15-750: Algorithms in the Real World **CMU, Spring 2023**
HW #2 (out Tuesday Feb 7, 2023) **Due:** Tuesday Feb 21, 2023

Please limit yourself to groups of two (or three) people. We adhere to a "whiteboard" collaboration policy: you should discuss in your group on whiteboards (or equivalents) but then should go away and write down your solutions yourself. You must not share written work. **You must write down the names of the person(s) with whom you collaborate.** In general, you should solve the homework problems using only to material we refer you to (or put on the course page), and not books or other online resources. If you have reason to use any resources we point you to (except the course notes), please cite them.

A word of advice: please try to solve the problem by yourself before working with your group. That is the way to improve your problem-solving skills.

Homeworks will be due at 11:59pm on the due date on `gradescope`. Corrections and changes will appear on the course webpage and on Piazza; please check them regularly.

Problems

1. **Targeted Shortest Path.** This problem describes a variant on Dijkstra's algorithm for finding the shortest path between a source $s \in V$ and a target $t \in V$ in a connected graph $G = (V, E)$ with nonnegative edge-lengths ℓ_e for each $e \in E$. It is not asymptotically faster in the worst case, but it can be a lot faster on many real-world instances, as it often explores much less of the graph. It is the kind of heuristic that is used in road-mapping software for finding shortest driving routes.

   Recall that Dijkstra's algorithm maintains an (over)estimate $d(v)$ of the shortest $s \to v$ path length for each vertex $v \in V$. This is initially $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s$. The algorithm then repeatedly picks the unmarked vertex u with the lowest $d(u)$ value, marks it, and "updates" all of u's unmarked neighbors v by replacing $d(v)$ with $\min\{d(v), d(u) + \ell_{uv}\}$.

   Suppose we have certain *target-estimates* $h_t(v) \geq 0$, which are supposed to be estimates of the shortest-path distances from each vertex v to the target t. (For now, don't be concerned with where these come from; just imagine we have them somehow.) Then the "targeted-Dijkstra" algorithm is the variation where at each step the unmarked vertex u with lowest value of $d(u) + h_t(u)$ is picked (and processed as before). (The algorithm may also stop once it has marked t.)

   Remark: If $h_t(v)$ magically equalled the shortest-path distance from v to t, then it's not hard to see that the algorithm will make a beeline for t. On the other hand, if $h_t(v) = 0$ for all v, then targeted-Dijkstra simply turns into the usual Dijkstra's algorithm.

   (a) Give an example showing that if the target-estimates are chosen poorly, the algorithm might not correctly compute shortest

   (b) Call a target-estimate function $h : V \to \mathbb{R}$ *sensible* if $h(v) \leq \ell_{vw} + h(w)$ for all edges $(v, w) \in E$. Prove that with a sensible target-estimate function, the targeted-Dijkstra algorithm is correct, in that once a node u is marked, $d(u)$ is the correct shortest-path distance from s to u.

   (c) We now know that as long as we have sensible target-estimates, we will get a correct algorithm. But how should we get these? Think back to the setting of software that is used to find shortest driving routes in a fixed large graph (like the road network of a state or a country). Suppose do the following:

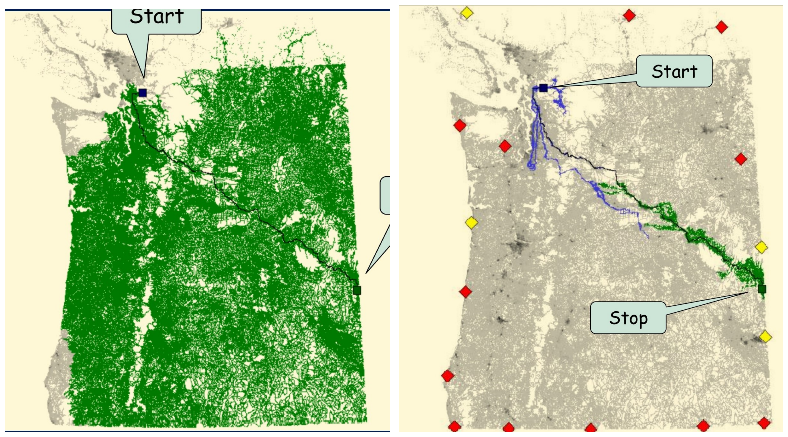
i. Fix a few dozen "landmarks" $L \subseteq V$ in the graph.

ii. For each vertex $u \in V$, precompute and store the exact shortest distance $D(u, a)$ from u to a for each landmark $a \in L$.

Now when we are tasked with finding the shortest path $s \to t$, and we need a target-estimate function h_t, we use $h_t(v) = \max_{a \in L}\{D(v, a) - D(t, a)\}$. Show that these target-estimates are sensible.

(Hint: first try the case when $|L| = 1$.)

By precomputing and storing these few dozen numbers for each vertex, the result is a correct shortest $s \to t$ path algorithm that is often much more efficient in practice than plain Dijkstra. Here are some pictures I found that give an example meant to illustrate how much the search-space is reduced; the red diamonds are the landmarks.



2. **On the Up and Up.** Madeline likes to jump upwards on platforms. The input to this problem is a sequence of n positive integers $p_1, p_2, \ldots, p_n$ (not necessarily distinct), representing the heights of a succession of platforms. Madeline starts to the left of all platforms at height 0. She would like to jump on as many platforms as she can, but the rules are that each jump can only go forwards (as far as she likes) and she can never go downward in height. So if the input sequence p is

$$3, 6, 17, 3, 9, 2, 19, 13, 4, 2$$

then one optimal plan for Madeline is to jump on the first 3, then on the next 3, then on the 9, and then on the 19, for a total of four jumps. (In this example there are several equally good 4-jump plans, such as $3, 6, 17, 19$, or $3, 3, 9, 13$.)

(a) Describe an algorithm that uses $O(n \log n)$ time and $O(n)$ extra space (beyond the input array) that finds the maximum number of jumps Madeline can do. You don't have to output an optimal plan, just the optimal number of jumps. (Here and in all future problems, when I say "Describe" I mean give the algorithm, and justify its correctness and time/space analysis.)

(b) *(Extra credit.)* Given the same kind of input sequence, suppose there are *two* people, Madeline and Badeline, who both jump according to the preceding rules, but who hate each other and so they cannot be on a platform the other has used (even at different times). Describe an algorithm that uses $O(n \log n)$ time and $O(n)$ extra space that computes the highest possible combined number of jumps the two can do. You don't have to output the plans, or even the number of jumps done by each, just the highest possible combined jump-total. (In the above example, the answer is 7, which can be achieved by having Madeline jump on the first 3 followed by $6, 17, 19$, and having Badeline jump on the second 3 followed by $9, 13$.)

(c) *(Extra credit.)* Back to just Madeline: Solve problem (a) but with an algorithm that uses $O(n^{1.5} \log n)$ time and only $O(\sqrt{n})$ extra space (beyond the input array).

3. **Counting Unique Elements.** In the algorithmic paradigm known as "streaming", we imagine that the inputs are arriving one by one in a "stream", and we want to use extremely little time and space to process each one, while still being able to come up with a good answer about the whole stream in the end.

In this problem, the elements streaming in are strings $s_1, s_2, \dots, s_n$. What you would really like to know is how many *distinct* strings are in the sequence. Since you really want to use extremely little time and space, you are willing to tolerate a somewhat inaccurate count. In fact, imagine you initially have in mind a fixed positive integer K (assumed even), and you just want to know if the number of distinct strings in the stream is "Low" (meaning at most $K/2$) or "High" (meaning at least $2K$).

Here's your method. Initially, you select a hash function $h : \{\text{all strings}\} \to \{0, 1, 2, \dots, 2K-1\}$. You may assume "SUHA". Now when a string s_t comes in, you compute $h(s_t)$. But actually you won't even allocate a hash table, you'll just use a single bit of memory: if ever you see a string that hashes to 0 you will output "High", and if you never see a string that hashes to 0 you will output "Low". (Note that you do not know in advance what "n" will be; once elements stop arriving is when you give your output.)

(a) Show that if the number of distinct elements is indeed "Low" (at most $K/2$), then

$$\mathbf{Pr}[\text{you wrongly output "High"}] \le 1/4.$$

(b) Show that if the number of distinct elements is indeed "High" (at least $2K$), then

$$\mathbf{Pr}[\text{you wrongly output "Low"}] \le 0.4.$$

(You may use that $1 - x \le 1 - x + x^2/2! - x^3/3! + x^4/4! - \cdots$ for all $x \ge 0$.)

(c) We can summarize the above by saying that the "error" probability is at most 0.4. (It's actually noticeably smaller on the "false-High" side, but never mind.) That's actually a pretty high error rate, so let's try to make it smaller as follows: We will instead run m copies of the above algorithm in parallel, using m "independent" hash functions $h_1, \dots, h_m$ and m bits of storage. (Assume m is an odd positive integer.) Then at the end, we output whatever the *majority* answer is among the m answers these runs would give. Prove that now the overall error probability is exponentially small in m; say, at most $O(0.98^m)$. *(Hint: One elementary way to do this involves reasoning through the following: for $k \ge m/2$, the value $\binom{m}{k}(0.4)^k(0.6)^{m-k}$ only gets smaller the larger k is (why?) and also $\binom{m}{k} \le 2^m$ (why?) and $2\sqrt{.4}\sqrt{.6} < 0.98\dots$)*

4. **TLS/SSL Certificate Revocations.** A common problem that web browsers face is checking if a certificate has been revoked. Let U be the set of all possible TLS Certificates and let $R \subset U$ be the set of certificates that have been revoked. The set of revoked certificates changes over time, so your goal is to periodically broadcast the current set to all web browsers. Ideally,

you would like to represent R using a data structure D that is both compact and fast at answering membership queries. Compactness is important for minimizing network traffic and speed is important for efficiency on the browser side.

Bloom filters are a natural choice for storing R, but we need a way to handle false positives (i.e., certificates that are identified as revoked even though they are not). Assuming we have access to both R and U, one idea is to store all the false positives from the first Bloom filter B_1 in a second Bloom filter B_2, and then storing all the false positives from B_2 in a sorted array L. More precisely, B_2 stores the set of certificates in $U \setminus R$ for which B_1 returns `true` and L stores the set of certificates in R for which B_2 returns `true`.

For this problem, you may operate at the same level of rigor that we did in lecture; that is, you may assume SUHA, that $1 + x \approx e^x$ for tiny x, and that fraction of bits set to 0 in a Bloom filter is equal to its expectation. Also, you may assume that the parameters k, k', m, m' are all integers (just for the simplicity of not having to write ceilings and floors), and that computing hash functions takes constant time.

(a) State an algorithm that, given $u \in U$, checks whether $u \in R$ using the multi-level Bloom filter described above. Your algorithm should have no false positive or false negatives, and should do the sensible Algorithms 101 thing when it accesses L. You do not have to analyze the algorithm further here; just stating it is sufficient.

(b) Let $r = |R|$, $n = |U|$ and $s = n - r$. For the remaining parts of this problem, suppose the Bloom filter B_1 uses $k = \log_2(s/r)$ hash functions and an array of $m = \frac{rk}{\ln 2}$ bits. Determine (as a function of r, n, and s) the probability of a false positive in B_1; that is, the probability for $u \in U \setminus R$ that B_1 will return `true` on u.

(c) Let r' be the number of certificates stored in B_2. Determine the expected value of r'.

(d) The number of revoked certificates is usually much smaller than the total number of certificates, so henceforth assume $r = \Theta(\sqrt{s})$. Also suppose we pick parameters for B_2 so that it uses $k' = k = \log_2(s/r)$ hash functions with an array of $m' = \frac{r'k}{\ln 2}$ bits. Determine the expected size of L, and also show that the running time of your algorithm from part (a) is $O(\log r)$. *(Hint: you may need that the average of a log is at most the log of an average; it's basically the upside-down version of this.)*