# Problem Set 6 Solutions

This problem set is due **at 11:59pm** on **Thursday, April 2, 2015.**

---

**Exercise 6-1.** Read CLRS, Chapter 25.

**Exercise 6-2.** Exercise 25.3-2.

**Exercise 6-3.** Exercise 25.3-5.

**Exercise 6-4.** Read CLRS, Chapter 23.

**Exercise 6-5.** Exercise 23.2-4.

**Exercise 6-6.** Exercise 23.2-5.

---

**Problem 6-1. Dynamic and Bounded-Hop All-Pairs Shortest Paths** [25 points]

This problem explores some extensions of the All-Pairs Shortest Paths (APSP) algorithms covered in lecture and in CLRS. All parts of this problem assume nonnegative real weights. We assume that the vertices are numbered $1, 2, \ldots, n$.

Weighted directed graphs may be used to model communication networks, and shortest distances (shortest-path weights) between nodes may be used to suggest routes for messages. However, most communication networks are dynamic, i.e., the weights of edges may change over time. So it is useful to have a way of modifying distance estimates to reflect these changes.

**(a)** [6 points] Give an efficient algorithm that, given a weighted directed graph $G = (V, E, W)$, a correct distance matrix $D$ for $G$, a corresponding predecessor matrix $\Pi$, and a triple $(i, j, r)$, where $i, j \in V$ and $r$ is a nonnegative real, modifies $D$ and $\Pi$ to reflect the effects of changing $w_{i,j}$ to $r$.

Your algorithm will need to handle three cases: $r = w_{i,j}$, $r < w_{i,j}$, and $r > w_{i,j}$. For each of the cases, analyze your algorithm. (Note: Your worst case running time for one of these cases may not be better than $O(V^3)$.)

**Solution:** The algorithm should behave differently in these three cases:

1. $r = w_{i,j}$.
   No changes are needed, since no shortest path can change.

2. $r < w_{i,j}$.
   In this case, we may improve the distances for some pairs of vertices: for each pair $(x, y)$ of vertices, $x \neq y$, set

   $$d_{x,y} = \min(d_{x,i} + r + d_{j,y}, d_{x,y}).$$

   The two terms in the min expression are as follows:
   - The first term is for the case where the new shortest path from $x$ to $y$ contains the edge $(i, j)$.
   - The second term is for the case where the new shortest path from $x$ to $y$ does not contain the edge $(i, j)$, in which case the shortest path remains unchanged.

   Note that, in the first case, the values of $d_{x,i}$ and $d_{j,y}$ do not change as a result of this adjustment, because the edge $(i, j)$ cannot be included in a shortest path from $x$ to $i$ or a shortest path from $j$ to $y$.
   The total time to recalculate $D$ is $O(V^2)$ (since we need to consider all pairs of vertices $i$ and $j$). The total extra space required is $O(1)$.
   To adjust the predecessor matrix $\Pi$, we need only consider matrix entries $\pi_{x,y}$ for which we reduce the value of $d_{x,y}$. For each of these, we consider two cases: if $y = j$, then $\pi_{x,y} = i$, and otherwise $\pi_{x,y} = \pi_{j,y}$. Note that, in this second case, $\pi_{j,y}$ does not change as a result of this adjustment, because $(i, j)$ does not appear in a shortest path from $j$ to $y$.
   The total time to adjust $\Pi$ is $O(V^2)$, and we need only $O(1)$ extra space.
3. $r > w_{i,j}$.
   In this case, we may worsen the distances for some pairs of nodes. Unfortunately, this seems to require recalculating best distances and parents for all nodes from scratch. $D$ and $\Pi$ can be recalculated in time $O(V^3)$ using Floyd-Warshall.

**(b)** [4 points] Give an example of a weighted directed graph $G = (V, E, W)$, a distance matrix $D$, and a triple $(i, j, r)$ such that any algorithm that modifies $D$ to reflect the effects of changing $w_{i,j}$ to $r$ must take $\Omega(V^2)$ time.

**Solution:** Let $G$ be a line graph with $n$ vertices $1, 2, \ldots, n$ and edges $(i, i + 1)$ for all $i$. Suppose the weights for all these directed edges are 1. All other weights are $\infty$. Let $e = (n, 1, 1)$. Then for every $i, j$, $i < j$, $d_{j,i} = \infty$ before the modification, but is a positive integer after the modification. That means that $\Omega(n^2)$ entries of $D$ must change.

Now suppose that we add a new constraint to the problem — an upper bound of $h$ on the number of hops (edges) in the paths we want to consider. More formally, given a weighted digraph $G = (V, E, W)$ and a positive integer $h$, $0 \leq h \leq n - 1$, we would like to produce a distance matrix $D$ giving the shortest distances for at-most-$h$-hop paths.

**(c)** [5 points] Adapt the Floyd-Warshall algorithm to solve the bounded-hop APSP problem. Analyze its complexity in terms of graph parameters and $h$.

**Solution:** The Floyd-Warshall algorithm introduces intermediate vertices in order, one at a time, in $|V|$ executions of an outer loop (see the code on p. 695 of CLRS). Within each execution of this loop, it considers all pairs $(i, j)$ of vertex numbers. Each such pair requires time $O(1)$ because it involves examining and comparing only three entries in $D$.

When we add the constraint that the complete path must consist of at most $h$ hops, the comparison on line 7 becomes more complicated. It must consider all the different ways of allocating the $h$ hops to the two parts of the path - the part that comes before the new vertex $k$ and the part that comes after $k$. So, we define $d_{i,j}^{k,h}$ to be the weight of a shortest path from vertex $i$ to vertex $j$ for which all intermediate vertices are in the set $1, \ldots, k$ with the added constraint that the path consists of at most $h$ hops. $d_{i,j}^{k,h}$ can now be expressed as,

$$d_{i,j}^{k,h} = \min(\{d_{i,j}^{k-1,h}\} \cup \{d_{i,k}^{k-1,h'} + d_{k,j}^{k-1,h-h'} \mid 0 \leq h' \leq h\}).$$

The total number of sub-problems is now $O(V^3 \cdot h)$, and the time required to solve each sub-problem is $O(h)$, which means the total runtime of this algorithm is $O(V^3 \cdot h^2)$.

For the space requirement, while calculating the results for $k$, we need distance information for $k-1$ and all values of $h$, so the space complexity is now $O(V^2 \cdot h)$.

**(d)** [5 points] Adapt the matrix multiplication strategy from Section 25.1 to solve the bounded-hop APSP problem. Analyze its complexity in terms of graph parameters and $h$. Try to get logarithmic dependence on $h$.

**Solution:** In the description in Section 25.1, the matrix $L^{(k)}$ contains shortest distances using paths of at most $k$ hops; specifically, $l_{i,j}^k$ is the shortest distance for a path from $i$ to $j$ consisting of at most $k$ hops. Thus, $L^{(0)}$ is the matrix with 0 on the main diagonal and $\infty$ everywhere else, and for every $k$, $L^{(k+1)} = L^k \cdot W$, where the matrix multiplication here uses $\min$ and $+$ instead of the usual $+$ and $\times$. Our goal is to produce $L^{(h)}$.

If $h = 0$ then the answer is simply the matrix with 0s on the main diagonal and $\infty$ elsewhere. Now assume $h \geq 1$.

We cannot use the successive squaring strategy from p. 689-690 directly, because that would "overshoot" $h$ if it is not a power of 2. However, we can get the logarithmic dependence on $h$ by using its binary representation, $h = h_l h_{l-1} \ldots h_1$. Since $h \geq 1$, we have $h_l = 1$. Start with $L^{(h_l)} = L^{(1)} = L^{(0)} \cdot W = W$. Then for $j = l-1, \ldots, 1$, calculate $L^{(h_l \ldots h_j)}$ as follows: If $h_j = 0$ then $(L^{(h_l)})^2$, else $(L^{(h_l)})^2 \cdot W$. The final answer is $L^{(h_l \ldots h_1)}$.

The number of matrix calculations is $O(\lg h)$, and each calculation takes time $O(V^3)$, for a total time complexity of $O(V^3 \lg h)$. Note that this is better than the modified Floyd-Warshall algorithm in Part (c), which is interesting given that vanilla Floyd-Warshall performs better on the standard All-pair Shortest Path problem. The space complexity is $O(V^2)$.

**(e)** [5 points] Finally, consider a dynamic version of the bounded-hop APSP problem. Design an algorithm that is given the following as input:

1. a weighted directed graph $G = (V, E, W)$;
2. a hop count $h$, $0 \leq h \leq n - 1$;
3. a correct distance matrix $D$ yielding shortest at-most-$h$-hop distances, and possibly additional distance information that is useful for solving this problem; and
4. a triple $(i, j, r)$, where $i, j \in V$ and $r$ is a nonnegative real.

Your algorithm should modify $D$ to reflect the effects of changing $w_{i,j}$ to $r$, and should also update any additional distance information that you have added. As for Part (a), your algorithm will need to handle three cases: $r = w_{i,j}$, $r < w_{i,j}$, and $r > w_{i,j}$. For each of the cases, analyze your algorithm's complexity, in terms of graph parameters and $h$.

**Solution:** The additional distance information might be the entire set of matrices $L^{(k)}$, $0 \leq k \leq h$, as defined in the solution to Part (d). Thus, $D = L^{(h)}$. All of these can be calculated from scratch in time $O(V^3 \cdot h)$, using a series of successive multiplications by matrix $W$.

Consider how to modify the solution to Part (a). The case where $r = w_{i,j}$ still involves no changes, and the case where $r > w_{i,j}$ involves recalculation of all the distances, in all the matrices, at a time cost of $O(V^3 \cdot h)$.

In the case where $r < w_{i,j}$, we must recalculate the distances for all ordered pairs $(x, y)$ in all of the matrices; thus, we must recalculate $V^2 \cdot h$ entries. For each $x, y$, we define the following:

$$l_{x,y}^{(k)} = \min(\{l_{x,y}^{(k)}\} \cup \{l_{x,i}^{(k')} + r + l_{j,y}^{(k-k'-1)} \mid 0 \leq k' \leq k-1\}).$$

The time cost for recalculating each entry is $O(k)$, which is $O(h)$. So the total time cost is $O(V^2 \cdot h^2)$. The space complexity is $O(V^2 \cdot h)$.

**Problem 6-2. Minimum Spanning Trees with Unique Edge Weights** [25 points]

Consider an undirected graph $G = (V, E)$ with a weight function $w$ providing nonnegative real-valued weights, such that the weights of all the edges are different.

**(a)** [5 points] Prove that, under the given uniqueness assumption, $G$ has a unique Minimum Spanning Tree.

**Solution:** Suppose for the sake of contradiction that $G$ has two different MSTs, $T_1$ and $T_2$. Let $e$ be the smallest weight edge that appears in exactly one of $T_1$ and $T_2$ - note that such an edge must exist since trees $T_1$ and $T_2$ are distinct. Without loss of generality, suppose that $e$ is in $T_1$ and not in $T_2$.

Then form a new graph $T_2'$ consisting of $T_2$ with $e$ added. Observe that $T_2'$ now has a cycle. Since $T_1$ can't contain the entire cycle, there must be some edge $e' \neq e$ in the cycle that is not in $T_1$. Since we assumed that $e$ is the smallest weight edge in one of the trees and that $e'$ is in $T_2$ but not in $T_1$, we see that $w(e') > w(e)$. (We are using the unique weight assumption here to get strict inequality.)

Then removing $e'$ from $T_2'$ yields a new spanning tree $T_2''$ with smaller weight than $T_2$. This contradicts our initial assumption that $T_2$ is an MST.

Each of the next three parts outlines an MST algorithm for graphs with unique edge weights. In each case, say whether this is a correct MST algorithm or not. If so, give a proof, a more detailed description of an efficient algorithm, and an analysis. If not, give a specific counterexample. (We are omitting the point values for these parts because we will assign more points to algorithms and fewer to counterexamples.)

**(b) [Batched Edge-Addition MST]**

The algorithm maintains a set $A$ of edges that are known to be in the MST. Initially, $A$ is empty. The algorithm operates in phases; in each, it adds a batch of one or more edges to $A$. Phases continue until we have a spanning tree.

Specifically, in each phase, the algorithm does the following: For each component tree $C$ in the forest formed by $A$, identify the lightest weight edge $e_C$ crossing the cut between $C$ and the rest of the components. After determining these edges for all component trees, add all of the edges $e_C$ to $A$, in one batch.

**Solution:** This does yield an MST.

We can show by induction on the number of phases that the set $A$ is always a subset of the unique MST. For the base case, the set $A$ contains no edges, which trivially is a subset of the unique MST.

For the inductive step, consider a set $A$ of edges at the beginning of some phase. By our inductive hypothesis, the set $A$ is a subset of the unique MST. We want to show that at the end of the same phase, the set $A'$ constructed by adding some edges to $A$, is still a subset of the unique MST.

For each component $C$ in the forest formed by $A$, the lightest edge is part of some MST, by CLRS Corollary 23.2. Since there is only one MST, all the chosen edges are part of the same unique MST. Thus, when we add them all, the new set $A'$ obtained at the end of the phase is still a subset of the unique MST.

**Detailed algorithm:**

We can implement this algorithm using a Union-Find set structure for the vertices. For each component in the structure, we maintain a list of edges that have exactly one endpoint in the component. Initially, $A$ is empty, and we create a separate set for each vertex, using the MAKESET operation described in Recitation 3. We sort the edges in ascending order of edge weights in $O(E \log E)$ time. For each single-vertex component, we compute an individual list of edges incident on that component. This can be done by a linear pass through the sorted edge list.

At each phase, we perform the following steps,

- For each component $C$, we find the lightest incident edge, by looking at the first entry in its edge list. This takes $O(1)$ time. All these edges are added to $A$.
- For each edge added to $A$, perform a union operation between the components associated with the end points of the edge.
- Now, we need to update the edge list for each of the merged components. This can be done using the two-finger algorithm from Merge sort. Each of these edge lists has $O(E_i)$ elements, so the merge step takes $O(E_i)$ time. (Here, we assume that component $i$ has $E_i$ edges, and WLOG, for arbitrary components $i$ and $j$, $E_i > E_j$) During this merge step, we need to remove all edges that now belong to the same component; this can be done by calling FIND on each edge of the merged list - the total number of edges in the merged list is $O(E_i)$ and the time complexity of each FIND operation is $O(\alpha(V)$, which means that the total time complexity of this merge step is $O(E_i(1 + \alpha(V)))$.

Note that the worst case time complexity of the merge operation per component per iteration is $O(E_i(1 + \alpha(V)))$, leading to a worst case time complexity per iteration of $O(\sum_i E_i(1 + \alpha(V))) = O(E(1 + \alpha(V)))$. Also note that the total number of iterations in the worst case is $O(\log V)$ (since the number of components is reduced by at least a factor of two at each phase), leading to a total worst case time complexity of $O(E \log V(1 + \alpha(V)))$.

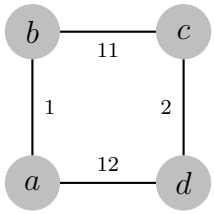
The total length of all individual edge lists is $O(E)$, and the space complexity of the Union-Find data structure that keeps track of which component a vertex belongs to is $O(V)$, which means that this algorithm has a total space complexity of $O(V + E)$, in addition to the list $A$, which has size $O(E)$.

**(c) [Divide-and-Conquer MST]**

The algorithm uses a simple *Divide-and-Conquer* strategy: Divide the set $V$ of vertices arbitrarily into disjoint sets $V_1$ and $V_2$, each of size roughly $V/2$. Define graph $G_1 = (V_1, E_1)$, where $E_1$ is the subset of $E$ for which both endpoints are in $V_1$. Define $G_2 = (V_2, E_2)$ analogously.

Recursively find (unique) MSTs for both $G_1$ and $G_2$; call them $T_1$ and $T_2$. Then find the (unique) lightest edge that crosses the cut between the two sets of vertices $V_1$ and $V_2$, and add that to form the final spanning tree $T$.

**Solution:** Incorrect. A simple counterexample is a four-node ring graph with nodes $a$, $b$, $c$, $d$, and edge weights $w(a,b) = 1$, $w(b,c) = 11$, $w(c,d) = 2$, $w(d,a) = 12$.




The graph could be split with $V_1 = \{a, d\}$ and $V_2 = \{b, c\}$. Then the MSTs of the two halves have weights 12 and 11 respectively. Adding in the lightest cut edge (edge between $a$ and $b$ with weight 1) gives us a Spanning Tree of weight $11 + 12 + 1 = 24$, which is clearly sub-optimal (edges $(a, b)$, $(b, c)$ and $(c, d)$ yield a spanning tree of weight 14).

**(d) [Cycle-Breaking MST]**

The algorithm operates in phases. In each phase, the algorithm first finds some nonempty subset of the simple cycles in the graph. Then it identifies the heaviest edge on each cycle, and removes all these heavy edges. Phases continue until we have a spanning tree.

**Solution:** This does yield an MST. Let $T$ denote the unique MST of $G$. We first state and prove the following claim.

*Claim:* For each simple cycle $C$ in $G$, the heaviest edge on cycle $C$ is not in $T$.

*Proof of Claim:* Suppose for the sake of contradiction that the heaviest edge of some simple cycle $C$ is in $T$. Since $T$ cannot contain the entire cycle $C$, there must be some other edge $e' \neq e$ in $C$ that is not in $T$. Then we can construct a new tree $T'$ from $T$ by adding edge $e'$ and removing edge $e$. $T'$ is also a spanning tree, and its weight is strictly less than that of $T$, which is a contradiction, since we assumed that $T$ is the MST.
*End of proof of Claim*

Now consider any edge $e$ that is removed during any phase $i$ of the algorithm. Edge $e$ must be the heaviest edge of some simple cycle $C$ of the graph $G'$ at the beginning of phase $i$. But cycle $C$ is also a simple cycle of the original graph $G$, so the removed edge $e$ is also the heaviest edge on a simple cycle of $G$. So by the Claim, edge $e$ is not in $T$.

Thus, the algorithm only removes edges that are not in $T$. Since the final graph is a spanning tree, and contains $T$, it must be equal to $T$.

**Detailed algorithm:**

We can implement this algorithm as follows,

CYCLEBREAKINGMST($G$)

```
1  while |G.E| > |V| − 1
2      C = FINDCYCLE(G).
3      e = FINDMAXIMUMEDGE(C)
4      G = G − e
5  return G
```

Here, FINDCYCLE finds some cycle in the graph $G$ and returns it - this can be implemented using Depth First Search (DFS); DFS allows us to find back edges within the graph $G$ in $O(E)$ time. Once we have a cycle, we can find the maximum weight edge in it in $O(V)$ time (since a simple cycle can have at most $V$ edges); we then remove this edge from the graph $G$ and repeat the above steps on the reduced graph, until the graph contains no more cycles.

The total runtime complexity of this algorithm is $O(E \cdot (E - V)) = O(E^2)$ time.
The total space complexity of the algorithm is $O(E)$, since we only need to keep track of one simple cycle in the graph, and a simple cycle in the worst case has $O(E)$ edges.

*Remark:* Note that in the above implementation, we remove one edge from the graph at a time. We can often do better than this by removing multiple edges from the graph in a single iteration (multiple cycles could be identified in a single iteration).

However, it is not clear how to improve the worst-case time complexity by doing this.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015