

Breaking a Long-Standing Barrier: 2- ϵ Approximation for Steiner Forest

Ali Ahmadi*
ahmadi@umd.edu

Iman Gholami*
igholami@umd.edu

MohammadTaghi Hajiaghayi*
hajiagha@umd.edu

Peyman Jabbarzade*
peymanj@umd.edu

Mohammad Mahdavi*
mahdavi@umd.edu

Abstract

The Steiner Forest problem, also known as the Generalized Steiner Tree problem, is a fundamental optimization problem on edge-weighted graphs where, given a set of vertex pairs, the goal is to select a minimum-cost subgraph such that each pair is connected. This problem generalizes the Steiner Tree problem, first introduced in 1811, for which the best known approximation factor is 1.39 [Byrka, Grandoni, Rothvoß, and Sanità, 2010] (Best Paper award, STOC 2010).

The celebrated work of [Agrawal, Klein, and Ravi, 1989] (30-Year Test-of-Time award, STOC 2023), along with refinements by [Goemans and Williamson, 1992] (SICOMP'95), established a 2-approximation for Steiner Forest over 35 years ago. Jain's (FOCS'98) pioneering iterative rounding techniques later extended these results to higher connectivity settings. Despite the long-standing importance of this problem, breaking the approximation factor of 2 has remained a major challenge, raising suspicions that achieving a better factor—similar to Vertex Cover—might indeed be hard. Notably, fundamental works, including those by Gupta and Kumar (STOC'15) and Groß et al. (ITCS'18), introduced 96- and 69-approximation algorithms, possibly with the hope of paving the way for a breakthrough in achieving a constant-factor approximation below 2 for the Steiner Forest problem.

In this paper, we break the approximation barrier of 2 by designing a novel deterministic algorithm that achieves a $2 - 10^{-11}$ approximation for this fundamental problem. As a key component of our approach, we also introduce a novel dual-based local search algorithm for the Steiner Tree problem with an approximation guarantee of 1.943, which is of independent interest.

*University of Maryland.

Contents

1	Introduction	1
2	Algorithms and Intuition	3
2.1	Overview of Techniques	6
3	Preliminaries	12
4	Monotonic Moat Growing Algorithms	12
4.1	Legacy Moat Growing: The 2-Approximation Algorithm	12
4.2	Generalize Moat Growing: Shadow Moat Growing Algorithm	14
4.3	Notation and Definitions for Moat Growing Algorithms	19
4.4	Legacy Execution	27
5	Local Search	32
5.1	Algorithm	32
5.2	Analysis of Local Search	36
5.3	Boosted Execution	40
6	Structural Results on Local Minima	49
6.1	Claw Property	49
6.2	Generalizing the Claw Property to Larger Sets	52
6.3	Approximation for Steiner Tree: Proof of Theorem 2	57
7	Extended Moat Growing	58
7.1	Algorithm	58
7.2	Properties and Definitions	60
7.3	Extended Execution	62
7.4	Extended-Boosted Execution	65
7.5	Cost Analysis of SOL_{XT}	74
8	Autarkic Pairs	87
8.1	Algorithm	88
8.2	Properties of Connected Components in \mathcal{B}_2	89
8.3	Cost Analysis of SOL_{AP}	102
9	Approximation for Steiner Forest: Proof of Theorem 1	108
10	Acknowledgements	112
A	Tight Examples and Counterexamples	115
A.1	The Strength of Local Search	115
A.2	A Lower Bound for the Local Search Algorithm	117
A.3	Necessity of Autarkic Pairs	120
A.4	A Lower Bound Worse than 2 for the Gluttonous Algorithm	121

1 Introduction

The Steiner Forest problem, also known as the the Generalized Steiner Tree problem, is a fundamental NP-hard problem in computer science. Given a weighted undirected graph and a set of vertex pairs (demands), the goal is to select a minimum-cost subset of edges that connects each pair. We can assume each vertex belongs to exactly one demand by duplicating vertices as needed, assigning each demand to a separate copy, and connecting copies with zero-cost edges. Additionally, vertices not in any demand can be treated as being paired with themselves. Formally, given a weighted undirected graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$ and an involution function $\text{PAIR} : V \rightarrow V$ indicating that each vertex v must be connected to PAIR_v , the objective is to find a minimum-cost subgraph that ensures v and PAIR_v are connected for all $v \in V$. In this work, we break the long-standing approximation barrier of 2 for Steiner Forest, marking a breakthrough after more than 35 years of efforts by the community.

The Steiner Forest problem generalizes the Steiner Tree problem, in which a set of terminals is given and the goal is to connect them using a tree of minimum total edge weight. In our model, we assume that one terminal is designated as the root, and we create a duplicate of it for each of the remaining terminals, pairing each duplicate with its corresponding terminal. All other vertices are paired with themselves.

The first studies on the Steiner Tree problem date back to 1811, and it was later recognized as one of Karp’s classic NP-complete problems [Kar72]. Moreover, the problem is MAX SNP-hard [BP89] and inapproximable within a factor of 96/95 unless $P = NP$ [CC08]. Compared to the Steiner Tree problem, variants of the Steiner Forest problem appear to be more challenging. Examples include the Prize-Collecting Steiner Forest¹ and the even more difficult k -Steiner Forest². This contrast highlights the inherent challenges in achieving better approximations for Steiner Forest, suggesting the possibility of a 2-approximation lower bound.

The Steiner Tree problem has been extensively studied in the context of approximation algorithms. A trivial 2-approximation algorithm was first proposed [KMB81]. Breaking this natural 2-approximation barrier became a major research focus, with Zelikovsky achieving an 11/6-approximation [Zel93], followed by Karpinski and Zelikovsky’s 1.65-approximation [KZ95]. Further refinements led to a 1.55-approximation by Robins and Zelikovsky [RZ05] and finally a 1.39-approximation by Byrka, Grandoni, Rothvoß, and Sanità [BGRS10]. While these improvements have been achieved for the Steiner Tree problem, determining whether a better-than-2 approximation exists for Steiner Forest remains one of the most prominent open problems in approximation algorithms (see e.g. [WS11]³).

The best-known approximation factor for Steiner Forest remains 2, achieved in a landmark paper by Agrawal, Klein, and Ravi [AKR91, AKR95] using the primal-dual method. This approach was later revisited as a general framework for constrained forest problems by Goemans and Williamson [GW92, GW95] and applied to other problems in this area, such as the Prize-Collecting Steiner Tree. Jain’s pioneering iterative rounding techniques [Jai98, Jai01] later extended these results to higher connectivity settings. Despite many attempts, no improvement over the 2-approximation factor has been achieved. The problem is so significant that researchers have published papers in top conferences with much worse approximation factors, hoping to pave the way for a breakthrough below 2 for the Steiner Forest problem. Notably, a greedy algorithm achieved

¹While the Prize-Collecting Steiner Tree has had a better-than-2 approximation for a long time [ABHK11, AGH⁺24], the Prize-Collecting Steiner Forest only recently achieved a 2-approximation [AGH⁺25] (JACM’25).

²The k -Steiner Tree problem has long been known to admit a 2-approximation [Gar05, BHL18], while the k -Steiner Forest problem is roughly as hard as the densest k -subgraph problem [HJ06], for which the best-known approximation is $O(n^{1/4})$ [BCC⁺10].

³See Problem 6 of Shmoys and Williamson [WS11], among their ten key open problems in approximation algorithms, with Problems 1 and 2 on TSP and Asymmetric TSP. Notably, some of these problems like TSP [KKG21] (Best Paper award, STOC 2021) and Asymmetric TSP [STV18] (Best Paper award, STOC 2018) have seen progress since the book’s 2011 publication.

a 96-approximation [GK15], and a local search algorithm attained a 69-approximation [GGK⁺18]. These works were published in the hope that tighter analyses or refinements might eventually help break the barrier of 2. For the greedy algorithm with a 96-approximation, we provide a counterexample in Appendix A.4, showing that its approximation factor is indeed worse than 2. Recently, even half-integral solutions to an LP relaxation for Steiner Forest have been studied [BGT24]. Needless to say, our approach takes a completely different path from these previous attempts to surpass the 2-approximation barrier.

In this paper, we present the first deterministic algorithm to break the long-standing approximation barrier of 2 for the Steiner Forest problem.

Theorem 1. There exists a deterministic polynomial-time algorithm that achieves an approximation factor of $2 - 10^{-11}$ for the Steiner Forest problem.

Along the way, we introduce a novel dual-based local search algorithm with a $1 + 2\frac{\sqrt{2}}{3} \approx 1.943$ approximation factor for the Steiner Tree problem, complemented by a lower bound of 1.5 on its approximation factor.

Theorem 2. There exists a deterministic polynomial-time algorithm that achieves a $(1 + 2\frac{\sqrt{2}}{3})$ -approximation for the Steiner Tree problem.

Other related works. A notable problem of similar caliber in combinatorial optimization to the Steiner Forest problem is the Traveling Salesman Problem (TSP). Recent advancements, such as the Best Paper at STOC 2021, which presented a randomized $(3/2 - \epsilon)$ -approximation algorithm for metric TSP [KKG21] (later derandomized in [KKG23]), highlight the progress in improving approximation algorithms for the classic TSP. While this algorithm does not break the integrality gap of the natural linear programming (LP) relaxation for TSP [KKG22], it represents a major step forward in surpassing a long-standing approximation barrier. In contrast, our approach for the Steiner Forest problem provides a deterministic algorithm that breaks the long-standing 2-approximation barrier by overcoming the integrality gap of its natural LP relaxation [AKR95, GW95].

Generalizations of the Steiner Tree and Steiner Forest problems have been extensively studied in the literature. One notable extension is the Prize-Collecting Steiner Tree problem, which initially admitted a 2-approximation algorithm [GW95], later improved to 1.96 [ABHK11], and more recently to a 1.79-approximation [AGH⁺24]. In contrast, the Prize-Collecting Steiner Forest problem saw a breakthrough only recently, with the first 2-approximation algorithm [AGH⁺25] (JACM'25), following numerous 3-approximation algorithms [HJ06, GKL⁺07, HN10]. See [BH12, HKKN12, SSW07] for additional work on special graph classes and generalizations of the Prize-Collecting Steiner Forest problem.

Similarly, the k -MST problem has undergone a long sequence of improvements [RSM⁺94, AABV95, BRV99, Gar96, AR98, AK06] to eventually achieve a 2-approximation algorithm [Gar05]. The k -Steiner Tree problem has been reduced to k -MST, benefiting from these advancements [BHL18]. However, the k -Steiner Forest problem has been shown to be roughly as hard as the densest k -subgraph problem [HJ06], for which the current best approximation factor is $O(n^{1/4})$ [BCC⁺10].

Finally, while no prior work has improved upon the 2-approximation factor for general Steiner Forest instances, better approximations have been achieved for special graph classes. For example, a PTAS is known for planar graphs and, more generally, for graphs with bounded genus [BHM11]. Additionally, the Euclidean Steiner Forest problem also admits a PTAS [BKM15].

2 Algorithms and Intuition

In this section, we present the intuition behind our algorithm, along with a more detailed explanation of its components and the techniques outlined in Section 2.1.

To begin, we introduce the concept of *moat growing*, originally defined by Jünger and Pulleyblank in 1991 [JP95]. We then describe the *Legacy Moat Growing* algorithm, which is equivalent to the primal-dual algorithms of [AKR95, GW95]. Moat growing algorithms maintain a forest, marking a subset of its connected components as *active sets* and expanding these sets uniformly over time. As the active sets grow, they *color* the uncolored portions of their adjacent edges. Once an edge is fully colored, it is added to the forest. Since the cost of the forest produced by a moat growing algorithm is upper bounded by twice the total growth of active sets, we introduce the notion of *assignments*, which distribute this growth among the vertices responsible for it. This allows us to bound the assigned values rather than the total growth, providing an upper bound on the solution cost. In this work, we leverage the versatility of moat growing and coloring. See Section 4 for further details on moat growing algorithms, particularly the `LEGACYMOATGROWING` procedure.

Starting with Legacy Moat Growing as the base algorithm, we introduce a novel *local search* method that iteratively attempts to increase the active duration of an active set. This operation, called a *boost action*, is applied only when increasing the growth of an active set significantly reduces the total growth across all active sets, thereby leading to a smaller upper bound on the solution cost. Figure 1 illustrates how a boost action can reduce the total growth of active sets, leading to a better solution. For a detailed explanation of the `LOCALSEARCH` procedure and its properties, see Section 5.

The local search terminates when no boost action can further reduce the total growth of active sets. At this point, we derive key structural properties of the resulting moat growing algorithm. Most importantly, for any subset of *actively connected* vertices—those that remain in active sets until reaching one another—their total assigned value, excluding the *maximum assigned value*, is at most a fraction (less than one) of the cost of any tree spanning them. We will define the maximum assigned value more formally later; for now, it can be viewed as roughly the time when these vertices become connected. See Section 6 for details of this property.

This property leads to a 1.943-approximation algorithm for the Steiner Tree problem, where all terminals are actively connected and the optimal solution spans them. While this bound may not be tight—the best lower bound we found is 1.5 (see Appendix A.2)—we did not pursue a tighter analysis, as the current guarantee already suffices for our main goal of improving the Steiner Forest approximation. The algorithm satisfying Theorem 2 is presented in Algorithm 1, with its proof provided in Section 6.3.

Algorithm 1 Steiner Tree Algorithm

Input: A graph $G = (V, E, c)$, with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, a demand function $\text{PAIR} : V \rightarrow V$, corresponds to a Steiner Tree instance, and algorithm parameter $0 < \beta < 1$.

Output: A tree T satisfying PAIR demands.

- 1: **procedure** `STEINERTREE`(G, PAIR, β)
 - 2: $F, t^- \leftarrow \text{LEGACYMOATGROWING}(G, \text{PAIR})$
 - 3: $\text{SOL}_{LS}, t^+ \leftarrow \text{LOCALSEARCH}(G, t^-, \beta)$
 - 4: **return** SOL_{LS}
-

Adapting our approach to the Steiner Forest problem introduces new challenges. Unlike in the Steiner Tree case, the vertices of each connected component in the optimal solution may not be actively connected in our algorithm, as some pairs can be satisfied earlier than others. This prevents us from directly applying the same bound developed for Steiner Tree. To address this, we introduce an *extension* step and rerun the local search

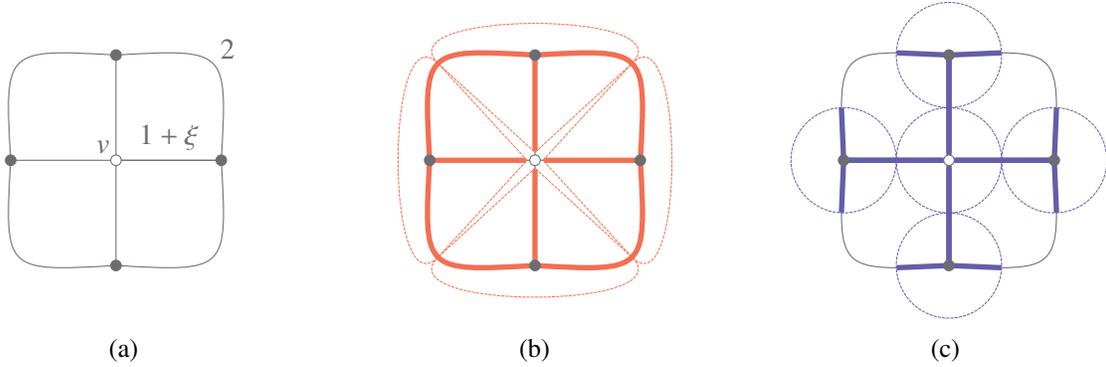


Figure 1: A Steiner Tree instance. (a) Initial configuration with four terminal vertices and a central vertex v , where $\xi > 0$ is sufficiently small. (b) After Legacy Moat Growing, moats centered on terminals expand, fully coloring the outer edges, resulting in a total growth of 4 and a solution cost of 8 . (c) A boost action on v connects moats earlier via central edges, reducing the total growth to $\frac{5}{2}(1 + \xi)$ and the solution cost to $4(1 + \xi)$.

algorithm afterward. In the extension step, each active set receives a potential equal to ϵ times its growth time, allowing it to continue expanding using this potential before deactivation. This ensures, at a small cost, that the vertices of each connected component in the optimal solution become actively connected, enabling us to apply the same bounding technique as in the Steiner Tree case. If some components remain not actively connected after the extension, we provide an alternative upper bound to complete the analysis. For details on the extension step, including the EXTEND procedure and its analysis, see Section 7.

Finally, as mentioned earlier, the bounding argument for Steiner Tree does not account for the maximum assigned value within each connected component of the optimal solution. This is not an issue if the connected components in our solution match those of the optimal solution. However, problems arise when they do not align, and the maximum assigned value for a component becomes disproportionately large compared to the total assigned value of its vertices. In such cases, vertices in those components often split into two groups that form quickly but take a long time to connect. This leads to a large maximum assigned value, meaning these vertices remain active and continue growing for a long time relative to the cost of their tree in the optimal solution. As a result, many edges are colored—edges that may later be used by pairs in other components—leading to inefficiencies. See Figure 2b for an example illustrating how these large values create challenges, preventing our local search and extension methods from achieving a better-than-2 approximation.

To address this, we introduce a novel technique called *autarkic pairs*, which refers to pairs of vertex subsets predicted to form the problematic structure described above. These subsets are selected based on the observation that vertices in one subset correspond to pairs of vertices in the other. While vertices within each subset connect quickly, it takes too long for the two groups to connect in our algorithm. From each autarkic pair, we select one vertex pair, add the shortest path between them to the solution, and directly connect them with a zero-cost edge. This ensures that all vertices in the autarkic pair use this shortcut to reach their counterparts, preventing moats from growing excessively and reducing the maximum assigned value for those vertices. Once autarkic pairs are connected, we proceed with Legacy Moat Growing algorithm on the modified graph with these added links to satisfy all demands. Figure 2 illustrates how this process works. For details on how AUTARKICPAIRS works and the proofs of its guarantees, see Section 8.

The idea behind autarkic pairs is that, for any minimization problem with a known χ -approximation algorithm, the approximation can be improved by identifying a structure whose addition allows part of the optimal solution to be removed without violating feasibility. If the cost of this structure is less than χ times the cost of the removed part, and the improvement accounts for a constant fraction of the optimal cost, applying the

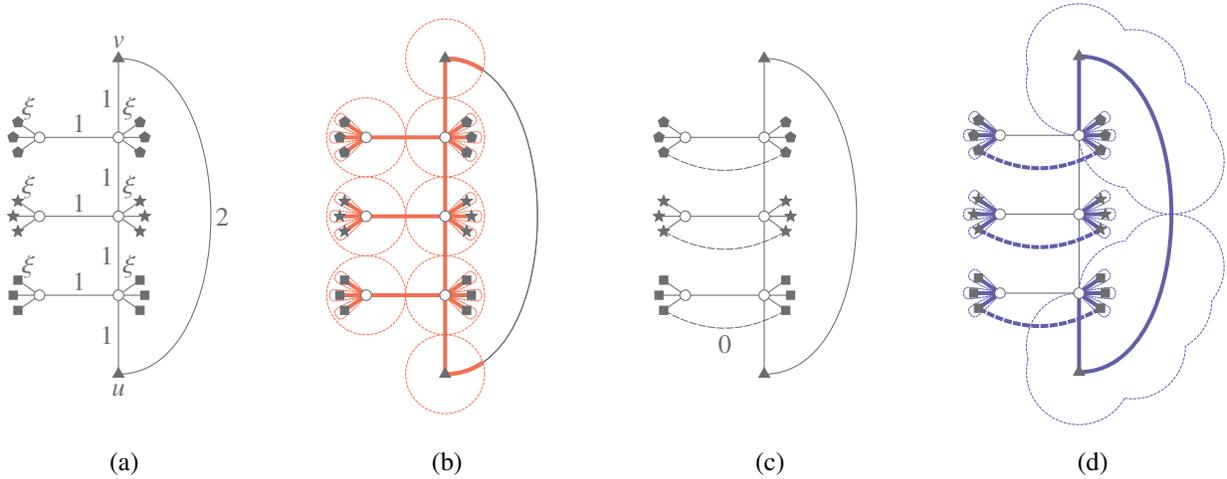


Figure 2: A Steiner Forest instance. (a) Initial configuration with n rows (here, $n = 3$), and small $\xi \in (0, 1/n]$. Each row contains three demand pairs, and vertices v and u also form a required pair. A vertical path through the middle rows costs $n + 1$ and connects v and u , while an edge of cost 2 offers an alternative. (b) After Legacy Moat Growing, vertices in each row quickly form two groups, but require significant growth to connect, resulting in coloring the vertical path. The total cost is $2n + 1 + O(1)$. (c) We detect vertices in each row as autarkic pairs and directly connect one vertex pair per row using their shortest paths, assuming a zero-cost edge connects them from this point on. (d) Running Legacy Moat Growing on the modified graph, the row vertices connect quickly via the new links, avoiding further growth and preventing coloring of the vertical path. Vertices u and v now connect via the edge of cost 2, and the total cost becomes $n + 2 + O(1)$.

χ -approximation algorithm to the remainder yields a better approximation. For Steiner Forest, we design autarkic pairs based on this natural idea and show that when the maximum assigned value for most optimal components is large enough, the resulting approximation is strictly better than 2.

These approaches lead to a $(2 - 2\alpha)$ -approximation algorithm for the Steiner Forest problem, where $\alpha \geq \frac{10^{-11}}{2}$. The algorithm satisfying Theorem 1 is presented in Algorithm 2. It has three parameters $0 < \beta, \epsilon, \eta < 1$, each used by different modules of the algorithm. The parameter values that achieve the desired approximation guarantee, along with the proof of the theorem, are provided in Section 9.

Algorithm 2 Main Algorithm

Input: A graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, a demand function $\text{PAIR} : V \rightarrow V$, and algorithm parameters $0 < \beta, \epsilon, \eta < 1$.

Output: A forest F satisfying PAIR demands.

- 1: **procedure** MAIN($G, \text{PAIR}, \beta, \epsilon, \eta$)
 - 2: $F, t^- \leftarrow \text{LEGACYMOATGROWING}(G, \text{PAIR})$
 - 3: $\text{SOL}_{LS}, t^+, y^{b+} \leftarrow \text{LOCALSEARCH}(G, t^-, \beta)$
 - 4: $t' \leftarrow \text{EXTEND}(G, t^-, t^+, y^{b+}, \epsilon)$
 - 5: $\text{SOL}_{XT}, t'', y^{b''} \leftarrow \text{LOCALSEARCH}(G, t', \beta)$
 - 6: $\text{SOL}_{AP} \leftarrow \text{AUTARKICPAIRS}(G, \text{PAIR}, y^{b+}, \eta)$
 - 7: **return** Best among SOL_{LS} , SOL_{AP} , and SOL_{XT}
-

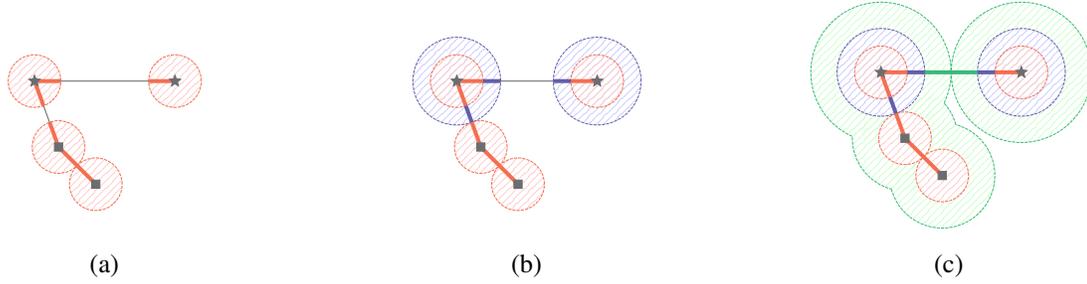


Figure 3: Here, there are two demand pairs: star pairs and square pairs. (a) Initially, all vertices form active sets and grow until the square pairs reach each other. (b) Once the connected component containing squares becomes inactive, the remaining connected components continue to grow until one of them reaches the square component. (c) At this point, there are two connected components that are both active and grow until they meet, causing the edge between the stars to become fully colored. Note that while all edges are added to F at this moment, the edge between a star and a square will be removed during the pruning phase since there was an inactive set containing squares that cut only this edge.

2.1 Overview of Techniques

In this section, we provide a comprehensive overview of the techniques that form the foundation of our algorithm. We begin by discussing the classic 2-approximation algorithms introduced by [AKR95] and revisited by [GW95], which serve as the basis for our work.

Legacy Moat Growing. The Legacy Moat Growing algorithm starts with an empty forest F and maintains its connected components, designating some as active sets. An active set is any component that contains at least one *unsatisfied* vertex, meaning a vertex that is not yet connected to its pair.

During the main phase of the algorithm, active sets grow uniformly, coloring their adjacent edges—those with exactly one endpoint in the set—at a constant rate. Edges are viewed as curves with lengths equal to their costs. Growth continues until an edge is fully colored, at which point it is added to F , merging the components at its endpoints. Only edges between distinct components are colored and added to F . Newly formed components become active or inactive depending on whether they contain any unsatisfied vertices.

The algorithm terminates when no active sets remain, ensuring all demand pairs are connected in the forest F . Finally, in a pruning phase, any edge of F that is the only edge crossing a set of vertices which was inactive at some point during the algorithm is removed, yielding the final solution. An illustration of the algorithm is provided in Figure 3, and a more detailed explanation appears in Section 4.1.

This algorithm achieves a 2-approximation because each portion of every edge is colored at most once, and each active set colors part of at least one edge from the optimal solution—specifically, from the paths connecting its unsatisfied vertices to their respective pairs. Moreover, the forest F is fully colored, with active sets coloring, on average, at most two edges in the final F . Thus, the cost of the solution is at most twice that of the optimal solution. Next, we take a closer look at the lower bound for the optimal solution and the upper bound for our solution.

As mentioned, the lower bound on the cost of the optimal solution is based on each active set coloring at least one edge of the optimal solution. However, if active sets color at least two such edges, a higher lower bound is achieved, leading to a better approximation factor. This idea was introduced by Ahmadi, Gholami, Hajiaghayi, Jabbarzade, and Mahdavi in their work on the Prize-Collecting Steiner Forest [AGH⁺25] and Prize-Collecting Steiner Tree [AGH⁺24]. They classify active sets into *single-edge sets* and *multi-edge sets*

based on how many edges they color from the optimal solution. If most of the active sets are multi-edge sets, meaning they color more than one edge of the optimal solution, then the optimal cost must be significantly larger than the total growth of active sets, implying that the moat growing algorithm's solution could be better than a 2-approximation. Conversely, if most active sets are single-edge sets, this offers insight into the structure of the optimal solution and its relationship to active set growth.

We now highlight and make use of the fact that active sets color, on average, two edges in the solution. This holds because, at any moment during the algorithm, if we contract the final forest F by its connected components at that time, all leaves in the resulting forest correspond to active sets. Consequently, the average degree of active sets—that is, the number of edges in the final F being colored at that moment—is at most 2. To maintain this property, we modify the growth rule so that the invariant of leaves being active always holds, ensuring the solution remains at most twice the total growth of active sets. A key condition that preserves this structure is that an inactive set never becomes active unless it merges with another active set to form a larger component. Furthermore, by allowing vertex sets to remain active longer than in Legacy Moat Growing, connectivity between vertices increases, resulting in valid solutions. Understanding the power of moat growing algorithms allows us to explore modifications that reduce the total growth of active sets. Since the solution cost is at most twice this value, such reductions lead to better upper bounds on the final cost.

Shadow Moat Growing. In Legacy Moat Growing algorithm, active sets are determined by unsatisfied demands, making it difficult to modify their behavior independently. To address this, we introduce a modified version called Shadow Moat Growing. This approach uses a *fingerprint* from a moat growing algorithm, assigning a time value to each vertex to ensure that active sets containing those vertices remain active until that time. Given this fingerprint, Shadow Moat Growing reruns a moat growing algorithm while preserving the intended active set behavior. It also allows increasing the time value of any vertex, enabling certain active sets to remain active longer and adapt their behavior accordingly. Details of this approach are provided in Section 4.2.

Assignment. Since our solution is at most twice the total growth of active sets, our goal is to effectively bound this growth. To do so, we define assignments, which distribute the growth of active sets across the vertices responsible for it. For example, in Legacy Moat Growing algorithm, each active component is active due to specific unsatisfied vertices, and we assign the corresponding growth to those vertices. However, modifying active set behavior complicates this process. Our algorithm involves multiple executions of moat growing, each using a different method of assignment. These assignments associate a value with each vertex such that the total assigned value captures the majority of the growth of active sets, allowing us to bound the solution cost by roughly twice this total. We therefore focus on bounding the assigned values.

To define assignments, we prioritize vertices according to Legacy Moat Growing execution, giving higher priority to vertices that are satisfied later. For each active set, we select the highest-priority vertex from each connected component of the optimal solution and decide how to distribute the growth among them. The distribution method varies: in some assignments, the growth is partitioned; in others, it is fully allocated to each selected vertex. Since these assignments are used solely for analysis and are not part of the actual algorithm, we assume access to the optimal solution for defining them. The general definition of assignments and the one corresponding to Legacy Moat Growing are provided in Sections 4.3 and 4.4, with additional assignments introduced later for other moat growing variants.

Now that we have demonstrated the flexibility of the moat growing algorithm and introduced the concept of assignment, we turn to improving the approximation factor for the Steiner Forest problem. While no algorithm is currently known to achieve this, significant progress has been made on a well-known special case: the Steiner Tree problem. The 2-approximation algorithm for Steiner Forest naturally extends to Steiner

Tree. Additionally, Steiner Tree admits a trivial 2-approximation via the minimum spanning tree on the metric closure of the terminals. Over the years, several algorithms have improved the approximation factor for this problem, many of which are based on local search [Zel93, KZ97, RZ05, TZ22].

Local Search. Local search algorithms for Steiner Tree typically begin with an initial solution, often the 2-approximation, and iteratively add a constant-size subgraph. Consequently, some edges can be removed as long as all demands remain satisfied, with the goal of improving the solution.

However, despite being well studied, this approach has not led to improvements for Steiner Forest. We revisit this paradigm with a broader perspective. Traditional local search relies on adding a small subgraph to improve the solution, but such subgraphs may not exist in Steiner Forest instances. More precisely, the optimal solution may consist of many connected components, and improving the solution could require breaking these components and regrouping vertices differently (see Figure 2). In Steiner Tree, knowing which vertices appear in the optimal solution is sufficient to reconstruct it. This gives hope that adding a small subset of vertices from the optimal solution could lead to improvement. In contrast, Steiner Forest requires knowing not only which vertices to include but also how they are grouped into components, making local improvements substantially more complex. The only known local search approach for Steiner Forest achieves a 69-approximation [GGK⁺18].

Here, we propose a novel local search algorithm for Steiner Tree. Our approach is more general than previous ones, yielding a solution better than the 2-approximation for Steiner Tree and introducing structural properties that help improve the approximation factor for Steiner Forest. It builds on a deep understanding of moat growing algorithms, specifically why their solution cost is at most twice the total growth of active sets and how to preserve this relationship while modifying growth behavior.

Unlike traditional local search, our algorithm aims to improve the upper bound on the solution rather than the cost itself. We define a *boost action*, which increases the fingerprint of a vertex, allowing its active set to remain active longer. In each iteration, we select a boost action that reduces the total growth of all active sets (see Figure 1). We define the *win* of a boost action as the reduction in total growth of existing active sets, and the *loss* as the additional growth introduced by the action. We apply a boost only when its win exceeds its loss by a factor of $1 + \beta$, for some $\beta \in (0, 1)$, calling it a *valuable boost action*. Each valuable boost reduces the total growth of active sets, and since the solution cost is at most twice this total, the bound improves accordingly. If the total growth becomes small enough, we obtain a good solution. Details of the local search algorithm are provided in Section 5.

Notably, the cost of the solution may increase in a single iteration, but the upper bound always decreases. Over multiple iterations, the cost typically improves as well. An example in Appendix A.1 demonstrates how repeated boosts can eventually yield the optimal solution, even if the first step does not reduce the cost. This illustrates the generality of our approach: unlike classical local search, which focuses on small local changes, our method can reshape the entire solution structure, even if it temporarily worsens. The key objective is to reduce the total growth of active sets, which upper bounds the solution cost, rather than minimizing the cost directly. This enables strategic decisions that ultimately lead to a good approximation.

Steiner Tree. The development of a general local search technique provides several useful properties for the resulting moat growing algorithm. In particular, we observe structural features in moat growing algorithms where no valuable boost action can be applied. These properties hold for the algorithm produced by our local search. We now present a key result for such algorithms, which allows us to prove an approximation factor strictly better than 2 for the Steiner Tree problem.

The main result, which we call the *claw property*, states that when no valuable boost action is available, the

cost of any tree connecting three actively connected vertices can be lower bounded by a factor greater than one times the sum of their assigned values, excluding the maximum. Here, actively connected means that the vertices remain in active sets until they all lie in the same connected component. We formalize this property in Section 6.1 and extend it to larger sets of actively connected vertices in Section 6.2. We then apply this result to the Steiner Tree problem in Section 6.3, proving Theorem 2. Note that the claw property is also used in our autarkic pair method, which we explain later.

To extend the claw property to more vertices, we assume the tree connecting them is a binary tree with the relevant vertices placed as leaves. Any tree can be transformed into such a binary tree by duplicating vertices and adding zero-cost edges. We analyze the structure by selecting a vertex as a center point and fixing one leaf in each of its three directions. Applying the claw property at this center bounds the sum of the assigned values of the selected leaves, excluding the maximum, by their total distance to the center. This follows from the assumption that boosting the center point is not valuable.

By applying this process to each vertex of the tree and selecting the closest leaf in each direction, we generate a set of inequalities. Summing the left-hand sides gives a bound on the total assigned value of the vertices, excluding the maximum. To make this summation meaningful, we show that at any time τ during the moat growing algorithm, if k active sets contain at least one of the relevant vertices (meaning at most k assigned values exceed τ), then there are $3(k - 1)$ center points whose inequalities contribute at that moment. We also show that the total of the right-hand sides is at most a constant factor times the cost of the tree.

Summing all inequalities, we find that the total assigned value to these vertices, minus the maximum, is at most $\frac{5+\beta}{6}$ times the cost of the tree, where β is the constant from our local search. Since all terminals in the Steiner Tree problem are actively connected in Legacy Moat Growing, and therefore also in the moat growing algorithm produced by our local search, we can apply this result to the terminals using the optimal solution as the connecting tree. This implies that our algorithm achieves a strictly better-than-2 approximation for Steiner Tree.

However, this analysis does not suffice to improve the approximation factor for Steiner Forest. The main issue is that we do not bound the maximum assigned value for each connected component of the optimal solution. This is not a problem when the connected components in our solution match those of the optimal solution, since a refined analysis shows that the cost of each component is bounded by twice the total assigned value minus the maximum. However, if our solution connects multiple components of the optimal solution, we need a bound on the maximum assigned value within each of those components (see Figure 2).

Another challenge is that the claw property requires all vertices in a component to be actively connected. In Steiner Forest, demands may not enforce this, and some vertices may be satisfied early and become inactive. To apply the claw property, we must first ensure that most vertices in each optimal component remain actively connected. We begin by addressing this issue and then return to the problem of bounding the maximum assigned value for each component of the optimal solution.

Extension. To address the issue of vertices within the same connected component of the optimal solution not being actively connected, we extend active sets by allocating an ϵ -fraction of their growth time as potential. This potential allows active sets, or their supersets, to continue growing before deactivation, ensuring that more vertices become actively connected. More precisely, based on the growth of an active set before extension, we allocate ϵ times this growth as potential to one of its vertices that remained active from the beginning until the set was formed. We then run a new moat growing algorithm in which a connected component is considered active at time τ under two conditions. The first is that it contains a vertex that was active at time τ in the moat growing algorithm before the extension. If this condition is not met, the component remains active as long as it has remaining potential, which it consumes until exhausted. Whenever

two active sets merge, their potentials are combined and transferred to the newly formed set.

After this extension step, which increases connectivity and merges some previously separate components, we hope that most vertices within each connected component of the optimal solution become actively connected. We then execute the local search to apply the same bounding technique used in the Steiner Tree case to the vertices of each optimal component that are actively connected.

On the other hand, if many vertices within a connected component remain not actively connected after the extension, we derive a stronger lower bound on the cost of the optimal solution. Since these vertices are not actively connected, those that become inactive earlier must, during their active sets' growth, color at least one edge of the optimal solution, as there must be a path connecting them to other vertices in the same component. Moreover, when these active sets grow solely due to potential, their vertices are already satisfied and remain active only because of the extension. In this case, they cannot color only one edge of the optimal solution, as such an edge would be redundant and contradict optimality. Therefore, these active sets must color at least two edges of the optimal solution and are classified as multi-edge sets. This provides a stronger lower bound than the previous assumption, where each active set was charged for coloring only one edge. With this improved lower bound, we can show that our solution is strictly better than twice the cost of the optimal solution. More details on the algorithm and analysis of our extension approach are provided in Section 7.

Note that while both the extension and the local search tend to increase the growth of active sets, they do so in different ways. The local search focuses on incremental improvement by extending the active duration of a single vertex at a time, with the amount of increase varying across steps. In contrast, the extension applies a small, uniform increase to the activity period of all active sets, relative to their previous growth.

To the best of our knowledge, the only prior work using a similar idea is by Bateni, Hajiaghayi, and Moharammi [BHM11], who developed a PTAS for the Steiner Forest problem. They introduced a procedure called PC-Clustering, which, after constructing a solution, assigns each component a potential equal to $1/\epsilon$ times its cost and lets components grow and color adjacent edges in a separate phase. Like us, their goal is to connect vertices that belong to the same component of the optimal solution. However, our approach differs in two key ways. First, in PC-Clustering, all components grow after the main algorithm has terminated, whereas we allow active sets to consume their potential immediately after deactivation, regardless of the state of other components. This ensures that the relevant vertices become actively connected, not just connected. Second, our potential is ϵ times the growth of each active set, a small value close to zero, while their potential is $1/\epsilon$ times the cost of the component's tree, which is large. As a result, they rely on unconnected vertices being far apart after PC-Clustering, while we only need to ensure that a small amount of additional growth cannot connect such vertices. We also leverage the fact that this small growth results in coloring multiple edges of the optimal solution, leading to a stronger lower bound.

Next, we address the issue of bounding the maximum assigned value for each connected component of the optimal solution.

Autarkic Pairs. As mentioned earlier, vertices in different connected components of the optimal solution can end up connected in our solution. This creates a challenge, as we need to bound the maximum assigned value of vertices within each connected component of the optimal solution. However, the bound derived from local search and used for the Steiner Tree case does not guarantee such a property. If the total assigned value of the vertices in a component of the optimal solution (including the maximum assigned value) is a constant factor smaller than the cost of its tree, then that component already has low assigned value and can be ignored. Likewise, if the maximum assigned value is small compared to the component's cost, its contribution is negligible, and the lack of a bound does not matter. The real difficulty arises when the total assigned value of a component is close to the cost of its optimal tree, and the maximum assigned value is large.

To address this, we try to identify such components despite not knowing the structure of the optimal solution. We do this by selecting groups of vertices whose growth times are largely aligned, meaning that the majority of growth from their active sets comes from active sets whose unsatisfied vertices are exactly those vertices, and their pairs behave symmetrically. We refer to these groups of vertices and their associated pairs as autarkic pairs. We show that any connected component of the optimal solution whose total assigned value nearly matches its tree cost and has a large maximum assigned value must contain such autarkic pairs.

Typically, these components consist of two groups of vertices, each of which connects internally very quickly during the moat growing process, while connecting the two groups takes significantly longer. This delay results in large assigned values for one vertex from each group. Note that if a component contains three actively connected vertices with large assigned values, we can apply the claw property to conclude that the cost of connecting them in the optimal solution must be large, contradicting the assumption that the assigned value closely matches the tree cost. This ensures that, in the moat growing algorithm produced by local search, such components must evolve as two early-formed groups that connect later. Moreover, if the active sets corresponding to these groups contain unsatisfied vertices from other components of the optimal solution, they must be multi-edge sets, as they color multiple components. This leads to a higher lower bound for the optimal solution, and similarly, the total assigned value no longer matches the tree cost. Therefore, we may assume that these two groups mostly grow together and do not include vertices from other components of the optimal solution, which results in them being selected as autarkic pairs. See Section 8.2 for further details.

After identifying autarkic pairs, we select the shortest path between an arbitrary pair of vertices from each pair and add it to our solution. We then insert a zero-cost edge between the endpoints of these paths and run Legacy Moat Growing again on the modified graph. In this new run, autarkic pairs no longer need to grow significantly to reach each other, as they are already connected. This ensures that the maximum assigned value within each connected component remains small relative to the total assigned value, effectively resolving the issue. See Figure 2 for further intuition and Section 8 for a full description and analysis.

To analyze the solution obtained through this approach, we focus on active sets whose unsatisfied vertices belong to a group of an autarkic pair. If a constant fraction of the total growth of these active sets comes from multi-edge sets, then similar to the extension case, this case is already handled. Otherwise, if these active sets are single-edge sets, meaning they color exactly one edge of the optimal solution, we can safely remove those edges from the optimal solution. Since we have already added connecting paths between autarkic pairs, removing these edges does not violate any demands. This gives a valid solution for the remaining instance, with cost significantly smaller than the original optimal solution. As Legacy Moat Growing yields a 2-approximate solution for the new instance, removing part of the optimal solution improves the upper bound on our solution by approximately twice the cost of the removed edges. While we do add some edges—the paths between autarkic pairs—their total cost is nearly equal to the cost of the removed edges. Since the gain is roughly twice the cost and the added cost is only equal to it, the overall solution cost is reduced. See Section 8.3 for more details.

It is worth noting that the autarkic pair method, like our local search, is completely novel. However, it follows a natural idea for improving approximation guarantees in minimization problems. The idea is to find a structure such that, when selected and added to any solution, and its cost set to zero, the optimal solution improves by more than $1/\chi$ of its value, where χ is the approximation factor of the base algorithm. If the cost of the added structure is less than χ times the saved cost, then combining the χ -approximate solution on the new instance with the added structure gives a better-than- χ approximation for the original instance. To achieve a constant-factor improvement, this saved cost must be a constant fraction of the optimal solution. For Steiner Forest, we design autarkic pairs to realize this idea in a problem-specific way. While this method does not always guarantee a constant-factor improvement, the extension and local search procedures complement it and handle the remaining cases.

3 Preliminaries

Graph Notation. We denote the cost of an edge e by c_e . For convenience, we extend this to a cost function $c : 2^E \rightarrow \mathbb{R}_{\geq 0}$ such that for any subset of edges $P \subseteq E$, we define $c(P) = \sum_{e \in P} c_e$. This allows us to compute the cost of trees, forests, and other edge sets more easily.

For any $S \subseteq V$, we denote the adjacent edges of S by $\delta(S)$, which is the set of edges with exactly one endpoint in S , i.e., the set of edges between S and $V \setminus S$ in G . Given sets $S, S' \subseteq V$, we say that S *cuts* S' , denoted $S \odot S'$, if $S \cap S' \neq \emptyset$ and $S' \not\subseteq S$. For a tree T , we write $S \odot T$ to mean $S \odot V(T)$, and for a forest F , we write $S \odot F$ to mean that S cuts at least one connected component of F .

We use $d(u, v)$ to denote the total cost of the shortest path between vertices u and v in the graph G .

Set Theory Notation. A family of sets is *disjoint* if its elements are pairwise disjoint. For a set of vertices V , we say a disjoint family \mathcal{F} is a *refinement* of another disjoint family \mathcal{H} if, for every $A \in \mathcal{F}$, there exists $B \in \mathcal{H}$ such that $A \subseteq B$. A disjoint family is a *partition* if the union of its sets equals the ground set.

Problem-Specific Notation. Throughout the paper, we fix a specific optimal solution, which is a forest in G . Among all optimal solutions of minimum cost, we select one with the fewest edges; this is the solution we refer to as *the* optimal solution. For simplicity, OPT refers both to the forest of the optimal solution and the partition of vertices it induces. Thus, we use $C \in \text{OPT}$ to denote a connected component in OPT, and $c(\text{OPT})$ to denote the cost of the optimal solution. For each component $C \in \text{OPT}$, we denote its tree by T_C . For any vertex v , $\text{COMP}(v)$ denotes the connected component in OPT that contains v .

We denote the pair of a vertex v by PAIR_v . We extend this notation and define a pair function $\text{PAIR} : 2^V \rightarrow 2^V$ such that for any subset $S \subseteq V$, we define $\text{PAIR}(S) = \{\text{PAIR}_v \mid v \in S\}$. A vertex is said to be *satisfied* if it is connected to its pair, and *unsatisfied* otherwise. We also define the following function:

Definition 3 (UNSATISFIED). For any subset of vertices $S \subseteq V$, we define $\text{UNSATISFIED}(S) \subseteq S$ as the subset of vertices in S whose pair is not also in S :

$$\text{UNSATISFIED}(S) = \{v \in S \mid \text{PAIR}_v \notin S\}.$$

4 Monotonic Moat Growing Algorithms

Here, we provide a detailed exploration of monotonic moat growing algorithms. We begin in Section 4.1 by introducing the `LEGACYMOATGROWING` algorithm, which yields a 2-approximate solution. In Section 4.2, we formalize the notion of monotonic moat growing and introduce the concept of fingerprints, which are used to implement Shadow Moat Growing, a technique capable of simulating any monotonic moat growing algorithm. Section 4.3 presents core definitions and properties relevant to monotonic moat growing algorithms. Finally, in Section 4.4, we analyze Legacy Execution, which corresponds to the call to `LEGACYMOATGROWING` in Line 2 of our main algorithm, and examine its key properties.

4.1 Legacy Moat Growing: The 2-Approximation Algorithm

In this section, we present Legacy Moat Growing algorithm, a 2-approximation method introduced by Agrawal, Klein, and Ravi [AKR95], and later used by Goemans and Williamson [GW95]. This algorithm forms the foundation of our approach, and its pseudocode is presented in Algorithm 3. Although the original

algorithm was introduced using a primal-dual approach, we omit the LP formulation here, as we can prove its approximation guarantee without it.

The algorithm maintains a forest F , which is initially empty, along with a collection of its connected components, denoted as FC . To track progress, the algorithm maintains a subset of FC called active sets, stored in $ActS$. These active sets correspond to connected components that contain vertices whose pairs are not yet in the same connected component. We also refer to these active sets as moats. Additionally, the algorithm keeps track of connected components that were inactive at some point during the algorithm in DS .

The algorithm proceeds as a continuous process where active sets grow and color their adjacent edges. Edges are modeled as curves with a length equal to their cost, and each portion of an edge can be colored only once. Let y_S represent the growth duration of the set $S \subseteq V$. When an edge e becomes fully colored, i.e., when $\sum_{S:e \in \delta(S)} y_S = c_e$, it is added to the forest F , and the connected components of its endpoints are merged within FC . The active sets $ActS$ are then updated by removing the previous components and adding the newly merged one. If the new component no longer contains any unsatisfied vertices, it becomes inactive. The process continues until all demands are satisfied.

After completing the main process, the algorithm enters a pruning phase. During this phase, as long as there exists a subset of vertices $S \in DS$ that cuts exactly one edge of F , that edge is removed from the forest. This removal is safe because the pairs of all vertices inside S are also contained within S , and removing the edge only disconnects vertices within S from those outside it. See Figure 3 for an illustration of the algorithm.

Since the algorithm operates as a continuous process, we define a *moment* as a conceptual unit of time that increases continuously throughout the moat growing process. This notion of time reflects the progress of active sets as they grow and color edges. We denote the state of the algorithm at a specific moment as time τ , with a small duration Δ such that no event occurs within the interval $(\tau, \tau + \Delta)$. During this interval, the variables F , FC , and $ActS$ remain unchanged, while the growth duration y_S of active sets increases by Δ , and the y_S of other sets stay the same.

To make the algorithm efficient and polynomial-time, we run it in discrete steps. The current time τ is maintained as a discrete variable that increments only when a new event occurs. At each step, the algorithm calculates the next event time Δ_e , which is the minimum duration needed to fully color an edge. The current time τ is then incremented by Δ_e , and the growth duration y_S of all active sets is updated by Δ_e . Subsequently, all edges that become fully colored at this moment are added to F , and the connected components and active sets are updated accordingly.

Finally, the algorithm returns a function $t : V \rightarrow \mathbb{R}_{\geq 0}$ that records the first moment when vertices v and PAIR_v become connected. This function is subsequently used in our main algorithm to facilitate running other moat growing algorithms on the given input.

Note that our explanation of pruning based on deactivated sets differs slightly from the 2-approximation algorithm commonly used for the Steiner Forest problem, although our approach has been widely applied in related settings. In the standard method for Steiner Forest, edges that are not required to satisfy any demands are removed. The edges we remove form a subset of these, yet we still obtain a 2-approximate solution. This distinction is important, as our goal is to design moat growing algorithms that operate independently of demand information during execution.

In each iteration of Algorithm 3, either an edge is added to the forest F , or an active set becomes inactive. Since both events can occur only a linear number of times in $|V|$, the algorithm runs in polynomial time.

Corollary 4. The `LEGACYMOATGROWING` procedure runs in polynomial time.

Algorithm 3 A 2-approximation Algorithm: Legacy Moat Growing

Input: A graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, and demand function $\text{PAIR} : V \rightarrow V$.

Output: A forest F that satisfies all demands and a function $t : V \rightarrow \mathbb{R}_{\geq 0}$ indicating the earliest moment v and PAIR_v are connected.

```
1: procedure LEGACYMOATGROWING( $G, \text{PAIR}$ )
2:    $\tau \leftarrow 0$ 
3:    $F \leftarrow \emptyset$ 
4:    $FC \leftarrow \{\{v\} \mid v \in V\}$ 
5:    $ActS \leftarrow \{\{v\} \mid v \in V, \text{PAIR}_v \neq v\}$ 
6:    $DS \leftarrow \{\{v\} \mid v \in V, \text{PAIR}_v = v\}$ 
7:    $t_v \leftarrow 0$  for all  $v \in V$ 
8:   Implicitly set  $y_S \leftarrow 0$  for all  $S \subseteq V$ 
9:   while  $ActS \neq \emptyset$  do ▷ While demands are not satisfied
10:     $\Delta_e \leftarrow \min_{e=uv \in E} \frac{c_e - \sum_{S \ni e} y_S}{|\{S_{u, S_v}\} \cap ActS|}$ , where  $u \in S_u \in FC, v \in S_v \in FC$ , and  $S_u \neq S_v$ 
11:     $\tau \leftarrow \tau + \Delta_e$ 
12:    for  $S \in ActS$  do
13:       $y_S \leftarrow y_S + \Delta_e$ 
14:      for  $e \in E$  do
15:        Let  $S_v, S_u \in FC$  be sets that contains each endpoint of  $e$ 
16:        if  $\sum_{S: e \in \delta(S)} y_S = c_e$  and  $S_v \neq S_u$  then ▷ Edge  $(v, u)$  become fully colored
17:           $F \leftarrow F \cup \{e\}$ 
18:           $FC \leftarrow (FC \setminus \{S_v, S_u\}) \cup \{S_v \cup S_u\}$ 
19:           $ActS \leftarrow (ActS \setminus \{S_v, S_u\}) \cup \{S_v \cup S_u\}$ 
20:          for  $w \in S_v$  such that  $\text{PAIR}_w \in S_u$  do
21:             $t_w, t_{\text{PAIR}_w} \leftarrow \tau$  ▷  $w$  and  $\text{PAIR}_w$  just connected
22:          for  $S \in ActS$  do
23:            if  $\text{UNSATISFIED}(S) = \emptyset$  then ▷  $S$  become inactive
24:               $ActS \leftarrow ActS \setminus \{S\}$ 
25:               $DS \leftarrow DS \cup \{S\}$ 
26:        while  $S \in DS$  exists such that  $|\delta(S) \cap F| = 1$  do ▷ Remove unnecessary edges
27:           $F \leftarrow F \setminus \delta(S)$ 
28:  return  $F, t$ 
```

4.2 Generalize Moat Growing: Shadow Moat Growing Algorithm

Here we propose a more abstract view of this algorithm, called monotonic moat growing algorithm. Monotonic moat growing algorithm is a more general concept that Legacy Moat Growing algorithm falls within it. This concept refer to set of algorithms similar to Legacy Moat Growing such that we will explore many other algorithms of this concept.

Definition 5 (Monotonic Moat Growing Algorithm). A continuous algorithm where a forest F , initially empty, is maintained. A set of connected components of F are called active sets. As time advance, y_S of active sets increase uniformly and they color their adjacent edges uniformly. When an edge become fully colored, it is added to F , the connected components of its endpoints are merged, and its new component become active. When an active set become inactive, it cannot activate again unless it merges with another active set, which form an active set that is union of those components. After completing the main process, the algorithm enters a pruning phase where while there is a subset of vertices that were an inactive connected

component at any moment of the algorithm and cuts exactly one edge of F , that edge is removed from F . The remaining F is the final solution of the monotonic moat growing algorithm.

Now, we want to define concept of fingerprint which then used by Shadow Moat Growing algorithm to simulate any monotonic moat growing algorithm on a given input.

Definition 6 (Fingerprint). Given a Steiner Forest instance and a monotonic moat growing algorithm, we define the *fingerprint* of the execution of the algorithm on the given input as a function $t : V \rightarrow \mathbb{R}_{\geq 0}$, which assigns to each vertex a value corresponds to time with the following properties:

- Every vertex v should be in an active set from the beginning until t_v in the algorithm.
- At any moment τ , every active set S must contain a vertex $v \in S$ such that $t_v \geq \tau$.

Note that a monotonic moat growing algorithm may admit infinitely many distinct fingerprints.

The procedure `LEGACYMOATGROWING` returns a function $t : V \rightarrow \mathbb{R}_{\geq 0}$, where t_v is the earliest moment at which vertex v becomes connected to PAIR_v in the algorithm. We now verify that this function is indeed a fingerprint.

Lemma 7. The output function t returned by `LEGACYMOATGROWING` is a fingerprint.

Proof. We want to prove that both conditions of Definition 6 holds.

- For each vertex v , it must be in active sets from moment 0 up to moment t_v : Initially, each vertex v which $\text{PAIR}_v \neq v$ is placed in an active singleton set in Line 5. A connected components deactivated only if all its vertices are satisfied (see Line 23). Since vertex v connect to PAIR_v at time t_v (see Line 21), it is unsatisfied until that time. Therefore, every connected component containing v before time t_v has an unsatisfied vertex and should be active.
- At any moment τ , for each active set in that moment S , there should be a vertex $v \in S$ with $t_v \geq \tau$: In `LEGACYMOATGROWING`, a connected component S is active only if it contains at least one unsatisfied vertex $v \in S$ (see Line 23). Since v and PAIR_v are not yet connected at time τ (see Line 21), we have $t_v > \tau$ which complete the proof.

Since both conditions are satisfied, the function t returned by Algorithm 3 is a fingerprint. □

Now we propose Shadow Moat Growing algorithm, which is a monotonic moat growing algorithm that utilizes fingerprints to simulate any monotonic moat growing algorithm. The pseudocode for this algorithm can be found in Algorithm 4. Although we never directly invoke this algorithm, it serves an essential purpose in demonstrating that monotonic moat growing algorithms, such as `LEGACYMOATGROWING`, can be interpreted through this framework. This understanding allows us to modify the moat growing process by adjusting the fingerprint. Furthermore, our algorithm invokes advanced versions of `SHADOWMOATGROWING`, such as Algorithms 5 and 7. These advanced algorithms take the fingerprint of a monotonic moat growing algorithm as input and assume that running `SHADOWMOATGROWING` with this fingerprint simulates the desired moat growing algorithm. They then modify the fingerprint to obtain a new monotonic moat growing algorithm, allowing for flexibility and customization in the moat growing process.

The process of `SHADOWMOATGROWING` closely resembles that of `LEGACYMOATGROWING`. The key difference lies in the criteria for marking a connected component as an active set. In `LEGACYMOATGROWING`, a connected component is active if it contains an unsatisfied vertex. In contrast, in `SHADOWMOATGROWING`, a connected component is considered an active set if it contains a vertex v such that $t_v > \tau$, where τ denotes the current moment of the algorithm and t represents the fingerprint given as input to `SHADOWMOATGROWING`. In essence, the fingerprint t_v enforces that any connected component containing vertex v must remain active until time t_v .

Algorithm 4 Shadow Moat Growing

Input: A graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, and a function $t : V \rightarrow \mathbb{R}_{\geq 0}$ specifying the minimum time each vertex v must be growing.

Output: F , the resulting forest of fingerprint t .

```
1: procedure SHADOWMOATGROWING( $G, t$ )
2:    $\tau \leftarrow 0$ 
3:    $F \leftarrow \emptyset$ 
4:    $FC \leftarrow \{\{v\} \mid v \in V\}$ 
5:    $ActS \leftarrow \{\{v\} \mid v \in V, t_v > 0\}$ 
6:    $DS \leftarrow \{\{v\} \mid v \in V, t_v = 0\}$ 
7:   Implicitly set  $y_S \leftarrow 0$  for  $S \subseteq V$ 
8:   while  $ActS \neq \emptyset$  ▷ While there exists an active set
9:      $\Delta_e \leftarrow \min_{e=uv \in E} \frac{c_e - \sum_{S \ni e} y_S}{|\{S_u, S_v\} \cap ActS|}$ , where  $u \in S_u \in FC, v \in S_v \in FC$ , and  $S_u \neq S_v$ 
10:     $\Delta_t \leftarrow \min_{v \in V, t_v > \tau} (t_v - \tau)$ 
11:     $\Delta \leftarrow \min(\Delta_e, \Delta_t)$ 
12:    for  $S \in ActS$  do
13:       $y_S \leftarrow y_S + \Delta$ 
14:       $\tau \leftarrow \tau + \Delta$ 
15:      for  $e \in E$  do
16:        Let  $S_v, S_u \in FC$  be sets that contain each endpoint of  $e$ 
17:        if  $\sum_{S: e \in \delta(S)} y_S = c_e$  and  $S_v \neq S_u$  then ▷ Edge  $(v, u)$  become fully colored
18:           $F \leftarrow F \cup \{e\}$ 
19:           $FC \leftarrow (FC \setminus \{S_v, S_u\}) \cup \{S_v \cup S_u\}$ 
20:           $ActS \leftarrow (ActS \setminus \{S_v, S_u\}) \cup \{S_v \cup S_u\}$ 
21:          for  $S \in ActS$  do
22:            if  $t_v \leq \tau$  for all  $v \in S$  then ▷  $S$  become inactive
23:               $ActS \leftarrow ActS \setminus \{S\}$ 
24:               $DS \leftarrow DS \cup \{S\}$ 
25:          while  $S \in DS$  exists such that  $|\delta(S) \cap F| = 1$  do ▷ Remove unnecessary edges
26:             $F \leftarrow F \setminus \delta(S)$ 
27:          return  $F$ 
```

Next, we aim to show that Shadow Moat Growing, given a fingerprint of a monotonic moat growing algorithm, accurately simulates that algorithm.

Lemma 8. If t is the fingerprint of a monotonic moat growing algorithm A on a given graph G , then for any moment during the process of A , the connected components, active sets, and the y_S values in A will be identical to those in SHADOWMOATGROWING(G, t) at that moment.

Proof. First, note that if at any moment until time τ , the active sets in both algorithms are the same, then the y_S values of sets $S \subseteq V$ are also the same in both algorithms at moment τ . This is because y_S represents the duration for which a component is active, and since active sets are identical in both algorithms, these values must also be the same.

Now, we show that at each moment τ , the set of all connected components and active sets are identical in both algorithms. This directly implies the identity of y_S , and consequently, the final forest F in both algorithms A and SHADOWMOATGROWING.

We prove this by induction on the events that change connected components and active sets in either of these algorithms. The induction base (at the start of the algorithm) is clear since all singleton sets are connected components in both algorithms. Additionally, in SHADOWMOATGROWING, the singleton sets for vertices with $t_v > 0$ start as active sets, and $t_v > 0$ for a vertex v if and only if it starts as an active set in A .

Now, assume for contradiction that there exists a first moment τ at which the connected components or active sets differ between the two algorithms. We consider three possible cases:

1. An active set S deactivates in A but remains active in SHADOWMOATGROWING: Since S is active in SHADOWMOATGROWING, there must be a vertex $v \in S$ such that $t_v > \tau$. By the first property in Definition 6, v must have remained in active sets of A until t_v . Therefore, S should still be active in A , which contradicts the assumption that S has deactivated in A . Hence, this case is not possible.
2. An active set S deactivates in SHADOWMOATGROWING but remains active in A : According to the implementation of SHADOWMOATGROWING, an active set deactivates only when there is no vertex $v \in S$ such that $t_v > \tau$. However, if S is still active in A , then by the second property in Definition 6, the fingerprint function t must assign a value larger than τ to some vertex in S . This contradicts the assumption that no vertex $v \in S$ has $t_v > \tau$. Therefore, this case is also not possible.
3. Two connected components merge in one algorithm but not in the other: We have established that until τ , the set of active sets, and therefore, y_S values are identical in both algorithms. A merge occurs when an edge becomes fully colored, meaning $\sum_{S:e \in \delta(S)} y_S = c_e$. Since being fully colored for an edge depends solely on y_S values, which are identical in both algorithms, any merge that occurs in one algorithm must also occur in the other. Thus, this case is also impossible.

Since all cases leading to a difference in connected components and active sets have been ruled out, it follows that these remain identical in both algorithms at all times. Consequently, the y_S values are the same. Additionally, the third point above shows that the same edge becomes fully colored in both algorithms, which ultimately proves that the forest F remains identical in both algorithms throughout their execution. Note that the pruning phase does not violate this consistency, as connected components and active sets remain identical at every moment in both algorithms, resulting in the same connected components becoming inactive at the same moments. Since the pruning phase removes edges based on inactive connected components, it follows that both algorithms prune the same set of edges. \square

The next lemma is a classic result used in previous works that leverage a similar primal-dual approach for related problems. It shows that the final forest cost is at most twice the total growth of active sets. We prove this for any monotonic moat growing algorithm and then focus on designing new moat growing algorithms that reduce this total growth while satisfying all demands, aiming to produce improved solutions with a better upper bound.

Lemma 9. For any monotonic moat growing algorithm and any vertex $v \in V$, the total cost of the resulting forest F is at most twice the total growth of active sets not containing v , that is

$$c(F) \leq 2 \sum_{\substack{S \subseteq V \\ v \notin S}} y_S.$$

Proof. Since all edges in F are fully colored, we can say

$$\begin{aligned} c(F) &= \sum_{e \in F} c_e \\ &= \sum_{e \in F} \sum_{\substack{S \subseteq V \\ e \in \delta(S)}} y_S \\ &= \sum_{S \subseteq V} |\delta(S) \cap F| \cdot y_S. \end{aligned}$$

Now, we show that at any moment τ of the monotonic moat growing, the increase in this value is at most the increase in

$$2 \sum_{\substack{S \subseteq V \\ v \notin S}} y_S.$$

Let $ActS_\tau$ be the active sets at moment τ . Then, if y_S of active sets increase by Δ at moment τ , the increase in $\sum_{S \subseteq V} |\delta(S) \cap F| \cdot y_S$ will be

$$\sum_{S \in ActS_\tau} |\delta(S) \cap F| \cdot \Delta,$$

while the increase in $2 \sum_{S: v \notin S} y_S$ is

$$2 \sum_{\substack{S \in ActS_\tau \\ v \notin S}} \Delta \geq 2(|ActS_\tau| - 1) \cdot \Delta.$$

Therefore, it suffices to show that

$$\sum_{S \in ActS_\tau} |\delta(S) \cap F| \leq 2(|ActS_\tau| - 1)$$

at any moment τ .

Given the components FC_τ at moment τ , define $F_{contracted}$ as the graph obtained from F by contracting each set in FC_τ into a single vertex and removing isolated vertices. Now, let V_a represent the vertices in $F_{contracted}$ corresponding to active sets $ActS_\tau$, and V_i the rest of the vertices corresponding to inactive components. Then, $F_{contracted}$ has the following properties:

- $F_{contracted}$ is a forest, since F is a forest and any contracted set cannot contain two vertices in one component of F without containing the path between them.
- For any vertex v of $F_{contracted}$, if S is the corresponding set in FC_τ , $\deg_{F_{contracted}}(v) = |\delta(S) \cap F|$. Additionally, isolated vertices excluded from $F_{contracted}$ represent sets S with $|\delta(S) \cap F| = 0$.
- For any vertex v in V_i , we have $\deg_{F_{contracted}}(v) \geq 2$. $F_{contracted}$ contains no isolated vertices, so $\deg_{F_{contracted}}(v) \geq 1$. Now, assume otherwise that $\deg_{F_{contracted}}(v) = 1$. Then, consider the set S corresponding to v . Since $v \in V_i$, S must be in DS at the end of the algorithm. However, the final check in the monotonic moat growing algorithm would remove the single edge attached to S from F if this was the case. Hence, $\deg_{F_{contracted}}(v)$ must be at least 2.

Now, combining these properties, we can say

$$\begin{aligned}
\sum_{S \in \text{Act}S_\tau} |\delta(S) \cap F| &= \sum_{v \in V_a} \text{deg}_{F_{\text{contracted}}}(v) \\
&= \sum_{v \in V_a \cup V_i} \text{deg}_{F_{\text{contracted}}}(v) - \sum_{v \in V_i} \text{deg}_{F_{\text{contracted}}}(v) \\
&\leq \sum_{v \in V_a \cup V_i} \text{deg}_{F_{\text{contracted}}}(v) - 2|V_i| && (\text{deg}_{F_{\text{contracted}}}(v) \geq 2 \text{ for all } v \in V_i) \\
&\leq 2(|V_a| + |V_i| - 1) - 2|V_i| && (F_{\text{contracted}} \text{ is a forest with at most } |V_a| + |V_i| - 1 \text{ edges}) \\
&\leq 2(|V_a| - 1) \\
&\leq 2(|\text{Act}S_\tau| - 1)
\end{aligned}$$

which completes the proof. \square

4.3 Notation and Definitions for Moat Growing Algorithms

In this section, we introduce notation and properties specific to moat growing algorithms. We begin by examining how often active sets cut the optimal solution, using this to classify active sets and to partition the optimal solution based on which class of active sets colors each portion. We then present a general definition of assignments, which serves as a key component of our analysis.

Since our approach focuses on variants of monotonic moat growing algorithms, we introduce common notation that will be used throughout the paper. For the remainder of the paper, we consider four specific executions of monotonic moat growing algorithms, each denoted by a distinct symbol. These symbols appear as superscripts on variables to indicate the corresponding execution. See Figure 4 for the list of symbols and their associated moat growing algorithms.

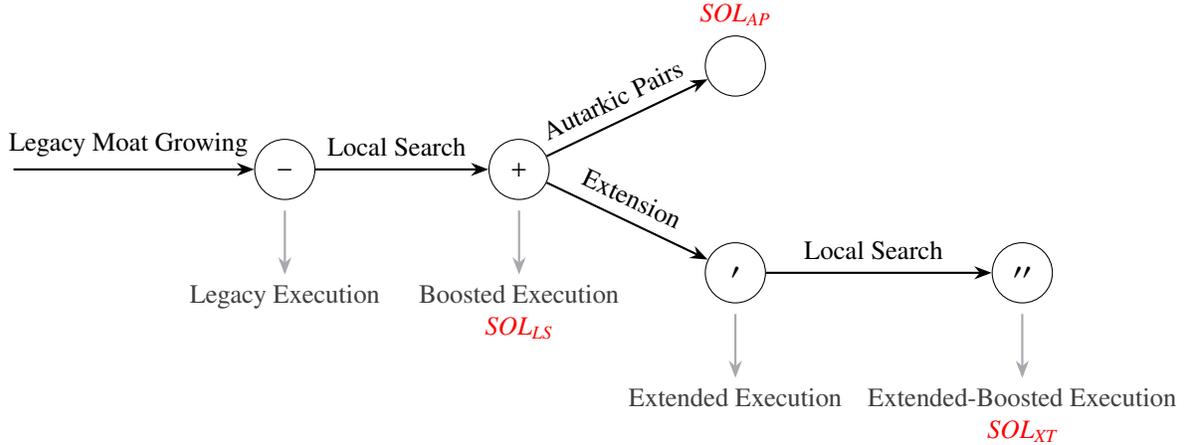


Figure 4: Flow of Algorithm 2, showing the different modules and their order. Each module results in an “execution” of a monotonic moat growing algorithm, with the name of each execution shown below the corresponding circle. The symbol inside each circle is used to refer to variables from that execution by placing it as a superscript. The three solutions compared at the end of the algorithm are also identified, along with their positions in the flow and how they are obtained.

Active and Superactive. We say that a vertex is *active* at moment τ in a monotonic moat growing algorithm if it has not belonged to any inactive set up to that point. A set of vertices is said to be *actively connected* if

its members remain in active sets from the beginning of the algorithm until they all reach one another and lie in the same connected component.

We now introduce the notion of *superactive sets*, which capture subsets of vertices that are actively connected.

Definition 10 (Superactive Sets). For an active set in an execution of a monotonic moat growing algorithm, its corresponding superactive set is the subset of its vertices that are active at the moment the active set is formed. We denote the superactive set of an active set S by $\text{SUPERACTIVE}(S)$, and for a particular execution $(*)$, we write $\text{SUPERACTIVE}^*(S)$. The term superactive sets refers to the collection of such subsets corresponding to all active sets throughout the algorithm.

Although some vertices may be immediately deactivated and never included in any active set, we assume that $\{v\}$ is a superactive set for every vertex $v \in V$.

In a monotonic moat growing algorithm, each connected component is formed by merging two existing components. As a result, the family of connected components that appear during the execution forms a *laminar* structure: any two such subsets are either disjoint or one is a subset of the other. Since each active set corresponds to a connected component at some moment, we obtain the following corollary:

Corollary 11. Active sets during a monotonic moat growing algorithm form a laminar family.

Similarly, the superactive sets in a monotonic moat growing execution also form a laminar family. Initially, superactive sets are disjoint. As the execution proceeds, when two active sets merge, their superactive sets merge as well. When an active set is deactivated, its superactive set becomes empty. These rules ensure that the laminar structure is preserved throughout the execution. This property allows us to define the following notion.

Definition 12 (Maximal Superactive Sets). In an execution of a monotonic moat growing algorithm, a superactive set is called *maximal* if it is not strictly contained in any other superactive set. The family of maximal superactive sets, denoted by \mathcal{M} , is obtained by filtering the collection of superactive sets to retain only the maximal ones. This family forms a partition of the vertex set.

For any maximal superactive set S , there exists an active set S' whose corresponding superactive set is S , and which becomes inactive at some moment τ . When we refer to the deactivation of a superactive set S , we mean the moment τ when the associated active set S' is deactivated. We also refer to S' as the active set from which S is derived, although S' may contain subsets whose superactive set is also S .

Comparison of Fingerprints and Refinement Families. We say that a fingerprint t^{out} is *larger* than a fingerprint t^{in} if $t_v^{out} \geq t_v^{in}$ for all $v \in V$. In this case, we also say that t^{in} is *smaller* than t^{out} .

We now compare pairs of monotonic moat growing executions whose fingerprints satisfy this relation. These comparisons play a central role in the remainder of the paper, as we will increase the value of t for some vertices to generate larger fingerprints and design new moat growing algorithms with the goal of obtaining better solutions. Figure 5 summarizes the key properties of such comparisons. The precise statements and their proofs are provided in the following lemmas.

The next lemma states that if the fingerprint of one execution is larger than that of another, then at any moment during the execution, any active set in the run with the smaller fingerprint is a subset of an active set in the run with the larger fingerprint at the same time. A similar statement holds for connected components.

Lemma 13. Let t^{in} and t^{out} be fingerprints of two monotonic moat growing algorithms on a graph G , where t^{out} is larger than t^{in} . At any moment τ during these executions, the following hold:

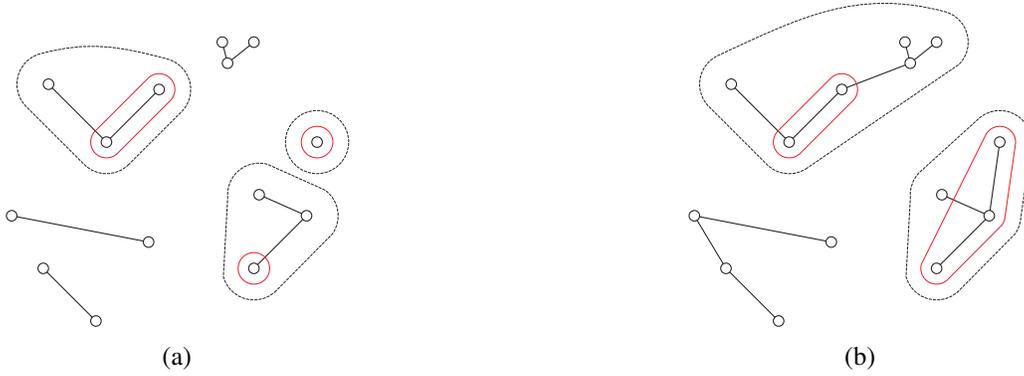


Figure 5: Connected components, active sets (black dashed lines), and their superactive sets (solid red lines) at the same moment in two monotonic moat growing executions. The fingerprint in (b) is larger than in (a). As shown, every connected component, active set, and superactive set in (a) is a subset of its counterpart in (b). The execution in (b) exhibits greater connectivity: any two vertices connected in (a) are also connected in (b), some vertices that are inactive in (a) are active in (b), and some vertices not included in any superactive set in (a) belong to one in (b).

- The active sets in the execution with fingerprint t^{in} form a *refinement* of the active sets in the execution with fingerprint t^{out} .
- The connected components in the execution with fingerprint t^{in} form a *refinement* of those in the execution with fingerprint t^{out} .

Proof. For each moment τ , let $ActS_\tau^{in}$ denote the active sets and FC_τ^{in} denote the connected components in the execution on t^{in} , and let $ActS_\tau^{out}$ and FC_τ^{out} denote the corresponding active sets and connected components in the execution on t^{out} .

Initially, when $\tau = 0$, we have $ActS_0^{in} \subseteq ActS_0^{out}$ since vertices with $t_v^{in} = 0$ may have $t_v^{out} > 0$. Also, $FC_0^{in} = FC_0^{out}$.

Assume τ is the minimum time at which neither $ActS_\tau^{in} \leq ActS_\tau^{out}$ nor $FC_\tau^{in} \leq FC_\tau^{out}$.

The relation \leq means the left-hand side is a refinement of the right-hand side. Additionally, we say a set S of vertices covers another set S' if $S' \subseteq S$.

There are four possible cases for the last event that could lead to this violation:

1. *Two connected components C_1 and C_2 in the execution on t^{out} are merged:* Then $C_1 \cup C_2$ covers everything C_1 and C_2 independently covered. Therefore, this case cannot break refinement.
2. *Two connected components C_1 and C_2 in the execution on t^{in} are merged:*
 - If both C_1 and C_2 were covered by the same active set C in the execution on t^{in} beforehand, then C also covers $C_1 \cup C_2$ at this moment. Thus, this case preserves refinement.
 - If C_1 and C_2 were covered by different sets beforehand, let the merging edge be $e = (u, v)$. At any earlier time, whenever u and v belonged to active sets in the execution on t^{in} , they have been in active sets cutting e on both executions (until now active sets hold the refinement condition). Thus, $\sum_{S:e \in \delta(S)} y_S = c_e$ must hold in both runs. This contradicts C_1 and C_2 being subsets of different active sets at moment τ . Hence, this case is also not possible.

3. *One active set C' in the execution on t^{in} becomes deactivated:* In this case, first, there is no change in FC_τ^{in} and FC_τ^{out} . Second, since only one set is removed from $ActS_\tau^{in}$, we still have $ActS_\tau^{in} \leq ActS_\tau^{out}$. Thus, this case cannot violate refinement.
4. *One active set C' in the execution on t^{out} becomes deactivated:* Again, there is no change in FC_τ^{in} and FC_τ^{out} . Since C' becomes deactivated, for all $v \in C'$, we have $t_v^{out} \leq \tau$. This condition also holds for any $C \subseteq C'$ as $t_v^{in} \leq out_v$. Therefore, there is no active $C \subseteq C'$ in execution on t^{in} .

Since none of these cases can lead to a violation, such a time τ cannot exist. Hence, for all τ , $ActS_\tau^{in}$ is a refinement of $ActS_\tau^{out}$, and FC_τ^{in} is a refinement of FC_τ^{out} . \square

The next three lemmas establish direct consequences of the above lemma.

Lemma 14. Let t^{in} and t^{out} be fingerprints of two monotonic moat growing algorithms on a graph G , where t^{out} is larger than t^{in} . If a vertex v is active at moment τ in the execution with fingerprint t^{in} , then v is also active at moment τ in the execution with fingerprint t^{out} .

Proof. Assume for contradiction that v is active at moment τ in the execution on t^{in} but not in the execution on t^{out} . Then v must belong to an active set S in the execution on t^{in} at time τ , but not to any active set in the execution on t^{out} at that time. However, by Lemma 13, S must be contained in an active set S' in the execution on t^{out} at the same moment. This implies $v \in S'$, so v is active in the execution on t^{out} , contradicting the assumption. \square

Lemma 15. Let t^{in} and t^{out} be fingerprints of two monotonic moat growing algorithms on a graph G , where t^{out} is larger than t^{in} . If two vertices are actively connected in the execution on t^{in} , then they are also actively connected in the execution on t^{out} .

Proof. Assume u and v are actively connected in the execution on t^{in} . This means that at some moment τ , they lie in the same active set S and both are still active.

By Lemma 13, in the execution on t^{out} , u and v must belong to an active set $S' \supseteq S$ at the same time τ . Furthermore, by Lemma 14, both u and v are active at time τ in the execution on t^{out} . Therefore, u and v are active and lie in the same connected component in the execution on t^{out} , so they are actively connected. \square

Lemma 16. Let t^{in} and t^{out} be fingerprints of two monotonic moat growing algorithms on a graph G , where t^{out} is larger than t^{in} . At any time τ during both executions, the superactive sets derived from the active sets in the execution with fingerprint t^{in} form a refinement of the superactive sets derived from the active sets in the execution with fingerprint t^{out} .

Proof. Let $SUPERACTIVE^{in}$ and $SUPERACTIVE^{out}$ be the functions that return the superactive set of a given active set in the executions on t^{in} and t^{out} , respectively.

At time τ , we claim that for any active set S in the execution on t^{in} , $SUPERACTIVE^{in}(S) \subseteq SUPERACTIVE^{out}(S')$ for the active set $S' \supseteq S$ in the execution on t^{out} , whose existence is guaranteed by Lemma 13. This is sufficient to show that the superactive sets under t^{in} form a refinement of those under t^{out} .

Take any vertex $v \in SUPERACTIVE^{in}(S)$. By definition, v is active at the time S is formed in the execution on t^{in} , and remains active up to τ . By Lemma 14, v must also be active at τ in the execution on t^{out} . Since $v \in S \subseteq S'$, it follows that $v \in SUPERACTIVE^{out}(S')$. This completes the proof. \square

Cut-Based Classification. Next, we begin by defining the notions of *single-edge set* and *multi-edge set*, originally introduced by [AGH⁺25] and later used by [AGH⁺24]. These concepts are fundamental to our analysis.

Definition 17 (Single-Edge Set and Multi-Edge Set). For any active set $S \subseteq V$, we call S a *single-edge set* if it cuts exactly one edge of the optimal solution, i.e., $|\delta(S) \cap \text{OPT}| = 1$, and a *multi-edge set* if it cuts more than one edge of the optimal solution, i.e., $|\delta(S) \cap \text{OPT}| > 1$.

While the works of [AGH⁺25, AGH⁺24] use these terms to classify sets based on how many edges of the optimal solution they cut, we require a more refined usage. In our setting, we consider specific forests—typically subgraphs of the optimal solution—and classify active sets based on how many edges of that subgraph they cut. Furthermore, we partition the forest based on the types of active sets that color its edges.

Definition 18. Let $\text{Act}S$ denote the family of all active sets that appear throughout a given monotonic moat growing algorithm. For a forest F , define:

$$\begin{aligned} \text{SAct}S_F &= \{S \in \text{Act}S \mid |\delta(S) \cap F| = 1\}, & (\text{active sets that cut exactly one edge of } F) \\ \text{MAct}S_F &= \{S \in \text{Act}S \mid |\delta(S) \cap F| > 1\}. & (\text{active sets that cut more than one edge of } F) \end{aligned}$$

We then define:

- $UC(F)$ as the total portion of F not colored by any active set,
- $SC(F) = \sum_{S \in \text{SAct}S_F} y_S$ as the total portion of F colored by sets in $\text{SAct}S_F$,
- $MC(F) = \sum_{S \in \text{MAct}S_F} |\delta(S) \cap F| \cdot y_S$ as the total portion of F colored by sets in $\text{MAct}S_F$,
- $UM(F) = UC(F) + MC(F)$.

For a particular execution $(*)$, we denote these quantities by UC^* , MC^* , and UM^* .

The following corollary follows directly from the above definitions:

Corollary 19. For any monotonic moat growing algorithm and any forest F , we can partition F into uncolored, single-colored, and multi-colored portions as follows:

$$c(F) = UC(F) + SC(F) + MC(F) = SC(F) + UM(F).$$

Next, we prove the monotonicity property of UM .

Lemma 20. For any monotonic moat growing algorithm, and any forests F and F' such that F' is a subgraph of F , we have

$$UM(F') \leq UM(F).$$

Proof. First, note that $UC(F') \leq UC(F)$ since every edge in F' is also in F , and any uncolored portion in F' must also be uncolored in F .

Furthermore, we have $\text{MAct}S_{F'} \subseteq \text{MAct}S_F$, because if an active set cuts more than one edge in F' , it must also cut more than one edge in F . Therefore, any portion of F' colored by $\text{MAct}S_{F'}$ is also part of the portion of F colored by $\text{MAct}S_F$. It follows that $MC(F') \leq MC(F)$.

Combining both bounds, we obtain:

$$UM(F') = UC(F') + MC(F') \leq UC(F) + MC(F) = UM(F). \quad \square$$

The next lemma shows that $UM(F)$ provides a lower bound on the cost of the optimal solution that exceeds the total growth of the active sets involved in coloring it.

Lemma 21. For any monotonic moat growing algorithm and any forest F , we have

$$\sum_{\substack{S \subseteq V \\ S \odot F}} y_S \leq c(F) - \frac{UM(F)}{2}.$$

Proof. We partition the active sets that cut F into those that cut exactly one edge and those that cut at least two edges:

$$\begin{aligned} \sum_{\substack{S \subseteq V \\ S \odot F}} y_S &= \sum_{\substack{S \subseteq V \\ |\delta(S) \cap F|=1}} y_S + \sum_{\substack{S \subseteq V \\ |\delta(S) \cap F| \geq 2}} y_S \\ &\leq \sum_{S \in SA_{ct} S_F} y_S + \sum_{S \in MA_{ct} S_F} \frac{|\delta(S) \cap F|}{2} \cdot y_S \\ &= SC(F) + \frac{MC(F)}{2} \\ &\leq SC(F) + \frac{UM(F)}{2} && \text{(Definition 18)} \\ &= c(F) - \frac{UM(F)}{2}. && \text{(Corollary 19)} \end{aligned}$$

□

Next, we prove that for any tree T , every active set contributing to $SC(T)$ must cut the set of leaves of T (see Figure 6). We then use this fact in Lemma 23 to derive a lower bound on $MC(T)$. This observation is crucial, as it implies that any active set that does not cut the set of leaves of a tree in the optimal solution must color more than one of its edges. Identifying such active sets enables us to establish a lower bound on the cost of the optimal solution that exceeds the total growth of the active sets involved in coloring it, similar in spirit to the above lemma.

Lemma 22. Let T be a tree with leaf set $S' \subseteq V$, and let $S \subseteq V$ be a set such that $|T \cap \delta(S)| = 1$. Then, $S \odot S'$.

Proof. Let $e = \{u, v\} \in T \cap \delta(S)$ be the unique edge of T cut by S . Since S cuts T , it must contain some, but not all, of the vertices of T . Removing e splits T into two connected components, each a subtree of T , such that one component lies entirely inside S , and the other lies entirely outside. Suppose this were not the case: that is, if one of the components contained vertices both inside and outside of S , then S would cut at least one additional edge of T , contradicting the assumption that $|T \cap \delta(S)| = 1$. On the other hand, if both components were on the same side of S (both fully inside or both fully outside), then S would not cut e at all. Therefore, the two components must lie entirely on opposite sides of S .

Now consider the two components created by removing e . If a component contains more than one vertex, it must have at least two leaves, and at least one of them is different from u and v . Such a vertex is also a leaf in the original tree T , i.e., a member of S' . If a component consists of a single vertex, then that vertex is necessarily a leaf in T and thus also belongs to S' . Therefore, one vertex from S' lies inside S and another lies outside S , implying $S \odot S'$. □

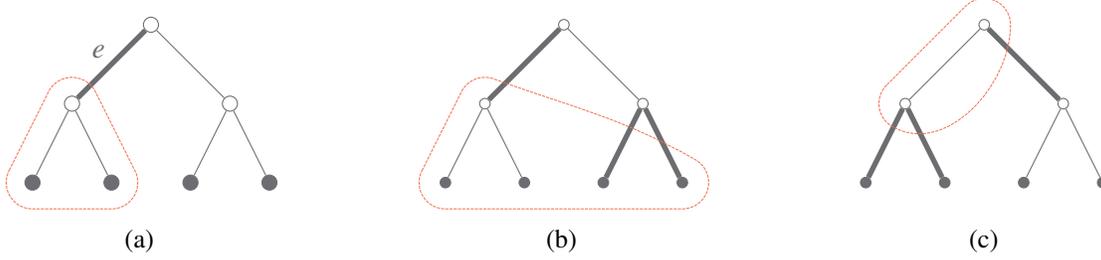


Figure 6: Tree T with three subsets of vertices. (a) A set that cuts the leaves of T can cut only one edge of T . (b, c) A set that contains all leaves of T (b) or none of them (c) cannot cut only one edge of T .

Lemma 23. For any monotonic moat growing algorithm and any tree T with leaves $S' \subseteq V$, we have

$$UM(T) \geq c(T) - \sum_{\substack{S \subseteq V \\ S \odot S'}} y_S.$$

Proof. We first show that

$$SC(T) \leq \sum_{\substack{S \subseteq V \\ S \odot S'}} y_S.$$

By Lemma 22, for any active set $S \in \text{Act}S_F$, we have $S \odot S'$. Since $SC(T)$ is the total portion of T colored by active sets in $\text{Act}S_F$, and each such active set colors exactly one edge of T , we can conclude the above inequality.

Now, using Corollary 19, we have

$$c(T) = UM(T) + SC(T) \leq UM(T) + \sum_{\substack{S \subseteq V \\ S \odot S'}} y_S,$$

which completes the proof. \square

The next lemma shows that if an active set cuts exactly one edge of a tree in the optimal solution, then removing that edge only disconnects those pairs that are cut by the active set itself. We use this property for two purposes. First, we show that if an active set does not contain any unsatisfied vertices, then it cannot cut exactly one edge of the optimal solution; otherwise, we could remove the edge it cuts and obtain a more optimal solution that satisfies all demands, leading to a contradiction. Second, we use this lemma to argue that if we remove edges cut by certain special single-edge active sets, then the only unsatisfied pairs are those cut by the same active set. This is useful because if we can find an alternative structure to satisfy those pairs, we can remove the corresponding edges from the optimal solution and obtain a lower-cost solution for the remaining pairs.

Lemma 24. Let $S \subseteq V$ be a subset of vertices such that, in a connected component of the optimal solution C (with tree T_C), S cuts exactly one edge of that component, denoted by e . That is, $\delta(S) \cap T_C = \{e\}$. Then, removing e only violates pair demands for those pairs that have one endpoint in $\text{UNSATISFIED}(S)$.

Proof. Similar to the proof of Lemma 22, removing e splits T_C into two subtrees, one entirely inside S and the other entirely outside. This means a pair $\{v, \text{PAIR}_v\}$ is disconnected by the removal of e only if v and PAIR_v lie on opposite sides of the cut. Without loss of generality, assume $v \in S$. Then $\text{PAIR}_v \notin S$, and so $v \in \text{UNSATISFIED}(S)$, completing the proof. \square

The next lemma provides a lower bound on the cost of the optimal solution based on the total portion of it colored by active sets.

Lemma 25. For any monotonic moat growing algorithm and any forest F , the cost of F can be bounded in terms of its coloring by the active sets of the algorithm as follows:

$$c(F) \geq \sum_{S \subseteq V} |\delta(S) \cap F| \cdot y_S.$$

Proof. For an active set $S \subseteq V$, the edges of F it colors are precisely those in $\delta(S) \cap F$. If S grows for a duration of y_S , then it contributes $|\delta(S) \cap F| \cdot y_S$ units to the total colored portion of F . Since each portion of F can be colored at most once, summing over all active sets gives a lower bound on $c(F)$, completing the proof. \square

Assignment. Another important concept that we introduce is *assignment*. The idea behind assignment is to allocate the growth of an active set to one or more of its vertices. This approach allows us to bound the total growth of all active sets using the assigned values. An assignment specifies how the growth of each active set is distributed among its vertices. Specifically, consider a particular moment during the execution of a monotonic moat growing algorithm. At this moment, each active set S assigns a fraction (possibly zero) of its growth to each of its vertices. It is worth noting that these fractions lie between 0 and 1, but the total fraction assigned by an active set is not necessarily bounded by 1.

Definition 26. We define $r : 2^V \times V \rightarrow \mathbb{R}_{\geq 0}$ as an assignment where $r_{S,v}$ represents the portion of growth assigned by the set S to a vertex $v \in \text{SUPERACTIVE}(S)$. We abuse the notation and write r_v to represent the total growth assigned to vertex v . Consequently,

$$r_v = \sum_{S \subseteq V} r_{S,v}.$$

We further use the same notation and define $r_{\max}(S)$ for a subset of vertices S to denote the maximum value of r_v for any $v \in S$, i.e.,

$$r_{\max}(S) = \max_{v \in S} r_v.$$

Generally, we have two types of assignments, defined as follows:

Definition 27 (Exclusive Assignment). An assignment is *exclusive* if the total fractional assignment across all vertices at any given moment for an active set is at most 1.

Definition 28 (Prefix-Time Assignment). A prefix-time assignment is an assignment satisfying the following conditions:

- At each moment in time, an active set assigns to each vertex a growth fraction of either 0 or 1.
- If, at some time τ , an active set assigns a growth fraction of 1 to a vertex v , then at any earlier time, the active set containing v must also assign its entire growth fraction of 1 to v .

The exclusive assignment roughly matches the total assigned value with the total growth of active sets, allowing us to bound the total assigned value instead of the total growth. It is used in the analysis throughout all subsequent sections and methods. On the other hand, the prefix-time assignment better captures temporal aspects and provides an upper bound on nearly the total growth of active sets, although this bound can be significantly weak. We define two types of prefix-time assignment, whose sole purpose is to analyze the autarkic pairs approach in Section 8. The first prefix-time assignment will be formally introduced in the next subsection.

4.4 Legacy Execution

Here, we discuss the execution of `LEGACYMOATGROWING` at Line 2 of our main algorithm. We refer to this execution as *Legacy Execution*, and for any variable x associated with this execution, we denote it as x^- . Let us formally define Legacy Execution and then provide some of its properties that will be useful in the subsequent sections.

Definition 29 (Legacy Execution). We define Legacy Execution as the execution of `LEGACYMOATGROWING` in Line 2 of Algorithm 2 on the given problem input. In this context, we denote:

- y_S^- as the total growth of set $S \subseteq V$,
- $ActS_\tau^-$ as the collection of active sets at time τ during this execution, and
- t^- as the fingerprint returned by this execution, which records the time at which each vertex connects to its pair (see Line 21).

Note that, given Lemma 7, t^- is a fingerprint for this execution. According to Lemma 8, executing `SHADOWMOATGROWING` on t^- results in an equivalent moat growing process. Therefore, all these variables can be considered as belonging to such a run of `SHADOWMOATGROWING` as well.

Next, we prove that monotonic moat growing algorithms whose fingerprint has a higher value for every vertex compared to their t^- satisfy all demand pairs. This property is crucial for demonstrating that the various solutions introduced in the remainder of the paper are feasible for the problem.

Lemma 30. Let t be a fingerprint of a monotonic moat growing algorithm on a graph G , where t is larger than t^- . Then, the forest produced by the execution with fingerprint t satisfies all demand pairs.

Proof. First note that we can assume a Shadow Moat Growing algorithm has been run on t and given Lemma 8, it is equivalent to the given monotonic moat growing algorithm.

Consider any vertex v . By Definition 29, t_v^- is the moment when v and its pair PAIR_v reach each other in Legacy Execution, and both are active until this moment. Consequently, at this moment, there exists an active set S such that S contains both v and PAIR_v (they may become deactivated immediately afterward).

According to Lemma 13, at any moment until t_v^- , v should be in an active set in Shadow Moat Growing execution on t , and at moment t_v^- , there exists an active set S' such that $S \subseteq S'$. Therefore, v and its pair reach each other while both are in active set until then.

As a result, even after removing unnecessary edges, they remain in the same component as no inactive set cuts their path. \square

Now, based on the fingerprint t^- obtained from Legacy Execution, we define the notion of the *base phase* for any monotonic moat growing process as follows.

Definition 31 (Base Phase). In an execution of a monotonic moat growing process, at moment τ , a vertex is said to be in the base phase if

$$\tau < t_v^-.$$

Similarly, an active set S is in the base phase if it contains a vertex in the base phase. Additionally, for an active set S and a given time τ , we define $\text{BASE}(S, \tau)$ as the set of vertices in the base phase.

Intuitively, $\text{BASE}(S, \tau)$ is the set of vertices in S that have not yet reached their initial fingerprint, which indicates the time they reach their pair in Legacy Execution. We use this notation to define different

assignments for different monotonic moat growing algorithms. Now, a basic observation regarding the base phase is the following.

Lemma 32. Let τ be any moment during the execution of a monotonic moat growing process. For any pair of vertex subsets S and S' such that $S \subseteq S'$, it holds that

$$\text{BASE}(S, \tau) \subseteq \text{BASE}(S', \tau),$$

Proof. Suppose $v \in \text{BASE}(S, \tau)$. By Definition 31, we have $\tau < t_v^-$. Since $v \in S$ and $S \subseteq S'$, it follows that $v \in S'$. Therefore, by the same definition, $v \in \text{BASE}(S', \tau)$. \square

Next, we introduce an assignment that maps the growth occurring in Legacy Execution to individual vertices, with respect to the optimal solution. Similar assignments will be defined for other executions in our algorithms, allowing us to compare them and ultimately derive the desired approximation factor.

To make this assignment meaningful and consistent, we introduce a notion of *priority*—a way to rank vertices based on when their demands are satisfied in Legacy Execution. Priority helps determine which vertices in a subset are most influential in driving further growth in the algorithm.

Definition 33 (Priority). The priority of a vertex v is determined by the time at which its associated demand is satisfied during Legacy Execution. Vertices satisfied later (larger t^-) are assigned higher priority.

In the event of a tie, priorities are resolved using a fixed total ordering over all vertices. This ordering is constructed by first ordering all demand pairs, and then, for each pair involving distinct vertices, arbitrarily ordering the two vertices within the pair.

We denote the priority of a vertex v by priority_v , and write $\text{priority}_v > \text{priority}_u$ to indicate that v has a higher priority than u .

Immediately, we observe the following relationship between priority and the fingerprint of Legacy Execution.

Corollary 34. For any pair of vertices v and u ,

$$t_v^- > t_u^- \Rightarrow \text{priority}_v > \text{priority}_u,$$

and

$$\text{priority}_v > \text{priority}_u \Rightarrow t_v^- \geq t_u^-.$$

Here, we define representatives for any subset of vertices based on priority, selecting vertices with the highest priority from each connected component within the subset. This notation is also crucial for defining assignments.

Definition 35 (Representatives). For a subset S of vertices, the set of *representatives* of S consists of the vertices in S that have the highest priority within their respective connected components of the optimal solution. Formally,

$$\text{REPS}(S) = \left\{ v \in S \mid v = \operatorname{argmax}_{u \in S \cap \text{COMP}(v)} \text{priority}_u \right\},$$

where $\text{COMP}(v)$ denotes the connected component of v in the optimal solution.

The following lemma can easily be concluded from the above definition.

Lemma 36. In a moat growing algorithm, for any time τ , any subset of vertices $S \subseteq V$, and any connected component of the optimal solution C , at most one vertex of C is in $\text{REPS}(\text{BASE}(S, \tau))$, which is the vertex with maximum priority. More precisely,

$$\begin{aligned} \text{If } \text{BASE}(S, \tau) \cap C = \emptyset, \text{ then} & \quad \text{REPS}(\text{BASE}(S, \tau)) \cap C = \emptyset \\ \text{Otherwise,} & \quad \text{REPS}(\text{BASE}(S, \tau)) \cap C = \{\text{argmax}_{v \in S \cap C} \text{priority}_v\} \end{aligned}$$

Proof. By Definition 35, $\text{REPS}(\text{BASE}(S, \tau))$ can contain at most one vertex from C . If $t_v^- \leq \tau$ for the vertex $v = \text{argmax}_{u \in S \cap C} \text{priority}_u$, which also has the maximum value of t^- among vertices in $S \cap C$ (based on Corollary 34), then $\text{BASE}(S, \tau) \cap C = \emptyset$, so the statement holds. Otherwise, since $t_v^- > \tau$, v is in $\text{BASE}(S, \tau)$ and, as it has the maximum priority among vertices in $\text{BASE}(S, \tau) \cap C$, the claim follows. \square

Next, we prove a consistency property of representatives throughout the algorithm.

Lemma 37. Let S be a subset of vertices. If $v \in \text{REPS}(S)$, then for any subset $S' \subseteq S$ with $v \in S'$, it follows that

$$v \in \text{REPS}(S').$$

Proof. Suppose, for the sake of contradiction, that $v \notin \text{REPS}(S')$. Then there exists a vertex $u \in S' \cap \text{COMP}(v)$ such that $\text{priority}_u \geq \text{priority}_v$. Since $S' \subseteq S$, we also have $u \in S$, which contradicts the assumption that $v \in \text{REPS}(S)$. \square

Next, we demonstrate that REPS exhibits cardinality monotonicity.

Lemma 38. Let $S \subseteq V$ be a subset of vertices and $S' \subseteq S$. Then,

$$|\text{REPS}(S')| \leq |\text{REPS}(S)|.$$

Proof. For any set S , $\text{REPS}(S)$ contains exactly one vertex from each connected component of the optimal solution that intersects with S . Consequently, a subset $S' \subseteq S$ can only contain vertices from a subset of the connected components that S intersects with, ensuring that $\text{REPS}(S')$ cannot have more elements than $\text{REPS}(S)$. \square

Now we can define our first assignment, r^- , which corresponds to Legacy Execution, using the above definitions.

Definition 39. Consider the moment τ of Legacy Execution. For each active set $S \in \text{Act}S_\tau^-$ in the *base phase*, we assign the growth of this moment of S to all vertices in $\text{REPS}(\text{BASE}(S, \tau))$ with fraction 1. We denote the total growth assigned to vertex v by set S as $r_{S,v}^-$, and the total accumulated growth assigned to v (over all sets) as r_v^- .

Note that for every active set S at time τ , we have $\text{BASE}(S, \tau) \neq \emptyset$, otherwise, all vertices in S would already be satisfied, and S would become inactive. This implies that y_s^- is assigned to at least one vertex, leading to the following corollary.

Corollary 40. The total assigned value of r^- is an upper bound for the total growth of active sets in Legacy Execution. That means,

$$\sum_{S \subseteq V} y_s^- \leq \sum_{v \in V} r_v^-.$$

Another observation for this assignment shows that the assignment r^- to a vertex only happens when the vertex is unsatisfied.

Lemma 41. The growth of a subset of vertices $S \subseteq V$ can be assigned to r^- of vertex v only if $v \in S$ and $\text{PAIR}_v \notin S$.

Proof. Let the growth of the active set S in Legacy Execution at moment τ be assigned to v . Since $\text{REPS}(\text{BASE}(S, \tau)) \subseteq S$ for any moment τ , the growth of S can only be assigned to vertices in S , meaning we must have $v \in S$. Additionally, $v \in \text{REPS}(\text{BASE}(S, \tau)) \subseteq \text{BASE}(S, \tau)$ if it is in the base phase, meaning $\tau < t_v^-$. Since v and PAIR_v reach each other at time t_v^- , we have $\text{PAIR}_v \notin S$. \square

As a reminder, we have introduced two general frameworks for assignments. We can now show that r^- satisfies the conditions of a prefix-time assignment as defined in Definition 28.

Lemma 42. r^- is a prefix-time assignment.

Proof. Consider a moment τ when an active set S exists and assigns its growth to v . By Definition 39,

$$v \in \text{REPS}(\text{BASE}(S, \tau)).$$

This implies $\tau < t_v^-$, meaning that v was in active sets until this moment.

For any earlier time $\tau' < \tau$, there exists an active set S' that contains v . Given Corollary 11, we know that $S' \subseteq S$. By way of contradiction, assume $v \notin \text{REPS}(\text{BASE}(S', \tau'))$. This implies the existence of some $u \in \text{BASE}(S', \tau') \cap \text{COMP}(v)$ such that $\text{priority}_u > \text{priority}_v$, or equivalently, $t_u^- \geq t_v^-$ (by Corollary 34). Furthermore, since

- $\tau < t_v^- \leq t_u^-$, and
- v and u remain in the same active sets after τ' ,

it follows that $u \in \text{BASE}(S, \tau) \cap \text{COMP}(v)$, contradicting the fact that $v \in \text{REPS}(\text{BASE}(S, \tau))$.

Finally, by Definition 39, an active set assigns a fraction of either 0 or 1 to each vertex, satisfying the two criteria of prefix-time assignment (Definition 28). \square

As a consequence of the above lemma, we can prove that all vertices $v \in C \in \text{OPT}$ are connected to their pair at moment $r_{\max}^-(C)$ in Legacy Execution.

Lemma 43. For any connected component of the optimal solution C and any vertex $v \in C$, vertex v and PAIR_v are in the same connected component at time $r_{\max}^-(C)$ in Legacy Execution.

Proof. Since for every vertex v , t_v^- is the time that v and PAIR_v connect together, we want to prove that $t_v^- \leq r_{\max}^-(C)$ for any vertex $v \in C$.

Let vertex $u \in C$ have the highest priority among vertices in C , meaning $u = \text{argmax}_{u' \in C} \text{priority}_{u'}$. Given Lemma 36, until u is in the base phase, any active set containing u should assign its growth to u . Therefore, since based on Lemma 42 we know r^- is a prefix-time assignment, we have

$$t_u^- \leq r_{\max}^-.$$

Additionally, using Corollary 34 we know that

$$u = \text{argmax}_{u' \in C} t_{u'}^-.$$

Therefore, for any vertex $v \in C$, we have

$$t_v^- \leq t_u^- \leq r_u^- \leq r_{\max}^-(C),$$

which completes the proof. \square

The next lemma gives a lower bound for the cost of any connected component of the optimal solution based on the total assigned value to its vertices in r^- .

Lemma 44. Let C be a connected component in the optimal solution and T_C be the tree of the optimal solution spanning the vertices in C . We have,

$$\sum_{v \in C} r_v^- \leq c(T_C).$$

Proof. Recall from Definition 39, by Lemma 36, each moment of growth of active set S in Legacy Execution is assigned to r^- of at most one vertex in C . Therefore,

$$\sum_{v \in C} r_{S,v}^- \leq y_S^-. \quad (1)$$

Additionally, based on Lemma 41, the growth of S can be assigned to $v \in C$ only if $v \in S$ and $\text{PAIR}_v \notin S$. Since $\text{PAIR}_v \in C$, we can conclude that $S \odot C$. Therefore, we can write a refined version of Definition 26 as follows:

$$\begin{aligned} \sum_{v \in C} r_v^- &= \sum_{v \in C} \sum_{\substack{S \subseteq V \\ S \odot C}} r_{S,v}^- \\ &= \sum_{\substack{S \subseteq V \\ S \odot C}} \sum_{v \in C} r_{S,v}^- \\ &\leq \sum_{\substack{S \subseteq V \\ S \odot C}} y_S^- && \text{(Equation 1)} \\ &\leq \sum_{\substack{S \subseteq V \\ S \odot C}} |\delta(S) \cap T_C| \cdot y_S^- && \text{(If } S \odot C \text{ then } |\delta(S) \cap T_C| \geq 1) \\ &\leq \sum_{S \subseteq V} |\delta(S) \cap T_C| \cdot y_S^- \\ &\leq c(T_C). && \text{(Lemma 25)} \end{aligned}$$

\square

The following lemma, giving a lower bound for the cost of the optimal solution, can easily be derived from the above lemma.

Lemma 45. The total growth of active sets in Legacy Execution is a lower bound for the cost of the optimal solution.

$$\sum_{S \subseteq V} y_S^- \leq c(\text{OPT}).$$

Proof. The proof can be derived by giving a lower bound for the cost of each connected component of the optimal solution as follows:

$$\begin{aligned}
\sum_{S \subseteq V} y_S^- &\leq \sum_{v \in V} r_v^- && \text{(Corollary 40)} \\
&= \sum_{C \in \text{OPT}} \sum_{v \in C} r_v^- \\
&\leq \sum_{C \in \text{OPT}} c(T_C) && \text{(Lemma 44)} \\
&= c(\text{OPT}).
\end{aligned}$$

□

5 Local Search

In this section, we introduce a novel *local search* approach that incrementally improves a given solution by modifying an initial monotonic moat growing algorithm. The key idea is to delay the deactivation of certain active sets, allowing them to remain active for longer. This can reduce the total growth of active sets by enabling them to connect and grow together more quickly. As a result, the upper bound on the solution becomes smaller, and the actual cost may also decrease.

More precisely, we select a vertex v and increase its fingerprint value t_v , an operation we call a *boost action*. We then check whether the moat growing algorithm corresponding to the new fingerprint yields a smaller total growth of active sets. If it does, we apply the boost and repeat the process until no further improvement is possible. This method not only reduces the upper bound and potentially lowers the cost, but also ensures useful structural properties in the resulting moat growing algorithm.

In the following subsections, we formally introduce boost actions, describe how the local search is carried out using a modified version of SHADOWMOATGROWING, analyze the resulting solution, and present key properties that are useful for the remainder of the paper.

5.1 Algorithm

Here, we introduce some necessary preliminary notation that follows the definition of our local search algorithm. We also present a modified version of Shadow Moat Growing algorithm, which is used by the local search.

From Section 4.2, we know that in SHADOWMOATGROWING, the input consists of a graph G and a fingerprint t , where each t_v specifies a time until which vertex v is required to grow. Since we consider increasing some of these values, we define a *boosted instance* as follows:

Definition 46 (Boosted Instance). In a boosted instance $I = (G, t, t^*)$, G represents the graph, t denotes the base fingerprint, and t^* , referred to as the *boosted fingerprint*, represents the required growth after boosting, where $t_v^* \geq t_v$ for all $v \in V$.

Initially, we can convert an input instance (G, t) of SHADOWMOATGROWING to a boosted input instance $I = (G, t, t^*)$ by letting $t_v^* = t_v$ for all $v \in V$. Now, we can apply boost actions to this instance to adjust the required growth time. We formally define *boost action* below:

Definition 47 (Boost). For a boosted instance $I = (G, t, t^*)$, a *boost action* $B = (v, \tau)$, where $\tau \geq t_v^*$, applied to a vertex v updates t_v^* to τ . This results in a new boosted instance where the value of t_v^* is replaced by τ . We denote this updated instance by $\text{WITHBOOST}(I, B)$.

To evaluate whether a boost action is beneficial, we first need to modify `SHADOWMOATGROWING`. Therefore, we introduce `BOOSTEDMOATGROWING`, which is essentially the same algorithm as `SHADOWMOATGROWING` described in Section 4.2, as both grow moats until they reach the times specified in the fingerprint. The difference lies in additional measurements used to assess whether a boost action is beneficial. Additionally, `BOOSTEDMOATGROWING` maintains a connected component S active until no vertex v within S satisfies $t_v^* > \tau$, where τ denotes the current moment in the algorithm.

We define two parameters, y_{base} and y_{boost} , along with a function $y^b : 2^V \rightarrow \mathbb{R}_{\geq 0}$. The value y_{base} denotes the total growth required to reach t_v for all $v \in V$, and y_{boost} represents the additional growth needed to reach t_v^* . More precisely, when an active set grows at time τ , its growth contributes to y_{base} if it contains at least one vertex v with $t_v > \tau$. If instead all vertices in the set satisfy $t_v^* > \tau \geq t_v$, the growth contributes to y_{boost} . It follows that the total growth of active sets is $y_{base} + y_{boost}$. We use y_S^b to denote the portion of growth from an active set S that contributes to y_{base} .

In Algorithm 5, similar to Algorithm 4, we continue expanding active sets until the time constraints are met. The main difference is that each vertex must remain active until t_v^* instead of t_v (see Line 29). We also split the total growth $\sum_{S \subseteq V} y_S$ into two quantities, y_{base} and y_{boost} , and compute the function y^b during execution (see Lines 15–19).

Now, we measure the benefit of a boost action against the additional cost it introduces. This is done by comparing the results of `BOOSTEDMOATGROWING` on two instances, the original boosted instance I and the instance $\text{WITHBOOST}(I, B)$ obtained after applying a boost action B . Based on this comparison, we define `WIN` and `LOSS` for a boost action as follows.

Definition 48 (Win, Loss, and Valuable Boost). For an instance I , let the boost action $B = (v, \tau)$ produce a new instance $I' = \text{WITHBOOST}(I, B)$. Suppose the procedure `BOOSTEDMOATGROWING` returns (y_{base}, y_{boost}) on I and (y'_{base}, y'_{boost}) on I' .

The *win* of the boost action is the total decrease in y_{base} , defined by

$$\text{WIN}(I, B) = y_{base} - y'_{base}.$$

The *loss* is the total increase in y_{boost} caused by applying B , given by

$$\text{LOSS}(I, B) = y'_{boost} - y_{boost}.$$

We say that the boost action B on instance I is *valuable* if

$$\text{WIN}(I, B) \geq (1 + \beta) \cdot \text{LOSS}(I, B),$$

where $\beta \in (0, 1)$ is a fixed constant.

It is important to note that we will present the appropriate value of β in Section 9, although this value is passed to `LOCALSEARCH` whenever we use it in our algorithm.

Algorithm 5 Boosted Moat Growing

Input: A boosted instance $I = (G, t, t^*)$ including a graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, a function $t : V \rightarrow \mathbb{R}_{\geq 0}$ specifying the original fingerprint, and a function $t^* : V \rightarrow \mathbb{R}_{\geq 0}$ specifying the boosted fingerprint.

Output: F is the resulting forest of fingerprint t^* , y_{base} is the amount of growth needed to satisfy fingerprint t , y_{boost} is the additional amount of growth to reach fingerprint t^* , $y^b : 2^V \rightarrow \mathbb{R}_{\geq 0}$ is a function that maps each set S to the amount of its growth counted toward y_{base} .

```
1: procedure BOOSTEDMOATGROWING( $I = (G, t, t^*)$ )
2:    $\tau \leftarrow 0$ 
3:    $F \leftarrow \emptyset$ 
4:    $FC \leftarrow \{\{v\} \mid v \in V\}$ 
5:    $ActS \leftarrow \{\{v\} \mid v \in V, t_v > 0\}$ 
6:    $DS \leftarrow \{\{v\} \mid v \in V, t_v = 0\}$ 
7:   Implicitly set  $y_S \leftarrow 0$  for  $S \subseteq V$ 
8:   Implicitly set  $y_S^b \leftarrow 0$  for  $S \subseteq V$ 
9:   while  $ActS \neq \emptyset$  do ▷ While there exists an active set
10:     $\Delta_e \leftarrow \min_{e=uv \in E} \frac{c_e - \sum_{S \ni e} y_S}{|\{S_u, S_v\} \cap ActS|}$ , where  $u \in S_u \in FC, v \in S_v \in FC$ , and  $S_u \neq S_v$ 
11:     $\Delta_t \leftarrow \min_{v \in V, t_v > \tau} (t_v - \tau)$ 
12:     $\Delta_{t^*} \leftarrow \min_{v \in V, t_v^* > \tau} (t_v^* - \tau)$ 
13:     $\Delta \leftarrow \min(\Delta_e, \Delta_t, \Delta_{t^*})$ 
14:    for  $S \in ActS$  do
15:      if  $v \in S$  exists such that  $t_v > \tau$  then
16:         $y_{base} \leftarrow y_{base} + \Delta$ 
17:         $y_S^b \leftarrow y_S^b + \Delta$ 
18:      else
19:         $y_{boost} \leftarrow y_{boost} + \Delta$ 
20:         $y_S \leftarrow y_S + \Delta$ 
21:       $\tau \leftarrow \tau + \Delta$ 
22:    for  $e \in E$  do
23:      Let  $S_v, S_u \in FC$  be sets that contain each endpoint of  $e$ 
24:      if  $\sum_{S: e \in \delta(S)} y_S = c_e$  and  $S_v \neq S_u$  then ▷ Edge  $(v, u)$  become fully colored
25:         $F \leftarrow F \cup \{e\}$ 
26:         $FC \leftarrow (FC \setminus \{S_v, S_u\}) \cup \{S_v \cup S_u\}$ 
27:         $ActS \leftarrow (ActS \setminus \{S_v, S_u\}) \cup \{S_v \cup S_u\}$ 
28:      for  $S \in ActS$  do
29:        if  $t_v^* \leq \tau$  for all  $v \in S$  then ▷  $S$  become inactive
30:           $ActS \leftarrow ActS \setminus \{S\}$ 
31:           $DS \leftarrow DS \cup \{S\}$ 
32:      while  $S \in DS$  exists such that  $|\delta(S) \cap F| = 1$  do ▷ Remove unnecessary edges
33:         $F \leftarrow F \setminus \delta(S)$ 
34:    return  $F, y_{base}, y_{boost}, y^b$ 
```

Using valuable boost actions, we design our local search algorithm. We start with a boosted instance where $t_v^* = t_v$ for all $v \in V$. In each iteration, we identify a valuable boost action and apply it to update the instance. This process is repeated until no further valuable boost actions remain.

Boost Actions Space. One natural question that arises is: given the infinite number of possible boost actions, how can we determine whether a valuable boost exists? To address this, we define a polynomially bounded set of boost actions (with size polynomial in the input), and later show that this restricted subspace is sufficient for the analysis in this paper (see Lemma 88).

Our method proceeds as follows. We consider all vertices as potential candidates for the boost action. For a given vertex v , we identify at most n specific time points such that checking only these is sufficient to determine whether a valuable boost action exists involving v .

To find these time points, we simulate Shadow Moat Growing algorithm with a modified parameter t^* , where we set $t_v^* = \infty$. Here, ∞ is chosen to be large enough so that the final moat grown around v includes all vertices while v is active. During this run, each time the active set containing v merges with another active set, we record the current time τ , but only if τ exceeds the previous value of t_v^* . Note that we only capture times when v becomes connected to another active set, not to a component that has already become inactive.

Let the resulting sequence of such time points be $\tau_1, \tau_2, \dots, \tau_k$. We then define the candidate boost actions for vertex v as $B \in \{(v, \tau_i)\}_{i=1}^k$. Since each recorded time corresponds to a merge event between two sets, and there are at most n such events, we have $k \leq n$. Repeating this process for all n vertices yields at most n^2 candidate boost actions to check.

Corollary 49. A valuable boost action, if one exists in the defined subspace, can be found in polynomial time or confirmed not to exist.

At the end of this process, we obtain a boosted instance $I = (G, t, t^*)$ in which no further valuable boost actions within the defined space can be applied. The final output includes the boosted fingerprint and the forest produced by running BOOSTEDMOATGROWING on it. This approach is summarized in Algorithm 6. We refer to the execution of BOOSTEDMOATGROWING in Line 7 as the *corresponding moat growing* in the local search, a term that will be further discussed.

Algorithm 6 Local Search

Input: Graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, function $t : V \rightarrow \mathbb{R}_{\geq 0}$ specifying a fingerprint, and parameter $0 < \beta < 1$.

Output: A forest F , a boosted fingerprint t^* , and a function $y : 2^V \rightarrow \mathbb{R}_{\geq 0}$ specifying the growth of active sets, all with respect to the corresponding moat growing algorithm.

```

1: procedure LOCALSEARCH( $G, t, \beta$ )
2:    $t^* \leftarrow t$ 
3:   Initialize  $I \leftarrow (G, t, t^*)$ , representing the instance before any boost action is applied.
4:   while  $B = (v, \tau)$  exists such that  $W_{\text{IN}}(I, B) \geq (1 + \beta) \cdot \text{Loss}(I, B)$  do
5:      $t_v^* \leftarrow \tau$ 
6:      $I \leftarrow \text{WITHBOOST}(I, B)$ 
7:    $F, y_{\text{base}}, y_{\text{boost}}, y^b \leftarrow \text{BOOSTEDMOATGROWING}(G, t, t^*)$ 
8:   return  $F, t^*, y^b$ 

```

Now, in LOCALSEARCH, we apply multiple boost actions. In our analysis, we are interested in the total win and loss across all these actions, leading to the following definition.

Definition 50 (Total Win and Loss). For a LOCALSEARCH on instance I , we define *win* and *loss* as the total W_{IN} and Loss , respectively, resulting from all boost actions applied during the LOCALSEARCH to the initial instance. Consequently, we have

$$\text{win} \geq (1 + \beta) \cdot \text{loss}.$$

5.2 Analysis of Local Search

In this section, we present key properties and analyses concerning boost actions, valuable boost actions, and the LOCALSEARCH procedure.

We first show that y_{base} and y_{boost} together account for the total growth of active sets.

Lemma 51. For an instance I , at the end of running BOOSTEDMOATGROWING on I , we have

$$\sum_{S \subseteq V} y_S = y_{base} + y_{boost}.$$

Proof. Based on the Lines 14- 20 of Algorithm 5, anytime we are adding Δ to one y_S , we add the same amount to one of y_{base} or y_{boost} , so the equality holds. \square

Since each boost action produces a larger fingerprint, we can derive the following two corollaries from Lemma 13.

Corollary 52. For any instance I and boost action B , the active sets at any moment τ during the execution of BOOSTEDMOATGROWING on I form a *refinement* of the active sets at the same moment during the execution of BOOSTEDMOATGROWING on WITHBOOST(I, B). Similarly, the connected components at the same moment in the execution on I form a refinement of the connected components in the other execution.

Corollary 53. For any instance (G, t, β) , let t^* be the boosted fingerprint produced by LOCALSEARCH on this instance. Then, at any moment τ , the active sets in the execution of SHADOWMOATGROWING on (G, t) form a *refinement* of the active sets at the same moment in the execution on (G, t^*) . Similarly, the connected components at the same moment in the execution on (G, t) form a refinement of those in the execution on (G, t^*) .

Boost Action Properties. We now focus on the properties of a boost action. Using Corollary 52, we derive bounds for WIN and Loss for any boost action. We begin by showing that WIN is always nonnegative.

Lemma 54. For any instance $I = (G, t, t^*)$ and boost action B , we have

$$W_{IN}(I, B) \geq 0.$$

Proof. By Definition 48, let $I' = WITHBOOST(I, B)$, and suppose that running BOOSTEDMOATGROWING on I yields (y_{base}, y_{boost}) while running it on I' yields (y'_{base}, y'_{boost}) . Since $W_{IN}(I, B) = y_{base} - y'_{base}$, it suffices to show that $y_{base} \geq y'_{base}$.

Consider an active set S' at moment τ in the execution on I' that contributes to y'_{base} . By definition, this means there exists a vertex $v \in S'$ such that $\tau < t_v$. At the same moment in the execution on I , vertex v must lie in a connected component S , and since $\tau < t_v$, its growth also contributes to y_{base} . By Corollary 52, we have $S \subseteq S'$. Hence, for every active set contributing to y'_{base} in I' , there exists a distinct subset in I that contributes to y_{base} . Since the active sets in the original execution are disjoint, their corresponding subsets in the boosted execution are also disjoint and distinct.

Therefore, each contribution to y'_{base} is matched by a corresponding (distinct) contribution to y_{base} , implying $y_{base} \geq y'_{base}$. This completes the proof. \square

We also use Corollary 52 to establish an upper bound on Loss.

Lemma 55. For any instance $I = (G, t, t^*)$ and boost action $B = (v, \tau)$, we have

$$\text{Loss}(I, B) \leq \tau.$$

Proof. By Definition 48, let $I' = \text{WITHBOOST}(I, B)$, and suppose that running `BOOSTEDMOATGROWING` on I yields (y_{base}, y_{boost}) , while on I' it yields (y'_{base}, y'_{boost}) . It suffices to show that

$$y'_{boost} - y_{boost} \leq \tau.$$

Consider any moment τ' during the execution on I' , and let S' be an active set at τ' that contributes to y'_{boost} . Suppose there exists a vertex $u \neq v$ in S' such that $t_u \leq \tau' < t_u^*$. Then, in the execution on I , the same vertex u is in an active set S at τ' .

By Corollary 52, we have $S \subseteq S'$, which ensures that no vertex w in S satisfies $\tau' < t_w$. Thus, S contributes to y_{boost} .

The only case where an active set contributes to y'_{boost} at time τ' but a corresponding subset does not contribute to y_{boost} is when v is the only vertex satisfying $t_v \leq \tau' < \tau$. That is, the moment lies between the original and updated boosted fingerprint values for v . The total growth over such moments is at most τ .

In summary, since every contribution to y'_{boost} either:

- Contains a subset that contributes to y_{boost} , or
- Is solely due to v during its extension period, contributing at most τ ,

we conclude that

$$y'_{boost} \leq y_{boost} + \tau,$$

as desired. □

Next, we state a property which shows that for any active set after a boost action, if it is independent of that boost action, then there exists a subset of it that was active before the boost action.

Lemma 56. Consider any instance I and boost $B = (v, \tau)$. Then, for any moment τ_0 and any active set S at this moment of the execution of `BOOSTEDMOATGROWING` on `WITHBOOST`(I, B), if either S does not include v or $\tau \leq \tau_0$, then there exists an active set S' at moment τ_0 of the execution of `BOOSTEDMOATGROWING` on I such that $S' \subseteq S$.

Proof. Let I' be `WITHBOOST`(I, B). We denote t as the fingerprint of I . We investigate two cases for S in the execution on I' at moment τ_0 .

First, assume $v \notin S$. In this case, a vertex $u \in S$ must exist where $t_u > \tau_0$. Consider the same moment in the execution on I . Since $t_u > \tau_0$, u must belong to an active set S' . As both S and S' have u , by Corollary 52, $S' \subseteq S$. Second, assume $v \in S$ and $\tau_0 \geq \tau$. Since $\tau_0 \geq \tau$, S is not active due to the boost B . Therefore, there again must exist a vertex u such that $t_u < \tau_0$. Similar to the first case, we can prove that active set S' exists such that $S' \subseteq S$. Thus, the proof is complete. □

Valuable Boost Action Properties. We now focus on valuable boost actions. To begin, we observe that for any valuable boost action, `WIN` is greater than or equal to `Loss`.

Lemma 57. For any instance I and valuable boost action B , we have

$$\text{WIN}(I, B) - \text{Loss}(I, B) \geq 0.$$

Proof. We consider two cases based on the sign of $\text{Loss}(I, B)$. If $\text{Loss}(I, B) < 0$, According to Lemma 54 we can conclude that $\text{WIN}(I, B) - \text{Loss}(I, B) \geq 0$. Otherwise, based on Definition 48 for a valuable boost action we have

$$\begin{aligned} \text{WIN}(I, B) - \text{Loss}(I, B) &\geq \text{WIN}(I, B) - (1 + \beta)\text{Loss}(I, B) && (\beta\text{Loss}(I, B) > 0) \\ &\geq 0. && (\text{Definition 48}) \end{aligned}$$

□

Next, we show how WIN and Loss capture the relationship between the total growth of active sets before and after a valuable boost action.

Lemma 58. For an instance I , let $I' = \text{WITHBOOST}(I, B)$ be the result of applying a *valuable* boost action B . Let y_S and y'_S denote the growth duration of an active set $S \subseteq V$ in the executions of $\text{BOOSTEDMOATGROWING}$ on I and I' , respectively. Then,

$$\sum_{S \subseteq V} y'_S = \sum_{S \subseteq V} y_S - \text{WIN}(I, B) + \text{Loss}(I, B).$$

Proof. Let (y_{base}, y_{boost}) and (y'_{base}, y'_{boost}) be the output of $\text{BOOSTEDMOATGROWING}$ on I and I' , respectively. By applying Lemma 51 and Definition 48, we have

$$\begin{aligned} \sum_{S \subseteq V} y'_S &= y'_{base} + y'_{boost} && (\text{Lemma 51}) \\ &= y'_{base} + y'_{boost} - \sum_{S \subseteq V} y_S + \sum_{S \subseteq V} y_S \\ &= y'_{base} + y'_{boost} - y_{base} - y_{boost} + \sum_{S \subseteq V} y_S && (\text{Lemma 51}) \\ &= \sum_{S \subseteq V} y_S - \text{WIN}(I, B) + \text{Loss}(I, B). && (\text{Definition 48}) \end{aligned}$$

□

Consequently, we can conclude that after a valuable boost action, the total growth of active sets is reduced. We use this perspective in the appendix, where we provide instances and argue that if no boost action reduces the total growth, then no boost is valuable.

Lemma 59. For an instance I , let $I' = \text{WITHBOOST}(I, B)$ be the result of applying a *valuable* boost action B to I . Let y_S and y'_S denote the growth duration of an active set $S \subseteq V$ in the executions of $\text{BOOSTEDMOATGROWING}$ on I and I' , respectively. Then,

$$\sum_{S \subseteq V} y'_S \leq \sum_{S \subseteq V} y_S.$$

Proof. According to Lemmas 57 and 58, we have

$$\begin{aligned} \sum_{S \subseteq V} y'_S &= \sum_{S \subseteq V} y_S - \text{WIN}(I, B) + \text{Loss}(I, B) && (\text{Lemma 58}) \\ &\leq \sum_{S \subseteq V} y_S. && (\text{Lemma 57}) \end{aligned}$$

□

Local Search Properties. Having established key properties of a single boost action, we now turn to the properties of the local search procedure, which applies multiple boost actions. First, from Lemma 58, we derive the following corollary by noting that, as stated in Definition 50, *win* and *loss* are the total W_{IN} and L_{OSS} accumulated over all boost actions during the local search.

Corollary 60. For an instance (G, t, β) , let t^* be the output fingerprint of $\text{LOCALSEARCH}(G, t, \beta)$. Consider the execution of SHADOWMOATGROWING with both t and t^* . Let y_S and y_S^* denote the growth of S during the respective runs of SHADOWMOATGROWING . Recalling that the total win and loss of this LOCALSEARCH are denoted by *win* and *loss*, we have

$$\sum_{S \subseteq V} y_S^* = \sum_{S \subseteq V} y_S - \text{win} + \text{loss}.$$

The next lemma shows that *loss* is equal to the total growth of active sets accumulated in y_{boost} .

Lemma 61. For an instance (G, t, β) , let t^* be the output fingerprint of $\text{LOCALSEARCH}(G, t, \beta)$. Recall that *loss* denotes the total loss accumulated during this LOCALSEARCH . Let y_{boost} be the value returned by $\text{BOOSTEDMOATGROWING}$ on (G, t, t^*) . Then,

$$\text{loss} = y_{\text{boost}}.$$

Proof. First, let $\text{BOOSTEDMOATGROWING}$ on (G, t, t^*) return $y_{\text{boost}}^{(0)}$. We observe that $y_{\text{boost}}^{(0)} = 0$ since active sets become deactivated when they contain no vertices v with $t_v > \tau$.

Next, let $y_{\text{boost}}^{(i)}$ denote the output of $\text{BOOSTEDMOATGROWING}$ after the i -th valuable boost action, and let $\text{Loss}^{(i)}$ be the corresponding loss for this action. Suppose there are k valuable boost actions. By the lemma's assumptions, we have $y_{\text{boost}} = y_{\text{boost}}^{(k)}$. Then, using Definitions 48 and 50, we have

$$\begin{aligned} \text{loss} &= \sum_{i=1}^k \text{Loss}^{(i)} && \text{(Definition 50)} \\ &= \sum_{i=1}^k (y_{\text{boost}}^{(i)} - y_{\text{boost}}^{(i-1)}) && \text{(Definition 48)} \\ &= y_{\text{boost}}^{(k)} - y_{\text{boost}}^{(0)} && \text{(Eliminating opposite terms)} \\ &= y_{\text{boost}}. \end{aligned}$$

□

We also show how the value of *win* can be computed from the total growth of active sets in the initial and final moat growing executions of the local search.

Lemma 62. For an instance (G, t, β) , let t^* be the output fingerprint of $\text{LOCALSEARCH}(G, t, \beta)$. Recall that *win* denotes the total win of this LOCALSEARCH . Let y_S represent the growth of S in $\text{SHADOWMOATGROWING}(G, t)$, and let $(y_{\text{base}}^*, y_{\text{boost}}^*)$ be the output of $\text{BOOSTEDMOATGROWING}(G, t, t^*)$, where y_S^* denotes the growth of S in this execution. Then

$$\text{win} = \sum_{S \subseteq V} y_S - y_{\text{base}}^*.$$

Proof. It can be concluded from the results so far that:

$$win = \sum_{S \subseteq V} y_S - \sum_{S \subseteq V} y_S^* + loss \quad (\text{Corollary 60})$$

$$= \sum_{S \subseteq V} y_S - y_{base}^* - y_{boost}^* + loss \quad (\text{Lemma 51})$$

$$= \sum_{S \subseteq V} y_S - y_{base}^* \quad (\text{Lemma 61})$$

□

Finally, we show that our local search algorithm runs in polynomial time by proving that the number of iterations is polynomial and using the fact that each iteration considers only a polynomial number of boost actions.

Lemma 63. The LOCALSEARCH procedure runs in polynomial time.

Proof. By Corollary 49, each iteration of Line 4 in LOCALSEARCH examines a polynomial number of candidate boost actions. For each evaluation of boost action, we run one instance of BOOSTEDMOATGROWING, which itself runs in polynomial time.

To complete the proof, we show that the algorithm applies only a polynomial number of valuable boost actions. Specifically, after applying a polynomial number of such actions, no further valuable boost exists within the defined subspace.

In each valuable boost action $B = (v, \tau)$, the chosen time τ corresponds to the moment when an active set S reaches another active set S' , which it had not previously reached while both were active. Additionally, since S' is active, there exists a vertex $u \in S'$ that has not yet reached its required growth time.

This means that v (from S) and u (from S') become a new pair of actively connected vertices, which was not the case before. Since there are at most $\binom{n}{2}$ such pairs, the number of valuable boost actions that can be applied is at most $\binom{n}{2}$, which is polynomial in n .

Therefore, the total number of iterations is polynomial, and each iteration takes polynomial time, implying that the entire local search algorithm runs in polynomial time. □

5.3 Boosted Execution

Up to this point, we have described the general structure and properties of LOCALSEARCH. We now turn our attention to its first execution in Algorithm 2. To support this part of the analysis, we introduce a set of definitions tailored to this specific run. While these definitions may resemble earlier ones, they are adapted to the current context and will be used throughout the remainder of this section.

Recall that in Line 7, we invoke BOOSTEDMOATGROWING on the fingerprint produced by the first call to LOCALSEARCH. We now define the corresponding moat growing execution associated with this invocation.

Definition 64 (Boosted Execution). We refer to the execution of BOOSTEDMOATGROWING at Line 7 of the LOCALSEARCH call made in Line 3 of Algorithm 2 as the *Boosted Execution* (+). The following notation is associated with this execution:

- SOL_{LS} denotes the output forest,
- y_S^+ denotes the growth of subset S ,

- $loss_1$ denotes the *loss* of this LOCALSEARCH,
- win_1 denotes the *win* of this LOCALSEARCH,
- $ActS_\tau^+$ denotes the family of active sets at time τ , and
- t^+ is the boosted fingerprint output by this LOCALSEARCH.

Given the above definition, we observe that Boosted Execution corresponds exactly to the run of BOOSTEDMOATGROWING(G, t^-, t^+).

Recall that multiple boost actions are performed during the execution of LOCALSEARCH. Since each boost action increases the required growth time of a vertex, the following corollary holds.

Corollary 65. For all vertices v , we have

$$t_v^+ \geq t_v^-.$$

Now, given t^+ , we can define the boost phase.

Definition 66 (Boost Phase). At time τ during the execution of a monotonic moat growing algorithm, a vertex v is said to be in the *boost phase* if

$$t_v^- \leq \tau < t_v^+.$$

An active set S is in the *boost phase* at time τ if:

- no vertex in S is in the base phase, and
- at least one vertex in S is in the boost phase.

Definition 67. We define the following quantities for Boosted Execution:

- y_{base}^+ denotes the total growth of active sets during the time they were in their base phase,
- y_S^{b+} denotes the total growth of active set S during the time it was in the base phase, and
- y_{boost}^+ denotes the total growth of active sets during the time they were in their boost phase.

As mentioned earlier, these quantities correspond to the output of BOOSTEDMOATGROWING(G, t^-, t^+).

Since y_S^{b+} represents the total time set S is in the base phase, and y_S^+ represents the total time S is an active set, we have the following corollary.

Corollary 68. For any $S \subseteq V$,

$$y_S^{b+} \leq y_S^+.$$

A Prefix-Time Assignment. Next, we define a new assignment based on Boosted Execution, show that it satisfies the properties of a prefix-time assignment, and compare it with the previously defined one.

Definition 69. Consider the moment τ of Boosted Execution. For each active set $S \in ActS_\tau^+$ in the *base phase* ($\exists v \in S : \tau < t_v^-$), we assign this moment of growth of this active set to all vertices in REPS(BASE(S, τ)) with fraction 1 and 0 for other vertices. We refer to the total growth assigned to vertex v by r_v^+ .

Similar to r^- , we show that r^+ is also a prefix-time assignment.

Lemma 70. Assignment r^+ is a prefix-time assignment.

Proof. Given Definition 28, the first criterion holds since the assigned fraction is either 0 or 1.

Now, suppose a vertex v is assigned a fraction of 1 at moment τ but not at an earlier moment τ' . According to Definition 69, we have $\tau < t_v^-$, which implies $\tau' < t_v^-$. Therefore, at time τ' , vertex v belongs to the base set of some active set S' , i.e., $v \in \text{BASE}(S', \tau')$.

Since v was assigned a fraction of 0 at τ' , there must exist another vertex $u \in \text{BASE}(S', \tau')$ such that $\text{COMP}(u) = \text{COMP}(v)$ and $\text{priority}_u > \text{priority}_v$. This implies $t_u^- \geq t_v^-$ by Corollary 34.

Given that we only merge sets in our process, it follows that $S' \subseteq S$ and $u \in S$. Since $\tau < t_v^- \leq t_u^-$, it follows that $u \in \text{BASE}(S, \tau)$, and hence $v \notin \text{REPS}(\text{BASE}(S, \tau))$, which contradicts the assumption that v was assigned a fraction of 1 at time τ . \square

The next lemma shows that the minimum assigned value r^+ among two vertices in the same connected component of the optimal solution is a lower bound on the time at which they become connected in Boosted Execution.

Lemma 71. For any two vertices u, v in a same connected component of the optimal solution, these vertices are not connected until time $\min(r_u^+, r_v^+)$ in Boosted Execution.

Proof. Without loss of generality, assume $\text{priority}_u > \text{priority}_v$. Let τ be the time at which these two vertices become connected in Boosted Execution. After time τ , they remain in the same connected component for the remainder of the execution. For any set S containing both u and v , Lemma 36 implies that y_S^+ cannot be assigned to v , as u has higher priority. Hence, $\tau \geq r_v^+$ because assigning to v is only possible before or at time r_v^+ . By symmetry, if $\text{priority}_u < \text{priority}_v$, we conclude that $\tau \geq r_u^+$. Therefore, $\tau \geq \min(r_u^+, r_v^+)$, and these vertices do not connect before time $\min(r_u^+, r_v^+)$. \square

Next, we show that the prefix-time assignment obtained from the first local search is strictly smaller than the previous assignment.

Lemma 72. For any vertex $v \in V$,

$$r_v^+ \leq r_v^-.$$

Proof. Since r^+ is a prefix-time assignment, the first moment not assigned to v is precisely r_v^+ . Let S be the last active set in Boosted Execution that assigns its growth to v , and let this happen at time $\tau = r_v^+$. Since v is in the base phase at this moment, we have $t_v^- > \tau$, which implies that v must also belong to an active set S' in Legacy Execution at time τ .

By Corollary 53, active sets in Legacy Execution form a refinement of those in Boosted Execution, which means $S' \subseteq S$. Then, by Lemma 32, we have $\text{BASE}(S', \tau) \subseteq \text{BASE}(S, \tau)$. Since v is in the base phase for both S and S' , and v belongs to $\text{REPS}(\text{BASE}(S, \tau))$, Lemma 37 implies that $v \in \text{REPS}(\text{BASE}(S', \tau))$ as well.

By Definition 39, this means that S' assigns its growth to v at time τ in Legacy Execution. Therefore, r_v^- , which accumulates growth assigned to v over time, must be at least r_v^+ . Hence, $r_v^+ \leq r_v^-$, as claimed. \square

We now show that, in each connected component of the optimal solution, the maximum assigned value (see Definition 26) is equal in both executions.

Lemma 73. For any connected component C in OPT,

$$r_{\max}^-(C) = r_{\max}^+(C).$$

Proof. Let $v \in C$ be a vertex such that $r_v^- = r_{\max}^-(C)$, and among all such vertices, assume v has the highest priority in C .

Consider a moment τ just before t_v^- , where v is in an active set S in Legacy Execution. Since $v \in \text{BASE}(S, \tau)$, S is assigning growth to a vertex in C . Furthermore, as r^- is a prefix-time assignment, S must assign growth to v , as otherwise the vertex in C to which growth is assigned would have a higher value of r^- than v . By Corollary 53, there exists an active set S' in Boosted Execution at moment τ such that $S \subseteq S'$.

We now show that S' also assigns its growth to v . By Lemma 32, $\text{BASE}(S, \tau) \subseteq \text{BASE}(S', \tau)$ so $v \in \text{BASE}(S', \tau)$. This implies that there is a vertex $u \in C$ in $\text{BASE}(S', \tau)$ to which S' assigns growth. Suppose for contradiction $u \neq v$. This would mean that u has a higher priority than v . Since u is active in Legacy Execution at time τ , and has higher priority than v , this contradicts our assumption that v is the highest-priority vertex among those with maximum r^- in C .

Therefore, S' must assign its growth to v , implying $r_v^- \leq r_v^+$. Thus, we can conclude that

$$\begin{aligned} r_{\max}^-(C) &= r_v^- \\ &\leq r_v^+ \\ &\leq r_{\max}^+(C). \end{aligned}$$

Now, let $w \in C$ be a vertex such that $r_w^+ = r_{\max}^+(C)$. Then:

$$\begin{aligned} r_{\max}^-(C) &\leq r_{\max}^+(C) \\ &= r_w^+ \\ &\leq r_w^- && \text{(Lemma 72)} \\ &\leq r_{\max}^-(C). \end{aligned}$$

Thus, all inequalities must be equalities, and we conclude that $r_{\max}^-(C) = r_{\max}^+(C)$. \square

An Exclusive Assignment. We now define our first exclusive assignment, corresponding to Boosted Execution, and establish several of its key properties.

Definition 74. Consider the moment τ during Boosted Execution. For each active set $S \in \text{Act}S^+$ in the *base phase*, we assign this moment of growth *proportionally* to all vertices in $\text{REPS}(\text{BASE}(S, \tau))$. Specifically, each vertex in $\text{REPS}(\text{BASE}(S, \tau))$ receives a fraction of the growth equal to $1/|\text{REPS}(\text{BASE}(S, \tau))|$. We denote the total growth assigned to a vertex v by \hat{r}_v^+ .

Since each active set divides its growth equally among its assigned vertices, this defines an exclusive assignment (see Definition 27). Moreover, since for any active set only the growth during its base phase is assigned, and the total fraction of assignment during those moments is 1, we conclude the following:

Corollary 75. Given a set $S \subseteq V$, we have

$$\sum_{v \in S} \hat{r}_{S,v}^+ = y_S^{b+}.$$

We can extend the above lemma and sum it over all sets to obtain the following lemma.

Lemma 76. For the exclusive assignment in Boosted Execution, we have

$$\sum_{v \in V} \hat{r}_v^+ = y_{base}^+.$$

Proof. We can prove this by expanding the total assignment over all vertices:

$$\begin{aligned}
\sum_{v \in V} \hat{r}_v^+ &= \sum_{v \in V} \sum_{S \subseteq V} \hat{r}_{S,v}^+ && \text{(Definition 26)} \\
&= \sum_{S \subseteq V} \sum_{v \in S} \hat{r}_{S,v}^+ && (\hat{r}_{S,v}^+ = 0 \text{ for all } v \notin S) \\
&= \sum_{S \subseteq V} y_S^{b+} && \text{(Corollary 75)} \\
&= y_{base}^+ && \text{(Definition 67)}
\end{aligned}$$

□

The next lemma, which bounds the sum of \hat{r}^+ over all vertices, follows directly from the above lemma.

Lemma 77. For the exclusive assignment in Boosted Execution, we have

$$\sum_{v \in V} \hat{r}_v^+ \leq c(\text{OPT}) - \text{win}_1.$$

Proof. We have

$$\begin{aligned}
\sum_{v \in V} \hat{r}_v^+ &= y_{base}^+ && \text{(Lemma 76)} \\
&= \sum_{S \subseteq V} y_S^- - \text{win}_1 && \text{(Lemma 62)} \\
&\leq c(\text{OPT}) - \text{win}_1. && \text{(Lemma 45)}
\end{aligned}$$

□

We can also use Lemma 76 to show that the difference between the total y^+ and the total \hat{r}^+ is exactly loss_1 .

Lemma 78. For Boosted Execution, we have

$$\sum_{v \in V} \hat{r}_v^+ + \text{loss}_1 = \sum_{S \subseteq V} y_S^+.$$

Proof. The result follows directly from previous lemmas:

$$\begin{aligned}
\sum_{v \in V} \hat{r}_v^+ + \text{loss}_1 &= y_{base}^+ + \text{loss}_1 && \text{(Lemma 76)} \\
&= y_{base}^+ + y_{boost}^+ && \text{(Lemma 61)} \\
&= \sum_{S \subseteq V} y_S^+. && \text{(Lemma 51)}
\end{aligned}$$

□

We now compare the exclusive and prefix-time assignments of Boosted Execution.

Lemma 79. For any vertex v ,

$$\hat{r}_v^+ \leq r_v^+.$$

Proof. By Definitions 69 and 74, at any moment when growth is assigned to v , the amount contributed to \hat{r}_v^+ is less than or equal to the amount contributed to r_v^+ , since in the exclusive assignment the growth is divided among potentially more vertices. As \hat{r}_v^+ accumulates less (or equal) growth than r_v^+ over time, $\hat{r}_v^+ \leq r_v^+$. \square

We can use the above lemma to show that for each connected component of the optimal solution, the total assigned value \hat{r}^+ to its vertices is at most the cost of its corresponding tree.

Lemma 80. For any connected component of the optimal solution C with tree T_C , we have

$$\sum_{v \in C} \hat{r}_v^+ \leq c(T_C).$$

Proof. It can be concluded as follows:

$$\sum_{v \in C} \hat{r}_v^+ \leq \sum_{v \in C} r_v^+ \quad (\text{Lemma 79})$$

$$\leq \sum_{v \in C} r_v^- \quad (\text{Lemma 72})$$

$$\leq c(T_C). \quad (\text{Lemma 44})$$

\square

The next lemma is independent of the assignment definitions and relies only on the basic setup of Boosted Execution. It shows that the total growth of active sets that do not color any edge of the optimal solution is bounded by the sum of win_1 and $loss_1$.

Lemma 81. We can bound the total growth of active sets that do not cut OPT by $win_1 + loss_1$, that is,

$$\sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^+ \leq win_1 + loss_1.$$

Proof. First, we expand the left-hand side:

$$\sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^+ = \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^+ - \sum_{S \subseteq V} y_S^+ + \sum_{S \subseteq V} y_S^+$$

$$= \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^+ - \sum_{S \subseteq V} y_S^+ + \sum_{S \subseteq V} y_S^{b+} + y_{boost} \quad (\text{Lemma 51})$$

$$= - \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} \neq \emptyset}} y_S^+ + \sum_{S \subseteq V} y_S^{b+} + y_{boost}$$

$$= - \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} \neq \emptyset}} y_S^+ + \sum_{S \subseteq V} y_S^{b+} + loss_1 \quad (\text{Lemma 61})$$

$$\leq - \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} \neq \emptyset}} y_S^{b+} + \sum_{S \subseteq V} y_S^{b+} + loss_1 \quad (\text{Corollary 68})$$

$$= \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^{b+} + loss_1.$$

Next, we expand the right-hand side:

$$\begin{aligned} \text{win}_1 + \text{loss}_1 &= \sum_{S \subseteq V} y_S^- - \sum_{S \subseteq V} y_S^{b+} + \text{loss}_1 && \text{(Lemma 62)} \\ &= \sum_{S \subseteq V} y_S^- - \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^{b+} - \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} \neq \emptyset}} y_S^{b+} + \text{loss}_1. \end{aligned}$$

It suffices to show:

$$\sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^{b+} + \text{loss}_1 \leq \sum_{S \subseteq V} y_S^- - \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^{b+} - \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} \neq \emptyset}} y_S^{b+} + \text{loss}_1.$$

Simplifying, we get:

$$2 \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} = \emptyset}} y_S^{b+} + \sum_{\substack{S \subseteq V \\ \delta(S) \cap \text{OPT} \neq \emptyset}} y_S^{b+} \leq \sum_{S \subseteq V} y_S^-$$

To prove this, consider an active set S in Boosted Execution at some moment τ , assumed to be in its base phase. By Definition 31, there exists $v \in S$ such that $t_v^- > \tau$. We aim to show that the contribution of S to the left-hand side has a corresponding contribution to the right-hand side. We distinguish two cases:

- If $\delta(S) \cap \text{OPT} \neq \emptyset$, then at the same moment in Legacy Execution, v is in an active set $S' \subseteq S$ by Corollary 53. Thus, we say S corresponds to S' .
- If $\delta(S) \cap \text{OPT} = \emptyset$, then $\text{PAIR}_v \in S$; otherwise, there must be a path between v and PAIR_v in the optimal solution, and S would cut that path. In Legacy Execution, at the same moment τ , v and PAIR_v belong to different active sets S' and S'' , respectively, by Definition 29 and the fact that $t_v^- > \tau$. Moreover, both S' and S'' are subsets of S by Corollary 53. In this case, we say S corresponds to S' and S'' .

Therefore, at any given moment, each active set in Boosted Execution either corresponds to one active set in Legacy Execution if it cuts OPT, or corresponds to two active sets in Legacy Execution if it does not cut OPT.

Since the corresponding active sets in Legacy Execution are subsets of distinct active sets in Boosted Execution at a fixed moment, it follows that Legacy active sets themselves are all distinct. This completes the proof. \square

Classifying Connected Components of the Optimal Solution. Next, using the exclusive assignment from Boosted Execution, we partition the connected components of OPT into two distinct families. This partition helps us analyze and bound the cost of our solutions, starting with SOL_{LS} .

Definition 82. For a constant λ , we classify a connected component C with tree T_C in the optimal solution as belonging to family \mathcal{A} if

$$\sum_{v \in C} \hat{r}_v^+ \leq (1 - \lambda) \cdot c(T_C),$$

and to family \mathcal{B} if

$$\sum_{v \in C} \hat{r}_v^+ > (1 - \lambda) \cdot c(T_C).$$

It follows that $\text{OPT} = \mathcal{A} \cup \mathcal{B}$. We also refer to the forests of connected components in \mathcal{A} and \mathcal{B} as $\text{OPT}_{\mathcal{A}}$ and $\text{OPT}_{\mathcal{B}}$, respectively. Consequently, $c(\text{OPT}) = c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}})$.

It is important to note that in Section 9, we determine an appropriate value for λ to ensure that the algorithm achieves the desired approximation guarantee.

We further partition the connected components in \mathcal{B} into two families.

Definition 83. For a constant γ , a connected component $C \in \mathcal{B}$ with tree T_C in the optimal solution is classified into family \mathcal{B}_1 if

$$r_{max}^+(C) \leq \gamma \cdot c(T_C),$$

and into family \mathcal{B}_2 if

$$r_{max}^+(C) > \gamma \cdot c(T_C).$$

Consequently, $\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_2$. We denote by $\text{OPT}_{\mathcal{B}_1}$ and $\text{OPT}_{\mathcal{B}_2}$ the forests consisting of the connected components in \mathcal{B}_1 and \mathcal{B}_2 , respectively. It follows that $c(\text{OPT}_{\mathcal{B}}) = c(\text{OPT}_{\mathcal{B}_1}) + c(\text{OPT}_{\mathcal{B}_2})$.

The value of γ will be defined in terms of other parameters in Definition 153, and its exact value will be determined in Section 9.

By combining Lemma 79 and Definition 82, we have the following property for connected components in \mathcal{B} .

Corollary 84. For any connected component $C \in \mathcal{B}$, we have

$$\sum_{v \in C} r_v^+ > (1 - \lambda) \cdot c(T_C).$$

The next three lemmas focus on bounding the cost of the solution SOL_{LS} , the forest returned by Boosted Execution. First, we show that if this solution does not meet the desired approximation factor, then $loss_1$ can be bounded in terms of the cost of the optimal solution.

Lemma 85. Consider the solution SOL_{LS} . Then either:

- $c(SOL_{LS}) \leq (2 - 2\alpha) \cdot c(\text{OPT})$, or
- $loss_1 \leq \frac{\alpha}{\beta} \cdot c(\text{OPT})$.

Proof. If $c(SOL_{LS}) \leq (2 - 2\alpha) \cdot c(\text{OPT})$, we are done. Otherwise, assume $c(SOL_{LS}) > (2 - 2\alpha) \cdot c(\text{OPT})$. Then,

$$\begin{aligned} c(SOL_{LS}) &\leq 2 \sum_{S \subseteq V} y_S^+ && \text{(Lemma 9)} \\ &= 2 \left(\sum_{S \subseteq V} y_S^- - win_1 + loss_1 \right) && \text{(Corollary 60)} \\ &\leq 2 \left(\sum_{S \subseteq V} y_S^- - \beta loss_1 \right) && \text{(Definition 50)} \\ &\leq 2(c(\text{OPT}) - \beta loss_1). && \text{(Lemma 45)} \end{aligned}$$

Combining this with the assumption $c(SOL_{LS}) > (2 - 2\alpha) \cdot c(\text{OPT})$, we get

$$(2 - 2\alpha) \cdot c(\text{OPT}) < 2(c(\text{OPT}) - \beta loss_1).$$

Dividing both sides by 2 and rearranging terms, we obtain

$$loss_1 < \frac{\alpha}{\beta} \cdot c(\text{OPT}).$$

This completes the proof. □

Next, we show that if SOL_{LS} fails to achieve the desired approximation guarantee, then win_1 can be bounded in terms of the cost of the optimal solution.

Lemma 86. Consider the solution SOL_{LS} . Then either:

- $c(SOL_{LS}) \leq (2 - 2\alpha) \cdot c(\text{OPT})$, or
- $win_1 \leq (\alpha + \frac{\alpha}{\beta}) \cdot c(\text{OPT})$.

Proof. If $c(SOL_{LS}) \leq (2 - 2\alpha) \cdot c(\text{OPT})$, we are done. Otherwise, assume $c(SOL_{LS}) > (2 - 2\alpha) \cdot c(\text{OPT})$. Then:

$$\begin{aligned}
(2 - 2\alpha) \cdot c(\text{OPT}) &< c(SOL_{LS}) \\
&\leq 2 \sum_{S \subseteq V} y_S^+ && \text{(Lemma 9)} \\
&= 2 \left(\sum_{S \subseteq V} y_S^- - win_1 + loss_1 \right) && \text{(Corollary 60)} \\
&\leq 2(c(\text{OPT}) - win_1 + loss_1). && \text{(Lemma 45)}
\end{aligned}$$

Dividing both sides by 2 and rearranging terms gives:

$$\begin{aligned}
win_1 &< loss_1 + \alpha \cdot c(\text{OPT}) \\
&\leq \left(\alpha + \frac{\alpha}{\beta} \right) \cdot c(\text{OPT}), && \text{(Lemma 85)}
\end{aligned}$$

which completes the proof. \square

Finally, we bound the cost of SOL_{LS} by separately analyzing the assigned values of vertices in classes \mathcal{A} and \mathcal{B} , as defined in Definition 82, and incorporating the bound on $loss_1$.

Lemma 87. For the cost of the solution SOL_{LS} , either

$$c(SOL_{LS}) \leq 2(1 - \lambda) \cdot c(\mathcal{A}) + 2c(\mathcal{B}) + 2\frac{\alpha}{\beta} \cdot c(\text{OPT}),$$

or SOL_{LS} is a $(2 - 2\alpha)$ approximation of the optimal solution.

Proof. If SOL_{LS} is a $(2 - 2\alpha)$ approximation, the proof is complete. Otherwise, we proceed as follows:

$$\begin{aligned}
c(SOL_{LS}) &\leq 2 \sum_{S \subseteq V} y_S^+ && \text{(Lemma 9)} \\
&= 2 \sum_{v \in V} \hat{r}_v^+ + 2loss_1 && \text{(Lemma 78)} \\
&\leq 2 \sum_{C \in \mathcal{A}} \sum_{v \in C} \hat{r}_v^+ + 2 \sum_{C \in \mathcal{B}} \sum_{v \in C} \hat{r}_v^+ + 2loss_1 \\
&\leq 2(1 - \lambda) \cdot c(\text{OPT}_{\mathcal{A}}) + 2 \sum_{C \in \mathcal{B}} \sum_{v \in C} \hat{r}_v^+ + 2loss_1 && \text{(Definition 82)} \\
&\leq 2(1 - \lambda) \cdot c(\text{OPT}_{\mathcal{A}}) + 2c(\text{OPT}_{\mathcal{B}}) + 2loss_1 && \text{(Lemma 80)} \\
&\leq 2(1 - \lambda) \cdot c(\text{OPT}_{\mathcal{A}}) + 2c(\text{OPT}_{\mathcal{B}}) + 2\frac{\alpha}{\beta} \cdot c(\text{OPT}). && \text{(Lemma 85)}
\end{aligned}$$

This completes the proof. \square

6 Structural Results on Local Minima

In this section, we analyze the properties of the minimal moat growing algorithms resulting from our local search. Our main property, called the *claw property*, states that for any triple of actively connected vertices, the total growth of active sets separating them lower bounds the cost of any tree connecting them by a constant factor. We generalize this property to larger sets of vertices and use it to bound the cost of our algorithm for the Steiner Tree problem.

6.1 Claw Property

We begin by stating the claw property in the following lemma. Given the result of our local search algorithm, it provides bounds on how long triples of vertices that connect actively can remain separated based on their distances to any other vertex.

Lemma 88 (Claw Property). Consider the final execution of `BOOSTEDMOATGROWING` during a call to `LOCALSEARCH` and let u, v, w be vertices that connect actively in this execution. Define τ as the first moment when at least two of u, v, w belong to the same active set, and let τ' be the first moment when all three belong to the same active set. Then, the following inequality holds for any vertex q

$$\tau + \tau' \leq \frac{d(q, u) + d(q, v) + d(q, w)}{2} + \beta \frac{\min(d(q, u), d(q, v), d(q, w))}{2}.$$

Proof. Let I be the final instance on which `BOOSTEDMOATGROWING` is run during the local search. Consider the boost action $B = (q, \tau^*)$, where τ^* is the smallest value for which q reaches u, v , or w actively in the boosted instance. If no such value exists or if $\tau^* > \tau'$, then q does not reach u, v , or w while all four grow actively for τ' . This implies that

$$d(q, u) \geq 2\tau', \quad d(q, v) \geq 2\tau', \quad \text{and} \quad d(q, w) \geq 2\tau'.$$

Consequently, we obtain

$$\begin{aligned} \tau + \tau' &\leq 2\tau' \\ &\leq \frac{d(q, u)}{2} + \frac{d(q, v)}{2} \\ &\leq \frac{d(q, u) + d(q, v) + d(q, w)}{2} + \beta \frac{\min(d(q, u), d(q, v), d(q, w))}{2}. \end{aligned}$$

From this point forward, we consider the case where $\tau^* \leq \tau'$. We observe that by construction,

$$\tau^* \leq \frac{\min(d(q, u), d(q, v), d(q, w))}{2}.$$

Let I^* be the boosted instance obtained after applying the boost B . Figure 7 illustrates a moment in the execution of `BOOSTEDMOATGROWING` on I^* , when q reaches one of u, v, w . We compare the executions of `BOOSTEDMOATGROWING` on I and I^* .

We use y to denote the y_S values in the execution on I and y^* for the values in the execution on I^* . To analyze the effect of the boost, we track the difference

$$\Delta_y = \sum_{S \subseteq V} y_S - \sum_{S \subseteq V} y_S^*$$

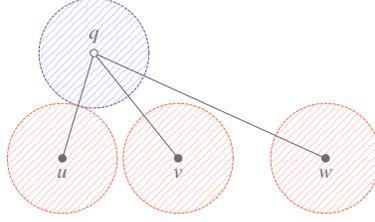


Figure 7: An illustration of the moat-growing process on a subgraph in a boosted instance, where a boost is applied to vertex q . Before the boost, vertices u , v , and w connect while active. The boost enables q to reach u while active, imposing constraints on the growth of sets containing u , v , or w but not q , determined by the distance of q to u , v , and w .

throughout the executions. This allows us to demonstrate that B is a valuable boost action unless the statement of the lemma holds. Since B is included in the boost actions space, and the local search has already terminated, it must not be valuable, completing the proof.

First, we show that at any moment after τ' , the increase in $\sum_{S \subseteq V} y_S$ is at least as large as the increase in $\sum_{S \subseteq V} y_S^*$, implying that Δ_y does not decrease after the moment τ' . To establish this, we observe that by Lemma 56, at any moment after τ' , each active set S in the run on I^* can be mapped to a distinct active set $S' \subseteq S$ at the same moment in the run on I .

Since Δ_y does not decrease after moment τ' , we use z_S and z_S^* to denote the values of y and y^* at moment τ' , and bound Δ_y using the fact that

$$\Delta_y \geq \sum_{S \subseteq V} z_S - \sum_{S \subseteq V} z_S^*. \quad (2)$$

We consider active sets including one of $\{u, v, w, q\}$ separately from those that do not. In the run on I , u , v , and w are contained in three different active sets until moment τ , and two different active sets from moment τ to τ' . Therefore, we have

$$\begin{aligned} \sum_{S \subseteq V} z_S &\geq \sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S + \sum_{\substack{S \subseteq V \\ |S \cap \{u,v,w\}| = 1}} z_S + \sum_{\substack{S \subseteq V \\ |S \cap \{u,v,w\}| = 2}} z_S \\ &\geq \sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S + 3\tau + 2(\tau' - \tau) \\ &= \sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S + 2\tau' + \tau. \end{aligned} \quad (3)$$

On the other hand, in the run on I^* , q connects actively with u , v , or w at a moment before τ' . Then, u , v , and w are guaranteed to be in the same active set as q after the moments $\frac{d(q,u)}{2}$, $\frac{d(q,v)}{2}$ and $\frac{d(q,w)}{2}$ respectively. Sets containing q can grow for at most τ' which is the entire duration we consider. Overall, we get

$$\begin{aligned} \sum_{S \subseteq V} z_S^* &\leq \sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S^* + \sum_{\substack{S \subseteq V \setminus \{q\} \\ u \in S}} z_S^* + \sum_{\substack{S \subseteq V \setminus \{q\} \\ v \in S}} z_S^* + \sum_{\substack{S \subseteq V \setminus \{q\} \\ w \in S}} z_S^* + \sum_{\substack{S \subseteq V \\ q \in S}} z_S^* \\ &\leq \sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S^* + \frac{d(q,u)}{2} + \frac{d(q,v)}{2} + \frac{d(q,w)}{2} + \tau'. \end{aligned} \quad (4)$$

Furthermore, we can once again use Lemma 56 to show that if S is an active set in the run on I^* that does not include q , there must exist a subset of S that is active at the same moment in the run on I . This implies that

$$\sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S^* \leq \sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S \quad (5)$$

as at any moment before τ' , the active sets not including u , v , w , or q in the run on I^* can be mapped to disjoint active sets not including these vertices in the run on I .

Now, we can combine the inequalities to show that

$$\sum_{S \subseteq V} y_S - \sum_{S \subseteq V} y_S^* \geq \sum_{S \subseteq V} z_S - \sum_{S \subseteq V} z_S^* \quad (\text{Equation 2})$$

$$\geq \left(\sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S + 2\tau' + \tau \right) - \left(\sum_{\substack{S \subseteq V \\ S \cap \{u,v,w,q\} = \emptyset}} z_S^* + \frac{d(q,u) + d(q,v) + d(q,w)}{2} + \tau' \right) \quad (\text{Equations 3 and 4})$$

$$\geq (2\tau' + \tau) - \left(\frac{d(q,u) + d(q,v) + d(q,w)}{2} + \tau' \right) \quad (\text{Equation 5})$$

$$= \tau' + \tau - \frac{d(q,u) + d(q,v) + d(q,w)}{2}.$$

On the other hand, we have

$$\sum_{S \subseteq V} y_S - \sum_{S \subseteq V} y_S^* = \text{WIN}(I, B) - \text{Loss}(I, B)$$

by Lemma 58. Since B cannot be a valuable boost, we have

$$\sum_{S \subseteq V} y_S - \sum_{S \subseteq V} y_S^* = \text{WIN}(I, B) - \text{Loss}(I, B) \leq \beta \text{Loss}(I, B).$$

Combining the two inequalities results in

$$\tau + \tau' - \frac{d(q,u) + d(q,v) + d(q,w)}{2} \leq \beta \text{Loss}(I, B).$$

which is equivalent to

$$\tau + \tau' \leq \frac{d(q,u) + d(q,v) + d(q,w)}{2} + \beta \text{Loss}(I, B).$$

Finally, $\text{Loss}(I, B)$ can be upper bounded by

$$\frac{\min(d(q,u), d(q,v), d(q,w))}{2}$$

using Lemma 55 and the fact that

$$\tau^* \leq \frac{\min(d(q,u), d(q,v), d(q,w))}{2}.$$

Therefore, we have

$$\tau + \tau' \leq \frac{d(q,u) + d(q,v) + d(q,w)}{2} + \beta \cdot \frac{\min(d(q,u), d(q,v), d(q,w))}{2}.$$

□

6.2 Generalizing the Claw Property to Larger Sets

Our main result in this section is Lemma 90, which gives a bound on the cost of the tree connecting any set of vertices that actively reach one another in a minimal moat growing algorithm. This bound can be stated for any assignment satisfying the following definition:

Definition 89. Given an execution of a monotonic moat growing and $S \subseteq V$ such that vertices in S connect while active, we say an assignment r is “priority-based on S ” if for any active set S' with $r_{S',v} > 0$ for a vertex $v \in S$, $v = \operatorname{argmax}_{u \in S \cap S'} \text{priority}_u$.

Lemma 90. Consider the final execution of BOOSTEDMOATGROWING during a call to LOCALSEARCH. Let S be a subset of vertices such that S is connected in OPT and the vertices in S are connected while active in this execution. Then, for any assignment r for this execution that is priority-based on S , we have

$$\frac{6}{5 + \beta} \left(\sum_{v \in S} r_v - r_{\max}(S) \right) \leq c(T_S)$$

where T_S is the minimal subtree of OPT connecting S .

To simplify our arguments, we normalize the structure of the tree T_S without loss of generality. We can transform T_S into a rooted perfect binary tree using the following operations:

- adding zero-cost edges and splitting high-degree vertices,
- duplicating vertices and connecting each duplicate to its original via a zero-cost edge, and
- bypassing degree-2 vertices by replacing their incident edges with a single edge.

These operations allow us to ensure that the leaves of the transformed tree form a set S' consisting of S along with possible duplicates of vertices in S . Moreover, all leaves can be assumed to be attached to their parents via zero-cost edges. These modifications do not affect the behavior of BOOSTEDMOATGROWING, since duplicates are immediately connected to their originals during the moat growing process. Additionally, vertices in S' would connect actively.

Therefore, throughout the remainder of this section, we assume without loss of generality that T_S is a rooted perfect binary tree with leaf set S , and that each leaf is attached to its parent via a zero-cost edge.

Let V_I denote the set of internal vertices in T_S , and let *root* be its root. For any non-root vertex v , we use b_v to represent the cost of the edge connecting v to its parent. For each vertex v in T_S , let σ_v be the closest leaf within its subtree, and let ϕ_v be the distance from v to σ_v in T_S . Furthermore, for each internal vertex v , we define τ_v as the first moment during the final execution of BOOSTEDMOATGROWING in the local search when a leaf from its left subtree and a leaf from its right subtree become connected.

As a first step toward proving Lemma 90, we establish a basic bound on the cost of T_S in terms of the ϕ_v values associated with its internal vertices.

Lemma 91. We have

$$\sum_{v \in V_I \setminus \{\text{root}\}} \phi_v \leq c(T_S).$$

Proof. To prove the claim, we use induction to show a slightly stronger statement: for each vertex v , let T_v be the subtree of T_S rooted at v , and let f_v denote the length of the path from v to the farthest leaf from v in T_v . Then,

$$\sum_{u \in T_v} \phi_u \leq c(T_v) - f_v.$$

In the base case, tree T_v is a leaf and both sides of the inequality are zero, so the claim holds trivially.

Suppose the claim holds for the left and right children ℓ and r of a vertex v . Then, we have:

$$\begin{aligned} \sum_{u \in T_v} \phi_u &= \sum_{u \in T_\ell} \phi_u + \sum_{u \in T_r} \phi_u + \phi_v \\ &\leq c(T_\ell) - f_\ell + c(T_r) - f_r + \phi_v && \text{(Induction Hypothesis)} \\ &= c(T_v) - f_\ell - f_r - b_\ell - b_r + \phi_v && (c(T_v) = c(T_\ell) + c(T_r) + b_\ell + b_r) \end{aligned}$$

Now, note that the farthest leaf from v lies in either the left or right subtree. Without loss of generality, assume it lies in the left subtree, so that $f_v = f_\ell + b_\ell$. Then

$$\begin{aligned} \sum_{u \in T_v} \phi_u &\leq c(T_v) - f_\ell - b_\ell - f_r - b_r + \phi_v \\ &= c(T_v) - f_v - f_r - b_r + \phi_v \\ &\leq c(T_v) - f_v \end{aligned}$$

where the last inequality uses the fact that $\phi_v \leq f_r + b_r$, since ϕ_v is the distance from v to its closest leaf in T_v , and $f_r + b_r$ is the distance from v to some leaf in $T_r \subseteq T_v$.

Applying this bound at the root of T_S , and noting that $f_{root} \geq 0$, we obtain:

$$\sum_{v \in V_I \setminus \{root\}} \phi_v \leq \sum_{v \in T_{root}} \phi_v \leq c(T_S).$$

□

To prove Lemma 90, we first establish a bound that relates the sum of τ_v values over the internal vertices of T_S to the cost of the tree $c(T_S)$ in Lemma 92. We then show that this same sum also serves as an upper bound on the sum of assignment values r in Lemma 93.

Lemma 92. We have

$$3 \sum_{v \in V_I} \tau_v \leq \frac{5 + \beta}{2} c(T_S).$$

Proof. For each non-root internal vertex q , let u and v be its children, w its sibling, and p its parent, as illustrated in Figure 8. Considering the leaves σ_u , σ_v , and σ_w , and applying Lemma 88, we obtain the bound:

$$\tau + \tau' \leq \frac{d(q, \sigma_u) + d(q, \sigma_v) + d(q, \sigma_w)}{2} + \beta \cdot \frac{\min(d(q, \sigma_u), d(q, \sigma_v), d(q, \sigma_w))}{2},$$

where τ denotes the first moment when any pair in $\{\sigma_u, \sigma_v, \sigma_w\}$ becomes connected and τ' denotes the first moment when all three are connected.

Now, before the moment $\min(\tau_q, \tau_p)$, none of the leaves in the set $\{\sigma_u, \sigma_v, \sigma_w\}$ can be connected, so $\tau \geq \min(\tau_q, \tau_p)$.

Additionally, since all three of σ_u , σ_v , and σ_w are connected at moment τ' , it follows that $\tau_q \leq \tau'$ and $\tau_p \leq \tau'$, which means $\tau' \geq \max(\tau_q, \tau_p)$. Combining these bounds shows that

$$\tau + \tau' \geq \min(\tau_q, \tau_p) + \max(\tau_q, \tau_p) = \tau_q + \tau_p.$$

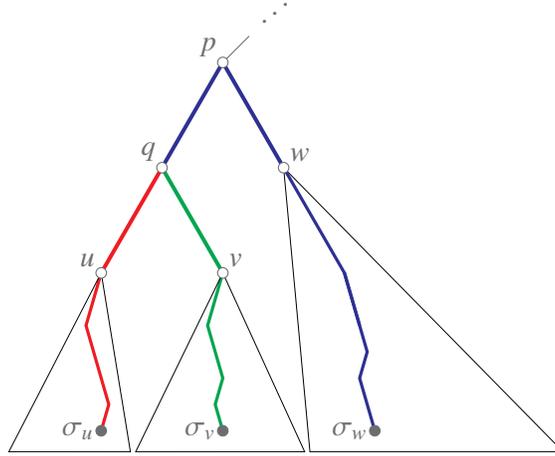


Figure 8: An illustration of a section of the tree T_S connecting its leaves in S , focused on the internal vertex q . Leaves σ_u and σ_v cannot connect before time τ_q , while u and w cannot connect before τ_p . Applying the claw property to q , σ_u , σ_v , and σ_w yields a bound on $\tau_p + \tau_q$ in terms of the costs of the paths from the leaves to q .

Substituting this into the earlier bound yields

$$\tau_q + \tau_p \leq \frac{d(q, \sigma_u) + d(q, \sigma_v) + d(q, \sigma_w)}{2} + \beta \cdot \frac{\min(d(q, \sigma_u), d(q, \sigma_v), d(q, \sigma_w))}{2}.$$

Next, we use the following bounds on the distances from q to leaves $\{\sigma_u, \sigma_v, \sigma_w\}$

$$d(q, \sigma_v) \leq \phi_v + b_v, \quad d(q, \sigma_u) \leq \phi_u + b_u, \quad d(q, \sigma_w) \leq \phi_w + b_q + b_w,$$

to show that

$$\tau_q + \tau_p \leq \frac{(\phi_u + b_u) + (\phi_v + b_v) + (\phi_w + b_q + b_w)}{2} + \beta \cdot \frac{\min(\phi_u + b_u, \phi_v + b_v, \phi_w + b_q + b_w)}{2}.$$

Lastly, since $\min(a_1, a_2, a_3) \leq \frac{a_1 + a_2}{2}$ for any values a_1, a_2, a_3 , we obtain the following bound:

$$\tau_q + \tau_p \leq \frac{\phi_u + b_u + \phi_v + b_v + \phi_w + b_q + b_w}{2} + \beta \cdot \frac{\phi_u + b_u + \phi_v + b_v}{4}. \quad (6)$$

Let ℓ and r be the children of the root and consider the leaves σ_ℓ and σ_r . These leaves belong to separate active sets and grow independently until at least moment τ_{root} , during which they both contribute to coloring the path between them. Therefore,

$$d(\sigma_\ell, \sigma_r) \geq 2\tau_{root}.$$

On the other hand, their distance is at most the length of the path connecting them in the tree T_S :

$$d(\sigma_\ell, \sigma_r) \leq \phi_\ell + b_\ell + \phi_r + b_r.$$

Consequently, we obtain

$$\begin{aligned} \tau_{root} &\leq \frac{\phi_\ell + b_\ell + \phi_r + b_r}{2} \\ &\leq \frac{\phi_\ell + b_\ell + \phi_r + b_r}{2} + \beta \cdot \frac{\phi_\ell + b_\ell + \phi_r + b_r}{4}, \end{aligned}$$

where the additional term with coefficient β is included to align with the structure of inequality (6).

Additionally, for each leaf v with parent p , we have $\tau_p \leq 0$, since leaves in T_S are connected to their parent by zero-cost edges and are thus immediately connected to their siblings.

Now, summing inequality (6) over all non-root internal vertices q , and including the bounds for the root and leaf vertices described above, yields the following overall bound:

$$3 \sum_{v \in V_I} \tau_v \leq \sum_{v \in V(T_S) \setminus \{root\}} \frac{2 + \beta}{4} (\phi_v + b_v) + \sum_{v \in V_I \setminus \{root\}} \frac{\phi_v + 2b_v}{2}.$$

Here, each term τ_v appears three times on the left-hand side of the summed inequalities: once in the inequality associated with vertex v itself, and once in the inequality for each of its two children.

On the right-hand side, for each non-root vertex v , the term $\phi_v + b_v$ is summed with a coefficient of $\frac{1}{2} + \frac{\beta}{4}$ in the inequality corresponding to its parent. Additionally, for each non-root internal vertex v , $\frac{\phi_v + b_v}{2}$ is counted in the inequality for its sibling and $\frac{b_v}{2}$ appears in its own inequality.

Now, since $\phi_v = b_v = 0$ for all $v \notin V_I$, we can simplify the previous bound to:

$$3 \sum_{v \in V_I} \tau_v \leq \frac{4 + \beta}{4} \sum_{v \in V_I \setminus \{root\}} \phi_v + \frac{6 + \beta}{4} \sum_{v \in V_I \setminus \{root\}} b_v.$$

Note that $\sum_{v \in V_I \setminus \{root\}} b_v \leq c(T_S)$, since each edge in T_S appears in this sum at most once. In addition, Lemma 91 implies that $\sum_{v \in V_I \setminus \{root\}} \phi_v \leq c(T_S)$.

Combining these bounds, we conclude that

$$\begin{aligned} 3 \sum_{v \in V_I} \tau_v &\leq \frac{4 + \beta}{4} \sum_{v \in V_I \setminus \{root\}} \phi_v + \frac{6 + \beta}{4} \sum_{v \in V_I \setminus \{root\}} b_v \\ &\leq \frac{10 + 2\beta}{4} c(T_S) \\ &= \frac{5 + \beta}{2} c(T_S). \end{aligned}$$

□

Next, we relate the τ_v values to the assignment r to complete the proof.

Lemma 93. For assignment r that is priority-based on S , we have

$$\sum_{v \in S} r_v - r_{\max}(S) \leq \sum_{v \in V_I} \tau_v.$$

Proof. Let τ' denote the final moment during which the moat-growing procedure runs. It is clear that $r_v \leq \tau'$ and $\tau_v \leq \tau'$ for all v . We now express both sides of the inequality in equivalent integral forms. For the right-hand side, we have

$$\begin{aligned} \sum_{v \in V_I} \tau_v &= \sum_{v \in V_I} \int_0^{\tau_v} 1 \, d\tau \\ &= \int_0^{\tau'} |\{v \in V_I \mid \tau_v \geq \tau\}| \, d\tau. \end{aligned}$$

On the other hand, for the left-hand side, we can write

$$\begin{aligned}
\sum_{v \in S} r_v - r_{\max}(S) &= \sum_{v \in S} \int_0^{r_v} 1 \, d\tau - \int_0^{r_{\max}(S)} 1 \, d\tau \\
&= \int_0^{\tau'} |\{v \in S \mid r_v \geq \tau\}| \, d\tau - \int_0^{r_{\max}(S)} 1 \, d\tau \\
&= \int_0^{\tau'} \max(0, |\{v \in S \mid r_v \geq \tau\}| - 1) \, d\tau.
\end{aligned}$$

In the final step, we use the fact that $|\{v \in S \mid r_v \geq \tau\}| = 0$ for any $\tau > r_{\max}(S)$.

It now suffices to show that, for any moment $\tau \leq \tau'$, the following inequality holds:

$$\max(0, |\{v \in S \mid r_v \geq \tau\}| - 1) \leq |\{v \in V_I \mid \tau_v \geq \tau\}|.$$

Furthermore, since the right-hand side is always non-negative, it suffices to prove that

$$|\{v \in S \mid r_v \geq \tau\}| - 1 \leq |\{v \in V_I \mid \tau_v \geq \tau\}|.$$

Fix a moment τ , and let K denote the family of active sets at moment τ that intersect S non-trivially, with $k = |K|$. Now consider any vertex $v \in S$ with $r_v \geq \tau$. Then, at some moment no earlier than τ , there must exist an active set S' that assigns growth to v ; otherwise, we would have $r_v < \tau$, contradicting the assumption. Since v is still assigned growth after τ , it must be active at moment τ and therefore belongs to an active set $S^* \in K$. Since active sets only grow larger during the moat growing process, we must have $S^* \subseteq S'$.

Moreover, since the assignment r is priority-based on S , v must have the highest priority in $S \cap S'$ by Definition 89 and hence also in $S \cap S^*$. This implies that any vertex v with $r_v \geq \tau$ is the maximum priority vertex in the intersection of some active set $S^* \in K$ with S . Since there are exactly k such active sets, and the maximum priority vertex is unique, it follows that

$$|\{v \in S \mid r_v \geq \tau\}| \leq k.$$

On the other hand, for each value τ_v , consider the pair of leaves $u, w \in S$ that define it—namely, a pair of leaves from the left and right subtrees of v that become connected at moment τ_v . We interpret each such pair as an edge between u and w , forming a graph on the vertex set S with one edge per internal vertex $v \in V_I$. This graph is connected, which can be shown by induction on the structure of the subtree rooted at each internal vertex of T_S .

Now consider the family of active sets K at moment τ , and merge all vertices in each set $S' \in K$ into a single vertex. Since merging vertices preserves connectivity, the resulting graph has $k = |K|$ vertices and remains connected. Thus, it must contain at least $k - 1$ non-self-loop edges.

Each such edge represents a pair u, w that lie in distinct active sets at time τ . Since these leaves are not yet connected at moment τ , the internal vertex v corresponding to this pair satisfies $\tau_v \geq \tau$. Therefore, there are at least $k - 1$ such vertices $v \in V_I$, implying:

$$|\{v \in V_I \mid \tau_v \geq \tau\}| \geq k - 1.$$

Combining this with the earlier bound on r , we obtain:

$$\begin{aligned}
|\{v \in S \mid r_v \geq \tau\}| - 1 &\leq k - 1 \\
&\leq |\{v \in V_I \mid \tau_v \geq \tau\}|,
\end{aligned}$$

which completes the proof. \square

Combining Lemmas 92 and 93 completes the proof of Lemma 90:

$$\begin{aligned} \sum_{v \in S} r_v - r_{\max}(S) &\leq \sum_{v \in V_I} \tau_v \\ &\leq \frac{5 + \beta}{6} c(T_S). \end{aligned}$$

6.3 Approximation for Steiner Tree: Proof of Theorem 2

We can now prove that Algorithm 1 achieves a better than 2 approximation for the Steiner Tree problem.

Proof of Theorem 2. We show that solution SOL_{LS} obtained in Algorithm 1 achieves approximation factor $1 + 2\frac{\sqrt{2}}{3}$. Since Algorithm 1 is identical to the initial part of Algorithm 2, we reuse the terminology and lemmas of Sections 4 and 5 and Legacy Execution and Boosted Execution.

By Lemma 9, we can bound the cost of solution SOL_{LS} as

$$c(SOL_{LS}) \leq 2 \left(\sum_{S \subseteq V} y_S^+ - \sum_{S \subseteq V; v^* \in S} y_S^+ \right)$$

for any vertex v . Now, consider the vertex v^* that maximizes \hat{r}_v^+ . We can plug v^* in to get

$$\begin{aligned} \sum_{S \subseteq V} y_S^+ - \sum_{S \subseteq V; v^* \in S} y_S^+ &\leq \sum_{v \in V} \hat{r}_v^+ + loss_1 - \sum_{S \subseteq V; v^* \in S} y_S^+ && \text{(Lemma 78)} \\ &\leq \sum_{v \in V} \hat{r}_v^+ + loss_1 - \hat{r}_{v^*}^+ && (\hat{r}_{v^*}^+ \leq \sum_{S \subseteq V; v^* \in S} y_S^+) \\ &= \sum_{v \in V} \hat{r}_v^+ + loss_1 - \max_{v \in V} \hat{r}_v^+ && (\hat{r}_{v^*}^+ = \max_{v \in V} \hat{r}_v^+) \end{aligned}$$

Now, let \mathcal{T} be the set of terminals for the Steiner Tree instance. Since vertices in \mathcal{T} must be connected by the algorithm, they are not deactivated before reaching each other and are actively connected in Legacy Execution. By Lemma 15, they must also actively connect in Boosted Execution, since it has a larger fingerprint. Additionally, \hat{r}^+ is priority-based on \mathcal{T} and satisfies the conditions for Lemma 90. Therefore, we can apply this lemma using $S = \mathcal{T}$ and $r = \hat{r}^+$ to get

$$\begin{aligned} \sum_{S \subseteq V} y_S^+ - \sum_{S \subseteq V; v^* \in S} y_S^+ &\leq \sum_{v \in V} \hat{r}_v^+ + loss_1 - \max_{v \in V} \hat{r}_v^+ \\ &\leq \frac{5 + \beta}{6} c(\text{OPT}) + loss_1 \end{aligned}$$

since the cost of the minimal tree $T_{\mathcal{T}}$ connecting \mathcal{T} is $c(\text{OPT})$. Furthermore, we can use Lemma 85 to achieve the following bound

$$\begin{aligned} c(SOL_{LS}) &\leq 2 \left(\frac{5 + \beta}{6} c(\text{OPT}) + loss_1 \right) \\ &\leq 2 \left(\frac{5 + \beta}{6} c(\text{OPT}) + \frac{\alpha}{\beta} c(\text{OPT}) \right) \\ &\leq 2 \left(\frac{5 + \beta}{6} + \frac{\alpha}{\beta} \right) c(\text{OPT}) \end{aligned}$$

when $c(SOL_{LS}) \geq (2 - 2\alpha)c(OPT)$. Equivalently, we can bound the cost of SOL_{LS} as follows

$$c(SOL_{LS}) \leq \max\left(2 - 2\alpha, 2\left(\frac{5 + \beta}{6} + \frac{\alpha}{\beta}\right)\right)c(OPT).$$

Finally, minimizing the coefficient of $c(OPT)$ gives us an approximation ratio of $1 + 2\frac{\sqrt{2}}{3} \approx 1.943$ when $\alpha = \frac{1}{2} - \frac{\sqrt{2}}{3}$ and $\beta = \sqrt{2} - 1$. \square

7 Extended Moat Growing

We show how our local search can yield a better-than-2 approximation for the Steiner Tree problem. However, this approach does not directly extend to the Steiner Forest problem. More precisely, the bound derived from our local search applies only to vertices that are actively connected, a property that does not necessarily hold for each connected component of the optimal solution in the Steiner Forest case.

To address this, we introduce a new monotonic moat growing algorithm, which extends an existing moat growing algorithm. In our case, we use Boosted Execution as the base algorithm. In the extended version, a small fraction of each active set's growth in the original algorithm is assigned as potential to one of its vertices. During the extended process, when an active set is about to become inactive, it can consume this potential to remain active for a longer period. This additional step tends to result in the majority of vertices within each connected component of the optimal solution becoming actively connected, which allows us to apply the same bound as in the Steiner Tree case to each component. Therefore, after performing the extension, we run our local search again to apply that bound.

Note that in some components, a significant portion of vertices may still fail to become actively connected. In these cases, we show that most of the growth introduced by the extension step ends up coloring more than one edge of the optimal solution, which in turn leads to a new bound for those components.

We now describe in detail how this extended moat growing procedure works.

7.1 Algorithm

In `EXTENDEDMOATGROWING`, described in Algorithm 7, each vertex v is provided with a fingerprint value t_v^{in} and a potential value π_v as input. Initially, potentials are assigned to singleton connected components. Whenever two components merge, their potentials are combined and assigned to the new component.

In this moat growing algorithm, active sets grow until they reach the maximum t^{in} value among their vertices. After that, they continue growing for an extended period by consuming their potential until it is fully exhausted. The output fingerprint of this method is the time at which each vertex becomes deactivated, which is larger than fingerprint t^{in} .

The algorithm uses the notation \mathcal{P}_S to store the remaining potential of each active set S . In Line 17, this value is reduced as active sets consume their potential, and in Line 24, the potentials of merging active sets are combined. Finally, in Line 26, we use \mathcal{P}_S to identify active sets that have both reached their maximum t^{in} and exhausted their potential, allowing us to deactivate them.

Algorithm 7 Extended Moat Growing

Input: An undirected graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, a function $t^{in} : V \rightarrow \mathbb{R}_{\geq 0}$ indicating the input fingerprint, and a potential $\pi : V \rightarrow \mathbb{R}_{\geq 0}$ for each vertex.

Output: A function $t^{out} : V \rightarrow \mathbb{R}_{\geq 0}$ indicating the latest moment each vertex was actively growing during the algorithm.

```
1: procedure EXTENDEDMOATGROWING( $G, t^{in}, \pi$ )
2:    $\tau \leftarrow 0$ 
3:    $FC \leftarrow \{\{v\} \mid v \in V\}$ 
4:    $DS \leftarrow \{v \mid v \in V, t_v^{in} = 0, \pi_v = 0\}$ 
5:    $ActS \leftarrow \{\{v\} \mid v \in V, v \notin DS\}$ 
6:    $t_v^{out} \leftarrow 0$  for all  $v \in V$ 
7:    $\mathcal{P}_{\{v\}} \leftarrow \pi_v$  for all  $v \in V$ 
8:   Implicitly set  $y_S \leftarrow 0$  for all  $S \subseteq V$ 
9:   while  $ActS \neq \emptyset$  ▷ While there exists an active set
10:      $\Delta_e \leftarrow \min_{e=uv \in E} \frac{c_e - \sum_{S \ni e} y_S}{|\{S_u, S_v\} \cap ActS|}$ , where  $u \in S_u \in FC, v \in S_v \in FC$ , and  $S_u \neq S_v$ 
11:      $\Delta_t \leftarrow \min_{v \in V, t_v^{in} > \tau} (t_v^{in} - \tau)$ 
12:      $\Delta_p \leftarrow \min_{S \in ActS, \mathcal{P}_S > 0} \mathcal{P}_S$ 
13:      $\Delta \leftarrow \min(\Delta_e, \Delta_t, \Delta_p)$ 
14:     for  $S \in ActS$  do
15:        $y_S \leftarrow y_S + \Delta$ 
16:       if  $\max_{v \in S} t_v^{in} \leq \tau$  then
17:          $\mathcal{P}_S \leftarrow \mathcal{P}_S - \Delta$ 
18:        $\tau \leftarrow \tau + \Delta$ 
19:     for  $e \in E$  do
20:       Let  $S_v, S_u \in FC$  be the sets that contains each endpoint of  $e$ 
21:       if  $\sum_{S: e \in \delta(S)} y_S = c_e$  and  $S_v \neq S_u$  then ▷ Edge  $(v, u)$  become fully colored
22:          $FC \leftarrow (FC \setminus \{S_u, S_v\}) \cup \{S_u \cup S_v\}$ 
23:          $ActS \leftarrow (ActS \setminus \{S_u, S_v\}) \cup \{S_u \cup S_v\}$ 
24:          $\mathcal{P}_{S_u \cup S_v} \leftarrow \mathcal{P}_{S_u} + \mathcal{P}_{S_v}$ 
25:       for  $S \in ActS$  do
26:         if  $\mathcal{P}_S = 0$  and  $\max_{v \in S} t_v^{in} \leq \tau$  then ▷  $S$  become inactive
27:            $ActS \leftarrow ActS \setminus \{S\}$ 
28:         for  $v \in S \setminus DS$  do
29:            $t_v^{out} \leftarrow \tau$ 
30:            $DS \leftarrow DS \cup \{v\}$ 
31:   return  $t^{out}$ 
```

In Algorithm 2, we do not directly call EXTENDEDMOATGROWING. Instead, we call EXTEND in Line 4, which internally uses EXTENDEDMOATGROWING. Within this call, we make use of the output of Boosted Execution. Based on the growth of active sets during their base phase in Boosted Execution—denoted by y^{b+} in Definition 67—we construct an exclusive assignment r such that for any set $S \subseteq V$, it holds that

$$\sum_{v \in S} r_{S,v} = y_S^{b+}.$$

Note that, according to Definition 26, we assign only to active vertices in S . We then run EXTENDEDMOATGROWING with fingerprint t^+ and potential values $\pi_v = \epsilon r_v$ for each vertex v , producing a new fingerprint t' .

We will later show that the output t' remains unchanged as long as the assignment r used to define π satisfies the condition above. This allows us to substitute another assignment in our analysis without affecting the resulting fingerprint. The pseudocode of `EXTEND` is provided in Algorithm 8.

Finally, we apply the `LOCALSEARCH` algorithm to the instance with fingerprint t' , resulting in the solution SOL_{XT} (Line 5 in Algorithm 2).

Algorithm 8 `Extend`

Input: Graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, function $t^- : V \rightarrow \mathbb{R}_{\geq 0}$ specifying the fingerprint of Legacy Execution, function $t^+ : V \rightarrow \mathbb{R}_{\geq 0}$ specifying the fingerprint of Boosted Execution, function $y^{b+} : 2^V \rightarrow \mathbb{R}_{\geq 0}$ specifying the growth in the base phase of Boosted Execution for each subset of vertices, and parameter $\epsilon > 0$.

Output: t' is the fingerprint resulting from the extension.

```

1: procedure EXTEND( $G, t^-, t^+, y^{b+}, \epsilon$ )
2:    $r_v \leftarrow 0$  for all  $v \in V$ 
3:   for  $S \subseteq V$  such that  $y_S^{b+} > 0$  do
4:      $v \leftarrow \arg \max_{v \in S} t_v^-$ 
5:      $r_v \leftarrow r_v + y_S^{b+}$ 
6:    $t' \leftarrow \text{EXTENDEDMOATGROWING}(G, t^+, \epsilon r)$ 
7:   return  $t'$ 

```

We observe that Algorithm 7 runs in polynomial time for any given input, since in each iteration of the loop, one of the following occurs: two active sets are merged, an active set becomes inactive, or τ passes the maximum t^{in} value among the vertices of an active set. Each of these three events can occur at most a linear number of times in $|V|$, ensuring the algorithm's overall polynomial runtime. Consequently, Algorithm 8 runs in polynomial time.

Corollary 94. The `EXTEND` procedure runs in polynomial time.

7.2 Properties and Definitions

Our first step is to show that the behavior of `EXTENDEDMOATGROWING` is the same if potential π_v is defined using any exclusive assignment where $\sum_{v \in S} r_{S,v} = y_S^{b+}$ for all sets $S \subseteq V$. Then, we will assume that $r_v = \hat{r}_v^+$ is used in the remainder of this section, as this assignment satisfies the required condition due to Corollary 75.

In proving this, we use the following corollary, which follows from Lemma 13 and the fact that the fingerprint of any run of `EXTENDEDMOATGROWING` is larger than its input fingerprint.

Corollary 95. Consider an execution of `EXTENDEDMOATGROWING` given fingerprint t_v^+ and any potential $\pi_v \geq 0$. Then, at any moment τ , the active sets of Boosted Execution form a refinement of the active sets in this execution at the same moment. Additionally, the connected components at the same moment in Boosted Execution are a refinement of the connected components in this execution.

Lemma 96. The behavior of `EXTENDEDMOATGROWING` given fingerprint t' is identical for all choices of exclusive assignment r_v used to define potential values $\pi_v = \epsilon r_v$, as long as the assignment satisfies

$$\sum_{v \in S} r_{S,v} = y_S^{b+}$$

for any set $S \subseteq V$. In particular, the output t^{out} will be the same.

Proof. Assume otherwise that the output differs for two exclusive assignments $r^{(1)}$ and $r^{(2)}$ satisfying the condition. Then, consider the first moment τ where active sets in the two runs of EXTENDEDMOATGROWING are not the same. There are two possible cases:

1. An edge e becomes fully colored in one instance, merging two active sets while it does not become fully colored in the other. This is not possible, as active sets have been the same in the two runs until τ , so the sum $\sum_{S:e \in \delta(S)} y_S$ is the same between the two runs and edge e can become fully colored in either both runs or neither.
2. An active set S becomes inactive in one run, while it remains active in the other. In this case, since $\max_{v \in S} t_v^{in}$ is the same between the two runs, we must have $\mathcal{P}_S = 0$ in one run while $\mathcal{P}_S \neq 0$ in the other, where \mathcal{P}_S shows the remaining potential of set S . Now, \mathcal{P}_S for any set S is $\sum_{v \in S} \pi_v$ minus the potential used by subsets of S . Since active sets at any moment before τ are the same in the two runs and use their potential at the same moments, the potential used by subsets of S is the same between runs. Therefore, it suffices to show that $\sum_{v \in S} \pi_v$ is the same in the different runs. Furthermore, since $\pi_v = \epsilon r_v$, we need to show that $\sum_{v \in S} r_v^{(1)} = \sum_{v \in S} r_v^{(2)}$.

Now, consider any vertex $v \in S$. Since v can become inactive at moment τ in the EXTENDEDMOATGROWING runs, it cannot be in an active set at the same moment in Boosted Execution by Corollary 95 and becomes inactive at τ at the latest. Therefore, any active set S' assigning growth to v in Boosted Execution must include v at a moment $\tau' \leq \tau$. Corollary 95 implies that $S' \subseteq S^*$ where S^* is the active set including v at moment τ' in the EXTENDEDMOATGROWING run. Furthermore, S^* must be a subset of S since $\tau' \leq \tau$. Therefore, for any active set S' that assigns growth to a vertex v in S , we have $S' \subseteq S$. Now, we have

$$\begin{aligned}
\sum_{v \in S} r_v^{(1)} &= \sum_{v \in S} \sum_{S' \subseteq V} r_{S',v}^{(1)} \\
&= \sum_{v \in S} \sum_{S' \subseteq S} r_{S',v}^{(1)} && (S' \subseteq S \text{ if } r_{S',v}^{(1)} > 0 \text{ for any } v \in S) \\
&= \sum_{S' \subseteq S} \sum_{v \in S} r_{S',v}^{(1)} && (\text{Change summation order}) \\
&= \sum_{S' \subseteq S} \sum_{v \in S'} r_{S',v}^{(1)} && (S' \subseteq S, r_{S',v}^{(1)} = 0 \text{ for all } v \notin S') \\
&= \sum_{S' \subseteq S} y_{S'}^{b+}.
\end{aligned}$$

Similarly, we can prove the same equality for $r^{(2)}$. Therefore, we have

$$\sum_{v \in S} r_v^{(1)} = \sum_{S' \subseteq S} y_{S'}^{b+} = \sum_{v \in S} r_v^{(2)}.$$

Therefore, set S cannot become inactive in one run and not in the other.

As neither case for active sets differing is possible, the active sets must be the same at any moment for different initial assignments used for calculating potential values. This means that the behavior of the algorithm will be the same, and the same output will be generated. \square

Given Lemma 96, we assume in our analysis that EXTENDEDMOATGROWING is called with potentials defined using \hat{r}^+ as opposed to the the assignment used in Algorithm 8.

7.3 Extended Execution

We now introduce the notations for the calls to EXTENDEDMOATGROWING, beginning with Extended Execution:

Definition 97 (Extended Execution). The execution of EXTENDEDMOATGROWING (Line 6 of Algorithm 8) in the call of EXTEND in Line 4 of Algorithm 2 is referred to as *Extended Execution*. In addition, we define the following notation:

- y'_S represents the growth of set S ,
- $ActS'_\tau$ is the family of active sets at time τ ,
- $SUPERACTIVE'(S)$ is the superactive set derived from the active set S ,
- \mathcal{M}' denotes the family of maximal superactive sets,
- t' is the output fingerprint such that $t'_v \geq t_v^+$ for all $v \in V$.

Based on this fingerprint, we define the potential phase as follows:

Definition 98 (Potential Phase). In any monotonic moat growing, vertex v is in *potential phase* at moment τ if $t_v^+ \leq \tau < t'_v$.

An active set S is considered to be in the *potential phase* if:

- no vertex in S is in the base phase or boost phase, and
- at least one vertex in S is in the potential phase.

Since the fingerprint t' is larger than the fingerprint t^+ , we can use Lemma 13 to obtain the following.

Corollary 99. The active sets at any moment τ during Boosted Execution form a *refinement* of the active sets at the same moment during Extended Execution. Similarly, the connected components at the same moment in Boosted Execution form a refinement of the connected components in Extended Execution.

Now, we define assignments for Extended Execution.

Definition 100. We simulate Extended Execution to compute this assignment. First, assume each vertex v has a *potential assignment capacity*, ρ_v , which initially is set to $\epsilon \hat{r}_v^+$.

Whenever two active sets, A and B , merge, for any pair of vertices u and v in $A \cup B$ such that $\text{COMP}(u) = \text{COMP}(v)$ and $\text{priority}_u < \text{priority}_v$, with both u and v still active, we transfer the remaining potential assignment capacity of u to v , i.e., $\rho_v \leftarrow \rho_v + \rho_u$ and $\rho_u \leftarrow 0$. After all transfers are completed, at most one vertex in each connected component of the optimal solution in the active set has a positive remaining potential assignment capacity.

Consider the moment τ during Extended Execution. For each active set $S \in ActS'_\tau$:

- If S is in the *base phase*, we assign its growth at this moment *proportionally* to all vertices in $\text{REPS}(\text{BASE}(S, \tau))$, where the fraction of growth assigned to each vertex is $1/|\text{REPS}(\text{BASE}(S, \tau))|$. We refer to the total growth assigned to vertex v in the base phase by S as $\hat{r}_{S,v}^{b'}$.
- If S is in the *boost phase*, no growth is assigned to any vertex.
- If S is in the *potential phase*, we assign this moment of growth *proportionally* to all vertices with positive remaining potential assignment capacity in S (with fraction $1/k$ if there are k such vertices). We then decrease the potential assignment capacity of these vertices accordingly. We refer to the total growth assigned to vertex v in the potential phase by S as $\hat{r}_{S,v}^{p'}$.

We denote the vertices to which active set S assigns its growth at moment τ by $Assignee'(S, \tau)$.

We also refer to the total amount of growth assigned to vertex v as \hat{r}'_v , and to the total amount of growth assigned to vertex v by active set S as $\hat{r}'_{S,v} = \hat{r}^{b'}_{S,v} + \hat{r}^{p'}_{S,v}$.

We first show that throughout the simulation described above, the potential remaining for an active set in Algorithm 7 matches the remaining potential assignment capacity. This ensures the validity of the assignment, and shows that any active set in the potential phase has capacity to which it can assign growth.

Lemma 101. In the simulation of Extended Execution (Definition 100), at any time τ , for any active set S ,

$$\sum_{v \in S} \rho_v = \mathcal{P}_S.$$

Proof. We have $\pi_v = \epsilon \hat{r}_v^+$. Initially, for each vertex v in Extended Execution, $\mathcal{P}_{\{v\}} = \pi_v = \epsilon \hat{r}_v^+$ (Line 8). Similarly, $\rho_v = \epsilon \hat{r}_v^+$ for all $v \in V$ (Definition 100), establishing the base case at $\tau = 0$.

At any later time, when two sets S_u and S_v merge, their potentials sum: $\mathcal{P}_{S_u \cup S_v} \leftarrow \mathcal{P}_{S_u} + \mathcal{P}_{S_v}$ (Line 24). Similarly, ρ can undergo capacity transfers that preserve the total capacity (Definition 100), ensuring that the condition of the lemma remains valid.

During a *potential phase* of duration Δ for active set S , we update $\mathcal{P}_S \leftarrow \mathcal{P}_S - \Delta$ (Line 17). In the beginning of this period, the total remaining capacity in S is equal to \mathcal{P}_S , which is at least Δ . During this period, S assigns growth with fraction one (Definition 100), and it follows that the total capacity decreases by Δ . Therefore, the condition of the lemma is preserved.

Since every modification to \mathcal{P} and ρ maintains

$$\sum_{v \in S} \rho_v = \mathcal{P}_S$$

for any active set S , the proof is complete. \square

Next, we focus on the relationship of assignment and growth of active sets in Extended Execution.

Lemma 102. The total growth in Extended Execution can be bounded as follows:

$$\sum_{S \subseteq V} y'_S \leq \sum_{v \in V} \hat{r}'_v + loss_1.$$

Proof. Consider any active set $S \in ActS'_\tau$ growing at any moment τ of Extended Execution. There are two cases for S :

In the first case, suppose S is in the base phase or the potential phase. By Definition 100, S assigns its growth at this moment, with total fraction 1, to a subset of its vertices. Consequently, the growth for this moment is calculated in both y'_S and $\sum_{v \in V} \hat{r}'_v$.

In the second case, consider S is in the boost phase. For any $v \in S$, we have $t_v^- \leq \tau$, and there exists a vertex u such that $\tau < t_u^+$. At the same moment in Boosted Execution, u is in an active set S' since $\tau < t_u^+$. By Corollary 99, we have $S' \subseteq S$. Any vertex $v \in S' \subseteq S$ satisfies $t_v^- \leq \tau$, meaning all vertices of S' are in the boost phase. Therefore, at this moment, there exists a subset $S' \subseteq S$ whose growth is calculated in y_{boost}^+ , while the growth of S is calculated in y'_S .

Thus, at any given moment during Extended Execution, the growth of each active set is accounted for in either $\sum_{v \in V} \hat{r}'_v$ or y_{boost}^+ . Additionally, since active sets at any moment are mutually exclusive, no calculation is considered multiple times, which leads to:

$$\begin{aligned} \sum_{S \subseteq V} y'_S &\leq \sum_{v \in V} \hat{r}'_v + y_{boost}^+ \\ &= \sum_{v \in V} \hat{r}'_v + loss_1. \end{aligned} \tag{Lemma 61}$$

□

In the following lemma, we prove that the total growth assigned to a vertex by active sets in the base phase of Extended Execution is bounded by the total growth assigned to that vertex in Boosted Execution.

Lemma 103. In assignment \hat{r}' , for any vertex v ,

$$\sum_{S \subseteq V} \hat{r}'_{S,v} \leq \hat{r}_v^+.$$

Proof. In Extended Execution, assume that at moment τ , active set $S \in ActS'_\tau$ in the base phase assigns its growth to vertex v . This implies that $v \in \text{BASE}(S, \tau)$, and therefore $\tau < t_v^-$. Consequently, v is active in Boosted Execution at moment τ , meaning there exists an active set $S' \in ActS_\tau^+$ such that $v \in S'$.

By Corollary 99, we have $S' \subseteq S$. Additionally, by Lemma 32, it follows that $\text{BASE}(S', \tau) \subseteq \text{BASE}(S, \tau)$. Since $v \in \text{REPS}(\text{BASE}(S, \tau))$, Lemma 37 tells us that $v \in \text{REPS}(\text{BASE}(S', \tau))$. Therefore, S' assigns its growth to v in assignment \hat{r}^+ .

To prove the inequality, it suffices to show that, at any given moment, the growth that set S assigns to vertex v in the assignment \hat{r}' is no greater than the growth that S' assigns to v at the same moment. This is because the growth assigned to v by S is calculated in $\hat{r}'_{S,v}$, while the growth assigned to v by S' is calculated in r_v^+ .

To proceed, since $\text{BASE}(S', \tau) \subseteq \text{BASE}(S, \tau)$, by Lemma 38, we have

$$|\text{REPS}(\text{BASE}(S', \tau))| \leq |\text{REPS}(\text{BASE}(S, \tau))|.$$

Since both S and S' assign growth proportionally (Definitions 74 and 100), it follows that S' assigns a larger fraction to v than S , completing the proof. □

Next, we aim to establish a similar bound on the growth assigned to a vertex by active sets during the potential phase in Extended Execution, this time using the potential the vertex receives based on growth assigned to in Boosted Execution. While such a bound may not hold for individual vertices, we prove an analogous inequality for certain subsets of vertices—specifically, the intersection of any maximal superactive set in Extended Execution with any connected component of the optimal solution.

Lemma 104. In assignment \hat{r}' , for the intersection of any connected component C of the optimal solution and any maximal superactive set S of Extended Execution,

$$\sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}'_{S',v} = \epsilon \sum_{v \in S \cap C} \hat{r}_v^+.$$

Proof. In Extended Execution, let S' be the active set S is derived from, and let τ be the time when S' becomes deactivated. We observe that $\mathcal{P}_{S'} = 0$ at this moment (Line 26). By Lemma 101, we also have $\sum_{v \in S'} \rho_v = \mathcal{P}_{S'} = 0$, indicating that no vertex in S' , and therefore none in $S \subseteq S'$, has any remaining capacity.

Now, we analyze the behavior of the assignment \hat{r}' based on Definition 100. First, no vertex transfers its capacity to any vertex outside its connected component of the optimal solution. Second, an active set is deactivated precisely when its \mathcal{P} value reaches zero, meaning no assignment capacity remains among its vertices. Thus, capacity is transferred only between vertices that have not been deactivated yet within the same active set. This means that whenever capacity is exchanged between two vertices, they must belong to the same superactive set in Extended Execution. It follows from laminarity of superactive sets that for any maximal superactive set in Extended Execution, capacity is not given to or received from vertices outside the superactive set, and its capacity remains confined to its vertices.

Combining these observations, we conclude that at moment τ , the assignment capacity of vertices in $S \cap C$ is used solely through growth assigned to vertices within the same set. Since these vertices reach a final capacity of zero, the total growth assigned to vertices in $S \cap C$ during the potential phase of active must be exactly

$$\epsilon \cdot \sum_{v \in S \cap C} \hat{r}_v^+.$$

□

Now, we can combine the bounds obtained in Lemma 103 and Lemma 104 for the total growth assigned to each vertex in Extended Execution.

Lemma 105. For the intersection of any connected component C of the optimal solution and any maximal superactive set S in Extended Execution,

$$\sum_{v \in S \cap C} \hat{r}'_v \leq (1 + \epsilon) \sum_{v \in S \cap C} \hat{r}_v^+.$$

Proof. We derive the inequality as follows:

$$\begin{aligned} \sum_{v \in S \cap C} \hat{r}'_v &= \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}'_{S',v} \\ &= \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{b'}_{S',v} + \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{p'}_{S',v} && \text{(Definition 100)} \\ &\leq \sum_{v \in S \cap C} \hat{r}_v^+ + \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{p'}_{S',v} && \text{(Lemma 103)} \\ &= \sum_{v \in S \cap C} \hat{r}_v^+ + \epsilon \sum_{v \in S \cap C} \hat{r}_v^+ && \text{(Lemma 104)} \\ &= (1 + \epsilon) \sum_{v \in S \cap C} \hat{r}_v^+. \end{aligned}$$

□

7.4 Extended-Boosted Execution

Next, we focus on the execution of LOCALSEARCH after extension.

Definition 106 (Extended-Boosted Execution). We refer to the corresponding execution of BOOSTEDMOAT-GROWING (Line 7; Algorithm 6) for the LOCALSEARCH in Line 5 of Algorithm 2 as Extended-Boosted Execution. Consequently, we define the following:

- The growth of set S is denoted by y''_S ,
- The family of active sets at moment τ is $ActS''_\tau$,

- The superactive set derived from the active set S is $\text{SUPERACTIVE}''(S)$,
- The family of maximal superactive sets is \mathcal{M}'' ,
- The solution found by this execution is SOL_{XT} ,
- The total loss in the LOCALSEARCH is loss_2 ,
- The total win in the LOCALSEARCH is win_2 ,
- The output fingerprint of LOCALSEARCH is t'' , where $t''_v \geq t'_v$ for all $v \in V$.

For any connected component C in OPT , we define the following notation.

Definition 107. For any component $C \in \text{OPT}$, let $S \in \mathcal{M}''$ be the maximal superactive set with a non-empty intersection with C that is deactivated the latest in Extended-Boosted Execution. Then, we define $LG(C)$ as the intersection $S \cap C$ and use $T_{LG(C)}$ to denote the minimal subtree of T_C that connects the vertices in $LG(C)$.

Next, we define the final boost phase as follows:

Definition 108. (Final Boost Phase) A vertex v is in the final boost phase at moment τ if $t'_v \leq \tau < t''_v$.

A set S is in the final boost phase at moment τ if it has no vertex in base, boost, or potential phases while it has at least one vertex in the final boost phase.

We refer to the total growth of active sets in Extended-Boosted Execution during the final boost phase as y''_{boost} . This corresponds to the y_{boost} in the output of $\text{BOOSTEDMOATGROWING}(G, t', t'')$ in the last LOCALSEARCH call in Line 5 of Algorithm 2, since in Line 19, the condition $t'_v \leq \tau$ holds for all vertices in an active set, which is equivalent to the final boost phase.

We next define the function DEPUTY , which is conceptually similar to the REPS function. This allows us to define an assignment for Extended-Boosted Execution that follows priority values.

Definition 109 (Deputy). Let U and S be sets of vertices such that $S \subseteq U \subseteq V$. Then, $\text{DEPUTY}(U, S)$ is the set of vertices in U that have the highest priority within the intersection of U and their respective connected components in OPT , for those components that intersect S non-trivially, i.e.,

$$\text{DEPUTY}(U, S) = \{u \in U \mid \exists v \in S : u = \underset{w \in U \cap \text{COMP}(v)}{\text{argmax}} \text{priority}_w\}.$$

Then, we prove the following for DEPUTY .

Lemma 110. For any sets of vertices $S \subseteq U \subseteq V$,

$$|\text{DEPUTY}(U, S)| \geq |\{\text{COMP}(v) \mid v \in S\}|.$$

Proof. Consider any vertex $u \in S$, and let $C = \text{COMP}(u)$. Since $u \in S \subseteq U$, it follows that $U \cap C \neq \emptyset$.

Let $v \in U \cap C$ be the vertex with the highest priority among those in the intersection. We claim that $v \in \text{DEPUTY}(U, S)$, which we will verify shortly. Assuming this claim, it follows that each distinct optimal component C corresponding to some $u \in S$ maps to a vertex $v \in \text{DEPUTY}(U, S)$ such that $v \in C$. As v is in C and the components are disjoint, these vertices are distinct. This implies the number of vertices in $\text{DEPUTY}(U, S)$ is at least the number of distinct optimal components over S , as desired.

To justify the claim that $v \in \text{DEPUTY}(U, S)$, we check that it satisfies the conditions of Definition 109:

- $v \in U$ since $v \in U \cap C$

- For the mentioned u , we have $C = \text{COMP}(u)$, and $v = \arg \max_{w \in U \cap \text{COMP}(u)} \text{priority}_w$.

Hence, $v \in \text{DEPUTY}(U, S)$, completing the proof. \square

Now, we define the growth assignment to vertices for this execution.

Definition 111. Consider the moment τ of Extended-Boosted Execution. For each active set S :

- If S is in the base phase, we divide the growth of this moment proportionally among all vertices in $\text{REPS}(\text{BASE}(S, \tau))$, meaning each vertex in this set is assigned a fraction of $1/|\text{REPS}(\text{BASE}(S, \tau))|$. We refer to the total growth assigned by active set S to vertex v in its base phase as $\hat{r}_{S,v}^{b''}$.
- If S is in the boost phase, no growth is assigned.
- If S is in the potential phase, define the family \mathcal{F} as:

$$\mathcal{F} = \{S' \in \text{Act}S'_\tau \mid S' \subseteq S\}.$$

We divide the growth of this moment of S proportionally among all vertices in

$$\text{DEPUTY}(\text{SUPERACTIVE}''(S), \bigcup_{S' \in \mathcal{F}} \text{Assignee}'(S', \tau)).$$

Each vertex in this set receives $1/k$ if the size of this set is k . We will later show that this set is non-empty, ensuring that any active set in the potential phase assigns its growth.

We refer to the total growth assigned by active set S to vertex v in its potential phase as $\hat{r}_{S,v}^{p''}$.

- If S is in the final boost phase, similar to the boost phase, no growth is assigned.

We denote \hat{r}_v'' as the total growth assigned to vertex v and $\hat{r}_{S,v}''$ as the total growth assigned to vertex v by set S .

In the following lemma, we demonstrate that each active set in the potential phase assigns its growth to a non-empty set.

Lemma 112. For active set S in the potential phase at moment τ of Extended-Boosted Execution, given $\mathcal{F} = \{S' \in \text{Act}S'_\tau \mid S' \subseteq S\}$,

$$\bigcup_{S' \in \mathcal{F}} \text{Assignee}'(S', \tau)$$

is a non-empty subset of $\text{SUPERACTIVE}''(S)$. Consequently,

$$\text{DEPUTY}(\text{SUPERACTIVE}''(S), \bigcup_{S' \in \mathcal{F}} \text{Assignee}'(S', \tau))$$

is defined and non-empty.

Proof. Since S is in the potential phase, there must exist a vertex $v \in S$ that is in the potential phase. Since v is in the potential phase, it is in an active set at moment τ in Extended Execution. Consider the set $S' \in \text{Act}S'_\tau$ that contains v . Then, by Corollary 53, $S' \subseteq S$ and therefore $S' \in \mathcal{F}$. Additionally, any $S' \in \mathcal{F}$ cannot be in the base or boost phase, and must be in the potential phase.

since \mathcal{F} is non-empty and each set $S' \in \mathcal{F}$ is in the potential phase,

$$\bigcup_{S' \in \mathcal{F}} \text{Assignee}'(S', \tau)$$

is non-empty.

Additionally, for any $S' \in \mathcal{F}$, $Assignee'(S', \tau) \subseteq \text{SUPERACTIVE}'(S')$. Furthermore, Lemma 16, implies that $\text{SUPERACTIVE}'(S') \subseteq \text{SUPERACTIVE}''(S)$ since Extended-Boosted Execution has a larger fingerprint. Therefore,

$$\bigcup_{S' \in \mathcal{F}} Assignee'(S', \tau) \subseteq \bigcup_{S' \in \mathcal{F}} \text{SUPERACTIVE}'(S') \subseteq \text{SUPERACTIVE}''(S).$$

Therefore, $\text{DEPUTY}(\text{SUPERACTIVE}''(S), \bigcup_{S' \in \mathcal{F}} Assignee'(S', \tau))$ is a valid use of DEPUTY . Finally, Lemma 110 shows that since $\bigcup_{S' \in \mathcal{F}} Assignee'(S', \tau)$ is non-empty, $\text{DEPUTY}(\text{SUPERACTIVE}''(S), \bigcup_{S' \in \mathcal{F}} Assignee'(S', \tau))$ is also non-empty. \square

We use the following bounds on the total growth during Extended-Boosted Execution.

Lemma 113. The total growth in Extended-Boosted Execution can be bounded as follows:

$$\sum_{S \subseteq V} y''_S = \sum_{S \subseteq V} y'_S - \text{win}_2 + \text{loss}_2.$$

Proof. Since y'' values are obtained after a local search, this follows from Corollary 60 with $y = y'$ and $y^* = y''$. \square

Lemma 114. The total growth in Extended-Boosted Execution can be bounded as follows:

$$\sum_{S \subseteq V} y''_S \leq \sum_{v \in V} \hat{r}''_v + \text{loss}_1 + \text{loss}_2.$$

Proof. First, we claim that

$$\sum_{S \subseteq V} y''_S \leq \sum_{v \in V} \hat{r}''_v + y_{boost}^+ + y_{boost}''. \quad (7)$$

Consider the moment τ of Extended-Boosted Execution. Let $S \in \text{Act}S''_\tau$ be an active set. There are three cases for S .

- S is in the base or potential phase. In this case, S assigns its growth with fraction one to a subset of its vertices. Therefore, the growth of S is calculated in both y''_S and $\sum_{v \in V} \hat{r}''_v$.
- S is in the boost phase. For any vertex $v \in S$, we have $t_v^- \leq \tau$ and there exists a vertex $u \in S$ such that $\tau \leq t_u^+$. This means that u is also active at the same time in Boosted Execution, in an active set S' . Since the fingerprint t'' is larger than t^+ , by Lemma 13, and because u appears in both S and S' , we conclude that $S' \subseteq S$. For any vertex $v \in S' \subseteq S$, we also have $t_v^- \leq \tau$, and for u , we have $\tau \leq t_u^+$. Thus, S' is in the boost phase.

On one hand, we know that the growth of this moment for S is calculated in y''_S . On the other hand, the growth of active set S' is calculated in y_{boost}^+ . Considering that active sets at a given moment are mutually exclusive and for all $S, S' \subseteq S$, this growth is calculated in both y''_S and y_{boost}^+ , with no growth being calculated multiple times.

- S is in the final boost phase. The growth of S is calculated in both y''_S and y_{boost}'' .

Therefore, since the growth of each active set at any moment is calculated in one term on the right-hand side of inequality 7, we have the claim proved.

Finally, applying Lemma 61 for both y_{boost}^+ and y_{boost}'' proves that

$$\sum_{S \subseteq V} y_S'' \leq \sum_{v \in V} \hat{r}_v'' + loss_1 + loss_2.$$

□

We now prove the following lemmas regarding the relation between the assignment of Extended Execution and Extended-Boosted Execution.

Lemma 115. In assignment \hat{r}'' , and any vertex v ,

$$\sum_{S \subseteq V} \hat{r}_{S,v}^{b''} \leq \sum_{S \subseteq V} \hat{r}_{S,v}^{b'}.$$

Proof. At time τ during Extended-Boosted Execution, consider active set $S \in ActS_\tau''$ in its base phase that assigns a fraction $0 < x \leq 1$ of its growth to vertex v . To prove the inequality, we show that for any such set S , there exists a corresponding active set $S' \in ActS_\tau'$ in Extended Execution, also in its base phase, that assigns a fraction $y \geq x$ of its growth to v at moment τ . Moreover, all such sets S' must be distinct members of $ActS_\tau'$ at this time.

We know that S is in its base phase. At the same moment in Extended Execution, vertex v is also in the base phase, so there exists an active set S' in its base phase that includes v . By Corollary 53, we have $S' \subseteq S$. Additionally, by Lemma 32, $BASE(S', \tau) \subseteq BASE(S, \tau)$.

Since S assigns growth to v in the base phase, v must be in $REPS(BASE(S, \tau))$. Then, by Lemma 37, we know that $v \in REPS(BASE(S', \tau))$, so S' also assigns growth to v in Extended Execution.

Furthermore, according to Lemma 38, we have

$$|REPS(BASE(S', \tau))| \leq |REPS(BASE(S, \tau))|.$$

Since the growth is assigned proportionally across vertices, this implies that the fraction assigned to v by S' is larger than or equal to the fraction assigned by S . □

Next, we prove a similar lemma for the growth assigned by active sets during their potential phase. The difference here is that instead of considering a single vertex, we consider the intersection of a maximal superactive set and a connected component of the optimal solution.

Lemma 116. For the intersection of any connected component C of the optimal solution and any maximal superactive set $S \in \mathcal{M}''$ in Extended-Boosted Execution, we have

$$\sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}_{S,v}^{p''} \leq \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}_{S,v}^{p'}.$$

Proof. At any moment τ , for any active set $S' \in ActS_\tau''$ in the potential phase that assigns a fraction x of growth to a vertex $v \in S \cap C$ in Extended-Boosted Execution, we find an active set $S'' \in ActS_\tau'$ in the potential phase such that $S'' \subseteq S'$, and S'' assigns a fraction $y \geq x$ of growth to a vertex $u \in S \cap C$ in Extended Execution. Since each active set only assigns growth to one vertex in C , and active sets at moment τ are disjoint, this is sufficient to prove our claim.

Since S' is in the potential phase and assigns growth to v , Definitions 109 and 111 imply that $v \in \text{SUPERACTIVE}''(S')$ and there exists a vertex u in $\text{Assignee}'(S'', \tau)$ such that:

- $u \in C$,
- $S'' \subseteq S$,
- $S'' \in \text{Act}S'_\tau$ is an active set in the potential phase of Extended Execution, and
- S'' assigns a fraction of its growth to u . It follows that u is in its potential phase.

We observe that both u and v must be active at moment τ in Extended-Boosted Execution, since u is in its potential phase and $v \in \text{SUPERACTIVE}''(S')$. Combined with the fact that S is the maximal superactive set containing v , it follows that $u \in S$. Now, it remains to show that S'' assigns a larger fraction of its growth to u compared to the fraction that S' assigns to v .

Let

$$U = \text{SUPERACTIVE}''(S')$$

and

$$H = \bigcup_{S^* \in \mathcal{F}} \text{Assignee}'(S^*, \tau),$$

where

$$\mathcal{F} = \{S^* \in \text{Act}S'_\tau \mid S^* \subseteq S'\}.$$

By Lemma 112, we know that $H \subseteq U$. Also, by Lemma 110, we have

$$|\text{DEPUTY}(U, H)| \geq |\{\text{COMP}(w) \mid w \in H\}|.$$

Since in assignment \hat{r}' , S'' assigns its growth to at most one vertex of each component,

$$|\text{Assignee}'(S'', \tau)| = |\{\text{COMP}(w) \mid w \in \text{Assignee}'(S'', \tau)\}|.$$

Since $\text{Assignee}'(S'', \tau) \subseteq H$, it follows that

$$|\text{Assignee}'(S'', \tau)| \leq |\text{DEPUTY}(U, H)|.$$

Therefore, at time τ , the growth fraction assigned by $S'' \subseteq S'$ to $u \in S \cap C$ is larger than the growth fraction assigned by S' to v , as both assign their growth proportionally. This completes the proof. \square

We proceed by combining Lemma 115 and Lemma 116.

Lemma 117. For the intersection of any connected component C of the optimal solution and any maximal superactive set $S \in \mathcal{M}''$ in Extended-Boosted Execution, we have

$$\sum_{v \in S \cap C} \hat{r}''_v \leq \sum_{v \in S \cap C} \hat{r}'_v.$$

Proof. We expand the left-hand side of the inequality as follows:

$$\begin{aligned}
\sum_{v \in S \cap C} \hat{r}'_v &= \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}'_{S',v} \\
&= \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{b''}_{S',v} + \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{p''}_{S',v} && \text{(Definition 111)} \\
&\leq \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{b'}_{S',v} + \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{p''}_{S',v} && \text{(Lemma 115 for all } v \in S \cap C) \\
&\leq \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{b'}_{S',v} + \sum_{v \in S \cap C} \sum_{S' \subseteq V} \hat{r}^{p'}_{S',v} && \text{(Lemma 116)} \\
&= \sum_{v \in S \cap C} \hat{r}'_v. && \text{(Definition 100)}
\end{aligned}$$

Consequently, the proof is complete. \square

Next, we prove that the family of maximal superactive sets in Extended Execution is a refinement of the family of superactive sets in Extended-Boosted Execution. This also holds when we observe the intersection of the sets in each family with any connected component of the optimal solution.

Lemma 118. Let U be the intersection of any connected component C of the optimal solution and any maximal superactive set $S \in \mathcal{M}''$ in Extended-Boosted Execution. Then, U can be decomposed into intersections of maximal superactive sets in Extended Execution and C , i.e., there exists a family $\mathcal{F} \subseteq \mathcal{M}'$ such that

$$U = \bigcup_{S' \in \mathcal{F}} (C \cap S').$$

Furthermore, \mathcal{F} includes all sets in \mathcal{M}' that intersect U non-trivially.

Proof. Let $\mathcal{F} \subseteq \mathcal{M}'$ be the family of maximal superactive sets in Extended Execution that intersect S . We claim that \mathcal{F} is a partition of S . To see this, note that \mathcal{F} is pairwise disjoint and covers S since \mathcal{M}' forms a partition over V .

Assume, for contradiction, that there exists $S' \in \mathcal{F}$ such that S' contains a vertex $v \in S$ and a vertex $u \notin S$. By Corollary 16, the superactive sets of Extended Execution are a refinement of the superactive sets of Extended-Boosted Execution, as this execution has a larger fingerprint. Therefore, at some moment, v and u must both belong to the same superactive set in Extended-Boosted Execution. The laminarity of superactive sets then leads to a contradiction with the maximality of S .

Now, since

$$S = \bigcup_{S' \in \mathcal{F}} S'$$

and $U = C \cap S$, we have

$$U = \bigcup_{S' \in \mathcal{F}} (C \cap S'),$$

which completes the proof. \square

We observe that since \mathcal{M}'' forms a partition over V , we can decompose any component C of OPT into intersections of C with maximal superactive sets in \mathcal{M}'' . Noting that one of these intersections is $LG(C)$, we can also write $C \setminus LG(C)$ as a union of intersections of C with maximal superactive sets in \mathcal{M}'' by putting aside the superactive set corresponding with $LG(C)$. Additionally, the intersection of each superactive

set in the \mathcal{M}' with C can be decomposed into intersections of superactive sets in \mathcal{M}' with C by Lemma 118. This implies that C and $C \setminus LG(C)$ can be further decomposed into intersections of C and superactive sets in \mathcal{M}' .

Based on this observation, we can derive the following corollaries from Lemma 105, Lemma 104, and Lemma 117.

Corollary 119. For any component C of the optimal solution, we have

$$\sum_{v \in C} \hat{r}'_v \leq (1 + \epsilon) \sum_{v \in C} \hat{r}^+_v.$$

Corollary 120. For any component C of the optimal solution, we have

$$\sum_{v \in C \setminus LG(C)} \hat{r}'_v \leq (1 + \epsilon) \sum_{v \in C \setminus LG(C)} \hat{r}^+_v.$$

Corollary 121. For any component C of the optimal solution, the total growth assigned to vertices of $C \setminus LG(C)$ in the potential phase of active sets in Extended Execution is exactly

$$\epsilon \sum_{v \in C \setminus LG(C)} \hat{r}^+_v.$$

Corollary 122. For any component C of the optimal solution, we have

$$\sum_{v \in C} \hat{r}''_v \leq \sum_{v \in C} \hat{r}'_v.$$

Corollary 123. For any component C of the optimal solution, we have

$$\sum_{v \in C \setminus LG(C)} \hat{r}''_v \leq \sum_{v \in C \setminus LG(C)} \hat{r}'_v.$$

Next, we provide bounds for the maximum assigned value to vertices in $LG(C)$ in both Extended Execution and Extended-Boosted Execution.

Lemma 124. For any connected component C of the optimal solution,

$$\hat{r}'_{\max}(LG(C)) \leq \hat{r}^+_{\max}(C) + \sum_{v \in LG(C)} \epsilon \hat{r}^+_v$$

Proof. Let vertex v^* be the vertex with the maximum \hat{r}' value in $LG(C)$. By Lemma 103, the total growth assigned to v^* in the base phase of Extended Execution is at most $\hat{r}^+_{v^*}$, and since $v^* \in C$, we have $\hat{r}^+_{v^*} \leq \hat{r}^+_{\max}(C)$.

We know that $LG(C)$ is the intersection of a maximal superactive set in \mathcal{M}' with C . By Lemma 118, we can decompose $LG(C)$ into intersections of maximal superactive sets in \mathcal{M}' with C . Therefore, applying Lemma 104 to the superactive sets in this decomposition, the total growth assigned to vertices in $LG(C)$ during the potential phase is

$$\epsilon \sum_{v \in LG(C)} \hat{r}^+_v.$$

This is also an upper bound for the growth assigned to v^* during the potential phase, since $v^* \in LG(C)$. Therefore, we have

$$\hat{r}'_{\max}(LG(C)) = \hat{r}'_{v^*} \leq \hat{r}^+_{\max}(C) + \sum_{v \in LG(C)} \epsilon \hat{r}^+_v.$$

□

Lemma 125. For any connected component C of the optimal solution,

$$\hat{r}_{\max}''(LG(C)) \leq \hat{r}_{\max}^+(C) + \sum_{v \in LG(C)} \epsilon \hat{r}_v^+$$

Proof. Consider a vertex v^* in $LG(C)$ such that $\hat{r}_{v^*}'' = \hat{r}_{\max}''(LG(C))$. We can bound the growth assigned to v^* in the base phase as follows:

$$\sum_{S \subseteq V} \hat{r}_{S,v^*}^{b''} \leq \sum_{S \subseteq V} \hat{r}_{S,v^*}^{b'} \quad (\text{Lemma 115})$$

$$\leq \hat{r}_{v^*}^+. \quad (\text{Lemma 103})$$

On the other hand, we can upper bound the growth assigned to v^* in the potential phase by the total growth assigned to any vertex in $LG(C)$. Then, we have

$$\sum_{S \subseteq V} \hat{r}_{S,v^*}^{p''} \leq \sum_{v \in LG(C)} \sum_{S \subseteq V} \hat{r}_{S,v}^{p''}$$

Since $LG(C)$ is the intersection of a superactive set in Extended-Boosted Execution, we can apply Lemma 116 to obtain

$$\sum_{S \subseteq V} \hat{r}_{S,v^*}^{p''} \leq \sum_{v \in LG(C)} \sum_{S \subseteq V} \hat{r}_{S,v}^{p'}$$

Furthermore, by Lemma 118, we can decompose $LG(C)$ into intersections of superactive sets in Extended Execution and C , so we can apply Lemma 104

$$\begin{aligned} \sum_{S \subseteq V} \hat{r}_{S,v^*}^{p''} &\leq \sum_{v \in LG(C)} \epsilon \hat{r}_v^+ \\ &= \epsilon \sum_{v \in LG(C)} \hat{r}_v^+. \end{aligned}$$

Finally, we have:

$$\begin{aligned} \hat{r}_{\max}''(LG(C)) &= \hat{r}_{v^*}'' \\ &= \sum_{S \subseteq V} \hat{r}_{S,v^*}'' \\ &= \sum_{S \subseteq V} \hat{r}_{S,v^*}^{b''} + \sum_{S \subseteq V} \hat{r}_{S,v^*}^{p''} \\ &\leq \hat{r}_{v^*}^+ + \epsilon \sum_{v \in LG(C)} \hat{r}_v^+ \\ &\leq \hat{r}_{\max}^+(C) + \epsilon \sum_{v \in LG(C)} \hat{r}_v^+. \quad (v^* \in C) \end{aligned}$$

□

The following lemma asserts that any vertex assigned growth by an active set in Extended Execution has the highest priority among all vertices in the intersection of the active set, the connected component of the optimal solution it belongs to, and the maximal superactive set it is contained in.

Lemma 126. For any connected component C of the optimal solution and superactive set S' in Extended Execution, if $\hat{r}'_{S,v} > 0$ for active set S and vertex $v \in C \cap S'$, we have

$$v = \operatorname{argmax}_{u \in S \cap C \cap S'} \operatorname{priority}_u.$$

Proof. Consider a moment τ when active set S assigns its growth to vertex v . Since S is assigning growth at moment τ , it is either in the base phase or in the potential phase.

First, assume S is in the base phase. In this case, S assigns its growth to vertices in $\operatorname{REPS}(\operatorname{BASE}(S, \tau))$. By Lemma 36, it follows that $v = \operatorname{argmax}_{w \in S \cap C} \operatorname{priority}_w$. Consequently, we have $v = \operatorname{argmax}_{w \in S \cap C \cap S'} \operatorname{priority}_w$.

Now, assume that S is in the potential phase. For contradiction, suppose there exists a vertex $u \in S \cap C \cap S'$ such that $\operatorname{priority}_u > \operatorname{priority}_v$. Since $u \in S'$, v and u are first connected while they are both active. Therefore, the potential capacity of v must be transferred to u , as $\operatorname{priority}_u > \operatorname{priority}_v$. However, this implies that $\rho_v = 0$ at moment τ , meaning that v cannot be assigned growth by S . This leads to a contradiction, so our assumption must be false.

Thus, $v = \operatorname{argmax}_{u \in S \cap C \cap S'} \operatorname{priority}_u$, as required. \square

Lemma 127. For any connected component C of the optimal solution, considering assignment \hat{r}'' , for an active set S , if $\hat{r}''_{S,v} > 0$ for some $v \in LG(C)$, we have

$$v = \operatorname{argmax}_{u \in S \cap LG(C)} \operatorname{priority}_u.$$

Consider a moment τ when active set S assigns its growth to v . Since S is assigning growth at moment τ , it is either in the base phase or the potential phase.

First, if S is in the base phase, it assigns its growth to vertices in $\operatorname{REPS}(\operatorname{BASE}(S, \tau))$. Therefore, by Lemma 36, $v = \operatorname{argmax}_{w \in S \cap C} \operatorname{priority}_w$ and therefore $v = \operatorname{argmax}_{w \in S \cap C \cap S'} \operatorname{priority}_w$.

Otherwise, S is in the potential phase. Let $\mathcal{F} = \{S' \mid S' \subseteq S, S' \in \operatorname{Act}S'_\tau\}$. Then, growth is assigned to vertices in

$$\operatorname{DEPUTY}(\operatorname{SUPERACTIVE}''(S), \bigcup_{S' \in \mathcal{F}} \operatorname{Assignee}'(S', \tau)).$$

Now, assume for contradiction that there exists $u \in S \cap LG(C)$ such that $\operatorname{priority}_u > \operatorname{priority}_v$. Since u and v are both in $LG(C)$, they are connected actively in Extended-Boosted Execution. Since $v \in \operatorname{SUPERACTIVE}''(S)$, u must also be in $\operatorname{SUPERACTIVE}''(S)$ because u and v are deactivated at the same moment. However, if $u \in \operatorname{SUPERACTIVE}''(S)$, then

$$\operatorname{DEPUTY}(\operatorname{SUPERACTIVE}''(S), \bigcup_{S' \in \mathcal{F}} \operatorname{Assignee}'(S', \tau))$$

cannot include v , because $\operatorname{priority}_u > \operatorname{priority}_v$ and $u \in \operatorname{SUPERACTIVE}''(S) \cap \operatorname{COMP}(v)$.

7.5 Cost Analysis of SOL_{XT}

Component Specific Bounds. To bound the cost of the solution after extension, SOL_{XT} , we analyze components of the optimal solution OPT one by one. For any component C of OPT , we recall that T_C is used to refer to the tree in OPT that contains C .

First, we provide the following bounds for components in \mathcal{A} .

Lemma 128. For any component C of OPT, if $C \in \mathcal{A}$, we have

$$\sum_{v \in C} \hat{r}'_v \leq (1 + \epsilon) \sum_{v \in C} \hat{r}^+_v \leq (1 + \epsilon)(1 - \lambda)c(T_C).$$

Proof. This follows from Corollary 119 and Definition 82. □

The second bound follows from the previous bound and Corollary 122.

Corollary 129. For any component C of OPT, if $C \in \mathcal{A}$, we have

$$\sum_{v \in C} \hat{r}''_v \leq (1 + \epsilon) \sum_{v \in C} \hat{r}^+_v \leq (1 + \epsilon)(1 - \lambda)c(T_C).$$

Next, we focus on components in \mathcal{B} . We provide several different lower bounds for these components and combine them to produce our final bound.

For each component C in \mathcal{B} , our bounds utilize the partitioning of C into $LG(C)$ and $C \setminus LG(C)$. For vertices in $C \setminus LG(C)$, we define the following value, which is utilized in several bounds.

Definition 130. We use $X(C)$ to denote the total growth assigned to vertices in $C \setminus LG(C)$ in assignment \hat{r}' corresponding to active sets S cutting at least two edges of T_C . That is,

$$X(C) = \sum_{v \in C \setminus LG(C)} \sum_{\substack{S \subseteq V \\ |\delta(S) \cap T_C| > 1}} \hat{r}'_{S,v}.$$

The following lemma establishes that vertices in $C \setminus LG(C)$ cannot connect to vertices in $LG(C)$ while they are active in Extended Execution. This is later used to identify active sets that cut T_C .

Lemma 131. Consider any component C of OPT, and let S be any active set assigning growth to a vertex v in $C \setminus LG(C)$ during Extended Execution, i.e., $\hat{r}'_{S,v} > 0$ for some $v \in C \setminus LG(C)$. Then, S does not contain any vertex in $LG(C)$.

Proof. Assume, for contradiction, that S contains a vertex $u \in LG(C)$ while $\hat{r}'_{S,v} > 0$ for some $v \in C \setminus LG(C)$. Since $\hat{r}'_{S,v} > 0$, we must have $v \in \text{SUPERACTIVE}'(S)$, and v is active at moment τ . By Lemma 14, v is also active at moment τ in Extended-Boosted Execution, since fingerprint t'' is larger than t' .

Let S' be the active set containing v at moment τ in Extended-Boosted Execution. By Corollary 53, we know that $S \subseteq S'$, and therefore $u \in S'$. Since $u \in LG(C)$, u cannot be deactivated before v in Extended-Boosted Execution. Consequently, u is also active at moment τ . This implies that u and v are connected while active, meaning they will be deactivated at the same moment. Hence, u can only be in $LG(C)$ if $v \in LG(C)$. However, we assumed that $v \in C \setminus LG(C)$ while $u \in LG(C)$, which leads to a contradiction. □

In the following lemma, during Extended Execution, we identify a class of active sets that cut a connected component of the optimal solution.

Lemma 132. For any component C of OPT, there exists a vertex v^* in $LG(C)$, such that for any active set S assigning growth to a vertex $v \in C \setminus \{v^*\}$ during Extended Execution, S cuts T_C .

Proof. Consider a vertex u in $LG(C)$ that remains active the longest in Extended Execution. Let S' be the maximal superactive set containing u , and let v^* be the vertex in $S' \cap C$ with the highest priority, i.e.,

$$v^* = \operatorname{argmax}_{w \in S' \cap C} \operatorname{priority}_w.$$

Since both u and v^* belong to S' , they are deactivated at the same moment during Extended Execution. As a result, whenever any vertex in $LG(C)$ is active in Extended Execution, v^* must also be active. Additionally, it follows from Lemma 118 that $S' \cap C \subseteq LG(C)$ and consequently $v^* \in LG(C)$.

Now, consider any active set S such that $\hat{r}'_{S,v} > 0$ for a vertex $v \in C \setminus \{v^*\}$. We show that S cannot include v^* . If $v \in C \setminus LG(C)$, this results from Lemma 131. Otherwise, if $v \in LG(C)$, assume for contradiction that $v^* \in S$. Since v is assigned growth by S , it must be active at this moment. Since v^* is active as long as any other vertex in $LG(C)$, it follows that v and v^* would connect while active. Therefore, v must also belong to the maximal superactive set S' . Now, by definition, $v^* = \operatorname{argmax}_{w \in S' \cap C} \operatorname{priority}_w$ and thus

$$v^* = \operatorname{argmax}_{w \in S \cap C \cap S'} \operatorname{priority}_w.$$

Thus, it follows from Lemma 126 that S cannot assign growth to v since $v \in S \cap C \cap S'$ and $v \neq v^*$. This is a contradiction, so v^* must not belong to S .

Finally, since S includes v but not v^* , and v and v^* are connected by T_C , it follows that S must cut T_C . \square

The following lemma presents our first bound for the cost of components C in \mathcal{B} . In this bound, we consider how the sets contributing to $X(C)$ color at least two edges of the tree T_C .

Lemma 133. For any component $C \in \mathcal{B}$, we have

$$\sum_{v \in C} \hat{r}'_v + X(C) - \hat{r}'_{\max}(LG(C)) \leq c(T_C).$$

Proof. We can lower bound $c(T_C)$ as follows:

$$c(T_C) \geq \sum_{S \subseteq V} |\delta(S) \cap T_C| \cdot y'_S. \quad (\text{Lemma 25})$$

Breaking down the sum based on $|\delta(S) \cap T_C|$, we get

$$c(T_C) \geq \sum_{\substack{S \subseteq V \\ S \not\cap T_C}} y'_S + \sum_{\substack{S \subseteq V \\ |\delta(S) \cap T_C| > 1}} y'_S.$$

Next, we lower bound y'_S values by their contribution to \hat{r}' values for subsets of V . Take the vertex $v^* \in LG(C)$ considered in Lemma 132. For the first summation term, we use the subset $C \setminus \{v^*\}$, since by Lemma 132, any set assigning growth to a vertex in this subset in Extended Execution must cut T_C . In the second summation,

we use $C \setminus LG(C)$ to find a term matching the definition of $X(C)$.

$$\begin{aligned}
c(T_C) &\geq \sum_{\substack{S \subseteq V \\ S \cap T_C}} \sum_{v \in C \setminus \{v^*\}} \hat{r}'_{S,v} + \sum_{\substack{S \subseteq V \\ |\delta(S) \cap T_C| > 1}} \sum_{v \in C \setminus LG(C)} \hat{r}'_{S,v} && (y'_S \geq \sum_{v \in V_1} \hat{r}'_{S,v} \text{ for any subset } V_1) \\
&\geq \sum_{S \subseteq V} \sum_{v \in C \setminus \{v^*\}} \hat{r}'_{S,v} + \sum_{\substack{S \subseteq V \\ |\delta(S) \cap T_C| > 1}} \sum_{v \in C \setminus LG(C)} \hat{r}'_{S,v} && \text{(Lemma 132)} \\
&\geq \sum_{v \in C \setminus \{v^*\}} \sum_{S \subseteq V} \hat{r}'_{S,v} + \sum_{v \in C \setminus LG(C)} \sum_{\substack{S \subseteq V \\ |\delta(S) \cap T_C| > 1}} \hat{r}'_{S,v} && \text{(Change order of summations)} \\
&= \sum_{v \in C \setminus \{v^*\}} \hat{r}'_v + \sum_{v \in C \setminus LG(C)} \sum_{\substack{S \subseteq V \\ |\delta(S) \cap T_C| > 1}} \hat{r}'_{S,v} && (\sum_{S \subseteq V} \hat{r}'_{S,v} = \hat{r}'_v \text{ for any vertex } v) \\
&\geq \sum_{v \in C \setminus \{v^*\}} \hat{r}'_v + X(C) && \text{(Definition 130)} \\
&= \sum_{v \in C} \hat{r}'_v + X(C) - \hat{r}'_{v^*}.
\end{aligned}$$

Finally, since $v^* \in LG(C)$, we have $\hat{r}'_{v^*} \leq \hat{r}'_{\max}(LG(C))$, and we can conclude that

$$\sum_{v \in C} \hat{r}'_v + X(C) - \hat{r}'_{\max}(LG(C)) \leq c(T_C)$$

as desired. \square

In the following lemma, during Extended Execution, we identify a class of active sets that cut either zero or at least two edges of $T_{LG(C)}$ for a connected component C of the optimal solution.

Lemma 134. Consider any component C of OPT, and let S be any active set assigning growth to a vertex v in $C \setminus LG(C)$ during Extended Execution, i.e., $\hat{r}'_{S,v} > 0$ for some $v \in C \setminus LG(C)$. Then, S cannot cut exactly one edge of $T_{LG(C)}$.

Proof. Assume, for contradiction, that S cuts exactly one edge of $T_{LG(C)}$. By Lemma 22, cutting exactly one edge of $T_{LG(C)}$ implies that S must cut one of the leaves of $T_{LG(C)}$. Therefore, S must include at least one of the leaves of $T_{LG(C)}$.

Since $T_{LG(C)}$ is the minimal tree connecting the leaves in T_C , the leaves of $T_{LG(C)}$ must be in the connected component $LG(C)$ of T_C . However, by Lemma 131, an active set assigning growth to a vertex in $C \setminus LG(C)$ during Extended Execution cannot include any vertex from $LG(C)$. This leads to a contradiction. Therefore, S cannot cut exactly one edge of $T_{LG(C)}$. \square

We can now prove our second lower bound on the cost of component $C \in \mathcal{B}$, which utilizes Lemma 90 to bound the cost of $T_{LG(C)}$, and lower bounds the coloring on the rest of T_C using the fact that growth assigned to vertices in $C \setminus LG(C)$ cannot contribute to coloring exactly one edge of $T_{LG(C)}$. Figure 9 illustrates this partition of T_C .

Lemma 135. For any component $C \in \mathcal{B}$, we have

$$\frac{6}{5 + \beta} \left[\sum_{v \in LG(C)} \hat{r}''_v - \hat{r}''_{\max}(LG(C)) \right] + \sum_{v \in C \setminus LG(C)} \hat{r}''_v - X(C) \leq c(T_C)$$

Furthermore, any active set with $\sum_{v \in C \setminus LG(C)} \hat{r}'_{S,v} > 0$ must include a vertex in $C \setminus LG(C)$, and no vertex in $LG(C)$ by Lemma 131, and must therefore cut T_C , which connects $C \setminus LG(C)$ and $LG(C)$. Thus, we can simplify the bound for the first summation to get

$$\begin{aligned}
c(T_C \setminus T_{LG(C)}) &\geq \sum_{S \subseteq V} \sum_{v \in C \setminus LG(C)} \hat{r}'_{S,v} - \sum_{\substack{S \subseteq V \\ |\delta(S) \cap T_C| > 1}} \sum_{v \in C \setminus LG(C)} \hat{r}'_{S,v} \\
&= \sum_{S \subseteq V} \sum_{v \in C \setminus LG(C)} \hat{r}'_{S,v} - X(C) && \text{(Definition 130)} \\
&\geq \sum_{v \in C \setminus LG(C)} \sum_{S \subseteq V} \hat{r}'_{S,v} - X(C) && \text{(Change order of summation)} \\
&= \sum_{v \in C \setminus LG(C)} \hat{r}'_v - X(C) && (\sum_{S \subseteq V} \hat{r}'_{S,v} = \hat{r}'_v \text{ for any } v) \\
&\geq \sum_{v \in C \setminus LG(C)} \hat{r}''_v - X(C). && \text{(Corollary 123)}
\end{aligned}$$

Finally, the desired inequality is found by combining the bounds on $c(T_{LG(C)})$ and $c(T_C \setminus T_{LG(C)})$ since $c(T_C) = c(T_C \setminus T_{LG(C)}) + c(T_{LG(C)})$. \square

The following lemma includes our third bound for the cost of component $C \in \mathcal{B}$, which does not depend on $X(C)$.

Lemma 136. For any component $C \in \mathcal{B}$, we have

$$\sum_{v \in C} \hat{r}''_v - \sum_{v \in LG(C)} \hat{r}''_v \leq (1 + \epsilon)c(T_C) - (1 + \epsilon) \sum_{v \in LG(C)} \hat{r}^+_v.$$

Proof. We have

$$\begin{aligned}
\sum_{v \in C} \hat{r}''_v - \sum_{v \in LG(C)} \hat{r}''_v &= \sum_{v \in C \setminus LG(C)} \hat{r}''_v \\
&\leq \sum_{v \in C \setminus LG(C)} \hat{r}'_v && \text{(Lemma 123)} \\
&\leq (1 + \epsilon) \sum_{v \in C \setminus LG(C)} \hat{r}^+_v && \text{(Corollary 120)} \\
&= (1 + \epsilon) \left(\sum_{v \in C} \hat{r}^+_v - \sum_{v \in LG(C)} \hat{r}^+_v \right) \\
&\leq (1 + \epsilon)c(T_C) - (1 + \epsilon) \sum_{v \in LG(C)} \hat{r}^+_v. && \text{(Lemma 80)}
\end{aligned}$$

\square

The next set of lemmas helps us bound the value of $X(C)$ for any component C of OPT.

Lemma 137. For any active set S in the potential phase of Extended Execution, we have

$$\text{UNSATISFIED}(S) = \emptyset.$$

Proof. Assume otherwise that there exists a vertex $v \in S$ such that $\text{PAIR}_v \notin S$. Let τ be a moment when S is in the potential phase. Since S is in the potential phase, we must have $t_v^+ \leq \tau$. Let S' be the set containing v in Legacy Execution at moment τ . By Corollary 53 and Corollary 99, it follows that $S' \subseteq S$. However, since $t_v^- \leq t_v^+ \leq \tau$, S' must contain both v and PAIR_v , because v and PAIR_v are connected by t_v^- in Legacy Execution. This implies that $\text{PAIR}_v \in S' \subseteq S$, which is a contradiction. \square

Lemma 138. For any component C of OPT, any active set in the potential phase of Extended Execution cannot cut exactly one edge of T_C .

Proof. Assume for contradiction that an active set S in the potential phase at moment τ cuts exactly one edge of T_C . By Lemma 24, removing this single edge from T_C can only disconnect demand pairs in C that are cut by S , which correspond precisely to $\text{UNSATISFIED}(S)$. However, since S is in the potential phase, Lemma 137 implies that $\text{UNSATISFIED}(S) = \emptyset$. Therefore, removing this edge from OPT does not disconnect any demand pairs, yielding a valid solution with either a strictly lower cost or fewer edges than OPT. This contradicts the definition of OPT as a minimal optimal solution (see Section 3). \square

Lemma 139. For any component C of OPT, we have

$$X(C) \geq \epsilon \sum_{v \in C \setminus LG(C)} \hat{r}_v^+.$$

Proof. This is trivially true if $C \setminus LG(C)$ is empty. Otherwise, consider any active set in the potential phase assigning growth to a vertex in $C \setminus LG(C)$ during Extended Execution. This active set cannot include any vertex in $LG(C)$ by Lemma 131. Therefore, it must cut T_C .

Moreover, by Lemma 138, it cannot cut exactly one edge of T_C . Thus, by Corollary 121, the growth assigned to vertices in $C \setminus LG(C)$ by sets in the potential phase is at least $\epsilon \sum_{v \in C \setminus LG(C)} \hat{r}_v^+$. Since all of this growth is counted in $X(C)$, we conclude that

$$X(C) \geq \epsilon \sum_{v \in C \setminus LG(C)} \hat{r}_v^+.$$

\square

Overall Bounds. Building on the bounds for the cost of individual components of OPT, we now introduce bounds for the total cost of OPT. To enhance readability, we begin by defining the following notations.

Definition 140. Given the optimal solution OPT, we define the following:

$$\begin{aligned} X_{sum} &= \sum_{C \in \mathcal{B}} X(C) \\ \hat{r}_{LG}^+ &= \sum_{C \in \mathcal{B}} \sum_{v \in LG(C)} \hat{r}_v^+ \\ \hat{r}_{LG}'' &= \sum_{C \in \mathcal{B}} \sum_{v \in LG(C)} \hat{r}_v'' \end{aligned}$$

Lemma 141. We can bound the sum of y'' values as

$$\sum_{S \subseteq V} y_S'' \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - X_{sum} + \epsilon \hat{r}_{LG}^+ + \sum_{C \in \mathcal{B}} r_{max}^+(C) - \beta \text{loss}_2 + \text{loss}_1.$$

Proof. We use Lemma 128 for components in \mathcal{A} and Lemma 133 for components in \mathcal{B} . Then, summing these inequalities over all components of OPT, we get

$$\begin{aligned} \sum_{C \in \mathcal{A}} \sum_{v \in C} \hat{r}'_v + \sum_{C \in \mathcal{B}} \left[X(C) - \hat{r}'_{\max}(LG(C)) + \sum_{v \in C} \hat{r}'_v \right] &\leq \sum_{C \in \mathcal{A}} (1 + \epsilon)(1 - \lambda)c(T_C) + \sum_{C \in \mathcal{B}} c(T_C) \\ &= (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) \end{aligned}$$

Next, gather the sum $\sum_{v \in V} \hat{r}'_v$ on the left-hand side and move the other terms to the right-hand side, which results in

$$\begin{aligned} \sum_{v \in V} \hat{r}'_v &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - \sum_{C \in \mathcal{B}} X(C) + \sum_{C \in \mathcal{B}} \hat{r}'_{\max}(LG(C)) \\ &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - X_{\text{sum}} + \sum_{C \in \mathcal{B}} \hat{r}'_{\max}(LG(C)) && \text{(Definition 140)} \\ &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - X_{\text{sum}} + \sum_{C \in \mathcal{B}} (\hat{r}_{\max}^+(C) + \epsilon \sum_{v \in LG(C)} \hat{r}_v^+) && \text{(Lemma 124)} \\ &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - X_{\text{sum}} + \epsilon \hat{r}_{LG}^+ + \sum_{C \in \mathcal{B}} \hat{r}_{\max}^+(C) && \text{(Definition 140)} \\ &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - X_{\text{sum}} + \epsilon \hat{r}_{LG}^+ + \sum_{C \in \mathcal{B}} r_{\max}^+(C). && \text{(Lemma 79)} \end{aligned}$$

Additionally, we have

$$\begin{aligned} \sum_{S \subseteq V} y''_S &\leq \sum_{S \subseteq V} y'_S - \text{win}_2 + \text{loss}_2 && \text{(Lemma 113)} \\ &\leq \sum_{S \subseteq V} y'_S - (1 + \beta)\text{loss}_2 + \text{loss}_2 && \text{(Definition 50)} \\ &\leq \sum_{v \in V} \hat{r}'_v - \beta \text{loss}_2 + \text{loss}_1. && \text{(Lemma 102)} \end{aligned}$$

Finally, we can combine the inequalities to get

$$\sum_{S \subseteq V} y''_S \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - X_{\text{sum}} + \epsilon \hat{r}_{LG}^+ + \sum_{C \in \mathcal{B}} r_{\max}^+(C) - \beta \text{loss}_2 + \text{loss}_1.$$

□

Lemma 142. We can bound the sum of y'' values as

$$\sum_{S \subseteq V} y''_S \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - \frac{1 - \beta}{5 + \beta} \hat{r}_{LG}'' + \frac{6\epsilon}{5 + \beta} \hat{r}_{LG}^+ + \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} r_{\max}^+(C) + X_{\text{sum}} + \text{loss}_1 + \text{loss}_2.$$

Proof. By Lemma 114, we have

$$\sum_{S \subseteq V} y''_S \leq \sum_{v \in V} \hat{r}''_v + \text{loss}_1 + \text{loss}_2.$$

Therefore, to prove the desired bound, it suffices to show that

$$\sum_{v \in V} \hat{r}''_v \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - \frac{1 - \beta}{5 + \beta} \hat{r}_{LG}'' + \frac{6\epsilon}{5 + \beta} \hat{r}_{LG}^+ + \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} r_{\max}^+(C) + X_{\text{sum}}.$$

We use Corollary 129 for components in \mathcal{A} and Lemma 135 for components in \mathcal{B} . Then, summing these inequalities over all components of OPT, we get

$$\begin{aligned} \sum_{C \in \mathcal{A}} \sum_{v \in C} \hat{r}_v'' + \sum_{C \in \mathcal{B}} \left(\frac{6}{5 + \beta} \left[\sum_{v \in LG(C)} \hat{r}_v'' - \hat{r}_{\max}''(LG(C)) \right] + \sum_{v \in C \setminus LG(C)} \hat{r}_v'' - X(C) \right) &\leq (1 + \epsilon)(1 - \lambda) \sum_{C \in \mathcal{A}} c(T_C) + \sum_{C \in \mathcal{B}} c(T_C) \\ &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) \end{aligned}$$

Then, gathering the sum $\sum_{v \in V} \hat{r}_v''$ on the left-hand side and moving the other terms to right-hand side results in

$$\sum_{v \in V} \hat{r}_v'' \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - \frac{1 - \beta}{5 + \beta} \sum_{C \in \mathcal{B}} \sum_{v \in LG(C)} \hat{r}_v'' + \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} \hat{r}_{\max}''(LG(C)) + \sum_{C \in \mathcal{B}} X(C)$$

which can be rewritten as

$$\sum_{v \in V} \hat{r}_v'' \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - \frac{1 - \beta}{5 + \beta} \hat{r}_{LG}'' + \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} \hat{r}_{\max}''(LG(C)) + X_{sum}$$

using Definition 140. Now, it suffices to show that

$$\frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} \hat{r}_{\max}''(LG(C)) \leq \frac{6\epsilon}{5 + \beta} \hat{r}_{LG}^+ + \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} r_{\max}^+(C)$$

to prove our desired bound. This can be established as follows to complete the proof:

$$\begin{aligned} \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} \hat{r}_{\max}''(LG(C)) &\leq \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} \left[\hat{r}_{\max}^+(C) + \epsilon \sum_{v \in LG(C)} \hat{r}_v^+ \right] && \text{(Lemma 125)} \\ &\leq \frac{6\epsilon}{5 + \beta} \hat{r}_{LG}^+ + \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} \hat{r}_{\max}^+(C) && \text{(Definition 140)} \\ &\leq \frac{6\epsilon}{5 + \beta} \hat{r}_{LG}^+ + \frac{6}{5 + \beta} \sum_{C \in \mathcal{B}} r_{\max}^+(C). && \text{(Lemma 79)} \end{aligned}$$

□

Lemma 143. We can bound the sum of y'' values as

$$\sum_{S \subseteq V} y_S'' \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon)c(\text{OPT}_{\mathcal{B}}) - (1 + \epsilon)\hat{r}_{LG}^+ + \hat{r}_{LG}'' + \text{loss}_1 + \text{loss}_2.$$

Proof. By Lemma 114, we have

$$\sum_{S \subseteq V} y_S'' \leq \sum_{v \in V} \hat{r}_v'' + \text{loss}_1 + \text{loss}_2.$$

Therefore, to prove the desired bound, it suffices to show that

$$\sum_{v \in V} \hat{r}_v'' \leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon)c(\text{OPT}_{\mathcal{B}}) - (1 + \epsilon)\hat{r}_{LG}^+ + \hat{r}_{LG}''.$$

We use Corollary 129 for components in \mathcal{A} and Lemma 136 for components in \mathcal{B} and sum up the inequalities to arrive at the following:

$$\begin{aligned}
\sum_{C \in \mathcal{A}} \sum_{v \in C} \hat{r}_v'' + \sum_{C \in \mathcal{B}} \left(\sum_{v \in C} \hat{r}_v'' - \sum_{v \in LG(C)} \hat{r}_v'' \right) &\leq (1 + \epsilon)(1 - \lambda) \sum_{C \in \mathcal{A}} c(T_C) + \sum_{C \in \mathcal{B}} \left((1 + \epsilon)c(T_C) - (1 + \epsilon) \sum_{v \in LG(C)} \hat{r}_v^+ \right) \\
&= (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon)c(\text{OPT}_{\mathcal{B}}) - (1 + \epsilon) \sum_{C \in \mathcal{B}} \sum_{v \in LG(C)} \hat{r}_v^+ \\
&= (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon)c(\text{OPT}_{\mathcal{B}}) - (1 + \epsilon)\hat{r}_{LG}^+. \tag{Definition 140}
\end{aligned}$$

Then, gathering the sum $\sum_{v \in V} \hat{r}_v''$ on the left-hand side and moving the other terms to right-hand side results in

$$\begin{aligned}
\sum_{v \in V} \hat{r}_v'' &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon)c(\text{OPT}_{\mathcal{B}}) - (1 + \epsilon)\hat{r}_{LG}^+ + \sum_{C \in \mathcal{B}} \sum_{v \in LG(C)} \hat{r}_v'' \\
&= (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon)c(\text{OPT}_{\mathcal{B}}) - (1 + \epsilon)\hat{r}_{LG}^+ + \hat{r}_{LG}'', \tag{Definition 140}
\end{aligned}$$

which completes the proof. \square

Taking a weighted average of the bounds in Lemmas 141 and 142 with weights $1 - w$ and w for $0 \leq w \leq 1$ respectively, we arrive at the following bound.

Corollary 144. For any $0 \leq w \leq 1$, we have

$$\begin{aligned}
\sum_{S \subseteq V} y_S'' &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - w \frac{1 - \beta}{5 + \beta} \hat{r}_{LG}'' + (2w - 1)X_{sum} \\
&\quad + (1 + w \frac{1 - \beta}{5 + \beta})(\epsilon \hat{r}_{LG}^+ + \sum_{C \in \mathcal{B}} r_{max}^+(C)) + loss_1 + (w - \beta + w\beta)loss_2.
\end{aligned}$$

Now, taking $w \leq \frac{1}{2}$ and using the bound on X from Lemma 139, we arrive at our next bound.

Lemma 145. For any $0 \leq w \leq \frac{1}{2}$, we have

$$\begin{aligned}
\sum_{S \subseteq V} y_S'' &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 - (1 - 2w)(1 - \lambda)\epsilon)c(\text{OPT}_{\mathcal{B}}) - w \frac{1 - \beta}{5 + \beta} \hat{r}_{LG}'' \\
&\quad + (2 - w \frac{9 + 3\beta}{5 + \beta})\epsilon \hat{r}_{LG}^+ + (1 + w \frac{1 - \beta}{5 + \beta}) \sum_{C \in \mathcal{B}} r_{max}^+(C) + loss_1 + (w - \beta + w\beta)loss_2.
\end{aligned}$$

Proof. By Corollary 144, we have

$$\begin{aligned}
\sum_{S \subseteq V} y_S'' &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + c(\text{OPT}_{\mathcal{B}}) - w \frac{1 - \beta}{5 + \beta} \hat{r}_{LG}'' + (2w - 1)X_{sum} \\
&\quad + (1 + w \frac{1 - \beta}{5 + \beta})(\epsilon \hat{r}_{LG}^+ + \sum_{C \in \mathcal{B}} r_{max}^+(C)) + loss_1 + (w - \beta + w\beta)loss_2.
\end{aligned}$$

Rewriting X_{sum} as a sum by Definition 140, we can use Lemma 139 to show that

$$\begin{aligned}
X_{sum} &= \sum_{C \in \mathcal{B}} X(C) \\
&\geq \sum_{C \in \mathcal{B}} \sum_{v \in C \setminus LG(C)} \epsilon \hat{r}_v^+ \\
&= \epsilon \sum_{C \in \mathcal{B}} \left[\sum_{v \in C} \hat{r}_v^+ - \sum_{v \in LG(C)} \hat{r}_v^+ \right] \\
&= \epsilon \sum_{C \in \mathcal{B}} \sum_{v \in C} \hat{r}_v^+ - \epsilon \hat{r}_{LG}^+. \tag{Definition 140}
\end{aligned}$$

Additionally, for any $C \in \mathcal{B}$, $\sum_{v \in C} \hat{r}_v^+ \geq (1 - \lambda)c(T_C)$ by Definition 82, which implies

$$\begin{aligned}
X_{sum} &\geq \epsilon \sum_{C \in \mathcal{B}} (1 - \lambda)c(C) - \epsilon \hat{r}_{LG}^+ \\
&= \epsilon(1 - \lambda)c(\text{OPT}_{\mathcal{B}}) - \epsilon \hat{r}_{LG}^+.
\end{aligned}$$

Since $2w - 1 \leq 0$, it follows that

$$(2w - 1)X_{sum} \leq (2w - 1)\epsilon(1 - \lambda)c(\text{OPT}_{\mathcal{B}}) - (2w - 1)\epsilon \hat{r}_{LG}^+.$$

Combining this with the inequality from Corollary 144 completes the proof. \square

Next, we introduce a bound that does not include \hat{r}_{LG}^+ and \hat{r}_{LG}'' terms by taking a convex combination of Lemma 145 and Lemma 143.

Lemma 146. For any $w \leq \frac{1}{2}$ and $0 \leq w' \leq 1$ such that $1/(1 + \frac{1-\beta}{5+\beta}w) \leq w' \leq (1 + \epsilon)/(1 + 3\epsilon - \epsilon w \frac{9+3\beta}{5+\beta})$, we have

$$\begin{aligned}
\sum_{S \subseteq V} y_S'' &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon - \epsilon w'(1 + (1 - 2w)(1 - \lambda)))c(\text{OPT}_{\mathcal{B}}) \\
&\quad + w'(1 + w \frac{1-\beta}{5+\beta}) \sum_{C \in \mathcal{B}} r_{max}^+(C) + loss_1 + (1 - w'(1 + \beta)(1 - w))loss_2.
\end{aligned}$$

Proof. Taking a convex combination of the bounds in Lemma 143 and Lemma 145, with w' as the coefficient for the second inequality results in the following bound:

$$\begin{aligned}
\sum_{S \subseteq V} y_S'' &\leq (1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon - \epsilon w'(1 + (1 - 2w)(1 - \lambda)))c(\text{OPT}_{\mathcal{B}}) \\
&\quad + w'(1 + w \frac{1-\beta}{5+\beta}) \sum_{C \in \mathcal{B}} r_{max}^+(C) + loss_1 + (1 - w'(1 + \beta)(1 - w))loss_2 \\
&\quad + (1 - w' - w'w \frac{1-\beta}{5+\beta})\hat{r}_{LG}'' + \left(w'\epsilon(2 - w \frac{9+3\beta}{5+\beta}) - (1 - w')(1 + \epsilon) \right) \hat{r}_{LG}^+.
\end{aligned}$$

It suffices for the coefficients of \hat{r}_{LG}^+ and \hat{r}_{LG}'' terms to be non-positive to arrive at our desired inequality. We can verify that the coefficient for \hat{r}_{LG}'' is non-positive when

$$1 - w' - w'w \frac{1-\beta}{5+\beta} \leq 0.$$

This can be rearranged into

$$w' \geq 1/(1 + \frac{1-\beta}{5+\beta}w),$$

observing that $1 + \frac{1-\beta}{5+\beta}w$ is always positive.

On the other hand, for the coefficient of \hat{r}_{LG}^+ to be non-positive, we must have

$$w'\epsilon(2 - w\frac{9+3\beta}{5+\beta}) - (1-w')(1+\epsilon) \leq 0.$$

This is equivalent to

$$w'(1 + 3\epsilon - \epsilon w\frac{9+3\beta}{5+\beta}) \leq (1+\epsilon).$$

Noting that $1 + 3\epsilon - \epsilon w\frac{9+3\beta}{5+\beta}$ is always positive since $\epsilon w\frac{9+3\beta}{5+\beta} \leq \frac{1}{2} \cdot 3 \cdot \epsilon < 3\epsilon$, we find the following upper bound for w' :

$$w' \leq (1+\epsilon)/(1 + 3\epsilon - \epsilon w\frac{9+3\beta}{5+\beta}).$$

Therefore, under the condition that $1/(1 + \frac{1-\beta}{5+\beta}w) \leq w' \leq (1+\epsilon)/(1 + 3\epsilon - \epsilon w\frac{9+3\beta}{5+\beta})$, the terms containing \hat{r}_{LG}^+ and \hat{r}_{LG}'' vanish and we arrive at the desired inequality. \square

We use the next two lemmas to upper bound the $loss_1$ and $loss_2$ terms in Lemma 146.

Lemma 147. If SOL_{XT} is not a $2 - 2\alpha$ approximation for the optimal solution OPT, then we must have

$$loss_2 \leq (\alpha + \epsilon)/\beta \cdot c(\text{OPT}) - (1 + \epsilon + \epsilon/\beta) \cdot loss_1.$$

Proof. By Lemma 9, the solution found by the algorithm after the local search has cost at most

$$\begin{aligned} c(SOL_{XT}) &\leq 2 \sum_{S \subseteq V} y_S'' \\ &= 2 \sum_{S \subseteq V} y_S' - 2win_2 + 2loss_2 && \text{(Lemma 113)} \\ &\leq 2 \sum_{v \in V} \hat{r}_v' - 2win_2 + 2loss_1 + 2loss_2. && \text{(Lemma 102)} \end{aligned}$$

Furthermore, we know by Corollary 119 that $\sum_{v \in V} \hat{r}_v' \leq \sum_v (1 + \epsilon)\hat{r}_v^+$. Then, we can say

$$\begin{aligned} c(SOL_{XT}) &\leq 2 \sum_{v \in V} (1 + \epsilon)\hat{r}_v^+ + 2loss_1 - 2win_2 + 2loss_2 \\ &\leq 2(1 + \epsilon)c(\text{OPT}) - 2(1 + \epsilon)win_1 + 2loss_1 - 2win_2 + 2loss_2 && \text{(Lemma 77)} \\ &\leq 2(1 + \epsilon)c(\text{OPT}) - 2(1 + \epsilon)(1 + \beta)loss_1 + 2loss_1 - 2win_2 + 2loss_2 && \text{(Definition 50)} \\ &\leq 2(1 + \epsilon)c(\text{OPT}) - 2(1 + \epsilon)(1 + \beta)loss_1 + 2loss_1 - 2(1 + \beta)loss_2 + 2loss_2 && \text{(Definition 50)} \\ &= 2(1 + \epsilon)c(\text{OPT}) - 2(\epsilon + \beta + \epsilon\beta)loss_1 - 2\beta loss_2. \end{aligned}$$

Finally, since we assumed that $c(SOL_{XT}) \geq (2 - 2\alpha)c(\text{OPT})$, we must have

$$2(1 + \epsilon)c(\text{OPT}) - 2(\epsilon + \beta + \epsilon\beta)loss_1 - 2\beta loss_2 \geq (2 - 2\alpha)c(\text{OPT})$$

and therefore

$$loss_2 \leq (\alpha + \epsilon)/\beta \cdot c(\text{OPT}) - (1 + \epsilon + \epsilon/\beta) \cdot loss_1.$$

\square

Lemma 148. If neither SOL_{LS} nor SOL_{XT} is a $2 - 2\alpha$ approximation for a given instance with optimal solution OPT , then for any coefficient ζ , we have

$$loss_1 + \zeta \cdot loss_2 \leq \max\left(\frac{\alpha}{\beta}, \frac{\zeta(\alpha + \epsilon)}{\beta}, \frac{\alpha + \zeta(1 - \alpha - \alpha/\beta)\epsilon}{\beta}\right)c(OPT).$$

Proof. We begin by observing that $loss_1$ and $loss_2$ are always non-negative. If $\zeta < 0$, we have

$$loss_1 + \zeta \cdot loss_2 \leq loss_1 \leq \frac{\alpha}{\beta}c(OPT)$$

by Lemma 85. Otherwise, we can use Lemma 147 and get

$$\begin{aligned} loss_1 + \zeta \cdot loss_2 &\leq loss_1 + \zeta \left(\frac{\alpha + \epsilon}{\beta}c(OPT) - (1 + \epsilon + \epsilon/\beta)loss_1 \right) \\ &\leq \frac{\zeta(\alpha + \epsilon)}{\beta}c(OPT) + (1 - \zeta(1 + \epsilon + \epsilon/\beta))loss_1. \end{aligned}$$

Then, either $1 - \zeta(1 + \epsilon + \epsilon/\beta) < 0$, in which case we have

$$\begin{aligned} loss_1 + \zeta \cdot loss_2 &\leq \zeta \frac{\alpha + \epsilon}{\beta}c(OPT) + (1 - \zeta(1 + \epsilon + \epsilon/\beta))loss_1 \\ &\leq \frac{\zeta(\alpha + \epsilon)}{\beta}c(OPT) \end{aligned}$$

or $1 - \zeta(1 + \epsilon + \epsilon/\beta) \geq 0$, in which case we can again use Lemma 85 to get

$$\begin{aligned} loss_1 + \zeta \cdot loss_2 &\leq \zeta \frac{\alpha + \epsilon}{\beta}c(OPT) + (1 - \zeta(1 + \epsilon + \epsilon/\beta))loss_1 \\ &\leq \frac{\zeta(\alpha + \epsilon)}{\beta}c(OPT) + (1 - \zeta(1 + \epsilon + \epsilon/\beta))\frac{\alpha}{\beta}c(OPT) \\ &= \frac{\alpha + \zeta(1 - \alpha - \alpha/\beta)\epsilon}{\beta}c(OPT). \end{aligned}$$

Since one of these three cases must apply, the maximum of the three values is always an upper bound for $loss_1 + \zeta \cdot loss_2$. □

Lemma 149. For any $w \leq \frac{1}{2}$ and $0 \leq w' \leq 1$ such that $1/(1 + \frac{1-\beta}{5+\beta}w) \leq w' \leq (1 + \epsilon)/(1 + 3\epsilon - \epsilon w \frac{9+3\beta}{5+\beta})$, either the following bound on $c(SOL_{XT})$ holds

$$\begin{aligned} c(SOL_{XT}) &\leq 2 \left((1 + \epsilon)(1 - \lambda)c(OPT_{\mathcal{A}}) + (1 + \epsilon - \epsilon w'(1 + (1 - 2w)(1 - \lambda)))c(OPT_{\mathcal{B}}) \right. \\ &\quad \left. + w'(1 + w \frac{1-\beta}{5+\beta}) \sum_{C \in \mathcal{B}} r_{max}^+(C) \right. \\ &\quad \left. + \max\left(\frac{\alpha}{\beta}, \frac{(1 - w'(1 + \beta)(1 - w))(\alpha + \epsilon)}{\beta}, \frac{\alpha + (1 - \alpha - \alpha/\beta)(1 - w'(1 + \beta)(1 - w))\epsilon}{\beta}\right)c(OPT) \right). \end{aligned}$$

or one of SOL_{LS} and SOL_{XT} is a $2 - 2\alpha$ approximation of the optimal solution OPT .

Proof. We know by Lemma 9 that $c(SOL_{XT}) \leq 2 \sum_{S \subseteq V} y_S''$. Then, it follows from Lemma 146 that

$$c(SOL_{XT}) \leq 2 \left((1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon - \epsilon w'(1 + (1 - 2w)(1 - \lambda)))c(\text{OPT}_{\mathcal{B}}) \right. \\ \left. + w'(1 + w \frac{1 - \beta}{5 + \beta}) \sum_{C \in \mathcal{B}} r_{max}^+(C) + loss_1 + (1 - w'(1 + \beta)(1 - w))loss_2 \right).$$

Now, using Lemma 148 with $\zeta = 1 - w'(1 + \beta)(1 - w)$ completes the proof. \square

Lemma 150. For any $w \leq \frac{1}{2}$ and $0 \leq w' \leq 1$ such that $1/(1 + \frac{1-\beta}{5+\beta}w) \leq w' \leq (1 + \epsilon)/(1 + 3\epsilon - \epsilon w \frac{9+3\beta}{5+\beta})$, either the following bound on $c(SOL_{XT})$ holds

$$c(SOL_{XT}) \leq 2 \left((1 + \epsilon)(1 - \lambda)c(\text{OPT}_{\mathcal{A}}) + (1 + \epsilon - \epsilon w'(1 + (1 - 2w)(1 - \lambda)))c(\text{OPT}_{\mathcal{B}}) \right. \\ \left. + \gamma w'(1 + w \frac{1 - \beta}{5 + \beta})c(\text{OPT}_{\mathcal{B}_1}) + w'(1 + w \frac{1 - \beta}{5 + \beta}) \sum_{C \in \mathcal{B}_2} r_{max}^+(C) \right. \\ \left. + \max\left(\frac{\alpha}{\beta}, \frac{(1 - w'(1 + \beta)(1 - w))(\alpha + \epsilon)}{\beta}, \frac{\alpha + (1 - \alpha - \alpha/\beta)(1 - w'(1 + \beta)(1 - w))\epsilon}{\beta}\right)c(\text{OPT}) \right).$$

or one of SOL_{LS} and SOL_{XT} is a $2 - 2\alpha$ approximation of the optimal solution OPT.

Proof. We start with the bound given in Lemma 149. We can decompose $\sum_{C \in \mathcal{B}} r_{max}^+(C)$ into

$$\sum_{C \in \mathcal{B}_1} r_{max}^+(C) + \sum_{C \in \mathcal{B}_2} r_{max}^+(C).$$

Now, by Definition 83, for any set $C \in \mathcal{B}_1$, we have $r_{max}^+(C) \leq \gamma c(T_C)$. Observing that the coefficient of this term is always non-negative, we can apply this upper bound for every component C in \mathcal{B}_1 to derive the desired bound. \square

8 Autarkic Pairs

In this section, we introduce a new approach designed specifically for the Steiner Forest problem, in contrast to the more general local search method discussed earlier. This method centers on the notion of *autarkic pairs*, which we identify and connect directly, while the remaining demands are handled by Legacy Moat Growing. Each autarkic pair consists of two subsets of vertices such that the vertices in one subset are the pairs of those in the other, and both subsets are expected to belong to the same connected component of the optimal solution. Within each subset, the vertices connect to each other significantly earlier than the time it takes for the two subsets to connect.

The motivation behind detecting and directly connecting autarkic pairs is that, in some connected components of the optimal solution, certain vertices may be merged into the same group early in a moat growing algorithm, but it may take a long time for these groups to connect to each other. This delay increases the maximum assigned value for the vertices in that component, leading to a large bound in Lemma 149. As a result, SOL_{XT} may not perform well.

To address this issue, we attempt to identify such connected components in the optimal solution. In these components, instead of allowing the last two active sets to grow further to reach each other, we select and add the shortest path between a pair of vertices, each belonging to one of these two active sets, to our solution.

This reduces the maximum assigned value of that connected component since the last two active sets do not need to grow to reach each other, as they can use the selected shortest path.

Since we do not have access to the optimal solution, we make rational guesses about these pairs to ensure our selection covers all relevant ones. However, it may include pairs that do not belong to the intended connected components of the optimal solution, as we lack explicit knowledge of it. Nonetheless, we carefully select autarkic pairs to ensure that false guesses do not interfere with our analysis.

In Section 8.1, we describe the `AUTARKICPAIRS` procedure and provide its pseudocode. Then, in Section 8.2, we identify properties for connected components in \mathcal{B}_2 and show that they have autarkic pairs. Finally, in Section 8.3, we use these properties to analyze the solution produced by executing `AUTARKICPAIRS` in Line 6, given that the algorithm previously calls `LOCALSEARCH` in Line 3 and we are finding autarkic pairs based on Boosted Execution.

8.1 Algorithm

In this section, we describe the `AUTARKICPAIRS` procedure, with its pseudocode provided in Algorithm 9. This method aims to identify specific pairs of subsets of vertices, referred to as autarkic pairs, and connect them directly. See also Figure 2 for an illustration of how this procedure works.

To formally define the selection of autarkic pairs, we first introduce the following notation.

Definition 151. Recall that for any vertices $v, u \in V$, $d(v, u)$ denotes their distance in the graph G . For any subset of vertices $S \subseteq V$, we define $d_{\max}(S)$ as the maximum distance between a vertex $v \in S$ and its pair:

$$d_{\max}(S) = \max_{v \in S} d(v, \text{PAIR}_v).$$

First, for each set that was active at any point during Boosted Execution, we accumulate its growth duration y_S^+ into $Y_{\text{UNSATISFIED}(S)}$. At the end, Y_S represents the total time during which the vertices in S were the only unsatisfied vertices in an active set. Importantly, if there exists a vertex $v \in S$ such that $\text{PAIR}_v \in S$, then $Y_S = 0$, as no set $S' \subseteq V$ satisfies $\text{UNSATISFIED}(S') = S$ in this case.

Then, for any set $S \neq \emptyset$ with $Y_S > 0$, we compare $Y_S + Y_{\text{PAIR}(S)}$ to $d_{\max}(S)$, using the threshold parameter $\eta \in (0, 1)$ provided by the main algorithm. If the condition in Line 8 is satisfied, we select S and $\text{PAIR}(S)$ as an autarkic pair. Next, we choose an arbitrary vertex $v \in S$, add the shortest path between v and its pair PAIR_v to the solution, and insert a zero-cost edge between them in E_{AP} , since they are now connected.

Finally, we run `LEGACYMOATGROWING` on $G' = G \cup E_{AP}$ to satisfy the remaining demands. The vertices in autarkic pairs are connected using the paths selected earlier, while the remaining pairs are handled by `LEGACYMOATGROWING`. Note that, instead of `LEGACYMOATGROWING`, other Steiner Forest algorithms—such as Algorithm 2, which achieves a better approximation factor—could be used on the new instance. However, for simplicity in our analysis, we continue to use `LEGACYMOATGROWING`.

Note that the for loop in Line 5 only needs to consider subsets $S \subseteq V$ with $y_S^+ > 0$, and the for loop in Line 7 only needs to consider subsets $S \subseteq V$ with $Y_S + Y_{\text{PAIR}(S)} > 0$. Since the number of such sets is polynomial, and by applying Corollary 4, we obtain the following corollary.

Corollary 152. The `AUTARKICPAIRS` procedure runs in polynomial time.

Moreover, if S is selected as an autarkic pair, we should not check $\text{PAIR}(S)$ as another subset in the for loop in Line 7. This prevents selecting a pair twice as an autarkic pair.

Algorithm 9 Autarkic Pairs

Input: A graph $G = (V, E, c)$ with edge costs $c : E \rightarrow \mathbb{R}_{\geq 0}$, demand function $\text{PAIR} : V \rightarrow V$, a function $y^+ : 2^V \rightarrow \mathbb{R}_{\geq 0}$ indicating the growth of active sets in Boosted Execution, and parameter $0 < \eta < 1$.

Output: A forest F that satisfies all demands.

```
1: procedure AUTARKICPAIRS( $G, \text{PAIR}, y^+, \eta$ )
2:    $F \leftarrow \emptyset$ 
3:    $E_{AP} \leftarrow \emptyset$ 
4:   Implicitly set  $Y_S \leftarrow 0$  for all  $S \subseteq V$ 
5:   for  $S \subseteq V$  do
6:      $Y_{\text{UNSATISFIED}(S)} \leftarrow Y_{\text{UNSATISFIED}(S)} + y_S^+$ 
7:   for  $S \subseteq V, S \neq \emptyset$  do
8:     if  $(1 + \eta)(Y_S + Y_{\text{PAIR}(S)}) > d_{\max}(S)$  then
9:       Select arbitrary vertex  $v \in S$ 
10:      Add the shortest path between  $v$  and  $\text{PAIR}_v$  to  $F$ 
11:      Insert a zero-cost edge  $(v, \text{PAIR}_v)$  into  $E_{AP}$ 
12:    $G' \leftarrow G \cup E_{AP}$ 
13:    $F \leftarrow F \cup \text{LEGACYMOATGROWING}(G', \text{PAIR}) \setminus E_{AP}$ 
14:   return  $F$ 
```

Furthermore, in Line 6, we pass y^{b+} instead of y^+ to AUTARKICPAIRS. This does not affect the outcome, since for sets with nonempty unsatisfied vertex sets, we have $y^{b+} = y^+$. Additionally, as we ignore Y_\emptyset (see Line 7), we can assume that the input to AUTARKICPAIRS is effectively y^+ .

To analyze our algorithm, we first need to identify properties of the connected components of the optimal solution in \mathcal{B}_2 . These properties demonstrate that these connected components contain autarkic pairs, allowing us to bound SOL_{AP} .

8.2 Properties of Connected Components in \mathcal{B}_2

In this section, we establish several properties for the connected components of the optimal solution in the class \mathcal{B}_2 . These properties rely on the fact that LOCALSEARCH has already been executed before invoking AUTARKICPAIRS, and that the values y_S^+ in the input to AUTARKICPAIRS represent the growth of active sets from Boosted Execution, which does not involve any valuable boost actions.

Recall that the class \mathcal{B}_2 , introduced in Definition 83, depends on the parameter γ , whose exact form was not specified earlier. Here, we define γ together with another parameter \varkappa , both expressed in terms of β , λ , and η . Their exact numerical values will be fixed in Section 9, once the remaining parameters have been determined.

Definition 153. We define two constants \varkappa and γ based on β , λ , and η as follows:

$$\varkappa = 4\lambda \frac{1 + \beta}{1 - \beta},$$
$$\gamma = \frac{\lambda}{\eta} \left(4 + 3\eta + 4(1 + \eta) \frac{1 + \beta}{1 - \beta} \right).$$

To simplify the presentation, we focus on a fixed connected component $C \in \mathcal{B}_2$ and denote its corresponding tree in the optimal solution by T_C . The following notations and definitions are specific to this component, starting with the selection of the two highest-priority vertices within C .

Definition 154. We define v_1 as the vertex with the maximum priority in C , and v_2 as the pair of v_1 . That is,

$$\begin{aligned} v_1 &= \operatorname{argmax}_{v \in C} \operatorname{priority}_v \\ v_2 &= \operatorname{PAIR}_{v_1}. \end{aligned}$$

Note that $v_2 \in C$, since $v_1 \in C$ and is connected to its pair v_2 in any valid solution. One straightforward observation from this definition is that v_1 has the maximum assigned value as stated formally below.

Lemma 155. Vertex v_1 has the maximum assigned value for r^- and r^+ . That means

$$\begin{aligned} r_{v_1}^- &= r_{\max}^-(C) \\ r_{v_1}^+ &= r_{\max}^+(C). \end{aligned}$$

Proof. Given Definition 154, v_1 has the maximum priority in C . Therefore, we can use Corollary 34 to conclude that $t_{v_1}^-$ is the maximum value between t^- of vertices in C . That means for any vertex $v \in C$ we have

$$t_{v_1}^- \geq t_v^- \tag{8}$$

Let us fix any time $\tau < t_{v_1}^-$. According to Corollary 65, we also have $\tau < t_{v_1}^+$. If S is the connected component containing v_1 in Legacy Execution (and similarly in Boosted Execution), then S is an active set since v_1 remains in an active set until time $t_{v_1}^-$ (and similarly $t_{v_1}^+$), as per Definition 6 and the fact that t^- (similarly t^+) is the fingerprint of the execution. Furthermore, we have $v_1 \in \operatorname{BASE}(S, \tau)$. Therefore, $v_1 \in \operatorname{REPS}(\operatorname{BASE}(S, \tau))$ because $\max_{u \in \operatorname{BASE}(S, \tau) \cap C} \operatorname{priority}_u \leq \max_{u \in C} \operatorname{priority}_u = \operatorname{priority}_{v_1}$. Thus, we can conclude that we are assigning to v_1 at any time $\tau < t_{v_1}^-$. Using Lemmas 42 and 70, we know that both r^- and r^+ are prefix-time assignments, respectively. Therefore, we can conclude that

$$r_{v_1}^-, r_{v_1}^+ \geq t_{v_1}^- \tag{9}$$

Moreover, for any vertex $v \in C$ and anytime $\tau \geq t_v^-$ and any subset of vertices $S \subseteq V$, we have $v \notin \operatorname{BASE}(S, \tau)$, which means, $v \notin \operatorname{REPS}(\operatorname{BASE}(S, \tau))$. Therefore we can conclude that

$$r_v^-, r_v^+ \leq t_v^- \tag{10}$$

By combining Equations 8, 9, and 10, for any vertex $v \in C$ we have

$$\begin{aligned} r_{v_1}^- &\geq t_{v_1}^- \geq t_v^- \geq r_v^- \\ r_{v_1}^+ &\geq t_{v_1}^- \geq t_v^- \geq r_v^+ \end{aligned}$$

which proves the lemma. □

Given Lemma 73, we have $r_{\max}^-(C) = r_{\max}^+(C)$. Using the above lemma, we can conclude the following corollary.

Corollary 156. The assignment to v_1 is the same in both r^- and r^+ . In other words,

$$r_{v_1}^- = r_{v_1}^+.$$

Now, we aim to identify important properties that demonstrate the presence of autarkic pairs within connected component $C \in \mathcal{B}_2$. The first property, detailed in Lemma 158, shows that $r_{v_1}^+$ and $r_{v_2}^+$ have almost identical values. Then, in Lemma 164, we demonstrate that other vertices of C , which are actively connected to v_1 or v_2 , connect with them early in Boosted Execution. Next, Lemma 167 reveals that most of the growth time of active sets containing v_1 and v_2 does not involve unsatisfied vertices from other connected components of the optimal solution. These three properties lead to the identification of two subsets of vertices, one containing v_1 and the other containing v_2 . We establish a lower bound for the total Y of these subsets in Lemma 170, provide an upper bound for the d_{\max} of these two subsets in Lemma 172, and finally demonstrate that they are autarkic pairs in Lemma 173. To facilitate the proof of these properties and lemmas, we present auxiliary lemmas throughout to simplify the process. Here is one such auxiliary lemma.

Lemma 157. We can relate the assigned values of v_2 and v_1 for Legacy Execution and Boosted Execution as follows:

$$r_{v_2}^- = r_{v_1}^+.$$

Proof. Based on Corollary 156, we have $r_{v_1}^- = r_{v_1}^+$. It remains to show that $r_{v_2}^- = r_{v_1}^-$, which completes the proof.

To prove that $r_{v_2}^- = r_{v_1}^-$, first note that t^- returned by LEGACYMOATGROWING indicates the time at which each vertex connects to its pair, which is the same for v_1 and v_2 . In other words, $t_{v_1}^- = t_{v_2}^-$. Additionally, by Lemma 42, r^- is a prefix-time assignment. Therefore, if we show that for any time $\tau_- < t_{v_1}^-$ both vertices are assigned, and for any time $\tau_+ > t_{v_1}^-$ neither is assigned, it follows that these two vertices receive the same r^- value.

For $\tau_+ > t_{v_1}^-$, both v_1 and v_2 are not members of $\text{BASE}(S, \tau_+)$ for any subset $S \subseteq V$, and consequently, are not members of $\text{REPS}(\text{BASE}(S, \tau_+))$.

For $\tau_- < t_{v_1}^-$, these two vertices have not yet reached each other. Thus, each active set containing one of them does not contain the other. Let $S \subseteq V$ be a subset of vertices containing one of them at time τ_- . We know that this vertex is also in $\text{BASE}(S, \tau_-)$, since $\tau_- < t_{v_1}^-, t_{v_2}^-$. Moreover, the vertex is in $\text{REPS}(\text{BASE}(S, \tau_-))$. This is immediate for v_1 , as it has the maximum priority in C . For v_2 , note that since v_1 has the maximum priority in C , its t^- is also maximal among vertices in C . Given that $t_{v_2}^- = t_{v_1}^-$ and by Definition 33, v_2 has the second-highest priority in C . Therefore, if $v_2 \in S$ at time τ_- , which implies $v_1 \notin S$, then v_2 is the vertex with the maximum priority from C in S , and thus belongs to $\text{REPS}(\text{BASE}(S, \tau_-))$.

This completes the proof that $r_{v_2}^- = r_{v_1}^-$, and consequently proves the lemma. \square

Now we provide the first important property, showing that $r_{v_1}^+$ and $r_{v_2}^+$ have roughly the same value.

Lemma 158. The difference between the assigned values r^+ of vertices v_1 and v_2 can be bounded by

$$r_{v_1}^+ - r_{v_2}^+ \leq \lambda c(T_C).$$

Proof. We prove this by showing that $r_{v_1}^+ - r_{v_2}^+$ is a lower bound on the difference between the total r^- of

vertices in C and their total r^+ , and then show that $\lambda c(T_C)$ is an upper bound for this value.

$$\begin{aligned}
r_{v_1}^+ - r_{v_2}^+ &= r_{v_2}^- - r_{v_2}^+ && \text{(Lemma 157)} \\
&\leq r_{v_2}^- - r_{v_2}^+ + \sum_{v \in C \setminus \{v_2\}} (r_v^- - r_v^+) && \text{(Lemma 72)} \\
&= \sum_{v \in C} r_v^- - \sum_{v \in C} r_v^+ \\
&\leq c(T_C) - (1 - \lambda)c(T_C) && \text{(Lemma 44, Corollary 84)} \\
&= \lambda c(T_C)
\end{aligned}$$

□

The next three lemmas establish connections between $UM^+(T_C)$ and the total assigned value r^+ of vertices in C , and derive bounds on $UM^+(T_C)$. Here, UM^+ refers to UM as defined in Definition 18, with respect to Boosted Execution. These lemmas play a key role in the proofs of Lemmas 164 and 167.

Lemma 159. For any subset of vertices $S' \subseteq C$ with $v_1 \notin S'$, we have

$$\sum_{v \in S'} r_v^+ \leq \sum_{\substack{S \subseteq V \\ S \cap S' \neq \emptyset \\ v_1 \notin S}} y_S^+.$$

Proof. First, we show that each y_S^+ value from sets on the right-hand side can be assigned to at most one r_v^+ of a vertex on the left-hand side. Then we show that only y_S^+ values from those sets can be assigned to r_v^+ of vertices in S' , completing the proof.

For any time τ and any subset of vertices $S \subseteq V$, according to Lemma 36, we have

$$(\text{REPS}(\text{BASE}(S, \tau)) \cap C) \in \{\emptyset, \{\text{argmax}_{v \in S \cap C} \text{priority}_v\}\}.$$

Since $S' \subseteq C$, we conclude that

$$|\text{REPS}(\text{BASE}(S, \tau)) \cap S'| \leq 1,$$

meaning the value of y_S^+ can contribute to r_v^+ of at most one vertex in S' .

Furthermore, for any subset of vertices $S \subseteq V$ and time τ , if $|\text{REPS}(\text{BASE}(S, \tau)) \cap S'| = 1$, then

$$(\text{REPS}(\text{BASE}(S, \tau)) \cap S') = \{\text{argmax}_{v \in S \cap C} \text{priority}_v\}.$$

This happens only if $\text{argmax}_{v \in S \cap C} \text{priority}_v$ is in S' . Therefore, $(S \cap C) \cap S' \neq \emptyset$, and consequently $S \cap S' \neq \emptyset$. Additionally, since v_1 has the maximum priority in C , if $v_1 \in S$, then $\text{argmax}_{v \in S \cap C} \text{priority}_v = v_1 \notin S'$, contradicting the assignment to S' . Thus, a y_S^+ value can be assigned to r_v^+ of some vertex $v \in S'$ only if $S \cap S' \neq \emptyset$ and $v_1 \notin S$. □

Lemma 160. The total assigned value r^+ to vertices in C is bounded by

$$\sum_{v \in C} r_v^+ \leq (r_{v_1}^+ - r_{v_2}^+) + c(T_C) - \frac{UM^+(T_C)}{2}.$$

Proof. We begin by bounding $r_{v_1}^+$ using Lemma 159 with $S' = \{v_2\}$:

$$\begin{aligned} r_{v_1}^+ &= r_{v_2}^+ + (r_{v_1}^+ - r_{v_2}^+) \\ &\leq \sum_{\substack{S \subseteq V \\ v_2 \in S \\ v_1 \notin S}} y_S^+ + (r_{v_1}^+ - r_{v_2}^+). \end{aligned} \quad (\text{Lemma 159})$$

Since v_1 and v_2 form a pair, they remain in active sets until they are connected in Legacy Execution. As a consequence, Corollary 53 ensures that v_1 and v_2 also remain active until they connect in Boosted Execution. Consequently, at any moment in Boosted Execution, a connected component containing v_2 but not v_1 is active, and another component containing v_1 but not v_2 is also active. It follows that $\sum_{\substack{S \subseteq V \\ v_2 \in S \\ v_1 \notin S}} y_S^+ = \sum_{\substack{S \subseteq V \\ v_1 \in S \\ v_2 \notin S}} y_S^+$, which

combining by above inequality lead to

$$r_{v_1}^+ \leq \sum_{\substack{S \subseteq V \\ v_1 \in S \\ v_2 \notin S}} y_S^+ + (r_{v_1}^+ - r_{v_2}^+). \quad (11)$$

Next, we bound the contribution of all vertices in C except for v_1 . By Lemma 159, we have

$$\sum_{\substack{v \in C \\ v \neq v_1}} r_v^+ \leq \sum_{\substack{S \subseteq V \\ S \cap C \neq \emptyset \\ v_1 \notin S}} y_S^+. \quad (12)$$

We now define the following families of sets:

$$\begin{aligned} \mathcal{S}_1 &= \{S \subseteq V \mid v_1 \in S, v_2 \notin S\}, \\ \mathcal{S}_2 &= \{S \subseteq V \mid S \cap C \neq \emptyset, v_1 \notin S\}, \\ \mathcal{S}_3 &= \{S \subseteq V \mid S \odot C\}. \end{aligned}$$

Observe that \mathcal{S}_1 and \mathcal{S}_2 are disjoint, since each set in \mathcal{S}_1 contains v_1 while each set in \mathcal{S}_2 does not. Furthermore, every set in $\mathcal{S}_1 \cup \mathcal{S}_2$ cuts C , and thus $(\mathcal{S}_1 \cup \mathcal{S}_2) \subseteq \mathcal{S}_3$. These observations imply

$$\sum_{S \in \mathcal{S}_1} y_S^+ + \sum_{S \in \mathcal{S}_2} y_S^+ \leq \sum_{S \in \mathcal{S}_3} y_S^+,$$

which, in expanded form, is

$$\sum_{\substack{S \subseteq V \\ v_1 \in S \\ v_2 \notin S}} y_S^+ + \sum_{\substack{S \subseteq V \\ S \cap C \neq \emptyset \\ v_1 \notin S}} y_S^+ \leq \sum_{\substack{S \subseteq V \\ S \odot C}} y_S^+. \quad (13)$$

Finally, we combine all these inequalities to complete the proof.

$$\begin{aligned}
\sum_{v \in C} r_v^+ &= r_{v_1}^+ + \sum_{\substack{v \in C \\ v \neq v_1}} r_v^+ \\
&\leq \left(\sum_{\substack{S \subseteq V \\ v_1 \in S \\ v_2 \notin S}} y_S^+ + (r_{v_1}^+ - r_{v_2}^+) \right) + \sum_{\substack{S \subseteq V \\ S \cap C \neq \emptyset \\ v_1 \notin S}} y_S^+ && \text{(Equations 11 and 12)} \\
&\leq (r_{v_1}^+ - r_{v_2}^+) + \sum_{\substack{S \subseteq V \\ S \cap C}} y_S^+ && \text{(Equation 13)} \\
&\leq (r_{v_1}^+ - r_{v_2}^+) + c(T_C) - \frac{UM^+(T_C)}{2}. && \text{(Lemma 21)}
\end{aligned}$$

□

Lemma 161. The quantity $UM^+(T_C)$ can be bounded as follows:

$$UM^+(T_C) \leq 4\lambda c(T_C).$$

Proof. The bound follows easily from previous lemmas:

$$\begin{aligned}
(1 - \lambda) \cdot c(T_C) &< \sum_{v \in C} r_v^+ && (C \in \mathcal{B}, \text{Corollary 84}) \\
&\leq (r_{v_1}^+ - r_{v_2}^+) + c(T_C) - \frac{UM^+(T_C)}{2} && \text{(Lemma 160)} \\
&\leq (1 + \lambda) \cdot c(T_C) - \frac{UM^+(T_C)}{2}. && \text{(Lemma 158)}
\end{aligned}$$

After reordering terms, we have

$$UM^+(T_C) \leq 4\lambda c(T_C).$$

□

Using Definition 153, we show a lower bound on the value of $r_{v_2}^+$.

Lemma 162. The value $r_{v_2}^+$ satisfies the following lower bound:

$$r_{v_2}^+ > \kappa c(T_C).$$

Proof.

$$\begin{aligned}
r_{v_2}^+ &\geq r_{v_1}^+ - \lambda c(T_C) && \text{(Lemma 158)} \\
&= r_{max}^+(C) - \lambda c(T_C) && \text{(Lemma 155)} \\
&> (\gamma - \lambda) c(T_C) && \text{(Definition 83)}
\end{aligned}$$

Therefore, it suffices to show that $\gamma - \lambda \geq \kappa$. Using Definition 153, this inequality reduces to:

$$\frac{\lambda}{\eta} \left(4 + 3\eta + 4(1 + \eta) \frac{1 + \beta}{1 - \beta} \right) - \lambda \geq 4\lambda \frac{1 + \beta}{1 - \beta}.$$

Multiplying both sides by the positive factor $\frac{\eta}{\lambda}$, we obtain:

$$4 + 3\eta + 4(1 + \eta)\frac{1 + \beta}{1 - \beta} - \eta \geq 4\eta\frac{1 + \beta}{1 - \beta},$$

which simplifies to:

$$4 + 2\eta + 4\frac{1 + \beta}{1 - \beta} \geq 0.$$

This inequality clearly holds, as all parameters are positive and $\beta < 1$. \square

Definition 163. Let ι_u denote the moment when a vertex u becomes connected to either v_1 or v_2 , and let S be the set of vertices $u \in C \setminus \{v_1, v_2\}$ for which ι_u is defined—that is, u connects to one of them—and u is unsatisfied just before ι_u (regardless of whether it becomes satisfied at ι_u). Then, we define

$$v_3 = \operatorname{argmax}_{u \in S} \iota_u.$$

Intuitively, v_3 is the last vertex in $C \setminus \{v_1, v_2\}$ that was not previously connected to either v_1 or v_2 , and then becomes connected to one of them for the first time while still unsatisfied just before that moment in Boosted Execution.

If no such vertex exists, we assume v_3 to be a dummy vertex in C that is connected to v_1 by a zero-cost edge.

Next, we aim to use the above lemmas to establish another important property, which gives an upper bound for $r_{v_3}^+$.

Lemma 164. The value of $r_{v_3}^+$ can be bounded as

$$r_{v_3}^+ \leq \kappa c(C).$$

Proof. We prove this by contradiction, assuming that $r_{v_3}^+ > \kappa c(C)$.

Since v_1, v_2 , and v_3 belong to C , they are all connected in the optimal solution. This means the subgraph of the optimal solution induced by these three vertices and the paths between them forms a star with three leaves, with a vertex $q \in C$ as the center. We refer to this star as ST . Note that it is possible that q coincides with one of these vertices. In this case, we can assume that they are distinct vertices connected by a zero-cost edge.

Let τ represent the first moment that two of these vertices are connected, and τ' the first time that all three are connected in Boosted Execution. Given Lemma 155, we have $r_{v_1}^+ \geq r_{v_2}^+$. Moreover, by Lemma 162, we have $r_{v_2}^+ > \kappa c(C)$, and by the contradiction assumption, $r_{v_3}^+ > \kappa c(C)$. Since the r^+ values of all these vertices are greater than $\kappa c(C)$, using Lemma 71, we can conclude that no pair among these three vertices connects until time $\kappa c(C)$. Therefore,

$$\tau \geq \min(r_{v_1}^+, r_{v_2}^+, r_{v_3}^+) > \kappa c(C). \quad (14)$$

Next, we aim to contradict the above inequality.

Since v_1 and v_2 are paired, they are actively connected in Legacy Execution, which implies they are also actively connected in Boosted Execution (by Lemma 15). Additionally, according to Definition 163, v_3 remains unsatisfied before connecting to the other two vertices. Thus, by Corollary 53, we can similarly conclude that v_3 is actively connected to v_1 and v_2 . Therefore, we can apply Lemma 88 as follows:

$$\tau + \tau' \leq \frac{d(q, v_1) + d(q, v_2) + d(q, v_3)}{2} + \beta \frac{\min(d(q, v_1), d(q, v_2), d(q, v_3))}{2}, \quad (15)$$

where $d(q, v_1)$, $d(q, v_2)$, and $d(q, v_3)$ are the distances from v_1 , v_2 , and v_3 to q , respectively.

Next, we bound $d(q, v_1) + d(q, v_2) + d(q, v_3)$ and $\min(d(q, v_1), d(q, v_2), d(q, v_3))$ based on τ , τ' , and $UM^+(T_C)$.

Let $S' = \{v_1, v_2, v_3\}$ be the set of leaves of ST . We aim to show that $\sum_{S \in S'} y_S^+ \leq \tau + 2\tau'$. In Boosted Execution, until time τ , there are three connected components containing at least one of these vertices, contributing at most 3τ to the sum. After time τ , two vertices become connected, forming a single component. Between τ and τ' , this contributes an additional $2(\tau' - \tau)$ to the sum. After τ' , all vertices are in the same component, and the sum remains unchanged. Hence,

$$\sum_{\substack{S \subseteq V \\ S \in S'}} y_S^+ \leq 3\tau + 2(\tau' - \tau) = \tau + 2\tau'. \quad (16)$$

Now, since ST is a subgraph of T_C , Lemma 20 implies:

$$\begin{aligned} UM^+(T_C) &\geq UM^+(ST) && \text{(Lemma 20)} \\ &\geq c(ST) - \sum_{\substack{S \subseteq V \\ S \in S'}} y_S^+ && \text{(Lemma 23)} \\ &\geq d(q, v_1) + d(q, v_2) + d(q, v_3) - (\tau + 2\tau'). && \text{(Equation 16)} \end{aligned}$$

Rearranging gives:

$$d(q, v_1) + d(q, v_2) + d(q, v_3) \leq \tau + 2\tau' + UM^+(T_C). \quad (17)$$

Next, we bound $\min(d(q, v_1), d(q, v_2), d(q, v_3))$. We know that two vertices of S' are connected at time τ in Boosted Execution. Without loss of generality, let us assume that these are v_1 and v_2 . We refer to the path between these vertices in the optimal solution as PT . We know that $c(PT) \geq d(q, v_1) + d(q, v_2)$.

Moreover, since these two vertices reach each other at time τ , the total y_S^+ of active sets containing exactly one of v_1 and v_2 is at most 2τ . This means

$$\sum_{\substack{S \subseteq V \\ S \in \{v_1, v_2\}}} y_S^+ \leq 2\tau. \quad (18)$$

Since PT is a subgraph of T_C , we can use Lemma 20 to conclude

$$\begin{aligned} UM^+(T_C) &\geq UM^+(PT) && \text{(Lemma 20)} \\ &\geq c(PT) - \sum_{\substack{S \subseteq V \\ S \in \{v_1, v_2\}}} y_S^+ && \text{(Lemma 23)} \\ &\geq d(q, v_1) + d(q, v_2) - 2\tau. && \text{(Equation 18)} \end{aligned}$$

By simple reordering, we obtain:

$$2\tau + UM^+(T_C) \geq d(q, v_1) + d(q, v_2) \geq 2 \min(d(q, v_1), d(q, v_2)).$$

Therefore,

$$\tau + UM^+(T_C) \geq \min(d(q, v_1), d(q, v_2)),$$

and it follows that

$$\min(d(q, v_1), d(q, v_2), d(q, v_3)) \leq \tau + UM^+(T_C). \quad (19)$$

Substituting inequalities from (17) and (19) into (15) yields:

$$\begin{aligned} \tau + \tau' &\leq \frac{d(q, v_1) + d(q, v_2) + d(q, v_3)}{2} + \beta \frac{\min(d(q, v_1), d(q, v_2), d(q, v_3))}{2} && \text{(Equation 15)} \\ &\leq \frac{\tau + 2\tau' + UM^+(T_C)}{2} + \beta \frac{\tau + UM^+(T_C)}{2}. && \text{(Equations 17 and 19)} \end{aligned}$$

Rearranging this results in

$$\begin{aligned} \tau &\leq \frac{1 + \beta}{1 - \beta} UM^+(T_C) \\ &\leq 4\lambda \frac{1 + \beta}{1 - \beta} c(T_C) && \text{(Lemma 161)} \\ &= \kappa c(T_C) && \text{(Definition 153)} \end{aligned}$$

This contradicts inequality (14), completing the proof. \square

Definition 165. Let g^{mix} denote the total growth of active sets that contain unsatisfied vertices from C as well as vertices outside of it. Specifically,

$$g^{mix} = \sum_{\substack{S \subseteq V \\ \text{UNSATISFIED}(S) \cap C \neq \emptyset \\ \text{UNSATISFIED}(S) \not\subseteq C}} y_S^+.$$

Note that according to Line 6, this is equivalent to the following:

$$g^{mix} = \sum_{\substack{S \subseteq V \\ S \cap C \neq \emptyset \\ S \not\subseteq C}} Y_S.$$

First, we establish a relation between the total assignment to vertices in C under r^+ and \hat{r}^+ using g^{mix} . Then, we use this relation to bound g^{mix} in Lemma 167, which plays a key role in proving that C contains an autarkic pair.

Lemma 166. We can bound the sum of \hat{r}^+ values for vertices in C in terms of their r^+ values and g^{mix} as follows:

$$\sum_{v \in C} \hat{r}_v^+ \leq \sum_{v \in C} r_v^+ - \frac{g^{mix}}{2}.$$

Proof. First, we define two families of subsets of vertices:

$$\begin{aligned} \mathcal{S}_1 &= \{S \subseteq V \mid \text{UNSATISFIED}(S) \cap C \neq \emptyset, \text{UNSATISFIED}(S) \not\subseteq C\}, \\ \mathcal{S}_2 &= \{S \subseteq V \mid \text{UNSATISFIED}(S) \cap C = \emptyset \text{ or } \text{UNSATISFIED}(S) \subseteq C\}. \end{aligned}$$

Note that

$$\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset \quad \text{and} \quad \mathcal{S}_1 \cup \mathcal{S}_2 = 2^V. \quad (20)$$

Intuitively, we partition all sets into those that contribute to g^{mix} (the sets in \mathcal{S}_1) and those that do not (the sets in \mathcal{S}_2). For sets in \mathcal{S}_1 , their growth is fully assigned to the r^+ values of vertices in C . However, due to the structure of \hat{r}^+ assignments, each such set contributes at most half of that growth to the \hat{r}^+ values of these vertices. This gap between the full assignment in r^+ and the partial assignment in \hat{r}^+ is what yields the factor of $g^{mix}/2$ in the inequality.

For any set $S \in \mathcal{S}_1$ active at time τ during Boosted Execution (i.e., $S \in ActS_\tau^+$), we have $UNSATISFIED(S) \cap C \neq \emptyset$. Therefore, there exists a vertex $v \in S \cap C$ with $PAIR_v \notin S$ (by Definition 3). In Legacy Execution, vertex v and $PAIR_v$ reach each other at time t_v^- (see Definition 29). Moreover, by Corollary 53, they are in the same connected component at time t_v^- in Boosted Execution. Since in Boosted Execution they are not yet connected at time τ , but are connected by time t_v^- , it follows that $\tau < t_v^-$. This implies $v \in BASE(S, \tau)$ (see Definition 31), and hence $BASE(S, \tau) \cap C \neq \emptyset$. Applying Lemma 36 then gives:

$$|REPS(BASE(S, \tau)) \cap C| = 1. \quad (21)$$

By Definition 69, the growth from S is assigned entirely to exactly one vertex in C , meaning:

$$\sum_{v \in C} r_{S,v}^+ = y_S^+ \quad \text{for all } S \in \mathcal{S}_1. \quad (22)$$

Similarly, for $S \in \mathcal{S}_1$, since $UNSATISFIED(S) \not\subseteq C$, there exists $v \in S \setminus C$ with $PAIR_v \notin S$. Using the same reasoning, $v \in BASE(S, \tau)$, and hence $BASE(S, \tau) \setminus C \neq \emptyset$. Consequently, by Lemma 36, we obtain that $|REPS(BASE(S, \tau)) \setminus C| \geq 1$. Combining this with Equation 21, we conclude that

$$|REPS(BASE(S, \tau))| \geq 2.$$

By Definition 74, the growth from such sets is distributed across multiple components, with vertices in C collectively receiving a fraction of the growth equal to $1/|REPS(BASE(S, \tau))|$. Therefore:

$$\sum_{v \in C} \hat{r}_{S,v}^+ \leq \frac{1}{2} y_S^+ \quad \text{for all } S \in \mathcal{S}_1. \quad (23)$$

We can now complete the proof:

$$\begin{aligned} \sum_{v \in C} r_v^+ - \frac{g^{mix}}{2} &= \sum_{v \in C} \sum_{S \subseteq V} r_{S,v}^+ - \frac{1}{2} \sum_{S \in \mathcal{S}_1} y_S^+ && \text{(Definitions 26 and 165)} \\ &= \sum_{S \in \mathcal{S}_1} \left(\sum_{v \in C} r_{S,v}^+ - \frac{1}{2} y_S^+ \right) + \sum_{S \in \mathcal{S}_2} \sum_{v \in C} r_{S,v}^+ && \text{(Equation 20)} \\ &= \sum_{S \in \mathcal{S}_1} \left(y_S^+ - \frac{1}{2} y_S^+ \right) + \sum_{S \in \mathcal{S}_2} \sum_{v \in C} r_{S,v}^+ && \text{(Equation 22)} \\ &\geq \sum_{S \in \mathcal{S}_1} \frac{1}{2} y_S^+ + \sum_{S \in \mathcal{S}_2} \sum_{v \in C} \hat{r}_{S,v}^+ && \text{(Lemma 79)} \\ &\geq \sum_{S \in \mathcal{S}_1} \sum_{v \in C} \hat{r}_{S,v}^+ + \sum_{S \in \mathcal{S}_2} \sum_{v \in C} \hat{r}_{S,v}^+ && \text{(Equation 23)} \\ &= \sum_{v \in C} \hat{r}_v^+. && \text{(Equation 20)} \end{aligned}$$

□

Lemma 167. We can bound the value of g^{mix} as follows:

$$g^{mix} \leq 4\lambda c(T_C).$$

Proof. The proof follows from the previous lemmas as outlined below:

$$\begin{aligned} (1 - \lambda)c(T_C) &< \sum_{v \in C} \hat{r}_v^+ && (C \in \mathcal{B}, \text{Definition 82}) \\ &\leq \sum_{v \in C} r_v^+ - \frac{g^{mix}}{2} && (\text{Lemma 166}) \\ &\leq (r_{v_1}^+ - r_{v_2}^+) + c(T_C) - \frac{g^{mix}}{2} && (\text{Lemma 160}, UM^+(T_C) \geq 0) \\ &\leq (1 + \lambda)c(T_C) - \frac{g^{mix}}{2}. && (\text{Lemma 158}) \end{aligned}$$

Reordering the terms results in

$$g^{mix} \leq 4\lambda c(T_C).$$

□

Definition 168. Let τ_{v_3} denote the moment immediately after $r_{v_3}^+$, meaning all merges and deactivations at time $r_{v_3}^+$ have already occurred in Boosted Execution. Let $S'_1, S'_2 \in ActS_{\tau_{v_3}}^+$ be the active sets containing v_1 and v_2 , respectively, at this moment. We define subsets $S_1, S_2 \subseteq V$ as the sets of unsatisfied vertices in C that lie in S'_1 and S'_2 , respectively. That is,

$$\begin{aligned} S_1 &= \text{UNSATISFIED}(S'_1) \cap C, \\ S_2 &= \text{UNSATISFIED}(S'_2) \cap C. \end{aligned}$$

We use the notation S_1, S_2, S'_1 , and S'_2 , and first show that S_1 and S_2 form pairs. Subsequently, in Lemma 170, we provide a lower bound for $Y_{S_1} + Y_{S_2}$, and in Lemma 172, we give an upper bound for $d_{\max}(S_1)$. Finally, using the value of γ from Definition 153 and the autarkic condition in Line 8 of AUTARKICPAIRS, we conclude in Lemma 173 that S_1 and S_2 are selected as autarkic pairs.

Lemma 169. Sets S_1 and S_2 are pairs of each other, meaning

$$\text{PAIR}(S_1) = S_2.$$

Proof. Recall the moment τ_{v_3} from Definition 168. Let $u \in S_1$. We claim that $\text{PAIR}_u \in S_2$ at this moment.

Since u is unsatisfied at time τ_{v_3} , PAIR_u cannot belong to S'_1 . On the other hand, PAIR_u must eventually be connected to u and thus to v_1 by the end of the algorithm. By Definition 163, PAIR_u cannot become connected to v_1 or v_2 for the first time after τ_{v_3} , as it is unsatisfied at this moment. Therefore, PAIR_u must already be in S'_2 at moment τ_{v_3} .

Moreover, since u is unsatisfied, so is PAIR_u , meaning $\text{PAIR}_u \in \text{UNSATISFIED}(S'_2)$. And since $u \in S_1 \subseteq C$, we also have $\text{PAIR}_u \in C$. Hence, $\text{PAIR}_u \in \text{UNSATISFIED}(S'_2) \cap C = S_2$.

By symmetry, for any $u \in S_2$, we similarly have $\text{PAIR}_u \in S_1$. This completes the proof. □

Lemma 170. There is a lower bound for the value that leads us to consider S_1 and S_2 as an autarkic pair as follows:

$$Y_{S_1} + Y_{S_2} \geq 2r_{v_1}^+ - (6\lambda + 2\kappa)c(T_C).$$

Proof. By Lemma 71, the vertices v_1 and v_2 do not reach each other until time $\min(r_{v_1}^+, r_{v_2}^+)$, which is equal to $r_{v_2}^+$ by Lemma 155, in Boosted Execution. Moreover, since these vertices are pairs of each other, they are actively connected in Legacy Execution and, by Lemma 15, also in Boosted Execution. Therefore, from the beginning of Boosted Execution until time $r_{v_2}^+$, they are both unsatisfied and belong to different active sets, which implies

$$\sum_{\substack{S \subseteq V \\ S \cap \{v_1, v_2\}}} Y_S = \sum_{\substack{S \subseteq V \\ \text{UNSATISFIED}(S) \cap \{v_1, v_2\}}} y_S^+ \geq 2r_{v_2}^+.$$

Furthermore, since $r_{v_3}^+$ is the last time an unsatisfied vertex from C reaches either v_1 or v_2 , after time $r_{v_3}^+$ no additional unsatisfied vertex from C reaches them. This means that for any time $\tau \in (r_{v_3}^+, r_{v_2}^+)$, if S'_τ is the active set containing v_1 at time τ and $S''_\tau = \text{UNSATISFIED}(S'_\tau) \cap C$, then $S''_\tau \subseteq S_1$. Additionally, no vertex in S_1 or S_2 becomes satisfied during the interval $(r_{v_3}^+, r_{v_2}^+)$, because as mentioned earlier, v_1 and v_2 do not meet until time $r_{v_2}^+$, meaning S_1 and S_2 do not meet either. By Lemma 169, the pair of every vertex in S_1 lies in S_2 and vice versa, so they cannot become satisfied until they meet. Therefore, $S_1 \subseteq S''_\tau$. We conclude that $S''_\tau = S_1$, which shows that throughout the interval $(r_{v_3}^+, r_{v_2}^+)$, the unsatisfied vertices of C in the active set containing v_1 are exactly S_1 . A symmetric argument holds for v_2 and S_2 . Thus,

$$\sum_{\substack{S \subseteq V \\ S \cap C \in \{S_1, S_2\}}} Y_S \geq 2r_{v_2}^+ - 2r_{v_3}^+. \quad (24)$$

Moreover, by Definition 165, the total amount of time during which a set S containing unsatisfied vertices from C also includes vertices from other connected components of the optimal solution is at most g^{mix} . Therefore,

$$\begin{aligned} Y_{S_1} + Y_{S_2} &\geq \sum_{\substack{S \subseteq V \\ S \cap C \in \{S_1, S_2\}}} Y_S - \sum_{\substack{S \subseteq V \\ S \cap C \in \{S_1, S_2\} \\ S \not\subseteq C}} Y_S \\ &\geq (2r_{v_2}^+ - 2r_{v_3}^+) - \sum_{\substack{S \subseteq V \\ S \cap C \neq \emptyset \\ S \not\subseteq C}} Y_S && \text{(Equation 24)} \\ &\geq (2r_{v_1}^+ - 2(r_{v_1}^+ - r_{v_2}^+) - 2r_{v_3}^+) - g^{mix} && \text{(Definition 165)} \\ &\geq 2r_{v_1}^+ - 2\lambda c(C) - 2\kappa c(C) - 4\lambda c(C) && \text{(Lemmas 158, 164, and 167)} \\ &= 2r_{v_1}^+ - (6\lambda + 2\kappa)c(C). \end{aligned}$$

□

The next lemmas establish an upper bound on $UM^-(T_C)$ in terms of the cost of T_C . Here, UM^- refers to UM as defined in Definition 18, with respect to Legacy Execution.

Lemma 171. The following inequality holds:

$$UM^-(T_C) \leq 2\lambda c(T_C).$$

Proof. First, note that for any moment τ and any active set $S \subseteq V$ at that moment in Legacy Execution, we have $|\text{REPS}(\text{BASE}(S, \tau)) \cap C| \leq 1$ by Lemma 36. This implies that the value of y_S^- can contribute to r_v^- for at most one vertex $v \in C$. Furthermore, by Lemma 41, y_S^- can be assigned to r_v^- for some vertex $v \in C$ only if $v \in S$ and $\text{PAIR}_v \notin S$. In other words, $S \odot C$. Therefore:

$$\sum_{v \in C} r_v^- \leq \sum_{\substack{S \subseteq V \\ S \odot C}} y_S^- \quad (25)$$

Now we have:

$$\begin{aligned} (1 - \lambda)c(T_C) &< \sum_{v \in C} r_v^+ && \text{(Corollary 84)} \\ &\leq \sum_{v \in C} r_v^- && \text{(Lemma 72)} \\ &\leq \sum_{\substack{S \subseteq V \\ S \odot C}} y_S^- && \text{(Equation 25)} \\ &\leq c(T_C) - \frac{UM^-(T_C)}{2}, && \text{(Lemma 21)} \end{aligned}$$

Rearranging terms yields:

$$UM^-(T_C) \leq 2\lambda c(T_C),$$

as desired. \square

Lemma 172. We can bound $d_{\max}(S_1)$ as follows:

$$d_{\max}(S_1) \leq 2\lambda c(T_C) + 2r_{v_1}^+.$$

Proof. First, we want to show that for every vertex $v \in C$, we can bound the distance between v and PAIR_v by $d(v, \text{PAIR}_v) \leq 2\lambda c(T_C) + 2r_{v_1}^+$.

By Lemma 43, the vertices v and PAIR_v are connected in Legacy Execution at time $r_{\max}^-(C)$. This implies that after time $r_{\max}^-(C)$, no set cuts the pair $\{v, \text{PAIR}_v\}$. Moreover, at each moment before time $r_{\max}^-(C)$, there are at most two active sets containing v or PAIR_v , since active sets at each moment are disjoint. Thus, we have

$$\sum_{\substack{S \subseteq V \\ S \odot \{v, \text{PAIR}_v\}}} y_S^- \leq 2r_{\max}^-(C). \quad (26)$$

In addition, since v and PAIR_v are connected in the optimal solution, there exists a path between them; let us denote this path by $path_v$. Then:

$$\begin{aligned} d(v, \text{PAIR}_v) &\leq c(path_v) \\ &\leq UM^-(path_v) + \sum_{\substack{S \subseteq V \\ S \odot \{v, \text{PAIR}_v\}}} y_S^- && \text{(Lemma 23)} \\ &\leq UM^-(path_v) + 2r_{\max}^-(C) && \text{(Equation 26)} \\ &\leq UM^-(T_C) + 2r_{v_1}^- && \text{(Lemmas 20 and 155)} \\ &\leq 2\lambda c(T_C) + 2r_{v_1}^+. && \text{(Lemma 171, Corollary 156)} \end{aligned}$$

Since this holds for every $v \in C$, by Definition 151, we conclude that

$$d_{\max}(S_1) = \max_{v \in S_1} d(v, \text{PAIR}_v) \leq 2\lambda c(T_C) + 2r_{v_1}^+,$$

completing the proof. \square

Lemma 173. Sets S_1 and S_2 are selected as an autarkic pair.

Proof. For contradiction, assume that S_1 and S_2 are not selected as an autarkic pair. Given Lemma 169, they are pairs of each other. Therefore, not getting selected as autarkic pairs would imply that they do not satisfy the condition in Line 8. This means that

$$(1 + \eta)(Y_{S_1} + Y_{S_2}) \leq d_{\max}(S_1) \quad (27)$$

We can use this to show that

$$\begin{aligned} (1 + \eta)(2r_{v_1}^+ - (6\lambda + 2\kappa)c(T_C)) &\leq (1 + \eta)(Y_{S_1} + Y_{S_2}) && \text{(Lemma 170)} \\ &\leq d_{\max}(S_1) && \text{(Equation 27)} \\ &\leq 2\lambda c(T_C) + 2r_{v_1}^+. && \text{(Lemma 172)} \end{aligned}$$

By moving $r_{v_1}^+$ to one side and other terms to the other side, we get

$$2\eta r_{v_1}^+ \leq (2\lambda + (6\lambda + 2\kappa)(1 + \eta))c(T_C).$$

Then, dividing both sides by 2η results in

$$\begin{aligned} r_{v_1}^+ &\leq \frac{1}{\eta}(\lambda + (3\lambda + \kappa)(1 + \eta))c(T_C) \\ &= \frac{1}{\eta} \left(4\lambda + 3\eta\lambda + 4(1 + \eta)\lambda \frac{1 + \beta}{1 - \beta} \right) c(T_C) && \text{(Definition 153)} \\ &= \frac{\lambda}{\eta} \left(4 + 3\eta + 4(1 + \eta) \frac{1 + \beta}{1 - \beta} \right) c(T_C) \\ &= \gamma c(T_C) && \text{(Definition 153)} \end{aligned}$$

However, since $C \in \mathcal{B}_2$, given Definition 83, we have $r_{\max}^+(C) > \gamma c(T_C)$ which, combined with Lemma 155, gives $r_{v_1}^+ > \gamma c(T_C)$. This contradicts the above inequality and proves that S_1 and S_2 are selected as an autarkic pair. \square

8.3 Cost Analysis of SOL_{AP}

Now that we have established properties for connected components in \mathcal{B}_2 , we first provide a lower bound for Y of all autarkic pairs in Lemma 175. Then, in Lemma 181, we derive an upper bound for the solution SOL_{AP} , which is obtained by executing `AUTARKICPAIRS` in Line 6, and a new upper bound for the solution SOL_{LS} in Lemma 182. Finally, since the cost of our final solution in Algorithm 2 is at most equal to the minimum cost between SOL_{AP} and SOL_{LS} , we combine their bounds to establish an upper bound on the minimum cost between them in Lemma 183.

First, we start by defining a notation for the total Y of autarkic pairs and bounding it.

Definition 174 (Total Autarkic Growth). We define \widehat{Y} as the total value of $Y_S + Y_{\text{PAIR}(S)}$ for subsets of vertices $S, \text{PAIR}(S) \subseteq V$ that are selected as autarkic pairs in Line 8.

Lemma 175. The total autarkic growth has a lower bound as follows:

$$\widehat{Y} \geq 2 \left(1 - \frac{3\lambda + \kappa}{\gamma}\right) \sum_{C \in \mathcal{B}_2} r_{max}^+(C).$$

Proof. For any connected component of the optimal solution $C \in \mathcal{B}_2$, let S_1^C and S_2^C be defined as S_1 and S_2 in Definition 168. According to Lemma 173, we know that they are selected as autarkic pairs. Furthermore, we have a lower bound on the value of $Y_{S_1^C} + Y_{S_2^C}$ using Lemma 170. Therefore, the total autarkic growth is at least the sum of these lower bounds for all connected components in \mathcal{B}_2 , which is

$$\begin{aligned} \widehat{Y} &\geq \sum_{C \in \mathcal{B}_2} \left(Y_{S_1^C} + Y_{S_2^C} \right) && \text{(Lemma 173)} \\ &\geq \sum_{C \in \mathcal{B}_2} \left(2r_{max}^+(C) - (6\lambda + 2\kappa)c(T_C) \right) && \text{(Lemmas 170 and 155)} \\ &> \sum_{C \in \mathcal{B}_2} \left(2r_{max}^+(C) - \frac{6\lambda + 2\kappa}{\gamma} r_{max}^+(C) \right) && \text{(Definition 83)} \\ &= 2 \left(1 - \frac{3\lambda + \kappa}{\gamma}\right) \sum_{C \in \mathcal{B}_2} r_{max}^+(C) \end{aligned}$$

□

Definition 176. Let \widehat{Y}_1 be the total value of y_S^+ for active sets such that $\text{UNSATISFIED}(S)$ is selected as an autarkic pair and S is a single-edge set. Similarly, let \widehat{Y}_2 be the total value of y_S^+ for active sets such that $\text{UNSATISFIED}(S)$ is selected as an autarkic pair and S is a multi-edge set.

The following lemma establishes that $\widehat{Y} = \widehat{Y}_1 + \widehat{Y}_2$ by showing any set S contributing to \widehat{Y} must be a single-edge or multi-edge set.

Lemma 177. We have

$$\widehat{Y} = \widehat{Y}_1 + \widehat{Y}_2.$$

Proof. Given Line 6, for any subset of vertices $S \subseteq V$ such that y_S^+ is added to $Y_{S'}$, the vertices in S' are unsatisfied in S . According to Line 7, if S' is an autarkic pair, it is not an empty set. Therefore, there exists a vertex $v \in S' = \text{UNSATISFIED}(S)$ such that v is unsatisfied, and hence $\text{PAIR}_v \notin S$. Since the optimal solution must connect v and PAIR_v , there is a path between them, and since $S \odot \{v, \text{PAIR}_v\}$, the set S cuts the path between them and consequently cuts the optimal solution. This implies that S either colors exactly one edge of the optimal solution, meaning S is a single-edge set and y_S^+ is added to \widehat{Y}_1 , or colors at least two edges of the optimal solution, meaning S is a multi-edge set and y_S^+ is added to \widehat{Y}_2 . Hence, we conclude that

$$\widehat{Y} = \widehat{Y}_1 + \widehat{Y}_2.$$

□

Recall that G' is defined in Line 12. We now analyze the cost of the optimal solution in G' .

Definition 178. We define OPT' as the optimal solution for G' .

Since we run `LEGACYMOATGROWING` on G' in Line 13, and `LEGACYMOATGROWING` has an approximation factor of 2, we can find a solution with cost at most $2c(\text{OPT}')$ for G' . Therefore, we first aim to bound $c(\text{OPT}')$. Note that instead of using `LEGACYMOATGROWING`, we could call our algorithm recursively on G' to obtain a better-than-2 approximation. However, we use `LEGACYMOATGROWING` to simplify the analysis.

The next lemma is necessary to establish an upper bound for $c(\text{OPT}')$.

Lemma 179. For any vertex $v \in V$, it can only belong to at most one autarkic pair.

Proof. We will prove this by contradiction. Assume that there are two autarkic pairs, $(S', \text{PAIR}(S'))$ and $(S'', \text{PAIR}(S''))$, such that $v \in S'$ and $v \in S''$, with $S' \neq S''$.

According to the autarkic pair condition in Line 8, we have:

$$\begin{aligned} Y_{S'} + Y_{\text{PAIR}(S')} &> \frac{1}{1+\eta} d_{\max}(S') \\ &\geq \frac{1}{1+\eta} d(v, \text{PAIR}_v) \end{aligned} \quad (v \in S, \text{Definition 151})$$

Similarly, we derive the same inequality for S'' :

$$Y_{S''} + Y_{\text{PAIR}(S'')} > \frac{1}{1+\eta} d(v, \text{PAIR}_v).$$

Summing these inequalities, we get:

$$Y_{S'} + Y_{\text{PAIR}(S')} + Y_{S''} + Y_{\text{PAIR}(S'')} > \frac{2}{1+\eta} d(v, \text{PAIR}_v). \quad (28)$$

Let $\mathcal{S} = \{S', \text{PAIR}(S'), S'', \text{PAIR}(S'')\}$. We know that \mathcal{S} contains four different subsets of vertices, and according to Line 6, the growth y_S^+ of any set $S \subseteq V$ is assigned to at most one of the sets in \mathcal{S} . Moreover, since every set in \mathcal{S} cuts the pair $\{v, \text{PAIR}_v\}$, any subset $S \subseteq V$ for which $\text{UNSATISFIED}(S) \in \mathcal{S}$ (meaning its growth y_S^+ is assigned to the Y of one of the sets in \mathcal{S}) must also cut $\{v, \text{PAIR}_v\}$. If S does not cut $\{v, \text{PAIR}_v\}$, then $\text{UNSATISFIED}(S) \cap \{v, \text{PAIR}_v\} = \emptyset$, implying that $\text{UNSATISFIED}(S) \notin \mathcal{S}$. Therefore,

$$\begin{aligned} \sum_{\substack{S \subseteq V \\ S \cap \{v, \text{PAIR}_v\} \neq \emptyset}} y_S^+ &\geq \sum_{\substack{S \subseteq V \\ \text{UNSATISFIED}(S) \in \mathcal{S}}} y_S^+ \\ &= \sum_{S \in \mathcal{S}} Y_S \\ &> \frac{2}{1+\eta} d(v, \text{PAIR}_v) \quad (\text{Equation 28}) \\ &> d(v, \text{PAIR}_v) \quad (0 < \eta < 1) \end{aligned}$$

However, this contradicts the fact that the sets $S \subseteq V$ that cut $\{v, \text{PAIR}_v\}$ in Legacy Execution color the shortest path between v and PAIR_v for the duration of y_S^+ , and each portion of an edge can only be colored once. Therefore, it is not possible for the sum to exceed the distance $d(v, \text{PAIR}_v)$.

Thus, our assumption is false, and we conclude that v can belong to at most one autarkic pair. \square

Lemma 180. We can bound the cost of OPT' as follows:

$$c(\text{OPT}') \leq c(\text{OPT}) - \widehat{Y}_1.$$

Proof. To prove the lemma, we first provide a valid solution for G' and show that its cost is at most $c(\text{OPT}) - \widehat{Y}_1$. Then, since OPT' is the optimal solution for G' , its cost must be less than or equal to the cost of any valid solution for G' . Let SOL' denote the constructed solution for G' .

Initially, we set SOL' equal to $\text{OPT} \cup E_{AP}$, which clearly satisfies all demands. Then, for any single-edge set S —that is, S cuts exactly one edge of OPT —such that $\text{UNSATISFIED}(S)$ is selected as an autarkic pair, we remove from SOL' the edge of OPT that is cut by S . Note that while S is a single-edge set with respect to OPT , it may cut more than one edge in SOL' since SOL' also includes the edges of E_{AP} . Moreover, for each such removed, we have added a corresponding edge to E_{AP} in Line 11, and thus to SOL' , and will later show that these additions preserve feasibility. Clearly, the cost of SOL' is at most $c(\text{OPT}) - \widehat{Y}_1$, since the total cost of removed edges is at least \widehat{Y}_1 , and all edges in E_{AP} have zero cost.

Next, we need to prove that SOL' satisfies all demands, given that some edges from OPT have been removed but edges of E_{AP} between autarkic pairs have been added.

Let e be an edge that was part of OPT and got removed from SOL' because a single-edge set S cuts e and $\text{UNSATISFIED}(S)$ is selected as an autarkic pair. Given Lemma 24, removing e only disconnects vertices in $\text{UNSATISFIED}(S)$ from their pair and does not affect other vertices. Therefore, we need to show that those vertices are still satisfied in SOL' . Since S was an autarkic pair, we have added a zero-cost edge from a vertex $v \in S$ to PAIR_v . If we show that all vertices in $\text{UNSATISFIED}(S)$ are connected to v and all vertices in $\text{PAIR}(\text{UNSATISFIED}(S))$ are connected to PAIR_v in SOL' , we can conclude that all vertices in $\text{UNSATISFIED}(S)$ are satisfied in SOL' . We observe that the leaves of T_1 are contained in $\text{UNSATISFIED}(S)$, and leaves of T_2 are contained in $\text{PAIR}(\text{UNSATISFIED}(S))$.

We observe that since S is a single-edge set, $\text{UNSATISFIED}(S)$ can only include vertices from a single component of \mathcal{C} . Therefore, we can define tree T_1 as the minimal subtree of OPT that connects vertices in $\text{UNSATISFIED}(S)$. Similarly, since $\text{PAIR}(\text{UNSATISFIED}(S))$ must lie within the same component, we define T_2 as the minimal subtree of OPT that connects the vertices in $\text{PAIR}(\text{UNSATISFIED}(S))$. Figure 10 illustrates the structure of an autarkic pair, highlighting the aforementioned notation.

Assume, for the sake of contradiction, that there is an edge in T_1 that is removed from SOL' . This edge is removed by some single-edge set $S' \subseteq V$ where $\text{UNSATISFIED}(S')$ is selected as an autarkic pair. Since S' only cuts one edge of the optimal solution and also cuts T_1 , it only cuts one edge of T_1 . According to Lemma 22, S' must cut the leaves of T_1 , which means $\text{UNSATISFIED}(S') \subseteq S'$ cannot be $\text{UNSATISFIED}(S)$ as $\text{UNSATISFIED}(S)$ contains all leaves of T_1 . Moreover, there must be a vertex $v \in S'$ that is a leaf of T_1 .

We can also show that $v \in \text{UNSATISFIED}(S')$. Otherwise, PAIR_v would need to be in S' . Since $v \in \text{UNSATISFIED}(S)$, $\text{PAIR}_v \notin S$, therefore S would have some vertices of S' but not all of them. Moreover, since S' cuts leaves of T_1 , it also has some vertices from S but not all of them. This would imply that S and S' intersect but are not subsets of each other, which contradicts the laminarity of active sets as per Corollary 11. Thus, $v \in \text{UNSATISFIED}(S')$.

Now that we know $\text{UNSATISFIED}(S)$ and $\text{UNSATISFIED}(S')$ are distinct autarkic pairs and v is in both of them, we can apply Lemma 179, which states that each vertex can belong to at most one autarkic pair. This contradiction shows that no edge from T_1 can be removed.

The same argument can be applied to T_2 . Therefore, no edge from T_1 and T_2 is removed from SOL' , and all vertices in $\text{UNSATISFIED}(S)$ are connected to v and all vertices in $\text{PAIR}(\text{UNSATISFIED}(S))$ are connected to PAIR_v . Since v and PAIR_v are connected in E_{AP} and consequently in SOL' , removing e does not violate any pair demand, and therefore, SOL' satisfies all demands. \square

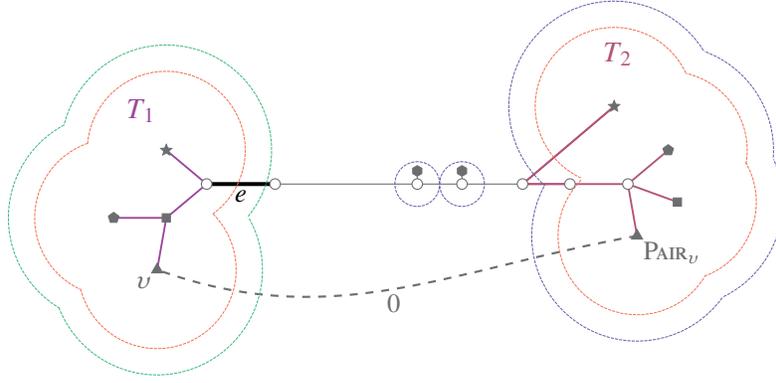


Figure 10: An autarkic pair within a component of the optimal solution. The moats on the right are multi-edge sets and therefore do not trigger any edge removal. However, the moats on the left cut only the edge e , allowing us to remove it from SOL' . This removal only impacts the unsatisfied vertices within that moat. To maintain feasibility, we add an edge between v and PAIR_v to both E_{AP} and our solution. By ensuring that all edges in T_1 and T_2 —the subtrees of the optimal solution connecting each subset of the autarkic pair—are retained in SOL' , we guarantee that all vertices in the autarkic pair remain satisfied.

Lemma 181. We can give an upper bound for SOL_{AP} as follows:

$$SOL_{AP} \leq 2c(\text{OPT}) - (1 - \eta)\widehat{Y} + 2\widehat{Y}_2.$$

Proof. Since we only select autarkic pairs that satisfy $(1 + \eta)(Y_S + Y_{\text{PAIR}(S)}) \geq d_{\max}(S)$ based on Line 8, and we add an edge between $v \in S$ and $\text{PAIR}_v \in \text{PAIR}(S)$ with a cost $d(v, \text{PAIR}_v) \leq d_{\max}(S)$, the cost of the selected edge in Line 10 for an autarkic pair is at most $(1 + \eta)(Y_S + Y_{\text{PAIR}(S)})$. This means the total cost of all selected edges is at most $(1 + \eta)\widehat{Y}$.

Since we run the 2-approximation algorithm on G' , we obtain a solution with cost $2c(\text{OPT}')$. Including the cost of the selected edges in Line 10, the total cost is at most:

$$\begin{aligned} SOL_{AP} &\leq 2c(\text{OPT}') + (1 + \eta)\widehat{Y} \\ &\leq 2(c(\text{OPT}) - \widehat{Y}_1) + (1 + \eta)\widehat{Y} && \text{(Lemma 180)} \\ &= 2c(\text{OPT}) - 2(\widehat{Y} - \widehat{Y}_2) + (1 + \eta)\widehat{Y} && \text{(Definition 176)} \\ &= 2c(\text{OPT}) - (1 - \eta)\widehat{Y} + 2\widehat{Y}_2 \end{aligned}$$

□

Lemma 182. We can either give an upper bound for SOL_{LS} as follows:

$$SOL_{LS} \leq \left(2 + 2\alpha + \frac{4\alpha}{\beta}\right)c(\text{OPT}) - 2\widehat{Y}_2,$$

or SOL_{LS} is a $2 - 2\alpha$ approximation of the optimal solution OPT .

Proof. If $c(SOL_{LS}) \leq (2 - 2\alpha)c(\text{OPT})$, the lemma is already proved. Therefore, we assume that this condition does not hold. In this case, we can bound $loss_1$ and win_1 using Lemmas 85 and 86.

Let MC^+ be MC as defined in Definition 18, corresponding to Boosted Execution. By Definition 176, during the duration of \widehat{Y}_2 , at least two edges of the optimal solution are being colored by each active set. Thus, the total portion of OPT colored by these active sets is at least $2\widehat{Y}_2$, since each active set colors at least two edges. Therefore,

$$2\widehat{Y}_2 \leq MC^+(OPT). \quad (29)$$

Additionally, each active set either cuts the optimal solution (meaning $S \odot OPT$) or does not cut it (meaning $\delta(S) \cap OPT = \emptyset$). Both cannot happen simultaneously. Hence, we divide all active sets and their corresponding y_S^+ between these two groups to bound SOL_{LS} .

$$\begin{aligned} SOL_{LS} &\leq 2 \sum_{S \in V} y_S^+ && \text{(Lemma 9)} \\ &= 2 \left(\sum_{\substack{S \in V \\ S \odot OPT}} y_S^+ + \sum_{\substack{S \in V \\ \delta(S) \cap OPT = \emptyset}} y_S^+ \right) \\ &\leq 2 \left(c(OPT) - \frac{MC^+(OPT)}{2} + \sum_{\substack{S \in V \\ \delta(S) \cap OPT = \emptyset}} y_S^+ \right) && \text{(Lemma 21)} \\ &\leq 2(c(OPT) + win_1 + loss_1) - MC^+(OPT) && \text{(Lemma 81)} \\ &\leq 2c(OPT) + 2\left(\alpha + \frac{\alpha}{\beta}\right)c(OPT) + 2\frac{\alpha}{\beta}c(OPT) - MC^+(OPT) && \text{(Lemmas 86 and 85)} \\ &\leq \left(2 + 2\alpha + \frac{4\alpha}{\beta}\right)c(OPT) - 2\widehat{Y}_2. && \text{(Equation 29)} \end{aligned}$$

□

Lemma 183. We can either bound the best solution between SOL_{AP} and SOL_{LS} as follows:

$$\min(c(SOL_{AP}), c(SOL_{LS})) \leq \left(2 + \alpha + \frac{2\alpha}{\beta}\right)c(OPT) - (1 - \eta) \left(1 - \frac{3\lambda + \kappa}{\gamma}\right) \sum_{C \in \mathcal{B}_2} r_{max}^+(C),$$

or SOL_{LS} is a $2 - 2\alpha$ approximation of the optimal solution OPT .

Proof. If $c(SOL_{LS}) \leq (2 - 2\alpha)c(OPT)$, the proof is done. Therefore, we assume that this condition does not hold. In this case, we can use the bound in Lemma 182 for SOL_{LS} . Therefore, we can prove our claim as follows:

$$\begin{aligned} \min(c(SOL_{AP}), c(SOL_{LS})) &\leq \frac{SOL_{AP} + SOL_{LS}}{2} \\ &\leq \frac{(2c(OPT) - (1 - \eta)\widehat{Y} + 2\widehat{Y}_2) + \left(\left(2 + 2\alpha + \frac{4\alpha}{\beta}\right)c(OPT) - 2\widehat{Y}_2\right)}{2} && \text{(Lemmas 181 and 182)} \\ &= \left(2 + \alpha + \frac{2\alpha}{\beta}\right)c(OPT) - \frac{1 - \eta}{2}\widehat{Y} \\ &\leq \left(2 + \alpha + \frac{2\alpha}{\beta}\right)c(OPT) - (1 - \eta) \left(1 - \frac{3\lambda + \kappa}{\gamma}\right) \sum_{C \in \mathcal{B}_2} r_{max}^+(C). && \text{(Lemma 175)} \end{aligned}$$

□

9 Approximation for Steiner Forest: Proof of Theorem 1

This section establishes that Algorithm 2 has an approximation factor of $(2 - 2\alpha)$ in polynomial time.

Proof of Theorem 1. We claim that Algorithm 2 deterministically solves the Steiner Forest problem in polynomial time with an approximation factor of $2 - 10^{-11}$.

Recall that our final solution is chosen from three possible solutions, as specified in Line 7. Our goal is to show that the minimum cost among these three solutions is at most $(2 - 2\alpha) \cdot c(\text{OPT})$ for some value of $\alpha \geq \frac{10^{-11}}{2}$. To this end, we define the forest ALG as the best choice among SOL_{LS} , SOL_{XT} , and SOL_{AP} .

First, all three solutions satisfy the demands as explained below:

- SOL_{LS} is obtained from Boosted Execution, where the fingerprint is larger than that of Legacy Execution. By Lemma 30, this ensures that the demands are met.
- SOL_{XT} is obtained from Extended-Boosted Execution, where the fingerprint is larger as that of Legacy Execution. This follows from the fact that $t''_v \geq t'_v \geq t^+_v \geq t^-_v$ for all $v \in V$. Thus, by Lemma 30, the demands are satisfied.
- SOL_{AP} selects some paths, adds a zero-cost edge between their endpoints, and then applies Legacy Moat Growing. Since Legacy Moat Growing satisfies all demands, replacing the zero-cost edges with the corresponding selected paths also satisfies all demands.

Second, all these three solutions are obtained in polynomial time since `LEGACYMOATGROWING`, `LOCALSEARCH`, `AUTARKICPAIRS`, and `EXTEND` run in polynomial time (see Lemma 63; Corollaries 4, 94, and 152; and Figure 4).

In the remainder, we prove the following:

$$c(ALG) \leq (2 - 2\alpha) \cdot c(\text{OPT})$$

In the previous sections, we analyzed each solution and established several bounds. Specifically, we have proven that:

1. either SOL_{LS} is a $(2 - 2\alpha)$ -approximation solution or the upper bound for SOL_{LS} in Lemma 87 holds,
2. either SOL_{LS} or SOL_{XT} is a $(2 - 2\alpha)$ -approximation solution, or the upper bound for SOL_{XT} in Lemma 150 holds, and
3. either SOL_{LS} is a $(2 - 2\alpha)$ -approximation solution or the upper bound given in Lemma 183 holds for the better solution between SOL_{AP} and SOL_{LS} .

First, if for the chosen α , either SOL_{LS} , SOL_{XT} , or SOL_{AP} provides a $(2 - 2\alpha)$ -approximation, then the theorem follows immediately:

$$c(ALG) \leq \min(c(SOL_{LS}), c(SOL_{XT}), c(SOL_{AP})) \leq (2 - 2\alpha) \cdot c(\text{OPT}).$$

Consequently, from the listed lemmas, we obtain three upper bounds: one for $c(SOL_{LS})$, one for $c(SOL_{XT})$, and one for $\min(c(SOL_{LS}), c(SOL_{AP}))$. Since

$$c(ALG) \leq \min(c(SOL_{LS}), c(SOL_{XT}), c(SOL_{AP})),$$

these bounds also apply to $c(ALG)$.

We index these bounds according to their positions in the list above. The i -th bound follows the general form

$$c(ALG) \leq \kappa_{i,\mathcal{A}} \cdot c(\mathcal{A}) + \kappa_{i,\mathcal{B}_1} \cdot c(\mathcal{B}_1) + \kappa_{i,\mathcal{B}_2} \cdot c(\mathcal{B}_2) + \kappa_{i,r} \cdot \sum_{C \in \mathcal{B}} r_{max}^+(C),$$

where each coefficient $\kappa_{i,j}$ is a function of the algorithm's parameters and is determined by the i -th bound.

We form a weighted sum of these three upper bounds by assigning a weight of ω_i to the i -th upper bound such that $\sum_{i=1}^3 \omega_i = 1$ and $\omega_i \geq 0$. It follows that

$$c(ALG) \leq \sum_{i=1}^3 \omega_i \cdot \left(\kappa_{i,\mathcal{A}} \cdot c(\mathcal{A}) + \kappa_{i,\mathcal{B}} \cdot c(\mathcal{B}) + \kappa_{i,r} \cdot \sum_{C \in \mathcal{B}} r_{max}^+(C) \right).$$

Therefore, it suffices to prove the existence of *proper values* for the parameters of the algorithm and the analysis ($\alpha, \beta, \eta, \lambda, \varepsilon$, and w), along with ω_1, ω_2 , and ω_3 , such that

$$\sum_{i=1}^3 \omega_i \cdot \left(\kappa_{i,\mathcal{A}} \cdot c(\mathcal{A}) + \kappa_{i,\mathcal{B}} \cdot c(\mathcal{B}) + \kappa_{i,r} \cdot \sum_{C \in \mathcal{B}} r_{max}^+(C) \right) \leq (2 - 2\alpha) \cdot c(OPT) \quad (30)$$

It is important to note that *proper values* must fit within the constraints of the algorithm's parameters, as explained in the lemmas' statements, as well as the constraints related to the weighted sum ω .

Observing Definitions 82 and 83, we obtain $c(OPT) = c(\mathcal{A}) + c(\mathcal{B}_1) + c(\mathcal{B}_2)$. To proceed, we show that there exist suitable values for the parameters and ω such that

$$\sum_{i=1}^3 \omega_i \cdot \kappa_{i,\mathcal{A}} \leq (2 - 2\alpha), \quad (31)$$

$$\sum_{i=1}^3 \omega_i \cdot \kappa_{i,\mathcal{B}_1} \leq (2 - 2\alpha), \quad (32)$$

$$\sum_{i=1}^3 \omega_i \cdot \kappa_{i,\mathcal{B}_2} \leq (2 - 2\alpha), \text{ and} \quad (33)$$

$$\sum_{i=1}^3 \omega_i \cdot \kappa_{i,r} \leq 0. \quad (34)$$

Summing these inequalities with the factors $c(\mathcal{A})$, $c(\mathcal{B}_1)$, $c(\mathcal{B}_2)$, and $\sum_{C \in \mathcal{B}} r_{max}^+(C)$ leads directly to inequality 30, which completes the proof.

For the left hand side of inequality 31, we have

$$\begin{aligned} \sum_{i=1}^3 \omega_i \cdot \kappa_{i,\mathcal{A}} &= 2\omega_1 \left((1 - \lambda) + \frac{\alpha}{\beta} \right) && \text{(Lemma 87)} \\ &+ 2\omega_2 (1 + \varepsilon)(1 - \lambda) && \text{(Lemma 150)} \\ &+ 2\omega_2 \cdot \max \left(\frac{\alpha}{\beta}, \frac{(1 - w'(1 + \beta)(1 - w))(\alpha + \varepsilon)}{\beta}, \frac{\alpha + (1 - \alpha - \alpha/\beta)(1 - w'(1 + \beta)(1 - w))\varepsilon}{\beta} \right) && \text{(Lemma 150)} \\ &+ \omega_3 \left(2 + \alpha + \frac{2\alpha}{\beta} \right). && \text{(Lemma 183)} \end{aligned}$$

For the left hand side of inequality 32, we have

$$\begin{aligned}
\sum_{i=1}^3 \omega_i \cdot \kappa_{i, \mathcal{B}_1} &= 2\omega_1 \left(1 + \frac{\alpha}{\beta}\right) && \text{(Lemma 87)} \\
&+ 2\omega_2 \left(1 + \epsilon - \epsilon w' (1 + (1 - 2w)(1 - \lambda)) + \gamma w' \left(1 + w \frac{1 - \beta}{5 + \beta}\right)\right) && \text{(Lemma 150)} \\
&+ 2\omega_2 \cdot \max\left(\frac{\alpha}{\beta}, \frac{(1 - w'(1 + \beta)(1 - w))(\alpha + \epsilon)}{\beta}, \frac{\alpha + (1 - \alpha - \alpha/\beta)(1 - w'(1 + \beta)(1 - w))\epsilon}{\beta}\right) && \text{(Lemma 150)} \\
&+ \omega_3 \left(2 + \alpha + \frac{2\alpha}{\beta}\right). && \text{(Lemma 183)}
\end{aligned}$$

Constraint	
$0 < \beta, \epsilon, \eta < 1$	Algorithm 2 Assumptions
$0 < w \leq \frac{1}{2}$	Lemma 150
$w' \geq 1 / \left(1 + \frac{1 - \beta}{5 + \beta} w\right)$	Lemma 150
$w' \leq (1 + \epsilon) / \left(1 + 3\epsilon - \epsilon w \frac{9 + 3\beta}{5 + \beta}\right)$	Lemma 150
$\varkappa = 4\lambda \frac{1 + \beta}{1 - \beta}$	Definition 153
$\gamma = \frac{\lambda}{\eta} \left(4 + 3\eta + 4(1 + \eta) \frac{1 + \beta}{1 - \beta}\right)$	Definition 153
$\sum_{i=1}^3 \omega_i = 1$	Weighted Sum
$\sum_{i=1}^3 \omega_i \cdot \kappa_{i, \mathcal{A}} \leq (2 - 2\alpha)$	Inequality 31, Explicit form
$\sum_{i=1}^3 \omega_i \cdot \kappa_{i, \mathcal{B}_1} \leq (2 - 2\alpha)$	Inequality 32, Explicit form
$\sum_{i=1}^3 \omega_i \cdot \kappa_{i, \mathcal{B}_2} \leq (2 - 2\alpha)$	Inequality 33, Explicit form
$\sum_{i=1}^3 \omega_i \cdot \kappa_{i, r} \leq 0$	Inequality 34, Explicit form

Table 1: *Conditions for a $(2 - 2\alpha)$ -Approximation.* If we can determine values for the previously defined parameters that satisfy the given constraints, the existence of a $(2 - 2\alpha)$ -approximation for PCSF is guaranteed.

For the left hand side of inequality 33, we have

$$\begin{aligned}
\sum_{i=1}^3 \omega_i \cdot \kappa_{i,\mathcal{B}_2} &= 2\omega_1 \left(1 + \frac{\alpha}{\beta}\right) && \text{(Lemma 87)} \\
&+ 2\omega_2 (1 + \epsilon - \epsilon w' (1 + (1 - 2w)(1 - \lambda))) && \text{(Lemma 150)} \\
&+ 2\omega_2 \cdot \max \left(\frac{\alpha}{\beta}, \frac{(1 - w'(1 + \beta)(1 - w))(\alpha + \epsilon)}{\beta}, \frac{\alpha + (1 - \alpha - \alpha/\beta)(1 - w'(1 + \beta)(1 - w))\epsilon}{\beta} \right) && \text{(Lemma 150)} \\
&+ \omega_3 \left(2 + \alpha + \frac{2\alpha}{\beta}\right). && \text{(Lemma 183)}
\end{aligned}$$

Finally, for the left hand side of inequality 34, we have

$$\begin{aligned}
\sum_{i=1}^3 \omega_i \cdot \kappa_{i,r} &= 2\omega_1 \cdot 0 && \text{(Lemma 87)} \\
&+ 2\omega_2 \cdot w' \left(1 + w \frac{1 - \beta}{5 + \beta}\right) && \text{(Lemma 150)} \\
&- \omega_3 (1 - \eta) \left(1 - \frac{3\lambda + \varkappa}{\gamma}\right). && \text{(Lemma 183)}
\end{aligned}$$

Now, we just need to provide values for the algorithm's parameters and weights such that all these conditions (Table 1) hold. This can be viewed as a numerical optimization problem with the goal of maximizing α . However, for the purpose of this theorem, we must provide values that satisfy the constraints while ensuring $\alpha \geq \frac{10^{-11}}{2}$.

To this end, we provide values for the algorithm's parameters and weights ω_i in Table 2. □

Parameter	Value	Fractional Value
α	$5.000000000000 \times 10^{-12}$	$\frac{1}{200000000000}$
β	$1.000000000000 \times 10^{-1}$	$\frac{1}{10}$
λ	$1.183079480386 \times 10^{-8}$	$\frac{3575638326237933}{302231454903657293676544}$
ϵ	$3.438743241429 \times 10^{-7}$	$\frac{6495602330607721}{18889465931478580854784}$
w	$1.000000000000 \times 10^{-2}$	$\frac{1}{100}$
w'	$9.999993185227 \times 10^{-1}$	$\frac{32112103126037549486258500}{32112125009721801303670549}$
η	$5.000000000000 \times 10^{-1}$	$\frac{1}{2}$
\varkappa	$5.783944126333 \times 10^{-8}$	$\frac{13110673862872421}{226673591177742970257408}$
γ	$3.036570666325 \times 10^{-7}$	$\frac{91774717040106947}{302231454903657293676544}$
ω_1	$4.960317460317 \times 10^{-1}$	$\frac{125}{252}$
ω_2	$7.936507936508 \times 10^{-3}$	$\frac{1}{126}$
ω_3	$4.960317460317 \times 10^{-1}$	$\frac{125}{252}$

Table 2: *Parameters and Values.* This table presents values for the algorithm's parameters that ensure the provided algorithm is an $(2 - 10^{-11})$ -approximation. Scientific notation provides an estimate for each parameter, while fractional representation helps verify the correctness of constraints without losing precision.

10 Acknowledgements

This work is partially supported by DARPA QuICC, ONR MURI 2024 award on Algorithms, Learning, and Game Theory, Army-Research Laboratory (ARL) grant W911NF2410052, NSF AF:Small grants 2218678, 2114269, 2347322.

References

- [AABV95] Baruch Awerbuch, Yossi Azar, Avrim Blum, and Santosh S. Vempala. Improved approximation guarantees for minimum-weight k -trees and prize-collecting salesmen. In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 277–283. ACM, 1995.
- [ABHK11] Aaron Archer, MohammadHossein Bateni, MohammadTaghi Hajiaghayi, and Howard J. Karloff. Improved approximation algorithms for prize-collecting steiner tree and TSP. *SIAM J. Comput.*, 40(2):309–332, 2011.
- [AGH⁺24] Ali Ahmadi, Iman Gholami, MohammadTaghi Hajiaghayi, Peyman Jabbarzade, and Mohammad Mahdavi. Prize-collecting steiner tree: A 1.79 approximation. In Bojan Mohar, Igor Shinkar, and Ryan O’Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1641–1652. ACM, 2024.
- [AGH⁺25] Ali Ahmadi, Iman Gholami, MohammadTaghi Hajiaghayi, Peyman Jabbarzade, and Mohammad Mahdavi. 2-approximation for prize-collecting steiner forest. *J. ACM*, March 2025. Just Accepted. Preliminary version appeared in SODA 2024.
- [AK06] Sanjeev Arora and George Karakostas. A $2 + \epsilon$ approximation algorithm for the k -mst problem. *Math. Program.*, 107(3):491–504, 2006.
- [AKR91] Ajit Agrawal, Philip 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.
- [AKR95] Ajit Agrawal, Philip 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.
- [AR98] Sunil Arya and H. Ramesh. A 2.5-factor approximation algorithm for the k -mst problem. *Inf. Process. Lett.*, 65(3):117–118, 1998.
- [BCC⁺10] Aditya Bhaskara, Moses Charikar, Eden Chlamtac, Uriel Feige, and Aravindan Vijayaraghavan. Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k -subgraph. In Leonard J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 201–210. ACM, 2010.
- [BGRS10] Jaroslav Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. An improved lp-based approximation for steiner tree. In Leonard 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.

- [BGT24] Jaroslav Byrka, Fabrizio Grandoni, and Vera Traub. On the bidirected cut relaxation for steiner forest. *CoRR*, abs/2412.06518, 2024.
- [BH12] MohammadHossein Bateni and MohammadTaghi Hajiaghayi. Euclidean prize-collecting steiner forest. *Algorithmica*, 62(3-4):906–929, 2012.
- [BHL18] Mohammad Hossein Bateni, Mohammad Taghi Hajiaghayi, and Vahid Liaghat. Improved approximation algorithms for (budgeted) node-weighted steiner problems. *SIAM J. Comput.*, 47(4):1275–1293, 2018.
- [BHM11] MohammadHossein Bateni, Mohammad Taghi Hajiaghayi, and Dániel Marx. Approximation schemes for steiner forest on planar graphs and graphs of bounded treewidth. *J. ACM*, 58(5):21:1–21:37, 2011.
- [BKM15] Glencora Borradaile, Philip N. Klein, and Claire Mathieu. A polynomial-time approximation scheme for euclidean steiner forest. *ACM Trans. Algorithms*, 11(3):19:1–19:20, 2015.
- [BP89] Marshall W. Bern and Paul E. Plassmann. The steiner problem with edge lengths 1 and 2. *Inf. Process. Lett.*, 32(4):171–176, 1989.
- [BRV99] Avrim Blum, R. Ravi, and Santosh S. Vempala. A constant-factor approximation algorithm for the k -mst problem. *J. Comput. Syst. Sci.*, 58(1):101–108, 1999.
- [CC08] Miroslav Chlebík and Janka Chlebíková. The steiner tree problem on graphs: Inapproximability results. *Theor. Comput. Sci.*, 406(3):207–214, 2008.
- [Gar96] Naveen Garg. A 3-approximation for the minimum tree spanning k vertices. In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 302–309. IEEE Computer Society, 1996.
- [Gar05] Naveen Garg. Saving an epsilon: a 2-approximation for the k -mst problem in graphs. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 396–402. ACM, 2005.
- [GGK⁺18] Martin Groß, Anupam Gupta, Amit Kumar, Jannik Matuschke, Daniel R. Schmidt, Melanie Schmidt, and José Verschae. A local-search algorithm for steiner forest. In Anna R. Karlin, editor, *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, volume 94 of *LIPICs*, pages 31:1–31:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- [GK15] Anupam Gupta and Amit Kumar. Greedy algorithms for steiner forest. In Rocco A. Servedio and Ronitt 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.
- [GKL⁺07] Anupam Gupta, Jochen Könemann, Stefano Leonardi, R. Ravi, and Guido Schäfer. An efficient cost-sharing mechanism for the prize-collecting steiner forest problem. In Nikhil Bansal, Kirk Pruhs, and Clifford 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.
- [GW92] Michel X. Goemans and David 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.

- [GW95] Michel X. Goemans and David P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995.
- [HJ06] Mohammad Taghi Hajiaghayi and Kamal 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.
- [HKKN12] Mohammad Taghi Hajiaghayi, Rohit Khandekar, Guy Kortsarz, and Zeev Nutov. Prize-collecting steiner network problems. *ACM Trans. Algorithms*, 9(1):2:1–2:13, 2012.
- [HN10] Mohammad Taghi Hajiaghayi and Arefeh A. Nasri. Prize-collecting steiner networks via iterative rounding. In Alejandro 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.
- [Jai98] Kamal Jain. Factor 2 approximation algorithm for the generalized steiner network problem. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 448–457. IEEE Computer Society, 1998.
- [Jai01] Kamal Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *Comb.*, 21(1):39–60, 2001.
- [JP95] Michael Jünger and William R. Pulleyblank. New primal and dual matching heuristics. *Algorithmica*, 13(4):357–386, 1995.
- [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James 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.
- [KKG21] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. A (slightly) improved approximation algorithm for metric TSP. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 32–45. ACM, 2021.
- [KKG22] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. A (slightly) improved bound on the integrality gap of the subtour LP for TSP. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 832–843. IEEE, 2022.
- [KKG23] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. A deterministic better-than-3/2 approximation algorithm for metric TSP. In Alberto Del Pia and Volker Kaibel, editors, *Integer Programming and Combinatorial Optimization - 24th International Conference, IPCO 2023, Madison, WI, USA, June 21-23, 2023, Proceedings*, volume 13904 of *Lecture Notes in Computer Science*, pages 261–274. Springer, 2023.
- [KMB81] Lawrence T. Kou, George Markowsky, and Leonard Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15:141–145, 1981.
- [KZ95] Marek Karpinski and Alexander Zelikovsky. New approximation algorithms for the steiner tree problems. *Electron. Colloquium Comput. Complex.*, TR95-030, 1995.

- [KZ97] Marek Karpinski and Alexander Zelikovsky. New approximation algorithms for the steiner tree problems. *J. Comb. Optim.*, 1(1):47–65, 1997.
- [RSM⁺94] R. Ravi, Ravi Sundaram, Madhav V. Marathe, Daniel J. Rosenkrantz, and S. S. Ravi. Spanning trees short or small. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*, pages 546–555. ACM/SIAM, 1994.
- [RZ05] Gabriel Robins and Alexander Zelikovsky. Tighter bounds for graph steiner tree approximation. *SIAM J. Discret. Math.*, 19(1):122–134, 2005.
- [SSW07] Yogeshwer Sharma, Chaitanya Swamy, and David P. Williamson. Approximation algorithms for prize collecting forest problems with submodular penalty functions. In Nikhil Bansal, Kirk Pruhs, and Clifford 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.
- [STV18] Ola Svensson, Jakub Tarnawski, and László A. Végh. A constant-factor approximation algorithm for the asymmetric traveling salesman problem. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 204–213. ACM, 2018.
- [TZ22] Vera Traub and Rico Zenklusen. Local search for weighted tree augmentation and steiner tree. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 3253–3272. SIAM, 2022.
- [WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.
- [Zel93] Alexander Zelikovsky. An $11/6$ -approximation algorithm for the network steiner problem. *Algorithmica*, 9(5):463–470, 1993.

A Tight Examples and Counterexamples

Here, we provide some instances to give a better intuition on different components of our algorithm.

A.1 The Strength of Local Search

Intuitively, a local search algorithm for the Steiner Forest problem might start with Legacy Moat Growing algorithm and try possible moves to improve the overall cost of the solution. One intuitive set of moves for such an algorithm would be our choice of Boost actions, which keep vertices active for longer, even if they have no unsatisfied demands. One important argument here is to determine whether keeping a vertex active for a longer period of time is beneficial. A natural idea is to compare the cost of the solution before and after a boost action. However, this example demonstrates that the idea does not work. Instead, it led us to observe a different objective: the total growth of active sets decreases after applying a boost action. Since twice this value serves as a bound for our solution, this observation resulted in an improvement that we leveraged to obtain our results.

No Cost-Based Boost Exists. For this example, we prove that Legacy Moat Growing results in a solution with a cost of at least $2 - \xi$ times the optimal solution for any $\xi > 0$ such that no boost action improving the

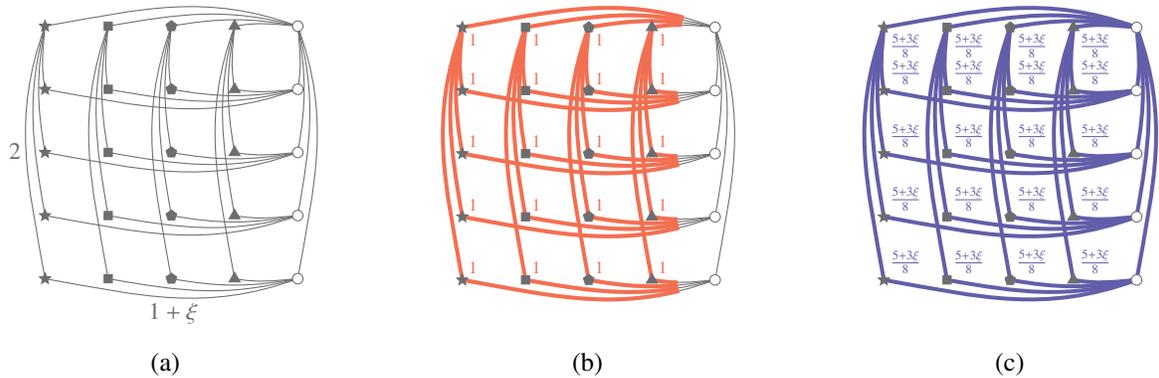


Figure 11: An illustration of an instance on a grid-like graph, which presents a challenging instance for local search algorithms that prioritize minimizing total cost. (a) The grid structure of the instance: edges within each column have length 2, and edges within each row have length $1 + \xi$, with sufficiently small $\xi > 0$. In the corresponding Steiner Forest instance, each column—except the rightmost one—must be connected. (b) The outcome of `LEGACYMOATGROWING` on this instance, which fails to achieve a better than 2 approximation. The vertical edges in each column (except the rightmost) are selected, while all horizontal edges remain partially uncolored. Vertex labels indicate a possible assignment of growth values. (c) The outcome of our local search algorithm, given appropriate values of β . All horizontal edges are selected, and vertical edges are fully colored; however, only a subset forming a spanning tree is ultimately chosen. The growth assignments differ from part (b), reflecting the total *win* value captured by our local search.

cost is possible. Then, the boost-based local search algorithm focusing on the cost of the solution will fail to produce a better than 2 approximation.

Consider the example illustrated in Figure 11 with n rows and m columns and parameter ξ . In this graph, vertices in each row are connected to the vertex in the last column with edges of length $1 + \xi$, and vertices in each column are connected to the first vertex in their column using edges of length 2. The demands for this instance require that each column except for the rightmost one form a connected component.

In the initial moat growing run on this instance, each vertex in all columns except the last one is active for a duration of 1, and each of these columns will form a connected component of the solution with cost $2(n - 1)$, for a total cost of $2(n - 1)(m - 1)$. On the other hand, for small enough values of ξ and large enough n and m , the optimal solution chooses the edges in each row plus the edges of one column, achieving a total cost of $2(n - 1) + (1 + \xi)n(m - 1)$. Therefore, this solution is a

$$\frac{2(n - 1)(m - 1)}{2(n - 1) + (1 + \xi)n(m - 1)} = 2 - \frac{4(n - 1) + 2\xi(n)(m - 1) + 2(m - 1)}{2(n - 1) + (1 + \xi)n(m - 1)}$$

approximation of the optimal, which given appropriate values of n , m , and ξ , this solution will not be a better than $2 - \xi'$ approximation.

Next, we show that no boost action will lead to an improvement in terms of solution cost. First, allowing any vertex not in the rightmost column to remain active longer cannot help achieve a lower solution, as the algorithm's behavior would not change before all demands are satisfied using the same forest, and additional growth afterward cannot decrease the cost of the solution. Additionally, for any vertex v on the rightmost column, allowing it to grow for a duration of less than ξ will not change the algorithm's behavior, as no vertex in the other columns will reach v before its demand is satisfied.

Now, if vertex v in the rightmost column is allowed to grow for at least ξ , all the vertices in the same row as v will reach v before being connected to their demands and then grow together. However, the final solution will not improve. To see why this is true, note that any other vertex in the rightmost column is separated from any active set by edges of at least $1 + \xi$ and, therefore, will not be connected to any vertex before this time. Since all demands are satisfied by time 1, adding any additional edges would not affect the final solution produced by the moat growing algorithm. Now, in the subgraph containing only one vertex of the last column and all the vertices in the other columns, the lowest cost forest that satisfies the demands is the forest that connects each column separately, which is the same as the solution found by the moat growing algorithm. Therefore, any boost action cannot improve the total cost of the solution.

Effectiveness of Our Local Search. In contrast, our local search algorithm focusing on minimizing y_{base} values can find the optimal solution given an appropriate value of β . To see why, consider a boost action of $\frac{1+\xi}{2}$ on any vertex in the rightmost column. Then, all vertices in the same row are connected by time $\frac{1+\xi}{4}$. Then, this combined set grows until moment 1, which is an additional duration of $\frac{3-\xi}{4}$. This means that the total growth of m for this row is reduced to

$$m \frac{1 + \xi}{4} + \frac{3 - \xi}{4} = \frac{(m + 3) + (m - 1)\xi}{4}.$$

This means we have a boost action with a win of

$$\frac{(3 - \xi)(m - 1)}{4}$$

while incurring a loss of

$$\frac{1 + \xi}{4}.$$

Now, for appropriate values of β , this will be a valuable boost action. Additionally, applying the boost action for each row does not detract from the value for the other rows, ensuring that the optimal solution can be found by the local search algorithm. Figure 11 also illustrates assignments before and after applying our local search, showcasing how the moves are beneficial.

A.2 A Lower Bound for the Local Search Algorithm

While our algorithm achieves a general approximation ratio better than 2 for the Steiner Tree problem, the example we have discovered demonstrates a lower bound of $3/2$, as described below. Note that in this example, there is no boost action with positive win. However, the condition for a boost action to be valuable is $W_{IN} \geq (1 + \beta) \cdot \text{Loss}$. This suggests that, for a fixed β , one could construct a tighter example that yields a higher lower bound.

Constructing the Instance. Consider a complete binary tree with a clear distinction between left and right children. This tree has 2^h leaves at the lowest layer. The edge weights decrease exponentially from bottom to top:

- The bottom-layer edges connecting leaves to their parents have weight 1.
- Each upper-layer edge weight is exactly half the weight of the edges directly below it (i.e., the immediate parents have edges of weight $1/2$, their parents $1/4$, and so forth).
- Additionally, every two consecutive leaves (from leftmost to rightmost) are directly connected with edges of weight $(2 - \xi)$.

- All the leaves are the set of terminal vertices.

Clearly, the given structure provides a Steiner Tree instance (See Figure 12a).

Running LEGACYMOATGROWING and LOCALSEARCH. The local search begins with the execution of the LEGACYMOATGROWING algorithm. At time $(2 - \xi)/2$, each leaf vertex's active set intersects with neighboring active sets, fully coloring the extra edges at the bottom layer (See Figure 12b). Hence, the fingerprint of the initial run sets the active times as follows:

$$t_v = \begin{cases} \frac{2-\xi}{2}, & \text{if } v \text{ is a leaf,} \\ 0, & \text{otherwise.} \end{cases}$$

This results in the initial forest with all additional edges of length $2 - \xi$ at the bottom of the input graph.

Analysis of Local Search. We now aim to show that our local search is unable to find a better solution.

Lemma 184. Local search on this instance, starting from the fingerprint produced by Legacy Moat Growing, returns the same initial forest and fingerprint.

Proof. We show that for any vertex v , boosting its value t_v to a certain threshold does not reduce the total growth $\sum y_S$. By Lemma 59, such a boost is not considered valuable. We then argue that neither increasing nor decreasing the value further yields a valuable boost, and thus no valid local search move exists.

Figure 12b illustrates this boost on vertex v .

Assume vertex v is at hop-distance d from the leaves in its subtree. Two observations are in order:

1. The distance from v to any leaf outside its subtree is at least 2.
2. The distance from v to any leaf within its subtree is

$$2 - \left(\frac{1}{2}\right)^{d-1}.$$

Now, suppose we boost v to:

$$t_v = \frac{2 - \left(\frac{1}{2}\right)^{d-1}}{2}.$$

At the moment of this boosted value, the active set containing v will merge with all active sets corresponding to the leaves of its subtree. And only in the next step, when time reaches $(2 - \xi)/2$ (assuming $\xi < (1/2)^{d-1}$), all active sets merge and moat growing ends. This implies only the subtree of v is affected by this boost (see Figure 12c).

Let's compare the total growth for this subtree before and after the boost (denoted as y and y').

Before the boost:

$$\sum y_S = 2^d \cdot \frac{(2 - \xi)}{2}.$$

After the boost:

$$\sum y'_S = \frac{2^d \cdot \left(2 - \left(\frac{1}{2}\right)^{d-1}\right)}{2} + \frac{2 - \xi}{2}.$$

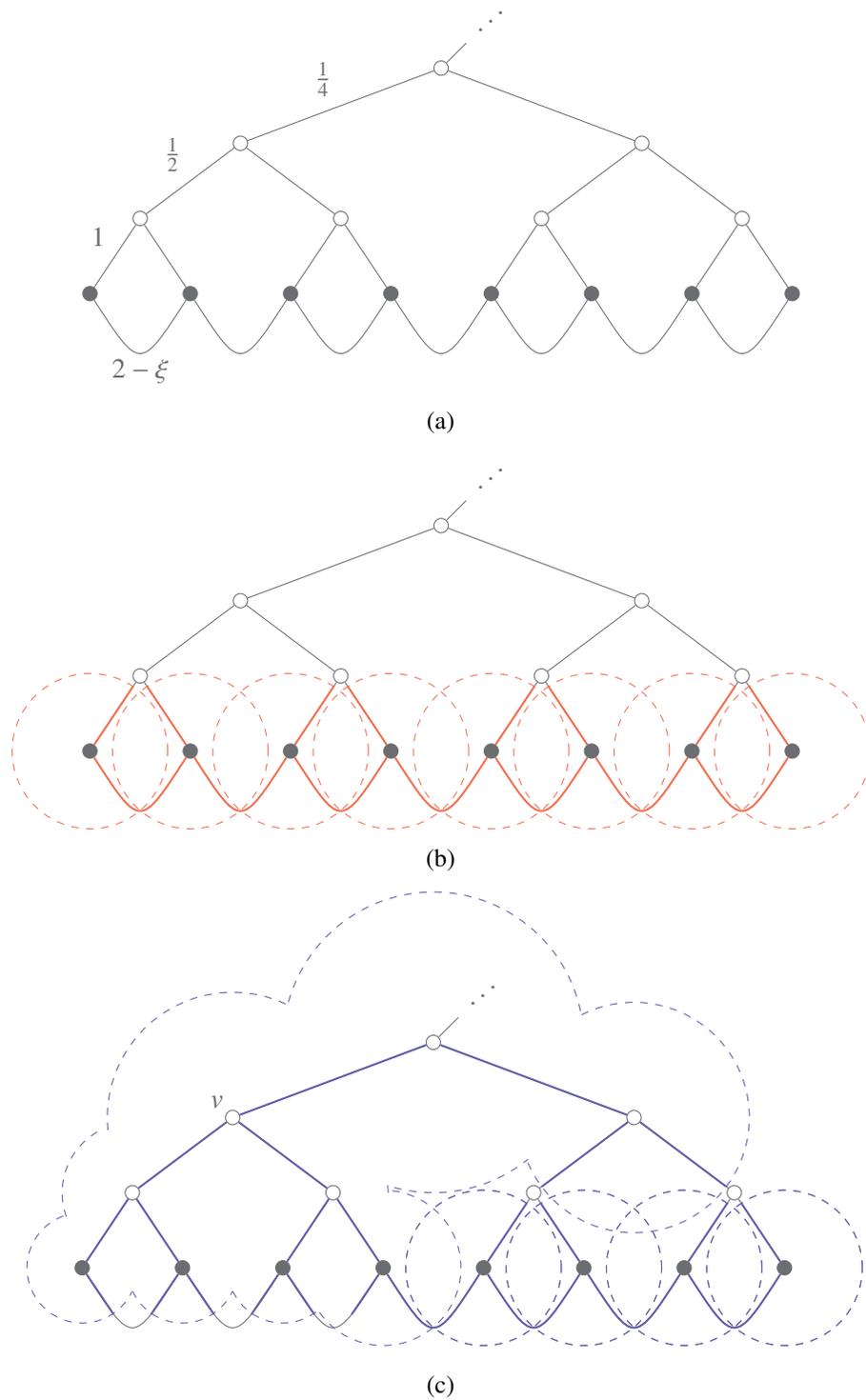


Figure 12: Illustration of the 3/2-approximation example. (A) The input graph: a complete binary tree with exponentially decreasing edge weights and additional horizontal edges of weight $2 - \xi$ between leaves. All leaves are terminals. (B) The initial forest constructed by LEGACYMOATGROWING, connecting leaves with horizontal $(2 - \xi)$ edges. (C) The effect of a boost action applied to an internal vertex v , which causes earlier merges within its subtree but does not decrease the total growth, and is thus not considered a valuable action.

Subtracting the pre-boost from the post-boost expression, and using the assumption $\xi < (1/2)^{d-1}$, one verifies:

$$2^d \cdot \frac{(2 - \xi)}{2} \leq \frac{2^d \cdot \left(2 - \left(\frac{1}{2}\right)^{d-1}\right)}{2} + \frac{2 - \xi}{2}.$$

Since the total growth does not decrease, this boost is not valuable. Any larger boost only increases the growth. A smaller boost may decrease the growth of v 's own active set, but increases it for all leaves in the subtree—which cancels out the gain.

Since the total growth does not decrease, this boost is not valuable. Any larger boost (i.e., increasing t_v beyond the value we just analyzed) only increases the total growth. On the other hand, any smaller boost may reduce the growth of v 's own active set, but increases the growth for all leaves in the subtree—compared to the analyzed boost value—thereby canceling out any potential decrease in total growth.

Thus, no boost action reduces the total cost. Local search halts without any change to the forest or fingerprint. \square

Optimal Solution Analysis. The optimal solution clearly corresponds to selecting only the original binary tree edges, excluding any additional leaf-to-leaf edges. Its total cost, denoted $c(\text{OPT})$, is:

$$c(\text{OPT}) = 2^h \left(1 + \frac{1}{4} + \frac{1}{16} + \cdots + \frac{1}{2^{2(h-1)}}\right) \leq \frac{4}{3} \cdot 2^h$$

Approximation Factor Analysis. The cost of our algorithm's forest, denoted as $c(F)$, is:

$$c(F) = (2^h - 1) \cdot (2 - \xi)$$

Hence, the approximation ratio is:

$$\frac{c(F)}{c(\text{OPT})} \leq \frac{(2^h - 1)(2 - \xi)}{2^h \cdot \frac{4}{3}} \leq \frac{3(2^h - 1)}{2(2^h)}$$

As h grows large, this ratio approaches $3/2$:

$$\lim_{h \rightarrow \infty} \frac{c(F)}{c(\text{OPT})} = \frac{3}{2}$$

This completes our analysis and demonstrates that the given instance achieves a $3/2$ approximation ratio.

Thus, this example establishes a lower bound of $3/2$ for the approximation ratio of our algorithm. Determining whether this bound is tight remains an intriguing open question.

A.3 Necessity of Autarkic Pairs

In this section, we present an instance where the `LEGACYMOATGROWING` algorithm produces a solution with a 2-approximation, but neither local search with boost actions nor extension is able to improve it. This example highlights the limitations of these techniques in certain cases and illustrates the necessity of handling autarkic pairs separately using the `AUTARKICPAIRS` algorithm.

Constructing the Instance. Consider a vertical path with $n + 1$ edges of length 1, connecting a top vertex v to a bottom vertex u . For each intermediate vertex along this path, attach a new vertex via an edge of length 1. Then, create multiple demand pairs over the endpoints of these added edges by duplicating the vertices and connecting each duplicate back with a near-zero-cost edge (with sufficiently small length ξ). Finally, add one more demand pair between v and u , and place a direct edge of length 2 between them (see Figure 2a).

Running LEGACYMOATGROWING, LOCALSEARCH and EXTENDEDMOATGROWING. When LEGACYMOATGROWING is executed on this instance, it assigns the same timestamp of $\frac{1}{2}$ to all demand vertices (see Figure 2b), and all are deactivated simultaneously at $\tau = \frac{1}{2}$. No vertex remains active beyond this point. Consequently, any attempt to boost a vertex is ineffective—the algorithm’s mechanism for boosting cannot trigger a valid change, as any smaller boost is ignored once all vertices are halted. This uniform fingerprint represents an edge case in which the LOCALSEARCH fails to improve the solution. Similarly, EXTENDEDMOATGROWING offers no improvement either, as it only increases the growth of active sets already finalized. Any additional growth preserves the previously selected forest without impacting the outcome.

Optimal Solution Analysis. The optimal solution in this instance connects each demand pair via its direct edge. Thus, the cost is:

$$c(\text{OPT}) = 2 + \sum_{i=1}^n 1 = n + 2.$$

Approximation Factor Analysis. In contrast, the solution produced by the moat growing algorithm includes all the edges of the base vertical path (which sum to $n + 1$) plus the n additional edges connecting each v_i to u_i . Therefore, the total cost of the forest F is:

$$c(F) = (n + 1) + n = 2n + 1.$$

This yields an approximation ratio of:

$$\frac{2n + 1}{n + 2} \approx 2 \quad \text{for sufficiently large } n.$$

Key Observation and Role of the Autarkic Pairs Algorithm. The main observation from this example is that—even though the moat growing procedure yields a 2-approximation—this specific structure causes the boost action to be ineffective, specifically not choosing the direct edge between v and u . In this edge case, the intermediate demands keep active sets growing over many unnecessary edges of the path and force the (v, u) demand to connect through these edges.

This is precisely where the AUTARKICPAIRS algorithm becomes essential: it detects these intermediate demands as autarkic pairs and connects them directly (see Figure 2c). This leads to immediate deactivation of their corresponding active sets and, as a result, allows the active sets of v and u to grow toward each other and connect via the direct edge of length 2 (see Figure 2d).

A.4 A Lower Bound Worse than 2 for the Gluttonous Algorithm

Given the high significance of Steiner Forest problem, long effort has been put even for finding another greedy algorithm that gives a constant approximation. Gupta and Kumar [GK15] showed the gluttonous algorithm preserves an $O(1)$ -approximation. They also left this as an important open problem whether this algorithm is a 2-approximation.

In this section, we present a counterexample showing that the gluttonous algorithm is not a 2-approximation for the Steiner Forest problem; in fact, its approximation ratio is at least $\frac{8}{3} \approx 2.666$.

First, we provide an exact presentation of the gluttonous algorithm.

The Gluttonous Algorithm. The algorithm proceeds by iteratively selecting the minimum-cost edge that connects previously disconnected components of unsatisfied vertices. This process continues until all unsatisfied vertices are connected to their pairs and get satisfied. The gluttonous algorithm is notable for its simplicity and intuitive approach.

The gluttonous algorithm can be formally defined as follows:

1. Initialize the forest $c(F)$ as an empty set.
2. While there exists unsatisfied (disconnected) demands:
 - (a) Find the minimum distance pair $e = (u, v)$ such that u and v belong to different components in the current forest $c(F)$ and neither v is connected to PAIR_v nor u is connected to PAIR_u —both are unsatisfied.
 - (b) Add the edge e to the forest $c(F)$ with cost of current shortest-path (equivalent to adding the edges of the current shortest-path to the forest).
 - (c) Connect u and v with a zero edge in the context graph.

Here, the context graph models the evolving connectivity state among vertices by introducing zero-cost virtual connections between already-joined vertices.

Previous research has established that the gluttonous algorithm is a 96-approximation for the Steiner Forest problem.

Counter Example. Here, we present a counterexample that demonstrates the gluttonous algorithm is not a 2-approximation for the Steiner Forest problem. The counterexample we provide exhibits a ratio of $\frac{8}{3}$ (approximately 2.666).

Description of the Counterexample. Let us consider a graph, H , defined in the following manner. The graph encompasses a path that consists of vertices denoted as v_0, v_1, \dots, v_k . Additionally, each index, i , has an associated vertex, u_i , which connects directly to the corresponding v_i vertex. This configuration constructs a tree-like structure (Actually, H corresponds to each column in Figure 13).

The costs of the edges within this graph adhere to a specific pattern. The edge cost connecting vertices v_i and v_{i+1} is defined as 2^i . A similar cost pattern is observed for the edge linking vertices v_i and u_i , wherein the cost is also defined as 2^i .

Building on the previous step, consider creating n copies of the graph H , labeled H_1, H_2, \dots, H_n . In addition to these copies, add a new extra vertex called v . In each copy H_j , let $u_{i,j}$ and $v_{i,j}$ represent the counterparts of the original vertices u_i and v_i , respectively.

For every j , form a connection between vertex $v_{0,j}$ and the extra vertex, v . The resultant structure upon these connections can then be defined as tree T (See Figure 13).

The graph G is generated as a weighted metric graph, derived by computing the shortest paths within tree T . Subsequently, for each fixed j , P_j is defined as a collection of terminal groups, where the j -th terminal group

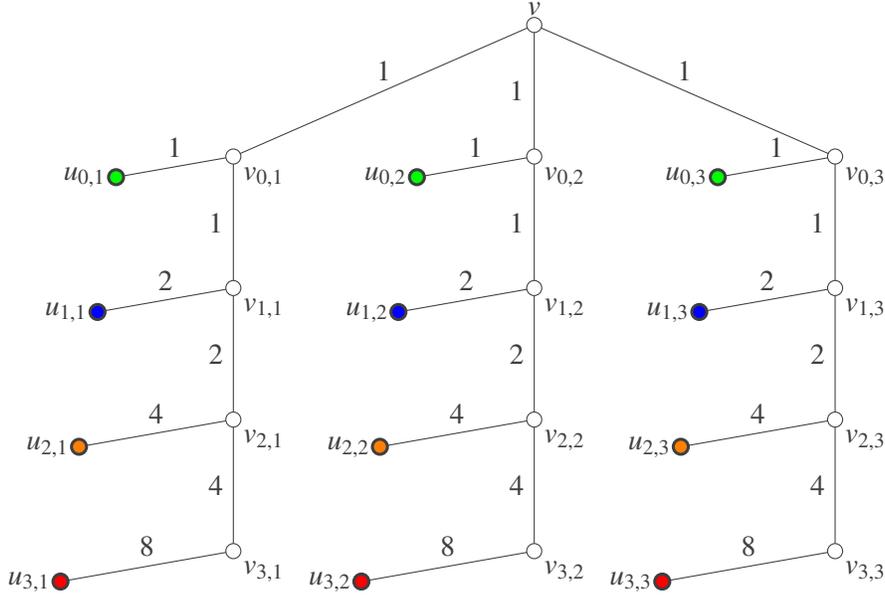


Figure 13: The figure shows a counterexample for which the Gluttonous algorithm yields an approximation ratio of at least $8/3$. Each column corresponds to one of the $n = 3$ replicas of a base tree structure with $k + 1 = 4$ layers and exponentially increasing edge weights. Each layer of leaves (i.e., vertices at the same depth across columns) forms a terminal group that must be pairwise connected. The Gluttonous algorithm connects each group of connectivity demands independently through directed edges between them, while the optimal solution reuses the underlying tree structure to connect all groups more efficiently. It is important to note that we can assume direct edges between demands are slightly smaller than the shortest path in the tree.

consists of the vertices $u_{i,j}$ for all possible values of i . Hence, there are $k + 1$ such terminal groups that are to be connected.

Here, a terminal group refers to a collection of vertices that must be pairwise connected. As discussed earlier, this setting can be transformed into our (v, PAIR_v) formulation by duplicating each vertex and adding zero-cost edges between the duplicates.

In this proposed counterexample, it will be demonstrated in the subsequent sections that the ratio of the output from the Gluttonous algorithm to the optimal solution is no less than $\frac{8}{3}$. This assertion underscores the performance characteristics of the algorithm within this specific context.

Gluttonous Algorithm Performance. The following characteristics of distances between vertices in the graph are observed:

- The distance from $u_{i,j}$ to v is 2^{i+1} .
- The distance between two vertices, u_{i,j_1} and u_{i,j_2} , is $2 \cdot 2^i = 2^{i+2}$.
- The distance between two vertices, $u_{i_1,j}$ and $u_{i_2,j}$ (with $i_1 < i_2$), is 2^{i+1} .
- Furthermore, the distance between two vertices, u_{i_1,j_1} and u_{i_2,j_2} (with $i_1 < i_2$, and $j_1 \neq j_2$), does not decrease when zero-edges are added between any pair of u_{i,j_1} and u_{i,j_2} (for $i < i_1$).

Based on the aforementioned observation, we notice that the gluttonous algorithm consecutively merges $u_{0,1}$ and $u_{0,2}$, then $u_{0,1}$ and $u_{0,3}$, and continues in this manner until it merges $u_{0,1}$ and $u_{0,n}$. In this progression, the

zeroth group, P_0 , becomes completely satisfied. Consequently, any later $u_{0,j}$ vertices are excluded from the requirements.

Subsequently, attention is turned to the remaining groups, namely P_1, P_2 , through to P_k . Throughout this process, each group necessitates $n - 1$ steps to reach full satisfaction. For each group P_i , each step incurs a cost of 2^{i+2} . Therefore, the total cost incurred is expressed as

$$(n - 1) \sum_{i=0}^k 2^{i+2} = (n - 1)(2^{k+3} - 4).$$

The Optimal Solution. It is apparent that tree T , as it stands, provides a feasible solution to the aforementioned requirements. Consequently, if we designate the optimal solution cost as $c(\text{OPT})$ and the total cost of T as $c(T)$, then it can be inferred that $c(\text{OPT}) \leq c(T)$.

More specifically, the upper bound of $c(\text{OPT})$ can be calculated as follows:

$$c(\text{OPT}) \leq n \cdot \left(1 + \sum_{i=0}^k 2^i + \sum_{i=0}^{k-1} 2^i \right) = n \cdot (2^k + 2^{k+1} - 1) = n \cdot (3 \cdot 2^k - 1).$$

Thus,

$$c(\text{OPT}) \leq n \cdot (3 \cdot 2^k - 1).$$

Approximation Factor Analysis. Let $c(F)$ denote the outcome cost of the Gluttonous algorithm. Hence,

$$c(F) = (n - 1)(2^{k+3} - 4) = (n - 1)(8 \cdot 2^k - 4).$$

Given that $c(\text{OPT}) \leq n \cdot (3 \cdot 2^k - 1)$, the approximation ratio can be expressed as:

$$\lim_{n,k \rightarrow \infty} \frac{c(F)}{c(\text{OPT})} = \lim_{n,k \rightarrow \infty} \frac{(n - 1)(8 \cdot 2^k - 4)}{n \cdot (3 \cdot 2^k - 1)} = \frac{8}{3}$$

Therefore, assuming sufficiently large values of n and k , it can be asserted that the approximation ratio is at least $\frac{8}{3}$.