CSE531 Homework 5 Solutions

Problem 1 Solution

1. This problem can be solved using Dijkstra's algorithm. To do the reduction, consider a undirected graph G produced by viewing every device as a node, and for any two nodes with distance smaller than r, there is an edge of weight cr^α connecting them. Apply Dijkstra's algorithm on this graph to find a lightest path from A to B. Clearly the path found gives the cheapest route to deliver the message.

2. Let k be the largest positive integer to make p^k not smaller than p_0. To ensure that the success probability is not smaller than p_0, a message should not be forwarded more than k times in the network. Let G be the graph produced above, our goal is now find the lightest path from A to B, with a new restriction that the path should not be longer than k. This problem is the same as problem 3 in homework 3(You can find the solution on the course page). By applying the same dynamic programming approach the problem is solved.

3. Our goal now is to find the minimum number(denoted as $k(A, B)$) of vertices(other than A and B) to remove to make A and B disconnected in G. The idea is to reduce the problem to a max-flow problem. To do this, construct a directed graph G' base on G:replace every edge (u_1, u_2) in G by two directed edges, one from u_1 to u_2 and the other from u_2 to u_1, then replace every vertex v other than A and B by two vertices v_1 and v_2, and every edge that goes to v now goes to v_1, and every edge that comes outs of v now comes out of v_2, and add an edge that goes from v_1 to v_2. Set the capacity of every edge in G' as 1. Set A as the source and B as the destination. It is clear that, the max-flow value of this max-flow problem is just $k(A, B)$.

Problem 24-4 Solution

a Create an array Q of $|E| + 1$ entries, where $Q[i]$ maintains a list of all vertices with $d[\cdot]$ value i. For $u \neq s$, when initialize $d[u]$, instead of using ∞, we use $|E| + 1$.

We keep the smallest index i such that there is an element in $Q[i]$. When we need to extract an element, we only need to pick one element in $Q[i]$. When $Q[i]$ becomes empty, we find the next (smallest) value of i such that $Q[i]$ is not empty. It is not hard to see that, this index will never decrease, therefore, we never need to scan backward to find such i. So the scanning is one-way, and therefore during the whole process, $O(|E|)$ time is spent on the queue operations.

b For any edge $(u, v) \in E$, $w_1(u, v) \in \{0, 1\}$, clearly $\delta_1(s, v) \leq |E|$. Simply apply the method in a.

c $w_i(u,v)$ is simply $w_{i-1}(u,v)$ followed by a bit, namely the i-th bit of $w(u,v)$(Probably starts with some zeros). If that bit is 0, then $w_i(u,v) = 2w_{i-1}(u,v)$ If it is 1, then $w_i(u,v) = 2w_{i-1}(u,v) + 1$. Since the weight of every edge is at least twice as much as before, the shortest path between any to vertices must be at least doubled. Since $w_i(u,v) \leq 2w_{i-1}(u,v) + 1$ and a shortest path can have at most $|V| - 1$ edges, it is clear that $\delta_i(s,v) \leq \delta_{i-1}(s,v) + |V| - 1$, for any v.

d

$$\hat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v) \geq 2w_{i-1}(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v) \geq 0$$

This is from part 3 and the fact that $\delta_{i-1}(s,v) \leq \delta_{i-1}(s,u) + w_{i-1}(u,v)$.

e By simple calculation we have $\hat{w}_i(p) = w_i(p) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v)$(Some terms are canceled during the process of add up all the terms in the path). From this we see that the path that minimizes the $\hat{w}_i$ weight between u and v will also minimize w_i weight, and vice versa. By letting $u = s$ we have $\hat{\delta}_i(s,v) = \delta_i(s,v) + 2\delta_{i-1}(s,s) - 2\delta_{i-1}(s,v) = \delta_i(s,v) - 2\delta_{i-1}(s,v)$. By the result in c we have $\hat{\delta}_i(s,v) \leq |V| - 1 \leq |E|$.

f The following routine compute $\delta_i(s,v)$ from $\delta_{i-1}(s_v)$ for all $v \in V$ in $O(E)$ time.

(a) Compute all $\hat{w}_i(u,v)$ as shown in part d.

(b) Apply part a to compute $\hat{\delta}_i(s,v)$, since $\hat{\delta}_i(s,v) \leq |E|$ by part e.

(c) Compute all $\delta_i(s,v)$ from $\hat{\delta}_i(s,v)$ and $\delta_{i-1}(s_v)$ as shown in part e.

To compute $\delta(s,v)$, simply compute in order $\delta_1(s,v), \delta_2(s,v) \ldots \delta_k(s,v) = \delta(s,v)$ as shown above.

Problem 24-6 Solution

The algorithm makes modification to the relaxing approach. Sort the edges by their weights. Then we run 4 passes of the relaxing procedure. In the first and the third passes, we relax the edges by the increasing order of the weights. In the second and the fourth pass, we relax the edges by the decreasing order of the edge weights. It is not hard to see that, if there is a shortest path that is actually bitonic, this approach will certainly find out this path.

Problem 25-1 Solution

a Initialize a $|V|$ by $|V|$ matrix T as

$$T[i,j] = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

We can update T in the following way to maintain T to represent a transitive closure, such that $T[i,j] = 1$ if and only if there is a path from node i to j. When an edge

(u, v) is added to the graph, we check for every possible node pair (i, j). If $T[i, u] = 1$ and $T[v, j] = 1$, then we update $T[i, j]$ as 1. Clearly in this way T always represent a transitive closure, and the running time for each update is clearly $|V|^2$.

b Consider a graph which is a chain $v_1 \to v_2 \to \ldots v_{|V|}$. The matrix T will only have $|V|(|V| + 1)/2$ that many entries are 1. However, by adding an edge $v_{|V|} \to v_1$, every two nodes are now reachable. Therefore $|V|^2$ entries are now 1. Clearly $\Theta(|V|^2)$ entries needs to be changed to 1 in this update, no matter what algorithm is used.

c The following makes a small modification to the method in a. Each that each time if we run the inner loop, it is guaranteed that at least 1 of the entries in T is changed from 0 to 1. Therefore it is not hard to see that it runs in $|V|^3$ time.

```
1 for i from 1 to |V|
2    if T[i, u] = 1 and T[i, v] = 0
3       for j from 1 to |V|
4          if T[v, j] = 1 then update T[i, j] as 1.
```

Problem 26-1 Solution

a Replace every vertex v other than the source and the destination by two vertices v_1 and v_2, and every edge that goes to v now goes to v_1, and every edge that comes outs of v now comes out of v_2, and add an edge that goes from v_1 to v_2, with capacity set as the node's capacity. Clearly, in this way, the problem becomes a ordinary max-flow problem.

b First, convert the grid to a directed graph, by replacing every edge by 2 edges in both direction. Then, create a source node s, and add an edge from s to every starting point in the grid. Then, create a destination node t, add an edge from every boundary node of the grid to t. Then set the capacity of every node and edge as 1. Now we have a flow problem with vertex capacity. Solve this problem by reducing it to the ordinary max-flow problem. The grid has an escape if and only if the max flow value equals the number of starting points.

Problem 26-2 Solution

a Run the algorithm suggested in the hint, by setting x_0 as source and y_0 as the sink. In this way we find a maximum matching between $x_1 \ldots x_n$ and $y_1 \ldots y_n$. Now construct the path cover in this way: if x_i and y_j is included in the maximum matching generated, then mark the edge (i, j) in the original graph. It is not hard to see that the marked edges naturally partition the vertices into several chains. That is, we have a path cover P_{min}.

This path cover is indeed a minimum one. From any path cover P, we are able to construct a matching between $x_1 \ldots x_n$ and $y_1 \ldots y_n$ in a reverse manner as above: If

(i, j) is in any path in P, we include (x_i, y_j) in the match(It is obviously a match indeed). The size of the match then is the total number of edges in all paths in P. Note that the size of the match then is the total number of edges in all paths in P equals the number of vertices in the original graph minus the number of paths in P, as one may easily verify. Therefore if the size of P if smaller than P_{min}, then we will have a match between $x_1 \dots x_n$ and $y_1 \dots y_n$ which is greater than the maximum matching we computed to generate P_{min}, which is a contradiction. Therefore P_{min} must be a minimum path cover.

b No. Consider the graph with vertices a, b, c, d and edges $a \to b \to c \to a, d \to a$. Then we have a edge cover with only one path:$d \to a \to b \to c$. However our algorithm will output 2 paths, as one may verify.