2-Approximation for Prize-Collecting Steiner Forest

Ali Ahamdi*
ahmadia@umd.edu

Iman Gholami*
igholami@umd.edu

MohammadTaghi Hajiaghayi*
hajiagha@umd.edu

Peyman Jabbarzade*
peymanj@umd.edu

Mohammad Mahdavi*
mahdavi@umd.edu

Abstract

Approximation algorithms for the prize-collecting Steiner forest problem (PCSF) have been a subject of research for over three decades, starting with the seminal works of Agrawal, Klein, and Ravi [1, 2] and Goemans and Williamson [13, 14] on Steiner forest and prize-collecting problems. In this paper, we propose and analyze a natural deterministic algorithm for PCSF that achieves a 2-approximate solution in polynomial time. This represents a significant improvement compared to the previously best known algorithm with a 2.54-approximation factor developed by Hajiaghayi and Jain [18] in 2006. Furthermore, Könemann, Olver, Pashkovich, Ravi, Swamy, and Vygen [23] have established an integrality gap of at least $9/4$ for the natural LP relaxation for PCSF. However, we surpass this gap through the utilization of a combinatorial algorithm and a novel analysis technique. Since 2 is the best known approximation guarantee for Steiner forest problem [2] (see also [14]), which is a special case of PCSF, our result matches this factor and closes the gap between the Steiner forest problem and its generalized version, PCSF.

1 Introduction

The Steiner forest problem, also known as the generalized Steiner tree problem, is a fundamental NP-hard problem in computer science and a more general version of the Steiner tree problem. In this problem, given an undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$ and a set of pairs of vertices $\mathcal{D} = \{(v_1, u_1), (v_2, u_2), \cdots (v_k, u_k)\}$ called demands, the objective is to find a subset of edges with the minimum total cost that connects v_i to u_i for every $i \leq k$. In this paper, our focus is on the prize-collecting Steiner forest problem (PCSF), which is a generalized version of the Steiner forest problem.

Balas [4] first introduced general prize-collecting problems in 1989 and Bienstock, Goemans, Simchi-Levi, and Williamson [8] developed the first approximation algorithms for these problems. In the prize-collecting version of the Steiner forest problem, we are given an undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$ and a set of pairs of vertices $\mathcal{D} = \{(v_1, u_1), (v_2, u_2), \cdots (v_k, u_k)\}$ called demands, along with non-negative penalties π_{ij} for each demand (i, j). The objective is to find a subset of edges and pay their costs, while also paying penalties for the demands that are not connected in the resulting forest. Specifically, we aim to find a subset of demands Q and a forest F such that if a demand (i, j) is not in Q, its endpoints i and j are connected in F, while minimizing the total penalty of the demands in Q and the sum of the costs of the edges in F. Without loss of generality, we assign a penalty of 0 to pairs that do not represent a demand, ensuring that there is a penalty associated with each pair of vertices. This allows us to define the penalty

*University of Maryland.

function $\pi : V \times V \to \mathbb{R}_{\geq 0}$, where $V \times V$ represents the set of all unordered pairs of vertices with $i \neq j$. In this paper, we significantly improve the approximation factor of the best-known algorithm for PCSF.

For the Steiner forest problem, the first approximation algorithm was introduced by Agrawal, Klein, and Ravi [2]. Their algorithm addressed a more generalized version of the Steiner forest problem and achieved a 2-approximation for Steiner forest. Later, Goemans and Williamson [14] provided a simplified simulation of their algorithm, which yields a $(2 - \frac{2}{n})$-approximate solution for the Steiner forest problem, where n is the number of vertices[1]. However, no further advancements have been made in improving the approximation factor of this problem since then. There has been a study focused on analyzing a natural algorithm for the problem, resulting in a constant approximation factor worse than 2 [16]. In this paper, we close the gap between the Steiner forest problem and its generalized version, PCSF, by presenting a 2-approximation algorithm for PCSF.

The Steiner tree problem is a well-studied special case of the Steiner forest problem. In the Steiner tree problem, one endpoint of every demand is a specific vertex known as *root*. In contrast to the Steiner forest problem, the approximation factor of the Steiner tree problem has seen significant progress since the introduction of the $(2 - \frac{2}{n})$-approximation algorithm by Goemans and Williamson [14]. Several improvements have been made [26, 24, 22], leading to a 1.39 approximation factor achieved by Byrka, Grandoni, Rothvoß, and Sanità [10]. Lower bounds have also been established, with [21] proving the NP-hardness of the Steiner tree problem and consequently the Steiner forest problem, and [11, 7] demonstrating that achieving an approximation factor within 96/95 is NP-hard. These advancements, along with the established lower bounds, underscore the extensive research conducted in the field of Steiner tree and Steiner forest problems.

Regarding the previous works in the prize-collecting version of these problems, Goemans and Williamson [14] provided a $(2 - \frac{1}{n-1})$-approximation algorithm for prize-collecting Steiner tree (PCST) and prize-collecting TSP problem (PCTSP) in addition to their work on the Steiner forest problem. However, they did not provide an algorithm specifically for the PCSF problem, leaving it as an open problem. Later, Hajiaghayi and Jain [18] in 2006 proposed a deterministic primal-dual $(3 - \frac{2}{n})$-approximation algorithm for the PCSF problem, which inspired our work. They also presented a randomized LP-rounding 2.54-approximation algorithm for the problem. In their paper, they mentioned that finding a better approximation factor, ideally 2, remained an open problem. However, no improvements have been made to their result thus far. Furthermore, other 3-approximation algorithms have been proposed using cost-sharing [15] or iterative rounding [17] (see e.g. [5, 19, 25] for further work on PCSF and its generalizations). Our paper is the first work that improves the approximation factor of [18].

Moreover, advancements have been made in the PCST problem since the initial $(2 - \frac{1}{n-1})$-approximation algorithm by Goemans and Williamson [14]. Archer, Bateni, Hajiaghayi, and Karloff [3] presented a 1.9672-approximation algorithm for PCST, surpassing the barrier of a 2-approximation factor. Additionally, there have been significant advancements in the prize-collecting TSP, which shares similarities with the LP formulation of PCST. Various works have been done in this area [3, 12], and the currently best-known approximation factor is 1.774 [9]. These works demonstrate the importance and interest surrounding prize-collecting problems, emphasizing their significance in the research community.

For a while, the best-known lower bound for the integrality gap of the natural LP relaxation for PCSF was 2. However, Könemann, Olver, Pashkovich, Ravi, Swamy, and Vygen [23] proved that the integrality gap of this LP is at least 9/4. This result suggests that it is not possible to achieve a 2-approximation algorithm for

[1] Indeed Goemans and Williamson [20](Sec 4.6.1) explicitly mention "... the primal-dual algorithm we have presented simulates an algorithm of Agrawal, Klein, and Ravi [AKR95]. Their algorithm was the first approximation algorithm for this [Steiner forest a.k.a. generalized Steiner tree] problem and has motivated much of the authors' research in this area."; the seminal work of Agrawal, Klein, and Ravi [1, 2] recently received *The 30-year STOC Test-of Time Award*.

PCSF solely through primal-dual approaches based on the natural LP, similar to the approaches presented in [18, 17]. This raises doubts about the possibility of achieving an algorithm with an approximation factor better than $9/4$.

However, in this paper, we provide a positive answer to this question. Our main result, Theorem 1, demonstrates the existence of a natural deterministic algorithm for the PCSF problem that achieves a 2-approximate solution in polynomial time.

Theorem 1. There exists a deterministic algorithm for the prize-collecting Steiner forest problem that achieves a 2-approximate solution in polynomial time.

We address the $9/4$ integrality gap by analyzing a natural iterative algorithm. In contrast to previous approaches in the Steiner forest and PCSF fields that compare solutions with feasible dual LP solutions, we compare our solution directly with the optimal solution and assess how much the optimal solution surpasses the dual. It is worth noting that our paper does not rely on the primal and dual LP formulations of the Steiner forest problem. Instead, we employ a coloring schema that shares similarities with primal-dual approaches. While LP techniques could be applied to various parts of our paper, we believe that solely relying on LP would not be sufficient, particularly when it comes to overcoming the integrality gap. Furthermore, although coloring has been used in solving Steiner problems [6], our approach goes further by incorporating two interdependent colorings, making it novel and more advanced.

In addition, we analyze a general approach that can be applied to various prize-collecting problems. In any prize-collecting problem, an algorithm needs to make decisions regarding which demands to pay penalties for and which demands to satisfy. Let us assume that for a prize-collecting problem, we have a base algorithm A. We propose a natural iterative algorithm that begins by running A on an initial instance and storing its solution as one of the options for the final solution. The solution generated by algorithm A pays penalties for some demands and satisfies others. Subsequently, we assume that all subsequent solutions generated by our algorithm will pay penalties for the demands that A paid, set the penalties of these demands to zero, and run A again on the modified instance. We repeat this procedure recursively until we reach a state where algorithm A satisfies every demand with a non-zero penalty, meaning that further iterations will yield the same solution. This state is guaranteed to be reached since the number of non-zero demands decreases at each step. Finally, we obtain multiple solutions for the initial instance and select the one with the minimum cost. This natural iterative algorithm could be effective in solving prize-collecting problems, and in this paper, we analyze its application to the PCSF problem using a variation of the algorithm proposed in [18] as our base algorithm.

One interesting aspect of our findings is that the current best algorithm for the Steiner forest problem achieves an approximation ratio of 2, and this approximation factor has remained unchanged for a significant period of time. It is worth noting that the Steiner forest problem is a specific case of PCSF, where each instance of the Steiner forest can be transformed into a PCSF instance by assigning a sufficiently large penalty to each demand. Since our result achieves the same approximation factor for PCSF, improving the approximation factor for the PCSF problem proves to be more challenging compared to the Steiner forest problem. In future research, it may be more practical to focus on finding a better approximation factor for the Steiner forest problem, which has been an open question for a significant duration. Additionally, investigating the tightness of the 2-approximation factor for both problems could be a valuable direction for further exploration.

1.1 Algorithm and Techniques

In this paper, we introduce a coloring schema that is useful in designing algorithms for Steiner forest, PCSF, and related problems. This coloring schema provides a different perspective from the algorithms proposed by Goemans and Williamson in [14] for Steiner forest and Hajiaghayi and Jain in [18] for PCSF. In Section 2, we provide a detailed representation of the algorithm proposed in [18] using our coloring schema. The use of coloring enhances the intuitiveness of the algorithm, compared to the primal-dual approach utilized in [18], and enables the analysis of our 2-approximation algorithm. Additionally, we introduce a modification to the algorithm of [18], which is essential for the analysis of our 2-approximation algorithm. Finally, in Section 3, we present an iterative algorithm and prove its 2-approximation guarantee for PCSF.

Here, we provide a brief explanation of how coloring intuitively solves the Steiner forest problem. We then present a 3-approximation algorithm and subsequently a 2-approximation algorithm for PCSF.

Steiner forest. We start with an empty forest F to hold our solution. The set FC represents the connected components of F at each moment. A connected component of F is considered an active set if it requires extension to connect with other components and satisfy the demands it cuts. We maintain a subset of FC as active sets in $ActS$. Starting from this point, we consider each edge as a curve with its length equal to its cost.

In each iteration of our algorithm, every active set $S \in ActS$ is assigned a distinct color, which is used to simultaneously color its cutting edges at the same speed. The cutting edges of a set S are defined as the edges that have exactly one endpoint within S. Our coloring procedure proceeds by coloring the remaining uncolored sections of these edges. An edge is in the process of getting colored at a given moment if it connects different connected components of F and has at least one endpoint corresponding to an active set. Additionally, if an edge is a cutting edge for two active sets, it is colored at a speed twice as fast as an edge that is a cutting edge for only one active set. We continue this coloring process continuously until an edge e is fully colored, and then we add it to the forest F. Afterwards, we update FC and $ActS$ accordingly, as defined earlier. It is important to note that we only add edges to F that connect different connected components, ensuring that F remains a forest. Furthermore, since the set of all connected components of F forms a laminar set over time, our coloring schema is also laminar. Refer to Figure 1 for clarity on the coloring process.

At the end of the algorithm, we construct F' from F by removing every edge that is not part of any path between the endpoints of any demand. We then analyze the cost of the optimal solution and our algorithm. Let y_S represent the amount of time that a set S was active and colored its cutting edges. We can show that the cost of the optimal solution is at least $\sum_{S \subset V} y_S$, while our algorithm will find a solution with a cost of at most $2 \sum_{S \subset V} y_S$.

For each active set S, there exists at least one edge in the optimal solution that has exactly one endpoint inside S and the other endpoint outside. This is due to the fact that every active set cuts at least one demand, and the optimal solution must connect all demands. While a set S is active, it colors all of its cutting edges. As the optimal solution includes a cutting edge from S, we can conclude that an amount of y_S from the optimal solution is colored by S. Since each set colors an uncolored portion of the edges, the cost of the optimal solution is at least $\sum_{S \subset V} y_S$.

Furthermore, considering the fixed final forest F', we can observe that at each moment of coloring, when we contract each connected component in FC, it results in a forest where every leaf corresponds to an active set. This observation is based on the fact that if a leaf does not correspond to an active set, it implies that the only edge adjacent to that leaf is unnecessary and should have been removed from F'. Based on this insight,

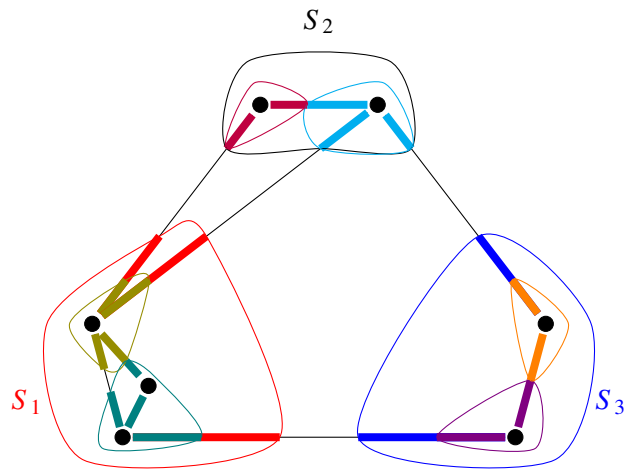

Figure 1: Illustration of the *static coloring* and the coloring used for Steiner forest problem. In the graph, S_2 is inactive and does not color its cutting edges, while S_1 colors edges in red and S_3 colors in blue. It is worth noting that edges within a connected component will not be further colored and will not be added to F.

we can conclude that the number of edges being colored from F' at that moment, which is equivalent to the sum of the degrees of the active sets in the aforementioned forest, is at most twice the number of active sets at that moment. This means that the amount of the newly colored portion of all edges at that moment is at most twice the total value added to all y_S. Therefore, considering that every edge in F' is fully colored, we can deduce that the total length of edges in F' is at most $2\sum_{S\subset V} y_S$.

A 3-approximation algorithm for prize collecting Steiner forest. Similar to the Steiner forest problem, we utilize coloring to solve PCSF. In PCSF, we encounter penalties that indicate it is not cost-effective to connect certain pairs (i, j) if the cost exceeds a specified threshold π_{ij}. To address this challenge, we introduce a coloring schema that assigns a color to each pair (i, j), ensuring that the color is not used to color edges for a duration exceeding its associated potential π_{ij}. However, assigning colors to pairs introduces some challenges. We are not aware of the distribution of potential between the endpoints of a pair, and each set may cut multiple pairs, making it unclear which color should be used at each moment.

To address these challenges, we use two types of coloring. The first type is called *static coloring*, which is similar to the coloring schema used in the Steiner forest problem. In *static coloring*, each set $S \subset V$ is assigned a distinct color. It is referred to as *static coloring* since the colors assigned to edges corresponding to a set S remain unchanged throughout the algorithm. The second type is *dynamic coloring*, which involves coloring edges based on pairs $(i, j) \in V \times V$. Unlike *static coloring*, *dynamic coloring* allows the colors of edges to change during the algorithm, adapting to the evolving conditions. By utilizing both *static coloring* and *dynamic coloring*, we can effectively handle the coloring requirements of PCSF, accounting for the potential constraints and varying edge coloring needs.

Similar to the Steiner forest algorithm, we begin by running the *static coloring* procedure. Whenever an edge is fully colored, we add it to the forest F. However, unlike *static coloring*, we do not maintain a separate *dynamic coloring* throughout the algorithm, as it would require constant reconstructions. Instead, we compute the *dynamic coloring* whenever needed. To obtain the *dynamic coloring*, we map each moment of coloring for each set S in the *static coloring* to a pair (i, j) such that $S \odot (i, j)$, which means S cuts (i, j). This assignment is achieved using a maximum flow algorithm, as described in Section 2.1. We ensure

that our *static coloring* can always be converted to a *dynamic coloring*. Let y_{ij} represent the total duration assigned to pair (i, j) in the *dynamic coloring*. It is important to ensure that y_{ij} does not exceed the potential π_{ij} associated with that pair. If we encounter an active set S for which assigning further coloring to any pair that S cuts would exceed the pair's potential, we deactivate S by removing it from $ActS$.

We define a pair as "tight" if $y_{ij} = \pi_{ij}$. At the end of the algorithm, when every set is inactive, our goal is to pay the penalty for every tight pair. To minimize the number of tight pairs, we perform a local operation by assigning an ϵ amount of color assignment for set S from pair (i, j) to another pair (i', j'), such that (i, j) was tight and after the operation, both pairs are no longer tight. Finally, we pay the penalty for every tight pair and construct F' from F by removing any edges that are not part of a path between pairs that are not tight. It is important to note that if a pair is not tight, it should be connected in F. Otherwise, the sets containing the endpoints of that pair would still be active. Thus, every pair is either connected or we pay its penalty. Let us assume the optimal solution chooses forest F^* and pays penalties for pairs in Q^*.

Since we do not assign more color to each pair (i, j) than its corresponding potential π_{ij}, i.e., $y_{ij} \leq \pi_{ij}$, we can conclude that the optimal solution pays at least $\sum_{(i,j)\in Q^*} y_{ij}$ in penalties. Moreover, similar to the argument for the Steiner forest, the cost of F^* is at least $\sum_{(i,j)\notin Q^*} y_{ij}$. Therefore, the cost of the optimal solution is at least $\sum_{S\subset V} y_S = \sum_{(i,j)\in V\times V} y_{ij}$, while, similar to the argument for the Steiner forest, the cost of F' is at most $2\sum_{S\subset V} y_S$. Moreover, the total penalty we pay is at most $\sum_{(i,j)\in V\times V} y_{ij}$, since we only pay for tight pairs. This guarantees a 3-approximation algorithm.

A 2-approximation algorithm for prize collecting Steiner forest. Let's refer to our 3-approximation algorithm as PCSF3. Our goal is to construct a 2-approximation algorithm called IPCSF, by iteratively invoking PCSF3. In IPCSF, we first invoke PCSF3 and obtain a feasible solution (Q_1, F'_1), where Q_1 represents the pairs for which we pay their penalty, and F'_1 is a forest that connects the remaining pairs. Next, we set the penalty for each pair in Q_1 to 0. We recursively call IPCSF with the updated penalties. Let's assume that (Q_2, F'_2) is the result of this recursive call to IPCSF for the updated penalties. It is important to note that (Q_2, F'_2) is a feasible solution for the initial instance, as it either connects the endpoints of each pair or places them in Q_2. Furthermore, it is true that $Q_1 \subseteq Q_2$, as the penalty of pairs in Q_1 is updated to 0, and they will be considered as tight pairs in further iterations of PCSF3. By induction, we assume that (Q_2, F'_2) is a 2-approximation of the optimal solution for the updated penalties. Now, we want to show that either (Q_1, F'_1) or (Q_2, F'_2) is a 2-approximation of the optimal solution for the initial instance. We will select the one with the lower cost and return it as the output of the algorithm.

To analyze the algorithm, we focus on the *dynamic coloring* of pairs in Q_1 that are connected in the optimal solution. Let $\mathcal{CP}$ denote the set of pairs $(i, j) \in Q_1$ that are connected in the optimal solution. We concentrate on this set because the optimal solution connects these pairs, and we will pay their penalties in both (Q_1, F'_1) and (Q_2, F'_2). Let's assume cp represents the total duration that we color with a pair in $\mathcal{CP}$ in *dynamic coloring*. Each moment of coloring with a pair $(i, j) \in \mathcal{CP}$ in *dynamic coloring* corresponds to coloring with a set S in *static coloring* such that $S \odot (i, j)$. Since $(i, j) \in \mathcal{CP}$ is connected in the optimal solution, we know that S cuts at least one edge of the optimal solution, and S colors that edge in *static coloring*, while (i, j) colors that edge in *dynamic coloring*. Thus, for any moment of coloring with pair $(i, j) \in \mathcal{CP}$ in *dynamic coloring*, we will color at least one edge of the optimal solution. Let cp_1 be the total duration when pairs in $\mathcal{CP}$ color exactly one edge of the optimal solution, and cp_2 be the total duration when pairs in $\mathcal{CP}$ color at least two edges. It follows that $cp_1 + cp_2 = cp$.

We now consider the values of cp_1 and cp_2 to analyze the algorithm. If cp_2 is sufficiently large, we can establish a stronger lower bound for the optimal solution compared to our previous bound, which was $\sum_{S\subset V} y_S$. In the previous bound, we showed that each moment of coloring covers at least one edge of the optimal

solution. However, in this case, we can demonstrate that a significant portion of the coloring process covers at least two edges at each moment. This improved lower bound allows us to conclude that the output of PCSF3, (Q_1, F'_1), becomes a 2-approximate solution.

Alternatively, if cp_1 is significantly large, we can show that the optimal solution for the updated penalties is substantially smaller than the optimal solution for the initial instance. This is achieved by removing the edges from the initial optimal solution that are cut by sets whose color is assigned to pairs in $\mathcal{CP}$ and that set only colored one edge of the optimal solution. By minimizing the number of tight pairs at the end of PCSF3, we ensure that no pair with a non-zero penalty is cut by any of these sets, and removing these edges will not disconnect those pairs. Consequently, we can construct a feasible solution for the updated penalties without utilizing any edges from the cutting edges of these sets in the initial optimal solution. In summary, since (Q_2, F'_2) is a 2-approximation of the optimal solution for the updated penalties, and the optimal solution for the updated penalties has a significantly lower cost than the optimal solution for the initial instance, (Q_2, F'_2) becomes a 2-approximation of the optimal solution for the initial input.

Last but not least, we conduct further analysis of our algorithm to achieve a more refined approximation factor of $2 - \frac{1}{n}$, which asymptotically approaches 2.

1.2 Preliminaries

For a given set $S \subset V$, we define the set of edges that have exactly one endpoint in S as the *cutting edges* of S, denoted by $\delta(S)$. In other words, $\delta(S) = \{(u, v) \in E : |\{u, v\} \cap S| = 1\}$. We say that S cuts an edge e if e is a cutting edge of S, i.e., $e \in \delta(S)$. We say that S cuts a forest F if there exists an edge $e \in F$ such that S cuts that edge.

For a given set $S \subset V$ and pair $\{i, j\} \in V \times V$, we say that S cuts (i, j) if and only if $|\{i, j\} \cap S| = 1$. We denote this relationship as $S \odot (i, j)$.

For a forest F, we define $c(F)$ as the total cost of edges in F, i.e., $c(F) = \sum_{e \in F} c_e$.

For a set of pairs of vertices $Q \subseteq V \times V$, we define $\pi(Q)$ as the sum of penalties of pairs in Q, i.e., $\pi(Q) = \sum_{(i,j) \in Q} \pi_{ij}$.

For a given solution SOL to a PCSF instance I, the notation $cost(\text{SOL})$ is used to represent the total cost of the solution. In particular, if SOL uses a forest F and pays the penalties for a set of pairs Q, then the total cost is given by $cost(\text{SOL}) = c(F) + \pi(Q)$.

For a graph $G = (V, E)$ and a vertex $v \in V$, we define $d_G(v)$ as the degree of v in G. Similarly, for a set $S \subset V$, we define $d_G(S)$ as the number of edges that S cuts, i.e., $|E \cap \delta(S)|$.

Since we use max-flow algorithm in Section 2.1, we provide a formal definition of the MaxFlow function:

Definition 2 (MaxFlow)**.** For the given directed graph G with source vertex *source* and sink vertex *sink*, the function MaxFlow(G, *source*, *sink*) calculates the maximum flow from *source* to *sink* and returns three values: (f^*, C_{min}, f). Here, f^* represents the maximum flow value achieved from *source* to *sink* in G, C_{min} represents the min-cut between *source* and *sink* in G that minimizes the number of vertices on the *source* side of the cut, and f is a function $f : E \to \mathbb{R}^+$ that assigns a non-negative flow value to each edge in the maximum flow.

Throughout this paper, it is important to note that whenever we refer to the term "minimum cut" or "min-cut," we specifically mean the minimum cut that separates *source* from *sink*. Furthermore, we refer to the minimum cut that minimizes the number of vertices on the *source* side of the cut as the "minimal min-cut".

2 Representing a 3-approximation Algorithm

In this section, we present an algorithm that utilizes coloring to obtain a 3-approximate solution. Although the main part of this algorithm closely follows the approach presented by Hajiaghayi and Jain in [18], our novel interpretation of the algorithm is crucial for the subsequent analysis of the 2-approximation algorithm in the next section. Furthermore, we introduce a modification at the end of the 3-approximation algorithm, which plays a vital role in achieving a 2-approximation algorithm in the next section.

From this point forward, we consider each edge as a curve with a length equal to its cost. In our algorithm, we use two types of *coloring*s: *static coloring* and *dynamic coloring*. Both of these *coloring*s are used to assign colors to the edges of the graph, where each part of an edge is assigned a specific color. It is important to note that both *coloring*s have the ability to assign different colors to different portions of the same edge.

First, we introduce some variables that are utilized in Algorithm 2, and we will use them to define the *coloring*s. Let F be a forest that initially is empty, and we are going to add edges to in order to construct a forest that is a superset of our final forest. Moreover, we maintain the set of connected components of F in FC, where each element in FC represents a set of vertices that forms a connected component in F. Additionally, we maintain a set of active sets $ActS \subseteq FC$, which will be utilized for coloring edges in *static coloring*. We will provide further explanation on this later. Initially, we set $ActS = FC$.

Static Coloring. We construct an instance of *static coloring* iteratively by assigning colors to portions of edges. In *static coloring*, each set $S \subset V$ is assigned a unique color. Once a portion of an edge is colored in *static coloring*, its color remains unchanged.

During the algorithm's execution, active sets color their cutting edges simultaneously and at the same speed, using their respective unique colors. As a result, only edges between different connected components are colored at any given moment. When an edge e is fully colored, we add it to F and update FC to maintain the connected components of F. Since e connects two distinct connected components of F, F remains a forest. Furthermore, we update $ActS$ by removing sets that contain an endpoint of e and replacing them with their union. Within the loop at Line 11 of Algorithm 2, we check if an edge has been completely colored, and then merge the sets that contain its endpoints. In addition, we provided a visual representation of the *static coloring* process in Figure 1.

Definition 3 (Static coloring duration)**.** For an instance of *static coloring*, define y_S as the duration during which set S colors its cutting edges using the color S.

It is important to note that we do not need to store the explicit portion of each edge that is colored. Instead, we keep track of y_S, which represents the amount of coloring associated with set S. The portion of edge e that is colored can be computed as $\sum_{S:e\in\delta(S)} y_S$.

Now, we will explain the procedure FindDeltaE, which determines the first moment in time, starting from the current moment, when at least one new edge will become fully colored. This procedure is essential for executing the algorithm in discrete steps.

Finding the maximum value for Δ_e. In FindDeltaE, we determine the maximum value of Δ_e such that continuing the coloring process for an additional duration Δ_e does not exceed the length of any edges. We consider each edge $e = (v, u)$ where v and u are not in the same connected component, and at least one of them belongs to an active set. The portion of edge e that has already been colored is denoted by $\sum_{S:e\in\delta(S)} y_S$. The remaining portion of edge e requires a total time of $(c_e - \sum_{S:e\in\delta(S)} y_S)/t$ to be fully colored, where t is the number of endpoints of e that are in an active set. It is important to note that the coloring speed is

doubled when both endpoints of e are in active sets compared to the case where only one endpoint is in an active set. To ensure that the edge lengths are not exceeded, we select Δ_e as the minimum time required to fully color an edge among all the edges.

Corollary 4. After coloring for Δ_e duration, at least one new edge becomes fully colored.

In Algorithm 1, we outline the procedure for FindDeltaE.

Algorithm 1 Fidning the maximum value for Δ_e

Input: An undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$, an instance of *static coloring* represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, active sets $ActS$, and connected components FC.

Output: Δ_e, the maximum value that can be added to y_S for $S \in ActS$ without violating edge lengths.

```
procedure FindDeltaE(G, y, ActS, FC)
    Initialize Δ_e ← ∞
    for e ∈ E do
        Let S_v, S_u ∈ FC be the sets that contain each endpoint of e.
        t ← |{S_v, S_u} ∩ ActS|
        if S_v ≠ S_u and t ≠ 0 then
            Δ_e ← min(Δ_e, (c_e − Σ_{S : e∈δ(S)} y_S)/t)
    return Δ_e
```

Now, we can utilize FindDeltaE to perform the coloring in discrete steps, as shown in Algorithm 2. In summary, during each step, at Line 6, we call FindDeltaE to determine the maximum duration Δ_e for which we can color with active sets without exceeding the length of any edge, ensuring that at least one edge will be fully colored. Similarly, at Line 7, we utilize FindDeltaP to determine the maximum value of Δ_p that ensures a valid *static coloring* when extending the coloring duration by Δ_p using active sets. The concept of a valid *static coloring*, which avoids purchasing edges when it is more efficient to pay penalties, will be further explained in Section 2.1.

Then, at Line 10, we advance the static coloring process for a duration of $\min(\Delta_e, \Delta_p)$. In the subsequent loop at Line 11, we identify newly fully colored edges and merge their endpoints' sets. Additionally, within the loop at Line 17, we will identify and deactivate sets that should not remain active, as their presence would lead to an invalid *static coloring*. We continue updating our *static coloring* until no active sets remain. Finally, we set Q equal to the set of pairs for which we need to pay penalties in Line 20, and we derive our final forest F' from F by removing redundant edges that are not necessary for connecting demands in $(V \times V) \setminus Q$.

In Algorithm 2, we utilize three functions other than FindDeltaE: FindDeltaP, CheckSetIsTight, and ReduceTightPairs. The purpose of FindDeltaP is to determine the maximum value of Δ_p that allows for an additional coloring duration of Δ_p resulting in a *valid static coloring*. CheckSetIsTight is responsible for identifying sets that cannot color their cutting edges while maintaining the validity of the static coloring. Lastly, ReduceTightPairs aims to reduce the number of pairs for which penalties need to be paid and determine the final set of pairs that we pay their penalty. All of these functions utilize *dynamic coloring*, which will be explained in Section 2.1.

It is important to note that we do not store a *dynamic coloring* within PCSF3 since it changes constantly. Instead, we compute a *dynamic coloring* based on the current *static coloring* within these functions, as they are the only parts of our algorithm that require a *dynamic coloring*. Note that at the end of PCSF3, we require a final *dynamic coloring* for the analysis in Section 3. This final coloring will be computed in ReduceTightPairs at Line 20.

Algorithm 2 A 3-approximation Algorithm

Input: An undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$ and penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$.
Output: A set of pairs Q with a forest F' that connects the endpoints of every pair $(i, j) \notin Q$.

1: **procedure** PCSF3($I = (G, \pi)$)
2: Initialize $F \leftarrow \emptyset$
3: Initialize $ActS, FC \leftarrow \{\{v\} : v \in V\}$
4: Implicitly set $y_S \leftarrow \emptyset$ for all $S \subset V$
5: **while** $ActS \neq \emptyset$ **do**
6: $\Delta_e \leftarrow$ FindDeltaE($G, y, ActS, FC$)
7: $\Delta_p \leftarrow$ FindDeltaP($G, \pi, y, ActS$)
8: $\Delta \leftarrow \min(\Delta_e, \Delta_p)$
9: **for** $S \in ActS$ **do**
10: $y_S \leftarrow y_S + \Delta$
11: **for** $e \in E$ **do**
12: Let $S_v, S_u \in FC$ be sets that contains each endpoint of e
13: **if** $\sum_{S : e \in \delta(S)} y_S = c_e$ **and** $S_v \neq S_u$ **then**
14: $F \leftarrow F \cup \{e\}$
15: $FC \leftarrow (FC \setminus \{S_p, S_q\}) \cup \{S_p \cup S_q\}$
16: $ActS \leftarrow (ActS \setminus \{S_p, S_q\}) \cup \{S_p \cup S_q\}$
17: **for** $S \in ActS$ **do**
18: **if** CheckSetIsTight(G, π, y, S) **then**
19: $ActS \leftarrow ActS \setminus \{S\}$
20: $Q \leftarrow$ ReduceTightPairs(G, π, y)
21: Let F' be the subset of F obtained by removing unnecessary edges for connecting demands $(V \times V) \setminus Q$.
22: **return** (Q, F')

Now, let's analyze the time complexity of FindDeltaE as described in Lemma 6. In Lemma 31, we will demonstrate that the overall time complexity of PCSF3 is polynomial. This will be achieved after explaining and analyzing the complexity of the subroutines it invokes.

Lemma 5. In the PCSF3 algorithm, the number of sets that have been active at some point during its execution is linear.

Proof. During the algorithm, new active sets are only created in Line 16 by merging existing sets. Initially, we start with n active sets in $ActS$. Symmetrically, for each creation of a new active set, we have one merge operation over sets in FC, which reduces the number of sets in FC by exactly one. Since we start with n sets in FC, the maximum number of merge operations is $n - 1$. Therefore, the total number of active sets throughout the algorithm is at most $2n - 1$. □

Lemma 6. The runtime of FindDeltaE is polynomial.

Proof. In Line 3, we iterate over the edges, and the number of edges is polynomial. In addition, since each set S with $y_S > 0$ has been active at some point, the number of these sets is linear due to Lemma 5. Consequently, for each edge, we calculate the sum in Line 7 in linear time by iterating through such sets. Therefore, we can conclude that FindDeltaE runs in polynomial time. □

2.1 Dynamic Coloring

The *dynamic coloring* is derived from a given *static coloring*. In *dynamic coloring*, each pair $(i, j) \in V \times V$ is assigned a unique color. The goal is to assign each moment of coloring in *static coloring* with each active set $S \subset ActS$, to a pair $(i, j) \in V \times V$ where $S \odot (i, j)$ holds, and color the same portion that set S colored at that moment in *static coloring* with the color of pair (i, j) in *dynamic coloring*. Furthermore, there is a constraint on the usage of each pair's color. We aim to avoid using the color of pair (i, j) for more than a total duration of π_{ij}. It's important to note that for a specific *static coloring*, there may be an infinite number of different *dynamic colorings*, but we only need to find one of them.

Now, we will introduce some notations that are useful in our algorithm and analysis.

Definition 7 (Dynamic Coloring Assignment Duration)**.** In a *dynamic coloring* instance, for each set S and pair (i, j) where $S \odot (i, j)$, y_{Sij} represents the duration of coloring with color S in *static coloring* that is assigned to pair (i, j) for coloring in *dynamic coloring*.

Definition 8 (Dynamic Coloring Duration)**.** In a *dynamic coloring* instance, y_{ij} represents the total duration of coloring with pair (i, j) in *dynamic coloring*, denoted as $y_{ij} = \sum_{S:S\odot(i,j)} y_{Sij}$.

Definition 9 (Pair Constraint and Tightness)**.** In a *dynamic coloring* instance, the condition that each pair (i, j) should not color for more than π_{ij} total duration (i.e., $y_{ij} \leq \pi_{ij}$) is referred to as the *pair constraint*. If this condition is tight in the *dynamic coloring* for a pair (i, j), i.e., $y_{ij} = \pi_{ij}$, we say that pair (i, j) is a tight pair.

Definition 10 (Valid Static Coloring)**.** A *static coloring* is considered valid if there exists a *dynamic coloring* for the given *static coloring*. In other words, the following conditions must hold:

- For every set $S \subset V$, we can distribute the duration of the static coloring for S among pairs (i, j) that satisfy $S \odot (i, j)$, such that $\sum_{(i,j):S\odot(i,j)} y_{Sij} = y_S$.
- For every pair $(i, j) \in V \times V$, the pair constraint is not violated, i.e., $y_{ij} = \sum_{S:S\odot(i,j)} y_{Sij} \leq \pi_{ij}$.

If there is no *dynamic coloring* that satisfies these conditions, the *static coloring* is considered invalid.

Note that in the definition of *valid static coloring*, the validity of a *static coloring* is solely determined by the duration of using each color, denoted as y_S, and the specific timing of using each color is not relevant. Moreover, a function $y : 2^V \to \mathbb{R}_{\geq 0}$ is almost sufficient to describe a *static coloring*, as it indicates the duration for which each set S colors its cutting edges. Therefore, this function provides information about the portion of each edge that is colored with each color. This information is enough for our algorithm and analysis. We are not concerned with the precise location on an edge where a specific color is applied. Instead, our focus is on determining the amount of coloring applied to each edge with each color. Similarly, the function $y : 2^V \times V \times V \to \mathbb{R}_{\geq 0}$ is enough for determining a *dynamic coloring*.

Definition 11 (Set Tightness)**.** For a valid instance of *static coloring*, we define a set $S \subset V$ as tight if increasing the value of y_S by any $\epsilon > 0$ in the *static coloring* without changing the coloring duration of other sets would make the *static coloring* invalid.

Lemma 12. In a *valid static coloring*, if set S is tight, then for any corresponding *dynamic coloring*, all pairs (i, j) such that $S \odot (i, j)$ are tight.

Proof. Consider an arbitrary *dynamic coloring* of the given *valid static coloring*. Using contradiction, assume there is a pair (i, j) such that $S \odot (i, j)$, and this pair is not tight. Let $\epsilon = \pi_{ij} - y_{ij}$. If we increase y_S, y_{Sij}, and y_{ij} by ϵ, it results in a new *static coloring* and a new *dynamic coloring*. In the new *dynamic coloring*, since for every set S' we have $\sum_{(i',j'):S'\odot(i',j')} y_{S'i'j'} = y_{S'}$, and for every pair (i', j') we have

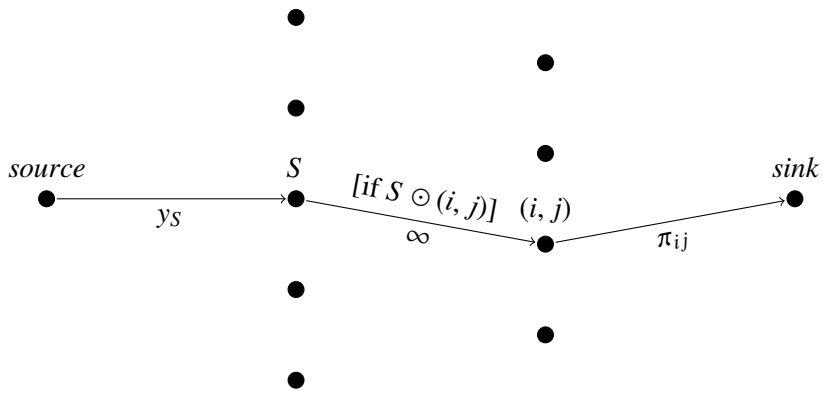

Figure 2: SetPairGraph

$y_{i'j'} = \sum_{S':S'\odot(i',j')} y_{S'i'j'} \leq \pi_{i'j'}$, based on Definition 10, increasing y_S by ϵ results in a valid *static coloring*. According to Definition 11, this contradicts the tightness of S. Therefore, we can conclude that all pairs (i, j) for which $S \odot (i, j)$ holds are tight. □

However, it is possible for every pair (i, j) satisfying $S \odot (i, j)$ to be tight in a *dynamic coloring*, while the set S itself is not tight. We will describe the process of determining whether a set is tight in Algorithm 4.

So far, we have explained several key properties and concepts related to *dynamic coloring*. Now, let us explore how we can obtain a *dynamic coloring* from a given *valid static coloring*. To accomplish this, we introduce the concept of SetPairGraph, which represents a graph associated with each *static coloring*. By applying the max-flow algorithm to this graph, we can determine a corresponding *dynamic coloring*. The definition of SetPairGraph is provided in Definition 13, and Figure 2 illustrates this graph.

Definition 13 (SetPairGraph)**.** Given a graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$, and penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, as well as an instance of *static coloring* represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, we define a directed graph $\mathcal{G}$ initially consisting of two vertices *source* and *sink*. For every set $S \subset V$ such that either $y_S > 0$ or $S \in ActS$, we add a vertex to $\mathcal{G}$ and a directed edge from *source* to S with a capacity of y_S. Additionally, for each pair $(i, j) \in V \times V$, we add a vertex to $\mathcal{G}$ and a directed edge from (i, j) to *sink* with a capacity of π_{ij}. Finally, we add a directed edge from each set S to each pair (i, j) such that $S \odot (i, j)$, with infinite capacity. We refer to the graph $\mathcal{G}$ as SetPairGraph(G, π, y).

Now, we compute the maximum flow from *source* to *sink* in SetPairGraph(G, π, y). We assign the value of y_{Sij} to the amount of flow from S to (i, j), representing the allocation of coloring from set S in *static coloring* to pair (i, j) in *dynamic coloring*. Similarly, we set y_{ij} equal to the amount of flow from (i, j) to *sink*, indicating the duration of coloring for pair (i, j) in *dynamic coloring*. In Lemma 14, we show that if the maximum flow equals $\sum_{S\subset V} y_S$, the *static coloring* is valid, and the assignment of y_{Sij} and y_{ij} satisfies all requirements.

Lemma 14. For a given graph $G = (V, E, c)$, penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, and an instance of *static coloring* represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, let $(f^*, C_{min}, f) =$ MaxFlow(SetPairGraph(G, π, y), *source*, *sink*), the *static coloring* is valid if and only if $f^* = \sum_{S\subset V} y_S$.

Proof. Assume that $f^* = \sum_{S\subset V} y_S$. Since the sum of the capacities of the outgoing edges from *source* is equal to the maximum flow, the amount of flow passing through S is y_S. Hence, given that the total flow

coming out from S is equal to $\sum_{(i,j)\in V\times V} y_{Sij}$, we have $y_S = \sum_{(i,j)\in V\times V} y_{Sij}$. It should be noted that $y_{Sij} > 0$ only if $S \odot (i, j)$, as we only have a directed edge from S to (i, j) in that case.

Furthermore, since the capacity of the edge from (i, j) to *sink* is π_{ij}, the sum of incoming flow to (i, j) is at most π_{ij}. Thus, $y_{ij} = \sum_{S\subset V: S\odot(i,j)} y_{Sij} \leq \pi_{ij}$.

Now, assume that the given *static coloring* is valid. Consider its corresponding SetPairGraph and *dynamic coloring*. Let the amount of flow from *source* to S be y_S. Let the amount of flow in the edge between S and (i, j) be y_{Sij}, representing the assignment duration in the *dynamic coloring*. Similarly, let the amount of flow in the edge between (i, j) and *sink* be y_{ij}, representing the duration in the *dynamic coloring*. According to Definition 10, in a *valid static coloring*, the following conditions hold: $\sum_{(i,j):S\odot(i,j)} y_{Sij} = y_S$ and $y_{ij} = \sum_{S:S\odot(i,j)} y_{Sij} \leq \pi_{ij}$. Therefore, in the SetPairGraph, the assignment of flow satisfies the edge capacities and the equality of incoming and outgoing flows for every vertex except *source* and *sink*. Furthermore, since we fulfill all outgoing edges from *source*, the maximum flow is $\sum_{S\subset V} y_S$. □

Let us define C_{source} as the cut in SetPairGraph that separates *source* from other vertices. Since the sum of the edges in C_{source} is $\sum_{S\subset V} y_S$, the following corollary can easily be concluded from Lemma 14.

Corollary 15. For a given graph $G = (V, E, c)$, penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, and an instance of *static coloring* represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, the *static coloring* is valid if and only if C_{source} is a minimum cut between *source* and *sink* in SetPairGraph(G, π, y).

To analyze the size of the SetPairGraph and the complexity of running the max-flow algorithm on it, we can refer to the following lemma.

Lemma 16. At any point during the algorithm, the size of the SetPairGraph remains polynomial.

Proof. In the SetPairGraph, vertices are assigned to sets that are either active or have $y_S > 0$. This implies that for each set that is active at least once, there is at most one corresponding vertex in the graph. According to Lemma 5, the number of such vertices is linear.

In addition to the active set vertices, the SetPairGraph also includes vertices *source*, *sink*, and pairs (i, j). The total number of such vertices is $2 + n^2$. Thus, the overall size of the graph is polynomial. □

Now that we understand how to find a *dynamic coloring* given a *static coloring* using the max-flow algorithm, we can use this approach to develop the functions FindDeltaP, CheckSetIsTight, and ReduceTightPairs.

Finding the maximum value for Δ_p. In FindDeltaP, our goal is to determine the maximum value of Δ_p such that if we continue coloring with active sets for an additional duration of Δ_p in the *static coloring*, it remains a *valid static coloring*. The intuition behind this algorithm is to start with an initial upper bound for Δ_p and iteratively refine it until we obtain a *valid static coloring*. This process involves adjusting the parameters and conditions of the coloring to gradually tighten the upper bound. Algorithm 3 presents the pseudocode for FindDeltaP. The proof of the following lemma illustrates how the iterations of this algorithm progress toward the correct value of Δ_p.

Lemma 17. In a *valid static coloring*, the maximum possible duration to continue the coloring process while ensuring the validity of the *static coloring* is Δ_p = FindDeltaP$(G, \pi, y, ActS)$.

Proof. Consider the SetPairGraph of the given *valid static coloring*. If increasing the capacity of edges from *source* to S for every set $S \in ActS$ by Δ_p results in an increase in the min-cut by $|ActS| \cdot \Delta_p$, then the

resulting *static coloring* remains valid since C_{source} remains a min-cut. This is based on Corollary 15. Thus, our goal is to find the maximum value for Δ_p that satisfies this condition.

First, in Line 3, we initialize Δ_p with an upper-bound value of $(\sum_{ij} \pi_{ij} - \sum_S y_S)/|ActS|$. This value serves as an upper-bound because the increase in min-cut cannot exceed $(\sum_{ij} \pi_{ij} - \sum_S y_S)$, as determined by the cut that separates *sink* from the other vertices. Next, we update the capacity of edges from *source* to the active sets in SetPairGraph by the value of Δ_p, and then calculate the minimal min-cut C_{min}. If C_{min} separates *source* from the other vertices, it indicates that the new *static coloring* is valid according to Corollary 15. At this point, the function terminates and returns the current value of Δ_p in Line 8.

Otherwise, if C_{source} is not a min-cut after updating the edges, we want to prove that C_{min} has at least one active set in the side of *source*. Let's assume, for contradiction, that C_{min} does not have any active sets on the side of *source*. According to Corollary 15, prior to updating the edges, C_{source} was a min-cut since we had a *valid static coloring*. Thus, the weight of C_{min} was at least equivalent to the weight of C_{source} before the edges were modified. Given that C_{min} and C_{source} include all the edges connecting *source* to the active sets, adding Δ_p to active sets leads to an increase in the weight of both C_{min} and C_{source} by $|ActS| \cdot \Delta_p$. Consequently, the weight of C_{min} remains at least as large as the weight of C_{source}. However, if C_{source} is not a minimum cut after updating the edges, it implies that C_{min} cannot be a minimum cut either, which contradicts the definition of C_{min}. Therefore, we can conclude that C_{min} must have at least one active set on the same side as *source*.

Let $k \geq 1$ represent the number of active sets on the same side as *source* in C_{min}. Referring to Definition 2, it is evident that C_{min} minimizes k. If we decrease Δ_p by ϵ, it deducts $|ActS| \cdot \epsilon$ from the weight of C_{source} and $(|ActS| - k) \cdot \epsilon$ from weight of C_{min}. Given that the weight of C_{source} is $|ActS| \cdot \Delta_p + \sum_{S \subset V} y_S$, and the weight of C_{min} is f^*, in order to establish C_{source} as a minimum cut, we want to find the minimum value of ϵ satisfying:

$$
\begin{aligned}
f^* - (|ActS| - k) \cdot \epsilon &\geq |ActS| \cdot \Delta_p + \sum_{S \subset V} y_S - |ActS| \cdot \epsilon \\
f^* + k\epsilon &\geq |ActS| \cdot \Delta_p + \sum_{S \subset V} y_S \\
k\epsilon &\geq |ActS| \cdot \Delta_p + \sum_{S \subset V} y_S - f^* \\
\epsilon &\geq \frac{|ActS| \cdot \Delta_p + \sum_{S \subset V} y_S - f^*}{k}
\end{aligned}
$$

Now, let us define $\epsilon^* = (|ActS| \cdot \Delta_p + \sum_{S \subset V} y_S - f^*)/k$. When we decrease Δ_p by ϵ^*, we effectively tighten the upper bound on Δ_p. This is crucial because reducing Δ_p by a smaller value would make it impossible for C_{source} to become a min-cut. Such a violation would contradict the validity of the *static coloring*, as indicated by Corollary 15. After updating Δ_p to its new value, if C_{source} indeed becomes a minimum cut, the procedure is finished. However, if the new minimal min-cut still contains active sets on the side of *source*, their number must be less than k.

To prove this by contradiction, let's assume that in the new minimal min-cut, the number of active sets on the side of *source* is greater than or equal to k. Due to the minimality of the new minimum cut, it can be observed that all minimum cuts for the updated Δ_p have at least k active sets on the *source* side. In other words, these minimum cuts have at most $|ActS| - k$ active sets on the other side. Consequently, the weight of these minimum cuts is reduced by at most $(|ActS| - k) \cdot \epsilon^*$. Since we have specifically reduced $(|ActS| - k) \cdot \epsilon^*$ from C_{min}, it remains a minimum cut. Moreover, after decreasing ϵ^* from Δ_p based on how we determine ϵ^*, C_{min} and C_{source} have the same weight. This implies that C_{source} is also a valid minimum cut and should

be the minimal min-cut. This contradiction suggests that after the reduction, the number of vertices on the *source* side has indeed decreased.

Finally, we repeat the same procedure until C_{source} becomes a min-cut. Given that there are at most n active sets in $ActS$, and each iteration reduces the number of active sets on the side of *source* in the minimal min-cut by at least one, after a linear number of iterations, all active sets will be moved to the other side, and the desired value of Δ_p will be determined. Since each time we have demonstrated that the value of Δ_p serves as an upper bound, it represents the maximum possible value that allows for a *valid static coloring*. □

Lemma 18. In a *valid static coloring*, the *static coloring* remains valid if we continue the coloring process by at most $\Delta_p = \textsc{FindDeltaP}(G, \pi, y, ActS)$.

Proof. Consider the SetPairGraph of the given *valid static coloring*. We want to show that for every value $\Delta'_p \leq \Delta_p$, the coloring remains valid. According to Lemma 17, we know that increasing the duration of the active sets by Δ_p results in a *valid static coloring*. Therefore, by increasing the capacity of edges from *source* to S for each set $S \in ActS$ by Δ_p, C_{source} represents a minimum cut. Decreasing the capacities of these edges by a non-negative value $d = \Delta_p - \Delta'_p$ results in a decrease in the weight of C_{source} by $|ActS| \cdot d$, while other cuts are decreased by at most this value. Consequently, C_{source} remains a minimum cut, and as indicated in Corollary 15, the static coloring remains valid after increasing y_S for active sets by $\Delta'_p \leq \Delta_p$. □

Based on the proof of Lemma 17, it is clear that the while loop in the FindDeltaP function iterates a linear number of times. Furthermore, during each iteration, we make a single call to the MaxFlow procedure on the SetPairGraph, which has a polynomial size according to Lemma 16. Consequently, we can deduce the following corollary.

Corollary 19. Throughout the algorithm, each call to the FindDeltaP function executes in polynomial time.

Now, we can prove a significant lemma that demonstrates that our algorithm behaves as expected.

Lemma 20. Throughout the algorithm, we always maintain a *valid static coloring*.

Proof. We can establish the validity of the *static coloring* throughout the algorithm using induction. Initially, since $y_S = 0$ for all sets S, assigning $y_{Sij} = 0$ and $y_{ij} = 0$ results in a *dynamic coloring*, satisfying the conditions of a *valid static coloring*.

Assuming that at the beginning of each iteration, we have a *valid static coloring* based on the induction hypothesis. Furthermore, during each iteration, we continue the coloring process for a duration of $\min(\Delta_p, \Delta_e) \leq \Delta_p$. According to Lemma 18, this coloring preserves the validity of the *static coloring*.

Therefore, by induction, we can conclude that throughout the algorithm, we maintain a *valid static coloring*. □

Check if a set is tight. Here, we demonstrate how to utilize max-flow on SetPairGraph to determine if a set S is tight. If it is indeed tight, we proceed to remove it from $ActS$ in Line 19 of PCSF3.

Checking the tightness of a set is straightforward, as outlined in Definition 11. To determine if set S is tight, we increase the capacity of the directed edge from *source* to S and check if the flow from *source* to *sink* exceeds $\sum_{S \subset V} y_S$. The pseudocode for this function is provided in Algorithm 4.

Lemma 21. Each call to the CheckSetIsTight function during the algorithm runs in polynomial time.

Algorithm 3 Finding the maximum value for Δ_p

Input: An undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$, penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, an instance of *static coloring* represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, and active sets $ActS$.

Output: Δ_p, the maximum value that can be added to y_S for $S \in ActS$ without violating the validity of the *static coloring*.

```
function FindDeltaP(G, π, y, ActS)
    𝒢 ← SetPairGraph(G, π, y)
    Δ_p ← (Σ_ij π_ij − Σ_S y_S) / |ActS|
    while true do
        Set the capacity of edges from source to S ∈ ActS equals to y_S + Δ_p in graph 𝒢.
        (f*, C_min, f) ← MaxFlow(𝒢, source, sink)
        if f* = |ActS| · Δ_p + Σ_{S⊂V} y_S then
            return Δ_p
        Let k represent the number of active sets on the same side of the cut C_min as source.
        Δ_p ← Δ_p − (|ActS| · Δ_p + Σ_{S⊂V} y_S − f*) / k
```

Algorithm 4 Check if set $S \in ActS$ is tight

Input: An undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$, penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, an instance of *static coloring* represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, and a set $S \subset V$.

Output: *True* if set S is tight, *False* otherwise.

```
function CheckSetIsTight(G, π, y, S)
    𝒢 ← SetPairGraph(G, π, y)
    Set the capacity of the edge from source to S in graph 𝒢 equals to y_S + 1.
    (f*, C_min, f) ← MaxFlow(𝒢, source, sink)
    if f* > Σ_{S⊂V} y_S then
        return False
    else
        return True
```

Proof. The CheckSetIsTight function call MaxFlow once on the SetPairGraph, whose size remains polynomial throughout the algorithm according to Lemma 16. Therefore, both the MaxFlow and the CheckSetIsTight function run in polynomial time. □

Reduce the number of tight pairs. At the end of PCSF3, we obtain a final *valid static coloring*, from which we can derive a corresponding final *dynamic coloring* which corresponds to a max-flow in SetPairGraph. Next, we present the process of reducing the number of tight pairs in the final *dynamic coloring*, aiming to achieve a *minimal dynamic coloring*. This step is essential to obtain a 2-approximate solution for PCSF in the next section.

Definition 22 (Minimal Dynamic Coloring)**.** A *dynamic coloring* is considered *minimal dynamic coloring* if there are no pairs $(i, j), (i', j') \in V \times V$ and set $S \subset V$ such that pair (i, j) is a tight pair while (i', j') is not a tight pair, $S \odot (i, j)$, $S \odot (i', j')$, and $y_{Sij} > 0$.

To obtain a *minimal dynamic coloring*, we first check if there exist pairs $(i, j), (i', j')$ and a set S meeting the following criteria: pair (i, j) is tight, pair (i', j') is not tight, $S \odot (i, j)$, $S \odot (i', j')$, and $y_{Sij} > 0$. If such pairs and set exist, we proceed with the following adjustments. Since (i', j') is not tight, we have $\pi_{i'j'} - y_{i'j'} > 0$. Additionally, $y_{Sij} > 0$ is assumed. Therefore, there exists $\epsilon > 0$ such that $\epsilon < \min(y_{Sij}, \pi_{i'j'} - y_{i'j'})$. Given

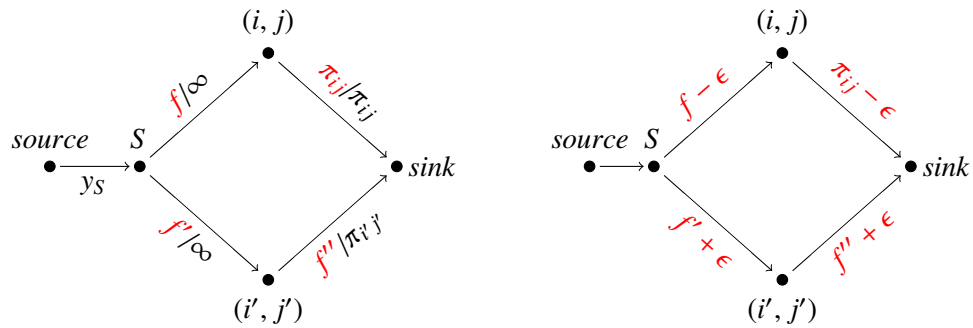

Figure 3: By choosing a small positive value $\epsilon < \min(f, \pi_{i'j'} - f'')$, we can remove one tight pair. The red variables represent the amounts of flow on each edge, while the black variables represent their capacity.

that $\epsilon < y_{Sij} \leq y_{ij}$, we can reduce y_{Sij} and y_{ij} by ϵ, while adding ϵ to $y_{Si'j'}$ and $y_{i'j'}$. Since $\epsilon > 0$, pair (i, j) is no longer tight, and since $\epsilon < \pi_{i'j'} - y_{i'j'}$ for the previous value of $y_{i'j'}$, pair (i', j') will not become tight. It is important to note that the *dynamic coloring* prior to these changes corresponds to a max-flow in SetPairGraph. Implementing these adjustments on the flow of edges associated with y_{Sij}, y_{ij}, $y_{Si'j'}$, and $y_{i'j'}$ results in a new max-flow that corresponds to the updated *dynamic coloring*. This provides an intuition for why the assignment in the new *dynamic coloring* remains valid. We illustrate these flow changes in Figure 3, and the complete process for achieving a minimal *dynamic coloring* is described in Algorithm 5.

After applying these adjustments, the number of tight pairs is reduced by one. If there are no tight pairs where their tightness can be removed through this operation, the result is a *minimal dynamic coloring*. Given that the number of tight pairs is at most n^2, and after each operation, the number of tight pairs is reduced by one, after a maximum of n^2 iterations in ReduceTightPairs, the number of tight pairs becomes minimal.

Considering that the number of iterations in the ReduceTightPairs function is polynomial and the MaxFlow operation on the SetPairGraph in Line 3 has a polynomial size (Lemma 16), it can be concluded that the runtime of ReduceTightPairs is polynomial.

Corollary 23. The ReduceTightPairs function runs in polynomial time.

Algorithm 5 Reduce the number of tight pairs

Input: An undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$ and penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$.
Output: The set Q of tight pairs for which we will pay penalties.

1: **function** ReduceTightPairs(G, π, y)
2: $\quad \mathcal{G} \leftarrow$ SetPairGraph(G, π, y)
3: $\quad (f^*, C_{min}, f) \leftarrow$ MaxFlow$(\mathcal{G}, source, sink)$
4: $\quad$ Let $y_{Sij} \leftarrow f(e)$ for each set S and each pair (i, j) that $S \odot (i, j)$, where e is the edge from S to (i, j).
5: $\quad$ Let $y_{ij} \leftarrow f(e)$ for each pair (i, j), where e is the edge from (i, j) to *sink*.
6: $\quad$ **for** $(i, j), (i', j') \in V \times V$ and $S \subset V$ that $S \odot (i, j)$, $S \odot (i', j')$, $y_{ij} = \pi_{ij}$, $y_{i'j'} < \pi_{i'j'}$, and $y_{S'ij} > 0$ **do**
7: $\quad\quad y_{Sij} \leftarrow y_{Sij} - \epsilon$
8: $\quad\quad y_{ij} \leftarrow y_{ij} - \epsilon$
9: $\quad\quad y_{Si'j'} \leftarrow y_{Si'j'} + \epsilon$
10: $\quad\quad y_{i'j'} \leftarrow y_{i'j'} + \epsilon$
11: $\quad$ Let $Q \leftarrow \{(i, j) \in V \times V : \sum_{S : S \odot (i,j)} y_{Sij} = \pi_{ij}\}$
12: $\quad$ **return** Q

Finally, after obtaining a *minimal dynamic coloring*, we consider it as our final *dynamic coloring*, which will be used in the analysis presented in Section 3.1.

Corollary 24. The final *dynamic coloring* obtained at the end of procedure PCSF3 is a *minimal dynamic coloring*.

Furthermore, in a *minimal dynamic coloring*, we establish the following lemma, which is necessary for the analysis presented in the next section.

Lemma 25. In a *minimal dynamic coloring*, if a set $S \subset V$ cuts a tight pair $(i, j) \in V \times V$ with $y_{Sij} > 0$, then all pairs (i', j') satisfying $S \odot (i', j')$ are also tight.

Proof. Assume there exists a pair (i', j') satisfying $S \odot (i', j')$ that is not tight. This implies that the pairs (i, j) and (i', j'), along with set S, contradict the definition of *minimal dynamic coloring* (Definition 22). □

2.2 Analysis

In this section, we demonstrate the validity of our algorithm's output for the given PCSF instance. We also present some lemmas that are useful for proving the approximation factor of PCSF3. However, we do not explicitly prove the approximation factor of PCSF3 in this section, as it is not crucial for our main result. Nonetheless, one can easily conclude the 3-approximation factor of PCSF3 using Lemmas 35, 37, and 34 provided in the next section. Additionally, in Lemma 31, we show that PCSF3 has a polynomial time complexity. The lemmas provided in this section are also necessary for the analysis of our 2-approximation algorithm, which is presented in the next section.

To conclude the correctness of our algorithm, it is crucial to show that our algorithm pays penalties for all pairs that are not connected in F'. In other words, every pair that is not tight will be connected in F'. This ensures that by paying the penalties for tight pairs and the cost of edges in F', we obtain a feasible solution.

To prove this, we introduce some auxiliary lemmas. First, in Lemma 28, we demonstrate that when a set becomes tight during PCSF3, it remains tight until the end of the algorithm. This lemma is essential because if a set becomes tight and is subsequently removed from the active sets, but then becomes non-tight again, it implies that some pairs could contribute to the coloring in the *dynamic coloring*, but their colors may no longer be utilized.

Furthermore, in Lemma 29, we establish that every connected component of F at the end of PCSF3 is a tight set. This provides additional evidence that the algorithm produces a valid solution.

Finally, we use these lemmas to prove the validity of the solution produced by PCSF3 in Lemma 30.

Let C_{source} be the cut in SetPairGraph that separates *source* from the other vertices.

Lemma 26. At every moment of PCSF3, in the SetPairGraph representation corresponding to the *static coloring* of that moment, C_{source} is a minimum cut between *source* and *sink*.

Proof. According to Lemma 20, the *static coloring* is always valid during PCSF3. Moreover, based on Corollary 15, when the *static coloring* is valid, C_{source} represents a minimum cut. □

Lemma 27. A set $S \subset V$ is tight if and only if there exists a minimum cut between *source* and *sink* in SetPairGraph representation of a valid *dynamic coloring* that does not contain the edge e from *source* to S.

Proof. Using contradiction, let's assume that S is tight and all minimum cuts contain the edge e. Let $\epsilon > 0$ be the difference between the weight of the minimum cut and the first cut whose weight is greater than the minimum cut. By increasing the capacity of the edge e by ϵ, the weight of any minimum cut increases by a positive value ϵ, as well as the maximum flow. This implies that we can increase y_S and still maintain a *valid static coloring*. Therefore, based on the definition of set tightness (Definition 11), we can conclude that set S is not tight. This contradicts the assumption of the tightness of S and proves that there exists a minimum cut that does not contain the edge e.

Furthermore, if we have a minimum cut that does not contain edge e, increasing the capacity of e does not affect the value of that minimum cut and respectfully the maximum flow. By using Lemma 14, we conclude that increasing y_S would result in an invalid *static coloring*. Therefore, based on Definition 11, we can conclude that set S is tight. □

Lemma 28. Once a set S becomes tight, it remains tight throughout the algorithm.

Proof. According to Lemma 26, C_{source} is always a minimum cut. Let us assume that at time t, the set S becomes tight. Based on Lemma 27, there exists a minimum cut C_S that has S on the side of *source*. Therefore, at time t, C_{source} and C_S have the same weight. Now, let us consider a contradiction by assuming that there is a time $t' > t$ when S is not tight. The only difference between SetPairGraph at time t and time t' is the increased capacity of some edges between *source* and sets $S' \subset V$. Let us assume that the total increase in all $y_{S'}$ from time t to t' is d. Since all of these edges are part of the cut C_{source}, the weight of the cut C_{source} is increased by d. Furthermore, since the total capacity of all edges in SetPairGraph from time t to t' has increased by d, the weight of C_S cannot have increased by more than d. That means, the weight of C_S cannot exceed the weight of C_{source} at time t'. Since C_{source} is a minimum cut at time t' according to Lemma 26, we can conclude that C_S remains a minimum cut at time t'. Therefore, based on Lemma 27, the set S is still tight at time t', which contradicts the assumption that it is not tight. □

Lemma 29. At the end of PCSF3, all remaining sets in FC are tight.

Proof. In Line 3 of the algorithm, both $ActS$ and FC are initialized with the same set of sets. Additionally, in Lines 15 and 16, the same sets are removed from $ActS$ and FC or added to both data structures. The only difference occurs in Line 19, where tight sets are removed from $ActS$ but not from FC. Given Lemma 28, these sets are tight at the end of PCSF3. Therefore, at the end of the algorithm, since there are no sets remaining in $ActS$, all sets in FC are tight. □

Lemma 30. After executing ReduceTightPairs, the endpoints of any pair that is not tight will be connected in the forest F'.

Proof. The forest F' is obtained by removing redundant edges from F, which are edges that are not part of a path between the endpoints of a pair that is not tight. Hence, we only need to show that every pair that is not tight is connected in F. Let us assume, for the sake of contradiction, that there exists a pair (i, j) that is not tight and the endpoints i and j are not connected in F. Consider the set $S \in FC$ at the end of the algorithm that contains i. Since i and j are not connected in F, and S is a connected component of F, it follows that S cuts the pair (i, j). According to Lemma 29, S is a tight set. This contradicts Lemma 12 because we have a tight set S such that $S \odot (i, j)$ is not tight. Therefore, our assumption is false, and every pair that is not tight is connected in F. As a result, after executing ReduceTightPairs, the endpoints of any pair that is not tight will be connected in the forest F'. □

Now we will prove that the running time of PCSF3 is polynomial.

Lemma 31. For instance I, the runtime of PCSF3 is polynomial.

Proof. We know that Δ_e denotes the time it takes for at least one new edge to be fully colored according to Corollary 4, and Δ_p signifies the time required for at least one active set to be deactivated based on the maximality of Δ_p demonstrated in Lemma 17. During each iteration of the while loop at Line 5, it is guaranteed that at least one of these events takes place.

If an edge becomes fully colored, it results in the merging of two sets into one in FC. As a result, two sets are removed and one set is added at Line 15, leading to a decrease in the size of FC. Alternatively, if an active set is deactivated, it is removed from $ActS$ at Line 19, which leads to a decrease in the size of $ActS$. It is important to note that the number of active sets in $ActS$ does not increase at Line 16 (it either decreases by one or remains the same). From this, we can conclude that after each iteration of the while loop, either the number of active sets in $ActS$ decreases by at least one, or the number of sets in FC decreases by one, or both events occur. Since both $ActS$ and FC initially contain n elements, the while loop can iterate for a maximum of $2n$ times.

In each iteration, we perform the following operations with polynomial runtime: FindDeltaE, which is polynomial due to Lemma 6; FindDeltaP, which is polynomial according to Corollary 19; iterating through active sets to extend the *static coloring*, which is polynomial based on the size of $ActS$; iterating through edges to update active sets if they fully color edges, which is polynomial; and checking if each active set is tight using CheckSetIsTight, which is polynomial according to Lemma 21.

In the end, we also run ReduceTightPairs, which is polynomial according to Corollary 23.

Therefore, we can conclude that PCSF3 runs in polynomial time. □

3 The Iterative Algorithm

In this section, we present our iterative algorithm which uses the PCSF3 procedure from Algorithm 2 as a building block. We then provide a proof of its 2-approximation guarantee in Section 3.1. Finally, in Section 3.2, we provide a brief overview of a more refined analysis to establish a $(2 - \frac{1}{n})$-approximation for an n vertex input graph.

Our algorithm, described in Algorithm 6, considers two solutions for the given PCSF instance I. The first solution, denoted as (Q_1, F'_1), is obtained by invoking the PCSF3 procedure (Line 2). If the total penalty of this solution, $\pi(Q_1)$, is equal to 0, the algorithm returns it immediately as the solution.

Otherwise, a second solution, denoted as (Q_2, F'_2), is obtained through a recursive call on a simplified instance R. The simplified instance is created by adjusting penalties: penalties are limited to pairs that Algorithm 2 does not pay, and the penalties for other pairs are set to 0 (Lines 6-12). Essentially, we assume that pairs whose penalties are paid in the first solution will indeed be paid, and our objective is to find a solution for the remaining pair connection demands. We note that setting the penalties for these pairs to 0 guarantees their inclusion in Q_2. This is because Q_2 represents the set of tight pairs for a subsequent invocation of PCSF3, and any pair with a penalty of 0 is trivially tight.

To compare the two solutions, the algorithm computes the values $cost_1 = c(F'_1) + \pi(Q_1)$ and $cost_2 = c(F'_2) + \pi(Q_2)$, which represent the costs of the solutions (Lines 5 and 14). In the final step, the algorithm simply selects and returns the solution with the lower cost.

Algorithm 6 Iterative PCSF algorithm

Input: An undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$ and penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$.
Output: A set of pairs Q with a forest F' that connects the endpoints of every pair $(i, j) \notin Q$.

1: **procedure** IPCSF($I = (G,\ \pi)$)
2: $(Q_1, F'_1) \leftarrow$ PCSF3(I)
3: **if** $\pi(Q_1) = 0$ **then**
4: **return** (Q_1, F'_1)
5: $cost_1 \leftarrow c(F'_1) + \pi(Q_1)$
6: Initialize π' as a new all-zero penalty vector
7: **for** $(i, j) \in V \times V$ **do**
8: **if** $(i, j) \in Q_1$ **then**
9: $\pi'_{ij} \leftarrow 0$
10: **else**
11: $\pi'_{ij} \leftarrow \pi_{ij}$
12: Construct instance R of the PCSF problem consisting of G and π'
13: $(Q_2, F'_2) \leftarrow$ IPCSF(R)
14: $cost_2 \leftarrow c(F'_2) + \pi(Q_2)$
15: **if** $cost_1 \leq cost_2$ **then**
16: **return** (Q_1, F'_1)
17: **else**
18: **return** (Q_2, F'_2)

3.1 Analysis

We now analyze the approximation guarantee of Algorithm 6. In the following, we consider an arbitrary instance $I = (G, \pi)$ of the PCSF problem, and analyze the solutions found by the IPCSF algorithm. In our analysis, we focus on **the first call** of IPCSF. By the output of PCSF3, we refer to the result of the first call of PCSF3 on instance I at Line 2. Similarly, when we mention the output of the recursive call, we are referring to the output of IPCSF on instance R at Line 13. We compare the output of IPCSF on I, which is the minimum of the output of PCSF3 and the output of the recursive call, with an optimal solution OPT of the instance I. We denote the forest selected in OPT as F^* and use Q^* to refer to the set of pairs not connected in F^*, for which OPT pays the penalties. Then, the cost of OPT is given by $cost(OPT) = c(F^*) + \pi(Q^*)$.

In the following, when we refer to coloring, we specifically mean the coloring performed in the first call of PCSF3 on instance I. In particular, when we mention *dynamic coloring*, we are referring to the final *dynamic coloring* of the first call of PCSF3 on instance I. The values y_S, y_{Sij}, and y_{ij} used in the analysis all refer to the corresponding values in the final *static coloring* and *dynamic coloring*.

Definition 32. For an instance I, we define four sets to categorize the pairs based on their connectivity in both the optimal solution OPT of I and the result of PCSF3(I), denoted as (Q_1, F'_1):

- Set $\mathcal{CC}$ contains pairs (i, j) that are connected in the optimal solution and are not in the set Q_1 returned by PCSF3.
- Set $\mathcal{CP}$ contains pairs (i, j) that are connected in the optimal solution and are in the set Q_1 returned by PCSF3.
- Set $\mathcal{PC}$ contains pairs (i, j) that are not connected in the optimal solution and are not in the set Q_1 returned by PCSF3.

- Set $\mathcal{PP}$ contains pairs (i, j) that are not connected in the optimal solution and are in the set Q_1 returned by PCSF3.

Based on the final *dynamic coloring* of PCSF3(I), we define the following values to represent the total duration of coloring with pairs in these sets.

$$cc = \sum_{(i,j)\in \mathcal{CC}} y_{ij}, \qquad cp = \sum_{(i,j)\in \mathcal{CP}} y_{ij}$$
$$pc = \sum_{(i,j)\in \mathcal{PC}} y_{ij}, \qquad pp = \sum_{(i,j)\in \mathcal{PP}} y_{ij}$$

The following table illustrates the connectivity status of pairs in each set:

		PCSF3	
		Connect	Penalty
Optimal Solution	Connect	$\mathcal{CC}$	$\mathcal{CP}$
	Penalty	$\mathcal{PC}$	$\mathcal{PP}$

So far, we have classified pairs into four categories. Now, we categorize the coloring moments involving pairs in set $\mathcal{CP}$ into two types: those that color exactly one edge of the optimal solution, and those that color at least two edges of the optimal solution. Since pairs in $\mathcal{CP}$ are connected in the optimal solution, they are guaranteed to color at least one edge of the optimal solution during their coloring moments. Furthermore, we allocate the value of cp between cp_1 and cp_2 based on this categorization.

Definition 33 (Single-edge and multi-edge sets)**.** For an instance I, we define a set $S \subset V$ as a single-edge set if it cuts exactly one edge of OPT, i.e., $d_{F^*}(S) = 1$, and as a multi-edge set if it cuts at least two edges of OPT, i.e., $d_{F^*}(S) > 1$. Let cp_1 represent the duration of coloring with pairs in $\mathcal{CP}$ in *dynamic coloring* that corresponds to coloring with single-edge sets in *static coloring*. Similarly, let cp_2 represent the duration of coloring with pairs in $\mathcal{CP}$ in *dynamic coloring* that corresponds to coloring with multi-edge sets in *static coloring*. These values are formally defined as follows:

$$cp_1 = \sum_{(i,j)\in \mathcal{CP}} \sum_{\substack{S : S \odot (i,j), \\ d_{F^*}(S)=1}} y_{Sij}$$
$$cp_2 = \sum_{(i,j)\in \mathcal{CP}} \sum_{\substack{S : S \odot (i,j), \\ d_{F^*}(S)>1}} y_{Sij}.$$

Figure 4 displays a single-edge set on the left and a multi-edge set on the right.

Lemma 34. For an instance I, we have $cp_1 + cp_2 = cp$.

Proof. Since pairs in $\mathcal{CP}$ are connected by the optimal solution OPT, any set S cutting a pair in $\mathcal{CP}$ must cut at least one edge of OPT. Therefore, S is either a single-edge set or a multi-edge set. Hence, we have $cp_1 + cp_2 = cp$. □

Now, we use these definitions and categorizations to analyze our algorithm. All of the following lemmas are based on the assumption that IPCSF is executed on instance I. First, in Lemma 35, we provide a lower

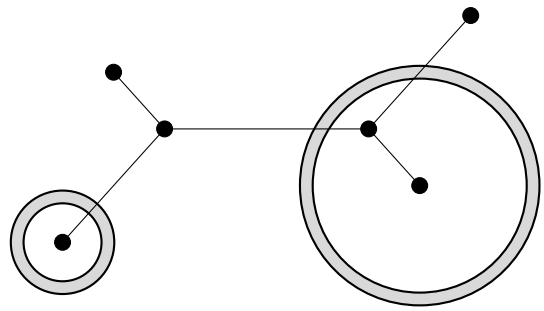

Figure 4: A comparison between a single-cut set (left) and a multi-cut set (right).

bound on the cost of the optimal solution, which is $cost(OPT) \geq cc + cp + cp_2 + pc + pp$. Next, in Lemma 37, we present an upper bound on the output of PCSF3(I), which is $cost_1 \leq 2cc + 2pc + 3cp + 3pp$. Moreover, in Lemma 38, we show that this value is at most $2cost(OPT) + cp_1 - cp_2 + pp$.

Next, we want to bound the output of the recursive call within IPCSF. In Lemma 40, we initially proof that $cost(OPT_R) \leq cost(OPT) - pp - cp_1$, where $cost(OPT_R)$ represents the cost of the optimal solution for the instance R defined at Line 12. Finally, in Theorem 41, we employ induction to demonstrate that $cost(\text{IPCSF}) \leq 2cost(OPT)$. Here, $cost(\text{IPCSF})$ denotes the cost of the output produced by IPCSF on instance I. To accomplish this, we use the same induction to bound the cost of the solution obtained through the recursive call at Line 14 by $cost_2 \leq 2cost(OPT_R) + cp + pp$, and by utilizing Lemma 40, we can then conclude that $cost_2 \leq 2cost(OPT) - cp_1 + cp_2 - pp$. Taking the average of $cost_1$ and $cost_2$ results in a value that is at most $2cost(OPT)$. Consequently, the minimum of these two values, corresponding to the cost of IPCSF(I), is at most $2cost(OPT)$.

Lemma 35. For an instance I, We can derive a lower bound for the cost of the optimal solution OPT as follows:

$$cost(OPT) \geq cc + cp + cp_2 + pc + pp.$$

Proof. The optimal solution pays penalties for pairs with labels $\mathcal{PC}$ and $\mathcal{PP}$ as it does not connect them. Since $y_{ij} \leq \pi_{ij}$ for each pair (i, j), we can lower bound the penalty paid by OPT as

$$\pi(Q^*) \geq \sum_{(i,j)\in(\mathcal{PC}\cup\mathcal{PP})} \pi_{ij} \geq \sum_{(i,j)\in(\mathcal{PC}\cup\mathcal{PP})} y_{ij} = pc + pp.$$

Now, we want to bound the cost of the forest in the optimal solution by $cc + cp + cp_2$. First, it is important to note that each part of an edge will be colored at most once. During the execution of the *static coloring*, each active set S colors the uncolored parts of all its cutting edges. Therefore, when S is an active set, it colors parts of exactly $d_{F^*}(S)$ edges of the optimal solution. Based on this observation, we can bound the total cost of the edges in the optimal solution by considering the amount of coloring applied to these edges.

$$\begin{aligned}
c(F^*) &\geq \sum_{S\subset V} d_{F^*}(S)\cdot y_S \\
&= \sum_{S\subset V}\sum_{(i,j):S\odot(i,j)} d_{F^*}(S)\cdot y_{Sij} && (y_S = \textstyle\sum_{(i,j):S\odot(i,j)} y_{Sij}) \\
&= \sum_{(i,j)\in V\times V}\sum_{S:S\odot(i,j)} d_{F^*}(S)\cdot y_{Sij} && \text{(change the order of summations)} \\
&\geq \sum_{(i,j)\in\mathcal{CC}}\sum_{S\odot(i,j)} d_{F^*}(S)\cdot y_{Sij} + \sum_{(i,j)\in\mathcal{CP}}\sum_{S\odot(i,j)} d_{F^*}(S)\cdot y_{Sij}. && (\mathcal{CC}\cap\mathcal{CP}=\emptyset)
\end{aligned}$$

For each pair $(i, j) \in (CC \cup C\mathcal{P})$, we know that in the optimal solution OPT, the endpoints of (i, j) are connected. This implies that for every set S satisfying $S \odot (i, j)$, the set S cuts the forest of OPT, i.e., $d_{F^*}(S) \geq 1$. Based on this observation, we bound the two terms in the summation above separately. For pairs in CC, we have

$$\sum_{(i,j)\in CC} \sum_{S\odot(i,j)} d_{F^*}(S) \cdot y_{Sij} \geq \sum_{(i,j)\in CC} \sum_{S\odot(i,j)} y_{Sij} = \sum_{(i,j)\in CC} y_{ij} = cc.$$

For pairs in $C\mathcal{P}$, we have

$$\begin{aligned}
\sum_{(i,j)\in C\mathcal{P}} \sum_{S\odot(i,j)} d_{F^*}(S) \cdot y_{Sij} &= \sum_{(i,j)\in C\mathcal{P}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)=1}} d_{F^*}(S) \cdot y_{Sij} + \sum_{(i,j)\in C\mathcal{P}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)>1}} d_{F^*}(S) \cdot y_{Sij} \\
&\geq \sum_{(i,j)\in C\mathcal{P}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)=1}} y_{Sij} + \sum_{(i,j)\in C\mathcal{P}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)>1}} 2y_{Sij} \\
&= cp_1 + 2cp_2 \\
&= cp + cp_2. && \text{(Lemma 34)}
\end{aligned}$$

Summing up all the components, we have:

$$cost(OPT) = c(F^*) + \pi(Q^*) \geq cc + cp + cp_2 + pc + pp$$

□

Lemma 36. Let F be an arbitrary forest and S be a subset of vertices in F. If S cuts only one edge e in F, then removing this edge will only disconnect pairs of vertices cut by S.

Proof. Consider a pair (i, j) that is disconnected by removing e. This pair must be connected in forest F, so there is a unique simple path between i and j in F. This path must include edge e, as otherwise, the pair would remain connected after removing e. Let the endpoints of e be u and v, where $u \in S$ and $v \notin S$. Without loss of generality, assume that i is the endpoint of the path that is closer to u than v. Then i is connected to u through the edges in the path other than e. As these edges are not cut by S and $u \in S$, it follows that i must also be in S. Similarly, it can be shown that j is not in S. Therefore, S cuts the pair (i, j). □

Lemma 37. For an instance I, during the first iteration of IPCSF(I) where PCSF3(I) is invoked, we can establish an upper bound on the output of PCSF3 as follows:

$$cost_1 \leq 2cc + 2pc + 3cp + 3pp.$$

Proof. Since $cost_1$ is the total cost of PCSF3(I), we should bound $\pi(Q_1) + c(F'_1)$. First, let's observe that PCSF3 pays the penalty for exactly the pairs (i, j) in $C\mathcal{P} \cup \mathcal{P}\mathcal{P}$, where $C\mathcal{P} \cup \mathcal{P}\mathcal{P} = Q_1$. Since every pair in Q_1 is tight, we have $\pi_{ij} = y_{ij}$ for these pairs. Therefore, the total penalty paid by PCSF3 can be bounded by

$$\pi(Q_1) = \sum_{(i,j)\in(C\mathcal{P}\cup\mathcal{P}\mathcal{P})} \pi_{ij} = \sum_{(i,j)\in(C\mathcal{P}\cup\mathcal{P}\mathcal{P})} y_{ij} = cp + pp.$$

Now, it suffices to show that $c(F'_1) \leq 2(cc + cp + pc + pp)$. We can prove this similarly to the proof presented by Goemmans and Williamson in [14]. Since each pair belongs to exactly one of the sets CC, $C\mathcal{P}$, $\mathcal{P}C$, and $\mathcal{P}\mathcal{P}$, we can observe that

$$cc + cp + pc + pp = \sum_{(i,j)\in V\times V} y_{ij} = \sum_{S\subset V} y_S.$$

Therefore, our goal is to prove that the cost of F_1' is at most $2\sum_{S\subset V} y_S$ using properties of *static coloring*. To achieve this, we show that the portion of edges in F_1' colored during each step of PCSF3 is at most twice the total increase in the y_S values during that step. Since every edge in the forest F_1' is fully colored by PCSF3, this will establish the desired inequality.

Now, let's consider a specific step of the procedure PCSF3 where the y_S values of the active sets in $ActS$ are increased by Δ. In this step, the total proportion of edges in F_1' that are colored by an active set S is $\Delta d_{F_1'}(S)$. Therefore, we want to prove that

$$\Delta \sum_{S\in ActS} d_{F_1'}(S) \leq 2\Delta \cdot |ActS|,$$

where the left-hand side represents the length of coloring on the edges of F_1' in this step, while the right-hand side represents twice the total increase in y_S values.

Consider the graph H formed from F_1' by contracting each connected component in FC at this step in the algorithm. As the edges of forest F at this step and F_1' are a subset of the forest F at the end of PCSF3, the graph H should be a forest. If H contains a cycle, it contradicts the fact that F at the end of PCSF3 is a forest.

In forest H, each vertex represents a set $S \in FC$, and the neighboring edges of this vertex are exactly the edges in $\delta(S) \cap F_1'$. We refer to the vertices representing active sets as active vertices, and the vertices representing inactive sets as inactive vertices. To simplify the analysis, we remove any isolated inactive vertices from H.

Now, let's focus on the inactive vertices in H. Each inactive vertex must have a degree of at least 2 in H. Otherwise, if an inactive vertex v has a degree of 1, consider the only edge in H connected to this vertex. For this edge not to be removed in the final step of Algorithm 2 at Line 21, there must exist a pair outside of Q_1 that would be disconnected after deleting this edge. However, since vertex v is inactive, its corresponding set S becomes tight before this step. According to Lemma 28, S will remain tight afterward. As a result, by Lemma 12, any pair cut by S will also be tight in the final coloring and will be included in Q_1. By applying Lemma 36, we can conclude that the only pairs disconnected by removing this edge would be the pairs cut by S, which we have shown to be in Q_1. Therefore, an inactive vertex cannot have a degree of 1, and all inactive vertices in H have a degree of at least 2. Let V_a and V_i represent the sets of active and inactive vertices in H, respectively. We have

$$\begin{aligned}
\sum_{S\in ActS} d_{F_1'}(S) &= \sum_{v\in V_a} d_H(v) \\
&= \sum_{v\in V_a\cup V_i} d_H(v) - \sum_{v\in V_i} d_H(v) \\
&\leq 2(|V_a| + |V_i|) - \sum_{v\in V_i} d_H(v) && (H \text{ is a forest}) \\
&\leq 2(|V_a| + |V_i|) - 2|V_i| && (d_H(v) \geq 2 \text{ for } v \in V_i) \\
&\leq 2(|V_a|) = 2|ActS|.
\end{aligned}$$

This completes the proof. □

Lemma 38. For an instance I, during the first iteration of IPCSF(I) where PCSF3(I) is invoked, we can establish an upper bound on the output of PCSF3 as follows:

$$cost_1 \leq 2cost(OPT) + cp_1 - cp_2 + pp$$

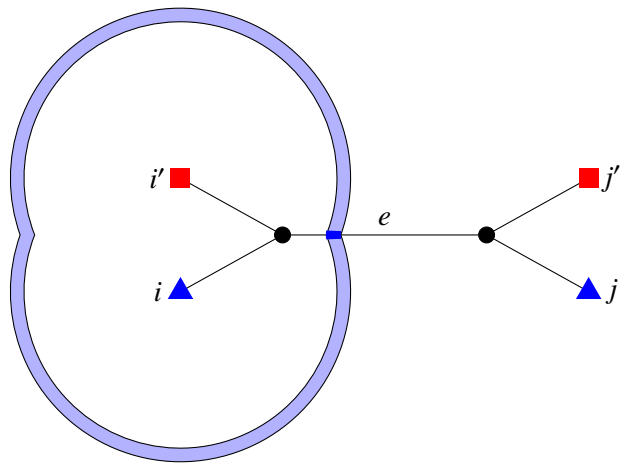

Figure 5: The figure shows the graph of F^* with pairs (i, j) and (i', j'), and a single-edge set colored with pair (i, j) in *dynamic coloring*. Tightness of (i, j) implies tightness of (i', j'), and removing edge e does not disconnect pairs in CC.

Proof. We can readily prove this by referring to the previous lemmas.

$$\begin{aligned} cost_1 \leq & 2cc + 2pc + 3cp + 3pp && \text{(Lemma 37)} \\ = & 2(cc + cp + cp_2 + pc + pp) + cp - 2cp_2 + pp \\ \leq & 2cost(OPT) + (cp - cp_2) - cp_2 + pp && \text{(Lemma 35)} \\ = & 2cost(OPT) + cp_1 - cp_2 + pp. && \text{(Lemma 34)} \end{aligned}$$

□

Lemma 39. For an instance I, it is possible to remove a set of edges from F^* with a total cost of at least cp_1 while ensuring that the pairs in CC remain connected.

Proof. Consider a single-edge set S that cuts some pair (i, j) in $\mathcal{CP}$ with $y_{Sij} > 0$. Since (i, j) is in $\mathcal{CP}$, it is also in Q_1 and therefore tight. By Lemma 25, any other pair cut by S will also be tight. Consequently, the pairs in CC will not be cut by S since they are not tight. Furthermore, according to Lemma 36, if S cuts only one edge e of F^*, then the only pairs that will be disconnected by removing edge e from F^* are the pairs that are cut by S. However, we have already shown that no pair in CC is cut by S. Therefore, all pairs in CC will remain connected even after removing edge e. See Figure 5 for an illustration.

For any single-edge set S that cuts a pair (i, j) in $\mathcal{CP}$ with $y_{Sij} > 0$, we can safely remove the single edge of F^* that is cut by S. The total amount of coloring on these removed edges is at least

$$\sum_{S : d_{F^*}(S)=1} \sum_{\substack{(i,j) \in \mathcal{CP} \\ S \odot (i,j) \\ y_{Sij} > 0}} y_{Sij} = \sum_{S : d_{F^*}(S)=1} \sum_{\substack{(i,j) \in \mathcal{CP} \\ S \odot (i,j)}} y_{Sij} = cp_1.$$

As the color on each edge does not exceed its length, the total length of the removed edges will also be at least cp_1. □

Now, we introduce some useful notation to analyze the output of the recursive call. During the execution of IPCSF on an instance I, it generates a modified instance R at Line 12, where the penalties for pairs in Q_1 are set to 0. We use the notation π' to represent the penalties in the instance R as they are defined in Lines 6-11. Since Line 3 ensures that $\pi(Q_1) \neq 0$, we can conclude that R is a reduced instance compared to I, meaning that the number of pairs with non-zero penalties is smaller in R than in I. Given that we recursively call IPCSF on instance R, we can bound the output of the recursive call by the optimal solution of R using induction. Let OPT_R be an optimal solution for R. We denote the forest of OPT_R as F^*_R and the set of pairs not connected by F^*_R as Q^*_R. The cost of OPT_R is given by $cost(OPT_R) = c(F^*_R) + \pi'(Q^*_R)$. We will use these notations in the following lemmas.

Lemma 40. For an instance I and the instance R constructed at Line 12 during the execution of IPCSF(I), we have

$$cost(OPT_R) \leq cost(OPT) - pp - cp_1.$$

Proof. To prove this lemma, we first provide a solution for the instance R given the optimal solution of the instance I, denoted as OPT, and we show that the cost of this solution is at most $cost(OPT) - pp - cp_1$. Since OPT_R is a solution for the instance R with the minimum cost, we can conclude that $cost(OPT_R) \leq cost(OPT) - pp - cp_1$.

To provide the aforementioned solution for the instance R, we start with the solution OPT consisting of the forest F^* and the set of pairs for which penalties were paid, denoted as Q^*. We create a new set $Q'_R = Q^* \cup \mathcal{CP} = \mathcal{PC} \cup \mathcal{PP} \cup \mathcal{CP}$ and a forest F'_R initially equal to F^*. Since F^* connects pairs in $\mathcal{CC}$ and $\mathcal{CP}$, but we add pairs in $\mathcal{CP}$ to Q'_R and pay their penalties, we can remove edges from F'_R that do not connect pairs in $\mathcal{CC}$.

Let's focus on Q'_R first. Since the penalties for pairs in $\mathcal{CP}$ and $\mathcal{PP}$ are set to 0 in π', we have

$$\begin{aligned}
\pi'(Q'_R) &= \pi'(\mathcal{CP}) + \pi'(\mathcal{PC}) + \pi'(\mathcal{PP}) && (Q'_R = \mathcal{CP} \cup \mathcal{PC} \cup \mathcal{PP}) \\
&= \pi'(\mathcal{PC}) && (\pi'(\mathcal{CP}) = \pi'(\mathcal{PP}) = 0) \\
&= \pi(\mathcal{PC}) \\
&= \pi(Q^*) - \pi(\mathcal{PP}) && (Q^* = \mathcal{PC} \cup \mathcal{PP}) \\
&= \pi(Q^*) - \sum_{(i,j)\in\mathcal{PP}} \pi_{ij} \\
&= \pi(Q^*) - \sum_{(i,j)\in\mathcal{PP}} y_{ij} && (\text{pairs in } \mathcal{PP} \text{ are tight}) \\
&= \pi(Q^*) - pp.
\end{aligned}$$

Moreover, using Lemma 39, we construct F'_R from F^* by removing a set of edges with a total length of at least cp_1, while ensuring that the remaining forest still connects all the pairs in $\mathcal{CC}$. Therefore, we can bound the cost of F'_R as

$$c(F'_R) \leq c(F^*) - cp_1.$$

Summing it all together, we have

$$cost(OPT_R) \leq c(F'_R) + \pi'(Q'_R) \leq (c(F^*) - cp_1) + (\pi(Q^*) - pp) = cost(OPT) - pp - cp_1,$$

where the first inequality comes from the fact that OPT_R is the optimal solution for the instance R, while (Q'_R, F'_R) gives a valid solution, i.e., F'_R connects every pair that is not in Q'_R. □

Finally, we can bound the cost of the output of IPCSF. For an instance I, let's denote the cost of the output of IPCSF(I) as $cost$(IPCSF). In Theorem 41, we prove that the output of IPCSF is a 2-approximate solution for the PCSF problem.

Theorem 41. For an instance I, the output of IPCSF(I) is a 2-approximate solution to the optimal solution for I, meaning that

$$cost(\text{IPCSF}) \leq 2cost(OPT).$$

Proof. We will prove the claim by induction on the number of pairs (i, j) with penalty $\pi_{ij} > 0$ in instance I.

First, the algorithm makes a call to the PCSF3 procedure to obtain a solution (Q_1, F'_1). If $\pi(Q_1) = 0$ for this solution, which means no cost is incurred by paying penalties, the algorithm terminates and returns this solution at Line 4. This will always be the case in the base case of our induction where for all pairs $(i, j) \in Q_1$, penalties π_{ij} are equal to 0. Since every pair $(i, j) \in Q_1$ is tight, we have $y_{ij} = \pi_{ij} = 0$. Given that $\mathcal{CP}$ and $\mathcal{PP}$ are subsets of Q_1, we can conclude that $cp = cp_1 = cp_2 = pp = 0$. Now, by Lemma 38, we have

$$cost_1 \leq 2cost(OPT) + (cp_1 - cp_2) + pp = 2cost(OPT).$$

Therefore, when IPCSF returns at Line 4, we have

$$cost(\text{IPCSF}) = cost_1 \leq 2cost(OPT),$$

and we obtain a 2-approximation of the optimal solution.

Now, let's assume that PCSF3 pays penalties for some pairs, i.e., $\pi(Q_1) \neq 0$. Therefore, since we set the penalty of pairs in Q_1 equal to 0 for instance R at Line 9, the number of pairs with non-zero penalty in instance R is less than in instance I. By induction, we know that the output of IPCSF on instance R, denoted as (Q_2, F_2), has a cost of at most $2cost(OPT_R)$. That means

$$c(F_2) + \pi'(Q_2) \leq 2cost(OPT_R).$$

In addition, we have

$$\pi(Q_2) = \pi(Q_2 \setminus Q_1) + \pi(Q_2 \cap Q_1) \leq \pi'(Q_2 \setminus Q_1) + \pi(Q_1) \leq \pi'(Q_2) + \pi(Q_1),$$

where we use the fact that $\pi'_{ij} = \pi_{ij}$ for $(i, j) \notin Q_1$. Now we can bound the cost of the solution (Q_2, F'_2), denoted as $cost_2$, by

$$\begin{aligned}
cost_2 &= c(F'_2) + \pi(Q_2) \\
&\leq c(F'_2) + \pi'(Q_2) + \pi(Q_1) \\
&\leq 2cost(OPT_R) + \pi(Q_1) && \text{(By induction)} \\
&\leq 2\left(cost(OPT) - pp - cp_1\right) + \sum_{(i,j)\in Q_1} \pi_{ij} && \text{(Lemma 40)} \\
&= 2\left(cost(OPT) - pp - cp_1\right) + \sum_{(i,j)\in Q_1} y_{ij} && \text{(pairs in } Q_1 \text{ are tight)} \\
&= 2cost(OPT) - 2pp - 2cp_1 + cp + pp \\
&= 2cost(OPT) - cp_1 + cp_2 - pp. && \text{(Lemma 34)}
\end{aligned}$$

Furthermore, according to Lemma 38, the cost of the solution (Q_1, F'_1), denoted as $cost_1$, can be bounded by

$$cost_1 \leq 2OPT + cp_1 - cp_2 + pp.$$

Finally, in Line 15, we return the solution with the smaller cost between (Q_1, F'_1) and (Q_2, F'_2). Based on the upper bounds above on both solutions, we know that

$$\begin{aligned} cost(\text{IPCSF}) &= \min(cost_1, cost_2) \leq \frac{1}{2}(cost_1 + cost_2) \\ &\leq \frac{1}{2}(2cost(OPT) + cp_1 - cp_2 + pp + 2cost(OPT) - cp_1 + cp_2 - pp) \\ &= \frac{1}{2}(4cost(OPT)) = 2cost(OPT), \end{aligned}$$

and we obtain a 2-approximation of the optimal solution. This completes the induction step and the proof of the theorem. □

Theorem 42. The runtime of the IPCSF algorithm is polynomial.

Proof. Let n be the number of vertices in the input graph. There are $O(n^2)$ pairs of vertices in total. Whenever IPCSF calls itself recursively, the number of pairs with non-zero penalties decreases by at least one, otherwise IPCSF will return at Line 4. Thus, the recursion depth is polynomial in n. At each recursion level, the algorithm only runs PCSF3 on one instance of the problem and performs $O(n^2)$ additional operations. By Lemma 31, we know that PCSF3 runs in polynomial time. Therefore, the total run-time of IPCSF will also be polynomial. □

3.2 Improving the approximation ratio

In this section, we briefly explain how a tighter analysis can be used to show that the approximation ratio of the IPCSF algorithm is at most $2 - \frac{1}{n}$, where n is the number of vertices in the input graph G. This approximation ratio more closely matches the approximation ratio of $2 - \frac{2}{n}$ for the Steiner Forest problem.

We first introduce an improved version of Lemmas 37 and 38.

Lemma 43. For an instance I, during the first iteration of IPCSF(I) where PCSF3(I) is invoked, we have the following upper bound

$$cost_1 \leq (2 - \frac{2}{n}) \cdot cc + (2 - \frac{2}{n}) \cdot pc + (3 - \frac{2}{n}) \cdot cp + (3 - \frac{2}{n}) \cdot pp.$$

Proof. We proceed similarly to the proof of Lemma 37 and make a slight change. In one of the last steps of that proof, we use the following inequality:

$$\sum_{v \in V_a \cup V_i} d_H(v) - \sum_{v \in V_i} d_H(v) \leq 2(|V_a| + |V_i|) - \sum_{v \in V_i} d_H(v).$$

This is true, as H is a forest and its number of edges is less than its number of vertices. However, as the number of edges in a forest is strictly less than the number of vertices, we can lower the right-hand side of this inequality to $2(|V_a| + |V_i| - 1) - \sum_{v \in V_i} d_H(v)$. Rewriting the main inequality in this step with this change

gives us

$$\begin{aligned}
\sum_{S \in ActS} d_{F_1'}(S) &\leq 2(|V_a| + |V_i| - 1) - \sum_{v \in V_i} d_H(v) \\
&\leq 2(|V_a| + |V_i| - 1) - 2|V_i| && (d_H(v) \geq 2 \text{ for } v \in V_i) \\
&\leq 2(|V_a| - 1) = 2|ActS| - 2 && (|V_a| = |ActS|) \\
&= (2 - \frac{2}{|ActS|})|ActS| \\
&\leq (2 - \frac{2}{n})|ActS|. && (|ActS| \leq n)
\end{aligned}$$

Based on the steps in the proof of Lemma 37, this leads to the desired upper bound. □

Lemma 44. For an instance I, during the first iteration of IPCSF(I) where PCSF3(I) is invoked, we can establish an upper bound on the output of PCSF3 as follows:

$$cost_1 \leq (2 - \frac{2}{n}) \cdot cost(OPT) + cp_1 - (1 - \frac{2}{n}) \cdot cp_2 + pp$$

Proof. We prove this lemma similarly to Lemma 38, except we use Lemma 43 instead of Lemma 37.

$$\begin{aligned}
cost_1 \leq& (2 - \frac{2}{n}) \cdot cc + (2 - \frac{2}{n}) \cdot pc + (3 - \frac{2}{n}) \cdot cp + (3 - \frac{2}{n}) \cdot pp && \text{(Lemma 43)} \\
=& (2 - \frac{2}{n})(cc + cp + cp_2 + pc + pp) + cp - (2 - \frac{2}{n}) \cdot cp_2 + pp \\
\leq& (2 - \frac{2}{n}) \cdot cost(OPT) + (cp - cp_2) - (1 - \frac{2}{n})cp_2 + pp && \text{(Lemma 35)} \\
=& (2 - \frac{2}{n}) \cdot cost(OPT) + cp_1 - (1 - \frac{2}{n}) \cdot cp_2 + pp. && \text{(Lemma 34)}
\end{aligned}$$

□

Finally, we improve Theorem 41.

Theorem 45. For an instance I, the output of IPCSF(I) is a $(2 - \frac{1}{n})$-approximate solution to the optimal solution for I, meaning that

$$cost(\text{IPCSF}) \leq (2 - \frac{1}{n}) \cdot cost(OPT).$$

Proof. Similarly to the proof of Theorem 41, we use induction on the number of non-zero penalties. If the algorithm terminates on Line 4 then by Lemma 44 we have

$$cost_1 \leq (2 - \frac{2}{n}) \cdot cost(OPT) + cp_1 - (1 - \frac{2}{n}) \cdot cp_2 + pp = (2 - \frac{2}{n}) \cdot cost(OPT)$$

since cp_1, cp_2, and pp are all 0 in this case. As $2 - \frac{2}{n} \leq 2 - \frac{1}{n}$, the desired inequality holds in this case. This establishes our base case for the induction.

Using the same reasoning as the proof of Theorem 41, based on the induction we have

$$\begin{aligned}
cost_2 &\leq (2-\frac{1}{n})\cdot cost(OPT_R)+\pi(Q_1)\\
&= (2-\frac{1}{n})\cdot cost(OPT_R)+cp+pp\\
&\leq (2-\frac{1}{n})(cost(OPT)-cp_1-pp)+cp+pp && \text{(By Lemma 40)}\\
&\leq (2-\frac{1}{n})\cdot cost(OPT)-(1-\frac{1}{n})\cdot cp_1+cp_2-(1-\frac{1}{n})\cdot pp.
\end{aligned}$$

We can combine this with the following upper bound from Lemma 44

$$cost_1 \leq (2-\frac{2}{n})\cdot cost(OPT)+cp_1-(1-\frac{2}{n})\cdot cp_2+pp.$$

As the algorithm chooses the solution with the lower cost between $cost_1$ and $cost_2$, we have

$$\begin{aligned}
cost(\text{IPCSF}) = \min(cost_1, cost_2) &\leq \frac{1}{2}(cost_1+cost_2)\\
&\leq \frac{1}{2}\Bigg[(2-\frac{2}{n})\cdot cost(OPT)+cp_1-(1-\frac{2}{n})\cdot cp_2+pp\\
&+(2-\frac{1}{n})\cdot cost(OPT)-(1-\frac{1}{n})\cdot cp_1+cp_2-(1-\frac{1}{n})\cdot pp\Bigg]\\
&= \frac{1}{2}\left((4-\frac{3}{n})\cdot cost(OPT)+\frac{2}{n}cp_2+\frac{1}{n}cp_1+\frac{1}{n}pp\right)\\
&\leq \frac{1}{2}\left((4-\frac{2}{n})\cdot cost(OPT)+\frac{1}{n}[2cp_2+cp_1+pp-cost(OPT)]\right)\\
&\leq \frac{1}{2}(4-\frac{2}{n})\cdot cost(OPT) && (cost(OPT)\geq 2cp_2+cp_1+pp \text{ by Lemma 35})\\
&= (2-\frac{1}{n})\cdot cost(OPT).
\end{aligned}$$

Therefore, the algorithm obtains a $(2-\frac{1}{n})$-approximation of the optimal solution. □

4 Acknowledgements

The work is partially support by DARPA QuICC, NSF AF:Small #2218678, and NSF AF:Small #2114269

References

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized steiner problem on networks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 134–144, New York, NY, USA, 1991. Association for Computing Machinery.

[2] A. Agrawal, P. N. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized steiner problem on networks. *SIAM J. Comput.*, 24(3):440–456, 1995.

[3] A. Archer, M. Bateni, M. Hajiaghayi, and H. J. Karloff. Improved approximation algorithms for prize-collecting steiner tree and TSP. *SIAM J. Comput.*, 40(2):309–332, 2011.

[4] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.

[5] M. Bateni and M. Hajiaghayi. Euclidean prize-collecting steiner forest. *Algorithmica*, 62(3-4):906–929, 2012.

[6] M. Bateni, M. T. Hajiaghayi, and D. Marx. Approximation schemes for steiner forest on planar graphs and graphs of bounded treewidth. *J. ACM*, 58(5):21:1–21:37, 2011.

[7] M. W. Bern and P. E. Plassmann. The steiner problem with edge lengths 1 and 2. *Inf. Process. Lett.*, 32(4):171–176, 1989.

[8] D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. P. Williamson. A note on the prize collecting traveling salesman problem. *Math. Program.*, 59:413–420, 1993.

[9] J. Blauth and M. Nägele. An improved approximation guarantee for prize-collecting TSP. In B. Saha and R. A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1848–1861. ACM, 2023.

[10] J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità. An improved lp-based approximation for steiner tree. In L. J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 583–592. ACM, 2010.

[11] M. Chlebík and J. Chlebíková. The steiner tree problem on graphs: Inapproximability results. *Theor. Comput. Sci.*, 406(3):207–214, 2008.

[12] M. X. Goemans. Combining approximation algorithms for the prize-collecting TSP. *CoRR*, abs/0910.0553, 2009.

[13] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, page 307–316, USA, 1992. Society for Industrial and Applied Mathematics.

[14] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995.

[15] A. Gupta, J. Könemann, S. Leonardi, R. Ravi, and G. Schäfer. An efficient cost-sharing mechanism for the prize-collecting steiner forest problem. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1153–1162. SIAM, 2007.

[16] A. Gupta and A. Kumar. Greedy algorithms for steiner forest. In R. A. Servedio and R. Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 871–878. ACM, 2015.

[17] M. Hajiaghayi and A. A. Nasri. Prize-collecting steiner networks via iterative rounding. In A. López-Ortiz, editor, *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, volume 6034 of *Lecture Notes in Computer Science*, pages 515–526. Springer, 2010.

[18] M. T. Hajiaghayi and K. Jain. The prize-collecting generalized steiner tree problem via a new approach of primal-dual schema. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 631–640. ACM Press, 2006.

[19] M. T. Hajiaghayi, R. Khandekar, G. Kortsarz, and Z. Nutov. Prize-collecting steiner network problems. *ACM Trans. Algorithms*, 9(1):2:1–2:13, 2012.

[20] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Co., USA, 1996.

[21] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[22] M. Karpinski and A. Zelikovsky. New approximation algorithms for the steiner tree problems. *J. Comb. Optim.*, 1(1):47–65, 1997.

[23] J. Könemann, N. Olver, K. Pashkovich, R. Ravi, C. Swamy, and J. Vygen. On the integrality gap of the prize-collecting steiner forest LP. In K. Jansen, J. D. P. Rolim, D. Williamson, and S. S. Vempala, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, volume 81 of *LIPIcs*, pages 17:1–17:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[24] G. Robins and A. Zelikovsky. Tighter bounds for graph steiner tree approximation. *SIAM J. Discret. Math.*, 19(1):122–134, 2005.

[25] Y. Sharma, C. Swamy, and D. P. Williamson. Approximation algorithms for prize collecting forest problems with submodular penalty functions. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1275–1284. SIAM, 2007.

[26] A. Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9(5):463–470, 1993.