CS41 Homework 4

This homework is due at 11:59PM on Tuesday, November 3. Write your solution using LaTeX. Submit this homework using **github** as a **.tex** file. Please do not include your name in the file you submit.

It's ok to discuss approaches at a high level with others. However, you should not reveal specific details of a solution, nor should you show your written solution to anyone else.

The main learning goals are to practice greedy algorithms.

> Any time you are asked for an algorithm, you should include a description of your algorithm, a proof of correctness, and a runtime analysis. This instruction holds for homeworks for the entire semester (unless the problem specifically says you do not need to include those proofs).

1. **Shortest Paths with Small Integer Weights.** Suppose we are given a directed graph $G = (V, E)$ with edge weights $\ell(e) > 0$ for all $e \in E$. Suppose we also have some integer $k > 0$ and we know that all edge weights are positive integers no larger than k, i.e., $\ell(e) \in \{1, 2, \ldots, k\}$ for all $e \in E$.

   As usual, we use n to denote the number of vertices, and m to denote the number of edges.

   (a) Suppose $k = 3$, so that all edge weights are either 1, 2, or 3. Design and analyze an algorithm that, given G and a starting vertex $s \in V$, computes the length of a shortest path from s to every other vertex. Your algorithm should run in time $O(m + n)$.

   (b) Now consider the general case, where k may be any positive integer. Extend your algorithm from the previous part to handle this, and analyze its running time *in terms of* k, m, and n.

   (c) Recall that Dijkstra's Algorithm, using a priority queue with a binary heap implementation, runs in time $O(m \log n)$. For which values of k is your algorithm from the previous part faster than Dijkstra's?

2. **Prim's Algorithm Running Time.** Recall that Prim's Algorithm can be viewed as a "generic search", building a tree by adding one vertex at a time to a "known region" R via an edge crossing the cut $(R, V \setminus R)$.

   Argue that Prim's Algorithm can be implemented in time $O(m \log n)$ by giving pseudocode that includes details on what data structures to use, and then showing that the total running time of all operations is $O(m \log n)$. You do not need to prove correctness of your pseudocode, as long as it works and you can show it achieves the desired running time.

3. **Kruskal's Algorithm and Union-Find.** Below is pseudocode for Kruskal's Algorithm using the Union-Find data structure.

```
KRUSKAL(G, c)
    // Initialize the Union-Find disjoint sets data structure with the vertices V
1   CREATEDISJOINTSETS(V)
2   F = ∅
3   for each edge e = {u, v} ∈ E, in increasing order of weight:
4       T1 = FIND(u); T2 = FIND(v)
5       if T1 ≠ T2
6           add e to F
7           UNION(T1, T2)
8   return F
```

The Union-Find data type consists of three operations: CREATEDISJOINTSETS initializes the disjoint-sets data structure from a single set, by placing each element in its own "singleton" set (so CREATEDISJOINTSETS(V) creates one singleton set for each vertex); FIND(u) returns the set containing u; and UNION merges two sets into one combined set.

In class, we saw the main idea behind the implementation of the Union-Find data structure. For reference, pseudocode for a basic implementation is provided at the end of the homework, though you don't need it to answer the below questions. In that basic implementation, the running times of the three operations are:

- CREATEDISJOINTSETS: Given a collection of n elements, runs in time $O(n)$.
- UNION: Runs in constant time $O(1)$.
- FIND: With n elements, may run in time $\Theta(n)$ in the worst case.

Two improvements, *union by rank* and *path compression*, help speed up the running time of FIND, which we will consider in this problem:

(a) In the basic version (as in the pseudocode at the end of the homework), FIND may take time $\Omega(n)$ in the worst case. With this implementation, what is the worst-case running time of Kruskal's Algorithm? Justify your answer by counting the number of times each operation is performed.

(b) **Union by rank** improves the running time of FIND to $O(\log n)$ in the worst case. What is the worst-case running time of Kruskal's Algorithm with this improvement? Justify your answer by counting the number of times each operation is performed.

(c) **Path compression** adds a further improvement to FIND, that improves its *amortized* running time to $O(\alpha(n))$, where $\alpha(n)$ is an extremely slowly growing function called the *inverse Ackermann* function ($\alpha(n) \leq 4$ for any n that could possibly be encountered in practice — e.g., this is true even when n is the number of atoms in the observable universe). You may recall from CS35 that *amortized* running time does not guarantee a specific running time on a single operation, but rather on a sequence of operations; thus, while a single FIND may still take $O(\log n)$ time (using union by rank), the amortized running time guarantees that with n elements, any sequence of k FIND operations takes time $O(k\alpha(n))$.

For this part, assume that the edges of E are given to us already sorted in increasing order of their weight. What is the total running time of Kruskal's Algorithm with this improvement? Again, justify your answer briefly by counting the individual operations.

(d) What is the total running time of Kruskal's Algorithm using the same path compression improvement as in the previous part when the edges are *not* pre-sorted and must first be sorted (e.g., with mergesort) in increasing order of weight?

(e) How does this compare to Prim's Algorithm? When would you choose to use Kruskal's Algorithm over Prim's, or vice versa?

(f) **(Optional, Extra challenge)** For the basic implementation of Union-Find, give an example of a sequence of UNION and FIND operations where subsequent FIND operations may require $\Omega(n)$ time in the worst case. Describe a family of graphs with edge weights that leads to an asymptotically worst-case running time for Kruskal's Algorithm with this basic implementation.

4. **Placing Cell Towers (K&T 4.5)** Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

   Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Reference: Union-Find Pseudocode

The basic version of the union-find data structure we mentioned briefly in class, for which FIND may take time $\Omega(n)$ in the worst case, is presented below. In this implementation, each element is a node in one of many rooted trees (NB: this is very different from the "tree" of the MST problem), and for each element x we keep track of its parent $\pi(x)$ in its tree. By convention, a root node's parent is itself.

The initialization sets each element's parent to itself, so that each one is in its own tree (and thus is a "singleton"):

```
CreateDisjointSets(S)
1  for each element x ∈ S
2      π(x) = x
```

Finding an element amounts to following the parent pointers until we reach the root node, which is used as the "representative" for its set:

```
Find(x)
1  if π(x) == x
2      return x
3  return Find(π(x))
```

And merging two sets is accomplished by setting the root node of one tree to be the parent of the other:

```
Union(T1, T2)
1  // T1 and T2 must be roots of different trees
2  π(T1) = T2
```