Quiz 2 Solutions

Problem 1. Hacking Skip Lists

After years of painstaking effort, you launch myferretplanet-spaces.com, an e-commerce site devoted to selling ferrets online. Your list of active customer orders is implemented as a skip list, with all data about an order stored as an element in the skip list. Every time a customer asks about the status of their order, your site queries the customer order list and displays the data. Every order gets a unique ID number that is used to place it in the customer order list. ID numbers are assigned by your web site and are not necessarily increasing, so the most recent order could go anywhere in the skip list. For a while, your site works well; orders can be managed quickly and customers are happy.

But the ferret e-commerce industry is a cutthroat one, and one dark and stormy night your competitors try to slow down your site. There are two operations they can do: add a dummy order to your site, and cancel an order that they added.

(a) Your system is storing information about $n$ legitimate orders. Your competitors add $m$ dummy orders to your system. What effect will this have on the runtime of query, insert, and delete operations for legitimate customer orders the morning after your competitors try to slow your site? Assume that $m >> n$.

Solution:

INSERT, DELETE, and SEARCH operations on a skip list normally take $\Theta(n)$ time, when a skip list has $n$ elements. Here the number of elements (customer orders) changes from $n$ to $n + m$, so all three operations take $\Theta(m + n) = \Theta(m)$ time. This could slow the site substantially. For instance, if $m = 2^n$, performance will change from $\Theta(\log n)$ to $\Theta(n)$

(b) Your competitors find a security loophole in your site that lets them see information about your skip list: specifically, which level each order has been promoted to. They again wish to perform $m$ operations, with the goal of slowing down later legitimate customer orders as much as possible. What should they do? What effect will this have on the runtime of query, insert, and delete operations for legitimate customer orders the morning after?

Solution:

To slow down the site, the competitors should try to make the skip list more like a linked list. They can do this using the following procedure:

1. Insert a dummy order.
2. Delete that order if it was promoted above the lowest level in the skip list.
3. Repeat until all $m$ operations have been used up.
Performance

In the morning, INSERT, DELETE, and SEARCH operations will still take $\log(n)$ time to traverse the levels of the skip list. At the bottom of the skip list, the list of elements that must be traversed will have an expected length of $\Theta(m/n)$.

So these operations will take $\Theta(m/n + \log n)$.

Another Acceptable Answer The problem didn’t say that the competitors know the size of $n$, the number of orders in the system when they start their attack.

But if they did, they would know the number of levels in the skip list would be about $\log(n)$. They could use this to

1. Insert a dummy order.
2. Delete that order if it was promoted above the lowest level in the skip list, unless it is promoted to a level higher than $\log(n)$ and is the first dummy element promoted to that level.
3. Repeat until all $m$ operations have been used up.

This strategy would slow performance to $\Theta(m/n + \log m)$. We gave full credit for this answer (for students who assumed competitors knew the size of $n$) and full credit for the other answer described (for students who assumed competitors did not know the size of $n$).
Problem 2. Balanced linear combination

Suppose you are given two vectors \( u, v \) of length \( n \), such that, \( u_i, v_i > 0 \) for all \( i = 1, 2, \ldots, n \), and \( n \) is odd. A vector \( w \) is a balanced linear combination of \( u \) and \( v \) if it is a non-trivial linear combination of \( u \) and \( v \), i.e.

\[
w = \alpha u + \beta v \text{ for some } \alpha, \beta \text{ such that } \alpha \neq 0 \text{ and } \beta \neq 0
\]

and the median of \( w \) is 0. For example, if \( n = 5 \), \( u = (1, 3, 4, 2, 5) \) and \( v = (4, 4, 4, 4, 4) \), then \( w = 4u - 3v = (-8, 0, 4, -4, 8) \) is a balanced linear combination of \( u \) and \( v \).

Give an efficient algorithm to find a balanced linear combination of \( u \) and \( v \).

**Solution:** Observe that for any balanced linear combination \( z \) of \( u \) and \( v \),

\[
z = \alpha u + \beta v
\]

There is a normalized (the coefficient of \( u \) is 1) balanced linear combination for \( u \) and \( v \)

\[
z_1 = u + (\beta/\alpha)v
\]

since \( \alpha \neq 0 \) and \( \beta \neq 0 \).

Therefore, we can narrow down our search to the set of normalized balanced linear combination, i.e. finding appropriate value of \( \beta \) such that the vector

\[
x = u + \beta v
\]

has median equal to zero. For each \( i = 1, 2, \cdots n \), calculate the array \( s \) such that \( s_i = v_i/u_i \). It is straightforward to verify that

- If \( \beta > s_i \), then \( x_i \) is positive.
- If \( \beta < s_i \), then \( x_i \) is negative.
- If \( \beta = s_i \), then \( x_i \) is 0.

Therefore, if \( \beta \) is the median of \( s \), then median of \( x \) is 0. The problem is equivalent to finding the median of the array \( s \).

Our algorithm works as follows.

1. Calculate \( s_i \), for all \( i = 1, 2, \cdots n \).
2. Find median of \( s \) by deterministic SELECTION.
3. Output \( u - \beta v \).

All three steps in the algorithm only takes \( O(n) \) time. Therefore, the total running time of this algorithm is \( O(n) \).
Problem 3. Graph Sparsification

You are given a connected graph $G = (V, E)$ such that each edge $e \in E$ has a distinct non-negative value $w(e)$. In order to reduce the cost of storing $G$, we want to remove edges from $G$ such that the number of remaining edges is exactly equal to the number of nodes. However, there are two constraints on which edges can be removed and which edges can be kept. First, each vertex chooses an edge incident to it to keep. An edge will only be kept when it is chosen by some vertex. Any edge that is not chosen by any vertex will be removed. Second, the set of remaining edges must keep $G$ connected, i.e. for any pair of vertices $u$ and $v$, there is a path from $u$ to $v$ only using the chosen edges.

Give an efficient algorithm to find which edge should be chosen by which vertex such that the total value of the remaining edges is maximized.

Solution:

The set of edges to be chosen:

In our algorithm, we find the maximum spanning tree in $G$ and keep all of its edges. Since this tree only contains $n - 1$ edges, one more edge needs to be chosen. To find such edge, we look at all edges in $G$ which are not in the tree and pick the one with maximum weight.

Find the maximum spanning tree: For each edge $e \in E$, let $w_1(e) = C - w(e)$, where $C$ is the maximum edge weight in $G$. For any spanning tree $S$ in $G$, $w_1(S) = (n - 1)C - w(S)$. Thus, the minimum spanning tree in $G$ w.r.t weight function $w_1$ is also the maximum spanning tree in $G$ w.r.t weight function $w$. Prim algorithm can be used to find the maximum spanning tree in $G$.

The mapping

Let $S_{max}$ be the maximum spanning tree and $e_{max} = (u, v)$ be the last edge added to the set. Consider the tree $S_{max}$ with $u$ as root. Repeatedly do the following step until only one vertex left in the tree: Pick a leaf $x$ in the tree, let $f$ map $x$ to the edge that connects $x$ to its parent, and delete $x$ (along with the edge) from the tree. After $n - 1$ such steps, the only vertex left is $u$. We can complete the mapping $f$ by letting it map $u$ to $e_{max}$.

Proof of correctness We will prove that the mapping $f$ satisfy all four constrains:

1. $f(u) = f(v)$ iff $u = v$.
2. $\sum_v w(f(v))$ is maximized.
3. $f(v)$ is incident to $v$, for all vertices $v$.
4. the edge set $\{f(v) | v \in V\}$ is connected.

The first and third constrains is straightforward from the construction of our mappings.

The fourth constrain is guaranteed by the fact that the set of chosen edges contains a spanning tree in $G$.

---

$^1$ $w(S)$ and $w_1(S)$ are the total edge weight of $S$ w.r.t weight function $w$ and $w_1$ respectively.
Therefore, we only need to prove that $\sum_v w(f(v))$ is maximized. We will prove that $\sum_v w(f(v)) \geq \sum_v w(g(v))$ for any mapping $g$ satisfying constrains 1,3 and 4. Let $A_g$ be the set of edges in $G$ chosen by $g$. Since $A_g$ has exactly $n$ edges, the (connected) subgraph of $G$ formed by $A_g$ has exactly one cycle. In that cycle, the must be at least one edge $e_1$ that is not in $S_{max}$, thus, $w(e_1) \leq w(e_{max})$. Also, consider the set $A_g - \{e_1\}$, it has exactly $n - 1$ edges and still forms a connected subgraph in $G$. Therefore, since $S_{max}$ is the maximum spanning tree in $G$, $w(A_g - \{e_1\}) \leq w(S_{max})$. Thus, $w(A_g) \leq w(S_{max} \cup e_{max})$.

**Running time**
The optimal running time is $O(E + V \log V)$ by using Prim algorithm to find MST. If Kruskal algorithm is used, then the running time is $O(E \log V)$, which is not optimal.
Problem 4. Detecting Compatible Subtrees

Given two rooted trees, we say that is a subtree of if there is a one-to-one mapping from the vertices of to the vertices of that preserves all parent-child relationships. In this problem, we consider rooted trees with the additional feature that every vertex is assigned a label from some set (for example, the labels may come from the set of colors ). Moreover, we are told that only certain pairs of labels are “compatible.” The set of compatible labels can be represented as an undirected graph , called a compatibility graph, with vertex set and edges connecting labels that are compatible.

Given two rooted trees and , both with labels in , we would like to determine whether is a subtree of , but with the additional constraint that each vertex of must be matched with a compatibly labeled vertex of . Formally, if there is a function with all of the properties in the previous paragraph, and the additional property that the labels of and are compatible for all , we say is a compatibly labeled subtree of . Figure 1 illustrates an example.

(a) Suppose you are given two labeled rooted trees, and , along with a compatibility graph . Let and , where . Further suppose that both and have depth . Give an efficient algorithm to determine if is a compatibly labeled subtree of . You may assume that for vertices and , it only takes time to check if the labels of and are compatible.

Solution: This problem can be solved in time by reducing it to bipartite matching. The algorithm is as follows.

First we check that the root of is compatible with the root of . As specified in the problem, this can be done in time. If the roots are compatible, then we need to check that there is a way to compatibly map the leaves of to the leaves of . This is exactly the maximum bipartite matching problem. To solve it we create a bipartite graph. The vertices are the leaves of and the vertices are the leaves of . For each and we check whether they are compatible, and if so, we put an edge between the corresponding nodes of the graph . It is clear that the leaves of can be compatibly matched to the leaves of iff the maximum matching in is of size .

A solution to the maximum bipartite matching problem is given in the book, and was also covered in recitation. To solve the problem, we make all the edges of the bipartite

---

2For completeness, we give some formal definitions here: a rooted tree is simply a tree where one vertex is designated as the root. For any vertex , the children of are defined as the neighbors of which are further from the root, and the parent of is defined as the neighbor of which is closer to the root (if is not the root). Every vertex has exactly one parent, except the root which has no parents. Rooted trees need not be binary or balanced - any vertex may have an arbitrary number of children. There is no ordering among the children.

The depth of a rooted tree is the depth of the deepest vertex (the root has depth 0, the children of the root have depth 1, their children have depth 2, etc.).

3In other words, must map every vertex of to a distinct vertex of , in such a way that if is the parent (or child) of in , then is the parent (or child, respectively) of in .
Figure 1: Two labeled rooted trees $S$ and $T$. As illustrated by the dark lines, $S$ is a compatibly labeled subtree of $T$. 
graph directed from $L$ to $R$. We add a source node $s$ connected to all nodes in $L$, and a sink node $t$ connected from all nodes in $R$. Then we give all edges capacity 1, and solve for the maximum flow on the resulting graph. See CLRS section 26.3 for details.

Now we analyze the algorithm. Constructing the graph $G$ takes $O(mn)$ time. The number of edges in $G$ is bounded by $O(mn)$, since in the worst case each node of $L$ is connected to every node of $R$. Solving for the maximum flow using any Ford-Fulkerson algorithm takes $O(E|f^*|)$ where $E$ is the number of edges in $G$ and $|f^*|$ is the size of the maximum flow. Here $|f^*|$ is at most $m - 1 = O(m)$. So in total, Ford-Fulkerson takes $O(m^2n)$ time.

Notes on alternative analysis and solutions:

- Some people noted that the Hopcraft-Karp algorithm referred to in the book takes only $O(\sqrt{VE})$ time, which in this case comes out to be $O(mn^{1.5})$. This answer also received full credit. However, this was not the preferred answer, since we didn’t actually see the details of the algorithm or analysis in class, nor do the details appear in the book (they only appear as a problem in the book, they are never fully explained). Similarly, some people cited the $O(n^\omega)$ bipartite matching algorithm (where $\omega$ is the best known matrix multiplication exponent) by Mucha and Sankowski. Again, this was unnecessary. This algorithm was mentioned casually in recitation, but we didn’t even come close to talking about the details.

- Some people used a variation on bipartite matching to achieve a better runtime when the number of labels is small. These people also created a bipartite graph $G = L \cup R$, but the number of nodes in both $L$ and $R$ was equal to the number of possible labels. As before, they created a source node $s$ connected to all nodes in $L$, and a sink node $t$ connected from all nodes in $R$. However the capacity of the edge $(s,i)$ was equal to the number of leaf nodes in $S$ with label $i$, and the capacity of the edge $(j,t)$ was equal to the number of leaf nodes in $T$ with label $j$. There were also edges with infinite capacity going from $L$ to $R$ corresponding to compatible labels. The maximum flow was then found on this graph $G$, and if it was equal to $m - 1$, the algorithm accepted.

To analyze the running time of this algorithm, let $k$ be the number of labels. There are then $O(k^2)$ possible edges on the compatibility graph. Thus the time to construct $G$ is $O(n + k^2)$. To find the maximum flow on $G$, we cannot necessarily use Hopcraft-Karp or Mucha-Sankowski, since we are no longer solving a strict matching problem (we are allowing multiple nodes in $L$ to match to the same node in $R$). Thus we should use Edmonds-Karp. The running time of Edmonds-Karp on any graph is $O(\min(E|f^*|, VE^2))$. In this case, if there are $k$ possible labels, this comes out to be $O(\min(k^2m, k^5))$.

Thus the total running time is $O(n + \min(k^2m, k^5))$. If $k$ is small enough (say $k < m^{2/5}n^{1/5}$), then this offers a speed advantage over the $O(m^2n)$ algorithm given earlier. However, in the worst case, $k$ could be as large as $n$, in which case the
running time could be \(O(n^2m)\), slightly worse than the \(O(m^2n)\) algorithm given earlier. Since this solution is incomparable to the previous one (and potentially better when \(k\) is small), it received full credit.

- Several people suggested algorithms that greedily selected compatible vertices, or tried to use some variant of the stable marriage algorithm. Magically these algorithms were purported to run in \(O(mn)\) time. Unfortunately, they did not work. If an algorithm actually did achieve that running time, it would be a major research result. Here’s why. Given any bipartite graph, you can turn the perfect-matching problem into a problem about depth-1 subtree compatibility. Just construct a tree \(S\) that has one leaf for every node in \(L\), and a tree \(T\) that has one leaf for every node in \(R\), and make the bipartite graph the compatibility graph (also give the roots of \(T\) and \(S\) colors that are known to be compatible). Then there is a perfect matching in the original graph iff \(S\) is a subtree of \(T\). The best known perfect matching algorithm runs in \(O(n^\omega)\), where \(\omega\) is around 2.38. So an \(O(mn) = O(n^2)\) algorithm beating that bound would be a major breakthrough.

**Solution:**

The trick here was to use dynamic programming AND bipartite matching. With both of these tools and some fine-grained analysis, it is possible to obtain an algorithm with \(O(m^2n)\) running time.

First we create an \(m \times n\) array \(M\). For any \(i \in S\) and \(j \in T\), the entry \(M[i, j]\) will tell us whether the subtree rooted at \(i\) in \(S\) can be compatibly mapped to the subtree rooted at \(j\) in \(T\). (Formally, we want \(M[i, j] = \text{TRUE}\) if there is a one-to-one function \(f\) that maps \(i\) to \(j\) and all descendents of \(i\) to descendents of \(j\) in such a way that parent-child relationships are preserved and \(i\) is always compatible with \(f(i)\). Otherwise, we want \(M[i, j] = \text{FALSE}\).) Clearly if we can compute \(M[i, j]\) for all \(i \in S\) and \(j \in T\), then we just need to check whether \(M[\text{root}(S), j] = \text{TRUE}\) for some \(j \in T\) to tell if \(S\) is a compatible subtree of \(T\).

To compute each entry of \(M[i, j]\), we first check whether node \(i\) is compatible to node \(j\). If so, then we need to solve a bipartite matching problem. We build a bipartite graph \(G = L \cup R\) where the vertices of \(L\) correspond to children of \(i\) in \(S\), and the vertices of \(R\) correspond to children of \(j\) in \(T\). We connect a vertex \(u\) of \(L\) to a vertex \(v\) of \(R\) if \(M[u, v] = \text{TRUE}\). Then we find the maximum matching in this bipartite graph, and if it’s equal to the number of children of \(i\), we set \(M[i, j] = \text{TRUE}\). To insure that we don’t recompute answers to subproblems, we iterate over the nodes in order of decreasing depth (from the leaves to the root). The pseudocode for this algorithm follows (it assumes that the nodes are in decreasing depth order, so for any node \(i\), child\((i) < i\):

\[(b)\] Same as part (a), but now let \(S\) and \(T\) be arbitrary (not necessarily depth 1) labeled rooted trees.
for $i = 1...m$
        for $j = 1...n$
            if $i$ is a leaf Then $M[i, j] = \text{Compatible}(i, j, H)$
            Else
                Let $G = \text{Construct-Bipartite-Graph}(\text{children}(i), \text{children}(j), M)$
                If (Compatible($i, j, H$) AND Max-Matching($G$) = $|\text{children}(i)|$)
                    Then $M[i, j] = \text{TRUE}$
                Else $M[i, j] = \text{FALSE}$

The “Compatible” subroutine simply takes two nodes and the compatibility graph, and returns True/False depending on whether the nodes are compatible. The “Construct-Bipartite-Graph” subroutine uses previously computed entries of $M$ to tell which children of $i$ are matchable to which children of $j$ (note that this subroutine does not need access to the compatibility graph, just to previously computed entries of $M$). The “Max-Matching” subroutine is the Ford-Fulkerson algorithm described in part (a).

Now we analyze the running time of this algorithm. First, we give a course analysis. To compute each entry $M[i, j]$, we are solving a bipartite matching problem no bigger than that in part (a). Thus we are performing $O(mn)$ bipartite matchings, each taking at most $O(m^2n)$ time, for a total of at most $O(m^3n^2)$. This analysis is correct (and received full credit), but not tight. We can do better as follows:

Let $m_i$ be the number of children of node $i$ in $S$, and let $n_j$ be the number of children of node $j$ in $T$. Then the running time of the algorithm is bounded by

$$\sum_i \sum_j O(m_i^2n_j) = O(\sum_i \sum_j m_i^2 n_j)$$
$$= O(\sum_i m_i^2 \sum_j n_j)$$
$$= O(n \sum_i m_i^2)$$
$$\leq O(n(\sum_i m_i)^2)$$
$$= O(nm^2)$$
Problem 5. Feasible Region

Consider an instance of the linear programming problem in two dimensions (i.e., with \( m \) constraints over \( n = 2 \) variables). Give an efficient algorithm that reports all vertices of the feasible region polygon.

Solution: The most efficient algorithm for this problem runs in \( O(m \log m) \) time. The algorithm actually solves a (harder) problem: given a set of given \( m \) half-spaces, find their intersection polygon, i.e., report the vertices sorted by the order they appear on the polygon.

Near-linear time algorithm. The \( O(m \log m) \)-time algorithm is based on sorting and the divide-and-conquer approach. First, we sort the half-spaces by their direction angles (ranging from 0 to 360 degrees), resulting in a sequence of half-spaces \( A \). Then, we split the sequence \( A \) into two sequences \( B \) and \( C \) where \( B \) contains all half-spaces with angles in range \([0, 180)\). We compute the intersection \( IB \) of halfspaces in \( B \) and intersection \( IC \) of halfspaces in \( C \) using the same algorithm. In the end we intersect \( IB \) and \( IC \) using essentially the same algorithm.

The intersection \( IB \) is computed using the divide and conquer approach. We split \( B \) into two equal-size sequences \( B' \) and \( B'' \), and recursively compute their intersections \( IB' \) and \( IB'' \). The combine step is as follows. Let \( S' = s'_1 \ldots s'_t' \) and \( S'' = s''_1 \ldots s''_t'' \) be the sequences of segments defining \( IB' \) and \( IB'' \), respectively. It now suffices to find the unique intersection between the segments in \( S' \) and \( S'' \). First, we check if if the first and last segments of \( S' \) and \( S'' \) contribute to such intersection; if yes, we are done. If not, then let \( s''_{i_1} \) be the segment in \( S'' \) intersected by the line along segment \( s'_1 \) (we can find it in \( O(m) \) time). We now observe the segment \( s''_{i_2} \) of \( S'' \) that is intersected by the line along \( s'_2 \) is such that \( i_2 \geq i_1 \). Therefore, we can find \( i_2 \) by incrementing \( i_1 \) and checking for intersection of respective segments in \( S'' \). By iterating this approach, we can find the intersection between segments in \( S' \) and \( S'' \) in \( O(m) \) time.

Quadratic-time algorithm. A simpler (but slower) approach is to use the incremental approach. Specifically, we can construct the intersection of the halfspaces by adding one halfspace at a time. Given the current intersection region \( I \) and a new halfspace \( h \) defined by a line \( l \), the algorithm finds the two points on the boundary of \( I \) that intersect \( l \). This can be done by enumerating all segments describing \( I \), and checking for intersection with \( l \). Since the number of segments defining \( I \) is at most \( m \), adding a new halfspace can be accomplished in \( O(m) \) time. The overall algorithm runs in time \( O(m^2) \).

Cubic-time algorithm. Even simpler (but slow) algorithm proceeds by enumerating all possible candidates for vertices in the feasible region, and checking their feasibility. See Lecture 17, slide 9 for details.