Instructions

• For your proofs, you may use any lower bound, algorithm or data structure from the text or in class, and their correctness and analysis, but please cite the result that you use.

• Each problem is worth 10 points. Remember that for Problems 2-4, you will not get full credit unless your solution is the most efficient one in terms of running time.

• If you do not prove that your algorithm is correct, we will assume that it is incorrect. If you do not provide an analysis of the running time, we will assume you do not know what the running time is.

Problem 1

You are given a set of \( n \) variables \( x_1, x_2, \ldots, x_n \) and a set of \( m_1 \) equality constraints of the form \( x_i = x_j \) and a set of \( m_2 \) inequality constraints of the form \( x_i \neq x_j \). Is it possible to satisfy all of them? For example, it is impossible to satisfy the constraints: \( x_1 = x_2, x_2 = x_3, x_1 \neq x_3 \).

Design an algorithm that takes as input the \( m_1 + m_2 \) constraints and decides whether the constraints can be satisfied. Prove that your algorithm is correct, and provide an analysis of its running time.

Solution

Let \( G = (V, E) \) be the following graph: \( G \) has a node corresponding to every variable \( x_i \), and there is an edge \((x_i, x_j)\) if \( x_i = x_j \) is an equality constraint. We do a Depth First Search of \( G \) to find all its connected components. For each node \( x_i \), if it is in the \( k \)-th connected component of \( G \), we assign it a mark \( k \).

Now for each inequality constraint \( x_i \neq x_j \), we check to see if \( x_i \) and \( x_j \) occur in different connected components of \( G \). If there is some inequality constraint \( x_i \neq x_j \) such that \( x_i \) and \( x_j \) occur in the same connected component of \( G \), then we report that the constraints are unsatisfiable. If there is no such constraint, then we report satisfiable.

Correctness: In this problem, the equality constraints form an equivalence class. If all the constraints are satisfiable, then, there can be no inequality constraint \( x_i \neq x_j \) such that \( x_i \) and \( x_j \) lie in the same equivalence class; this is because such a constraint is not satisfiable, which leads to a contradiction. Therefore, if all the constraints are satisfiable, then the algorithm behaves correctly.

Now suppose that all the constraints are not satisfiable; we claim that this can only happen if there is an inequality constraint \( x_i \neq x_j \) where \( x_i \) and \( x_j \) lie in the same equivalence class. Suppose this is not the case, and that for all inequality constraints \( x_i \neq x_j \), \( x_i \) and \( x_j \) lie in different connected components. Then, an assignment that assigns a value \( k \) to all the variables in connected component \( k \) satisfies all the constraints, thus leading to a contradiction. Therefore, if all the constraints are not satisfiable, then, the algorithm behaves correctly as well.

Problem 2

A traveling salesman is getting ready for a big tour. Starting at his hometown he will conduct a journey in which each of his target cities is visited exactly once before he returns home. His problem is as follows: given the pairwise distances between all pairs of cities he will visit, what is the best order in which to visit them, so as to minimize the overall distance traveled?
Describe and analyze an algorithm to solve the traveling salesman’s problem in \(O(2^n \text{poly}(n))\) time. You are given an undirected \(n\)-vertex graph \(G\) with weighted edges, where each node in \(G\) is a city, and the weight of an edge \((u, v)\) is the distance between cities \(u\) and \(v\). Your algorithm should return the weight of the lightest traveling salesman tour in \(G\).[Hint: The obvious recursive algorithm takes \(O(n!)\) time.]

**Solution**

**Subproblems.** To solve this problem, we will use two subproblems \(A(z, S)\) and \(B(z, S)\), defined as follows. Let \(x\) be a fixed node in \(V\). For \(z \in V \setminus \{x\}\) and \(S \subseteq V \setminus \{x\}\), let \(A(z, S)\) be the length of the minimum-length path from \(x\) to \(z\) that includes exactly the nodes in \(S\) in between, and let \(B(z, S)\) be the length of the minimum-length path from \(z\) to \(x\) that includes all nodes in \(S\) in between.

**Recurrence Relations.** Let \(y\) be the next-to-last node on the minimum length path from \(x\) to \(z\) that includes all nodes in \(S\). Then, this minimum length path consists of the minimum length path from \(x\) to \(y\) that includes all nodes in \(S \setminus \{y\}\), appended with the edge \((y, z)\). Thus, we can update \(A(z, S)\) as follows:

\[
A(z, S) = \min_{y \in S} A(y, S \setminus \{y\}) + w(y, z)
\]

A similar relationship can be shown for \(B(z, S)\) and we can update \(B(z, S)\) as follows:

\[
B(z, S) = \min_{y \in S} w(z, y) + B(y, S \setminus \{y\})
\]

**Final Solution, Base Case and Evaluation Order.** Our final solution is to output for any fixed \(z\),

\[
\min_{S \subseteq V \setminus \{x, z\}} A(z, S) + B(z, V \setminus (S \cup \{x, z\}))
\]

Our base case is \(A(z, \emptyset) = w(x, z)\) and \(B(z, \emptyset) = w(z, x)\). Our order for solving the recurrences is by order of increasing size of \(S\); first, we solve \(A(z, \emptyset)\) and \(B(z, \emptyset)\) for all \(z\), then \(A(z, S)\) and \(B(z, S)\) for all sets \(S\) of size 1 and so on.

**Correctness.** The correctness proof is by induction over the size of \(S\). For \(S = \emptyset\), \(A(z, S)\) and \(B(z, S)\) are computed correctly for all \(z\). Now, suppose that \(A(z, S)\) and \(B(z, S)\) are computed correctly for all sets \(S\) of length \(j\), and for all \(z\), and let \(S\) be a set of length \(j + 1\). Suppose the shortest path between \(x\) and \(z\) that includes all elements of \(S\) places \(y \in S\) at the end; then, \(A(z, S) = A(y, S \setminus \{y\}) + w(y, z)\). Thus, as the algorithm takes the minimum over all \(y \in S\), it computes \(A(z, S)\) correctly. A similar fact can be shown for \(B(z, S)\) as well. The proof follows by induction.

**Running Time.** There are at most \(2^n\) sets \(S\), and at most \(n\) values of \(z\). Thus there are at most \(O(n2^n)\) possible parameters \((z, S)\) for which \(A(z, S)\) and \(B(z, S)\) must be computed. Computing each \(A(z, S)\) or \(B(z, S)\) from the previous subtasks takes \(O(|S|) = O(n)\) time. Thus, the time to construct all the \(A\) and \(B\) values is at most \(O(n2^n)\). Computing the final solution involves a single iteration over all \(S\) and thus takes at most \(O(2^n)\) time; the total running time is thus \(O(n^2 2^n)\).

**Problem 3**

A vertex cover of a graph \(G = (V, E)\) is a subset of vertices \(S \subseteq V\) that includes at least one endpoint of every edge in \(E\). For instance, in the following tree, possible vertex covers include \(\{A, B, C, D, E, F, G\}\) and \(\{A, C, D, F\}\), but not \(\{C, E, F\}\). The smallest vertex cover has size 3: \(\{B, E, G\}\).
Give a linear-time algorithm for the following task: given an input an undirected tree $T = (V, E)$, find the smallest vertex cover of $T$.

**Solution**

**Subproblems.** For each node $u$ in the tree, we define a subproblem $V(u)$ as the size of the minimum vertex cover for the subtree rooted at node $u$.

**Recurrence Relation.** The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes. Hence, for any internal node $i$, the recurrence relation is:

$$V(i) = \min \left\{ \sum_{j \in \text{children}(i)} \left( 1 + \sum_{k \in \text{children}(j)} V(k) \right), 1 + \sum_{j \in \text{children}(i)} V(j) \right\}$$

(1)

**Final Solution, Base Case, Evaluation Order.** The base case is when $u$ is a leaf; we can have $V(u) = 0$ if $u$ is a leaf, as the subtree rooted at $u$ has no edges to cover. The algorithm can then solve all the subproblems in order of decreasing depth in the tree and output as the final solution $V(r)$, where $r$ is the root of the tree. Thus $V(r)$ is the size of the minimum vertex cover.

To find the actual cover, for each node $u$, one can maintain a variable $Y(u)$ that records whether the first term or the second in the right hand side of Equation 1 led to the minimum; the vertex cover can be reconstructed by following these pointers top-down from the root.

**Correctness.** The proof of correctness is by induction in order of decreasing depth from the tree. At a leaf node $u$, the subtree is empty, and a vertex cover has size 0; the algorithm thus behaves correctly. Suppose for a node $v$, the algorithm correctly computes the size of the smallest vertex cover of all subtrees rooted at all children and grandchildren of $v$. Then, a vertex cover of the subtree rooted at $v$ either:

- Includes $v$. In this case, all edges between $v$ and its children are covered, and the solution is a union of $v$ and the minimum vertex covers of the subtrees rooted at the children of $v$. This is case (2) in Equation 1.

- Does not include $v$. In this case, the solution being a vertex cover, has to include all children of $v$, along with a vertex cover of the subtrees rooted at $v$’s grandchildren. This is case (1) in Equation 1.
Running Time. The running time is linear in $n$ because while calculating $V(i)$ and $Y(i)$ for all $i$ we look at each edge at most twice; thus, the running time is most at $2 \cdot |E| = O(n)$ in total. Calculating the actual cover using the $Y(i)$ pointers takes another $O(n)$ time.