CS 170 Efficient Algorithms and Intractable Problems
Spring 2017 Prasad Raghavendra and Sanjam Garg

Final Review

May 6, 2017

1. DAG Revisited

Design an algorithm that takes a directed acyclic graph, $G = (V, E)$, and determines whether G contains a directed path that touches every vertex exactly once. The algorithm should run in $O(|V| + |E|)$ time.

Solution: Linearize the DAG. The answer to the question is YES iff the DAG has an edge $(i, i+1)$ for every pair of consecutive vertices labelled i and $i+1$ in the linearized order.

Reason: The edges in the linearized order can only go in the increasing direction in the linearized order, and the required path must touch all of the vertices.

Complexity: Both linearization and checking outgoing edges from every vertex takes linear time.

2. Another MST Algorithm

In this problem, we will develop a new algorithm for finding minimum spanning trees. It is based upon the following property:

Pick any cycle in the graph, and let e be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain e.

(a) Prove this property carefully.

(b) Here is the new MST algorithm. The input is an undirected graph $G = (V, E)$ with edge weights $\{w_e\}$.
In decreasing order of edge weights, if edge e is part of a cycle, remove it from G. When you cannot remove any more edges, return G.
Prove that this algorithm is correct.

(c) What is the overall time taken by this algorithm, in terms of $|E|$?

Solution:

(a) Consider an MST T which contains e. Removing e breaks the tree into two connected components, say S and $V \setminus S$. Since all the vertices of the cycle cannot still be connected after removing e, at least one edge, say e' in the cycle must cross from S to $V \setminus S$. However, then replacing e by e' gives a tree T' such that $\text{cost}(T') \leq \text{cost}(T)$. Since T is an MST, T' is also an MST which does not contain e.

(b) If e is the heaviest edge in some cycle of G, then there is some MST T not containing e. However, then T is also an MST of $G - e$ and so we can simply search for an MST of $G - e$. At every step, the algorithm creates a new graph $(G - e)$ such that an MST of the new graph is also an MST of the old graph (G). Hence the output of the algorithm (when the new graph becomes a tree) is an MST of G.

(c) The time for sorting is $O(ElogE)$ and checking for a cycle at every step takes $O(E)$ time. Finally, we remove $E - V + 1$ edges and hence the running time is $O(ElogE + (E - V)E) = O(E^2)$.

3. Minimal Graphs

(a) A city has n intersections, and m undirected roads. However, road maintenence is getting really expensive, so the city would like to reduce the number of roads. They want to do this without affecting the overall connectivity of the intersections.

More formally, you are given as input a graph $G = (V, E)$, with edges undirected. You would like to create a new graph $G' = (V, E')$, with edges undirected and $|E'|$ minimized with the following constraint that if there exists a path from u to v in G, then there exists a path from u to v in G'.

Give an efficient algorithm to determine the minimum number of undirected edges we need in the modified graph. Note that if there isn't a path from u to v in G, then there may or may not be a path from u to v in G'.

(b) Repeat the same excercise, but now for directed graphs. More specifically, you are given as input a graph $G = (V, E)$ with edges **directed**. You would like to create a new graph $G' = (V, E')$ with edges **directed** and $|E'|$ minimized with the following constraint that if there is a path from u to v in G, then there must be a path from u to v in G'. (Note that we don't necessarily need a path from v to u).

Give an efficient algorithm to determine the minimum number of directed edges we need in the modified graph. Note again that if there isn't a path from u to v in G, then there may or may not be a path from u to v in G'.

Solution:

(a) Let k be the number of connected components of G (found using DFS). Then, the answer is $n - k$.

Each edge at the very best would merge two different connected components. Thus, we start with n components initially (isolated vertices), and when we add edges in one by one, we will decrease the number of components by at most 1 at each step. At the end, we are left with k components, thus we need at least $n - k$ edges.

We can check that this is indeed achievable. Doing a DFS from one particular node will find all edges in that component. G' can just include all these edges in the DFS tree. A tree on c vertices has $c - 1$ edges. Thus, let c_i be the number of nodes in component i. Then, the total number of edges is $\sum -i = 1^k (c_i - 1) = (\sum_{i=1}^{k} c_i) - k = n - k$.

The runtime is linear since a DFS takes linear time.

(b) *Main Idea:* Let's define the notion of a "weakly connected component" (abbreviated as wcc) as the connected components in the graph when we think of the edges as undirected. Let a be the number of wcc's in G, and let b the number of wcc's in G that have a cycle. Then, the answer is $n - a + b$.

Pseudocode:

Algorithm MinimialGraphDirected(G):
1. Let G' be undirected version of G (i.e. add all edges from G, but also add reverse edges into G').
2. Find the number of connected components in G' (using a DFS). Record this number as a
3. For each connected component in G', run a DFS in G to check if there's a cycle (there exists a back edge). Record the number of connected components with a cycle number as b.
4. Return $n - a + b$

Correctness: Let's suppose we only had a single weakly connected component. Suppose that this wcc had c nodes. There are two cases

- Suppose that this wcc did not have a cycle. Then, we can topologically sort the vertices, and connect them in that order with $c - 1$ edges and satisfy all the conditions. We definitely need at least this many edges (since if we had fewer, the resulting graph wouldn't be weakly connected).

- Suppose that this wcc had at least one cycle. Here, we can't do a topological sort, $c-1$ edges (since if we could, we would have a tree, which can be topologically sorted). However, we can definitely do c edges, since we can connect all the vertices in a cycle.

Now, we've solved this problem for each wcc. Now, it's clear that we don't need any edges between any two different weakly connected component. Thus, we've solved the problem overall on any general graph. The quantity $a-b$ is precisely the number of wcc's with no cycles, thus we have shown that $n-(a-b) = n-a+b$ is indeed the minimum number of edges required.

Note: A common idea is to decompose the input graph into a DAG on SCCs, then to put a cycle on each SCC and then take a spanning forest from the DAG structure. This algorithms uses too many edges.

4. Revisiting zero-sum games

Suppose we have a zero-sum game with payoff matrix M. The row player tries to maximize the expected payoff whereas the column player tries to minimize it. Under the following conditions, indicate (YES or NO) whether entry $m_{i,j]}$ is the optimal expected payoff value in general:

Condition	Is $m_{i,j}$ optimal?
$m_{i,j}$ is **smallest** in row i and **smallest** in column j	
$m_{i,j}$ is **largest** in row i and **smallest** in column j	
$m_{i,j}$ is **smallest** in row i and **largest** in column j	
$m_{i,j}$ is **largest** in row i and **largest** in column j	

Solution:

Optimal only when $m_{i,j}$ is smallest in row and largest in column. Row player plays i showing that its expected payoff is at least $m_{i,j}$, column player plays j showing that its expected payoff is at most $m_{i,j}$. Since optimal expected payoff is the same for both players, it is exactly $m_{i,j}$.

5. Assigning workers

Assume we have N workers. Each worker is assigned to work at one of M factories. For each of the M factories, they will produce a different profit depending on how many workers are assigned to that factory. We will denote the profits of factory i with j workers by P_i^j.

(a) How would you find the assignment of workers that produces the most profit?

Solution:

We will use dynamic programming where $B(m,j)$ denotes the best cost of the best assignment of j workers to the first m factories. An optimal solution of j workers to the first m factories assigns some number of workers, k, to factory m, and must assign $j-k$ workers with maximal profit to the first $m-1$ factories, which, by induction, has cost $B(m-1,j-k)$. Thus, we have

$$B(m,j) = \max_{0\le k\le j} P_m^k + B(m-1,j-k)$$

Furthermore, the base case is $B(0,0)=0$; i.e. the profit of assigning no worker to no factory is 0. The time to compute each value $B(m,j)$ is $O(N)$ and since there are MN subproblems, so the total runtime is $O(MN^2)$. To recover the assignment, one backtracks from $m=M$ and $j=N$ and assigns k workers to machine m if $B(m,j) = B(m-1,j-k)+P_m^k$ and continues to backtrack from $m-1, j-k$, etc.

(b) How can this algorithm be improved if we enforce the law of diminishing returns? Namely that for each factory each additional worker (past the first worker) cannot result in a larger increase in profits than the previous worker. (For example, $P_1^1 = 5$ and $P_1^2 = 7$ obeys diminishing returns as the first worker adds profit 5 and the second adds profit 2.)

Solution:

The solution is to assign the N workers greedily according to the largest available increase in profit: that is, for a factory i with j workers the increase in profit is $P_i^{j+1} - P_i^j$. (We assume $P_i^0 = 0$.) Consider the first point where an optimal solution, S, differs from this algorithms solution A. That is, A assigns a worker to position j for factory i and S assigns no worker here. S must assign some worker to a position j' in factory i' where A assigns no worker. The profit increase for factory i' at position j' is less than profit increase for i' for the current number of workers assigned to i' by the diminishing return condition, which, in turn, is at most the profit increase for worker j on factory i since the algorithm producing A has both choices available to it and it chooses the largest. Thus, we can move a worker assigned to factory i' in S to factory i and produce an improved solution that is consistent with greedy for at least one more step. By induction, we can conclude that repeatedly choosing the largest increase in profit produces an optimal solution.

Implementation:

An efficient implementation of this algorithm uses a max-heap, with an entry for each factory i with initial key P_i^1. Then we repeatedly assign each worker to the factory i with the maximum key. When a is assigned to factory i, we re-insert factory i into the queue with key $P_i^{j+1} - P_i^j$ where j is the number of workers currently assigned to factory i. The time for each extract-max and insert is $O(\log M)$, thus the total time to implement the algorithm is $O((N+M)\log M)$ as there are at most $(N+M)$ inserts and N extract-maxs.

Comment: A faulty argument is that that since greedy chooses the maximal profit that it must be optimal. The problem with this argument is that it does not use the diminishing returns condition explicitly. One needs to use diminishing returns to argue that any future worker will have a profit increase that is at most the current best option. Without diminishing returns, one could find that by scheduling a worker who adds a very low profit increase may provide a later opportunity to assign a very high profit worker. Indeed, this is why the greedy algorithm fails for part (a).

6. Assigning backups

Horizon Wireless is building a 5G network. The company has a set V of wireless towers; the distance $d(i, j)$ between any two of them is known, and each tower is capable of transmitting to other towers within a distance r.

- To make this network fault-tolerant, Horizon wants to assign each tower $v \in V$ to **two** other backup towers, so that if v is about to fail, it can transmit its data to them.
- Due to storage constraints, each tower can only serve as a backup for up to **three** other towers.

(a) Suppose Horizon partitions its towers V into active towers A and backup towers B. Devise an algorithm which, given $V = A \cup B$, a distance function $d(i, j)$ on V, and tower radius r, assigns each active tower $a \in A$ to two backup towers in B (subject to the storage constraint), or reports that no assignment is possible. (*Hint: build a graph and use a known algorithm.*

Solution:

We want to reduce the problem to max flow. Create a directed graph with vertex set $\{s,t\} \cup V$. Create an edge from s to every vertex $a \in A$ with capacity 2. Create an edge from each $b \in B$ to t with capacity 3. Finally, for every pair of vertices (a,b) with $a \in A$, $b \in B$, create an edge from a to b with capacity 1

if and only if $d(a,b) \leq r$. The existence of an assignment is equivalent to the existence of a flow with value at least $2|A|$.

If there is a flow with value $2|A|$, then it must saturate all of the edges into each of the vertices $|V|$. We know from class that there is an integer-valued flow with this value. In particular, on every edge from a vertex in A to a vertex of B, the flow value is either 0 or 1. Therefore, this flow has exactly two outgoing edges from every vertex in A with nonzero flow. Furthermore, this flow assigns at most 3 active towers to any backup tower. Therefore, there is a valid assignment.

Conversely, suppose that there is an assignment of active towers to backup towers. Make a flow in the network we created by putting a flow of 1 on each edge corresponding to assigned pair (a,b) and 0 for all other pairs $(a,b), a \in A, b \in B$. For all incoming edges to A, assign 2 units of flow. For all outgoing edges from $b \in B$, assign the right flow to achieve flow conservation. This is an $s-t$ flow with value $2|A|$ because flow conservation is achieved at all vertices in A. This completes the proof of correctness.

Note that you didn't need to give this justification to receive full credit.

(b) To use its network more efficiently, Horizon wants to use all towers in V as active towers, but still wants to assign each tower $v \in V$ to two other towers in V as backups. Again, no tower can be a backup for more than three other towers. Give an algorithm to find the assignment of backups for every tower.

Solution:

We split every vertex $v \in V$ into two vertices, v_a and v_b. We then run the algorithm from the previous part, passing in $\{v_a\}$ as A and $\{v_b\}$ as B.