Steiner Tree Approximation via Iterative Randomized Rounding

1 Problem Description

The Steiner Tree Problem is a fundamental combinatorial optimization problem with significant applications in various fields such as network design, electronic circuit layout, communication networks, and transportation systems. The objective is to find the minimum-weight tree that spans a given subset of vertices (called terminals) in a weighted graph. This problem is NP-hard, meaning that finding an exact solution is computationally infeasible for large instances.

In this project, we address the Steiner Tree Problem using an iterative randomized rounding algorithm. The main tools and techniques used in this approach include:

- **Linear Programming Relaxation**: The problem is first relaxed into a linear programming (LP) problem. This relaxation allows us to solve a simpler version of the problem, where the solution can take fractional values.
- **Randomized Rounding**: The fractional solution obtained from the LP relaxation is then converted into an integer solution through a process of randomized rounding. This technique involves making probabilistic decisions to round the fractional values to integers while maintaining feasibility.
- **Iterative Refinement**: The algorithm iteratively refines the solution to improve its quality. In each iteration, the solution is adjusted to better meet the required constraints and reduce the overall weight of the tree.

The iterative randomized rounding algorithm provides an efficient approximation to the Steiner Tree Problem, balancing the trade-off between solution quality and computational efficiency.

2 Solution Description

The solution to the Steiner Tree Problem is implemented using the following steps:

1. **Linear Programming Relaxation**: Formulate the Steiner Tree Problem as a linear programming problem. Solve the LP relaxation to obtain a fractional solution.
2. **Randomized Rounding**: Apply randomized rounding to convert the fractional solution into an integer solution. This involves making probabilistic decisions based on the fractional values.
3. **Iterative Refinement**: Iteratively refine the solution by adjusting the tree structure and re-evaluating the weights. This step ensures that the solution meets the required constraints and improves the overall quality.

The algorithm is designed to handle large graphs efficiently, making it suitable for practical applications in network design and optimization.

3 Function Descriptions

The code is structured to solve the Steiner Tree Problem using various helper functions that manage different aspects of the algorithm. Below are detailed descriptions of the main functions:

getChildren This function retrieves the children of a given node u in the tree represented by the adjacency matrix A. It identifies all nodes directly connected to u and filters out the node itself and any nodes with a lower index to ensure proper tree traversal.

getLeaf This function identifies and returns the leaf nodes in the tree represented by the adjacency matrix A. A leaf node is defined as a node with exactly one connection (degree of one).

findPath This function finds the shortest path from node u to node v using Breadth-First Search (BFS). It maintains a queue to explore nodes level by level and tracks the parent of each node to reconstruct the path once the target node is reached.

isOnPath This function checks if node w is on the path from node u to node v. It uses the `findPath` function to get the path and then checks if w is included in this path.

getWeight This function computes the weight vector $w(v)$ for node v in the witness tree. It counts the number of paths passing through v among all pairs of final Steiner nodes and adds an additional weight if v is a final node.

computeCost This function computes the cost $C^1_{u_j}$ for a given child and its subtree W_{u_j}. It sums the inverse of the weight vector for all nodes on the path from u to each final Steiner node.

costVector This function computes the cost vector for the children of a given node. It uses the `computeCost` function to calculate the cost for each child and stores these values in a vector.

selectMarkedChild This function selects the marked child based on cost conditions. It chooses the child with the minimum cost if none of the children are final nodes. If all costs are above a certain threshold, it selects the child with the minimum cost; otherwise, it selects the child that satisfies the cost condition.

getLeafDescendant This function gets the leaf descendant of a given node in the tree. It recursively follows the path from the node to its children until a leaf node is reached.

mergeAdjacencyMatrices This function merges the adjacency matrices of subtrees to construct the witness tree. It combines the connections from all subtrees and ensures that the leaf descendants of the marked child are connected to the leaf descendants of other children.

computeWitnessTree This function computes the witness tree for a given node u by recursively constructing the witness trees for its children and merging them. It uses the previously described functions to manage the tree structure and calculate costs.

4 Example Usage

To run the code, you can use the following example in Julia:

Listing 1: Example usage of the Steiner Tree approximation algorithm

```
using SteinerTreeApproximation

# Define the graph and terminals
graph = [
    (1, 2, 1.0),
    (2, 3, 2.0),
    (3, 4, 1.5),
    (4, 1, 2.5),
    (1, 3, 2.2)
]
terminals = [1, 3, 4]

# Run the algorithm
steiner_tree = steiner_tree_approximation(graph, terminals)
```

```
# Output the result
println("Approximate Steiner Tree: ", steiner_tree)
```

5 Final Observations

It is important to note that in the current version of the code, it is only possible to consider trees in which the nodes are enumerated such that a parent always has lower index then a child. This issue could be overcame if we recoded the project using dictionaries instead of adjacency matrices. The problem in case we didn't have this type of enumeration would arise in our code due to the function "get children", where we force any child to have higher degree then a parent.

It is of great interest for my research to use the code that I implement on some instances to verify the integrality gap that was found from the authors of the paper that I'm basing my work on. The fact that there is a little integrality gap suggests that either the analysis of the authors is not tight, or there is space of improvemet in the code itself.

Finally it in our plans to compare the same examples that we will run in the code that I'm presenting here using an alternative dynamic approach, that is known to be tight for the edge witness tree, but was not yet implemented for the node witness tree problem.