Homework 5

ECE345 - Group 16

December 9th, 2023

Total pages: 9

Team Member	**Student Number**
Ardavan Alaei Fard	1007934620
Rowan Honeywell	1007972945
Isaac Muscat	1007897135

Contents

Question 1

Proof of $\omega(T_{\text{approx}}) \leq 2(1 - \frac{1}{l})\omega(T_{\text{opt}})$:

Firstly, let us consider a clockwise traversal of an optimal Steiner tree, T_{opt}, with the minimum number of leaves. We will partition this traversal into sections denoted by C_i for the ith section. Let each section be the set of edges between two consecutive leaves in the order of traversal such that, for l total leaves:

$$C_i = \begin{cases} E \in (r_i \rightsquigarrow r_{i+1}) & i \leq l - 1 \\ E \in (r_i \rightsquigarrow r_1) & i = l \end{cases} \tag{1}$$

Where r_i refers to the ith visited leaf. From Eqn. 1, it is clear that this traversal forms a cycle through T_{opt} since the last section, C_l, connects the final and initial visited leaves. Finally, let A denote the concatenation of all C_i into one set.

Lemma 1: The set A contains exactly two instances of each edge in T_{opt}.

Proof of Lemma 1 by induction:

Basis: Let T_2 be a Steiner tree with two nodes. Since the two nodes must be connected by a single edge, they will each be leaves. Because the clockwise traversal implies that there will be one section for each leaf, T_2 will produce two sections (C_1 and C_2). Section C_1 will travel along the edge from the first node to the second, and C_2 will be the reverse. Therefore, C_1 and C_2 each contain the same single edge with $A_{T_2} = \{C_1, C_2\}$ containing the edge exactly twice. It has thus been shown that Lemma 1 holds for the basis.

Hypothesis: For a Steiner tree with n nodes, T_n, there will be exactly two instances of each edge (in T_n) in A_{T_n}.

Inductive Step: Assuming the hypothesis is true, we must prove that this property will hold for T_{n+1}. To do this, we examine the different mechanisms in which a new node can be added to the Steiner tree T_n:

1. Adding a non-leaf node
2. Adding a leaf node connected to a non-leaf node
3. Adding a leaf node connected to leaf node

A 'leaf node' denotes a node that, when inserted, will only be connected to one other node in the tree. For **case 1**, we grow the tree by adding a new node that will not be a leaf. To preserve the properties of the tree, this must be done by bisecting an existing edge with the new node. Each instance of the edge in A_{T_n} will simply be split into two different edges, allowing $A_{T_{n+1}}$ to obey the hypothesis. Next, for **case 2**, the tree is grown by connecting a new leaf node to a non-leaf node. By increasing the total number of leaves by one, this will bisect some section C_i, splitting it into two new sections. Each of these new sections will have exactly one instance of the edge that was added to support the new leaf node. Therefore, in this case the only change to A_{T_n} is adding exactly two instances of the new edge. It has thus been shown that case 2 respects the hypothesis. Finally, in **case 3** the tree grows from the addition of a new leaf node connected to an existing leaf node. By adding a new leaf in such a way, the existing leaf will have two adjacent nodes, thereby losing its leaf property. This addition then implies that the total number of leaves in the tree, l, will remain the same. Therefore, this addition simply extends two sections, C_i and C_{i+1}, by the same edge, adding two instances of this new edge to A_{T_n}. Since all three cases preserve the hypothesis, $A_{T_{n+1}}$ will have exactly two instances of each edge in T_{n+1}.

Q.E.D (Lemma 1)

Now that it has been shown that A contains exactly two instances of each edge in T_{opt}, consider the largest section in A, denoted as $C_{\max}$. Let us define the following set:

$$P = A \backslash \{C_{\max}\}$$

In the **minimal** case, $\omega(C_{\max}) = \omega(C_i), \forall i \in [1, l]$. By definition, this occurs when all C_i have the same weight. In this case, since each section takes up an equal fraction of the weight of A, an upper bound can be derived for the weight of P:

$$C_{\max} \geq \frac{\omega(A)}{l}$$
$$\omega(P) \leq \omega(A) - \frac{\omega(A)}{l}$$
$$\omega(P) \leq (1 - \frac{1}{l})\omega(A)$$

Finally, by Lemma 1, the weight of A should be exactly double the weight of T_{opt}. This leads to an upper bound on the weight of P with respect to the weight of T_{opt}:

$$\omega(P) \leq 2(1 - \frac{1}{l})\omega(T_{\text{opt}}) \tag{2}$$

Let us now examine the Steiner tree approximation algorithm.

Lemma 2: $\omega(T_{\text{approx}}) \leq \omega(G_2)$

Proof of Lemma 2: Since G_2 is a minimum spanning tree of G_1, a complete graph, its total weight must be less than G_1. Next, the weight of G_2 after being mapped back into G remains the same since the edges of G_2 will be replaced by equally-weighted paths. With G_2 being mapped into G, the minimum spanning tree G_4 of G_3 will have a weight less than or equal to G_2. Finally, the weight of G_4 may be further reduced in T_{approx} by removing any leaves that are not vertices in R. Therefore $\omega(T_{\text{approx}}) \leq \omega(G_2)$.

Q.E.D (Lemma 2)

The total path weight between any two required nodes (in R) in P is **at minimum** the smallest path weight between them in G. Equivalently, the lightest path in G between two nodes in R has the weight of their connecting edge in G_1. Therefore, consider removing any connections in G_1 whose equivalent connection in P requires passing through a separate node in R. The result is a graph ${G_1}'$ that spans G_1 and has a weight (by Inequality 2):

$$\omega({G_1}') \leq \omega(P) \leq 2(1 - \frac{1}{l})\omega(T_{\text{opt}})$$

This is because, by removing the aforementioned connections in G_1, we now guarantee that any path from one required node to the next is as light as possible. Finally, since G_2 is a minimum spanning tree of G_1, $\omega(G_2) \leq \omega({G_1}')$. Therefore, incorporating Lemma 2 and Inequality 2, we are left with the following relationship:

$$\omega(T_{\text{approx}}) \leq \omega(G_2) \leq \omega({G_1}') \leq \omega(P) \leq 2(1 - \frac{1}{l})\omega(T_{\text{opt}})$$

Simplifying, we achieve the bound:

$$\omega(T_{\text{approx}}) \leq 2(1 - \frac{1}{l})\omega(T_{\text{opt}})$$

Q.E.D

Question 2

(a)

The form of the certificate are the vertices (V) and edges (E) representing the common subgraph G_{sub}. The verification algorithm first affirms that the number of edges in G_{sub} is at least k. This can be done in polynomial time with BFS or DFS. Next, the verifcation algorithm iterates over each edge $(u, v) \in G_{\text{sub}}$ and checks that $(u, v) \in G_1$ and $(u, v) \in G_2$. This can be accomplished by using DFS or BFS to search for the edge in G_1 and G_2. The DFS or BFS take $\mathcal{O}(V + E)$ time and at most E edges need to be iterated over. This makes the overall runtime of the verifcation algorithm to be $\mathcal{O}(E^2 + VE)$. Therefore, we can verify the certificate in polynomial time.

(b)

Let L-C-S denote the longest common subgraph problem. To show that L-C-S is NP-hard, we can show that $\textsc{Clique} \leq_P$ L-C-S. Let $\langle G, k\rangle$ be an instance of $\textsc{Clique}$, we can construct an instance of L-C-S through the following steps in polynomial time.

First, we know that by deleting edges from a clique of size k, we can construct an arbitrary graph beacuse a clique of size k maximizes the number of edges. Second, We know that the number of edges associated with a clique of size k is bounded by $\mathcal{O}(k^2)$ so deleting these edges runs in polynomial time. In G_1, we can delete edges in the clique to make an arbitrary subgraph of size k.

Creating a copy of the subgraph takes polynomial time since the graph can be traversed in $\mathcal{O}(V + E)$ time with DFS or BFS. Then a series of edges and nodes can be added to the copied subgraph which also takes polynomial time to create a secondary graph G_2. We need to show the following:

$$\langle G, k \rangle \in \textsc{Clique} \Leftrightarrow \langle G_1, G_2, k \rangle \in \text{L-C-S}$$

Proof of $\Rightarrow$: If a clique of size k exists in G, then a subgraph of size at least k is guaranteed to be in both subgraph G_1 and G_2 because the clique was modified to have an arbitrary subgraph of size k, and then it was copied for graph G_2.

Proof of $\Leftarrow$: If G_1 and G_2 both have a common subgraph of size greater than or equal to k, then connecting k nodes between the subgraphs yields a graph with a clique of size k.

Question 3

(a)

In order to prove that $B \in \text{NP}$, we need to show that it is verifiable with a given certificate. The certificate for this question should have the following language:

$$\text{certificate} = \{\langle C, W, V\rangle : C \subseteq S \text{ and } W, V \in \mathbb{R}\}$$

We can see that this certificate is polynomial in size as it has at most n elements. Therefore, we can simply sum the weights and values and compare the sums with W and V for verification. This algorithm has a time complexity of $\mathcal{O}(n)$ which means this problem is verifiable in polynomial time and a member of NP.

(b)

For showing that SubsetSum $\leq_P$ Knapsack, we need a polynomial-time function that transforms an instance of SubsetSum to an instance of Knapsack. This way we can show that Knapsack is at least as hard as SubsetSum. Given a set S and target k with a subset $C \subseteq S$ whose sum is equal to k, we can construct the Knapsack language as:

$$\textsc{Knapsack} = \{\langle S', W, V\rangle : W = V\} \tag{3}$$

Where S' is a set of tuples each containing a weight and a value. For the ith element, both the weight (w_i) and the value (v_i) will be equal to the ith element of S. In other words, S' is achieved by duplicating S and packaging each pair into tuples. Finally, let the bounds on the sums of the weights and values (W, V) be equal to k. This reduction function is $\mathcal{O}(n)$ in time as it must iterate through S exactly once.

In order to prove that the proposed reduction is valid, we must show that:

$$\langle S, k \rangle \in \textsc{SubsetSum} \iff \langle S', W, V \rangle \in \textsc{Knapsack}$$

Proof of $\Rightarrow$: Suppose there exists a $C \subseteq S$ such that $\sum C = k$. By constructing S', it is guaranteed that there will be a $C' \subseteq S'$ whose weight and value sums are exactly equal to W and V respectively since $W = V = k$.

Proof of $\Leftarrow$: Now suppose there exists a $C' \subseteq S'$ whose weight sum is less than or equal to W and whose value sum is greater than or equal to V. Recall that the weights and values of every element in S' are equal and $V = W$. Since $V = W$, the only way that the sum of this list can be at most W and at least V is if it would be exactly equal to them. By setting a new set S as $S_i = v_i$, there would be a subset $C \subseteq S$ such that $\sum C = k$ due to the condition that $W = V = k$.

Therefore, since $\textsc{Knapsack} \in$ NP and $\textsc{SubsetSum} \leq_P \textsc{Knapsack}$, we can conclude that the $\textsc{Knapsack}$ problem is NP-Complete.

Q.E.D