Deterministic Graph Sparsification algorithm of Dynamic fractional matching

Hanumat Lal Vishwakarma
2021csm1019@iitrpr.ac.in
M.Tech CSE, IIT Ropar
Punjab, India
Dr. Anil Shukla
anilshukla@iitrpr.ac.in
Assistant Professor, IIT Ropar
Punjab, India

Abstract

We have developed a new algorithm for sparsifying graphs deterministically in order to maintain dynamic fractional matchings. Dynamic fractional matchings are an important component of the well-known problem of preserving maximum matching on dynamic graphs under edge updates. Before our work, there existed two common rounding methods for dynamic fractional matching that have been applied to various scenarios, both of which were randomized: one by Arar et al. and another by David Wajc in the years 2018 and 2020 respectively.
However, in 2021, Shayan Bhattacharya and Peter Kiss introduced the first deterministic graph sparsification algorithm called "degree_split_subroutine()", which inspired us to develop our own deterministic algorithm. The primary objective of our algorithm is to produce a sparser graph without reducing the size of fractional matching. It is worth noting that before 2021, all existing algorithms for preserving maximum matching on dynamic graphs under edge updates were randomized.
Highlighted Keywords: Fractional matching, Graph sparsification algorithm, Maximum matching, Dynamic graph

1 Introduction

Any sort of graph that experiences changes over time is referred to as dynamic. It is characterized by its capacity to change the number of vertices and edges in its structure. Dynamic graphs are frequently used in complex system modeling, including social, transportation, and communication networks. But creating effective algorithms that can solve issues on dynamic graphs is difficult due to the unpredictability of such graphs. Dynamic graph algorithms, therefore, need to be able to handle a lot of modifications in real-time while preserving a precise representation of the graph. Computer science is actively researching dynamic graphs, which has led to the creation of cutting-edge algorithms and data structures.

A fundamental idea in graph theory known as maximum matching has several uses in engineering and computer science. In order for two edges to be distinct from one another, the biggest collection of edges in a graph must be found. To put it another way, it entails choosing a portion of a graph's edges so that no two edges are next to one another. Due to its many real-world applications, such as task scheduling, resource allocation, and network flow

optimization, maximum matching is a well-researched subject in computer science. The effectiveness of several effective algorithms that may be employed to address the Maximum Matching issue relies on the features of the input graph.

In the field of dynamic algorithms, the main concern is to find an efficient way in order to ensure that a computational problem is consistently solved despite changes in input, it is important to maintain a robust solution. One specific area of focus in the past decade has been on dynamic matching, which involves finding a collection of edges within a graph with no overlapping endpoints. The major goal is to maintain an approximately maximum matching in the dynamic graph where, edges are added or removed at runtime, with a small update time. To achieve this goal, existing algorithms for dynamic matching typically follow a three-step template, which we will discuss further.

In Step (I), The goal here is to create an algorithm that can efficiently handle a maximum approximation fractional matching in a given dynamic graph G=(V, E). For this task, all existing algorithms are deterministic. In this step, we ensure the preservation of a dynamic maximum fractional matching.
In Step (II), it is necessary to maintain a sparse subgraph S=(V, ES) of G that approximately preserves the maximum fractional matching size. The subset of edges ES in S should have the property that the size of the maximum fractional matching in S is nearly equal to the size of the maximum fractional matching w that is obtained in Step (I). This type of subgraph is called a matching-sparsifier. There are two randomized algorithms available for this step, which have been proposed separately by Arar et al. and David Wajc. The objective of this stage is to obtain a well-defined subgraph of the original input graph that maximizes the fractional matching without significantly reducing its size.
In Step (III), Using a technique created by Gupta et al., the goal of Step (III) is to preserve a close match in the bounded-degree sparsifier S that was established in Step (II). On dynamic graphs with a maximum degree less than or equal to d, this deterministic algorithm updates in O(d) time and is deterministic. The update time overhead is very low since S has a finite degree.

It is important to point out that as of 2021, there exists only one deterministic algorithm for the second step. This algorithm was developed by Shayan and Bhattacharya, and all the literature published before 2021 relied on randomized techniques. The question of whether it is feasible to create a successful and deterministic method for Step (II) emerges. The development of an effective deterministic method for Step (II) would eliminate the unpredictability in many present results in the dynamic matching literature because Steps (I) and (III) are already deterministic.

1.1 Fractional Matching

Fractional matching is a well-studied problem in graph theory that deals with finding the largest possible matching in a given graph, where matching is a collection of edges that are non-adjacent to each other. The objective of the fractional matching problem is to identify a matching that maximizes the total weight of its edges. Each edge in the matching is assigned a real number between 0 and 1 as its weight.
Fractional matching has important applications in various fields, including computer science, operations research, and combinatorial optimization. It is used in the design of algorithms

for solving problems such as job scheduling, resource allocation, and network flow maximization. The problem also arises in the study of combinatorial designs and the theory of matroids.
The fractional matching problem can be solved using linear programming techniques, which involve formulating the problem as a linear program and then applying algorithms such as the simplex method to get a solution that is optimal. In addition, several heuristic algorithms have been proposed that is solving the problem efficiently in practical. Fractional matching is a fundamental problem in graph theory and has important applications in various fields. Its study continues to attract the attention of researchers and practitioners alike, and advances in the field are expected to have significant implications for the development of new algorithms and optimization techniques.

Algorithm for Fractional Matching in a Static Setting:
Input: Let G is an input graph defined as G = (V, E), where V is vertex-set and E is edge-set in graph G.
Output: Assign a weight from the range [0,1] to every edge e belonging to edge-set E, such that the total weight of all vertices is at most 1. And maximize their edge weights to obtain the size of fractional matching maximum.
Step 1: Initialize all edge weights to 0 for every edge e in E.
Step 2: Repeat the following steps until a tight node is reached. Continuously increase the edge weights at the same rate and freeze the edges incident on the tight nodes.

The core concept of the algorithm involves allocating weights to the vertices in such a manner that the total weight assigned to all vertices does not exceed 1, while keeping the weights proportional to the lengths of the edges incident to the vertices. This is done by gradually increasing the weights of all edges at the same rate until a tight node is reached.
Let's look at an example to demonstrate the above algorithm as below.

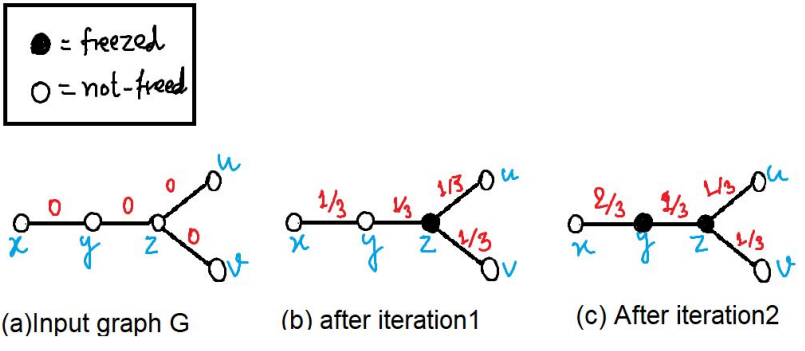

Figure 1: Demonstration of the above algorithm on an example graph.

The computational efficiency of the algorithm is determined by the quantity of iterations necessary to attain a precise node(tight or freezed node). In the worst case, all edges may be tight, and the algorithm would require O(E) iterations. In each iteration, the weight of each edge is updated, which takes O(1) time. Hence, the algorithm's time complexity can be expressed as O(E), where E denotes the number of edges in the input. In terms of space

complexity, the algorithm only requires O(E) space in order to store the weights of all edges. As a result, the space required by the algorithm is proportional to the number of edges in the graph and can be expressed as O(E). The above algorithm is a simple and efficient way to assign weights to the vertices of a graph such that the total weight is at most 1.

Discretizing the above algorithm:

To avoid increasing the edge weight continuously, we propose to raise them to the power of 2 instead. For notation purposes, let L be the logarithm of n, and let G(S) be a subgraph induced by a node-set S, a subset of V. To carry out the algorithm, one must follow these steps in order.

Step 1: One way to initialize the weights of the edges is to set them all to 1/n, where n represents the number of vertices in the set V.
Step 2: Now, Partition the vertex-set V into two sets as S_L and T_L. Here, S_L contains vertices with tight nodes (i.e., nodes with weights in the range [1/2,1]), And T_L contains vertices with non-tight nodes (i.e., nodes with weights in the range [0,1/2]).
Step 3: Identify the edges induced by the node T_L and set their weights to 2*w(e). Then, partition T_L into S_{L-1} and T_{L-1}. Repeat step 3 until all nodes become tight.

Upon analyzing the aforementioned algorithm, it becomes evident that the input graph is divided into subsets T0=S_{-1}, S_0, S_1,.., S_i,..S_j,..S_{logn} consisting of edges with weights of $1/(2^i)$, while edges connecting S_i to S_{i-1} possess edge weights of $1/(2^i)$. T0, being a subset of the graph with an edge weight of $1/(2^0)=1$, will not have any edges as we know for each vertex there must be a weight at most 1. There exist two methods to render the aforementioned algorithm dynamic, namely the Greedy approach and the Lazy approach.

A Greedy Approach for maintaining hierarchical partition Dynamically:

If the weight of a node, denoted as w(v), falls within the interval [0,1], it is deemed as a "clean" node. If the weight is outside this range, the node is considered dirty. At the start, the graph is empty and all nodes are at level 1, hence they are clean. If an edge is added or deleted from the graph, assuming all nodes are clean, on the deletion of an edge, the weights of the endpoints decrease. On insertion of an edge between level i and level j, the weights of both endpoints increase by $1/2^i$, which may result in dirty nodes. To fix dirty nodes, a subroutine called dirty-node-fix subroutines is used. This subroutine increases the level of a node if its weight is too large, and decreases the level if the weight is too small until it becomes clean.
dirty-node-fix subroutine is as followed;

while(dirty node x)
{
1. if w(x) is too large, then l(x)=l(x)+1;where, l(x) =level of node x,w(e)=$1/2^{l(x)}$
2. if w(x) is too small, then l(x)=l(x)-1;
}

A Lazy Approach for maintaining hierarchical partition Dynamically:

The second approach for maintaining a dynamic partition is aimed at keeping the output of the algorithms nearly unchanged when an edge is deleted. Instead of evaluating the output of a given graph G=(V, E) using the original graph, an alternative approach involves utilizing the shadowgraph G*=(V, E U D), which includes the set of dead edges (D) in addition to the set of active edges (E). The algorithm operates as follows: at the preprocessing stage, D is an empty set, and subsequently, when an edge e is removed from E, it is added to D. The invariant condition is that the weight of the dead edges in D is less than or equal to a certain constant multiple of the weight of the active edge e. When this invariant is violated, a subroutine called REBUILD-SUBROUTINE removes some dead edges and re-evaluates the output on the new shadow graph.

2 Problem Statement

The problem at hand is to maintain the maximum matching on a dynamic graph under edge insertion and deletion operations only. A dynamic graph is a graph that undergoes changes to its edge set over time. Edge insertion adds a new edge to the graph, while edge deletion removes an existing edge from the graph. The maximum matching refers to the largest possible set of non-overlapping edges in the graph where each edge in the graph is connected to exactly two vertices, and no two edges have a vertex in common. The goal is to devise an algorithm that can efficiently update the maximum matching of the graph in response to edge insertion and deletion operations while minimizing computational overhead and maintaining a correct solution throughout. The algorithm should work on a wide variety of dynamic graph instances and handle the real-time processing of large-scale graphs.

Before delving into the specific problem, it is necessary to define the main problem, which is "Maintaining Maximum Matching under Edge Updation Only."

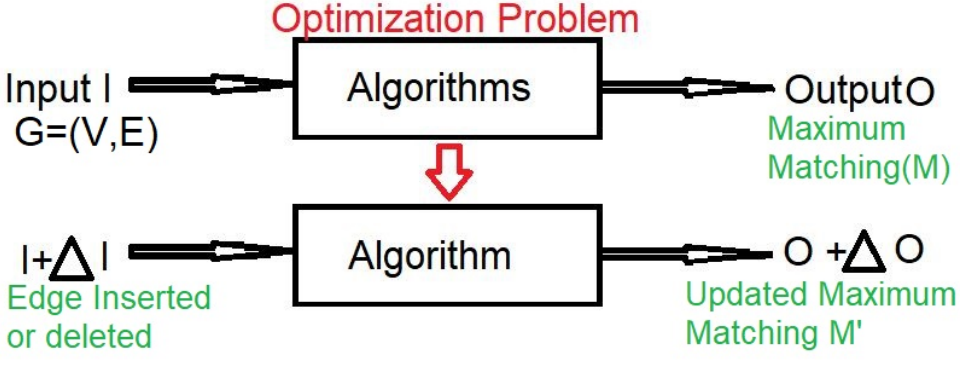

Figure 2: Problem: "Preserving maximum matching in Dynamic graph"

Input: Consider an input graph G=(V,E) where V is the set of vertices and E is the set of edges. Let M denote the maximum matching graph of G.
Output: The objective is to minimize the time taken for edge updates (insertion/deletion) while ensuring that the maximum matching of G is maintained.

One way to solve the problem is to apply the matching algorithm from scratch on the updated graph upon every update, but this approach is time-consuming. Therefore, the idea is to have an updation algorithm that can update the matching simultaneously on every edge update while taking less time (probably polylog n time or linear time).

One Randomized trivial method for keeping track of the maximum matching in a changing graph is a straightforward approach that consists of the following steps:

When an edge (u,v) is added to the graph, the effect on the matching depends on the status of vertices u and v. If neither u nor v is currently matched, the matching M is updated to include (u,v), i.e., M = M U (u,v). However, if either u or v are already matched, the matching remains unchanged.
When an edge (u,v) is deleted from the graph, If (u,v) is not part of the matching M, then M remains unchanged. But if (u,v) is part of M, then M is updated to M = M (u,v), and for every vertex x in u,v that has an unmatched neighbor y, M is updated to M = M U x,y.

The time complexity of the first step (i) is constant, denoted as O(1). The second step (ii) involves calculating the degrees of nodes u and v, which has a time complexity of O(degree(u)) and O(degree(v)), respectively. As a result, the overall time complexity of step (ii) is O(degree(u)) + O(degree(v)) = O(|V|), where |V| is the number of vertices in the graph. Combining the time complexities of both steps, we obtain an overall time complexity of O(|V|). However, this algorithm is randomized since in step (ii), we randomly select one neighbor of u or v after deleting the edge (u,v). Although there is much-existing literature, most of them are randomized, and the current market lacks a deterministic algorithm for preserving the maximum matching, indicating a notable gap in the existing solutions.

The algorithm discussed above is a basic randomized technique employed for maintaining the maximum matching in a dynamic graph. When a new edge is introduced to the graph, its impact on the matching is contingent upon the current matching status of the involved vertices. In cases where both vertices are unpaired, the matching is updated to include the new edge. Conversely, if either vertex is already paired, the matching remains unchanged. On the other hand, when an existing edge is removed from the graph, its effect on the matching depends on whether it is part of the existing matching or not. If the edge is not part of the matching, the matching remains unaltered. However, if the edge is indeed part of the matching, it is removed, and for each vertex that previously had an unmatched neighbor among the endpoints of the removed edge, a new edge is introduced into the matching.

Recent research in this area has adopted a common strategy for addressing this problem, which involves three steps.

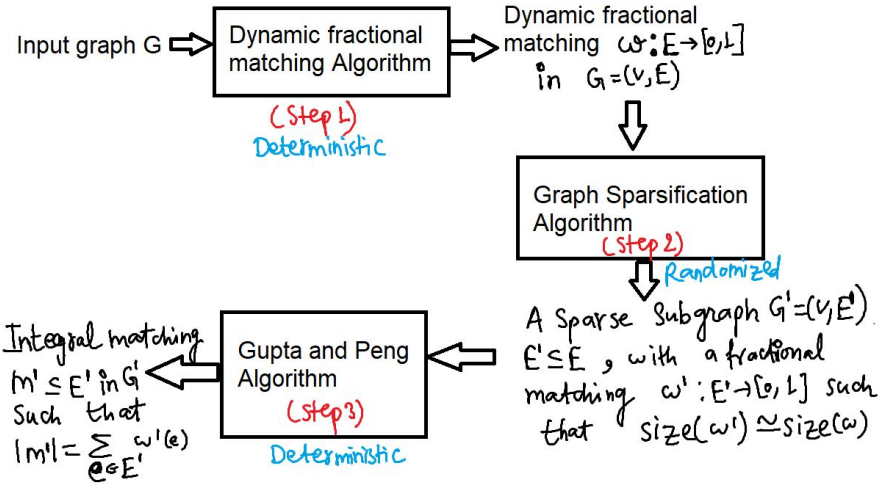

Figure 3: Common three-step strategy followed in recent literature for the problem "Preserving maximum matching in Dynamic graph".

There is an existing deterministic primal-dual algorithm for step 1, and Gupta and Peng's algorithm from 2013 can be used for step 2. In this three-step strategy, steps 1 and 3 are already deterministic, while step 2 is randomized. In 2021, Shayan Bhattacharya and Peter Kiss introduced the first deterministic graph sparsification algorithm called the "degree-split subroutines." This inspired us to develop our own deterministic graph sparsification algorithm specifically for step 2 of the three-step strategy.

Let's define the problem statement on which we are working;

Input: λ-uniform weighted graph G=(V,E), with fractional matching w: E->[0,1].
Output: 2λ-uniform weighted graph G'=(V,E'), E' is a subset of with fractional matching w': E'->[0,1] such that size(w')=size(w), where size(w) is defined as the sum of all the edge-weights in graph G.

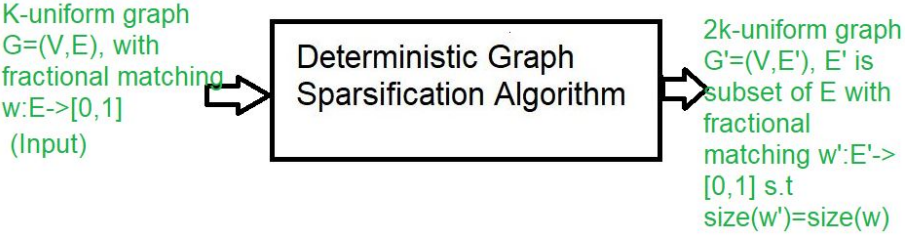

Figure 4: Our Problem Statement(Step 2 of common strategy mentioned above). We want to design a deterministic graph sparsification algorithm for step 2.

3 Literature Review

There existed two common rounding methods for dynamic fractional matching that have been applied to various scenarios, both of which were randomized: one by Arar et al. and another by David Wajc in the years 2018 and 2020 respectively.

Arar et al. [ICALP'18] developed a randomized rounding scheme that is based on a combination of the primal-dual algorithm and a random walk approach. Their approach has two main phases. In the first phase, the primal-dual algorithm is used to find a fractional matching. In the second phase, a random walk is conducted on the graph, and edges are included in the matching based on the random walk's outcome. The probability of selecting an edge is proportional to the fractional value of the edge. The algorithm repeats this process multiple times to improve the approximation ratio. Their algorithm has a time complexity of O(n^2 log^2 n log(1/Epsilon)), where n represents the number of elements in vertex-set V, and Epsilon denotes the error probability.

Wajc [STOC'20] proposed a randomized scheme that involves finding a fractional matching followed by a "random sampling" phase. In the first phase, fractional matching is found using an algorithm such as the primal-dual algorithm. In the second phase, edges are sampled randomly based on their fractional value. The probability of selecting an edge is proportional to the square of its fractional value. The algorithm repeats this process multiple times to improve the approximation ratio. The time complexity of their algorithm is O(n^2 log(1/Epsilon)), where n represents the number of elements in vertex-set V, and Epsilon denotes the error probability.
Two popular randomized rounding schemes were proposed in previous work: one by Arar et al. in their paper published at ICALP'18, and another by David Wajc in his paper published at STOC'20 that involve finding a fractional matching and then using randomness to select edges for inclusion in the maximum matching. While Arar et al.'s algorithm uses a combination of the primal-dual algorithm and random walks, Wajc's algorithm involves random sampling. Both algorithms have a similar time complexity of O(n^2 log(1/Epsilon)), with Arar et al.'s algorithm having an additional factor of log^2 n.

In 2021, **Shayan Bhattacharya and Peter Kiss** proposed the first deterministic graph sparsification algorithm, which was a significant advancement in the field of graph algorithms. Here is a more detailed explanation of their algorithm along with its time complexity. Their algorithm called the degree-split subroutine, is based on a simple idea of splitting vertices of high degree into multiple copies while preserving the structure of the original graph. The algorithm first assigns a weight to each vertex based on its degree. It then splits each vertex with degree d into d copies, each having a weight of 1/d. The algorithm then selects a subset of edges from the original graph such that each vertex copy has a degree at most O(polylog(n)) in the subgraph. The selected edges are then used to form the output sparsified graph. The degree-split subroutine has several desirable properties. It is deterministic, which means that the output graph is identical for the same input graph and the same random seed. It also has a provable guarantee on the size of the sparsified graph, which is within a constant factor of the size of the input graph. Moreover, it can be efficiently implemented, with a time complexity of O(m polylog(n)), where m represents the number of elements in edge-set E and n represents the number of elements in vertex-set V. The degree-split subroutine has several potential applications in the field of dynamic graph algorithms. It can be used as a build-

ing block for constructing more complex algorithms that require sparsification of the graph. For example, it can be used in the three-step template proposed by Gupta and Peng (2013) for maintaining maximum matching on a dynamic graph. The degree-split subroutine can replace the randomized sparsification step in their template, resulting in a fully deterministic algorithm. In conclusion, the degree-split subroutine proposed by Bhattacharya and Kiss is the first deterministic graph sparsification algorithm. It is based on a simple idea of splitting high-degree vertices and has several desirable properties, including a provable guarantee on the size of the sparsified graph and an efficient implementation with a time complexity of O(m polylog(n)). It has potential applications in the field of dynamic graph algorithms and can be used as a building block for constructing more complex algorithms.

4 First Deterministic Solution

Until 2021, existing literature on graph sparsification algorithms was based on randomness and probability concepts. However, in 2021, Shayan Bhattacharya and Peter Kiss introduced the first deterministic graph sparsification algorithm, known as the degree-split subroutine. The problem of maintaining maximum matching in a dynamic graph is crucial, and this algorithm offers a fully deterministic solution to it, which was previously solved using randomized algorithms.

The degree-split subroutine is specifically designed to operate on a dynamic weighted graph G=(V, E), where G is a uniform graph with weight λ to each edge with size(w) as the size of fractional matching. Its primary objective is to produce a sparsified graph G'=(V, E'), where G' is a uniform graph with weight 2λ to each edge with size(w') as the size of fractional matching, ensuring that the size of w' remains the same as w. The key idea behind this subroutine is to remove half of the edges for each vertex in G and simultaneously increase the weights of the remaining edges by a factor of 2 in G'. By doing so, the size of the fractional matching is preserved, while reducing the number of edges in the graph. In G', the degree of each vertex is either (degree(G)/2)+1 or (degree(G)/2)-1, introducing a slack of 1 or -1. The algorithm for the degree-split subroutine consists of four steps: first, partition E into a set of disjoint maximal walks; second, for each walk Wi, collect the alternate edges to form Wi*; third, let E' be the union of all the sets Wi*; and finally, return the sparsified graph G'=(V, E).

The time complexity of the degree-split subroutine can be analyzed as follows:

Step 1: Partitioning E into a set of maximal walks

To partition the edges into maximal walks, we can perform a depth-first search (DFS) traversal of the graph starting from each unmatched vertex. The DFS takes O(|E| + |V|) time in total, since each edge and vertex is visited at most twice during the traversal.

Step 2: Collecting alternate edges for each walk

For each maximal walk Wi, we can collect its alternate edges using a hash table or an array. The expense of gathering alternate edges for a walk is directly proportional to the degree of the vertices within the walk, with a maximum degree of k, representing the largest possible fractional matching size. As a result, the total time for this step is O(k|Wi|).

Step 3: Computing the union of all the sets Wi

To compute the union of all the sets Wi*, we can use a hash table or a union-find data structure. The cost of adding an edge to or looking up an edge in the data structure is proportional to the length of the walk it belongs to, which is at most 2k. Therefore, this step takes O(k|E|) time in total.

Step 4: Sorting the edges by their endpoints
To construct the sparsified graph G', we need to sort the edges by their endpoints. We can use a priority queue to perform this step in O(|E| log |V|) time, since there are at most |E| edges and each edge is inserted into the priority queue once.

The time complexity of the degree-split subroutine can be expressed as O(|E| log |V| + k|E|), where |E| is the number of edges in the graph, |V| is the number of vertices, and k is a constant, which is dominated by the first term for sufficiently large values of |V| and |E|.
Implications of the research include the potential introduction of multiple new deterministic algorithms for dynamic sparsification. Additionally, there may be opportunities for the derandomization of existing randomized algorithms.

Proof of Correctness of degree-split subroutine:

Let W be the set of all maximal walks in G. By construction, every edge in E is included in exactly one walk in W.
Step 1: First, we divide the edge set E into a collection of disjoint maximal walks. To achieve this, we partition E into a set of such walks, which we denote by W This means that every edge is in exactly one maximal walk.
Step 2: For each walk Wi, collect the alternate edges to form Wi*.
For each maximal walk Wi in W, we collect alternate edges from Wi to form a set Wi*. Since the vertices in Wi have alternating degrees, it is possible to collect alternate edges from Wi in this way.
Step 3: Let E' be the union of all the sets Wi*.
We let E' be the union of all sets Wi*. By construction, each edge in E is in exactly one walk, so each edge in E appears in exactly one set Wi*.
Step 4. The resulting output is the sparsified graph, denoted as G'=(V,E'). By construction, G' has at most half as many edges as G, since we remove half of the edges incident to each vertex. Furthermore, the degree of every vertex in the graph G' is either (degree(G)/2)+1 or (degree(G)/2)-1, which implies that G' possesses a 2λ-uniform fractional matching w':E'->[0,1], where size(w')=size(w).

Thus, the degree-split subroutine algorithm generates a sparse graph G'=(V,E') along with a 2λ-uniform fractional matching w':E'->[0,1] such that the size of w' is equal to the size of w, thereby validating the algorithm's correctness.

5 Our Technique

After examining the degree-split idea, we decided to remove half of the edges of input graph G and doubled the edge weights for the surviving edges. However, the challenge was to ensure that the algorithm was deterministic, meaning that the selection of edges must be a deterministic choice. We could not rely on random sampling, probability, or threshold concepts.
The input for the algorithm is a weighted graph G=(V,E), where V=1,2,...V and a λ-uniform fractional matching w: E -> [0,1]. The output is a graph G'=(V,E'), where E' is a set of edges with a 2λ-uniform fractional matching w': E -> [0,1] that maintains the same w (i.e. size(w')=size(w)); where, size(w)=sum of all edge-weights in G.

For simplicity, We can assume the weights assigned to each edge in the input graph G must be set to a value of 1 /max-degree(G). Therefore, input graph G is 1/max-degree(G)-uniform.

The algorithm proceeds as follows:

1. For each vertex v in V, create a list L-v of its neighbors in non-increasing order of their degree.
2. Initialize E' as an empty set of edges
3. For each vertex v in V:
a. Compute k=ceil(degree(v)/2), where degree(v) is the degree of the vertex in G.
b. If v has already been relaxed, proceed to the next vertex (i.e., degree(v) is already sparsified).
c. Initialize S as an empty set of vertices containing the selected neighbors of v.
d. For each neighbor u in L-v:
i. If u<v and the edge u,v has not been added to E' yet, then continue to the next neighbor.
ii. If u<v and the edge u,v is already present in E', then reduce k by 1 and continue to the next neighbor.
iii. If u>=v and |S| <= k, add u to S.
iv. If |S| = k, break the loop.
e. Add the edge v,u for each u in S to the edge set E' of G'.
4. Return the new graph G' with vertex set V and edge set E', and double the weight of each edge in the graph G'.

Let's demonstrate our proposed technique with some of examples;

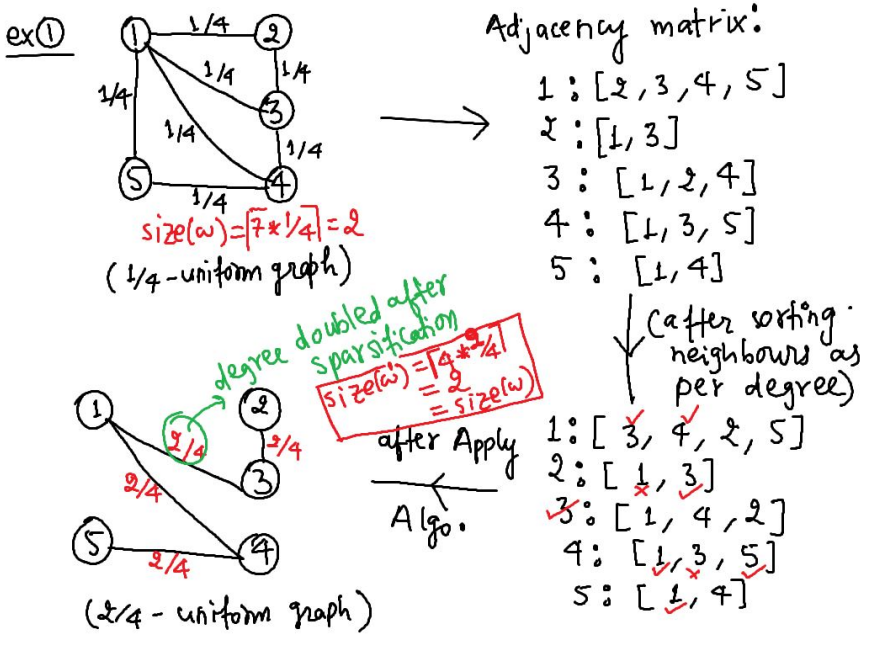

Figure 5: This is example 1 on which we were applying our proposed technique.

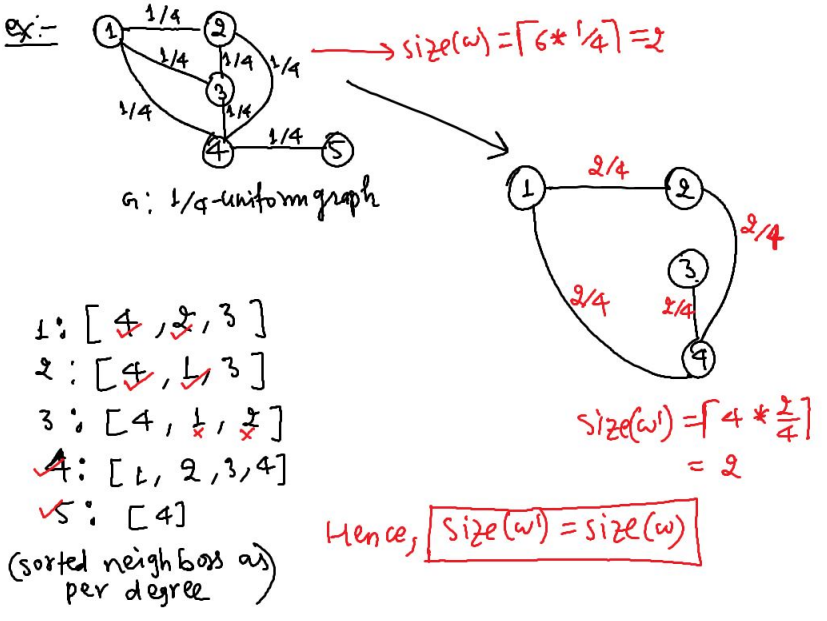

Figure 6: This is example 2 on which we were applying our proposed technique.

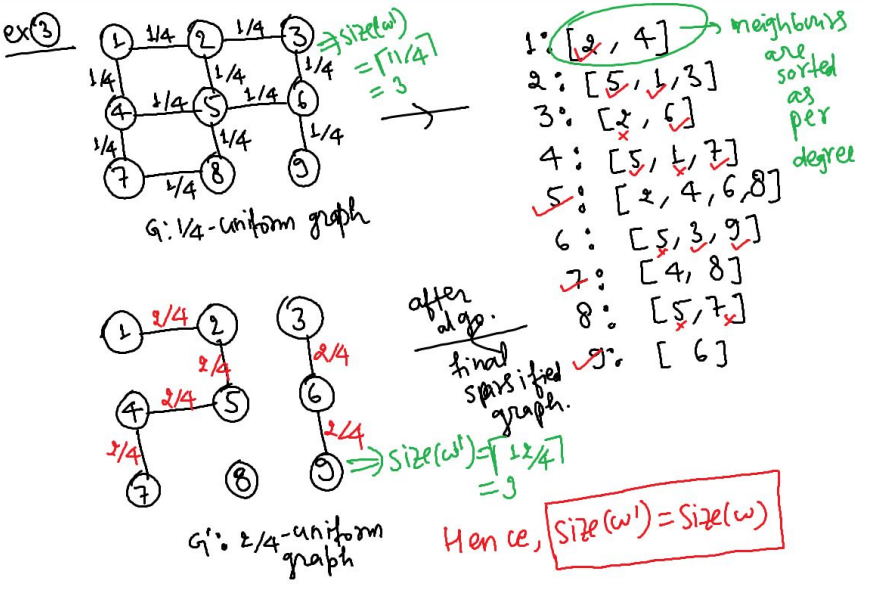

Figure 7: This is example 3 on which we were applying our proposed technique.

Time and Space Complexity Analysis of Our Technique:

Time complexity Analysis is as followed;

Step 1: For each vertex v, the algorithm sorts its neighbors in non-increasing order of their degree. The time complexity of this step is O(|E| log |V|) since it involves sorting the neighbors of each vertex, and the total number of neighbors in the graph is |E|.
Steps 2-4: For each vertex v, the algorithm selects a set of neighbors S to keep in the sparsified graph G'. This involves iterating over the neighbors of v in L-v and performing constant-time operations, such as checking whether an edge has already been added to E', adding a neighbor to S, and breaking the loop when |S| = k. Since each edge is considered at most twice, once for each of its endpoints, the total number of operations in Steps 2-4 is O(|E|).

Thus, the overall time complexity of the algorithm is O(|E| log |V|), which dominates the O(|E|) time complexity of the degree-split algorithm. The reason for the higher time complexity is the sorting step in Step 1, which is necessary to ensure deterministic selection of the neighbors to keep in the sparsified graph. However, the O(|E| log |V|) time complexity is still efficient for most practical graphs, and the algorithm provides a guarantee of deterministic sparsification without using random sampling or probability.

Space Complexity Analysis is as followed;

The algorithm stores the input graph G=(V,E) and its λ-uniform fractional matching w, which requires O(|E|) space for storing the edges and O(|E|) space for storing the fractional matching.
For each vertex v in V, the algorithm creates a list L-v of its neighbors in non-increasing order of their degree, which requires O(|E|) space for storing the lists.
The algorithm also stores the sparsified graph G'=(V,E') and its 2λ-uniform fractional matching w', which requires O(|E|) space for storing the edges and O(|E|) space for storing the fractional matching.

The space complexity of the algorithm is, therefore, O(|E|) for storing the input graph and its matching, O(|E|) for storing the neighbor lists, and O(|E|) for storing the sparsified graph and its matching, for a total of O(|E|) space complexity.
Therefore, the space complexity of the algorithm is linear in the size of the input graph, which is efficient for most practical graphs.

A Formal Proof of Correctness of Our Algorithm:

In order to establish the algorithm's validity, it is necessary to demonstrate the following:
1. The size of the fractional matching w' must be identical to that of w. 2. The proposed algorithm is determinate.

Proof:
1. To prove that the size of fractional matching is maintained, we need to show that the sum of the edge weights in G' is equal to the sum of the edge weights in G. Let's consider the edge weight of the sparsified graph G'. The weight of each edge in G' is twice the weight of the corresponding edge in G. Since we are doubling the weight of all the edges in G, the sum

of the edge weights in G' is equal to 2 times the sum of the edge weights in G. Therefore, the size of fractional matching in G' is the same as in G, i.e., size(w')=size(w).

2. To prove that the proposed algorithm is deterministic, we need to demonstrate that the output graph G' with its edges E' is uniquely determined by the input graph G and its λ-uniform fractional matching w.

The algorithm starts by computing a list L-v of the neighbors of each vertex v in non-increasing order of their degree, a step that is deterministic and will generate the same list for each vertex v in every execution of the algorithm.
The algorithm then initializes E' as an empty set of edges, which is also a deterministic step that will result in an empty set of edges in every execution of the algorithm.

For each vertex v in V, the algorithm selects a set of edges to add to E', depending on the degree of vertex v, the fractional matching w, and the list L-v. Since the degree of v, the fractional matching w, and L-v are fixed inputs, the selection of edges for each vertex v is deterministic.

Specifically, the algorithm selects the top k=ceil(degree(v)/2) neighbors of vertex v from the list L-v, where k is a fixed value that depends only on the degree of vertex v. The algorithm selects these neighbors in a deterministic way by iterating through the list L-v from the beginning until it has selected k neighbors or exhausted the list. If a neighbor u<v and the edge u,v has not been added to E' yet, then the algorithm skips this neighbor and continues to the next neighbor. If a neighbor u<v and the edge u,v is already present in E', then the algorithm reduces k by 1 and continues to the next neighbor. If a neighbor u>=v and |S|<=k, where S is the set of selected neighbors, then the algorithm adds u to S. If |S|=k, the loop is terminated.

Since the selection of edges for each vertex v is deterministic, the resulting set of edges E' is also deterministic. The output graph G' with vertex set V and edge set E' is also uniquely determined by the input graph G and its λ-uniform fractional matching w. Finally, the weights of the edges in G' are doubled, which is a deterministic operation. Therefore, the proposed algorithm is deterministic.

6 Conclusions

In conclusion, we have presented a new deterministic algorithm for sparsifying graphs while preserving dynamic fractional matchings. There exist two well-known general-purpose rounding schemes for dynamic fractional matchings that are Arar et al. and Wajc. Both algorithms are randomized, and there is no deterministic algorithm for preserving maximum matching on dynamic graphs under edge updates before 2021.
Our technique is the second deterministic algorithm for dynamic fractional matching after the "degree_split_subroutine()" introduced by Shayan Bhattacharya and Peter Kiss in 2021. The algorithm produces a sparser graph without reducing the size of fractional matching, unlike all existing algorithms for preserving maximum matching on dynamic graphs under edge updates before 2021, which were randomized.

Our algorithm selects edges deterministically by iterating over the neighbors of each vertex and removing nearly half of its edges to reduce its degree nearly by half. The primary objective in sparsifying graphs is to ensure that the fractional matching size does not reduce significantly. It is crucial to maintain the fractional matching size of the original input graph in the sparsified graph.
The time complexity of the algorithm is O(|E| log |V|), dominated by the sorting step in Step 1, which ensures the deterministic selection of the neighbors to keep in the sparsified graph. However, the O(|E| log |V|) time complexity is still efficient for most practical graphs, and the algorithm provides a guarantee of deterministic sparsification without using random sampling or probability.

Dynamic graphs, which undergo changes over time, are extensively utilized in complex system modeling. Efficient algorithms are necessary to address issues related to dynamic graphs while maintaining a precise depiction of the graph. Future work includes exploring the possibility of improving the time complexity of the algorithm by optimizing the sorting step in Step 1 or by introducing parallelization techniques. Also, to explore more deterministic algorithms for preserving maximum matching on dynamic graphs under edge updates and improve the time complexity of existing algorithms. Another direction of future research is to investigate the application of our algorithm to other problems that require deterministic graph sparsification, such as preserving other graph properties or designing deterministic graph algorithms.

7 References

1. Bhattacharya, S., Kiss, P. (2021, May 4). Deterministic Rounding of Dynamic Fractional Matchings. Paper presented at the International Colloquium on Automata, Languages, and Programming (ICALP).

2. Wajc, D. (2020). Rounding Techniques for Dynamic Fractional Matchings. In Proceedings of the 52nd Annual Symposium on Theory of Computing (STOC).

3. Arar, M. A., Mastrolilli, M., & Liaghat, V. (2018). General-Purpose Rounding Schemes for Dynamic Fractional Matchings. In Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP).

4. Bhattacharya, S., Kiss, P., Saranurak, T., Wajc, D. (2022, November 9). Dynamic Matching with Better-than-2 Approximation in Polylogarithmic Update Time.

5. Sleator, D. D., Tarjan, R. E. (1985). Self-Adjusting Binary Search Trees. Journal of the Association for Computing Machinery, 32(3), 652-686.

6. Vazirani, V. V. (2022). Online Bipartite Matching and Adwords.

7. Karp, R. M., Vazirani, U. V., Vazirani, V. V. (1990). An optimal algorithm for online bipartite matching. In Proceedings of the Symposium on the Theory of Computing (STOC) (pp. 352-358). ACM.