Quiz 2 Solutions

This take-home quiz contains 4 problems for a total of 80 points. Each problem accounts for 20 points.

Problem 1. Nearest T?

As severe budget cuts hit the MBTA, the T authority is planning to phase out several T stops along the Red line. To compensate for the inconvenience, the MBTA is planning a user friendly 1-800-service that you can call at any time to find if your favorite subway stop is still in service, and if not, the names of the two stops on either side of your favorite stop. However, they would like your help is designing the data structures underlying this state-of-the-art service. Below is a formal description of the task.

Assume that the original locations of the stops of the Red line were numbered 1 to n and all stops are active initially. Your data structure should support the following two operations:

- **DELETE(i)** - deactivates the subway stop at location i.
- **NEARBY(i)** - returns a pair (j, k) with \( j \leq i \leq k \) such that \( j \) is the largest numbered active subway stop satisfying \( j \leq i \) and \( k \) is the smallest numbered active subway stop with \( i \leq k \).

Your data structure may use \( O(n) \) space. Your goal is to minimize the total time taken to execute a sequence of \( D \) DELETE operations and \( N \) NEARBY queries (interleaved arbitrarily), with \( n, D \ll N \) (say \( n, D \leq N/\sqrt{n} \)). Express your running time as a function of \( n, N, \) and \( D \).

Solution:

After removing all the transportation-related vocabulary, we are left with the problem of designing a data structure which maintains (under deletions) a set \( T \) of numbers from \( \{1 \ldots n\} \) (i.e., operational stops), and supports the successor and predecessor queries. Since successor and predecessor queries are symmetric, we focus on successor. There are several ways of solving this problem, with various degrees of efficiency.

**Linear scan.** One of the simplest solution involves one array \( A[1 \ldots n] \), such that \( A[i] \) stores the successor of \( i \). The cost of successor operation is then \( O(1) \). The cost of delete(i) is much higher, since we need to

- find the successor \( s \) of \( i \) (excluding \( i \) itself, since it just has been deleted)
- set \( A[j] := s \) for all \( j \)'s who had \( i \) as their successor
This take time $O(n)$ in the worst case (or $O(D)$, after more careful analysis). Therefore, the total time is $O(n + N + D^2)$, which is somewhat slow if e.g., $D,n$ are close to $N/\sqrt{n}$. But if $D$ is very small, this runs in $O(N)$ time.

A much less efficient (but even simpler) solution is to maintain only a binary array $B[1 \ldots n]$, such that $B[i] = 1$ if and only if $i$ has not been deleted. In this data structure deletion takes only $O(1)$ time, but finding the successor could take up to $D$ steps. This gives a total time of $O(n + ND)$, which is strictly larger than before, under the assumption $n, D \leq N/\sqrt{n}$.

**Binary trees.** Another solution to maintain a binary tree. One can use here 2-3 trees, red-black trees (which support insertion, deletion and successor in $O(\log n)$ time). In fact, one can use any binary search tree which initially has $O(\log n)$ depth. This is due to the fact that we only perform deletions, so the depth can never increase.

This method results in a total time of $O(n + (N + D) \log n)$. For most settings of $N, D, n$ it beats the previous solution, and can be worse by at most a factor of $O(\log n)$.

**Union-find.** The fastest solution relied on using Union-Find data structure (CLR, p. 445). That data structure maintains a partition of a set $\{1 \ldots n\}$ into disjoint sets $S_1 \ldots S_k$, and supports two operations:

- $\text{Find}(i)$ - returns a pointer to the set containing $i$
- $\text{Union}(p,q)$ - replaces $S_p, S_q$ by $S_p \cup S_q$

To use the data structure, we maintain a partition of $\{1 \ldots n\}$ such that each set $S_i$ contains only numbers $j$ having same successor (in the “operational” set $T$). E.g., if we have $n = 5$ and 1 and 4 are the only operational stops, we have partition $\{1\}, \{2, 3, 4\}, \{5\}$. Initially, each set has only one element. Moreover, for each set we maintain its maximum element (which is precisely the operational stop in that set).

In this scenario, deleting an operational stop $i$ corresponds to the following sequence of operations:

- $p = \text{Find}(i)$ - returns the pointer to the set containing $i$
- $q = \text{Find}(i + 1)$ - returns the pointer to the successor of $i$ (excluding $i$)
- $\text{Union}(p,q)$ - merges $S_p$ with $S_q$. The maximum element of the new set is the maximum of $S_q$

In order to find the successor of $i$, it is sufficient to compute $p = \text{Find}(i)$ and then return the maximum element of $S_p$ (which is maintained, so we do not have to recompute it).

If we use the data structure from CLR, p. 445, we perform Find in $O(1)$ time and Union in (amortized) $O(\log n)$ time. Thus, the total running time is $O(n + N + D \log n)$ time, which is better than both previous solutions.
**Constant time solutions.** There were a couple of incorrect solutions with $O(1)$ time per deletion and successor. The bug was usually in the delete operation, which did not update the successors for all elements whose successor changed.

**Scoring.** The rough scoring guide is as follows:

- $O(n + N + D \log n)$ - 20 points; some points were subtracted for different versions of Union/Find, which resulted in worse running times for $D$ much smaller than $N$
- $O(n + (N + D) \log n)$ - 13 points
- $O(n + N + D^2)$ - 5 points
- $O(n + DN)$ - 2-3 points
- Incorrect solutions - 0 points

**Problem 2. Sequence alignment**

The following problem is a (highly) simplified version of a problem arising in genomics.

You are given two sequences $S_1$ and $S_2$ of characters from $\{A,C,G,T,*\}$, that reflect partial understanding of some unknown DNA sequence $S_0$ (a sequence of characters from $\{A,C,G,T\}$), where the strings $S_1$ and $S_2$ have been obtained by replacing some substrings of $S_0$ by *’s. (See example below.) Give an algorithm to reconstruct a sequence $S_0$ of minimal length that is an expansion of both $S_1$ and $S_2$.

For example, the given sequences may be

$$S_1 = *A * CGTA * AA * G * \quad \text{and} \quad S_2 = *A * GTCG * AG *.$$

The sequences may have been obtained from either

$$S_0^{(1)} = ACGT CGTCT A T T A A G$$

or $S_0^{(2)} = ACGT A A G T C G A G A G \quad \text{or} \quad S_0^{(3)} = AG T C G T A A A G.$$

Of all such strings, your algorithm should find the shortest possible string that is consistent with both $S_1$ and $S_2$.

Analyze the running time of your algorithm as a function of $m = |S_1|$ and $n = |S_2|$.

**Solution:**

We will first derive a recurrence for the Shortest Common Expansion (SCE) of two strings. For two empty strings $\epsilon$, clearly

$$\text{SCE}(\epsilon, \epsilon) = \epsilon.$$
If only one of the strings is empty, the other must be comprised exclusively of wildcards. This can be represented recursively by requiring the last character to be a wildcard, and referring to the SCE with the last character removed:

\[
\text{SCE}(\epsilon, Y) = \begin{cases} 
\text{SCE}(\epsilon, Y[-1]) & \text{if } Y[-1] = * \\
\infty & \text{if } Y[-1] \neq *
\end{cases}
\]

where \(\infty\) indicates no common expansion exists. Symmetrically:

\[
\text{SCE}(X, \epsilon) = \begin{cases} 
\text{SCE}(X[-1], \epsilon) & \text{if } X[-1] = * \\
\infty & \text{if } X[-1] \neq *
\end{cases}
\]

For two non-empty sequences, we will consider several cases depending on their terminal characters \(X[-1]\) and \(Y[-1]\):

\[
\text{SCE}(X, Y) = \begin{cases} 
X[-1] + \text{SCE}(X[-1], Y[-1]) & \text{if } X[-1] = Y[-1] \neq * \\
\minlen \left( \begin{array}{l}
\text{SCE}(X[-1], Y) \\
Y[-1] + \text{SCE}(X, Y[-1])
\end{array} \right) & \text{if } * = X[-1] \neq Y[-1] \\
\minlen \left( \begin{array}{l}
\text{SCE}(X, Y[-1]) \\
X[-1] + \text{SCE}(X[-1], Y)
\end{array} \right) & \text{if } X[-1] \neq Y[-1] = * \\
\minlen \left( \begin{array}{l}
\text{SCE}(X[-1], Y) \\
\text{SCE}(X, Y[-1])
\end{array} \right) & \text{if } X[-1] = Y[-1] = * \\
\infty & \text{if } * \neq X[-1] \neq Y[-1] \neq *
\end{cases}
\]

Where \(\minlen(\ )\) is the string of minimum length. If only \(X[-1] = *\), then all the common expansions either have \(Y[-1]\) unmatched (i.e. the wildcard \(X[-1]\) is used) or are common expansions of \(X[-1]\) and \(Y\). The shortest common expansion from the first set is \(\text{SCE}(X, Y[-1]) + Y[-1]\), and from the second set it is \(\text{SCE}(X[-1], Y)\). The case of only \(Y[-1] = *\) is symmetrical. If both are wildcards, then at least one of them is unused, and so all common expansions are common expansions of either \((X, Y[-1])\) or \((X[-1], Y)\) (or possibly both, but that is OK).

Based on this recurrence, a dynamic programming approach can be taken, similar to the dynamic program for finding the Longest Common Subsequence. We will use two arrays. \(c[i, j]\) will be the length of \(\text{SCE}(X[: i], Y[: j])\) and \(b[i, j]\) will be pointer hints for reconstructing the \(\text{SCE}\).
FILL-SCE(X, Y)
c[0, 0] ← 0
b[0, 0] ← “□”
for i ← 1 to |X|
c[i, 0] ← ∞
if X[i] = * then c[i, 0] ← c[i − 1, 0], b[i, 0] ← “↑”
for j ← 1 to |Y|
c[0, j] ← ∞
if Y[j] = * then c[0, j] ← c[0, j − 1], b[0, j] ← “←”
for i ← 1 to |X|, for j ← 1 to |Y|
c[i, j] ← ∞
if X[i] = Y[j] ≠ * then
c[i, j] ← c[i − 1, j − 1] + 1, b[i, j] ← “↖”
if X[i] =*
if c[i − 1, j] < c[i, j] then c[i, j] ← c[i − 1, j], b[i, j] ← “↑”
if c[i, j − 1] + 1 < c[i, j] then c[i, j] ← c[i, j − 1] + 1, b[i, j] ← “←”
if Y[j] =*
if c[i, j − 1] < c[i, j] then c[i, j] ← c[i, j − 1], b[i, j] ← “←”
if c[i − 1, j] + 1 < c[i, j] then c[i, j] ← c[i − 1, j] + 1, b[i, j] ← “↑”
return b

Note that although the case of two wildcards is not explicitly covered, it is implicitly covered by taking the minimum of the two single-wildcard cases.

If c(|X|, |Y|) < ∞, the SCE exists and can be extracted from b using:

PRINT-SCE(X, Y, b, i, j)
if i = 0 and j = 0 then return
switch on b[i, j]:
“↖” : s ← X[i], PRINT-SCE(X, Y, b, i − 1, j − 1)
“↑” : s ← X[i], PRINT-SCE(X, Y, b, i − 1, j)
“←” : s ← Y[j], PRINT-SCE(X, Y, b, i, j − 1)
if s ≠ * then print s

The running time, as for LCS, is O(|X||Y|).

Solutions that got the recurrence right (or some other equivalent representation of it) got most of the points. A few points were taken off if you stored the SCE in the table, and not only a pointer (as this requires string copy operations for filling in each cell, and so a runtime of O(|X||Y|(|X| + |Y|))), or if you did not fill in the margin correctly (some solutions ignored the margin, or incorrectly filled it with zeros).

Some solutions involved a more complicated recurrence with a (implicit or explicit) ‘look-ahead’ that only dropped the wildcard if it foresaw that the next non-wildcard character
matches. Although these solutions involve more cases, and provide for a more complicated solution, they were accepted if they were correct. A possible flaw with some of these solutions is that consecutive wildcards need to be specifically handled (i.e. checking if the previous character matches is not enough if it is a wildcard, one needs to check the previous non-wildcard character). A way around this is to collapse consecutive wildcards (or at least mention why they can be assumed to not exist). Also, because of the look-ahead, the boundary conditions become some complicated, since we need to have special cases in a margin of width two.

Another possible approach is to build a graph, in which each cell in the matrix is a node, and each allowed transition is an edge. Edges have weight zero or one, appropriately. A path from \((0,0)\) to \((|X|,|Y|)\) correspond to a common expansion and so a shortest-path algorithm can be used to find the SCE. Since the resulting graph is a DAG, Bellman’s single-round algorithm can be used (\textit{DAG-Shortest-Paths}, page 536 in the book), yielding an identical run time. In fact, the two approaches perform almost the same operations and are very much equivalent.

Many wrong solutions involved matching up wildcard-surrounded blocks in various greedy techniques.

\textbf{Problem 3. Decycling}

An edge \(e\) of an undirected graph \(G = (V, E)\) is a decycling edge if its removal makes the graph acyclic.

Give an efficient algorithm to compute a list of all the decycling edges in a graph \(G\). Express your running time as a function of \(V\) and \(E\).

\textbf{Solution:}

The key to this problem is to realize that a graph that has decycling edges is special. In particular, an undirected graph that is acyclic is a forest. So removal of a decycling edge leaves a graph that is a forest. Turning that around, a graph will have decycling edges if and only if it is a forest plus one edge.

Recall from lecture and the book that depth-first search on an undirected graph builds a forest, classifying edges as tree edges or back edges (CLR Theorem 23.9). Therefore a graph has decycling edges if and only if the classification of edges by depth-first search yields precisely one back edge. And in the case that there is one back edge, there is clearly one cycle, made up of the back edge and the tree edges between its endpoints. It is obvious that removal of any edge not on this cycle doesn’t break it, so is not decycling, and the removal of any edge on the cycle on the cycle does break it, and since there are no other cycles must be decycling.

Our algorithm is therefore a slight modification to depth first search. When we notice the first back edge we record it; if we notice a second back edge we exit everything and return NIL. At the end of the depth first search, if we didn’t find a back edge we return NIL. If we
did find a back edge \((u, v)\), we use the predecessor variables to read off the cycle. That is, to the end of DFS we add:

\[
\begin{align*}
ans &\leftarrow (u, v) \\
x &\leftarrow u \\
\text{while } (x! = v) &
\text{ add } (x, \pi[x]) \text{ to } ans \\
x &\leftarrow \pi[x] \\
\text{return } ans
\end{align*}
\]

Correctness: It is clear that when there are multiple back edges our algorithm returns NIL, that when there are no back edges our algorithm returns NIL, and that when there is one back edge we report every edge on the one cycle. We argued above that these are the correct things to do.

Analysis: Depth-first search is ordinarily \(O(V + E)\), but we do a little better in this case. Note that we look at tree edges and at most 2 back edges. Since there can only be \(V - 1\) tree edges, this means that we only look at \(V + 1\) edges. So the running time of our depth first search is \(O(V + V + 1) = O(V)\). Note that the extra work we do per edge to check for back edges and write down the first one is \(O(1)\), so it falls under the \(O\), and our extra code to output the cycle looks at each node at most once, so is also \(O(V)\).

Comments on technicalities, for the picky: It is a little unfair to assume that two nodes tells us an edge. That is, we refer to the edge \((x, \pi[x])\), and this is slightly cheating. But instead of having \(\pi[x]\) store the parent node of \(x\), we can have it store the edge \((x, \pi[x])\). We have this information at the time \(\pi[x]\) is set, and then we’ll have the edge, and we’ll still be able to perform our extra loop.

Less efficient solutions: We gave partial credit for solutions that did not achieve \(O(V)\) run time. One such solution is to test each edge individually to see if it is decycling. If this test is performed using an \(O(V + E)\) test such as DFS or BFS, the algorithm runs in \(O(E(V + E))\) time, and the solution received a maximum of 8 points. If the test terminates as soon as a back edge is discovered, however, it can be executed in \(O(V)\) time. This yields a \(O(VE))\) algorithm and was worth a maximum of 10 points. Finally, since any undirected graph with more than \((V)\) edges has no decycling edges, we can first count the edges and terminate immediately if we see too many for total run time of \(O(V^2)\) and a maximum of 12 points.

Linear time algorithms that examined all edges and did not terminate early if too many back edges or cycles were found received a maximum of 17 or 18 points depending on the quality of the rest of the solution. Solutions that did terminate early when too many back edges were discovered, and thus ran in \(O(V)\) time, but were analyzed to run in \(O(V + E)\) time received a maximum of 19 points.

One common mistake was to state without justification that an undirected graph with more than one cycle contains no decycling edges. It was also common to make this statement with the short justification that in such a graph, the removal of any edge can destroy at most one
cycle, leaving at least one cycle behind. This is an inadequate justification since one must also consider the case of an edge that is a member of multiple cycles.

The 20 possible points were divided into three areas with (approximately) 10 points allocated to the algorithm, 5 for the statement of correctness, and 5 for the analysis.

**Problem 4. Streaming Video**

Suppose that you wanted to broadcast video of a live event over the Internet. One of the big problems with doing this today is that parts of the Internet tend to break, and then viewers miss part of the show, and then they give up on Internet broadcasts and go back to their television, which is far more reliable. Your goal, as the streaming video broadcaster, is to pick a set of reliable paths to send the video over, so as to minimize the probability of a failure. Note that once the paths are chosen, they are fixed through the transmission. You don’t know how to switch between alternate paths or merge paths reaching the same destination. Also it is imperative that once you start transmission your video reaches every one. If even one path to one destination breaks down, then regulations force you to bring down your server and all transmission must stop immediately. This is a broadcast failure. Your goal is to choose the best set of paths under all these constraints that minimizes the probability of a broadcast failure. This problem is abstracted below.

You have a network of computers, and for each link $l$ in the network, the probability it will fail on you in the middle of the event is $p_l$. Starting from the source computer $s$, you wish to select a set of paths $P_{st}$, one for every other computer $t$ on the network, where $P_{st}$ is a path connecting $s$ to $t$. Now for every link $l$, with probability $p_l$, the link $l$ fails. Say this happens simultaneously, and independently, for all links in the network. If this results in the one of the links used in one of the paths $P_{st}$ failing, then the entire broadcast is a failure. Give an efficient algorithm to calculate the optimal set of paths, i.e., the one that minimizes the probability of failure.

**Solution:**

We represent the computer network by an undirected, connected graph, in adjacency-list format.

For any set of paths $P_{st}$, each link in the network is either on some path, or it is not. Therefore the task is to find a set $T$ of edges such that:

- for all vertices $t \neq s$, there is a path from $s$ to $t$ using only edges in $T$, and
- the probability that any edge in $T$ fails is minimized.

Given a set $T$ of edges, the probability $f_T$ that some edge in $T$ fails is one minus the probability that all edges stay alive. Since the edges fail independently, the probability of failure is $f_T = 1 - \prod_{e \in T} (1 - p_e)$.

If $T$ contains a cycle, then we can remove any edge on that cycle to get $T'$, and we have $f_{T'} \leq f_T$. Recall also that $T$ must contain a path from $s$ to every other vertex $t$. Combining these two facts implies that an optimal solution will be a spanning tree.
We minimize the quantity $f_T = 1 - \prod_{i \in T} (1 - p_i)$ iff we maximize $\prod_{i \in T} (1 - p_i)$, iff we maximize $\sum_{i \in T} \log(1 - p_i)$, iff we minimize $\sum_{i \in T} -\log(1 - p_i)$. Since $0 \leq -\log(1 - p_i)$, we may assign edge weights $w_i = -\log(1 - p_i)$ and simply run Prim’s algorithm to find the minimum spanning tree. This tree minimizes the probability of failure, as desired.

Finding the actual path $P_{st}$ for any $t$ is a simple matter of running a breadth-first search.

Alternate analysis. Instead of using the “log trick” above, we could directly assign the probabilities $p_i$ as edge weights, and prove a greedy-choice property similar to Theorem 24.1 in CLR. Doing so would prove the correctness of Prim’s algorithm on a graph with edge weights $p_i$. However, it is insufficient to run Prim’s algorithm on such a graph, without proving why the output minimizes the actual probability of network failure.

Common errors. Many students used a modified single-source shortest-path algorithm to compute the paths $P_{st}$. Instead of choosing paths $P_{st}$ with minimal $\sum_{i \in P_{st}} p_i$, they chose paths with minimal $1 - \prod_{i \in P_{st}} (1 - p_i)$. This solution does not meet the stated goal because one edge may be used by multiple paths, which in effect counts that edge’s failure probability multiple times. Consider the graph $V = \{s, a, b\}$ with failure probabilities $p(s, a) = 0.5$, $p(a, b) = 0.5$, $p(s, b) = 0.01$. Then a shortest-paths solution would return the edges $(s, a), (s, b)$ (failure probability = 3/4), while a correct solution is $(s, a), (a, b)$ (failure probability = $1 - 0.5 \times 0.99 = 0.505$). It is interesting to note that this algorithm, however, does minimize the expected number of computers that will be cut-off (but not the probability that some computer will be cut-off, as desired).

Another common error was arguing that the probability of network failure is just the sum of the probabilities of all the chosen edges. This is obviously incorrect, since the sum could be greater than 1 if several edges have large failure probabilities.

Finally, some solutions used network flow algorithms. This is misguided for several reasons, most notably because a maximal flow may push commodity through many different paths to reach a sink, instead of selecting a single path.

Grading standards. Correctly understanding and arguing the minimum spanning tree nature of the problem was worth about 10 points. A correct expression of failure probabilities (whether over the network, or over a path) was worth about 5 points. Finding the “log trick” in the proper context, or alternately, proving the proper greedy-choice property, was worth about 5 points.

Therefore a shortest-paths solution that properly minimized the path failures received about 5 points. A minimum spanning tree solution that minimized the wrong quantity was worth about 15 points. A network flow solution received no points.