CS535 Design and Analysis of Algorithms
Fall 2019
HomeWork 1 Sample Solutions

1. (a) Finding a sequence of currencies such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

is equivalent to find a sequence of currencies such that

$$\lg \frac{1}{R[i_1, i_2]} + \lg \frac{1}{R[i_2, i_3]} + \cdots + \lg \frac{1}{R[i_{k-1}, i_k]} + \lg \frac{1}{R[i_k, i_1]} < 0$$

By this means, the original problem can be reduced into the problem of detecting existence of negative cycles. We first build the graph G with each currency as vertex and $\lg \frac{1}{R[i_j, i_k]}$ as the weight of edge between v_j and v_k.

Add a new node t and connect all nodes to t with 0 cost edges. Then we can simply apply Bellman-Ford algorithm over the graph G to detect whether there exist a negative cycle, if so, then there exists such a sequence of currencies; otherwise, no such a sequence of currencies could be found.

It takes $O(n^2)$ time to construct the graph, then $O(nm)$ to run Bellman-Ford algorithm. Thus, the total time is $O(n^3)$, since $m = O(n^2)$.

(b) By solving problem (a), we can determine whether such a sequence exists in $O(n^3)$ time. In (a), when applying the Bellman-Ford algorithm, we record each vertex's successor (referring Lec. 1 P. 20.) Since if there exist a negative cycle, then running the $(n+1)^{th}$ iteration, some vertices' d value will change. We randomly choose one vertex u whose d value changes in $(n+1)^{th}$ iteration, and if we repeatedly follow the vertices' successor, we can get back to vertex u. We can use a recursive method to print the path vertex until we reach to vertex u (referring to algorithm PRINT-PATH in section 22.2 in CLRS P. 601). We only need $O(n)$ time to print the vertices of the cycle. So the total running time is $O(n^3)$.

2. The problem can be reduced into shortest path between two determined source and target vertex. We first build the graph G as follows:

Each interval $[x_i, y_i]$ is treated as a vertex i in graph G. For any two intervals $[x_i, y_i]$ and $[x_j, y_j]$, if they overlap (*i.e.* $x_i \leq x_j \leq y_i \leq y_j$), then we add an edge between vertex i and j and its weight is $c_i + c_j$. By this means, we build the graph G. Then we find all intervals covering x_0, meanwhile get the corresponding vertices as source set S; Similarly, we nd all intervals covering y_0 and get the corresponding vertices as target set T. The we add a source vertex s connecting to each vertex v_i in source set S, and weight the corresponding edge c_i; we also add a target vertex t connecting to each vertex v_j in target set T, and weight the corresponding edge c_j.

Then to find a subset of intervals with minimum total cost which together still cover the interval $[x_0, y_0]$, we can solve it by finding shortest path between vertex s and t. Since the cost for any interval is positive, thus we can use Dijkstra's algorithm to calculate the shortest path. The actual minimum total cost should be the length of the shortest path divided by 2, since we consider each interval's cost twice.

3. (a) Note that, set F contains edges in all shortest $s-t$ paths. One way to find F is to run BFS twice: once in graph D from s and once in $D'(V, A')$ from t, where A' is the set of edges in A with reversed direction. In each BFS, label each vertex with its level; each vertex appears in some shortest path should have the same sum of the two levels labeled on it, and this sum, say k, equals to the level of t in the first BFS. Our algorithm outputs those edges between adjacent levels during the second BFS (note that, the direction of each edge needs to be the same as in A) such that both endpoints have a label-sum equals to k. To create D' and to run BFS both need time $O(|V|+|A|)$, so the algorithm has running time $O(|V|+|A|)$.

   (b) A simple algorithm can be running DFS on $G(V, F)$, where F is the set of edges output from question (a). Note that, G is a dag graph from s to t and every $s-t$ path in G is a shortest $s-t$ path in $D(V, A)$. We run a slightly updated DFS on G from s: we never mark t to black; and each time we reach an end (either a black vertex or t) we go all the way back to s and we mark each interior vertex on this path to black at the same time. Specially, when the end of a path is t, we add this $s-t$ path to the output set on the way back to s. The algorithm will cease because all neighbors of s will be marked black. The running time is $O(|V|+|F|) \leq O(|V|+|A|)$ because that each vertex other than s and t can be marked black once, thus each edge in F can be visited at most twice (once from s, once back to s).

4. Assume circuit C contains n ordered vertex $v_1, v_2, \cdots, v_n$, edge between v_i and v_i+1 is notated as a_i. Then according to the definition, we get n following equations:

$$\begin{aligned} l^*(a_1) &= l(a_1) + p(v_1) - p(v_2) \\ l^*(a_2) &= l(a_2) + p(v_2) - p(v_3) \\ &\cdots \\ l^*(a_n) &= l(a_n) + p(v_n) - p(v_1) \end{aligned}$$

   Considering the fact that circuit C is 0-length, $l(a_1) + l(a_2) + \cdots + l(a_n) = 0$, which implies

$$l^*(a_1) + l^*(a_2) + \cdots + l^*(a_n) = 0$$

   The reweighted length has the property that $\forall i$, $l^*(a_i) \geq 0$, therefore

$$l^*(a) = 0, \forall a \in C$$

5. We can reduce the original problem into a *Longest Path* problem. First, we build a digraph G as follows: We sort all the points in the plane by x-coordinates, and denote ordered points as $v_1, v_2, ..., v_n$ (if any two points have the same y-coordinates, order them from top to bottom). So we get an ordering for all points $\sigma = v_1, v_2, \cdots, v_n$.

Each point in the plane will be a vertex in G, and for any two vertices v_i and v_j, an edge is constructed **iff** in the plane the corresponding two points' Euclidean distance is greater than one. The direction of the edge points from left point to the right one following the σ order (*i.e.*, if $i < j$, then we build edge v_iv_j instead of edge v_jv_i). Edge v_iv_j is then weighted as $w(v_i) + w(v_j)$. Because the height of the plane is $\sqrt{3}/2$, so if any two points have the same y-coordinates, there will be no edge between them. We add a source vertex s and a sink vertex t to graph G, where s is connected to every 0 in-degree vertex v_i, and the weight of edge (s, v_i) is $w(v_i)$, every 0 out-degree vertex v_j is connected to t, and the weight of edge (v_j, t) is $w(v_j)$.
Now to find a well-separated subset U of V with the largest total weight, is no more than finding the longest path between vertex s and t. The longest path in a dag graph can be found in linear time.

Algorithm 1 Longest-Path(G)

Input: Weighted DAG G
Output: Longest path cost $d[z]$ and subset U

```
1: Topologically sort vertices in G
2: for v ∈ V do
3:    d[v] ← −∞
4:    successor[v] ← ∅
5: end for
6: d[s] = 0
7: for v ∈ V in topological order do
8:    if d[u] < d[v] + l(vu) then
9:       d[u] = d[v] + l(vu)
10:      successor[u] = v
11:   end if
12: end for
13: z ← argmax_{z∈V}{d[z]}
14: U ← following succsssor[z] to get all vertices back to s
15: return U and d[z]
```

The actual largest total weight should be $d[z]/2$, as we considered each vertex's weight twice.
The proof of correctness can be found in Chapter 24.2 with a replacing of the shortest with the longest. The running time is $O(|V| \log |V|)$ for sorting vertices by x-coordinates, $O(|V|^2)$ for dag graph construction, and $O(|V| + |E|)$ for topological sort and the relaxing operations. So the total running time will be $O(|V|^2)$.