Introduction to Algorithms November 29, 2003
Massachusetts Institute of Technology 6.046J/18.410J
Professors Shafi Goldwasser and Silvio Micali Handout 27

Problem Set 7 Solutions

Problem 7-1. Shortest Paths

Consider a given directed weighted graph $G = (V, E)$ in which all edge weights are positive. Suppose that you have already computed a distance matrix D, where $D_{i,j} = d(i, j)$ is the length of the shortest path from node i to node j.

(a) Give an $O(1)$ algorithm that, on input s, u, v, t, finds the length of the shortest path from s to t that passes through both u and v.

Solution: Return $\min(d(s, u)+d(u, v)+d(v, t), d(s, v)+d(v, u)+d(u, t))$. Running time is obviously $O(1)$, as we need only look up six values in the matrix D. For correctness, any s-t path that passes through nodes x and y in that order must consist of an s-x path, a x-y path, and a y-t path; the shortest s-t path passing through x and y must consist of the shortest s-x path, etc. (This is just the fundamental fact about shortest paths discussed in lecture.) For our problem, the path can pass through u and v in either order, so the shortest such path is just the shorter of the shortest s-u-v-t path and the shortest s-v-u-t path.

(b) Give an $O(n)$ algorithm that, on input s and t, outputs a list of all vertices $v \in V$ such that v is on *some* shortest path from s to t.

Solution: The key observation is the following: a node v lies on a shortest pash from s to t if and only if $d(s, v) + d(v, t) = d(s, t)$. This is another application of the fundamental fact about shortest paths. Obviously $d(s, v) + d(v, t) \geq d(s, t)$—this is just the triangle inequality—since to be otherwise would contradict the optimality of the purported shortest s-t path. If $d(s, v) + d(v, t) = d(s, t)$, then v is on a shortest s-t path since going from s to t via v is as short as any way of going from s to t; the converse directly follows from the fundamental fact.

Thus the algorithm is simply the following: for each node v, add v to the list if $d(s, v) + d(v, t) = d(s, t)$. Correctness follows from the above, and the running time is obviously $O(n)$.

(c) Give an $O(n \log n)$ algorithm that, on input s and t, outputs a shortest s-t path in G, i.e. your algorithm should output a list of vertices $s = v_0, v_1, \ldots, v_k = t$ such that $\sum_{i=1}^{k} d(v_{i-1}, v_i) = d(s, t)$.)

Solution: We use a divide and conquer approach. For a set S of vertices and two vertices u and v, define

FINDPATH(u, v, S)

1. If $u = v$, just return the trivial path u.
2. Using the algorithm from part (b), find a list X of all nodes $x \in S$ such that x is on some shortest path from u to v.
3. Find the median m of X.
4. Partition X into $L := \{x \in X : d(u,x) < d(u,m)\}$ and $R := \{x \in X : d(u,x) > d(u,m)\}$.
5. Return the concatenation of (1) FINDPATH(u, m, L), (2) FINDPATH(m, v, R) without the initial m.

For the given problem, we return FINDPATH(s, t, V).

First, the running time analysis, which is easy: the recurrence for the running time is $T(n) \le 2T(n/2) + n = O(n \log n)$, since L and R both have size at most half of the size of S.

For correctness, note that m is one *some* shortest u-v path by Part (b). Thus correctness follows straightforwardly by induction and the fundamental fact. Inductively, we have a shortest path from u to m and a shortest path from m to v, where m is on some shortest path from u to v. Thus putting these together yields some shortest path from u to v.

Problem 7-2. All-Pairs Shortest Paths

Give an implementation of the FLOYD-WARSHALL algorithm that uses $O(n^2)$ space.

Solution: As it appears in CLRS, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k \in \{1, 2, \ldots, n\}$. We will show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL'(W)
n ← rows[W]
D ← W
for k ← 1 to n
    do for i ← 1 to n
           do for j ← 1 to n
                  d_ij ← min(d_ij, d_ik + d_kj)
return D
```

The procedure is correct because any update it makes to the D matrix is made to an entry which will not affect any other entry in that k-loop. In other words, during the k-loop, entry d_{ij} is updated by setting $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$. This entry d_{ij} for $i, j \ne k$ is not used again until k is incremented, i.e. the next k-loop. Additionally, during this k-loop, entry d_{ik} is not changed because this would require that $d_{ik} > d_{ik} + d_{kk}$, which only occurs if there is a negative cycle, i.e. if d_{kk} is negative. Thus, there is no need to distinguish between $d_{ik}^{(k)}$ and $d_{ik}^{(k-1)}$ as done in the original algorithm.

Problem 7-3. Transitive Closure

We define a directed graph G^- to be **edge-parsimonious** if among all directed graphs with the same transitive closure as G^-, graph G^- has a minimum number of edges. (See CLR pages 632–633 for a definition of transitive closure.)

Given a directed acyclic graph G, give an $O(VE)$ algorithm that finds an edge-parsimonious graph G^- with the same transitive closure as G.

Solution: As we will prove, when G is a DAG, G^- is unique and $E^- \subseteq E$. (Both these facts do not hold for arbitrary graphs). More specifically, we will show that the edges of G^- are exactly:

$$E^- = \{(u, v) \in E \mid \text{there exists no other path } u \leadsto v \text{ in } G\}$$

Therefore, we want to remove from G any edge (u, v) that has another (longer) path from u to v, and the remaining graph will be G^-.

There are several possible $O(|V||E|)$ algorithms that do that, and we describe one of them below.

From the characterization of G^- above, we see that an edge $(u, v) \in E$ should *not* be removed if and only if the longest path from u to v is of length 1 (the edge itself). Thus, if we find the longest path between every pair of vertices, we can go over all edges in E and remove the redundant ones. In the following, we find the longest path for each pair by assigning a weight of -1 to every edge, and running ALL-PAIRS-SHORTEST-PATHS (APSP). (G is a DAG so there are no negative cycles).

FIND-PARSIMONIOUS(G)
1 For every $(u, v) \in E$ set $w(u, v) = -1$
2 $L \leftarrow APSP(G, w)$
3 $E^- \leftarrow \emptyset$
4 For every $(u, v) \in E$
5   **if** $L[u, v] = -1$ $\triangleright$ longest path from u to v is the edge itself
6   **then** $E^- \leftarrow E^- \cup \{(u, v)\}$ $\triangleright$ add the edge to E^-
7 **return** E^-

Running time: The loops in lines 1, and 4–6 take $O(|E|)$. Thus, the APSP is dominating. Note that since G is a DAG, the SINGLE-SOURCE-SHORTEST-PATH can be done in $O(|V| + |E|)$ time, using topological sort (see CLR). Thus APSP, and therefore FIND-PARSIMONIOUS, can be done in $O(|V|^2 + |V||E|)$.

Correctness

The correctness of the above algorithms relies on the following theorem, which gives the characterization of the unique edge-parsimonious graph G^- corresponding to a given DAG G.

Theorem 1*Let $G = (V, E)$ be a DAG, and let $G^- = (V, E^-)$ be any edge-parsimonious graph that has the same transitive closure as G. Then for all $u, v \in V$,*

$$(u, v) \in E^- \iff (u, v) \in E, \text{ and there is no path } u \leadsto v \text{ of length} > 1 \text{ in } G$$

Proof: We start by observing that since G and G^- have the same transitive closure, then if one of them has a path $u \leadsto v$ of length k, the other must have a path $u \leadsto v$ of length $\geq k$. Using this, we can prove both directions of the theorem.

First assume the righthandside holds. $(u, v) \in E$ implies that there exists a path $u \leadsto v$ in G^-. If this path was of length > 1 then, by the observation above, G would also contain a path of length > 1 from u to v, contradicting our assumption. Thus the path in G^- is of length 1, i.e. $(u, v) \in E^-$.

For the other direction, assume $(u, v) \in E^-$. Then there is a path $u \leadsto v$ in G of length $k \geq 1$. But if $k > 1$, using the observation above, G^- also has a longer path $u \leadsto v$ of length ≥ 1. Since G^- must be acyclic, this additional path cannot use the edge (u, v) (or else there would be a cycle on u or v). It follows that we could safely remove (u, v) from G^-, without affecting the transitive closure, since we can always get from u to v using the alternate path. This is a contradiction to the minimality of G^-. Thus, we must have $k = 1$, i.e. there is an edge $(u, v) \in E$, and no longer path exists.

Problem 7-4. Greedy Maximal Matching

Give a linear time algorithm that, on input graph $G = (V, E)$, finds a matching with size at least half that of a maximum matching.

Solution:

MAXIMAL-MATCHING(G)
1 $M \leftarrow \emptyset$
2 While there is an edge (u, v) in G so that neither u nor v is matched in M,
3 Set $M \leftarrow M \cup \{(u, v)\}$.

Note that this greedy algorithm may not produce an optimal (i.e., maximum) matching. However, we will show that the size of M is at least half the size of a maximum matching in G. Let M be the matching output by the MAXIMAL-MATCHING algorithm when run on G and let M' be a maximum matching in G. We will show that if $|M| < |M'|/2$, then there exists an edge $(u, v) \in M'$ such that u and v are unmatched in M.

Write the matching M as $(1, \pi_1), (2, \pi_2), \ldots, (k, \pi_k)$, where $|M| = k$. We call an index i *bad* if either i or π_i is matched in M'. Each edge (i, π_j) in M' makes at most two indices (i and j) bad. Thus if $|M| < |M'|/2$ there must be a good index. And if i is a good index, then both endpoints of the edge (i, π_i) are unmatched, as required.

Thus, the MAXIMAL-MATCHING algorithm terminates when $|M| \geq |M'|/2$.

Problem 7-5. Flying Friends to San Francisco

You are stuck in San Francisco and you want to fly as many friends as possible from Boston to San Francisco to celebrate your birthday tommorrow. You have a schedule F of n flights available to you today. An entry F_i in this schedule is described by five values $F_i = (s_i, t_i, d_i, a_i, x_i)$:

- s_i = starting city
- t_i = ending city
- d_i = departure time
- a_i = arrival time
- x_i = number seats available

Give an efficient algorithm that determines the maximum number of friends you can fly from Boston to San Francisco in time for your birthday party using the flights from schedule F. You may assume that all flights run precisely on time, and that transfers between flights are instant (i.e., if $a_i \leq d_j$ for flights F_i and F_j and $t_i = s_j$, then it is possible for one student to take flight F_i and then flight F_j). Be sure to argue correctness and analyze the running time of your algorithm.

Hint: You might want to transform the schedule into a max-flow instance and then solve the max-flow instance. You may use any max-flow algorithm as a black box.

Solution: To solve this problem, we will create an instance G of max-flow such that the value of the maximum flow of G is exactly the maximum possible number of students that we can get from Boston to San Francisco. Furthermore, when we solve this max-flow instance, we will be able to tell (from the solution) which flights each student should take in order to obtain the maximum. Our output will take the form of a set of flight itineraries, one for each student.

We must provide a transformation procedure that converts F, the flight schedule, into a directed graph G with capacities (a flow instance) that meets the criteria we describe. The intuition behind the transformation is to make an edge for each flight with capacity equal to the seat capacity of that flight, and connect two flights (via an edge of infinite capacity) if it is possible to tranfer from one to the other.

There are many ways to do this transformation. First we describe a clear but slightly inefficient transformation (a better one is given at the end):

- First we create an edge in G for each flight. Since there are only a limited number of seats on this flight, we set the capacity of the edge to the number of available seats. Formally, for each flight $f_i = (s_i, t_i, d_i, a_i, c_i)$ create two vertices u_i and v_i, with an edge between them. The capacity of this edge is c_i.
- We wish to connect, in G, two flights f_i and f_j if it is possible to transfer from f_i to f_j. This is possible if flight f_i arrives where flight f_j departs, and flight f_i arrives before (or exactly when) flight f_j departs. Formally, for each flight pair f_i, f_j such that $t_i = s_j$ and $a_i \leq d_j$, create the edge (v_i, u_j) with infinite capacity.

- Now we need source and sink vertices representing Boston and San Francisco. The source should connect to all flights leaving Boston, and the sink should have a connection from all flights arriving in San Francisco. So, create source node s, and infinite-capacity edges (s, u_i) for all i such that s_i = Boston. Create sink node t and infinite-capacity edges (v_i, t) for all i such that t_i = San Francisco.

The algorithm performs this transformation, then computes a max-flow on the graph G. A unit of flow on a "flight edge" (u_i, v_i) represents a student taking that flight f_i. A unit of flow on a "transfer edge" (v_i, u_j) represents a student transferring from flight f_i to flight f_j.

If we wanted the itinerary of each student, we could find a path of unit flow from the source to the sink, subtracting that unit from each edge on the path, and repeating until there is no more flow left in the graph.

Correctness. We must prove that the max-flow of G represents the way to get the most students possible from Boston to San Francisco. To show this, we need two claims: (1) Given a feasible flow with value f, it represents a legal itinerary for f students. (2) If the optimal itinerary gets q students to San Francisco, there exists a feasible flow of value q. Together, these two claims imply the correctness of the algorithm.

The first claim is clear from the construction. Given a feasible flow with value f, we just described how to generate a legal itinerary for f students. By the way the flow instance G was constructed, we know this itinerary is legal: Since all flow paths begin at s and end at t, this means all generated itineraries begin in San Francisco and end in Boston. Since no more than c_i flow paths use a flight f_i, we know that at most c_i of the itineraries use flight f_i, and hence no flight is over-booked. Finally, since the only flow connections between "flight edges" are those representing legal flight transfers, we know that each itinerary is reasonable (all transfers are possible).

The second claim can be seen as follows. Take the legal itinerary for q students, and create, for each student's trip, a path of unit flow in G corresponding to that trip. This is certainly a flow of value q, so we just have to show that it is legal. The only way it could be illegal is if it violates a capacity constraint of a "flight edge." However, since the itinerary does not put more than c_i students on any flight f_i, then no such capacities are violated.

Running time. The first step of the algorithm is the transformation. For each of the n flights, we generate two nodes of G, one "flight edge" and at most n "connection" edges (at most one for every other flight). We also generate the source and the sink, and at most $2n$ edges incident to them. Therefore generating G takes time $O(n^2)$. We can also see that G has $O(n)$ vertices and $O(n^2)$ edges.

The next step is to compute a maximum flow of G. Using Ford-Fulkerson, we can get a max-flow in time $O(|E|f^*)$, where f^* is the value of the maximum flow of G. This may not be so good, since the flow value f^* might be huge. So, let's use Edmonds-Karp, which runs in time $O(|V||E|^2)$. Since $|E| = O(n^2)$ and $|V| = O(n)$, this gives a total running time of $O(n^5)$.

We can get this down to $O(n^4)$ by using a better transformation. This transformation will only create $O(n)$ edges in G, and so using Edmonds-Karp will only take $O(n^4)$ time:

- We create vertices of the form $\langle$ city, time $\rangle$. Each vertex represents a departure or arrival (the city and the time). We create one of these vertices for each unique city/time combination among all of our flights. For example, if flight A went from Boston to New York, departed at 3pm and arrived at 4pm, there would be two nodes $\langle$ Boston, 3pm $\rangle$ and $\langle$ New York, 4pm $\rangle$. However, if some other flight leaves Boston at 3pm, we do not create an additional node.
- Now we need "flight edges" as before. For each flight $f_i = (s_i, t_i, d_i, a_i, c_i)$, create the edge $\langle s_i, d_i \rangle \rightarrow \langle t_i, a_i \rangle$, with capacity c_i.
- Allowing transfers with a linear number of edges is a bit confusing, but it works if we do it carefully. The idea is that we hook up all the vertices from the same city into a chain, in time-order. This chain will contain edges with infinite capacity. So, when a unit of flow "transfers flights", it travels down this chain until it gets to the departure time of the flight it wishes to take. Formally for each city x, let X be the set of nodes of the form $\langle x$, time $\rangle$. Sort X in increasing order by time. For each adjacent pair $\langle x, i \rangle, \langle x, i \rangle$ in this sorted list, make an edge $\langle x, i \rangle \rightarrow \langle x, i \rangle$ with infinite capacity.
- Finally, we need a source and a sink representing Boston and San Francisco. In fact, we already have them in the graph. The source is the first vertex in the Boston chain, and the sink is the last vertex of the San Francisco chain.

This construction may take different amounts of time depending on how you make sure there are no duplicate city/time nodes, and how you create the "chain" edges. However, even simple inefficient algorithms are more efficient than the max-flow step, so optimizing the running time of the transformation is not important. What is important is that we create $O(n)$ edges, which makes our max-flow computation (and therefore our total computation time) more efficient.