Applications of a Novel Sampling Technique to Fully Dynamic Graph Algorithms

by

Benjamin Mountjoy
B.Sc., University of Victoria, 2011

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisory Committee

Dr. Valerie King, Supervisor

(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member

(Department of Computer Science)

Supervisory Committee

Dr. Valerie King, Supervisor

(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member

(Department of Computer Science)

ABSTRACT

In this thesis we study the application of a novel sampling technique to building fully-dynamic randomized graph algorithms. We present the following results:

1. A randomized algorithm to estimate the size of a cut in an undirected graph $G = (V, E)$ where V is the set of nodes and E is the set of edges and $n = |V|$ and $m = |E|$. Our algorithm processes edge insertions and deletions in $O(\log^2 n)$ time. For a cut $(U, V \setminus U)$ of size K for any subset U of V, $|U| < |V|$ our algorithm returns an estimate x of the size of the cut satisfying $K/2 \leq x \leq 2K$ with high probability in $O(|U| \log n)$ time.

2. A randomized distributed algorithm for maintaining a spanning forest in a fully-dynamic synchronous network. Our algorithm maintains a spanning forest of a graph with n nodes, with worst case message complexity $\tilde{O}(n)$ per edge insertion or deletion where messages are of size $O(\text{polylog}(n))$. For each node v we require memory of size $\tilde{O}(degree(v))$ bits. This improves upon the best previous

algorithm with respect to worst case message complexity, given by Awerbuch, Cidon, and Kutten, which has an amortized message complexity of $O(n)$ and worst case message complexity of $O(n^2)$.

Contents

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Valerie King, for her dedication and patience. She has done a excellent job in introducing me to a variety of research topics and open problems of which I had very little knowledge when I started my degree. I would also like to thank her for her hard work in finding funding to support my research.

DEDICATION

I dedicate this thesis to Astrid, who has been a source of great support and who has patiently been a participant of many one-sided conversations over the last two years.

Chapter 1

Introduction

A *fully dynamic* graph algorithm is a data structure that maintains a property of a graph, that can process the insertion or deletion of an edge faster than the property can be re-computed from scratch. Fully dynamic graph algorithms have received considerable attention over the past couple of decades and are a natural extension of static graph algorithms for maintaining certain properties of a graph when updates to the graph are frequent. Input events to our data structures can be viewed as a sequence of insertions, deletions, and queries about the property being maintained, of which the data structure has no prior knowledge.

In this thesis, we study the application of a novel sampling technique in building two fully dynamic algorithms. The first, is a fully dynamic graph algorithm for estimating the size of a cut in an undirected graph. For this problem, our goal is to build a data structure so as to minimize the cost per operation. The second, is a fully dynamic algorithm for maintaining a spanning tree in a distributed synchronous network. For this problem, our goal is to minimize the number of messages required to rebuild the data structure after an insertion or deletion and to minimize the worst case memory required at a node.

Formally, our main contributions are:

1. A fully dynamic algorithm for estimating the size of a cut in an undirected graph $G = (V, E)$. The algorithm processes edge insertions and deletions in $O(\log^2 n)$ time and returns an estimate of the size of the cut $(U, V \setminus U)$ for a subset U of V where $|U| < |V|$ in $O(|U| \log n)$ time.

2. A randomized distributed algorithm for maintaining a spanning forest in a fully dynamic synchronous network. Our algorithm maintains a spanning forest of a graph with n nodes, with worst case message complexity $\tilde{O}(n)$ bits per edge insertion or deletion. For each node v we require memory of size $\tilde{O}(degree(v))$ bits. This improves upon the best previous algorithm with respect to worst message complexity, given by Awerbuch, Cidon, and Kutten [6], which has an amortized message complexity of $O(n)$ and worst case message complexity of $O(n^2)$. With respect to the worst case space requirement of a node this algorithm also improves upon [6] which has a worst space requirement of $O(n^2)$ per node.

The remainder of this thesis is organized as follows: In Chapter 2, we define our notation. In Chapter 3 we present related work. In Chapter 4, we present a fully dynamic algorithm for estimating the size of a cut in an undirected graph. In Chapter 5, we present a fully dynamic algorithm for maintaining a spanning tree in an synchronous network. In Chapter 6, we summarize our results.

Chapter 2

Definitions

Let $G = (V, E)$ be an undirected graph where $V = \{1, 2, ..., n\}$ is the set of nodes and E is the set of edges consisting of unordered pairs of nodes in V. We denote an undirected edge between the nodes x and y as $\{x, y\}$ and say that x and y are the endpoints of $\{x, y\}$. We say that the edge $\{x, y\}$ is *incident* to both x and y and that x and y are *adjacent* or *neighbors* in G. We use n to denote $|V|$ and m to denote $|E|$. The algorithms presented in this thesis are designed for undirected graphs.

A *path* of length k from a node x to a node y is a sequence $\langle v_0, v_1, ..., v_k \rangle$ of nodes such that $x = v_0$ and $y = v_k$ and $\{v_{i-1}, v_i\} \in E$ for $i = 1, .., k$. We denote a path that begins at a node x and ends at a node y as $x \rightsquigarrow y$. If there exist two nodes x and y such that there is no path $x \rightsquigarrow y$ then G is *disconnected*, otherwise G is *connected*. A *maximally connected component* of G is a maximal subset $C \subseteq V$ such that for every pair of nodes $x, y \in C$ there exists a path $x \rightsquigarrow y$. If there is a path $x \rightsquigarrow y$ in G we say the x and y are connected. Given a connected undirected graph $G = (V, E)$, a spanning tree $T = (V, E_T)$ of G, $E_T \subseteq E$, is a connected acyclic subgraph which contains all the nodes of G. If G is disconnected the collection of spanning trees of each maximally connected component of G is called a *spanning forest*. We refer to

any edge in E_T as a *tree edge*. An *update* to G is either the insertion or deletion of an edge. For any disjoint subsets U_1 and U_2 of V the *cutset* or *cut* between U_1 and U_2, denoted (U_1, U_2), is the set containing all edges with one endpoint in U_1 and one endpoint in U_2.

A *dynamic* graph algorithm is an algorithm that maintains a property of a graph and is able to recompute the property after an update in less time than can be done from scratch. A dynamic graph algorithm that only allows edge insertions is called an *incremental* graph algorithm. Incremental dynamic algorithms are *partially dynamic* graph algorithms. Dynamic graph algorithms that allow both edge insertions and edge deletions are called *fully dynamic* graph algorithms. In this thesis we only consider fully dynamic graph algorithms.

In the *synchronous* communication model computation proceeds in steps governed by a global clock where each step takes one time unit. During each time step a node can examine messages received from its neighbors, perform any required processing, and send messages to each of its neighbors. These messages are then available at their destination at the beginning of the next time step. In the *asynchronous* communication model there is no global clock and a sequence of messages sent from a node x to a node y across the edge $\{x, y\}$ will arrive in order at y some arbitrary but finite time later. A communications network whose topology is fixed is a *static network* and is a *dynamic network* otherwise. The *time complexity* of a distributed graph algorithm is the number of time steps the algorithm requires to complete its execution. The *message complexity* is the number of messages sent by the algorithm during its execution.

The algorithms presented here are randomized, which means their success is dependent on randomly generated bits. A *query* is a question to the algorithm about the property being maintained. The randomized algorithms presented in this thesis

are *Monte Carlo* type randomized algorithms which means their running time is deterministic but their query responses may be incorrect. Given a query with possible responses $\{no, yes\}$, we say an algorithm has a *one-sided* error if when it responds *yes* it is always correct and when is responds *no* it is wrong with some probability. We say a randomized algorithm is correct with *high probability* if the probability that any query response is incorrect is always less than $1/n^c$ for any constant c.

Chapter 3

Related Work

3.1 Review of Related Sequential Algorithms

For an undirected graph $G = (V, E)$, we are not aware of any previous fully dynamic algorithms that explicitly estimate the size of a cut $(U, V \setminus U)$ for any subset U of V, $|U| \leq |V|$.

However, research relating to computing and maintaining graph properties related to connectivity has received considerable attention over the years and continues to receive attention now. Henzinger [14] gives incremental algorithms for determining approximate and exact minimum cuts in undirected unweighted graphs. In 1994, Karger [17] introduced the concept of randomized sparsification to generate sparse graphs that closely approximate the minimum cut of the original graph. This technique was used to give an algorithm to estimated the size of the minimum cut of an undirected weighted graph within a $(1+\epsilon)$ multiplicative factor in $O(m + n \log^3 n/\epsilon)$ time and a fully dynamic algorithm to maintain a $O(\sqrt{1+2/\epsilon})$-approximation of the minimum cut with $\tilde{O}(n^\epsilon)$ time per update. In 1997, Eppstein et al. [10] used a sparsification technique to give fully dynamic algorithms for graph connectivity and

minimum spanning forests that processed updates in worst case $O(n^{1/2})$ time. In 1995, Henzinger and King gave the first fully dynamic connectivity algorithm with polylogarithmic expected update time of $O(log^3 n)$. This was later improved by Holm et al. [15], who gave deterministic fully dynamic algorithms for connectivity and maintaining a minimum spanning tree with an amortized cost of $O(\log^2 n)$ per update. Kapron, King, and Mountjoy [16] gave a fully dynamic connectivity algorithm with worst case time of $O(\log^4 n)$ time per insertion and $O(\log^5 n)$ time per deletion.

Recent work by Ahn et al. [3] gives a sparsification construction algorithm that uses a similar sampling technique to the sampling technique we use. Their work relies on the use of *graph sketches* that can be applied to distributed graph algorithms in a similar way that we have applied our sampling technique in Chapter 5. To provide an overview of their work, we introduce the concept of *graph sketches* and *dynamic graph streams* in the context of their work. A *dynamic graph stream* is a sequence of updates (insertions and deletions) to a graph. The position in the dynamic graph stream defines the state of the graph, specifically the edges belonging to the graph. A *graph sketch* is a linear projection of the graph defined by a dynamic graph stream, i.e. a compact representation of the graph from which relevant properties of the graph can be approximated. Linearity of the sketches allows sketches for multiple dynamic graph streams to be added together to form a sketch for the combined stream. As a consequence, these graph sketches are applicable to distributed algorithms where sketches representing graph streams at different nodes can be added or subtracted to represent sketches for different subnets of the network. The primary motivation for graph sketches is in processing dynamic graph streams representing large graphs using $O(n \cdot \text{polylog}(n))$ space, that without compression would require $O(n^2)$ space in memory.

Definition 3.1.1. *(*Ahn et al. *[2]) Given a graph $G = (V, E)$, the weighted sub-*

graph $H = (V, E')$ *is an* ϵ*-sparsification for* G *if*

$$\forall A \subset V, \qquad (1-\epsilon)\lambda_A(G) \leq \lambda_A(H) \leq (1+\epsilon)\lambda_A(G)$$

where λ_A *denotes the size of the cut* $(A, V \setminus A)$.

Ahn et al. [3] give a sketch based ϵ-sparsification construction algorithm over a dynamic graph stream that $(1+\epsilon)$ approximates all cuts and requires $O(n(\log^6 n + \epsilon^{-2}\log^5 n))$ space. A drawback to their algorithm is that the ϵ-sparsification can not be updated dynamically to reflect edge insertions and deletions. We note, the running time of their algorithm is not considered in their analysis.

3.2 Review of Related Distributed Algorithms

Distributed algorithms for dynamic networks have been studied extensively for decades. One of the primary challenges in distributed dynamic graph algorithms is designing an algorithm that can process single updates quicker than can be done from scratch. We assume unless otherwise stated that the messages sent by the following distributed graph algorithms are of size $O(\log n)$.

In 1979, Finn [11] was the first to show we could achieve loss-free and duplicate-free packet communication in a distributed network subject to asynchronous node and edge failures. His result relied on a distributed *resynch procedure* that could be used to bring all nodes of the network to a known state in the event that a link had failed or recovered. In 1983, Gallager et al. [12] gave a static distributed algorithm to construct a minimum spanning tree in an undirected graph $G = (V, E)$ with message complexity $O(m + n\log n)$ and time complexity of $O(n\log n)$. At this time there was no distributed algorithm for constructing a spanning tree in a dynamic network. This changed in 1987 when Afek et al. [1] introduced a distributed reset procedure,

RESET, for adapting static distributed algorithms for dynamic networks with message and time complexity of $O(m)$ and $O(n)$ respectively. The idea is to run RESET in conjunction with an existing static dynamic graph algorithm. If at anytime during its execution an edge is inserted or deleted, Reset "freezes" the nodes involved in the execution of the algorithm and returns the distributed data structure to the state it was in before the execution was started, purging all messages and undoing any other effects. This "blast-away" approach, as it is referred to by [6], represents one of the first forms of dynamic distributed graph algorithms. The goal was to build an efficient static dynamic graph algorithm and restart it every time there is an update to the graph. Although this approach is effective at converting static distributed graph algorithms to dynamic distributed graph algorithms, it does not reduce the message complexity of processing a single update below $O(m)$.

In 1988, Awerbuch et al. [7] improved upon the results of [1] giving a simulation technique called a *dynamic synchronizer* that achieves a local simulation of a global clock in a dynamic asynchronous network. Their results showed that any task performed on a static synchronous network can be performed as fast, up to a constant multiplicative factor, in a dynamic asynchronous network. Although their technique performs better than the "blast-away" technique Afek et al. [1] with respect to time complexity, it does not perform better with respect to message complexity, incurring an overhead of $O(m)$. It is interesting to note, other attempts at using past computation to improve algorithm performance actually performed worse than the "blast-away" approach of [1] ([6]) with respect to message complexity.

At this time, it was unknown if processing a single update required less communication than re-computation from scratch. It was not until 1990 that progress was made, when Awerbuch et al. [5] gave the first distributed algorithm for maintaining a spanning tree, where processing a single update required less communication

than re-computation from scratch. Awerbuch et al. [6] extended this work giving distributed spanning tree algorithm with amortized message complexity of $O(n)$ and time complexity of $O(n^2)$. The worst case message complexity of their algorithm is $O(n^2)$ and the worst case space requirement for a node is $O(n^2)$. In 1999, Kutten and Porat [19] further improved this result, reducing the amortized message complexity to $O(A)$ and the time complexity to $O(A \log^3 A)$, where A is the size of the connected component the algorithm is performed.

We note the distributed algorithms discussed so far are asynchronous.

More recent work on minimum spanning tree algorithms has ignored message complexity, and has focused on reducing time complexity. Each of [13, 18, 9] present distributed MST algorithms for synchronous networks, leaving adaptation to dynamic asynchronous networks to synchronizers such as [4] or [7]. Garay et al. [13] give the first sub-linear time distributed minimum spanning tree algorithm for an undirected graph with time complexity $O(Diam + n^{\epsilon} \log^* n)$, where $Diam$ is the diameter of the graph and $\epsilon = \ln 3 / \ln 6 \simeq 0.61$. Kutten and Peleg [18] later improved this result to $O(Diam + \sqrt{n} \log^* n)$ as a consequence of giving a faster distributed k-dominating set algorithm. This result was later improved in 2006 by Elkin [9], who gave a randomized distributed MST algorithm with time complexity $\tilde{O}(\mu(G, w), \sqrt{n})$ where $\mu(G, w)$ is a slightly more complicated metric called the MST-radius of the weighted undirected graph G. This improvement is somewhat conditional. The time complexity of [9] may be up to $\tilde{\Omega}(\sqrt{n})$ times faster than [18] and is never more than a polylogarithmic factor of n times slower depending on the network topology. It is noted, that the protocols [18] and [9] can be combined such that the resulting protocol is no more than twice as slow as the minimum of both.

Chapter 4

Cutset Size Estimation

Consider an undirected graph $G = (V, E)$. Given any partition of V into two sets we can label the sets U and $V \setminus U$ such that $|U| \leq |V \setminus U|$. Given any sequence of updates and queries of the form: "What is the size of the cut $(U, V \setminus U)$?" If the updates are independent of the query answers then there exists an algorithm that will return an estimate x that is within a factor of 2 times the cut size with high probability in $O(|U| \log n)$ time. The algorithm supports the following functions:

- Delete({x,y}): Delete an edge $\{x, y\}$ from E.
- Insert({x,y}): Insert an edge $\{x, y\}$ into E.
- Estimate(U): Estimate the size of the cut $(U, V \setminus U)$.

4.1 Description of the Algorithm

To illustrate the intuition behind the algorithm we present a simplified version first. Let K be the size of the cut $(U, V \setminus U)$ for any set $U \subset V$ such that $|U| < |V \setminus U|$. At each node $x \in V$ we maintain a bit $S(x)$ initially set to 0. For any subset U of V we define $S(U) = \bigoplus_{x \in U} S(x)$ to be the XOR of the bits $S(x)$ at each node $x \in U$. When

an edge $\{x, y\}$ is inserted into E we set $S(x) = S(x) \oplus 1$ and $S(y) = S(y) \oplus 1$ with probability 1/2. Notice that because $0 \oplus 0 = 0$ and $1 \oplus 1 = 0$ if every edge incident to a node in U has both endpoints in U then $S(U) = 0$ regardless of how the bits at each node are set. Intuitively, if the cut is empty then $S(U) = 0$ and if the cut is not empty then $S(U) \neq 0$ if and only if there an odd number of edges {x,y} in the cut such that $x \in U$ set $S(x) \leftarrow S(x) \oplus 1$. We denote the case when $K > 0$ and $S(U) = 0$ as a false positive. For example, if $K = 2$ then the probability of a false positive is 1/2.

To reduce the probability of false positives we replace the bit stored at each node $x \in V$ with bit vector $S(x) = S_1(x), ..., S_{c''}(x)$ of size $c'' = O(1)$. Similarly, we define $S_k(U) = \bigoplus_{x \in U} S_k(x)$ for $k = 1, ..., c''$. Now when an edge $\{x, y\}$ is added to E we *record* it at x and y as follows: for each k we set $S_k(x) \leftarrow S_k(x) \oplus 1$ and $S_k(y) \leftarrow S_k(y) \oplus 1$ independently with probability 1/2. We conclude that the cut is empty if and only if $S_k(U) = 0$ for every k. For example, if $K = 2$ the probability that we incorrectly conclude the cut is empty is $Pr(S_1(U) = 0) * Pr(S_2(U) = 0) * \cdots * Pr(S_{c''}(U) = 0) = 1/2^{c''}$. This allows us to effectively determine if the cut $(U, V \setminus U)$ is empty or not but does not allow us to accurately determine the size K of the cut.

To accommodate cuts of arbitrary size at each node x we introduce $\lceil \ln n + 2 \rceil$ levels i where $S_i(x)$ is the level i bit vector $S_{i,1}(x), ..., S_{i,c''}(x)$. When an edge $\{x, y\}$ is inserted into E for $i = 0, 1, ..., \lceil \ln n + 2 \rceil$ with probability $1/2^i$ we record $\{x, y\}$ at $S_i(x)$ and $S_i(y)$. The intuition behind the levels is as follows: for a cut of size $K \simeq 2^i$ there is a constant probability that no edge from the cut has been recorded at $S_{i+1}(U)$. That is, for any K there exists a smallest i such that $Pr(S_i(U) = 0) \geq T$ for some constant T, $0 < T < 1$ to be set later.

We would like to determine this smallest level i in order to approximate K. To do this we introduce $c \ln n$ versions j for each level i. Formally, at each node x in V we

maintain the table $S_{i,j,k}(U)$ for $i = 1, ..., \lceil \ln n + 2 \rceil$, $j = 1, ..., c \ln n$, and $k = 1, ..., c''$. We say $S_{i,j}(U) = \langle S_{i,j,1}(U), ..., S_{i,j,c''}(U) \rangle$ is the version j bit vector on level i. To insert an edge $\{x, y\}$ for each i, j with probability $1/2^i$ we record $\{x, y\}$ at $S_{i,j}(x)$ and $S_{i,j}(y)$ by setting $S_{i,j,k}(x) \leftarrow S_{i,j,k}(x) \oplus 1$ and $S_{i,j,k}(y) \leftarrow S_{i,j,k}(y) \oplus 1$ for each k independently with probability $1/2$. We say that a bit vector $S_{i,j}$ is a 0-vector if $S_{i,j,k} = 0$ for each k.

To estimate the size of the cut K we determine the smallest level i such that the fraction of 0-vectors on level i is greater than or equal to T. To track the bits in $S(x)$ and $S(y)$ that were set when $\{x, y\}$ was inserted we keep the table $A_{i,j,k}(\{x, y\})$. To delete $\{x, y\}$ we simply consult $A(\{x, y\})$ and unset any bits in $S(x)$ and $S(y)$ that were set when $\{x, y\}$ was inserted.

The code for INSERT, DELETE, and ESTIMATE is shown below.

Algorithm 1 RECORD$(S_{i,j}(x), S_{i,j}(y))$

for For $k = 1, ..., c''$ **do**
  With probability $1/2$
    Set $S_{i,j,k}(x) \leftarrow S_{i,j,k}(x) \oplus 1$, $S_{i,j,k}(y) \leftarrow S_{i,j,k}(y) \oplus 1$, and $A_{i,j,k}(\{x, y\}) \leftarrow 1$
end for

Algorithm 2 INSERT$(\{x, y\})$

Set $E \leftarrow E \cup \{x, y\}$
for For $i = 1, ..., c \ln n$, $j = 1, ..., \lceil \ln n + 2 \rceil$ **do**
  With probability $1/2^i$ call $Record(S_{i,j}(x), S_{i,j}(y))$
end for

Algorithm 3 DELETE$(\{x, y\})$

Set $E \leftarrow E \setminus \{x, y\}$
for each i, j, k **do**
  Set $S_{i,j,k}(x) \leftarrow S_{i,j,k}(x) \oplus A_{i,j,k}(\{x, y\})$ and $S_{i,j,k}(y) \leftarrow S_{i,j,k}(y) \oplus A_{i,j,k}(\{x, y\})$.
end for

Algorithm 4 ESTIMATE(U)

$T \leftarrow 0.3528$
Initialize $Count_i(U) = 0$ for $i = 1, ..., \ln n + 2$
for $i = 1, ..., \ln n + 2$ and $j = 1, ..., c \ln n$ **do**
  if for all k, $S_{i,j,k}(U) = 0$ **then**
    Set $Count_i(U) \leftarrow Count_i(U) + 1$
  end if
end for
Find the smallest i such that $Count_i \geq T(c \ln n)$
if $Count_1 = 0$ **then**
  return 0
else if $i = 1$ **then**
  return 2
else
  return $3 \cdot 2^{i-2}$
end if

4.2 Analysis

Theorem 4.2.1. *This algorithm requires $O((n+m)\log^2 n)$ bits.*

Proof. We maintain for each node x the table $S(x)$ of size $O(\log^2 n)$ bits and for each edge $\{x, y\}$ the table $A(\{x, y\})$ of size $O(\log^2 n)$ bits. □

Theorem 4.2.2. *Insert and delete operations have a running time of $O(\log^2 n)$.*

Proof. With each node x in V we associate $O(\log^2 n)$ bit vectors of size $O(1)$. When inserting or deleting an edge we update each bit vector at most once. □

Theorem 4.2.3. *The query time of the algorithm is $O(|U| \log n)$.*

Proof. To estimate the size of the cut we must compute the bitwise XOR of the $O(|U| \log^2 n)$ bit vectors at $S(x)$ for each node $x \in U$. Using words of size $O(\log n)$ we can pack $O(\log n)$ bit vectors into each word, therefore, we can compute $S(U)$ in $O(|U| \log n)$ operations. To find the smallest i such that $Count_i(U) > Tc \ln n$ we must visit each bit vector $S_{i,j}(U)$ at most once requiring $O(\log^2 n)$ time. □

Theorem 4.2.4. *Let $K > 0$ be the size of the cut $(U, V \setminus U)$ and suppose $K' > 0$ edges $\{x, y\}$ from the cut were recorded at $S_{i,j}(x)$ and $S_{i,j}(y)$ for a fixed i, j. Then the probability that $S_{i,j}(U)$ is a 0-vector is less than or equal to $0.75^{c''}$.*

Proof. Let $[K] = \{0, 1, ..., K\}$. Consider any bit $S_{i,j,k}(U)$ of $S_{i,j}(U)$. Then $Pr(S_{i,j,k}(U) = 0)$ is the probability that $S_{i,j,k}(U)$ was set to 1 an even number of times. Let S_E be the set of non-negative even integers less than or equal to K' and S_O be the set of positive odd integers less than or equal to K'. Let $even = \sum_{e \in S_E} \binom{K}{e}$ and $odd = \sum_{o \in S_O} \binom{K}{o}$. We will say that and term $\binom{K}{i}$ is even if i is even and odd if i odd. Then

$$Pr(S_{i,j,k}(U) = 0) = \sum_{k \in S_E} (1/2)^{K'} \binom{K'}{k} = \frac{even}{2^{K'}}.$$

Therefore, to bound $Pr(S_{i,j,k}(U) = 0)$ we need to bound $even$. To bound $even$ we break the analysis into three cases.

Case 1: K' is odd.

Then $even = 2^{n-1}$ because each even term in $\sum_{i=0}^{i \leq K} \binom{K'}{i}$ can be paired with an odd term of equal value as follows: $\binom{K'}{\lfloor K'/2 \rfloor - i} = \binom{K'}{\lfloor K'/2 \rfloor + i + 1}$ for $i = 0, ..., \lfloor K'/2 \rfloor$.

Case 2: K' is even and $K'/2$ is even.

Let $2x = 2^n - \binom{K'}{K'/2}$ and $y = \binom{K'}{K'/2}$. Note the sum of the terms in the multiset $I = \left\{ \binom{K'}{i} | i \in [K'] \setminus (K'/2) \right\}$ is $2x$. We can show that the sum of the even terms in I is less than x by pairing every even term with a larger odd terms as follows: $\binom{K'}{0} < \binom{K'}{1}, ..., \binom{K'}{K'/2-2} < \binom{K'}{K'/2-1}$ and $\binom{K'}{K'/2+1} > \binom{K'}{K'/2+2}, ..., \binom{K'}{K'-1} > \binom{K'}{K'}$. Therefore, because y is an even term $even < x + y = x + 2^n - 2x = 2^n - x$. Since $0 \leq \binom{K'}{K'/2} \leq 2^{n-1}$ we have that $2^{n-2} \leq x \leq 2^{n-1}$ which implies that $even < 3 \cdot 2^{n-2}$.

Case 3: K' is even and $K/2$ is odd.

Let x be the sum of the terms in the multiset $I_x = \{\binom{K'}{k} | i \in [K'] \setminus \{K'/2 - 1, K'/2, K'/2 + 1\}\}$ and $y = 2^n - x$. We can show that the sum of the even terms in

I_x is less than $x/2$ by pairing all even terms with a larger odd term as follows:

- $\binom{K'}{0} < \binom{K'}{1}$, $\binom{K'}{2} < \binom{K'}{3}$, ..., $\binom{K'}{K/2-1} < \binom{K}{K/2-2}$
- $\binom{K'}{K'/2+2} < \binom{K'}{K'/2+3}$, $\binom{K'}{K'/2+4} < \binom{K'}{K'/2+5}$, ..., $\binom{K'}{K'-1} < \binom{K'}{K'}$

We can show that the sum of the even terms in $[K] \setminus I_x$ is at most $2y/3$ because $\binom{K'}{K/2} > \binom{1}{2}\left[\binom{K}{K/2-1} + \binom{K'}{K'/2+1}\right]$ since $\binom{K'}{K'/2}$ is greater than $\binom{K'}{K'/2-1}$ and $\binom{K'}{K'/2+1}$. Therefore $even < x/2 + 2y/3 < (1/3)(x+y) < 2^n/3$.

It follows that regardless of the size of the cut $even < 3 \cdot 2^{n-2}$ and

$$Pr(S_{i,j,k}(U) = 0) \leq \left(\frac{3 \cdot 2^{n-2}}{2^n}\right) = 0.75.$$

Therefore, the probability that $S_{i,j}(U)$ is a 0-vector when $S_{i,j}(x)$ has been set at K' nodes $x \in U$ (equivalently, $S_{i,j}(U)$ is a false positive) is the probability that $S_{i,j,k}(U) = 0$ for every k which is $0.75^{c''}$. □

Lemma 4.2.5. *Let K be the size of the cut $(U, V \setminus U)$. Then* ESTIMATE(U) *returns an estimate x such that $K/2 \leq x \leq 2K$ if one of the following cases are true:*

1. *$2^i \leq K < 3 \cdot 2^{i-1}$ where $1 \leq i \leq \lfloor \log K \rfloor$ and all $Count_1(U), Count_2(U), ..., Count_{i-1}(U)$ are less than $Tc \ln n$ and either $Count_i(U)$ or $Count_{i+1}(U)$ are greater than or equal to $Tc \ln n$.*
2. *$3 \cdot 2^{i-1} \leq K < 2^{i+1}$ where $1 \leq i \leq \lfloor \log K \rfloor$ and all $Count_1(U), Count_2(U), ..., Count_i(U)$ are less than $Tc \ln n$ and either $Count_{i+1}(U)$ or $Count_{i+2}(U)$ are greater than $Tc \ln n$.*
3. *$K = 0$ and $Count_1(U) = 0$.*
4. *$K = 1$ and $Count_1(U) \geq Tc \ln n$.*

Proof. We handle each case separately.

Case 1. Let $l = i$ be the smallest value such that $Count_l(U) \geq Tc \ln n$. Then ESTIMATE(U) returns the estimate $x = 3 \cdot 2^{i-2}$. We have

$$K/2 < 3 \cdot 2^{i-2} \leq x < 3 \cdot 2^{i-1} < K.$$

Let $l = i+1$ be the smallest value such that $Count_l(U) \geq Tc \ln n$. Then ESTIMATE(U) returns the estimate $x = 3 \cdot 2^{i-1}$. We have

$$K < 3 \cdot 2^{i-1} \leq x < 2^{i+1} \leq 2K$$

Case 2. Let $l = i + 1$ be the smallest value such that $Count_l(U) \geq Tc \ln n$. Then ESTIMATE(U) returns the estimate $x = 3 \cdot 2^{i-1}$. We have

$$K/2 < 2^i < x \leq 3 \cdot 2^{i-1} \leq K$$

Let $l = i+2$ be the smallest value such that $Count_l(U) \geq Tc \ln n$. Then ESTIMATE(U) returns the estimate $x = 3 \cdot 2^i$. We have

$$K < 2^{i+1} < x \leq 3 \cdot 2^i \leq 2K$$

Case 3. $K = 0$ then $Count_1(U) = 0$ with probability 1 and ESTIMATE(U) returns 0.

Case 4. If $K = 1$ and $Count_1(U) > Tc \ln n$ then ESTIMATE(U) returns 2. □

Lemma 4.2.6. *Exactly one of cases 1-4 of Lemma 4.2.5 is true with high probability.*

Proof. The size K of the cut satisfies exactly one of $K = 0$, or $K = 1$, or $2^i \leq K < 3 \cdot 2^{i-1}$, or $3 \cdot 2^{i-1} \leq K < 2^i$ and therefore at most one case from Lemma 4.2.7 can

be true for any fixed K. For each case we compute bounds on the expected value of $Count_l(U)$ for the appropriate values of l and use Chernoff bounds from [20] to prove with high probability that these expected values do not deviate too far from expected. Before we can compute these expected values we need to bound the expected number of false positives on any level. Let F denote the fraction of false positives for any level i of $S(U)$. For $j = 1, ..., c \ln n$ let F_j equal 1 if $S_{ij}(U)$ is a false positive and 0 otherwise. Then

$$\begin{aligned} E[F] &= E[\sum_{i=1}^{c \ln n} F_i]/c \ln n \\ &= \sum_{i=1}^{c \ln n} E[F_i]/c \ln n. \end{aligned}$$

By Theorem 4.2.4 the probability that $S_{i,j}(U)$ for any fixed i, j is a false positive is at most $0.75^{c''}$ and therefore $E[F_i] \leq 0.75^{c''}$. Therefore

$$\begin{aligned} \sum_{i=1}^{c \ln n} E[F_i]/c \ln n &\leq \sum_{i=1}^{c \ln n} 0.75^{c''}/c \ln n \\ &= 0.75^{c''} \end{aligned}$$

For the remainder of the proof we assume that $c'' \geq 17$ and therefore $E[F] < 0.01$. Let E_i be the event that $Count_i(U) < Tc \ln n$ and $E_i^{'}$ be the event that $Count_i(U) \geq Tc \ln n$. We handle each case from Lemma 4.2.5 separately.

Case 1.

Let S_1 be the event that ESTIMATE(U) returns an estimate x such that $K/2 \leq$

$x \leq 2K$ when $2^i \leq K < 3 \cdot 2^{i-1}$. Then

$$\begin{aligned}
Pr(S_1) &= Pr(E_1 \cap E_2 \cap \cdots \cap E_{i-1} \cap (E_i^{'} \cup E_{i+1}^{'})) \\
&\geq Pr(E_1 \cap E_2 \cap \cdots \cap E_{i-1} \cap E_{i+1}^{'}) \\
&\qquad \text{since } E_{i+1}^{'} \subset (E_i^{'} \cup E_{i+1}^{'}) \\
&= Pr(E_1) \cap Pr(E_2) \cap \cdots \cap Pr(E_{i-1}) \cap Pr(E_{i+1}^{'}) \\
&\qquad \text{since each } E_i \text{ is independent} \\
&\geq Pr(Count_1(U) < Tc \ln n) * \cdots * Pr(Count_{i-1}(U) < Tc \ln n) * \\
&\quad Pr(Count_{i+1}(U) \geq Tc \ln n) \\
&\geq Pr(Count_{i-1}(U) < Tc \ln n)^{i-1} * Pr(Count_{i+1}(U) \geq Tc \ln n)
\end{aligned}$$

The expected value of $Count_i(U)$ is the expected number 0-vectors on level i. Let X_i denote the number of level i bit vectors $S_{i,j}(x)$ that were not set at any node $x \in U$. Then the $E[Count_i(U)] = E[X_i] + E[F]$. We compute an upper bound on $Count_{i-1}(U)$ as follows:

$$\begin{aligned}
E[Count_{i-1}(U)] &= E[X_{j-1}] + E[F] \\
&\leq \left(1 - \frac{1}{2^{j-1}}\right)^{2^j} c \ln n + 0.01 c \ln n \\
&\leq \exp\left\{-\frac{2^j}{2^{j-1}}\right\} c \ln n + 0.01 c \ln n \\
&= \exp\{-2\} c \ln n + 0.01 c \ln n \\
&\leq 0.1454 c \ln n
\end{aligned}$$

Next we compute a lower bound on $E[Count_{i+1}(U)]$. We consider the cases $i = 1$

and $i > 1$ separately. We have

$$\begin{aligned} E[Count_{i+1}(U)] &= E[X_{i+1}] + E[F] \\ &> E[X_{i+1}] \\ &\geq \left(1 - \frac{1}{2^{i+1}}\right)^{3 \cdot 2^{i-1} - 1} c \ln n \qquad \text{since } K \leq 3 \cdot 2^{i-1} - 1 \end{aligned}$$

If $i = 1$ then $E[Count_{i+1}(U)] \geq \left(1 - \frac{1}{4}\right)^2 \cdot c \ln n > 0.5625 c \ln n$. If $i > 1$ then

$$\begin{aligned} E[Count_{i+1}(U)] &> \left(1 - \frac{1}{2^{j+1}}\right)^{3 \cdot 2^{j-1} - 1} c \ln n \\ &\geq \exp\left\{-\frac{1}{2^{j+1}} - \frac{1}{2^{2j+2}}\right\}^{3 \cdot 2^{j-1} - 1} c \ln n \qquad \text{since } 1 - x \geq e^{-x - x^2}; x \leq 1/2 \\ &= \exp\left\{-\frac{3 \cdot 2^{j-1} - 1}{2^{j+1}} - \frac{3 \cdot 2^{j-1} - 1}{2^{2j+2}}\right\} c \ln n \\ &\geq 0.4950 c \ln n \end{aligned}$$

Therefore, $E[Count_{i+1}(U)] > 0.4950$.

Case 2.

Let S_2 be the event that ESTIMATE(U) returns an estimate x such that $K/2 \leq x \leq 2K$ when $3 \cdot 2^{i-1} \leq K < 2^{i+1}$. Then

$$
\begin{aligned}
Pr(S_2) &= Pr(E_1 \cap E_2 \cap \cdots \cap E_i \cap (E_{i+1}^{'} \cup E_{i+2}^{'})) \\
&\geq Pr(E_1 \cap E_2 \cap \cdots \cap E_i \cap E_{i+2}^{'}) \\
&\qquad \text{since } E_{i+2}^{'} \subset (E_{i+1}^{'} \cup E_{i+2}^{'}) \\
&= Pr(E_1) \cap Pr(E_2) \cap \cdots \cap Pr(E_i) \cap Pr(E_{i+2}^{'}) \\
&\qquad \text{since each } E_1 \text{ is independent} \\
&\geq Pr(Count_1(U) < Tc \ln n) * \cdots * Pr(Count_i(U) < Tc \ln n) * \\
&\quad Pr(Count_{i+2}(U) \geq Tc \ln n) \\
&\geq Pr(Count_i(U) < Tc \ln n)^i * Pr(Count_{i+2}(U) \geq Tc \ln n)
\end{aligned}
$$

We first compute an upper bound for $E[Count_i(\mathrm{U})]$.

$$
\begin{aligned}
E[Count_i(U)] &= E[X_i] + E[F] \\
&< \left(1 - \frac{1}{2^i}\right)^{3 \cdot 2^{i-1}} c \ln n + 0.01 c \ln n \qquad \text{since } K \geq 3 \cdot 2^{i-1} \\
&\leq \exp\left\{-\frac{3 \cdot 2^{i-1}}{2^i}\right\} c \ln n + 0.01 c \ln n \\
&= \exp\left\{-\frac{3}{2}\right\} c \ln n + 0.01 c \ln n \\
&\leq 0.2332 c \ln n
\end{aligned}
$$

Next, we compute a lower bound for $E[Count_{i+2}(\mathrm{U})]$. We consider the cases $i = 1$

and $i > 1$ separately. We have

$$\begin{aligned} E[Count_{i+2}(U)] &= E[X_{i+2}] + E[F] \\ &\geq E[X_{i+2}] \\ &\geq \left(1 - \frac{1}{2^{i+2}}\right)^{2^{i+1}-1} && \text{since } K < 2^{i+1} \end{aligned}$$

If $i = 1$ then $E[Count_{i+2}(U)] > \left(1 - \frac{1}{2^3}\right)^3 c \ln n \geq 0.6699 c \ln n$. If $i > 1$ then

$$\begin{aligned} E[Count_{i+2}(U)] &> \left(1 - \frac{1}{2^{j+2}}\right)^{2^{j+1}-1} c \ln n \\ &> \left(1 - \frac{1}{2^{j+2}}\right)^{2^{j+1}} c \ln n && \text{since } \left(1 - \frac{1}{2^{j+2}}\right) < 1 \\ &\geq \exp\left\{-\frac{1}{2^{j+2}} - \frac{1}{2^{2j+4}}\right\}^{2^{j+1}} c \ln n && \text{since } 1 - x \geq e^{-x-x^2}; x \leq 1/2 \\ &= \exp\left\{-\frac{2^{j+1}}{2^{j+2}} - \frac{2^{j+1}}{2^{2j+4}}\right\} c \ln n \\ &\geq 0.5878 \cdot c \ln n \end{aligned}$$

Case 3.

If $K = 0$ then $Count_i(U) = 0$ with probability 1 and ESTIMATE(U) returns 0.

Case 4.

Let S_3 be the even that $Count_1(U) > Tc \ln n$ when $K = 1$. Then $Pr(S_3) = Pr(E_1)$. We have

$$\begin{aligned} E[Count_1(U)] &= E[X_1] + E[F] \\ &\geq E[X_1] \\ &= \left(1 - \frac{1}{2}\right) \cdot c \ln n \\ &= \frac{c \ln n}{2} \end{aligned}$$

To guarantee that S_1, S_2, and S_3 succeed with probability greater than $1 - 1/n^{c'}$ it suffices to choose c such that $Pr(Count_i(U) \geq T) < 1/n^{c'+1}$ and the $Pr(Count_{i+2}(U) < T) < 1/n^{c'+1}$. Then by union bound we have the probability of failure at each level is bounded by $\lceil \ln n \rceil / n^{c'+1} < 1/n^{c'}$. We show using Chernoff bounds that setting $T = 0.3528$ minimizes the constant c required for the algorithm to succeed with high probability. By Chernoff bound (4.2) from [20] where $\mu = 0.2332c \ln n$ and $\delta = (T - 0.2332)/0.2332$ we have

$$Pr(Count_{i+1}(U) \geq Tc \ln n) \leq \exp\left\{ -\frac{(0.2332c \ln n)\left(\frac{T-0.2332}{0.2332}\right)^2}{3} \right\} \leq \frac{1}{n^{c'+1}}$$
$$\implies -\frac{(0.2332c \ln n)\left(\frac{T-0.2332}{0.2332}\right)^2}{3} \leq -(c'+1)\ln n.$$

By Chernoff bound (4.5) from [20] where $\mu = 0.4950c \ln n$ and $\delta = (0.4950 - T)/0.4950$ we have

$$Pr(Count_i(U) < Tc \ln n) \leq \exp\left\{ -\frac{(0.4950c \ln n)\left(\frac{0.4950-T}{0.4950}\right)^2}{2} \right\} \leq \frac{1}{n^{c'+1}}$$
$$\implies -\frac{(0.4950c \ln n)\left(\frac{0.4950-T}{0.4950}\right)^2}{2} \leq -(c'+1)\ln n$$

Solving the above equations for c we get

$$c \geq \frac{3(c'+1)}{0.2332(\frac{T-0.2332}{0.2332})^2} \tag{4.1}$$

and

$$c \geq \frac{2(c'+1)}{0.4950(\frac{T-0.4950}{0.4950})^2} \tag{4.2}$$

Setting equations 4.1 and 4.2 equal to each other and solving for T we get

$$T = \frac{0.4950 + (0.2332)\left(\sqrt{\frac{(2)(0.4950)}{(3)(0.2332)}}\right)}{\sqrt{\frac{(2)(0.4950)}{(3)(0.2332)}} + 1} = 0.3528$$

Substituting T into equations 4.1 and 4.2 we get $c \geq 49c'$. Therefore, by union bound setting $c \geq 49(c'+1)$ guarantees that exactly one case from Lemma 4.2.5 is true with probability at least $1 - 1/n^{c'}$. □

Theorem 4.2.7. *Let K be the size of the cut $(U, V \setminus U)$. Given a series of insertions, deletions and calls to Estimate(U); if the cut C(U, V-U) queried is chosen independently of the result of previous calls to* ESTIMATE *then* ESTIMATE(U) *returns an approximation x of $|C(U, V-U)|$ that satisfies $K/2 \leq x \leq 2K$ with probability at least $1 - 1/n^{c'}$ for any constant c'.*

Proof. The result follows from Lemma 4.2.5 and Lemma 4.2.6. □

Chapter 5

Maintaining a Spanning Forest in a Distributed Network

We begin this chapter by discussing previous joint work with Bruce Kapron and Valerie King [16] on a Monte Carlo type randomized algorithm for fully dynamic graph connectivity. Given an undirected graph $G = (V, E)$ the algorithm supports updates and queries of the following form:

- DELETE(e): Delete an edge e from E.
- INSERT(e): Insert an edge e into E.
- QUERY(x, y): Is there a path $x \rightsquigarrow y$ in G?

In Section 5.1 we present a sampling technique for finding an edge in a cut called the Cutset Data Structure. In Section 5.2 we discuss how the Cutset Data Structure can be extended to build a Monte Carlo type randomized algorithm for fully dynamic connectivity. Then, in section 5.3 we present our distributed algorithm for maintaining a spanning forest in a fully dynamic synchronous network.

5.1 Cutset Data Structure

The most complicated task in maintaining a spanning forest in a dynamic graph is finding a replacement edge when an edge in the spanning forest is deleted. Suppose a tree edge $\{x, y\}$ is deleted splitting the tree containing x and y in G into the tree T_x containing x and the tree T_y containing y. The challenge is then to find an edge with one endpoint in T_x and one in T_y. A naive approach might visit each edge incident to a node in T_x and determine if its other endpoint is in T_y. This approach is doomed to $o(n^2)$ worst case running time as $O(n^2)$ edges may have to be checked before finding a replacement.

We develop a Cutset Data Structure which relies on the observation that all possible replacement edges for $\{x, y\}$ have exactly one endpoint in T_x (and one in T_y) and all other edges have 0 or 2. To exploit this observation (1) each edge $\{x, y\}$ where $x < y$ is assigned a label $x_b \cdot y_b$ where x_b and y_b are the $\lceil \log n \rceil$ bit vectors containing the binary representation of x and y respectively and $x_b \cdot y_b$ is the $\lceil 2 \lg n \rceil$ bit vector formed by the concatenation of y_b to x_b; and (2) each node maintains the bitwise XOR of the labels of each of its incident edges.

Because the bitwise XOR of any binary vector with itself is 0, we know that for any tree T in F, if the cut $(T, V \setminus T)$ is empty, then the bitwise XOR of the values stored at each node in T is 0 and if it contains exactly 1 edge then the bitwise XOR of the values stored at each node in T will identify the edge in the cut.

To accommodate cuts of arbitrary size, at each node x we introduce $O(\log n)$ levels i where $S_i(x)$ is the level i bit vector at x. When an edge $\{x, y\}$ is inserted into G for $i = 0, ..., \lceil \log n \rceil$ with probability $1/2^i$ we *record* $\{x, y\}$ at $S_i(x)$ and $S_i(y)$ by setting $S_i(x) \leftarrow S_i(x) \oplus x_b \cdot y_b$ and $S_i(y) \leftarrow S_i(y) \oplus x_b \cdot y_b$. The intuition behind the levels is as follows: for a cut of size $K \simeq 2^i$ with constant probability exactly one edge $\{x, y\}$ from the cut will be recorded at its endpoints on level i and therefore the bitwise

XOR of the level i values stored at each node in the tree containing x (or the tree containing y) will reveal the label of $\{x, y\}$.

To extend the data structure to work with high probability for each level i we keep $c \ln n$ versions j. Formally, at each node x in V we maintain the table $S(x) = S_{i,j}(x)$ for $i = 0, 1, ..., \lceil \log n \rceil$ and $j = 1, ..., c \ln n$ where $S_{i,j}(x)$ is the bitwise XOR or all labels recorded at $S_{i,j}(x)$. When an edge $\{x, y\}$ is inserted into G, for each i, j with probability $1/2^i$ we record $\{x, y\}$ at $S_{i,j}(x)$ and $S_{i,j}(y)$.

To find a replacement edge in the cut $(T_x, V \setminus T_x)$ induced by the deletion of a spanning tree edge $\{x, y\}$, for each i, j we determine if the bit vector $S_{i,j}(T_x)$ reveals an edge in the cut. We show that if a replacement edge exists, there exists some i, j such that $S_{i,j}(T_x)$ reveals an edge in the cut with high probability.

Theorem 5.1.1. *If the size of the cutset $(U, V \setminus U)$ is $K > 0$ then there exists a bit vector $S_{i,j}(U)$ equal to the label of an edge in the cut with probability at least $1 - 1/n^{c'}$ for any constant c'.*

Proof. Let K be the size of the cut $(U, V \setminus U)$ and let $i = \lfloor \log K \rfloor$. Then it suffices to prove that with high probability there exists a bit vector $S_{i,j}(U)$ for some j on level i such that $S_{i,j}(U)$ reveals an edge in the cut. Let $E'_{\{x,y\}}$ be the event that for any fixed j, $S_{i,j}(U) = x_b \cdot y_b$ for an edge $\{x, y\}$ in the cut. The probability of $E'_{\{x,y\}}$ is equal to the probability that $\{x, y\}$ is sampled at $S_{i,j}$ multiplied by the probability that remaining $K - 1$ edges in the cut were not. Therefore

$$Pr\left(E'_{\{x,y\}}\right) = \left(\frac{1}{2^i}\right)\left(\frac{2^i - 1}{2^i}\right)^{K-1}.$$

Let E_j for $j = 1, ..., c \ln n)$ be the event that $S_{i,j}(U) = x_b \cdot y_b$ for some edge $\{x, y\}$ in the cut. Then $Pr(E_j) = \sum_{\{\{x,y\} \in C\}} Pr(E'_{\{x,y\}})$ because for any two edges e_i and e_j in the cut, E'_{e_i} and E'_{e_j} are mutually exclusive.

Now we can determine a lower bound for E_i. First note if $K = 1$ and $\{x, y\}$ is the only edge in C then $Pr(E_j) = Pr\left(E'_{\{x,y\}}\right) = 1$ because for each j, $\{x, y\}$ is added to $S_{0,j}(U)$ with probability 1. If $K > 1$ then

$$\begin{aligned}
Pr(E_j) &= \left(\frac{K}{2^i}\right)\left(1 - \frac{1}{2^i}\right)^{K-1} \\
&> \left(1 - \frac{1}{2^i}\right)^{K-1} && \text{since } 1 \leq K/2^i < 2 \\
&> \left(1 - \frac{1}{2^i}\right)^{2^{i+1}} && \text{since } 2^i \leq K < 2^{i+1} \\
&\geq (1/4)^2 && \text{since } x \geq 2 \implies \left(1 - \frac{1}{x}\right)^x \geq \frac{1}{4}.
\end{aligned}$$

Let Y be the event that for $j = 1, ..., c \ln n$ no label $S_{i,j}(U)$ identifies an edge in the cut, that is, that E_j fails for each j. Then the probability of Y is the probability that E_j fails for every j. Because $Pr(E_j) > 1/4^2$ we have $Pr(1 - E_j) < (1 - 1/4^2)$. Therefore,

$$\begin{aligned}
Pr(Y) &= \prod_{i=1}^{c' \ln n} (1 - Pr(S_i)) \\
&< \left(1 - \frac{1}{4^2}\right)^{c \ln n} \\
&< \left(e^{-\frac{1}{4^2}}\right)^{c \ln n} && \text{since } 1 - x < e^{-x} \\
&< \left(e^{\ln n}\right)^{-c/4^2} \\
&= n^{-c/4^2}.
\end{aligned}$$

Setting $c = 16c'$ we have for any constant c', $Pr(Y) < 1/n^{c'}$. □

5.2 Fully Dynamic Connectivity

Now we wish to develop a fully dynamic connectivity algorithm using the Cutset Data Structure described in Section 5.1. The algorithm attempts to maintain a spanning forest F of an undirected graph $G = (V, E)$. Consider the following naive approach:

- **To insert an edge** $\{x, y\}$. Add $\{x, y\}$ to E. If $\{x, y\}$ are not connected in F then add $\{x, y\}$ to F.
- **To delete an edge** $\{x, y\}$. Remove $\{x, y\}$ from E. If $\{x, y\}$ is in F then, remove $\{x, y\}$ from F and use the Cutset Data Structure to find a replacement edge $\{v, w\}$ connecting x and y in F. If a valid replacement $\{v, w\}$ is found then add it to F.

Unfortunately, this does not solve dynamic connectivity. When a tree edge is deleted, we would query the cutset induced by one component of the broken tree and use it to find a replacement edge. This may not work because the probabilistic analysis we present will be erroneous if the cutset queries and the edge updates are correlated with the random bits used by the algorithm. For example, we cannot use the random bits used to form T to search for a replacement edge in the cut $(T, V \setminus T)$.

To ensure the random bits used to find a replacement edge are independent of the structure of the component queried to find a replacement edge, F is constructed in $tiers$ $0, 1, ..., top$ where $top = \lceil \log n \rceil$. On each tier t a Cutset Data Structure $CutsetDS_t$ and forest F_t are maintained such that the random bits used by $CutsetDS_t$ are independent of random bits used by Cutset Data Structures on other tiers and the structure of F_t is dependent only on the random bits of Cutset Data Structures on lower tiers.

To insert an edge $\{x, y\}$ we add $\{x, y\}$ to E and if x and y are not connected in F_{top} we add $\{x, y\}$ to F_t for all $t > 0$. When an edge is deleted we use the Cutset Set

Data structures to find replacement edges. The Cutset Data Structures on each tier follow the algorithm described in Section 5.1 to find edges crossing the cuts induced by trees in the forest on that tier. Thus, on tier 0 are the nodes in V, and on tier t is the forest F_t formed by merging trees in F_{t-1} with tier t edges found by $CutsetDS_{t-1}$. A tree in F_t is called a tier t tree and an edge found by $CutsetDS_{t-1}$ connecting two trees in F_{t-1} is called a tier t tree edge. A tree T on tier t is called *unmatched* if it is not maximal and is not connected to another tier t tree edge by an edge on tier $t+1$, otherwise, T is *matched*.

To ensure $\lceil \log n \rceil$ tiers suffice in guaranteeing that F_{top} is a spanning forest of G the algorithm maintains the invariant that each non-maximal tree in F_{t-1} is matched. When a tier t edge is deleted this invariant may be violated as one or two trees in $F_{t'}$, $t' < t$ may become unmatched. Let T be one of the tier t' trees that has become unmatched, then T must be matched with another tier t' tree inorder to restore the invariant. To restore this invariant, $CutsetDS_{t'}$ is queried to find a tier $t'+1$ edge $\{x, y\}$ connecting T to another level $t'+1$ tree. However, inserting $\{x, y\}$ into $F_{t'+1}$ may cause a cycle in $F_{t''}$ where $t'' > t$ if there is already a path $x \rightsquigarrow y$ in $F_{t''}$.

This cycle is broken by removing the tier t'' edge in the cycle. This may cause a component on a higher tier to become unmatched, therefore, requiring this procedure to be repeated at most *top* times until there exist no unmatched component containing T. To summarize the following invariants are maintained:

1. The tier 0 trees are the nodes of G.

2. On each tier t, $F_t \subset F_{t+1}$.

3. Every tree F_t is joined to at least one other tree in F_t by a tier $t+1$ tree edge unless F_t is a spanning tree of a maximally connected component.

4. The structure of the forest on tier t is independent of the random bits of tiers

t and higher.

5. F_{top} is a spanning tree.

The code for INSERT, DELETE, and RECONNECT are shown below.

Algorithm 5 INSERT($\{x, y\}$)

```
1: insert {x, y} into the cutset data structure on each tier
2: if {x, y} connects two unconnected trees in F_top then
3:    add {x, y} to F_t for all t > 0
4: end if
```

Algorithm 6 DELETE($\{x, y\}$)

```
1: delete {x, y} from all trees containing it.
2: for u ∈ {x, y} do
3:    while there exists an unmatched tree containing u do
4:       A ← the lowest unmatched containing u
5:       k ← (tier of A)
6:       Reconnect(A, k)
7:    end while
8: end for
```

Algorithm 7 RECONNECT(A, k)

```
1: e = {v, w} ← search(A, S^k) (assume that v is the endpoint of e in A)
2: if e = null then
3:    Mark A as maximal
4: else
5:    if there is a path from v to w in F_top then
6:       e' ← an edge of maximum tier on the path between v and w.
7:       Remove e' from all F_t that contain it
8:    end if
9:    Add e to F_k' for all k' > k
10: end if
```

5.3 Maintaining a Spanning Forest in a Distributed Network

In this section we present a Monte Carlo type randomized algorithm for maintaining a spanning forest in a fully dynamic synchronous network. We model the network with the undirected graph $G = (V, E)$. Our algorithm maintains a spanning forest F of G with worst case message complexity of $\tilde{O}(n)$ bits per update. For each node v we require memory of size $\tilde{O}(degree(v))$ bits. This work extends the work of Kapron, King, and Mountjoy [16] to the distributed setting. Our algorithm supports the following operations:

- INSERT($\{x, y\}$): Insert the edge $\{x, y\}$ to E.
- DELETE($\{x, y\}$): Delete the edge $\{x, y\}$ from E.

We assume that updates are made by an oblivious adversary that are independent of the random bits used by the algorithm, that all messages sent are of size $O(\text{polylog}(n))$, and that there is enough time between each update for the algorithm to perform all required processing. Each node knows the number of nodes in the graph. Initially the network starts with no edges. Input events at a node are as follows: (1) the deletion of an incident edge, (2) the insertion of an incident edge, and (3) the reception of a message. An edge that has failed and not recovered cannot send messages. We assume that whenever an edge fails or recovers a lower-layer link protocol is in place that notifies each endpoint of the edge.

Each node in V marks a subset of its incident edges as tree edges and associates with each marked edge a tier t. As described in Section 5.2 we define a hierarchy of forests F_t for all t, $0 \leq t \leq top$ such that the distributed forest F_t is the union of marked edges of tier $\leq t$ for each node in V. Each tree in $F = F_{top}$ is rooted. To

track which tree in F it belongs each node x maintains the label $root(x)$ identifying the root of tree in F containing it. Initially each node x sets $root(x) \leftarrow x$. Each node x in V maintains for each tier t a Cutset Data Structure $CutsetDS_t$ comprised of $S^t(x)$ and $A^t_x(\{x, y\})$ as described in Section 5.1. We refer to the tables $S^t(x)$ and $A^t_x(\{x, y\})$ on each tier as the *local tables* of x. We let $A_x(\{x, y\})$ denote the set $\{A^0_x(\{x, y\}), ..., A^{top}_x(\{x, y\})\}$.

5.3.1 Subroutines

In this section we define a set of distributed routines required to perform some computation over a subset of a tier t component. These routines use a simple distributed protocol, outlined in [8], for communicating over a tier t component: A distributed routine is started at a node x which invokes the routine in adjacent nodes reachable by tree edges of tier at most t by sending a message. Each node that receives a message begins the distributed routine locally and repeats the process of invoking the routine in its neighbors reachable by tree edges of tier at most t until the routine is invoked in a *terminating* node. When the distributed routine reaches a terminating node it signals its completion by sending ACK to its sender, possibly performing some local computation and returning additional information. A non-terminating node will send an ACK only after having received ACKs from all nodes it invoked the procedure in. A *leaf node* in the calling sequence is a node that has no neighbors reachable by tree edges in which to invoke the distributed routine. A leaf node is always a terminating node but a terminating node is not always a leaf node. If a node x invokes a distributed routine in a node y we say that x is the *parent* of y and y is the *child* of x in the calling sequence. Each of the following subroutines rely on this simple communication protocol.

- COUNT(t) at tier t started at a node x determines the size of the connected

component containing x in F_t.

- PATH(y, t) at tier t started at a node x returns "yes" if there is a path from x to y in F_t and "no" otherwise.
- FIND(y) started at a node x determines the highest tier edge on the path from x to y in F_{top}.
- UNMARK$(\{v, w\})$ started at a node x coordinates the unmarking of $\{v, w\}$ as a tree edge at v and w in F_{top}.
- UPDATE(t) at tier t started at a node x in a tree T in F_t computes $S = \bigoplus_{x \in T} S^t(x)$. S is used to find a replacement edge in the cut $(T, V \setminus T)$.
- MARK$(\{x, y\}, t)$ at tier t started at a node x handles the addition of $\{x, y\}$ to F_t by coordinating the marking of $\{x, y\}$ as a tier t tree edge at x and y.
- ROOTCHANGE(y) at tier *top* started at a node x notifies the tree containing x in F that its new root is y.
- ISVALID$(\{v, w\})$ at tier *top* started at a node x returns "yes" if the edge $\{v, w\}$ is incident to the tree in F containing x.

Implementation Details

In this section we describe the implementation details of the distributed procedures listed above.

- COUNT(t) invoked on tier t at a node x returns 1 plus 1 for each ACK received from adjacent nodes it invoked COUNT(t) in.
- PATH(y, t) invoked on tier t at a node x terminates at x if $x = y$ or if x is a leaf node in the calling sequence. When PATH terminates at x, if $x = y$ then x

returns "yes", otherwise, x returns "no". A non-terminating node returns "yes" if it received "yes" as an ACK from any of its children in the calling sequence of PATH and "no" otherwise.

- FIND(y) invoked on tier *top* at a node x terminates at x if y is adjacent to x in F or x is a leaf node in the calling sequence. If FIND terminates at x because x it is a leaf node in the calling sequence it returns $(\phi, -1)$. If FIND terminates at x because y is adjacent to x in F then x returns $(\{x, y\}, t(\{x, y\}))$ where $t(\{x, y\})$ is the tier of $\{x, y\}$. Suppose x is a non-terminating node. If all children of x in the calling sequence of FIND return $(\phi, -1)$ then x returns $(\phi, -1)$. Otherwise, some child u of x will return $(\{v, w\}, t(\{v, w\}))$ where $t(\{v, w\}) > 0$. If $t(\{x, u\}) > t(\{v, w\})$ then x will return $(\{x, u\}, t(\{x, u\}))$, otherwise, x will return $(\{v, w\}, t(\{v, w\}))$.

- UNMARK($\{v, w\}$) invoked on tier *top* at a node x terminates at x if x is a leaf node in the calling sequence. If $x \in \{v, w\}$ then after receiving an ACK from all of its children in the calling sequence x unmarks $\{v, w\}$ locally as a tree edge.

- UPDATE(t) invoked on tier t at a node x returns $S^t(x) \oplus (\bigoplus S^t(y))$ for each node y that x received an ACK from.

- ROOTCHANGE(x) invoked on tier *top* at a node y sets $root(y) \leftarrow x$.

- MARK($\{x, y\}, t$) is always invoked at x or y on tier *top*. Assume without loss of generality that MARK is started at x. Then x will mark $\{x, y\}$ as a tier t tree edge and invoke MARK($\{x, y\}, t$) in y. When MARK is invoked in y, y marks $\{x, y\}$ as a tier t tree edge and terminates.

- ISVALID($\{v, w\}$) invoked on tier *top* at a node x terminates at x if x is a leaf node in the calling sequence or $x \in \{v, w\}$. If x is a terminating node and

$x \notin \{v, w\}$ then x returns "no". Suppose $x \in \{v, w\}$ and let *other* equal the node in $\{v, w\}$ such that $other \neq x$. Then x returns "yes" if *other* is a neighbor of x and "no" otherwise. If x is a non-terminating node x returns "yes" if it receives a "yes" with an ACK from any of its children in the calling sequence of IsValid and "no" otherwise.

5.3.2 Handling Updates

Edge Insertions

When notification of the insertion of $\{x, y\}$ reaches x (resp. y), x (resp. y) invoke the procedure Insert($\{x, y\}$) to update their local tables and coordinate a possible merge of the tree T_x containing x and the tree T_y containing y in F. When updating the local tables some synchronization is required to ensure both x and y use the same random bits. By convention the node with the smaller ID assumes the responsibility of determining the random bits used (in general this convention is used to dictate which node acts first and which waits when distributed functions are required to operate in sequence). Assume without lose of generality that $x < y$. Then x will update its local tables and send the message $(A_x(\{x, y\}), root(x))$ to y. When y receives the message $(A_x(\{x, y\}), root(x))$ from x it updates its local tables using the random bits of $A_x(\{x, y\})$. Then y compares $root(x)$ to $root(y)$ to see if they belong to the same tree in F. If $root(x) \neq root(y)$ then y will call RootChange($root(x)$) to notify each node in the component containing y that it belongs to the tree in F rooted at x. Then y will mark $\{x, y\}$ as a tier 1 tree edge and return $TRUE$ to x indicating T_x and T_y should be merged. Otherwise, if $root(x) = root(y)$ then y simply returns $FALSE$ inidicating that x and y are already connected in F. When x finally receives the message *merge* from y it determines if a merge of T_x and T_y is necessary. If $merge = TRUE$ then x marks $\{x, y\}$ as a tier 1 tree edge merging T_x and T_y over

$\{x, y\}$. The code for INSERT is shown in Algorithm 8.

Algorithm 8 INSERT($\{x, y\}$) at x

```
1: if x < y then
2:    add {x, y} to A_x^t({x, y}) on each tier t
3:    send (A_x({x, y}), root(x)) to y
4:    wait for message merge from y
5:    if merge = TRUE then
6:       mark {x, y} as a tier 1 tree edge
7:    end if
8: else
9:    wait for message (A_y({x, y}), root(y)) from y
10:   update local tables with random bits A_y({x, y})
11:   if root(x) ≠ root(y) then
12:      call RootChange(root(y))
13:      mark {x, y} as a tier 1 tree edge
14:      send TRUE to y
15:   else
16:      send FALSE to y
17:   end if
18: end if
```

Edge Deletions

Deletion of an edge $\{x, y\}$ may cause a violation of invariant (3), i.e. a component containing x and/or a component containing y may become unmatched on its tier. In [16] violations of invariant (3) are recursively fixed until no non-maximal component containing x is unmatched and then repeats this procedure for y. When x and y are notified of the deletion of $\{x, y\}$ they invoke the local procedure DELETE($\{x, y\}$) which initiates a series of routines to fix violations of invariant (3). To an extent we would like to replicate this sequential process used by [16] in the distributed setting. To achieve this portions of rebuilding process are synchronized. By convention the node with the smaller ID acts first while the endpoint with the higher ID waits a predetermined number of time steps for the node with the smaller ID to finish.

DELETE($\{x, y\}$)

Suppose that the edge $\{x, y\}$ fails and assume that $x < y$. When x (resp. y) are notified of the failure of $\{x, y\}$ they invoke DELETE($\{x, y\}$). First x (resp. y) remove $\{x, y\}$ from their local tables. If $\{x, y\} \in F$ then its deletion will split the component T in F containing $\{x, y\}$ into a component T_x containing x and a component T_y containing y. In this case a replacement edge connecting T_x and T_y must be found if it exists. Note that root of T will belong to exactly one of T_x and T_y and therefore we must decide on a new root for the component where the root is absent. To do this, x (resp. y) determine if there is a path in F to $root(x)$ (resp. $root(y)$). If not x (resp. y) becomes the new root, invoking ROOTCHANGE(x) (resp. ROOTCHANGE(y)) to inform their component that x (resp. y) is the new root. In the next step DELETE will attempt to fix violations of invariant (3). We require that T_x and T_y run sequentially and therefore x acts first immediately invoking the local procedure $Fix(0)$ and y waits $\tilde{O}(n)$ time steps before calling $Fix(0)$. The code for DELETE is shown in Algorithm 10.

FIX(t)

FIX(t) initiated at a node x begins by invoking UNMATCHED(t) to determine the lowest tier t' of the tree $A \in F_{t'}$ containing x which has become unmatched on its tier. FIX attempts to match A to another tier t' tree using a tier $t'+1$ tree edge. To do this FIX invokes UPDATE(t') to compute the table S, equal the bitwise XOR of the local tables of $CutsetDS_{t'}$ at each node in A, which is used to find a level $t'+1$ replacement edge $e = \{v, w\}$. If e is a valid replacement edge then x broadcasts a MERGE($\{v, w\}$, $t'+1$) message over the edges of A. Assume without loss of generality that $v \in A$ (and consequently $w \notin A$), then after at most $n-1$ time steps v will receive the message and invoke MERGE($\{x, y\}, t'+1$). If e is not a valid edge, i.e. a replacement edge was not found, then FIX concludes its has found a maximally connected component

and terminates. The code for FIX is shown in Algorithm 11.

MERGE($\{v, w\}, t$)

MERGE($\{v, w\}, t$) initiated at a node v at tier t coordinates the merging of the tier $t-1$ tree containing x and the tier $t-1$ tree containing y in F_{t-1}. Before $\{v, w\}$ can be added to F_t, we must determine if there is path from v to w in some forest $F_{t'}$ $t' > t$ to avoid creating a cycle. Therefore, MERGE invokes FIND(w) to determine the edge with highest tier on the path from v to w. If FIND(w) returns a valid edge $\{a, b\}$, v will first call UNMARK($\{a, b\}$) to remove $\{a, b\}$ from F making it safe to add $\{v, w\}$ to F without creating a cycle and then call MARK($\{v, w\}, t$) to mark $\{v, w\}$ as a level t tree edge at v and w. If FIND(w) does not find an edge, then v and w are not already connected in F. In this case MERGE calls MARK($\{v, w\}, t$) then ROOTCHANGE($root(v)$) to indicate that v has become the root of the merged tree. Finally, MERGE calls FIX(t), repeating this process until no non-maximal component containing x is unmatched. The code for MERGE is shown in Algorithm 12.

Algorithm 9 UNMATCHED(t) at x

```
1: for t' = t, t+1, ..., top-1 do
2:    if COUNT(t) =COUNT(t+1) then
3:       return t
4:    end if
5: end for
6: return top
```

5.4 Refreshing Random Bits and Fixing Errors

As shown by Kapron, King, and Mountjoy [16] our algorithm maintains invariants (1) - (5) with high probability over any polynomial length sequence of updates. In [16], to maintain the probability of success indefinitely edges are incrementally added to a

Algorithm 10 DELETE($\{x, y\}$) at x

```
1: remove {x, y} from all local tables
2: if {x, y} ∉ F then
3:    return
4: end if
5: if PATH(root(x)) = "no" then
6:    call ROOTCHANGE(x)
7: end if
8: if x > y then
9:    wait Õ(n) time steps
10: end if
11: call FIX(0)
```

Algorithm 11 FIX(t)

```
1: set t' ← UNMATCHED(t)
2: set S ← UPDATE(t')
3: set {x, y} ← SEARCH(S, t')
4: if {x, y} ≠ φ then
5:    broadcast MERGE({x, y}, t' + 1) on all tree edges t' ≤ t
6: end if
```

Algorithm 12 MERGE($\{x, y\}, t$) at x

```
1: set {v, w} ← FIND(y)
2: if {v, w} ≠ φ then
3:    call UNMARK({v, w})
4:    call MARK({x, y}, t)
5: else
6:    call MARK({x, y}, t)
7:    call ROOTCHANGE(root(x))
8: end if
9: call FIX(t)
```

second data structure which replaces the primary data structure every $O(n^2)$ updates. Unfortunately this technique does not work in the distributed setting because it is impossible to coordinate between different maximally connected components of the graph.

Therefore, instead of coordinating the refreshing of the random bits for each edge and swapping out the entire data structure every $O(n^2)$ updates, we periodically

Algorithm 13 SEARCH(S, t) at w

```
1: for i = 1, ..., ⌈log n⌉ and j = 1, ..., c log n do
2:    set {x, y} ← S_{i,j}
3:    if one of PATH(x, t) and PATH(y, t) returns "yes" and ISVALID({x, y}) returns "yes" then
4:       return {x, y}
5:    end if
6: end for
7: return φ
```

refresh the random bits used by the algorithm and fix mistakes. The main challenges of this approach are (1) ensuring that no random bits of $CutsetDS_t$ for any tier t are are used to find a replacment edge more than a polynomial number of times before they are refreshed and (2) determining how F should be rebuilt when a mistake is found.

Our technique is similar to that used in [16]; after the deletion of tree edge we refresh the random bits of a single edge $\{x, y\}$ but instead of adding the edge to a secondary data structure we determine if an error has been made that can be fixed by adding $\{x, y\}$ to F_t on some tier t. That is, we determine if $\{x, y\}$ belongs the cut of an unmatched component in F_{t-1} for some tier $t - 1$. If so, the component is matched (and the error fixed) by marking $\{x, y\}$ as a tier t tree edge. Unfortunately, adding $\{x, y\}$ to F_t may create a cycle in F or cause a component on a higher tier to become unmatched. However, these consequences are exactly the same as those faced when adding a replacement edge found by $CutsetDS_{t-1}$ to F_t as the result of tree edge being deleted. Therefore to add $\{x, y\}$ to F_t we call MERGE$(\{x, y\}, t)$, which initiates the same rebuilding process that is used when a tree edge is deleted, to fix any violations of invariant (3) on tiers $t' \geq t$ caused by adding $\{x, y\}$ to F_t.

5.4.1 Keeping Count

In this section we describe modifications to our data structure that allow us to ensure no random bits of $CutsetDS_t$ for any tier t are are used to find a replacment edge more than a polynomial number of times before they are refreshed. We say that the random bits $A_x(\{x, y\})$ are *queried* when x belongs to an unmatched component $A \in F_t$ and $CutsetDS_t$ is queried for a replacement edge that will match A on its tier. To keep track of how many times the random bits $A_x(\{x, y\})$ have been queried we keep a *count*, $count_x(\{x, y\})$ at x intially set to 0 when $\{x, y\}$ is inserted. Therefore, for each edge $\{x, y\} \in E$ there are exactly two counts; $count_x(\{x, y\})$ at x and $count_y(\{x, y\})$ at y. Intuitively, $count_x(\{x, y\})$ represents an upper bound on the number of times the random bits $A_x(\{x, y\})$ have been queried and is used by the algorithm to determine when the random bits $A_x(\{x, y\})$ and $A_y(\{x, y\})$ will be refreshed.

To update the appropriate counts when a tree edge is deleted we introduce the distributed procedure INCREMENT. The purpose of INCREMENT is to increment all the counts at each node in a tree $T \in F$ and to return the node $y \in T$ with the highest count out of all nodes in T. The implementation details of INCREMENT are as follows:

INCREMENT() invoked on tier *top* at a node x increments $count_x(\{x, y\})$ for each edge $\{x, y\}$ incident to x. If x is a leaf node in the calling sequence of INCREMENT x returns $(x, max(x))$ where $max(x)$ is the value of the maximum count at x. Suppose x is not a leaf node. Let A=$\{(y_1, max(y_1)), (y_2, max(y_2)), ..., (y_k, max(y_k))\}$ be the set of values returned to x from children of x in the calling sequence of FINDMAX. Let $(y, max(y)) = (y_i, max(y_i)) \in A$ where $max(y_i) \geq max(y_j)$ for all $i \neq j$. Then x will return $(x, max(x))$ if $max(x) > max(y)$ and will return $(y, max(y))$ otherwise.

In order to control the refreshing of the random bits of each edge and the fixing of mistakes we modify the procedure FIX and define the local routines REFRESH and REFRESHBITS discussed in the next section.

5.4.2 Handling Deletions

Consider the deletion of the tree edge $\{x, y\}$. Let T_x be the tree in F containing x and T_y be the tree in F containing y. The deletion of $\{x, y\}$ may cause violations to invariant (3) which are fixed by the rebuilding process defined by a series of calls to FIX and MERGE. Consider the series of calls to FIX and MERGE initiated by x calling FIX(0). Such a series of calls terminates when FIX terminates as the result of SEARCH failing to find a valid replacement edge. At this point the algorithm is done, satisfied that it has fixed any violations to invariant (3) containing x. Now instead of having FIX terminate when SEARCH fails to find a valid replacement edge, we modify FIX to call INCREMENT and REFRESH. Let u be the node returned by INCREMENT. FIX then broadcasts the message REFRESH(u) over the edges of T. When this message arrives at u, u calls REFRESH($\{u, v\}$) where $count_u(\{u, v\})$ is the count at u with the highest value. The modified code for FIX is show in Algorithm 14.

REFRESH($\{u, v\}$)

The responsibility of REFRESH($\{u, v\}$) called by a node u is to refresh the random bits $A_u(\{u, v\})$ and $A_v(\{u, v\})$ and to determine if $\{u, v\}$ belongs to the cut of an unmatched component containing u in F_t on some tier t, in which case a mistake was found. To determine if $\{u, v\}$ belong to the cut of an unmatched component containing u we must consider two cases: either (1) u and v are not connected in F_{top} or (2) u and v are connected in F_{top} and $\{u, v\}$ belongs to an unmatched component containing u on a lower tier. Errors of form (1) are easily detected by determining

if there exists a path $u \rightsquigarrow v$ in F_{top}. If such a path does not exist then we let $t \leftarrow \text{UNMATCHED}(0)$ be the smallest tier of the unmatched component containing u. Errors of form (2) are slightly more difficult to detect. Similar to the first case, we let $t \leftarrow \text{UNMATCHED}(0)$ be the smallest tier of the potentially unmatched component containing u. Let $A \in F_t$ be the potentially unmatched component containing u. We conclude that A is unmatched and that $\{u, v\} \in (A, V \setminus A)$ if the maximum tier edge on the path $u \rightsquigarrow v$ in F is greater than $t + 1$.

In order to match A, u calls $\text{MERGE}(\{x, y\}, t + 1)$ which initiates the same rebuilding process used to fix violations of invariant (3) when a tree edge is deleted. The only modification being that when FIX terminates because SEARCH failed to find a valid replacement edge it does not call INCREMENT and REFRESH again. The code for REFRESH is show in Algorithm 16.

Algorithm 14 $\text{FIX}(t)$

```
1: set t' ← UNMATCHED(t)
2: set S ← UPDATE(t')
3: set {x, y} ← SEARCH(S, t')
4: if {x, y} ≠ φ then
5:    broadcast MERGE({x, y}, t' + 1) in F_{t'}
6: else
7:    call INCREMENT()
8:    set {x, max(x)} ← FINDMAX()
9:    broadcast message REFRESH(x) in F_top
10: end if
```

Algorithm 15 $\text{REFRESHBITS}(A_y(\{x, y\}))$ at node x

```
1: A_x({x, y}) ← A_y({x, y})
```

Algorithm 16 REFRESH($\{x, y\}$) at x

1: $count_x(\{x, y\}) \leftarrow 0$
2: refresh the random bits of the tables in $A_x(\{x, y\})$
3: invoke REFRESHBITS($A_x\{(x, y)\}$) in y
4: **if** $\{x, y\} \in F$ **then**
5: end
6: **end if**
7: $tpath(x) \leftarrow t(e)$ where $(e, t(e)) \leftarrow$ FIND(y)
8: $tunm(x) \leftarrow$ UNMATCHED(x)
9: **if** $tpath(x) = -1$ **or** $(tunm(x) + 1 < tpath(x))$ **then**
10: MERGE($\{x, y\}, tunm(x) + 1$)
11: **end if**

5.5 Analysis

Theorem 5.5.1. *The message complexity of* INSERT *is* $O(n)$.

Proof. Consider the invocation of INSERT($\{x, y\}$) at x and y after the insertion of $\{x, y\}$ such that $x < y$. First consider the messages sent by x. It requires one message of size $O(\log^4 n)$ to communicate the message $(A_x(\{x, y\}), root(x))$ to y. Now consider the messages sent by y. INSERT may call ROOTCHANGE which sends $O(n)$ messages of size $O(\log n)$. Upon termination y will send the message *merge* of size $O(1)$ to x. Therefore the message complexity of INSERT at x and y is $O(n)$. □

Theorem 5.5.2. *The message complexity of processing a deletion is* $\tilde{O}(n)$.

Proof. Consider the deletion of the tree edge $\{x, y\}$ such that $x < y$. DELETE will be invoked at both x and y. DELETE may call PATH and ROOTCHANGE each with message complexity $O(n)$. Next DELETE calls FIX(0). Consider the consider the number of messages sent by Fix(t) on some tier t. FIX first calls UNMATCHED which may call COUNT $O(\log n)$ times. However, we note that COUNT can be called at most twice on each tier and therefore the number of messages sent as a result of calling UNMATCHED is $O(n \log n)$. To see this, note that the parameter t in the call UMATACHED(t) is increasing and the parameter $t' + 1$ of the next call to UNMATCHED represents the largest tier in which COUNT was previously invoked. Next FIX calls UPDATE and SEARCH. UPDATE sends at most $O(n)$ messages of size $O(\log^3 n)$. SEARCH sends at most $O(n \log^2 n)$ message of size $O(\log n)$. Next FIX may call MERGE. It requires $O(n)$ messages to broadcast the message MERGE($\{x, y\}, t$) on some tier t. MERGE may call FIND, UNMARK, and ROOTCHANGE each of which send $O(n)$ messages of size $O(\log n)$. MERGE may also call MARK which sends a single message of size $O(\log n)$. Therefore, we can bound the message complexity of FIX and its call to MERGE on a single tier by $O(n \log^2 n)$. FIX can be called at most twice on each tier

requiring $O(n \log^3 n)$ messages. When SEARCH does not find a valid replacment edge FIX finally terminates and calls INCREMENT and REFRESH. INCREMENT sends $O(n)$ message so size $O(\log n)$. REFRESH calls REFRESHBITS, FIND, and UMATCHED. It requires $O(\log^4 n)$ messages to send the parameter $A_x(\{x, y\})$ of REFRESHBITS to y. FIND sends $O(n)$ messages of size $O(\log n)$ and UNMATCHED sends at most $O(n \log^2 n)$ message of size $O(\log n)$. Finally REFRESH calls MERGE to rebuild F which shown above requires $O(n \log^3 n)$ messages. Therefore the total message complexity of processing a deletion is $O(n \log^3 n)$. □

5.6 Correctness

It follows from [16] that our algorithm maintains invariants (1) - (5) with high probability over any polynomial length sequence of updates. What is left to show is that our algorithm maintains invariants (1) - (5) with high probability over any arbitrary length sequence of updates. Consider the i-th deletion in any sequence of updates to G. We say that a count c was incremented as a result of deletion i if it was incremented after deletion i and before deletion $i + 1$. Let C_i be the set of counts that were incremented as a result of the i-th deletion d_i in any sequence of updates to G. We say that d_i is *active* if there exists a count $count_x(\{x, y\}) \in C_i$ that has not been set to 0 as the result of x calling REFRESH$(\{x, y\})$ after d_i. If all counts $count_x(\{x, y\}) \in C_i$ have been set to 0 after d_i then d_i is not active.

Theorem 5.6.1. *The invocation of* REFRESH$(\{x, y\})$ *at x does not result in any additional violations of invariants (1) - (5).*

Proof. Consider the invocation of REFRESH$(\{x, y\})$ at x. If $\{x, y\}$ does not belong to the cut of an unmatched component the random bits in $A_x(\{x, y\})$ and $A_y(\{x, y\})$ are simply refreshed and none of the invariants are affected. Suppose $\{x, y\}$ be-

longs to the cut of the unmatched component A on tier $t-1$ resulting in x calling MERGE($\{x,y\},t$). Invariants (1) and (2) are not effected by the addition of $\{x,y\}$ to F_t because $t>0$ and any edge added to F_t is added to $F_{t'}$ for all $t'>t$. Adding $\{x,y\}$ to F_t cannot result in any trees in $F_{t''}$ where $t''<t$ to become unmatched but may cause a component on a higher tier to become unmatched thus violating invariant (3). Such an unmatched component will be detected and matched by subsequent calls to FIX and MERGE. Finally, the fact that A is unmatched is dependent on the failure of $CutsetDS_{t-1}$ for A finding an edge in the cut $(A, V \setminus A)$. Therefore, by invariant (4) this failure can only effect the structure of $F_{t'}$ for $t' \geq t$. Adding $\{x,y\}$ to F_t and the subsequent rebuilding process only effect the structure of F on tiers $\geq t$ and therefore invariant (4) is satisfied. Invariants (1) - (4) imply (5). □

Theorem 5.6.2. *Any errors made by the algorithm as the result of a non-active delete have been fixed.*

Proof. Let A be the unmatched component SEARCH failed to match while rebuilding F as the result of the non-active delete d. Consider any count $count_x(\{x,y\})$ such that $x \in A$ and $\{x,y\} \in (T, V \setminus A)$ that was incremented as a result of d. Then by the definition of a non-active delete REFRESH($\{x,y\}$) must have been called at x and therefore A must have been matched. □

Theorem 5.6.3. *No count can exceed $O(n^2)$.*

Proof. : Let A be the set of all possible counts kept at all nodes and let $N = n^2 - n = 2 \cdot \binom{n}{2}$. Then $|A| = N$ because for each of the $\binom{n}{2}$ possible edges $\{x,y\} \in E$ there are two distinct counts, $count_x(\{x,y\})$ and $count_y(\{x,y\})$, in A. For each edge $\{x,y\}$ not in E, those that have been deleted and not yet re-inserted or those that were never inserted, we set $count_x(\{x,y\})$ and $count_y(\{x,y\})$ to 0. This allows for insertions to have no effect on the count values because when an edge is inserted its

corresponding counts are intialized to 0. Throughout the algorithm we maintain the following invariant:

Invariant: For any i, $0 \leq i \leq N-1$, the number of counts with value at least i after completion of REFRESH is no greater than $N - i$.

We prove by induction on the number of refreshes that the invariant holds throughout the course of the algorithm.

Base Case: Initially all counts are 0. Therefore, for $i = 1, ..., N-1$ we have $a_i = 0 \leq N - i$ and $a_0 = N \leq N - 0$.

Induction Step: Suppose the invariant holds after exactly t refreshes. We show that it must hold after exactly $t+1$ refreshes for $i = 0, ..., N-1$. Note the that number of counts with value at least 0 is always N and therefore the invariant is always satisfied for $i = 0$. Let b_i for $i = 1, ..., N-1$ denote the number of counts in A with value at least i after exactly t refreshes. By the invariant, $b_i \leq N - i$ for each i. Let $m - 1$ denote the largest value of a count in A that was incremented and not set to 0 during refresh $t + 1$. To show the invariant is satisfied for $i = 1, ..., N-1$ after refresh $t + 1$ we consider the following two cases:

Case $i \leq m$. Note each refresh results in at least one count being set to 0. For each $i \leq m$ the number of counts with values at least i after exactly $t + 1$ refreshes is at most the number counts in A with values at least $i - 1$ after exactly t refreshes minus at least one count that was set to 0. Therefore, the number of counts with value at least i after exactly $t + 1$ updates is at most $b_{i-1} - 1 \leq N - i + 1 - 1 = N - i$.

Case $i > m$. For each $i > m$ the number of counts with value at least i does not change and therefore satisfies the invariant.

□

Theorem 5.6.4. *There are at most $O(n^4)$ active deletes at any stage of the algorithm.*

Proof. Let S be the sum of all counts at each node in V. By the invariant of Theorem 5.6.3, S is bounded by $O(n^4)$. Each non-zero count with value c was incremented as a result of at most c distinct active deletes after last being set to 0. Therefore, the number of active deletes is bounded by $S = O(n^4)$. □

Theorem 5.6.5. *The algorithm maintains the invariants (1)-(5) with probability at least $1 - 1/n^c$.*

Proof. The random bits for $CutsetDS_t$ on each tier are generated and queried for replacement edges in the same way as is done in [16]. Therefore by Lemma 4.1 of [16] if we let $c' > -(c+k)\log 8/9$ be the constant used by the cutset data structure the deletion of a tree edge will result in the violation of invariants (1) - (5) with probability at most $1/n^{c+k-1}$. By union bound on the number of active deletes (any one of which introduces an error with probability $\leq 1/n^{c+k-1}$) we have that invariants (1) - (5) are correct with probability at least $1 - n^4/n^{c+k-1}$. By setting $k = 5$ we have a probability of success of $1 - 1/n^c$ for any constant c. □

Chapter 6

Concluding Remarks

In this thesis we presented a novel sampling technique and demonstrated its application to the following fully dynamic graph algorithms:

1. A randomized algorithm to estimate the size of a cut in an undirected graph $G = (V, E)$ where V is the set of nodes and E is the set of edges and $n = |V|$ and $m = |E|$. Our algorithm processes edge insertions and deletions in $O(\log^2 n)$ time. For a cut $(U, V \setminus U)$ of size K for any subset U of V, $|U| < |V|$ our algorithm returns an estimate x of the size of the cut satisfying $K/2 \leq x \leq 2K$ with high probability in $O(|U| \log n)$ time.

2. A randomized distributed algorithm for maintaining a spanning forest in a fully dynamic synchronous network. Our algorithm maintains a spanning forest of a graph with n nodes, with worst case message complexity $\tilde{O}(n)$ per update and worst case memory requirement of $\tilde{O}(degree(v))$ bits at each node v.

In future work we would like to extend the functionality of our distributed spanning forest algorithm presented in Chapter 5 by removing some of the restrictions of our model. Currently, we are working on extending this algorithm to support batch

updates. Our goal is to give an algorithm that can process a batch of insertions and deletions with the same message and time complexity as our algorithm presented here can process a single update. Ideally we would like to extend our algorithm to asynchronous networks. One of the main challenges is coordinating the merging of trees when replacement edges are chosen at random, while maintaining the invariants outlined in Chapter 5 that control the structure of the spanning forests on each tier.

Chapter 7

Bibliography

[1] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 358–370, 1987.

[2] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 459–467. SIAM, 2012.

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st symposium on Principles of Database Systems*, PODS '12, pages 5–14, New York, NY, USA, 2012. ACM.

[4] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[5] Baruch Awerbuch, I. Cidon, and S. Kutten. Communication-optimal maintenance of replicated information. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 492–502 vol.2, 1990.

[6] Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of a spanning tree. *J. ACM*, 55(4):18:1–18:45, September 2008.

[7] Baruch Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 206–219, 1988.

[8] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1 – 4, 1980.

[9] Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. *Journal of Computer and System Sciences*, 72(8):1282 – 1308, 2006.

[10] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsificationa technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, September 1997.

[11] S. Finn. Resynch procedures and a fail-safe network protocol. *Communications, IEEE Transactions on*, 27(6):840–845, 1979.

[12] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[13] J.A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 659–668, 1993.

[14] MonikaRauch Henzinger. Approximating minimum cuts under insertions. In Zoltn Flp and Ferenc Gcseg, editors, *Automata, Languages and Programming*, volume 944 of *Lecture Notes in Computer Science*, pages 280–291. Springer Berlin Heidelberg, 1995.

[15] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.

[16] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1142, 2013.

[17] David R. Karger. Using randomized sparsification to approximate minimum cuts. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 424–432, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[18] Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 238–251, New York, NY, USA, 1995. ACM.

[19] Shay Kutten and Avner Porat. Maintenance of a spanning tree in dynamic networks. In Prasad Jayanti, editor, *Distributed Computing*, volume 1693 of *Lecture Notes in Computer Science*, pages 342–355. Springer Berlin Heidelberg, 1999.

[20] Eli Upfal Michael Mitzenmacher. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.