Discussion On The Minimum Label Spanning Tree Problem

Jonathan Cheng, Guangzhe Liu, Shengnan Liu

1. Introduction and Motivation

The problem of finding spanning trees is intrinsically linked with network design. One such extension of the spanning tree problem is the minimum label spanning tree (MLST). In this problem, we are given a graph with labeled/colored edges, with the goal to find a spanning tree that uses the fewest number of labels/colors possible. This problem often arises in contexts where achieving uniformity across connections is essential, such as by minimizing the number of protocols in a communication network or streamlining the types of resources used in a supply chain.

Hence, it is evident that the importance of minimizing label diversity naturally stems from the need for efficiency, simplicity, reliability, and cost minimization in real-world systems. In communication networks, for instance, edges with different labels might correspond to distinct technologies, such as fiber optics, satellite links, or wireless connections. Using fewer technologies can significantly reduce costs and complexity. Similarly, in supply chain management, edge labels may represent different logistics providers, and minimizing these labels can streamline operations and improve coordination.

To be more specific, imagine a network where nodes represent routers, and edges represent the communication links between them. These links may use different technologies or protocols, each with varying costs and efficiencies. Colors assigned to edges symbolize these differences—one might represent high-speed fiber optics, while another represents slower, less expensive wireless links. Since uniform costs across links are impractical, minimizing the number of colors (or link types) can streamline the network, lowering costs and improving compatibility and efficiency. This approach translates practical network design challenges into a theoretical problem, such as minimizing edge diversity in a graph, to find efficient solutions. In this case, the connection is that it abstracts practical communication setups into a more general, theoretical graph problem.

In this report, we will cover in depth the topics presented in Krumke and Wirth's paper: [2] *On the minimum label spanning tree problem*. We will detail two example heuristic methods, first given by [1] Change and Leu, to obtain approximate solutions (one sub-optimal and one near-optimal) in polynomial time. Alongside these, we will present the evaluations of these algorithms and prove their results through theoretical performance analysis. Then, we cover a NP-hardness proof for the MLST problem to show there cannot exist a polynomial time algorithm to find the exact solution unless P=NP.

2. Problem Definition

We can formally define the minimum label spanning tree problem as follows: We are given as input: $G = (V, E)$ a connected undirected graph, and $c: E \rightarrow N$ a function that takes as input an edge $e \in E$ and maps it to a corresponding label/color $n \in N$.

We define a K-colored spanning tree (V, T) as a spanning tree of G with the following property: $|\{c(e) \mid e \in T\}| \leq K$, or that the number of unique colors/labels in the spanning tree T does not exceed K. Thus, the goal of the MLST is to find a K-colored spanning tree with the minimum K out of all possible spanning trees. Note that adjacent edges in such a problem are not required to have different colors, which is fundamentally different from the edge coloring problem in graph theory of mathematics.

Below (in Fig. 1), we give an example of a possible input graph $G = (V, E)$, an edge labeling function $c(e)$, and the resultant minimum label spanning tree of G.

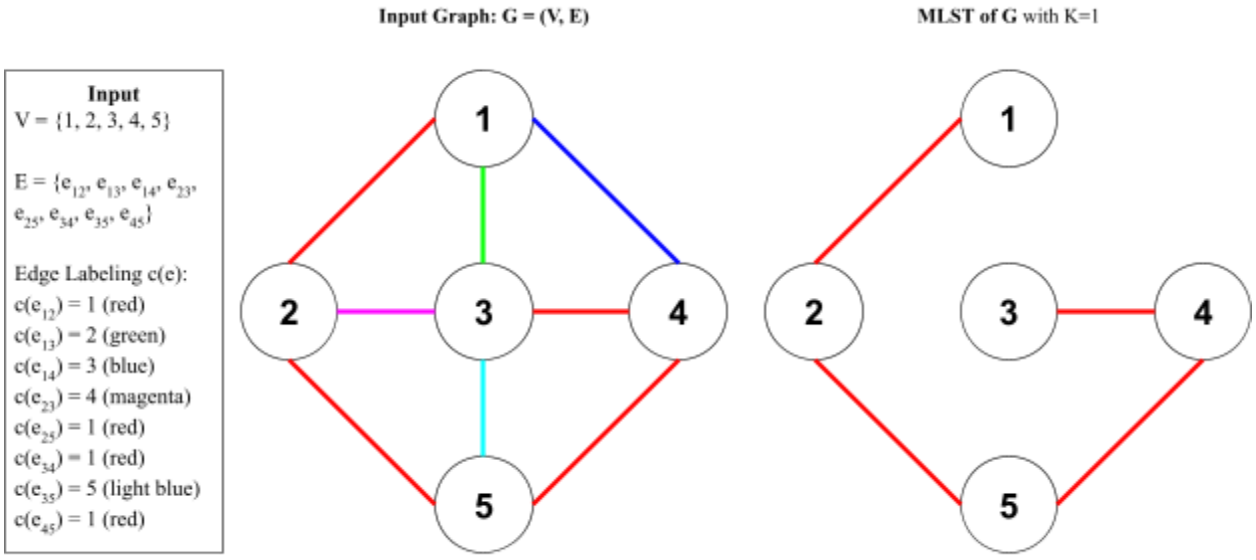


Fig. 1. Example MLST

Despite its simplicity, the MLST problem has been proven to be NP-hard, as we will discuss later. Thus, to obtain a polynomial time solution, we will examine approximation algorithms. To recap: for a minimization problem, an approximation algorithm is an algorithm that guarantees a solution whose cost is at most n * OPT, where:

- n is a constant ≥ 1 representing the performance ratio of the algorithm, and
- OPT is the cost of the optimal solution.

In the MLST problem, approximation algorithms aim to produce a spanning tree with a small number of distinct colors, within a bounded factor of the optimal K.

With this in mind, we will proceed to discuss two possible heuristics to find approximate solutions to MLST in polynomial time.

3. 1st Heuristic Solution

The first heuristic we will discuss takes the following naive approach: We begin by constructing an arbitrary spanning tree T. Next, we will iterate over each edge in G not included in T (which we will denote E') and add that edge to T, forming T'. Adding an edge to a spanning tree forms a cycle, thus we must remove an edge from the newly-formed cycle to restore the tree structure. Thus, we choose to remove the edge whose color appears the least in the current tree T'. If this edge swap results in a decrease in the total number of colors used in T', then we keep the swap and set $T = T'$. Otherwise, we undo the swap, reverting the tree back to T. Once all edges in E' have been considered (E' is set at the start), we terminate the algorithm.

At first glance, the reasoning behind this algorithm appears to be intuitive – we consider each non-tree edge and check if we can reduce the total number of colors used in our tree by swapping this edge with an edge in our tree. Thus, starting from our original tree T, the number of unique colors used only decreases or stays the same when we apply this algorithm.

However, in actuality, this heuristic is suboptimal, sometimes even outputting the worst possible spanning tree. Take the following example illustrated in fig. 2. Here, we construct a star shaped graph where each of the $n - 1$ edges has a unique color (in our example, $n = 4$). Then, we make the graph complete by adding edges of a new color (so now every vertex has an edge to every other vertex).

If we choose the star as our initial spanning tree (using $n - 1$ labels), then the algorithm will perform no swaps, as there is no single edge swap that decreases the number of unique colors used. This is because all edges not in the initial spanning tree are a "new color" (blue), so swapping any edge out from the spanning tree for one of the non-tree edges will always result in adding one new color and removing one unique color. Thus, the number of colors never decreases from swapping any edge of this initial tree, and our algorithm terminates with this initial tree with $n - 1$ unique colors.

However, if we instead use only edges of the new color and one additional edge from the star, we are able to construct a spanning tree that uses only 2 unique colors. The issue becomes clear that the heuristic never makes a swap that keeps the number of unique colors the same, even if making that swap would benefit the algorithm down the line. As a result, this initial heuristic is off by a factor of $\Omega(n)$, as it outputs a tree using $n - 1$ unique colors, while the true optimal solution uses 2 unique colors.

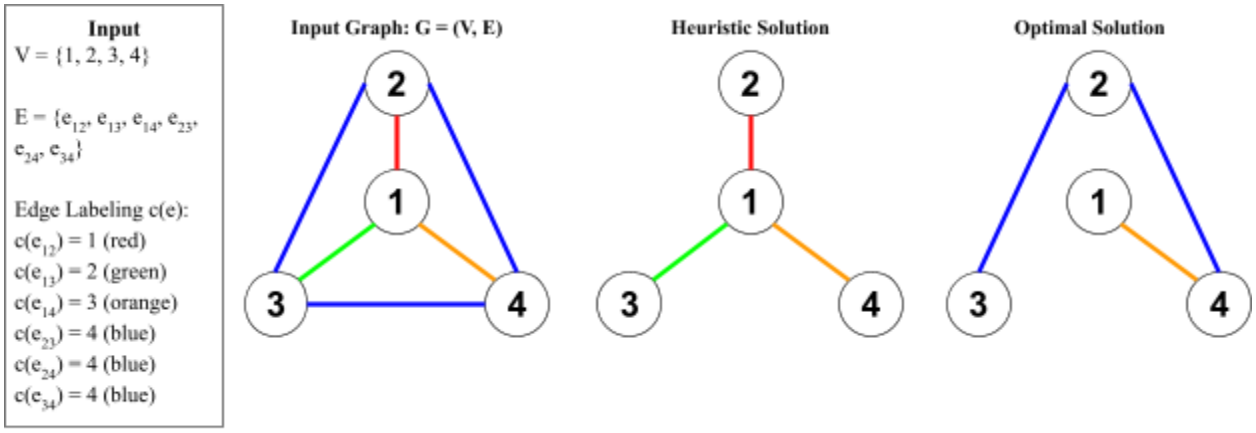


Fig. 2. Suboptimal Solution

Thus, to obtain a more optimal solution, we must pursue a different heuristic algorithm.

4. 2nd Heuristic Solution

In this section, we discuss a second approximation algorithm to find an approximate solution to the MLST problem. Additionally, we will prove that the algorithm finds this solution in polynomial time, and that it has a logarithmic performance guarantee.

First, we will cover some preliminary terms and techniques necessary to understand the algorithm.

I. Preliminaries:

1. Union-Find data structure (Also known as Disjoint-Set Union (DSU)):
   Definition: A structure that supports two primary operations:
   - Find operation: Determines which subset (or "set representative") a particular element belongs to, which helps to check if two elements are in the same subset.
   - Union Operation: Merges two subsets into a single subset, which helps to join two disjoint sets when adding connections, such as edges in a graph.

   Example:
   Imagine you have 5 elements: {1, 2, 3, 4, 5} and an initial state: {1}, {2}, {3}, {4}, {5}.
   - Union(1, 2) = Merge sets containing 1 and 2. New state: {1, 2}, {3}, {4}, {5}
   - Union(3, 4) = Merge sets containing 3 and 4. New state: {1, 2}, {3, 4}, {5}
   - Find(1) = Determine the subset of 1. Result: {1, 2}
   - Union(2, 3) = Merge sets containing 2 and 3. New state: {1, 2, 3, 4}, {5}
   - Find(4) = Determine the subset of 4. Result: {1, 2, 3, 4}

   Analysis:
   - This structure is essential for MLST problems because it efficiently manages and tracks the connectivity of graph components. In the MLST problem, we aim to iteratively construct a spanning tree by adding edges of specific colors while

ensuring that the graph remains connected. The Union-Find data structure allows us to quickly check if adding an edge merges different connected components or creates a cycle, which is critical in graph-based algorithms.

2. The inverse of Ackerman's function:
   Definition: $\alpha(m, n) = \min\{i \geq 1: A(i, \lfloor m / n \rfloor) > \log_2 n\}$, where A(i, j) is Ackermann's function.
   Note: It is a function of two parameters whose value grows very slowly.

3. Ackerman's function:
   Definition:
   - $A(0, j) = j + 1$ for $j \geq 0$
     $A(i, 0) = A(i - 1, 1)$ for $i > 0$
     $A(i, j) = A(i - 1, A(i, j - 1))$ for $i, j > 0$

   Note: It is a function of two parameters whose value grows very fast.

4. Using techniques like path compression and union by rank for t "Union-Find" operations on s elements results in $O(t * \alpha(t, s))$ time performance, ensuring the algorithm to be computationally efficient. α is the inverse of Ackermann's function.

We will now provide an outline of the approximation algorithm. Given the NP-hardness of MLST, this heuristic provides an approximate solution with a proven logarithmic performance guarantee. The intuition behind this approach is as follows: the heuristic employs a greedy strategy to iteratively add edges of the same color to the graph, aiming to connect all nodes into a single connected component. At each step, it selects the color that maximally reduces the number of disconnected components in the graph. This approach ensures that the graph becomes connected as quickly as possible while using fewer colors. Following this passage is a detailed explanation of each component/step of this approximation algorithm.

II. Algorithm:

1. Initialization:
   - Create an empty graph $H = (V, \emptyset)$, containing all nodes but no edges.
   - Let $R = \{1, 2, \ldots, l\}$ be the set of colors that will be inspected later.
   - Let $C = \emptyset$ be the set of selected colors.
   - Set $comp(H)$, the number of connected components in H, initially to $|V|$.
   - Preprocessing:
     - For each color $i \in R$, let Gi be the subgraph of G containing only edges of color i: $Gi = (V, \{e \in E \mid c(e) = i\})$.

- For each Gi, reduce $\{e \in E \mid c(e) = i\}$ to a spanning forest with at most $|V| - 1$ edges. This removes redundant edges while preserving connectivity.
- Denote Ei to be the reduced edge set $\{e \in E \mid c(e) = i\}$.
- Now, $Gi = (V, Ei)$

2. Iterative Color Selection:
   - For each unused color $i \in R \setminus C$, simulate adding edges of Gi to H:
     - For $e = (u, v) \in Gi$:
       - Check whether u and v belong to the same connected component in H using the **Find** operation in the union-find data structure.
       - If u and v are in different components, mark e as reducing the number of components and add it to H.
       - Track the number of connected components that would result after adding qualified edges of Gi to H.
       - Note: we do not lose generality by not adding $e = (u, v) \in Gi$ where u and v are in the same components. This is because such an e does not contribute to reducing the number of connected components of H.
   - Select the color i that minimizes the number of connected components after simulating, add i to C, and select Gi.
     - For all $e = (u, v) \in Gi$ that are marked during simulation, add e to H, and then perform a **Union** operation to merge the connected components of u and v. Otherwise, skip the corresponding e to avoid forming a cycle.
     - Update $comp(H)$ to reflect the current number of connected components.
3. Termination: Repeat the iterative color selection steps until $comp(H) = 1$.

Next, we will proceed to prove our earlier claim that this algorithm finds a solution in polynomial time.

III. Time Complexity Analysis:

1. Let $|V| = n$, $|E| = m$ and the number of colors l.
2. Initialization:
   - Creating $H = (V, \emptyset)$ for one color: $O(n)$ because adding or listing each vertex is an $O(1)$ operation and it is done n times.
   - Thus, total cost across l colors: $O(ln)$.
   - Preprocessing: For each color i, reducing Gi to a spanning forest with $O(n - 1)$ edges:
     - For each Gi, connected components are determined using a graph traversal (Depth-First Search or Breadth-First Search).

- Graph Traversal Complexity: $O(mi)$, mi: the number of edges in Gi.
- Total Complexity for l colors: The sum of edges across all l colors is $\leq m$. Thus, finding connected components for all colors takes at most $O(m)$.

- Combining initialization and preprocessing gives $O(m + ln)$.

3. Iterative Color Selection:
   - Simulating Edge Additions:
     - Cost per Edge: Each **Find** operation takes $O(\alpha(m, n))$.
     - Total Cost per Color: For $O(|Ei|)$ edges, this step requires $O(|Ei| * \alpha(m, n))$.
     - Total Cost Across All Colors: Since the sum of edges across all l colors is $\leq m$, the cost is $O(m * \alpha(m, n))$.
   - Updating H:
     - Cost per Edge: Each **Union** operation takes $O(\alpha(m, n))$.
     - Cost for Each Iteration: Since at most $n - 1$ edges can be added in one iteration, the cost is $O((n - 1) * \alpha(m, n))$
     - Total Cost for All Iterations: Since there are l colors, the total cost is $O((n - 1) * l * \alpha(m, n))$.

   - Final Iterative Complexity: $O(l * min(ln, m) * \alpha(m, n))$
     - Because it involves testing min(ln, m) edges :
       - ln if each color has edges for all n vertices
       - m if the graph is sparse.
4. Overall Complexity: $O(m + ln) + O(l * min(ln, m) * \alpha(m, n))$, which simplifies to $O(m + l * min(ln, m) * \alpha(m, n))$.

Lastly, we will prove that this algorithm has a logarithmic performance guarantee. To help us prove this claim, we will establish a few lemmas.

IV. Performance Guarantee:

1. Lemma 2: Let H be a connected graph with r nodes that contains a p-colored spanning tree. Then there exists a color i such that H, restricted to edges of color i, has no more than $r * (1 - 1/p) + 1/p$ connected components.
   Proof:
   - Consider a p-colored spanning tree T of H. The tree T consists of $r - 1$ edges.
   - Since there are p colors available in T, by the pigeonhole principle, at least one color i must appear on at least $\lfloor (r - 1)/p \rfloor$ edges of T.
   - To analyze the effect of edges of color i, we start with an empty graph containing all nodes of H, and iteratively add the $\lfloor (r - 1)/p \rfloor$ edges of color i from T. Each

time a new edge is added, it connects two previously disconnected components of the graph, reducing the total number of connected components by exactly one.

- Initially, there are r connected components (one for each node).
- After adding $\lfloor (r-1)/p \rfloor$ edges of color i, the number of components is reduced by $\lfloor (r-1)/p \rfloor$.
- Thus, the final number of components is:

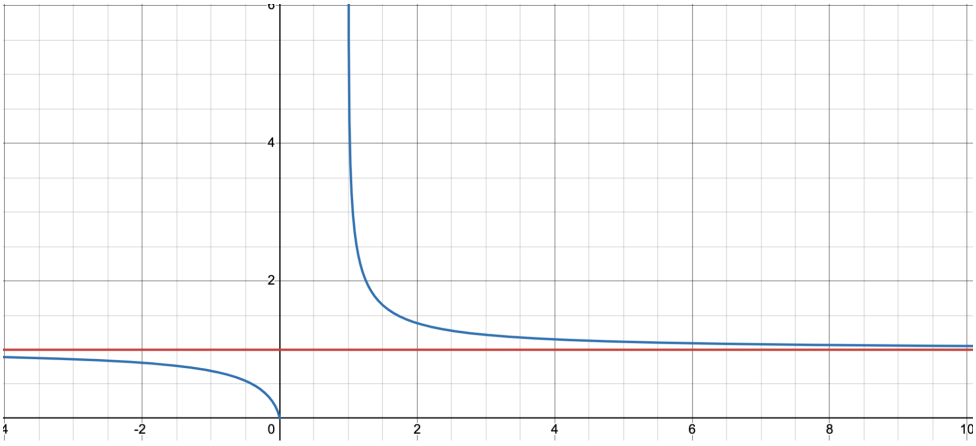
$$r - \lfloor (r-1)/p \rfloor \leq r * (1 - 1/p) + 1/p$$

2. Lemma 3: Let $f(n) = n * (1 - 1/p) + 1/p$ for some fixed $p \geq 1$. Then, for all $k \geq 0$, applying f iteratively k times results in $f^k(n) \leq n * (1 - 1/p)^k + k/p$, where $f^k(n)$ is the k-fold iteration of f, representing an upper bound on the number of connected components after k iterations.
   Proof: We prove the claim of Lemma 3 by mathematical induction:
   - Base Case: For $k = 0$, we have: $f^0(n) = n \leq n * (1 - 1/p)^0 + 0/p = n$.
   - Inductive Step: Assume the inequality holds for k, i.e., $f^k(n) \leq n * (1 - 1/p)^k + k/p$. Prove that it is true for $k + 1$:
     - From the definition of f, we know: $f^{k+1}(n) = f(f^k(n)) = f^k(n) * (1 - 1/p) + 1/p$.
     - By the inductive hypothesis, substitute $f^k(n)$: $f^{k+1}(n) \leq (n * (1 - 1/p)^k + k/p) * (1 - 1/p) + 1/p$.
     - Distribute $(1 - 1/p)$: $f^{k+1}(n) \leq n * (1 - 1/p)^{k+1} + k/p * (1 - 1/p) + 1/p$.
     - Since $k/p * (1 - 1/p) + 1/p = (k+1)/p - k/p^2$, $f^{k+1}(n) \leq n * (1 - 1/p)^{k+1} + (k+1)/p - k/p^2$.
     - Then clearly, $f^{k+1}(n) \leq n * (1 - 1/p)^{k+1} + (k+1)/p$.
     - Thus, the inequality holds for k + 1.
   - Thus, by mathematical induction, we proved Lemma 3.

3. Lemma 4: For $n, p \in N$, we have: $n(1 - 1/p)^k \leq p$ if $k \geq p \ln n/p$
   Proof:
   - It holds that the expression $n(1 - 1/p)^k$ is decreases exponentially as k increases. Hence, if the inequality holds for $k = p \ln n/p$, it will hold for any value of k greater than this. As a result, to prove our lemma, it is sufficient to show that it is sufficient to show that $n(1 - 1/p)^k \leq p$ for $k = p \ln n/p$.
     To do this, we plug in the value of k, and simplify the inequality:
     - $n(1 - 1/p)^{p \ln n/p} \leq p$
     - $\ln(n(1 - 1/p)^{p \ln n/p}) \leq \ln(p)$

- $ln(n) + ln(1 - 1/p)^{p\ ln\ n/p}) \leq ln(p)$
- $ln(n) - ln(p) \leq -ln(1 - 1/p)^{p\ ln\ n/p}$
- $ln(n/p) \leq -p * ln(n/p) ln(1 - 1/p)$
- $1 \leq pln(p/(p-1))$

- The right hand side of the inequality: $p\ ln(\frac{p}{p-1})$ is a non-increasing function with a limit of 1 as p approaches infinity. If we graph this function, it like such:



- Thus, it holds that for all $p \geq 1$, $p\ ln\ p/(p-1) \geq 1$, and the lemma follows.

4. Lemma 5: The number of iterations is at most $p * ln(n/p) + p + ln(n/p)$.
   Proof:
   - After $\lceil p * ln(n/p) \rceil$ iterations of our algorithm, we have reduced the number of connected components from n to at most $\lceil p + ln(n/p) \rceil$. This statement follows from lemmas 3 and 4.
   - Recall: Lemma 3 states that $f^k(n)$, the number of connected components in our graph after iteration k, is upper bounded by: $n(1 - \frac{1}{p})^k + \frac{k}{p}$.
   - Thus, if we plug in $k = \lceil p * ln(n/p) \rceil$, then we get
     $$f^k(n) \leq n(1 - \frac{1}{p})^{pln(n/p)} + \frac{pln(n/p)}{p}.$$
   - By lemma 4, we know that the term
     $$n(1 - \frac{1}{p})^{pln(n/p)} \leq p.$$
   - Thus, if substitute this term and simplify the inequality, we get that $f^k(n) \leq p + ln(n/p)$, proving the claim that after $\lceil p * ln(n/p) \rceil$ iterations of the algorithm, the number of connected components is at most $\lceil p + ln(n/p) \rceil$.
   - Because each iteration of the algorithm decreases the number of connected components by at least 1, this means that to completely connect the graph (i.e. get the number of connected components to one), we need at most $\lceil p + ln(n/p) \rceil$ - 1 additional iterations.

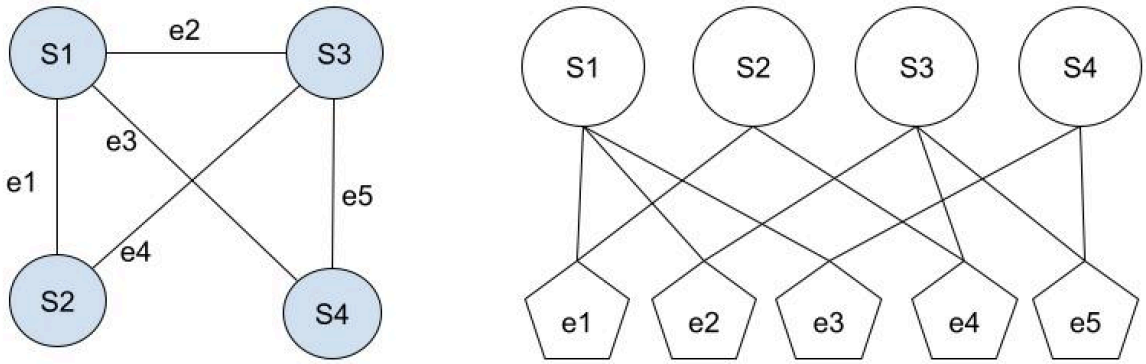
- Thus, if we sum up the upper bounds on the number of iterations we need to complete the algorithm, we get:
  $$\lceil p * ln(n/p) \rceil + \lceil p + ln(n/p) \rceil - 1 \text{ iterations}$$
- If we simplify the ceiling, we get:
  $$p * ln(n/p) + p + ln(n/p) \text{ iterations}$$
- Thus proving our lemma.

Using these lemmas, we can prove the logarithmic performance guarantee of the algorithm:

5. Theorem 6:
   - The algorithm for the Minimum Label Spanning Tree (MLST) problem has a performance guarantee of $2\ln n+1$. Specifically, for an n - node graph G with a p-colored spanning tree, the algorithm produces a spanning tree with at most:
     $$p * ln(n/p) + p + ln(n/p) \leq p * (2\, ln(n) + 1) \text{ colors.}$$
   - We know that our algorithm adds one color/label to the resultant tree each iteration. Thus, the number of colors/labels in our final solution has the same upper bound as our iterations. Therefore, the number of unique colors our algorithm finds is at most: $p * ln(n/p) + p + ln(n/p)$ colors.
   - Because we know p is positive and $p \geq 1$, it holds that:
     $$p * ln(n/p) \geq ln(n/p).$$
   - Thus, the following inequality holds:
     $$p * ln(n/p) + p + ln(n/p) \leq p * ln(n/p) + p + p * ln(n/p).$$
   - This right hand equality simplifies to
     $$p(2ln(n) + 1).$$
   - Thus giving us our performance ratio of $2ln(n) + 1$, proving our theorem, and providing the performance guarantee.

5. NP-Hardness Discussion

The NP-hardness of this problem is given by a polynomial reduction from MINSETCOVER problem. MINSETCOVER is defined as given a Universe U and a family S of subsets of U, find the subfamily $C \subseteq S$ whose union is U such that $|C|$ has the smallest possible size. MINSETCOVER's NP-completeness is given by a simple reduction to the Vertex Cover problem. For a Vertex Cover problem on a Graph G, construct a bipartite of every vertex S_i in G and edges e_j in G, connect edge (S_i, e_j) in bipartite when S_i is a vertex of e_j, it can be redacted to a MINSETCOVER problem where each set is a set of two elements. Assuming $P \neq NP$, there are no polynomial algorithms for MINSETCOVER. For this example, the VERTEX COVER in the original problem is redacted to a MINSETCOVER problem described by the bipartite.

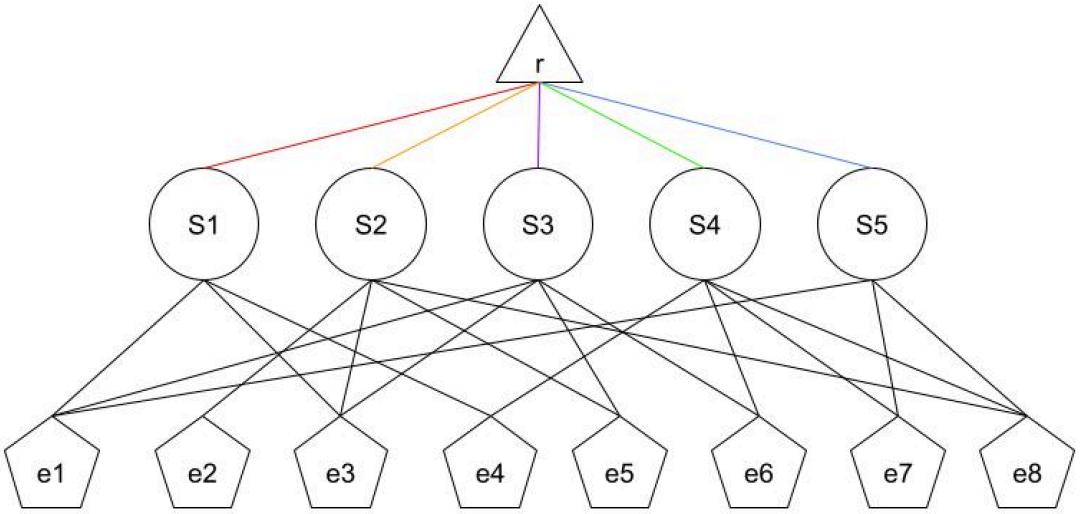


Theorem: MINSETCOVER is NP-hard.

MINSETCOVER can also be formulated to an Integer Linear Programming problem and thus can be shown to be an FPT when parameterized by the size of the union and set family. In other words, MINSETCOVER cannot be polynomial approximated with a factor of $\eta ln|Q|$.

A polynomial reduction from MINSETCOVER to MLST can be given this way: For a Universe U and a family S of subsets of U, construct a vertex in Graph G for each element $S_i \in S$, $e_j \in U$, and connect edge (S_i, e_j) of color 0 when $e_j \in S_i$, and then create a root node r and connect edge (r, S_j) of color j($1 \leq j \leq |S|$). In this structure G, an MLST provides an MINSETCOVER to the original problem: every (r, S_j) selected in the MLST means selecting S_j for the original MINSETCOVER problem. Figure shows an example reduction of MINSETCOVER $U = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $S_1 = \{e_1, e_3, e_4\}$, $S_2 = \{e_2, e_5, e_8\}$, $S_3 = \{e_1, e_3, e_5, e_6\}$, $S_4 = \{e_4, e_6, e_7, e_8\}$, $S_5 = \{e_1, e_7, e_8\}$. In this constructed colored graph, the MLST of this graph contains the least possible types of

color but connects all the leaf nodes in the Universe to the root, therefore it will select the least possible subsets while covering all the elements in the Universe.



This reduction shows that assuming there is a polynomial algorithm for MLST, there is a polynomial algorithm for MINSETCOVER and then P=NP. Assuming P!=NP, MLST is an NP-hard problem.

Theorem: MLST is NP-hard.

This proof of NP-hardness also provides some insights to a parameterized solution for the MLST: Since MINSETCOVER is not an FPT when parameterized by the size of the size of Universe and family, Assuming FPT is not equal to w[1], MLST is also w[1]-hard when parameterized by the size of the graph. Solutions to MLST can be given in multiple ways:

Randomized algorithm is the most straightforward method for coloring problems. Given an integer k, our task is to determine whether there exists a Spanning Tree in the original graph G of less than k colors. We can randomly select a subset of colors C_0 s.t. $|C| = k$ and determine whether there exists a spanning tree in the graph G of only colors in C_0 in $O(|V| + |E|)$. For one solution of k colors, the hitting possibility is at least $2^{-k-|C|}$, so in general the total complexity for solving this problem is $O(2^{k+|C|}(|V| + |E|))$. The hitting possibility can be proved this way: consider $a_1, \ldots, a_n$ is a sequence of n random 0/1 values, chosen uniformly at random, $b_1, \ldots, b_k$ are k different integers within 1 to n, discussion the possibility of $P(a_{b_1} = \ldots = a_{b_k} = 1) = 2^{-k-|C|}$.

Integer linear programming provides another way to normalize this problem: use x_e to represent whether an edge is selected in the spanning tree, use y_c to represent whether a color is selected in the spanning tree, the following constraint can be used to ensure that the graph is connect: Pick an arbitrary root r as the root of the spanning tree of the graph, for each edge $e_{uv} \in E$, let f_{uv} denote the amount of flow sent from u to v alongside edge e_{uv}, ensuring the root flows out $|V| - 1$ flow and each vertex in the graph receives one can ensure the graph is connected and there is a spanning tree in the graph. All the constraints can be given this way:

$$x_e \in \{0, 1\}, e \in E$$

$$y_c \in \{0, 1\}, c \in C$$

$x_e \leq y_c$ when edge e has color c

$$f_{uv} \leq x_{uv}(|V| - 1), uv \in E$$

$$\sum_{rv} f_{rv} = |V| - 1, rv \in E$$

$$\sum_{uv} f_{uv} - \sum_{vw} f_{vw} = 1, uv, vw \in E$$

$$MINIMIZE \sum_{c} y_c, c \in C$$

This integer linear programing normalization provides an FPT for MLST when parameterized by the number of edges |E|, but it is not FPT when parameterized by the number of used colors.The integration of flow in this process also reveals the connection between linear algebra and general graph connectivity problems.

Parameterized searching is one common way to solve the MLST problem. Given an integer k, our task is to determine whether there exists a Spanning Tree in the original graph G of less than k colors. The following searching algorithm can be used to find the smallest possible k:

MLST(G, k):
if G is disconnected -> NO
if remaining types of color <= k and G is connected -> YES
Pick a color c -> MLST(G merge all connected block of e of c, k-1) or MLST(G-all e of c, k)

The solution will be $O(2^{k+|C|}(|V| + |C|))$ and it is w[1]-hard when parameterized by the number of used colors.The following lemmas can provide several reduction and branching rules to optimize the algorithm:

1. Lemma 1: if deleting all the edges of a specific color c disconnects the graph, then color c must be in the MLST of the graph.

2. Lemma 2: There exists a color i such that H, restricted to edges of color i, has no more than $r * (1 - 1/p) + 1/p$ connected components.

MLST(G, k):
if G is disconnected -> NO
if remaining types of color <= k and G is connected -> YES
If there's a color c deleting all edges of color c the graph is disconnected -> MLST(G-all e of c, k-1)
Pick a color i such that i has no more than $r * (1 - 1/k) + 1/k$ connected components-> MLST(G merge all connected block of e of c, k-1) or MLST(G-all e of i, k)

After applying all these lemmas, we got a searching algorithm with complexity less than $O(2^{k+ln|C|}(|V| + |C|))$.

6. Conclusion

The Minimum Label Spanning Tree (MLST) problem addresses a fundamental challenge in optimizing network design by minimizing label diversity in spanning trees. Through the exploration of heuristic approaches, we demonstrated the strengths and limitations of both suboptimal and near-optimal solutions. While the first heuristic illustrates the pitfalls of local optimization, the second heuristic leverages a greedy, component-reduction strategy with proven logarithmic performance guarantees. By formalizing the problem, analyzing its complexity, and providing approximative solutions, this report underscores the theoretical and practical significance of MLST. Future research can focus on refining heuristics and exploring hybrid methodologies to bridge the gap between computational efficiency and solution quality.

Works Cited

[1] Chang, R.-S., Leu, S.-J.. “The minimum labeling spanning trees.” *Information Processing Letters, 63 (1997) 277-282.*

[2] Krumke, Sven O., Wirth, Hans-Christoph. “On the minimum label spanning tree problem.” *Information Processing Letters, 66 (1998) 81-85.*

[3] Michael R. Fellows, Jiong Guo, Iyad Kanj. “The parameterized complexity of some minimum label problems”,

Journal of Computer and System Sciences, Volume 76, Issue 8, 2010,