ANLY-550 Homework 2

Yi Li, Feb. 14

1. Prove that if a graph G is undirected, the any DFS of G will never encounter a cross edge.

Proof: Assume there is a cross edge (u, v), then u is the ancestor of v. As the graph G is undirected, v is also the ancestor of u, so (u, v) is a back edge (in the DFS tree). So, there is no cross edge in undirected graph.

2.1 Give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list.

Solution:

Step1: Build a min-heap with the smallest element from each list. (So initially, there are k elements in this heap.)

Step2: Delete-min on the heap to find and delete the smallest element.

Step3: Insert the next element from the list of the deleted one to the heap.

Repeat step2 and step3 until the heap is empty.

Proof: We get the smallest element from all lists after the first run. Suppose we get m sorted elements after the mth run ($m < n$). Step3 makes sure that the new inserted element is no smaller than the deleted elements. After the $(m + 1)$th run, we get the smallest element e from the remaining $(n - m)$ elements from the step2, and e is no smaller than the m sorted elements, so we get $(m + 1)$ sorted elements.

Analysis: It takes $O(k)$ to build the heap; for every element, it takes $O(\log k)$ to delete-min and $O(\log k)$ to insert the next element. In total it takes $O(k + n \log k) = O(n \log k)$.

2.2 Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.

Solution:

Step1: Build a min-heap with first k elements. (So initially, there are k elements in this heap.)

Step2: Delete-min on the heap to find and delete the smallest element.

Step3: Insert the next element from the remaining elements in the list to the heap.

Repeat step2 and step3 until the heap is empty.

Proof: As each number in the list is less than k positions from its actual place in the sorted order. It is true that we get the smallest element after the first run. Suppose we get m sorted elements after the mth run ($m < n$). Step3 makes sure that all the k elements in the heap are $k - close$ to sorted, so the new inserted element is no smaller than the deleted elements. So after the $(m + 1)$th run, we get the smallest element e from the remaining elements, and e is no smaller than the m sorted elements, so we get $(m + 1)$ sorted elements.

Analysis: It takes $O(k)$ to build the heap; for every element, it takes $O(\log k)$ to delete-min and $O(\log k)$ to insert the next element. In total it takes $O(k + n \log k) = O(n \log k)$.

3. Design an efficient algorithm to find the longest path in a directed acyclic graph. (including negative weighted edges)

We can negate the edge weights and run the DAG shortest-path algorithm.

Step1: Negate all the edge weights.

Step2: Topologically sort the DAG.

Step3: Assign a zero distance to some source node s ($dist(s) = 0$), and an infinite distance to other vertices ($dist(v) = \infty$).

Step4: For each vertex v in sorted order, for each outgoing edge (v, u), if $dist(v) + weight(v, u) < dist(u)$, set $dist(u) = dist(v) + weight(v, u)$ and the predecessor of u to v.

Step5: Repeat step3 and step4 to take each vertex in G as the source node to find the longest path in the DAG.

Correctness: Step2 makes sure that vertices with no incoming edges are first and vertices with only incoming edges are last. Step3 and step4 follow the Bellman-Ford's algorithm to find the single-source shortest-path. As we negate all the edge weights in step1, we actually find the single-source longest-path. To find the longest path in the DAG, step5 repeat $|V|$ times to take each vertex as the source node, and finally output the largest single-source longest-path.

Analysis: The topological sort takes $O(|V| + |E|)$. Step3 takes $O(|V|)$. Step4 takes $O(|V| * |E|)$. Step5 repeats step3 and step4 for $|V|$ times. So totally, it takes $O(|V|^2 + |V|^2|E|)$ time.

4. Traverse all the streets of Sunnydale in such a way that she would walk on each side of each block exactly one.

The layout of Sunnydale should satisfy that each street has two sides (to walk on each side once, one needs to walk on the same street twice: forward and backward). We consider the layout as an undirected graph, that each street is an edge, and each crossroad or an end of a street is a vertex. To walk through each edge of the graph exactly twice, we can use a variant DFS as following:

Step1: Start from a source node s, walk through one of its edges to another node v, and mark this edge as visited.

Step2: If v has new edges, choose one edge to walk through, and mark this edge as visited.

Step3: If v does not have new edges, go back to the previous nodes one by one, until find a node u that has at least one new edge has not been visited, and walk through the new edge, also mark this edge as visited. (If one cannot not find such a node u, one must follow the previous nodes and return back to the source node s, and the task ends.)

Repeat step2 and step3 until all edges have been visited and return to s.

Proof: Suppose there is an edge (u, v) has not been visited. As the DFS algorithm makes sure every node has been visited, so if the node u is visited, (u, v) must can be visited either by step2 or by step3. As one always needs to go back to the previous nodes, each edge is been visited only twice.

5. Design an efficient algorithm to find the paths from a source to every other node such that each path has the smallest possible bottleneck.

We can change the Dijkstra's algorithm as following:

Set an array $dist[1..n]$ that $dist[u]$ stores the length of the bottleneck edge on a bottleneck path from s to u.

Set an array $prev[1..n]$ that $prev[u]$ stores the predecessor of u on the bottleneck path from s to u.

Initialization: For the source s, set $dist(s) = 0$. For all other vertices, set $dist(v) = \infty, prev[v] = Null$.

For each (u, v) in E: If $max(dist[u], length(u, v)) < dist[v]$, set $dist[v] = max(dist[u], length(u, v)), prev[v] = u$.

Correctness: For every vertex v, if u is the predecessor of v in a bottleneck path from s to v, the bottleneck edge from s to v is either the bottleneck edge on the path from s to u or the edge (u, v), whichever is larger.

Analysis: As we just change what stores in the $dist[v]$ from the Dijkstra's algorithm, the time is as the same as $O((n + m)logn)$ in the Dijkstra's algorithm.

6. Consider the shortest paths problem in the special case where all edge costs are non-negative integers. Describe a modification of Dijkstra's algorithm that works in time $O(|E|+M|V|)$, where M is the maximum cost of any edge in the graph.

The maximum cost of any shortest path from the source s is at most $M(|V| - 1)$.

Set $M(|V| - 1) + 1$ doubly-linked lists, each doubly-linked list $L[i]$ keeps track of all vertices that are currently at distance i from the source s. $L[0] = s$.

Set an array $dist[1..n]$ that $dist[u]$ stores the shortest distance (so far) from the source s to vertex u.

Initialization: For the source s, set $dist(s) = 0$. For all other vertices, set $dist(v) = \infty$.

Loop: For each $L[i]$ (start from $L[0]$ to $L[M(|V|-1)]$), for each edge (u, v), if $dist(v) > dist(u)+length(u, v)$, set $dist(v) = dist(u) + length(u, v)$, and push v to $L[dist[v]]$.

Correctness: This algorithm is changed from the Dijkstra's algorithm. It uses doubly-linked lists instead of a heap to store the data.

Analysis: Each edge and vertex of the graph will be visited only once, and the outer loop runs in time $O(M|V|)$. So totally, the running time is $O(|E| + M|V|)$.

7. Design an efficient algorithm to detect if a risk-free currency exchange exists.

We can consider this problem as a directed graph G(V, E) that each currency c_i is a vertex and each exchange rate $r_{i,j}$ is an edge. We can set the weight of each edge as $(-\lg r_{i,j})$. And then use the Bellman-Ford's algorithm to detect whether the graph exists a negative cycle.

Proof: As

$$r_{i_1,i_2} * r_{i_2,i_3} * \cdots * r_{i_k-1,i_k} * r_{i_k,i_1} > 1$$

can be transferred to

$$(-\lg r_{i_1,i_2}) + (-\lg r_{i_2,i_3}) + \cdots + (-\lg r_{i_{k-1},i_k}) + (-\lg r_{i_k,i_1}) < 0,$$

we can use the Bellman-Ford's algorithm to detect negative cycles.

Analysis: It takes the same time as the Bellman-Ford algorithm, that is, $O(|V| * |E|)$.