Problem 1

Let $G = (V, E)$ be a connected graph and $g, h$ be two edge-cost functions such that for all $e, e' \in E$, $h(e) \leq h(e') \iff g(r) \leq g(e')$. Show there is a single tree which is an MST with respect to $g, h$.

Solution: We simply show that any tree constructed by Kruskal’s algorithm using the function $g$, is also a valid output of Kruskal’s algorithm using the function $h$ (recall that Kruskal’s algorithm does not produce a unique tree). To see this, we notice that the cost-functions in Kruskal’s algorithm are only used to select a minimal cost edge. But a minimal cost-edge under the function $g$ is also a minimal-cost edge under the function $h$ and hence, by the correctness of Kruskal, if we run it with cost function $g$, we also get an MST under the function $h$.

Problem 1.1

Assume that the edge-costs are time varying and that the cost of an edge $e$ at time $t$ is given by some degree 2 polynomial $f_e(t)$. Find the time $t$ at which the cost of the MST of $G$ is minimized.

Solution: The main idea is to partition the time-line into intervals so that, for all values of $t$ within an interval, the ordering of the edges with respect to cost is preserved. Using the observation from the first part, we can then find a single candidate MST per interval and this will be an MST for all times $t$ within that interval. Lastly, we take the smallest candidate.

So we are only left with the task of partitioning the time-line. We notice that, if at some time $t$ the ordering of the edge costs changes, then $f_e(t) = f_{e'}(t)$ for some $e, e' \in E$. Since $f_e$ is quadratic, there are at most two solutions to any such equation. Hence, by taking all pairs $e, e' \in E$ and finding the solutions of $f_e(t) = f_{e'}(t)$, we get at most $m = 2\lvert E \rvert\lvert E \rvert - 1) / 2$ points $t_1, \ldots, t_m$ at which the ordering can change. By sorting the points, we can also assume that $t_1 \leq t_2, \ldots, \leq t_m$ and hence we get our $m + 1$ time intervals $[-\infty, t_1], [t_1, t_2], \ldots, [t_{m-1}, t_m], [t_m, \infty]$.

For the run-time analysis, we run $m = \mathcal{O}(\lvert E \rvert^2)$ copies of Kruskal each of which takes $\mathcal{O}(\lvert E \rvert \log \lvert E \rvert)$ for a total of $\mathcal{O}(\lvert E \rvert^3 \log \lvert E \rvert)$. The other steps are: finding the $m$ intervals, sorting the intervals, finding the smallest of the candidate MSTs. They take $\mathcal{O}(\lvert E \rvert^2), \mathcal{O}(\lvert E \rvert^2 \log(\lvert E \rvert)), \mathcal{O}(\lvert E \rvert^2)$ respectively and hence the total run time is $\mathcal{O}(\lvert E \rvert^3 \log \lvert E \rvert)$.

Problem 2

Let $G$ be an $n$-vertex connected graph with costs on the edges. Assume that all the edge costs are distinct.
1. Prove that $G$ has a unique minimum cost spanning tree.

2. Give a polynomial time algorithm to find a spanning tree whose cost is the second smallest.

3. Give a polynomial time algorithm to find a cycle in $G$ such that the maximum cost of edges in the cycle is minimum amongst all possible cycles. Assume that the graph has at least one cycle.

Solution:

1. Let $T$ be the MST that is computed by Kruskal and assume that $T' \neq T$ is some other MST. Let us look at the first iteration of Kruskal which adds an edge $e$ to $T$ such that $e \not\in T'$ (such an iteration must exist if $T \neq T'$). Then $e$ is the unique minimal edge crossing some two different trees $C_1, C_2$ such that $C_1, C_2$ are subtrees of both $T$ and $T'$ i.e. $e = (u, v)$ with $u \in C_1, v \in C_2$. Since $T'$ is also a tree, there must be some edge $e' = (u', v') \in T'$ crossing between $C_1, C_2$ i.e. $u' \in C_1, v' \in C_2$. Furthermore the cost of $e'$ is strictly greater than that of $e$. Now we claim that $\hat{T} := T' \cup \{e\} - \{e'\}$ is also a spanning tree. To see this, we note that $C_1, C_2$ are subtrees of $T'$, and hence there are paths $u' \rightarrow u$ and $v' \rightarrow v$ in $T'$. Hence for any path crossing $e' = (u', v')$ in $T'$, we can re-route it to a path crossing $e = (u, v)$ in $\hat{T}$. Lastly, it is easy to see that the cost of $\hat{T}$ is strictly smaller than that of $T'$, which contradicts the fact that $T'$ is an MST. Therefore there is no MST $T' \neq T$.

2. The simplest solution to this is to just take each edge $e_i \in E$ and find a candidate minimum spanning tree $T_i$ using Kruskal on the set of edges $E - \{e_i\}$. Also find the minimal spanning tree $T$ on the full graph. Then sort the trees $T_i$ by their cost and take the smallest one whose cost is greater than that of $T$. To see correctness, let $T'$ be a second smallest spanning tree. Then there must be some edge $e_i \in T - T'$ (otherwise $T'$ contains $T$ and hence has a cycle). When we run Kruskal on $E - \{e_i\}$ we get a tree $T_i$ whose cost is at most that of $T'$ (by correctness of Kruskal) but strictly greater than $T$ (by the uniqueness of the MST $T$ as we saw in part (1)). Therefore, since $T'$ is second-smallest, the cost of $T_i$ is the same as that of $T'$, and hence $T_i$ is second smallest.

3. Run Kruskal's algorithm until the first time it picks and edge $(u, v)$ such that $u, v$ are in the same connected component. Then the path $u \rightarrow v$ in the component together with the edge $(u, v)$ forms a cycle $C$ which we claim satisfies the desired property. To see this, assume there is some other cycle $C'$ whose maximal edge cost is strictly smaller than $C$. Then all of edges of $C'$ would have been considered by Kruskal before the edge $(u, v)$ (since Kruskal picks edges in order of increasing cost) and hence $(u, v)$ could not be the first edge which causes a cycle.

Problem 3

You are given a set of $n$ intervals on a line: $(a_1, b_1], \ldots, (a_n, b_n]$. Design a polynomial time greedy algorithm to select minimum number of intervals whose union is the same as the union of all intervals.

PS2.3-2
Solution:

First, sort the intervals in increasing order by $a_i$. Our algorithm will essentially keep track of a right-most frontier $s$ which we try to grow greedily while ensuring that we always cover everything possible to the left of $s$. Set $s$ to initially be the minimal value of $a_i$. Then repeat the following:

Find the interval with maximal $b_i$ satisfying $a_i \leq s < b_i$.

1. If such an interval $(a_i, b_i]$ exists then add it to the solution set and update $s := b_i$.
2. If such an interval does not exist then set $s$ to be the smallest value of $a_i$ which is greater than $s$.

Correctness:

First, we need to show that our solution set indeed includes the union of all intervals. To do so we analyze the invariant that, after each iteration, our current solution set covers all the points possible to the left of $s$. This holds in the beginning since $s$ is set to the minimal $a_i$ (i.e. there are no points to the left of $s$). If it holds in the beginning of some iteration then, during the iteration, $s$ gets updated either:

1. Because we add an interval $(a_i, b_i]$ satisfying $a_i \leq s < b_i$ and update $s_{\text{new}} := b_i$. But then we cover all points between $s$ and $s_{\text{new}}$ so the invariant continues to hold.
2. Because no interval with $a_i \leq s < b_i$ exists and hence we update $s_{\text{new}}$ to be the smallest value of $a_i$ which is greater than $s$. Therefore there are no points between the $s$ and $s_{\text{new}}$ in the union of all intervals and, again, the invariant then continues to hold.

Now we need to show that our solution is minimal. Let $S$ be the set of intervals in our solutions. Assume that some other set $I$ of intervals is optimal. Sort both $S$ and $I$ in increasing order by the interval start points. Let $S_j, I_j$ denote the union of the first $j$ intervals in $S, I$ respectively (according to the above sorted order). Then we will show that $I_j \subseteq S_j$ by induction. This is true for $j = 0$ since we pick the interval with maximal $b_i$ value. Assume it is true for $j$. Let $s_j$ be the value of $s$ in our algorithm after the first $j$ intervals were added. Let $t_j$ be the largest point contained in $I_j$ so that $t_j \leq s_j$ by the inductive hypothesis. Then for $j + 1$st interval in $S$ is the interval $[a, b]$ with maximal $b$ satisfying $a \leq s_j < b$ while $j + 1$st interval of $I$ has $a'_j \leq b_j \leq s_j$ and so $b'_j \leq b_i$. Therefore $I_{j+1} \subseteq S_{j+1}$. This completes the induction and shows that, if $|I| = n$ then $S_n$ covers $I$ and (since we never include an interval unless it adds points) $S = S_n$ and $|S| = |I| = n$ is optimal.

Run-Time:

It is easy to see that the run-time is polynomial. By being careful with the implementation and sorting the intervals, we can make sure that each interval is only looked at once after sorting and hence we can make the algorithm run in $O(n \log(n))$ time.

Problem 4

Given a non-increasing list of natural numbers $(d_1, \ldots, d_n)$ decide if this sequence is the degree sequence of some $n$ vertex graph.

Solution:

First let us prove the following lemma.
**Lemma 1** A non-increasing sequence \((d_1, \ldots, d_n)\) is a degree sequence of some \(n\)-vertex graph if and only if
\[
(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)
\]
is a degree sequence of some \((n-1)\)-vertex graph.

⇒ Assume there is a graph \(G'\) with vertices \(v_2, \ldots, v_n\) whose degrees follow the sequence (1). Then, by adding a new vertex \(v_1\) with edges to the vertices \(v_2, \ldots, v_{d_1+1}\), we get a graph \(G\) whose degree sequence is \((d_1, \ldots, d_n)\).

⇐ Assume that the graph \(G\) has vertices \(v_1, \ldots, v_n\) of degrees \(d_1, \ldots, d_n\). Let us first show that there is a graph \(G'\) with the same degree-sequence as \(G\) and where the \(d_1\) neighbors of \(v_1\) form the set \(S = \{v_2, \ldots, v_{d_1+1}\}\). We do this by a series of transformations. Assume that \(v_i \in S\) is not neighbor of \(v_1\) while \(v' \not\in S\) is a neighbor of \(v_1\). Then, by deleting an edges \((v_1, v'), (v_i, v'')\) and adding the edges \((v_1, v_2), (v', v'')\) we preserve the degrees of all vertices and the number of neighbors of \(v\) which are also in \(S\) increases by 1. By repeating this transformation, we end up with a graph \(G'\) with the same vertex degrees as \(G\) such that the neighbor set of \(v_1\) is exactly \(S\). Now, by removing \(v_1\) and all of its edges, we get a graph \(G''\) whose degree-sequence is given by (1).

Given the above lemma, it becomes clear that we can use the following decision algorithm:
\[
f(d_1, \ldots, d_n) := \begin{cases} 
  \text{true} & n = 1, d_1 = 0 \\
  \text{false} & n = 1, d_1 \neq 0 \\
  f(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n) & n > 1 
\end{cases}
\]
which runs in \(O(n)\) time.

Problem 5

Given a tree with (possibly negative) weights assigned to its vertices, give a polynomial time algorithm to Find a subtree with maximum weight. Note that a subtree is a connected subgraph of a tree.

**Solution:** Let \(T\) be a tree with root \(v\) and let \(v_1, \ldots, v_t\) be the children of \(v\). Let \(T_1, \ldots, T_n\) denote the \(n\) trees rooted at \(v_1, \ldots, v_n\).

The key realization is that the maximal-weight subtree of \(T\) – call it \(\text{maxWeight}(T)\) – either includes the root \(v\) or excludes it. In the latter case, it must just be the maximal-weight subtree of one of the children:

\[
\text{maxWeightSans}(T) = \begin{cases} 
  v & v \text{ is a leaf, weight}(v) \geq 0 \\
  \emptyset & v \text{ is a leaf, weight}(v) < 0 \\
  \max\{\text{maxWeight}(T_1), \ldots, \text{maxWeight}(T_n)\} & \text{otherwise}
\end{cases}
\]

In the former case, if the maximal-weight subtree of \(T\) does include \(v\), then the question is what other “subtrees” to add under \(v\). To do so, we define \(\text{maxWeightWith}(T)\) to be the maximal-weight subtree of \(T\) which includes the root \(v\). Then this is simply the union of all \(\text{maxWeightWith}(T_i)\) which have weight greater than 0 together with \(v\).
Then

\[
\text{maxWeightWith}(T) = \begin{cases} 
  v & \text{if } v \text{ is a leaf} \\
  \bigcup_{i \geq 0} \{\text{maxWeightWith}(T_i)\} \cup v & \text{otherwise}
\end{cases}
\]

Lastly, putting this together we get:

\[
\text{maxWeight}(T) = \max\{\text{maxWeightWith}(T) , \text{maxWeightSans}(T)\}
\]

We note that the number of distinct recursive calls is at most \(3|T|\) (three recursive functions which can get applied to any node) so, by cashing the answers on each recursive call (i.e. using dynamic programming) the run time of the algorithm is \(3|T|\).

**Problem 6**

Let \(G = (V, E)\) be a directed acyclic graph (i.e. it does not contain any directed cycle).

1. Prove that the graph must have a vertex \(t\) that has no outgoing edge.

2. Suppose \(|V| = n\). A topological ordering of the acyclic graph is a labeling of its vertices by integers from 1 to \(n\) such that
   - Any two distinct vertices receive distinct labels.
   - Every (directed) edge goes from a vertex with a lower label to a vertex with a higher label.

   Give a polynomial time algorithm to Find a topological ordering of the graph.

3. Fix a node \(t\) that has no outgoing edge. For every node \(v \in V\), let \(P(v)\) be the number of distinct paths from \(v\) to \(t\). Define \(P(v) = 0\) if no such path exists and define \(P(t) = 1\) for convenience. Give a polynomial time algorithm to compute \(P(v)\) for every node \(v\).

**Solution:**

1. Pick an arbitrary vertex \(v\) and follow an arbitrary path “away from” \(v\) until you reach a vertex \(t\) that has no outgoing edges. Since there are no cycles, the path will never visit any vertex twice and hence the above process must terminate after some finite number of steps proving the existence of \(t\).


3. Run a topological sort on \(G\). Let \(\text{label}(v)\) be the value assigned to \(v\) by the sort, and \(N(v)\) be the neighbor-set of \(v\). Then

\[
\text{npaths}(v, t) = \begin{cases} 
  0 & \text{label}(v) \geq \text{label}(t) \\
  1 & v = t \\
  \sum_{v' \in N(v)} \text{npaths}(v', t) & \text{otherwise}
\end{cases}
\]

\(\square\)

PS2.3-5
Problem 7

Let \( p(1), \ldots, p(n) \) be positive real numbers. A \( k \)-shot strategy \( S \) is a sequence of at most \( k \) ordered integer pairs \( (b_1, s_1), \ldots, (b_m, s_m) \), with \( 1 \leq b_1 < s_1 < b_2 < s_2 < \ldots < b_m < s_m \). Let \( \text{val}(S) = \sum_{i=1}^{m} (p(s_i) - p(b_i)) \). For any \( k \) we want to find the \( k \)-shot strategy which maximizes \( \text{val}(S) \).

Solution:

Let \( M(i,j) := \max_{i \leq b < j} (p(j) - p(b)) \) be the maximum amount of money you can make in one buy/sell transaction with sell date \( j \) and buy date \( b \) : \( i \leq b < j \). It is easy to compute \( M(i,j) \) for all \( 1 \leq i \leq j \) in time \( \mathcal{O}(n^3) \).

Now the best \( k \)-shot strategy in the days \( i, i+1, \ldots, n \) must consist of making the best possible transaction with a sell date prior to some date \( b \) and the following the best \( k-1 \)-shot strategy in the days \( b+1, \ldots, n \). Formally,

\[
\text{BEST}(k, i) = \begin{cases} 
0 & k \leq 0 \text{ or } i \geq n \\
\max_{i \leq b \leq n} (M(i,b) + \text{BEST}(k-1, b+1)) & \text{otherwise}
\end{cases}
\]

There are at most \( nk \) distinct values of \( \text{BEST}(k, i) \) that need to be computed and each runs in time \( n \) for a total run time of \( \mathcal{O}(n^3 + kn^2) \). The above algorithm needs to be modified to return the actual strategy rather than just the profit, but this just requires some simple book-keeping.

Problem 8

Given a graph \( G \) with some edge weights such that the all cycles in \( G \) have positive weight, together with vertices \( s, t \) find the number of shortest paths from \( s \) to \( t \).

Solution:

Use Bellman-Ford to find the length \( \text{best}(u, t, n) \) of the shortest path from any node \( u \) to the node \( t \) which uses fewer than \( n \) edges. Now we define \( \text{npaths}(u, t, n) \) to be the number of shortest-paths from \( u \) to \( t \) using fewer than \( n \) edges. Then \( \text{npaths}(u, t, n) = \sum_{w \in S(u)} (\text{npaths}(w, t, n-1)) \) is the sum of the number of shortest paths from \( w \) to \( t \) using fewer than \( n-1 \) edges, for all neighbors \( w \) such that some shortest path from \( u \) to \( t \) goes through \( w \). We call this set \( S(u) \). But \( w \in S(u) \iff c(u, w) + \text{best}(w, t, n-1) = \text{best}(w, t, n) \) (where \( c(u, w) \) is the cost of the edge \((u, w)\)). So it is easy to check if a vertex is in \( S(u) \). Therefore we get

\[
\text{npaths}(u, t, n) = \begin{cases} 
1 & u = t \\
0 & u \neq t, n = 0 \\
\sum_{w \in S(u)} \text{npaths}(w, t, n-1) & \text{otherwise}
\end{cases}
\]

There are \( |V|^2 \) distinct problems each of which takes at most \(|V| \) steps to compute for a run-time of \( \mathcal{O}(|V|^3) \).
Problem 9

Given \( n \) jobs such that job \( i \) takes time \( t_i \) and must finish before deadline \( d_i \) find a schedule which runs the maximum number of jobs.

Solution:

1. First we show that there is an optimal schedule in which the jobs run in order of increasing deadlines. Imagine that \( S \) is an optimal schedule and that, in \( S \), tasks do not run in order of increasing deadlines. Then there must be two tasks which run adjacent in \( S \) such that the later one has an earlier deadline. But we can always switch the order of these tasks and they finish within their deadlines (and the rest of the schedule is unchanged). By performing this re-ordering operation many times, we get a schedule where jobs run in order of increasing deadlines.

2. Sort tasks in order of increasing deadlines. In sorted order, let the deadlines be \( d_1, \ldots, d_n \) and the run-times \( t_1, \ldots, t_n \). Let \( \text{sched}(i, s) \) be the optimal (value) of the schedule for tasks \( i, i + 1, \ldots, n \) starting at time \( s \). Then the optimal schedule either runs the first task, and then runs the optimal schedule of the remaining \( n - 1 \) tasks from time \( s + t_i \), or it does not run the first task and just runs the optimal schedule of the remaining tasks from time \( s \).

\[
\text{sched}(i, s) = \begin{cases} 
0 & \text{if } s > d_n \text{ or } i > n \\
\max(1 + \text{sched}(i + 1, s + t_i), \text{sched}(i+1, s)) & \text{otherwise}
\end{cases}
\]

Figuring out the actual schedule requires simple additional book-keeping which we skip. We see that there are at most \( n \times d_n \) possible problems each of which takes \( O(1) \) time so, using dynamic programming the run-time of the above recursion is \( O(n \times d_n) \) together with sorting we then get a run time of \( O(n \log n + n d_n) \) (where \( d_n \) is the maximal deadline).

Problem 10

An independent set \( I \) in a graph is called maximal if the graph does not contain an independent set \( I' \) such that \( I \subseteq I' \), \( |I| < |I'| \). Given a tree on \( n \) vertices, and an integer \( 0 \leq k \leq n \), give a polynomial time algorithm to determine whether the tree has a maximal independent set of size \( k \).

(Hint: Design an algorithm that solves the problem for all possible values of \( k \).)

Solution:

As the main idea, each node \( v \) of the tree will store two sets:

- \( \text{maxWith}(v) \) : the set of all \( k \) such that the subtree rooted at \( v \) contains some maximal independent subset of size \( k \) which includes \( v \).
- \( \text{maxSans}(v) \) : the set of all \( k \) such that the subtree rooted at \( v \) contains some maximal independent subset of size \( k \) which excludes \( v \).

We also define \( \text{maxAny}(v) = \text{maxSans}(v) \cup \text{maxWith}(v) \).

It is clear that, if \( v \) is a leaf then \( \text{maxWith}(v) = \{1\} \) and \( \text{maxSans}(v) = \{0\} \). If \( v \) has children \( v_1, \ldots, v_m \) then

\[ \text{maxAny}(v) = \max \{ \text{maxAny}(v_i) \mid i = 1, \ldots, m \} \]
\[ \text{maxWith}(v) = \{1 + t_1 + t_2 + \ldots + t_m : t_1 \in \text{maxSans}(v_1), \ldots, t_n \in \text{maxSans}(v_n) \} \]

since an independent subset containing \( v \) must not include its children, and the maximal subset of the subtree rooted at \( v \) must contain some maximal independent subset of the trees rooted at \( v_1, \ldots, v_m \). Also

\[ \text{maxSans}(v) = \{t_1 + t_2 + \ldots + t_m : t_1 \in \text{maxAny}(v_1), \ldots, t_n \in \text{maxAny}(v_n) \} \]

Now we just need to recursively compute \( \text{maxAny}(\text{root}) \) and check if \( k \) is included in the answer. By caching the values at the nodes (i.e. using dynamic programming) we see that we actually only solve two problems per node. Also, the amount of work done at the nodes is only the merging of the sets \( \text{maxWith}(v_i) \) \( \text{maxSans}(v_i) \) computed for the children \( v_i \). Each such set is of size at most \( |V| \) and hence the merging as well as the total algorithm run in polynomial time. \( \square \)