Fully Dynamic Randomized Algorithms for Graph Spanners in Centralized as well as Distributed environments*

Surender Baswana† Soumojit Sarkar‡

Abstract

Spanner of an undirected graph $G = (V, E)$ is a sub graph 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.

We present two randomized algorithms for maintaining a sparse t-spanner of an undirected unweighted graph under insertion and deletion of edges. Our algorithms significantly improve the existing fully dynamic algorithms for graph spanners in centralized as well as distributed environment. The expected size of the t-spanner maintained at each stage by our algorithms matches the worst case optimal size of a t-spanner up to poly-logarithmic factor, and the expected time required to process an update (insertion/deletion of an edge) 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.

‡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 to compute a subgraph with certain properties. These properties could be defined by some function which one aims to optimize for the given graph. For example, the minimum spanning tree problem aims to compute a connected subgraph having least total weight. The single source shortest paths problem aims to compute a rooted tree which stores the shortest paths from a source vertex to all the vertices in the graph. In this paper, we consider dynamic algorithms for one such well known subgraph called *spanner*. A spanner of an undirected graph is a sub graph which is *sparse* and yet preserves the all-pairs distance information of the given graph *approximately*. In precise words, for any $\alpha \geq 1, \beta \geq 0$, an (α, β)-spanner of the graph $G = (V, E)$ is a subgraph $H = (V, E_S)$ such that for all vertices $u, v \in V$,

$$\delta_G(u, v) \leq \delta_H(u, v) \leq \alpha\delta_G(u, v) + \beta$$

where δ_G is the distance in graph G. For an (α, β)-spanner, the parameters α and β are called multiplicative stretch and additive stretch respectively. We call a $(1, \beta)$-spanner an additive β-spanner. If $\beta = 0$, this definition reverts to the usual definition of a multiplicative α-spanner [30, 2].

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, 39, 34, 13, 14, 31], schemes for simulating synchronized protocols in unsynchronized networks [31]. Spanners are also used in parallel and distributed algorithms for computing approximate shortest paths [10, 18]. A recent application of spanners is the construction of labeling schemes and distance oracles [39, 7, 35, 4], 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 in 1989, the research area of spanner has witnessed many elegant and important results. In addition, many new structures similar to spanners have also evolved. For example, Bollobás et al. [9] 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. Efficient construction of d-preservers has been given by Coppersmith and Elkin [11]. Thorup and Zwick [40] introduced a variant of additive spanners where the additive error is sub-linear in terms of the distance being approximated. Though there are so many diverse applications of spanners (and related structures), from algorithmic perspective each of them have a common goal - to efficiently design sparsest spanner for a given additive and/or multiplicative stretch.

Among all (α, β)-spanners and related structures, it is the spanner with multiplicative stretch whose construction and optimality has attracted the maximum attention of the researchers[1]. For these spanners, the simple and elegant algorithm of Althöfer et al. [2] is well known. Throughout this paper, we shall use n and m to denote the number of vertices and edges respectively of the given graph. For any positive integer k and any weighted graph, the algorithm of Althöfer [2] computes a $(2k-1)$-spanner of $O(n^{1+1/k})$ edges. Based on a 45 year old girth conjecture of Erdős [23], 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. The algorithm of Althöfer is similar in spirit to the Kruskal's algorithm for minimum spanning tree. However, the best known implementation [2, 37] of this algorithm has $O(\min(kn^{2+1/k}, mn^{1+1/k}))$ running time. For unweighted graphs Halperin and Zwick [26] designed an $O(m)$ time algorithm to compute a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges. Finally, Baswana and Sen [7] designed a randomized $O(km)$ time algorithm for weighted graphs which computes a $(2k-1)$-spanner of $O(kn^{1+1/k})$ size. This algorithm is optimal in all respects, save a factor of k, in the running time and the size. The algorithm was derandomized by Roditty et al. [35].

In addition to the centralized algorithms for spanners as mentioned above, many efficient distributed algorithms have also been designed. These distributed algorithms are motivated by various applications of spanners in distributed computing. The centralized algorithm of Baswana and Sen [7] can be adapted easily in distributed environment to construct a $(2k-1)$-spanner in $O(k)$ rounds and $O(km)$ communication complexity. Derbel et al. [16] recently gave a deterministic distributed

[1]From now onwards, we shall omit the word multiplicative from α-multiplicative spanner

algorithm with matching efficiency. Note that the size of the sparsest spanner computed by these algorithms is of the order of $n \log n$. Recently Dubhashi et al. [17] and Pettie [32] independently showed how to compute a linear size $O(\log n)$-spanner in $O(\text{polylog}\, n)$ rounds.

As outlined above, many elegant static algorithms exist for computing spanners in centralized and distributed environment. However, these algorithms are not suitable for graphs which undergo updates. These updates could be deletion of existing edges and insertion of new edges. A naive way to handle these updates is to execute the best static algorithm from scratch after each update. Certainly this is a wasteful solution to solve any dynamic graph problem because the new solution might not be completely different from the previous one. This necessitates the design of a separate dynamic algorithm for the problem. The aim of a dynamic algorithm is to maintain some efficient data structure for the given graph problem such that the updates can be processed in time much faster than the static algorithm.

Various real life applications use graphs which are dynamic. Therefore, it becomes important to design an efficient dynamic algorithms for the problem underlying these applications. Motivated by the dynamic nature of real life graphs, many elegant dynamic algorithms have been designed in the past for various fundamental problems [25, 27, 28, 15, 38, 36]. 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 to update the spanner (and its associated data structures) upon insertion/deletion of an edge. This measure is commonly called the *update time* of the algorithm. Most of the graph problems, though having a simple solution in the static setting, pose nontrivial challenges in dynamic settings. Unfortunately, as will soon become clear, the same is true for the graph spanners also. A dynamic graph problem, which seems closer to the problem of maintaining graph spanner is the all-pairs approximate shortest paths (APASP) problem. Most of the static solutions of this problem [18, 6, 4] have employed spanners. However, there does not exist any dynamic algorithm for APASP which achieves sub-quadratic update time. The potential difficulty in handling updates efficiently is due to the fact that a single edge deletion/insertion can lead to change in distances for $\Theta(n^2)$ pairs of vertices.

Given the existence of a linear time static algorithm to compute a $(2k-1)$-spanner, an ideal goal of a fully dynamic centralized algorithm would be to achieve constant or poly-logarithmic update time per edge insertion/deletion. Similarly an ideal goal of a fully dynamic algorithm in distributed environment would be to achieve constant communication complexity per edge update. Naturally, these goals 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. In particular, 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, surprisingly and contrary to these intuitions, this paper presents fully dynamic algorithms for spanners which nearly achieve these goals.

Remark. For the class of geometric graphs, the problem of dynamic maintenance of spanners has been addressed by various researchers, and many insightful algorithms exist for this problem (see the recent work of Roditty [33] and references therein). However, the underlying techniques of all these algorithms employ the geometry of the (geometric) graphs in a crucial manner, and don't seem to be extensible to general graphs.

Prior work and our contributions. Figure 1 provides a comparative description of the previously existing fully dynamic algorithms for graph spanners and the new algorithms designed in this paper. These algorithms are for unweighted graphs.

Fully dynamic centralized algorithms. Ausiello *et al.* [3] are the first to design fully dynamic centralized algorithms for graph spanners. They present a fully dynamic deterministic algorithms for maintaining spanners with stretch at most 5 only, and the update time guaranteed is $O(n)$.

Fully Dynamic Centralized 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]
$2k-1$	$O(kn^{1+1/k})$	$O(mn^{-1/k})$	Elkin [21]
$2k-1$	$O(n^{1+1/k}\operatorname{polylog} n)$	$O(\operatorname{polylog} n)$	**New**
$2k-1$	$O(n^{1+1/k}\operatorname{polylog} n)$	$O(7^{k/2})$	**New**

Fully Dynamic Distributed Algorithms

Stretch	Size	Quiescence time complexity	Message Complexity per update	Notes
$2k-1$	$O(kn^{1+1/k})$	$3k$	$O(mn^{-1/k})$	Elkin [20]
$2k-1$	$O(n^{1+1/k}\operatorname{polylog} n)$	k	$O(7^k)$	**New**

Figure 1: Current state of the art dynamic algorithms for maintaining spanner of an unweighted graph in centralized and distributed environments.

Their algorithm does not seem extensible to arbitrary stretch. Recently Elkin [21] designed dynamic centralized algorithms for $(2k-1)$ spanners for any k. In the same paper, Elkin also designed an optimal streaming algorithm for computing spanners. If only insertion of edges is allowed, his dynamic algorithm achieves $O(1)$ update time per insertion. However, for fully dynamic environment, the expected update time ensured by his algorithm is $O(mn^{-1/k)})$. Thus there had been no fully dynamic centralized algorithm for graph spanners till date which could even achieve sub-linear update time.

We present two fully dynamic centralized algorithms which improve the update time to the extent of near optimality. The expected size of the $(2k-1)$-spanner maintained at each stage by our algorithms matches the worst-case optimal size of a $(2k-1)$-spanner up to a poly-logarithmic factor.

Our starting point is the static linear time algorithm of Baswana and Sen [5]. Unlike its seamless adaptability in parallel, distributed and external memory environment, it poses nontrivial challenges in order to adapt it in a dynamic environment. These challenges are posed by hierarchical structure of the algorithm wherein there is a huge dependency of upper level structures on lower level. 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 are able to design two fully dynamic algorithms for maintaining $(2k-1)$-spanners. The reader may please note that the ideas underlying the two algorithms are mutually disjoint, therefore, each of them can be studied in any order.

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 update time achieved for $(2k-1)$-spanner is of the order of $7^{\frac{k}{2}}$ per edge insertion/deletion. Hence constant stretch spanners can be maintained fully dynamically in expected constant time per edge update. The algorithm is particularly of interest for small stretch spanners. From theoretical perspective, note that $\log n$ stretch is necessary and sufficient for achieving the sparsest spanner (of size $O(n)$), and for this stretch our algorithm achieves $O(n^{0.7})$ update time. However, the following point is worth consideration: If an application requires 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 update time achieved by our 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 update time for any value of $\epsilon > 0$.

Our second fully dynamic algorithm guarantees expected $O(k\log^3 n)$ update time for maintaining a $(2k-1)$-spanner. At the core of the algorithm lies an efficient decremental algorithm which maintains a $(2k-1)$-spanner of expected $O(kn + n^{1+1/k})$ size. The expected update time ensured by the algorithm is $O(k\log^2 n)$ per edge deletion. The fully dynamic algorithm extends this new

decremental algorithm to a fully dynamic environment at the expense of increasing the update time and size by a factor of $O(\log n)$ each.

Fully dynamic distributed algorithms. In the distributed model of computation, each vertex corresponds to a node hosting a processor and each edge corresponds to a link. The computation proceeds in rounds in this distributed environment. In each round, each processor sends and receives messages, and performs some local computation. As mentioned in [20], a common way to measure the performance of a distributed dynamic algorithm is through *quiescence* complexities. Suppose that all updates stop occurring in round α, and that the algorithm reaches a state in which the (distributed) structure that it is maintaining starts again to satisfy the properties of the problem at hand at the end of round β. 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 paper [20], most dynamic distributed algorithms are built by using simulation techniques on top of a static distributed algorithm. These simulation technique transform the static algorithm to dynamic algorithm. Unfortunately these simulation techniques have numerous drawbacks (see [20]). This necessitates the search for designing dynamic distributed algorithm from scratch. Pursuing this direction, Elkin [20] designed the first fully dynamic distributed algorithm for spanners which is quite elegant and efficient compared to the algorithm formed by combining the simulation technique and the static algorithm [7].

The fully dynamic randomized distributed algorithm of Elkin [20] achieves $3k$ quiescence time complexity. The quiescence message complexity achieved by the algorithm is $O(km)$. The lower bound on quiescence time complexity is $k - 1$ assuming Erdős's girth conjecture [23]. The lower bound on quiescence message complexity per update is $\Omega(m)$. However, note that the $\Omega(m)$ lower bound on quiescence message complexity is derived for the worst case when there are $\Theta(m)$ simultaneous updates in a short span of rounds. However, it is unlikely that such a huge number of simultaneous updates occur frequently. Therefore, a more precise as well as more realistic measure of message complexity of dynamic algorithm should be in terms of the number of updates. The fully dynamic distributed algorithm of Elkin [20] achieves expected $O(mn^{-1/k})$ bound on message complexity per update which is quite large. We present a distributed fully dynamic algorithm for $(2k-1)$-spanner which achieves significantly improved bounds compared to the algorithm of Elkin [20]. Our algorithm achieves quiescence time equal to k and expected message complexity of $O(7^k)$ per update. It can be observed that the improvement in message complexity is really drastic, especially for small value of k. However, this improvement comes by paying some extra price - the expected size of the spanner ensured by our algorithm is larger than that of [20] by a poly-logarithmic factor. Our distributed algorithm is derived from one of our fully dynamic centralized algorithm seamlessly.

Additional features of our algorithms. An important observation, of independent interest, which follows from our fully dynamic algorithms is that a single insertion or deletion of an edge requires very few and local updates in the spanner. This observation is in complete contrast with the all-pairs shortest paths problem where a single edge update may lead to change in $\Theta(n^2)$ entries in the all-pairs distance matrix.

Our poly-logarithmic update time algorithm leads to improved bounds on update time for the problem of fully dynamic maintenance of all-pairs approximate shortest paths (APASP) in undirected unweighted graphs. In particular, we are able to achieve sub-quadratic update time and sub-linear query time for this problem. Previously no such bounds were known for this dynamic problem.

Our fully dynamic centralized algorithms can be employed for maintaining spanners for weighted graphs also, though the bounds will become slightly inferior. Let $w_{\max}$ and $w_{\min}$ be the weights of the heaviest and lightest edge respectively in the graph. For any $\epsilon > 0$, we can 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, we maintain a $(2k-1)$-spanner ignoring the weights on edges. The union of these spanners will be a $(2k-1)(1+\epsilon)$-spanner for the given graph. The update time of the algorithm and the size of

spanner thus maintained will be larger by a factor of $\log_{1+\epsilon} \frac{w_{\max}}{w_{\min}}$.

The decremental algorithm, which is a part of our second fully dynamic algorithm, achieves $O(k \log^2 n)$ update time and the expected size of the spanner maintained is $O(kn + n^{1+1/k})$ which is optimal, save a factor of k.

The eventual design of each of our algorithms is simple though their underlying ideas and analysis are nontrivial, making them ideal for practical applications.

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 [7] 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 sake of 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. We shall maintain the set of edges of the dynamic graph $G = (V, E)$ using a dynamic hash table (see [29]). For each edge of the graph, we shall maintain a bit to denote whether it is a spanner edge at the current moment of time.

The task 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 subgraph (V, E_S) by a path consisting of at most $(2k-1)$ edges

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

Definition 2.1 *[7] 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 $C[\,]$ 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).*

For each clustering, we shall associate a spanning forest $\mathcal{F} \subset E$ such that each cluster 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.

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.*
$S_0 = V, S_k = \emptyset$. S_i for $0 < i < k$ is formed by selecting each vertex from S_{i-1} independently with probability $n^{-1/k}$.

For sake of compactness, we use $G' \subseteq G''$ to denote that the graph G' is a sub graph of G''. The following notations will be used throughout this paper. In these notations, C correspond to a clustering, $H \subseteq E$, $X \subset V$, c is any cluster of C, u is any vertex not belonging to c, and $\mathcal{R}$ is any subset of clusters from C.

- $H(u, c)$: the set of edges from H which are between u and vertices of cluster c. cluster c is said to be *adjacent* or *neighboring* to u if $H(u, c) \neq \emptyset$.

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

The main problem in computing and maintaining spanner is to keep the size of the spanner small without increasing the stretch too much. The clustering plays a crucial role to solve this problem as suggested by the following observation.

Observation 2.1 *For a graph (V, E), let C be a clustering of radius i for vertices V, and let $\mathcal{F}$ 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 sub graph $E_S \cup \mathcal{F}$ ensures that $\mathcal{P}_{2i+1}$ holds for all the edges E.*
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 sub graph $E_S \cup \mathcal{F}$ ensures that $\mathcal{P}_{4i+1}$ holds for all the edges E.*

Though it does not seem feasible to use a single clustering to compute or maintain a $(2k-1)$-spanner for any arbitrary k, a hierarchy of clusterings achieves the goal [7] as shown in the following subsection.

2.1 A Hierarchy of Clusterings induced by $\mathcal{H}_k$

Given a graph $G = (V, E)$, the hierarchy $\mathcal{H}_k$, in a very natural manner, defines $k+1$ levels of subgraphs $G_0 \supseteq G_1 \supseteq \cdots \supseteq G_k$, clusterings $C_0, C_1, \ldots, C_k$, and the spanning forests $\mathcal{F}_0, \ldots, \mathcal{F}_{k-1}$ as follows.

Clustering at level i will be a clustering centered at vertices of set S_i. For level 0, the subgraph G_0 is (V, E), the clustering C_0 is $\{\{v\} | v \in V\}$, and $\mathcal{F}_0$ is empty. Graph G_k is an empty graph - $V_k = \emptyset = E_k$, and so is the clustering C_k. The clustering and the subgraph at level $i+1, 0 \leq i < k$ is defined by the clustering and subgraph at level i, and S_{i+1} in the following manner.

Among the clusters of C_i, the clusters whose centers belong to S_{i+1} are called the *sampled* clusters of level i and this set is denoted by R_i.

- 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.
- Each cluster of C_{i+1} is a cluster from R_i with an additional layer of some neighboring vertices joining it 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, and then each vertex $v \in \mathcal{N}_i$ concurrently joins (hooks on) to a neighboring cluster, say $c \in R_i$ through some edge called $hook(v, i)$ from $E_i(v, c)$. This defines C_{i+1}.
- The set E_{i+1} is defined as $E_i(C_{i+1}, C_{i+1})$.
- Spanning forest F_{i+1} for C_{i+1} is the union of the spanning forest F_i confined to R_i and the set of hooks selected by $\mathcal{N}_i$. That is, $F_{i+1} = F_i(R_i) \bigcup \{hook(v, i) | v \in \mathcal{N}_i\}$

Conceptually each level looks the same : The subgraph (V_i, E_i) with V_i partitioned into clusters by C_i, and E_i consists of only the inter-cluster edges and F_i is the spanning forest for C_i ensuring a radius i for each cluster. We shall use $\mathcal{C}_k$ to denote the hierarchy of clusterings $C_i, 0 \leq i \leq k$ 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 $F = \cup F_i$. Let us form a subset $E_S \subseteq E$ which contain F and the edges implied by the following rule - Each vertex v which belongs to V_i but not to V_{i+1} adds one edge from $E_i(v, c)$ to E_S for each neighboring $c \in C_i$.

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

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

The hierarchical clustering $\mathcal{C}_k$ forms the back bone of the static algorithms [7, 8], and will be the starting point for our dynamic algorithm as well. Therefore, for better understanding of the dynamic algorithms, we first provide the intuition as to why such a hierarchy of clustering is a very natural way to compute and maintain a $(2k-1)$-spanner. Firstly the very idea of clustering is based on achieving sparseness as follows from Observation 2.1. 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. Consider a vertex v 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 follows that we have ensured $\mathcal{P}_{2k-1}$ for all edges in $E_i(v, c)$. However, the problem in this naive approach is that this would lead to a large size spanner if there are too *many* clusters from C_i which are neighboring to a vertex. To overcome this problem, the random sampling underlying the formation of S_i's plays a crucial role - if v has large number of clusters neighboring to it from C_i, it is likely to have at least one neighboring cluster which will be a sampled cluster. So v just joins that 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 until it reaches a level where there are a *few* neighboring clusters where it can afford to add one edge per neighboring cluster and its role ends there. In this manner, the hierarchy of clustering $\mathcal{C}_k$ suffices to obtain a $(2k-1)$-spanner. Now for the size of the spanner, let us take a vertex centric approach. Note that each cluster in C_i is a sampled cluster independently with probability $n^{-1/k}$. A vertex at any level i would contribute single edge (and joins clustering C_{i+1}) if it has any sampled cluster from C_i neighboring to it, and contributes one edge per neighboring cluster otherwise. It follows from simple probability that the expected number of edges contributed by a vertex in this manner is bounded by $k + n^{1/k}$, and so the expected size of the spanner is $O(n^{1+1/k} + kn)$.

From the above insight into the hierarchical clusterings $\mathcal{C}_k$, it can be seen that the following two invariants at each level i are crucial to ensure $O(n^{1+1/k} + kn)$ size and $(2k-1)$ stretch of a spanner.

- CLUSTERING-INVARIANT - each vertex $v \in V_i$ neighboring to a sampled cluster at level i must be a member of the clustering at level $i+1$.
- SPANNER-EDGE-INVARIANT - each vertex $v \in V_i$ not neighboring to any sampled cluster at level i adds one edge from $E_i(v, c)$ to the spanner for each $c \in C_i$ neighboring to it. v will not be present at level $i+1$ in this case.

It can be observed that the spanner consists of precisely those edges which get added as a consequence of the two invariants mentioned above. Hence, in the dynamic scenario, maintaining

the hierarchy of clustering $\mathcal{C}_k$ and the two invariants mentioned above will directly imply a $(2k-1)$-spanner of expected $O(n^{1+1/k}+kn)$ size at any stage. As will become clear from the following section, this approach, though simple in words, is quite challenging.

Data structure. As the reader may easily observe that the clustering and the two invariants have a local and vertex centric approach - each vertex needs clustering information of only its neighbors to determine its clustering at the next level and also for maintaining the two invariants. Therefore, each vertex has to have an efficient data structure to maintain information about the clusters to which each of its neighbors belong. We now give a description of this data structure which will be useful in both the dynamic algorithms for $(2k-1)$-spanner.

- An array $C_i[\,]$ is used for storing the clustering at level i.
- Each vertex u keeps a hash table h_u for storing the centers of all clusters neighboring to it. In addition we also store 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 [29], ensure worst case constant query/search time and expected constant update time.
- We keep the edges $E_i(u)$ arranged in a doubly linked list in such a way that the edges emanating from the same cluster appear contiguously in the list. In addition, for each cluster c neighboring to u, we also keep pointers from its entry in the hash table h_u to the beginning and end of the portion of the adjacency list storing $E_i(u,c)$.

3 Overview of the two algorithms

For maintaining a $(2k-1)$-spanner, let us explore the hurdles in maintaining the hierarchy of clustering $\mathcal{C}_k$ 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 (graphs and clustering) on the lower level objects of the hierarchy. It follows from the hierarchy of clusterings that $E_i(u) \supseteq E_{i+1}(u)$ 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 we delete the edge (u,v), we just discard 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. In this case, vertex u might have added (u,v) to spanner as a candidate edge from $E_i(u,c), v \in c$. So upon deletion of (u,v), we just have to find some other candidate edge from $E_i(u,c)$ to be included in the spanner in order to maintain SPANNER-EDGE-INVARIANT. Using the simple data structure at u as mentioned above, this can also be accomplished in $O(1)$ time.

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 $hook(u,i)$. This is the costliest update due to the following reason. This deletion may forbid u to join its present cluster at level $i+1$. However, if u is still having some neighboring sampled cluster at level i, it must join one such cluster at the next level in order to maintain CLUSTERING-INVARIANT. So in this case let us assume that vertex u changes its cluster from c to c' at level $i+1$. This change leads to change in vertex-cluster neighborhood information for each vertex which is present at level $i+1$ and was neighbor of u at level i as well. More precisely, for each such vertex this change is equivalent to deletion of an edge incident from c and/or insertion of an edge incident from c'. Each such neighbor has to process these updates individually for maintaining CLUSTERING-INVARIANT and SPANNER-EDGE INVARIANT at level $i+1$. Note that potentially all neighbors of u at level i may be present 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 $i+1$ as follows. Suppose u is a vertex which belongs

to C_i but does not belong to C_{i+1}, so presently there is no *hook* available for u at level i. If the new edge inserted is such that it is a potential hook, so u will have to join the next level clustering for maintaining CLUSTERING-INVARIANT. This is equivalent to introducing potentially $\Theta(|E_i(u)|)$ updates at $i+1$ level. Furthermore, each of these updates when processed can in turn cause yet higher number of updates to be introduced at the next higher level. 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 [7].

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

First algorithm. We start with a dynamic algorithm for 3-spanner. For this, we augment the static algorithm described in [7] with the following randomization. Each unsampled vertex u at level 0 which is a member of clustering at level 1 maintains the following invariant at every moment : Each edge incident on u from a sampled cluster at level 0 is equally likely to be $hook(u, 0)$. In simple words, we ensure that a vertex selects and maintain a random hooking edge among all potential hooks present at any moment. This minor augmentation in the static algorithm has the following powerful consequence : Deletion or insertion of a single edge on u will cause change in the clustering of u at level 1 with probability which is close to the inverse of $E(u)$. As a result of this consequence, the expected number of updates to be performed due to a single edge deletion or insertion becomes constant. This leads to expected $O(1)$ update time for a fully dynamic algorithm for maintaining a 3-spanner. However, this simple and elegant idea of selecting a *random hooking edge* which works miraculously for spanners of stretches 3 or 5 fails for higher values of k due to the possible skewed distribution of the edges of a vertex among its neighboring clusters. To overcome this problem, we employ the idea of pruning some edges at each level in our fully dynamic algorithm for $(2k-1)$-spanner as follows. We maintain a data structure for each vertex u which partitions the neighboring unsampled clusters of a vertex into buckets such that the clusters which are incident on u with *nearly* same number of edges belong to the same bucket. We allow only those edges to be part of E_{i+1} which are incident from clusters belonging to buckets of large size (the number of clusters). For the edges of u belonging to smaller sized buckets we ensure $\mathcal{P}_{2i+1}$ by adding *a few* edges in the spanner at level i itself. The selective pruning of edges combined with the idea of randomly selecting the hooking edge form the core of our first fully dynamic algorithm. It ensures that for any edge update at a level i, the expected number of updates introduced at level $i+1$ is a constant ≤ 7. This leads to $O(7^k)$ update time algorithm with a careful analysis and calculations. 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.

Second algorithm. Our second algorithm takes another route altogether to achieve efficiency. From a slightly different perspective 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. In our second algorithm we essentially try to get rid of this dependency. For this, we design a completely new clustering where the dependency among the levels is minimal so that we need to maintain clustering and follow SPANNER-EDGE INVARIANT at each level in a way almost *independent* of other levels. We design a decremental algorithm for spanners using this clustering and incorporate another randomization principle. Using this principle in the new clustering, we show that a vertex will change its cluster at any level expected $O(\text{polylog}\, n)$ times for any arbitrary sequence of edge deletions. We then extend the decremental algorithm to fully dynamic scenario 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 is similar to those used by Henzinger and King [27] and Holm et al. [28] in the context of fully dynamic connectivity.

A common point between the two algorithms is the following. Both of them maintain hierarchies of subgraphs, clustering and slightly *modified* CLUSTERING-INVARIANT and SPANNER-EDGE-INVARIANT. However, the approach of first algorithm is totally vertex centric. This makes it adaptable in distributed environment seamlessly. The second algorithm uses a mixture of vertex centric

and global approach. The global approach is used for maintaining the new clustering which is maintained using a truncated BFS tree.

4 Fully Dynamic Centralized Algorithm - I

In this section we shall design expected $O(7^{\frac{k}{2}})$ update time fully dynamic algorithm for maintaining a $(2k-1)$-spanner of expected size $O(n^{1+1/k}\operatorname{polylog} n)$. The starting point of our fully dynamic algorithm is the following very simple fully dynamic algorithm for 3-spanner.

4.1 Fully dynamic algorithm for 3-spanner

The fully dynamic algorithm for 3-spanner is essentially designed by adding slight randomization to the static algorithm for 3-spanner. Recall that the static algorithm uses the hierarchy $\mathcal{H}_2 = \{S_0 = V, S1, S_2 = \emptyset\}$, where 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 has essentially 2 levels of subgraphs and clusterings as follows.

- Level 0 consist 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-VARIANT), 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 for 3-spanner maintains two levels of subgraphs and clusterings, the SPANNER-EDGE-INVARIANT, and slightly augmented CLUSTERING-INVARIANT. Clustering at level 0 is $\{\{u\}|u \in V\}$ just like the static algorithm. However, clustering at level 1 is slightly different. Though it consists of clusters centered at S_1 like in the static algorithm, but the way the vertices of set $V\backslash S_1$ join clusters employs randomization as follows. Each vertex $u \in V\backslash S_1$ with $|R(u)| > 1$ selects the cluster from $R(u)$ uniformly randomly and hooks on to it at level 1. Throughout the sequence of updates, the following invariant is maintained by each such vertex u.

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

An important consequence of the above invariant and the randomization used in construction of S_1 is the following Lemma.

Lemma 4.1 *At each stage of the algorithm the following assertion holds for each edge $(u,v) \in E$.* $\mathbf{Pr}[(u,v) = hook(u,0)|u \in V_1] = \frac{1}{\deg(u)}$

The proof of the above lemma is based directly on the following observation whose proof can be easily derived from elementary probability. Consider a two level sampling to select a *winner* ball from a bag containing arbitrary number of balls - first select a sample of j balls uniformly randomly from the bag 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 We now state the following corollary of Lemma 4.1 which we shall employ in the analysis of our dynamic algorithm.

Corollary 4.0.1 *Insertion or deletion of an edge (u,v) will lead to change in clustering of u at level 1 with probability $1/\deg(u)$.*

Let us analyze the computation involved in handling a single update - insertion or deletion of an edge, say, (u,v). The update is processed at level 0 first, where it takes $O(1)$ time. The following Lemma bounds the number of updates passed to level 1 due to a single update at level 0.

Lemma 4.2 *For each edge insertion or deletion in the graph, the expected number of updates to be processed at level 1 in the dynamic algorithm is at most 2.*

Proof: Let us consider deletion of an edge (u, v). Let us analyze how u handles the deletion of (u, v) (the vertex v handles it analogously). There are two cases. The first case is when (u, v) is $hook(u, 0)$ before the deletion of (u, v). The probability of this is $1/\deg(u)$ (see Corollary 4.0.1). Maintaining the new clustering invariant, vertex u will change its cluster at level 1 (unless $R(u) = \emptyset$ now). For each neighbors of u at level 1, this will introduce a single update corresponding to the change in clustering of u, and thus the updates to be passed to level 1 in this case will be at most $\deg(u)$. The second case is when (u, v) is not $hook(u, 0)$. In this case, the only update passed to level 1 is deletion of (u, v) itself. Hence the expected number of updates to be processed at level 1 by u is at most 2.

Let us consider an edge insertion. Let (u, v) be the edge inserted. First we restore the invariant 1. Probability that $hook(u, 0) = (u, v)$ holds after this restoration is $1/\deg(u)$, and if so the number of updates introduced any level 1 is at most $\deg(u)$. On the other hand if $hook(u, 0) \neq (u, v)$, the only update introduced at level 1 is insertion of (u, v) itself. Hence the expected number of updates to be processed at level 1 by u is at most 2. •

It can be seen that at level 1, we have to just maintain Spanner-edge-invariant only since there is no clustering at higher level. Using the data structure mentioned earlier, each update at level 1 can be processed in $O(1)$ time. So it follows from Lemma 4.2 that the expected time spent at level 1 in processing a single edge update from level 0 is $O(1)$.

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.*

4.2 On generalizing the dynamic algorithm for arbitrary k

From the fully dynamic algorithm for 3-spanner described above, it follows that the key principle which ensured expected $O(1)$ update time is the following : Each vertex which joins clustering at level 1 selects a random hooking edge from all edges incident from neighboring sampled vertices (singleton clusters). The result of this principle is that an edge $(u, v) \in E(u)$ is $hook(u, 0)$ with probability which is equal to the inverse of $|E(u)|$. The efficacy of this consequence was demonstrated in the rigorous analysis above. One would like to achieve similar results at each level of clustering for the fully dynamic algorithm for $(2k - 1)$-spanner. Unfortunately the random hooking edge principle alone does not work for arbitrary k when the distribution of edges incident on a vertex among its neighboring clusters is skewed. The reader may consider the following example in order to realize this fact. Consider any vertex u at any level i where $0 < i < k - 1$. 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 the distribution of edges $E_i(u)$ be 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 endpoint in c_ℓ. In addition, suppose most of the neighbors of u at level i are also present at level $i+1$ so that $|E_{i+1}(u)| \approx |E_i(u)|$. 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 polylog n edges incident on u from sampled clusters at level i. In this scenario, the random hooking edge principle would imply that the probability that edge (u, v) is $hook(u, i)$ is approximately $1/\ell$ which is much larger than $1/|E_{i+1}(u)|$. Therefore the expected number of updates introduced at level $i + 1$ due to deletion of an edge incident from $c_j, j < \ell$ could be $|E_{i+1}|/\ell$ which is not a constant.

The skewed distribution of edges incident on a vertex among its neighboring clusters at a level is hard to be ruled out since it depends upon the edge distribution in the original graph and the sampling done at previous levels. To overcome this hurdle, we shall use a technique of classifying the clusters according to the number of edges they share with a vertex, and a randomization principle as mentioned in the following Theorem. This theorem can be viewed as an extension of Chernoff's bound and its proof is given in Appendix.

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.*

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

Bucket structure. 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. It can be seen that the ranges of two adjacent buckets are overlapping, this will facilitate their efficient maintenance in a dynamic environment. Because the ranges are overlapping, cluster may not need to hop back to its old bucket immediately after it moved to a new one. So we add sufficient inertia for change which will help us in reducing the time-complexity. This is an adaptation of the well-known data structures of dynamic tables [12], a detailed description of the bucket-structure given later will make this point clear. For initially assigning clusters to buckets, we break the ties arbitrarily. 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. In this manner, let $\text{ACT}(u)$ and $\overline{\text{ACT}}(u)$ be the edges incident on u from active and inactive buckets respectively. Let C_i, $G_i = (V_i, E_i)$ be the clustering and the subgraph at level i, with R_i be the set of sampled clusters at level i. Based on the bucket structure we define E_i^{act}, which is a subset of E_i, as follows.

$$E_i^{act} = E_i(R_i) \cup \{(u,v) \in E_i | (u,v) \in \text{ACT}(u) \wedge (u,v) \in \text{ACT}(v)\}$$

Remark. 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. It can be seen that a cluster can belong to active buckets of some vertices and to inactive buckets of others.

The following lemma will be crucially used on the analysis of our fully dynamic algorithm.

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

Proof: To prove the theorem, we describe a procedure for the formation of bucket structure of vertex u and analyze it using Theorem 4.2. Partition all the clusters neighboring to u at level i into groups such that ith group has all those clusters which are incident on u with edges in the range $[(\frac{1}{\epsilon})^{i-1}, (\frac{1}{\epsilon})^{i+1} - 1]$. Note that ith groups has the same range as that of ith bucket. 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_i being the number of edges incident from ith 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. Applying union bound for each 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 try to relate the set $E_i^{big}(u)$ with the set $E_i^{act}(u)$. If the number of unsampled clusters of a big group is less than $a\gamma n^{1/k} \frac{\ln n}{\epsilon^6}$, then these clusters won't contribute any edge to $E_i^{act}(u)$ since it will correspond to an *inactive* bucket. Now consider a small group, the edges incident from sampled clusters of this group will also be present in $E_i^{act}(u)$. Thus it can be seen that obtaining $E_i^{act}(u)$ from $E_i^{big}(u)$ involves removing some edges incident on u from unsampled clusters and adding some edges incident on u from sampled clusters. Thus if we consider the fraction of edges incident from sampled clusters, this fraction may only be bigger in $E_i^{act}(u)$ than $E_i^{big}(u)$. This concludes the proof of this lemma. •

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

The backbone of the fully dynamic algorithm for $(2k-1)$-spanner is a hierarchy of clustering induced by $\mathcal{H}_k$ which differs from that of $\mathcal{C}_k$ due to added randomization incorporated in the clustering - where a vertex selects a random hooking edge. To make this idea work, each vertex maintains a bucket structure for storing its neighboring clusters.

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 $\mathcal{N}_i$ consists 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 from $\mathcal{N}_i$ selects the neighboring sampled cluster to hook onto randomly ensuring the following invariant.

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

  where $R_i(u)$ is the set of edges incident on u from clusters of set R_i.
- Each vertex maintains bucket structure for its neighboring clusters which defines the set of active edges E_i^{act}. The set E_{i+1} is defined as

$$E_{i+1} = E_i^{act}(C_{i+1}, C_{i+1})$$

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

The spanner E_S at any stage would consist of the union of the spanning forests at each level and the edges contributed by the following invariant.
NEW-SPANNER-EDGE-INVARIANT : Each vertex u of set V_i but not present in V_{i+1} keeps one edge from $E_i(u,c)$ in the spanner for each neighboring cluster $c \in C_i$. In addition, vertices joining some cluster at level $i+1$ also keep some edges in the spanner : Each vertex $u \in \mathcal{N}_i$ keep one edge from $E_i(u,c)$ in the spanner for each neighboring cluster $c \in C_i$ belonging to inactive bucket.

Along same lines to Lemma 2.2, it follows that E_S as defined above will be a $(2k-1)$-spanner at each stage. However, due to the NEW-SPANNER-EDGE-INVARIANT, the expected size of the spanner will increase to $O(k/\epsilon^6 n^{1+1/k} \log n)$. 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 $\frac{1+2\epsilon}{|E_i^{act}(u)|}$, for any constant $0 < \epsilon < 1/2$.*

Proof: First note that no edge from $E_i \backslash E_i^{act}$ can be $hook(u,i)$. 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 $(1-\epsilon)|E_i^{act}(u)|$ with probability at most n^{-a}. Let us now consider an edge $(u,v) \in E_i^{act}(u)$, 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 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.

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

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

$$\mathbf{Pr}[hook(u,i) = (u,v)] \quad = \quad \left(\sum_{d \geq 1} \frac{\mathbf{Pr}[X = d | c \in \mathcal{R}_i]}{d} \right) \mathbf{Pr}[c \in \mathcal{R}_i]$$

$$
\begin{aligned}
&= n^{-1/k} \sum_{d \geq 1} \frac{\mathbf{Pr}[X = d | c \in R_i]}{d} \\
&\leq n^{-1/k} \frac{\mathbf{Pr}[X \leq d | c \in R_i]}{1} + n^{-1/k} \frac{\mathbf{Pr}[X > d | c \in R_i]}{d} \quad \{\text{for any } d \geq 1\} \\
&\leq n^{-1/k} \mathbf{Pr}[X \leq d] + n^{-1/k} \frac{1}{d} \quad \{\text{using Equation 2}\} \\
&\leq n^{-1/k} n^{-a} + n^{-1/k} \frac{1}{(1-\epsilon) n^{-1/k} |E_i^{act}(u)|} \quad \{\text{for } d = (1-\epsilon) n^{-1/k} |E_i^{act}|\} \\
&\leq n^{-a} + \frac{1}{(1-\epsilon)|E_i^{act}(u)|} \leq \frac{1+2\epsilon}{|E_i^{act}(u)|} \quad \{\text{for } \epsilon \leq \frac{1}{2}\}.
\end{aligned}
$$

•

4.4 Handling the updates by the dynamic algorithm

An update (insertion or deletion of an edge) in the graph is processed by our algorithm in a bottom up fashion to maintain a $(2k-1)$-spanner. As can be seen that each update reaching a level $i > 0$ will be of one of the following types : insertion of an edge, deletion of an edge, and change in clustering of some vertex. Processing of these updates will be intrinsically local - each vertex being affected by an update will restore its two invariants and update its own bucket structure. However, from view point of this local processing, none of the updates mentioned above are isolated or atomic updates in the following sense. Insertion (or deletion) of an edge will require processing by both the endpoints individually instead of by only one endpoint. A change in clustering of a vertex entails processing by all of its neighbors and not by the vertex whose clustering changed (We mentioned it briefly in section 3 as well). Therefore, following the local and vertex centric approach, we shall express these *macro* updates in terms of the corresponding *atomic* updates - each of which needs to be processed by exactly one vertex locally. Each level i will receive a list L_i of the vertices which have to process one or more updates, and an array $U_i[1..n]$ whose jth entry will store the list of updates to be processed by jth vertex locally. Any macro update will be represented implicitly in the array U_i as follows.

- For a macro update which is deletion (or insertion) of edge (u, v), there will be one entry in the list $U_i[u]$ for the deletion of (u, v) to be processed by u, and one entry in $U_i[v]$ 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', there will be an entry in the list $U_i[u]$ for the change of cluster of u from c to c'. From what we need in our algorithm, each neighbor of u at level i can simply perceive this change in the clustering of u as deletion of a single edge incident from c and insertion of a single edge from c'. Therefore, for each neighbor v of u there will be two such update entries in the list $U_i[v]$ for the change in cluster of u.

There are two advantages of using the above mentioned representation of the updates at any level. Firstly, from perspective of any vertex u, each update will be either insertion or deletion of an edge incident on it[2] and thus the description as well as analysis of the algorithm becomes simpler. Secondly, as we will see later, this representation will assist in getting a bound on the time complexity of the algorithm as well - the total processing time spent at a level will be of the order of the number of these updates reaching the level and the number of updates generated for the next level in their processing.

Upon a single edge insertion/deletion in the graph, we first add the corresponding updates at level 0 for the endpoints of the edge. Subsequently, this is how our fully dynamic algorithm processes the updates at a level i. Our algorithm scans the list L_i and for each vertex $x \in L_i$, it processes the updates of list $U_i[x]$ sequentially. This processing involves restoring the two invariants and updating

[2]except a change of $C_i[u]$ in case of change in clustering (which takes $O(1)$ time only)

the bucket structure of vertex x. This processing will in turn generate the updates for level $i+1$ (the list L_{i+1}, and array U_{i+1}). We now elaborate on processing of a single update at level i, and analyze the expected number of updates to be passed onto level $i+1$ in this processing. For maintenance of bucket structure, we shall use the following Theorem which we prove at the end of this section.

Theorem 4.3 *In processing one edge update for maintaining the bucket structure $B_i(u)$ at level i, the amortized number of updates introduced at level $i+1$ is at most $5+8\epsilon$ for any $\epsilon \leq \frac{1}{6}$.*

Handling an edge deletion. Consider an update from $U_i[u]$ which is deletion of an edge, say (u, v). The processing of this update by u is shown in Figure 2.

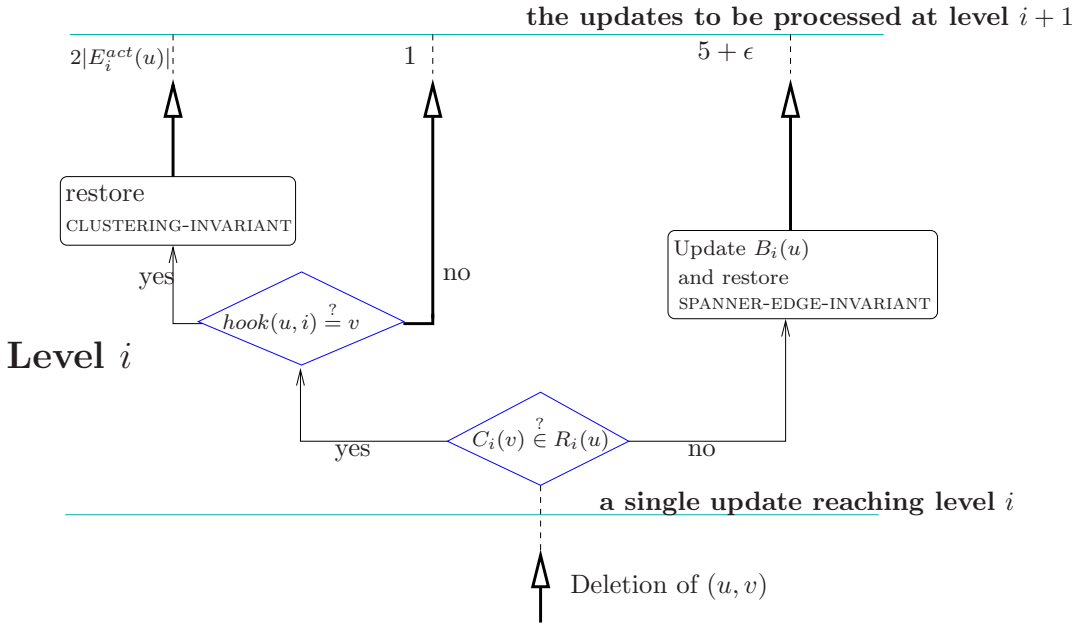

Figure 2: Handling deletion of an edge (u, v) by the endpoint u at level i

If $C_i(v)$ is a sampled cluster, then the deletion of (u, v) may violate the CLUSTERING-INVARIANT if $hook(u, i) = v$. Restoring the CLUSTERING-INVARIANT would require selecting another hook for u, changing the cluster of u at level $i+1$. This will also introduce two updates for each vertex at level $i+1$ which is a neighbor of u at level i. In this way, a total of $2|E_i^{act}(u)|$ new update entries will be generated in U_{i+1}. However, note that there is no update introduced due to bucket maintenance in this case because bucket structure of u stores edges incident from unsampled neighboring clusters only.

If $C_i(v)$ is not a sampled cluster, then this update is not going to affect the CLUSTERING-INVARIANT. In this case we just update the bucket structure $B_i(u)$ and restore SPANNER-EDGE-INVARIANT if needed. Using Theorem 4.3, the maintenance of bucket structure will introduce amortized $5+8\epsilon$ number of updates for level $i+1$ (to be stored in U_{i+1}).

It follows from Lemma 4.4 that the probability $hook(u, i) = (u, v)$ before deletion of (u, v) is bounded by $\frac{1+2\epsilon}{|E_i^{act}(u)|}$, and only in that case $2|E_i^{act}(u)|$ updates are generated for level $i+1$. Hence combining together the various cases and their associated probabilities, the expected number of updates introduced when vertex u processes deletion of an edge from $E_i(u)$ is

$$\frac{1+2\epsilon}{|E_i^{act}(u)|} 2|E_i^{act}(u)| + 5 + 8\epsilon \leq 7 + 12\epsilon$$

Handling an edge insertion. Consider an update from $U_i[u]$ which is insertion of an edge, say (u, v). Figure 3 describes the complete picture of how insertion of (u, v) is handled 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

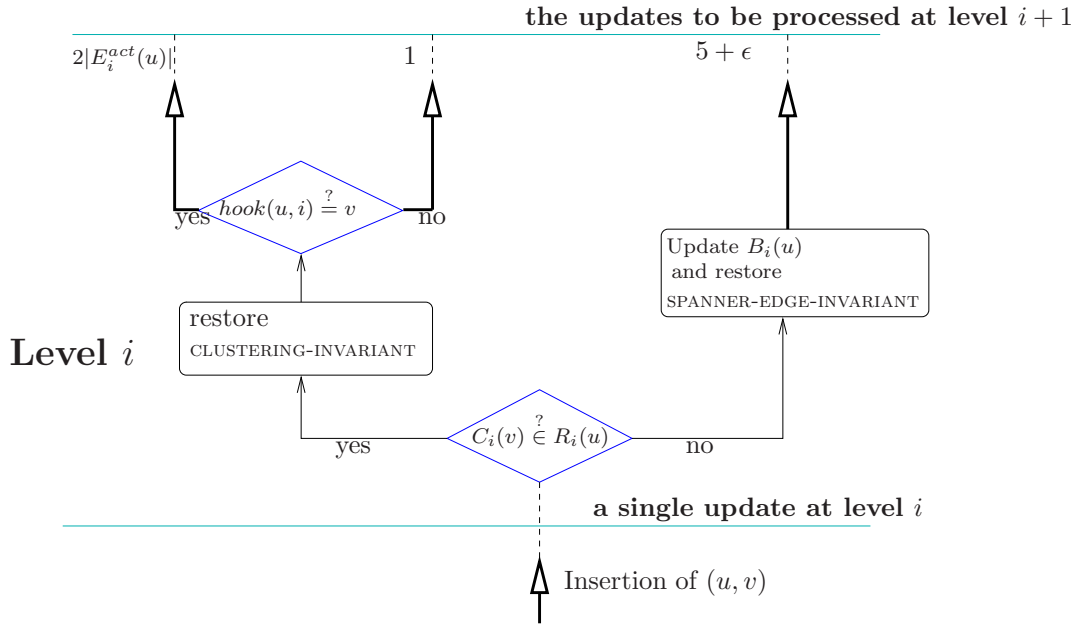

Figure 3: Handling insertion of an edge (u, v) by the endpoint u at level i

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 $2|E_i^{act}(u)|$ updates at level $i+1$ are introduced. Note that the bucket structure of u at level i remains intact in this case. If v belongs to sampled cluster but $hook(u, i)$ 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, then CLUSTERING-INVARIANT remains intact but the bucket structure of u has to handle the insertion and u may have to restore SPANNER-EDGE-INVARIANT if needed. Using Theorem 4.3, the maintenance of bucket structure will introduce amortized $5 + 8\epsilon$ number of updates for level $i + 1$.

To analyze the expected number of updates, we apply Lemma 4.4 after the insertion, it follows that $(u, v) = hook(u, i)$ with probability $(1 + 2\epsilon)/|E_i^{act}(u)|$, and only in that case at most $2|E_i^{act}(u)|$ updates would have to be passed to level $i+1$. Hence combining together various cases and their associated probabilities, the expected number of updates introduced when vertex u processes insertion of an edge into $E_i(u)$ is

$$\frac{1+2\epsilon}{|E_i^{act}(u)|} 2|E_i^{act}(u)| + 5 + 8\epsilon \leq 7 + 12\epsilon$$

By processing each update of $U_i[u]$ according to the procedures given above, each vertex u generates a list of possible updates for itself and its neighbors at level $i + 1$. There may be some superfluous updates in this list as shown at the end of this section. However, in the analysis above, we assumed that all updates generated are essential and hence get a worst case bound on the expected number of updates passed to level $i + 1$. So we can conclude the following Lemma.

Lemma 4.5 *While processing an update from $U_i[u]$ for maintaining the two invariants and the bucket structure of a vertex u, the expected number of updates introduced at level $i + 1$ is at most $7 + 12\epsilon$ for any $\epsilon > 0$.*

Remark. It is worth observing that the above Lemma does not assume anything about the clustering or subgraph at level below $i - 1$ or above level $i + 1$ leave alone the updates at those level. It gives a bound on the expected number of updates introduced at level $i + 1$ in terms of updates processed at level i.

Let Z_i be the random variable for the expected number of updates that reach level i upon a single edge update in the graph. Then using Lemma 4.5 it follows that

$$\begin{aligned}\mathbf{E}[Z_i] &= \sum_{\alpha} \mathbf{E}[Z_i | Z_{i-1} = \alpha]\mathbf{Pr}[Z_{i-1} = \alpha] \\ &= \sum_{\alpha} (7 + 12\epsilon)\mathbf{Pr}[Z_{i-1} = \alpha] = (7 + 12\epsilon)E[Z_{i-1}]\end{aligned}$$

At the topmost level, we just have to maintain SPANNER-EDGE-INVARIANT and there is no bucket structure needed. (Note that there is no need to keep bucket structure at level 0 either). Using the data structure given at end of section 2, recall that it takes $O(1)$ time to restore SPANNER-EDGE-INVARIANT for a violating neighboring cluster. So each update reaching the top level can be processed in worst case $O(1)$ time. At any other level, in addition to it, we maintain bucket structure and CLUSTERING-INVARIANT as well. For maintaining the bucket structure and restoring CLUSTERING-INVARIANT while processing an update, the processing cost is of the order of the number of updates generated at the next level. Hence we can conclude that 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 time to update the spanner upon a single edge update is of the order of $(7+12\epsilon)^k$ for any $\epsilon > 0$. Choosing $\epsilon < \frac{1}{12k}$, 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 $O(7^k)$ amortized time per update. The expected size of the spanner at any stage will be $O(k^7 n^{1+1/k} \log n)$.*

The expected update time can be improved further to $O(7^{k/2})$ by reducing the number of levels in the hierarchy of clusterings. Note that we used $k-1$ levels since at level $k-1$, the expected number of clusters is only $n^{1/k}$ and we can add 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 $c \in C_{k/2}, c' \in C_{k/2}$ 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 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, it can be seen that the number of levels at which we need to keep bucket structure will be reduced to $k/2-2$. Hence we can conclude 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 $O(7^{k/2})$ amortized time per update. The expected size of the spanner at any stage will be $O(k^7 n^{1+1/k} \log n)$.*

Analysis of dynamic bucket data structure.

First we briefly explain how our data structure at any level i as described at the end of section 2 (Preliminaries) 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 structure, it is easy to determine in $O(1)$ time if an edge incident on u at level i is active or inactive.

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 facilitate efficient dynamic maintenance, we shall use different activation and deactivation thresholds for a bucket.

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 j^{th} 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 touches $a\ell$; and an active bucket will be deactivated when its size falls to ℓ.

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. And 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 the next level $i + 1$. In this way, an edge update can trigger a huge number of updates to be passed to the next level. 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 [12]. 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. One unit of it will be consumed when the same update is passed to the next level for the first time. We call this the *registration cost* of the update. The remaining credit will be passed onto the cluster (in the bucket structure of u) to which v belong. Let the amount of this residual 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 (and the bucket to which the cluster will move in next hop) to pay for any possible change in the activation status of the bucket.

Clearly there is a hierarchical structure in the movement of credits: an edge update passes some credit 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. In addition, to each of these $(a-1)\ell$ new clusters some new edges might have got added ever since it joined the bucket. However, while activating the entire bucket, the activation cost for these new edges will be paid by their registration cost. 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 ith bucket.

Now let us analyze movement of a cluster from bucket j to $j+1$ (or $j-1$). When cluster had moved to bucket j, it had exactly α_j edges. If it moves to the next 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 that bucket, and transfer a credit of $\frac{g}{a-1}\alpha_j$ for deactivating the old bucket. In addition, if the 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 in addition to one credit for its registration cost. 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, and a credit of $\frac{g}{a-1}\alpha_j$ has to be paid to the old bucket for its deactivation. 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 in addition to one credit for its registration cost. Comparing π_i with π_d, and using $g > 1$ we can conclude the following Lemma.

Lemma 4.6 *A single edge insertion or deletion in the bucket structure of a vertex u leads to a change in the activation status of amortized $\frac{2g(a-1)+g^2+g}{(a-1)(g-1)}$ number of edges in $E_i(u)$.*

In other words, a single edge update in the bucket structure leads to insertion or deletion of amortized $\frac{2g(a-1)+g^2+g}{(a-1)(g-1)}$ edges in $\textsc{act}(u)$. Recall that whether an edge (u,v) belongs to E_i^{act} is determined by both u and v - the edge belongs to $E_i^{act}(u)$ if $(u,v) \in \textsc{act}(u)$ and $(u,v) \in \textsc{act}(v)$. So whether the deletion (or insertion) of an edge, say (u,v) to $\textsc{act}(u)$ will be reflected in E_i^{act} 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 data structure. In case the insertion (or deletion) of the edge in $\textsc{act}(u)$ implies insertion (or deletion) of the edge in E_i^{act}, it would be equivalent to two updates for level $i+1$ to be processed by each end point of the edge separately. Thus we may conclude the following lemma.

Lemma 4.7 *For each edge update processed by bucket structure of u at level i, the amortized number of updates to be sent to the level $i+1$ will be at most $1 + 2\frac{2g(a-1)+g^2+g}{(a-1)(g-1)}$.*

Choosing $g = 1/\epsilon$ and $a = g^2$ for suitably small value of ϵ, Lemma 4.7 leads to Theorem 4.3.

A note on the superfluous updates. Recall that the updates for level $i+1$ are generated as a result of the processing of the updates at level i. For each vertex $x \in L_i$, we process the updates $U_i[x]$ maintaining the two invariants for vertex x and the bucket structure. In this processing, let $\Delta_{i+1}(x)$ be the list of updates generated which are to be processed by x and its neighbors at level $i+1$. However, there may be many superfluous updates in $\Delta_{i+1}(x)$ as explained by the following two cases.

1. While handling the sequence of updates from $U_i[x]$ 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. The updates in $\Delta_{i+1}(x)$ corresponding to the intermediate changes in the activation status of the cluster are superfluous.

2. While handling updates from $U_i[x]$ which involve maintenance of CLUSTERING-INVARIANT, let x changes $hook(x,i)$ more than once. In this situation it is only the final $hook(x,i)$ which it selects after the entire processing of $U_i[x]$ 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.

In our algorithm, we avoid any distinction between a superfluous update and a normal/essential update. To avoid any inconsistency, we just have to impose the following rule on the processing of the updates by each concerned vertex.

Each vertex processes its updates in the order they are generated

To implement this rule, we distribute $\Delta_{i+1}(x)$ among the concerned vertices (x and neighbors of x at level $i+1$) in the order they are generated and each vertex v processes the updates $U_{i+1}[v]$ in the sequential order. In this manner, all the the superfluous updates will be taken care of automatically. For example, if there is a sequence of alternating (insertion and deletion) updates of an edge in $\Delta_{i+1}(x)$, the final update of the edge will survive. Similar assertion will hold in case a vertex changes its cluster multiple times.

5 Fully Dynamic Centralized Algorithm - II

We first present a decremental algorithm for maintaining a $(2k-1)$-spanner with expected $O(k^2 \log n)$ update time. Extending it to the fully dynamic environment, we obtain our second fully dynamic algorithm. This extension is achieved at the expense of increasing the size of spanner and update time by only a logarithmic factor.

At the core of the efficient decremental algorithm lies a new clustering which, with the help of some additional randomization, can be maintained efficiently under edge deletions. We describe this clustering in the following section.

5.1 New clustering

The new clustering is defined by a triplet (S, i, σ), where $S \subseteq V$ is the set of centers of the clusters, $i \in \mathbb{N}$ the radius of clusters, 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. The triplets (S, i, σ) defines clustering on the set of vertices of the graph which are within distance i from any vertex in S. For a vertex v of this set, the cluster center to which it belongs is the vertex $x \in S$ such that for every $y \in S \backslash \{x\}$, the following relation is satisfied

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

In other words, vertex v, in order to select its cluster center from S, gives first priority to their nearness, and in case there are multiple nearest vertices it prefers that vertex among them which appears first in the permutation σ. The uniqueness of the cluster center for v follows immediately. At any instance, the distance $\delta(v, S)$ will be termed as $d[v]$. As the edges are being deleted, the distance $d[v]$ and/or the cluster membership of a vertex v may change in $C(\sigma(S), i)$. We shall say that clustering $C(\sigma(S), i)$ has changed with respect to a vertex v due to an edge deletion if the deletion leads to either an increase in $d[v]$ or a change in its cluster membership (change in $C[v]$). For any fixed σ, there exists a sequence of edge deletions for which a vertex v may undergo $|S|i$ number of changes in clustering. However, we select the permutation σ randomly uniformly. As a result, the expected number of changes the clustering of v in $C(\sigma(S), i)$ will be quite small for any arbitrary sequence of edge deletions. The following Lemma formalizes this fact.

Lemma 5.1 *For an undirected unweighted graph (V, E) on n vertices and a set $S \subset V$, consider any arbitrary 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 with respect to v is $O(i \log n)$.*

Proof: Consider a vertex $v \in V$. It is easy to observe that distance $d[v]$ in the clustering $C(\sigma(S), i)$ can only increase during any sequence of edge deletion. Furthermore it can increase at most $i-1$ times before v leaves the clustering. We shall estimate the number of times a vertex changes its cluster (membership) within $C(\sigma(S), i)$ while keeping $d[v] = \ell$, for some fixed $\ell \leq i$. Consider the first time when $d[v]$ becomes ℓ. Choose any order in which the edges will be deleted from this moment onwards. Corresponding to this order, let $o_1, ..., o_t$ be the sequence of all the vertices of S at distance ℓ from v at present, and arranged in the chronological order of their cessation from being at distance ℓ from v. The number of times v changes its cluster while keeping $d[v] = \ell$ is the same as the number of vertices in this sequence whose clusters v joins during the period for which $d[v] = \ell$. The vertex v will join cluster centered at o_j if and only if o_j appears first among $\{o_j, ..., o_t\}$ in the permutation σ. Since σ is a uniformly random permutation of S, the probability of this event is $1/(t-j+1)$. Hence the expected number of cluster changes for the vertex v while remaining at a fixed distance ℓ from S is $\sum_{j=1}^{t} \frac{1}{t-j+1} = O(\log n)$. Since the vertex v may change $d[v]$ at most $i-1$ times before losing membership from the clustering $C(\sigma(S), i)$, the lemma follows. •

Though the above lemma proves that the expected number of changes in the clustering $C(\sigma(S), i)$ will be quite *small*, we need a data structure to detect these changes and to update the clustering efficiently.

Efficient construction and maintenance of the new clustering. First we state the following observation which follows immediately from the definition of clustering $C(\sigma(S), i)$.

Observation 5.1 *In order to compute the array $C[v]$ for 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., $\delta(x, S) = \delta(v, S) - 1$.*

The above mentioned observation suggests that the clustering $C(\sigma(S), i)$ can be constructed in the increasing order of distances of the vertices, as shown in **Algorithm 1**. A careful look at the algorithm can convince the reader that the clustering $\mathcal{C}(\sigma, i)$ and the forest $\mathcal{F}$ can be associated with a breadth first search (BFS) tree in an augmented 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 from g where we visit the neighbors of g in the order $\sigma(S)$. It is easy to see that the BFS tree we obtain consists of exactly the forest $\mathcal{F}$ and edges $\{(g, s) | s \in S\}$. A simple proof by induction on the distance from S shows that the array

Algorithm 1: Computing $C(\sigma(S), i)$. The queue Q is initialized to contain elements of S in the order defined by σ.

```
1  foreach v ∈ V do visited(v)← false;
2  foreach s ∈ S do C[s] = s;
3  F ← ∅;
4  while Q ≠ ∅ do
5      x ← Dequeue(q);
6      foreach (x, y) ∈ E do
7          if visited(y)=false then
8              visited(y) ← true;
9              C[y] ← C[x];
10             F ← F ∪ {(x, y)};
11             d[y] ← d[x] + 1;
12             if d[y] < i then Enqueue(y)
```

$C[\,]$ stores the clustering $C(\sigma(S), i)$. Moreover, the forest $\mathcal{F}$ spans each cluster by a tree rooted at

its center such that for each vertex $v \in \mathcal{C}(\sigma(S), i)$, there is a path in $\mathcal{F}$ of length $\delta(v, S)$ connecting v to $C[v]$.

Maintaining the clustering amounts to maintaining the BFS tree described above. An arbitrary BFS tree of depth i can be maintained in total $O(mi)$ cost over any sequence of edge deletions [24]. We shall tune this algorithm to maintain the clustering described above. In this algorithm, for each vertex $v \in V$, we shall maintain the following two additional fields as well. First field is $parent(v)$ which is the set of those vertices $w \in adj(v)$ with $d[w] = d[v] - 1$ and $C[w] = C[v]$. For $\mathcal{F}$, we may select one edge from $parents(v)$ for each v in the clustering. Another field maintained is $children(v)$, which is just inverse of $parent(v)$, that is, the set of vertices $w \in adj(v)$ with $v \in parents(w)$. It can be noted that each vertex in $parents(v)$ is a certificate for membership of v to its present cluster as well as for $d[v]$. In this manner the $parents$ field at each vertex serves as the foundation for the clustering. So we may have to process an edge deletion only if one endpoint of the edge is in the parent set of the other endpoint. Consider deletion of such an edge, say (u, v), with $u \in parents(v)$. In this case, we have to delete u from $parents(v)$ and v from $children(u)$. If $parents(v)$ is still non-empty, it implies that no change in the clustering is needed. However, if $parents(v)$ is empty, then the clustering of v (its cluster membership and, possibly $d[v]$) will change. It can be observed that this change may lead to changes in clustering of children of v as well. This wave of changes in clustering can go further down leading to a sort of chain reaction in some cases. **Algorithm 2** updates the clustering when edge (u, v) is deleted. It first detects in lines 1-5 whether clustering of v has to be updated. If so, it iterate the while loop with a queue $\mathcal{Q}_j$. In the first iteration $\mathcal{Q}_j$ consists of v only.

From the perspective of a vertex, say x, whose clustering has been changed due to the deletion of edge (u, v), the algorithm works as follows. Let the distance $d[x]$ prior to deletion of (u, v) was ℓ_{old}, and new distance be ℓ_{new}. The vertex x joins queue $\mathcal{Q}_{\ell_{old}}$. When dequeued from the queue $\mathcal{Q}_{\ell_{old}}$, the first task (lines 10-13) it does is to inform its children that its clustering information will change and so they better update their clustering too if necessary. It does so by deleting itself from their parents field. Following this point, the vertex x has its children field empty (recall its parent field was already empty in the beginning) and in the current iteration and each of the following iterations it just does the following. It determines whether it has reached its correct distance, that is, whether $d[x]$ is equal to $\delta(x, S)$. It does so by checking if it has some neighbor w with distance $d[w] = d[x] - 1$ (line 14). If it does not have such a neighbor, it concludes that the distance $d[x]$ is going to be more than its current value. So it increments $d[x]$, and enters $\mathcal{Q}_{j+1}$ - the queue to be processed in the following iteration. Once it $d[x]$ reached the $\delta(x, S)$, it scans its adjacency list to compute its new cluster-membership using Observation 5.1, computes $parents(x)$, and leaves the algorithm (lines 18-22). In this manner, the wave of updates originates from v and travels downward with $\mathcal{Q}_j$ as its wavefront. When this wave reaches a level l, some vertices leave it and get settled at the level l, while some new vertices may join it. The latter vertices are those which were at level l prior to the edge deletion whose clustering (cluster membership and possibly distance $d[]$ also) has got changed. Having given the description of the algorithm formally and from the perspective of a vertex, we have to prove that each vertex correctly updates clustering $C(\sigma(S), i)$ upon deletion of an edge. Let Δ_C^j be the set of those vertices x with $d[x] \leq j$ prior to the deletion of (u, v) whose clustering has changed. The following invariant holds in the beginning of each iteration of the while loop.

All those vertices of Δ_C^j whose new distance $d[]$ is $< j$ have their clustering updated correctly and queue $\mathcal{Q}_j$ consists of exactly the remaining vertices of Δ_C^j (whose new distance will be $\geq j$).

The above invariant, whose proof follows easily by induction on j, ensures the correctness of Algorithm 2. Let us analyze its time complexity. Suppose it performed t executions of the main while loop. Note that the time complexity of the algorithm is dominated by processing of the vertices of queues $\mathcal{Q}_{\ell(v)}, \ldots, \mathcal{Q}_{\ell(v)+t}$. It follows from the above invariant that only those vertices enter the queue $\mathcal{Q}_j$ in Algorithm 2 whose clustering (cluster membership or $d[]$) has changed due to the recent edge deletion. Each time a vertex x is enqueued or dequeued, it is charged $O(\deg(x))$ amount of work. Now it follows from Lemma 5.1 that vertex x will change its clustering expected $O(i \log n)$ times. So the expected processing cost charged to a vertex x for maintaining clustering $C(\sigma(S), i)$

Algorithm 2: Updating the clustering upon deletion of an edge (u, v), with $d[v] = d[u] + 1$.

```
1  j ← ℓ(v);
2  if v ∈ children(u) then
3      delete v from children(u);
4      delete u from parents(v);
5      if parents(v) = ∅ then add v to Q_j
6  while j ≤ i and Q_j ≠ ∅ do
7      Q_{j+1} ← ∅;
8      while Q_j ≠ ∅ do
9          x ⟵ Dequeue(Q_j);
10         for each y ∈ children(x) do
11             delete y from children(x);
12             delete x from parents(y);
13             if parents(y) = ∅ then add v to Q_{j+1}
14         if there is no neighbor w of x with d[w] = j − 1 then
15             d[x] ← d[x] + 1;
16             Enqueue(x,Q_{j+1});
17         else
18             compute C[x];
19             for each v ∈ adj(x) do
20                 if d[v] = j − 1 and C[v] = C[x] then
21                     add v to parents(x);
22                     add x to children(v);
23     j ← j + 1;
```

throughout any sequence of edge deletions is $O(i \deg(x) \log n)$. Thus the expected computational cost 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 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 $\mathcal{F}$ with amortized $O(i \log n)$ time per edge deletion.*

5.2 Decremental Algorithm

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 [7] as well as the first fully dynamic algorithm for $(2k-1)$-spanner. The hierarchy $\mathcal{H}_k$ and the set E of edges present in the graph at any stage define a hierarchy of subgraphs $G_i = (V_i, E_i), 0 \leq i \leq k$ in a natural way as follows.

1. G_0 is same as G, that is, $V_0 = V, E_0 = E$, and G_k is empty graph.
2. For $0 < i < k$, $G_i = (V_i, E_i)$ is defined in terms of G_{i-1} and S_i as follows.
   $V_i = \{v \in V_{i-1} | \delta_{E_{i-1}}(v, S_i) \leq i\}$, and
   $E_i = E_{i-1} \cap (V_i \times V_i)$

In simple words, V_i consists of those vertices in the graph G_{i-1} lying within distance i from S_i, and E_i is the set of edges from E_{i-1} with both endpoints present in V_i. It can be seen that G_i is a subgraph of G_{i-1} for each $i < k$. Let $h(v)$ for $v \in V$ be the highest $i \geq 0$ such that $v \in V_i$. In a similar manner $h(x, y)$ for $(x, y) \in E$ be the highest $i \geq 0$ such that $(x, y) \in E_i$.

As its randomization ingredient, our decremental algorithm construct a uniformly random permutation σ_i of vertices of set S_i for each $i < k$. This will be used to construct a hierarchy of new types of clusterings which have minimal dependencies across levels, and therefore, are efficient to

be maintained under deletion of edges. The algorithm maintains the subgraph (V_i, E_i), a clustering C_i and the following two invariants at each level $0 \leq i < k$ which will ensure a $(2k-1)$-spanner of expected size $O(n^{1+1/k} + kn)$ for the graph at each stage.

NEW-CLUSTERING-INVARIANT : Clustering C_i is the clustering $C(\sigma_i, i)$ of the set V_i. (Let $\mathcal{F}$ be the union of all edges from the spanning forests of these clusterings).

NEW-SPANNER-EDGE-INVARIANT : Every vertex v with $h(v) = i$, includes one edge from $E_i(v, c)$ in the spanner for each $c \in C_i$ neighboring to v.

Let E_S be the union of $\mathcal{F}$ and edges contributed according to NEW-SPANNER-EDGE-INVARIANT. The reader may note that NEW-CLUSTERING-INVARIANT is quite different from the CLUSTERING-INVARIANT used in the static algorithm described in [7] as well as the first fully dynamic algorithm. It is because the new clustering is quite different from the clustering used in the previous algorithms. However, combined with NEW-SPANNER-EDGE-INVARIANT, it still leads to sparse size of the spanner as will be evident from the following lemma.

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

Proof: First we shall show that E_S is a $(2k-1)$-spanner for the graph at each stage. Note that $E_0 = E$ and $E_k = \emptyset$, so it follows from hierarchy of the subgraphs that each edge (u, v) has $h(u, v) = j$, for some $j < k$. It thus follows that both u and v belong to C_j. Furthermore, at least one of u or v must have its highest level equal to j. Without loss of generality, let j be the highest level of u. So as part of NEW-SPANNER-EDGE-INVARIANT, vertex u must have contributed a spanner edge from the set $E_i(u, c)$, where c is the cluster of clustering C_j containing v. Now note that the clustering $C_j = C(\sigma_j, j)$ has radius j by construction. So it follows from Observation 2.1 that the proposition $\mathcal{P}_{2j+1}$ holds for edge (u, v). Since $j < k$, it follows that the set E_S is a $(2k-1)$-spanner. Let us analyze the size of this spanner. The collection of spanning forests $\mathcal{F}$ would contribute $O(nk)$ edges to the spanner. We shall show that any vertex $v \in V$ contributes expected $O(n^{1/k})$ edges to the set E_S. Let $S_i(v)$ be the set of vertices from S_i within distance $i+1$ from v in graph (V_i, E_i). The number of edge contributed by v is thus the set $|S_{h(v)}(v)|$. Note that both $h(v)$ as well as $S_{h(v)}$ are random variables. In order to get a bound on the expected size of $S_{h(v)}(v)$, it suffices to analyze the following randomized experiment. The experiment has a bag (for storing vertices) and a coin which gives HEADS with probability $p = n^{-1/k}$ independently in each toss. The experiment consists of at most k rounds where description of ith round is described in procedure Round given below.

Procedure `Round(`i`)`

Scan the vertices of S_i in the increasing order of distance from v in (V_i, E_i). For each vertex w scanned, query its distance from v, toss a coin, and proceed as follows;
if $\delta_i(v, w) > i + 1$ **then**
  stop the experiment and return the bag
else
  if HEADS **then**
    empty the bag and start $(i+1)$th round
  else
    add w to the bag

The set of vertices present in the bag at the end of the experiment is exactly the set $S_{h(v)}(v)$. Note that the number of vertices in the bag during a round is stochastically dominated by the number of coin tosses to get a HEAD. Hence $\mathbf{E}[|S_{h(v)}(v)|] = O(n^{1/k})$ and we are done. •

It thus follows from Lemma 5.2 that in order to maintain a $(2k-1)$-spanner, it suffices to maintain the hierarchy of subgraphs and clusterings, and the two invariants under deletion of edges.

The clustering $C(\sigma_i, i)$ at level i used by our decremental algorithm is such that it is solely defined by E_{i-1}, σ_i and has little influence of the clustering of previous levels. It is in contrast with the first fully dynamic algorithm where there was huge dependency between clusterings C_i and C_{i-1} - each cluster of C_i was a cluster of C_{i-1} with a few additional vertices hooking onto it from level $i-1$. The new clustering provides independence between clusterings of adjacent levels, and as a result their maintenance and the maintenance of the two invariants at each level becomes an easy task in dynamic scenario as will soon become clear from the following analysis.

As the edges are being deleted, the subgraphs and clustering at various levels in the hierarchy will change. Just like the first fully dynamic algorithm, it can be seen that a single edge deletion may cause deletion of many edges and vertices from higher levels. To observe this fact, note that a single edge deletion may potentially lead to the situation where one or more vertices cease to be members of a clustering at a level i. This happens exactly when their distance from S_i becomes greater than i. The highest level of these vertices decreases and they have to maintain SPANNER-EDGE-INVARIANT at their new levels. Let us now analyze, in full details, the processing done when handling an edge deletion.

Let us consider deletion of an edge, say (u, v). While processing this deletion, we perform updates in a bottom up fashion. We describe the processing to be done at level i. Among various structures at any level i, the following order in the dependencies can be easily observed. E_{i-1} and S_i define V_i and C_i. The set V_i defines E_i. We follow this order of dependency while performing updates. Let Δ_{i-1} be the set of edges deleted from E_{i-1} as a result of deletion of (u, v). Following the description of the new clustering, maintenance of V_i and E_i is done implicitly by our dynamic algorithm for maintaining $C(\sigma_i, i)$ - using the BFS tree like structure from S_i in the graph (V_{i-1}, E_{i-1}). It follows from Theorem 5.1 that it will take amortized $O(i \log n)$ time to maintain the clustering C_i (and V_i, E_i) for a single edge deletion in E_{i-1}. This takes care of our first invariant - NEW-CLUSTERING-INVARIANT. Now we need to restore the second invariant - NEW-SPANNER-EDGE-INVARIANT. For this task, there are two kinds of updates to be performed.

- The first kind of updates are caused by those edge deletions from E_i which do not lead to any change in the clustering C_i. For any such deletion, say deletion of (u, v), vertices u (and v) will have to restore NEW-SPANNER-EDGE-INVARIANT if $h(u) = i$ or $h(v) = i$. It takes $O(1)$ time per edge deletion for this task using the data structure mentioned earlier. There will be total $O(|E|)$ such kind of deletions at level i for any sequence of edge deletions. Hence the total update time of handling such edge deletions is $O(E)$.

- The second kind of updates are those which are caused by change in clustering C_i - a vertex x leaves its present cluster, say c, and joins another cluster, say c' within C_i or leaves the clustering C_i permanently. As discussed in earlier sections, from perspective of a neighbor y of x, such a change is equivalent to deletion of an edge from $E_i(y, c)$ and insertion of an edge in $E_i(y, c')$. Each neighbor y of x which has $h(y) = i$ will have to restore SPANNER-EDGE-INVARIANT for such update and it will take $O(1)$ time to accomplish it using the data structure mentioned earlier. Hence, it is easy to observe that a change in clustering of a vertex x at level i would lead to $O(\deg_i(x))$ operations to maintain SPANNER-EDGE-INVARIANT. However, if vertex x ceases to be member of C_i, there is (a one time) additional task of restoring SPANNER-EDGE-INVARIANT for $E_{i'}(x)$ where i' is the new level $h(x)$ of x after the deletion of (u, v).

It follows from Theorem 5.1 that a vertex can change its cluster in C_i expected $O(i \log n)$ number of times only. Hence the total time complexity of maintaining SPANNER-EDGE-INVARIANT at level $i < k$ is $O(m + mi \log n)$ which also matches the complexity of maintaining NEW-CLUSTERING-INVARIANT as well. Since there are total k levels, the total computational work 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 $\mathcal{D}(V, E)$ of size $O(|E|)$ such that for any arbitrary sequence of edge deletions, it maintains a $(2k-1)$-spanner of expected size $O(n^{1+1/k} + kn)$ and the total expected update time required is $O(k^2 m \log n)$.*

In other words, the amortized cost to maintain the spanner is expected $O(k^2 \log n)$ per edge deletion.

5.3 Extending the decremental algorithm to the fully dynamic environment

We shall now employ the decremental algorithm from Theorem 5.2 above 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 maintain a partition of the graph into $O(\log n)$ subgraphs, and concurrently maintain a decremental data-structure as defined in Theorem 5.2 for each subgraph. In order to handle edge insertions, these subgraphs are redefined periodically after certain intervals of updates and the corresponding data structure are rebuilt accordingly. We provide complete details of the algorithm below.

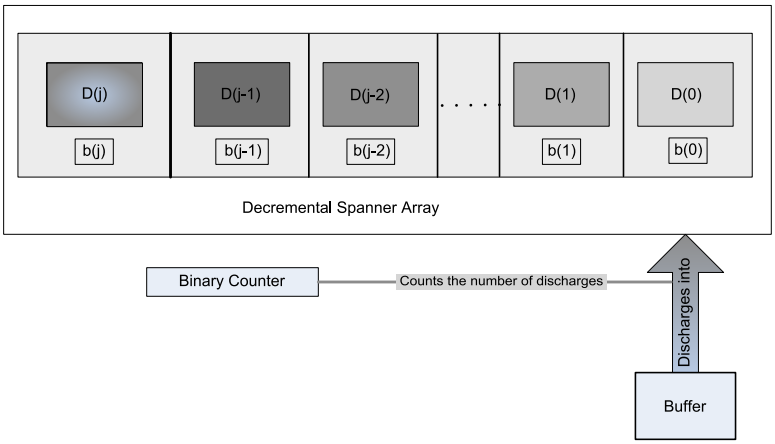

Figure 4: Data-structures used by the second fully dynamic algorithm

Let ℓ_0 be the greatest integer such that $2^{\ell_0} \leq n^{1+1/k}$. The algorithm will maintain the following objects at each stage.

1. A partition of edges E into subsets $E_0, ..., E_j$, $j = \log_2 n^{1-1/k}$ such that $|E_i| \leq 2^{i+\ell_0}$ holds for each i throughout the sequence of edge updates. Some of these subsets may be empty.
2. For each subset $E_i, i > 0$, the data-structure $D(i)$ from Theorem 5.2 which keep a $(2k-1)$-spanner $(V, \mathcal{E}_i)$ for the subgraph $G_i = (V, E_i)$.
3. A binary counter $\mathbf{C}$ which counts from 0 to $\frac{n(n-1)}{2}$, with the least significant bit at 0th place.
4. A hash table H to determine for an edge the subset E_i to which the edge belongs.

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. Deletion and insertion of edges are handled by the algorithm as detailed in Figure 5.3. It follows easily from the algorithm that $\{E_i | i \leq j\}$ is a partition of E at each stage. The data-structure $D(i)$ maintains a $(2k-1)$-spanner 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 the following Lemma.

Handling deletion of an edge

Let (u, v) be the edge which is deleted.

1. Let $(u, v) \in E_i$ (it takes $O(1)$ time to compute i using the hash table H).
2. The data-structure $D(i)$ handles the deletion of edge (u, v) and updates the spanner $\mathcal{E}_i$ accordingly.

Handling insertion of an edge

Let (u, v) be the edge which is inserted.

1. Increment the counter **C**. Let g be the largest integer such that gth bit of the counter **C** makes a transition (from 0 to 1 or vice verse).
2. **If** $g \leq \ell_0$
   add (u, v) to E_0 and $\mathcal{E}_0$.
   Else
   $h \longleftarrow g - \ell_0$;
   $E_h \longleftarrow \cup_{0 \leq i \leq h} E_i$;
   For $0 \leq i < h$ **do** $E_i \longleftarrow \emptyset$;
   Build $D(h)$.

Figure 5: Handling updates by the fully dynamic algorithm

Lemma 5.3 *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 $D(i)$ for each i. Consider any arbitrary sequence of μ edge updates. The data-structure $D(i)$ 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 5.3, this cost 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 $D(i)$ 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 $O(\log n)$ data-structures $D()$ used in the algorithm, it follows that expected update time for handling μ edge updates (insertion/deletion) is $O(\mu k^2 \log^2 n)$. In other words, amortized update time per edge insertion/deletion by the algorithm is expected $O(k^2 \log^2 n)$.

Theorem 5.3 *Given an integer $k > 1$ and a graph on n vertices undergoing edge insertions and deletions in any arbitrary way, there exists a fully dynamic algorithm which can maintain a $(2k-1)$-spanner of the graph such that amortized time to update the spanner is expected $O(k^2 \log^2 n)$ per edge insertion/deletion and the expected size of the spanner at each stage is $O((n^{1+1/k} + kn) \log n)$.*

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

In the distributed model of computation, each vertex corresponds to a node hosting a processor and a local memory, and each edge corresponds to a link through which messages can be sent. The computation proceeds in rounds. In a round, a processor sends and receives messages, and performs some local computation. In this manner, a given problem is solved in a distributed environment. There are two models of distributed computation. At first we consider the 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 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 asynchronous model, the transmission of messages takes place in an asynchronous manner. Each

processor on receiving a message does some local computation, and then transmits messages to the neighbors. However, the time spent in processing a message received may vary for different nodes as well as for different messages. The maximum time period between receiving a message and sending of the messages generated by the local computation on the messages received is called a single *cycle* in this model.

We shall now show that our first fully dynamic centralized algorithm (Theorem 4.4) adapts seamlessly to the distributed environment. The quiescence time complexity achieved is k and communication complexity per update is expected $O(7^k)$. This is an improvement over the previous best algorithm of Elkin [20] which has $3k$ quiescence time complexity and expected $O(mn^{-1/k})$ communication complexity.
It is expected that the reader fully understands the fully dynamic centralized algorithm I before studying its implementation in the distributed environment.

Hierarchy of sampled vertices and clusterings in distributed environment. The hierarchy $\mathcal{H}_k = \{S_0 = V, S_1, \ldots, S_k = \emptyset\}$ of subsets of vertices used by algorithm I can be formed in a distributed manner easily in the beginning due to the independence employed in the underlying random sampling. For each vertex $u \in V$, let $\max(u)$ 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 $j = \max(v)$. If $j > i$, the cluster centered at v (to which the vertex u belongs at level i) is a sampled cluster at level i. Let x be a neighbor of u at level i. If vertex x needs to know the clustering information of its neighbor u and whether the cluster of u is a sampled cluster at a level i, the information (v, j) can be communicated by u.

We now state the key properties of the centralized algorithm I which makes it easily adaptable in the distributed environment.
Key Properties

1. *Hierarchical structures.* The structures (clustering and subgraphs) maintained by our fully dynamic algorithm I at level $i+1$ are defined solely by the structures at level i only. As a result, the processing of updates by our fully dynamic centralized algorithm I is easily accomplished in a bottom up fashion in k iterations. During the ith iteration, the updates for level i are processed. Note that processing of an update by a vertex v at level i does not lead to any new update at level i. This processing can only generate updates for v and its neighbors at level $i + 1$ only. This acyclic feature of the updates, and the hierarchical structure facilitate processing of updates in a pipe-lined fashion in distributed environment.

2. *Local and atomic nature of updates.* In our centralized algorithm, each update reaching a level is atomic and vertex specific as elaborated in section 4.4. Taking vertex centric approach, each update at a level is essentially deletion or insertion of an edge. Such an update - deletion or insertion of an edge - at a level influences only the two endpoints of the edge. Moreover, processing of an update by a vertex, namely maintenance of the invariants and the bucket structure, is based on very local information - the clustering of neighbors of the vertex. This local and atomic nature of an update combined with the acyclic nature of updates mentioned above ensures that the processing of the updates reaching a level i can be carried out independently and concurrently by all the vertices at a level.

3. *The bound on the expected number of updates generated holds independent of the sequence of updates.* Suppose U_i be a set of updates to be processed at level i. Recall that the updates are processed by each vertex one at a time. Consider any update from U_i which is to be processed by some vertex, say v. This update will lead to expected 2 updates while restoring CLUSTERING-INVARIANT, and amortized at most 5 updates while restoring SPANNER-EDGE-INVARIANT and the bucket structure. These bounds are derived on the basis that the two invariants hold just before processing the update. So these bounds will hold regardless of the updates preceding or succeeding the update under consideration. Therefore, the expected

number of updates generated for level $i+1$ while processing U_i is at most $7|U_i|$ irrespective of what updates constitute the set U_i.

Remark. Though the expected number of updates generated at level $i+1$ is at most $7|U_i|$, the following subtle point needs to be understood here. The actual set of updates generated for level $i+1$ while processing U_i may differ depending upon the order in which the updates arrive for a vertex. This is essentially because of the bucket structure maintained by each vertex at a level. As an example, suppose there are some insertions and deletions of edges incident on a vertex v at level i. Furthermore, suppose the end points of these edges belong to same cluster $c \in C_i$, and the number of edge insertions is at least the same as the number of edges deleted. If the cluster c was on the verge of become inactive just before the sequence of these updates and all the deletions appear before all the insertions, then the activation status of c will change, and this will lead to 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.

In both synchronous and asynchronous version of the fully dynamic algorithm I, each vertex v will keep the following data locally for each level $i < k$.

- $E_i(v)$.
- the bucket structure for those edges from $E_i(v)$ which are incident from unsampled clusters.
- clustering information of its own and its neighbors.
- For each edge $(u,v) \in E_i(v)$, the vertex v also keeps the activation status of (u,v) in the bucket structure of u. While processing some update at level i, if activation status of the edge (u,v) in bucket structure of u changes, it communicates the same to v. This protocol helps in determining the changes in the set E_i^{act} and the corresponding updates for level $i+1$ as follows. At any moment, let activation status of (u,v) is ACTIVE in the bucket structure of u, but is INACTIVE in the bucket structure of v. Thus $(u,v) \notin E_i^{act}$, and hence $(u,v) \notin E_{i+1}$.
  - Suppose as a result of processing of some updates, the edge (u,v) becomes INACTIVE in the bucket structure of u as well. This change is communicated to v as well. This communication occurs immediately in the asynchronous environment, and in the beginning of the next round in the synchronous distributed environment. Both the vertices realize that this update does not lead to any change at level $i+1$ since (u,v) was already absent at level $i+1$. So no update at level $i+1$ is introduced in this case.
  - Suppose as a result of processing of some updates, the activation status of (u,v) becomes ACTIVE in the bucket structure of v. This change in communicated to v as well. As a result, both the vertices can realize that they need to process the update corresponding to insertion of (u,v) at level $i+1$.

  In our fully dynamic distributed algorithms for spanners, the update introduced at level $i+1$ due to change in the bucket structure of a vertex at level i will be handled according to the protocol described above.

6.1 Fully dynamic algorithm in synchronous distributed environment

The algorithm can be viewed as the k iterations of the dynamic centralized algorithm being executed concurrently with updates being processed in a pipe-lined fashion as shown in Figure 6.

Consider any round τ. let i_τ and d_τ be respectively the sets of edges inserted and deleted in the round τ. Along the lines of the centralized algorithm I, we define the following terminologies. Let $U_i(v)$ denote the updates for vertex v at level i received in the beginning of round τ, and let $\Delta_{i+1}(v)$ denote the updates generated for level $i+1$ as a result of the processing of the updates $U_i(v)$ by v. Now we describe the distributed computation performed in round τ. For level 0, each vertex v processes $i_\tau(v)$ insertions and $d_\tau(v)$ deletions. For any $0 < i < k$, the following activities are performed concurrently in round τ.

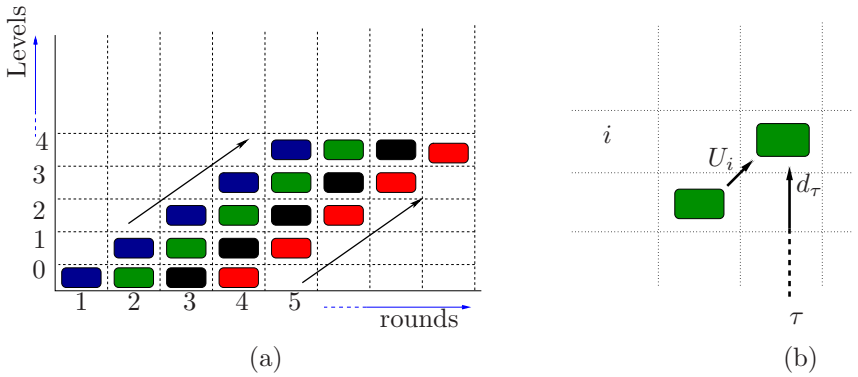

Figure 6: (a) pipeline execution of fully dynamic algorithm in synchronized distributed model (b) closer description of the updates processed at level i in round τ.

---

foreach $v \in V$ *and* $0 < i < k$ **do**
- v transmits the updates $\Delta_i(v)$ to the concerned neighbors;
- v, in turn, receives the updates $U_i(v)$;
- $U_i(v) \longleftarrow U_i(v) \cup (d_\tau(v) \cap E_i(v))$;
- v processes the updates $U_i(v)$ and generates the update $\Delta_{i+1}(v)$;

It is easy to follow from the above description that if all updates stop at the end of round α, then at the end of round $\alpha + k$, the spanner maintained is a $(2k-1)$-spanner of the graph present at that time. Thus quiescence time complexity of the algorithm is k.

The expected number of messages communicated in handling an edge insertion/deletion is of the order of the number of updates generated in the hierarchy. Insertion of an edge is processed starting from level 0, so it will lead to expected $O(7^k)$ updates as follows from Lemma 4.5. Deletion of an edge is processed concurrently at all the levels where the edge was present. So it will lead to $\sum_{i=0}^{k} 7^{k-i}$ updates which is still $O(7^k)$. Hence the expected number of messages communicated in handling an edge insertion/deletion is of the order of 7^k. The length of messages sent along an edge in a round is $O(\log n)$ bits. We can now conclude the following theorem.

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

6.2 Fully dynamic algorithm in asynchronous distributed environment

In the synchronous distributed algorithm, the entire computation proceeds in rounds of fixed duration in a very synchronized manner. However, in asynchronous model, it no longer holds - the time taken to process an update and then communicating messages may differ widely among processors. So all the updates originated by a single insertion/deletion of an edge do not reach a level at the same time. However, we shall show that this asynchronous nature does not cause any problem due to the three crucial properties of our fully dynamic centralized algorithm I mentioned above. Firstly, due to the hierarchical feature of our algorithm, it suffices to focus only on two consecutive levels of the entire hierarchy. Consider a level i. The vertices at ith level will receive updates U_i from level $i-1$ ordered according to the time of their creation. Each vertex will process these updates in the same order in which they are received. Due to acyclic and local nature of the updates in our first centralized algorithm, the processing of the updates by a vertex is unaffected by its neighbors, in particular by the variation in the processing speed of the neighbors. Moreover, as follows from 3rd property, the processing of a single update by a vertex at level i will lead to expected at most 7 updates at level $i+1$ irrespective of the order in which the updates arrive at level i. So, sooner

or later, whenever the processing of U_i ends, at that moment each vertex has restored its invariants and bucket structures and the expected total number of updates generated for level $i+1$ will be at most $7|U_i|$. So the only rule that needs to be followed while processing the updates in asynchronous environment is the following, which any way needs to be followed for sake of maintaining consistency. *Each vertex processes its updates in the order they are received and each vertex communicates the updates to its neighbors in the order they are generated by it.*

We now describe the distributed dynamic algorithm for $(2k-1)$-spanner in asynchronous model. In asynchronous model, we maintain the same hierarchy of subgraphs and clustering as maintained in the synchronous model. Each vertex will process the updates at a level in the order it receives them. Updates meant for different levels are processed concurrently by a vertex. The processing of an update at level i by a vertex v is described as follows.

---

foreach *update at level i received by $v \in V$* **do**
- v processes the update (maintaining bucket structure and invariants at level i);
- Let $\Delta_{i+1}(v)$ be the updates generated for level $i+1$ by this processing;
- v communicates each update from $\Delta_{i+1}(v)$ to the corresponding neighbor;

Similar to the synchronous distributed algorithm, insertion of an edge, say (u, v), is handled starting from level 0, whereas the deletion of an edge (u, v) will be handled concurrently by u and v at each level wherever it was present.

For getting a bound on the quiescence time complexity, it can be seen that after k cycles all the necessary updates in the spanner due to a single edge deletion or insertion are performed. The arguments for the bound on quiescence message complexity per update are the same as in the case of synchronous model. Hence we can state the following Theorem.

Theorem 6.2 *A $(2k-1)$-spanner of expected $O(n^{1+1/k}\operatorname{polylog} n)$ size can be maintained in asynchronous distributed environment with quiescence time $O(t)$ cycles and quiescence message complexity $O(7^k)$ per update. The worst case length of a message communicated along an edge in a cycle is $O(\log n)$ bits.*

To provide a better understanding, we would like to conclude with the following remark to highlight a subtle point in the fully dynamic asynchronous distributed algorithm for spanners.
Remark. Recall the protocol in the distributed environment for the updates at level $i+1$ which are generated by changes in the bucket structures at level i. In the asynchronous environment, there may be a change in the set of these updates due to variation in the processing time of the updates. This can be understood from the following example. Let $(u, v) \in E_i$ at a given moment. However, $(u, v) \in \text{ACT}(u)$ and $(u, v) \notin \text{ACT}(v)$. Hence (u, v) is not present in E_i^{act}, and hence is not present in E_{i+1}. Suppose some updates arrive at level i such that, at the end of processing these updates, $(u, v) \notin \text{ACT}(u)$ but $(u, v) \in \text{ACT}(v)$. So at the beginning as well as end of processing the updates, (u, v) does not belong to E_i^{act}, and hence not present at level $i+1$. However, if v processed its updates before u, then there will be a moment at which (u, v) will belong to $\text{ACT}(u)$ as well as $\text{ACT}(v)$, and hence to E_i^{act} at level i. This will introduce an update at level $i+1$ corresponding to the insertion of (u, v). Later on when u also finishes its updates, another update corresponding to deletion of (u, v) will be introduced at level $i+1$. On the other hand, if vertex u finishes the processing of its updates at level i before v, then no such update will be introduced. This is because at no instant in this time interval the edge (u, v) belongs to $\text{ACT}(u)$ as well as $\text{ACT}(v)$. Therefore, although the bound of 7 on the expected number of updates generated at level $i+1$ as a result of processing of a single update holds true, there may be a variation in the set of updates generated at level $i+1$ for a given U_i due to differences in the processing time taken by processors in asynchronous environment.

7 Improved fully dynamic algorithm for APASP in unweighted graphs

A simple consequence of the poly-logarithmic fully dynamic algorithm is the following observation.

Observation 7.1 *For each edge inserted/deleted into/from the graph, expected number of edges inserted to or deletion from a* $(2k-1)$*-spanner is* $O(\text{polylog}\, n)$.

This observation and our second fully dynamic algorithm for $(2k-1)$-spanner leads to an improved fully dynamic algorithm for the problem of maintaining all-pairs approximate shortest paths in unweighted graphs. In this problem, we receive an on-line sequence of edge insertions/deletions interspersed with query about exact/approximate distance between any two vertices in the graph. This problem has gained lot of attention in the past fifteen years. Camil Demetrescu and Italiano [15] presented an amortized $O(n^2\,\text{polylog}\, n)$ update time and $O(1)$ query time algorithm based on very novel ideas, and 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. For the simpler problem of undirected unweighted graphs, Roditty and Zwick [36] designed an algorithm which could ensure better update time at the expense of larger query time and introducing approximation in the distance reported. The algorithm achieves amortized $O(\frac{mn}{\epsilon t})$ update time and $O(t)$ query time for any $t < \sqrt{m}$, and for any pair of vertices the ratio of the distance reported to the actual distance is at most $O(1+\epsilon)$. The query time and update time are sub-linear and sub-quadratic respectively for sparse graphs. But for the dense graphs ($m = \Theta(n^2)$), the query time is linear and update time is quadratic. However, we can design a new fully dynamic algorithm which achieves sub-linear query time and sub-quadratic update time irrespective of whether the graph is dense/sparse. We achieve it at the expense of reporting approximate distance with larger stretch. The algorithm is the following : Maintain a $(2k-1)$-spanner using our fully dynamic algorithm and maintain the data structure of Roditty and Zwick [36] on the $(2k-1)$-spanner with $\epsilon = \frac{1}{2k}$. Each edge insertion and deletion is processed by our fully dynamic polylogarithmic algorithm which in turn passes the changes in the spanner to the latter data-structure. Using Theorem 5.3 and Observation 7.1, our fully dynamic algorithm for spanner takes polylogarithmic update time, and amortized number of edge updates in the spanner upon deletion/insertion of an edge in the graph is expected $O(\text{polylog}\, n)$. We can thus state the following theorem.

Theorem 7.1 *There exists a fully dynamic algorithm for* $2k$*-approximate distances in an undirected unweighted graph with* $O(n^{\frac{k+1}{2k}})$ *query time and expected* $O(n^{\frac{3k+1}{2k}}\,\text{polylog}\, n)$ *update time.*

For example, all-pairs 4-approximate distances can be maintained with $O(n^{3/4})$ query time and expected $O(n^{7/4}\,\text{polylog}\, n)$ update time, all-pairs 6-approximate distances can be maintained with $O(n^{2/3})$ query time and expected $O(n^{5/3}\,\text{polylog}\, n)$ update time. Previously there did not exist any such algorithm for fully dynamic all-pairs approximate distances which could simultaneously achieve sub-quadratic update time and sub-linear query time for any graph.

8 Conclusion and open problems

In this paper, 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. Like so many other instances in theoretical computer science, the simple randomization principles combined with new ideas have helped us achieve the impressive bounds of our algorithms. It would be interesting to explore whether similar results are achievable deterministically too. It would also be interesting to ged rid of the poly-logarithmic factor in the size $O(kn^{1+1/k}\,\text{polylog})$ of the $(2k-1)$-spanner maintained by the algorithms discussed in this paper.

As far as sparseness is concerned, the sparsest spanner would be a spanning tree. The existing static algorithms for spanners ensure that it is possible to construct an $O(n)$ size spanner with stretch $\log n$. However, none of these algorithms lead to a spanning tree with stretch $\log n$. In fact,

there are graphs for which such a spanning tree which guarantee $\log n$ worst case stretch don't even exist. As a simple example, consider a graph which is a cycle of length n. So one needs to *weaken* the notion of stretch slightly. In a very impressive result, Elkin et al. [22] 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 grateful to Michael Elkin for providing a copy of his work [19] which motivated them to extend their fully dynamic centralized algorithms 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. Newblock 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*, volume 3669 of *LNCS*, pages 532–543. Springer, 2005.

[4] 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.

[5] 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.

[6] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in expected $\mathrm{O}(n^2)$ time. *ACM Transactions on Algorithms (TALG)*, 2:557–577, 2006.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

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

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

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

[27] 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.

[28] 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.

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

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

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

[32] 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.

[33] L. Roditty. Fully dynamic geometric spanners. In *Proceedings of 23rd ACM Symposium on Computational Geometry (to appear)*, 2007.

[34] L. Roditty, M. Thorup, and U. Zwick. Roundtrip spanners and roundtrip routing in directed graphs, 2002.

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

[36] 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.

[37] 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.

[38] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput*, 37(5):1455–1471, 2008.

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

[40] 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.

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 $t = \ln \frac{1}{1-\epsilon}$.

$$\begin{aligned}
A(t) &= \mathbf{E}[\exp(-\sum_i tX_i)] \\
&= \mathbf{E}[\Pi_i \exp(-tX_i)] \\
&= \Pi_i \mathbf{E}[\exp(-tX_i)]
\end{aligned}$$

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

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

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

•