Bubbles: Adaptive Routing Scheme for High-Speed Dynamic Networks*

Shlomi Dolev[†] Evangelos Kranakis[‡] Danny Krizanc[‡] David Peleg[§]

December 17, 1997

Abstract

This paper presents the first dynamic routing scheme for high-speed networks. The scheme is based on a hierarchical *bubbles* partition of the underlying communication graph. Dynamic routing schemes are ranked by their *adaptability*, i.e., the maximum number of sites to be updated upon a topology change. An advantage of our scheme is that it implies a small number of updates upon a topology change. In particular, for the case of a bounded degree network it is proved that our scheme is optimal in its adaptability by presenting a matching tight lower bound. Our bubble routing scheme is a combination of a distributed routing data-base, a routing strategy and a routing data-base update. It is shown how to perform the routing data-base update on a dynamic network in a distributed manner.

1980 Mathematics Subject Classification: 68Q99

CR Categories: C.2.1

Key Words and Phrases: Adaptability, Dynamic, Fiber Optic, Communication Network, Routing

---

*An extended abstract of this work was presented at the Twenty-Seventh annual ACM Symposium on the Theory Of Computing, May 1995.

[†]Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel. Part of this work was done while this author visited the School of Computer Science, Carleton University, and the Department of Computer Science, Texas A&M University. Email: `dolev@cs.bgu.ac.il`. Part of this research was supported by NSERC (Natural Sciences and Engineering Research Council of Canada), by TAMU Engineering Excellence funds and by NSF Presidential Young Investigator Award CCR-9396098.

[‡]School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada. Email: {`kranakis, krizanc`}`@scs.carleton.ca`. Part of this research was supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

[§]Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. Email: `peleg@wisdom.weizmann.ac.il`. Supported in part by a grant from the Israel Science Foundation.

1 Introduction

1.1 Motivation

The advent of fiber-optic technology dramatically changes the characteristics of distributed networks. It also improves the capabilities of distributed networks because it gives them the potential of supporting new services such as multi-media and real-time applications. Traditional algorithms designed for the point-to-point classical model of distributed networks may neither fit the new characteristics of the high-speed network nor support the new tasks it is capable of achieving. The relation between the bandwidth of a fiber-optics cable (on the order of hundreds of Gigabits per second) and the speed of a processor implies a bottleneck in the process time as opposed to the communication time. Therefore, high-speed networks use a fast switching subsystem in order to utilize the power of the fiber optic cables.

Algorithms for basic tasks that match the new network structure are of interest. In (high-speed) distributed networks messages are used for communication between different sites. A message sent from one site to another is transferred through the network according to a *routing scheme*. The routing scheme ensures that the message is forwarded towards its destination. The routing scheme serves the basic communication primitive in the network — message delivery. Being such a basic component, the performance of the distributed network as a whole may be dominated by the quality of the routing scheme. Thus, finding an efficient routing scheme is one of the most important tasks in distributed networks.

Imagine a network in which users may be connected and disconnected upon request. Users may migrate from one geographical region to another causing a change in the demand for services at different parts of the network. Assume further that the network spans the entire world and a single user (or a network junction) changes its location from one street of New York to another: is it reasonable to update the *entire* network with a new routing data-base? We would not like the *entire* distributed network to be updated upon each such dynamic change. In fact we would like to *minimize* the effect of a topology change as much as possible.

Beyond planned topology changes, such as users migration, some transient topology changes may take place due to a failure of communication links or processors. One would like the network to automatically change the routing data-base to reflect the new topology upon the change. The resources used by such a distributed routing data-base update (messages and time) have an inherent relation with the number of sites that have to change their portion of the distributed routing data-base.

We distinguish between *static* and *dynamic* routing schemes. A *static* routing scheme is a combination of a distributed routing data-base and a fitting routing strategy. The routing data-base is tailored to the network topology. Whenever the network changes its topology, a new distributed routing data-base is assigned to the network possibly changing the routing data-base portion of each processor. A *dynamic* routing scheme has in addition a fitting routing data-base update. Upon a topology change (e.g., link addition or removal) this fitting routing data-base update would change the distributed data-base only in a *limited* number of sites. We rank the dynamic routing schemes by their *adaptability*, i.e., the maximum number of sites to be updated upon a topology change. By this definition static routing schemes are associated with adaptability that is in the order of the number of nodes in the network.

The efficiency of a dynamic routing scheme is measured not only by its adaptability. It is also measured by the time and memory complexities associated with it. The time performance is measured by a *super-hop count*, defined as the maximum number of super-hops, or routing steps, needed to route a message from a given source to a given destination, where the maximum is taken over all possible origin-destination pairs. A more precise definition of the routing process in high-speed networks and the notion of super-hops will be given later on. The memory complexity is the total number of bits used for the routing data-base. We remark that there is an interesting relation between the adaptability and the memory requirement e.g., it is clear that the worst case in terms of adaptability and memory requirement is when the entire topology is stored in the memory of each processor. Roughly speaking, both the adaptability and the memory requirements gain when the processors have less information on the system.

1.2 Previous work

Many clever routing schemes and lower bounds for the resources required for routing in point-to-point networks were presented in the past. Following the pioneering work of [13, 14], which originated the approach of using hierarchical clustering strategies for memory-efficient routing, came a number of papers which attempted to characterize and bound the resource tradeoffs involved. The first set of studies along this line were mostly designed for special classes of networks like trees [16], complete networks [18], and grids [19]. Routing schemes for general networks were presented in [15, 4, 6]. These studies focused on the design of routing schemes with compact routing tables and low stretch factor. The *stretch factor* of a routing scheme is defined as the maximum ratio, over all pairs of nodes in the network, between the length of the route provided for them by the routing scheme, and the actual distance between them in the network.

Unfortunately, most of the success in this field is for static networks. Few papers consider the dynamic property of the network. The following quotation is taken from [16]: "In actual network the topology may vary in time; in particular, nodes or links may be added or deleted". [16] present a partial solution for limited cases of topology changes that keep the network in a tree structure. In [7], the routing scheme of [6] is extended to the dynamic case for general networks. However, as argued in [1], network changes in static routing schemes (such as [15, 3, 6] or the derived dynamic scheme [7]) "require expensive pre-processing to reconstruct the routing scheme over the whole network. The newly constructed structure is used until the next change...". In contrast, [1] design a routing scheme for the restricted case of dynamic growing trees. The solution of [1] can handle neither link nor processor failures, nor can it be applied to the case of general graphs.

The papers mentioned above all deal with the traditional point-to-point communication model. Those solutions are not applicable for fiber optic high-speed networks, since the stretch measure (based on actual distances) is no longer the predominant cost parameter relevant to such networks, given their specific characteristics. Furthermore, the issue of adaptability is not handled by any of the above papers. In particular, the solutions provided in the above papers for the dynamic version of the problem might require data-base updates in all the nodes of the network following a single topology change.

Recently, [12, 10] presented static routing schemes for high-speed networks. The schemes statically assign links along a path to act as a virtual long link. Both papers attempt to simultaneously optimize three cost parameters, namely, the super-hop count, the memory size and the link load. However, [12, 10] as well do not address adaptability explicitly, and in the resulting solutions a single topology change may in some cases require the entire routing data-base to be updated. Hence the solution might be impractical in a dynamically changing environment.

1.3 Contributions of this paper

In this work we present the first dynamic routing scheme for high-speed networks. We present a family of hierarchical *bubbles* schemes. The intuition behind this structure is the behavior of natural bubbles. Assume a partition of a space into bubbles. Whereas bubbles may shrink (members are disconnected) or expand and/or blow up (new members are inserted) we want to avoid total change of the bubble structure upon a change at a single bubble. We define an upper and lower threshold for the size of a bubble. When the upper threshold is reached the bubble is split into two bubbles. When the lower threshold is reached the bubble is combined with a single neighboring bubble which "swallows" the small bubble (and then is split if necessary).

The Bubbles scheme exhibits a tradeoff between the number of super-hops needed for routing on the one hand, and the adaptability and memory requirement parameters. Namely, the more super-hops are allowed, the less memory is needed to store the routing data-base, and the more efficient it becomes to adapt the scheme to dynamic changes. Denote the maximum number of neighboring nodes with which each node in the network may be directly connected by δ. For the case of constant δ we prove that our scheme is optimal in its adaptability by presenting a matching tight lower bound. Note that this is the case in many settings in reality, where the number of communication ports of a single processor is limited. Our bubble routing scheme is a combination of a distributed routing data-base, a routing strategy and a routing data-base update. We present a distributed routing data-base update strategy for dynamic changing networks.

The rest of the paper is organized as follows. The definition of the problem and two examples for basic routing schemes appear in Section 2. These two schemes are given as examples for our complexity measures and a base for comparison with the bubbles scheme, which is presented in Section 4. Section 5 presents an improved scheme, based on a somewhat more involved bubbles partitioning algorithm. See Figure 1 for a summary of the comparison among the different schemes. Note that we typically choose k to be a small constant, with a value much smaller than $\log n$. Section 6 presents a lower bound on the adaptability of any routing scheme. The distributed update of the bubbles routing data-base appears in Section 7. Concluding remarks are given in Section 8.

	Super-hop count	Memory Required	Adaptability	Model
Multiple Trees	1	$O(n^2 \log n)$	n	
Single Leader	2	$O(n^2 \log \delta)$	n	
Basic Bubbles	k	$O(k\delta n^{1+1/k} \log \delta)$	$O(3^k \delta^2 n^{1/k})$	Node failures
Basic Bubbles	k	$O(k\delta n^{1+1/k} \log \delta)$	$O(3^k \delta n^{1/k})$	Link failures
Edge-Bubbles	k	$O(kn^{1+1/k} \log \delta)$	$O(3^k \delta n^{1/k})$	Node Failures
Edge-Bubbles	k	$O(kn^{1+1/k} \log \delta)$	$O(3^k n^{1/k})$	Link failures

Figure 1: Comparison of Routing Schemes

2 Definition of the Problem

2.1 High-Speed Dynamic Network

We consider the high-speed model of a communication network as described in [8, 5, 9]. The network is described by an undirected graph $G = (V, E)$. The nodes, $V = \{1, \dots, n\}$, represent the processors of the network (where n is essentially an upper bound on the number of processors in a connected component). The edges of the graph represent bidirectional communication channels between the processors.

The network is *dynamic* in the strong sense: processors and links may crash and recover arbitrarily. However, the number of edges connected to a node P, which we call the *degree* of P, does not exceed some predefined value. We denote the maximum degree of nodes in the network by δ.

Each processor consists of two components, a *switching subsystem* and a *node control unit* that are connected by a virtual link. The *switching subsystem* is a fast and simple hardware device that switches arriving packets to the appropriate communication link or to its node control unit. Within the switching subsystem each communication link (including the virtual link to the switching subsystems' node control unit) has a unique label. When a packet p arrives at a switching subsystem and the header of p contains at least one label, then the switching subsystem removes the first label, l, and the shortened packet is sent on the link with label l. The *node control unit* of each processor contains the processing hardware and software necessary to extract the information content of messages (delivered in the packets), do internal computation, and generate packets to be forwarded to other nodes via the processor's switching subsystem.

In this model a packet consists of the entire path from the sender to the receiver. [[For example when the packet length is 50 bytes and the degree of nodes in the network is $\delta = 5$ then a packet may contain 5^{50} different addreses.]] This is more than enough for every existing network. Moreover, note that the first packet in a connection may mark the path (by the connection identifier) for the next packets that are related to the same connection — thus not every packet contains the entire path. Due to the above network architecture, it is assumed that a message sent from any processor, P, to any destination, Q, in the network may arrive in one time unit provided the labels along the entire path from P to Q are known to P. We refer to such a routing step as a *super-hop*.[1] In case the entire path is not known to P, the message may be sent through a number of super-hops. In each super-hop, the sender packs the message in a packet whose header contains the route description to the next intermediate destination, and advancing the message to that destination. Since the overhead of preparing the packets dominates the transmission time, the cost of the entire routing process is proportional to the number of super-hops it involves.

2.2 Complexity Measures for Routing Schemes

The routing of packets in the network is done according to a *distributed routing data-base* maintained by the node control unit and a fitting *routing strategy* and *data-base update*.

Super-hop count: The *super-hop count* of a routing scheme bounds the maximum number of super-hops (or, packet generation and transmission steps) required in order to send a message from one node control unit to another. Obviously, a single packet transmission is sufficient when every processor knows the entire topology (including the link labels). Thus, the super-hop count can be thought of as measuring the ratio between the maximum number of packets *generated* by the routing scheme and the optimal number of packets that must be generated in order to deliver a message sent from one node control unit to another. It is interesting to note that in that sense, the super-hop count of a routing scheme in the high-speed model essentially plays a similar role to that played by the *stretch factor* measure in routing schemes for traditional computer networks, cf. [15, 3, 6].

Memory: The memory complexity is the total number of bits maintained by the node control units for the routing data-structure.

Adaptability: A single topology change $\mathcal{C}$ occurs when a single processor or a single link joins (or recovers) or leaves (or crashes) the system. Note that for any two topologies G_1 and G_2 there exist a finite sequence of single topology changes $\mathcal{C}_1, \mathcal{C}_2, \ldots$ that transfer G_1 into G_2. For example the sequence can start with adding all the processors that do not appear in G_1 but do appear in G_2, one processor at a time. Then links are added in a similar fashion. Finally, links and then processors are removed to form G_2. Ideally a topology change causes only a very limited number of processors to change their portion of the routing distributed data-base. Thus, we choose the *adaptability measure* to be the maximum number of processors that have to change their portion of the routing distributed data-base upon a single topology change. A similar measure, called the *adjustment measure*, is proposed in [11].

We shall first ignore the issue of *how* the routing data-base is updated upon a topology change and only count the number of processors that have to change their routing data-base. Then we present a distributed update for the bubbles routing scheme which we call *bubbles update*.

Distributed adaptability: is the maximal number of node control units that have to participate in the distributed routing data-base update upon a single topology change. Clearly,

[1] Certain routing models devised for ATM networks use more elaborate mechanisms, and distinguish between several types of "super-hops", e.g., routing along virtual channels or virtual paths. Here, we follow the simple, unified model of *Automatic Network Routing (ANR)* of [8, 9].

the distributed adaptability is greater than the adaptability since every node control unit that has to change its portion of the routing distributed data-base participates in the distributed data-base update, and in addition, other node control units may participate in the distributed update without changing their routing data-base portion.

3 Examples of Basic Routing Schemes

In this section we present two basic routing schemes, whose adaptability is n.

3.1 The Multiple Spanning Trees Scheme

The simplest routing scheme for our network is the *multiple spanning trees* routing scheme.

Routing Distributed Data-base: The routing distributed data-base is a description of a spanning tree of the entire topology at each processor.

Routing Strategy: The routing strategy is to use the entire path given by the routing distributed data-base.

Routing Update: The routing update has to change the distributed routing data-base upon every processor addition and removal as well as upon removal of a link used as part of a spanning tree by some processor. The update would change the spanning trees representation in an obvious way.

The following lemma states the super-hop count, memory and adaptability properties of the multiple trees routing scheme.

Lemma 3.1 *The multiple trees routing scheme has the following properties:*
(1) Super-hop count = 1.
(2) Memory requirement = $n^2(\log n + \log \delta)$*.*
(3) Adaptability = n*.*

Proof: The super-hop count of this scheme is clearly one, since a single packet is generated for the delivery of a message.

Property (2) holds since $n(\log n + \log \delta)$ bits are required in order to describe a spanning tree with link labels for every processor (say, in parenthesized form).

Property (3) follows from the fact that a single recovery of a processor or a link requires the update of the tree of every processor. ■

3.2 The Single Leader Scheme

Routing Distributed Data-base: Only a single processor, L, has a spanning tree of the entire topology. Every other processor, P, has the path description to L, $path_L$. $path_L$ is a list of link labels that defines a path from P to L.

Routing Strategy: To deliver a message m to a processor Q, a packet $(path_L, Q, m)$ is sent to the leader L, and then L sends a packet $(path_Q, m)$ to Q.

Routing Update: The routing update has to change the distributed routing data-base upon the leader failure, processors' addition, processors' removal, a failure of a link along a path used by some processor to reach the leader, an addition of a link that connects two connected components. For a given choice of a single leader the update is defined in a natural way.

The following lemma states the super-hop count, memory requirements and adaptability properties of the single leader routing scheme.

Lemma 3.2 *The single leader routing scheme has the following properties:*
(1) Super-hop count $= 2$.
(2) Memory requirement $= n(\log n + \log \delta) + n(n-1)\log \delta$.
(3) Adaptability $= n$.

Proof: The super-hop count is two since at most two packets are generated for each message.

The memory requirements of the scheme are as follows. The description of a spanning tree of the entire topology requires $n(\log n + \log \delta)$ bits as in the proof of Lemma 3.1. The description of a path from a processor P to L includes at most n link labels. Each link label requires $\log \delta$ bits. Thus, each processor but L uses at most $n \log \delta$ bits for the routing data-base.

Finally, if all the paths to the leader L, as well as the spanning tree stored by L, include a certain link e (say, adjacent to L), then the failure of e (while preserving the connectivity of the communication graph) is a topology change that will cause every processor to change its topology data-base, hence the adaptability is n . ∎

4 The Bubbles Routing Scheme

In this section we present our main result which is a novel dynamic graph partitioning technique, used in conjunction with a new routing scheme for dynamic high speed networks. We introduce a new type of hierarchical partitioning scheme for general graphs, with certain desirable properties. We then present a graph partitioning algorithm that constructs such a partition. Then we explain how to maintain the properties of the partition (most notably, the size bounds on connected components) upon a dynamic change. Finally, we present our routing scheme, which is based on the hierarchical partition.

The partitioning scheme uses a spanning tree of the communication graph. The partition of the spanning tree into connected components called bubbles is done in levels. The first level is a single bubble that includes $x_1 = n$ nodes. The second level partitions the single bubble into bubbles (i.e. connected components) of at least x_2 (where $x_2 < x_1/\delta$) nodes and no more than δx_2 nodes. In general, the bubble partition of level i partitions the bubble of level $i-1$ into bubbles of at least x_i nodes and no more than δx_i nodes. A failure of a link that belongs to the spanning tree may disconnect a number of bubbles, on several levels. We present a scheme to maintain the bubble partition with few changes as long as the communication graph is still connected. Assume that a bubble of level i is partitioned because of a link failure. One or two of the resulting bubble parts may contain fewer than x_i nodes. In such a case each part is combined with a neighboring bubble. Each of the two new combined bubbles may include more than δx_i nodes and therefore require a split. This fact and the fact that the partition on level $i+1$ is a refinement of the partition on level i imply that (at most) three edges should be removed from level $i+1$, namely, the failed link of level i and the two edges that are removed from bubbles of level i because of the split operations. The details follow.

4.1 The Bubbles Partition

Definition 4.1 *Given a graph G and parameters $0 < a < b$, an $[a,b]$-*bubbles partition *of G is a partition of the nodes of G into disjoint* connected *components called* bubbles, *$\mathcal{P} = \{\mathcal{B}^1, \ldots, \mathcal{B}^r\}$, such that the size of each bubble $\mathcal{B}$ (i.e., the number of nodes it contains), satisfies $a \le |\mathcal{B}| \le b$.*

Let us next describe a partitioning algorithm based on a technique presented in [17]. Given a n-node graph G with maximum degree δ and a parameter $1 \le x \le n$, the algorithm produces an $[x, \delta x]$-bubble partition for G. To be more precise, as a preprocessing step we choose an arbitrary node of the graph, R, and construct a spanning tree, $\mathcal{T}_R$, rooted at R with edges directed towards the leaves. The partitioning algorithm is then applied to this tree. The algorithm, PARTITION($\mathcal{T}, x$), is described in Fig. 2.

(1) While $\mathcal{T}$ contains more than δx nodes **do:**

- **(a)** Mark every node P in $\mathcal{T}$ by M_P, the total number of nodes in the subtree $\mathcal{T}_P$ rooted at P.
- **(b)** Find a node P with the property that $M_P \ge x$, but all of its children Q satisfy $M_Q < x$.
- **(c)** Make $\mathcal{T}_P$ into a bubble.
- **(d)** Remove this bubble and the edge connecting P to its parent from $\mathcal{T}$.

End_while

(2) Make $\mathcal{T}$ into a bubble and terminate.

Figure 2: Algorithm PARTITION($\mathcal{T}, x$).

Lemma 4.1 *Algorithm* PARTITION($\mathcal{T}_R, x$) *produces an $[x, \delta x]$-bubbles partition of G.*

Proof: Obviously, each bubble constructed in step (1c) is of size at least x and no larger than δx.

We now show that every execution of the loop in step (1) succeeds in finding a candidate node P as required in (1b). Since Step (1) is performed only as long as $M_R > \delta x$, R itself trivially satisfies the first requirement, namely, $M_R > x$. It now suffices to show that every tree $\mathcal{T}_Q$ of size $M_Q \geq x$ has a node P as required. This is proved by simple induction on the size of the tree, beginning at size x. For the inductive step we note that if Q has no child with more than x nodes in its subtree $\mathcal{T}_Q$ then Q can be taken as the required P, and otherwise, let S be the child of Q with $M_S \geq x$ and apply the inductive hypothesis to $\mathcal{T}_S$, to find a suitable node P in it.

The fact that any execution of the loop in step (1) results in a new bubble and that $\mathcal{T}_R$ is finite implies the termination of the algorithm.

Finally, we show that when the algorithm terminates, $x \leq M_R \leq \delta x$, hence the last bubble is legal as well. The termination condition of the loop implies that when it terminates, $M_R \leq \delta x$. If the number of nodes in the entire graph G is also smaller than δx then this is the only bubble, and we are done (since $x \leq n$ by choice of x). Otherwise, there exists at least one more bubble. Examine the last removal executed at step (1d) before the termination of the algorithm. Right before this removal, M_R is at least $\delta x + 1$, and as a result of the removal, no more than $(\delta - 1)(x - 1) + 1$ nodes are removed from $\mathcal{T}_R$, leaving at least $\delta x + 1 - (\delta - 1)x + (\delta - 1) - 1 = x + \delta - 1 \geq x$ nodes that are still connected to R. ■

Theorem 4.2 is implied by the existence of the partition algorithm.

Theorem 4.2 *For every n-node graph G with maximum degree δ and parameter $1 \leq x \leq n$, G has an $[x, \delta x]$-bubble partition.* ■

Next we define special hierarchies of bubbles partitions. Given two partitions $\mathcal{P}$ and $\mathcal{P}'$, we say that $\mathcal{P}$ is a *refinement* of $\mathcal{P}'$ if each bubble of $\mathcal{P}$ is fully contained in some bubble of $\mathcal{P}'$ (or equivalently, each bubble of $\mathcal{P}'$ is partitioned into bubbles of $\mathcal{P}$).

Definition 4.2 *Given a graph G and a list of k integer parameters $\bar{x} = (x_1, x_2, \cdots, x_k, x_{k+1})$, where $x_1 = n$, $x_i > \delta x_{i+1}$ and $x_{k+1} = 1$, an $\bar{x}$-*hierarchical bubbles partition *of G is a collection of k partitions $\mathcal{P}_i$, $1 \leq i \leq k$, with the following properties.*

1. *For every $1 \leq i \leq k$, $\mathcal{P}_i$ is an $[x_i, \delta x_i]$-bubbles partition.*
2. *For every $1 < i \leq k$, $\mathcal{P}_i$ is a* refinement *of $\mathcal{P}_{i-1}$.*

In what follows, we choose the parameters $\bar{x}$ as $x_i = \lceil n^{(k-i+1)/k} \rceil$. This choice satisfies the requirements of the definition, so long as $n^{1/k} \geq \delta$. We use $\mathcal{B}_i$ to denote a bubble at level i. For every bubble $\mathcal{B}_i$ let $\mathcal{T}(\mathcal{B}_i)$ be a representation of the subtree of the spanning tree $\mathcal{T}$ that spans the bubble $\mathcal{B}_i$. (Hereafter we refer to both the description of the spanning tree and the spanning tree itself by $\mathcal{T}(\mathcal{B}_i)$.) Thus each of the k partitions $\mathcal{P}_i$, $1 \leq i \leq k$, defines a *forest* $\mathcal{F}_i = \bigcup_{\mathcal{B} \in \mathcal{P}_i} \mathcal{T}(\mathcal{B})$. Note that $\mathcal{F}_1$ is identical to the spanning tree $\mathcal{T}$.

We note the following properties of the hierarchical partition. Since each bubble of level i is composed of the union of some bubbles of level $i+1$, the spanning tree $\mathcal{T}(\mathcal{B}_i)$ is a combination of the spanning trees of those bubbles. Hence $\mathcal{F}_{i+1} \subseteq \mathcal{F}_i$ for every $1 \leq i \leq k-1$, and $\mathcal{F}_1$, corresponding to the single bubble containing the whole network, is the entire tree $\mathcal{T}$.[2] These properties will be maintained as invariants of the hierarchical partition throughout the updates performed upon topology changes, and are important for the analysis of the scheme, as will be shown later on.

The hierarchical bubble partition of a graph is constructed level by level, by Algorithm Hierarchy($\bar{x}$) described in Fig. 3.

(1) Choose an arbitrary node of the graph, R.

(2) Construct a spanning tree $\mathcal{T}_R$ rooted at R with edges directed towards the leaves.

(3) Let the first level partition $\mathcal{P}_1$ consist of a single bubble covering the entire graph.

(4) **For** $i = 2$ to k **do:**

For every bubble $\mathcal{B}$ of partition $\mathcal{P}_{i-1}$ **do:**

- **(a)** Let $\mathcal{T}(\mathcal{B})$ be the portion of $\mathcal{T}_R$ that spans the bubble $\mathcal{B}$.
- **(b)** Apply Algorithm Partition($\mathcal{T}(\mathcal{B}), x_i$) to $\mathcal{T}(\mathcal{B})$.
- **(c)** Add the resulting bubbles to the current level partition $\mathcal{P}_i$.

Figure 3: Algorithm Hierarchy($\bar{x}$).

It is clear that $\mathcal{P}_1$ is an $[n, \delta n]$-bubbles partition. Since for every $i > 1$ the partition $\mathcal{P}_i$ is obtained from $\mathcal{P}_{i-1}$ by splitting each bubble $\mathcal{B}$ separately using Algorithm Partition($\mathcal{T}(\mathcal{B}), x_i$), the requirements of definition 4.2 are clearly met, establishing the following:

Theorem 4.3 *Every graph has a $\bar{x}$-hierarchical bubbles partition.* ■

Figure 13 depicts a partition of a graph into bubbles. First a spanning tree is constructed (the upper part of the figure) then the spanning tree is partitioned into bubbles of level two and three (lower parts of the figure).

4.2 Dynamic Maintenance of the Partition

In order to use the Bubbles partition over time in a dynamically changing environment, it is necessary to be able to maintain a legal partition in the presence of link and node failures and recoveries. One point that we need to address first when dealing with the dynamic case is that of failures that disconnect the network. Our policy in such a case is to treat each connected component separately, but still using the global parameters $\bar{x}$. This has the implication, that certain connected components cannot be partitioned in some of the lower levels of the hierarchy, since they are too small. In particular, if a certain connected component G' of the graph has

[2] We say that $\mathcal{F}_{i+1} \subseteq \mathcal{F}_i$ when every tree in $\mathcal{F}_{i+1}$ is a subtree in $\mathcal{F}_i$.

less than x_i (but at least x_{i+1}) nodes, then on levels 1 through i there is a single bubble encompassing the entire connected component, and this bubble is said to be *illegal*, since it is smaller than the allowed lower bound. The implication of such situation is that once a connecting link recovers, it is necessary to reorganize the partition in order to "legalize" the bubble structure.

We now present the strategy for handling each topology change. For now, the strategy we describe is centralized, as if an outside operator changes the distributed routing data-base. Later, in Section 7, we discuss a distributed implementation of our maintenance strategy. We use the following definition.

Definition 4.3 *Two bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$ both at level $l > 1$ are said to be* tree-neighboring *w.r.t. the level $l-1$ forest $\mathcal{F}_{l-1}$ if there exists an edge of $\mathcal{F}_{l-1}$ that connects them. We refer to this tree link as the* connecting link *of $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$.*

Our maintenance activities are based on employing two basic operations, or "procedures", COMBINE($\mathcal{B}'_l, \mathcal{B}''_l$) and SPLIT($\mathcal{B}_l$), described in Figures 4 and 5, responsible for combining and splitting bubbles, respectively. These two basic procedures are used as building blocks for our update operations.

Input: Two tree-neighboring level l bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$

Action:
Connect $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$ using their connecting link.

Output: A combined bubble $\mathcal{B}_l$.

Figure 4: Procedure COMBINE($\mathcal{B}'_l, \mathcal{B}''_l$).

Input: A (typically oversized) bubble $\mathcal{B}_l$.

Action:
Select one of the nodes of $\mathcal{B}_l$ to be the root R.
Direct $\mathcal{T}(\mathcal{B}_l)$ from R towards the leaves.
Execute Algorithm PARTITION($\mathcal{T}(\mathcal{B}_l), x_l$).

Output: An $[x_l, \delta x_l]$-bubbles partition of $\mathcal{B}_l$.

Figure 5: Procedure SPLIT($\mathcal{B}_l$).

Let us next describe another procedure, named Procedure REMOVE($e, \mathcal{F}_l, \mathcal{F}_{l-1}$), that handles the removal of a link e from the forest of spanning trees of the level $l > 1$ bubbles partition, $\mathcal{F}_l$. This removal can be the result of either the link's failure, or the restructuring of bubbles at the next lower level, $l-1$, that caused the removal of this link from $\mathcal{F}_{l-1}$. In any case, it is assumed that e does not appear in $\mathcal{F}_{l-1}$. The computation of the new forest $\mathcal{F}'_l$, that no longer contain e, is based on the current forest for level $l-1$, $\mathcal{F}_{l-1}$, in the sense that the only edges that are added to $\mathcal{F}'_l$ during the procedure's execution are also edges of $\mathcal{F}_{l-1}$. Procedure $\text{Remove}(e, \mathcal{F}_l, \mathcal{F}_{l-1})$, described next in Figure 6, relies on the assumption that both $\mathcal{F}_{l-1}$ and $\mathcal{F}_l$ span legal bubbles partitions (of level $l-1$ and l, respectively). However, it is not assumed that $\mathcal{F}_l$ is a refinement of $\mathcal{F}_{l-1}$.

Input: Edge e for removal, forests $\mathcal{F}_l$ and $\mathcal{F}_{l-1}$ spanning legal partitions.

Action:

The removal of e causes the partition of a single bubble, say $\mathcal{B}_l$, into two portions, denoted $\mathcal{B}'_l$ and $\mathcal{B}''_l$.

For each of these two portions $\hat{\mathcal{B}}$ **do:**

1. If the size of $\hat{\mathcal{B}}$ is still greater than x_l, then make it into an independent bubble without change.
2. Otherwise, if it is too small, then do the following:
   (a) Find some bubble $\tilde{\mathcal{B}}$ of $\mathcal{F}_l$ that is a tree-neighbor of $\hat{\mathcal{B}}$ w.r.t. $\mathcal{F}_{l-1}$.
   (b) Merge $\tilde{\mathcal{B}}$ and $\hat{\mathcal{B}}$ into $\bar{\mathcal{B}}$ by invoking procedure $\text{Combine}(\tilde{\mathcal{B}}, \hat{\mathcal{B}})$.
   (c) If $\bar{\mathcal{B}}$ includes more than δx_l nodes, then split it by invoking procedure $\text{Split}(\bar{\mathcal{B}})$.

Output: Forest $\mathcal{F}'_l$.

Figure 6: Procedure $\text{Remove}(e, \mathcal{F}_l, \mathcal{F}_{l-1})$.

It is important to understand that procedure Remove ends up not only removing the specified link e from $\mathcal{F}_l$, but also removing some *additional* links, as well as adding some links to $\mathcal{F}_l$, in its attempt to regain the legality of the partition. In particular, the following scenario may take place during the operation of removing e on level l. Suppose the removal of e breaks bubble $\mathcal{B}_l$ into two portions. We then apply Combine operations to the broken portions, combining them with tree-neighboring level l bubbles. The resulting combined bubbles might be too large, in which case we apply a Split operation to each of them, and thus remove additional edges from each.

In the next lemma we establish some basic properties of procedure Remove, and in particular, prove that the above mentioned phenomenon is bounded.

Lemma 4.4 *(1) If $\mathcal{F}_l$ is the spanning forest of a legal $[x_l, \delta x_l]$-bubbles partition, then after the execution of procedure* $\text{Remove}(e, \mathcal{F}_l, \mathcal{F}_{l-1})$ *and removing the edge e, the output forest $\mathcal{F}'_l$ represents a legal $[x_l, \delta x_l]$-bubbles partition as well.*
(2) Every edge added to $\mathcal{F}'_l$ during the execution of $\text{Remove}(e, \mathcal{F}_l, \mathcal{F}_{l-1})$ *is an edge of $\mathcal{F}_{l-1}$.*
(3) The number of edges removed from $\mathcal{F}_l$ during the execution of $\text{Remove}(e, \mathcal{F}_l, \mathcal{F}_{l-1})$ *is at most three.*

Proof: It is easy to verify (by simple case by case inspection) that, assuming we start with a legal bubbles partition, then each bubble generated during the update process is connected, and the different bubbles are disjoint and cover the entire graph.

For proving (1), we need to argue that in the output partition, every bubble is in the right size range. For establishing the upper size bound, $|\mathcal{B}| \le \delta x_l$, we rely on the fact that whenever the algorithm merges a bubble and it becomes too large, it is split.

For proving that $|\mathcal{B}| \ge x_l$, we rely on the fact that whenever the algorithm creates a bubble $\hat{\mathcal{B}}$ that is too small (as the result of the edge removal), it is combined with a tree-neighboring bubble $\tilde{\mathcal{B}}$, such that their combined size is at least x_l at this time.

The only point that needs to be verified in order to complete the above proof, is that whenever some portion $\hat{\mathcal{B}}$ of a broken bubble is too small, and needs to be combined with a tree-neighboring bubble, then the existence of a suitable tree-neighboring bubble $\tilde{\mathcal{B}}$ is guaranteed.

To see this, let $\mathcal{B}_{l-1}$ be a level $l-1$ bubble that includes at least one node of $\hat{\mathcal{B}}$, say v. Since level $l-1$ bubbles are always larger than level l bubbles, $\mathcal{B}_{l-1}$ is a connected component that includes more nodes than $\hat{\mathcal{B}}$ does, hence there must be a node w that is not within $\hat{\mathcal{B}}$ but is within $\mathcal{B}_{l-1}$. A path from v to w in $\mathcal{T}(\mathcal{B}_{l-1})$ must include a link connecting $\mathcal{B}'_l$ to a tree-neighboring level l bubble.

Property (2) is immediate from the procedure and the definition of a tree-neighboring bubble.

It remains to prove property (3), namely, to show that the procedure removes at most two more edges in addition to the intended e itself. To see this, observe that if a bubble $\bar{\mathcal{B}}$ is created in step 2 by combining bubbles $\tilde{\mathcal{B}}$ and $\hat{\mathcal{B}}$, then $|\hat{\mathcal{B}}| \le x_l$ and $|\tilde{\mathcal{B}}| \le \delta x_l$, hence $|\bar{\mathcal{B}}| \le (\delta+1)x_l$. This implies that Procedure SPLIT($\bar{\mathcal{B}}$) will erase a single edge, splitting $\bar{\mathcal{B}}$ into precisely two new bubbles. This is because the very first splitting operation in Procedure PARTITION($\mathcal{T}(\bar{\mathcal{B}}), x_l$) will create a bubble of size at least x_l, leaving the remaining portion of the bubble $\bar{\mathcal{B}}$ with fewer than δx_l nodes and thus terminating the procedure. Since steps 1 and 2 of Procedure REMOVE are executed at most twice, the claim follows. ■

The observation made just before the lemma, combined with Claim (3) of the lemma, has the following important implication. Consider a single topology change, in the form of the failure of a link e_1, requiring us to remove e_1 on some level l (and possibly also in other levels). This change might in fact cause us to remove up to two other edges, e_2 and e_3, on level $l+1$. But removing each of these three edges e_j on level $l+1$ may in turn cause the removal of two other edges, resulting in a total of nine removals on level 2. More generally, this single topology change might result in up to (but no more than) 3^{i-1} link removal operations in each level i.

Consequently, whenever we need to remove an edge from the partition of some level i, we typically need to immediately fix the partitions of all higher levels, $i+1$ through k, as well. This is done via procedure REMOVEALL. The procedure fixes the Bubbles partitions $\mathcal{P}_l$ at each level $i \le l \le k$, one at a time, starting from level i up. At each level l, the procedure outputs a corrected forest $\mathcal{F}'_l$. The computation of the new bubble partition of level l, $\mathcal{P}'_l$, and the corresponding forest $\mathcal{F}'_l$, is based on the recently computed forest for level $l-1$, $\mathcal{F}'_{l-1}$, and the old forest $\mathcal{F}_l$ of level l before the topology change. Suppose the procedure already computed the new partition up to and including level $l-1$. As a result, there is a number of edges that appear in $\mathcal{F}_l$ and do not appear in $\mathcal{F}'_{l-1}$ (since they have already been removed on that level). The procedure removes each tree link of $\mathcal{F}_l - \mathcal{F}'_{l-1} = \{e_1, e_2, \cdots, e_m\}$ from $\mathcal{F}_l$. This is done sequentially, edge by edge, and the forest $\mathcal{F}_l$ is modified at each edge removal step to reflect the change. Hence the sequence of link removals creates a sequence of forests $\mathcal{F}_l = \mathcal{F}_l^{(0)}, \mathcal{F}_l^{(1)}, \ldots, \mathcal{F}_l^{(m)} = \mathcal{F}'_l$, where the update for removing e_j leads from $\mathcal{F}_l^{(j-1)}$ to $\mathcal{F}_l^{(j)}$.

It remains to describe the modification of the forest $\mathcal{F}_l^{(j-1)}$ due to the removal of e_j. If e_j is not in $\mathcal{F}_l^{(j-1)}$, then nothing need be done. Otherwise, if the removal of e_j causes the partition of some bubble, then we invoke procedure $\textsc{Remove}(e_j, \mathcal{F}_l^{(j-1)}, \mathcal{F}_{l-1})$ to remove the edge and correct $\mathcal{F}_l^{(j-1)}$ into $\mathcal{F}_l^{(j)}$. Again, it is assumed that e_j does not appear in $\mathcal{F}_{l-1}$, and all edges added to $\mathcal{F}'_l$ or higher partitions during the procedure's execution belong to $\mathcal{F}_{l-1}$. The procedure relies on the assumption that initially, $\mathcal{F}_l$ (for every $i-1 \leq l \leq k$) spans a legal bubbles partition. Procedure $\textsc{RemoveAll}$ is described formally in Figure 7.

Input: Edge e for removal,
forests $\mathcal{F}_l$ for $i-1 \leq l \leq k$ spanning legal partitions.

Action:

Let $\mathcal{F}'_i \leftarrow \textsc{Remove}(e, \mathcal{F}_i, \mathcal{F}_{i-1})$.

For $l = i+1$ to k **do:**

1. Let $\{e_1, e_2, \cdots, e_m\} = \mathcal{F}_l - \mathcal{F}'_{l-1}$.
2. Let $\mathcal{F}_l^{(0)} \leftarrow \mathcal{F}_l$.
3. **For** $j = 1$ to m **do:**
   If e_j is in $\mathcal{F}_l^{(j-1)}$, then invoke $\mathcal{F}_l^{(j)} \leftarrow \textsc{Remove}(e_j, \mathcal{F}_l^{(j-1)}, \mathcal{F}_{l-1})$.
4. Let $\mathcal{F}'_l \leftarrow \mathcal{F}_l^{(m)}$.

Output: Forests $\mathcal{F}'_l$ for $i \leq l \leq k$.

Figure 7: Procedure $\textsc{RemoveAll}(e, i)$.

Lemma 4.5 *(1) If $\mathcal{F}_l$ is the spanning forest of a legal $[x_l, \delta x_l]$-bubbles partition for every $i-1 \leq l \leq k$, then after the execution of procedure* $\textsc{RemoveAll}(e, i)$ *and removing the edge e, the output forest $\mathcal{F}'_l$ represents a legal $[x_l, \delta x_l]$-bubbles partition as well, for every $i \leq l \leq k$.*
(2) the output partition $\mathcal{P}'_l$ is a refinement of $\mathcal{P}'_{l-1}$ (i.e., $\mathcal{F}'_l \subseteq \mathcal{F}'_{l-1}$) for every $i \leq l \leq k$.
(3) The number of edges removed from $\mathcal{F}_l$ during the execution of $\textsc{RemoveAll}(e, i)$ *is at most 3^{l-i+1}, for every $i \leq l \leq k$.*

Proof: Claim (1) follows directly from claim (1) of Lemma 4.4. Moreover, this property holds throughout the process. That is, after the procedure completes handling levels 1 through l and works on level $l+1$, in the internal stage after the procedure has completed handling the removal of the first j edges, and has reached the intermediate forest $\mathcal{F}^{(j)}$, the claim holds for this $\mathcal{F}^{(j)}$.

Claim (2) holds only for the final partition constructed for each level. That is, in intermediate points along the computation of the procedure for some level $l+1$, the intermediate level $l+1$ forest $\mathcal{F}_{l+1}^{(j)}$ might not obey this relationship w.r.t. $\mathcal{F}_l$ (i.e., it may not be that $\mathcal{F}_{l+1}^{(j)} \subseteq \mathcal{F}_l$).

For the final partitions, the claim follows from the fact that in constructing the new level l forest $\mathcal{F}'_{l+1}$, the procedure removes from the original level $l+1$ forest $\mathcal{F}_{l+1}$ any edge of $\mathcal{F}_{l+1} - \mathcal{F}'_l$.

Claim (3) follows directly from claim (3) of Lemma 4.4. ■

Note that the lemma above holds for the case of legal bubbles i.e., the connected component includes at least x_l nodes. Similar arguments hold for the case in which the connected component is smaller.

Until this stage we described the algorithm to construct the bubble partition and basic procedures for maintaining the partition. Now we detail how those procedures are invoked in each of the four possibilities for topology changes.

Link Removal: If the removal of the link e does not partition the spanning tree of any bubble then no change of the hierarchical bubbles partition is required. If the spanning tree of a bubble is disconnected by removing e, then the link removal is handled at each level starting from level one up. At level one, if the communication graph is still connected then a non-tree link is promoted to be a tree link in order to retain the connectivity of the entire spanning tree, $\mathcal{T}$. The removal itself is then performed by applying procedure RemoveAll$(e, 2)$.

Link Addition: An addition of a link that does not connect two previously separated connected components does not change the routing data base. An addition of a link that does connect two previously disconnected components G' and G'' is handled as follows. Let $\mathcal{T}(\mathcal{B}'_1)$ and $\mathcal{T}(\mathcal{B}''_1)$ be the spanning trees of G' and G'', respectively, before the link addition. Connect $\mathcal{T}(\mathcal{B}'_1)$ with $\mathcal{T}(\mathcal{B}''_1)$ by invoking procedure Combine$(\mathcal{B}'_1, \mathcal{B}''_1)$ using the new link, to form a spanning tree $\mathcal{T}(\mathcal{B}_1)$ of the new connected communication graph.

Continue with levels 2 up to k one at a time. For level $2 \leq i \leq k$, if the new link connects two legal bubbles then no update operations are triggered by level i. Otherwise, when the new link connects at least one illegal bubble, then combine the illegal bubble with the new tree-neighboring bubble (using procedure Combine), and split the combined bubble (using procedure Split) if necessary. The split operation may result in the need to remove a link e at the next level. This is handled by invoking the link removal procedure RemoveAll(e, i) described above. Then the update for level $i+1$ may be started.

Node Addition or Removal: Both the addition and the removal of a node can be described in terms of addition and removal of links. The addition is handled by first adding a link that connects two separated connected components one of which is the single node. Then adding the rest of the links one by one. Node removal is done by removing one link at a time and then removing the node.

In order to argue about the correctness of the dynamic update procedures, two main claims need to be established, namely, that at the end of each update operation, each partition $\mathcal{P}_i$

is a legal $[x_i, \delta x_i]$-bubbles partition, and that the entire hierarchy is a hierarchical bubbles partition. These claims are naturally implied by the properties established earlier regarding procedures REMOVE and REMOVEALL.

Theorem 4.6 *At the end of each update operation, each partition $\mathcal{P}_i$ is a legal $[x_i, \delta x_i]$-bubbles partition and the entire hierarchy is a hierarchical bubbles partition.*

Proof: We prove the theorem for link removal and addition updates since node additions or removals are reduced to links removals or additions. Link removal employs the procedure REMOVEALL. By Lemma 4.5 the result of the procedure is a legal bubble partition. In the case of link addition, each COMBINE execution is followed by SPLIT and REMOVEALL (if required) to result in a legal partition. ■

4.3 The Routing Scheme

Next we describe the distributed routing data-base by the use of the bubble partition.

Routing Distributed Data-base: Start by constructing a hierarchical bubbles partition for the network. For every bubble $\mathcal{B}_k$ at level k, define the nodes that reside in $\mathcal{B}_k$ as its *members*, and choose one of these nodes to be the *leader* of the bubble, denoted $L(\mathcal{B}_k)$. In general, for level $i < k$, define the *members* of a bubble $\mathcal{B}_i$ to be all the leaders of level $i + 1$ bubbles that reside in the connected component of $\mathcal{B}_i$, and choose one of these members to be the bubble's leader, $L(\mathcal{B}_i)$. (The single bubble of level one, $\mathcal{B}_1$, will have no use for a leader.)

In Figure 14 we illustrate the way the members of the bubbles are chosen. The members of a bubble at level three are the nodes that reside in the bubble (omitted from the figure description). For each bubble at level three a single leader is chosen (e.g., A, D, E) this leader is a member of the bubble of level two in which it resides. For every bubble at level two a leader is chosen among its members (e.g., D, G). These chosen leaders are members of level one. The tree $\mathcal{T}(\mathcal{B}_i)$ is known to every member of $\mathcal{B}_i$. Note that in particular, every member of $\mathcal{B}_1$ maintains in its memory $\mathcal{T}(\mathcal{B}_1)$, which is a spanning tree of the entire communication graph. Note that for load balancing reasons, the representation of the spanning tree can be extended to include the full topology of the bubble, if desired, thereby increasing the memory requirement by a factor of δ. The precise method of choosing paths to avoid congestion is out of the scope of this paper.

Routing Strategy: P delivers a message m to a processor Q by the following procedure:
Let $i_1 \geq k$ be the lowest level in which P is a member, and let $\mathcal{B}_{i_1}$ be its *home bubble.* Then P searches for Q in the spanning tree description $\mathcal{T}(\mathcal{B}_{i_1})$. If Q is not found in $\mathcal{T}(\mathcal{B}_{i_1})$ then P sends the packet $(path_{L_1}, Q, m)$ to $L_1 = L(\mathcal{B}_{i_1})$, the leader of $\mathcal{B}_{i_1}$.
This leader, L_1, repeats the process. I.e., let $i_2 < i_1$ be the lowest level in which L_1 is a member, and let $\mathcal{B}_{i_2}$ be its home bubble. Then L_1 searches for Q in the spanning tree description $\mathcal{T}(\mathcal{B}_{i_2})$, and if Q is not found then L_1 sends the packet $(path_{L_2}, Q, m)$ to $L_2 = L(\mathcal{B}_{i_2})$.
This process must terminate (at a member of $\mathcal{B}_1$ in the worst case) since every member of $\mathcal{B}_1$ has a spanning tree description of the entire communication graph, namely, $\mathcal{T}(\mathcal{B}_1)$.

Routing Update: Whenever we apply procedure $\textsc{Combine}(\mathcal{B}'_l, \mathcal{B}''_l)$, the members of the combined bubble have to be updated regarding the new tree. Likewise, whenever we apply $\textsc{Split}(\mathcal{B}_l)$, the members of the two split bubbles have to be updated. Finally, upon adding a new link for reconnecting the tree $\mathcal{T}$, every member of $\mathcal{B}_1$ (namely, every level 2 leader) is updated with the new tree $\mathcal{T}$, similar to any other COMBINE operation.

Analysis:

We now turn to analyzing the complexity of the routing scheme. Let us first establish some basic properties of the hierarchical bubbles partition. For uniformity of treatment, let us define $x_{k+1} = 1$. Recall that the size of each bubble at level i is in the range $[x_i, \delta x_i]$. Define $\pi_i = \delta x_i / x_{i+1}$. As shown in the next lemma, π_i serves as an estimate for the population of level i bubbles.

Lemma 4.7 *The number of members in a bubble of level i is at most π_i.*

Proof: A bubble $\mathcal{B}$ of level i has at most δx_i nodes. For $i = k$ every node is a member, and we are done. Now consider $i < k$. The nodes of $\mathcal{B}$ are partitioned into bubbles of level $i+1$, each of which contains at least x_{i+1} nodes. Hence the number of level $i+1$ bubbles contained in $\mathcal{B}$ is at most $\delta x_i / x_{i+1}$. The members of $\mathcal{B}$ are precisely the leaders of these bubbles. The claim follows. ■

Lemma 4.8 *The total number of members in all level i bubbles is at most n/x_{i+1}.*

Proof: For $i = k$ all nodes are members. For $i < k$, the members of level i bubbles are precisely all leaders of level $i+1$ bubbles. Their number is at most n/x_{i+1} since each such bubble is of size at least x_{i+1}. ■

The next three lemmas state the super-hop count, the memory requirement and the adaptability of the bubble routing scheme.

Lemma 4.9 *The super-hop count of the bubble routing scheme is bounded by k.*

Proof: In the worst case a processor P that is only a member in level k sends a packet to its bubble leader, Q, that is a member at level $k-1$, and so on. With no more than $k-1$ packets a member of level 1 is reached; this member knows $\mathcal{T}(\mathcal{B}_1)$ and sends a direct packet to the destination of the message. ■

Lemma 4.10 *The memory requirement for the bubble routing scheme is bounded from above by $n(\log n + \log \delta) \sum_{1 \le i \le k} \pi_i$.*

Proof: Every processor maintains a spanning tree of the lowest bubble it is a member of. The members of bubbles at level i maintain a spanning tree of at most δx_i nodes. This requires $\delta x_i (\log n + \log \delta)$ bits. Since there are at most n/x_{i+1} members altogether at level i, the memory requirement for the ith level is $\delta x_i (\log n + \log \delta) \cdot n/x_{i+1} = n(\log n + \log \delta)\pi_i$ bits. ■

Lemma 4.11 *The adaptability of the bubble routing scheme is bounded from above by $\delta \sum_{1\le i\le k} 3^{i-1}\pi_i$, and only $\sum_{1\le i\le k} 3^{i-1}\pi_i$ in the link-failure model.*

Proof: The number of processors that change their routing data-base is greatest when a processor is removed. This number is bounded from above by the effect of removal of δ links one at a time.

The removal of a link that partitions a spanning tree of a bubble at every level implies the largest number of updates. On level 1, such a link removal requires at most n/x_2 updates of the members of $\mathcal{B}_1$.

We continue our analysis according to the description of the link removal procedure. Note that by Claim (3) of Lemma 4.5, at most 3^{i-1} edges are removed at level i as the result of a single edge removal at level 1. Since each link belongs to at most one bubble on each level, this bounds also the number of level i bubbles whose topology has undergone changes as a result of this removal.

The routing data-base must be updated at every member of a bubble whose topology was changed. Hence the total number of members (on all levels) whose routing data-base must be updated upon a link removal is bounded by $\sum_{1\le i\le k} 3^{i-1}\pi_i$.

The total number of processors that are updated upon a node removal is no more than $\delta \sum_{1\le i\le k} 3^{i-1}\pi_i$. Both claims of the lemma follow. ■

We now fix $x_i = \lceil n^{(k-i+1)/k} \rceil$ for $1 \le i \le k$. Then $\sum_{1\le i\le k} \pi_i = k\delta n^{1/k}$, and $\sum_{1\le i\le k} 3^{i-1}\pi_i = \frac{1}{2}(3^k - 1)\delta n^{1/k}$. The following theorem summarizes the properties of the bubbles routing scheme.

Theorem 4.12 *The bubble routing scheme has the following properties:*
(1) Super-hop count $= k$.
(2) Memory requirement $= k\delta n^{1+1/k}(\log n + \log \delta)$.
(3) Adaptability $= 3^k\delta^2 n^{1/k}$ in the node-failure model, and $3^k \delta n^{1/k}$ in the link-failure model. ■

5 Improved Partitioning: Edge-Bubbles Partition

The Bubbles routing scheme as described above is efficient when δ is small, and in particular, for bounded degree networks there is a little significance to this factor. However, if δ is large (close to n), and we consider only link failures, the Bubbles scheme may not compare favorably even with the simple Multiple Trees scheme, because of the occurrences of δ in the expression for adaptability (and in the expression for memory). For such a case it is desirable to get rid of the dependence of the complexities on δ. Note that a δ factor for the adaptability is inherent to the problem if we consider a model that allows node failures and recoveries. This δ factor is based on the fact that a node addition can cause the connection of δ previously disconnected components.

We now present a modified partitioning scheme called the *edge-bubbles partition*, that eliminates a δ factor (occurring in both the bound of adaptability and memory) at the cost of using a somewhat more complex partition and algorithm. Recall that in order to partition the communication graph into node disjoint connected components, the bubble scheme presented above allows the range of the bubble size to be $[x, \delta x]$. It turns out that it is possible to form a bubble-like partition of a tree into *edge-disjoint* trees (i.e., with bubbles possibly sharing nodes), such that the size (i.e., number of edges) of each tree is between x and $3x$.

Definition 5.1 *Given a graph $G = (V, E)$ and parameters $0 < a < b$, an $[a, b]$* edge-bubbles partition *of G is a (possibly node-overlapping) cover of V by a collection of edge-disjoint* connected *trees called* bubbles, *$\{\mathcal{B}_1, \ldots, \mathcal{B}_q\}$, where $\mathcal{B}_i = (V_i, E_i)$, with the following property: $E_i \cap E_j = \emptyset$ for every $1 \le i < j \le q$.*

Intuitively, the modified scheme can be thought of as utilizing the idea of transforming the tree $\mathcal{T}$ at hand into a *logical* tree $\hat{\mathcal{T}}$ of maximum degree 3, and applying the previous solution to this logical tree. The transformation can be achieved by splitting each node P of degree d into d copies $P_1, \ldots, P_d$, connected by a simple chain, and hanging exactly one child Q_i of P off each copy P_i. (See Fig. 15.)

The bubbles solution on the logical tree is simulated on the actual network by requiring each node P to simulate the behavior of its d logical copies. Some care is still needed with respect to the modification procedures, and in particular, it is necessary to specify how a node P reacts to topological changes in terms of reorganizing its internal (logical) structure, for example, as a result of a combine operation that changed its root (and made its former parent into a child, and one of its former children into its new parent). In what follows, the resulting algorithm is described directly (rather than via the intuitive simulation method discussed above), in order to gain better understanding concerning its behavior.

We next present an overview of a partitioning algorithm, that given a n-node graph G and a parameter $1 \le x \le n$, produces an $[x, 3x]$ edge-bubbles partition for G. The new algorithm is similar in nature to Algorithm PARTITION of Figure 2. The essential differences between the two algorithms are that in the new algorithm (i) M_P is the total number of *edges* in the subtree $\mathcal{T}_P$, and (ii) The treatment of a node P found in step (2) of the algorithm, namely, a node P satisfying $M_P \ge x$, such that all of P's children Q satisfy $M_Q < x$, is different. For a particular edge e from P to one of its children, Q, we define a subtree, $\mathcal{T}_e$, rooted at e to be the subtree rooted at Q together with e. Recall that the previous algorithm simply makes the subtree $\mathcal{T}_P$ rooted at P into one bubble, containing, in particular, all the subtrees rooted at P's children. In contrast, the new algorithm will select *some* of the subtrees $\mathcal{T}_e$ rooted at P's outgoing edges of total size greater than x but not exceeding $2x$. The selected subtrees are now merged into an edge-connected bubble. This bubble is removed from the tree. We note that the resulting bubbles are of size $[x, 3x]$ since the partition of the last bubble into two bubbles of size in the range $[x, 2x]$ might not be possible in the case P has three outgoing edges for which the subtree rooted at each of them is of size less than x but greater than say, $3x/4$.

We proceed with a more formal description of the algorithm. As in the previous algorithm, a preprocessing step involves constructing a downward-directed spanning tree $\mathcal{T}_R$ for the graph, rooted at an arbitrary node R. The algorithm then partitions this tree into bubbles as described next (in Fig. 8).

(1) While $\mathcal{T}$ contains more than $3x$ nodes **do:**

- **(a)** Mark every node P in $\mathcal{T}$ by M_P, the total number of edges in the subtree $\mathcal{T}_P$ rooted at P.
- **(b)** Find a node P with the property that $M_P \geq x$, but all of its children Q satisfy $M_Q < x$.
- **(c)** Let $E_P = (e_1, e_2, \cdots, e_i)$ be the edges connecting P to its children. Let n_i be the number of edges in the subtree connected to P by e_i (including e_i). Let j be the minimal index j such that $x \leq \Sigma_{i=1}^{j} n_i \leq 2x$.
- **(d)** Make the edges of the subtrees connected to P by $e_1, e_2, \cdots, e_j$ (including the edges $e_1, e_2, \cdots, e_j$ themselves) into a bubble.
- **(e)** Remove this bubble from $\mathcal{T}$.

End_while

(2) Make $\mathcal{T}$ into a bubble and terminate.

Figure 8: Algorithm EdgePartition(x).

Lemma 5.1 *Algorithm* EdgePartition(x) *produces an* $[x, 3x]$ *edge-bubbles partition of* G.

Proof: The proof is a variant of that of Lemma 4.1. Step (2c) guarantees, by the choice of j, that the selected collection is of total size no greater than $2x$.

Showing that every execution of step (1) succeeds in finding a candidate node P as required in (1b) follows just as in the proof of Lemma 4.1. Hence termination of the algorithm follows in the same way too.

Next, we show that when the algorithm terminates, $x \leq M_R \leq 3x$, hence the last bubble is within the correct size range as well. The termination condition of step (1) implies that when the algorithm terminates, $M_R \leq 3x$. If the number of nodes in the entire graph G is also smaller than $3x$ then this is the only bubble, and we are done. Otherwise, there exists at least one more bubble. Examine the last removal executed at step (2c) before the termination of the algorithm. Right before this disconnection M_R is greater than $3x$, and during the disconnection no more than $2x$ edges are removed from $\mathcal{T}_R$, leaving at least (a set of) x edges that are still connected to R (i.e. a path from R to each of the edges in this set [[[consists of]]] only edges of the set). ■

[[Theorem 5.2 is implied by the existence of the partition algorithm.]]

Theorem 5.2 *For every n-node graph G and parameter $1 \leq x \leq n$, G has an $[x, 3x]$ edge-bubbles partition.*

Figure 16 depicts a step in the partition of a graph into edge-bubbles. The tree of the (remaining) graph is depicted in the top portion of the figure. It is assumed that $x = 6$ and hence the partition takes place only in the node of level two for which every subtree has less than 6 edges. The partition gathered the first two subtrees to be a bubble of size greater than 6 and not exceeding 12.

Again, for our routing schemes we will use hierarchies of edge-bubbles partitions, defined as follows.

Definition 5.2 *Given a graph G and a list of k integer parameters $\bar{x} = (x_1, x_2, \cdots, x_k, x_{k+1})$, where $x_1 = n$, $x_{k+1} = 1$ and $x_i > 3x_{i+1}$, an $\bar{x}$-*hierarchical edge-bubbles partition *of G is a collection of k edge-bubbles partitions $\mathcal{P}_i$, $1 \le i \le k$, with the following properties.*
(1) For every $1 \le i \le k$, $\mathcal{P}_i$ is an $[x_i, 3x_i]$ edge-bubbles partition.
(2) For every $1 < i \le k$, $\mathcal{P}_i$ is a refinement of $\mathcal{P}_{i-1}$.

Similarly to the previous section we choose the parameters $\bar{x}$ as $x_i = \lceil n^{(k-i+1)/k} \rceil$.

The hierarchical edge-bubbles partition of a graph is constructed by an algorithm EdgeHier($\bar{x}$) very similar to algorithm Hierarchy($\bar{x}$) of Section 4, except that the partitioning algorithm invoked is EdgePartition rather than Partition. [[The requirements of definition 5.2 are clearly met, and thus:]]

Theorem 5.3 *Every graph has a hierarchical edge-bubbles partition.*

5.1 Dynamic Maintenance

Dynamic maintenance of the bubbles is carried out in much the same way as with the basic algorithm of Section 4.

The combine and split procedures for the edge partition bubbles appear in Figure 9 and Figure 10. We use the following definition for *edge-tree-neighboring.*

Definition 5.3 *Two bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$ both at level l are said to be* edge-tree-neighboring *w.r.t. the level $l-1$ forest $\mathcal{F}_{l-1}$ if there exist two edges $e' \in \mathcal{B}'_l$ and $e'' \in \mathcal{B}''_l$ adjacent to the same node such that both e' and e'' are in the same bubble of $\mathcal{F}_{l-1}$. We refer to the shared adjacent node of e' and e'' as the* connecting node *of $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$.*

It turns out that the order of the edges $e_1, e_2, \cdots e_i$ of step (1c) of Algorithm EdgePartition is crucial in the context of dynamic maintenance of the hierarchical partition. When a bubble of level l is split it might split bubbles of level $l+1$. We would like to minimize the number of bubbles at level $l+1$ that are split because a split of level l took place. Towards this end, we specify the order of the edges $e_1, e_2, \cdots e_i$ in a way that ensures that at most one bubble of level $l+1$ is split because of a split operation of a bubble in level l. The next definition uses e_0 to denote the edge connecting P with its parent in $\mathcal{T}$.

Definition 5.4 *The execution of the algorithm* EdgePartition *is said to be* ordered w.r.t. $\mathcal{F}_{l+1}$ *if in step (1c) of Algorithm* EdgePartition, *the edges* $E_P = (e_1, e_2, \cdots, e_i)$ *are ordered according to* $\mathcal{F}_{l+1}$, *in such a way that for every bubble* $\mathcal{B}_{l+1}$, *the edges of* E_P *that belong to* $\mathcal{B}_{l+1}$ *appear in one contiguous subsequence of* E_P. *In addition, if* e_0 *belongs to* $\mathcal{B}'_{l+1}$ *then the contiguous subsequence of edges in the ordered* E_P *that belong to* $\mathcal{B}'_{l+1}$ *(if any) is ordered last among the other contiguous subsequences.*

Input: Two edge-tree-neighboring level l bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$

Action: Connect $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$ using their connecting node.

Output: A combined bubble $\mathcal{B}_l$.

Figure 9: Procedure EdgeCombine($\mathcal{B}'_l, \mathcal{B}''_l$).

Input: A (typically oversized) bubble $\mathcal{B}_l$ and $\mathcal{F}_{l+1}$.

Action: Select one of the nodes of $\mathcal{B}_l$ to be the root R.
Direct $\mathcal{T}(\mathcal{B}_l)$ from R towards the leaves.
Execute steps (1) to (2) of the Algorithm EdgePartition(x_l) (ordered w.r.t. $\mathcal{F}_{l+1}$).

Output: An $[x_l, 3x_l]$ edge-bubbles partition of $\mathcal{B}_l$.

Figure 10: Procedure EdgeSplit($\mathcal{B}_l, \mathcal{F}_{l+1}$).

The procedures EdgeRemove and EdgeRemoveAll appear in figures 11 and 12. [[[The next Lemma states some basic properties of procedure Remove.]]]

Lemma 5.4 *(1) If* $\mathcal{F}_l$ *is the spanning forest of a legal* $[x_l, 3x_l]$ *edge-bubbles partition, then after the execution of procedure* EdgeRemove($e, \mathcal{F}_l, \mathcal{F}_{l-1}, \mathcal{F}_{l+1}$) *and removing the edge* e, *the output forest* $\mathcal{F}'_l$ *represents a legal* $[x_l, 3x_l]$ *edge-bubbles partition as well.*
(2) Every two bubbles combined into a single bubble of $\mathcal{F}'_l$ *during the execution of* EdgeRemove($e, \mathcal{F}_l, \mathcal{F}_{l-1}, \mathcal{F}_{l+1}$) *resides in a single bubble of* $\mathcal{F}_{l-1}$.

Proof: It is easy to verify (by simple case by case inspection) that, assuming we start with a legal bubbles partition, then each bubble generated during the update process is connected, and the different bubbles are disjoint and cover the entire graph.

For proving (1), we need to argue that in the output partition, every bubble is in the right size range. For establishing the upper size bound, $|\mathcal{B}| \leq 3x_l$, we rely on the fact that whenever the algorithm merges a bubble and it becomes too large, it is split.

Input: Edge e for removal, forest $\mathcal{F}_l$, $\mathcal{F}_{l-1}$ and $\mathcal{F}_{l+1}$ spanning a legal partition.

Action:

The removal of e_i causes the partition of a single bubble, say $\mathcal{B}_l$, into two portions, denoted $\mathcal{B}_l'$ and $\mathcal{B}_l''$. Let $\hat{\mathcal{B}}_l$ be an edge-tree-neighboring of $\mathcal{B}_l$ and w.l.o.g. of $\mathcal{B}_l'$.

For $\hat{\mathcal{B}} = \mathcal{B}_l'$ and then $\hat{\mathcal{B}} = \mathcal{B}_l''$ **do:**

1. If the size of $\hat{\mathcal{B}}$ is still greater than x_l, then make it into an independent bubble without change.
2. Otherwise, if it is too small, then do the following:
   (a) If $\hat{\mathcal{B}} = \mathcal{B}_l'$ then let $\tilde{\mathcal{B}}$ be $\hat{\mathcal{B}}_l$.
   Else find some bubble $\tilde{\mathcal{B}}$ of $\mathcal{F}_l$ that is an edge-tree-neighbor of $\hat{\mathcal{B}}$ w.r.t. $\mathcal{F}_{l-1}$.
   (b) Merge $\tilde{\mathcal{B}}$ and $\hat{\mathcal{B}}$ into $\bar{\mathcal{B}}$ by invoking procedure EdgeCombine($\tilde{\mathcal{B}}, \hat{\mathcal{B}}$).
   (c) If $\bar{\mathcal{B}}$ includes more than $3x_l$ nodes, then split it by invoking procedure EdgeSplit (ordered w.r.t. $\mathcal{F}_{l+1}$).

Output: Forest $\mathcal{F}_l'$.

Figure 11: Procedure EdgeRemove($e, \mathcal{F}_l, \mathcal{F}_{l-1}, \mathcal{F}_{l+1}$).

For proving that $|\mathcal{B}| \geq x_l$, we rely on the fact that whenever the algorithm creates a bubble $\hat{\mathcal{B}}_j$ that is too small (as the result of an edge removal operation), it is combined with a tree-neighboring bubble $\tilde{\mathcal{B}}$ that is of size of at least x_l.

The only point that needs to be verified in order to complete the above proof, is that whenever some portion $\hat{\mathcal{B}}$ of a broken bubble is too small, and needs to be combined with a tree-neighboring bubble, then the existence of a suitable tree-neighboring bubble $\tilde{\mathcal{B}}$ is guaranteed.

To see this, let $\mathcal{B}_{l-1}$ be a level $l-1$ bubble that includes at least one node of $\hat{\mathcal{B}}$, say v. Since level $l-1$ bubbles are always larger than level l bubbles, $\mathcal{B}_{l-1}$ is a connected component that includes more edges than $\hat{\mathcal{B}}$ does, hence there must be an edge w that is not within $\hat{\mathcal{B}}$ but is within $\mathcal{B}_{l-1}$. A path in $\mathcal{T}(\mathcal{B}_{l-1})$ from v to one of the nodes attached to w must include a node connecting $\hat{\mathcal{B}}$ to an edge-tree-neighboring level l bubble.

Property (2) is immediate from the procedure and the definition of a edge-tree-neighboring bubble. ■

[[[The next lemma uses Lemma 5.4 to prove some properties of procedure RemoveAll.]]]

Lemma 5.5 *(1) If $\mathcal{F}_l$ is the spanning forest of a legal $[x_l, 3x_l]$ edge-bubbles partition for every $i \leq l \leq k$, then after the execution of procedure* RemoveAll(e,i) *and removing the edge e, the output forest $\mathcal{F}_l'$ represents a legal $[x_l, 3x_l]$ edge-bubbles partition as well, for every $i \leq l \leq k$. (2) The output partition $\mathcal{P}_l'$ is a refinement of $\mathcal{P}_{l-1}'$ (i.e., $\mathcal{F}_l' \subseteq \mathcal{F}_{l-1}'$) for every $i \leq l \leq k$.*

Input: Edge e for removal, forests $\mathcal{F}_l$ for $i-1 \leq l \leq k$ spanning legal partitions.

Action:

Let $\mathcal{F}'_i \leftarrow$ EDGEREMOVE$(e, \mathcal{F}_i, \mathcal{F}_{i-1})$.

For $l = i+1$ to k **do:**

1. Let $\{\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_m\}$ be all the bubbles such that, $\mathcal{B}_j$, $1 \leq j \leq m$, resides in more than a single bubble of $\mathcal{F}_{l-1}$.
2. Let $\mathcal{F}_l^{(0)} \leftarrow \mathcal{F}_l$.
3. **For** $j = 1$ to m **do:**
   (a) Partition $\mathcal{B}_j$ into $\{\mathcal{B}_{j1}, \mathcal{B}_{j2}, \cdots, \mathcal{B}_{jx}\}$ such that each $\mathcal{B}_{j_y}$, $1 \leq y \leq x$, is the maximal portion of $\mathcal{B}_j$ that resides in a single bubble of $\mathcal{F}_{l-1}$.
   (b) **For** $y = 1$ to x **do:**
      i. Make $\mathcal{B}_{j_y}$ a bubble.
      ii. If the size of $\mathcal{B}_{j_y}$ is too small then find an edge-tree-neighboring bubble of $\mathcal{B}_{j_y}$, $\mathcal{B}'_j$, EDGECOMBINE$(\mathcal{B}_{j_y}, \mathcal{B}'_j)$. If the resulting bubble, $\mathcal{B}''_j$, is a too big bubble then EDGESPLIT$(\mathcal{B}''_j, \mathcal{F}_{l+1})$ (ordered w.r.t. $\mathcal{F}_{l+1}$).
4. Let $\mathcal{F}'_l \leftarrow \mathcal{F}_l^{(m)}$.

Output: Forests $\mathcal{F}'_l$ for $i \leq l \leq k$.

Figure 12: Procedure EDGEREMOVEALL(e, i).

Proof: Claim (1) follows directly from claim (1) of Lemma 5.4 and the EDGECOMBINE and EDGESPLIT procedures applied to the portions of the bubbles. Note that in the execution of EDGECOMBINE$(\mathcal{B}_{j_y}, \mathcal{B}'_j)$, $\mathcal{B}'_j$ is not a portion of $\mathcal{B}_j$ and thus is a legal sized bubble.

Moreover, as in Lemma 4.5, this property holds throughout the process, namely, for each intermediate forest $\mathcal{F}^{(j)}$.

Claim (2) holds (again as in Lemma 4.5) only for the final partition constructed for each level.

For the final partitions, the claim follows from the fact that in constructing the new level l forest $\mathcal{F}'_{l+1}$, the procedure handles every bubble portion. ■

The update operations are handled in a similar fashion to the first bubble partition operations, [[yielding:]]

Theorem 5.6 *At the end of each update operation, each partition $\mathcal{P}_i$ is a legal $[x_i, 3x_i]$edge$-$bubbles partition and the entire hierarchy is a hierarchical edge-bubbles partition.*

5.2 The Routing Scheme

The routing proceeds as in Section 4. The only points to note are the following.

A node with adjacent edges $e_1, e_2 \cdots e_j$ is a member of every bubble at level k for which one of its edges belongs. For each bubble of level k a single *leader-edge* is elected among the edges of the bubble — then one of the adjacent node of the leader-edge is elected to be the leader (node) of the bubble. Note that a single node may function as a leader of several (up to δ) bubbles.

The elected leader-edge and the elected leader (node) are members of the bubble of level $k-1$ in which the elected leader-edge resides. For each bubble of level $k-1$, $\mathcal{B}_{k-1}$, a single leader-edge is elected among its edge members. The elected edge and the leader of level $k-1$ attached to it are members of the bubble of level $k-2$ in which the elected edge resides. This membership definition is naturally extended to the next lower levels. Similarly, to the node partition bubbles, a member of a bubble maintains the spanning tree portion of the bubble.

Analysis:

The next three lemmas state the super-hop count, the memory requirement and the adaptability of the edge-bubble routing scheme. The end result is very similar to the analysis of Section 4, except that π_i should be redefined as $\pi_i = 3x_i/x_{i+1}$. The proof of the next lemma is identical to the proof of Lemma 4.9.

Lemma 5.7 *The super-hop count of the edge-bubble routing scheme is bounded by k.*

Lemma 5.8 *The memory requirement for the edge-bubble routing scheme is bounded from above by $n(\log n + \log \delta)\sum_{1\le i\le k}\pi_i$.*

Proof: For every edge e such that the lowest level bubble e is a member in is $\mathcal{B}_i$ there is an adjacent processor that maintains a description of $\mathcal{T}(\mathcal{B}_i)$. A bubble of level i includes at most $3x_i$ edges. The total number of bits required for such a description of a tree is $3x_i(\log n+\log\delta)$. Since there are at most n/x_{i+1} edge members altogether at level i, the memory requirements for the ith level is $3x_i(\log n + \log\delta)\cdot n/x_{i+1} = n(\log n + \log\delta)\pi_i$ bits. ∎

Lemma 5.9 *The adaptability of the edge-bubble routing scheme is bounded from above by $\delta\sum_{1\le i\le k}3^{i-1}\pi_i$, and only $\sum_{1\le i\le k}3^{i-1}\pi_i$ in the link-failure model.*

Proof: The number of processors that change their routing data-base is greatest when a processor is removed. This number is bounded from above by the effect of removal of δ links one at a time.

The removal of a link that partitions a spanning tree of a bubble at every level implies the largest number of updates. On level 1, such a link removal requires at most n/x_2 updates of the members of $\mathcal{B}_1$.

We continue our analysis according to the description of the link removal procedure. Since we use ordered partition whenever EdgeRemove is invoked a partition of a bubble at level l

results with partition of up to three bubbles of level $l+1$. The bubble in which the partition of level l occurred and two other neighboring bubbles.

Thus, at most 3^{i-1} bubbles are partitioned at level i as the result of a single edge removal at level 1. Hence the total number of members (on all levels) whose routing data-base must be updated upon a link removal is bounded by $\sum_{1\leq i\leq k} 3^{i-1}\pi_i$.

The total number of processors that are updated upon a node removal is no more than $\delta\sum_{1\leq i\leq k} 3^{i-1}\pi_i$. Both claims of the lemma follow. ■

[[We now fix $x_i = \lceil n^{(k-i+1)/k}\rceil$ for $1 \leq i \leq k$. Then $\sum_{1\leq i\leq k}\pi_i = k\delta n^{1/k}$, and $\sum_{1\leq i\leq k} 3^{i-1}\pi_i = \frac{1}{2}(3^k-1)\delta n^{1/k}$. The following theorem summarizes the properties of the edge-bubbles routing scheme.]]

Theorem 5.10 *The edge-bubbles routing scheme has the following properties:*
(1) Super-hop count $= k$.
(2) Memory requirement $= kn^{1+1/k}(\log n + \log\delta)$.
(3) Adaptability $= 3^k\delta n^{1/k}$ *in the node-failure model, and* $3^k n^{1/k}$ *in the link-failure model.* ■

6 Lower Bound

In this section we prove a lower bound on the adaptability for graphs with bounded degree δ and super-hop count k. For the purpose of establishing the lower bound, we consider a single destination for all messages, and then derive a tree of superhops, rather than of links.

Given any source-oblivious routing scheme with super-hop count k we build a spanning tree T_k as follows. The root of the spanning tree is a processor P. Assume that every processor is to send a message m to P, and connect each processor Q with the destination of its first packet when Q sends m to P. The *depth* of a tree is the maximal number of edges from the root to a leaf.

Claim 6.1 *T_k is of depth no more than k.*

Proof: Otherwise the super-hop count is greater than k. ■

Claim 6.2 *There exists at least one node with at least $(n/2)^{1/k}$ children in T_k.*

Proof: There are n processors in T_k and the depth of T_k is k. Letting Δ denote the maximum out-degree of T_k, the number of nodes in the tree satisfies $n \leq \sum_{0\leq i\leq k}\Delta^i \leq 2\Delta^k$ (as $\Delta \geq 2$, implied by $n > k$). Hence $\Delta \geq (n/2)^{1/k}$. ■

Theorem 6.3 *Any source oblivious routing scheme with super-hop count k has adaptability $\Omega(n^{1/k})$ in a communication graph of constant degree.*

Proof: Let Q be a processor that has at least $(n/2)^{1/k}$ children in T_k. The existence of Q is implied by Claim 6.2. The node Q is connected to the rest of the processors in G by at most δ links. Thus, by the pigeon-hole principle there must be a link, (Q, R), used by a set C of at least $(n/2)^{1/k}/\delta$ processors in sending their first packet to carry the message to P. We now show a sequence of at most three topology changes involving *edge* removals and additions only, that implies $(n/2)^{1/k}/(3\delta)$ adaptability.

If a disconnection of (Q, R) does not partition the communication graph then obviously this topology change alone forces all $(n/2)^{1/k}/\delta$ processors of the set C to change their routing data base.

Now we analyze the case in which the removal of the link (Q, R) does partition the graph. We need to show that there exists a sequence of changes that preserve the bound on the degree δ and forces an omission of the link (Q, R) from the routing tables. We claim there are two different cases.

Case 1: Before the disconnection of (Q, R), the graph G is a tree, and the number of processors is greater than two. In this case, following the disconnection of (Q, R) there must be at least one leaf $X \notin \{Q, R\}$, say w.l.o.g. in the connected component of Q. Then one more topology change, involving the reconnection of X to R, is possible without exceeding δ. As a result of this change, the graph becomes connected again, and the nodes of the set C are forced to change their routing data-base to use the resulting tree for reaching P.

Case 2: Before the disconnection of (Q, R), the graph G contains at least one cycle. In this case , following the disconnection one of the connected components must contain a cycle (otherwise no partition would take place). Let (X, Y) be a link in this cycle. W.l.o.g assume that (X, Y) is in Q's connected component and $X \neq Q$. Disconnect (X, Y) and connect X to R. Again the connection of X to R is possible without exceeding δ.

Thus at most three topology changes cause $(n/2)^{1/k}/\delta$ processors to change their routing data-base. ∎

7 Distributed Bubbles Update

So far we have not been concerned with the issue of how to distribute the bubbles update algorithm. Still the above upper bound for the adaptability of the bubble routing scheme is useful for the case of manually adding and removing links and nodes. This is the case for telephone networks where new users may be connected and existing users may be disconnected. However, our results are made stronger by handling automatically topology changes while keeping the low adaptability. I.e., the number of node control units that participate in the distributed automatic update together with the number of node control units that must change their portion of the routing distributed data-base is low. For the sake of keeping a low adaptability we introduce a distributed bubbles update algorithm that can cope with transient failures and recoveries as well as permanent disconnection and connections.

The main idea for the distributed bubble update is to have the processors neighboring a topology change to be the *monitors* of the change. The task of a monitor is (1) to collect information on the tree structure it belongs to and on the existing routing data-base, (2) to try to merge with other neighboring trees and (3) to modify the existing routing data-base to fit the current topology. The modification should be performed in a way that involves the least number of node control units.

Towards this end it is assumed that each switching subsystem has a *sack variable* which may be modified by the node control unit. This sack variable contains the processor's portion of the distributed routing data-base. The sack variable can be collected by a special arriving message. We also assume that an arriving message may be concatenated with the label of the link through which it arrived. Note that no direct modification of the set of labels or the sack variable by an arriving message is allowed.

The links of the spanning tree $\mathcal{T}(\mathcal{B}_1)$ are distributively marked on the edges of the system. The two end points of a tree link are marked by a label t. In the sequel, we use the term *marked tree* to be the graph obtained by the set of marked links and the nodes they connect. Upon failure of a tree link the marked tree of a connected component may essentially be a marked forest that has to be fused to a new marked tree. Each tree in the forest is an autonomous entity with a monitor. The monitors of trees in the forest negotiate in order to promote a non tree link into a tree link. Upon such a promotion the number of trees (and monitors) in the forest is reduced by one. More details follow.

A monitor is associated with the time of the topology change that created it. We assume the existence of a synchronized clock at every processor. Each monitor uses the marked tree it belongs to for *tree broadcast with feedback* (in short TBF). The TBF collects the information in the sacks of every processor that belongs to the tree and delivers the information to the monitor. The monitor initiates the TBF by sending a message with its identifier, the timestamp of its creation and a TB (tree-broadcast) label on every marked tree link. Every switch that receives a message m of type TB concatenates the unique label of the link through which m arrived to the end of the message and forwards a copy of the message to each link that is labeled t except the link through which m arrived. If no other tree link exists (the switch is a leaf in the marked tree) then the switch modifies the message as follows: TB is replaced by TF (tree-feedback) the labels at the end become the address of the message (that transfer it back to the TBF initiator), and the content of the sack is concatenated to the end of the message. When a switch receives a TF message it removes the first label of the message arrived and concatenates the value of its sack to the tail of the message.

A processor P starts to act as a monitor when: Either one of the tree links attached to it fails or an attached link recovers. A monitor marks the label that leads from its switching subsystem to itself by the label m. This enables the switching subsystem to deliver a copy of every BF message to its node control unit when the node control unit is a monitor. Then the monitor collects information on its tree and the distributed routing data-base of this tree. This is done by the tree broadcast with feedback mechanism. Whenever a tree broadcast with feedback of another monitor arrives to (the node control unit of) P and the timestamp of the arriving TBF is greater (breaking ties by the monitors identifiers) then P stops being a monitor.

Upon receiving the information from the tree broadcast with feedback, P checks whether there are neighboring processors that do not belong to its tree. If such neighboring processor exists, then P sends a *promote* message to the processor Q attached to the link with the highest lexicographic order that leads to a neighboring tree. (For the sake of this lexicographically ordering, the name of a link is defined as the pair of identifiers of the two processors that are attached to it, where the first identifier in the pair is always greater than the second.) Q becomes the monitor and queries the other side on the possibilities to merge. Upon receiving an accept message the link changes its status to a tree link and the monitor with the smaller identifier sends the information collected in its TBF to its neighbor. At the same time it stops being a monitor. When there is no further possibility to merge the corrections of the routing data base is sent by the monitor to the appropriate node control units, and the processor that acted as a monitor resumes operation as a regular processor.

Lemma 7.1 *In every instance of time and for every connected component, if the marked tree does not span the entire connected component then for each tree in the forest there exists at least one monitor.*

Proof: This is true in the first instance of the system. Further topology changes keep this invariant. A monitor stops existing only when there exists another monitor in the same tree or when the marked tree spans the entire connected component. ■

Lemma 7.2 *If no monitor exists in a connected component then the distributed routing data-base of this connected component is correct.*

Proof: For every connected component of more than a single processor there has been an instance in which one processor of this connected component has been a monitor. The instance that follows a connection of any two neighboring processors by a link. The last processor that stopped being a monitor in this connected component must have finished the corrections of the distributed routing data-base. ■

Lemma 7.3 *Some time following the last topology change no monitor exists.*

Proof: The tree broadcast with feedback terminates. Then the monitor may wait for a promotion of a link into a tree link. Assume towards contradiction that the monitor waits for some link promotion forever. This in turn implies that the monitor of the other tree portion is waiting on another edge that has greater identifier. Since links that are to be promoted into tree links are chosen according to lexicographical order this chain of waiting monitors is finite and must end with two monitors waiting on the same edge. ■

Theorem 7.4 *The distributed adaptability is of the adaptability order.*

Proof: Only monitors and the node control units that have to change their distributed routing data-base participate in the distributed bubbles update. The number of monitors is no more than 2δ for a topology change: At most δ are directly influenced and another δ are because of monitors migration.

Note that during the correction process of the distributed routing data-base by one monitor a topology change might take place stopping the update before it is completed. Nevertheless, the partial correction process took place in a limited number of bubbles (as explained for the centralized case) leaving most of the bubbles unaffected. Thus, the number of node control units that have to change their distributed routing data-base following c topology changes is $O(ck3^{k-1}\delta^2 n^{1/k})$.

The theorem is proved since the additional $O(\delta)$ monitors do not change the $O(k3^{k-1}\delta^2 n^{1/k})$ adaptability. ■

8 Concluding Remarks

In this paper we defined new measures for the efficiency of routing schemes for high-speed dynamic networks. We presented the concept of the bubble-partition of a graph to support routing in high-speed dynamic networks. We believe that the resulting bubble partitioning technique may be of independent inerest in other contexts as well. Our routing scheme is efficient in terms of the number of routing steps (super-hop count), memory, and adaptability. The routing scheme also supports load balancing of traffic by allowing each processor to forward a message along a randomly chosen path within its bubble (the path does not necessarily belong to the bubble tree). The scheme has been proven optimal in its adaptability for a network of bounded degree and for a small constant k by a matching lower bound.

A number of variants of our scheme are possible in different contexts. We briefly discuss some of them next.

8.1 The programmable model

The basic model as described assumes that the labels associated with each link are fixed and permanent. A natural variant of this model is one in which it is assumed that the switching subsystem permits the node control unit to configure the link labels dynamically, or in other words, that the link labels are programmable. This assumption is commonly made in the high-speed model, including in PARIS (cf. [5]), and in fact, it is often assumed that each link may have a *set* of usable labels. We shall refer to this variant of the model as the *programmable model*.

A possible implementation enabling this assumption is as follows. Each port in the processor has a permanent (hard-wired) *physical id*. In addition, the link has a logical label, which can be changed from time to time. The switching subsystem maintains a *conversion mechanism*, enabling it to translate any given logical label into the appropriate physical port id. Hence upon the arrival of a message, the switching subsystem removes the first label l, consults the mechanism for the appropriate physical port id, and forwards the message onto that port for transmission.

For the programmable model, a simple variant of the Single-Leader routing scheme can be implemented with $O(n \log n)$ memory requirements (as opposed to $O(n^2 \log n)$ of the original Single-Leader scheme). The necessary modifications are as follows. Given the spanning tree, each processor labels its link upwards in the tree (toward the root L) by 0. (The rest of the links may be labeled arbitrarily with a non zero value.) A message with zero in its header that arrives to a non-root is forwarded through the link labeled by zero without shortening the message. For the first part of each message transmission, namely, the path from v to L, the processor v attaches the label zero to the message header. The root maintains the entire tree topology and forwards arriving messages to their destinations using non-zero labels.

8.2 Constrained bubble partitions

The second variant is related to special restrictions on the bubble partition. For some cases the bubbles partition might be restricted due to other constraints e.g., geographical constraints. For instance, one would not like to have two bubbles (say at level two) to cover the network in the USA, such that one of the bubbles includes Japan and the other includes England, instead a single bubble for the USA is preferred. Our bubble partition can take into account such considerations during the partition by ignoring some of the communication links during the bubble partition in some levels.

References

[1] Y. Afek, E. Gafni, and M. Ricklin. "Upper and lower bounds for routing schemes in dynamic networks," *Proc. 30th Symp. on Foundations of Computer Science,* pp. 370-375, 1989.

[2] B. Awerbuch. "Complexity of network synchronization," *JACM*, 32(4):804-823, 1985.

[3] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. "Compact Distributed Data Structure for Adaptive Routing," *Proc. 21th Symp. on Theory of Computing,* pp. 479-489, 1989.

[4] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. "Improved routing strategies with succinct tables," *J. of Algorithms*, 11:307-341, 1990.

[5] B. Awerbuch, I. Cidon, I. Gopal, M. Kaplan and S. Kutten, "Distributed Control for PARIS," *Proc. 9th PODC*, Quebec, Aug. 1990, 145-159.

[6] B. Awerbuch and D. Peleg. "Routing with polynomial communication-space tradeoff," *SIAM J. on Discrete Math,* 5(2) pp. 151-162, 1992.

[7] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks. "Adapting to asynchronous dynamic networks," *Proc. 24th Symp. on Theory of Computing,* pp. 557-570, 1992.

[8] I. Cidon and I. Gopal. "PARIS: An approach to private integrated networks," *Journal of Analog and Digital Cabled Systems* 1(2), pp. 77-86, 1988.

[9] I. Cidon, I. Gopal and S. Kutten, "New Models and Algorithms for Future Networks," *IEEE Transactions on Information Theory* 41(3), pp. 769-780, 1995.

[10] I. Cidon, O. Gerstel and S. Zaks. "A Scalable Approach to Routing in ATM Networks," *Proc. 8th Int. Workshop on Distributed Algorithms*, 1994.

[11] S. Dolev and T. Herman, "SuperStabilizing Protocols for Dynamic Distributed Systems" University of Iowa, Department of Computer Science Technical Report 95-02, 1995.

[12] O. Gerstel and S. Zaks. "The Virtual Path Layout Problem in Fast Networks," *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 1994.

[13] L. Kleinrock and F. Kamoun, "Hierarchical routing for large networks," *Computer Networks* 1, (1977), 155–174.

[14] L. Kleinrock and F. Kamoun, "Optimal clustering structures for hierarchical topological design of large computer networks," *Networks* 10, (1980), 221–248.

[15] D. Peleg and E. Upfal. "A tradeoff between size and efficiency for routing tables," *JACM,* 36:510-530, 1989.

[16] M. Santoro and R. Khatib. "Labeling and implicit routing in networks," *The Computer Journal,* 28:5-8, 1985.

[17] L. G. Valiant. "Universality consideration in VLSI circuits," *IEEE Transactions on Computers,* 30, 135-140, 1981.

[18] J. van Leeuwen, and R.B. Tan. "Routing with compact routing tables," *The Book of L*, G. Rozenberg and A. Salomaa, eds. Springer-Verlag, New York, 1986, pp. 259-273.

[19] J. van Leeuwen, and R.B. Tan. "Interval Routing," *The Computer J.*, 30 (1987), 298-307.

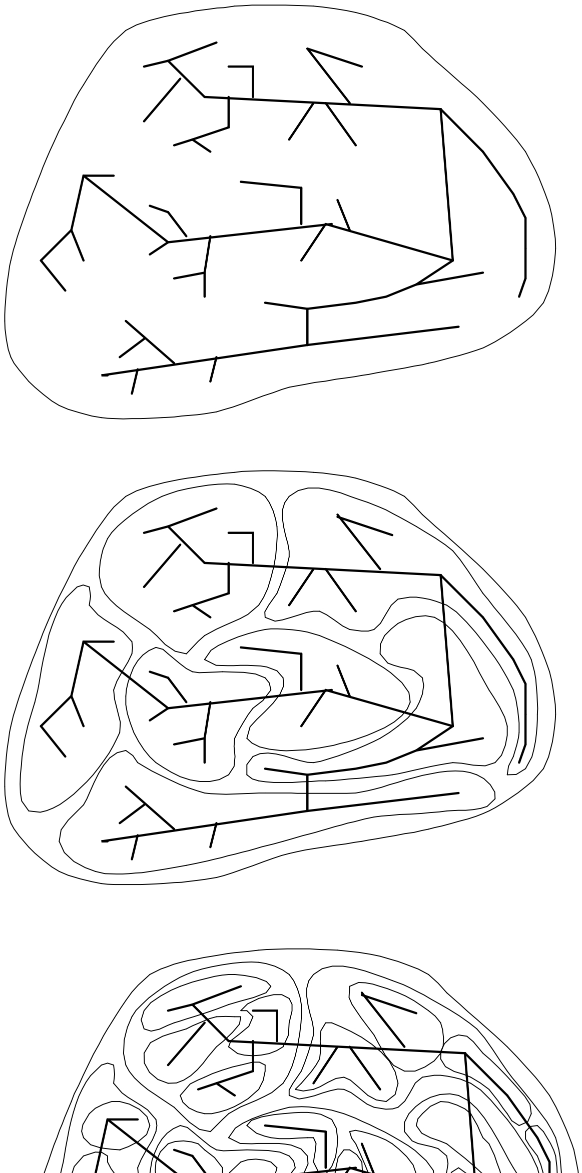

Figure 13: Bubble Partition $k = 3$

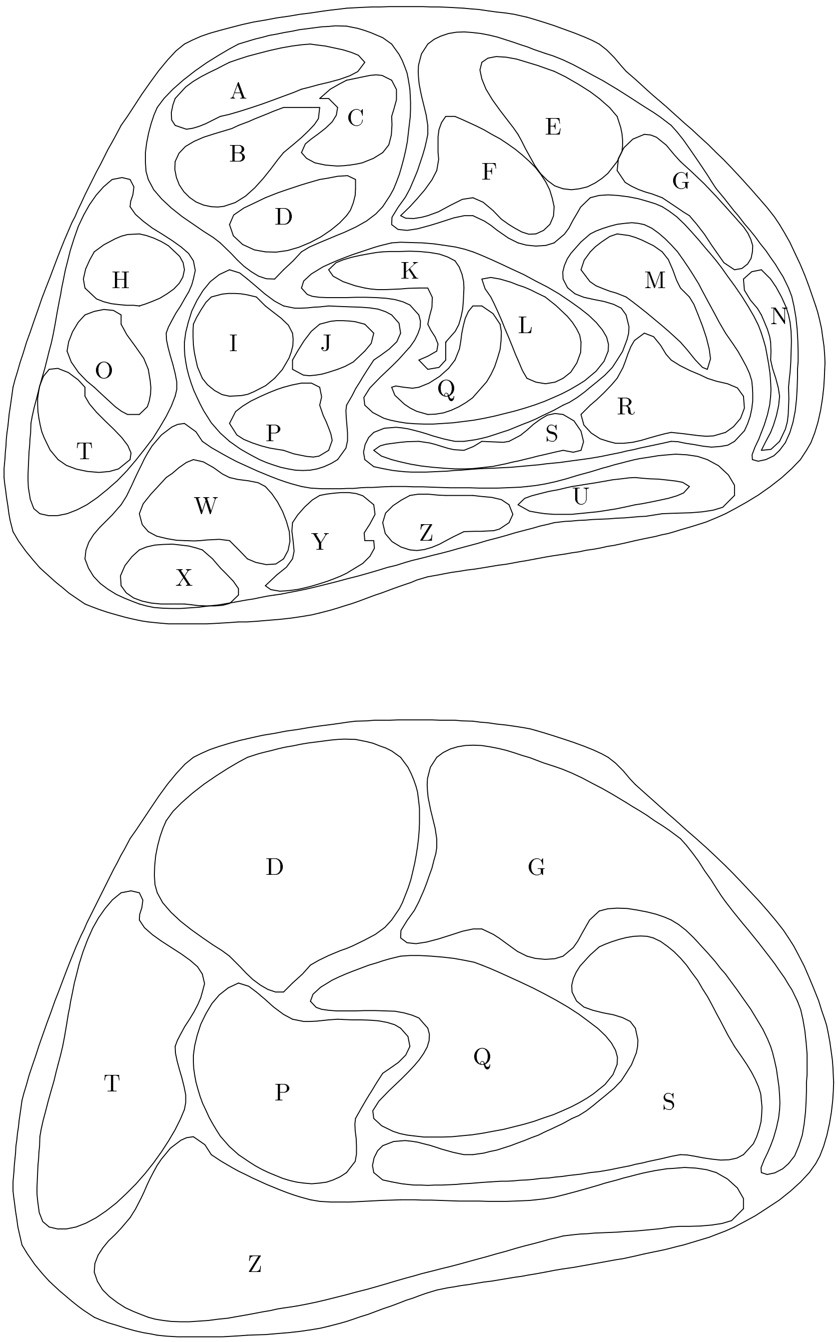

Figure 14: The members at level 2 and level 1

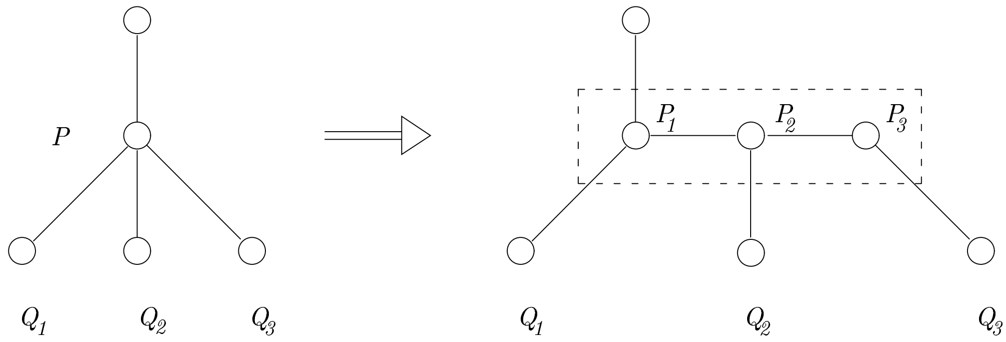

Figure 15: (a) A node P of degree 3. (b) P is split into 3 logical copies P_1, P_2, P_3.

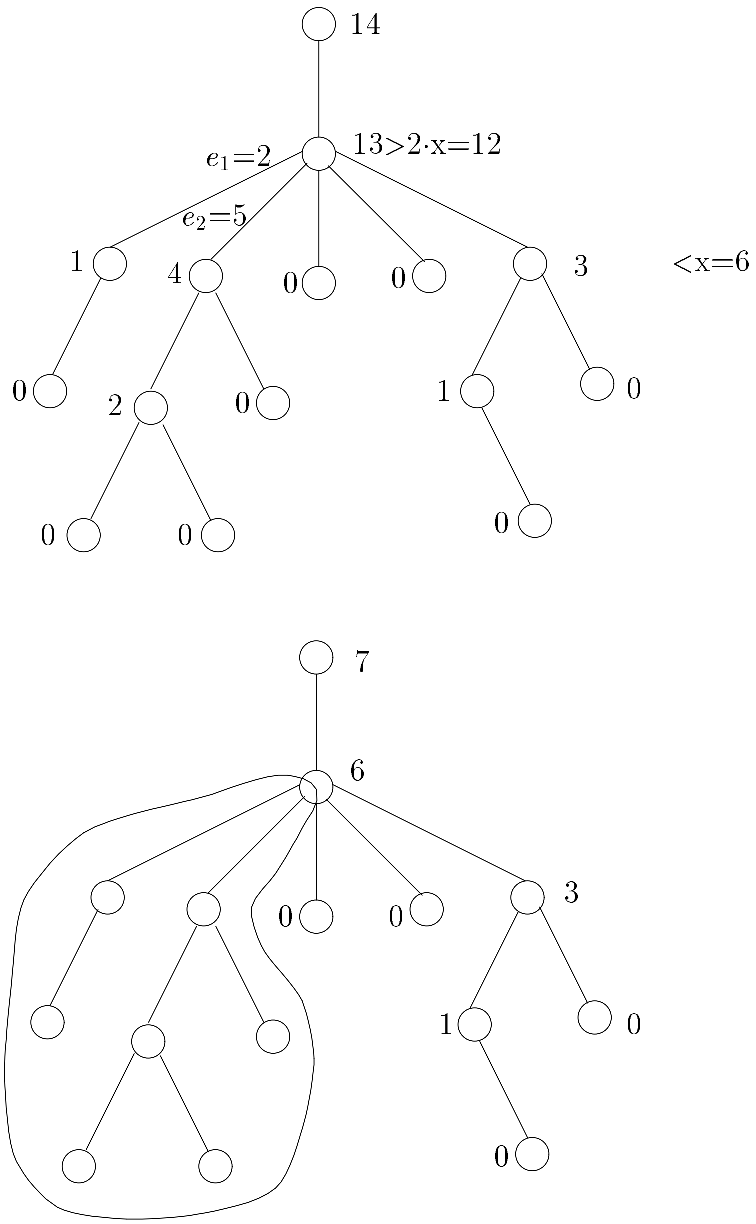

Figure 16: A Step of the Edge-Bubbles Partition