1 Single Source Shortest Path

The input for an instance of the SSSP is a connected weighted graph $G$ of $n$ vertices and $m$ edges, positive integer weights (representing lengths) on the edges, and a particular vertex $v_0$ marked as the source. The output assigns to each vertex its shortest distance from $v_0$.

(a) [10 points] Show that $\Omega(m + n \log n)$ is a lower bound on the running time of the Dijkstra’s algorithm for SSSP.

(b) [Extra Credit] Is $\Theta(m + n \log n)$ the optimal time for any algorithm solving the SSSP problem?

(c) [10 points] Given a supposed solution to the SSSP problem (the distances from the source to all vertices are specified), verify if the solution is correct in $O(m + n)$ time.

\[\text{SOLUTION:} \text{ (a) In the case where } n \log n = o(m), \text{ the running time of Dijkstra is } O(m). \text{ This is optimal because any algorithm would need to read all edge weights before computing the answer. Now we only need to show that the cost of Dijkstra is at least } O(n \log n). \text{ An instance of sorting } n \text{ numbers can take as much } O(n \log n) \text{ time. If we manage to reduce any instance of the sorting problem to an instance of Dijkstra’s, then the same lower bound applies to Dijkstra’s.} \]

\[\text{Reduction: given a set of } n \text{ numbers to be sorted, construct a star graph with } n \text{ edges and assign the numbers as the lengths of these edges. Mark the internal node as the source. Since Dijkstra’s algorithm discovers vertices (in this case leaves) in the increasing order of their distances, and there is only path to a leaf edge, tracking the order in which the leaves are added to the discovered set gives us the order of the } n \text{ numbers.} \]

\[\text{(c) Notation: Suppose } d_v \text{ is the actual shortest distance from } v_0 \text{ (source)} \]**
to \( v \), and \( D_v \) denotes the shortest distance proposed by the solution to be verified. Let \( l_{u,v} \) be the length of edge \((u,v)\). We perform the following check.

**Check:** For every vertex \( v \neq v_0 \), check that 
\[
d_v = \min_{u \in N(v)} \{d_u + l_{u,v}\},
\]
where \( N(v) \) is the set of vertices adjacent to \( v \). Takes time \( O(m + n) \).

**Claim:** If check holds, then \( D_v \leq d_v \) for all vertices.

**Proof:** Suppose not. Then there exists \( v \) such that \( D_v > d_v \). Consider a shortest path from \( v_0 \) to \( v \). Suppose that this path is \( v_0, v_1, \ldots, v_k, v \), and the lengths of edges along this path are \( l_0, l_1, \ldots, l_k \) in the same order. Note that 
\[
d_v = l_0 + l_1 + \cdots + l_k.
\]
Since the solution passed the check, \( D_{v_k} > d_v - l_k \). Also 
\[
D_{v_{k-1}} > d_v - l_k - l_{k-1},
\]
Continuing further, \( D_{v_1} > d_v - (l_1 + l_2 + \cdots + l_k) > l_1 \) which is a contradiction.

**Claim:** If check holds, then \( D_v \geq d_v \) for all vertices.

**Proof:** Suppose there exists \( v \) such that \( D_v < d_v \). If the check succeeds, then it is possible to find a path from \( v_0 \) to \( v \) (just follow back the edges that determined the min starting from \( v \)) of length only \( D_v \). This can not possible be the case since all paths from \( v_0 \) to \( v \) are at least \( d_v > D_v \) long. Therefore, if \( D_v < d_v \), the check has to fail somewhere.

From the above two claims, we infer that if the proposed solution passes the check, \( D_v = d_v \) for all vertices. Also note that every valid solution passes this check.

## 2 Median Data Structure

(a) [10 points] Suppose that we wanted a data structure to maintain a set of elements and quickly query for the median of the set (in case of even sized set of elements, call either of the two middle elements a median). We want to support the following operations with the specified amortized costs:

- **Build\((n)\):** Build a data structure consisting of \( n \) elements in time \( O(n) \).
- **makeDS\((x)\):** Build a data structure consisting of 1 element in time \( O(1) \)
- **findMedian\((D)\):** Find the median of \( D \) in time \( O(1) \)
- **insert\((x, D)\):** Insert element \( x \) in to a structure \( D \) consisting \( n \) elements in time \( O(\log n) \)
• \textbf{deleteMedian}(D): Delete the median of \(D\) in time \(O(\log n)\).

(b) [2 points] Build a data structure that supports all the above operations, but with the specified costs being worst case instead of amortized.

(c) [8 points] Is it possible to improve the amortized cost of the \texttt{insert} operation to \(O(1)\), all other requirements being the same? If not, why?

\textbf{SOLUTION:} The high level idea is that the data structure with \(n\) elements is the triplet \((m, L, H)\) where \(m\) is a median, \(L\) is a max-heap consisting of all the elements not larger than median and \(H\) is a min-heap consisting of all the elements not smaller than the median. We also maintain the data structure such that \(L\) and \(H\) differ in size by atmost 1. Its sufficient to implement the heap using binary heaps here.

• \textbf{Build}(n): Find the median in \(O(n)\) time, separate the smaller and larger set and build two heaps in \(O(n)\) time.

• \textbf{makeDS}(x): The data structure is just \((x, \emptyset, \emptyset)\).

• \textbf{findMedian}(D): If \(D = (m, L, H)\), return \(m\). From here on we denote the

• \textbf{insert}(x, D): Compare \(x\) to \(m(D)\). If it is equal, insert it in to the smaller of \(L\) and \(H\) (if both \(L\) and \(H\) are of same size, pick one randomly). If \(x\) is larger than \(m(D)\), insert \(x\) in to \(H\). If the operation makes \(|H| = |L| + 2\), then insert \(m(D)\) in to \(L\), remove the min element of \(H\) and assign it to \(m(D)\). This operation rebalances \(H\) and \(L\) while maintaing the invariants described above. The case where \(x < m(D)\) is handled similarly. Since we perform at most 2 insert operations and one deleteMin operation on heaps of size \(n/2\), the whole operation takes \(O(\log n)\) time.

• \textbf{deleteMedian}(D): Remove \(m(D)\). If \(|L| \leq |H|\), then do a deleteMax operation on \(L\) and use the root of \(L\) as the new median. Similar when \(|H| > |L|\).

(b) Note all the above costs are worst case too.

(c) Suppose that \(O(1)\) inserts were possible, we will show that sorting can be done in \(O(n)\) time. To sort a set \(A\) of \(n\) numbers, we construct a set \(L\) of \(n\) numbers that are smaller than elements of \(A\), and insert \(A \cup L\) in to the
median data structure one by one. This takes $O(n)$ time. Now we come up with another set $U$ of $2n$ numbers, each of them larger than numbers in $A$. Now, we alternately insert an element (or two) from $U$ in to the median data structure and query for the median. After inserting one element from $U$, the `findMedian` returns the least element of $A$. After inserting 3 elements from $U$, the `findMedian` returns rank 2 (among elements of $A$) element of $A$, and so on. All inserts take $O(n)$ elements, and we end up with the rank of each element in $A$.

### 3 Union-Find

Lecture 10 (Kozen’s book) on Union-Find uses two heuristics to improve the performance: (a) in an union, always merge the smaller tree into the larger, and (b) in a find, use path compression. Consider a sequence of $m$ union and find operations on a set of $n$ elements.

(a) [5 points] If we do not perform path compression and randomly pick one of the two elements to link to the other when we perform an union, what is the worst-case (over all sequences of operations) average cost (averaged over all choices of tree merges in union).

**SOLUTION:** Consider the sequence: union($x_1, x_2$), $x = \text{find}(x_1)$, union($x, x_3$),$x = \text{find}(x_1), \ldots, $union($x, x_n$), $x = \text{find}(x_1)$. The cost is $\Omega(mn)$ in worst case, because the depth of $x_1$ during the $i$th union-find pair can go up approximately $i/2$.

(b) [15 points] If we perform path compression but still randomly pick one of the two elements to link to the other when we perform an union, what is the worst-case (over all sequences of operations) average cost (averaged over all choices of tree merges in union).

**SOLUTION:** The answer is $O((m + n) \log n)$.

Each element $x$ in the union-find data structure is assigned potential $\log(s(x))$ where $s(x)$ is the number of elements in $x$’s subtree (the maximum potential is $O(n \log n)$ and the minimum potential is 0).

The amortized cost of an arbitrary union operation is at most $1 + O(\log n)$. When performing an arbitrary union, the only element whose subtree changes is the resulting root. The maximum change in potential is $O(\log n)$. And the unit cost of linking two roots together is 1.
The amortized cost of an arbitrary find is $2 + O(\log n)$. Suppose we traverse the path $(x_1, \cdots, x_k)$ to the root of $x_1$'s tree, when performing an arbitrary find operation on element $x_1$. No element's potential increases, and the change in potential at $x_i$ for $i \in \{2, \cdots, x_{k-1}\}$ is $\log(s(x_i) - s(x_{i-1})) - \log(s(x_i))$. If $s(x_i) \geq s(x_{i+1})/2$, then $\log(s(x_{i+1}) - s(x_i)) - \log(s(x_{i+1})) \leq \log(s(x_{i+1}/2) - \log(s(x_{i+1})) = -1$, so that we recapture one unit of potential to pay for the traversal and linking of $x_{i+1}$. If $s(x_{i+1}) > 2s(x_i)$ (let us call $x_i$ light if so), then $x_{i+1}$ may not recapture much potential, but there are at most $\log n$ light children on any path to a root of a tree because $x_i$ is light and no element’s size is greater than $n$. And the cost incurred from traversing and linking $x_1$ and $x_k$ is 2.

Thus the amortized cost of any arbitrary union or find is $O(\log n)$. 