges between the currencies. Then, we detect the presence or absence of a negative weight cycle by applying the Bellman-Ford algorithm. To understand that the existence of an arbitrage situation is equivalent to the existence of a negative weight cycle in the original graph, consider the following sequence of steps:

$$\begin{aligned} R[i_1,i_2]\cdot R[i_2,i_3]\cdots R[i_k,i_1] &> 1 \\ \ln(R[i_1,i_2]) + \ln(R[i_2,i_3]) + \cdots + \ln(R[i_k,i_1]) &> 0 \\ -\ln(R[i_1,i_2]) - \ln(R[i_2,i_3]) - \cdots - \ln(R[i_k,i_1]) &< 0 \end{aligned}$$

b. To perform this task, we first apply the same modification to all edge weights as described in part a of this problem. Next, to detect a negative weight cycle, we relax all the edges $|V|-1$ times, following the procedure of the Bellman-Ford algorithm. Subsequently, we record the distance values (d values) of all vertices. Afterward, we relax all the edges an additional $|V|$ times. We then verify which vertices

experienced a decrease in their d values since we recorded them. These vertices are part of one or more potentially disjoint sets of negative weight cycles, denoted as set S.

To identify a specific cycle within this set, we select any vertex in S and greedily continue to choose any adjacent vertex also within S, vigilantly watching for any repetition. This method ensures the identification of a cycle, as we never reach a dead end due to the nature of S, where every vertex is guaranteed to be part of some cycle, thereby possessing an out-degree of at least one.

3. Algorithmic consulting

Solution:

a. Suppose, to the contrary, that there exists some $J_i \in T$, and some $A_k \in R_i$ such that $A_k \notin T$. By the definition of the flow network, there exists an edge of infinite capacity from A_k to J_i because $A_k \in R_i$. Consequently, this entails that an edge with infinite capacity crosses the specified cut. This scenario implies that the capacity of the cut is infinite, which contradicts the given fact that the cut has finite capacity.

b. Though it may seem intuitive, it is insufficient to merely consider the experts located on the source side of the cut. To illustrate this, imagine a scenario involving a specialized role, such as a "Computer power switch operator," essential for every job. Consequently, any finite cut that includes any job completion would necessitate hiring this expert. Moreover, given that this expert has infinite capacity edges connecting to every other job, all other required experts for these jobs would also need to be employed. Therefore, if this ubiquitous employee is needed, any minimum cut would either include all or none, but it is straightforward to identify a counterexample where this is not the optimal solution.

For the problem to be effectively resolved, it must be assumed that every hired expert completes all the jobs they are required for. Under this assumption, define $S_k \subseteq [n]$ as the indices of the experts on the source side of the cut, and $S_i \subseteq [m]$ as the indices of jobs on the source side of the cut. The net revenue can then be expressed as:

$$\sum_{i \in S_i} p_i - \sum_{k \in S_k} c_k \tag{6}$$

This configuration ensures that moving any set of experts and tasks from the sink side to the source side results in a net decrease in cut capacity by the cost of those experts, while simultaneously increasing by the revenue of those jobs. If the cut was minimal, such a shift would represent a positive change, indicating that the revenue generated does not suffice to justify the hiring costs. This demonstrates that the jobs included on the source side in the minimal cut are precisely those that should be targeted.

c. Again, to arrive at a solution, we must assume that for every expert hired, all requisite jobs for that expert are completed. This entails running the relabel-to-front algorithm, which operates in $O(V^3)$ time as described in section 26.5 on the flow network, and employing the experts on the source side of the cut. As indicated by the previous discussion, this strategy guarantees the optimal outcome.

The flow network comprises $m+n+r$ edges and $2+m+n$ vertices, yielding a runtime of $O(((2+m+n)^3))$, thus the complexity is cubic in terms of $\max(m, n)$. It's important to note that this algorithm does not depend on R, which is reasonable given the intrinsic constraint $r < mn$; a factor considered to be of lower order. Absent this assumption, it is doubtful that an efficient solution exists, although the specific NP-complete problem applicable for a reduction remains uncertain.

4. Augmenting paths

Solution:

The way to find out recreating a series of augmenting paths which has the maximum number of augmenting paths, can be consider the worst case of the worst case of Algorithm for Maximum Flow. Recall Ford-Fulkerson Algorithm for Maximum Flow (It is shown below).

Algorithm 1 Ford-Fulkerson Algorithm for Maximum Flow

Inputs: Network $G = (V, E)$ with flow capacity c, a source node s, and a sink node t
Output: Compute a flow f from s to t of maximum value
$f(u, v) \leftarrow 0$ for all edges (u, v)
while there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$ **do**
    Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
    for each edge $(u, v) \in p$ **do**
        $f(u, v) \leftarrow f(u, v) + c_f(p)$ ▷ Send flow along the path
        $f(v, u) \leftarrow f(v, u) - c_f(p)$ ▷ The flow might be "returned" later
    end for
end while

Time complexity of the above algorithm is $O(f * E)$. We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes $O(f * E)$.Now let's look backwards. The idea is to set the flows of all edges to 0 one by one. It will take $|E|$ iterations to do so. Once all edges are set to zero it becomes clear that the highest number of augmenting paths can not exceed the number of edges that are 0. And since there are $|E|$ such edges the maximum number of possible paths is $|E|$ as well.

5. Formulating LPs

Solution:

(a) We will follow a similar approach to the method used when finding the shortest path between two specific vertices.

$$\begin{aligned} \text{maximize} \quad & \sum_{v \in V} d_v \\ \text{subject to} \quad & d_y \leq d_u + w(u,v) \quad \text{for each edge } (u,v) \\ & d_s = 0. \end{aligned} \tag{7}$$

The first type of constraint ensures that we never designate a vertex as being further away than it would be if we took the edge corresponding to that constraint directly. Moreover, by aiming to maximize all of the variables, we eliminate any slack, thereby ensuring that all d_v values represent the lengths of the shortest paths to each vertex v. This is achieved because the only factor limiting the variables is the requirement to relax along the edges, which fundamentally determines the shortest paths.

(b) We can solve the maximum-bipartite-matching problem by viewing it as a network flow problem, where we append a source s and a sink t, each connected to every vertex in sets L (left vertices) and R (right vertices) respectively by an edge with capacity 1. Additionally, every edge that already exists in the bipartite graph is also assigned a capacity of 1. Integral maximum flows thus correspond directly to maximum bipartite matchings. In this setup, the linear programming problem to be solved is as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{v \in L} f_{sv} \\ \text{subject to} \quad & f_{(u,v)} \leq 1 \text{ for each } u, v \in \{s\} \cup L \cup R \cup \{t\} = V, \\ & \sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \text{ for each } uinL \cup R, \\ & f_{uv} \geq 0 \text{ for each } u, v \in V. \end{aligned} \tag{8}$$

This formulation ensures that the flow from the source to each vertex in L does not exceed the capacity, and the flow into each vertex is equal to the flow out, which is critical for ensuring that the system is balanced and obeys the laws of flow conservation.