**ASSIGNMENT 2: Solutions** CMSC 451 (Section 0101, Fall 2020)

1. *Gears.* [20 points]

   Suppose you are given a large collection of $n$ circular gears. The gears all operate in the same plane; each rotates about a fixed center position with its teeth pointing outward. For each pair of gears, you are told whether or not they mesh (so that they will turn together). One of the gears has a crank on it. Design an efficient algorithm that will determine (i) whether it is possible to turn the crank without breaking the gears, and (ii) whether *all* the gears move when the crank is turned. Prove that your algorithm is correct and determine its asymptotic running time.

   **Solution:**

   (i) Gears that mesh must turn in opposite directions (one clockwise and the other counterclockwise). We represent the gears as vertices and introduce an edge between two vertices if the corresponding gears mesh. Let $s$ denote the gear with the crank. Let $S$ be the component that $s$ belongs to. We claim that the gears can turn if and only if $S$ is bipartite. If $S$ has vertex bipartition $(A, B)$, then all the gears in $A$ can turn in one direction (say, clockwise) and the gears in $B$ can turn the other direction (counterclockwise). On the other hand, suppose $S$ is not bipartite. Then, by the characterization discussed in class, it contains an odd cycle of gears $g_1, g_2, \ldots, g_{2k+1}$. To turn freely, gears $g_1, g_3, \ldots, g_{2k+1}$ must all turn in the same direction, but $g_1$ and $g_{2k+1}$ must turn in opposite directions, so in fact the gears cannot turn freely. Having established this characterization, we can use BFS to determine whether $S$ is bipartite by looking for a non-tree edge bewteen vertices at the same level of the search tree, as discussed in class. Using the given adjacency matrix representation, we can perform BFS in time $O(n^2)$, since for each vertex it suffices to consider each of its $O(n)$ potential neighbors once.

   (ii) To determine whether all the gears turn, we just have to find whether the graph described above is connected (i.e., whether the component $S$ is the entire graph). BFS also solves this in time $O(n^2)$.

2. *Route planning with variable travel times.* [20 points]

   Suppose you would like to navigate a system of roads, represented by a directed graph $G = (V, E)$. Because of variable traffic and weather conditions, the time required to travel along an edge varies with time. Fortunately, you are given access to a procedure that can reliably predict how long it will take to travel along an edge at any chosen starting time. Given an edge $e = (u, v) \in E$ and a starting time $t$, the procedure returns the amount of time $r_e(t) \geq 0$ that it will take to travel from vertex $u$ to vertex $v$ along edge $e$, assuming you leave $u$ at time $t$. The travel times have the property that if $t' > t$, then $t' + r_e(t') > t + r_e(t)$ (i.e., if you leave later, you will arrive later), but are otherwise arbitrary. The running time of the procedure is $O(1)$. Design a polynomial-time algorithm to find the fastest way to get from a given starting vertex to a given destination vertex, starting at time 0. Prove that your algorithm is correct and analyze its running time. (Hint: consider adapting Dijkstra's algorithm to this setting.)

   **Solution:** Dijkstra's algorithm can be modified slightly to handle time-varying edge lengths $r_e(t)$. Let $s$ denote the designated starting vertex. Algorithm 1 shows the complete algorithm.

Initially $S = \{s\}$ and $d(s) = 0$ and $\text{prev}(s) = \varnothing$;
**while** $S \neq V$ **do**
    Select a node $v \notin S$ for which $d'(v) = \min_{e=(u,v):\, u \in S} d(u) + r_e(d(u))$;
    Add $v$ to $S$;
    Set $d(v) = d'(v)$;
    Set $\text{prev}(v) = u$ where $u$ is a neighbor of $v$ achieving the minimum $d'(v)$;
Let $w$ be the destination vertex and $P^* = \varnothing$;
**while** $w \neq s$ **do**
    $P^* = \text{prev}(w) + P^*$;
    $w = \text{prev}(w)$;
Return $P^*$;

**Algorithm 1:** Modified Dijkstra's Algorithm

*Correctness:*

**Claim 1.** *At any point in the algorithm's execution, for any vertex $u \in S$, $d(u)$ denotes the earliest time required to reach $u$ from $s$.*

*Proof.* We prove this by induction on the size of $S$. The base case of $|S| = 1$ is easy as we have $S = \{s\}$ and $d(s) = 0$. Let us assume that the claim holds when $|S| = k$ for some value of $k \geq 1$. We now grow $S$ to size $k + 1$ by adding the node $v$ and let $(u, v)$ be the final edge considered, i.e., $u = \text{prev}(v)$.

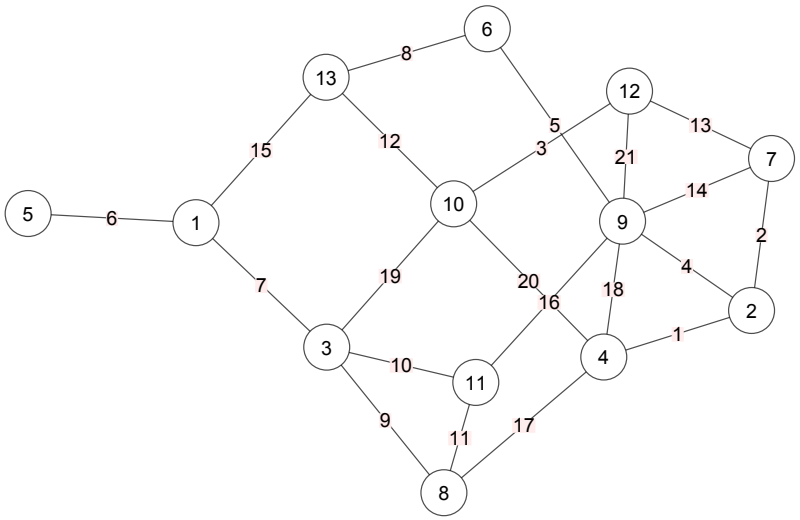
By the induction hypothesis, $d(u)$ is the earliest time at which we can reach $u$ from $s$. Since the variable edge lengths satisfy the property that if $t' > t$, then $t' + r_e(t') > t + r_e(t)$, we are guaranteed that $d(v) = d(u) + r_e(d(u))$ is the earliest we can reach $v$ via a path that contains edge $(u, v)$. Now consider any other path $P$ from $s$ to $v$; we need to show that following any such path, we cannot reach $v$ before $d(v)$. Let $y \notin S$ be the first vertex on $P$ that is not in $S$ and let $x \in S$ be the node just before $y$. In the current iteration, since we chose to add vertex $v$ to $S$ instead of $y$, we know that $d(x) + r_{(x,y)}(d(x)) \geq d'(y) \geq d(v) = d(u) + r_{(u,v)}(d(u))$. Since we have $r_e(t) \geq 0$ for all edges $e$ and times $t$, we are guaranteed that following path $P$, the time required to reach $v$ is at least $d'(y) \geq d(v)$. This completes the proof by induction. □

The actual path that needs to be taken to reach the destination by the earliest time is obtained by traversing back through the edges chosen by the algorithm. $P^*$ denotes such an optimal path.

*Running Time:* Just as for Dijkstra's algorithm as described in Kleinberg and Tardos (4.15), the above algorithm can be implemented to run in $O(m \log n)$ time using a priority queue.

3. *Minimum spanning tree.* [15 points]

Run Kruskal's algorithm to find a minimum spanning tree of the following weighted graph.




In addition to showing the minimum spanning tree produced as output, indicate the order in which the edges are added to the tree. (*Note:* You should either run the algorithm by hand, or implement it in a programming language of your choice. You may *not* use someone else's implementation.)

**Solution:** The edges are added in the order $\{2,4\}, \{2,7\}, \{10,12\}, \{2,9\}, \{6,9\}, \{1,5\}, \{1,3\}, \{6,13\}, \{3,8\}, \{3,11\}, \{10,13\}, \{1,13\}$; the output is the tree with these edges.

4. *Scheduling fire drills.* [20 points]

Suppose you are given a set of intervals $[s_i, f_i]$ specifying starting and finishing times for classes indexed by $i \in \{1, 2, \ldots, n\}$. You would like to schedule fire drills at times $t_1, t_2, \ldots, t_k$ so that a drill happens during every class (i.e., for all $i \in \{1, 2, \ldots, n\}$ there is a $j \in \{1, 2, \ldots, k\}$ such that $t_j \in [s_i, f_i]$). It is okay if more than one drill is scheduled during a given class, but you would like to minimize the total number of fire drills (i.e., the value $k$). Design an efficient algorithm for this problem, prove its correctness, and analyze its running time.

**Solution:** *The algorithm:*
The algorithm schedules a drill at the earliest finishing time of any class, deletes all intervals it overlaps from the list, and repeats until the list is empty. Algorithm 2 presents an optimized implementation of this.

```
1 sort the intervals by finishing time;
2 let f_prev = −∞;
3 for i = 1 to n do
4     if s_i > f_prev then
5         schedule a fire drill at f_i;
6         f_prev = f_i;
```

**Algorithm 2:** Fire drill scheduler

*Correctness:*
First we show that every class must have a fire drill scheduled during its interval. For a class to not have a fire drill scheduled at its finishing time, it must already have a fire drill scheduled

at some point within its interval. The algorithm iterates over all classes, so when it concludes, each class must have a fire drill scheduled during its interval.

Now we must show that this is the least number of fire drills that suffice to achieve this task. Suppose the greedy algorithm produces a schedule $G = (g_1, \ldots, g_\ell)$ (with drill times listed in increaseing order), and there exists a schedule $B = (b_1, \ldots, b_m)$ that uses the minimal number $m \leq \ell$ of drills (again listed in increasing order).

We know that $b_1 \leq g_1$ because otherwise $B$ would not have a fire drill in the first class. Now suppose that for all $i \leq k$, $b_i \leq g_i$. Then we claim that $b_{k+1} \leq g_{k+1}$, since otherwise $B$ would not have any fire drills in the interval $(g_k, g_{k+1}]$, and therefore the class that caused the greedy algorithm to schedule a drill at time $g_{k+1}$ (which starts after $g_k$ and ends at $g_{k+1}$ by construction of the greedy schedule) would not have a drill in $B$. Thus by induction, $b_k \leq g_k$ for all $k \leq m$.

Finally, we claim that the greedy algorithm must terminate after assigning $g_m$. This follows because $b_m \leq g_m$ is the latest fire drill in $B$, so there can be no intervals that begin after $b_m$, and therefore the greedy algorithm will not schedule any more drills after scheduling $g_m$. Therefore $\ell = m$, which means that our greedy algorithm finds an optimal solution.

*Running time:*
This algorithm first sorts all of the finishing times (in time $O(n \log n)$) and then iterates over all $n$ intervals, taking a constant amount of time to process each. The latter loop takes time $O(n)$, so the overall running time is $O((n \log n) + n) \in O(n \log n)$.

5. *Finding a majority.* [20 points]

Given an $n$-element array $A$, a majority element is one that appears *strictly more than* $n/2$ times. Note that the majority element is unique if it exists. Your task is to design an efficient algorithm to tell whether a given array $A$ has a majority element, and if so, to find that element. The elements of $A$ are large and complicated, so you cannot easily make comparisons of the form "is $A[i] \geq A[j]$?". However, you can answer questions of the form "is $A[i] = A[j]$?" in constant time.

Show how to solve this problem in $O(n \log n)$ time. Prove correctness and the bound on the running time.

**Solution:** If $|A| = 0$, then there is no majority; we indicate this by returning null. If $|A| = 1$, we simply return $A[1]$. Otherwise, we split the given array $A$ into $A_1$ and $A_2$ by putting the first $\lfloor |A|/2 \rfloor$ elements into $A_1$ and the rest into $A_2$. We then recursively determine the majority of both $A_1$ and $A_2$. Suppose $x_1$ and $x_2$ are elements returned from running our algorithm on $A_1$ and $A_2$, respectively. We then check whether either $x_1$ or $x_2$ is a majority element in linear time by comparing them with all elements in $A$. If one of them is a majority, then we return it. If neither of them is an overall majority, we conclude that $A$ does not have a majority, and we return null.

*Correctness:* Assuming our algorithm works properly for the subproblems, we expect that a majority element, if it exists, must be either $x_1$ or $x_2$. We prove this in Claim 2 below. In the base case where we have only one element, it is obvious that this element is the majority. Our recursion reaches the base case $|A| = 1$ eventually, so our algorithm must be correct.

*Running time:* We take linear time to divide our array of size $n = |A|$ into two subarrays and process them separately. The merge step, as described above, can also be done in linear time. Hence our running time $T(n)$ satisfies the recurrence

$$T(n) = 2T(n/2) + O(n).$$

Thus the algorithm runs in $O(n \log n)$ time.

**Claim 2.** *Suppose $x$ is the majority of $A$. Then $x$ must be the majority of either $A_1$ or $A_2$ (or both).*

*Proof.* $A_1$ has size $\lfloor n/2 \rfloor$ and $A_2$ has size $\lceil n/2 \rceil$. If $x$ is not a majority element of $A_1$ or $A_2$, then the number of times $x$ can appear in $A$ is at most $(1/2)\lfloor n/2 \rfloor + (1/2)\lceil n/2 \rceil = (1/2)(\lfloor n/2 \rfloor + \lceil n/2 \rceil) = n/2$, so it is not a majority element of $A$. □

6. *Collaboration.* [5 points]

Write "I understand the course collaboration policy and have followed it when working on this assignment." List the other students with whom you discussed the problems, or else indicate that you did not discuss any problems with your classmates.