Fully Dynamic Randomized Algorithms for Graph Spanners *

Surender Baswana[†] Sumeet Khurana [‡] Soumojit Sarkar[§]

Abstract

Spanner of an undirected graph $G = (V, E)$ is a subgraph which is sparse and yet preserves all-pairs distances approximately. More formally, a spanner with *stretch* $t \in \mathbb{N}$ is a subgraph $(V, E_S), E_S \subseteq E$ such that the distance between any two vertices in the subgraph is at most t times their distance in G. Though G is trivially a t-spanner of itself, the research as well as applications of spanners invariably deal with a t-spanner which has as small number of edges as possible.

We present fully dynamic algorithms for maintaining spanners in centralized as well as synchronized distributed environments. These algorithms are designed for undirected unweighted graphs and use randomization in a crucial manner.

Our algorithms significantly improve the existing fully dynamic algorithms for graph spanners. The expected size (number of edges) of a t-spanner maintained at each stage by our algorithms matches, up to a poly-logarithmic factor, the worst case optimal size of a t-spanner. The expected amortized time (or messages communicated in distributed environment) to process a single insertion/deletion of an edge by our algorithms is *close* to optimal.

---

*The results of the preliminary version of this article appeared in ESA 2006 and SODA 2008

[†]Dept. of Comp. Sc. and Engg., IIT Kanpur, India. Email : sbaswana@cse.iitk.ac.in. This work was supported by a Young Faculty Chair Award from Research I Foundation, CSE, IIT Kanpur.

[‡]Dept. of Comp. Sc. and Engg., IIT Kanpur, India. Email : skhurana.net@gmail.com.

[§]Department of Comp. Sc., University of Texas at Austin, Texas 78712, USA. Email : soumojitsarkar@gmail.com. This Work was done while the author was an M.Tech student at CSE, IIT Kanpur, India.

1 Introduction

There are numerous well known algorithmic graph problems which aim at computing a subgraph of a given graph with certain properties. These properties are usually defined by some function which one tries to optimize for the given graph. For example, the objective of the minimum spanning tree problem is to compute a connected subgraph having the least total weight. The objective of the single source shortest paths problem is to compute a rooted tree which stores the shortest paths from a given source vertex to all the vertices in the graph. In this paper, we consider one such well known subgraph called *spanner*, and design its efficient dynamic algorithms. Let $G = (V, E)$ be an undirected graph with nonnegative weights on edges and $t \geq 1$ be any integer. A t-spanner of the graph $G = (V, E)$ is a subgraph $H = (V, E_S)$ such that for all vertices $u, v \in V$,

$$\delta_G(u, v) \leq \delta_H(u, v) \leq t\delta_G(u, v) \tag{1}$$

where δ_G is the distance in graph G. Though G is trivially a t-spanner of itself, we are invariably interested in a t-spanner which has the smallest number of edges. So a spanner can also be viewed as a subgraph which is *sparse* and yet preserves the all-pairs distance information of the given graph *approximately.* The parameter t associated with a t-spanner is called the *stretch* of the spanner. The number of edges in the spanner is called the *size* of the spanner.

In addition to being beautiful mathematical objects, spanners are useful in various domains of computer science. They are the basis of space-efficient routing tables that guarantee nearly shortest routes [1, 16, 17, 38, 47, 41], schemes for simulating synchronized protocols in unsynchronized networks [38]. Spanners are also used in parallel and distributed algorithms for computing approximate shortest paths [13, 21]. A recent application of spanners is the construction of labeling schemes and distance oracles [8, 5, 42, 47], which are the data structures that can report approximately accurate distances in constant time.

Ever since the notion of spanner was defined formally by Peleg and Schaffer [37] in 1989, the research area of spanner has witnessed many important results. Moreover, many new structures similar to spanners have also evolved. One such structure is additive spanner. A subgraph H is said to be additive b-spanner for graph G if $\delta_H(u, v) \leq \delta_G(u, v) + b$. Note that a t-spanner as defined in Equation 1 can thus be seen as a *multiplicative* t-spanner. For additive spanners, the reader is referred to the work of Baswana et al. [6] and Woodruff [49]. Elkin and Peleg [27] introduced the notion of $(1 + \epsilon, \beta)$ spanner which is a hybrid of multiplicative and additive spanner. Bollobás et al. [11] introduced the notion of distance preserver. A subgraph is said to be a d-preserver if it preserves exact distance for each pair of vertices which are separated by distance at least d. Coppersmith and Elkin [14] study different notions of distance preservers. Thorup and Zwick [48] introduced a variant of additive spanners where the additive error is sublinear in terms of the distance being approximated. Though there are so many diverse applications of spanners and related structures, each of them have a common algorithmic goal - to efficiently design or maintain the sparsest spanner of a given additive and/or multiplicative stretch for a given graph. In this paper, we focus only on multiplicative t-spanner. Henceforth we shall omit the word multiplicative while dealing with t-spanner. We shall use n and m to denote the number of vertices and edges respectively of the given undirected graph.

For computing a t-spanner, the algorithm of Althöfer et al. [2] is well known. This algorithm is similar in spirit to the well known algorithm of Kruskal [35] for computing minimum spanning tree. For any positive integer k, this algorithm [2] computes a $(2k-1)$-spanner of $O(n^{1+1/k})$ edges. Based on a 48 year old girth conjecture of Erdős [28], this upper bound on the size of $(2k-1)$-spanner is worst-case optimal: there are graphs whose sparsest $(2k-1)$-spanner and $2k$-spanner have $\Omega(n^{1+1/k})$ edges. However, the best known implementation of the algorithm of Althöfer [2] has $O(\min(kn^{2+1/k}, mn^{1+1/k}))$ running time [2, 44]. For the special case of unweighted graphs, Halperin and Zwick [32] designed an $O(m)$ time algorithm to compute a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges. For general graphs, Baswana and Sen [8] designed a randomized algorithm which takes expected $O(km)$ time to compute a $(2k-1)$-spanner of $O(kn^{1+1/k})$ size. This algorithm takes a very simple and local approach. As a result, it has been adapted easily to design near optimal algorithms for spanners in external memory, parallel, and distributed computational environments as well [8]. The algorithm was later derandomized by Roditty et al. [42].

In addition to the centralized algorithms mentioned above, many efficient distributed algorithms have also been designed for computing spanner. These distributed algorithms are motivated by various applications of spanners in distributed computing. The centralized algorithm of Baswana and Sen [8] can be adapted easily in distributed environment to construct a $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size in $O(k)$ rounds and $O(km)$ communication complexity. Derbel et al. [19] designed a deterministic distributed algorithm with matching efficiency in synchronous environment. Note that the size of the sparsest spanner computed by these algorithms is of the order of $n \log n$ and it is achieved for stretch $\log n$. For computing an $O(n)$ size spanner with stretch $\log n$, Dubhashi et al. [20] and Pettie [40] independently designed algorithms which run in $O(\text{polylog}\, n)$ rounds.

Though various efficient algorithms exist for computing spanners and related structures for a given graph, these algorithms are not suitable for many real life graphs which are dynamic in their nature. These graphs undergo updates which are insertion and deletion of edges. A naive way to handle an update is to execute the best static algorithm for the given problem from scratch after each update. Certainly this approach seems quite wasteful because the new solution might not be much different from the previous one. This necessitates the design of a separate dynamic algorithm for a graph problem in addition to its static counterpart. The aim of such a dynamic algorithm is to maintain some efficient data structure such that the updates can be processed in time much faster than the best known static algorithm. Many dynamic algorithms have been designed in the past for various fundamental graph problems, namely, undirected connectivity [31, 33, 34], all-pairs shortest paths [18], transitive closure [45, 43]. Following the terminologies of any other dynamic graph problem, the problem of maintaining spanners in dynamic graphs can be stated formally as follows.

Given an undirected graph $G = (V, E)$ under any arbitrary sequence of updates - insertion and deletion of edges, design an efficient algorithm to maintain a $(2k-1)$-spanner of the graph in an online fashion with the aim to preserve $O(n^{1+1/k})$ bound on the size of the spanner.

The measure of efficiency of a dynamic algorithm is the time required (or messages communicated in distributed environment) to update the spanner upon insertion or deletion of an edge. This measure is commonly called the *update time* of the dynamic algorithm. Given the existence of a linear time static algorithm to compute a $(2k-1)$-spanner in centralized environment, an ideal goal of a fully dynamic algorithm would be to achieve constant or poly-logarithmic update time per edge insertion or deletion. Similarly an ideal goal of a fully dynamic algorithm in distributed environment would be to achieve constant communication complexity per edge insertion or deletion. Naturally, such a goal may look too ambitious, if not counter-intuitive, as there is no apparent way to argue that a single update in the graph will lead to a *small* change in the spanner. For example, upon deletion of an edge in the spanner, it is not obvious that we need to add just a few edges of the graph to the spanner to preserve its stretch, leave aside the issue of finding those edges efficiently. However, defying all these impressions, this paper presents fully dynamic algorithms for graph spanners with extremely small update time (or message complexity in distributed environment).

Remark 1.1: The best known dynamic algorithms for various fundamental problems [10, 18, 31, 33, 34, 43, 45] achieve a bound only on the expected and/or amortized time complexity per update. The time complexity bounds guaranteed by our fully dynamic algorithms are also expected amortized instead of worst case.

1.1 Prior work and our contributions

1.1.1 Centralized Algorithms

Ausiello *et al.* [3] are the first to design fully dynamic algorithms for graph spanners. They present a fully dynamic deterministic algorithm for maintaining spanners with stretch at most 5 only, and the update time guaranteed is $O(n)$. Their algorithm does not seem extensible to arbitrary stretch.

In the recent past, a couple of partially dynamic algorithms have also been designed for graph spanners. Incremental (similarly decremental) algorithm for graph spanner is for those applications which involve only insertion (only deletion) of edges. Feigenbaum et al. [30] are the first to design an incremental algorithm for $(2k-1)$-spanner which takes $O(k^2 n^{1/(k-1)} \log n)$ time per edge insertion and maintain spanners with nearly optimal size. Baswana [4] and Elkin [24, 25] independently de-

signed incremental algorithms which guarantee $(kn^{1+1/k})$ bound on the expected size of the spanner and achieve $O(1)$ update time per insertion. However, the algorithm of Elkin [24, 25] is superior in that it guarantees worst case $O(1)$ time which is better than amortized $O(1)$ time per edge insertion guaranteed by Baswana [4].

Elkin [25] also designed an efficient fully dynamic algorithm for $(2k-1)$-spanners for any k. This algorithm handles any edge insertion in $O(1)$ time. However, for deletion of an edge, the algorithm achieves $O(1)$ time with probability $1 - n^{-1/k}$ and $O(m)$ time with probability $n^{-1/k}$. Therefore, the expected time per update guaranteed by the algorithm of Elkin [25] is $O(m/n^{1/k})$. Due to this reason, this algorithm is suitable only if the parameter k as well as the number of edge deletions is very small. In particular, for arbitrary sequence of sufficiently large number of updates, algorithm of Elkin [25] guarantees only $O(m/n^{1/k})$ bound on the average time per update, which is not even sublinear in n if the graph is dense.

We present two fully dynamic centralized algorithms which improve the update time to the extent of near optimality. The update time of our first fully dynamic centralized algorithm is independent of n, and depends only on the stretch of the spanner. In particular, the expected amortized update time achieved for $(2k-1)$-spanner is of the order of $7^{k/2}$ per edge insertion/deletion. Hence constant stretch spanners can be maintained fully dynamically in $O(1)$ expected amortized time per edge update. Our second fully dynamic algorithm guarantees expected amortized $O(k^2 \log^2 n)$ update time for maintaining a $(2k-1)$-spanner. Note that $k = O(\log n)$ always, therefore, our second fully dynamic algorithm achieves expected amortized $O(\text{polylog}\, n)$ update time for all values of k.

The starting point for our dynamic algorithms is the static linear time algorithm of Baswana and Sen [7]. Unlike its seamless adaptability in parallel, distributed and external memory environment, it poses nontrivial challenges to adapt it in a dynamic environment. These challenges are due to the hierarchical structure of the algorithm wherein there is a huge dependency of upper level structures on lower level structures. For a dynamic scenario, such a huge dependency becomes a source of enormous update cost. We investigate the source of these challenges. Using extra randomization and new ideas, we design two fully dynamic algorithms for maintaining a $(2k-1)$-spanner. The reader may please note that the ideas underlying the two algorithms are mutually disjoint, therefore, each of them can be studied independently. Figure 1 provides a comparative description of the previously existing dynamic algorithms for graph spanners and the new algorithms in centralized environment.

Fully Dynamic Algorithms

Stretch	Size	Update Time	Notes
3	$O(n^{3/2})$	$O(n)$	Ausiello et al. [3]
5	$O(n^{5/2})$	$O(n)$	Ausiello et al. [3]
$2k-1$	$O(kn^{1+1/k})$ expect.	$O(mn^{-1/k})$ expect.	Elkin [25]
$2k-1$	$O(k^9 n^{1+1/k} \log^2 n)$ expect.	$O(7^{k/2})$ expect. amort.	**New** (Algorithm-I)
$2k-1$	$O(kn^{1+1/k} \log n)$ expect.	$O(k^2 \log^2 n)$ expect. amort.	**New** (Algorithm-II)

Partially Dynamic Algorithms

Incremental Algorithms			
Stretch	**Size**	**Update Time**	**Notes**
$2k-1$	$O(kn^{1+1/(k-1)})$ expect.	$O(k^2 n^{1/(k-1)} \log n)$	Feigenbaum et al. [30]
$2k-1$	$O(kn^{1+1/k})$ expect.	$O(1)$ amort.	Baswana [4]
$2k-1$	$O(kn^{1+1/k})$ expect.	$O(1)$	Elkin [25]
Decremental Algorithms			
$2k-1$	$O(kn^{1+1/k})$ expect.	$O(k^2 \log n)$ expect. amort.	**New** (Algorithm-II)

Figure 1: Current state-of-the-art dynamic algorithms for spanner in centralized environment.

Remark 1.2: The dynamic algorithms mentioned above are for undirected unweighted graph.

However, they can be used for maintaining spanners for undirected weighted graphs also. Let $w_{\max}$ and $w_{\min}$ be the weights of the heaviest and lightest edge respectively in the graph. For any $\epsilon > 0$, maintain a partition of the edges of the graph into $\log_{1+\epsilon} \frac{w_{\max}}{w_{\min}}$ subgraphs based on their weights. For each subgraph, maintain a $(2k-1)$-spanner ignoring the weights. The union of these spanners will always be a $(2k-1)(1+\epsilon)$-spanner for the graph. However, the update time and the size of spanner thus maintained will increase by a factor of $\log_{1+\epsilon} \frac{w_{\max}}{w_{\min}}$.

1.1.2 Distributed Algorithms

An algorithmic graph problem in the distributed model of computation is defined as follows. Each vertex in the graph corresponds to a node hosting a processor and local memory, and each edge corresponds to a bi-directional link along which messages can be sent. Each node is not aware of the global picture of the distributed network, and communication of messages is the only mean of sharing any information between two neighboring nodes. The computation in the distributed model proceeds in rounds. In each round, a processor may send and receive messages to and from its neighbors, and based on this information it performs some computation locally. A given algorithmic graph problem is solved in a distributive manner - message communications and local computations over a sequence of rounds. The standard measures of complexity for a distributed algorithm are the number of messages communicated and the number of rounds taken to compute the solution of the problem.

There are two models of distributed computation. The first model is synchronous model of distributed computation. In this model, all the processors share a common clock so that transmission (sending and receiving) of messages takes place at the same time on all processors. The time spent in the local computation at a node in each round is assumed to be bounded by the duration of the round. In another model, called asynchronous model, there is no global clock, and therefore the transmission of messages takes place in an asynchronous manner.

Two well known measures of the performance of a distributed dynamic algorithm are quiescence time and quiescence message complexity. Let all updates stop occurring in round α, and let $\beta \geq \alpha$ be the round in which the distributed structure maintained by the algorithm starts satisfying the properties of the problem at hand. The worst-case difference $\beta - \alpha$ is called the quiescence time complexity of the algorithm, and the worst-case number of messages that are sent in between rounds β and α is called the quiescence message complexity of the algorithm.

Though the area of distributed algorithms in static environment is quite rich and vast, there are no explicit dynamic distributed algorithms for majority of the problems. As observed by Elkin in his seminal paper [23], most dynamic distributed algorithms are built by using simulation techniques on top of a static distributed algorithm. Unfortunately these simulation techniques have numerous drawbacks (see [23]). This necessitates the search for designing dynamic distributed algorithm from scratch. Elkin [23] designed the first fully dynamic distributed algorithm for spanners in synchronous as well as asynchronous environment. This algorithm is quite elegant and significantly more efficient compared to the algorithm formed by combining the simulation technique and the static algorithm of Baswana and Sen [8]. The quiescence time complexity of this algorithm [23] is $2k$ in synchronous environment and $3k$ in asynchronous environment. The algorithm of Elkin [23] achieves even superior bounds on quiescence time for some interesting cases of partially dynamic environments. In purely incremental environment, it takes just $O(1)$ rounds to process insertion of multiple edges simultaneously if the number of newly inserted edges incident on each vertex is a constant. In the purely decremental environment, the algorithm [23] achieves expected $1 + o(1)$ rounds of quiescence time if the number of edge deletions in a round is $o(n^{1/k})$. Elkin [23] also showed that for any constant k any distributed algorithm that maintains sparse spanner of near optimal size must need quiescence time greater or equal to $\lfloor \frac{2k}{3} \rfloor$. There is also a lower bound of $\Omega(m)$ on quiescence message complexity for any dynamic algorithm. Note that this lower bound is derived for the worst case when there are $\Theta(m)$ updates in the graph in a single round. The quiescence message complexity of the fully dynamic algorithm of Elkin [23] is $O(km)$. In this manner, quiescence message complexity and quiescence time complexity of the fully dynamic distributed algorithm of Elkin [23] are nearly optimal.

An important measure of complexity of dynamic distributed algorithm has to be the message complexity per update. The message complexity per update achieved by the fully dynamic dis-

tributed algorithm of Elkin [23] are the same as the time complexity per update achieved by the centralized algorithm of Elkin [25]. Thus, the dynamic distributed algorithm of Elkin [23] achieves expected $O(mn^{-1/k})$ bound on message complexity per update, which is quite large. We present a fully dynamic algorithm for $(2k-1)$-spanner in synchronous distributed environment which achieves expected amortized $O(7^k)$ message complexity per update (see Figure 2). It can be observed that the improvement in message complexity per update is really drastic, especially for small value of k. Moreover, the quiescence time complexity of our algorithm is k. In our algorithm, at most one message of size $O(\log n)$ bits is communicated by a vertex along an edge during a round. Hence, the quiescence message complexity of our algorithm is $O(km)$.

Expected Size	Quiescence Time Complexity	Quiescence Message Complexity	Message Complexity per update	Model	Notes
$O(kn^{1+1/k})$	$2k$	$O(km)$	$O(mn^{-1/k})$ expect.	Synchronous	Elkin [23]
$O(kn^{1+1/k})$	$3k$	$O(km)$	$O(mn^{-1/k})$ expect.	Asynchronous	Elkin [23]
$O(k^9 n^{1+1/k} \log^2 n)$	k	$O(km)$	$O(7^k)$ expect. amort.	Synchronous	**New**

Figure 2: Current state-of-the-art fully dynamic distributed algorithms for $(2k-1)$-spanner.

Organization of the paper. In the following section, we describe concepts and notations used in this paper. We also describe the notion of clustering which is the fundamental concept underlying both the algorithms. In section 3, we sketch the challenges in extending the optimal static algorithm of Baswana and Sen [8] to dynamic scenarios, and provide an overview of the approaches taken by our dynamic algorithms. In sections 4 and 5, we describe our two centralized algorithms separately. The fully dynamic distributed algorithm is described in section 6. In section 7, we describe improved fully dynamic centralized algorithms for APASP problem.

All the algorithms designed in this paper are randomized. We assume that the adversary who generates the updates in the graph is oblivious of the random bits used by the algorithm. For clarity of exposition, we have not tried to optimize the value of the leading coefficients as functions of k in our bounds (Theorems 4.2, 4.4, 4.5).

2 Concepts and notations

Throughout this paper, we deal with graphs which are undirected and unweighted. We assume that the vertices are numbered from 1 to n. The objective of building or maintaining a $(2k-1)$-spanner can be achieved by ensuring the following, somewhat local, proposition for each edge $(x, y) \in E$.

$\mathcal{P}_{2k-1}(x, y)$: the vertices x and y are connected in the spanner by a path consisting of at most $(2k-1)$ edges.

In order to maintain a $(2k-1)$-spanner in dynamic environment, our algorithms will ensure that $\mathcal{P}_{2k-1}$ holds at each stage for each edge currently present in the graph. An important ingredient of our algorithms is a suitable grouping of vertices, called *clustering*, which is defined as follows.

Definition 2.1 *[8] a* cluster *is a subset of vertices. A* clustering C*, is a union of disjoint clusters. Each cluster will have a unique vertex which will be called its* center*. A clustering can be represented by an array* $C[1..n]$ *such that* $C[v]$ *for any* $v \in V$ *is the center of the cluster to which* v *belongs, and* $C[v] = 0$ *if* v *is unclustered (does not belong to any cluster).*

A vertex u is said to be *present* in a clustering C, if there is some cluster $c \in C$ such that $u \in c$. For each clustering C we design, we shall always associate a spanning forest $\mathcal{F} \subseteq E$ such that each cluster $c \in C$ corresponds to a tree in this forest. Radius of a clustering is the smallest integer r such that the distance from center of a cluster to any vertex of the same cluster in the forest $\mathcal{F}$ is at most r. See Figure 3 for better understanding of these terminologies associated with a clustering.

The following notations will be used throughout this paper. In order to achieve a better understanding of the two dynamic algorithms, the reader is advised to get familiarized with these notations

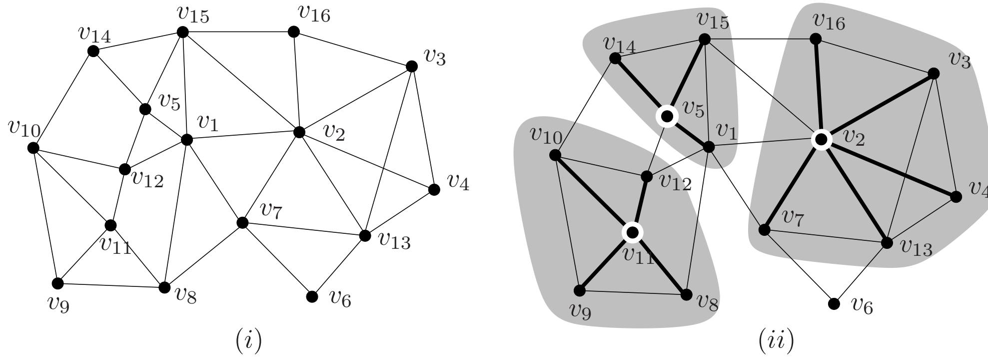

Figure 3: For the graph shown in Figure (i), a clustering consisting of three clusters of radius 1, centered at v_2, v_5, and v_{11} are shown in Figure (ii). Note that v_6 is unclustered (does not belong to the clustering). The thick edges constitute the spanning forest associated with the clustering.

before proceeding further. In these notations, C corresponds to a clustering, $E' \subseteq E$, $X \subset V$, c is any cluster of C, u is any vertex not belonging to cluster c, and $\mathcal{R}$ is any subset of clusters from C.

- $E'(u,c)$: the set of edges from E' which are between u and vertices of cluster c. The cluster c is said to be *adjacent* or *neighboring* to u if $E'(u,c) \neq \emptyset$.
- $E'(c,c')$: the set of edges from E' with one endpoint in cluster c and another endpoint in cluster c'. The cluster c is said to be *adjacent* or *neighboring* to cluster c' if $E'(c,c') \neq \emptyset$.
- $E'(C,C)$: $\{(u,v) \in E' | \ u$ and v belong to different clusters in $C\}$.
- $E'(\mathcal{R})$: the set of edges from E' with at least one endpoint in some cluster $c \in \mathcal{R}$.
- $\delta(u,X)$: $\min\{\delta(u,v) \mid v \in X\}$.

The challenge in building and maintaining a spanner is to keep its size as well as stretch small simultaneously. The idea of clustering helps in meeting this challenge as suggested by the following observation which underlies many static algorithms for computing spanners. The credit of this observation goes to Halperin and Zwick [32], and Peleg and Schaffer [37].

Observation 2.1 *For a graph (V,E), let C be a clustering of radius i for vertices V, and let $\mathcal{F} \subseteq E$ be its spanning forest.*

1. *Let E_S be the set of edges formed by selecting one edge from $E(v,c)$ for each vertex v and its neighboring cluster $c \in C$. Then the subgraph $E_S \cup \mathcal{F}$ ensures that $\mathcal{P}_{2i+1}$ holds for each edge in E (see Figure 4).*
2. *Let E_S be the set of edges formed by selecting one edge from $E(c,c')$ for each pair of neighboring clusters $c,c' \in C$. Then the subgraph $E_S \cup \mathcal{F}$ ensures that $\mathcal{P}_{4i+1}$ holds for each edge in E.*

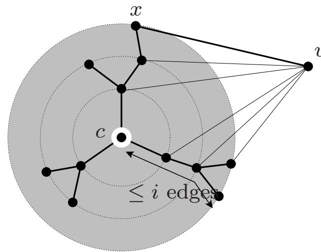

Figure 4: vertex v and a cluster c neighboring to it; adding a single edge (v,x) from $E(v,c)$ ensures $\mathcal{P}_{2i+1}$ for each edge in the set $E(u,c)$.

In order to design a static algorithm for $(2k-1)$-spanner, Baswana and Sen [8] used the above observation and a hierarchy of clusterings. This hierarchy will play an important role in our fully dynamic algorithms as well, so we describe it in the following section.

2.1 A Hierarchy of Subgraphs and Clusterings

Given a graph $G = (V, E)$, consider a hierarchy $\mathcal{H}_k$ of subsets of vertices defined as follows.

Definition 2.2 *Given a graph $G = (V, E)$, $\mathcal{H}_k$ denotes a hierarchy of $k+1$ subsets of vertices $\{S_0, S_1, \ldots, S_k\}$ formed by random sampling as follows. S_0 is the same as V, whereas S_k is $\emptyset$. For $0 < i < k$, the set S_i is formed by selecting each vertex from S_{i-1} independently with probability $n^{-1/k}$.*

We can use $\mathcal{H}_k$ to define $k+1$ levels of subgraphs and clusterings as follows. Level i will have a subgraph $G_i = (V_i, E_i)$, and a clustering C_i which partitions V_i into clusters each centered at some vertex of set $S_i \in \mathcal{H}_k$. The entire hierarchy of subgraphs and clusterings is built in a bottom-up fashion. In order to describe it, we introduce a terminology here. A cluster $c \in C_i$ is said to be a *sampled* cluster at level i if its center belongs to S_{i+1} as well. Let R_i be the set of sampled clusters in clustering C_i.

For level 0, the subgraph G_0 is (V, E), the clustering C_0 is $\{\{v\} | v \in V\}$. Graph G_k is defined as an empty graph - $V_k = \emptyset = E_k$, and so is the clustering C_k. The clustering and subgraph at level $i+1, 0 \le i < k-1$, are defined by subgraph G_i, clustering C_i and the sampled clusters R_i as follows.

1. V_{i+1} is the set of those vertices from V_i which either belong to or are adjacent to some cluster from set R_i in subgraph G_i.

2. The clustering C_{i+1} is defined as follows. Let $\mathcal{N}_i$ be the set of vertices at level i which do not belong to any cluster from R_i, but are neighboring to one or more clusters from the set R_i. C_{i+1} is first initialized to R_i. Then each vertex $v \in \mathcal{N}_i$ concurrently joins (*hooks* on) to a neighboring cluster, say $c \in R_i$ through some edge from $E_i(v, c)$, which will be called *hook*(v, i) henceforth. This defines C_{i+1}. Essentially, each cluster of C_{i+1} can be viewed as a cluster from R_i with some additional layer of neighboring vertices from V_i. See Figure 5 for a visual description of this process.

3. The set E_{i+1} is defined as $E_i(C_{i+1}, C_{i+1})$.

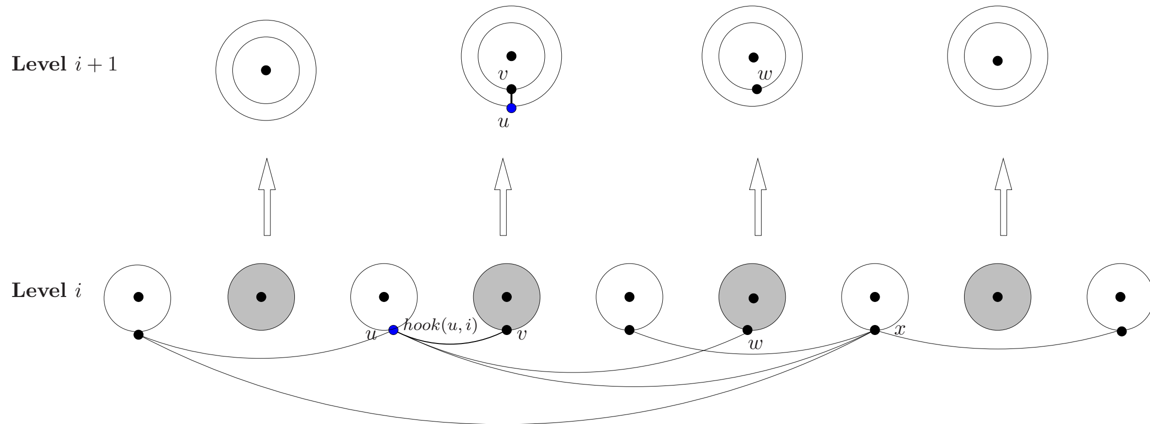

Figure 5: Formation of clusters at level $i+1$. The shaded clusters at level i are the sampled clusters. Vertex u is present at level $i+1$ as well since it is neighboring to a sampled cluster, whereas vertex x is not present at level $i+1$.

The spanning forest F_{i+1} for clustering C_{i+1} is the union of the set of hooks selected during step 2 above and the spanning forest F_i confined to R_i. That is, $F_{i+1} = F_i(R_i) \bigcup \{hook(v, i) | v \in \mathcal{N}_i\}$.

It follows from step 1 and 3 above that $V_{i+1} \subseteq V_i, E_{i+1} \subseteq E_i$, and hence G_{i+1} is subgraph of G_i for all $i < k$. We shall use $\mathcal{C}_k$ to denote the hierarchy of clusterings $C_i, 0 \le i \le k$ as described above. This hierarchy has the following important property which can be proved easily by induction on i.

Lemma 2.1 *C_i with forest F_i is a clustering of V_i with radius i.*

Let $F = \cup_i F_i$. Now along the lines of Observation 2.1 (part *1*), let us form a subset $E_S \subseteq E$ which contains the edges implied by the following rule:

Each vertex $v \in V_i \backslash V_{i+1}$ adds one edge from $E_i(v,c)$ to E_S for each neighboring cluster $c \in C_i$.

Lemma 2.2 *The subgraph $(V, F \cup E_S)$ constructed as described above is a $(2k-1)$-spanner.*

Proof: Consider any edge $(u,v) \in E$ which is not present in $F \cup E_S$. Let $i < k$ be the level such that edge (u,v) is present in E_j, $\forall j \leq i$, but is absent from E_{i+1}. Such a unique i must exist since $(u,v) \in E_0$, $E_k = \emptyset$, and $E_{j+1} \subseteq E_j$, $\forall 0 \leq j < k$. It can be seen that the only reason (u,v) is not present in E_{i+1} is that either u and v both belong to the same cluster in C_{i+1} or at least one of them is not present in V_{i+1}. In the former case, it follows from Lemma 2.1 that there is a path of length $2(i+1)$ between u and v in F. For the latter case, let us assume without loss of generality that $v \notin V_{i+1}$. Let $c \in C_i$ be the cluster containing u. It follows that c is neighboring to v at level i. So the rule mentioned above implies that v will add an edge, say (v,w), from $E_i(v,c)$ to E_S. The vertices u and w belong to the same cluster, so it follows from Lemma 2.1 that there is a path of length $2i$ between u and w in F. This path concatenated with the edge $(v,w) \in E_S$ is a path of length $2i+1$ between u and v in $F \cup E_S$. Hence $\mathcal{P}_{2i+1}$ holds for the edge (u,v) in the subgraph $(V, F \cup E_S)$. Since $i < k$, we can conclude that $(V, F \cup E_S)$ is a $(2k-1)$-spanner. •

The hierarchy $\mathcal{C}_k$ of clusterings forms the back bone of the static algorithms [8, 9], and will be the starting point for our dynamic algorithms as well. Therefore, for better understanding of the dynamic algorithms, we provide a brief intuition as to why such a hierarchy of clusterings is a very natural way to compute and maintain a $(2k-1)$-spanner. An important outcome of this will be the two invariants whose maintenance in a dynamic environment will ensure a $(2k-1)$-spanner of nearly optimal size.

Firstly, as follows from Observation 2.1, the very idea of clustering is aimed at achieving sparseness for the spanner. However, we need to achieve precisely $O(kn^{1+1/k})$ bound on the size and a bound of $2k-1$ on the stretch of the spanner. The hierarchy $\mathcal{C}_k$ achieves these two tasks simultaneously in the following manner. Consider any clustering $C_i, i < k$ from this hierarchy. Let v be a vertex which belongs to V_i. If we add one edge from $E_i(v,c)$ for each cluster $c \in C_i$ neighboring to v, it can be seen that the proposition $\mathcal{P}_{2k-1}$ gets ensured for all edges from $E_i(v,c)$. However, the problem in this naive approach is that this would lead to a large size spanner if v has too *many* neighboring clusters in C_i. To overcome this problem, the random sampling of clusters plays a crucial role - if v has large number of clusters neighboring to it from C_i, then at least one of them must be sampled as well. So v just joins one such sampled cluster at the next level and contributes only one edge to the spanner (in particular, to F_{i+1}). In this way the vertex v continues becoming a member of clustering at higher levels as long as it has *many* neighboring clusters. Eventually, it will reach a level where it has a *few*, i.e. $O(n^{1/k})$, neighboring clusters so that it can afford to add one edge per neighboring cluster. This event is bound to happen at some level $\leq k-1$ since at level $k-1$ there are expected $O(n^{1/k})$ clusters and none of them is sampled. Using elementary probability, it follows that the expected size of the spanner would be $O(kn^{1+1/k})$. However, using a more sophisticated analysis, Pettie [39] recently showed that the size would be $O(kn + n^{1+1/k}\log k)$.

From the above insight into the hierarchical clusterings $\mathcal{C}_k$, we can infer the following key observation which holds at each level i. The vertices belonging to sampled clusters at level i *do not* do any processing; they move to level $i+1$ directly. However, each $v \in V_i$ not belonging to any sampled clusters at level i maintains *exactly* one of the following two invariants.

- CLUSTERING-INVARIANT - If v is neighboring to one or more sampled cluster at level i, then it hooks onto one of them and is present in the clustering at level $i+1$.

- SPANNER-EDGE-INVARIANT - If v is not neighboring to any sampled cluster at level i, then it adds one edge from $E_i(v,c)$ to the spanner for each $c \in C_i$ neighboring to it. v will be absent from level $i+1$ onwards in this case.

The resulting $(2k-1)$-spanner consists of precisely those edges which get added as a consequence of the two invariants mentioned above. Also note that, at the level $k-1$, vertices maintain only SPANNER-EDGE-INVARIANT since kth level is just empty.

In the dynamic environment, maintaining the hierarchy of clusterings $\mathcal{C}_k$ and the two invariants mentioned above will directly imply a $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size at any stage. As will become clear from the following section, this objective, though simple to state, is quite challenging. First we describe the data structure kept by our algorithms in dynamic environment in the following subsection.

2.2 Data structure

For a level i of the hierarchy, let C_i be the clustering and E_i be the edges, and u be a vertex belonging to a cluster centered at w. There may be many clusters neighboring to u at level i as shown in Figure 6 (i). We now give a description of the global data structure stored at level i, and the local data structure stored at the vertex u.

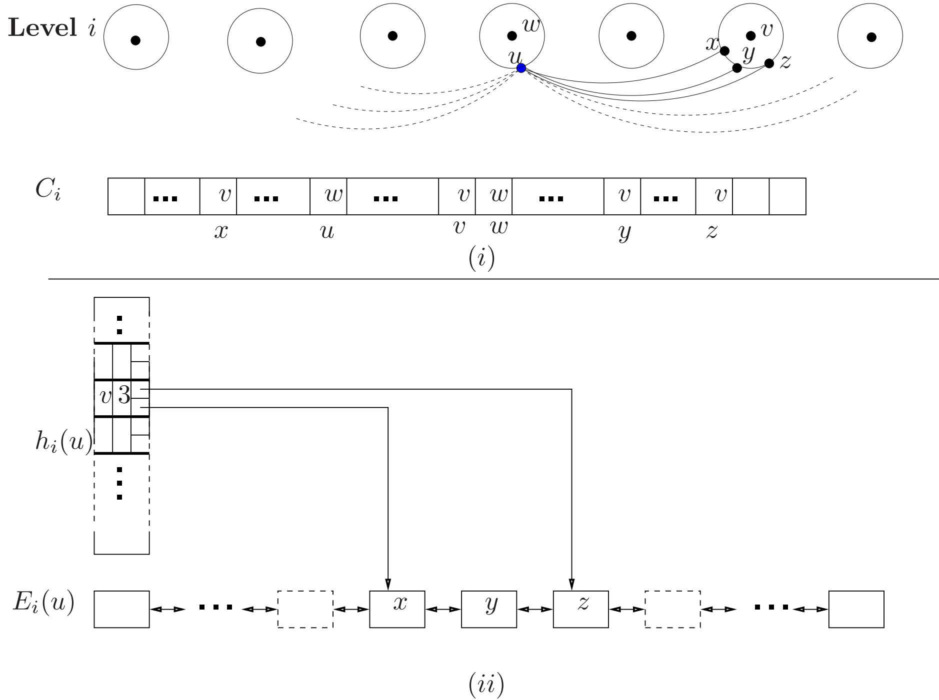

Figure 6: (i)vertex u and its neighbors in clustering at level i, (ii) Data structure $D_i(u)$.

As a global data structure at level i in the hierarchy, we keep array $C_i[\,]$ for storing the clustering at level i. See Figure 6 (i). In addition, we also keep an array $M[\,]$ to store the sampling status of each vertex in the hierarchy $\mathcal{H}_k$. For a vertex w, $M[w] = j$ would imply that for all $i \leq j,\ w \in S_i$. Each vertex u at level i keeps a data structure $D_i(u)$ locally which consists of the following components (see Figure 6 (ii)).

1. a dynamic hash table $h_i(u)$:
   Each vertex u keeps a dynamic hash table $h_i(u)$ for storing the centers of all clusters in C_i neighboring to it. In addition, it stores the number of the edges with which the cluster is adjacent to u. Note that the existing dynamic hash tables, say by Pagh and Rodler [36], guarantee worst case $O(1)$ search time, and expected amortized $O(1)$ time per insertion/deletion.

2. a linked list for $E_i(u)$:
   Vertex u keeps the edges $E_i(u)$ arranged in a doubly linked list in such a way that, for each neighboring $c \in C_i$, the edges $E_i(u,c)$ will appear as a contiguous sublist in $E_i(u)$. We shall also keep pointers from the entry of c in the hash table $h_i(u)$ to the beginning and end of this sublist storing $E_i(u,c)$. Any insertion to $E_i(u,c)$ will be made at the end of this sublist. We shall also maintain a dynamic hash table storing neighbors of u at level i. The entry in this hash table for a neighbor, say v, will point to the edge (u,v) in the list $E_i(u)$. This will facilitate removal or insertion of any edge from $E_i(u)$ in $O(1)$ time.

3. spanner-label for every edge:
   For each edge (u, v) present in the graph, we maintain a spanner-label which would determine whether the edge is in the spanner currently. This label will be an integer variable such that (u, v) is in the spanner if the value stored in the variable is positive. This label gets updated by u and v as follows. While maintaining any of its invariants at some level i, if u decides to add the edge (u, v) to the spanner, it increments the spanner-label of (u, v), and whenever it decides to delete the edge from spanner, it decrements the spanner-label of (u, v).

The data structrue described above will prove to be very helpful in handling updates. Our fully dynamic algorithms will employ a subroutine `Restore-span-edge-invariant`(u, c, i). This subroutine will be invoked in the following scenario. Consider a vertex u which is maintaining SPANNER-EDGE-INVARIANT at level i. Suppose some edge incident on u from some cluster $c \in C_i$ gets deleted (or inserted). This may lead to violation of SPANNER-EDGE-INVARIANT for u at level i. However, using the data structure $D_i(u)$ described above, procedure `Restore-span-edge-invariant`(u, c, i) restores it in $O(1)$ time as follows. If c ceases to be neighboring to u due to the edge deletion, it removes c from the hash table $h_i(u)$. Otherwise, using hash table $h_i(u)$, it accesses the sublist $E_i(u, c)$ and increments the spanner-label of the first edge in this sublist.

3 The Challenges in Maintaining Spanner Dynamically

For maintaining a $(2k-1)$-spanner, let us explore the hurdles in maintaining the hierarchy $\mathcal{C}_k$ of clustering and the two invariants mentioned above.

Consider any edge update, say deletion of an edge (u, v). The updates have to be processed starting from level 0, and have to be passed to higher levels due to the dependency of upper level objects (subgraphs and clustering) on the lower level objects of the hierarchy. It follows from the hierarchy of clusterings that $E_{i+1}(u) \subseteq E_i(u)$ for all $i < k$ and $u \in V$. Therefore, for each edge (u, v), there exists an integer i such that it is present in E_j for all $j \leq i$, and absent in E_ℓ for all $\ell > i$. So when the edge (u, v) is deleted, we remove it from all levels $j < i$. Let us consider the processing required at level i. If the edge is not included in the spanner, then we just delete it from E_i as well and stop. Let us consider the situation in which the edge is included in the spanner at this level. Recall that it must have been included in the spanner because of one of the two invariants.

If the edge (u, v) was included in the spanner due to SPANNER-EDGE-INVARIANT, it must be that i is the last level for u or v. Without loss of generality let i be the last level for u and let v belongs to cluster c at level i. In this case, vertex u executes `Restore-span-edge-invariant`(u, c, i) which will take $O(1)$ time using $D_i(u)$.

If the edge (u, v) was included in the spanner due to CLUSTERING-INVARIANT, it must be serving as a hook for one of its endpoints to join the clustering at next level. Without loss of generality, let us assume that (u, v) was $hook(u, i)$, and u belonged to cluster c at level $i+1$. In this case, deletion of (u, v) is a very costly update as follows. If u has no sampled neighboring cluster at level i now, then u can't be present at level $i+1$ and so will have to settle at level i only. As a consequence, we shall have to delete all the edges incident on u at level $i+1$, and restore SPANNER-EDGE-INVARIANT for each edge from $E_i(u)$ at level i itself. Let us consider the other, more generic, case wherein u still has some sampled neighboring clusters at level i. In this case u must join one such cluster, say c', at the next level in order to maintain CLUSTERING-INVARIANT. So deletion of (u, v) leads to moving u from cluster c to c' at level $i+1$. Consider any neighbor of u at level i which is present at level $i+1$ too. This vertex will perceive this change as deletion of an edge incident from c and insertion of an edge incident from c'. See Figure 7. Each such vertex has to process these updates individually for maintaining CLUSTERING-INVARIANT or SPANNER-EDGE INVARIANT at level $i+1$. Thus a single edge deletion at level i may lead to $\Theta(|E_i(u)|)$ updates at level $i+1$. In a similar fashion, a single edge insertion at level i may force $\Theta(|E_i(u))$ updates at level $i+1$.

Therefore, it is obvious that a single edge insertion/deletion may cost $\Omega(n)$ computational work for maintaining $(2k-1)$-spanner if we have the naive dynamic version of the static algorithm [8].

In short, the main problem in dynamizing the static algorithm is that the updates at a level which lead to change in clustering at the next level are very expensive. To overcome this hurdle,

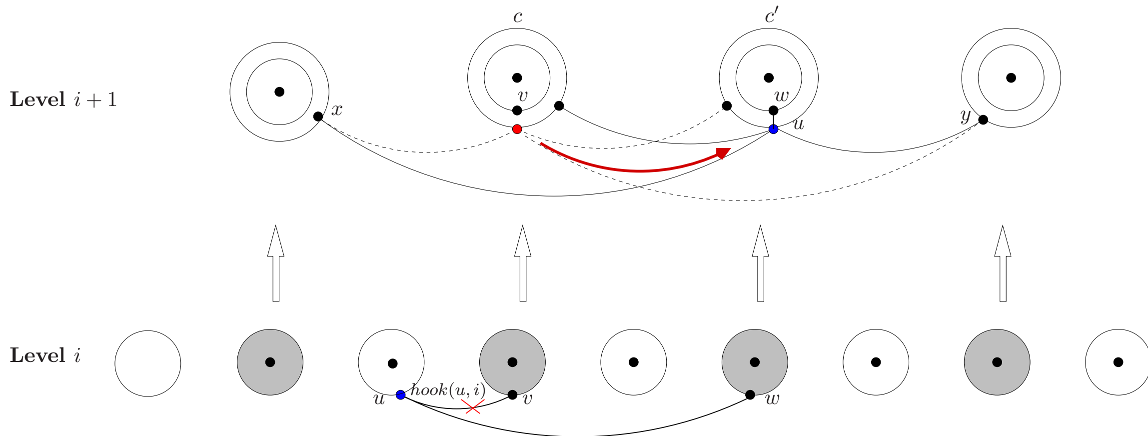

Figure 7: Upon deletion of $hook(u,i) = (u,v)$, u might leave its old cluster c at level $i+1$ and join cluster c'. For each neighbor x of u at level $i+1$, it is as if a new edge (shown solid) incident from c' has been inserted and an edge (shown dotted) incident from c has been deleted.

our two algorithms take completely different approaches and use randomization in a powerful way.

Algorithm I. Our First fully dynamic algorithm maintains a hierarchy of clusterings and subgraph similar to that of the static algorithm [8]. However, it ensures that the probability of any edge update at level i to change clustering at level $i+1$ is *very small.* To achieve this objective, the algorithm strives to maintain the following property at every stage. For any edge (u,v) at level i, the probability that "$hook(u,i) = (u,v)$" is bounded by $O(1/|E_i(u)|)$. This property turns out to be quite easy to maintain for 3-spanner. However, its extension to $(2k-1)$-spanner requires a couple of novel ideas. This property has the following important consequence. While processing an update at level i, the expected amortized number of updates generated for level $i+1$ is bounded by a constant. The value of this constant turns out to be close to 7. This leads to an $O(7^k)$ update time algorithm with a careful analysis. The update time is later improved to $O(7^{k/2})$ by reducing the levels of the clustering to $k/2$ and using part (ii) of Observation 2.1 at the top most level.

Algorithm II. One may note that the main source of the costly update operations in the straightforward dynamization of the static algorithm is the following. There is a huge dependency of objects (clustering, subgraphs) at a level on the objects at the previous level, so a single edge insertion/deletion at level i can lead to a huge number of updates at level $i+1$. In our second algorithm we essentially try to get rid of this dependency. For this, we use a novel clustering such that there is no dependency between clusterings present at any two different levels. As a result we need to maintain clustering and SPANNER-EDGE INVARIANT at each level in a way almost *independent* of other levels. We first design a decremental algorithm for spanners using this new clustering. This clustering uses extra randomization which helps in achieving the following result. For any arbitrary sequence of edge deletions, a vertex will change its cluster at any level expected $O(\text{polylog}\, n)$ times. We then extend the decremental algorithm to fully dynamic environment at the expense of increasing the expected update time and size by $\log n$ factor. The ideas used in extending the decremental algorithm to fully dynamic environment are similar to those used by Henzinger and King [33] and Holm et al. [34] in the context of fully dynamic connectivity.

A common point between the two algorithms is the following. Both of them maintain hierarchies of subgraphs, clusterings, and slightly *modified* CLUSTERING-INVARIANT and SPANNER-EDGE-INVARIANT. However, the approach of first algorithm is totally local. Exploiting this feature and a couple of new observations, we later adapt it in distributed environment as well. The second algorithm uses a mixture of local and non-local approaches. In particular, the second algorithm employs a BFS tree like structure for maintaining the new clustering; and this is the reason for its non-local nature.

4 Fully Dynamic Algorithm - I

A distinct feature of this algorithm is its vertex centric approach described below.

4.1 Vertex centric approach in processing updates

An update (insertion or deletion of an edge) in the graph will be processed in a bottom up fashion from level 0 to $k-1$ of the underlying hierarchy maintained by the algorithm. An update reaching a level $i > 0$ will be of one of the following types.

1. insertion of an edge or deletion of an edge
2. change in clustering of a vertex

Each update of the above type is meant for two or more vertices as follows. Insertion (or deletion) of an edge will require processing by both endpoints of the edge. A change in clustering of a vertex would require processing by all its neighbors at that level. Our fully dynamic algorithm will follow a vertex centric approach wherein each vertex receives updates solely meant for it and it processes them to maintain its invariants at that level. We shall call these updates *atomic* updates. For this purpose, we shall express the two types of *macro* updates mentioned above in terms of atomic updates as follows.

- For the macro update which is deletion (or insertion) of edge (u, v), there will be one atomic update for the deletion of (u, v) to be processed by u, and one atomic update for the deletion of (u, v) to be processed by v.
- For the macro update which is change of cluster of a vertex u, say from c to c' at level i, there will be a single atomic update for u for changing $C_i[u]$ from c to c'. This update will take just $O(1)$ time. In addition to this atomic update which is solely for u, there will be two atomic updates for every neighbor of u at level i: deletion of a single edge incident from c and insertion of a single edge from c'.

It follows that, from perspective of any vertex u, each atomic update will be an insertion or deletion of an edge incident on it from some cluster at level i or a change of $C_i[u]$. The latter atomic update takes $O(1)$ time now. The concept of atomic updates simplifies the description of the algorithm : all that we need is to describe the processing of each atomic update by the respective vertex. Every vertex u will process an atomic update so as to restore its invariants. Interestingly, this processing will be completely local and won't affect any other vertex at this level. However, it might generate update for some neighbors of u only at the next level. The atomic representation of the updates helps in the analysis for the time complexity as well. It turns out that the time complexity of processing an atomic update at a level will be of the order of updates that its processing would generate for the next level. From now onwards, each update would implicitly mean an atomic update. First as a warm up, we describe our fully dynamic algorithm for 3-spanner. Later we shall use some novel ideas to extend it to fully dynamic algorithm for $(2k-1)$-spanner for any positive integer k.

4.2 Fully dynamic algorithm for 3-spanner

The fully dynamic algorithm for 3-spanner is designed by adding slight randomization to the static algorithm for 3-spanner. Recall from subsection 2.1 that the static algorithm for 3-spanner uses the hierarchy $\mathcal{H}_2 = \{S_0, S_1, S_2\}$, where $S_0 = V, S_2 = \emptyset$, and S_1 is formed by selecting each vertex independently with probability $1/\sqrt{n}$. Let $R(u)$ be the set of vertices from S_1 neighboring to u. The algorithm computes two levels of subgraphs and clusterings as follows.

- Level 0 consists of original graph and clustering $\{\{u\} | u \in V\}$. Each vertex $u \in V \backslash S_1$ does the following. If $R(u) \neq \emptyset$ then u hooks onto some vertex $x \in R(u)$ (following the CLUSTERING-INVARIANT), and this defines clustering at level 1, otherwise it adds all its edges to spanner (following the SPANNER-EDGE-INVARIANT).
- At level 1, each vertex adds one edge per neighboring cluster to spanner (following the SPANNER-EDGE-INVARIANT).

This completes the description of the static algorithm of 3-spanner. The fully dynamic algorithm will also maintain two levels of subgraphs and clusterings. However, clustering at level 1 will be slightly different from the static algorithm. Though it will consist of clusters centered at S_1 but the way the vertices of set $V \backslash S_1$ join clusters will employs randomization as follows. Each vertex $u \in V \backslash S_1$ with $|R(u)| \geq 1$ will maintain the following, somewhat restricted, CLUSTERING-INVARIANT throughout the sequence of edge updates:

$$\forall v \in R(u) \qquad \mathbf{Pr}[(u,v) = hook(u,0)] = \frac{1}{|R(u)|}$$

In other words, at any moment of time, u will belong to a cluster centered at a uniformly and randomly selected vertex from $R(u)$. An important consequence of the above invariant and the randomization used in the construction of S_1 is the following lemma.

Lemma 4.1 *For a vertex $u \in V \backslash S_1$ with $R(u) \neq \emptyset$, any edge (u,v) is $hook(u,0)$ with probability $\frac{1}{\deg(u)}$.*

The proof of the above lemma is based on the following observation from elementary probability. Let there be a bag containing arbitrary number of balls. Consider a two level sampling to select a *winner* ball from the bag - first select a sample of j balls uniformly randomly from the bag (for some $j \geq 1$), and then select one ball uniformly from the sampled balls and call it the winner. In this two-level sampling experiment, each ball of the bag is equally likely to be the winner ball.

While processing the updates, the fully dynamic algorithm will maintain the SPANNER-EDGE-INVARIANT and the new, somewhat restricted, CLUSTERING-INVARIANT mentioned above. Therefore, the stretch of the spanner will still be 3, and the expected size of the 3-spanner will still remain $O(n^{3/2})$. We shall now describe how the dynamic algorithm handles the updates.

4.2.1 Handling deletion and insertion of edges

First observe that each vertex $u \in V$ present at level 1 follows only SPANNER-EDGE-INVARIANT because there is no sampled cluster at level 1. Using data structure $D_1(u)$, it will take $O(1)$ time to process each update reaching level 1. Keeping this in mind, we need to focus on just the number of updates reaching level 1 upon any edge insertion or deletion. We now describe the procedures to handle any update.

Handling deletion of an edge. Suppose an edge (u,v) gets deleted. This will introduce two updates at level 0, one each for u and v. Vertex u processes this update as follows (the vertex v handles it in a similar fashion). There are the following two cases.

- **Case 1:** $v \in S_1$
  If vertex u also belongs to S_1, then no invariant gets violated for u. Vertex u just deletes the edge from $D_0(u)$. In this case a single update corresponding to deletion of (u,v) is passed onto level 1. Let us consider the more interesting case when u is not a sampled vertex at level 0. There are the following two cases here.

  If $hook(u,0) = (u,v)$, then the deletion of (u,v) is quite catastrophic. If $R(u)$ has become $\emptyset$, then vertex u will have to maintain SPANNER-EDGE-INVARIANT at level 0. This implies that u will add all its edges to spanner and leave C_1. However, if $R(u)$ still has some vertices, then vertex u will have to restore CLUSTERING-INVARIANT at level 0, and so it selects some neighbor, say z, from $R(u)$ uniformly randomly and hooks onto it at level 1; thus u remains present in C_1 but changes its cluster. In each of these cases, one or two updates will be generated for each neighbor of u at level 1. Thus a total of $O(\deg(u))$ updates get generated for level 1.

  If $(u,v) \neq hook(u,0)$, then deletion of (u,v) violates neither of the two invariants for u. In this case, (u,v) is deleted from $D_0(u)$. Only a single update corresponding to deletion of (u,v) is passed onto level 1.

- **Case 2:** $v \notin S_1$
  u just deletes (u,v) from $D_0(u)$. If u is not present in C_1, the edge (u,v) is deleted from spanner as well. If u and v are present at level 1, the deletion of (u,v) is passed onto level 1.

So handling deletion of an edge (u, v) by vertex u will require $O(1)$ processing at levels 0 and 1 except when $hook(u, 0) = (u, v)$; in which case it requires $O(\deg(u))$ amount of processing. Applying Lemma 4.1 just before deletion of (u, v), it follows that probability "$(u, v) = hook(u, 0)$" is $1/\deg(u)$. Hence the expected amount of computation performed upon deletion of any edge (u, v) is of the order of $1/\deg(u) \cdot O(\deg(u)) + O(1) = O(1)$.

Handling insertion of an edge. Let (u, v) be the edge inserted. We describe the processing done by u. There are the following two cases.

- **Case 1:** $v \in S_1$
  If vertex u was itself a sampled vertex at level 0, then no processing is required at level 0. The insertion of edge (u, v) is passed onto level 1. Let us consider more interesting case when u was not a sampled vertex at level 0.

  If u was not neighboring to any sampled cluster at level 0, then it must have added all its edges to the spanner. For this purpose, u would have incremented the spanner-label of each of its edges. Insertion of (u, v) has made u neighboring to a sampled cluster $\{v\}$, so u decrements the spanner-label of all its edges at level 0, makes $hook(u, 0) = (u, v)$, and thus joins the cluster centered at v in C_1. All this will require $O(\deg(u))$ amount of processing at level 0. If u was neighboring to any sampled cluster at level 0, then in order to restore CLUSTERING-INVARIANT u does the following. With probability $1/|R(u) \cup \{v\}|$, u selects (u, v) as $hook(u, 0)$, leaves its earlier cluster and joins the cluster centered at v in C_1. In each of these cases, one or two updates will be generated for each neighbor of u at level 1. Thus a total of $O(\deg(u))$ updates get generated for level 1.

- **Case 2:** $v \notin S_1$
  u adds (u, v) to $D_0(u)$. If u is not present in C_1, the edge (u, v) is added to the spanner and no update is passed to level 1. However, if u and v are present at level 1, the insertion of (u, v) is passed onto level 1.

So handling insertion of edge (u, v) by u will require $O(1)$ processing at levels 0 and 1 except when (u, v) becomes $hook(u, 0)$; in which case it requires $O(\deg(u))$ amount of processing. Applying Lemma 4.1 just after insertion of (u, v), it follows that probability "$(u, v) = hook(u, 0)$" is $1/\deg(u)$. Hence the expected amount of computation performed upon insertion of any edge (u, v) will be $1/\deg(u) \cdot O(\deg(u)) + O(1) = O(1)$. We can thus conclude the following theorem.

Theorem 4.1 *For an unweighted undirected graph on n vertices, a 3-spanner of expected size $O(n^{3/2})$ can be maintained fully dynamically with expected $O(1)$ update time per edge insertion/deletion.*

4.3 On generalizing the dynamic algorithm for arbitrary k

Inspired by the fully dynamic algorithm for 3-spanner, our fully dynamic algorithm for $(2k-1)$-spanner should aim to maintain the following property at any level i. For any edge (u, v) incident on u at level i, probability that "$hook(u, i) = (u, v)$" is at most $1/|E_i(u)|$. For the case of 3-spanner, $i = 1$, and this property could be ensured by the *random hooking edge* principle. So, it is natural to explore if the same principle alone would work for the fully dynamic algorithm for $(2k-1)$-spanner. A careful reader might have realized that this simple principle worked so well for 3-spanner due to the following reasons. Firstly, there is uniformity in sampling the (singleton) clusters at level 0. Secondly, from perspective of u, there is a symmetry among all neighboring clusters at level 0, that is, each of them is adjacent to u with the same number (just one) of edges. However, such a symmetry can not be guaranteed at level $i > 0$. In order to realize how this asymmetry, i.e. the skewed distribution of edges $E_i(u)$ among neighboring clusters of u, causes a serious problem, here is an example. Consider a level i where $0 < i < k-1$, and let u does not belong to any sampled cluster at this level. Let there be $\ell = \Theta(n^{1/k} \log n)$ clusters - $c_1, \ldots, c_\ell$ neighboring to u, and let the edges $E_i(u)$ are distributed among these clusters as follows. There is exactly one edge between u and $c_j, \forall j < \ell$ and all the remaining edges incident on u have their other endpoints in c_ℓ. Potentially, $\ell \ll |E_i(u)|$. Now consider an edge (u, v) with $v \in c_j, j < \ell$. The probability that c_ℓ is not sampled is $1 - n^{-1/k} \approx 1$. So it can be seen that with high probability there will be $O(\log n)$ edges incident on u from sampled clusters at level i. As a result, with high probability, u will hook

onto some cluster $c_j, j < \ell$. So the random hooking edge principle would imply that the probability that "$hook(u,i) = (u,v)$" is approximately $1/\ell$ which is much larger than $1/|E_i(u)|$.

We overcome the problem arising due to skewed distribution of $E_i(u)$ by a novel idea of *filtering* of edges. Interestingly, the reason this idea works has its roots in randomization too.

4.3.1 Key idea of filtering of edges

Let there be a coin which gives HEADS with probability p and TAILS otherwise. The following fact is a folklore in probability theory. If there are *large*, i.e., $\Omega(1/p \log n)$ numbers of coin tosses, then with high probability, the number of HEADS will be concentrated around pth fraction of the total number of tosses. We shall use the following reformulation of this fact. If the total number of TAILS were *large*, i.e., $\Omega(1/p \log n)$, then the number of HEADS is concentrated around pth fraction of the total number of coin tosses.

Recall that each cluster at level i is sampled (corresponds to HEADS) with probability $p = n^{-1/k}$. We partition the set of unsampled (corresponds to TAILS) clusters neighboring to vertex u at level i into *buckets* so that the clusters incident with *nearly* the same number of edges belong to the same bucket. There will be total $O(\log n)$ buckets. Now consider a bucket consisting of clusters which are incident on u with $\Theta(s)$ edges for some s. If the bucket has *large* (i.e., $\Omega(n^{1/k} \log n)$) number of clusters, then the above probability fact implies the following. Out of all those clusters incident on u with $\Theta(s)$ edges at level i, with high probability, close to $n^{-1/k}$th fraction would be sampled ones. Now consider a bucket having few, i.e., $\ll n^{1/k} \log n$ clusters. For such a *small* bucket, we can't make such an inference with high probability. We take care of small buckets by *filtering* away all their edges at level i itself: We won't let any of the edges incident on u from the clusters of small buckets move to level $i+1$. Instead, vertex u will add one edge per neighboring clusters from every small bucket to the spanner so that proposition $\mathcal{P}_{2k-1}$ holds for all the filtered edges at level i itself. Using the facts that a small bucket has very few clusters and there are $O(\log n)$ buckets, this strategy will lead to an increase in the size of the spanner by poly-logarithmic factors only. However, this filtering will have the following useful consequences. Let $\mathcal{E}_i(u)$ be the set of remaining edges from $E_i(u)$ after filtering away the edges incident from clusters of *small* buckets. Firstly, observe that in case of change in clustering of u at level $i+1$, u will introduce $O(|\mathcal{E}_i(u)|)$ updates instead of $\Theta(|E_i(u)|)$. Secondly, with high probability $(1-\epsilon)n^{-1/k}$th fraction of $\mathcal{E}_i(u)$ edges are incident on u from sampled clusters at level i. As a result, the random hooking edge principle ensures that any edge $(u,v) \in \mathcal{E}_i(u)$ is $hook(u,i)$ with probability $O(1/|\mathcal{E}_i(u)|)$. Hence a single edge update on u at level i will lead to expected $O(1)$ updates at level $i+1$. Recall, that this is what we wished to achieve for our fully dynamic algorithm.

In the following subsection, we describe the bucket data structure for classifying the neighboring clusters of a vertex based on the number of edges incident on it. This data structure will help in implementing the idea of filtering of edges. Though the concept of bucket structure is used to achieve better update time, this structure has to be dynamic. Maintaining bucket structures at a level produces additional updates for the next level. However, we present an analysis of these dynamic buckets in subsection 4.7 and show that the amortized number of these updates is quite small.

4.4 Bucket structure associated with a vertex

First we state a theorem from probability theory. This theorem will be used to show how the idea of filtering of edges achieves an important goal (stated in Lemma 4.3) on the way to design our first fully dynamic algorithm for graph spanners.

Theorem 4.2 *Let $o_1, \ldots, o_\ell$ be ℓ positive numbers such that the ratio of the largest to the smallest number is at most b for some $b > 1$, and $X_1, ..., X_\ell$ be ℓ independent random variables such that X_i takes value o_i with probability p and zero otherwise. Let $\mathcal{X} = \sum_i X_i$, and $\mu = \mathbf{E}[\mathcal{X}] = \sum_i o_i p$, and $a > 1$. There exists a constant γ such that if $\ell \geq ab\gamma \frac{\ln n}{\epsilon^4 p}$ for any $\epsilon > 0$ then the following bound holds.*

$$\mathbf{Pr}[\mathcal{X} < (1-\epsilon)\mu] < O(n^{-a-1})$$

The above mentioned theorem can be viewed as an extension of the Chernoff bound [12]. In fact for the case, when $o_1 = o_2 = \cdots = o_\ell = 1$, this theorem is identical to the Chernoff bound [12]. The proof of this theorem is provided in Appendix.

Consider a vertex u at level i. The edges $E_i(u)$ incident on u at level i are of two types : the edges which are incident from sampled clusters and the edges which are incident from unsampled clusters. We describe a bucket data structure $B_i(u)$ for storing edges incident on u from all the neighboring unsampled clusters at level i. It consists of $\log_{\frac{1}{\epsilon}} n$ buckets, wherein jth bucket stores adjacency information between u and all those neighboring unsampled clusters which have at least $(\frac{1}{\epsilon})^{j-1}$ (the Lower-limit of jth bucket) and at most $(\frac{1}{\epsilon})^{j+1} - 1$ (Upper-limit of jth bucket) edges onto u. The bucket structure $B_i(u)$ can be built easily in time of the order of $O(|E_i(u)|)$. We shall call a bucket *active* if there are $\ell \geq a\gamma n^{1/k}\frac{\ln n}{\epsilon^6}$ clusters in it for some constant a, and *inactive* otherwise.

Remark 4.2: The bucket structure is vertex specific, so the active or inactive status of a cluster is not a universal categorization of clusters at a level - a cluster can belong to active buckets of some neighboring vertices and to inactive buckets of other neighboring vertices.

Let $\mathcal{E}_i(u)$ be the set of all those edges which are incident on u from a sampled cluster or a cluster belonging to an active bucket at level i. The following lemma establishes a lower bound on the number of edges incident on u from sampled clusters in terms of $\mathcal{E}_i(u)$.

Lemma 4.3 *The number of edges incident on a vertex u from sampled clusters at level i is at least $(1-\epsilon)n^{-1/k}|\mathcal{E}_i(u)|$ with probability at least $1 - n^{-a}$.*

Proof: Partition all the clusters neighboring to u at level i into $O(\log_{1/\epsilon} n)$ groups such that jth group has all those clusters which are incident on u with edges in the range $[(\frac{1}{\epsilon})^{j-1}, (\frac{1}{\epsilon})^{j+1} - 1]$. We call a group *big* if it has at least $a\gamma n^{1/k}\frac{\ln n}{\epsilon^6}$ clusters, and *small* otherwise. Consider any big group and apply Theorem 4.2 with o_t being the number of edges incident on u from tth cluster of the group, $b = 1/\epsilon^2$, and the sampling probability $p = n^{-1/k}$. It follows that with probability at least $1 - n^{-a-1}$, the number of edges incident on u from sampled clusters of a *big* group will be at least $(1-\epsilon)n^{-1/k}$ fraction of all the edges incident from (the clusters of) the group. Let $E_i^{big}(u) \subset E_i(u)$ be the edges incident on u from all big groups. Since there are only $O(\log n)$ big groups, applying union bound for each big group, it follows that with probability at least $1 - n^{-a}$ the number of edges incident from sampled clusters of all *big* groups is at least $(1-\epsilon)n^{-1/k}$ fraction of $|E_i^{big}(u)|$.

Now let us explore the relationship between the set $E_i^{big}(u)$ and the set $\mathcal{E}_i(u)$. Consider any edge from $\mathcal{E}_i(u)$ which is incident on u from an unsampled cluster at level i. We shall show that this edge is definitely present in $E_i^{big}(u)$ as well. In addition, observe that all edges incident on u from sampled clusters at level i are present in $\mathcal{E}_i(u)$. Thus if we consider the fraction of edges incident from sampled clusters, this fraction may only be bigger in $\mathcal{E}_i(u)$ than $E_i^{big}(u)$ (see Figure 8). From the above analysis, this will conclude the proof of this lemma.

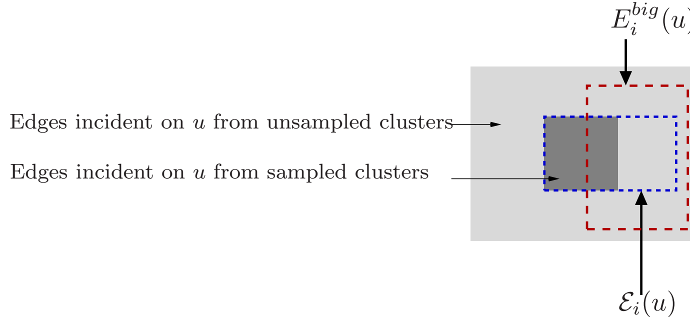

Figure 8: The relationship between $E_i^{big}(u)$ and $\mathcal{E}_i(u)$

Consider any edge $(u, v) \in \mathcal{E}_i(u)$ incident on u from some unsampled cluster, say $c \in C_i$, containing v. Let c belong to jth bucket. It follows from the construction of $\mathcal{E}_i(u)$ that this bucket must be an active bucket and so has at least $a\gamma n^{1/k}\frac{\ln n}{\epsilon^6}$ unsampled clusters. Notice that jth group has the same range as the jth bucket. The only distinction is that the latter stores only unsampled

clusters. Therefore, all those clusters that belong to jth bucket are present in jth group. So the jth group will be a *big* group and all its edges, and hence (u, v), will be present in $E_i^{big}(u)$ as well. •

A note on dynamic bucket structures.
The bucket structure will have its own overhead in dynamic scenario which will contribute to the update time achieved by the fully dynamic algorithm. To realize this fact, consider bucket structure associated with vertex u at level i. Note that the bucket structure assigns each unsampled cluster, say c, neighboring to u to its respective bucket according to the size of set $E_i(u, c)$. As vertices change clusters at level i, and as the edges are being inserted or deleted at level i, the size of $E_i(u, c)$ keeps changing. As a result, c may have to hop from one bucket to another bucket (of possibly different activation status). This may lead to change in activation status of all the edges of set $E_i(u, c)$. Furthermore, due to such movement of clusters neighboring to u, activation status of an entire bucket may also change. This may lead to change in activation status of even larger number of edges incident on u. In this manner a single edge deletion/insertion from/to $E_i(u, c)$ may lead to a huge number of updates in the set $\mathcal{E}_i(u)$. All these updates may have to be communicated to level $i + 1$. Notice that the cause of these updates is only the maintenance of bucket structure at level i. However, by selecting parameters of the bucket structure carefully, we shall show that the amortized number of updates generated for level $i+1$ due to maintenance of the bucket structure will be quite *small*. For this purpose, we shall exploit the fact that the ranges of two adjacent buckets are overlapping. In addition, we shall use different thresholds for activation and deactivation of buckets. These two measures will ensure that sufficiently large number of insertion or deletions will be required into the bucket structure of a vertex before a huge number of edges incident on it change their activation status. The entire analysis is similar in spirit to the analysis of dynamic tables [15]. However, in order to avoid distraction from our main problem at this point, we have deferred the complete analysis to subsection 4.7. Here we just state the following theorem.

Theorem 4.3 *In processing one edge update for maintaining the bucket structure $B_i(u)$ at level i, the amortized number of additional updates introduced at level $i + 1$ is at most $4 + 10\epsilon$ for any $0 < \epsilon \leq \frac{1}{4}$.*

4.5 Hierarchies and additional invariants maintained for $2k - 1$-spanner

The backbone of the fully dynamic algorithm for $(2k-1)$-spanner is a hierarchy of clusterings induced by $\mathcal{H}_k$. As we shall soon see, this hierarchy differs from that of $\mathcal{C}_k$ due to added randomization incorporated in the clustering and the idea of filtering of the edges at each level as described above. The clustering and subgraph maintained at level $i+1$ by our algorithm are defined by the clustering and subgraph at level i and the new invariants as follows.

- The set V_{i+1} consists of those vertices of set V_i which belong to or are adjacent to sampled clusters at level i.
- Clustering at level $i + 1$ is defined as follows. Let R_i be the set of sampled clusters in C_i and let $\mathcal{N}_i$ be the set of those vertices from V_i which don't belong to, but are adjacent to one or more clusters from R_i. Each cluster of C_{i+1} will be a sampled cluster from R_i with an additional layer of some neighboring vertices from $\mathcal{N}_i$ hooking onto it. However, each vertex $u \in \mathcal{N}_i$ selects the neighboring sampled cluster to hook onto randomly ensuring the following invariant.

  $$\text{NEW-CLUSTERING-INVARIANT}: \quad \forall (u, v) \in E_i(u) \text{ with } C_i(v) \in R_i, \ \mathbf{Pr}[(u, v) = \mathit{hook}(u, i)] = \frac{1}{|R_i(u)|}$$

  where $R_i(u)$ is the set of edges incident on u from clusters of set R_i.
- Each vertex $u \in \mathcal{N}_i$ keeps the bucket structure $B_i(u)$ for its neighboring clusters to maintain $\mathcal{E}_i(u)$. The set E_{i+1} will be defined as follows. Consider an edge $(u, v) \in E_i$ such that both u and v are present in V_{i+1} too. The edge (u, v) will be present in E_{i+1} only if u and v belong to different clusters in C_{i+1} and one of the following conditions hold.
  - at least one of u and v belongs to a sampled cluster at level i.

– (u, v) belongs to $\mathcal{E}_i(u)$ as well as $\mathcal{E}_i(v)$.

- The spanning forest for C_i is defined as $F_{i+1} = F_i(R_i) \bigcup \{hook(u, i) | u \in \mathcal{N}_i\}$

Let $F = \cup_i F_i$. The edges of this set are contributed by the NEW-CLUSTERING-INVARIANT. We now define the set E_S which contains the edges contributed by NEW-SPANNER-EDGE-INVARIANT. Each vertex $u \in V_i$ belonging to unsampled cluster at level i maintains this invariant defined as follows. NEW-SPANNER-EDGE-INVARIANT : If u is not present at level $i+1$, then u contributes one edge from $E_i(u, c)$ in the set E_S for each neighboring cluster $c \in C_i$. If u is present at level $i+1$, then u contributes one edge from $E_i(u, c)$ to E_S for each neighboring cluster $c \in C_i$ belonging to inactive bucket in $B_i(u)$.

The set $F \cup E_S$ will be the spanner maintained by the algorithm at any stage. Along same lines to Lemma 2.2, it follows that $F \cup E_S$ will be a $(2k-1)$-spanner at each stage. We now prove the following lemma which follows from these invariants and the bucket structure.

Lemma 4.4 *Probability that an edge $(u, v) \in E_i$ is* hook*(u, i) is bounded by $\frac{1+2\epsilon}{|\mathcal{E}_i(u)|}$, for any constant $0 < \epsilon < 1/2$.*

Proof: First note that an edge (u, v) can be *hook*(u, i) only if it is present in $\mathcal{E}_i(u)$. Let X be the random variable for the number of edges incident on u from sampled clusters at level i. It follows from Lemma 4.3 that X is less than $(1-\epsilon)n^{-1/k}|\mathcal{E}_i(u)|$ with probability at most n^{-a}. Let us now consider an edge $(u, v) \in \mathcal{E}_i(u)$, and let v belong to cluster $c \in C_i$. The necessary condition for (u, v) to be *hook*(u, i) is that the cluster c must be a sampled cluster at level i, the probability of which is $n^{-1/k}$. Furthermore, the following inequality follows due to independence incorporated in sampling the clusters.

$$\mathbf{Pr}[X \le d | c \in R_i] \le \mathbf{Pr}[X \le d] \qquad \text{for any constant } d > 0 \tag{2}$$

Applying the new clustering invariant, we can bound the probability for edge (u, v) to be *hook*(u, i) as follows.

$$\begin{aligned}
\mathbf{Pr}[hook(u,i) = (u,v)] &= \left(\sum_{d \ge 1} \frac{\mathbf{Pr}[X = d | c \in R_i]}{d} \right) \mathbf{Pr}[c \in R_i] \\
&= n^{-1/k} \sum_{d \ge 1} \frac{\mathbf{Pr}[X = d | c \in R_i]}{d} \\
&\le n^{-1/k} \frac{\mathbf{Pr}[X \le d | c \in R_i]}{1} + n^{-1/k} \frac{\mathbf{Pr}[X > d | c \in R_i]}{d} \qquad \{\text{for any } d \ge 1\} \\
&\le n^{-1/k} \mathbf{Pr}[X \le d] + n^{-1/k} \frac{1}{d} \qquad \{\text{using Equation 2}\} \\
&\le n^{-1/k} n^{-a} + n^{-1/k} \frac{1}{(1-\epsilon) n^{-1/k} |\mathcal{E}_i(u)|} \qquad \{\text{for } d = (1-\epsilon) n^{-1/k} |\mathcal{E}_i(u)|\} \\
&\le n^{-a} + \frac{1}{(1-\epsilon)|\mathcal{E}_i(u)|} \le \frac{1+2\epsilon}{|\mathcal{E}_i(u)|} \qquad \{\text{for } \epsilon < \frac{1}{2}\}.
\end{aligned}$$

•

We now analyse the size of the spanner maintained by our algorithm.

Lemma 4.5 *The expected size of the spanner built by following the new invariants and bucket structure is $O(k/\epsilon^8 n^{1+1/k} \log^2 n)$.*

Proof: A vertex, at any level, adds at most one edge to maintain NEW-CLUSTERING-INVARIANT, and one edge per neighboring cluster belonging to inactive buckets. The latter edge is added to maintain NEW-SPANNER-EDGE-INVARIANT. Thus the edges added by any vertex at any level is bounded by the number of neighboring clusters belonging to inactive buckets.

We shall use different thresholds for activation and deactivation of buckets. An active bucket will be marked as inactive as soon as it has fewer than β clusters, where $\beta = \Theta(n^{1/k} \frac{\log n}{\epsilon^6})$. However, an inactive buckets is not marked as active until it contains at least β/ϵ^2 clusters. Thus in the worst

case, an inactive bucket can have at most $\Theta(n^{1/k}\frac{\log n}{\epsilon^8})$ clusters. There are $O(\log n)$ buckets storing the clusters neighboring to u at level i, and there are k levels. Thus a vertex contributes at most $O(k/\epsilon^8 n^{1/k}\log^2 n)$ edges to the spanner, and the lemma follows. •

4.6 Handling the updates by the dynamic algorithm

Upon a single edge insertion/deletion in the graph, we first add the two updates, each meant for one of the endpoints of the edge, at level 0. The fully dynamic algorithm processes the updates in a bottom up fashion starting from level 0. The processing of updates at level i will restore the invariants at level i and generate the updates for level $i+1$. Once all the updates for level i have been processed, we move to level $i+1$ and proceed in a similar manner. Instead of analysing the updates generated and processed at all levels together, we may focus on just two consecutive levels i and $i+1$ only. This is because our fully dynamic algorithm essentially achieves the following objective which is somewhat stronger than we require.

Consider any subgraph and clustering at level i. Let each cluster at level i be sampled independently with probability $n^{-1/k}$. Each vertex at level i maintains NEW-CLUSTERING-INVARIANT and/or NEW-SPANNER-EDGE-INVARIANT and this completely defines subgraph and clustering at level $i+1$. Consider any set of updates at level i. These include insertion/deletion of edges and change in clustering of vertices. As remarked earlier, change in clustering of a vertex v at level i is equivalent to at most two updates for each of its neighbors. Each vertex will process these updates in a sequential manner to restore its invariants. Our algorithm ensures that by processing each update at level i, the expected amortized number of updates generated for level $i+1$ would be $7+14\epsilon$.

We now highlight one more interesting feature of our algorithm. Consider deletion or insertion of an edge $(u,v)\in E_i$ to be processed by u. If this edge is incident on u from sampled cluster, then NEW-CLUSTERING-INVARIANT for u gets violated and has to be restored but the bucket structure of u will remain intact. On the other hand, if the edge is incident from unsampled cluster, the bucket structure may have to be updated in order to maintain NEW-SPANNER-EDGE-INVARIANT, but the NEW-CLUSTERING-INVARIANT remains intact. Therefore, NEW-CLUSTERING-INVARIANT and the bucket structure can be maintained independent of each other. This feature helps in a cleaner description and analysis of the algorithm. We now describe the procedure for handling deletion or insertion of an edge at level i.

Handling an edge deletion. Consider deletion of an edge, say (u,v), to be processed by u.

If v belongs to a sampled cluster in C_i, then the deletion of (u,v) may violate the CLUSTERING-INVARIANT only if $\mathit{hook}(u,i)=v$. We proceed as follows in this situation. If $R_i(u)$ is non-empty, then we select another hook for u from $R_i(u)$ uniformly randomly, and change the cluster of u at level $i+1$ accordingly. If $R_i(u)$ has become empty, then vertex u leaves clustering C_{i+1} and starts maintaining NEW-SPANNER-EDGE-INVARIANT at level i. In each of these cases, for each $(u,w)\in\mathcal{E}_i(u)$, at most two updates get generated for level $i+1$.

If v belongs to an unsampled cluster in C_i, we restore NEW-SPANNER-EDGE-INVARIANT and update the bucket structure $B_i(u)$ if needed. Using Theorem 4.3, amortized $4+10\epsilon$ number of additional updates get generated for level $i+1$ in this case.

The deletion of (u,v) will have to be passed to level i if $(u,v)\in E_{i+1}$. In addition to this update, additional updates will also be passed onto level $i+1$ which get generated due to its processing at level i as described above. It follows from Lemma 4.4 that the probability of "$\mathit{hook}(u,i)=(u,v)$" before deletion of (u,v) is bounded by $\frac{1+2\epsilon}{|\mathcal{E}_i(u)|}$, and only in that case, at most $2|\mathcal{E}_i(u)|$ updates are generated for level $i+1$. Hence, the expected amortized number of updates introduced at level $i+1$ when vertex u processes deletion of an edge from $E_i(u)$ is bounded by

$$1+\left(\frac{1+2\epsilon}{|\mathcal{E}_i(u)|}2|\mathcal{E}_i(u)|+4+10\epsilon\right)\leq 7+14\epsilon$$

Handling an edge insertion. Consider insertion of an edge, say (u,v), to be processed by u.

If v belongs to a sampled cluster at level i, then CLUSTERING-INVARIANT has to be restored. If there were s edges incident from sampled clusters at level i prior to the insertion, then with

probability $1/(1+s)$, the edge (u,v) would be selected as $hook(u,i)$. In this case, the cluster of u at level $i+1$ changes and at most $2|\mathcal{E}_i(u)|$ updates are passed to level $i+1$. However, if $hook(u,i)$ is not changed to (u,v), then just a single update is passed onto the next level.

If v belongs to an unsampled cluster at level i, vertex u processes the insertion of (u,v) to restore NEW-SPANNER-EDGE-INVARIANT and updates the bucket structure if needed. Using Theorem 4.3, amortized $4+10\epsilon$ number of additional updates gets generated for level $i+1$ in this case.

The insertion of (u,v) will have to be passed to level i if u and v belong to different clusters in C_{i+1}. In addition to this update, additional updates will also be passed onto level $i+1$ which get generated due to its processing at level i as described above. To analyze the expected number of total updates, we apply Lemma 4.4 after the insertion. It follows that "$hook(u,i) = (u,v)$" with probability $(1+2\epsilon)/|\mathcal{E}_i(u)|$, and only in that case at most $2|\mathcal{E}_i(u)|$ updates would have to be passed to level $i+1$. Hence combining together various cases and their associated probabilities, the expected amortized number of updates introduced at level $i+1$ when vertex u processes insertion of an edge into $E_i(u)$ is

$$1 + \left(\frac{1+2\epsilon}{|\mathcal{E}_i(u)|} 2|\mathcal{E}_i(u)| + 4 + 10\epsilon\right) \le 7 + 14\epsilon$$

So we can conclude the following lemma.

Lemma 4.6 *While processing any update at level i for maintaining the two invariants and the bucket structure of a vertex u, the expected amortized number of updates introduced at level $i+1$ is at most $7+14\epsilon$ for any $0 < \epsilon \le 1/4$.*

Computing the updates for level $i+1$ from updates at level i.
Consider any sequence of updates at level i to be processed by a vertex, say x. It processes these updates in a sequential manner as described above and generates updates for its neighbors at level $i+1$. However, many of these new updates will be superfluous as explained by the following two cases.

1. While processing the updates which involve maintenance of NEW-SPANNER-EDGE-INVARIANT, a cluster neighboring to x may hop to many buckets of different activation status. However, it is only the final bucket which it joins that determines the new activation status of (the edges incident on x from) that cluster. All the intermediate changes are superfluous.

2. While processing the updates which involve maintenance of NEW-CLUSTERING-INVARIANT, let x changes $hook(x,i)$ more than once. In this situation it is only the final $hook(x,i)$ that determines the clustering of x at level $i+1$; the rest of them are all transient. From perspective of all neighbors of u, the updates associated with these transient hooks of x are superfluous.

Among all the updates which x generates for level $i+1$ by processing updates at level i, all the superfluous updates will be eliminated. The task of eliminating the superfluous updates can be done easily in time of the order of the number of updates generated by x. This produces the list of updates generated by x for level $i+1$. All lists of updates, generated by different vertices at level i, are joined together to form the list of updates for level $i+1$. An interesting consequence of eliminating these superfluous updates is that any vertex x at level i will generate at most $O(\deg(x))$ updates for level $i+1$. This leads to the following important corollary.

Corollary 4.3.1 *Upon any insertion or deletion of an edge, the number of updates generated by the algorithm for level i will be $O(m)$.*

4.6.1 Subtle technical points of the algorithm

Before we analyse the time complexity of our fully dynamic algorithm, we would like to mention the following subtle technical points about the fully dynamic algorithm described above.

1. *Processing of updates by vertices of a sampled cluster*

   If v belongs to a sampled cluster at level i, then it *does not* have to do any processing. This is because it does not maintain any invariant. However, in this case, v updates its data structure $D_i(v)$ whenever it receives any edge update at level i and passes on the same update to level $i+1$ if necessary.

2. *Order of processing updates at a level*

   Each vertex processes its updates at level i in a sequential manner. The set of updates generated for level $i+1$ may depend upon the order in which vertex v processes its updates at level i. This can be understood through the following example.

   Suppose there are some insertions and deletions of edges incident on a vertex v at level i. Furthermore, suppose the other endpoint of each of these edges belong to the same cluster $c \in C_i$, and the number of edge insertions is at least the same as the number of edge deletions. Let the cluster c be on the verge of becoming inactive in $B_i(v)$ just before the sequence of these updates. If all the edge deletions appear before all the edge insertions in this sequence, then the activation status of c will become inactive, and this will lead to a substantial number of updates for level $i+1$. On the other hand, if all the edge insertions appear before all the deletions, activation status of c will not change.

3. *The bound on the number of updates generated*

   It follows from point *2* above that at any given intermediate stage, the number of updates generated for level $i+1$ may depend upon the type (insertion/deletion) of updates processed at level i and the order in which they are processed. However, this does not contradict the claim of Lemma 4.6 that processing an update at a level i generates expected amortized $7+14\epsilon$ updates for level $i+1$. In particular, the bound $7+14\epsilon$ holds independent of the type (insertion or deletion) of updates we process. The reason for this is as follows. This bound is derived using the fact that the invariants associated with the processing vertex hold just before the update, and the processing of each update restores the invariants after each update as well.

4. *conflicting updates*

   Notice that each vertex processes its updates quite locally, as a result, it is possible that endpoints of an edge, say (u, v), generate conflicting update for level $i+1$. To understand this point, consider the following example. Let (u, v) belongs to active bucket in the data structure of u and inactive bucket in the data structure of v. While processing some updates at level i, the activation status of edge (u, v) changes in bucket structures of u as well as v. As a result, u would generate an update of deletion of edge (u, v) for level $i+1$ while v generates an update of insertion of edge (u, v). As follows from definition of E_{i+1}, such a conflict is resolved by allowing deletion to override insertion.

5. *bucket structures at level 0 and $k-1$*

   Notice that the role of a bucket structure at any level i is to ensure the claim stated in Lemma 4.4. But maintaining the bucket structures also have an additional overhead in terms of update time and space. However, there is no need to keep bucket structure for any vertex at levels $k-1$ and 0 due to the following reasons. At level $k-1$, we just have to maintain SPANNER-EDGE-INVARIANT and so there is no need to ensure Lemma 4.4 for $i = k-1$. At level 0, Lemma 4.4 already holds as shown in the fully dynamic algorithm for 3-spanner.

6. *a vertex hoping from a sampled cluster to an unsampled cluster*

   Suppose u belongs to a sampled cluster at level i. Notice that vertices belonging to a sampled cluster do not have to maintain any invariant. Suppose there is an update that makes u leave its current cluster and join some unsampled cluster at level i. In this case, it will have to maintain NEW-SPANNER-EDGE-INVARIANT and hence the bucket structure. So there is an additional overhead of building the bucket structure which will take $O(|E_i(u)|)$ time. But notice the following two points here. Firstly this overhead is for level i only. Secondly, there will certainly be $\Theta(|E_i(u)|)$ updates in this situation at level i. This is because whenever a vertex changes its cluster at level i, there are definitely $\Theta(|E_i(u)|)$ updates to be processed by its neighbors at level i. Considering these two points, we can see that the overhead of building the bucket structure is subsumed by the computation which the neighbors of u will anyway be doing.

4.6.2 Analysis of the algorithm

Let Z_i be the random variable for the number of updates that reach level i upon a single edge update in the graph. Then using Lemma 4.6 it follows that

$$\begin{aligned}\mathbf{E}[Z_i] &= \sum_{\alpha} \mathbf{E}[Z_i|Z_{i-1} = \alpha]\mathbf{Pr}[Z_{i-1} = \alpha] \\ &= \sum_{\alpha} (7 + 14\epsilon)\ \alpha\ \mathbf{Pr}[Z_{i-1} = \alpha] = (7 + 14\epsilon)\mathbf{E}[Z_{i-1}]\end{aligned}$$

The processing cost of an update at a level is of the order of the number of updates it generates for the next level. Hence the time complexity of maintaining $(2k-1)$-spanner upon an edge insertion/deletion is of the order of the sum of the number of updates generated at all levels. Combining this observation with the above equation, it follows that the expected amortized time to update the spanner upon any single edge insertion or deletion is of the order of $(7 + 14\epsilon)^k$ for any $0 < \epsilon \leq 1/4$. Choosing $\epsilon < \frac{1}{2k}$ and using Lemma 4.5, we can conclude the following Theorem.

Theorem 4.4 *Given an undirected unweighted graph on n vertices and an integer k, we can maintain a $(2k-1)$-spanner of the graph fully dynamically in expected amortized $O(7^k)$ time per update. The expected size of the spanner at any stage will be $O(k^9 n^{1+1/k} \log^2 n)$.*

The expected update time can be improved further to $O(7^{k/2})$ by reducing the number of levels in the hierarchy of clusterings from k to $k/2$. Note that we used k levels since at level $k-1$, the expected number of clusters is only $n^{1/k}$ and we can add a single edge between a vertex and every neighboring clusters at this level. We basically used part (i) of Observation 2.1. There is an alternate solution to achieve expected $O(kn^{1+1/k})$ size of spanner based on part (ii) of Observation 2.1. In this solution, we keep hierarchy up to level $\lfloor \frac{k}{2} \rfloor$ only. There will be expected $n^{1-\frac{1}{k}\lfloor \frac{k}{2} \rfloor}$ number of clusters at top level in this solution.

- if k is odd, then for each pair of neighboring clusters $c \in C_{\lfloor k/2 \rfloor}, c' \in C_{\lfloor k/2 \rfloor}$ at level $\lfloor \frac{k}{2} \rfloor$, we keep one edge between them in the spanner.
- if k is even, then for each pair of neighboring clusters $c \in C_{k/2}$, and $c' \in C_{k/2-1}$, we keep one edge between them in the spanner.

It follows from part (ii) of Observation 2.1 that for all edges at the highest level $\lfloor \frac{k}{2} \rfloor$ of hierarchy, the proposition $\mathcal{P}_{2k-1}$ holds true and the expected number of edges added to the spanner at this level will be $O(n^{1+1/k})$. Following this and 5th technical point, it can be seen that the number of levels at which we need to keep bucket structure will be reduced to $\lfloor k/2 \rfloor - 2$. We conclude with the following theorem.

Theorem 4.5 *Given an undirected unweighted graph on n vertices and an integer k, we can maintain a $(2k-1)$-spanner of the graph fully dynamically in expected amortized $O(7^{k/2})$ time per update. The expected size of the spanner at any stage will be $O(k^9 n^{1+1/k} \log^2 n)$.*

Remark 4.7: It follows from Corollary 4.3.1 that the worst case update time guaranteed by our algorithm will be $O(km)$. The worst case update time guaranteed by the algorithm of Elkin [25] is $O(m)$.

4.7 Analysis of dynamic buckets

First we briefly explain how the data structure D_i associated with a vertex at any level i (see Subsection 2.2) is augmented and extended to incorporate bucket data structure. Each vertex u will keep an array $b_u[\,]$ of size $O(\log_{1/\epsilon} n)$ such that $b_u[j]$ would store a doubly linked circular list for the centers of those clusters neighboring to u which are present in jth bucket. It also stores the total count of these clusters. In addition to it, the hash table h_u storing the clusters neighboring to u also stores, for each neighboring cluster, the bucket to which it belongs at a given moment. As clusters move from one bucket to another, the entries of array $b_u[]$ are updated accordingly. Using these data

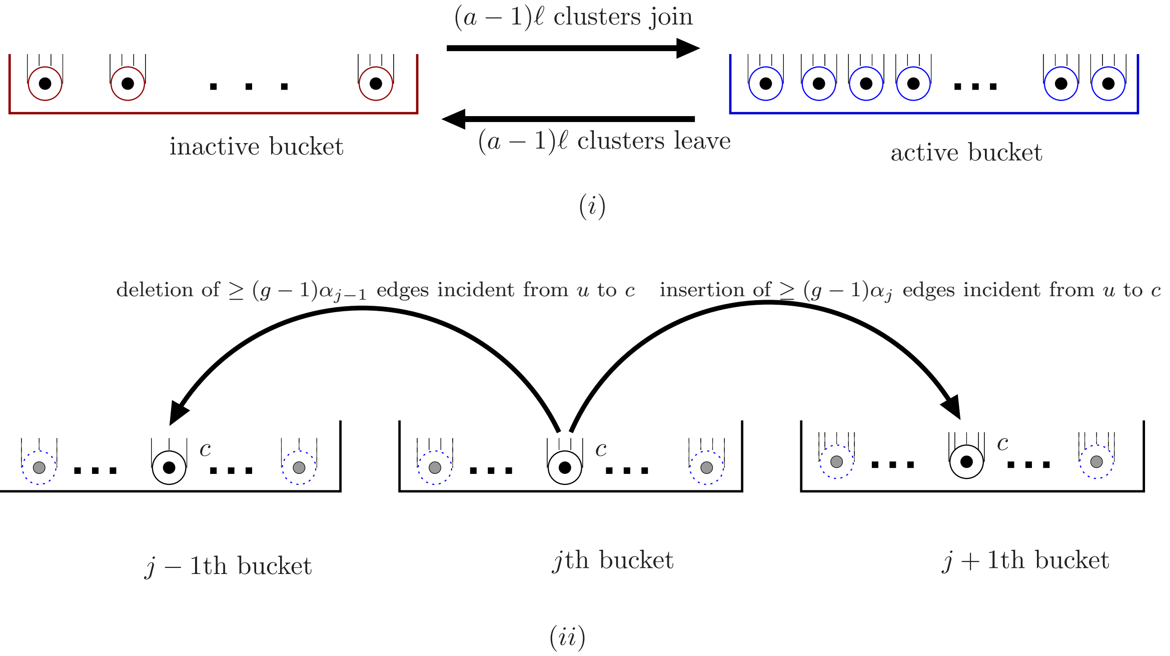

Figure 9: (i) Change in the activation status of a bucket, (ii) cluster c hopping from jth bucket to its neighboring buckets.

structures, it is easy to determine in $O(1)$ time if an edge incident on u at level i is incident from a cluster of active bucket or inactive bucket.

Let us now address the maintenance of bucket data structure under deletion/insertion of edges. It can be seen that there may be the following kinds of changes in the bucket data structure : The number of edges $E_i(u, c)$ between u and a cluster c changes as we insert or delete edges. It may change to such an extent that $|E_i(u, c)|$ either falls below LOWER-LIMIT of the bucket or goes over the UPPER-LIMIT of the bucket. So we have to move the cluster to another bucket and if the new bucket has different activation status than the previous one, then the activation status of all the edges of the cluster will also change. Furthermore, due to the movement of these clusters, the size of buckets also change, and so an active bucket may become inactive and vice versa. In order to enhance clarity and compactness of notations, we shall describe and analyze the following generic bucket structure maintained by a vertex u at a level $j > 0$. The exact bound (mentioned in Theorem 4.3) will seamlessly follow if we just set proper values to the free parameters used here.

- The jth bucket will have clusters whose neighborhood size (number of edges with which the cluster is incident on u) is in the range $[\alpha_j/g, g\alpha_j]$ for some constant $g > 1$. The geometric mean of the two extremes of this range is α_j, and hence we call it the *mid-size* of the bucket. It is that critical size at which clusters hop into this bucket. The mid-sizes are related as follows: $\alpha_{j+1} = g\alpha_j$. So a cluster enters into jth bucket when its size is α_j and hops to $(j+1)$th bucket (or $(j-1)$th bucket) when its size rises to $g\alpha_j$ (drops to α_j/g).

- We shall use the convention that a bucket is *inactive* if its size (the number of clusters in it) is less than ℓ, and it is active if its size is greater than $a\ell$ for some $a, \ell > 1$. The bucket with size in the range $[\ell, a\ell]$ is allowed to have any activation status in the beginning. An inactive bucket will be activated when its size reaches $a\ell$; and an active bucket will be deactivated when its size falls to ℓ.

Figure 9 provides a sketch of the generic dynamic bucket structure described above. There are two major events that we need to handle and analyze in the dynamic maintenance of bucket structure.

1. A cluster hopping to a new bucket.

2. A bucket changing its activation status.

When the first event occurs, the activation status of all the edges coming from the cluster has to be changed if the activation status of the new bucket is different from that of the old one. When the second event occurs, the activation status of all the edges in the bucket has to be changed. So whenever these major events occur, a large number of edges could change their activation status. Potentially all these changes may have to be communicated to level $i+1$. In this way, an edge update can trigger a huge number of updates to be passed to level $i+1$. However, by using a credit based amortized analysis, it can be shown that the number of activation changes of all the edges in the bucket structure of u at level i is of the order of the number of edge updates (insertion/deletion) reaching u at level i. The whole analysis is similar in spirit to the analysis of the well-known data structure of dynamic tables (Chapter 18, [15]). Consider an edge update reaching level i which is to be processed by u. Let it be insertion or deletion of some edge (u, v). We shall give some credits to this update. These credits will be passed onto the cluster (in the bucket structure of u) to which v belongs. Let the amount of this credit be π_i for edge insertions, and π_d for edge deletions. The credits obtained by a cluster will be partially used for the change in its own activation status when it moves to another bucket. And the remaining credit will be passed to the current bucket as well as the bucket to which the cluster will move in next hop in order to pay for any possible change in the activation status of these buckets.

Clearly there is a hierarchical structure in the movement of credits: an edge update passes credits to its cluster, and a cluster passes credits to the buckets that contains it (or will contain it in immediate future). So to find out the appropriate values for π_i and π_d, we take the following backward approach.

We have to ensure that whenever a bucket changes its activation status, it has accumulated enough credits to pay for the change in the activation status of all its edges. Let us analyze the transition of jth bucket from being inactive to becoming active. Consider the moment when the bucket had just become inactive. At that moment it had ℓ clusters, and so a maximum of $g\ell\alpha_j$ edges. From that moment to the present, when the bucket becomes active, there must be at least $(a-1)\ell$ clusters which have joined it. Each such cluster joined with α_j edges. To each of these $(a-1)\ell$ new clusters, some new edges incident from u might have got inserted ever since it joined the bucket. However, while activating the entire bucket, the activation cost for these new edges will be paid by themselves. So we need to only bother about the activation of the remaining edges in the bucket : which consists of old $g\ell\alpha_j$ edges and α_j edges per new cluster. Dividing this cost equally over the new $(a-1)\ell$ clusters, each cluster must transfer a credit of $\frac{(a-1)+g}{a-1}\alpha_j$ to the jth bucket at the time of joining it. Now let us consider deactivation of jth bucket. From the moment it was activated to the moment it became inactive, it must have lost at least $(a-1)\ell$ clusters. Now there are ℓ clusters, each with at most $g\alpha_j$ edges, left in the bucket, and their deactivation cost has to be paid by the clusters that left the bucket when it was active. So a credit of $\frac{g}{a-1}\alpha_j$ must be paid by each cluster that leaves the jth bucket.

Now let us analyze movement of a cluster from bucket j to $j+1$ (or $j-1$). At the time the cluster moved to bucket j, it had exactly α_j edges. If it moves to the $(j+1)$th bucket by increasing its size to $g\alpha_j$, it must carry with it a credit of $\frac{(a-1)+g}{a-1}g\alpha_j$ for activating the $(j+1)$th bucket in future, and transfer a credit of $\frac{g}{a-1}\alpha_j$ for deactivating the old bucket. In addition, if the current activation status of $(j+1)$th bucket differs from that of jth bucket, it must pay for the change in its own activation status with a credit of $g\alpha_j$. All these costs must be attributed to the edges that are inserted to it ever since it joined jth bucket, these edges are at least $(g-1)\alpha_j$ in number. So for each edge insertion it suffices to assign $\pi_i = \frac{2g(a-1)+g^2+g}{(a-1)(g-1)}$ credits. If the cluster moved to $(j-1)$th bucket after decreasing its size to α_j/g, then it must carry with it a credit of $\frac{a-1+g}{a-1}\frac{\alpha_j}{g}$ for activating the new bucket in future, and a credit of $\frac{g}{a-1}\alpha_j$ has to be paid to the old bucket for its deactivation. At the time it moves to bucket $j-1$, it must also use α_j/g credits for the possible change of its own activation status. All these costs have to be paid by edges that were deleted from the cluster during this period, they are at least $\alpha_j - \alpha_j/g$ in number. So for each edge deletion it suffices to assign $\pi_d = \frac{2(a-1)+g^2+g}{(a-1)(g-1)}$ credits. Comparing π_i with π_d, and using $g > 1$ we can conclude the following Lemma.

Lemma 4.8 *A single edge insertion or deletion in the bucket structure of a vertex u leads to a change in the activation status of amortized at most $\frac{2g(a-1)+g^2+g}{(a-1)(g-1)}$ additional edges in $E_i(u)$.*

In other words, a single edge update in the bucket structure leads to insertion or deletion of amortized $\frac{2g(a-1)+g^2+g}{(a-1)(g-1)}$ additional edges in $\mathcal{E}_i(u)$. Recall that whether an edge (u, v) belongs to E_{i+1} is determined by both u and v - the edge belongs to E_{i+1} if $(u, v) \in \mathcal{E}_i(u)$ and $(u, v) \in \mathcal{E}_i(v)$. So whether the deletion (or insertion) of an edge, say (u, v) to $\mathcal{E}_i(u)$ will be reflected in E_{i+1} is determined by the status of (u, v) in the bucket structure of v as well. The latter information can be computed in $O(1)$ time using the augmented data structure D_i. In case the insertion (or deletion) of the edge in $\mathcal{E}_i(u)$ implies insertion (or deletion) of the edge in E_{i+1}, it would be equivalent to two updates for level $i+1$ to be processed by each endpoint of the edge separately. Thus we may conclude the following lemma.

Lemma 4.9 *For each edge update processed by bucket structure of u at level i, the amortized number of additional updates to be sent to the level $i+1$ will be at most $2\frac{2g(a-1)+g^2+g}{(a-1)(g-1)}$.*

Choosing $g = 1/\epsilon$ and $a = g^2$ for suitably small value of ϵ, Lemma 4.9 leads to Theorem 4.3.

5 Fully Dynamic Algorithm - II

We first present a decremental algorithm for maintaining a $(2k-1)$-spanner with expected amortized $O(k^2 \log n)$ update time per edge deletion. We then extend it to the fully dynamic environment using the idea of periodic rebuilding which is a standard idea in the area of dynamic algorithms [33, 34]. Interestingly, this extension is achieved at the expense of increasing the spanner size and update time by just a logarithmic factor. Since $k = O(\log n)$, the expected amortized update time guaranteed by this fully dynamic algorithm is always $O(\log^4 n)$. Therefore, for large values of stretch, this algorithm has a significant advantage over fully dynamic algorithm I described earlier.

At the core of the efficient decremental algorithm lies a new kind of clustering of vertices. We now describe this new clustering along with its construction and maintenance.

5.1 New clustering

Consider a triplet (S, i, σ), where S is a subset of vertices, i is a positive integer, and σ is a permutation of S. We shall use the notation $\sigma(x) < \sigma(y)$ for $x, y \in S$, if vertex x precedes y in the permutation σ. Let us now consider the set of those vertices of the graph which are within distance i from any vertex in S. The triplet (S, i, σ) defines a clustering on this set as follows. For a vertex v of this set, the center of the cluster to which v belongs is the vertex $x \in S$ such that for every $y \in S \backslash \{x\}$, the following relation is satisfied.

$$\delta(v, x) < \delta(v, y) \quad \textbf{or} \quad \delta(v, x) = \delta(v, y) \textbf{ and } \sigma(x) < \sigma(y)$$

In other words, vertex v selects the nearest vertex from S as its cluster center; and in case v has multiple nearest vertices, it selects that vertex among them which appears first in the permutation σ. The uniqueness of the cluster center for v follows immediately. This clustering defined by the triplet (S, i, σ) will be denoted by $C(\sigma(S), i)$ henceforth.

Let us address the maintenance of clustering $C(\sigma(S), i)$ while the edges are being deleted. Let $C[\,]$ denote the array used for storing the clustering $C(\sigma(S), i)$. At any instance, let $d[v]$ denote the distance $\delta(v, S)$. We say that clustering $C(\sigma(S), i)$ *changes* for vertex v due to an edge deletion if the deletion leads to either an increase in $d[v]$ or a change in $C[v]$. For any fixed σ, there may exist a sequence of edge deletions during which the clustering may change for v arbitrarily large number of times. However, if we select the permutation σ randomly uniformly, this number will be quite small on expectation as shown by the following lemma.

Lemma 5.1 *Let v be a vertex in $V \backslash S$ and $d[v] \le i$ in the beginning. Consider any sequence of edge deletions. If σ is a uniformly random permutation of S, the expected number of times the clustering $C(\sigma(S), i)$ changes for vertex v during this sequence is $O(i \log n)$.*

Proof: Observe that $d[v] \ge 1$ in the beginning, and $d[v]$ can only increase during edge deletions. So we can partition any given sequence of edge deletions into maximal contiguous subsequences such that $d[v]$ remains unchanged during each subsequence. Note that v ceases to belong to $C(\sigma(S), i)$

once $d[v]$ exceeds i, so we just need to consider at most i such subsequences. Consider any $\ell \leq i$, and let $\mathcal{A}$ be the subsequence during which $d[v] = \ell$. We shall show that the clustering $C(\sigma(S), i)$ would change for v only $O(\log n)$ times on expectation during the entire subsequence $\mathcal{A}$.

Let $O \subseteq S$ be the set of vertices lying at distance exactly ℓ from v just before the beginning of subsequence $\mathcal{A}$. So vertices of set O are the only potential centers whose clusters v can join during $\mathcal{A}$. Furthermore, if v ever leaves a cluster, say centered at $x \in O$, it will never join the cluster centered at x again during $\mathcal{A}$. Thus we just need to show that there are expected $O(\log n)$ cluster centers from O whose cluster v joins during $\mathcal{A}$.

For each $o \in O$, there exists an edge in the subsequence $\mathcal{A}$ whose deletion causes $\delta(v, o)$ to become greater than ℓ. Let $\langle o_1, ..., o_t \rangle$ be the sequence of vertices of O arranged in the (chronological) order in which they cease to be at distance ℓ from v. Conditioned on this sequence, what is the probability that v ever joins the cluster centered at o_j during $\mathcal{A}$ for a given $1 \leq j \leq t$? For this to happen, o_j must appear first among $\{o_j, ..., o_t\}$ in the permutation σ. Since σ is a uniformly random permutation, the probability of this event is $1/(t - j + 1)$. Hence, applying linearity of expectation, the expected number of centers from O whose clusters v joins during $\mathcal{A}$ is $\sum_{j=1}^{t} \frac{1}{t-j+1} = O(\log n)$. •

Though Lemma 5.1 ensures that the expected number of changes in the clustering $C(\sigma(S), i)$ will be quite *small* when the edges are being deleted, we would need a data structure to detect these changes and to update the clustering efficiently. It turns out that computing and maintaining the clustering $C(\sigma(S), i)$ is similar to building and maintaining a breadth first search (BFS) tree upto depth $i + 1$.

5.2 Efficient construction and maintenance of the new clustering

We first address the issue of efficient initialization of the array $C[\ ]$ which stores the clustering $C(\sigma(S), i)$. Note the following observation which can be inferred from the definition of $C(\sigma(S), i)$.

Observation 5.1 *In order to compute $C[v]$ for a vertex v in the clustering $C(\sigma(S), i)$ it suffices to know $C[x]$ for each neighbor x of v which lies closer to S than v, i.e., $d[x] = d[v] - 1$.*

This observation suggests that if we know $C[x]$ for each x with $d[x] = j$, then we may compute $C[v]$ for all vertices with $d[v] = j + 1$. Based on this idea, **Algorithm 1** computes the clustering $C(\sigma(S), i)$ in the beginning. A proof by induction on the distance of vertices from S shows that $C[\ ]$

Algorithm 1: `Computing` $C(\sigma(S), i)$.

```
foreach v ∈ V do  visited(v) ← false;
foreach s ∈ S do
    C[s] = s;
    visited[s] ← true;
    d[s] ← 0;
F ← ∅;
Let Q be a queue which stores vertices of S in the order defined by σ;
while Q ≠ ∅ do
    x ← Dequeue(Q);
    foreach (x, y) ∈ E do
        if visited(y)=false then
            visited(y) ← true;
            C[y] ← C[x];
            F ← F ∪ {(x, y)};
            d[y] ← d[x] + 1;
            if d[y] < i then Enqueue(y, Q)
```

stores the clustering $C(\sigma(S), i)$. This algorithm also computes a forest $\mathcal{F}$ spanning (and defining) the clustering $C(\sigma(S), i)$. Each tree in this forest corresponds to a spanning tree of a cluster of radius i centered at some vertex in S. Hence radius of clustering $C(\sigma(S), i)$ is i.

A careful reader might find **Algorithm** 1 similar to the standard algorithm for computing a BFS tree from a vertex in a graph. In fact a forest $\mathcal{F}$ defining the clustering $C(\sigma(S), i)$ can be associated with a truncated BFS tree in a slightly modified graph in the following way. Add a dummy vertex g and the edges $\{(g, s)|s \in S\}$ to G; now do a BFS traversal upto depth $i+1$ from g with the following constraints. Firstly, visit the neighbors of g in the order $\sigma(S)$; and secondly, process the vertices in the order they are visited. As a result, we shall obtain a BFS tree all of whose edges, excluding $\{(g, s)|s \in S\}$, correspond to the forest $\mathcal{F}$ defining the clustering $C(\sigma(S), i)$. This insight about **Algorithm** 1 implies that maintaining $C(\sigma(S), i)$ under deletion of edges amounts to maintaining this BFS tree under deletion of edges. Even and Shiloach [29] designed an algorithm which maintains one BFS tree (out of potentially many BFS trees) of depth i under edge deletions. However, a BFS tree associated with the clustering $C(\sigma(S), i)$ has some additional constraints as explained above. Therefore, in order to maintain the clustering we suitably modify this algorithm.

We shall maintain the following two additional fields for each vertex v. First field is $parents(v)$ which is the set consisting of every neighbor w of v such that $d[w] = d[v] - 1$ and $C[w] = C[v]$. For $\mathcal{F}$, we may select one edge from $parents(v)$ for each v in the clustering. The second field is $children(v)$ which is the set of neighbors x of v such that $v \in parents(x)$.

Consider deletion of an edge (u, v); without loss of generality, let $d[u] \leq d[v]$ at the time of deletion of (u, v). Procedure `Update-clustering` updates the clustering upon deletion of edge (u, v). It returns two sets Y, Z defined as follows. Y consists of triplets of the form $\langle x, c, c' \rangle$ which represents that vertex x has changed its cluster from c to c' in clustering $C(\sigma(S), i)$ due to the edge deletion. Similarly Z consists of pairs of the form $\langle x, c \rangle$ which represents that vertex x, which belonged to cluster $c \in C(\sigma(S), i)$ earlier, now ceases to belong to any cluster in clustering $C(\sigma(S), i)$ after deletion of (u, v). We now provide an overview of this procedure. Observe that deletion of edge (u, v) may lead to a change in the clustering only if $u \in parents(v)$. The procedure begins with deletion of u from $parents(v)$ and deletion of v from $children(u)$. If $parents(v)$ is still non-empty, there is no change in the clustering. However, if $parents(v)$ is empty, then the clustering of v will change. Note that the change in clustering of v will lead to change in clustering of those children of v whose *parents* field consists of v only. In this manner, the deletion of (u, v) may cause a change in clustering of many descendants of v. During the procedure `Update-clustering` all such vertices are computed in the increasing order of their distance $d[]$ and their new clustering information is updated as follows. A while loop is iterated starting from $j = d[v]$. The jth iteration processes a queue $\mathcal{Q}_j$ of vertices. The following invariant holds before execution of jth iteration for all $j \leq i+1$.

$\mathcal{I}(j)$: The clustering has been updated for every vertex x whose new distance $d[x] \leq j-1$. Queue $\mathcal{Q}_j$ consists of all those vertices y with initial $d[y] \leq j$ and new $d[y] \geq j$ whose clustering changes due to deletion of (u, v).

We now describe the processing done in jth iteration. Vertices are extracted from $\mathcal{Q}_j$, and this is how any vertex x from $\mathcal{Q}_j$ is processed. Firstly, for each $y \in children(x)$, we delete x from $parents(y)$ and delete y from $children(x)$. If $parents(y)$ is empty now, it would imply that clustering of y will also change, and so we add y to $\mathcal{Q}_{j+1}$. If x has some neighbor at level $j-1$, then x will settle at level j and compute its new cluster center (see Observation 5.1). Otherwise, final $d[x]$ has to be greater than j and so x enters queue $\mathcal{Q}_{j+1}$ so as to be processed in the next iteration.

Figure 10 shows an example of execution of **Procedure** `Update-clustering` on a clustering. This clustering is defined by $\sigma(S) = \prec w, u, x \succ$, and has radius 3. It is captured by a BFS tree rooted at a dummy vertex g (see Figure 10(i)). The solid edges constitute the forest spanning the clustering. Upon deletion of edge (u, v), the clustering gets updated as shown in the Figure 10(ii). Vertices v, q, and p move from the cluster centered at u to the cluster centered at x, whereas vertex h leaves the clustering since its distance from S gets more than 3, the radius of the clustering.

Let us analyze the total time spent in maintaining $C(\sigma(S), i)$ over any sequence of edge deletions. Consider an execution of procedure `Update-clustering` for deletion of an edge (u, v). Let the main while loop got iterated from $j = d[v]$ to $j = t$. Then the computation time spent by the procedure is dominated by the processing of the vertices of queues $\mathcal{Q}_j, \ldots, \mathcal{Q}_t$. It follows from invariant $\mathcal{I}(j)$ that only those vertices enter the queue $\mathcal{Q}_j$ during `Update-clustering` whose clustering (cluster membership or $d[]$) has changed due to the deletion of (u, v). A vertex x dequeued

Procedure `Update-clustering(`C,i,(u,v)`)`: updates clustering C of radius i upon deletion of edge (u, v)

```
j ← d[v];  Q_j ← ∅;
Y ← ∅;  Z ← ∅;
if u ∈ parents(v) then
    delete u from parents(v);
    delete v from children(u);
    if parents(v) = ∅ then
        Enqueue(v,Q_j);
        while j ≤ i and Q_j ≠ ∅ do
            Q_{j+1} ← ∅;
            while  Q_j ≠ ∅ do
                x ⟵ Dequeue(Q_j);
                for each y ∈ children(x) do    /* Updating clustering for children(x) */
                    delete y from children(x);
                    delete x from parents(y);
                    if parents(y) = ∅ then Enqueue(y,Q_{j+1});
                if ∃ some neighbor w of x with d[w] = j − 1 then  /* C_i[x] gets updated */
                    c ← C[x];
                    Compute C[x] by scanning all neighbors of x;
                    Y ← Y ∪ {⟨x, c, C[x]⟩};
                    for each neighbor y of x do
                        if d[y] = j − 1 and C[y] = C[x] then
                            add y to parents(x);
                            add x to children(y);
                else                                   /* distance d[x] increases by 1 */
                    d[x] ← d[x] + 1;
                    Enqueue(x,Q_{j+1});
            j ← j + 1;
    while Q_{i+1} ≠ ∅ do       /* Processing vertices which left the clustering C_i */
        x ← Dequeue(Q_{i+1});
        Z ← Z ⋃{⟨x, C[x]⟩};
        C[x] ← 0;
return (Y, Z);
```

from $\mathcal{Q}_j$ involves $O(\deg(x))$ amount of computation which we charge to x itself. After this computation, either its new clustering has been computed or it enters $\mathcal{Q}_{j+1}$. In the latter case, $d[x]$ increases by 1. Thus we conclude that over the entire sequence of edge deletions, a vertex will incur $O(\deg(x))$ amount of computation every time clustering $C(\sigma(S), i)$ changes for x. The clustering will change for x expected $O(i \log n)$ times only as follows from Lemma 5.1. So the expected processing cost charged to a vertex x while maintaining clustering $C(\sigma(S), i)$ over any sequence of edge deletions is $O(i \deg(x) \log n)$. Analysing each vertex along similar lines, we can conclude that the expected computation for maintaining clustering $C(\sigma(S), i)$ under any sequence of edge deletions is $O(im \log n)$.

Theorem 5.1 *Given a graph $G = (V, E)$, an integer $i \geq 0$ and a uniformly random permutation σ of a set $S \subseteq V$, we can maintain the clustering $C(\sigma(S), i)$ and its spanning forest $\mathcal{F}$ in expected amortized $O(i \log n)$ time per edge deletion.*

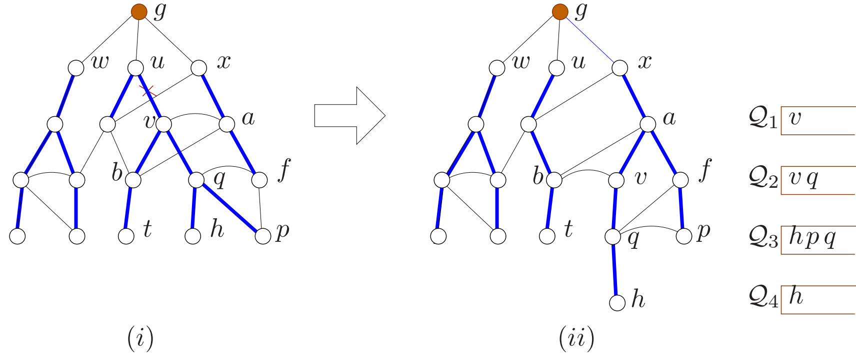

Figure 10: Updating the clustering upon deletion of (u, v). Thick edges define the spanning forest for the clustering

5.3 Decremental Algorithm for $(2k-1)$-spanner

The algorithm uses the hierarchy $\mathcal{H}_k = \{S_i | 0 \leq i \leq k\}$ of subsets of vertices which was used in the static algorithm described in [8] as well as the first fully dynamic algorithm for $(2k-1)$-spanner. Hierarchy $\mathcal{H}_k$ induces a hierarchy of subsets of vertices $V_i, 0 \leq i \leq k$ in a natural way as follows.

- $V_0 = V$ and $V_k = \emptyset$.
- For $0 < i < k$, $V_i = \{v \in V | \delta(v, S_i) \leq i\}$.

In simple words, V_i is the subset of all those vertices in the graph G which lie within distance i from S_i. Notice that, for any $i > 1$, V_{i+1} need not be a subset of V_i though S_{i+1} is a subset of S_i. As its randomization ingredient, our decremental algorithm constructs a uniformly random permutation σ_i of vertices of set S_i for each $i < k$. The algorithm maintains a clustering C_i for V_i and the following two invariants at each level $0 \leq i < k$.

> NEW-CLUSTERING-INVARIANT : Clustering C_i is the clustering $C(\sigma_i, i)$ of the set V_i in G. Let $\mathcal{F}$ be the union of all edges from the spanning forests of these clusterings.
>
> NEW-SPANNER-EDGE-INVARIANT : For every integer $i < k$ and every $v \in V_i$, if $v \notin V_{i+1}$ then v includes one edge from $E(v, c)$ in the spanner for each neighboring cluster $c \in C_i$.

Let E_S be the set of edges contributed by all vertices for maintaining NEW-SPANNER-EDGE-INVARIANT. We now state and prove a crucial lemma.

Lemma 5.2 *For a given graph undergoing edge deletions, the subgraph $(V, E_S \cup \mathcal{F})$ is a $(2k-1)$-spanner at every stage, and its expected size is $O(kn^{1+1/k})$.*

Proof: First we shall show that $E_S \cup F$ is a $(2k-1)$-spanner for the graph at each stage. Note that $V_0 = V$ and $V_k = \emptyset$, so it follows that there is at least one level $j < k$ such that both u and v belong to V_j and at least one of u or v is not present in V_{j+1}. Without loss of generality, let u be absent from C_{j+1}. Let c be the cluster to which v belongs in clustering C_j. Therefore, as part of NEW-SPANNER-EDGE-INVARIANT, vertex u must have contributed an edge from the set $E(u, c)$ to E_S. Now note that the clustering $C_j = C(\sigma_j, j)$ has radius j by construction. So it follows from Observation 2.1 (part 1) that the proposition $\mathcal{P}_{2j+1}$ holds for edge (u, v). Since $j < k$, it follows that the set $E_S \cup \mathcal{F}$ is a $(2k-1)$-spanner.

Now we analyze the size of the spanner. The collection of spanning forests $\mathcal{F}$ would contribute $O(nk)$ edges to the spanner. We shall now show that the expected number of edge contributed to E_S at any level $i < k$ is $O(n^{1+1/k})$. For this purpose, fix the clustering C_i for any arbitrary sample set S_i and its permutation σ_i. Let v be any vertex in V_i. Let $x_1, \ldots, x_\ell$ be the vertices from S_i lying within distance $i+1$ from v. From the construction of hierarchy $\mathcal{H}_k$, it follows that each vertex from $\{x_1, \ldots, x_\ell\}$ is present in S_{i+1} independently with probability $n^{-1/k}$. If at least one of $\{x_1, \ldots, x_\ell\}$ is present in S_{i+1}, v is bound to be present in C_{i+1} and so won't contribute any edge to E_S. Otherwise, for maintaining NEW-SPANNER-EDGE-INVARIANT v will add (to E_S) one edge

```
Procedure Handling-edge-deletion(u,v)
for i = 0 to k − 1 do
    if u ∈ V_i and v ∈ V_i then
        if C_i[u] ≠ C_i[v] then
            Delete (u, v) from D_i(u);
            if C_{i+1}[u] = 0 then Restore-span-edge-invariant(u, C_i[v], i);
            Delete (u, v) from D_i(v);
            if C_{i+1}[v] = 0 then Restore-span-edge-invariant(v, C_i[u], i)
        else
            (Y, Z) ← Update-clustering(C_i, i, (u, v));
            for each ⟨x, c⟩ ∈ Z do          //Processing vertex x which left clustering
                for each (x, w) ∈ E(x) with w ∈ V_i do
                    Delete (x, w) from D_i(w);
                    if C_{i+1}[w] = 0 then Restore-span-edge-invariant(w, c, i);
                C_i[x] ← 0 ;                        //x leaves clustering at level i
                if C_{i−1}[x] ≠ 0 then
                    for each c ∈ C_{i−1} neighboring to x do
                        Restore-span-edge-invariant(x, c, i − 1)
            for each ⟨x, c, c′⟩ ∈ Y do      //Processing vertex x which changed cluster
                for each (x, w) ∈ E(x) and w ∈ V_i do
                    D_i(w) processes the change in cluster of x;
                    if C_{i+1}[w] = 0 then
                        Restore-span-edge-invariant(w, c, i);
                        Restore-span-edge-invariant(w, c′, i);
```

from $E(v,c)$ for each neighboring cluster $c \in C_i$. Now observe that the center of each cluster $c \in C_i$ neighboring to v must belong to the set $\{x_1, ..., x_\ell\}$. Hence there are ℓ clusters neighboring to v at level i, and the probability that none of $\{x_1, \ldots, x_\ell\}$ is selected in S_{i+1} is $(1 - n^{-1/k})^\ell$. This implies that the expected number of edges contributed to E_S at level i by vertex v is bounded by $\ell(1 - n^{-1/k})^\ell$, which is $O(n^{1/k})$ for all possible values of ℓ. Hence the expected number of edges contributed to E_S at level i by all vertices is $O(n^{1+1/k})$. This completes the proof. •

Lemma 5.2 suggests that for maintaining a $(2k-1)$-spanner under deletion of edges, it suffices to maintain the hierarchy of subgraphs and clusterings, and the two invariants as described above.

5.3.1 Handling deletion of an edge

We describe Procedure `Handling-edge-deletion` for handling deletion of any edge (u, v). It proceeds in a bottom up fashion starting from level 0 to $k-1$. Unlike fully dynamic algorithm I, processing at any level i is done quite independent of other levels. Observe that if either u or v does not belong to V_i, then the clustering C_i as well as the invariants remain unaffected, so nothing needs to be done at level i in this case. In case $u, v \in V_i$, the deletion of (u, v) is processed at level i as follows.

If $C_i[u] \neq C_i[v]$, the deletion of (u, v) does not change clustering C_i. However, it may violate NEW-SPANNER-EDGE-INVARIANT if either of u or v is absent at level $i+1$. Using data structure $D_i(u)$ (and $D_i(v)$), it would require $O(1)$ time to restore NEW-SPANNER-EDGE-INVARIANT for u (for v).

If $C_i[u] = C_i[v]$, the deletion of (u, v) may change the clustering C_i. So we execute the procedure `Update-clustering`$(C_i, i, (u, v))$. This procedure outputs the changes in clustering C_i through two subsets Y, Z (see Figure 10). A triplet $\langle y, c, c' \rangle \in Y$ represents that y changed its cluster from c to c'. So we need to communicate this information to neighbors of y in V_i and restore their NEW-

SPANNER-EDGE-INVARIANT if needed. A pair $(z, c) \in Z$ represents that z ceases to belong to C_i. We need to communicate this information to neighbors of z in V_i and restore their NEW-SPANNER-EDGE-INVARIANT. In addition, we need to restore NEW-SPANNER-EDGE-INVARIANT for z at level $i-1$ if it is present in C_{i-1}.

In order to analyse the time complexity of the decremental algorithm described above, we analyse the total computation time spent by the algorithm at a level $i < k$ over any sequence of edge deletions in the graph. Processing an edge deletion at level i takes $O(1)$ time except when it leads to a change in clustering C_i. In this case, the procedure `Update-clustering` gets invoked. It follows from Theorem 5.1 that the expected amortized computation performed by this procedure will be $O(i \log n)$ per edge deletion. This procedure returns the set of vertices which either changed their cluster in C_i or cease to belong to it. For each such vertex x, a computation of the order of $|E(x)|$ is incurred in communicating this change to its neighbors at level i and restoring their NEW-SPANNER-EDGE-INVARIANT if needed. For analysis purpose we shall assign this cost to vertex x itself. It follows from Lemma 5.1 that a vertex can change its cluster in C_i expected $O(i \log n)$ number of times before leaving it. Hence the expected amount of the total computation time spent in the processing done at level i for any sequence of edge deletions is $O(mi \log n)$. Since there are total k levels, expected amount of total computation performed in maintaining $(2k-1)$ spanner by the decremental algorithm is $O(mk^2 \log n)$. So we can conclude the following Theorem.

Theorem 5.2 *An undirected unweighted graph $G = (V, E)$ can be processed to build a data structure $\mathcal{B}(V, E)$ which maintains a $(2k-1)$-spanner of expected size $O(kn^{1+1/k})$ under deletion of edges. For any arbitrary sequence of edge deletions, the expected total time spent in maintaining this data structure is $O(k^2 m \log n)$.*

Remark 5.3: Randomization is used at two places in the decremental algorithm described above. The randomization underlying the hierarchy $\mathcal{H}_k$ is used to get a bound on the expected size of the spanner. The randomization used in the clustering $C(\sigma_i, i)$ helps in achieving better update time.

5.4 Extending the decremental algorithm to the fully dynamic environment

We shall now employ the decremental algorithm from Theorem 5.2 to design a fully dynamic algorithm for $(2k-1)$-spanner. The algorithm is based upon the following simple observation.

Observation 5.2 *For a given graph $G = (V, E)$, let $E_1, ..., E_j$ be a partition of the set of edges E, and let $\mathcal{E}_1, ..., \mathcal{E}_j$ be respectively the t-spanners of subgraphs $G_1 = (V, E_1), ..., G_j = (V, E_j)$. Then $\cup_i \mathcal{E}_i$ is a t-spanner of the original graph $G = (V, E)$.*

Based on the above observation, the fully dynamic algorithm can be summarized as follows : The algorithm maintains a partition of the graph into $O(\log n)$ subgraphs, and concurrently maintains a decremental data structure as defined in Theorem 5.2 for each subgraph. In order to handle edge insertions, each of these subgraphs is redefined periodically after certain intervals of updates and the corresponding data structure is rebuilt accordingly. Now we provide complete details of the algorithm. Let ℓ_0 be the greatest integer such that $2^{\ell_0} \leq n^{1+1/k}$. The algorithm will maintain the following objects and data structures at each stage.

1. A partition of edges E into subsets $E_0, ..., E_j$, $j = \lceil \log_2 n^{1-1/k} \rceil$ such that $|E_i| \leq 2^{i+\ell_0}$ holds for each $i \leq j$ throughout the sequence of edge updates. Some of these subsets may be empty. For each edge (u, v), we maintain an additional field, $index(u, v)$ which stores the integer i such that $(u, v) \in E_i$.

2. For each subset $E_i, i > 0$, the data structure $\mathcal{B}_i = \mathcal{B}(V, E_i)$ from Theorem 5.2 which maintains a $(2k-1)$-spanner $(V, \mathcal{E}_i)$ for the subgraph (V, E_i).

3. A binary counter $\mathbf{C}$ which counts from 0 to $\frac{n(n-1)}{2}$, with the least significant bit at 0th place.

In the beginning, the partition is : $E_j = E$ and $E_i = \emptyset$ for all $i < j$; and the counter $\mathbf{C}$ is set to 0. The fully dynamic algorithm handles deletion and insertion of edges using procedures

Procedure `Handling deletion of an edge`(u,v)

$i \leftarrow index(u,v)$;
update $\mathcal{B}_i$ for deletion of (u,v);
update the spanner $\mathcal{E}_i$ accordingly.

Procedure `Handling insertion of an edge`(u,v)

Increment the counter **C**;
Let g be the highest bit of counter **C** which gets flipped;
if $g \leq \ell_0$ **then** $//\ell_0 = \lfloor \log_2 n^{1+1/k} \rfloor$
  $E_0 \leftarrow E_0 \cup \{(u,v)\}$;
  $\mathcal{E}_0 \leftarrow \mathcal{E}_0 \cup \{(u,v)\}$;
else
  $h \leftarrow g - \ell_0$; $E_h \leftarrow E_h \cup \{(u,v)\}$;
  $E_h \leftarrow \cup_{0 \leq i \leq h} E_i$;
  for $0 \leq i < h$ **do**
    $E_i \longleftarrow \emptyset$;
    $\mathcal{E}_i \leftarrow \emptyset$
  rebuild $\mathcal{B}_h$.

Figure 11: Handling insertion and deletion of edges

shown in Figure 11. Consider deletion of an edge. We first compute its index, say i. Then the data structure $\mathcal{B}_i$ processes the deletion of the edge and updates the spanner $\mathcal{E}_i$ accordingly. Let us consider insertion of an edge. First, we increment counter C, and let g be the highest bit of C which gets flipped. If $g \leq \ell_0$, we just insert the edge to E_0 and $\mathcal{E}_0$. Otherwise, we insert the edge to E_h, where $h = g - \ell_0$, and move all the edges from $E_i, i < h$ to E_h. At this moment, $E_i = \emptyset$ for all $i < h$. The data structure $\mathcal{B}_h$ and the spanner $\mathcal{E}_h$ associated with (V, E_h) are then rebuilt.

It follows easily from the fully dynamic algorithm described above that $\{E_i | i \leq j\}$ is a partition of E at each stage. For each $i \leq j$, the data structure $\mathcal{B}_i$ maintains a $(2k-1)$-spanner $\mathcal{E}_i$ for subgraph (V, E_i). So using Observation 5.2, it follows that at each stage $\cup_i \mathcal{E}_i$ is a $(2k-1)$-spanner of E. To analyze the update time of the new fully dynamic algorithm, we first state an important lemma.

Lemma 5.4 *For each* $0 \leq i \leq j$, $|E_i| \leq 2^{i+\ell_0}$ *holds at each stage.*

Proof: During the algorithm, maximum number of edges in the graph is at most $\frac{n(n-1)}{2}$ and so $|E_j| \leq 2^{j+\ell_o}$. Let us analyze E_i for $i < j$. Observe that tth bit of the binary counter is reset after every 2^t increment operations. Each increment operation in the counter is triggered by an edge insertion operation. Therefore, it follows from the algorithm that each set E_i is set to empty set after every $2^{i+\ell_0}$ edge insertions. Hence $|E_i| \leq 2^{i+\ell_0}$ holds at each stage. •

In order to bound the update time of the fully dynamic algorithm, it suffices to bound the update time incurred in maintaining $\mathcal{B}_i$ for each i. Consider any arbitrary sequence of μ edge updates. The data structure $\mathcal{B}_i$ is rebuilt after every $2^{i+\ell_0}$ insertions, and hence will be rebuilt at most $\frac{\mu}{2^{i+\ell_0}}$ times. It follows from Theorem 5.2 that the expected total number of operations for building the data structure and maintaining it under arbitrary sequence of edge deletions (till next rebuilding) is bounded by $O(|E_i| k \log^2 n)$. Applying Lemma 5.4, $O(|E_i| k \log^2 n)$ is bounded by $O(2^{i+\ell_0} k \log^2 n)$. Hence for any sequence of μ edge updates, the expected number of operations performed for rebuilding and maintaining data structure $\mathcal{B}_i$ is $O(\frac{\mu}{2^{i+\ell_0}} 2^{i+\ell_0} k^2 \log n) = O(\mu k^2 \log n)$. Since there are $O(\log n)$ instances of the data structure $\mathcal{B}$ used in the algorithm, it follows that the expected update time for handling μ edge updates (insertion/deletion) is $O(\mu k^2 \log^2 n)$. In other words, expected amortized update time per edge insertion/deletion achieved by the algorithm is $O(k^2 \log^2 n)$.

Theorem 5.3 *There exists a fully dynamic algorithm which can maintain a* $(2k-1)$*-spanner of an undirected unweighted graph on* n *vertices such that the expected amortized time per edge insertion/deletion is* $O(k^2 \log^2 n)$ *and the expected size of the spanner at each stage is* $O(kn^{1+1/k} \log n)$.

6 Fully Dynamic Distributed Algorithm for $(2k-1)$-Spanner

As a warm up, first we show that our fully dynamic centralized algorithm I (Theorem 4.4) adapts seamlessly in a synchronous distributed environment. The quiescence time complexity achieved is k and expected amortized communication complexity per update is $O(7^k)$. However, this algorithm would achieve $O(k^2m)$ quiescence message complexity. We then extend this algorithm suitably to reduce quiescence message complexity to $O(km)$ as well. In this algorithm, a message communicated, if any, along an edge during a round consists of $O(\log n)$ bits.

Note 6.1 *It is expected that the reader fully understands the fully dynamic centralized algorithm I before studying its implementation in the distributed environment.*

6.1 Key properties of the fully dynamic centralized algorithm I

We now summarize various key properties of our fully dynamic algorithm I which will play an important role in its extension to distributed environment.

- *Atomic and local nature of updates.* In the centralized algorithm, let v be a vertex present at level i. An update concerning v there is either deletion/insertion of an edge incident on v or a change in clustering of v at level i. Moreover, the processing of any such update by v requires only local information - the clustering information of itself and its neighbors at level i.
- *Acyclic nature of updates.* The processing of an update by a vertex at a level i does not generate updates for itself or its neighbors at level i. Instead, this processing may generate update only for level $i+1$.
- *Superfluousness of the transient updates.* Processing of a sequence of updates at level i by a vertex during a round may generate multiple updates for a single neighbor at level $i+1$. As discussed earlier, only the final update needs to be communicated to the neighbor. The remaining updates, which are just transient, can be ignored.

Recall the hierarchy $\mathcal{H}_k = \{S_0 = V, S_1, \ldots, S_k = \emptyset\}$ of subsets defined by our first centralized algorithm. $\mathcal{H}_k$ and the hierarchy of clusterings form the backbone of our centralized algorithm. We first show that these can be built and maintained in the distributed environment as well.

Hierarchy of sampled vertices and clusterings in distributed environment. The hierarchy $\mathcal{H}_k$ is constructed by performing random sampling at each level from 0 to $k-1$ wherein each vertex from set S_i is sampled for the set S_{i+1} independently with probability $n^{-1/k}$. The independence employed in the random sampling enables construction of $\mathcal{H}_k$ in the distributed environment as well.

For each vertex $u \in V$, let $\textsc{max}(u)$ be the highest non-negative integer j such that $u \in S_j$. For maintaining clustering in distributed environment, we require some mechanism using which a vertex can know if its own or its neighbor's cluster is sampled at level i. For this purpose, each vertex u maintains its clustering information at a level i using an ordered pair (v, j) where v is the center of the cluster to which u belongs and $j = \max(v)$. It can be observed that the cluster centered at v (to which the vertex u belongs at level i) is a sampled cluster at level i if $j > i$.

6.2 Adapting the fully dynamic centralized algorithm I in distributed environment

The fully dynamic algorithm I processes each update in the graph in a bottom-up fashion. The entire algorithm can be seen as execution of k iterations - ith iteration begins with a set of updates at ith level of the hierarchy. These updates are processed so as to restore the invariants at level i of the hierarchy. Notice that the updates concerning a vertex at level i are generated either by its neighbors or itself at level $i-1$. As a result, each such update can be communicated directly to the vertex by the respective neighbor in the distributed environment. The atomic and local nature of updates implies the following. Firstly, length of each message corresponding to an update will be $O(\log n)$ bits. Secondly, once every vertex receives its updates at level i, each vertex can process these updates independent of other vertices present at level i. Therefore, the entire computation

of ith iteration of the centralized algorithm can be performed concurrently in a single round of the synchronous distributed environment.

Consider the entire time scale partitioned into rounds. Let some updates take place in the graph during round τ. It follows from acyclic nature of updates that by the end of $(\tau + i)$th round, the invariants have been restored at all levels upto $i - 1$ and the *wave* of updates which originated at level 0 during round τ has propagated to level i. Let there be some additional updates in the graph in some subsequent round τ'. Notice that the two waves of updates will always have a time gap of $\tau' - \tau$ rounds in their reaching any level i of the hierarchy (see Figure 12(i)). During any round, processing done at level i will be for those updates only which occurred i round ago. Therefore, the updates that occur in different rounds can also be processed concurrently without any interference. This concurrent processing at all levels looks very much like a pipe-line of length k as shown in Figure 12(ii). Based on these insights, we can summarize the adaptation of the centralized algorithm in the distributed environment in just one sentence- *In each round updates are communicated through respective edges; and each vertex plays its role at all levels concurrently and independently.* In particular, v does the following activity during a round.

- In the beginning of each round, v concurrently sends (and receives) updates to (and from) its neighbors for each level $i < k$.
- v concurrently does the following computation for each level $i < k$:
  v processes updates for level i sequentially. This results in generation of updates for level $i + 1$ to be communicated in the next round. However, if there are multiple updates generated for a neighbor at a level $i + 1$, only the final update will be communicated to that neighbor, and the transient ones will be discarded. (This is justified due to the superfluousness of the transient updates as mentioned above)

The following lemma about the distributed algorithm follows from the discussion above.

Lemma 6.1 *Let the last update occurred in round q. At the end of round $(q + i)$, there will be no update message to be processed at any level i or below.*

It is a simple corollary of Lemma 6.1 that the quiescence time complexity of the distributed algorithm is k. Notice that a message communicated during any round is always associated with some update generated at some level in the previous round. The distributed algorithm described above is an exact adaptation of our first centralized algorithm (Theorem 4.4). Hence expected amortized $O(7^k)$ number of messages will be communicated per edge insertion/deletion in the graph. An important consequence of the superfluousness of the transient updates which is exploited in the algorithm is the following lemma.

Lemma 6.2 *Let v be a vertex at level i and x be some neighbor of v which is present at level $i + 1$. After processing any sequence of updates at level i during a round, v will have at most one message (update) of $O(\log n)$ bits for x at level $i + 1$.*

It follows from Lemma 6.2 that the number of messages, each of length $O(\log n)$, communicated along an edge in a round will be $O(k)$. Thus the quiescence message complexity of the algorithm is $O(k^2 m)$. This leads us to the following theorem.

Theorem 6.1 *A $(2k - 1)$-spanner of expected $O(k^9 n^{1+1/k} \log^2 n)$ size can be maintained in synchronous distributed environment with quiescence time k and quiescence message complexity $O(k^2 m)$. The message complexity per update in the graph will be expected amortized $O(7^k)$. The worst case length of all messages communicated along an edge in a round will be $O(k \log n)$ bits.*

With the above warm-up, we now design our fully dynamic distributed algorithm for maintaining $(2k - 1)$-spanner with $O(km)$ quiescence message complexity.

6.3 Fully dynamic algorithm in synchronous distributed environment

The reason for $O(k^2 m)$ quiescence message complexity is that, during a round, a vertex plays its role at each level i concurrently and thus can generate a total of $\Theta(k)$ updates to be communicated

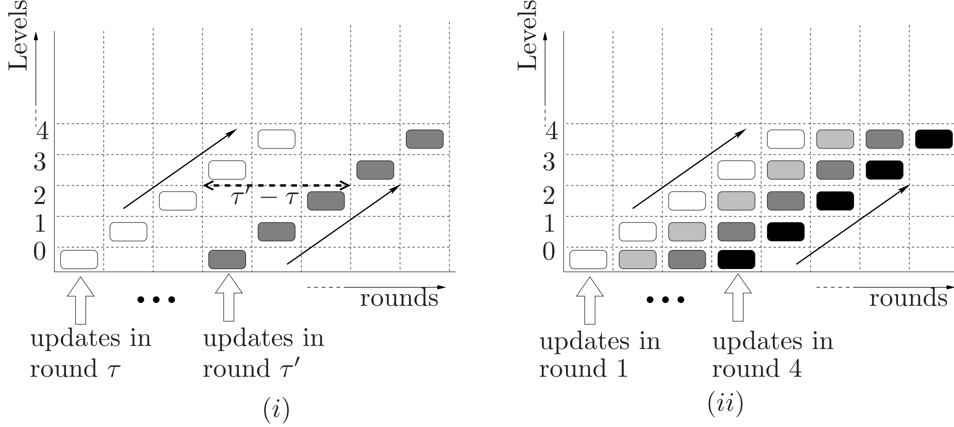

Figure 12: (i)two waves of updates originating during round τ and τ', (ii)processing the updates in a pipe-line manner in the synchronous distributed model

along an edge. To reduce this bound, in our distributed algorithm, a vertex will play its role at only one level during a round. The algorithm can be summarized for each vertex $v \in V$ as follows.

After exchanging messages with its neighbors in the beginning of a round, let i be the lowest level for which there exists an unprocessed update for v. v processes all updates of level i in a sequential manner. If v received, during the same round, an update for level $j > i$ communicated by a neighbor, say u, then it is stored in a buffer. This update will either be processed in some future round when there is no update for v for any level below j. However, this update may get overwritten in case u communicates another update for v at level j.

We now provide complete details of the algorithm. First we provide description of the types of messages exchanged during the algorithm.

6.3.1 Types of messages

In the distributed environment, we just need the following two types of messages, each of size $O(\log n)$, to communicate any update along an edge.

- *ClusteringInfo* message. This message informs the receiver about the cluster to which sender belongs at a level i. For example, consider an edge (u, v). The *ClusteringInfo*$(u, v, x, i, \text{MAX}(x))$ message from u to v means that vertex u belongs to cluster centered at x at level i. Recall that $\text{MAX}(x)$ is the highest level upto which the cluster centered at x remains sampled.
- *EdgeDeletion* message. An *EdgeDeletion* message at level i communicates to the receiver that the sender has deleted the edge from level i and above in the hierarchy while maintaining either of the invariants at level $i - 1$.

Consider a vertex v at level i and consider any neighbor of v, say u, at level $i-1$. Let us explore the situations during the algorithm in which u will have to communicate the messages mentioned above to v along the edge (u, v). Recall that u maintains the set $\mathcal{E}_{i-1}(u)$ of active edges at level $i-1$. While processing updates at level $i - 1$, edge (u, v) may enter and leave the set $\mathcal{E}_{i-1}(u)$. Since the edges at level i are defined in terms of the active edges maintained at level $i-1$, therefore, whenever (u, v) enters $\mathcal{E}_{i-1}(u)$, it can potentially lead to insertion of (u, v) at level i. Vertex u communicates this update by *ClusteringInfo* message. Vertex u also uses this message whenever u changes its cluster at level i. Similarly, whenever edge (u, v) leaves $\mathcal{E}_{i-1}(u)$, it leads to deletion of (u, v) at level i. Vertex u communicates this update by *EdgeDeletion* message to v.

Note that updates for a vertex v at level i could also be generated by vertex v itself at level $i - 1$. These update could be insertion/deletion of an edge or a change in clustering of v at level i. These updates are basically like other updates which are communicated by neighbors of v though they need not be communicated through any edge. Hence these updates will be stored by v locally.

6.3.2 Data structure kept by a vertex

In addition to the local data structures, namely $D_i(v)$ and bucket structure $\mathcal{B}_i(v)$, each vertex v will keep the following data structures in the distributed setting.

- an array L^v such that $L^v[i]$ would store the list of updates for v at level i which got generated while v restores its invariants at level $i-1$.
- For each edge (u,v), an array Q_u^v such that $Q_u^v[i]$ stores the most recent message received by v along the edge $(u,v) \in E_{i-1}$. It also stores a bit with $Q_u^v[i]$ which is set whenever a new message is received along the edge (u,v) for level i and is reset as soon as the corresponding update gets processed by v.
- An array *Output* to store the update messages generated for the neighboring vertices during processing of any round.

Procedure SynRound gives the complete description of the protocol which a vertex v follows during the dynamic distributed algorithm.

Procedure `SynRound(`v`)`

```
/*Recording the insertion/deletion of edges incident on v                    */
foreach edge (u,v) added to the network in previous round do
    add ClusteringInfo(u,v,u,0,MAX(u)) message to Output;
foreach edge (x,v) deleted from network in previous round do
    add EdgeDeletion message of level-0 in the list L^v[0];
/*Communicating the updates                                                  */
foreach message ∈ Output do
    send the message to the concerned neighbor;
foreach message received from neighbor, say u, for level i do
    write the message in Q_u^v[i] and set the corresponding bit;
/*Processing the updates                                                     */
Let i be the lowest level for which v has some unprocessed update;
foreach unprocessed update at level i do
    process the update and restore the invariants for level i
Among the new updates generated for level i+1 eliminate the transient ones;
The new updates for neighbors at level i+1 get stored in Output;
The new updates for v at level i+1 get stored in L^v[i+1] ;
```

A few technical points about processing the updates

1. In our first fully dynamic centralized algorithm, consider any two vertices u, v belonging to different unsampled clusters at level i. Note that u as well as v maintain subsets $\mathcal{E}_{i-1}(u)$, $\mathcal{E}_{i-1}(v)$ of active edges at level $i-1$ using their bucket structures. A necessary condition for the edge (u,v) to be present at level i is that it is present in $\mathcal{E}_{i-1}(u)$ as well as $\mathcal{E}_{i-1}(v)$. In case the edge (u,v) gets removed from either of the sets $\mathcal{E}_{i-1}(u)$ and $\mathcal{E}_{i-1}(v)$, the edge should remain absent from E_i. So to maintain consistency between $E_i(u)$ and $E_i(v)$, u (as well as v) must know the activation status of the edge (u,v) in the bucket structure of v (of u) at level $i-1$. In the distributed environment, the latest message stored in $Q_u^v[i]$ can be used to infer this information by v (likewise by u) as follows. If $Q_u^v[i]$ stores *ClusteringInfo* message, it implies that $(u,v) \in \mathcal{E}_{i-1}(u)$; whereas if $Q_u^v[i]$ stores *EdgeDeletion* message then it would imply that $(u,v) \notin \mathcal{E}_{i-1}(u)$.

2. At each level i, each vertex v adapts the following overwriting policy in the buffers $L_v[i]$ and $Q_u^v[i]$ for the updates associated with any edge $(u,v) \in E_{i-1}$. Let some update be generated for edge (u,v) at level i. This update could be generated by v itself at level $i-1$ (and hence

to be stored in $L_v[i]$) or it could be generated by u at level $i-1$ (and hence to be stored in $Q_u^v[i]$). In case there is already some update in the corresponding buffer, then the new update overwrites the old update.

3. $Q_u^v[i]$ has to store the most recent update received along an edge $(u,v) \in E_{i-1}$ even after it is processed. This is to infer the activation status of (u,v) in the bucket structure of u at level $i-1$ (see point 1 above). However, an update stored in $L_v[i]$ may be discarded as soon as it is processed.

6.3.3 Analysis of the algorithm

Lemma 6.1 holds for the new distributed algorithm as well, so its quiescence time complexity is also k. Since along each edge in a round, at most one message of $O(\log n)$ bits is communicated in either direction, the quiescence message complexity is $O(km)$. Using Lemma 4.6, it would follow that the expected amortized number of messages communicated per update will be $O(7^k)$.

Though the new distributed algorithm is efficient and has a compact description, it is not immediate if the spanner maintained is a $(2k-1)$-spanner of expected size $O(k^9 n^{1+1/k} \log^2 n)$. The reason for this is as follows. Consider the set of all the updates communicated for any level i during any round by the algorithm. While some of these updates may be processed in the next round, the remaining ones may either be processed in some future round or may even be overwritten as well. However, as we shall show, the algorithm is robust against such apparent irregularities. The algorithm assumes the validity of only the following principle which holds for all synchronous distributed environment.

Well ordering principle along an edge
Consider a vertex v. Along an edge (u,v), v receives messages in the order they are communicated by u, and it processes these update messages in the order they are received. However, there is a possibility of an update message to be overwritten by an update message which follows it.

Let the last network update be observed during round q. Notice that each vertex present at a level i maintains the invariants depending upon its neighborhood at level i. Recall from the static as well as the fully dynamic centralized algorithm I that these invariants ensure that the spanner maintained at any moment is a $(2k-1)$-spanner and has expected size $O(k^9 n^{1+1/k} \log^2 n)$. The structures (subgraph and clustering) at level $(i+1)$ are defined solely by the invariants maintained over the structures at level i. Thus, all that is needed to establish the correctness of our fully dynamic distributed algorithm is the following assertion. At the end of $(q+i+1)$th round, the structures at level $(i+1)$ are in *accordance with* the final updates communicated from level i. Using *well ordering principle along an edge*, we can prove the validity of this assertion as follows.

Consider the snapshot of neighborhood of a vertex v at level $i+1$ in the beginning of the algorithm. Compared to it, the neighborhood of v might have changed by the end of round $(q+i)$. There might have been many updates for v at level $(i+1)$ during this time span. Some of these updates would have been processed by v, while some updates would have been overwritten. What can we say about the final update for level $i+1$ communicated by u to v along edge (u,v)? Note that the final update message along (u,v) communicated by u represents the final *status* of u at level $i+1$ as follows. If the final message is *ClusteringInfo* message, then it would convey to v the cluster to which u belongs at level $i+1$. If the final message is *EdgeDeletion* message, then it would mean that (u,v) is not present at level $i+1$. Now it follows from the well ordering principle mentioned above that the final message communicated by u will also be the final message received and processed by v through edge (u,v). Hence, once all the update messages for level $i+1$ received from level i have been processed, the clustering of v and its neighborhood at level $(i+1)$ is in complete accordance with the final updates communicated from level i. Therefore, the spanner maintained at the end of $q+k$ rounds will be a $(2k-1)$-spanner of expected size $O(k^9 n^{1+1/k} \log^2 n)$.

Theorem 6.2 *A $(2k-1)$-spanner of expected $O(k^9 n^{1+1/k} \log^2 n)$ size can be maintained in synchronous distributed environment with quiescence time k rounds and quiescence message complexity $O(km)$. In addition, the expected amortized message complexity achieved per update is $O(7^k)$, with worst case $O(\log n)$ bit message communicated along an edge in any round.*

Remark 6.3: Our fully dynamic centralized algorithm I (and distributed algorithm) for $(2k-1)$-spanner achieves time complexity (and message complexity) per update which is exponential in k. This makes the algorithm suitable for small stretch spanner only. However, the following point is worth consideration: If an application has to maintain a spanner of size close to $O(n \operatorname{polylog} n)$ instead of $O(n)$, stretch value can be made smaller than $\log n$ by a suitable constant factor. In this case, the time (message complexity) per update achieved by the algorithm can be made arbitrarily close to $O(n^\epsilon)$, for any constant $\epsilon > 0$. In particular, a $O(2^{\frac{3}{2\epsilon}} n \operatorname{polylog} n)$ size spanner of stretch $< \frac{4\epsilon}{3} \log n$ can be maintained with $O(n^\epsilon)$ expected amortized time per update for any value of $\epsilon > 0$.

7 Improved fully dynamic algorithm for approximate shortest paths

The problem of maintaining all-pairs shortest paths (APSP) in graphs is defined as follows. There is an on-line sequence of edge insertions/deletions interspersed with query about shortest path or distance between any pair vertices in the graph. The objective is to maintain a data structure which can answer any shortest path query efficiently and handle any update efficiently too. This problem has gained a lot of attention in the past fifteen years. The best known amortized update time achieved for the fully dynamic APSP with $O(1)$ query time is $O(n^2 \operatorname{polylog} n)$. This bound is achieved by an algorithm designed by Demetrescu and Italiano [18], followed by a slight improvement by Thorup [46]. The algorithm works for directed weighted graphs. The update time is almost the best one can achieve if one has to explicitly maintain an all-pairs distance matrix. However, for undirected unweighted graphs, there exist algorithms which can report approximate distance and achieve subquadratic update time if the graph is sparse. Two most efficient fully dynamic algorithms for the problem of all-pairs approximate shortest paths (APASP) are as follows.

- Roditty and Zwick [43] designed the first fully dynamic algorithm for all-pairs approximate distances. The algorithm achieves better update time at the expense of larger query time. For any $t < \sqrt{m}$ and $\epsilon > 0$, the algorithm achieves amortized $O(\frac{mn}{\epsilon t})$ time per update and takes $O(t)$ time to report $O(1+\epsilon)$-approximate distance between any pair of vertices.
- Bernstein [10] designed an algorithm for maintaining all-pairs approximate distances with the fastest update time. This algorithm works for weighted graphs as well. The algorithm takes $O(\log\log\log n)$ time to report $(2+\epsilon)$-approximate distance between any pair of vertices and achieves $O(mn^{\sqrt{\log \frac{3}{\epsilon} / \log n}})$ expected amortized time per update for any $\epsilon > 0$.

Elkin [25] showed that a dynamic algorithm for $(2k-1)$-spanners can prove to be useful for improving the update time of algorithm of Roditty and Zwick [43] even further, but at the expense of increased stretch. The overall strategy devised by Elkin [25] is quite generic, and can be described as follows.

Maintain a spanner of the graph dynamically and use the dynamic algorithm for APASP on this spanner instead of the original graph. The updates in the graph are communicated to the dynamic algorithm for spanner, which in turn generates updates in the spanner. The updates in the spanner are communicated to the data structure for maintaining APASP which is used for answering approximate distance query between any pair of vertices. See Figure 13 for details.

Let a fully dynamic algorithm for $(2k-1)$-spanner achieves $O(\mu_{n,k})$ update time and generates $\nu_{n,k}$ updates in the spanner upon a single update in the graph. It can be observed that combining this algorithm with algorithm of Bernstein [10] as described above would achieve expected amortized time $O(\nu_{n,k} \cdot n^{1+1/k+\sqrt{\log \frac{3}{\epsilon} / \log n}} + \mu_{n,k})$ per update. The distance reported between any two vertices will have stretch $(2k-1)(2+\epsilon)$. In order to achieve subquadratic update time, both $\nu_{n,k}$ and $\mu_{n,k}$ have to be *suitably* small. Interestingly, our fully dynamic algorithm II meets both these requirements.

It follows from Theorem 5.3 that our fully dynamic algorithm II for $(2k-1)$-spanner achieves $\mu_{n,k} = O(k^2 \log^2 n)$. Observe that $\nu_{n,k}$ is upper bounded by $\mu_{n,k}$. Now consider the fully dynamic algorithm designed by Bernstein [10] for all-pairs $(2+\epsilon)$-approximate shortest paths and set $\epsilon = 1/k$. This algorithm combined with our fully dynamic algorithm II in the way described above leads to the following theorem.

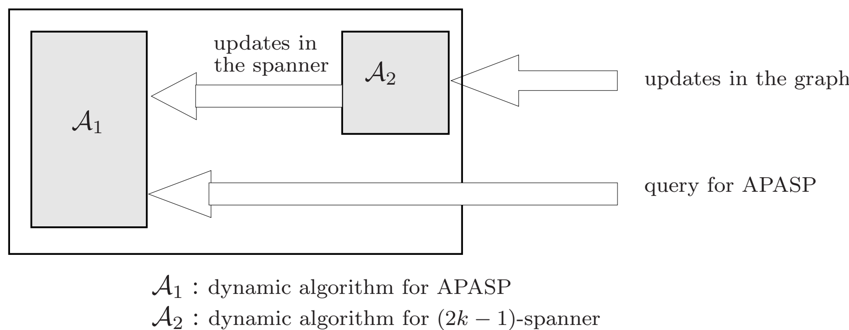

Figure 13: Combining dynamic algorithms for spanner and APASP for better update time.

Theorem 7.1 *Given any undirected unweighted graph on n vertices, and any integer $k > 1$, all-pairs $4k$-approximate distances can be maintained in the graph fully dynamically with $O(\log\log\log n)$ query time and $O(n^{1+1/k+\sqrt{\log 3k/\log n}}\,\text{polylog}\, n)$ expected amortized time per update.*

Remark 7.1: Algorithm stated in Theorem 7.1 is the first fully dynamic algorithm for APASP which guarantees subquadratic update time and *nearly* constant query time for any unweighted undirected graph.

8 Conclusion and open problems

We presented fully dynamic algorithms for maintaining spanners of unweighted graphs in centralized as well as distributed environments. These algorithms achieve *near* optimal performance for maintaining a $(2k-1)$-spanner. We would like to conclude with the following open problems.

1. Our fully dynamic centralized algorithm I (and distributed algorithm) achieves a huge poly-logarithmic blow up in the size of the spanner that is maintained. It would be interesting to get rid of this poly-logarithmic factor.

2. Simple randomization principles played key roles in our algorithms. It would be interesting to explore whether similar results are achievable deterministically too.

3. Our fully dynamic distributed algorithm is for synchronous environment only. It would be interesting to extend it for asynchronous environment as well.

4. In a very impressive result, Elkin et al. [26] showed that every weighted connected graph on n vertices contains as a subgraph a spanning tree which achieves average stretch $O(\log^2 n \log\log n)$. They also designed a very efficient static algorithm for constructing such a spanning tree. It would be an interesting and challenging problem to design dynamic algorithms for maintaining such spanning trees.

Acknowledgments

The authors are very grateful to anonymous referee for providing very valuable comments on a preliminary version of this paper which helped in improving its readability and rectifying many inaccuracies as well. The authors are also grateful to Michael Elkin for providing a copy of his paper [22] which provided motivation to extend the fully dynamic centralized algorithm for spanners to distributed environment.

References

[1] I. Abraham, Y. Bartal, H. T.-H. Chan, K. Dhamdhere, A. Gupta, J. M. Kleinberg, O. Neiman, and A. Slivkins. Metric embeddings with relaxed guarantees. In *Proceedings of 46th Annual (IEEE) Symposium on Foundations of Computer Science (FOCS)*, pages 83–100, 2005.

[2] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9:81–100, 1993.

[3] G. Ausiello, P. G. Franciosa, and G. F. Italiano. Small stretch spanners on dynamic graphs. In *Proceedings of 13th Annual European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 532–543. Springer, 2005.

[4] S. Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Inf. Process. Lett.*, 106(3):110–114, 2008.

[5] S. Baswana and T. Kavitha. Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *Proceedings of the 47th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 591–602, 2006.

[6] S. Baswana, T. Kavitha, K. Mehlhorn, and S. Pettie. Additive spanners and (alpha, beta)-spanners. *ACM Transactions on Algorithms*, 7(1):5, 2010.

[7] S. Baswana and S. Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(kn^{1+1/k})$ size in weighted graphs. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 384–396, 2003.

[8] S. Baswana and S. Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures and Algorithms*, 30:532–563, 2007.

[9] S. Baswana, K. Telikepalli, K. Mehlhorn, and S. Pettie. New construction of (α, β)-spanners and purely additive spanners. In *Proceedings of 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 672–681, 2005.

[10] A. 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)*, pages 693–702, 2009.

[11] B. Bollobás, D. Coppersmith, and M. Elkin. Sparse distance preserves and additive spanners. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 414–423, 2003.

[12] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:4:493–507, 1952.

[13] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.

[14] D. Coppersmith and M. Elkin. Sparse source-wise and pairwise distance preservers. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 660–669, 2005.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. In *Introduction to Algorithms*. The MIT Press, 1990.

[16] L. J. Cowen. Compact routing with minimum stretch. *Journal of Algorithms*, 38:170–183, 2001.

[17] L. J. Cowen and C. G. Wagner. Compact roundtrip routing in directed networks. *Journal of Algorithms*, 50:79–95, 2004.

[18] C. Demetrescu and G. F. Italiano. A new approach to dynamic all-pairs shortest paths. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 159–166, 2003.

[19] B. Derbel, C. Gavoille, D. Peleg, and L. Viennot. On the locality of distributed sparse spanner construction. In *Proceedings of 27th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 273–282, 2008.

[20] D. Dubhashi, A. Mei, A. Panconesi, J. Radhakrishnan, and A. Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *Journal of Computer and System Sciences*, 71:467–479, 2005.

[21] M. Elkin. Computing almost shortest paths. *ACM Transactions on Algorithms*, 1:282–323, 2005.

[22] M. Elkin. A near-optimal fully dynamic distributed algorithm for maintaining sparse spanners. *CoRR abs/cs/0611001*, 2006.

[23] M. Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proceedings of 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 185–194, 2007.

[24] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. In *Proceedings of 34th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 4596 of *LNCS*, pages 716–727. Springer, 2007.

[25] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 7(2):20, 2011.

[26] M. Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng. Lower-stretch spanning trees. *SIAM Journal of Computing*, 38:608–628, 2008.

[27] M. Elkin and D. Peleg. $(1+\epsilon,\beta)$-spanner construction for general graphs. *SIAM Journal of Computing*, 33:608–631, 2004.

[28] P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.

[29] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of association for computing machinery*, 28:1–4, 1981.

[30] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *Proceedings of 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 745–754, 2005.

[31] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14:781–798, 1985.

[32] S. Halperin and U. Zwick. Linear time deterministic algorithm for computing spanners for unweighted graphs. *unpublished manuscript*, 1996.

[33] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of association for computing machinery*, 46:502–516, 1999.

[34] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of association for computing machinery*, 48:723–760, 2001.

[35] J. Kruskal. On the shortest spanning subtree of a graph and the travelinf salesman problem. In *Proceedings of the American Mathematical Society*, volume 7, pages 48–50, 1956.

[36] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[37] D. Peleg and A. A. Schaffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[38] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of Association of Computing Machinery*, 36(3):510–530, 1989.

[39] S. Pettie. Distributed algorithms for ultra sparse spanners and linear size skeletons. In *Proceedings of ACM Symposium on Principle of Distributed Computing (PODC)*, pages 253–262, 2008.

[40] S. Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. In *Proceedings of 27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 253–262, 2008.

[41] I. Roditty, M. Thorup, and U. Zwick. Roundtrip spanners and roundtrip routing in directed graphs. *ACM Transaction on Algorithms*, 4:29:1–29:17, 2008.

[42] L. Roditty, M. Thorup, and U. Zwick. Deterministic construction of approximate distance oracles and spanners. In *Proceedings of 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *LNCS*, pages 261–272. Springer, 2005.

[43] L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *Proceedings of the 45th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 499–508, 2004.

[44] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proceedings of 12th Annual European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 580–591. Springer, 2004.

[45] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal of Computing*, 37(5):1455–1471, 2008.

[46] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT*, pages 384–396, 2004.

[47] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of Association of Computing Machinery*, 52:1–24, 2005.

[48] M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *Proceedings of 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 802–809, 2006.

[49] D. P. Woodruff. Additive spanners in nearly quadratic time. In *Proceedings of the 37th international colloquium conference on Automata, Languages and Programming (ICALP)*, pages 463–474, 2010.

Appendix A: Proof of Theorem 4.2

We first prove the following Theorem.

Theorem 8.1 *Let $o_1, \ldots, o_h$ be h positive numbers in the range $[1, b]$ where $b \leq 3/2$. Let X_i, $1 \leq i \leq h$ be h independent random variables such that X_i takes value o_i with probability p and zero otherwise. Let $\mathcal{X} = \sum_i X_i$ and $\mu = \mathbf{E}[\mathcal{X}] = \sum_i o_i p$. Then the following variant of Chernoff like bounds holds for a given $0 < \epsilon \leq 1/4$.*

$$\mathbf{Pr}[\mathcal{X} < (1-\epsilon)\mu] < e^{-\frac{\epsilon^2}{5}\mu}$$

Proof:

$$\begin{aligned}
\mathbf{Pr}[X < (1-\epsilon)\mu] &= \mathbf{Pr}[-X > -(1-\epsilon)\mu] \\
&= \mathbf{Pr}[\exp(-tX) > \exp(-(1-\epsilon)t\mu)] \\
&\leq \frac{\mathbf{E}[\exp(-tX)]}{\exp(-(1-\epsilon)t\mu)} = \frac{A(t)}{B(t)}
\end{aligned}$$

The last inequality follows from applying Markov Inequality. We shall now simplify $A(t)$ and $B(t)$ and evaluate them at $t = \ln \frac{1}{1-\epsilon}$.

$$A(t) = \mathbf{E}[\exp(-\sum_i tX_i)]$$

$$
\begin{aligned}
&= \mathbf{E}[\Pi_i \exp(-tX_i)] \\
&= \Pi_i \mathbf{E}[\exp(-tX_i)] \\
&= \Pi_i (p.e^{-to_i} + (1-p).1) \\
&= \Pi_i (1 + p(e^{-to_i} - 1)) \\
&\leq \Pi_i \exp^{p(e^{-to_i}-1)} \quad \{\text{since } 1 + x < e^x \ \forall\ x \in \mathbb{R}\}. \\
&= \exp\left(p \sum_i (e^{-to_i} - 1)\right) \\
&= \exp\left(p \sum_i ((1-\epsilon)^{o_i} - 1)\right) \\
&= \exp\left(p \sum_i ((1 - o_i\epsilon + \frac{o_i(o_i-1)}{2}\epsilon^2 + ...) - 1)\right)
\end{aligned}
$$

In the expansion of $(1-\epsilon)^{o_i}$, the sum of the terms with exponent of ϵ greater than 2 can be bounded as follows.

$$
\begin{aligned}
&\leq \frac{o_i(o_i-1)}{2 \cdot 3}\epsilon^2[\epsilon + \epsilon^2 + \cdots] \\
&\leq \frac{o_i(o_i-1)}{2 \cdot 3}\epsilon^2 \frac{\epsilon}{1-\epsilon} \\
&\leq \frac{o_i(o_i-1)}{2 \cdot 3}\epsilon^2 \frac{1}{3} \qquad \{\text{for } \ \epsilon \leq 1/4\}
\end{aligned}
$$

Using the above bound, we can get an upper bound on $A(t)$ as follows.

$$
\begin{aligned}
A(t) &\leq \exp\left(p \sum_i (-o_i\epsilon + \frac{10}{9}\frac{o_i(o_i-1)}{2}\epsilon^2\right) \\
&= \exp\left(p \sum_i (-o_i\epsilon + \frac{5o_i}{18}\epsilon^2)\right) \quad \{\text{since } o_i \leq \frac{3}{2}\} \\
&= \exp\left((-\epsilon + \frac{5}{18}\epsilon^2)\sum_i po_i\right) = \exp\left(-(\epsilon - \frac{5}{18}\epsilon^2)\mu\right)
\end{aligned}
$$

Let us simplify $B(t)$ at $t = \ln \frac{1}{1-\epsilon}$.

$$
\begin{aligned}
B(t) &= (1-\epsilon)^{(1-\epsilon)\mu} > (\exp(-\epsilon + \epsilon^2/2))^{\mu} \quad \{\text{for every } \ \delta \in (0,1] \ \} \\
&= \exp(-(\epsilon - \frac{1}{2}\epsilon^2)\mu)
\end{aligned}
$$

Combining the simplified expressions of $A(t)$ and $B(t)$ at $t = \ln \frac{1}{1-\epsilon}$, we get

$$\mathbf{Pr}[X < (1-\epsilon)\mu] \leq e^{-\frac{2\epsilon^2}{9}\mu} < e^{-\frac{\epsilon^2}{5}\mu}$$

•

Corollary 8.1.1 *Let $o_1, \ldots, o_h$ be h positive numbers such that ratio of the largest number to the smallest number is at most 3/2. Let X_i, $1 \leq i \leq h$ be h independent random variables such that X_i takes value o_i with probability p and zero otherwise. Let $\mathcal{X} = \sum_i X_i$ and $\mu = \mathbf{E}[\mathcal{X}] = \sum_i o_i p$. For a constant a and $\epsilon \leq \frac{1}{4}$, if $\mu > \frac{5a \ln h}{\epsilon^2}$,*

$$\mathbf{Pr}[\mathcal{X} < (1-\epsilon)\mu] < h^{-a}$$

We shall now extend the above results to the case where the ratio of the largest to smallest constant can be arbitrary value b. This is exactly the Theorem 4.2.

Theorem 8.2 *Let $o_1, \ldots, o_\ell$ be ℓ positive numbers such that the ratio of the largest to the smallest number is at most b, and $X_1, ..., X_\ell$ be ℓ independent random variables such that X_i takes value o_i with probability p. Let $\mathcal{X} = \sum_i X_i$ and $\mu = \mathbf{E}[\mathcal{X}] = \sum_i c_i p$. If $\ell = \Omega(\frac{b \log b}{\epsilon^3 p} \ln n)$ for any $n > \log b$ then the following bound holds for any $a > 1$.*

$$\mathbf{Pr}[X < (1-\epsilon)\mu] < O(n^{-a})$$

Proof: We fix ℓ to be a number greater or equal to $\frac{40(a+1) b \log b}{\epsilon^3 p} \ln n$. Let us group the numbers (and the associated random variables) into sub-groups G_j, $j < \log b$ such that ith sub-group consists of o_i's in the range $[(\frac{3}{2})^{i-1}, (\frac{3}{2})^i - 1]$. We shall use $G(o_i)$ to denote the group to which o_i belongs. Let $Y_j = \sum_{o_i \in G_j} X_i$ and $\mu_j = \mathbf{E}[Y_j]$. Further we shall say that a sub-group G_j is *big* if there are more than $\frac{20(a+1)\ln n}{\epsilon^2 p}$ constants in this sub-group, and *small* otherwise. Let $\mathcal{X}_{big}$ and $\mathcal{X}_{small}$ be the sum of the random variables from big groups and small groups respectively. Clearly $\mathcal{X} = \mathcal{X}_{big} + \mathcal{X}_{small}$. Let $\mu_{big} = \mathbf{E}[\mathcal{X}_{big}]$ and $\mu_{small} = \mathbf{E}[\mathcal{X}_{small}]$. In order to get an upper bound on the probability of X being less than $(1-\epsilon)\mu$, we proceed as follows. Firstly we show that the probability of $\mathcal{X}_{big}$ being less than $(1 - \frac{\epsilon}{2})\mu_{big}$ is very small. Here we essentially use Corollary 8.1.1 for each *big* subgroup and use the union bound. Then we use the fact that $\mu_{big} > (1 - \frac{\epsilon}{2})\mu$ by showing that $\mu_{small} < \frac{\epsilon}{2}\mu$. Together it leads to very small probability for deviation of X from $(1-\epsilon)\mu$. we provide the details below.

For a *big* subgroup G_j, it follows from the Corollary 8.1.1 that

$$\mathbf{Pr}[Y_j < (1 - \frac{\epsilon}{2})\mu_j] \leq n^{-(a+1)}$$

Hence arguing for each big group, and using union bound, we get

$$\mathbf{Pr}[\mathcal{X}_{big} < (1 - \frac{\epsilon}{2})\mu_{big}] < n^{-(a+1)} \log b \leq n^{-a} \tag{3}$$

Now let us bound μ_{small} as follows.

$$\mu_{small} = p \sum_{i:G(o_i) \text{ is } small} o_i \leq pb \log b \frac{20(a+1)\ln n}{\epsilon^2 p} \leq p\frac{\epsilon}{2}\ell \leq \frac{\epsilon}{2}\mu$$

Here we used the fact that there are at most $\log b$ *small* sub-groups and each *small* sub-group has less than $\frac{20(a+1)\ln \ell}{\epsilon^2 p}$ elements. It follows that $\mu_{big} \geq (1 - \frac{\epsilon}{2})\mu$. Combining it with Equation 3 we get,

$$\mathbf{Pr}[\mathcal{X}_{big} < (1-\epsilon)\mu] < \mathbf{Pr}[\mathcal{X}_{big} < (1 - \frac{\epsilon}{2})^2\mu] \leq \mathbf{Pr}[\mathcal{X}_{big} < (1 - \frac{\epsilon}{2})\mu_{big}] < n^{-a}$$

Since $\mathcal{X} \geq \mathcal{X}_{big}$, so $\mathcal{X} < (1-\epsilon)\mu$ implies $\mathcal{X}_{big} < (1-\epsilon)\mu$. Hence

$$\mathbf{Pr}[\mathcal{X} < (1-\epsilon)\mu] < n^{-a}$$

•