Steiner Problems and Submodularity

Peyman Jabbarzade

Acknowledgements

Firstly, I would like to express my deepest gratitude to my advisor, MohammadTaghi Hajiaghayi, for his invaluable guidance and advice throughout our research projects. His support has been crucial to my development as a researcher.

I would also like to extend my thanks to all of my collaborators who contributed to my research: MohammadAli Abam, Ali Ahmadi, Antonios Antoniadis, MohammadReza Bahrami, Kiarash Banihashem, Leyla Biabani, Samira Goudarzi, Iman Gholami, MohammadTaghi Hajiaghayi, Mohammad Mahdavi, Morteza Monemizadeh, and Golnoosh Shahkarami.

A special thank you goes to MohammadAli Abam and Antonios Antoniadis, whose advice and encouragement helped me start my research journey and find my path.

Abstract

Studying the generalization of fundamental problems is fascinating because it involves understanding the potential of algorithms and how they can be adapted to handle more complex cases. In real-world applications, we rarely encounter clean and simple problems; instead, we must consider many factors. This is why generalizing problems and solving them with multiple decision points becomes crucial. I have focused on generalizing the minimum spanning tree problem and maximizing submodular functions, which model many real-world scenarios.

The minimum spanning tree problem is a foundational problem in graph theory, with numerous algorithms developed to find its optimal solution. While this problem can be solved in polynomial time, many generalizations, such as k-MST, Steiner tree, and Steiner forest, are NP-hard. Finding approximation algorithms for these problems is a significant research focus. In collaboration with Ali Ahmadi, Iman Gholami, MohammadTaghi Hajiaghayi, and Mohammad Mahdavi, we explored further generalizations of these problems, specifically their prize-collecting versions. Our work on the prize-collecting Steiner forest problem was published at SODA'24 (Ahmadi et al., 2024a), and our study on the prize-collecting Steiner tree problem was published at STOC'24 (Ahmadi et al., 2024b).

Additionally, I have explored submodular functions, which generalize a wide range of problems. Submodular functions have applications in theoretical computer science, such as modeling the max-cut problem, as well as in practical scenarios like video summarization, where we have tested our results. Specifically, in joint work with Kiarash Banihashem, Leyla Biabani, Samira Goudarzi, MohammadTaghi Hajiaghayi, and Morteza Monemizadeh, we studied maximizing submodular functions in dynamic models, where elements are added or removed from the ground set, requiring us to update our solution after each change. We achieved various results for both monotone and non-monotone submodular functions under different constraints. For example, our work on dynamic monotone submodular maximization under cardinality and matroid constraints was published at SODA'24 (Banihashem et al., 2024).

Contents

1 Introduction **2**

2 Prize-Collecting Steiner Forest **4**

2.1 Introduction 4

2.1.1 Algorithm and Techniques 6

2.1.2 Preliminaries 10

2.2 Representing a 3-approximation Algorithm 10

2.2.1 Dynamic Coloring 13

2.2.2 Analysis 20

2.3 The Iterative Algorithm 22

2.3.1 Analysis 23

2.3.2 Improving the approximation ratio 32

3 Submodular Maximization **34**

3.1 Introduction 34

3.1.1 Preliminaries 37

3.1.2 Our contribution and overview of techniques 39

3.1.3 Related Work 44

3.2 Dynamic algorithm for submodular matroid maximization 47

3.3 Analysis of dynamic algorithm for submodular matroid 49

3.3.1 Monotone property and binary search argument 51

3.3.2 Correctness of invariants after MatroidConstructLevel is called 53

3.3.3 Correctness of invariants after an update 54

3.3.4 Application of Uniform Invariant: Query complexity 61

3.3.5 Application of Level Invariants: Approximation guarantee 62

3.4 Parameterized dynamic algorithm for submodular maximization under cardinality constraint 66

3.5 Parameterized Lower Bound 67

Chapter 1

Introduction

I explore the generalization of fundamental problems in computer science, focusing on designing approximation algorithms for Steiner problems, which are generalizations of the Minimum Spanning Tree problem, and optimizing submodular functions in dynamic models. Studying these generalizations is fascinating because it involves understanding the potential of algorithms and how they can be adapted to handle more complex cases. In real-world applications, problems are rarely simple and clean, making it crucial to consider multiple factors and decision points. This is why generalizing problems and solving them effectively is so important.

Steiner Problems. In the Minimum Spanning Tree (MST) problem, the goal is to select a subset of edges that span all vertices while minimizing the total weight. While this problem can be solved in polynomial time, its generalizations—such as the Steiner tree, Steiner forest, and k-MST problems—are NP-hard. These problems still involve selecting a subset of edges with minimum total weight, but with additional objectives, like satisfying connectivity requirements between specific sets or numbers of vertices. Furthermore, the prize-collecting versions of these problems introduce flexibility by allowing penalties to be paid instead of satisfying certain demands, making the models more applicable to real-world scenarios where not all demands need to be met.

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

This natural iterative algorithm has proven effective in solving prize-collecting problems. We used it to improve the best-known approximation factor for the prize-collecting Steiner tree problem from 2.54 to 2, which we published at SODA'24. More details on this work can be found in Chapter 2. Additionally, we reduced the approximation factor for the prize-collecting Steiner tree from 1.96 to 1.799, and this work was published at STOC'24. Recently, we further enhanced our approach, generalizing it to show that the prize-collecting forest problem with a submodular penalty function has a 2-approximation algorithm. This result has been submitted to SODA'25.

Given that we have established a 2-approximation factor for a more generalized version of the Steiner forest problem, a natural question arises: can the approximation factor for the Steiner forest problem itself be further reduced?

Submodular Optimization. Submodular functions are powerful tools for solving real-world problems, as they model the "diminishing returns" phenomenon common in various practical settings. These functions are central to many theoretical problems, including graph cuts, entropy-based clustering, and coverage functions, and have been increasingly applied in machine learning tasks such as data summarization, feature selection, and recommendation systems.

We have studied the maximization of submodular functions in dynamic models, where elements may be added or removed, requiring continuous updates to the solution. Our work on monotone submodular maximization under cardinality and matroid constraints, published at SODA'24, is detailed in Chapter 3. Additionally, we have continued this research by improving update times and exploring non-monotone submodular functions and weighted submodular covers, with findings published in ICML'23, NeurIPS'23, and ICML'24.

While much of our work has focused on developing algorithms that operate efficiently in these dynamic models, future research should aim to improve the approximation factors for these problems.

Figure 1.1: Video summarization of Susan Boyle's performance on Britain's Got Talent. These frames were selected by our algorithm for dynamic non-monotone submodular maximization, published at NeurIPS'23 (Banihashem et al., 2023a).

Chapter 2

Prize-Collecting Steiner Forest

2.1 Introduction

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

Balas (Balas, 1989) first introduced general prize-collecting problems in 1989 and Bienstock, Goemans, Simchi-Levi, and Williamson (Bienstock et al., 1993) developed the first approximation algorithms for these problems. In the prize-collecting version of the Steiner forest problem, we are given an undirected graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$ and a set of pairs of vertices $\mathcal{D} = \{(v_1, u_1), (v_2, u_2), \cdots (v_k, u_k)\}$ called demands, along with non-negative penalties π_{ij} for each demand (i, j). The objective is to find a subset of edges and pay their costs, while also paying penalties for the demands that are not connected in the resulting forest. Specifically, we aim to find a subset of demands Q and a forest F such that if a demand (i, j) is not in Q, its endpoints i and j are connected in F, while minimizing the total penalty of the demands in Q and the sum of the costs of the edges in F. Without loss of generality, we assign a penalty of 0 to pairs that do not represent a demand, ensuring that there is a penalty associated with each pair of vertices. This allows us to define the penalty function $\pi : V \times V \to \mathbb{R}_{\geq 0}$, where $V \times V$ represents the set of all unordered pairs of vertices with $i \neq j$. In this paper, we significantly improve the approximation factor of the best-known algorithm for PCSF.

For the Steiner forest problem, the first approximation algorithm was introduced by Agrawal, Klein, and Ravi (Agrawal et al., 1995). Their algorithm addressed a more generalized version of the Steiner forest problem and achieved a 2-approximation for Steiner forest. Later, Goemans and Williamson (Goemans and Williamson, 1995) provided a simplified simulation of their algorithm, which yields a $(2 - \frac{2}{n})$-approximate solution for the Steiner forest problem, where n is the number of vertices[1]. However, no further advancements have been made in improving the approximation factor of this problem since then. There has been a study focused on analyzing a natural algorithm for the problem, resulting in a constant approximation factor worse

[1] Indeed Goemans and Williamson (Hochbaum, 1996)(Sec 4.6.1) explicitly mention "... the primal-dual algorithm we have presented simulates an algorithm of Agrawal, Klein, and Ravi [AKR95]. Their algorithm was the first approximation algorithm for this [Steiner forest a.k.a. generalized Steiner tree] problem and has motivated much of the authors' research in this area."; the seminal work of Agrawal, Klein, and Ravi (Agrawal et al., 1991, 1995) recently received *The 30-year STOC Test-of Time Award*.

than 2 (Gupta and Kumar, 2015). In this paper, we close the gap between the Steiner forest problem and its generalized version, PCSF, by presenting a 2-approximation algorithm for PCSF.

The Steiner tree problem is a well-studied special case of the Steiner forest problem. In the Steiner tree problem, one endpoint of every demand is a specific vertex known as *root*. In contrast to the Steiner forest problem, the approximation factor of the Steiner tree problem has seen significant progress since the introduction of the $(2 - \frac{2}{n})$-approximation algorithm by Goemans and Williamson (Goemans and Williamson, 1995). Several improvements have been made (Zelikovsky, 1993; Robins and Zelikovsky, 2005; Karpinski and Zelikovsky, 1997), leading to a 1.39 approximation factor achieved by Byrka, Grandoni, Rothvoß, and Sanità (Byrka et al., 2010). Lower bounds have also been established, with (Karp, 1972) proving the NP-hardness of the Steiner tree problem and consequently the Steiner forest problem, and (Chlebík and Chlebíková, 2008; Bern and Plassmann, 1989) demonstrating that achieving an approximation factor within $96/95$ is NP-hard. These advancements, along with the established lower bounds, underscore the extensive research conducted in the field of Steiner tree and Steiner forest problems.

Regarding the previous works in the prize-collecting version of these problems, Goemans and Williamson (Goemans and Williamson, 1995) provided a $(2 - \frac{1}{n-1})$-approximation algorithm for prize-collecting Steiner tree (PCST) and prize-collecting TSP problem (PCTSP) in addition to their work on the Steiner forest problem. However, they did not provide an algorithm specifically for the PCSF problem, leaving it as an open problem. Later, Hajiaghayi and Jain (Hajiaghayi and Jain, 2006) in 2006 proposed a deterministic primal-dual $(3 - \frac{2}{n})$-approximation algorithm for the PCSF problem, which inspired our work. They also presented a randomized LP-rounding 2.54-approximation algorithm for the problem. In their paper, they mentioned that finding a better approximation factor, ideally 2, remained an open problem. However, no improvements have been made to their result thus far. Furthermore, other 3-approximation algorithms have been proposed using cost-sharing (Gupta et al., 2007) or iterative rounding (Hajiaghayi and Nasri, 2010) (see e.g. (Bateni and Hajiaghayi, 2012; Hajiaghayi et al., 2012; Sharma et al., 2007) for further work on PCSF and its generalizations). Our paper is the first work that improves the approximation factor of (Hajiaghayi and Jain, 2006).

Moreover, advancements have been made in the PCST problem since the initial $(2 - \frac{1}{n-1})$-approximation algorithm by Goemans and Williamson (Goemans and Williamson, 1995). Archer, Bateni, Hajiaghayi, and Karloff (Archer et al., 2011) presented a 1.9672-approximation algorithm for PCST, surpassing the barrier of a 2-approximation factor. Additionally, there have been significant advancements in the prize-collecting TSP, which shares similarities with the LP formulation of PCST. Various works have been done in this area (Archer et al., 2011; Goemans, 2009; Blauth and Nägele, 2023), and the currently best-known approximation factor is 1.599 (Blauth et al., 2023). These works demonstrate the importance and interest surrounding prize-collecting problems, emphasizing their significance in the research community.

For a while, the best-known lower bound for the integrality gap of the natural LP relaxation for PCSF was 2. However, Könemann, Olver, Pashkovich, Ravi, Swamy, and Vygen (Könemann et al., 2017) proved that the integrality gap of this LP is at least $9/4$. This result suggests that it is not possible to achieve a 2-approximation algorithm for PCSF solely through primal-dual approaches based on the natural LP, similar to the approaches presented in (Hajiaghayi and Jain, 2006; Hajiaghayi and Nasri, 2010). This raises doubts about the possibility of achieving an algorithm with an approximation factor better than $9/4$.

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

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

We address the 9/4 integrality gap by analyzing a natural iterative algorithm. In contrast to previous approaches in the Steiner forest and PCSF fields that compare solutions with feasible dual LP solutions, we compare our solution directly with the optimal solution and assess how much the optimal solution surpasses the dual. It is worth noting that our paper does not rely on the primal and dual LP formulations of the Steiner forest problem. Instead, we employ a coloring schema that shares similarities with primal-dual approaches. While LP techniques could be applied to various parts of our paper, we believe that solely relying on LP would not be sufficient, particularly when it comes to overcoming the integrality gap. Furthermore, although coloring has been used in solving Steiner problems (Bateni et al., 2011), our approach goes further by incorporating two interdependent colorings, making it novel and more advanced.

In addition, we analyze a general approach that can be applied to various prize-collecting problems. In any prize-collecting problem, an algorithm needs to make decisions regarding which demands to pay penalties for and which demands to satisfy. Let us assume that for a prize-collecting problem, we have a base algorithm A. We propose a natural iterative algorithm that begins by running A on an initial instance and storing its solution as one of the options for the final solution. The solution generated by algorithm A pays penalties for some demands and satisfies others. Subsequently, we assume that all subsequent solutions generated by our algorithm will pay penalties for the demands that A paid, set the penalties of these demands to zero, and run A again on the modified instance. We repeat this procedure recursively until we reach a state where algorithm A satisfies every demand with a non-zero penalty, meaning that further iterations will yield the same solution. This state is guaranteed to be reached since the number of non-zero demands decreases at each step. Finally, we obtain multiple solutions for the initial instance and select the one with the minimum cost. This natural iterative algorithm could be effective in solving prize-collecting problems, and in this paper, we analyze its application to the PCSF problem using a variation of the algorithm proposed in (Hajiaghayi and Jain, 2006) as our base algorithm.

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

2.1.1 Algorithm and Techniques

In this paper, we introduce a coloring schema that is useful in designing algorithms for Steiner forest, PCSF, and related problems. This coloring schema provides a different perspective from the algorithms proposed by Goemans and Williamson in (Goemans and Williamson, 1995) for Steiner forest and Hajiaghayi and Jain in (Hajiaghayi and Jain, 2006) for PCSF. In Section 2.2, we provide a detailed representation of the algorithm proposed in (Hajiaghayi and Jain, 2006) using our coloring schema. The use of coloring enhances the intuitiveness of the algorithm, compared to the primal-dual approach utilized in (Hajiaghayi and Jain, 2006), and enables the analysis of our 2-approximation algorithm. Additionally, we introduce a modification to the algorithm of (Hajiaghayi and Jain, 2006), which is essential for the analysis of our 2-approximation algorithm. Finally, in Section 2.3, we present an iterative algorithm and prove its 2-approximation guarantee for PCSF.

Here, we provide a brief explanation of how coloring intuitively solves the Steiner forest problem. We then

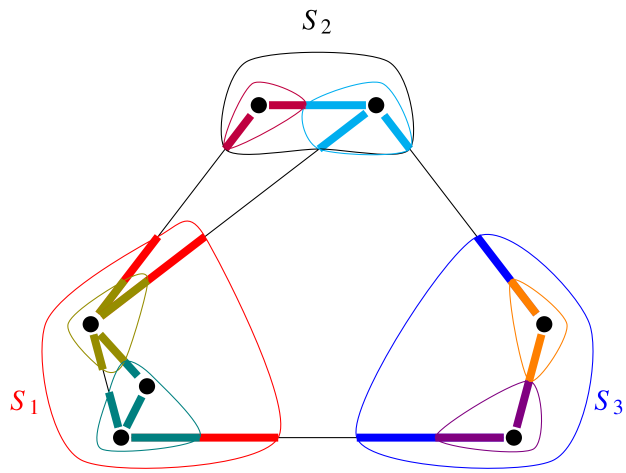

Figure 2.1: Illustration of the *static coloring* and the coloring used for Steiner forest problem. In the graph, S_2 is inactive and does not color its cutting edges, while S_1 colors edges in red and S_3 colors in blue. It is worth noting that edges within a connected component will not be further colored and will not be added to F.

present a 3-approximation algorithm and subsequently a 2-approximation algorithm for PCSF.

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

In each iteration of our algorithm, every active set $S \in ActS$ is assigned a distinct color, which is used to simultaneously color its cutting edges at the same speed. The cutting edges of a set S are defined as the edges that have exactly one endpoint within S. Our coloring procedure proceeds by coloring the remaining uncolored sections of these edges. An edge is in the process of getting colored at a given moment if it connects different connected components of F and has at least one endpoint corresponding to an active set. Additionally, if an edge is a cutting edge for two active sets, it is colored at a speed twice as fast as an edge that is a cutting edge for only one active set. We continue this coloring process continuously until an edge e is fully colored, and then we add it to the forest F. Afterwards, we update FC and $ActS$ accordingly, as defined earlier. It is important to note that we only add edges to F that connect different connected components, ensuring that F remains a forest. Furthermore, since the set of all connected components of F forms a laminar set over time, our coloring schema is also laminar. Refer to Figure 2.1 for clarity on the coloring process.

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

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

Furthermore, considering the fixed final forest F', we can observe that at each moment of coloring, when we contract each connected component in FC, it results in a forest where every leaf corresponds to an active set. This observation is based on the fact that if a leaf does not correspond to an active set, it implies that the only edge adjacent to that leaf is unnecessary and should have been removed from F'. Based on this insight, we can conclude that the number of edges being colored from F' at that moment, which is equivalent to the sum of the degrees of the active sets in the aforementioned forest, is at most twice the number of active sets at that moment. This means that the amount of the newly colored portion of all edges at that moment is at most twice the total value added to all y_S. Therefore, considering that every edge in F' is fully colored, we can deduce that the total length of edges in F' is at most $2 \sum_{S \subset V} y_S$.

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

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

Similar to the Steiner forest algorithm, we begin by running the *static coloring* procedure. Whenever an edge is fully colored, we add it to the forest F. However, unlike *static coloring*, we do not maintain a separate *dynamic coloring* throughout the algorithm, as it would require constant reconstructions. Instead, we compute the *dynamic coloring* whenever needed. To obtain the *dynamic coloring*, we map each moment of coloring for each set S in the *static coloring* to a pair (i, j) such that $S \odot (i, j)$, which means S cuts (i, j). This assignment is achieved using a maximum flow algorithm, as described in Section 2.2.1. We ensure that our *static coloring* can always be converted to a *dynamic coloring*. Let y_{ij} represent the total duration assigned to pair (i, j) in the *dynamic coloring*. It is important to ensure that y_{ij} does not exceed the potential π_{ij} associated with that pair. If we encounter an active set S for which assigning further coloring to any pair that S cuts would exceed the pair's potential, we deactivate S by removing it from $ActS$.

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

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

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

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

We now consider the values of cp_1 and cp_2 to analyze the algorithm. If cp_2 is sufficiently large, we can establish a stronger lower bound for the optimal solution compared to our previous bound, which was $\sum_{S\subset V} y_S$. In the previous bound, we showed that each moment of coloring covers at least one edge of the optimal solution. However, in this case, we can demonstrate that a significant portion of the coloring process covers at least two edges at each moment. This improved lower bound allows us to conclude that the output of PCSF3, (Q_1, F'_1), becomes a 2-approximate solution.

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

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

2.1.2 Preliminaries

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

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

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

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

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

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

Since we use max-flow algorithm in Section 2.2.1, we provide a formal definition of the MaxFlow function:
Definition 2.1.1 (MaxFlow). *For the given directed graph G with source vertex source and sink vertex sink, the function* $\textsc{MaxFlow}(G, source, sink)$ *calculates the maximum flow from source to sink and returns three values:* (f^*, C_{min}, f). *Here,* f^* *represents the maximum flow value achieved from source to sink in G,* C_{min} *represents the min-cut between source and sink in G that minimizes the number of vertices on the source side of the cut, and* f *is a function* $f : E \to \mathbb{R}^+$ *that assigns a non-negative flow value to each edge in the maximum flow.*

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

2.2 Representing a 3-approximation Algorithm

In this section, we present an algorithm that utilizes coloring to obtain a 3-approximate solution. Although the main part of this algorithm closely follows the approach presented by Hajiaghayi and Jain in (Hajiaghayi and Jain, 2006), our novel interpretation of the algorithm is crucial for the subsequent analysis of the 2-approximation algorithm in the next section. Furthermore, we introduce a modification at the end of the 3-approximation algorithm, which plays a vital role in achieving a 2-approximation algorithm in the next section.

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

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

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

During the algorithm's execution, active sets color their cutting edges simultaneously and at the same speed, using their respective unique colors. As a result, only edges between different connected components are colored at any given moment. When an edge e is fully colored, we add it to F and update FC to maintain the connected components of F. Since e connects two distinct connected components of F, F remains a forest. Furthermore, we update $ActS$ by removing sets that contain an endpoint of e and replacing them with their union. Within the loop at Line 11 of Algorithm 2, we check if an edge has been completely colored, and then merge the sets that contain its endpoints. In addition, we provided a visual representation of the *static coloring* process in Figure 2.1.

Definition 2.2.1 (Static coloring duration)**.** *For an instance of static coloring, define y_S as the duration during which set S colors its cutting edges using the color S.*

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

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

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

Corollary 2. *After coloring for Δ_e duration, at least one new edge becomes fully colored.*

In Algorithm 1, we outline the procedure for FindDeltaE.

Now, we can utilize FindDeltaE to perform the coloring in discrete steps, as shown in Algorithm 2. In summary, during each step, at Line 6, we call FindDeltaE to determine the maximum duration Δ_e for which we can color with active sets without exceeding the length of any edge, ensuring that at least one edge will be fully colored. Similarly, at Line 7, we utilize FindDeltaP to determine the maximum value of Δ_p that ensures a valid *static coloring* when extending the coloring duration by Δ_p using active sets. The concept of a valid *static coloring*, which avoids purchasing edges when it is more efficient to pay penalties, will be further explained in Section 2.2.1.

Algorithm 1 Fidning the maximum value for Δ_e

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

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

procedure FindDeltaE(G, y, $ActS$, FC)
    Initialize $\Delta_e \leftarrow \infty$
    for $e \in E$ **do**
        Let $S_v, S_u \in FC$ be the sets that contain each endpoint of e.
        $t \leftarrow |\{S_v, S_u\} \cap ActS|$
        if $S_v \neq S_u$ **and** $t \neq 0$ **then**
            $\Delta_e \leftarrow \min(\Delta_e, (c_e - \sum_{S : e \in \delta(S)} y_S)/t)$
    return Δ_e

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

In Algorithm 2, we utilize three functions other than FindDeltaE: FindDeltaP, CheckSetIsTight, and ReduceTightPairs. The purpose of FindDeltaP is to determine the maximum value of Δ_p that allows for an additional coloring duration of Δ_p resulting in a *valid static coloring*. CheckSetIsTight is responsible for identifying sets that cannot color their cutting edges while maintaining the validity of the static coloring. Lastly, ReduceTightPairs aims to reduce the number of pairs for which penalties need to be paid and determine the final set of pairs that we pay their penalty. All of these functions utilize *dynamic coloring*, which will be explained in Section 2.2.1.

It is important to note that we do not store a *dynamic coloring* within PCSF3 since it changes constantly. Instead, we compute a *dynamic coloring* based on the current *static coloring* within these functions, as they are the only parts of our algorithm that require a *dynamic coloring*. Note that at the end of PCSF3, we require a final *dynamic coloring* for the analysis in Section 2.3. This final coloring will be computed in ReduceTightPairs at Line 20.

Now, let's analyze the time complexity of FindDeltaE as described in Lemma 4. In Lemma 22, we will demonstrate that the overall time complexity of PCSF3 is polynomial. This will be achieved after explaining and analyzing the complexity of the subroutines it invokes.

Lemma 3. *In the* PCSF3 *algorithm, the number of sets that have been active at some point during its execution is linear.*

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

Lemma 4. *The runtime of* FindDeltaE *is polynomial.*

Algorithm 2 A 3-approximation Algorithm

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

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

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

2.2.1 Dynamic Coloring

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

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

Definition 2.2.2 (Dynamic Coloring Assignment Duration)**.** *In a dynamic coloring instance, for each set S and pair (i, j) where $S \odot (i, j)$, y_{Sij} represents the duration of coloring with color S in static coloring that is assigned to pair (i, j) for coloring in dynamic coloring.*

Definition 2.2.3 (Dynamic Coloring Duration)**.** *In a dynamic coloring instance, y_{ij} represents the total duration of coloring with pair (i, j) in dynamic coloring, denoted as $y_{ij} = \sum_{S : S \odot (i,j)} y_{Sij}$.*

Definition 2.2.4 (Pair Constraint and Tightness)**.** *In a dynamic coloring instance, the condition that each pair (i, j) should not color for more than π_{ij} total duration (i.e., $y_{ij} \leq \pi_{ij}$) is referred to as the pair constraint. If this condition is tight in the dynamic coloring for a pair (i, j), i.e., $y_{ij} = \pi_{ij}$, we say that pair (i, j) is a tight pair.*

Definition 2.2.5 (Valid Static Coloring)**.** *A static coloring is considered valid if there exists a dynamic coloring for the given static coloring. In other words, the following conditions must hold:*

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

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

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

Definition 2.2.6 (Set Tightness)**.** *For a valid instance of static coloring, we define a set $S \subset V$ as tight if increasing the value of y_S by any $\epsilon > 0$ in the static coloring without changing the coloring duration of other sets would make the static coloring invalid.*

Lemma 5. *In a valid static coloring, if set S is tight, then for any corresponding dynamic coloring, all pairs (i, j) such that $S \odot (i, j)$ are tight.*

Proof. Consider an arbitrary *dynamic coloring* of the given *valid static coloring*. Using contradiction, assume there is a pair (i, j) such that $S \odot (i, j)$, and this pair is not tight. Let $\epsilon = \pi_{ij} - y_{ij}$. If we increase y_S, $y_{S\,ij}$, and y_{ij} by ϵ, it results in a new *static coloring* and a new *dynamic coloring*. In the new *dynamic coloring*, since for every set S' we have $\sum_{(i',j'):S'\odot(i',j')} y_{S'i'j'} = y_{S'}$, and for every pair (i', j') we have $y_{i'j'} = \sum_{S':S'\odot(i',j')} y_{S'i'j'} \leq \pi_{i'j'}$, based on Definition 2.2.5, increasing y_S by ϵ results in a valid *static coloring*. According to Definition 2.2.6, this contradicts the tightness of S. Therefore, we can conclude that all pairs (i, j) for which $S \odot (i, j)$ holds are tight. T

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

So far, we have explained several key properties and concepts related to *dynamic coloring*. Now, let us explore how we can obtain a *dynamic coloring* from a given *valid static coloring*. To accomplish this, we introduce the concept of SetPairGraph, which represents a graph associated with each *static coloring*. By applying the max-flow algorithm to this graph, we can determine a corresponding *dynamic coloring*. The definition of SetPairGraph is provided in Definition 2.2.7, and Figure 2.2 illustrates this graph.

Definition 2.2.7 (SetPairGraph)**.** *Given a graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$, and penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, as well as an instance of static coloring represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, we define a directed graph $\mathcal{G}$ initially consisting of two vertices source and sink. For every set $S \subset V$ such that either $y_S > 0$ or $S \in ActS$, we add a vertex to $\mathcal{G}$ and a directed edge from source to S with a capacity of y_S. Additionally, for each pair $(i, j) \in V \times V$, we add a vertex to $\mathcal{G}$ and a directed edge from (i, j) to sink with a capacity of π_{ij}.*

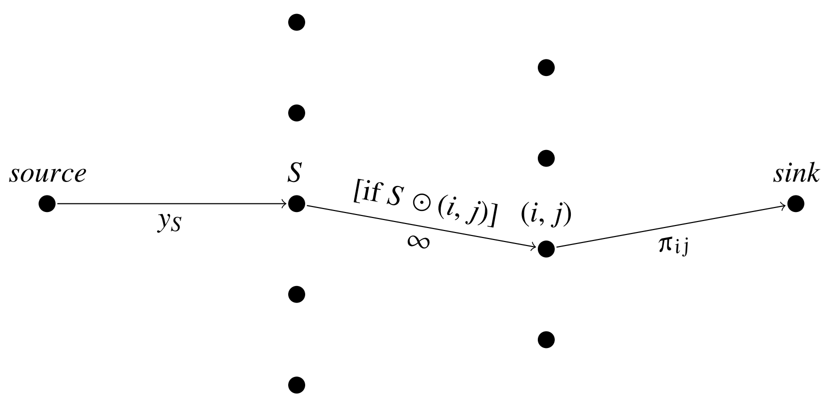

Figure 2.2: SetPairGraph

Finally, we add a directed edge from each set S to each pair (i, j) such that $S \odot (i, j)$, with infinite capacity. We refer to the graph $\mathcal{G}$ as SetPairGraph(G, π, y).

Now, we compute the maximum flow from *source* to *sink* in SetPairGraph(G, π, y). We assign the value of y_{Sij} to the amount of flow from S to (i, j), representing the allocation of coloring from set S in *static coloring* to pair (i, j) in *dynamic coloring*. Similarly, we set y_{ij} equal to the amount of flow from (i, j) to *sink*, indicating the duration of coloring for pair (i, j) in *dynamic coloring*. In Lemma 6, we show that if the maximum flow equals $\sum_{S\subset V} y_S$, the *static coloring* is valid, and the assignment of y_{Sij} and y_{ij} satisfies all requirements.

Lemma 6. *For a given graph $G = (V, E, c)$, penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, and an instance of static coloring represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, let $(f^*, C_{min}, f) =$* MaxFlow(SetPairGraph(G, π, y), *source*, *sink*)*, the static coloring is valid if and only if $f^* = \sum_{S\subset V} y_S$.*

Proof. Assume that $f^* = \sum_{S\subset V} y_S$. Since the sum of the capacities of the outgoing edges from *source* is equal to the maximum flow, the amount of flow passing through S is y_S. Hence, given that the total flow coming out from S is equal to $\sum_{(i,j)\in V\times V} y_{Sij}$, we have $y_S = \sum_{(i,j)\in V\times V} y_{Sij}$. It should be noted that $y_{Sij} > 0$ only if $S \odot (i, j)$, as we only have a directed edge from S to (i, j) in that case.

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

Now, assume that the given *static coloring* is valid. Consider its corresponding SetPairGraph and *dynamic coloring*. Let the amount of flow from *source* to S be y_S. Let the amount of flow in the edge between S and (i, j) be y_{Sij}, representing the assignment duration in the *dynamic coloring*. Similarly, let the amount of flow in the edge between (i, j) and *sink* be y_{ij}, representing the duration in the *dynamic coloring*. According to Definition 2.2.5, in a *valid static coloring*, the following conditions hold: $\sum_{(i,j):S\odot(i,j)} y_{Sij} = y_S$ and $y_{ij} = \sum_{S: S\odot(i,j)} y_{Sij} \leq \pi_{ij}$. Therefore, in the SetPairGraph, the assignment of flow satisfies the edge capacities and the equality of incoming and outgoing flows for every vertex except *source* and *sink*. Furthermore, since we fulfill all outgoing edges from *source*, the maximum flow is $\sum_{S\subset V} y_S$. T

Let us define C_{source} as the cut in SetPairGraph that separates *source* from other vertices. Since the sum of the edges in C_{source} is $\sum_{S\subset V} y_S$, the following corollary can easily be concluded from Lemma 6.

Corollary 7. *For a given graph $G = (V, E, c)$, penalties $\pi : V \times V \to \mathbb{R}_{\geq 0}$, and an instance of static coloring represented by $y : 2^V \to \mathbb{R}_{\geq 0}$, the static coloring is valid if and only if C_{source} is a minimum cut between source and sink in* SetPairGraph(G, π, y).

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

Lemma 8. *At any point during the algorithm, the size of the* SetPairGraph *remains polynomial.*

Proof. In the SetPairGraph, vertices are assigned to sets that are either active or have $y_S > 0$. This implies that for each set that is active at least once, there is at most one corresponding vertex in the graph. According to Lemma 3, the number of such vertices is linear.

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

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

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

Lemma 9. *In a valid static coloring, the maximum possible duration to continue the coloring process while ensuring the validity of the static coloring is* $\Delta_p = \text{FindDeltaP}(G, \pi, y, ActS)$.

Proof. Consider the SetPairGraph of the given *valid static coloring*. If increasing the capacity of edges from *source* to S for every set $S \in ActS$ by Δ_p results in an increase in the min-cut by $|ActS| \cdot \Delta_p$, then the resulting *static coloring* remains valid since C_{source} remains a min-cut. This is based on Corollary 7. Thus, our goal is to find the maximum value for Δ_p that satisfies this condition.

First, in Line 3, we initialize Δ_p with an upper-bound value of $(\sum_{ij} \pi_{ij} - \sum_S y_S)/|ActS|$. This value serves as an upper-bound because the increase in min-cut cannot exceed $(\sum_{ij} \pi_{ij} - \sum_S y_S)$, as determined by the cut that separates *sink* from the other vertices. Next, we update the capacity of edges from *source* to the active sets in SetPairGraph by the value of Δ_p, and then calculate the minimal min-cut C_{min}. If C_{min} separates *source* from the other vertices, it indicates that the new *static coloring* is valid according to Corollary 7. At this point, the function terminates and returns the current value of Δ_p in Line 8.

Otherwise, if C_{source} is not a min-cut after updating the edges, we want to prove that C_{min} has at least one active set in the side of *source*. Let's assume, for contradiction, that C_{min} does not have any active sets on the side of *source*. According to Corollary 7, prior to updating the edges, C_{source} was a min-cut since we had a *valid static coloring*. Thus, the weight of C_{min} was at least equivalent to the weight of C_{source} before the edges were modified. Given that C_{min} and C_{source} include all the edges connecting *source* to the active sets, adding Δ_p to active sets leads to an increase in the weight of both C_{min} and C_{source} by $|ActS| \cdot \Delta_p$. Consequently, the weight of C_{min} remains at least as large as the weight of C_{source}. However, if C_{source} is not a minimum cut after updating the edges, it implies that C_{min} cannot be a minimum cut either, which contradicts the definition of C_{min}. Therefore, we can conclude that C_{min} must have at least one active set on the same side as *source*.

Let $k \geq 1$ represent the number of active sets on the same side as *source* in C_{min}. Referring to Definition 2.1.1, it is evident that C_{min} minimizes k. If we decrease Δ_p by ϵ, it deducts $|ActS| \cdot \epsilon$ from the weight of C_{source} and $(|ActS| - k) \cdot \epsilon$ from weight of C_{min}. Given that the weight of C_{source} is $|ActS| \cdot \Delta_p + \sum_{S \subset V} y_S$, and the weight of C_{min} is f^*, in order to establish C_{source} as a minimum cut, we want to find the minimum value of ϵ satisfying:

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

Now, let us define $\epsilon^* = (|ActS| \cdot \Delta_p + \sum_{S \subset V} y_S - f^*)/k$. When we decrease Δ_p by ϵ^*, we effectively tighten the upper bound on Δ_p. This is crucial because reducing Δ_p by a smaller value would make it impossible for C_{source} to become a min-cut. Such a violation would contradict the validity of the *static coloring*, as indicated by Corollary 7. After updating Δ_p to its new value, if C_{source} indeed becomes a minimum cut, the procedure is finished. However, if the new minimal min-cut still contains active sets on the side of *source*, their number must be less than k.

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

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

Lemma 10. *In a valid static coloring, the static coloring remains valid if we continue the coloring process by at most* $\Delta_p = \textsc{FindDeltaP}(G, \pi, y, ActS)$.

Proof. Consider the SetPairGraph of the given *valid static coloring*. We want to show that for every value $\Delta'_p \leq \Delta_p$, the coloring remains valid. According to Lemma 9, we know that increasing the duration of the active sets by Δ_p results in a *valid static coloring*. Therefore, by increasing the capacity of edges from *source* to S for each set $S \in ActS$ by Δ_p, C_{source} represents a minimum cut. Decreasing the capacities of these edges by a non-negative value $d = \Delta_p - \Delta'_p$ results in a decrease in the weight of C_{source} by $|ActS| \cdot d$, while other cuts are decreased by at most this value. Consequently, C_{source} remains a minimum cut, and as indicated in Corollary 7, the static coloring remains valid after increasing y_S for active sets by $\Delta'_p \leq \Delta_p$. T

Based on the proof of Lemma 9, it is clear that the while loop in the FindDeltaP function iterates a linear number of times. Furthermore, during each iteration, we make a single call to the MaxFlow procedure on the SetPairGraph, which has a polynomial size according to Lemma 8. Consequently, we can deduce the following corollary.

Corollary 11. *Throughout the algorithm, each call to the* FindDeltaP *function executes in polynomial time.*

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

Lemma 12. *Throughout the algorithm, we always maintain a valid static coloring.*

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

Assuming that at the beginning of each iteration, we have a *valid static coloring* based on the induction hypothesis. Furthermore, during each iteration, we continue the coloring process for a duration of $\min(\Delta_p, \Delta_e) \leq \Delta_p$. According to Lemma 10, this coloring preserves the validity of the *static coloring*.

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

Algorithm 3 Finding the maximum value for Δ_p

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

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

function FindDeltaP$(G, \pi, y, ActS)$
  $\mathcal{G} \leftarrow$ SetPairGraph(G, π, y)
  $\Delta_p \leftarrow \left(\sum_{ij} \pi_{ij} - \sum_S y_S\right) / |ActS|$
  while true **do**
    Set the capacity of edges from *source* to $S \in ActS$ equals to $y_S + \Delta_p$ in graph $\mathcal{G}$.
    $(f^*, C_{min}, f) \leftarrow$ MaxFlow$(\mathcal{G}, source, sink)$
    if $f^* = |ActS| \cdot \Delta_p + \sum_{S \subset V} y_S$ **then**
      return Δ_p
    Let k represent the number of active sets on the same side of the cut C_{min} as *source*.
    $\Delta_p \leftarrow \Delta_p - \left(|ActS| \cdot \Delta_p + \sum_{S \subset V} y_S - f^*\right) / k$

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

Checking the tightness of a set is straightforward, as outlined in Definition 2.2.6. To determine if set S is tight, we increase the capacity of the directed edge from *source* to S and check if the flow from *source* to *sink* exceeds $\sum_{S \subset V} y_S$. The pseudocode for this function is provided in Algorithm 4.

Lemma 13. *Each call to the* CheckSetIsTight *function during the algorithm runs in polynomial time.*

Proof. The CheckSetIsTight function call MaxFlow once on the SetPairGraph, whose size remains polynomial throughout the algorithm according to Lemma 8. Therefore, both the MaxFlow and the CheckSetIsTight function run in polynomial time.
T

Reduce the number of tight pairs. At the end of PCSF3, we obtain a final *valid static coloring*, from which we can derive a corresponding final *dynamic coloring* which corresponds to a max-flow in SetPairGraph. Next, we present the process of reducing the number of tight pairs in the final *dynamic coloring*, aiming to

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

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

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

function CheckSetIsTight(G, π, y, S)
  $\mathcal{G} \leftarrow$ SetPairGraph(G, π, y)
  Set the capacity of the edge from *source* to S in graph $\mathcal{G}$ equals to $y_S + 1$.
  $(f^*, C_{min}, f) \leftarrow$ MaxFlow$(\mathcal{G}, source, sink)$
  if $f^* > \sum_{S \subset V} y_S$ **then**
    return False
  else
    return True

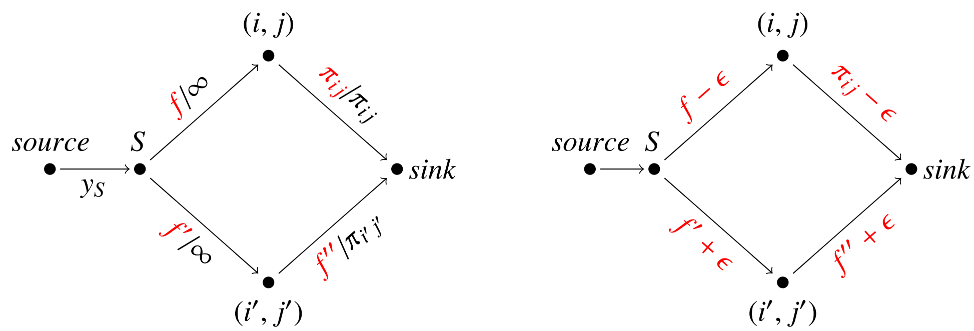

Figure 2.3: By choosing a small positive value $\epsilon < \min(f, \pi_{i'j'} - f'')$, we can remove one tight pair. The red variables represent the amounts of flow on each edge, while the black variables represent their capacity.

achieve a *minimal dynamic coloring*. This step is essential to obtain a 2-approximate solution for PCSF in the next section.

Definition 2.2.8 (Minimal Dynamic Coloring)**.** *A dynamic coloring is considered minimal dynamic coloring if there are no pairs $(i, j), (i', j') \in V \times V$ and set $S \subset V$ such that pair (i, j) is a tight pair while (i', j') is not a tight pair, $S \odot (i, j)$, $S \odot (i', j')$, and $y_{Sij} > 0$.*

To obtain a *minimal dynamic coloring*, we first check if there exist pairs $(i, j), (i', j')$ and a set S meeting the following criteria: pair (i, j) is tight, pair (i', j') is not tight, $S \odot (i, j)$, $S \odot (i', j')$, and $y_{Sij} > 0$. If such pairs and set exist, we proceed with the following adjustments. Since (i', j') is not tight, we have $\pi_{i'j'} - y_{i'j'} > 0$. Additionally, $y_{Sij} > 0$ is assumed. Therefore, there exists $\epsilon > 0$ such that $\epsilon < \min(y_{Sij}, \pi_{i'j'} - y_{i'j'})$. Given that $\epsilon < y_{Sij} \leq y_{ij}$, we can reduce y_{Sij} and y_{ij} by ϵ, while adding ϵ to $y_{Si'j'}$ and $y_{i'j'}$. Since $\epsilon > 0$, pair (i, j) is no longer tight, and since $\epsilon < \pi_{i'j'} - y_{i'j'}$ for the previous value of $y_{i'j'}$, pair (i', j') will not become tight. It is important to note that the *dynamic coloring* prior to these changes corresponds to a max-flow in SetPairGraph. Implementing these adjustments on the flow of edges associated with y_{Sij}, y_{ij}, $y_{Si'j'}$, and $y_{i'j'}$ results in a new max-flow that corresponds to the updated *dynamic coloring*. This provides an intuition for why the assignment in the new *dynamic coloring* remains valid. We illustrate these flow changes in Figure 2.3, and the complete process for achieving a minimal *dynamic coloring* is described in Algorithm 5.

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

Considering that the number of iterations in the ReduceTightPairs function is polynomial and the MaxFlow operation on the SetPairGraph in Line 3 has a polynomial size (Lemma 8), it can be concluded that the runtime of ReduceTightPairs is polynomial.

Corollary 14. *The* ReduceTightPairs *function runs in polynomial time.*

Algorithm 5 Reduce the number of tight pairs

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

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

Finally, after obtaining a *minimal dynamic coloring*, we consider it as our final *dynamic coloring*, which will be used in the analysis presented in Section 2.3.1.

Corollary 15. *The final dynamic coloring obtained at the end of procedure* PCSF3 *is a minimal dynamic coloring.*

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

Lemma 16. *In a minimal dynamic coloring, if a set $S \subset V$ cuts a tight pair $(i, j) \in V \times V$ with $y_{Sij} > 0$, then all pairs (i', j') satisfying $S \odot (i', j')$ are also tight.*

Proof. Assume there exists a pair (i', j') satisfying $S \odot (i', j')$ that is not tight. This implies that the pairs (i, j) and (i', j'), along with set S, contradict the definition of *minimal dynamic coloring* (Definition 2.2.8). T

2.2.2 Analysis

In this section, we demonstrate the validity of our algorithm's output for the given PCSF instance. We also present some lemmas that are useful for proving the approximation factor of PCSF3. However, we do not explicitly prove the approximation factor of PCSF3 in this section, as it is not crucial for our main result. Nonetheless, one can easily conclude the 3-approximation factor of PCSF3 using Lemmas 24, 26, and 23 provided in the next section. Additionally, in Lemma 22, we show that PCSF3 has a polynomial time complexity. The lemmas provided in this section are also necessary for the analysis of our 2-approximation algorithm, which is presented in the next section.

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

To prove this, we introduce some auxiliary lemmas. First, in Lemma 19, we demonstrate that when a set becomes tight during PCSF3, it remains tight until the end of the algorithm. This lemma is essential because if a set becomes tight and is subsequently removed from the active sets, but then becomes non-tight again, it implies that some pairs could contribute to the coloring in the *dynamic coloring*, but their colors may no longer be utilized.

Furthermore, in Lemma 20, we establish that every connected component of F at the end of PCSF3 is a tight set. This provides additional evidence that the algorithm produces a valid solution.

Finally, we use these lemmas to prove the validity of the solution produced by PCSF3 in Lemma 21.

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

Lemma 17. *At every moment of* PCSF3*, in the* SetPairGraph *representation corresponding to the static coloring of that moment,* C_{source} *is a minimum cut between source and sink.*

Proof. According to Lemma 12, the *static coloring* is always valid during PCSF3. Moreover, based on Corollary 7, when the *static coloring* is valid, C_{source} represents a minimum cut. T

Lemma 18. *A set* $S \subset V$ *is tight if and only if there exists a minimum cut between source and sink in* SetPairGraph *representation of a valid dynamic coloring that does not contain the edge* e *from source to* S*.*

Proof. Using contradiction, let's assume that S is tight and all minimum cuts contain the edge e. Let $\epsilon > 0$ be the difference between the weight of the minimum cut and the first cut whose weight is greater than the minimum cut. By increasing the capacity of the edge e by ϵ, the weight of any minimum cut increases by a positive value ϵ, as well as the maximum flow. This implies that we can increase y_S and still maintain a *valid static coloring*. Therefore, based on the definition of set tightness (Definition 2.2.6), we can conclude that set S is not tight. This contradicts the assumption of the tightness of S and proves that there exists a minimum cut that does not contain the edge e.

Furthermore, if we have a minimum cut that does not contain edge e, increasing the capacity of e does not affect the value of that minimum cut and respectfully the maximum flow. By using Lemma 6, we conclude that increasing y_S would result in an invalid *static coloring*. Therefore, based on Definition 2.2.6, we can conclude that set S is tight. T

Lemma 19. *Once a set* S *becomes tight, it remains tight throughout the algorithm.*

Proof. According to Lemma 17, C_{source} is always a minimum cut. Let us assume that at time t, the set S becomes tight. Based on Lemma 18, there exists a minimum cut C_S that has S on the side of *source*. Therefore, at time t, C_{source} and C_S have the same weight. Now, let us consider a contradiction by assuming that there is a time $t' > t$ when S is not tight. The only difference between SetPairGraph at time t and time t' is the increased capacity of some edges between *source* and sets $S' \subset V$. Let us assume that the total increase in all $y_{S'}$ from time t to t' is d. Since all of these edges are part of the cut C_{source}, the weight of the cut C_{source} is increased by d. Furthermore, since the total capacity of all edges in SetPairGraph from time t to t' has increased by d, the weight of C_S cannot have increased by more than d. That means, the weight of C_S cannot exceed the weight of C_{source} at time t'. Since C_{source} is a minimum cut at time t' according to Lemma 17, we can conclude that C_S remains a minimum cut at time t'. Therefore, based on Lemma 18, the set S is still tight at time t', which contradicts the assumption that it is not tight. T

Lemma 20. *At the end of* PCSF3*, all remaining sets in* FC *are tight.*

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

Lemma 21. *After executing* ReduceTightPairs*, the endpoints of any pair that is not tight will be connected in the forest F'.*

Proof. The forest F' is obtained by removing redundant edges from F, which are edges that are not part of a path between the endpoints of a pair that is not tight. Hence, we only need to show that every pair that is not tight is connected in F. Let us assume, for the sake of contradiction, that there exists a pair (i, j) that is not tight and the endpoints i and j are not connected in F. Consider the set $S \in FC$ at the end of the algorithm that contains i. Since i and j are not connected in F, and S is a connected component of F, it follows that S cuts the pair (i, j). According to Lemma 20, S is a tight set. This contradicts Lemma 5 because we have a tight set S such that $S \odot (i, j)$ is not tight. Therefore, our assumption is false, and every pair that is not tight is connected in F. As a result, after executing ReduceTightPairs, the endpoints of any pair that is not tight will be connected in the forest F'. T

Now we will prove that the running time of PCSF3 is polynomial.
Lemma 22. *For instance I, the runtime of* PCSF3 *is polynomial.*

Proof. We know that Δ_e denotes the time it takes for at least one new edge to be fully colored according to Corollary 2, and Δ_p signifies the time required for at least one active set to be deactivated based on the maximality of Δ_p demonstrated in Lemma 9. During each iteration of the while loop at Line 5, it is guaranteed that at least one of these events takes place.

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

In each iteration, we perform the following operations with polynomial runtime: FindDeltaE, which is polynomial due to Lemma 4; FindDeltaP, which is polynomial according to Corollary 11; iterating through active sets to extend the *static coloring*, which is polynomial based on the size of $ActS$; iterating through edges to update active sets if they fully color edges, which is polynomial; and checking if each active set is tight using CheckSetIsTight, which is polynomial according to Lemma 13.

In the end, we also run ReduceTightPairs, which is polynomial according to Corollary 14.

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

2.3 The Iterative Algorithm

In this section, we present our iterative algorithm which uses the PCSF3 procedure from Algorithm 2 as a building block. We then provide a proof of its 2-approximation guarantee in Section 2.3.1. Finally, in Section 2.3.2, we provide a brief overview of a more refined analysis to establish a $(2 - \frac{1}{n})$-approximation for an n vertex input graph.

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

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

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

Algorithm 6 Iterative PCSF algorithm

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

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

2.3.1 Analysis

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

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

Definition 2.3.1. *For an instance I, we define four sets to categorize the pairs based on their connectivity in both the optimal solution OPT of I and the result of* PCSF3(I)*, denoted as* (Q_1, F'_1)*:*

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

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

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

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

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

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

Definition 2.3.2 (Single-edge and multi-edge sets)**.** *For an instance I, we define a set $S \subset V$ as a single-edge set if it cuts exactly one edge of OPT, i.e., $d_{F^*}(S) = 1$, and as a multi-edge set if it cuts at least two edges of OPT, i.e., $d_{F^*}(S) > 1$. Let cp_1 represent the duration of coloring with pairs in $\mathcal{CP}$ in dynamic coloring that corresponds to coloring with single-edge sets in static coloring. Similarly, let cp_2 represent the duration of coloring with pairs in $\mathcal{CP}$ in dynamic coloring that corresponds to coloring with multi-edge sets in static*

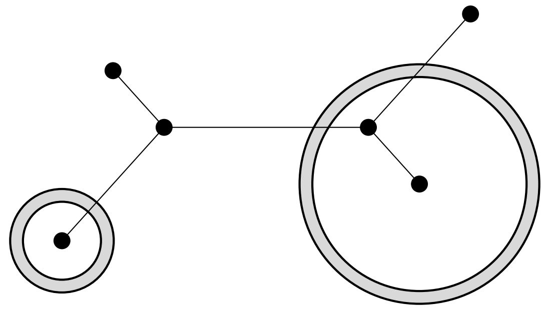

Figure 2.4: A comparison between a single-cut set (left) and a multi-cut set (right).

coloring. These values are formally defined as follows:

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

Figure 2.4 displays a single-edge set on the left and a multi-edge set on the right.

Lemma 23. *For an instance I, we have $cp_1 + cp_2 = cp$.*

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

Now, we use these definitions and categorizations to analyze our algorithm. All of the following lemmas are based on the assumption that IPCSF is executed on instance I. First, in Lemma 24, we provide a lower bound on the cost of the optimal solution, which is $cost(OPT) \geq cc + cp + cp_2 + pc + pp$. Next, in Lemma 26, we present an upper bound on the output of PCSF3(I), which is $cost_1 \leq 2cc + 2pc + 3cp + 3pp$. Moreover, in Lemma 27, we show that this value is at most $2cost(OPT) + cp_1 - cp_2 + pp$.

Next, we want to bound the output of the recursive call within IPCSF. In Lemma 29, we initially proof that $cost(OPT_R) \leq cost(OPT) - pp - cp_1$, where $cost(OPT_R)$ represents the cost of the optimal solution for the instance R defined at Line 12. Finally, in Theorem 30, we employ induction to demonstrate that $cost$(IPCSF) $\leq 2cost(OPT)$. Here, $cost$(IPCSF) denotes the cost of the output produced by IPCSF on instance I. To accomplish this, we use the same induction to bound the cost of the solution obtained through the recursive call at Line 14 by $cost_2 \leq 2cost(OPT_R) + cp + pp$, and by utilizing Lemma 29, we can then conclude that $cost_2 \leq 2cost(OPT) - cp_1 + cp_2 - pp$. Taking the average of $cost_1$ and $cost_2$ results in a value that is at most $2cost(OPT)$. Consequently, the minimum of these two values, corresponding to the cost of IPCSF(I), is at most $2cost(OPT)$.

Lemma 24. *For an instance I, We can derive a lower bound for the cost of the optimal solution OPT as follows:*

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

Proof. The optimal solution pays penalties for pairs with labels $\mathcal{PC}$ and $\mathcal{PP}$ as it does not connect them.

Since $y_{ij} \leq \pi_{ij}$ for each pair (i, j), we can lower bound the penalty paid by OPT as

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

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

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

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

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

For pairs in $\mathcal{CP}$, we have

$$\begin{aligned}
\sum_{(i,j)\in\mathcal{CP}} \sum_{S\odot(i,j)} d_{F^*}(S) \cdot y_{Sij} &= \sum_{(i,j)\in\mathcal{CP}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)=1}} d_{F^*}(S) \cdot y_{Sij} + \sum_{(i,j)\in\mathcal{CP}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)>1}} d_{F^*}(S) \cdot y_{Sij} \\
&\geq \sum_{(i,j)\in\mathcal{CP}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)=1}} y_{Sij} + \sum_{(i,j)\in\mathcal{CP}} \sum_{\substack{S\odot(i,j),\\ d_{F^*}(S)>1}} 2y_{Sij} \\
&= cp_1 + 2cp_2 \\
&= cp + cp_2. && \text{(Lemma 23)}
\end{aligned}$$

Summing up all the components, we have:

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

T

Lemma 25. *Let F be an arbitrary forest and S be a subset of vertices in F. If S cuts only one edge e in F, then removing this edge will only disconnect pairs of vertices cut by S.*

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

Lemma 26. *For an instance I, during the first iteration of* IPCSF(I) *where* PCSF3(I) *is invoked, we can establish an upper bound on the output of* PCSF3 *as follows:*

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

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

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

Now, it suffices to show that $c(F'_1) \leq 2(cc + cp + pc + pp)$. We can prove this similarly to the proof presented by Goemmans and Williamson in (Goemans and Williamson, 1995). Since each pair belongs to exactly one of the sets $\mathcal{CC}$, $\mathcal{CP}$, $\mathcal{PC}$, and $\mathcal{PP}$, we can observe that

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

Therefore, our goal is to prove that the cost of F'_1 is at most $2\sum_{S\subset V} y_S$ using properties of *static coloring*. To achieve this, we show that the portion of edges in F'_1 colored during each step of PCSF3 is at most twice the total increase in the y_S values during that step. Since every edge in the forest F'_1 is fully colored by PCSF3, this will establish the desired inequality.

Now, let's consider a specific step of the procedure PCSF3 where the y_S values of the active sets in $ActS$ are increased by Δ. In this step, the total proportion of edges in F'_1 that are colored by an active set S is $\Delta d_{F'_1}(S)$. Therefore, we want to prove that

$$\Delta \sum_{S\in ActS} d_{F'_1}(S) \leq 2\Delta \cdot |ActS|,$$

where the left-hand side represents the length of coloring on the edges of F'_1 in this step, while the right-hand side represents twice the total increase in y_S values.

Consider the graph H formed from F'_1 by contracting each connected component in FC at this step in the algorithm. As the edges of forest F at this step and F'_1 are a subset of the forest F at the end of PCSF3, the graph H should be a forest. If H contains a cycle, it contradicts the fact that F at the end of PCSF3 is a forest.

In forest H, each vertex represents a set $S \in FC$, and the neighboring edges of this vertex are exactly the edges in $\delta(S) \cap F'_1$. We refer to the vertices representing active sets as active vertices, and the vertices representing inactive sets as inactive vertices. To simplify the analysis, we remove any isolated inactive vertices from H.

Now, let's focus on the inactive vertices in H. Each inactive vertex must have a degree of at least 2 in H. Otherwise, if an inactive vertex v has a degree of 1, consider the only edge in H connected to this vertex. For this edge not to be removed in the final step of Algorithm 2 at Line 21, there must exist a pair outside of Q_1

that would be disconnected after deleting this edge. However, since vertex v is inactive, its corresponding set S becomes tight before this step. According to Lemma 19, S will remain tight afterward. As a result, by Lemma 5, any pair cut by S will also be tight in the final coloring and will be included in Q_1. By applying Lemma 25, we can conclude that the only pairs disconnected by removing this edge would be the pairs cut by S, which we have shown to be in Q_1. Therefore, an inactive vertex cannot have a degree of 1, and all inactive vertices in H have a degree of at least 2. Let V_a and V_i represent the sets of active and inactive vertices in H, respectively. We have

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

This completes the proof. T

Lemma 27. *For an instance I, during the first iteration of* IPCSF(I) *where* PCSF3(I) *is invoked, we can establish an upper bound on the output of* PCSF3 *as follows:*

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

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

$$\begin{aligned}
cost_1 \leq& 2cc + 2pc + 3cp + 3pp && \text{(Lemma 26)} \\
=& 2(cc + cp + cp_2 + pc + pp) + cp - 2cp_2 + pp \\
\leq& 2cost(OPT) + (cp - cp_2) - cp_2 + pp && \text{(Lemma 24)} \\
=& 2cost(OPT) + cp_1 - cp_2 + pp. && \text{(Lemma 23)}
\end{aligned}$$

T

Lemma 28. *For an instance I, it is possible to remove a set of edges from F^* with a total cost of at least cp_1 while ensuring that the pairs in $\mathcal{CC}$ remain connected.*

Proof. Consider a single-edge set S that cuts some pair (i, j) in $\mathcal{CP}$ with $y_{Sij} > 0$. Since (i, j) is in $\mathcal{CP}$, it is also in Q_1 and therefore tight. By Lemma 16, any other pair cut by S will also be tight. Consequently, the pairs in $\mathcal{CC}$ will not be cut by S since they are not tight. Furthermore, according to Lemma 25, if S cuts only one edge e of F^*, then the only pairs that will be disconnected by removing edge e from F^* are the pairs that are cut by S. However, we have already shown that no pair in $\mathcal{CC}$ is cut by S. Therefore, all pairs in $\mathcal{CC}$ will remain connected even after removing edge e. See Figure 2.5 for an illustration.

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

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

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

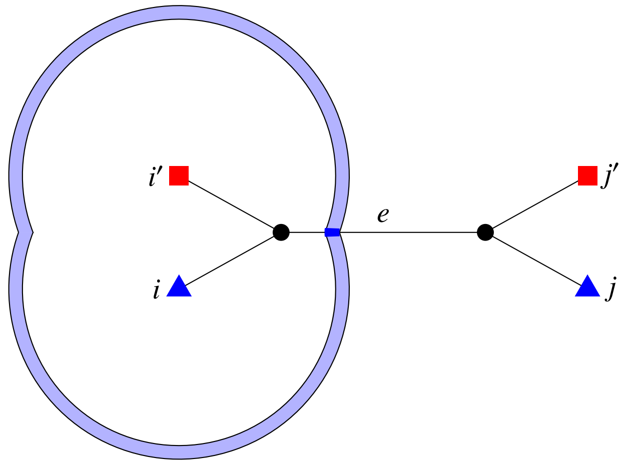

Figure 2.5: The figure shows the graph of F^* with pairs (i, j) and (i', j'), and a single-edge set colored with pair (i, j) in *dynamic coloring*. Tightness of (i, j) implies tightness of (i', j'), and removing edge e does not disconnect pairs in $\mathcal{CC}$.

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

Lemma 29. *For an instance I and the instance R constructed at Line 12 during the execution of* IPCSF(I)*, we have*

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

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

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

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

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

Moreover, using Lemma 28, we construct F'_R from F^* by removing a set of edges with a total length of at least cp_1, while ensuring that the remaining forest still connects all the pairs in $\mathcal{CC}$. Therefore, we can bound the cost of F'_R as

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

Summing it all together, we have

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

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

Finally, we can bound the cost of the output of IPCSF. For an instance I, let's denote the cost of the output of IPCSF(I) as $cost$(IPCSF). In Theorem 30, we prove that the output of IPCSF is a 2-approximate solution for the PCSF problem.

Theorem 30. *For an instance I, the output of* IPCSF(I) *is a 2-approximate solution to the optimal solution for I, meaning that*

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

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

First, the algorithm makes a call to the PCSF3 procedure to obtain a solution (Q_1, F'_1). If $\pi(Q_1) = 0$ for this solution, which means no cost is incurred by paying penalties, the algorithm terminates and returns this solution at Line 4. This will always be the case in the base case of our induction where for all pairs $(i, j) \in Q_1$, penalties π_{ij} are equal to 0. Since every pair $(i, j) \in Q_1$ is tight, we have $y_{ij} = \pi_{ij} = 0$. Given that $\mathcal{CP}$ and $\mathcal{PP}$ are subsets of Q_1, we can conclude that $cp = cp_1 = cp_2 = pp = 0$. Now, by Lemma 27, we have

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

Therefore, when IPCSF returns at Line 4, we have

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

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

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

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

In addition, we have

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

where we use the fact that $\pi'_{ij} = \pi_{ij}$ for $(i, j) \notin Q_1$. Now we can bound the cost of the solution (Q_2, F_2'), denoted as $cost_2$, by

$$\begin{aligned}
cost_2 &= c(F_2') + \pi(Q_2) \\
&\leq c(F_2') + \pi'(Q_2) + \pi(Q_1) \\
&\leq 2cost(OPT_R) + \pi(Q_1) && \text{(By induction)} \\
&\leq 2\left(cost(OPT) - pp - cp_1\right) + \sum_{(i,j)\in Q_1} \pi_{ij} && \text{(Lemma 29)} \\
&= 2\left(cost(OPT) - pp - cp_1\right) + \sum_{(i,j)\in Q_1} y_{ij} && \text{(pairs in } Q_1 \text{ are tight)} \\
&= 2cost(OPT) - 2pp - 2cp_1 + cp + pp \\
&= 2cost(OPT) - cp_1 + cp_2 - pp. && \text{(Lemma 23)}
\end{aligned}$$

Furthermore, according to Lemma 27, the cost of the solution (Q_1, F_1'), denoted as $cost_1$, can be bounded by

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

Finally, in Line 15, we return the solution with the smaller cost between (Q_1, F_1') and (Q_2, F_2'). Based on the upper bounds above on both solutions, we know that

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

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

Theorem 31. *The runtime of the* IPCSF *algorithm is polynomial.*

Proof. Let n be the number of vertices in the input graph. There are $O(n^2)$ pairs of vertices in total. Whenever IPCSF calls itself recursively, the number of pairs with non-zero penalties decreases by at least one, otherwise IPCSF will return at Line 4. Thus, the recursion depth is polynomial in n. At each recursion level, the algorithm only runs PCSF3 on one instance of the problem and performs $O(n^2)$ additional operations. By Lemma 22, we know that PCSF3 runs in polynomial time. Therefore, the total run-time of IPCSF will also be polynomial. T

2.3.2 Improving the approximation ratio

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

We first introduce an improved version of Lemmas 26 and 27.

Lemma 32. *For an instance I, during the first iteration of* IPCSF(I) *where* PCSF3(I) *is invoked, we have the following upper bound*

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

Proof. We proceed similarly to the proof of Lemma 26 and make a slight change. In one of the last steps of that proof, we use the following inequality:

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

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

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

Based on the steps in the proof of Lemma 26, this leads to the desired upper bound. T

Lemma 33. *For an instance I, during the first iteration of* IPCSF(I) *where* PCSF3(I) *is invoked, we can establish an upper bound on the output of* PCSF3 *as follows:*

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

Proof. We prove this lemma similarly to Lemma 27, except we use Lemma 32 instead of Lemma 26.

$$\begin{aligned}
cost_1 &\leq (2 - \frac{2}{n}) \cdot cc + (2 - \frac{2}{n}) \cdot pc + (3 - \frac{2}{n}) \cdot cp + (3 - \frac{2}{n}) \cdot pp && \text{(Lemma 32)} \\
&= (2 - \frac{2}{n})(cc + cp + cp_2 + pc + pp) + cp - (2 - \frac{2}{n}) \cdot cp_2 + pp \\
&\leq (2 - \frac{2}{n}) \cdot cost(OPT) + (cp - cp_2) - (1 - \frac{2}{n})cp_2 + pp && \text{(Lemma 24)} \\
&= (2 - \frac{2}{n}) \cdot cost(OPT) + cp_1 - (1 - \frac{2}{n}) \cdot cp_2 + pp. && \text{(Lemma 23)}
\end{aligned}$$

T

Finally, we improve Theorem 30.

Theorem 34. *For an instance I, the output of* IPCSF(I) *is a* $(2 - \frac{1}{n})$*-approximate solution to the optimal solution for I, meaning that*

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

Proof. Similarly to the proof of Theorem 30, we use induction on the number of non-zero penalties. If the algorithm terminates on Line 4 then by Lemma 33 we have

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

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

Using the same reasoning as the proof of Theorem 30, based on the induction we have

$$\begin{aligned}
cost_2 &\le (2 - \frac{1}{n}) \cdot cost(OPT_R) + \pi(Q_1) \\
&= (2 - \frac{1}{n}) \cdot cost(OPT_R) + cp + pp \\
&\le (2 - \frac{1}{n})(cost(OPT) - cp_1 - pp) + cp + pp && \text{(By Lemma 29)} \\
&\le (2 - \frac{1}{n}) \cdot cost(OPT) - (1 - \frac{1}{n}) \cdot cp_1 + cp_2 - (1 - \frac{1}{n}) \cdot pp.
\end{aligned}$$

We can combine this with the following upper bound from Lemma 33

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

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

$$\begin{aligned}
cost(\text{IPCSF}) &= \min(cost_1, cost_2) \le \frac{1}{2}(cost_1 + cost_2) \\
&\le \frac{1}{2}\Big[(2 - \frac{2}{n}) \cdot cost(OPT) + cp_1 - (1 - \frac{2}{n}) \cdot cp_2 + pp \\
&\quad +(2 - \frac{1}{n}) \cdot cost(OPT) - (1 - \frac{1}{n}) \cdot cp_1 + cp_2 - (1 - \frac{1}{n}) \cdot pp\Big] \\
&= \frac{1}{2}\left((4 - \frac{3}{n}) \cdot cost(OPT) + \frac{2}{n}cp_2 + \frac{1}{n}cp_1 + \frac{1}{n}pp\right) \\
&\le \frac{1}{2}\left((4 - \frac{2}{n}) \cdot cost(OPT) + \frac{1}{n}[2cp_2 + cp_1 + pp - cost(OPT)]\right) \\
&\le \frac{1}{2}(4 - \frac{2}{n}) \cdot cost(OPT) && (cost(OPT) \ge 2cp_2 + cp_1 + pp \text{ by Lemma 24}) \\
&= (2 - \frac{1}{n}) \cdot cost(OPT).
\end{aligned}$$

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

Chapter 3

Submodular Maximization

3.1 Introduction

Submodularity is a fundamental notion that arises in many applications such as image segmentation, data summarization (Kumari and Bilmes, 2021; Schreiber et al., 2020), RNA and protein sequencing (Yang et al., 2020; Libbrecht et al., 2018) hypothesis identification (Barinova et al., 2012; Chen et al., 2014), information gathering (Radanovic et al., 2018), and social networks (Kempe et al., 2003). A function $f : 2^{\mathcal{V}} \to \mathbb{R}^+$ is called *submodular* if for all $A \subseteq B \subseteq \mathcal{V}$ and $e \notin B$, it satisfies $f(A \cup \{e\}) - f(A) \geq f(B \cup \{e\}) - f(B)$, and it is called *monotone* if for every $A \subseteq B$, it satisfies $f(A) \leq f(B)$.

Given a monotone submodular function $f : 2^{\mathcal{V}} \to \mathbb{R}^+$ that is defined over a ground set $\mathcal{V}$ and a parameter $k \in \mathbb{N}$, in the *submodular maximization problem under the cardinality constraint* k, we would like to report a set $I^* \subseteq \mathcal{V}$ of size at most k whose submodular value is maximum among all subsets of $\mathcal{V}$ of size at most k.

Matroid (Oxley, 1992) is a basic branch of mathematics that generalizes the notion of linear independence in vector spaces and has basic links to linear algebra (MINTY, 1966), graphs (Edmonds, 1971), lattices (Maeda and Maeda, 1970), codes (Kashyap, 2007), transversals (Edmonds and Fulkerson, 1965), and projective geometries (MacLane, 1936). A matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$ consists of a *ground set* $\mathcal{V}$ and a nonempty downward-closed set system $\mathcal{I} \subseteq 2^V$ (known as the independent sets) that satisfies the *exchange axiom*: for every pair of independent sets $A, B \in \mathcal{I}$ such that $|A| < |B|$, there exists an element $x \in B \backslash A$ such that $A \cup \{x\} \in \mathcal{I}$.

A growing interest in machine learning (Tohidi et al., 2020; Han et al., 2020; Elenberg et al., 2017; Krause, 2013; Bateni et al., 2019; Lin and Bilmes, 2011; Sipos et al., 2012; El-Arini and Guestrin, 2011; Agrawal et al., 2009; Wei et al., 2015; Dueck and Frey, 2007), online auction theory (Bateni et al., 2013; Gupta et al., 2010; Babaioff et al., 2018; Kleinberg and Weinberg, 2012; Gharan and Vondrák, 2013; Kleinberg and Weinberg, 2019; Babaioff et al., 2018; Kleinberg and Weinberg, 2012; Ehsani et al., 2018), and combinatorial optimization (Lee et al., 2010b; Chekuri et al., 2011; Lee et al., 2010a; Kempe et al., 2015; Kveton et al., 2014; Papadimitriou and Steiglitz, 1982; Magos et al., 2006) is to study the problem of maximizing a monotone submodular function $f : 2^V \to \mathbb{R}^+$ under a matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$ constraint. In particular, the goal in the *submodular maximization problem under the matroid constraint* is to return an independent set $I^* \in \mathcal{I}$ of the maximum submodular value $f(I^*)$ among all independent sets in $\mathcal{I}$.

The seminal work of Fisher, Nemhauser and Wolsey (Nemhauser et al., 1978) in the 1970s, was the first that considered the submodular maximization problem under the matroid constraint problem in the offline model. Indeed, they developed a simple, elegant leveling algorithm for this problem that in time $O(nk)$ (where k is the rank of the matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$), returns an independent set whose submodular value is a 2-approximation of the optimal value $OPT = \max_{I^* \in \mathcal{I}} f(I^*)$.

However, despite the simplicity and optimality of this celebrated algorithm, there has been a surge of recent research efforts to reexamine these problems under a variety of massive data models motivated by the unique challenges of working with massive datasets. These include streaming algorithms (Badanidiyuru et al., 2014; Feldman et al., 2020; Alaluf et al., 2020; Kazemi et al., 2019), dynamic algorithms (Mirzasoleiman et al., 2017; Kazemi et al., 2018; Lattanzi et al., 2020; Monemizadeh, 2020; Chen and Peng, 2022), sublinear time algorithms (Stan et al., 2017), parallel algorithms (Kupfer et al., 2020; Balkanski and Singer, 2018b,a; Ene and Nguyen, 2020, 2019; Ene et al., 2019; Chekuri and Quanrud, 2019), online algorithms (Harvey et al., 2020), private algorithms (Chaturvedi et al., 2021), learning algorithms (Balcan and Harvey, 2012, 2011; Balkanski et al., 2017) and distributed algorithms (Mirrokni and Zadimoghaddam, 2015; Ene et al., 2017; da Ponte Barbosa et al., 2016, 2015).

Among these big data models, the *(fully) dynamic model* (Rauch, 1992; Henzinger and King, 1999) has been of particular interest recently. In this model, we see a sequence $\mathcal{S}$ of updates (i.e., inserts and deletes) of elements of an underlying structure (such as a graph, matrix, and so on), and the goal is to maintain an approximate or exact solution of a problem that is defined for that structure using a fast update time. For example, the influential work of Onak and Rubinfeld (Onak and Rubinfeld, 2010)(STOC'10) studied dynamic versions of the matching and the vertex cover problems. Some other new advances in the dynamic model have been seen by developing dynamic algorithms for matching and vertex cover (Onak and Rubinfeld, 2010; Bhattacharya et al., 2017; Bernstein and Stein, 2015; Solomon, 2016; Neiman and Solomon, 2016; Charikar and Solomon, 2018; Bernstein et al., 2021a,b; Bhattacharya and Kiss, 2021), graph connectivity (Kapron et al., 2013; Ahn et al., 2012), graph sparsifiers (Bernstein et al., 2022; Abraham et al., 2016; Durfee et al., 2019; Chuzhoy et al., 2020; Chen et al., 2020; Gao et al., 2021; van den Brand et al., 2022), set cover (Bhattacharya et al., 2021; Gupta and Levin, 2020; Bhattacharya et al., 2019; Gupta et al., 2017; Gupta and Levin, 2020; Abboud et al., 2019), approximate shortest paths (Bernstein, 2009; Henzinger et al., 2013; Bernstein, 2016a,b; Henzinger et al., 2016a; van den Brand and Nanongkai, 2019; Henzinger et al., 2016b; Abraham et al., 2017), minimum spanning forests (Nanongkai et al., 2017; Nanongkai and Saranurak, 2017; Chechik and Zhang, 2020), densest subgraphs (Bhattacharya et al., 2015; Sawlani and Wang, 2020), maximal independent sets (Assadi et al., 2018; Behnezhad et al., 2019; Chechik and Zhang, 2019), spanners (Bernstein et al., 2021b; Baswana, 2006; Bodwin and Krinninger, 2016; Baswana et al., 2012), and graph coloring (Solomon and Wein, 2020; Bhattacharya et al., 2022a), to name a few[1].

However, for the very basic problem of submodular maximization under the matroid constraint, there is no (fully) dynamic algorithm known. This problem was repeatedly posed as an open problem at STOC'22 (Chen and Peng, 2022) as well as NeurIPS'20 (Lattanzi et al., 2020). Indeed, Chen and Peng (Chen and Peng, 2022)(STOC'22) raised the following open question:

> **Open question (Chen and Peng, 2022; Lattanzi et al., 2020):** "For fully dynamic streams [sequences of insertions and deletions of elements], there is no known constant-factor approximation algorithm with poly(k) amortized queries for matroid constraints."

In this paper, we answer this question as well as the open problem of Lattanzi et al. (Lattanzi et al., 2020) (NeurIPS'20) affirmatively. As a byproduct, we also develop a dynamic algorithm for the submodular maximization problem under the cardinality constraint. We next state our main result.

[1]Interestingly, the best paper awards at SODA'23 were awarded to two dynamic algorithms (Behnezhad, 2022; Bhattacharya et al., 2022b) for the matching size problem in the dynamic model.

Theorem 35 (Main Theorem)**.** *Suppose we are provided with oracle access to a monotone submodular function $f : 2^{\mathcal{V}} \to \mathbb{R}^+$ that is defined over a ground set $\mathcal{V}$. Let $\mathcal{S}$ be a sequence of insertions and deletions of elements of the underlying ground set $\mathcal{V}$. Let $0 < \epsilon \leq 1$ be an error parameter.*

- *We develop the first parameterized (by the rank k of a matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$) dynamic $(4+\epsilon)$-approximation algorithm for the submodular maximization problem under the matroid constraint using a worst-case expected $O(k \log(k) \log^3(k/\epsilon))$ query complexity.*
- *We also present a parameterized (by the cardinality constraint k) dynamic algorithm for the submodular maximization under the cardinality constraint k, that maintains a $(2+\epsilon)$-approximate solution of the sequence $\mathcal{S}$ at any time t using a worst-case expected complexity $O(k\epsilon^{-1}\log^2(k))$.*

The seminal work of Fisher, Nemhauser and Wolsey (Nemhauser et al., 1978), which we mentioned above, developed a simple and elegant leveling algorithm for the submodular maximization problem under the cardinality constraint that achieves the optimal approximation ratio of $\frac{e}{e-1} \approx 1.58$ in time $O(nk)$ (Nemhauser et al., 1978; Feige, 1998).

The study of the submodular maximization in the dynamic model was initiated at NeurIPS 2020 based on two independent works due to Lattanzi, Mitrovic, Norouzi-Fard, Tarnawski, and Zadimoghaddam (Lattanzi et al., 2020) and Monemizadeh (Monemizadeh, 2020). Both works present dynamic algorithms that maintain $(2+\epsilon)$-approximate solutions for the submodular maximization under the cardinality constraint k in the dynamic model. The amortized expected query complexity of these two algorithms are $O(\epsilon^{-11}\log^6(k)\log^2(n))$ and $O(k^2\epsilon^{-3}\log^5(n))$, respectively.

Our dynamic algorithm for the cardinality constraint improves upon the dynamic algorithm that Monemizadeh (Monemizadeh, 2020) (NeurIPS'20) developed for this problem using an expected query complexity $O(k^2\epsilon^{-3}\log^5(n))$. In particular, our dynamic algorithm is the first one for this problem whose query complexity is independent of the size of the ground set $\mathcal{V}$.

We develop our dynamic algorithm for the submodular maximization problem under the matroid or cardinality constraint by designing a randomized leveled data structure that supports insertion and deletion operations, maintaining an approximate solution for the given problem. In addition, we develop a fast construction algorithm for our data structure that uses a one-pass over a random permutation of the elements and utilizes a monotonicity property of our problems which has a subtle proof in the matroid case. We believe these techniques could also be useful for other optimization problems in the area of dynamic algorithms.

Interestingly, our results can be seen from the lens of parameterized complexity (Marx, 2008; Fomin and Korhonen, 2022; Korhonen, 2021; Kawarabayashi and Thorup, 2011; Fafianie and Kratsch, 2014; Chitnis et al., 2016a, 2015, 2016b; Gupta et al., 2018). In particular, the query complexity of our dynamic algorithms for the submodular maximization problems under the matroid and cardinality constraints (1) are independent of the size of the ground set $\mathcal{V}$ (i.e., $|\mathcal{V}| = n$) , and (2) are parameterized by the rank k of the matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$ and the cardinality k, respectively. We hope that our work sheds light on the connection between dynamic algorithms and the Fixed-Parameter Tractability (FPT) (Downey and Fellows, 2012; Flum and Grohe, 2006; Cygan et al., 2015) world. We should mention that streaming algorithms (Fafianie and Kratsch, 2014; Chitnis et al., 2016a, 2015) through the lens of the parameterized complexity have been considered before where vertex cover and matching parameterized by their size were designed in these works.

Finally, one may ask whether we can obtain a dynamic c-approximate algorithm for the cardinality constraint for $c < 2$ with a query complexity that is polynomial in k. Let $g : \mathbb{N} \to \mathbb{R}^+$ be an arbitrary function. Building on a hardness result recently obtained by Chen and Peng (Chen and Peng, 2022), we show in Appendix 3.5 that there is no randomized $(2-\epsilon)$-approximate algorithm for the dynamic submodular maximization under cardinality constraint k with amortized expected query time of $g(k)$ (e.g., not even doubly exponentially in k), even if the optimal value is known after every insertion/deletion.

Concluding remark. This paper is a merge of two papers NeurIPS'20 and SODA'24. However, since the second paper subsumes the first paper, we explain the algorithm in context and algorithms of the second paper. After the neurips submission, this area was expanded.

Concurrent work in footnote. In a concurrent work, Dütting, Fusco, Lattanzi, Norouzi-Fard, and Zadimoghaddam (Dütting et al., 2023) also provide an algorithm for dynamic submodular maximization under a matroid constraint. Their algorithm obtains a $4+\epsilon$ approximation with $\frac{k^2}{\epsilon}\log(k)\log^2(n)\log^3(\frac{k}{\epsilon})$ amortized expected query comlpexity. [2]

Our query complexity of $k\log(k)\log^3(\frac{k}{\epsilon})$ is strictly better as **(a)** it does not depend on n and **(b)** its dependence on k is nearly linear rather than nearly quadratic and the dependence on ϵ^{-1} is polylogarithmic. Additionally, our guarantees are worst-case expected, rather than amortized expected.

3.1.1 Preliminaries

Notations. For two natural numbers $x<y$, we use $[x,y]$ to denote the set $\{x,x+1,\cdots,y\}$, and $[x]$ to denote the set $\{1,2,\cdots,x\}$. For a set A and an element e, we denote by $A+e$, the set that is the union of two sets A and $\{e\}$. Similarly, for a set A and an element $e\in A$, we denote by $A-e$ or $A\backslash e$, the set A from which the element e is removed. For a level L_i, we represent by $L_{1\leq j\leq i}$ the levels $L_1,L_2,\cdots,L_i$, and we simplify $L_{1\leq j\leq i}$ and show it by $L_{\leq i}$. The levels $L_{i\leq j\leq T}$ and its simplification $L_{i\leq}$ are defined similarly. For a function x and a set A, we denote by $x[A]$ the function x that is restricted to domain A. For an event E, we use $\mathbb{1}[E]$ as the *indicator function* of E. That is, $\mathbb{1}[E]$ is set to one if E holds and is set to zero otherwise. For random variables and their values, we use bold and non-bold letters, respectively. For example, we denote a random variable by $\mathbf{X}$ and its value by X. We will use the notations $\mathbb{P}[\mathbf{X}]$ and $\mathbb{E}[\mathbf{X}]$ to denote the probability and the expectation of a random variable $\mathbf{X}$. For two events A and B, we will use the notation $\mathbb{P}[A|B]$ to denote "the conditional probability of A given B" or "the probability of A under the condition B". For an event A with nonzero probability and a discrete random variable $\mathbf{X}$, we denote by $\mathbb{E}[\mathbf{X}|A]$ conditional expectation of X given A, which is $\mathbb{E}[\mathbf{X}|A]=\sum_x x\cdot\mathbb{P}[\mathbf{X}=x|A]$. Similarly, if $\mathbf{X}$ and $\mathbf{Y}$ are discrete random variables, the conditional expectation of $\mathbf{X}$ given $\mathbf{Y}$ is denoted by $\mathbb{E}[\mathbf{X}|\mathbf{Y}=y]$.

Submodular function. Given a ground set $\mathcal{V}$, a function $f:2^{\mathcal{V}}\to\mathbb{R}^+$ is called *submodular* if it satisfies $f(A\cup\{e\})-f(A)\geq f(B\cup\{e\})-f(B)$, for all $A\subseteq B\subseteq\mathcal{V}$ and $e\notin B$. In this paper, we assume that f is *normalized*, i.e., $f(\emptyset)=0$. When f satisfies the additional property that $f(A\cup\{e\})-f(A)\geq 0$ for all A and $e\notin A$, we say f is *monotone*. For a subset $A\subseteq\mathcal{V}$ and an element $e\in\mathcal{V}\backslash A$, the function $f(A\cup\{e\})-f(A)$ is often called the *marginal gain* (Badanidiyuru et al., 2014; Kazemi et al., 2019) of adding e to A.

Let $f:2^{\mathcal{V}}\to\mathbb{R}^+$ be a monotone submodular function defined on the ground set $\mathcal{V}$. The *monotone submodular maximization problem under the cardinality constraint* k is defined as finding $OPT=\max_{I\subseteq\mathcal{V}:|I|\leq k}f(I)$. We denote by I^* an optimal subset of size at most k that achieves the optimal value $OPT=f(I^*)$. Note that we can have more than one optimal set.

The leveling algorithm of the seminal work of Nemhauser, Wolsey, and Fisher (Nemhauser et al., 1978) that can approximate OPT to a factor of $(1-1/e)$, is as follows. In the beginning, we let $S=\emptyset$. We then take k passes over the set V, and in each pass, we find an element $e\in V$ that maximizes the marginal gain $f(S\cup\{e\})-f(S)$, add it to S and delete it from V.

[2] The two works appeared on arxiv at the same time; We had submitted an earlier version of our work to SODA'23, in July 2022.

Access Model. We assume the access to a monotone submodular function $f : 2^{\mathcal{V}} \to \mathbb{R}^+$ is given by an *oracle*. That is, we consider the oracle that allows *set queries* such that for every subset $A \subseteq \mathcal{V}$, one can query the value $f(A)$. The marginal gain $f(A \cup \{e\}) - f(A)$ for every subset $A \subseteq \mathcal{V}$ and an element $e \in \mathcal{V}$ in this query access model can be computed using two queries $f(A \cup \{e\})$ and $f(A)$.

Matroid. A matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$ consists of a *ground set* $\mathcal{V}$ and a nonempty downward-closed set system $\mathcal{I} \subseteq 2^{\mathcal{V}}$ (known as the independent sets) that satisfies the *exchange axiom*: for every pair of independent sets $A, B \in \mathcal{I}$ such that $|A| < |B|$, there exists an element $x \in B \backslash A$ such that $A \cup \{x\} \in \mathcal{I}$. A subset of the ground set $\mathcal{V}$ that is not independent is called *dependent*. A maximal independent set—that is, an independent set that becomes dependent upon adding any other element—is called a *basis* for the matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$. A *circuit* in a matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$ is a minimal dependent subset of $\mathcal{V}$—that is, a dependent set whose proper subsets are all independent. Let A be a subset of V. The rank of A, denoted by $rank(A)$, is the maximum cardinality of an independent subset of A.

Let $f : 2^{\mathcal{V}} \to \mathbb{R}^{\geq 0}$ be a monotone submodular function defined on the ground set $\mathcal{V}$ of a matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$. We denote by $OPT = \max_{I \in \mathcal{I}} f(I)$ the maximum submodular value of an independent set in $\mathcal{I}$. We denote by $I^* \in \mathcal{I}$ an independent set that achieves the optimal value $OPT = f(I^*)$.

Here, we bring two lemmas about matroids that will be used in our paper.
Lemma 36 ((Oxley, 1992))**.** *The family C of circuits of a matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$ has the following properties:*

- *(C1) $\emptyset \notin C$.*
- *(C2) if C_1, $C_2 \in C$ and $C_1 \subseteq C_2$, then $C_1 = C_2$.*
- *(C3) if C_1, $C_2 \in C$, $C_1 \neq C_2$ and $e \in C_1 \cap C_2$, then there exists $C_3 \in C$ such that $C_3 \subseteq C_1 \cup C_2 \setminus \{e\}$.*

Lemma 37. *Let $e \in V$ be an element, and $I \in \mathcal{I}$ be an independent set. Then $I \cup \{e\}$ has at most one circuit.*

Proof. For the sake of contradiction, suppose there are two circuits $C_1, C_2 \subseteq I \cup \{e\}$, where $C_1 \neq C_2$. As I is an independent set, $C_1, C_2 \nsubseteq I$, which means $e \in C_1 \cap C_2$. Then using Lemma 36, there exists a circuit $C_3 \subseteq C_1 \cup C_2 \setminus \{e\}$. Since $C_1 \cup C_2 \setminus \{e\} \subseteq I$, we have $C_3 \subseteq I$, which is a contradiction to the fact that the set I is an independent set in $\mathcal{I}$. T

Dynamic Model. Let $\mathcal{S}$ be a sequence of insertions and deletions of elements of an underlying ground set $\mathcal{V}$. Let $\mathcal{S}_t$ be the sequence of the first t updates (insertion or deletion) of the sequence $\mathcal{S}$. By time t, we mean the time after the first t updates of the sequence $\mathcal{S}$, or simply when the updates of $\mathcal{S}_t$ are done. We define V_t as the set of elements that have been inserted until time t but have not been deleted after their latest insertion.

Given a monotone submodular function $f : 2^{\mathcal{V}} \to \mathbb{R}^+$ defined on the ground set $\mathcal{V}$, the aim of *dynamic monotone submodular maximization problem under the cardinality constraint k* is to have (an approximation of) $OPT_t = \max_{I_t \subseteq V_t : |I| \leq k} f(I_t)$ at any time t. Similarly, the aim of *dynamic monotone submodular maximization problem under a matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$ constraint* for a monotone function f defined over the ground set $\mathcal{V}$ is to have (an approximation of) $OPT_t = \max_{I_t \subseteq V_t : I_t \in \mathcal{I}} f(I_t)$ at any time t. We also define MAX_t to be $\max_{e \in V_t} f(e)$. For simplicity, during the analysis for a fixed time t, we use V, OPT, and MAX instead of V_t, OPT_t, and MAX_t respectively.

Our dynamic algorithm is in the oblivious adversarial model as is common for analysis of randomized data structures such as universal hashing (Carter and Wegman, 1977). The model allows the adversary, who is aware of the submodular function f and the algorithm that is going to be used, to determine all the arrivals and departures of the elements in the ground set $\mathcal{V}$. However, the adversary is unaware of the random bits used in the algorithm and so cannot choose updates adaptively in response to the randomly guided choices

of the algorithm. Equivalently, we can suppose that the adversary prepares the full input (insertions and deletions) before the algorithm runs.
The *query complexity* of an α-approximate dynamic algorithm is the number of oracle queries that the algorithm must make to maintain an α-approximate of the solution at time t, given all computations that have been done till time $t-1$.
We measure the *time complexity* of our dynamic algorithm in terms of its *query complexity*, taking into account queries made to either the submodular oracle for f or the matroid independence oracle for $\mathcal{I}$.

The query complexities of the algorithms in our paper will be worst-case expected. An algorithm is said to have worst-case expected update time (or query time) α if for every update x, the expected time to process x is at most α. We refer to Bernstein, Forster, and Henzinger (Bernstein et al., 2021b) for a discussion about the worst-case expected bound for dynamic algorithms.

3.1.2 Our contribution and overview of techniques

Our dynamic algorithms for the submodular maximization problems with cardinality and matroid constraints consist of the following three building blocks.

- *Fast leveling algorithms:* We first develop linear-time leveling algorithms for these problems based on random permutations of elements. These algorithms are used in *rare occasions* when we need to (partially or totally) reset a solution that we maintain.
- *Insertion and deletion subroutines:* We next design subroutines for inserting and deleting a new element. Upon insertion or deletion of an element, these subroutines often perform *light* computations, but in rare occasions, they perform *heavy* operations by invoking the leveling algorithms to (partially or totally) reset a solution that we maintain.
- *Relax OPT or MAX assumptions:* For the leveling algorithms, and the insertion and the deletion subroutines, we assume we know either an approximation of OPT (as for the cardinality constraint) or an approximation of the maximum submodular value $MAX = \max_{e\in V} f(e)$ of an element (as for the matroid constraint). In the final block of our dynamic algorithms, we show how to relax such an assumption.

Submodular maximization problem under the cardinality constraint

Designing and analyzing the above building blocks for the cardinality constraint is simpler than designing and developing them for the matroid constraint. Therefore, we outline them first for the cardinality constraint. That gives the intuition and sheds light on how we develop these building blocks for the matroid constraint which are more involved. Since the main contribution of our paper is developing a dynamic algorithm for the matroid constraint, we explain the algorithms (in Section 3.2) and the analysis (in Section 3.3) for the matroid first. The dynamic algorithm for the cardinality constraint is given in Section 3.4.

Suppose for now, we know the optimal value $OPT = \max_{I^*\subseteq V:|I^*|\le k} f(I^*)$ of any subset of the set V of size at most k. We consider the fixed threshold $\tau = \frac{OPT}{2k}$.

First building block: Fast leveling algorithm. Our leveling algorithm constructs a set of levels $L_0, L_1, \cdots, L_T$, where T is a random variable guaranteed to be $T \le k$. Every level L_ℓ consists of two sets R_ℓ and I_ℓ, and an element e_ℓ so that:

1. $R_0 = V$, $I_0 = \emptyset$, and $R_1 = \{e \in R_0 : f(e) \ge \tau\}$
2. $R_0 \supseteq R_1 \supset \cdots \supset R_T \supset R_{T+1} = \emptyset$

3. For $1 \leq \ell \leq T$, we have $I_\ell = I_{\ell-1} \cup \{e_\ell\}$

4. We report the set I_T as the solution

The illustration of our construction is shown in Figure 3.1.

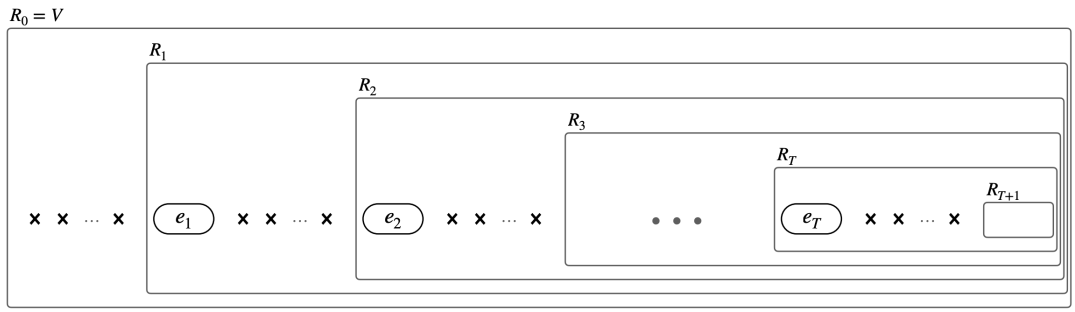

Figure 3.1: The illustration of our leveling algorithm.

The key concept in constructing the levels is the notion of *promoting elements*.

Definition 3.1.1 (Promoting elements)**.** *Let $L_{1\leq\ell\leq T}$ be a level. We call an element $e \in R_\ell$, a promoting element for the set I_ℓ if $f(I_\ell \cup \{e\}) - f(I_\ell) \geq \tau$ and $|I_\ell| < k$.*

The levels are constructed as follows: Let $\ell = 1$. We first randomly permute the elements of the set R_1 and denote by P this random permutation. We next iterate through the elements of P and for every element $e \in P$, we check if e is a promoting element with respect to the set $I_{\ell-1}$ or not.

- If e is a promoting element for the set $I_{\ell-1}$, we then let e_ℓ be e and let I_ℓ be $I_{\ell-1} \cup \{e_\ell\}$. Observe that now we have the set I_ℓ and the element e_ℓ, however, the set R_ℓ is not complete yet, as some of its elements may come after e in the permutation P. We create the next level $L_{\ell+1}$ by setting $R_{\ell+1} = \emptyset$. We then proceed to the next element in P. Note that in this way, for all levels $L_{1<j\leq\ell}$, the sets R_j are not complete, and they will be complete when we reach the end of the permutation P.
- Next, we consider the case when e is not a promoting element for the set $I_{\ell-1}$. This essentially means that we need to find the largest $z \in [1, \ell-1]$ so that e is promoting for the set I_{z-1}, but it is not promoting for the set I_z. A naive way of doing that is to perform a linear scan for which we need one oracle query to compute $f(I_x \cup \{e\}) - f(I_x)$ for every $x \in [1, \ell - 1]$. However, we do a binary search in the interval $[i, \ell - 1]$, which needs $O(\log k)$ oracle queries to find z. Once we find such z, we add e to all sets R_r for $2 \leq r \leq z$[3]. Observe that adding e to all these sets may need $O(k)$ time, but we do not need to do oracle queries in order to add e to these sets.

The permutation P has at most n elements, and we do the above operations for every such element. Thus, the leveling algorithm may require a total of $O(n \log k)$ oracle queries. Observe that the implicit property that we use to perform the binary search is the *monotonicity property* which says if an element is a promoting for a set I_{z-1}, it is promoting for all sets $I_{\leq z-1}$. For the cardinality constraint, the monotone property is trivial to see. However, it is ***intricate*** for the ***matroid*** constraint. We will develop a leveling algorithm for the matroid constraint, which satisfies a monotonicity property, allowing us to perform the binary search.

Second building block: Insertion and deletion of an element. Next, we explain the insertion and deletion subroutines. Let $\mathcal{S}$ be a sequence of insertions and deletions of elements of an underlying ground set $\mathcal{V}$. First,

[3]Observe that z is already in R_1

suppose we would like to delete an element v. Observe that the set R_0 should contain all elements that have been inserted but not deleted so far. Thus, we remove v from R_0. Now, two cases can occur:

- *Light computation:* The first case is when $v \notin I_i$ for all $i \in [T]$. Then, we do a light computation by iterating through the levels $L_1, \cdots, L_T$, and for each level L_i, we delete v from R_i. Handling this light computation takes zero query complexity as we do not make any oracle query.
- *Heavy computation:* However, if there exists a level $i \in [T]$ where $e_i = v$, we then rebuild all levels $L_{i \le j \le T}$. To this end, we invoke the leveling algorithm for the level L_i to rebuild the levels $L_i, \cdots, L_T$. This computation is heavy, for which we need to make $O(|R_i| \log k)$ oracle queries.

When we invoke the leveling algorithm for a level L_i, it randomly permutes the elements R_i and iterates through this random permutation to compute I_ℓ, R_ℓ, and e_ℓ for $i \le \ell \le T$. We prove that this means for every level $L_{\ell \ge i}$, the promoting element e_ℓ that we picked is sampled uniformly at random from the set R_ℓ. This ensures that the probability that the sampled element e_ℓ being deleted is $\frac{1}{|R_\ell|}$. Therefore, the expected number of oracle queries to reconstruct the levels $[i, T]$ is at most $O(\frac{1}{|R_i|}|R_i| \log k) = O(\log k)$. Since there are at most $T \le k$ levels, by the linearity of expectation, the expected oracle queries that a deletion can incur is $O(k \log k)$.

Next, suppose we would like to insert an element v. First of all, the set R_0 should contain all elements that have been inserted but not deleted so far. Thus, we add v to R_0. Later, we iterate through levels $L_1, \cdots, L_{T+1}$, and for each level L_i, we check if v is a promoting element for the previous level or not. If it is not, we break the loop and exit the insertion subroutine. However, if v is a promoting element for the level L_{i-1}, we then add it to the set R_i and with probability $\frac{1}{|R_i|}$, we let e_i be v and invoke the leveling algorithm for the level L_{i+1} to rebuild the levels $L_{i+1}, \cdots, L_T$. The proof of correctness for insertion uses similar techniques to the proof for deletion.

Third building block: Relax the assumption of having OPT. Our dynamic algorithm assumes the optimal value $OPT = \max_{I^* \subseteq V : |I^*| \le k} f(I^*)$ is given as a parameter. However, in reality, the optimal value is not known in advance and it may change after every update. To remove this assumption, we use the well-known technique that has been also used in (Lattanzi et al., 2020). Indeed, we run parallel instances of our dynamic algorithm for different guesses of the optimal value OPT_t for the set V_t of elements that have been inserted till time t, but not deleted, such that for any time t, $\max_{I^* \subseteq V : |I^*| \le k} f(I^*) \in (OPT_t/(1+\epsilon), OPT_t]$ in one of the runs. These guesses are $(1+\epsilon)^i$ where $i \in \mathbb{Z}$. We apply each update on only $O(\log(k)/\epsilon)$ instances of our algorithm. See Section 3.4 for the details.

Submodular maximization problem under the matroid constraint

The dynamic algorithm that we develop for the matroid constraint has similar building blocks as the cardinality constraint, but it is more intricate. We outline these building blocks for the matroid constraint next.

First building block: Leveling algorithm. Let $\mathcal{M}(\mathcal{V}, \mathcal{I})$ be a matroid whose rank is $k = rank(\mathcal{M})$. We first assume that we have the maximum submodular value $MAX = \max_{e \in V} f(e)$. We relax this assumption later. Our leveling algorithm builds a set of levels $L_0, L_1, \cdots, L_T$, where T is a random variable guaranteed to satisfy $T = O(k \log(k/\epsilon))$. Every level L_i consists of three sets R_i, I_i, and I'_i, and an element e_i. For these sets, we have the following properties:

1. $V = R_0 \supseteq R_1 \supset \cdots \supset R_T \supset R_{T+1} = \emptyset$
2. The sets I_i are independent sets, i.e., $I_i \in \mathcal{I}$
3. Each I'_i is the union of all I_j for $j \le i$, i.e., $I'_i = \bigcup_{j \le i} I_j$

4. The sets I'_i *are not necessarily* independent
5. We report the set I_T as the solution

The illustration of our construction is similar to the one for the cardinality constraint and is shown in Figure 3.1. A key concept in our algorithm is again the notion of *promoting elements*. However, the definition of promoting elements for the matroid constraint is more complex than that of the cardinality constraint, and is inspired by the streaming algorithm of Chakrabarti and Kale (Chakrabarti and Kale, 2015). The complexity comes from the fact that adding an element e to an independent set, say I may preserve the independency of I or it may violate it by creating a circuit[4]. In Lemma 37, we prove that adding e to an independent set can create at most one circuit. Thus, we need to handle both cases when we define the notion of promoting elements.

Definition 3.1.2 (Promoting elements). *Let $L_{1\leq \ell\leq T}$ be a level. We call an element e, a promoting element for the level L_ℓ if*

- ***Property 1:*** $f(I'_\ell + e) - f(I'_\ell) \geq \frac{\epsilon}{10k} \cdot MAX$*, and*
- *One of the following properties hold:*
  - ***Property 2:*** *$I_\ell + e$ is independent set (i.e., $I_\ell + e \in \mathcal{I}$)* or
  - ***Property 3:*** *$I_\ell + e$ is not independent and the minimum weight element $\hat{e} = \arg\min_{e'\in C} w(e')$ of the set $C = \{e' \in I_\ell : I_\ell + e - e' \in \mathcal{I}\}$ satisfies $2w(\hat{e}) \leq f(I'_\ell + e) - f(I'_\ell)$.*

We next explain the leveling algorithm. We first initialize I_0 and I'_0 as empty sets and let R_0 be the set of existing elements V. We then let R_1 be all elements of the set R_0 that are promoting with respect to L_0. Observe that since I_0 and I'_0 are empty sets, an element is filtered out from level L_0 because of Property 1.

The leveling algorithm can be called for any level L_i and starting at that level, it builds the rest of levels $L_{i\leq j\leq T}$. Suppose our leveling algorithm is called for a level L_i for $i \geq 1$. Let $\ell = i$. We randomly permute the set R_i and let P be this random permutation. We iterate through the elements of P and upon seeing a new element e, we check if e is a promoting element for the level $L_{\ell-1}$.

- The first case occurs if e is a promoting element for the level $L_{\ell-1}$. Note that e is promoting if satisfies Property 1 and one of Properties 2 and 3.
  - If the element e satisfies Properties 1 and 2, we set $I_\ell = I_{\ell-1} + e$.
  - If e satisfies Properties 1 and 3, we set $I_\ell = I_{\ell-1} + e - \hat{e}$.

  In both cases, the resulting I_ℓ is an independent set in $\mathcal{I}$. We then fix the weight of e to be $w(e) = f(I'_{\ell-1} + e) - f(I'_{\ell-1})$. Later, we let $I'_\ell := I'_{\ell-1} + e$, and $e_\ell = e$. Similar to the leveling algorithm that we develop for the cardinality case, we now have the sets I_ℓ and I'_ℓ, and the element e_ℓ. However, the set R_ℓ is not complete yet, as some of its elements may come after e in the permutation P. We create the next level $L_{\ell+1}$ by setting $R_{\ell+1} = \emptyset$. We then proceed to the next element in P. Note that in this way, for all levels $L_{i<j\leq\ell}$, the sets R_j are not complete and they will be complete when we reach the end of the permutation P.

- The second case is when e is not a promoting element for $L_{\ell-1}$. Here, similar to the cardinality constraint, our goal is to perform the binary search to find the smallest $z \in [i, \ell - 1]$ so that e is promoting for the level L_{z-1}, but it is not promoting for the level L_z. ***Interestingly, we prove the monotonicity property holds for the matroid constraint***. (The proof of this subtle property is given in Section 3.3.1.) That is, we prove if e is promoting for a level L_{x-1}, it is promoting for all levels $L_{r\leq x-1}$

[4] A *circuit* in a matroid $\mathcal{M}$ is a minimal dependent subset of V—that is, a dependent set whose proper subsets are all independent.

and if e is not promoting for a level L_x, it is not promoting for all levels $L_{r\geq x}$. Thus, we can do the binary search to find the smallest $z \in [i, \ell - 1]$ so that e is promoting for the level L_{z-1}, but it is not promoting for the level L_z, which needs $O(\log(T)) = O(\log(k \log(k/\epsilon))$ steps of binary search. Once we find such z, we add e to all sets R_r for $i < r \leq z$. Unlike the previous case, however, we stay in the current level L_ℓ and proceed to the next element of P. Observe that adding e to all these sets may need $T = O(k \log(k/\epsilon))$ time, but we do not need to do oracle queries in order to add e to these sets.

Overview of the analysis: In order to prove the correctness of our leveling algorithm and compute its query complexity, we define two invariants; ***level*** and ***uniform*** invariants. The level invariant itself is a set of 5 invariants ***starter***, ***survivor***, ***independent***, ***weight***, and ***terminator***. We show that these invariants are fulfilled by the end of the leveling algorithm (in Section 3.3.2) and after every insertion and deletion of an element (in Section 3.3.3).

The level invariants assert that all elements that are added to R_i at a level L_i are promoting elements for the previous level. In other words, those elements of the set $R_{i-1} \backslash e_{i-1}$ that are not promoting will be filtered out and not be seen in R_i. Intuitively, this invariant provides us the approximation guarantee. The uniform invariant asserts that for every level $L_{i\in[T]}$, conditioned on the random sets $R_{j\leq i}$ and random elements $e_{j<i}$, the element e_i is chosen uniformly at random from the set R_i. That is, $\mathbb{P}\left[e_i = e | R_{j\leq i} \wedge e_{j<i}\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right]$[5]. Intuitively, this invariant provides us with the randomness that we need to fool the adversary in the (fully) dynamic model which in turn helps us to develop a dynamic algorithm for the submodular maximization problem under a matroid constraint.

The proof that the level and uniform invariants hold after every insertion and deletion is ***novel*** and ***subtle***. This proof is given in Sections 3.3.2 and 3.3.3. The technical part is to show that all promoting elements that are added to R_i at a level L_i (from the previous level) will be promoting after every update (i.e., insert or delete) and also, the sets I_i will remain independent after updates. In addition, we need to show that uniformly chosen elements e_i from survivor set R_i will be uniform after every update.

Now, we overview how we analyze the query complexity of our leveling algorithm. Checking if an element e is promoting for a level L_i can be done using $O(\log(k))$ oracle queries using a binary search argument. The proof is given in Section 3.3.1. The binary search that we perform in order to place an element e in the correct level requires $O(\log T)$ such promoting checks. Thus, if we initiate the leveling algorithm with a set R_i, our algorithm needs $O(|R_i| \log(k) \log(T))$ oracle queries to build the levels $L_i, \cdots, L_T$ for $T = O(k \log(k/\epsilon))$.

Second building block: Insertion and deletion of an element. Now, we explain how to maintain the independent set I_T upon insertions and deletions of elements. First, suppose we would like to delete an element e. We iterate through levels $L_1, \cdots, L_T$ and for each level L_i we delete e from R_i and we later check if e is the element e_i that we have picked for the level L_i. If this is the case, we then invoke the leveling algorithm for the set R_i to reset the levels $L_i, \cdots, L_T$. Since, the invocation of the leveling algorithm for the level L_i may initiate $O(|R_i| \log(k) \log(T))$ oracle queries (to build the levels $L_i, \cdots, L_T$) and since the element e_i is chosen uniformly at random from the set R_i and we iterate through levels $L_1, \cdots, L_T$, thus, the worst-case expected query complexity of deletion is $\sum_{i=1}^{T} \frac{1}{|R_i|} \cdot O(|R_i| \cdot \log(k) \cdot \log(T)) = O(k \log(k) \log^2(k/\epsilon))$.

Next, suppose we would like to insert an element e. First of all, the set R_0 should contain all elements that have been inserted but not deleted so far. Thus, we add e to R_0. Later, we iterate through levels $L_1, \cdots, L_T$ and for each level L_i, we check if e is a promoting element for that level or not. If it is not, we break the loop and exit the insertion subroutine. However, if e is indeed, a promoting element for the level L_i, we then add it to the set R_i and with probability $1/|R_i|$, we set $e_i = e$ and invoke the leveling algorithm (with the input index

[5]For an event A, we define $\mathbb{1}[A]$ as the *indicator function* of A. That is, $\mathbb{1}[A]$ is set to one if A holds and is set to zero otherwise.

$i+1$) to <u>reset</u> the subsequent levels $L_{i+1}, \cdots, L_T$. The query complexity of an insertion is proved similar to what we showed for a deletion.

The third block of our dynamic algorithm for the matroid constraint is to relax the assumption of knowing MAX. Relaxing this assumption is similar to what we did for the cardinality constraint. See Section 3.2 for the details.

3.1.3 Related Work

In this section, we state some known results for the submodular maximization problem under the matroid and cardinality constraints or some other related problems in the streaming, distributed, and dynamic models. In Table 3.1, we summarize the results in streaming and dynamic models for the submodular maximization problem under the matroid or cardinality constraint.

model	problem	approx.	query complexity	ref.
dynamic streaming model	cardinality	$2+\epsilon$	$O(\epsilon^{-1} dk \log(k))$	(Mirzasoleiman et al., 2017)
	cardinality	$2+\epsilon$	$O(dk \log^2(k) + d \log^3(k))$	(Kazemi et al., 2018)
	matroid	$5.582+\epsilon$	$O(k + \epsilon^{-2} d \log(k))$	(Duetting et al., 2022)
insertion-only dynamic model	matroid	$2+\epsilon$	$k^{\tilde{O}(1/\epsilon)}$	(Chen and Peng, 2022)
		$\frac{e}{e-1}+\epsilon$	$k^{\tilde{O}(1/\epsilon^2)} \cdot \log(n)$	(Chen and Peng, 2022)
fully dynamic model	matroid	$4+\epsilon$	$O(k \log(k) \log^3(k/\epsilon))$	this work
			$O(\frac{k^2}{\epsilon} \log(k) \log^2(n) \log^3(\frac{k}{\epsilon}))$	(Dütting et al., 2023)
	cardinality	$2+\epsilon$	$O(\epsilon^{-3} k^2 \log^4(n))$	(Monemizadeh, 2020)
			$O(\epsilon^{-4} \log^4(k) \log^2(n))$	(Lattanzi et al., 2023)
			$O(\text{Poly}(\log(n), \log(k), \frac{1}{\epsilon}))$	(Banihashem et al., 2023c)
			$O(k\epsilon^{-1} \log^2(k))$	this work
	cardinality lower-bound	$2-\epsilon$	$n^{\tilde{\Omega}(\epsilon)}/k^3$	(Chen and Peng, 2022)
		1.712	$\Omega(n/k^3)$	(Chen and Peng, 2022)

Table 3.1: Results for the submodular maximization subject to cardinality and matroid constraints. The lower bounds presented in (Chen and Peng, 2022) assume that we know the optimal submodular maximization value of the sub-sequence $\mathcal{S}_t$, where $\mathcal{S}_t$ is the set of elements that are inserted but not deleted from the beginning of the sequence $\mathcal{S}$ till any time t.

Known dynamic algorithms. The study of the submodular maximization in the dynamic model is initiated at NeurIPS 2020 based on two independent works. The first work is due to Lattanzi, Mitrovic, Norouzi-Fard, Tarnawski, and Zadimoghaddam (Lattanzi et al., 2020) who present a randomized dynamic algorithm that maintains an expected $(2+\epsilon)$-approximate solution of the maximum submodular (under the cardinality constraint k) of a dynamic sequence $\mathcal{S}$. The amortized expected query complexity of their algorithm is $O(\epsilon^{-11} \log^6(k) \log^2(n))$. The second work is due to Monemizadeh (Monemizadeh, 2020) who presents a randomized dynamic algorithm with approximation guarantee $(2+\epsilon)$. The amortized expected query complexity of his algorithm is $O(\epsilon^{-3} k^2 \log^5(n))$. The original version of the algorithm in Lattanzi et al. (Lattanzi et al., 2020) has some correctness issues, as pointed out by Banihashem, Biabani, Goudarzi, Hajiaghayi, Jabbarzade, and Monemizadeh (Banihashem et al., 2023c) at ICML 2023, who also provide an alternative algorithm for solving this problem with polylogarithmic update time. Those issues were also subsequently fixed by Lattanzi et al. (Lattanzi et al., 2023) by modifying their algorithm. The query complexity of their new algorithm is $O(\epsilon^{-4} \log^4(k) \log^2(n))$ per update. Peng's work at NeurIPS 2021 (Peng, 2021) focuses on the dynamic influence maximization problem, which is a white box dynamic submodular maximization problem. Work of

Banihashem, Biabani, Goudarzi, Hajiaghayi, Jabbarzade, and Monemizadeh (Banihashem et al., 2023b) at NeurIPS 2023 solves dynamic non-monotone submodular maximization under cardinality constraint k.

At STOC 2022, Chen and Peng (Chen and Peng, 2022) show two lower bounds for the submodular maximization in the dynamic model. Both of these lower bounds hold even if we know the optimal submodular maximization value of the sequence $\mathcal{S}$ at any time t. Their first lower bound shows that any randomized algorithm that achieves an approximation ratio of $2-\epsilon$ for dynamic submodular maximization under cardinality constraint k requires amortized query complexity $n^{\tilde{\Omega}(\epsilon)}/k^3$. They also prove a stronger result by showing that any randomized algorithm for dynamic submodular maximization under cardinality constraint k that obtains an approximation guarantee of 1.712 must have amortized query complexity at least $\Omega(n/k^3)$.

Chen and Peng (Chen and Peng, 2022) also studied the complexity of the submodular maximization under matroid constraint in the insertion-only dynamic model (a restricted version of the fully dynamic model where deletions are not allowed) and they developed two algorithms for this problem. The first algorithm maintains a $(2+\epsilon)$-approximate independent set of a matroid $\mathcal{M}(\mathcal{V},\mathcal{I})$ such that the expected number of oracle queries per insertion is $k^{\tilde{O}(1/\epsilon)}$. Their second algorithm is a $(\frac{e}{e-1}+\epsilon)$-approximation algorithm using an amortized query complexity of $k^{\tilde{O}(1/\epsilon^2)} \cdot \log(n)$, where k is the rank of $\mathcal{M}(\mathcal{V},\mathcal{I})$ and $n=|\mathcal{V}|$. However, these results do not work for the classical (fully) dynamic model, and they posed developing a dynamic algorithm for the submodular maximization problem under the matroid constraint in the (fully) dynamic model as an open problem.

And as discussed previously, the concurrent work of Dütting et al. (Dütting et al., 2023) at ICML 2023 provides an algorithm for dynamic submodular optimization under matroid constraint. Their algorithm has a $4+\epsilon$ approximation guarantee and $O(\frac{k^2}{\epsilon}\log(k)\log^2(n)\log^3(\frac{k}{\epsilon}))$ amortized expected query complexity.

Known (insertion-only) streaming algorithms. The first streaming algorithm for the submodular maximization under the cardinality constraint was developed by Badanidiyuru, Mirzasoleiman, Karbasi, and Krause (Badanidiyuru et al., 2014). In this seminal work, the authors developed a $(2+\epsilon)$-approximation algorithm for this problem using $O(k\epsilon^{-1}\log k)$ space. Later, Kazemi, Mitrovic, Zadimoghaddam, Lattanzi and Karbasi (Kazemi et al., 2019) proposed a space streaming algorithm for this problem that improves the space complexity down to $O(k\epsilon^{-1})$.

In a groundbreaking work, Chakrabarti and Kale (Chakrabarti and Kale, 2015) at IPCO'14 designed a streaming framework for submodular maximization problems under the matroid and matching constraints, as well as other constraints where independent sets are given either by a hypermatching constraint in p-hypergraphs or by the intersection of p matroids. In particular, their streaming framework gives a 4-approximation streaming algorithm for the submodular maximization under the matroid constraint using $O(k)$ space, where k is the rank of the underlying matroid $\mathcal{M}(\mathcal{V},\mathcal{I})$. The approximation ratio was recently improved to 3.15 by Feldman, Liu, Norouzi-Fard, Svensson, and Zenklusen (Feldman et al., 2021).

Later, Chekuri, Gupta, and Quanrud (Chekuri et al., 2015) developed one-pass streaming algorithms for (non-monotone) submodular maximization problems under p-matchoid[6] constraint as well as simpler streaming algorithms for the monotone case that have the same bounds as those of Chakrabarti and Kale (Chakrabarti and Kale, 2015). (These two works (Chakrabarti and Kale, 2015; Chekuri et al., 2015) were inspiring works for us as well).

[6] A set system $(N,\mathcal{I})$ is p-matchoid if there exists m matroids $(N_1,\mathcal{I}_1),\cdots,(N_m,\mathcal{I}_m)$ such that every element of N appears in the ground set of at most p of these matroids and $\mathcal{I}=\{S\subseteq 2^N : \forall_{1\le i\le m} S\cap N_i\in\mathcal{I}_i\}$.

Known streaming algorithms for related submodular problems. For non-monotone submodular objectives, the first streaming result was obtained by Buchbinder, Feldman, and Schwartz (Buchbinder et al., 2015), who designed a randomized streaming algorithm achieving an 11.197-approximation for the problem of maximizing a non-monotone submodular function subject to a single cardinality constraint.

Chekuri, Gupta, and Quanrud (Chekuri et al., 2015) further extended the work of Chakrabarti and Kale by developing $(5p + 2 + 1/p)/(1 - \epsilon)$-approximation algorithm for the non-monotone submodular maximization problems under p-matchoid constraints in the insertion-only streaming model. They also devised a deterministic streaming algorithm achieving $(9p + O(\sqrt{p}))/(1 - \epsilon)$-approximation for the same problem. Later, Mirzasoleiman, Jegelka, and Krause (Mirzasoleiman et al., 2018) designed a different deterministic algorithm for the same problem achieving an approximation ratio of $4p + 4\sqrt{p} + 1$.

At NeurIPS'18, Feldman, Karbasi and Kazemi (Feldman et al., 2018) improved these results for monotone and non-monotone submodular maximization under the p-matchoid constraint with respect to the space usage and approximation factor. As an example, their streaming algorithm for non-monotone submodular under p-matchoid achieves $4p + 2 - o(1)$-approximation that improves upon the randomized streaming algorithm proposed in (Chekuri et al., 2015).

Known dynamic streaming algorithms. Mirzasoleiman, Karbasi and Krause (Mirzasoleiman et al., 2017) and Kazemi, Zadimoghaddam and Karbasi (Kazemi et al., 2018) proposed dynamic streaming algorithms for the cardinality constraint. In particular, the authors in (Mirzasoleiman et al., 2017) developed a dynamic streaming algorithm that given a stream of inserts and deletes of elements of an underlying ground set $\mathcal{V}$, $(2 + \epsilon)$-approximates the submodular maximization under cardinality constraint using $O((dk\epsilon^{-1} \log k)^2)$ space and $O(dk\epsilon^{-1} \log k)$ average update time, where d is an upper-bound for the number of deletes that are allowed.

The follow-up paper (Kazemi et al., 2018) studies approximating submodular maximization under cardinality constraint in three models, (1) centralized model, (2) dynamic streaming where we are allowed to insert and delete (up to d) elements of an underlying ground set $\mathcal{V}$, and (3) distributed (MapReduce) model. In order to design a generic framework for all three models, they compute a coreset for submodular maximization under cardinality constraint. Their coreset has a size of $O(k \log k + d \log^2 k)$. Out of this coreset, we can extract a set S of size at most k whose $f(S)$ in expectation is at least 2-approximation of the optimal solution. The time to extract such a set S from the coreset is $O(dk \log^2 k + d \log^3 k)$.

The algorithms presented in (Mirzasoleiman et al., 2017) and (Kazemi et al., 2018) are dynamic streaming algorithms (not fully dynamic algorithms) whose time complexities depend on the number of deletions (Theorem 1 of the second reference). Therefore, their query complexities will be high if we recompute a solution after each insertion or deletion. Indeed, if the number of deletions is linear in terms of the maximum size of the ground set $\mathcal{V}$, it is in fact better to re-run the known leveling algorithms (say, (Nemhauser et al., 1978)) after every insertion and deletion. A similar result was recently obtained for the submodular maximization under the matroid constraint. At ICML 2022, Duetting, Fusco, Lattanzi, Norouzi-Fard, Zadimoghaddam (Duetting et al., 2022) presented a streaming $(5.582 + O(\epsilon))$-approximation algorithm for the deletion robust version of this problem, where the number of deletions is known to the algorithm, and they are revealed at the end of the stream. The space usage of their algorithm is $O(k + \epsilon^{-2} d \log(k))$, which is again linear if the number of deletions (d) is linear in terms of the maximum size of the ground set $\mathcal{V}$. This was subsequently improved by Zhang, Tatti, and Gionis (Zhang et al., 2022).

Known MapReduce algorithms. The first distributed algorithm for the cardinality constrained submodular maximization was due to Mirrokni and Zadimoghaddam (Mirrokni and Zadimoghaddam, 2015) who gave a 3.70-approximation in 2 rounds without duplication and a 1.834-approximation with significant duplication of the ground set (each element being sent to $\Theta(\frac{1}{\epsilon}\log(\frac{1}{\epsilon}))$ machines). Later, Barbosa, Ene, Nguyen and Ward (da Ponte Barbosa et al., 2016) achieves a $(2+\epsilon)$-approximation in 2 rounds and was the first to achieve a $(\frac{e}{e-1}+\epsilon)$ approximation in $O(\frac{1}{\epsilon})$ rounds. Both algorithms require $\Omega(\frac{1}{\epsilon})$ duplication. (da Ponte Barbosa et al., 2016) mentions that without duplication, the two algorithms could be implemented in $O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon}))$ and $O(\frac{1}{\epsilon^2})$ rounds, respectively.

In a subsequent work, Liu and Vondrak (Liu and Vondrák, 2019) develop a simple thresholding algorithm that with one random partitioning of the dataset (no duplication) achieves the following: In 2 rounds of MapReduce, they obtain a $(2+\epsilon)$-approximation and in $2/\epsilon$ rounds, they achieve $(\frac{e}{e-1}-\epsilon)$-approximation. Their algorithm is inspired by the streaming algorithms that are presented in (Kumar et al., 2015) and (McGregor and Vu, 2019). It is also similar to the algorithm of Assadi and Khanna (Assadi and Khanna, 2018) who study the communication complexity of the maximum coverage problem.

3.2 Dynamic algorithm for submodular matroid maximization

In this section, we present our dynamic algorithm for the submodular maximization problem under the matroid constraint. The pseudocode of our algorithm is provided in Algorithms 7 and 8. The overview of our dynamic algorithm is given in Section 3.1.2 "Our contribution".

Promoting Elements As we explained in Section 3.1.2 "Our contribution", a key concept in our algorithm is the notion of *promoting elements*.

Definition 3.2.1 (Promoting elements). *Let $L_{1\le\ell\le T}$ be a level. We call an element e, a promoting element for the level L_j if*
- ***Property 1:*** $f(I'_\ell + e) - f(I'_\ell) \ge \frac{\epsilon}{10k}\cdot MAX$, *and*
- *One of the following properties hold:*
  - ***Property 2:*** *$I_\ell + e$ is independent set (i.e., $I_\ell + e \in \mathcal{I}$)* or
  - ***Property 3:*** *$I_\ell + e$ is not independent and the minimum weight element $\hat{e} = \arg\min_{e'\in C} w(e')$ of the set $C = \{e' \in I_\ell : I_\ell + e - e' \in \mathcal{I}\}$ satisfies $2w(\hat{e}) \le f(I'_\ell + e) - f(I'_\ell)$.*

We define the function $\textsc{Promote}(I_\ell, I'_\ell, e, w[I_\ell])$ for an element $e \in V$ with respect to the level L_ℓ which

- returns $\emptyset$ if properties 1 and 2 hold;
- returns $\hat{e}$ if properties 1 and 3 hold;
- returns $\textsc{Fail}$ otherwise.

Subroutine $\textsc{Promote}$ in Algorithm 7 implements this function. This subroutine checks if an element $e \in V$ is a promoting element for a level L_ℓ or not. In case that e is a promoting element for L_ℓ, the subroutine $\textsc{Promote}$ finds an element e' (if it exists) that satisfies Property 3 of definition 3.2.1 and replaces it by e.

Our leveling algorithm consists of three subroutines, $\textsc{Init}$, $\textsc{MatroidConstructLevel}$, and $\textsc{Promote}$. We explained in above Subroutine $\textsc{Promote}$. In Subroutine $\textsc{Init}$, we first initialize I_0 and I'_0 as empty set and set R_0 to the ground set V. We then let R_1 be all elements of the set R_0 that are promoting with respect to L_0. Observe that since I_0 and I'_0 are empty sets, if an element e filtered out from the level L_0, i.e., $e \in L_0$ but $e \notin L_1$, then e was filtered because of Property 1. Finally, we invoke $\textsc{MatroidConstructLevel}$ for the level L_1, to build all the remaining levels. Subroutine $\textsc{MatroidConstructLevel}$ implements our leveling algorithm that we gave an overview of it in Section 3.1.2 "Our contribution".

Algorithm 7 MatroidLeveling($\mathcal{M}(\mathcal{V}, \mathcal{I}), MAX$)

```
function Init(V)
    I_0 ← ∅,   I'_0 ← ∅,   R_0 ← V
    R_1 ← {e ∈ R_0 : Promote(I_0, I'_0, e, w[I_0]) ≠ Fail}
    Invoke MatroidConstructLevel(i = 1)

function MatroidConstructLevel(i)
    Let P be a random permutation of elements of R_i and ℓ ← i
    for e in P do
        if Promote(I_{ℓ−1}, I'_{ℓ−1}, e, w[I_{ℓ−1}]) ≠ Fail then
            y ← Promote(I_{ℓ−1}, I'_{ℓ−1}, e, w[I_{ℓ−1}])) and z ← ℓ
            Fix the weight w(e) ← f(I'_{ℓ−1} + e) − f(I'_{ℓ−1}), and set the element e_ℓ ← e
            Let I_ℓ ← (I_{ℓ−1} + e)\y,   I'_ℓ ← I'_{ℓ−1} + e,   R_{ℓ+1} ← ∅,   and then ℓ ← ℓ + 1
        else
            Run binary search to find the lowest z ∈ [i, ℓ − 1] such that Promote(I_z, I'_z, e, w[I_z]) = Fail
        for r ← i + 1 to z do
            R_r ← R_r + e
    return T ← ℓ − 1 which is the final ℓ that the for-loop above returns subtracted by one

function Promote(I, I', e, w[I])
    if f(I' ∪ {e}) − f(I') ∉ [ε/(10k) · MAX, MAX] then
        return Fail
    if I + e ∈ 𝓘 then
        return ∅
    C ← {e' ∈ I : I + e − e' ∈ 𝓘} and let ê ← arg min_{e'∈C} w(e')
    if 2 · w(ê) ≤ f(I' + e) − f(I') then
        return {ê}
    else
        return Fail
```

Relaxing MAX assumption. Our dynamic algorithm assumes the maximum value $\max_{e \in V} f(e)$ is given as a parameter. However, in reality, the maximum value is not known in advance and it may change after every insertion or deletion. To remove this assumption, we run parallel instances of our dynamic algorithm for different guesses of the maximum value MAX_t at any time t of the sequence $\mathcal{S}_t$, such that $\max_{e \in V_t} f(e) \in (MAX_t/2, MAX_t]$ in one of the runs. Recall that V_t is the set of elements that have been inserted but not deleted from the beginning of the sequence till time t. These guesses that we take are 2^i where $i \in \mathbb{Z}$. If ρ is the ratio between the maximum and minimum non-zero possible value of an element in V, then the number of parallel instances of our algorithm will be $O(\log \rho)$. This incurs an extra $O(\log \rho)$-factor in the query complexity of our dynamic algorithm.

Next, we show how to replace this extra factor with an extra factor of $O(\log (k/\epsilon))$ which is independent of ρ. We use the well-known technique that has been also used in (Lattanzi et al., 2020). In particular, for every element e, we add it to those instances i for which we have $\frac{\epsilon}{10k} \cdot 2^i \leq f(e) \leq 2^i$. The reason is if the maximum value of V_t is within the range $(2^{i-1}, 2^i]$ and $f(e) > 2^i$, then $f(e)$ is greater than the maximum value and can safely be ignored for the instance i that corresponds to the guess 2^i. On the other hand, we can safely ignore all elements e whose $f(e) < \frac{\epsilon}{10k} \cdot 2^i$, since these elements will never be a promoting element in the run with $MAX = 2^i$. This essentially means that every element e is added to at most $O(\log (k/\epsilon))$

Algorithm 8 MatroidUpdates($\mathcal{M}(\mathcal{V}, \mathcal{I})$, MAX)

function Delete(v)
    $R_0 \leftarrow R_0 - v$
    for $i \leftarrow 1$ **to** T **do**
        if $v \notin R_i$ **then**
            break
        $R_i \leftarrow R_i - v$
        if $e_i = v$ **then**
            Invoke MatroidConstructLevel(i)
            break

---

function Insert(v)
    $R_0 \leftarrow R_0 + v$.
    for $i \leftarrow 1$ **to** $T + 1$ **do**
        if Promote($I_{i-1}, I'_{i-1}, v, w[I_{i-1}]$) = Fail **then**
            break
        $R_i \leftarrow R_i + v$.
        Let $p_i = 1$ with probability $\frac{1}{|R_i|}$, and otherwise $p_i = 0$
        if $p_i = 1$ **then**
            $e_i \leftarrow v, \quad w(e_i) \leftarrow f(I'_{i-1} + v) - f(I'_{i-1}), \quad y \leftarrow$ Promote($I_{i-1}, I'_{i-1}, v, w[I_{i-1}]$)
            $I_i \leftarrow I_{i-1} + v - y, \quad I'_i \leftarrow I'_i + v$
            $R_{i+1} = \{e' \in R_i :$ Promote($I_i, I'_i, e', w[I_i]$) $\neq$ Fail$\}$
            MatroidConstructLevel($i + 1$)
            break

Algorithm 9 Unknown MAX

Let $\mathcal{A}_i$ be the instance of our dynamic algorithm, for which $MAX = 2^i$

---

function UpdateWithoutKnowingMAX(e)
    for each $i \in \left[\lceil \log f(e) \rceil, \left\lfloor \log\left(\frac{10k}{\epsilon} \cdot f(e)\right)\right\rfloor\right]$ **do** ▷ $\frac{\epsilon}{10k} \cdot 2^i \leq f(e) \leq 2^i$
        Invoke Update(e) for instance $\mathcal{A}_i$

parallel instances. Thus, after every insertion or deletion, we need to update only $O(\log(k/\epsilon))$ instances of our dynamic algorithm.

3.3 Analysis of dynamic algorithm for submodular matroid

In this section, we prove the correctness of our MatroidConstructLevel, Insert, and Delete algorithms. We will also compute the query complexity of each one of them. To analyze our randomized algorithm, for any variable x in our pseudo-code, we use $\mathbf{x}$ to denote it as a random variable and use x itself to denote its value in an execution. The most frequently used random variables in our analysis are as follows:

- We denote by $\mathbf{e}_i$ the random variable corresponding to the element e_i picked at level L_i.
- We denote by $\mathbf{R}_i$ the random variable that corresponds to the set R_i.
- The random variable $\mathbf{T}$ corresponds to T, which is the index of the last non-empty level created. Indeed, for a level L_i to be existent and non-empty, $\mathbf{T} \geq i$ should hold.
- We define $H_i = (e_1, \ldots, e_{i-1}, R_0, \ldots, R_i)$ as *the partial configuration up to the level L_i*. Note that R_i is included in this definition, while e_i is not. $\mathbf{H}_i := (\mathbf{e}_1, \ldots, \mathbf{e}_{i-1}, \mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_i)$ is the random variable corresponding to the partial configuration H_i.

We break the analysis of our algorithm into a few steps.

Step 1: Analysis of binary search. In the first step, we prove that the binary search that we use to speed up the process of finding the right levels for non-promoting elements works. Indeed, we prove that if $e \in V$ is a promoting element for a level L_{z-1}, it is promoting for all levels $L_{r \leq z-1}$ and if e is not promoting for the level L_z, it is not promoting for all levels $L_{r \geq z}$. Therefore, because of this monotonicity property, we can do a binary search to find the smallest $z \in [i, \ell - 1]$ so that e is promoting for the level L_{z-1}, but it is not promoting for the level L_z. Additionally, we show that checking whether e is promoting for a level L_z can be done with $O(\log(k))$ queries using a binary search argument.

Step 2: Maintaining invariants. We define six invariants, and we show that these invariants *hold* when Init is run, and our whole data structure gets built, *and are preserved* after every insertion and deletion of an element.

Invariants:

1. **Level invariants.**
   - 1.1 **Starter.** $R_0 = V$ and $I_0 = I'_0 = \emptyset$
   - 1.2 **Survivor.** For $1 \leq i \leq T + 1$, $R_i = \{e \in R_{i-1} - e_{i-1} : \textsc{Promote}(I_{i-1}, I'_{i-1}, e, w[I_{i-1}]) \neq \textsc{Fail}\}$
   - 1.3 **Independent.** For $1 \leq i \leq T$, $I_i = I_{i-1} + e_i - \textsc{Promote}(I_{i-1}, I'_{i-1}, e_i, w[I_{i-1}])$, and $I'_i = \cup_{j \leq i} I_j$
   - 1.4 **Weight.** For $1 \leq i \leq T$, $e_i \in R_i$ and $w(e_i) = f(I'_{i-1} + e_i) - f(I'_{i-1})$
   - 1.5 **Terminator.** $R_{T+1} = \emptyset$
2. **Uniform invariant.** For all $i \geq 1$, conditioned on the random variables $\mathbf{T}$ and $\mathbf{H}_i$, the element e_i is chosen uniformly at random from the set R_i. That is, $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i \text{ and } \mathbf{H}_i = H_i] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$.

The survivor invariant says that all elements that are added to R_i at a level L_i are promoting elements for that level. In other words, those elements of the set $R_{i-1} - e_{i-1}$ that are not promoting will be filtered out and not be seen in R_i. The terminator invariant shows that the recursive construction of levels stops when the survivor set becomes empty. The independent invariant shows that the sets I_i are independent sets of the matroid $\mathcal{M}(\mathcal{V}, \mathcal{I})$, and I'_i is equal to the union of $I_1, \ldots, I_i$. The weight invariant explains that the weight of every element e_i added to the independent set I_i is defined with respect to the marginal gain it adds to the set I'_{i-1}, and it is fixed later on. Intuitively, the level invariants provide the approximation guarantee.

The uniform invariant asserts that, conditioned on $\mathbf{T} \geq i$ which means that L_i is a non-empty level and $\mathbf{H}_i = H_i$, which implies that $e_1, \cdots, e_{i-1}$ are chosen and R_i is well-defined, the element $\mathbf{e}_i$ is uniform random variable over the set R_i. That is, $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i \text{ and } \mathbf{H}_i = H_i] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$. Intuitively, this invariant provides us with the randomness that we need to fool the adversary in the (fully) dynamic model which in turn helps us to develop a dynamic algorithm for the submodular matroid maximization.

Step 3: Query complexity. In the third part of the proof, we show that if the **uniform** invariant holds, we can bound the worst-case expected query complexity of the leveling algorithm, and later, the worst-case expected query complexity of the insertion and deletion operations.

Step 4: Approximation guarantee. Finally, in the last step of the proof, we show that if the survivor, terminator, independent and weights invariants hold, we can report an independent set $I_T \in \mathcal{I}$ whose submodular value is an $(4 + \epsilon)$-approximation of the optimal value.

3.3.1 Monotone property and binary search argument

Recall that we defined the function $\textsc{Promote}(I_j, I'_j, e, w[I_j])$ for an element $e \in V$ with respect to the level L_j which

- returns $\emptyset$ if properties 1 and 2 hold;
- returns $\hat{e}$ if properties 1 and 3 hold;
- returns $\textsc{Fail}$ otherwise.

Here properties 1, 2, and 3 are the ones that we defined in Definition 3.2.1. Recall that if the first two cases occur, we say that e is a promoting element with respect to the level L_j. In this section, we consider a boolean version of the function $\textsc{Promote}(I_j, I'_j, e, w[I_j])$. We denote this boolean function by $\textsc{BoolPromote}(e, L_j)$ which is $True$ if either of the first two cases happen. That is, when $\textsc{Promote}(I_j, I'_j, e, w[I_j])$ returns either $\emptyset$ or $\hat{e}$; otherwise, $\textsc{BoolPromote}(e, L_j)$ returns $False$.

Lemma 38. *Let L_j be an arbitrary level of the Algorithm* $\textsc{DynamicMatroid}$*, where $1 \le j \le T$. Let e be an arbitrary element of the ground set. If* $\textsc{BoolPromote}(e, L_{j-1})$ *returns $False$, then* $\textsc{BoolPromote}(e, L_j)$ *returns $False$.*

Suppose for the moment that this lemma is correct. Then by applying a simple induction, we can show the function $\textsc{BoolPromote}(e, L_j)$ is monotone which means that the function $\textsc{Promote}(I_j, I'_j, e, w[I_j])$ is monotone. Thus, for every arbitrary element e, it is possible to perform a binary search on the interval $[i, \ell - 1]$ to find the smallest $z \in [i, \ell - 1]$ such that $\textsc{BoolPromote}(e, L_{z-1}) = True$ and $\textsc{BoolPromote}(e, L_z) = False$.

Now we prove the lemma.

Proof. Suppose that $\textsc{BoolPromote}(e, L_{j-1})$ returns $False$. It means that either property 1 or both properties 2 and 3 do not hold. If property 1 does not hold, then $f(I'_{j-1} + e) - f(I'_{j-1}) < \frac{\varepsilon}{10k} \cdot MAX$. Since $I'_{j-1} \subseteq I'_j$ and f is submodular, we have $f(I'_j + e) - f(I'_j) \le f(I'_{j-1} + e) - f(I'_{j-1}) < \frac{\varepsilon}{10k} \cdot MAX$, which means that $\textsc{BoolPromote}(e, L_j) = False$.

For the remainder of the proof, we assume that both properties 2 and 3 do not hold. This means that $I_{j-1} + e$ is not independent, and for the minimum weight element $\hat{e} := \arg\min_{e' \in C} w(e')$ of the set $C := \{e' \in I_{j-1} : I_{j-1} + e - e' \in \mathcal{I}\}$, we have $f(I'_{j-1} + e) - f(I'_{j-1}) < 2w(\hat{e})$. Now, let us consider level L_j. There are two cases to consider: $|I_j| = |I_{j-1}| + 1$ and $|I_j| = |I_{j-1}|$.

For the first case, we have $I_j = I_{j-1} + e_j$. Thus, we have $I_{j-1} \subseteq I_j$. Now, let us consider the element e. For the set $C := \{e' \in I_{j-1} : I_{j-1} + e - e' \in \mathcal{I}\}$, we have $C \subseteq I_{j-1} \subseteq I_j$ which means that the circuit (dependent set) $C + e \subseteq I_j + e$. Note that since I_j is an independent set, we also know that $C + e$ is the only circuit of $I_j + e$ according to Lemma 37. Recall that $f(I'_{j-1} + e) - f(I'_{j-1}) < 2w(\hat{e})$ where $\hat{e}$ is the minimum weight element $\hat{e} := \arg\min_{e' \in C} w(e')$. Since $I'_{j-1} \subseteq I'_j$, then by the submodularity of the function f, we have

$$f(I'_j + e) - f(I'_j) \le f(I'_{j-1} + e) - f(I'_{j-1}) < 2w(e') \ .$$

Hence, BoolPromote(e, L_j) returns $False$ in this case.

For the second case, we have $I_j = I_{j-1} - \hat{e}_j + e_j$. This means that $I_{j-1} + e_j$ is not an independent set. Thus, the set $C' := \{e' \in I_{j-1} : I_{j-1} + e_j - e' \in \mathcal{I}\}$ has a minimum weight element $\hat{e}_j$ that is replaced by e_j to obtain the independent set I_j.

Now, we consider two subcases. Case (I) is $\hat{e}_j \in C$ and Case (II) is $\hat{e}_j \notin C$.

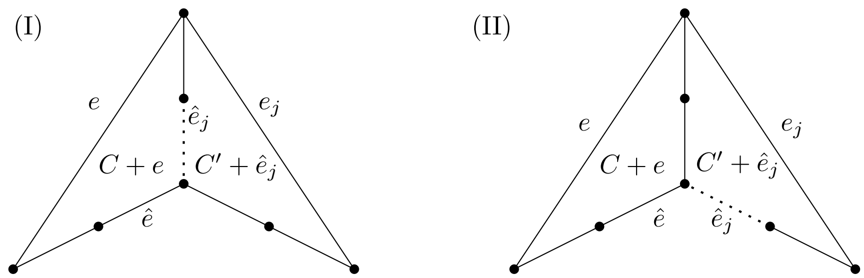

Figure 3.2: Illustration of $I_j + e$ for the subcases (I) and (II) in Lemma 38. $C + e$ and $C' + \hat{e}_j$ are circuits. Case (I) is $\hat{e}_j \in C$. Then there is a circuit $C'' \subseteq (C + e) \cup (C' + e_j) - \hat{e}_j$. Case (II) is $\hat{e}_j \notin C$. Then $C + e \subseteq I_j + e$.

First, we consider Case (I) which is $\hat{e}_j \in C$. Thus, $\hat{e}_j \in C \cap C'$. Note that $C \subseteq I_{j-1}$ and $e_j \notin I_{j-1}$, so $e_j \notin C$, which implies that $(C+e)$ and $(C'+e_j)$ are two different circuits. Since $\hat{e}_j \in (C+e) \cap (C'+e_j)$, there is a circuit $C'' \subseteq (C+e) \cup (C'+e_j) - \hat{e}_j$ according to Lemma 36. In addition, $(C+e) \cup (C'+e_j) \subseteq I_{j-1} + e + e_j = I_j + e + \hat{e}_j$. Since $\hat{e}_j \notin C''$, we then have $C'' \subseteq I_j + e$. Recall that $\hat{e}$ and $\hat{e}_j$ are the minimum weight element in C and C', respectively. Since $\hat{e}_j \in C$, then $w(\hat{e}) \leq w(\hat{e}_j)$.

Let e'' be the minimum weight element in $C'' - e$. Since $C'' \subseteq (C + e) \cup (C' + e_j)$ and $w(e_j) > w(\hat{e}_j)$, we have $w(e'') \geq \min(w(\hat{e}), w(\hat{e}_j)) = w(\hat{e})$. Since $I'_{j-1} \subseteq I'_j$ and f is a submodular function, we obtain the following:

$$f(I'_j + e) - f(I'_j) \leq f(I'_{j-1} + e) - f(I'_{j-1}) < 2 \cdot w(\hat{e}) \leq 2 \cdot w(e'') \ .$$

This essentially means that $I_j + e$ is not independent as $C'' \subseteq I_j + e$, and $f(I'_j + e) - f(I'_j) < 2 \cdot w(e'')$, where e'' is the minimum weight element in $C'' - e$. Thus, BoolPromote(e, L_j) returns $False$.

Finally, we consider Case (II) which is $\hat{e}_j \notin C$. In this case, $C + e \subseteq I_{j-1} - \hat{e}_j + e \subseteq I_j + e$. Note that $C + e$ is the only circuit of $I_j + e$ by Lemma 37. Recall $f(I'_{j-1} + e) - f(I'_{j-1}) < 2 \cdot w(\hat{e})$ and $I'_{j-1} \subseteq I'_j$. Hence, by the submodularity of f we have $f(I'_j + e) - f(I'_j) \leq f(I'_{j-1} + e) - f(I'_{j-1}) < 2 \cdot w(\hat{e})$. Thus, BoolPromote$(e, L_j)$ returns $False$ proving the lemma. T

Lemma 39. *Let $I \in \mathcal{I}$ be an independent set and e be an element such that $I \cup \{e\} \notin \mathcal{I}$. Define $C := \{e' : I + e - e' \in \mathcal{I}\}$. Let $w : I \cup \{e\} \to \mathbb{R}^{\geq 0}$ be an arbitrary weight function and define $\hat{e} := \arg\min_{e' \in C} w(e')$. The element $\hat{e}$ can be found using at most $O(\log(|I|))$ oracle queries.*

Proof. Let $e_1, \ldots, e_{|I|+1}$ denote an ordering of $I \cup \{e\}$ such that $w(e_1) \geq w(e_2) \cdots \geq w(e_{|I|+1})$. Let i denote the smallest index such that $\{e_1, \ldots, e_i\} \notin \mathcal{I}$. Such an index exists because $\{e_1, \ldots, e_{|I|+1}\} = I \cup \{e\} \notin \mathcal{I}$. We claim that $\hat{e} = e_i$. We note that the element e_i can be found using a binary search over $[|I| + 1]$ because for any j, if $\{e_1, \ldots, e_j\} \notin \mathcal{I}$, then $\{e_1, \ldots, e_{j+1}\} \notin \mathcal{I}$ as well.

To prove this, we first claim that $e_i \in C$. To see why this holds, we first observe that since $\{e_1, \ldots, e_{i-1}\}$ is independent but $\{e_1, \ldots, e_i\}$ is not, we have $e_i \in$ Span$(\{e_1, \ldots, e_{i-1}\}) \subseteq$ Span$(I + e - e_i)$. Therefore, since

$e_j \in \textsc{Span}(I + e - e_i)$ for all $j \neq i$, we have $I + e \subseteq \textsc{Span}(I + e - e_i)$, which implies

$$rank(I + e - e_i) \geq rank(I + e) \geq rank(I) = |I| = |I + e - e_i|,$$

which implies $I + e - e_i \in \mathcal{I}$ as claimed.

We need to show that for any $e' \in C$, we have $w(e_i) \leq w(e')$. Assume for contradiction that $w(e') < w(e_i)$. It follows that $e' = e_j$ for some $j > i$. By definition of C, we must have $I + e - e_j \in \mathcal{I}$, which implies $\{e_1, \ldots, e_{j-1}\} \in \mathcal{I}$. Since $i < j$, this further implies $\{e_1, \ldots, e_i\} \in \mathcal{I}$, which is not possible by definition of i. T

3.3.2 Correctness of invariants after MatroidConstructLevel is called

In this section, we focus on the previously defined invariants at the end of the execution of the algorithm $\textsc{MatroidConstructLevel}(j)$. We first provide a definition explaining what we mean by stating that level invariants partially hold.

Definition 3.3.1. *For $j \geq 1$, we say that* the level invariants partially hold for the first j levels *if the followings hold.*

1. ***Starter.*** *$R_0 = V$ and $I_0 = I'_0 = \emptyset$*
2. ***Survivor.*** *For $1 \leq i \leq j$, $R_i = \{e \in R_{i-1} - e_{i-1} : \textsc{Promote}(I_{i-1}, I'_{i-1}, e, w[I_{i-1}]) \neq \textsc{Fail}\}$*
3. ***Independent.*** *For $1 \leq i \leq j - 1$, $I_i = I_{i-1} + e_i - \textsc{Promote}(I_{i-1}, I'_{i-1}, e_i, w[I_{i-1}])$, and $I'_i = \cup_{j \leq i} I_j$*
4. ***Weight.*** *For $1 \leq i \leq j - 1$, $e_i \in R_i$ and $w(e_i) = f(I'_{i-1} + e_i) - f(I'_{i-1})$*

Next, we have the following theorem, in which we ensure that all level invariants hold after the execution of $\textsc{MatroidConstructLevel}(j)$ given the assumption that level invariants partially hold for the first j levels when $\textsc{MatroidConstructLevel}(j)$ is invoked. This theorem will be of use in the following sections in showing that level invariants hold after each update. It can also independently prove that level invariants hold after $\textsc{Init}$ is run.

Theorem 40. *If before calling* $\textsc{MatroidConstructLevel}(j)$*, the level invariants partially hold for the first j levels, then after the execution of* $\textsc{MatroidConstructLevel}(j)$*, level invariants fully hold.*

Proof. Considering that the starter invariant holds by the assumption of the theorem and needs no further proof, we have broken the proof of this theorem into four lemmas, each considering one of the survivor, independent, weight, and terminator invariants separately. T

Finally, we prove a lemma that says knowing that the level invariants are going to hold after the execution of $\textsc{MatroidConstructLevel}(j)$, a modified version of uniform invariant will also hold after this execution. We use this lemma in the next sections to prove that the uniform invariant holds after each update. It also shows that uniform invariant holds after $\textsc{Init}$ is run since the previous theorem had proved that level invariants would hold.

Lemma 41 (Uniform invariant)**.** *If* $\textsc{MatroidConstructLevel}(j)$ *is invoked and the level invariants are going to hold after its execution, then for any $i \geq j$ we have $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i \text{ and } \mathbf{H}_i = H_i] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$.*

Proof. At the beginning of $\textsc{MatroidConstructLevel}(j)$, we take a random permutation of elements in R_j. Making a random permutation is equivalent to sampling all elements without replacement. In other words, instead of fixing a random permutation P of R_j and iterating through P in Line 7, we can repeatedly sample a random element e from the unseen elements of R_j until we have seen all of the elements. Hence, in the following proof, we assume our algorithm uses sampling without replacement.

Given this view, we make the following claims.

Observation 1. e_i is the first element of R_i seen in the permutation.
This is because before e_i is seen, the value of ℓ is at most i. It is also clear from the algorithm that when an element e is considered, it can only be added to sets R_x for $x \leq \ell$, both when $y = \text{Fail}$ and when $y \neq \text{Fail}$. Furthermore, e can only be added to R_ℓ if $e = e_\ell$. Therefore, no element can be added to R_i before e_i is seen.

Observation 2. Once $e_1, \ldots, e_{i-1}$ have been seen, the set R_i is uniquely determined.
Note that R_i is uniquely determined *even though the algorithm has not observed its elements yet*. This is because regardless of the randomness of MatroidConstructLevel(j), the level invariants will hold after its execution. This implies that the content of the set R_i only depends on the value of $(\mathbf{e}_1, \ldots, \mathbf{e}_{i-1})$, which is not going to change after it is set to be equal to $(e_1, \ldots, e_{i-1})$.

Let the random variable $\mathbf{M}_i$ denote the sequence of elements that our algorithm observes until setting $\mathbf{e}_{i-1}$ to be e_{i-1}, including e_{i-1} itself. In other words, if e_{i-1} is the x-th element of the permutation P, M_i is the first x elements of P.
Based on the above facts, conditioned on $\mathbf{M}_i = M_i$, **(a)** the value of $\mathbf{R}_i$, or in other words R_i is uniquely determined. **(b)** e_i is going to be the first element of R_i that the algorithm observes. Therefore, since we assumed that the algorithm uses sampling without replacement, $\mathbf{e}_i$ is going to have a uniform distribution over R_i, i.e.,

$$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{M}_i = M_i] = \frac{1}{|R_i|} \mathbb{1}[e \in R_i] \quad .$$

By the law of total probability, we have

$$\mathbb{P}[\mathbf{e}_i = e_i | \mathbf{T} \geq i, \mathbf{H}_i = H_i] = \mathbb{E}_{M_i}[\mathbb{P}[\mathbf{e}_i = e_i | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{M}_i = M_i]] \quad ,$$

where the expectation is taken over all M_i with positive probability.

Also, note that knowing that $\mathbf{M}_i = M_i$ uniquely determines the value of $\mathbf{H}_i$ as well. This is because M_i includes $(e_1, \ldots, e_{i-1})$ and, with similar reasoning to what we used for Observation 2, we can say that $R_1, \ldots, R_i$ are uniquely determined by $(e_1, \ldots, e_{i-1})$.

Since we are only considering M_i with positive probability, and $\mathbf{H}_i$ is a function of $\mathbf{M}_i$ given the discussion above, all the forms of M_i that we consider in our expectation are the ones that imply $\mathbf{H}_i = H_i$. Therefore, we can drop the condition $\mathbf{H}_i = H_i$ from the condition $\mathbf{H}_i = H_i, \mathbf{M}_i = M_i$, which implies

$$\mathbb{P}[\mathbf{e}_i = e_i | \mathbf{T} \geq i, \mathbf{H}_i = H_i] = \mathbb{E}_{M_i}[\mathbb{P}[\mathbf{e}_i = e_i | \mathbf{T} \geq i, \mathbf{M}_i = M_i]] = \mathbb{E}_{M_i}\left[\frac{1}{|R_i|} \mathbb{1}[e_i \in R_i]\right] = \frac{1}{|R_i|} \mathbb{1}[e_i \in R_i] \quad ,$$

as claimed. T

3.3.3 Correctness of invariants after an update

In our dynamic model, we consider a sequence $\mathcal{S}$ of updates to the underlying ground set V where at time t of the sequence $\mathcal{S}$, we observe an update which can be the deletion of an element $e \in V$ or insertion of an element $e \in V$. We assume that an element e can be deleted at time t, if it is in V meaning that it was not deleted after the last time it was inserted.

We use several random variables for our analysis, including $\mathbf{e}_i$, $\mathbf{R}_i$, $\mathbf{T}$, and $\mathbf{H}_i$. Upon observing an update at time t, we should distinguish between each of these random variables and their corresponding values before and after the update. To do so, we use the notations $\mathbf{Y}^-$ and Y^- to denote a random variable and its value before time t when e is either deleted or inserted, and we keep using $\mathbf{Y}$ and Y to denote them at the current time after the execution of update. As an example, $\mathbf{H}_i^- := (\mathbf{e}_1^-, \ldots, \mathbf{e}_{i-1}^-, \mathbf{R}_0^-, \mathbf{R}_1^-, \ldots, \mathbf{R}_i^-)$ is the random variable that corresponds to the partial configuration $H_i^- = (e_1^-, \ldots, e_{i-1}^-, R_0^-, \ldots, R_i^-)$.

Correctness of invariants after every insertion

We first consider the case when the update at time t of the sequence $\mathcal{S}$ is an insertion of an element v. In this section, we prove the following theorem.

Theorem 42. *If before the insertion of an element v, the level invariants and uniform invariant hold, then they also hold after the execution of* Insert(v).

We break the proof of this theorem into Lemmas 43 and 44. Note that we use Lemma 43 in the proof of Lemma 44. However, Lemma 44 would not be used in the proof of 43, so no loop would form when combined to prove the theorem.

Lemma 43 (Level invariants)**.** *If before the insertion of an element v the level invariants (i.e., starter, survivor, independent, weight, and terminator) hold, then they also hold after the execution of* Insert(v).

Lemma 44 (Uniform invariant)**.** *If before the insertion of an element v the level and uniform invariants hold, then the uniform invariant also holds after the execution of* Insert(v).

Proof. By the assumption that the uniform invariant holds before the insertion of the element v, we mean that for any arbitrary i and any arbitrary element e, the following holds:

$$\mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-\right] = \frac{1}{|R_i^-|} \cdot \mathbb{1}\left[e \in R_i^-\right] \quad .$$

We aim to prove that given our assumptions, after the execution of Insert(v), for each arbitrary i and each arbitrary element e, we have

$$\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right] \quad .$$

Note that $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i\right]$, is only defined when $\mathbb{P}\left[\mathbf{T} \geq i, \mathbf{H}_i = H_i\right] > 0$, which means that given the input and considering the behavior of our algorithm including its random choices, it is possible to reach a state where $\mathbf{T} \geq i$ and $\mathbf{H}_i = H_i$. In this proof, we use $\mathbf{p}_i$ to denote to the variable p_i used in the Insert as a random variable.

Fix any arbitrary i and any arbitrary element e. Since $\mathbf{H}_i^- = (\mathbf{e}_1^-, \ldots, \mathbf{e}_{i-1}^-, \mathbf{R}_0^-, \mathbf{R}_1^-, \ldots, \mathbf{R}_i^-)$ refers to our data structure levels before the insertion of the element v, it is clear that the following facts hold about $\mathbf{H}_i^-$.

Fact 45. *For any $j < i$, $\mathbf{e}_j^- \neq v$.*

Fact 46. *For any $j \leq i$, $v \notin \mathbf{R}_j^-$.*

We consider the following cases based on which of the following holds for $H_i = (e_1, \ldots, e_{i-1}, R_0, R_1, \ldots, R_i)$:

- Case 1: If the $e_j = v$ for some $j < i$.
- Case 2: If $v \notin \{e_1, \ldots, e_{i-1}\}$.

We handle these two cases separately (Lemma 3.3.1 for the first case and Lemma 3.3.3 for the second case). We show that, no matter the case, $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i\right]$ is equal to $\frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right]$, which completes the proof of the Lemma.

Claim 3.3.1. *If H_i is such that there is a $1 \le j < i$ that $e_j = v$, then* $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right]$.

Proof. We know that, $\mathbf{p_j}$ must have been equal to 1, as otherwise, instead of having $\mathbf{e_j} = e_j = v$, we would have had $\mathbf{e}_j = \mathbf{e}_j^-$, which would not have been equal to v as stated in Fact 45. According to our algorithm, since $\mathbf{p_j}$ has been equal to 1, we have invoked MatroidConstructLevel$(j+1)$. By Lemma 43, we know that the level invariants hold at the end of the execution of Insert, which is also the end of the execution of MatroidConstructLevel$(j+1)$. Thus, Lemma 41, proves that $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right]$. T

Claim 3.3.2. *Assume that H_i is such that $e_j \neq v$ for any $1 \le j < i$ and define H_i^- based on H_i as $H_i^- := (R_0 \backslash \{v\}, \dots, R_i \backslash \{v\}, e_1, \dots, e_{i-1})$. The events $[\mathbf{T} \ge i, \mathbf{H}_i = H_i]$ and $[\mathbf{T}^- \ge i, \mathbf{H}_i^- = H_i^-, \mathbf{p_1} = 0, \dots, \mathbf{p_{i-1}} = 0]$ are equivalent and imply each other, thusly they are interchangeable.*

Proof. First, we show that if $\mathbf{T} \ge i, \mathbf{H}_i = H_i$, then $\mathbf{T}^- \ge i, \mathbf{H}_i^- = H_i^-, \mathbf{p_1} = 0, \dots, \mathbf{p_{i-1}} = 0$. Considering that case 2 holds for H_i, $\mathbf{H}_i = H_i$, means that for any $j < i$, $\mathbf{e}_i \neq v$, which means there is no $j < i$ with $\mathbf{p_j} = 1$. Note that if $\mathbf{p_j} = 1$, then we would have set $\mathbf{e_j}$ to be equal to v, and we would have invoked MatroidConstructLevel$(j+1)$. Thus, in addition to knowing that for any $j < i$, $\mathbf{p_j} = 0$, we also know that, we have not invoked MatroidConstructLevel$(j+1)$ for any $j < i$. As for any $j < i$, $\mathbf{p_j} = 0$ and MatroidConstructLevel$(j+1)$ was not invoked, we have the following results:

1. Level i also existed before the insertion of v, i.e. $\mathbf{T}^- \ge i$.
2. We have made no change in the values of $(\mathbf{e_1}, \dots, \mathbf{e_{i-1}})$, and they still have the values they had before the insertion of v, i.e. for any $j < i$, $\mathbf{e_j} = \mathbf{e_j^-}$, and so $\mathbf{e_j^-} = e_j$.
3. All the change we might have made in our data structure is limited to adding the element v to a subset of $\{\mathbf{R_0^-}, \dots, \mathbf{R_i^-}\}$. Hence, for any $j \le i$, whether $\mathbf{R_j}$ is equal to $\mathbf{R_j^-}$ or $\mathbf{R_j^-} \cup \{v\}$, $\mathbf{R_j^-} = \mathbf{R_j} \backslash \{v\} = R_j \backslash \{v\}$.

So far, we have proved that throughout our algorithm, we reach the state, where $\mathbf{T} \ge i, \mathbf{H}_i = H_i$, only if $\mathbf{T}^- \ge i, \mathbf{H}_i^- = H_i^-, \mathbf{p_1} = 0, \dots, \mathbf{p_{i-1}} = 0$.

We know that in our insertion algorithm, there is not any randomness other than setting the value of $\mathbf{p_j}$ as long as we have not invoked MatroidConstructLevel, which only happens when for a j, $\mathbf{p_j}$ is set to be 1. It means that the value of $\mathbf{H}_i$ can be determined uniquely if we know the value of $\mathbf{H}_i^-$, and we know that $\mathbf{p_1}, \dots, \mathbf{p_{i-1}}$ are all equal to 0. Since we have assumed that $\mathbf{T} \ge i, \mathbf{H}_i = H_i$ is a valid and reachable state in our algorithm, $\mathbf{T}^- \ge i, \mathbf{H}_i^- = H_i^-$ must have been a reachable state as well. Plus, $\mathbf{T}^- \ge i, \mathbf{H}_i^- = H_i^-, \mathbf{p_1} = 0, \dots, \mathbf{p_{i-1}} = 0$, should imply that $\mathbf{T} \ge i$ and $\mathbf{H}_i = H_i$. Otherwise, $\mathbf{T} \ge i, \mathbf{H}_i = H_i$ could not be a reachable state, which is in contradiction with our assumption. T

Claim 3.3.3. *If H_i is such that $e_j \neq v$ for any $1 \le j < i$, then* $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right]$.

Proof. Define H_i^- based on H_i as $H_i^- := (R_0 \backslash \{v\}, \dots, R_i \backslash \{v\}, e_1, \dots, e_{i-1})$.
We calculate $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right]$. As stated above, considering that Case 2 holds for H_i, we know that $\mathbf{T} \ge i, \mathbf{H}_i = H_i$ implies that MatroidConstructLevel has not been invoked for any $j < i$. Thus, the value of $\mathbf{e}_i$ will be determined based on the random variable $\mathbf{p_i}$. And we have:

$$\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right] = \sum_{p_i \in \{0,1\}} \left(\mathbb{P}\left[\mathbf{p_i} = p_i | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right] \cdot \mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \ge i, \mathbf{H}_i = H_i, \mathbf{p_i} = p_i\right]\right) .$$

According to the algorithm, if $v \in H_i$, then $\mathbb{P}\left[\mathbf{p_i} = 1 | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right]$ is equal to $\frac{1}{|R_i|}$. Otherwise, if $v \notin H_i$, then $\mathbf{p_i}$ would be zero by default, and $\mathbb{P}\left[\mathbf{p_i} = 1 | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right] = 0$. Hence, we can say that:

$$\mathbb{P}\left[\mathbf{p_i} = 1 | \mathbf{T} \ge i, \mathbf{H}_i = H_i\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[v \in R_i\right] .$$

Additionally, Having $\mathbf{T} \geq i, \mathbf{H}_i = H_i$, if $\mathbf{p_i} = 1$, then $\mathbf{e}_i$ would be v. Otherwise, if $\mathbf{p_i} = 0$, then $\mathbf{e}_i^-$ would remain unchanged, i.e. $\mathbf{e}_i = \mathbf{e}_i^-$. Hence, $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i]$ is equal to

$$\frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i] \cdot \mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{p_i} = 1] + (1 - \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i]) \cdot \mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{p_i} = 0\right].$$

We consider the following cases based on the value of e:

- Case (i): $e = v$

  In this case $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{p_i} = 1] = 1$, and $\mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{p_i} = 0\right] = 0$. Thus, we have:

  $$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i] = \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i] \cdot 1 + (1 - \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i]) \cdot 0 = \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i].$$

- Case (ii): $e \neq v$ In this case, $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{p_i} = 1] = 0$. So we have:

  $$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i] = \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i] \cdot 0 + (1 - \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i]) \cdot \mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{p_i} = 0\right].$$

  According to the claim that we proved beforehand, $\mathbf{T} \geq i, \mathbf{H}_i = H_i$ and $\mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{p_1} = 0, \ldots, \mathbf{p_{i-1}} = 0$ are interchangeable. So we have:

  $$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i] = (1 - \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i]) \cdot \mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{p_1} = 0, \ldots, \mathbf{p_i} = 0\right].$$

  Since for any $j \leq i$, $\mathbf{e}_i^-$ and $\mathbf{p_i}$ are independent random variables, we have:

  $$\begin{aligned}\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i] &= (1 - \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i]) \cdot \mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-\right] \\ &= (1 - \frac{1}{|R_i|} \cdot \mathbb{1}[v \in R_i]) \cdot \left(\frac{1}{|R_i^-|} \cdot \mathbb{1}\left[e \in R_i^-\right]\right) ,\end{aligned}$$

  where the last equality holds because of the assumption stated in Lemma. From the definition of H_i^-, we have $R_i^- = R_i \backslash \{v\}$. Therefore,

  $$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i] = \frac{|R_i| - \mathbb{1}[v \in R_i]}{|R_i|} \cdot \left(\frac{1}{|R_i| - \mathbb{1}[v \in R_i]} \cdot \mathbb{1}[e \in R_i \backslash \{v\}]\right) .$$

  And since, $e \neq v$, we have:

  $$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i] .$$

T

As stated before, proof of these claims completes the Lemma's proof. T

Correctness of invariants after every deletion

Now, we consider the case when the update at time t of the sequence $\mathcal{S}$, is a deletion of an element v, and prove the following theorem.

Theorem 47. *If before the deletion of an element v, the level invariants and the uniform invariant hold, then they also hold after the execution of* Delete(v).

Similar to Theorem 42, we break the proof of this theorem into Lemmas 48 and 49.

Lemma 48 (Level invariants)**.** *If before the deletion of an element v the level invariants (i.e., starter, survivor, independent, weight, and terminator) hold, then they also hold after the execution of* Delete(v).

Lemma 49 (Uniform invariant)**.** *If before the deletion of an element v, the level and uniform invariants hold, then the uniform invariant also holds after the execution of* Delete(v).

Proof. In other words, we want to prove that if for any i and any element e

$$\mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-\right] = \frac{1}{|R_i^-|} \cdot \mathbb{1}\left[e \in R_i^-\right] \quad ,$$

then, after execution Delete(v), for each i and each element e, we have

$$\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right] \quad .$$

Fix any arbitrary i and e. We define a random variable $\mathbf{X_i}$ attaining values from the set $\{0, 1, 2\}$, as follows:

1. If the execution of Delete(v) has terminated after invoking MatroidConstructLevel(j), then we set $\mathbf{X_i}$ to 2.
2. If the execution of Delete(v) has terminated in a level $L_{j\leq i}$ because $v \notin R_j^-$, then we set $\mathbf{X_i}$ to 1.
3. Otherwise, we set $\mathbf{X_i}$ to 0. That is, this case occurs if $v \in R_i^-$ and Delete(v) terminates because in a level $L_{j>i}$, either $e_j = v$ or $v \notin R_j$.

In Claims 3.3.4, 3.3.7, and 3.3.8, we show that for each value $X_i \in \{0, 1, 2\}$, $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = X_i\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right]$. This would imply the statement of our Lemma and completes the proof since

$$\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i\right] = \mathbb{E}_{X_i \sim \mathbf{X_i}}\left[\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = X_i\right]\right]$$

by the law of total probability.

Claim 3.3.4. $\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 0\right] = \frac{1}{|R_i|} \cdot \mathbb{1}\left[e \in R_i\right]$.

Proof. First, we prove the following claim.

Claim 3.3.5. *If $\mathbf{X_i} = 0$, then for every $j < i$, $e_j \neq v$ and $v \notin R_i$.*

Proof. Since $\mathbf{X_i} = 0$, then MatroidConstructLevel(j) has not been invoked for any $j \leq i$. Thus, $\mathbf{e_j^-} = \mathbf{e_j} = e_j$ for any $j < i$. However, if $e_j = v$ for a level index $j < i$, then $\mathbf{e_j^-} = v$ would have held for that $j < i$, which means that MatroidConstructLevel(j) would have been executed for that j. This contradicts the assumption that $\mathbf{X_i} = 0$. Therefore, for all $j < i$, we must have $e_j \neq v$ proving the first part of this claim.

Next, we prove the second part. Since $\mathbf{X_i} = 0$, the algorithm Delete(v) neither has called MatroidConstructLevel nor it terminates its execution until level L_i. Thus, $\mathbf{R_i} = \mathbf{R_i^-} - v$, which implies that $v \notin \mathbf{R_i}$. However, if we had $v \in R_i$, then the event $[\mathbf{H}_i = H_i, \mathbf{X_i} = \mathbf{0}]$ would have been impossible. T

Using Claim 3.3.5, we know that $e_j \neq v$ for $j < i$ and $v \notin R_i$. However, we also know that $v \in R_j^-$ for $j \leq i$. Thus, we can define $H_i^- = (e_1^-, \dots, e_{i-1}^-, R_0^-, \dots, R_i^-)$ based on $H_i = (e_1, \dots, e_{i-1}, R_0, \dots, R_i)$ as follows:

$$H_i^- = (e_1, \dots, e_{i-1}, R_0 \cup \{v\}, \dots, R_i \cup \{v\}) \ .$$

Claim 3.3.6. *Two events* $[\mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 0]$ *and* $[\mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{e}_i^- \neq v]$ *are equivalent (i.e., they imply each other).*

Proof. We first prove that the event $[\mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 0]$ implies the event $[\mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{e}_i^- \neq v]$. Indeed, since $\mathbf{X_i} = 0 \neq 2$ we know that the algorithm MatroidConstructLevel(j) was not invoked for any $j \leq i$ and the element v was contained in $\mathbf{R_j^-}$ for all $j \leq i$. In this case, according to the algorithm Delete(v), we conclude that for any $j \leq i$, we have $\mathbf{e_j^-} \neq v$ and $\mathbf{e_j^-} = \mathbf{e_j}$, and $\mathbf{R_j} = \mathbf{R_j^-} - v$. This means that $\mathbf{R_j^-} = \mathbf{R_j} \cup \{v\}$. Therefore, since $\mathbf{H}_i = H_i$, we must have $\mathbf{H}_i^- = H_i^-$, $\mathbf{e}_i^- \neq v$, and $\mathbf{e}_i^- = \mathbf{e}_i$.

Next, we prove the other way around. That is, the event $[\mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{e}_i^- \neq v]$ implies the event $[\mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 0]$. Indeed, since $\mathbf{H}_i^- = H_i^- = (e_1, \dots, e_{i-1}, R_0 \cup \{v\}, \dots, R_i \cup \{v\})$, then, for any $j \leq i$, $v \in \mathbf{R_j^-}$ and for any $j < i$, $\mathbf{e_j^-} = e_j$.

Recall from Claim 3.3.5 that for all $j < i$, $e_j \neq v$ and $v \notin R_i$. Thus, for any $j < i$, we know that $\mathbf{e_j^-} \neq v$. However, we also know that $\mathbf{e}_i^- \neq v$. Thus, $\mathbf{e_j^-} \neq v$ for any $j \leq i$. This essentially means that the algorithm Delete(v) neither invokes MatroidConstructLevel nor terminates its execution till the level L_i. This implies that $\mathbf{X_i} = 0$. On the other hand, the algorithm Delete(v) only removes v from R_i^- and does not make any change in $\mathbf{e_1^-}, \dots, \mathbf{e_i^-}$. Thus, $\mathbf{R_i} = R_i^- - \{v\} = R_i \cup \{v\} - v = R_i$ and $\mathbf{e}_i = \mathbf{e}_i^-$. Therefore, we have $\mathbf{H}_i = H_i$. T

Therefore, we have the following corollary.

Corollary 50. $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 0] = \mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{e}_i^- \neq v\right]$.

Thus, in order to prove $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 0] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$, we can prove

$$\mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{e}_i^- \neq v\right] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i] \ .$$

Recall that the assumption of this lemma is $\mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-\right] = \frac{1}{|R_i^-|} \cdot \mathbb{1}\left[e \in R_i^-\right]$. That is, conditioned on the event $[\mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-]$, the random variable $\mathbf{e}_i^- \sim U(R_i^-)$ is a uniform random variable over the set R_i^-. (i.e., the value e_i of the random variable $\mathbf{e}_i^-$ takes ones of the elements of the set R_i^- uniformly at random.) However, since $X_i = 0$ and using Claim 3.3.6, we have $\mathbf{e}_i^- \neq v$. Thus, conditioned on the event $[\mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{e}_i^- \neq v]$, we have that the random variable $\mathbf{e}_i^- \sim U(R_i^- \backslash \{v\}) = U(R_i)$ should be a uniform random variable over the set $R_i^- \backslash \{v\} = R_i$. Indeed, we have

$$\begin{aligned}
\mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{e}_i^- \neq v\right] &= \frac{\mathbb{P}\left[\mathbf{e}_i^- = e, \mathbf{e}_i^- \neq v | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-\right]}{\mathbb{P}\left[\mathbf{e}_i^- \neq v | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-\right]} = \frac{\frac{1}{|R_i^-|} \cdot \mathbb{1}\left[e \in R_i^- \backslash \{v\}\right]}{1 - \frac{1}{|R_i^-|}} \\
&= \frac{1}{|R_i^-| - 1} \cdot \mathbb{1}\left[e \in R_i^- \backslash \{v\}\right] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i] \ ,
\end{aligned}$$

where the second equality holds because of our assumption that the uniform invariant holds before the deletion, and the fourth invariant holds because $R_i^- = R_i \cup \{v\}$ and $v \notin R_i$ proving the case $X = 0$.

T

Claim 3.3.7. $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 1] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$.

Proof. We will be conditioning on possible values of $\mathbf{H}_i^-$.

$$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, X_i = 1] = \mathbb{E}_{H_i^-}\left[\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 1, \mathbf{H}_i^- = H_i^-\right]\right] \ ,$$

where the expectation is taken over all H_i for which $\mathbb{P}\left[\mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 1, \mathbf{H}_i^- = H_i^-\right] > 0$. For all such H_i^-, we claim that this can be further rewritten as $\mathbb{P}\left[\mathbf{T} \geq i, \mathbf{H}_i^- = H_i^-\right]$. This is because Delete(v) is executed deterministically if it does not invoke the algorithm MatroidConstructLevel. Furthermore, the value of $\mathbf{X_i}$ is deterministically determined by $\mathbf{H}_i^-$. Therefore, for any value of H_i^-, either $\mathbf{H}_i^- = H_i^-$ implies $\mathbf{X_i} \neq 1$, in which case $\mathbb{P}\left[\mathbf{T} \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{X_i} = 1\right] = 0$, which is in contradiction with our assumption, or $\mathbf{H}_i^- = H_i^-$ imply $\mathbf{X_i} = 1$. Therefore, for all such H_i^- implies $\mathbf{X_i} = 1$, which also means that MatroidConstructLevel never gets invoked, in which case $\mathbf{H}_i$ is uniquely determined. Hence $\mathbf{H}_i^- = H_i^-$ should also imply that $\mathbf{H}_i = H_i$, as otherwise $\mathbb{P}\left[\mathbf{T} \geq i, \mathbf{H}_i^- = H_i^-, \mathbf{H}_i = H_i\right] = 0$. We therefore obtain:

$$\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 1, \mathbf{H}_i^- = H_i^-\right] = \mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i^- = H_i^-\right]$$

as claimed.

Also, we know that $\mathbf{H}_i = H_i, \mathbf{X_i} = 1$, implies that:

$$\mathbf{T}^- = \mathbf{T}, \quad \mathbf{R_i^-} = \mathbf{R_i}, \quad \mathbf{e}_i^- = \mathbf{e}_i,$$

since it means that the execution of Delete(v) has terminated before level i, thus no change has been made for that level. Therefore, for a H_i^- used in our expectation, we know that $\mathbf{T} \geq i, \mathbf{H}_i^- = H_i^-$ also implies

$$\mathbf{T}^- \geq i, \quad R_i^- = R_i, \quad \mathbf{e}_i^- = \mathbf{e}_i,$$

we have:

$$\mathbb{P}\left[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i^- = H_i^-\right] = \mathbb{P}\left[\mathbf{e}_i^- = e | \mathbf{T}^- \geq i, \mathbf{H}_i^- = H_i^-\right] = \frac{1}{|R_i^-|} \cdot \mathbb{1}\left[e \in R_i^-\right] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i] \ ,$$

where the third equality holds because of our assumption that the uniform invariant holds before the deletion of element v. Therefore, $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 1] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$.

T

Claim 3.3.8. $\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 2] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$.

Proof. By Lemma 48, we know that the level invariants hold at the end of the execution of Delete, which is also the end of the execution of MatroidConstructLevel(j). Using Lemma 41, we know that since the level invariants are going to hold after the execution of MatroidConstructLevel(j), for i which is greater than j, we have:

$$\mathbb{P}[\mathbf{e}_i = e | \mathbf{T} \geq i, \mathbf{H}_i = H_i, \mathbf{X_i} = 2] = \frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i] \ ,$$

which proves this claim. T

T

3.3.4 Application of Uniform Invariant: Query complexity

As for the query complexity of this algorithm, observe that checking if an element e is promoting for a level L_z needs $O(\log(k))$ oracle queries because of Lemma 39 and the fact that the size of the independent set I_z is at most k. The binary search that we perform needs $O(\log T)$ number of such suitability checks for the element e. Thus, if we initiate the leveling algorithm with a set R_i, our algorithm needs $O(|R_i| \cdot \log(k) \cdot \log(T))$ oracle queries to build the levels $L_i, \cdots, L_T$.

Lemma 51. *The number of levels T is at most $k \log(\frac{k}{\epsilon})$.*

Proof. Consider a directed graph G with elements $I'_T = \{e_1, \cdots, e_T\}$ as vertices of this graph. For each element $e_i \in I'_T$, we know that e_i is a promoting element for L_{i-1}, i.e. $\textsc{Promote}(I_{i-1}, I'_{i-1}, e_i, w[I_{i-1}]) \neq \textsc{Fail}$. Therefore, we define $parent(e_i) = \textsc{Promote}(I_{i-1}, I'_{i-1}, e_i, w[I_{i-1}])$. This value is $\emptyset$ if $I_i = I_{i-1} + e_i$. Otherwise, if $I_i = I_{i-1} - e' + e_i$, this value would be e'. For each $e_i \in I'_T$, if $parent(e_i) \neq \emptyset$, we add an edge $e_i \to parent(e_i)$ to the graph.

Since an element can only be replaced once, we have $|\{e' | e' \in I'_T, parent(e') = e\}| = 1$, i.e. the in-degree of each $e \in I_T$ is at most 1. Furthermore, the out-degree of each vertex is 1, because for each element $e \in I'_T$, $|parent(e)| \leq 1$. Therefore, it follows that the graph is a union of disjoint paths and each $e_i \in I'_T$ is in exactly one path.

An element e is a starting element in a path (its in-degree is 0), if and only if it has not been replaced by another element. That means, e remains in I_T at the end of the algorithm. Given that $|I_T| \leq k$, there are at most k paths in G. Furthermore, for two successive elements (u, v) in the path where $parent(u) = v$, $w(u) \geq 2w(v)$. As the weights of all elements in I'_T satisfy $w(e) \in [\frac{\epsilon}{10k} MAX, MAX]$, the length of each path is bounded by $\log(k/\epsilon) + 4$. Consequently, it follows that the total number of vertices in the graph is at most $O(k \log(\frac{k}{\epsilon}))$. T

Next, we analyze the query complexity of $\textsc{MatroidConstructLevel}$.

Lemma 52. *The total cost of calling* $\textsc{MatroidConstructLevel}(i)$ *is at most* $O\left(|R_i| \log(k) \log\left(\frac{k}{\epsilon}\right)\right)$.

Proof. Checking if an element e is promoting needs $O(\log(k))$ query calls, because of Lemma 39 and the fact that $|I| \leq k$ for any $I \in \mathcal{I}$. The algorithm $\textsc{MatroidConstructLevel}(i)$ iterates over all elements in R_i. For each element e, it first calls the $\textsc{Promote}$ function, and select e if it is a promoting element, i.e. $\textsc{Promote}(I_{\ell-1}, I'_{\ell-1}, e, w[I_{\ell-1}]) \neq \textsc{Fail}$. In this case, we only need $O(\log(k))$ query calls. However, if e is not a promoting element, it reaches Line 13 and runs the binary search on the interval $[i, \ell - 1]$. Based on Lemma 51, the length of this interval is $O\left(k \log\left(\frac{k}{\epsilon}\right)\right)$. Therefore, the number of steps in binary search is at most $O\left(\log\left(k \log\left(\frac{k}{\epsilon}\right)\right)\right) = O\left(\log\left(\frac{k}{\epsilon}\right)\right)$. In each step of the binary search, the algorithm calls $\textsc{Promote}$ one time. Thus, for each element we need $O\left(\log(k) \log\left(\frac{k}{\epsilon}\right)\right)$, and for all elements, we need $O\left(|R_i| \log(k) \log\left(\frac{k}{\epsilon}\right)\right)$ query calls. T

Lemma 53. *For a specified value of MAX, each update operation in Algorithm 8 has query complexity at most $O\left(k \log(k) \log^2\left(\frac{k}{\epsilon}\right)\right)$.*

Proof. We divide the queries made by the algorithm into two categories: the queries made directly by the update operations $\textsc{Insert}$ and $\textsc{Delete}$, and the queries made indirectly, if the update triggers a call to $\textsc{MatroidConstructLevel}$. For the first category, the number of queries for each update is always $O(T)$ for insertion which can be bounded by $O\left(k \log\left(\frac{k}{\epsilon}\right)\right)$, and there are no queries made for deletion. We therefore focus on the second category.

Based on uniform invariant, when we insert/delete an element, for each natural number $i \leq T$, we call MatroidConstructLevel(i) with probability $\frac{1}{|R_i|} \cdot \mathbb{1}[e \in R_i]$ which is at most $\frac{1}{|R_i|}$. Using Lemma 52, the query complexity for calling MatroidConstructLevel(i) is $O\left(|R_i| \log(k) \log\left(\frac{k}{\epsilon}\right)\right)$. Therefore, the expected number of queries caused by level i is bounded by $\frac{1}{|R_i|} \cdot O\left(|R_i| \log(k) \log\left(\frac{k}{\epsilon}\right)\right) = O\left(\log(k) \log\left(\frac{k}{\epsilon}\right)\right)$. As the Lemma 51 bounded the number of levels by $T = O\left(k \log\left(\frac{k}{\epsilon}\right)\right)$, we calculate the expected number of query calls for each update by summing the expected number of query calls at each level:

$$\sum_{i=1}^{T} O\left(\log(k) \log\left(\frac{k}{\epsilon}\right)\right) \leq O\left(k \log(k) \log^2\left(\frac{k}{\epsilon}\right)\right) \quad .$$

T

In order to obtain an algorithm that works regardless of the value of MAX, we guess MAX up to a factor of 2 using parallel runs. Each element is inserted only to $\log(k/\epsilon)$ copies of the algorithm. Therefore, we obtain the total query complexity claimed in Theorem 54.

Theorem 54. *The expected query complexity of each insert/delete for all runs is* $O\left(k \log(k) \log^3\left(\frac{k}{\epsilon}\right)\right)$.

3.3.5 Application of Level Invariants: Approximation guarantee

Recall that we run parallel instances of DynamicMatroid for different guesses of the maximum value MAX such that after each update, there is a run with $\max_{e \in V_t} f(e) \in (MAX/2, MAX]$, where V_t is the set of elements that have been inserted but not deleted yet. In this section, we only talk about the run with $\max_{e \in V_t} f(e) \in (MAX/2, MAX]$. We prove that if the level invariants hold, then after each update the submodular value of the set I_T in this run is a $(4+\epsilon)$-approximation of the optimal value OPT. Formally, we state this claim as follows:

Theorem 55. *Suppose that the level invariants hold in every run of* DynamicMatroid*. Let I_T be the independent set of the final level L_T in the run with* $\max_{e \in V} f(e) \in (MAX/2, MAX]$*. Then, the set I_T satisfies* $(4+\epsilon) \cdot f(I_T) \geq OPT$*, where* $OPT = \max_{I^* \in \mathcal{I}} f(I^*)$.

To this end, we first define a few notations.

Definition 3.3.2. *For an element $e \in \mathcal{V}$, we let $z(e)$ denote the largest i such that $e \in R_i$. In Algorithms 7 and 8, $w(e)$ is defined for all elements $e \in I'_T$, but we need to define it for other elements as well. Therefore, if $e_{z(e)} = e$, we set $w(e) = f(I'_{z(e)-1} + e) - f(I'_{z(e)-1})$, to match the value defined in the Algorithm. Otherwise, we set $w(e) = f(I'_{z(e)} + e) - f(I'_{z(e)})$. For a set $E \subseteq \mathcal{V}$, we define $w(E) = \sum_{e \in E} w(e)$.*

We split the proof of Theorem 55 into four steps. We first (in Lemma 56) prove that $w(I'_T) \leq 2w(I_T)$. Later, in Lemma 57 we show that the sum of the weight of the elements in I_T is upper-bounded by the submodular function of I_T. That is, $w(I_T) \leq f(I_T)$. Recall that $OPT = \max_{I \in \mathcal{I}} f(I)$ and we used the notation $I^* = \arg\max_{I \in \mathcal{I}} f(I)$ for an independent set in $\mathcal{I}$ whose submodular value is maximum. In the third step of the proof of Theorem 55, we show that $f(I^*) \leq 2w(I_T) + w(I^*)$. We prove this in Lemma 58. Our proofs for these lemmas are inspired by the analysis in Chakrabarti and Kale (Chakrabarti and Kale, 2015) who study the streaming version of the problem. Finally, we show that $w(I^*) \leq 2w(I_T) + \frac{\epsilon}{5} \cdot f(I^*)$. This is proven in Lemma 59 using an argument inspired by the analysis of Ashwinkumar (Badanidiyuru Varadaraja, 2011).

Having all these tools in hand, we can then finish the proof of Theorem 55. Indeed, we have

$$f(I^*) \overset{(a)}{\leq} 2w(I_T) + w(I^*) \overset{(b)}{\leq} 4w(I_T) + \frac{\epsilon}{5} \cdot f(I^*) \overset{(c)}{\leq} 4f(I_T) + \frac{\epsilon}{5} \cdot f(I^*) \quad , \tag{3.1}$$

where (a), (b), and (c) follow from Lemmas 58, 59 and 57 respectively.

This essentially means that $f(I^*) \leq \frac{4}{1-\frac{\epsilon}{5}} \cdot f(I_T)$. Now observe that $\frac{4}{1-\frac{\epsilon}{5}} \leq 4+\epsilon$. Indeed, if we want to have this claim correct, we must have $20 - 4\epsilon + 5\epsilon - \epsilon^2 \geq 20$ which means we must have $\epsilon(\epsilon - 1) \leq 0$. However, this is correct since $0 < \epsilon \leq 1$, which finishes the proof of Theorem 55.

Next, we prove the four steps that we explained above.

Definition 3.3.3 (Span)**.** *Let $E \subseteq V$ be a set of elements. We define* $\textsc{Span}(E) = \{e \in V : rank(E+e) = rank(E)\}$.

Lemma 56. $w(I'_T) \leq 2w(I_T)$.

Proof. We prove by induction on i that $w(I'_i) \leq 2w(I_i)$ for all i. Setting $i = T$ will finish the proof.
The claim holds for $i = 0$ as $w(I_i) = w(I'_i) = w(\emptyset) = 0$. Assume that the claim holds for $i-1$, we prove it holds for i as well. Given independent invariant, $I'_i = I'_{i-1} + e_i$ and either $I_i = I_{i-1} + e_i$ or $I_i = I_{i-1} + e_i - \hat{e}$ for some $\hat{e}$ satisfying $w(\hat{e}) \leq \frac{w(e_i)}{2}$. In either case,

$$\begin{aligned} w(I_i) &\geq w(I_{i-1}) + w(e_i) - \frac{w(e_i)}{2} \geq w(I_{i-1}) + \frac{w(e_i)}{2} \\ &\overset{(a)}{\geq} \frac{1}{2}w(I'_{i-1}) + \frac{w(e_i)}{2} = \frac{1}{2}w(I'_i) \ , \end{aligned}$$

where for (a), we have used the induction assumption for $i-1$. T

Lemma 57. *The sum of the weight of the elements in I_T is upper-bounded by the submodular function of I_T. That is, $w(I_T) \leq f(I_T)$.*

Proof. For each $i \in [T]$, define $\widetilde{I_i}$ as $I_i \cap I_T$. We prove by induction on i that $w(\widetilde{I_i}) \leq f(\widetilde{I_i})$. Setting $i = T$ proves the claim.

The case of $i = 0$ holds trivially as $w(\widetilde{I_0}) = f(\widetilde{I_0}) = 0$. Assume that $w(\widetilde{I_{i-1}}) \leq f(\widetilde{I_{i-1}})$, we will prove that $w(\widetilde{I_i}) \leq f(\widetilde{I_i})$. If $e_i \notin I_T$, then the claim holds trivially as $\widetilde{I_i} = \widetilde{I_{i-1}}$. Note that in this case, if an element has appeared in I_{i-1}, but it is removed from I_i, then it is not included in I_T and hence $\widetilde{I_{i-1}}$. We therefore assume that $e_i \in I_T$. In this case, we note that

$$I'_{i-1} = \bigcup_{j \leq i-1} I_j \supseteq I_{i-1} \supseteq \widetilde{I_{i-1}} \ . \tag{3.2}$$

Therefore,

$$w(\widetilde{I_i}) - w(\widetilde{I_{i-1}}) = w(e_i) \overset{(a)}{=} f(I'_{i-1} + e) - f(I'_{i-1}) \overset{(b)}{\leq} f(\widetilde{I_{i-1}} + e) - f(\widetilde{I_{i-1}}) = f(\widetilde{I_i}) - f(\widetilde{I_{i-1}}) \ ,$$

where for (a) we have used weight invariant, and for (b) we have used the definition of submodularity together with (3.2). Summing the above inequality with the induction hypothesis $w(\widetilde{I_{i-1}}) \leq f(\widetilde{I_{i-1}})$ proves the claim. T

Lemma 58. *Recall that $OPT = \max_{I \in \mathcal{I}} f(I)$ and we used the notation $I^* = \arg\max_{I \in \mathcal{I}} f(I)$ for an independent set in $\mathcal{I}$ whose submodular value is maximum. Then, $f(I^*) \leq 2w(I_T) + w(I^*)$.*

Proof. We first note that

$$f(I'_T) = \sum_{i=1}^{T} f(I'_i) - f(I'_{i-1}) = \sum_{i=1}^{T} f(I'_{i-1} + e_i) - f(I'_{i-1}) \overset{(a)}{=} \sum_{i=1}^{T} w(e_i) = w(I'_T) \overset{(b)}{\leq} 2w(I_T) \ , \tag{3.3}$$

where (a) follows from weight invariant, and (b) follows from Lemma 56.

We now bound $f(I^*)$. Enumerate $I^* \backslash I'_T$ as $\{e^*_1, \ldots, e^*_{|I^* \backslash I'_T|}\}$ in an arbitrary order. Define $D_0 = I'_T$ and $D_i = I'_T \cup \{e^*_1, \ldots e^*_i\}$. It is clear that $D_{i-1} \supseteq I'_T \supseteq I'_{z(e^*_i)}$. Therefore,

$$f(D_i) - f(D_{i-1}) = f(D_{i-1} + e^*_i) - f(D_{i-1}) \overset{(a)}{\leq} f(I'_{z(e^*_i)} + e^*_i) - f(I'_{z(e^*_i)}) \overset{(b)}{=} w(e^*_i) \ ,$$

where for (a) we have used the definition of submodularity, and (b) holds because $e^*_i \notin I'_T$. Summing over all i, we obtain

$$\begin{aligned} \sum_{i=1}^{|I^* \backslash I'_T|} f(D_i) - f(D_{i-1}) &\leq \sum_{i=1}^{|I^* \backslash I'_T|} w(e^*_i) \\ f(D_{|I^* \backslash I'_T|}) - f(D_0) &\leq w(I^* \backslash I'_T) \\ &\leq w(I^*) \ . \end{aligned}$$

Given that $D_0 = I'_T$ and $D_{|I^* \backslash I'_T|} = I^* \cup I'_T$, we have

$$f(I^*) \leq f(I^* \cup I'_T) \leq f(I'_T) + w(I^*) \leq 2w(I_T) + w(I^*) \ ,$$

where the last inequality follows from (3.3).

T

Lemma 59. *$w(I^*) \leq 2w(I_T) + \frac{\epsilon}{5} \cdot f(I^*)$.*

We first give a sketch of the proof of Lemma 59.

We split the I^* into two parts. The first part consists of elements with weights $w(e) \leq \frac{\epsilon}{10k} MAX$. As we will show, the total weight of these elements can be bounded by $\frac{\epsilon}{5} \cdot f(I^*)$. As for the second group, we will show that each element can be mapped one-to-one to an element in I_T with at least half its weight.
Formally, let I^*_W consist of all the elements in I^* such that $w(e) \leq \frac{\epsilon}{10k} MAX$. We first bound $w(I^*_W)$:

$$\begin{aligned} w(I^*_W) = \sum_{e \in I^*_W} w(e) &\leq |I^*_W| \cdot \frac{\epsilon}{10k} \cdot MAX \\ &\leq |I^*| \cdot \frac{\epsilon}{10k} \cdot MAX \\ &\leq \frac{\epsilon}{10} \cdot MAX < \frac{\epsilon}{5} \cdot f(I^*) \ . \end{aligned} \tag{3.4}$$

The last conclusion comes from the fact that $f(I^*) \geq \max_{e \in V} f(e) \in (\frac{MAX}{2}, MAX]$.

In order to bound $w(I^* \backslash I^*_W)$, we will use the following lemmas:
Lemma 60. *Let sets $E_1, E_2 \subseteq V$ and elements $e_1, e_2 \in V$. If $e_1 \in \textsc{Span}(E_1)$ and $e_2 \in \textsc{Span}(E_2 - e_2)$, then $e_1 \in \textsc{Span}((E_1 \cup E_2) - e_2)$.*

Proof. Note that if we have two sets $S_1, S_2 \subseteq E$ such that $S_1 \subseteq S_2$, then $\textsc{Span}(S_1) \subseteq \textsc{Span}(S_2)$ and $rank(S_1) \leq rank(S_2)$. Now, using Definition 3.3.3 with the fact $E_1 \subseteq (E_1 \cup E_2)$ gives us $e_1 \in \textsc{Span}(E_1) \subseteq \textsc{Span}(E_1 \cup E_2)$. Therefore, we have $rank((E_1 \cup E_2) + e_1) = rank(E_1 \cup E_2)$.

In a similar way, since $E_2 - e_2 \subseteq ((E_1 \cup E_2) - e_2)$, we can conclude that $e_2 \in \textsc{Span}(E_2 - e_2) \subseteq \textsc{Span}((E_1 \cup E_2) - e_2)$ what implies that $rank(E_1 \cup E_2) = rank((E_1 \cup E_2) - e_2)$.

By using these two results, we conclude that $rank((E_1 \cup E_2) - e_2) = rank((E_1 \cup E_2) + e_1)$. Furthermore,

$$rank((E_1 \cup E_2) - e_2) \le rank((E_1 \cup E_2) - e_2 + e_1) \le rank((E_1 \cup E_2) + e_1)$$

where the first and third parts are equal. Therefore, all of them are equal and $rank((E_1 \cup E_2) - e_2 + e_1) = rank((E_1 \cup E_2) - e_2)$, which implies that $e_1 \in \textsc{Span}((E_1 \cup E_2) - e_2)$. T

Lemma 61. *There is a function $N : I^* \backslash I^*_W \to 2^{I_T}$ such that for all $e \in I^* \backslash I^*_W$, $e \in \textsc{Span}(N(e))$ and for all $e' \in N(e)$, $w(e) \le 2w(e')$.*

Proof. Define $\widetilde{I_i} := \{e \in I^* \backslash I^*_W : z(e) \le i\}$. We prove by induction on $i \in [T]$ that there is a function $N_i : \widetilde{I_i} \to 2^{I_i}$ such that $e \in \textsc{Span}(N_i(e))$ and $w(e) \le 2w(e')$ for all $e' \in N_i(e)$.

The induction base holds trivially as $\widetilde{I_0} = \emptyset$. Assume the claim holds for $i-1$. We show it holds for i. Let e be an element of $\widetilde{I_i}$. We define $N_i(e)$ based on three cases as follows.

- **Assume that $\mathbf{e} \in \widetilde{\mathbf{I}}_{\mathbf{i-1}}$.** If $I_i = I_{i-1} + e_i$ or $I_i = I_{i-1} + e_i - \hat{e}$ for some $\hat{e} \notin N_{i-1}(e)$, we set $N_i(e) = N_{i-1}(e)$. N_i has the desirable properties for e by the induction hypothesis. Otherwise, assuming that $I_i = I_{i-1} + e_i - \hat{e}$, for some $\hat{e} \in N_{i-1}(e)$. By Lemma 37 there is a unique circuit in $I_{i-1} + e_i$, named C. Define $N_i(e)$ as $(N_{i-1}(e) \cup C) - \hat{e}$. Since $\hat{e} \in N_{i-1}(e)$, by induction hypothesis, $w(e_i) \le 2w(\hat{e})$. Also, given independent invariant, $\hat{e} = \textsc{Promote}(I_{i-1}, I'_{i-1}, e_i, w[I_{i-1}])$, i.e. $\hat{e} \leftarrow \arg\min_{e' \in C} w(e')$. Thus, $w(e_i) \le 2w(\hat{e}) \le 2w(e')$ for all $e' \in C - \hat{e}$. Moreover, $w(e_i) \le 2w(e')$ for all $e' \in N_{i-1}(e)$ by induction hypothesis. Hence, $w(e_i) \le 2w(e')$ for all $e' \in ((N_{i-1}(e) \cup C) - \hat{e})$. Furthermore, since $e \in \textsc{Span}(N_{i-1}(e))$ and $\hat{e} \in \textsc{Span}(C - \hat{e})$, we can use Lemma 60 to conclude that $e \in \textsc{Span}(N_i(e))$.
- **Assume that $\mathbf{e} = \mathbf{e_i}$.** In this case, we set $N_i(e) = e$.
- **If neither of the two cases above hold,** then $z(e) = i$ but $e \neq e_i$. According to the survivor invariant, e is not a promoting element for L_i. It follows that $I_i + e$ is not independent. By Lemma 37 there is a unique circuit in $I_i + e$. Let C denote this circuit, and let $N_i(e) = C - e$. It is clear that $e \in \textsc{Span}(N_i(e))$, and $w(e) \le 2w(e')$ for all $e' \in C - e$ since otherwise, e would be promoting element for L_i.

Finally, we set $N = N_T$ to get the desired function. T

Lemma 62. *Assume that $E, E' \subseteq V$. If E be an independent set such that $E \subseteq \textsc{Span}(E')$, then $|E| \le |E'|$.*

Proof. Given that E is independent, we know that $|E| = rank(E)$. In addition, given that $E \subseteq \textsc{Span}(E')$, we have $rank(E) \le rank(\textsc{Span}(E'))$. Therefore, $|E| = rank(E) \le rank(\textsc{Span}(E')) = rank(E') \le |E'|$. T

Proof of Lemma 59. Let $N : I^* \backslash I^*_W \to 2^{I_T}$ be the function described in Lemma 61. Recall that $e \in \textsc{Span}(N(e))$ for all $e \in I^* \backslash I^*_W$. This further implies that for all $E \subseteq I^* \backslash I^*_W$, we have $E \subseteq \textsc{Span}(N(E))$. Therefore, as E is independent, we can use Lemma 62 to conclude that $|E| \le |N(E)|$.

By Hall's marriage theorem, we conclude that there is an injection $H : I^* \backslash I^*_W \to I_T$ such that $H(e) \in N(e)$ for all $e \in I^* \backslash I^*_W$. Therefore,

$$w(I^* \backslash I^*_W) = \sum_{e \in I^* \backslash I^*_W} w(e) \overset{(a)}{\le} \sum_{e \in I^* \backslash I^*_W} 2w(H(e)) \overset{(b)}{\le} \sum_{e' \in I_T} 2w(e') = 2w(I_T) \ .$$

where for (a) we have used the fact that $w(e) \le 2w(e')$ for all $e' \in N(e)$, and for (b) we have used the fact that H is an injection. Summing the above inequality with (3.4) finishes the proof.

$$w(I^*) = w(I^* \backslash I^*_W) + w(I^*_W) \le 2w(I_T) + \frac{\epsilon}{5} \cdot f(I^*) \ .$$

T

3.4 Parameterized dynamic algorithm for submodular maximization under cardinality constraint

In this section, we present our dynamic algorithm for the maximum submodular problem under the cardinality constraint k. The pseudo-code of our algorithm is provided in Algorithm 10. The overview of our dynamic algorithm is given in Section "Our contribution" 3.1.2. The analysis of this algorithm is similar to the dynamic algorithm that we designed for the matroid constraint.

Algorithm 10 CardinalityConstraintLeveling(k, OPT)

function Init(V)
  $\tau \leftarrow \frac{OPT}{2k}$
  $I_0 \leftarrow \emptyset$ and $R_0 \leftarrow V$
  $R_1 \leftarrow \{e \in R_0 : \text{Promote}(I_0, e) = True\}$
  Invoke ConstructLevel($i = 1$)

---

function ConstructLevel(i)
  Let P be a random permutation of elements of R_i and $\ell \leftarrow i$
  for e in P **do**
    if Promote($I_{\ell-1}, e$) $= True$ **then**
      $e_\ell \leftarrow e, \quad I_\ell \leftarrow I_{\ell-1} + e_\ell,$ and $z \leftarrow \ell$
      $\ell \leftarrow \ell + 1$ and $R_\ell \leftarrow \emptyset$
    else
      Run binary search to find the lowest $z \in [i, \ell - 1]$ such that Promote(I_z, e) $= False$
    for $r \leftarrow i + 1$ **to** z **do**
      $R_r \leftarrow R_r + e$.
  return $T \leftarrow \ell - 1$ which is the final ℓ that the for-loop above returns subtracted by one

---

function Promote(I, e)
  if $f(I + e) - f(I) \geq \tau$ and $|I| < k$ **then**
    return True
  return False

Relaxing OPT assumption. Our dynamic algorithm assumes the optimal value $OPT = \max_{I^* \subseteq V : |I^*| \leq k} f(I^*)$ is given as a parameter. However, in reality, the optimal value is not known in advance and may change after every insertion or deletion. To remove this assumption in Algorithm 12, we run parallel instances of our dynamic algorithm for different guesses of the optimal value OPT_t at any time t of the sequence $\mathcal{S}_t$, such that $\max_{I^* \subseteq V_t : |I^*| \leq k} f(I^*) \in (OPT_t/(1 + \epsilon), OPT_t]$ in one of the runs. Recall that V_t is the set of elements that have been inserted but not deleted from the beginning of the sequence till time t. These guesses that we take are $(1 + \epsilon)^i$ where $i \in \mathbb{Z}$. If ρ is the ratio between the maximum and minimum non-zero possible value of a subset of V with at most k elements, then the number of parallel instances of our algorithm will be $O(\log_{1+\varepsilon} \rho)$. This incurs an extra $O(\log_{1+\varepsilon} \rho)$-factor in the query complexity of our dynamic algorithm.

In fact, we can replace this extra factor with an extra factor of $O(\log(k)/\epsilon)$ which is independent of ρ. To this end, we use the well-known technique that has been also used in (Lattanzi et al., 2020). In particular, for every element e, we add it to those instances i for which we have $\frac{(1+\epsilon)^i}{2k} \leq f(e) \leq (1 + \epsilon)^i$. The reason is if the optimal value of V_t is within the range $((1 + \epsilon)^{i-1}, (1 + \epsilon)^i]$ and $f(e) > (1 + \epsilon)^i$, then $f(e)$ is greater than

Algorithm 11 CardinalityConstraintUpdates(k, OPT)

function Delete(v)
  $R_0 \leftarrow R_0 - v$
  for $i \leftarrow 1$ **to** T **do**
    if $v \notin R_i$ **then**
      break
    $R_i \leftarrow R_i - v$
    if $e_i = v$ **then**
      Invoke ConstructLevel(i).
      break

---

function Insert(v)
  $R_0 \leftarrow R_0 + v$.
  for $i \leftarrow 1$ **to** $T + 1$ **do**
    if Promote(I_{i-1}, v) = False **then**
      break
    $R_i \leftarrow R_i + v$.
    Let $p = 1$ with probability $\frac{1}{|R_i|}$, and otherwise $p = 0$.
    if $p = 1$ **then**
      $e_i \leftarrow v, \quad I_i \leftarrow I_{i-1} + v$
      $R_{i+1} = \{e' \in R_i : \text{Promote}(I_i, e') = True\}$
      ConstructLevel($i + 1$)
      break

the optimal value and can safely be ignored for the instance i that corresponds to the guess $(1+\epsilon)^i$. On the other hand, we can safely ignore all elements e whose $f(e) < \frac{(1+\epsilon)^i}{2k} = \tau$, since these elements will never be a promoting element in the run with $OPT = (1+\epsilon)^i$. This essentially means that every element e is added to at most $O(\log_{1+\epsilon}(2k)) = O(\log(k)/\epsilon)$ parallel instances. Thus, after every insertion or deletion, we need to update only $O(\log(k)/\epsilon)$ instances of our dynamic algorithm.

Algorithm 12 Unknown OPT

Let $\mathcal{A}_i$ be the instance of our dynamic algorithm, for which $OPT = (1+\epsilon)^i$.

---

function UpdateWithoutKnowingOPT(e)
  for each $i \in [\lceil \log_{1+\epsilon} f(e) \rceil, \lfloor \log_{1+\epsilon}(2k \cdot f(e)) \rfloor]$ **do**   ▷ $\frac{(1+\epsilon)^i}{2k} \leq f(e) \leq (1+\epsilon)^i$
    Invoke Update(e) for instance $\mathcal{A}_i$.

3.5 Parameterized Lower Bound

In above, we presented our dynamic 0.5-approximation algorithm that has an amortized query complexity of $O(k \log k)$ if we know the optimal value of the sequence $\mathcal{S}$ after every insertion or deletion, and incurs an extra $O(\log(k)/\epsilon)$-factor in the case that we do not know the optimal value. One may ask if we can obtain a dynamic algorithm for this problem that provides better than 0.5-approximation factor having a query complexity that is still linear, or even polynomial in k. Interestingly, we show there is no dynamic algorithm that maintains a $(0.5+\epsilon)$-approximate submodular solution of the sequence $\mathcal{S}$ using a query complexity that is an arbitrary function $g(k)$ of k (e.g., not even doubly exponentially in k). This hardness holds even when we know the optimal value of the sequence after every insertion or deletion. Thus, the approximation ratio of our parameterized dynamic algorithm is tight. We first state the lower bound due to Chen and Peng (Chen and Peng, 2022, Theorem 1.1) in the following lemma.

Lemma 63 (Theorem 1.1 of (Chen and Peng, 2022))**.** *For any constant $\epsilon > 0$, there is a constant $C_\epsilon > 0$ with the following property. When $k \geq C_\epsilon$, any randomized algorithm that achieves an approximation ratio of $(0.5 + \epsilon)$ for dynamic submodular maximization under cardinality constraint k requires amortized query complexity n^{α_ϵ}/k^3, where $\alpha_\epsilon = \tilde{\Omega}(\epsilon)$ and n is the number of elements in V.*

Chen and Peng proved their theorem by considering a sequence that has the optimal value 1 after every insertion or deletion. Building on this lower bound, we next prove the following theorem.

Theorem 64. *Let $g : \mathbb{N} \to \mathbb{R}^+$ be an arbitrary function. There is no randomized $(0.5 + \epsilon)$-approximate algorithm for dynamic submodular maximization under cardinality constraint k with an expected amortized query time of $g(k)$, even if the optimal value is known after every insertion/deletion.*

Proof. Assume for the sake of contradiction, there exists a constant ϵ, a function $g : \mathbb{N} \to \mathbb{R}^+$, and a $(0.5 + \epsilon)$-approximation algorithm for dynamic submodular maximization with at most $g(k)$ amortized query per insertion/deletion.

According to Lemma 63, there is a constant $C_\epsilon > 0$ such that for all $k > C_\epsilon$ and $n \geq k^{2/\epsilon}$, any $(0.5 + \epsilon)$-approximation algorithm requires at least n^{α_ϵ}/k^3 amortized query. Let k be an arbitrary natural number such that $k > C_\epsilon$, we define $n_0 := \max((g(k) \cdot k^3)^{-\alpha_\epsilon}, k^{2/\epsilon})$. By the definition of n_0, we have $n_0 \geq (g(k) \cdot k^3)^{-\alpha_\epsilon}$, therefore $n_0^{\alpha_\epsilon} \geq g(k) \cdot k^3$, and then $n_0^{\alpha_\epsilon}/k^3 \geq g(k)$. In conclusion, for any $n > n_0$, as $k^2 \leq n^\epsilon$ constraint holds, Lemma 63 implies that the amortized query complexity is at least $n^{\alpha_\epsilon}/k^3 > n_0^{\alpha_\epsilon}/k^3 \geq g(k)$, even if we know the optimal value. Thus, the algorithm requires more than $g(k)$ amortized query complexity which is in contradiction with the assumption.

T

As a result of Theorem 64, for any $\epsilon > 0$, even if k is a constant, the required amortized query complexity to find a $0.5 + \epsilon$-approximate solution increases by increasing n. Therefore, even if we know the optimal value, it is not possible to find a parameterized algorithm with an approximation factor better than 0.5 while query complexity is a function of only k; even if the query complexity is a double exponential of k. For example, if we are looking for an algorithm with amortized query complexity of 2^{2^k}, when n goes up enough, it is not possible to get a $(0.5 + \epsilon)$-approximate solution for any $\epsilon > 0$. Hence, the best approximation ratio of an algorithm that is parameterized by only k is 0.5, even with the assumption of knowing the optimal value. Surprisingly, the parameterized dynamic 0.5-approximation algorithm that we presented above has expected amortized query complexity of $O(k \log k)$ if we know the optimal value.

Bibliography

Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: improved algorithms and lower bounds. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 114–125. ACM, 2019. doi: 10.1145/3313276.3316376. URL `https://doi.org/10.1145/3313276.3316376`.

Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 335–344. IEEE Computer Society, 2016. doi: 10.1109/FOCS.2016.44. URL `https://doi.org/10.1109/FOCS.2016.44`.

Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 440–452. SIAM, 2017. doi: 10.1137/1.9781611974782.28. URL `https://doi.org/10.1137/1.9781611974782.28`.

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. ISBN 0897913973. doi: 10.1145/103418.103437. URL `https://doi.org/10.1145/103418.103437`.

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. doi: 10.1137/S0097539792236237. URL `https://doi.org/10.1137/S0097539792236237`.

Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proceedings of the Second International Conference on Web Search and Web Data Mining, WSDM 2009, Barcelona, Spain, February 9-11, 2009*, pages 5–14. ACM, 2009. doi: 10.1145/1498759.1498766. URL `https://doi.org/10.1145/1498759.1498766`.

Ali Ahmadi, Iman Gholami, MohammadTaghi Hajiaghayi, Peyman Jabbarzade, and Mohammad Mahdavi. 2-approximation for prize-collecting steiner forest. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 669–693. SIAM, 2024a. doi: 10.1137/1.9781611977912.25. URL `https://doi.org/10.1137/1.9781611977912.25`.

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, 2024b. doi: 10.1145/3618260.3649789. URL `https://doi.org/10.1145/3618260.3649789`.

Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 459–467. SIAM, 2012. doi: 10.1137/1.9781611973099.40. URL https://doi.org/10.1137/1.9781611973099.40.

Naor Alaluf, Alina Ene, Moran Feldman, Huy L. Nguyen, and Andrew Suh. Optimal streaming algorithms for submodular maximization with cardinality constraints. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 6:1–6:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ICALP.2020.6. URL https://doi.org/10.4230/LIPIcs.ICALP.2020.6.

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. doi: 10.1137/090771429. URL https://doi.org/10.1137/090771429.

Sepehr Assadi and Sanjeev Khanna. Tight bounds on the round complexity of the distributed maximum coverage problem. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2412–2431. SIAM, 2018. doi: 10.1137/1.9781611975031.155. URL https://doi.org/10.1137/1.9781611975031.155.

Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 815–826. ACM, 2018. doi: 10.1145/3188745.3188922. URL https://doi.org/10.1145/3188745.3188922.

Moshe Babaioff, Nicole Immorlica, David Kempe, and Robert Kleinberg. Matroid secretary problems. *J. ACM*, 65(6):35:1–35:26, 2018. doi: 10.1145/3212512. URL https://doi.org/10.1145/3212512.

Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Streaming submodular maximization: massive data summarization on the fly. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 671–680. ACM, 2014. doi: 10.1145/2623330.2623637. URL https://doi.org/10.1145/2623330.2623637.

Ashwinkumar Badanidiyuru Varadaraja. Buyback problem-approximate matroid intersection with cancellation costs. In *International Colloquium on Automata, Languages, and Programming*, pages 379–390. Springer, 2011.

Egon Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989. doi: 10.1002/net.3230190602. URL https://doi.org/10.1002/net.3230190602.

Maria-Florina Balcan and Nicholas J. A. Harvey. Learning submodular functions. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 793–802. ACM, 2011. doi: 10.1145/1993636.1993741. URL https://doi.org/10.1145/1993636.1993741.

Maria-Florina Balcan and Nicholas J. A. Harvey. Learning submodular functions. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part II*, volume 7524 of *Lecture Notes in Computer Science*, pages 846–849. Springer, 2012. doi: 10.1007/978-3-642-33486-3_61. URL https://doi.org/10.1007/978-3-642-33486-3_61.

Eric Balkanski and Yaron Singer. Approximation guarantees for adaptive sampling. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 393–402. PMLR, 2018a. URL http://proceedings.mlr.press/v80/balkanski18a.html.

Eric Balkanski and Yaron Singer. The adaptive complexity of maximizing a submodular function. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 1138–1151. ACM, 2018b. doi: 10.1145/3188745.3188752. URL https://doi.org/10.1145/3188745.3188752.

Eric Balkanski, Aviad Rubinstein, and Yaron Singer. The limitations of optimization from samples. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1016–1027. ACM, 2017. doi: 10.1145/3055399.3055406. URL https://doi.org/10.1145/3055399.3055406.

Kiarash Banihashem, Leyla Biabani, Samira Goudarzi, MohammadTaghi Hajiaghayi, Peyman Jabbarzade, and Morteza Monemizadeh. Dynamic non-monotone submodular maximization. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023a. URL http://papers.nips.cc/paper_files/paper/2023/hash/387982dbf23d9975c7fc45813dd3dabc-Abstract-Conference.html.

Kiarash Banihashem, Leyla Biabani, Samira Goudarzi, MohammadTaghi Hajiaghayi, Peyman Jabbarzade, and Morteza Monemizadeh. Dynamic non-monotone submodular maximization. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023b. URL https://openreview.net/forum?id=0K1ZTfHZ0N.

Kiarash Banihashem, Leyla Biabani, Samira Goudarzi, Mohammadtaghi Hajiaghayi, Peyman Jabbarzade, and Morteza Monemizadeh. Dynamic constrained submodular optimization with polylogarithmic update time. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 1660–1691. PMLR, 23–29 Jul 2023c. URL https://proceedings.mlr.press/v202/banihashem23a.html.

Kiarash Banihashem, Leyla Biabani, Samira Goudarzi, MohammadTaghi Hajiaghayi, Peyman Jabbarzade, and Morteza Monemizadeh. Dynamic algorithms for matroid submodular maximization. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 3485–3533. SIAM, 2024. doi: 10.1137/1.9781611977912.125. URL https://doi.org/10.1137/1.9781611977912.125.

Olga Barinova, Victor S. Lempitsky, and Pushmeet Kohli. On detection of multiple object instances using hough transforms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(9):1773–1784, 2012. doi: 10.1109/TPAMI.2012.79. URL https://doi.org/10.1109/TPAMI.2012.79.

Surender Baswana. Dynamic algorithms for graph spanners. In *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2006. doi: 10.1007/11841036_10. URL https://doi.org/10.1007/11841036_10.

Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012. doi: 10.1145/2344422.2344425. URL https://doi.org/10.1145/2344422.2344425.

MohammadHossein Bateni and MohammadTaghi Hajiaghayi. Euclidean prize-collecting steiner forest. *Algorithmica*, 62(3-4):906–929, 2012. doi: 10.1007/s00453-011-9491-8. URL https://doi.org/10.1007/s00453-011-9491-8.

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. doi: 10.1145/2027216.2027219. URL https://doi.org/10.1145/2027216.2027219.

MohammadHossein Bateni, Mohammad Taghi Hajiaghayi, and Morteza Zadimoghaddam. Submodular secretary problem and extensions. *ACM Trans. Algorithms*, 9(4):32:1–32:23, 2013. doi: 10.1145/2500121. URL https://doi.org/10.1145/2500121.

MohammadHossein Bateni, Lin Chen, Hossein Esfandiari, Thomas Fu, Vahab S. Mirrokni, and Afshin Rostamizadeh. Categorical feature compression via submodular optimization. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 515–523. PMLR, 2019. URL http://proceedings.mlr.press/v97/bateni19a.html.

Soheil Behnezhad. Dynamic algorithms for maximum matching size. *CoRR*, abs/2207.07607, 2022. doi: 10.48550/arXiv.2207.07607. URL https://doi.org/10.48550/arXiv.2207.07607.

Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 382–405. IEEE Computer Society, 2019. doi: 10.1109/FOCS.2019.00032. URL https://doi.org/10.1109/FOCS.2019.00032.

Marshall W. Bern and Paul E. Plassmann. The steiner problem with edge lengths 1 and 2. *Inf. Process. Lett.*, 32(4):171–176, 1989. doi: 10.1016/0020-0190(89)90039-2. URL https://doi.org/10.1016/0020-0190(89)90039-2.

Aaron Bernstein. Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 693–702. IEEE Computer Society, 2009. doi: 10.1109/FOCS.2009.16. URL https://doi.org/10.1109/FOCS.2009.16.

Aaron Bernstein. *Dynamic Algorithms for Shortest Paths and Matching*. PhD thesis, Columbia University, USA, 2016a. URL https://doi.org/10.7916/D8QF8T2W.

Aaron Bernstein. Dynamic approximate-apsp. In *Encyclopedia of Algorithms*, pages 602–605. 2016b. doi: 10.1007/978-1-4939-2864-4_563. URL https://doi.org/10.1007/978-1-4939-2864-4_563.

Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 167–179. Springer, 2015. doi: 10.1007/978-3-662-47672-7_14. URL https://doi.org/10.1007/978-3-662-47672-7_14.

Aaron Bernstein, Aditi Dudeja, and Zachary Langley. A framework for dynamic matching in weighted graphs. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 668–681. ACM, 2021a. doi: 10.1145/3406325.3451113. URL https://doi.org/10.1145/3406325.3451113.

Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. *ACM Trans. Algorithms*, 17(4):29:1–29:51, 2021b. doi: 10.1145/3469833. URL https://doi.org/10.1145/3469833.

Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 20:1–20:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ICALP.2022.20. URL https://doi.org/10.4230/LIPIcs.ICALP.2022.20.

Sayan Bhattacharya and Peter Kiss. Deterministic rounding of dynamic fractional matchings. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 27:1–27:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.ICALP.2021.27. URL https://doi.org/10.4230/LIPIcs.ICALP.2021.27.

Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 173–182. ACM, 2015. doi: 10.1145/2746539.2746592. URL https://doi.org/10.1145/2746539.2746592.

Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 470–489. SIAM, 2017. doi: 10.1137/1.9781611974782.30. URL https://doi.org/10.1137/1.9781611974782.30.

Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. A new deterministic algorithm for dynamic set cover. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 406–423. IEEE Computer Society, 2019. doi: 10.1109/FOCS.2019.00033. URL https://doi.org/10.1109/FOCS.2019.00033.

Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Xiaowei Wu. Dynamic set cover: Improved amortized and worst-case update time. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2537–2549. SIAM, 2021. doi: 10.1137/1.9781611976465.150. URL https://doi.org/10.1137/1.9781611976465.150.

Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. Fully dynamic $(\Delta+1)$-coloring in $O(1)$ update time. *ACM Trans. Algorithms*, 18(2):10:1–10:25, 2022a. doi: 10.1145/3494539. URL https://doi.org/10.1145/3494539.

Sayan Bhattacharya, Peter Kiss, Thatchaphol Saranurak, and David Wajc. Dynamic matching with better-than-2 approximation in polylogarithmic update time. *CoRR*, abs/2207.07438, 2022b. doi: 10.48550/arXiv.2207.07438. URL https://doi.org/10.48550/arXiv.2207.07438.

Daniel Bienstock, Michel X. Goemans, David Simchi-Levi, and David P. Williamson. A note on the prize collecting traveling salesman problem. *Math. Program.*, 59:413–420, 1993. doi: 10.1007/BF01581256. URL https://doi.org/10.1007/BF01581256.

Jannis Blauth and Martin Nägele. An improved approximation guarantee for prize-collecting TSP. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1848–1861. ACM, 2023. ISBN 978-1-4503-9913-5. doi: 10.1145/3564246.3585159. URL https://doi.org/10.1145/3564246.3585159.

Jannis Blauth, Nathan Klein, and Martin Nägele. A better-than-1.6-approximation for prize-collecting TSP. *CoRR*, abs/2308.06254, 2023. doi: 10.48550/ARXIV.2308.06254. URL https://doi.org/10.48550/arXiv.2308.06254.

Greg Bodwin and Sebastian Krinninger. Fully dynamic spanners with worst-case update time. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.ESA.2016.17. URL https://doi.org/10.4230/LIPIcs.ESA.2016.17.

Niv Buchbinder, Moran Feldman, and Roy Schwartz. Online submodular maximization with preemption. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1202–1216. SIAM, 2015. doi: 10.1137/1.9781611973730.80. URL https://doi.org/10.1137/1.9781611973730.80.

Jaroslaw 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. ISBN 978-1-4503-0050-6. doi: 10.1145/1806689.1806769. URL https://doi.org/10.1145/1806689.1806769.

Larry Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 106–112, 1977. doi: 10.1145/800105.803400. URL https://doi.org/10.1145/800105.803400.

Amit Chakrabarti and Sagar Kale. Submodular maximization meets streaming: Matchings, matroids, and more. *Mathematical Programming*, 154(1):225–247, 2015.

Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPIcs.ICALP.2018.33. URL https://doi.org/10.4230/LIPIcs.ICALP.2018.33.

Anamay Chaturvedi, Huy Le Nguyen, and Lydia Zakynthinou. Differentially private decomposable submodular maximization. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6984–6992. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16860.

Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 370–381. IEEE Computer Society, 2019. doi: 10.1109/FOCS.2019.00031. URL https://doi.org/10.1109/FOCS.2019.00031.

Shiri Chechik and Tianyi Zhang. Dynamic low-stretch spanning trees in subpolynomial time. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA,* *January 5-8, 2020*, pages 463–475. SIAM, 2020. doi: 10.1137/1.9781611975994.28. URL https://doi.org/10.1137/1.9781611975994.28.

Chandra Chekuri and Kent Quanrud. Parallelizing greedy for submodular set function maximization in matroids and beyond. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 78–89. ACM, 2019. doi: 10.1145/3313276.3316406. URL https://doi.org/10.1145/3313276.3316406.

Chandra Chekuri, Jan Vondrák, and Rico Zenklusen. Multi-budgeted matchings and matroid intersection via dependent rounding. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 1080–1097. SIAM, 2011. doi: 10.1137/1.9781611973082.82. URL https://doi.org/10.1137/1.9781611973082.82.

Chandra Chekuri, Shalmoli Gupta, and Kent Quanrud. Streaming algorithms for submodular function maximization. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 318–330. Springer, 2015. doi: 10.1007/978-3-662-47672-7_26. URL https://doi.org/10.1007/978-3-662-47672-7_26.

Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1135–1146. IEEE, 2020. doi: 10.1109/FOCS46700.2020.00109. URL https://doi.org/10.1109/FOCS46700.2020.00109.

Xi Chen and Binghui Peng. On the complexity of dynamic submodular maximization. In *Proceedings of the Fifty-Fourth Annual ACM on Symposium on Theory of Computing, STOC 2022, to appear*, 2022. URL https://arxiv.org/abs/2111.03198.

Yuxin Chen, Hiroaki Shioi, Cesar Fuentes Montesinos, Lian Pin Koh, Serge A. Wich, and Andreas Krause. Active detection via adaptive submodularity. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 55–63. JMLR.org, 2014. URL http://proceedings.mlr.press/v32/chena14.html.

Rajesh Chitnis, Graham Cormode, Hossein Esfandiari, MohammadTaghi Hajiaghayi, Andrew McGregor, Morteza Monemizadeh, and Sofya Vorotnikova. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1326–1344. SIAM, 2016a. doi: 10.1137/1.9781611974331.ch92. URL https://doi.org/10.1137/1.9781611974331.ch92.

Rajesh Chitnis, Marek Cygan, MohammadTaghi Hajiaghayi, Marcin Pilipczuk, and Michał Pilipczuk. Designing fpt algorithms for cut problems using randomized contractions. *SIAM Journal on Computing*, 45(4):1171–1229, 2016b. doi: 10.1137/15M1032077. URL https://doi.org/10.1137/15M1032077.

Rajesh Hemant Chitnis, Graham Cormode, Mohammad Taghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1234–1251. SIAM, 2015. doi: 10.1137/1.9781611973730.82. URL https://doi.org/10.1137/1.9781611973730.82.

Miroslav Chlebík and Janka Chlebíková. The steiner tree problem on graphs: Inapproximability results. *Theor. Comput. Sci.*, 406(3):207–214, 2008. doi: 10.1016/j.tcs.2008.06.046. URL https://doi.org/10.1016/j.tcs.2008.06.046.

Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1158–1167. IEEE, 2020. doi: 10.1109/FOCS46700.2020.00111. URL https://doi.org/10.1109/FOCS46700.2020.00111.

Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

Rafael da Ponte Barbosa, Alina Ene, Huy L. Nguyen, and Justin Ward. The power of randomization: Distributed submodular maximization on massive datasets. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1236–1244. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/barbosa15.html.

Rafael da Ponte Barbosa, Alina Ene, Huy L. Nguyen, and Justin Ward. A new framework for distributed submodular maximization. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 645–654. IEEE Computer Society, 2016. doi: 10.1109/FOCS.2016.74. URL https://doi.org/10.1109/FOCS.2016.74.

Rodney G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer Publishing Company, Incorporated, 2012. ISBN 1461267986.

Delbert Dueck and Brendan J. Frey. Non-metric affinity propagation for unsupervised image categorization. In *IEEE 11th International Conference on Computer Vision, ICCV 2007, Rio de Janeiro, Brazil, October 14-20, 2007*, pages 1–8. IEEE Computer Society, 2007. doi: 10.1109/ICCV.2007.4408853. URL https://doi.org/10.1109/ICCV.2007.4408853.

Paul Duetting, Federico Fusco, Silvio Lattanzi, Ashkan Norouzi-Fard, and Morteza Zadimoghaddam. Deletion robust submodular maximization over matroids. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 5671–5693. PMLR, 2022. URL https://proceedings.mlr.press/v162/duetting22a.html.

David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. Fully dynamic spectral vertex sparsifiers and applications. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 914–925. ACM, 2019. doi: 10.1145/3313276.3316379. URL https://doi.org/10.1145/3313276.3316379.

Paul Dütting, Federico Fusco, Silvio Lattanzi, Ashkan Norouzi-Fard, and Morteza Zadimoghaddam. Fully dynamic submodular maximization over matroids. *arXiv preprint arXiv:2305.19918*, 2023.

Jack Edmonds. Matroids and the greedy algorithm. *Math. Program.*, 1(1):127–136, 1971. doi: 10.1007/BF01584082. URL https://doi.org/10.1007/BF01584082.

Jack Edmonds and Delbert Ray Fulkerson. Transversals and matroid partition. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics*, page 147, 1965.

Soheil Ehsani, MohammadTaghi Hajiaghayi, Thomas Kesselheim, and Sahil Singla. Prophet secretary for combinatorial auctions and matroids. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 700–714. SIAM, 2018. doi: 10.1137/1.9781611975031.46. URL `https://doi.org/10.1137/1.9781611975031.46`.

Khalid El-Arini and Carlos Guestrin. Beyond keyword search: discovering relevant scientific literature. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 439–447. ACM, 2011. doi: 10.1145/2020408.2020479. URL `https://doi.org/10.1145/2020408.2020479`.

Ethan R. Elenberg, Alexandros G. Dimakis, Moran Feldman, and Amin Karbasi. Streaming weak submodularity: Interpreting neural networks on the fly. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4044–4054, 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/c182f930a06317057d31c73bb2fedd4f-Abstract.html`.

Alina Ene and Huy L. Nguyen. Submodular maximization with nearly-optimal approximation and adaptivity in nearly-linear time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 274–282. SIAM, 2019. doi: 10.1137/1.9781611975482.18. URL `https://doi.org/10.1137/1.9781611975482.18`.

Alina Ene and Huy L. Nguyen. Parallel algorithm for non-monotone dr-submodular maximization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 2902–2911. PMLR, 2020. URL `http://proceedings.mlr.press/v119/ene20a.html`.

Alina Ene, Huy L. Nguyen, and László A. Végh. Decomposable submodular function minimization: Discrete and continuous. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 2870–2880, 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/c1fea270c48e8079d8ddf7d06d26ab52-Abstract.html`.

Alina Ene, Huy L. Nguyen, and Adrian Vladu. Submodular maximization with matroid and packing constraints in parallel. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 90–101. ACM, 2019. doi: 10.1145/3313276.3316389. URL `https://doi.org/10.1145/3313276.3316389`.

Stefan Fafianie and Stefan Kratsch. Streaming kernelization. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, volume 8635 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2014. doi: 10.1007/978-3-662-44465-8_24. URL `https://doi.org/10.1007/978-3-662-44465-8_24`.

Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998. doi: 10.1145/285055.285059. URL `https://doi.org/10.1145/285055.285059`.

Moran Feldman, Amin Karbasi, and Ehsan Kazemi. Do less, get more: Streaming submodular maximization with subsampling. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 730–740, 2018. URL `https://proceedings.neurips.cc/paper/2018/hash/d1f255a373a3cef72e03aa9d980c7eca-Abstract.html`.

Moran Feldman, Ashkan Norouzi-Fard, Ola Svensson, and Rico Zenklusen. The one-way communication complexity of submodular maximization with applications to streaming and robustness. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 1363–1374. ACM, 2020. doi: 10.1145/3357713.3384286. URL `https://doi.org/10.1145/3357713.3384286`.

Moran Feldman, Paul Liu, Ashkan Norouzi-Fard, Ola Svensson, and Rico Zenklusen. Streaming submodular maximization under matroid constraints. *arXiv preprint arXiv:2107.07183*, 2021.

Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN 978-3-540-29952-3. doi: 10.1007/3-540-29953-X. URL `https://doi.org/10.1007/3-540-29953-X`.

Fedor V Fomin and Tuukka Korhonen. Fast fpt-approximation of branchwidth. In *Proceedings of 54th Annual ACM Symposium on Theory of Computing (STOC)*, 2022. to appear.

Yu Gao, Yang P. Liu, and Richard Peng. Fully dynamic electrical flows: Sparse maxflow faster than goldberg-rao. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 516–527. IEEE, 2021. doi: 10.1109/FOCS52979.2021.00058. URL `https://doi.org/10.1109/FOCS52979.2021.00058`.

Shayan Oveis Gharan and Jan Vondrák. On variants of the matroid secretary problem. *Algorithmica*, 67(4):472–497, 2013. doi: 10.1007/s00453-013-9795-y. URL `https://doi.org/10.1007/s00453-013-9795-y`.

Michel X. Goemans. Combining approximation algorithms for the prize-collecting TSP. *CoRR*, abs/0910.0553, 2009. URL `http://arxiv.org/abs/0910.0553`.

Michel X. Goemans and David P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995. doi: 10.1137/S0097539793242618. URL `https://doi.org/10.1137/S0097539793242618`.

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. ISBN 978-1-4503-3536-2. doi: 10.1145/2746539.2746590. URL `https://doi.org/10.1145/2746539.2746590`.

Anupam Gupta and Roie Levin. Fully-dynamic submodular cover with bounded recourse. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1147–1157. IEEE, 2020. doi: 10.1109/FOCS46700.2020.00110. URL `https://doi.org/10.1109/FOCS46700.2020.00110`.

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. ISBN 978-0-898716-24-5. URL `http://dl.acm.org/citation.cfm?id=1283383.1283507`.

Anupam Gupta, Aaron Roth, Grant Schoenebeck, and Kunal Talwar. Constrained non-monotone submodular maximization: Offline and secretary algorithms. In *Internet and Network Economics - 6th International Workshop, WINE 2010, Stanford, CA, USA, December 13-17, 2010. Proceedings*, volume 6484 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2010. doi: 10.1007/978-3-642-17572-5_20. URL `https://doi.org/10.1007/978-3-642-17572-5_20`.

Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 537–550. ACM, 2017. doi: 10.1145/3055399.3055493. URL https://doi.org/10.1145/3055399.3055493.

Anupam Gupta, Euiwoong Lee, and Jason Li. An fpt algorithm beating 2-approximation for k-cut. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, page 2821–2837, USA, 2018. Society for Industrial and Applied Mathematics. ISBN 9781611975031.

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. ISBN 0-89871-605-5. URL http://dl.acm.org/citation.cfm?id=1109557.1109626.

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. doi: 10.1145/2390176.2390178. URL https://doi.org/10.1145/2390176.2390178.

MohammadTaghi 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. ISBN 978-3-642-12199-9. doi: 10.1007/978-3-642-12200-2_45. URL https://doi.org/10.1007/978-3-642-12200-2_45.

Kai Han, Zongmai Cao, Shuang Cui, and Benwei Wu. Deterministic approximation for submodular maximization over a matroid in nearly linear time. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/05128e44e27c36bdba71221bfccf735d-Abstract.html.

Nicholas J. A. Harvey, Christopher Liaw, and Tasuku Soma. Improved algorithms for online submodular maximization via first-order regret bounds. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/0163cceb20f5ca7b313419c068abd9dc-Abstract.html.

Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 538–547. IEEE Computer Society, 2013. doi: 10.1109/FOCS.2013.64. URL https://doi.org/10.1109/FOCS.2013.64.

Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. *SIAM J. Comput.*, 45(3):947–1006, 2016a. doi: 10.1137/140957299. URL https://doi.org/10.1137/140957299.

Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrierand derandomization. In *Encyclopedia of Algorithms*, pages 600–602. 2016b. doi: 10.1007/978-1-4939-2864-4_565. URL https://doi.org/10.1007/978-1-4939-2864-4_565.

Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999. doi: 10.1145/320211.320215. URL `https://doi.org/10.1145/320211.320215`.

Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Co., USA, 1996. ISBN 0534949681.

Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142. SIAM, 2013. doi: 10.1137/1.9781611973105.81. URL `https://doi.org/10.1137/1.9781611973105.81`.

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. ISBN 0-306-30707-3. doi: 10.1007/978-1-4684-2001-2_9. URL `https://doi.org/10.1007/978-1-4684-2001-2_9`.

Marek Karpinski and Alexander Zelikovsky. New approximation algorithms for the steiner tree problems. *J. Comb. Optim.*, 1(1):47–65, 1997. doi: 10.1023/A:1009758919736. URL `https://doi.org/10.1023/A:1009758919736`.

Navin Kashyap. Code decomposition: Theory and applications. In *2007 IEEE International Symposium on Information Theory*, pages 481–485, 2007. doi: 10.1109/ISIT.2007.4557271.

Ken-ichi Kawarabayashi and Mikkel Thorup. The minimum k-way cut of bounded size is fixed-parameter tractable. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS '11, page 160–169, USA, 2011. IEEE Computer Society. ISBN 9780769545714. doi: 10.1109/FOCS.2011.53. URL `https://doi.org/10.1109/FOCS.2011.53`.

Ehsan Kazemi, Morteza Zadimoghaddam, and Amin Karbasi. Scalable deletion-robust submodular maximization: Data summarization with privacy and fairness constraints. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2549–2558. PMLR, 2018. URL `http://proceedings.mlr.press/v80/kazemi18a.html`.

Ehsan Kazemi, Marko Mitrovic, Morteza Zadimoghaddam, Silvio Lattanzi, and Amin Karbasi. Submodular streaming in all its glory: Tight approximation, minimum memory and low adaptive complexity. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3311–3320. PMLR, 2019. URL `http://proceedings.mlr.press/v97/kazemi19a.html`.

David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, pages 137–146. ACM, 2003. doi: 10.1145/956750.956769. URL `https://doi.org/10.1145/956750.956769`.

David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. *Theory of Computing*, 11:105–147, 2015. doi: 10.4086/toc.2015.v011a004. URL `https://doi.org/10.4086/toc.2015.v011a004`.

Robert Kleinberg and S. Matthew Weinberg. Matroid prophet inequalities. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 123–136. ACM, 2012. doi: 10.1145/2213977.2213991. URL https://doi.org/10.1145/2213977.2213991.

Robert Kleinberg and S. Matthew Weinberg. Matroid prophet inequalities and applications to multi-dimensional mechanism design. *Games Econ. Behav.*, 113:97–115, 2019. doi: 10.1016/j.geb.2014.11.002. URL https://doi.org/10.1016/j.geb.2014.11.002.

Jochen Könemann, Neil Olver, Kanstantsin Pashkovich, R. Ravi, Chaitanya Swamy, and Jens Vygen. On the integrality gap of the prize-collecting steiner forest LP. In Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, volume 81 of *LIPIcs*, pages 17:1–17:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. ISBN 978-3-95977-044-6. doi: 10.4230/LIPIcs.APPROX-RANDOM.2017.17. URL https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2017.17.

Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 184–192. IEEE, 2021. doi: 10.1109/FOCS52979.2021.00026. URL https://doi.org/10.1109/FOCS52979.2021.00026.

Andreas Krause. Submodularity in machine learning and vision. In *British Machine Vision Conference, BMVC 2013, Bristol, UK, September 9-13, 2013*. BMVA Press, 2013. doi: 10.5244/C.27.2. URL https://doi.org/10.5244/C.27.2.

Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *ACM Trans. Parallel Comput.*, 2(3):14:1–14:22, 2015. doi: 10.1145/2809814. URL https://doi.org/10.1145/2809814.

Lilly Kumari and Jeff A. Bilmes. Submodular span, with applications to conditional data summarization. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 12344–12352. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/17465.

Ron Kupfer, Sharon Qian, Eric Balkanski, and Yaron Singer. The adaptive complexity of maximizing a gross substitutes valuation. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/e56954b4f6347e897f954495eab16a88-Abstract.html.

Branislav Kveton, Zheng Wen, Azin Ashkan, Hoda Eydgahi, and Brian Eriksson. Matroid bandits: Fast combinatorial optimization with learning. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence, UAI 2014, Quebec City, Quebec, Canada, July 23-27, 2014*, pages 420–429. AUAI Press, 2014. URL https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2477&proceeding_id=30.

Silvio Lattanzi, Slobodan Mitrovic, Ashkan Norouzi-Fard, Jakub Tarnawski, and Morteza Zadimoghaddam. Fully dynamic algorithm for constrained submodular optimization. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*

2020, December 6-12, 2020, virtual, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/9715d04413f296eaf3c30c47cec3daa6-Abstract.html.

Silvio Lattanzi, Slobodan Mitrovic, Ashkan Norouzi-Fard, Jakub Tarnawski, and Morteza Zadimoghaddam. Fully dynamic algorithm for constrained submodular optimization. *CoRR*, abs/2006.04704v2, 2023. URL https://arxiv.org/abs/2006.04704v2.

Jon Lee, Maxim Sviridenko, and Jan Vondrák. Matroid matching: the power of local search. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 369–378. ACM, 2010a. doi: 10.1145/1806689.1806741. URL https://doi.org/10.1145/1806689.1806741.

Jon Lee, Maxim Sviridenko, and Jan Vondrák. Submodular maximization over multiple matroids via generalized exchange properties. *Math. Oper. Res.*, 35(4):795–806, 2010b. doi: 10.1287/moor.1100.0463. URL https://doi.org/10.1287/moor.1100.0463.

Maxwell W. Libbrecht, Jeffrey A. Bilmes, and William Stafford Noble. Choosing non-redundant representative subsets of protein sequence data sets using submodular optimization. In *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB 2018, Washington, DC, USA, August 29 - September 01, 2018*, page 566. ACM, 2018. doi: 10.1145/3233547.3233717. URL https://doi.org/10.1145/3233547.3233717.

Hui Lin and Jeff A. Bilmes. A class of submodular functions for document summarization. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, pages 510–520. The Association for Computer Linguistics, 2011. URL https://www.aclweb.org/anthology/P11-1052/.

Paul Liu and Jan Vondrák. Submodular optimization in the mapreduce model. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019, January 8-9, 2019 - San Diego, CA, USA*, volume 69 of *OASICS*, pages 18:1–18:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi: 10.4230/OASIcs.SOSA.2019.18. URL https://doi.org/10.4230/OASIcs.SOSA.2019.18.

Saunders MacLane. Some interpretations of abstract linear dependence in terms of projective geometry. *American Journal of Mathematics*, 58(1):236–240, 1936. ISSN 00029327, 10806377. URL http://www.jstor.org/stable/2371070.

Fumitomo Maeda and Shûichirô Maeda. *Matroid Lattices*, pages 56–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 1970. ISBN 978-3-642-46248-1. doi: 10.1007/978-3-642-46248-1_3. URL https://doi.org/10.1007/978-3-642-46248-1_3.

Dimitris Magos, Ioannis Mourtos, and Leonidas S. Pitsoulis. The matching predicate and a filtering scheme based on matroids. *J. Comput.*, 1(6):37–42, 2006. doi: 10.4304/jcp.1.6.37-42. URL http://www.jcomputers.us/index.php?m=content&c=index&a=show&catid=93&id=1208.

Dániel Marx. Parameterized complexity and approximation algorithms. *The Computer Journal*, 51(1):60–78, 2008.

Andrew McGregor and Hoa T. Vu. Better streaming algorithms for the maximum coverage problem. *Theory Comput. Syst.*, 63(7):1595–1619, 2019. doi: 10.1007/s00224-018-9878-x. URL https://doi.org/10.1007/s00224-018-9878-x.

GEORGE J. MINTY. On the axiomatic foundations of the theories of directed linear graphs, electrical networks and network-programming. *Journal of Mathematics and Mechanics*, 15(3):485–520, 1966. ISSN 00959057, 19435274. URL `http://www.jstor.org/stable/24901346`.

Vahab S. Mirrokni and Morteza Zadimoghaddam. Randomized composable core-sets for distributed submodular maximization. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 153–162. ACM, 2015. doi: 10.1145/2746539.2746624. URL `https://doi.org/10.1145/2746539.2746624`.

Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Deletion-robust submodular maximization: Data summarization with "the right to be forgotten". In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2449–2458. PMLR, 2017. URL `http://proceedings.mlr.press/v70/mirzasoleiman17a.html`.

Baharan Mirzasoleiman, Stefanie Jegelka, and Andreas Krause. Streaming non-monotone submodular maximization: Personalized video summarization on the fly. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1379–1386. AAAI Press, 2018. URL `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17014`.

Morteza Monemizadeh. Dynamic submodular maximization. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/6fbd841e2e4b2938351a4f9b68f12e6b-Abstract.html`.

Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o(n$^{1/2 - \epsilon}$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129. ACM, 2017. doi: 10.1145/3055399.3055447. URL `https://doi.org/10.1145/3055399.3055447`.

Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 950–961. IEEE Computer Society, 2017. doi: 10.1109/FOCS.2017.92. URL `https://doi.org/10.1109/FOCS.2017.92`.

Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016. doi: 10.1145/2700206. URL `https://doi.org/10.1145/2700206`.

George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.*, 14(1):265–294, 1978. doi: 10.1007/BF01588971. URL `https://doi.org/10.1007/BF01588971`.

Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 457–464. ACM, 2010. doi: 10.1145/1806689.1806753. URL `https://doi.org/10.1145/1806689.1806753`.

James G. Oxley. *Matroid theory*. Oxford University Press, 1992. ISBN 978-0-19-853563-8.

Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982. ISBN 0-13-152462-3.

Binghui Peng. Dynamic influence maximization. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 10718–10731, 2021. URL `https://proceedings.neurips.cc/paper/2021/hash/58ec72df0caca51df569d0b497c33805-Abstract.html`.

Goran Radanovic, Adish Singla, Andreas Krause, and Boi Faltings. Information gathering with peers: Submodular optimization with peer-prediction constraints. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1603–1610. AAAI Press, 2018. URL `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16678`.

Monika Rauch. Fully dynamic biconnectivity in graphs. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 50–59. IEEE Computer Society, 1992. doi: 10.1109/SFCS.1992.267819. URL `https://doi.org/10.1109/SFCS.1992.267819`.

Gabriel Robins and Alexander Zelikovsky. Tighter bounds for graph steiner tree approximation. *SIAM J. Discret. Math.*, 19(1):122–134, 2005. doi: 10.1137/S0895480101393155. URL `https://doi.org/10.1137/S0895480101393155`.

Saurabh Sawlani and Junxing Wang. Near-optimal fully dynamic densest subgraph. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 181–193. ACM, 2020. doi: 10.1145/3357713.3384327. URL `https://doi.org/10.1145/3357713.3384327`.

Jacob M. Schreiber, Jeffrey A. Bilmes, and William Stafford Noble. apricot: Submodular selection for data summarization in python. *J. Mach. Learn. Res.*, 21:161:1–161:6, 2020. URL `http://jmlr.org/papers/v21/19-467.html`.

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. URL `http://dl.acm.org/citation.cfm?id=1283383.1283520`.

Ruben Sipos, Adith Swaminathan, Pannaga Shivaswamy, and Thorsten Joachims. Temporal corpus summarization using submodular word coverage. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 754–763. ACM, 2012. doi: 10.1145/2396761.2396857. URL `https://doi.org/10.1145/2396761.2396857`.

Shay Solomon. Fully dynamic maximal matching in constant update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 325–334. IEEE Computer Society, 2016. doi: 10.1109/FOCS.2016.43. URL `https://doi.org/10.1109/FOCS.2016.43`.

Shay Solomon and Nicole Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3): 41:1–41:24, 2020. doi: 10.1145/3392724. URL `https://doi.org/10.1145/3392724`.

Serban Stan, Morteza Zadimoghaddam, Andreas Krause, and Amin Karbasi. Probabilistic submodular maximization in sub-linear time. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3241–3250. PMLR, 2017. URL http://proceedings.mlr.press/v70/stan17a.html.

Ehsan Tohidi, Rouhollah Amiri, Mario Coutino, David Gesbert, Geert Leus, and Amin Karbasi. Submodularity in action: From machine learning to signal processing applications. *IEEE Signal Process. Mag.*, 37(5):120–133, 2020. doi: 10.1109/MSP.2020.3003836. URL https://doi.org/10.1109/MSP.2020.3003836.

Jan van den Brand and Danupon Nanongkai. Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 436–455. IEEE Computer Society, 2019. doi: 10.1109/FOCS.2019.00035. URL https://doi.org/10.1109/FOCS.2019.00035.

Jan van den Brand, Yu Gao, Arun Jambulapati, Yin Tat Lee, Yang P. Liu, Richard Peng, and Aaron Sidford. Faster maxflow via improved dynamic spectral vertex sparsifiers. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 543–556. ACM, 2022. doi: 10.1145/3519935.3520068. URL https://doi.org/10.1145/3519935.3520068.

Kai Wei, Rishabh K. Iyer, and Jeff A. Bilmes. Submodularity in data subset selection and active learning. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1954–1963. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/wei15.html.

Wei Yang, Jeffrey A. Bilmes, and William Stafford Noble. Submodular sketches of single-cell rna-seq measurements. In *BCB '20: 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, Virtual Event, USA, September 21-24, 2020*, pages 61:1–61:6. ACM, 2020. doi: 10.1145/3388440.3412409. URL https://doi.org/10.1145/3388440.3412409.

Alexander Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9 (5):463–470, 1993. doi: 10.1007/BF01187035. URL https://doi.org/10.1007/BF01187035.

Guangyi Zhang, Nikolaj Tatti, and Aristides Gionis. Coresets remembered and items forgotten: submodular maximization with deletions. In *2022 IEEE International Conference on Data Mining (ICDM)*, pages 676–685. IEEE, 2022.