Dynamic Routing in Metrics of Low Doubling Dimension

Goran Konjevod Andréa W. Richa Donglin Xia

November 19, 2007

Abstract

We consider dynamic compact routing in metrics of low doubling dimension. Given a set of nodes V in a metric space with nodes joining, leaving and moving, we show how to maintain a set of links E that allows compact routing on the graph $G(V, E)$ in both labeled and name-independent models.

Given a constant $\epsilon \in (0, 1)$ and a dynamic node set V with normalized diameter Δ in a metric of doubling dimension α, we achieve

- a dynamic graph $G(V, E)$ with maximum degree $(1/\epsilon)^{O(\alpha)} \log \Delta$, and an optimal $(1+\epsilon)$-stretch compact labeled routing scheme on G with $O(\log^2 \Delta)$-bit label size and $O(\log^2 \Delta)$-bit storage at each node; and
- a dynamic graph $G(V, E)$ with maximum degree $2^{O(\alpha)} \log^2 \Delta$, and an optimal $(9 + \epsilon)$-stretch compact name-independent routing scheme on G with $(1/\epsilon)^{O(\alpha)} \log^3 \Delta$-bit storage at each node.

(Note that the lower bound of 9 on the stretch of compact name-independent routing schemes in our PODC'06 paper can be extended to routing on a metric.) Moreover, the amortized number of messages for a node joining, leaving and moving is polylogarithmic in the diameter Δ. Finally, the cost of a node movement operation is locality-sensitive. A k-level node-move protocol at a node u is performed only if u has moved to a point within distance at least 2^k from the point where u was where the last node-move protocol at level no less than k was executed at u. The amortized number of messages in a k-level node-move protocol is proportional to k^2 and k for the name-independent and labeled schemes, respectively, and each message traverses a path of distance $O(2^k/\epsilon)$.

Our dynamic routing schemes rely on a new dynamic search tree structure that is less sensitive to changes in the network topology than search trees resulting from standard network decomposition techniques (such as search trees that directly mimic a hierarchical r-net decomposition of the network), and thus supports a locality-sensitive load-balancing procedure.

1 Introduction

A metric space (M, d) consists of a set of points M and a distance function $d : M \times M \to R^+$. Given a set of nodes V, where a node is a processor with memory and corresponds to a distinct point in M, the routing algorithm designer is allowed to assign a set of links $E \subseteq V \times V$ so that the two end nodes of any link $(u, v) \in E$ can communicate with each other with delay $d(u, v)$[1]. A routing scheme on the graph $G(V, E)$ is a distributed algorithm running at each node that allows any source node to deliver packets to any destination node along the links in E. There are two models for routing schemes: *labeled* routing, and *name-independent* routing. The former allows the designer of the routing scheme to label the nodes so that they contain additional routing (e.g. topological) information. In the latter case, the routing scheme must use the (arbitrary) original naming.

For routing on a metric, we consider the trade-off among stretch, space and the number of links at each node. The *stretch* of a routing scheme is the maximum ratio of the length of a routing path between u and v to the distance $d(u, v)$, over all pairs of nodes u, v. The *space* requirement of a scheme refers to the size of routing tables maintained at each node and the size of packet headers used by the scheme. A routing scheme is *compact* if the routing table at each node and every packet header have size at most a polylogarithmic function of the number of nodes. In general, one would like to limit the number of links at each node to be polylogarithmic in the number of nodes, while achieving a compact routing scheme with optimal stretch.

Recently, there have been several new developments on compact routing schemes in graphs or metrics of low doubling dimension [11, 3, 9, 1, 6, 7, 8, 10], where the *doubling dimension* of a metric space is the minimum α such that any ball of radius r can be covered by at most 2^α balls of radius $r/2$. However, all the proposed schemes assume that the node set V is fixed. Moreover, with exception of [10], most existing work assumes that there is a centralized pre-configuration procedure for configuring the routing table at each node.

1.1 Our Contributions

In this paper, given a dynamic set of nodes V in a metric (M, d) of constant doubling dimension α, we present the *first fully dynamic optimal-stretch distributed compact routing schemes in both the labeled and name-independent models with polylogarithmic storage, label and packet header sizes, and polylogarithmic amortized number of messages for each node-join, leave or move operation.* Furthermore, the *total cost of updates caused by the movement of a node v is proportional to the total distance v moves.* These results follow directly from the three theorems below.

We use V_t to denote the particular configuration of the node set V at time t (whenever clear from context, we omit the parameter t). Let Δ_t be the diameter of V_t at time t, i.e. $\Delta_t = \max_{u,v \in V_t} d(u, v)$. Let Δ be the maximum diameter of V over time, i.e. $\Delta = \max_t \Delta_t$. A node is a processor with compact memory and a distinct name. As we will show in Section 5, we can use a distributed hash function which will enable us to represent each of the node names using $O(\log \Delta_t)$ bits at any time t — the amortized cost of updating this hash function whenever necessary has been taken into consideration in Theorem 1.3 below.

(Note that at each time, we have $\log|V_t| = O(\log \Delta_t)$ because of the constant doubling dimension.)

We provide node join/leave/move protocols for nodes joining/leaving/moving within the metric. For a node to join the network, it must have some arbitrary bootstrap node in the network it can connect with. We assume that a node leaves the network in a graceful way, that is, the node always performs the node-leave protocol before it leaves. We will maintain a hierarchical data structure with $O(\log \Delta_t)$ levels (we will specify the data structure used in Section 2). When a node moves, a *locality-sensitive* node-move protocol at an appropriate level of the hierarchy is performed. We guarantee that (i) for each movement of node u, a node-move protocol at u is executed at a single level; and (ii) a level k node-move protocol at node u will only be executed if u *has moved to a point within distance at least 2^k from the point where the last node-move protocol at level no less than k was executed at node u.*

Our main technical results on dynamic compact routing schemes are described in the following two theorems, plus Theorem 1.3.

Theorem 1.1 *Given a constant $\epsilon \in (0, 1)$ and a dynamic node set V in a metric with doubling dimension α, we maintain a dynamic graph $G(V, E)$ with $(1/\epsilon)^{O(\alpha)} \log \Delta$ degree, and achieve an optimal $(1 + \epsilon)$-stretch compact* labeled *routing scheme on G with $O(\log^2 \Delta)$-bit label size and $O(\log^2 \Delta)$-bit storage at each node. Moreover, for each node-move protocol at level k, it takes $(1/\epsilon)^{O(\alpha)} \log \Delta \cdot k$ amortized number of messages, and each message traverses a path of distance $O(2^k/\epsilon)$. In particular, it takes $(1/\epsilon)^{O(\alpha)} \log^2 \Delta$ messages for a node to join or to leave the network.*

[1]In this paper, we may abuse notation slightly by indistinctly referring to a node and the point it is mapped to, if clear from the context.

Theorem 1.2 *Given a constant $\epsilon \in (0,1)$ and a dynamic node set V in a metric with doubling dimension α, we maintain a dynamic graph $G(V,E)$ with $2^{O(\alpha)} \log^2 \Delta$ degree, and achieve an optimal $(9+\epsilon)$-stretch compact* name-independent *routing scheme on G with $(1/\epsilon)^{O(\alpha)} \log^3 \Delta$-bit storage at each node. Moreover, for each node-move protocol at level k, it takes $(1/\epsilon)^{O(\alpha)} \log^4 \Delta \cdot k^2$ amortized number of messages, and each message traverses a path of distance $O(2^k/\epsilon)$. In particular, it takes $(1/\epsilon)^{O(\alpha)} \log^6 \Delta$ amortized messages for a node to join or to leave the network.*

The stretch 9 of the name independent routing scheme in Theorem 1.2 is asymptotically optimal, since the lower bound result of compact name-independent routing on *graphs* from [6] naturally extends to compact name-independent routing in *metrics*.

One unique feature that distinguishes the name-independent routing scheme presented in this paper is that it does *not* rely on an underlying labeled scheme for the actual routing in the network (as basically all of the previous name-independent compact routing schemes in the literature [1, 6, 7, 8, 12] do): Since our scheme does not have to update a node label (and inform all other relevant nodes in the data structure of the label change) every time a nodes moves, our scheme is able to efficiently adapt to node movements, avoiding updates at higher levels of the hierarchy whenever a node has not moved a long distance.

As hinted above, we provide a scale-control procedure which scales the above schemes according to the dynamic variations in network size and diameter, as stated in Theorem 1.3. Let $\{\Delta_{i_j} : j = 0, 1, \cdots\}$ be a subsequence of $\{\Delta_t : t = 0, 1, \cdots\}$ such that (i) $\Delta_{i_0} = \Delta_0$; (ii) $|\log \Delta_{i_{j+1}} - \log \Delta_{i_j}| \geq \log \frac{1}{\epsilon}$ for all j; and (iii) $|\log \Delta_{i_j} - \log \Delta_k| \leq \log \frac{1}{\epsilon} + O(1)$ for all j and all $i_j \leq k \leq i_{j+1}$.

Theorem 1.3 *The scale-control procedure for the network size is executed whenever the number of nodes is squared or square-rooted. A constant amortized number of messages per node in the network is enough to perform the operation.*

The scale-control procedure for the diameter is executed at each time i_j, for $j = 0, 1, \cdots$. If $\Delta_{i_j} < \Delta_{i_{j-1}}$, a constant amortized number of messages per node in the network is enough to perform the scale-control procedure at time i_j. If $\Delta_{i_j} > \Delta_{i_{j-1}}$, the scale-control procedure at time i_j takes $(1/\epsilon)^{O(\alpha)} \log^6 \Delta_{i_j}$ amortized number of messages per node in the current network.

Hence, *the results in Theorem 1.1 and 1.2 still hold if we take to Δ to be the* current *diameter of the network, rather than the maximum diameter over time.*

We believe that some of the novel data structures presented in this paper are a contribution in their own right, since they open new directions for the design of dynamic network algorithms. In particular, the *skeletal trees* presented in Section 3.3 appear much more suitable for the storage of dynamic information (such as routing location information) than prior search trees which resulted from standard network decomposition (such as search trees that directly mimic a hierarchical r-net decomposition of the network [9, 1, 6, 7, 8]). The main idea behind the skeletal tree is that it recursively keeps *large* (in terms of number of nodes) branches of a cluster tree and omits all the small branches. This makes a skeletal tree less sensitive to changes in the network topology and more suitable for dynamic applications. In addition, we also present a locality-sensitive load-balancing procedure which is used to (re-)balance our dynamic storage structure. The current distribution of load on the storage structure determines (in a distributed fashion) when, where, and at which level to trigger load-balancing, thus guaranteeing efficient costs for the load-balancing operations.

1.2 Related Work

A more general version of our problem is that of designing compact routing schemes on a given graph $G(V,E)$. Hence all compact routing schemes for graphs whose induced shortest-path metric is of low doubling dimension also apply to our problem. However, all of the existing compact routing schemes [11, 3, 9, 1, 6, 7, 8] only consider the static version of the problem. Nevertheless, our scheme not only efficiently adapts to dynamic changes in the node set, but does so while still achieving optimal stretch factors and maintaining the polylogarithmic guarantees on storage, label and packet header size (note however that the best-known static schemes are scale-free, i.e. independent of the network diameter, while ours is not.). Another feature of most static schemes is that they assume that there is a centralized preconfiguration procedure for configuring the routing tables. One exception is the recent work by Slivkins [10] where he presents a decentralized procedure to build the routing tables.

Given a dynamic set of nodes V in a metric of doubling dimension α, Gottlieb and Roditty [4] provide a dynamic $(1+\epsilon)$-spanner[1] with degree $O(1/\epsilon^\alpha)$ polylogarithmic update time for both insertions and deletions. Based on their spanner, they propose a labeled routing scheme. However, the proposed labeled routing scheme itself is static and the authors do not provide any efficient mechanisms for maintaining the routing information and updating node labels as the spanner topology changes.

[1]A t-spanner of a set of nodes V in a metric (M,d) is a graph $G(V,E)$ with weight $d(u,v)$ for each edge $(u,v) \in E$ such that

2 Dynamic Graph

Given a dynamic set of nodes V in a metric space (M, d) of doubling dimension α, for each routing scheme, we define a virtual graph $G' = (V', E')$, and a host mapping $\phi : V' \to V$ that associates each virtual vertex in V' to a host node in V. This virtual graph will give us the necessary data structures to achieve the results outlined in Theorems 1.1, 1.2, and 1.3. Thus it is natural to define the link set $E = \{(\phi(x), \phi(y)) \ : \ \forall (x, y) \in E'\}$.

The elements of the metric space M are called points. To avoid confusion between $G(V, E)$ and the virtual graph $G'(V', E')$, the elements of V and E are called *nodes* and *links* respectively, while the elements of V' and E' are called *vertices* and *edges* respectively. Each vertex $x \in V'$ (resp., each node $u \in V$) corresponds to a point $pnt(x)$ (resp., $pnt(u)$) in M.

For any point $x \in M$ and $r > 0$, let the ball $B_x(r)$ denote the set of points in M within distance r to x, i.e. $B_x(r) = \{y \in M \ : \ d(x, y) \leq r\}$. In the following, we give the definition of an r-net and present some of the structural properties of an r-net, on which our hierarchical data structures are based.

Definition 2.1 (r-net) *For any $r > 1$, a set $Y \subseteq M$ is an* r-net *if the distance of any pair of points $y, y' \in Y$ is at least r, i.e. $d(y, y') \geq r$.*

We say the r-net Y *covers* a set $X \subseteq M$ if for any point $x \in X$ there exists an r-net point $y \in Y$ within distance of r, i.e. $|Y \cap B_x(r)| \geq 1$.

Lemma 2.2 ([5]) *Let Y be an* r-net *in a metric space with doubling dimension α. For any point x in the metric space and $r' \geq r$, we have $|Y \cap B_x(r')| \leq \left(\frac{4r'}{r}\right)^{O(\alpha)}$.*

2.1 Virtual Graph and Host Mapping

The virtual graph G' consists of two hierarchies of 2^i-nets: the *parent* hierarchy $X = \cup_{i=0}^{h} X_i$ and the *cluster header* hierarchy $Y = \cup_{i=0}^{h} Y_i$, where $h = \log \Delta$. Let the *parent set* X_i be a 2^i-net, for each $i \in [h]$[2], and $X_0 = V$. Let the *cluster header set* Y_i be a 2^i-net covering X_i, for each $i \in [h]$. Actually we abuse the notation slightly; when we say that, for example, X_i is a 2^i-net, we mean that its point set $pnt(X_i) \subseteq M$ is a 2^i-net. Note that we treat two vertices from different levels of X_i or Y_i, or from X and Y respectively, as different vertices, even if their corresponding points in M are identical.

We define three kinds of edge relationships as follows. For each node $u \in V$ and each $i \in [h]$, let $p_i(u) \in X_i$ denote the *parent* of u at level i, which is selected by our dynamic protocols and initially has $d(p_i(u), p_{i-1}(u)) \leq 2^i$. For each cluster header $y \in Y_i$ and $i \in [h]$, let $N(y) = X_i \cap B_y(2^i/\epsilon)$ be the *neighborhood set* of y. Since Y_i is a 2^i-net covering X_i, for $i \in [h]$, let $f : X_i \to Y_i$ be the *header mapping* that maps each $x \in X_i$ to a cluster header $y = f(x) \in Y_i$ that covers x, i.e. $d(y, x) \leq 2^i$. Thus we defines edge sets: $E'_1 = \{(p_i(u), p_{i-1}(u)) \ : \ \forall u \in V, \forall i \in [h]\}$, $E'_2 = \{(y, x) \ : \ \forall y \in Y, \forall x \in N(y)\}$, and $E'_3 = \{(x, f(x)) \ : \ \forall x \in X\}$.

For each vertex $x \in X_i$ and $i \in [h]$, the host $\phi(x)$ is one of the nodes whose parent at level i is x, i.e. $p_i(\phi(x)) = x$. For each vertex $y \in Y_i$ and $i \in [h]$, let $\phi(y) = \phi(x)$, for $x \in X_i$ that minimizes $d(x, y)$.

For labeled routing schemes, the virtual graph $G' = (V', E')$ consists of $V' = X \cup Y$, and $E' = E'_1 \cup E'_2 \cup E'_3$. For name-independent routing schemes, moreover, we maintain a *cluster tree* $CT(y)$ for each cluster header $y \in Y$. For each $x \in X_i$ and $i \in [h]$, the *descendant tree*, denoted $T(x)$, consists of paths $\langle p_i(u) = x, p_{i-1}(u), \cdots, p_0(u)\rangle$ for all nodes $u \in V$ s.t. $x = p_i(u)$, and the maximal common prefix of any two paths is merged. That is, for any two such paths w.r.t. nodes u, v, let j be the minimal index such that $d(p_k(u), p_k(v)) = 0$ for all $j \leq k \leq i$; then vertices $p_k(u)$ and $p_k(v)$ are identical in $T(x)$ for all $j \leq k \leq i$. For each cluster header $y \in Y$, the *cluster tree*, denoted $CT(y)$, consists of root y, edges (y, x) and descendant trees $T(x)$ for all $x \in N(y)$.

2.2 Restricted vs. General Models

We consider two models. In the *restricted* model, nodes are allowed to join or leave the network. In the *general* model, we also allow nodes to move. For ease of explanation, we first present a labeled and a name-independent scheme for the restricted model. The name-independent routing scheme relies on the labeled one. If we allow nodes to move in the network, then the routing labels of these nodes may need to be changed, making any dynamic name-independent scheme which relies on these routing labels rather inefficient. Hence, we show in Section 4.3 how to modify the name-independent scheme presented for the restricted model to also work efficiently on the general model, by "embedding" the routing labels of the nodes into the hierarchy of search tree data structure used.

In the restricted model, the vertices in descendant trees are identical to their corresponding vertices in X. Thus we still have $V' = X \cup Y$ and $E' = E'_1 \cup E'_2 \cup E'_3$ for the name-independent routing scheme in the

the shortest path distance of any pair of nodes in G is their distance in the metric by a factor at most t.

[2]For any integer $k \geq 0$, let $[k]$ denote the set $\{0, 1, \cdots, k\}$.

restricted model. However, in the general model, the vertices in each descendant tree are a distinct copy of their corresponding vertices in X. The host map on these vertices is similar to the host map on X. However, the number of links at each node for the general model is a factor of $\log \Delta$ larger than that for the restricted model.

Therefore we have $V' = (X \cup Y) \bigcup \cup_{y \in Y} V(CT(y))$, and $E' = (E'_1 \cup E'_2 \cup E'_3) \bigcup \cup_{y \in Y} E(CT(Y))$ for the name-independent routing scheme in the general model.

3 Routing in the Restricted Model

In the restricted model, we consider routing schemes in metrics with nodes joining and leaving. Subsection 3.1 provides node join/leave protocols to maintain the hierarchies X and Y dynamically. In Subsection 3.2, a labeled routing scheme is provided. For the name-independent routing scheme, Subsection 3.3 discusses how to maintain a search tree for each cluster tree $CT(y)$, $\forall y \in Y$, while Subsection 3.4 provides the routing algorithm.

3.1 Node Join/Leave Protocols

Given the dynamic node set V with nodes joining and leaving, the node join/leave protocols maintain the virtual graph G', the host mapping ϕ, and therefore the link set E. In the restricted model, we maintain a root vertex r such that $p_{h+1}(u) = r$ for any $u \in V$. We update $X = X \cup \{r\}$ and $E'_1 = E'_1 \cup \{(p_h(u), p_{h+1}(u)) \ : \ \forall u \in V\}$.

Invariant 3.1 *In the restricted model, we preserve the following invariants:*

- *The graph (X, E'_1) is a tree rooted at r, i.e. the descendant tree $T(r)$.*
- *$d(p_0(u), u) = 0$, and $d(r, u) \leq 2^h$, for all $u \in V$.*
- *$d(p_i(u), p_{i-1}(u)) \leq 2^i$ for $i \in [h]$, and $d(p_{h+1}(u), p_h(u)) \leq 2^{h+1} + 2^h$.*
- *For a node $u \in Y$ and $\forall i \in [h]$, if vertex $p_i(u)$ is hosted at u, i.e. $\phi(p_i(u)) = u$, then for all $k \leq i$ vertices $p_k(u)$ are hosted at u, i.e. $\phi(p_k(u)) = u$.*

3.1.1 Node Join Protocol

The node join protocol is provided in Algorithm 1. When a new node u joins V, let i be the minimum index s.t. $\exists z \in X_{i+1} \cap B_u(2^{i+1})$. Let $p_k(u)$ be the vertices on the path from t to z in $T(r)$, for $k = h+1, h, \cdots, i+1$. We call procedure `AddNewParent` to add a new parent node x with $pnt(x) = pnt(u)$ as $p_k(u)$, for $k = i$ down to 0. Lines 5 to 10 of `AddNewParent` find a cluster header y in $Y_k \cap B_x(2^k)$ for the newly added parent vertex x, or add a new cluster header y if $Y_k \cap B_x(2^k)$ is empty. Note that we append a new path $\langle p_i(u), p_{i-1}(u), \cdots, p_0(u) \rangle$ to the tree $T(r)$ at $z = p_{i+1}(u)$. It is easy to verify that Invariant 3.1 is preserved.

3.1.2 Node Leave Protocol

The node leave protocol is provided in Algorithm 2. When an existing node u leaves V, let i be the maximum index s.t. $\phi(p_i(u)) = u$ and let j be the minimum index s.t. $\exists v \neq u \in V$, $p_j(v) = p_j(u)$. In general, the leave protocol deletes all parent vertices $p_k(u)$ for $k = j-1$ down to 0 using procedure `DeleteParent`, while it uses procedure `UpdateHost` to assign v as the host for all parent vertices and cluster headers with levels between i and j that were previously hosted by u.

Note that when u is leaving, we remove the path $\langle p_{j-1}(u), \cdots, p_1(u), p_0(u) \rangle$ from $T(r)$, and update the host of $p_k(u)$, for each $j \leq k \leq i$, to such a v that $p_k(u) = p_k(v)$ and $\phi(p_{j-1}(v)) = v$. In addition, if $i = h+1$, i.e. previously $\phi(r) = u$, we also update $pnt(r) = pnt(v)$ as well as updating $\phi(r) = v$. Thus it is easy to verify Invariant 3.1.

3.2 Labeled Routing

3.2.1 Labeling

We color each vertex x in X, using a color function $c : X \to [10^{O(\alpha)}]$ such that no two siblings in the descendant tree $T(r)$ share a color. That is, for any vertex $x \in T(r)$, its children have different colors. Since the number of children of any vertex in $T(r)$ is at most $10^{O(\alpha)}$ by Lemma 2.2, we can always find a valid color for a newly added vertex.

Thus for each $u \in V$, we assign its label $\ell(u) = \langle c(p_h(u)), c(p_{h-1}(u)), \cdots, c(p_0(u)) \rangle$. In addition, for each vertex $x \in X_i$ of $T(r)$, define its label $\ell(x) = \langle c(x_h), c(x_{h-1}), \cdots, c(x_i) \rangle$, where $\langle r, x_h, x_{h-1}, \cdots, x_i = x \rangle$ is a path from the root r to x in $T(r)$.

When a new node u joins V, we append a path to the tree $T(r)$, as in Algorithm 1. When a node u leaves V, a path from $p_0(u)$ up to a first vertex with at least two children in $T(r)$ is removed, as in Algorithm 2. Thus these topological changes do not affect the labeling of nodes other than u itself.

Algorithm 1: A new node u joins V

```
1: begin
2:     Find minimum index $i$ s.t. $\exists z \in X_{i+1} \cap B_u(2^{i+1})$
3:     Set $p_k(u)$ be the vertices on the path from $root$ to $z$, for $k = h+1$ downto $i+1$
4:     for $k = i$ downto $0$ do
5:         AddNewParent($u, k$)
6: end
```

```
1: Procedure AddNewParent($u, k$)
2: begin
3:     Add a new vertex $x$ with $pnt(x) = pnt(u)$ into $X_k$, and set $\phi(x) = u$
4:     Set $p_k(u) = x$ and add $x$ to $N(y)$ for all $y \in Y_k \cap B_x(2^k/\epsilon)$
5:     if $\exists y \in Y_k \cap B_x(2^k)$ then
6:         Set $f(x) = y$
7:         if $d(x, y) \le d(p_k(v), y)$, where $v = \phi(y)$ then Update $\phi(y) = \phi(x)$
8:     else
9:         Add a new vertex $y$ with $pnt(y) = pnt(u)$ into $Y_k$, and set $\phi(y) = u$
10:        Set $N(y) = X_k \cap B_y(2^k/\epsilon)$, and $f(x) = y$
11: end
```

Algorithm 2: A node u leaves V

```
1: begin
2:     Let $i$ be the maximum index s.t. $\phi(p_i(u)) = u$
3:     Let $j$ be the minimum index s.t. $\exists v \neq u \in V$, $p_j(v) = p_j(u)$
4:     W.l.o.g., pick $v$ s.t. $\phi(p_{j-1}(v)) = v$
5:     for $k = i$ downto $j$ do
6:         UpdateHost($u, k, v$)
7:     for $k = j-1$ downto $0$ do
8:         DeleteParent($u, k$)
9:     if $i$=$h$+1 then Set $pnt(r) = pnt(v)$
10: end
```

```
1: Procedure UpdateHost($u, k, v$)
2: begin
3:     Set $\phi(p_k(u)) = v$
4:     Set $\phi(y) = v$, $\forall y \in Y_k$ with $\phi(y) = u$
5: end
```

```
1: Procedure DeleteParent($u, k$)
2: begin
3:     Remove $x = p_k(u)$ from $X_k$
4:     for all $y \in Y_k \cap B_x(2^k/\epsilon)$ do
5:         Remove $x$ from $N(y)$
6:         if $N(y) = \emptyset$ then Remove $y$ from $Y_k$
7:         else if $\phi(y) = u$ then
8:             Set $\phi(y) = \phi(z)$, where $z \in N(y)$ that minimizes $d(z, y)$
9: end
```

3.2.2 Routing Algorithm

Let each cluster header $y \in Y$ store all labels of vertices $x \in N(y)$. The labeled routing algorithm is given in Algorithm 3. Given the label of the destination v, it searches for a parent of v along the path $\langle p_0(u), p_1(u), \cdots, p_i(u), \cdots \rangle$ of the source node u by checking whether $\exists x \in N(f(p_i(u)))$ s.t. $\ell(x)$ is a prefix of $\ell(v)$. If yes, then x is the parent of v at level i. Thus it simply go down to v along the path $\langle p_i(v), p_{i-1}(v), p_0(v) \rangle$ according to the label of v.

Algorithm 3: Labeled routing from a node u to v, given the label $\ell(v)$

```
1: for $i = 0$ to $h$ do
2:     if $\exists x \in N(f(p_i(u)))$ s.t. $\ell(x)$ is a prefix of $\ell(v)$ then break ;      /* $x = p_i(v)$ */
3: for $k = i-1$ downto $0$ do
4:     Let $x' \in X_k$ be a child of $x$ s.t. $c(x') = c(p_k(v))$
5:     $x \leftarrow x'$
6: return $\phi(x)$ ;                                                                  /*$\phi(x) = v$*/
```

Theorem 3.2 *Given a constant $\epsilon \in (0, 1)$ and a dynamic node set V with nodes joining and leaving in a metric with doubling dimension α, we maintain a dynamic graph G with degree $(1/\epsilon)^{O(\alpha)} \log \Delta$, and achieve a $(1 + O(\epsilon))$-stretch compact labeled routing scheme on G with $O(\alpha \log \Delta)$-bit label size and $(1/\epsilon)^{O(\alpha)} \log^2 \Delta$-bit storage at each node. In addition, $(1/\epsilon)^{O(\alpha)} \log \Delta$ messages suffice for a node to join or to leave the network.*

Proof: Note that the degree of the virtual graph G' is $(1/\epsilon)^{O(\alpha)}$. At each level i, a node u hosts at most one parent vertex $p_i(u)$, and at most all cluster headers in $Y_i \cap B_{p_i(u)}(2^i/\epsilon)$ which has size at most $(1/\epsilon)^{O(\alpha)}$. Thus the degree of G is at most $(1/\epsilon)^{O(\alpha)} \log \Delta$.

Since there are 10^α colors, the label can be expressed in $O(\alpha \log \Delta)$ bits. Since $N(y) = (1/\epsilon)^{O(\alpha)}$ by Lemma 2.2, each cluster header $y \in Y$ stores $(1/\epsilon)^{O(\alpha)} \log \Delta$ bits of information. Since one node hosts at most $(1/\epsilon)^{O(\alpha)} \log \Delta$ cluster headers, it stores at most $(1/\epsilon)^{O(\alpha)} \log^2 \Delta$ bits of information.

Let i be the minimum index such that $p_i(v) \in N(f(p_i(u)))$. By minimality, we have $d(p_{i-1}(v), f(p_{i-1}(u))) > 2^{i-1}/\epsilon$, which implies $d(u, v) > 2^{i-1}(1/\epsilon + O(1))$. In the virtual graph G', the routing cost is $d(p_i(u), p_i(v)) + O(2^i) \leq d(u, v) + O(2^i)$, while the cost in the real graph is still at most $d(u, v) + O(2^i)$ by Invariant 3.1. This implies stretch $1 + O(\epsilon)$.

In addition, for a node u joining or leaving the network, we just add or delete the label of u at each cluster header $Y_i \cap B_{p_i(u)}(2^i/\epsilon)$, for each $i \in [h]$. Thus $(1/\epsilon)^{O(\alpha)} \log \Delta$ messages suffice to complete the operation. $\blacksquare$

3.3 Search Trees

For name-independent routing schemes, we maintain a *search tree* for each cluster tree. The search tree stores the routing data of nodes within its corresponding cluster tree keyed by the node names. In this Subsection, we introduce the concept of *skeletal trees*, which is used for search trees. Then we give a dynamic load balancing procedure on search trees. Finally we show how to update search trees to work with Algorithms 1 and 2 and what the cost is.

3.3.1 Skeletal Trees

It might be natural to use cluster trees themselves as search trees in static networks. However, in order to compensate for frequent changes in dynamic networks, we maintain a search tree on a subgraph of its corresponding cluster tree to make it relatively insensitive to network changes. Intuitively, the subgraph, called *skeletal tree*, contains large (in terms of the number of nodes) branches but omits small branches of the cluster tree.

Definition 3.3 (Skeletal Tree) *Given a cluster tree $CT(y)$, for $y \in Y_i$ and $i \in [h]$, let $CT_y(x)$ denote the subtree rooted at x of $CT(y)$, and let $s_y(x)$ denote the number of leaves in the subtree $CT_y(x)$, for all vertices $x \in CT(y)$. The* skeletal tree *of $CT(y)$, denoted $ST(y)$, is the subgraph of $CT(y)$ including (i) the root y; and (ii) any edge (x, z) of $CT(y)$, where z is a child of x in $CT(y)$, if $x \in ST(y)$ and $s_y(z)/s_y(x) \geq \frac{1}{b \cdot (i+1)}$, where $b = \left(\frac{4}{\epsilon}\right)^\alpha$. We denote the subtree of $ST(y)$ rooted at x by $ST_y(x)$, for any vertex $x \in ST(y)$.*

Note that the degree of $CT(y)$ is b, and its height is i. Thus we have the following lemma:

Lemma 3.4 *Given any $y \in Y_i$ and $i \in [h]$, the ratio of the number of leaves in the skeletal tree $ST(y)$ and the number of leaves in the cluster tree $CT(y)$ is no less than $(1 - \frac{1}{i+1})^i > e^{-1}$ (where $e \approx 2.71828$).*

For each cluster tree $CT(y)$, $\forall y \in Y$, we maintain a search tree on its skeletal tree $ST(y)$ to store the routing data of all leaves (i.e. the real nodes) in $CT(y)$ using node *name* as the key. For each vertex $x \in ST(y)$, let $Range_y(x)$ be the minimal interval that contains all keys stored in $ST_y(x)$. Then the interval of the root vertex, $Range_y(y)$, is the whole range of the key space, while for any vertex $x \in ST(y)$, the set $\{Range_y(z) : z \text{ is a child of } x\}$ is a partition of $Range_y(x)$. A key k together with its associated data is inserted into the leaf z of $ST(y)$ such that $k \in Range_y(z)$ along the path from the root y to the leaf z. Let each leaf z of $ST(y)$ keep its stored keys in a sorted list, denoted $List_y(z)$. Let $List_y(x)$ denote the sorted list of all keys stored in $ST_y(x)$. Note that we can enumerate keys in $List_y(x)$ by enumerating keys in each list $List_y(z)$, for all leaves z in $ST_y(x)$ in a preorder traversal of the subtree.

Given a key k and a search tree $ST(y)$, $y \in Y_i$ and $i \in [h]$, the search procedure on $ST(y)$ searches for k along the path from the root to a leaf such that any vertex x on the path has $k \in Range_y(x)$, and goes back to the root with the data or with the "not-found" message. This takes $2i$ messages and $2^{i+1}(1/\epsilon + O(1))$ delay.

3.3.2 Load Balancing

Whenever the structure of a search tree changes due to the network changes, we want to perform a balancing procedure to balance the load on nodes. In addition, if the load on a node is too heavy due to the unbalance of newly inserted keys, we also want to trigger load balancing across regions with heavy load. The following lemma gives a load balancing procedure and its performance:

Lemma 3.5 (Load Balancing) *Given a subtree $ST_y(x)$ of any search tree $ST(y)$, for $y \in Y$ and $x \in ST(y)$, there is a load balancing procedure on $ST_y(x)$ for keys in $List_y(x)$ that restores these keys in $ST_y(x)$ such that every leaf stores an equal number of keys; and it takes $O(k \cdot |List_y(x)|)$ messages, where k is the height of $ST_y(x)$.*

Proof: Recall that the list $List_y(x)$ consists of lists $List_y(z)$, for all leaves z of $ST_y(x)$, in a preorder traversal of $ST_y(x)$. Thus we consider the load balancing procedure that, given a list of keys $List_y(x)$, stores an equal number of keys in each leaf of $ST_y(x)$ in a preorder traversal. Note that it takes $O(k)$ messages to find the next key in the original list, and $O(k)$ messages to go to the next available storage leaf, since the successor operator at $ST_y(x)$ takes $O(k)$ messages. In addition, it takes $O(k)$ messages to copy a key leaf-to-leaf in $ST_y(x)$. Overall it takes $O(k \cdot |List_y(x)|)$ messages to complete the load balancing procedure at $ST_y(x)$ for keys in $List_y(x)$. ■

Now we consider when to trigger the load-balancing procedure. For example, there are too many new nodes arriving at the cluster tree $CT(y)$ with their names within $Range_y(x)$ for a fixed leaf x of $ST(y)$. However, by Lemma 3.4 a load-balancing procedure on the whole search tree is able to average the load on each leaf to be a constant number of $(key, data)$ pairs. On the other hand, since the cost of a balancing procedure is proportional to the number of keys stored in the tree where the procedure is performed, we trigger the procedure on a restricted area of the search tree, rather than always on the whole search tree.

For each cluster tree $CT(y)$, $\forall y \in Y$, we maintain a counter $t_y(x)$ for each vertex x in the search tree $ST(y)$: (i) initially $t_y(x)$ is set to zero; (ii) whenever a key inserted in the search tree $ST(y)$ is stored in the subtree $ST_y(x)$, $t_y(x)$ is increased by one; (iii) when $t_y(x)$ reaches $s(x)$, i.e. the number of leaves of $T(x)$, load-balancing is executed on the subtree $ST_y(x)$ for the keys in $List_y(x)$, and $t_y(z)$ is reset to zero for all vertices $z \in ST_y(x)$.

3.3.3 Dynamic Maintenance of Search Trees

In the restricted model, we reuse the vertices in $X \cup Y$ for cluster trees. Here we discuss the maintenance of search trees according to Algorithms 1 and 2.

Node Joins When a new node u joins V, there are three kinds of updates with regard to each search tree $ST(y)$, $\forall y \in Y_i$ and $\forall i \in [h]$, s.t. $p_i(u) \in N(y)$:

- Insert the $(key, data)$ pair of u into $ST(y)$, which takes $O(i)$ messages. By Lemma 2.2, there are at most $(1/\epsilon)^{O(\alpha)}$ nodes $y \in Y_i$ such that $p_i(u) \in N(y)$. Thus it takes $O((1/\epsilon)^{O(\alpha)} \log^2 \Delta)$ messages to insert the routing information of u into all such search trees $ST(y)$ over all $i \in [h]$.
- The structure of the skeletal tree $ST(y)$ might change due to the insertion of u. The number of leaves of each subtree $CT_y(z)$, $\forall z \in CT(y)$, that contains $p_0(u)$ is increased, which might result in the addition of the branch $ST_y(z)$ to the skeletal tree $ST(y)$ at z's parent x if $s_y(z)/s_y(x) \geq \frac{1}{b\cdot(i+1)}$. In that case, we perform load balancing on the updated subtree $ST_y(x)$ for the list $List_y(x)$, which takes $k \cdot |List_y(x)|$ messages, where k is the height of $ST_y(x)$, i.e. $x \in X_k$.
- If the cluster header y is just added as in Line (9) of `AddNewParent` procedure, the search tree $ST(y)$ is created for the new cluster tree $CT(y)$. Each node in $CT(y)$ inserts its routing information into $ST(y)$.

In addition, after the insertion of a key, load balancing might be triggered.

Node Leaves When a node u leaves V, there are three kinds of updates with regard to each search tree $ST(y)$, $\forall y \in Y_i$ and $\forall i \in [h]$, s.t. $p_i(u) \in N(y)$:

- Delete the routing information of u from $ST(y)$, which takes $O(i)$ messages.
- The structure of the skeletal tree $ST(y)$ might change due to the deletion of u. The number of leaves of each subtree $CT_y(z)$, $\forall z \in CT(y)$, that previously contained $p_0(u)$ is increased, which might result in the removal of the branch $ST_y(z)$ from the skeletal tree $ST(y)$ at z's parent x. However, we remove the branch $ST_y(z)$, only if $s_y(z)/s_y(x) < \frac{1}{2b\cdot(i+1)}$, instead of $s_y(z)/s_y(x) < \frac{1}{b\cdot(i+1)}$. This helps avoid repeated addition and removal of the same branch with only several nodes joining and leaving. In addition, a load-balancing procedure on the updated subtree $ST_y(x)$ is performed for the old list $List_y(x)$, which takes $k \cdot |List_y(x)|$ messages, where k is the height of $ST_y(x)$, i.e. $x \in X_k$.
- If y is being removed as in Line (6) of `DeleteParent` procedure, we do nothing, since $N(y) = \emptyset$, i.e. u was the only node in $CT(y)$ previously.

3.3.4 Cost of Dynamic Maintenance

Lemma 3.6 *For each search tree $ST(y)$, $\forall y \in Y_i$ and $i \in [h]$, each leaf stores at most $(1/\epsilon)^{O(\alpha)} \cdot i^2$ keys. For the dynamic maintenance of $ST(y)$, it takes $(1/\epsilon)^{O(\alpha)} \cdot i^5$ amortized messages per node.*

Proof: By Lemma 3.4, each leaf stores a constant number of keys in average. Let z be a child of x, for a node x in $CT(y)$. Whenever $t_y(x) = s(x)$, we trigger a load-balancing procedure at the subtree $CT_y(x)$. Thus in the worst case, between any two load-balancing procedures on $CT_y(x)$, there are at most $s(x) - 1$ keys inserted into $CT_y(z)$. Since the subtree $CT_y(z)$ also performs load-balancing whenever there are $s(z)$ keys inserted, and since $s(x)/s(z) \leq (1/\epsilon)^{O(\alpha)} \cdot i$, in the worst case, these $s(x)-1$ keys give $(1/\epsilon)^{O(\alpha)} \cdot i$ additional keys per leaf in $CT_y(z)$ just after the last load-balancing procedure at $CT_y(z)$. Thus by counting all levels, in the worst case, a leaf in $ST(y)$ stores at most $(1/\epsilon)^{O(\alpha)} \cdot i^2$ keys.

The cost of maintaining $ST(y)$ comes from the load-balancing caused by the changes in the tree structure, and by the unbalanced insertions of keys. We use amortized analysis with one credit for each message.

Let's consider the changes in the search tree structure. During the insertion of a node u into the cluster tree $CT(y)$, whenever we increase $s_y(z)$ by one, where $z = p_k(u) \in CT(y)$ and $k \in [i]$, we deposit $(1/\epsilon)^{O(\alpha)} \cdot i^3 \cdot k$ credits. Thus when we add the branch $ST_y(z)$ into the search tree $ST(y)$ at vertex $x = p_{k+1}(u)$, we have $(1/\epsilon)^{O(\alpha)} \cdot i^3 \cdot k \cdot s_y(z)$ credits to do the balancing at $CT_y(x)$. The procedure costs at most $(1/\epsilon)^{O(\alpha)} \cdot i^2 \cdot k \cdot s_y(x)$ messages since each leaf has at most $(1/\epsilon)^{O(\alpha)} \cdot i^2$ keys. Since $s_y(x)/s_y(z) = (1/\epsilon)^{O(\alpha)} \cdot i$, we have enough credits for the cost of the procedure. Thus the total number of credits deposited by the insertion of a node is $\sum_{k=0}^{i}(1/\epsilon)^{O(\alpha)} \cdot i^3 \cdot k = (1/\epsilon)^{O(\alpha)} \cdot i^5$. Since we remove the branch $ST_y(z)$ from $ST(y)$ at x only when $s_y(x)/s_y(z) < \frac{1}{2b\cdot(i+1)}$ compared to the ratio $\frac{1}{b\cdot(i+1)}$ at which we add the branch $ST_y(z)$, by a similar argument, we pay $(1/\epsilon)^{O(\alpha)} \cdot i^5$ credits for the deletion of a node.

Now consider the cost caused by the unbalanced insertions of keys. During the insertion of a key k into $ST(y)$, whenever we increase $t_y(x)$ by one, for $x \in X_k \cap CT(y)$ s.t. $k \in Range_y(x)$ and $k \in [i]$, we deposit $(1/\epsilon)^{O(\alpha)} \cdot i^2 \cdot k$ credits. Thus when $t_y(x) = s(x)$, we have $(1/\epsilon)^{O(\alpha)} \cdot i^2 \cdot k \cdot s(x)$ credits to do the balancing at $CT_y(x)$, which costs at most $(1/\epsilon)^{O(\alpha)} \cdot i^2 \cdot k \cdot s(x)$ messages. Hence the total credits deposited by the insertion of a key is $\sum_{k=0}^{i}(1/\epsilon)^{O(\alpha)} \cdot i^2 \cdot k = (1/\epsilon)^{O(\alpha)} \cdot i^4$. Therefore each dynamic maintenance operation on $ST(y)$ takes $(1/\epsilon)^{O(\alpha)} \cdot i^5$ amortized messages per node. ■

Since a node belongs to at most $(1/\epsilon)^{O(\alpha)}$ cluster trees at each level, we have the following corollary.

Corollary 3.7 *Each node in V stores $(1/\epsilon)^{O(\alpha)} \log^3 \Delta$-bit routing information. It takes $(1/\epsilon)^{O(\alpha)} \log^6 \Delta$ amortized messages for a node to join or to leave the network.*

3.4 Name-Independent Routing

For each node $u \in V$, let $n(v)$ denote the arbitrary original node name, and assume that each name is represented in $O(\log n)$ bits; the label $\ell(u)$ defined in Section 3.2 has size of $O(\alpha \log \frac{1}{\epsilon} \log \Delta)$ bits. For each $y \in Y$, we maintain a search tree on the skeletal tree $ST(y)$ to store the labels $\ell(u)$ of all real nodes u (i.e. the leaf vertices) in the cluster tree $CT(y)$ using the name of u as the key. Thus every time a node u wants to communicate with a node v given $n(v)$, node u queries search trees along its parents to retrieve the label $\ell(v)$ using $n(v)$, and then routes to v using the underlying labeled routing scheme. The routing algorithm is presented in Algorithm 4.

Algorithm 4: Name-independent routing from a node u to v, given the name of v

```
1: for $i = 0$ to $h$ do
2:     Search on the search tree $ST(y)$ for the key $n(v)$, where $y = f(p_i(u))$
3:     if the data $\ell(v)$ for the key $n(v)$ is found then
4:         Go to $v$ using the underlying labeled routing scheme, and terminate the algorithm
```

Lemma 3.8 *The name-independent routing scheme has stretch $9 + O(\epsilon)$.*

Proof: Let j be the index such that the *if* condition in Line 3 of Algorithm 4 is satisfied, i.e. the label of v is found. Since the label of v is not found at level $j-1$, we have that $p_{j-1}(v)$ is not in $N(f(p_{j-1}(u)))$. Thus $d(p_{j-1}(v), p_{j-1}(u)) \geq 2^{j-1}/\epsilon - 2^{j-1}$. Since the cost of the search procedure at a search tree of level i is $2^{i+1}/\epsilon + O(2^i)$, the total routing cost in the virtual graph is $\sum_{i=0}^{j} 2^{i+1}/\epsilon + d(p_j(u), p_j(v)) + O(2^j) \leq 2^{j+2}/\epsilon + d(u,v) + O(2^j) \leq (9 + O(\epsilon))d(u,v)$.

By Invariant 3.1, the cost in the real graph is still bounded by $(9 + O(\epsilon))d(u,v)$. ■

4 Routing in the General Model

In this section, we consider dynamic routing in metrics with nodes joining, leaving **and** moving.

4.1 Dynamic Graph

In the general model, we allow nodes to move, and seek locality-sensitive protocols, in that the performance bounds depend on the range of motion of a node. Thus we are neither able to maintain (X, E_1) as the tree structure of $T(r)$, nor able to reuse the vertices $\in X$ for the cluster trees.

4.1.1 Dynamic Protocols

The node join/leave protocols are presented in Algorithms 5 and 6 respectively, which are based on the subprocedures `AddNewParent` and `DeleteParent` in Section 3.1. The main difference to the restricted model versions of these protocols is that we are not able to just append or delete a path into the global tree $T(r)$ when a node joins or leaves as in the restricted model, but we update its parent vertices level by level.

The node move protocol at node u is presented in Algorithm 7. Let i be the maximal index s.t. $d(p_j(u), u) > 2^{j+1}$, $\forall j \in [i]$. Then for level k from i down to 0, we combine both the loop actions of Algorithms 5 and 6, i.e. deleting the old parent vertex $p_k(u)$, and assigning a new parent vertex $p_k(u)$ within $B_u(2^k)$.

By Algorithm 7, whenever a node u updates $p_i(u)$ due to the movement of u, we have $d(p_i(u), u) > 2^{i+1}$ before the update; and by Algorithms 5, 6 and 7, we always assign a new $p_i(u)$ in $X_i \cap B_u(2^i)$ to u. Thus we have:

Invariant 4.1 *Whenever a node u updates $p_i(u)$ due to the movement of u, we have $d(p_i(u), u) > 2^{i+1}$ before the update and $d(p_i(u), u) \le 2^i$ after the update.*

```
Algorithm 5: A new node $u$ joins $V$
1: for $k = h$ downto 0 do
2:     if $\exists x \in X_k \cap B_u(2^k)$ then Set $p_k(u) = x$
3:     else AddNewParent($u, k$)
```

```
Algorithm 6: A node $u$ leaves $V$
1: for $k = h$ downto 0 do
2:     if $\exists v \neq u \in V$ s.t. $p_k(v) = p_k(u)$ then
3:         UpdateHost($u, k, v$)
4:     else DeleteParent($u, k$)
```

```
Algorithm 7: A node $u$ moves
1: Let $i$ be the maximal index s.t.
   $d(p_j(u), u) > 2^{j+1}$, $\forall j \in [i]$
2: for $k = i$ downto 0 do
3:     if $\exists v \neq u \in V$ s.t. $p_k(v) = p_k(u)$ then
4:         UpdateHost($u, k, v$)
5:     else DeleteParent($u, k$)
6:     if $\exists x \in X_k \cap B_u(2^k)$ then Set $p_k(u) = x$
7:     else AddNewParent($u, k$)
```

4.2 Labeled Routing

For each node $u \in V$, let its label $\ell(u) = \langle pnt(p_h(u)), pnt(p_{h-1}(u)), \cdots, pnt(p_0(u)) \rangle$. Assume that any point in the metric can be expressed in $\log \Delta$ bits. Thus the label has $\log^2 \Delta$ bits. The structure of the routing algorithm remains the same as Algorithm 3, except that (i)in Line 2 we check whether $p_i(v) \in N(f(p_i(u)))$ by testing whether $d(pnt(p_i(v)), f(p_i(u))) \le 2^i/\epsilon$; and (ii) in Line 4 the algorithm goes down from $p_{k+1}(v)$ to $p_k(v)$ by selecting a child of $p_{k+1}(v)$ with its point equal to $pnt(p_k(v))$. By a similar argument as in Theorem 3.2, we have Theorem 1.1.

4.3 Name-Independent Routing

We color each vertex x in X, using a color function $c : X \to [20^{O(\alpha)}]$ such that no two siblings in X share a color, where we say two vertices $x, x' \in X_i$, $\forall i \in [h-1]$, are sibling if $\exists z \in X_{i+1}$ s.t. edges (z, x) and (z, x') are in E_1'. Note that the number of siblings of any vertex in X is at most $20^{O(\alpha)}$ by Lemma 2.2 and Invariant 4.1. Thus we can always find a valid color for a newly added vertex.

For each cluster tree $CT(y)$, $\forall y \in Y_i$ and $\forall i \in [h]$, the search tree $ST(y)$ still uses $n(u)$ as the key, for all real nodes u in $CT(y)$, but stores the ID of $p_i(u)$ and colors $\langle c(p_i(u)), c(p_{i-1}(u)), \cdots, c(p_{i-\log(1/\epsilon)}(u)) \rangle$ as the data. The ID of $p_i(u)$ takes $O(\alpha \log(1/\epsilon))$ bits, since $p_i(u) \in N(y)$ and $|N(y)| = (1/\epsilon)^{O(\alpha)}$.

The improved name-independent routing scheme is presented in Algorithm 8, which is similar to Algorithm 4. The main difference is that we are no longer able to use the labeled routing scheme. Thus once we get the color data of v at level i, we are only able to go down to $p_{i-\log(1/\epsilon)}(v)$. Then we recursively get the color data of v at level $i - \log(1/\epsilon)$ by querying the search tree $ST(f(p_{i-\log(1/\epsilon)}(v)))$ and go down $\log(1/\epsilon)$ levels further.

Proof of Theorem 1.2: Note that a node u might have $2^{O(\alpha)}i$ links in each descendant tree $T(p_i(u))$, $\forall i \in [h]$. Thus, overall the number of links at u is $2^{O(\alpha)} \log^2 \Delta$.

By Lemma 3.6, we achieve the same result for the name-independent routing scheme in the general model as in Corollary 3.7. By a similar argument of Lemma 3.6, for each node-move protocol at level k, it takes $(1/\epsilon)^{O(\alpha)} \log^4 \Delta \cdot k^2$ amortized number of messages, and each message traverses a path of distance $O(2^k/\epsilon)$. By a more careful argument as in 3.8, Algorithm 8 achieves $9 + O(\epsilon)$ stretch. Thus Theorem 1.2 follows. ■

Algorithm 8: Improved name-independent routing from a node u to v, given the name of v

```
1: for $i = 0$ to $h$ do
2:     $y \leftarrow f(p_i(u))$
3:     Go to $y$, and search on the search tree $ST(y)$ for the key $n(v)$
4:     if the data $\langle c(p_i(v)), c(p_{i-1}(v)), \cdots, c(p_{i-\log(1/\epsilon)}(v))\rangle$ for the key $n(v)$ is found then
5:         Go to $p_i(v)$; and break
6: while true do
7:     Go down to $p_{i-\log(1/\epsilon)}(v)$ from $p_i(v)$ using the color data
8:     if we reach $p_0(v)$ then terminate the algorithm
9:     $i \leftarrow i - \log(1/\epsilon)$
10:    Search on the search tree $ST(f(p_i(v)))$ for the key $n(v)$
```

5 Scale-Control Procedure

In this section, we provide a scale-control procedure to adapt our schemes to dynamic network size and diameter.

First we discuss how to adapt our schemes to a dynamic number of nodes in the network. Note that the original node name might be expressed in $\log n$ bits, where n is the total number of nodes over all time. However the number n_t of nodes at the current time t, might be much less than n. We use a universal hash function to hash the original names into a value of $c \log n_t$ bits, where $c > 2$ is a constant. Carter and Wegman [2] provides such universal hash function that is represented in $O(\log n_t)$ bits. Whenever the number of nodes in the network is squared or square-rooted, we update the hash function so that the number of bits for each hash value increases or decreases c bits.

Second, we discuss how to adapt our schemes to dynamic network diameter. We update the hierarchical level h according to the changes in the network diameter.

Invariant 5.1 *We preserve two invariants: (i) $N(f(x)) = X_h, \forall x \in X_h$; and (ii) $|X_{h-\log\frac{1}{\epsilon}}| > 1$.*

Whenever we insert a new vertex into X_h, we check whether Invariant 5.1 (i) is preserved. The invariant is not preserved, iff the diameter increases to $2^h/\epsilon$. We recursively define the parent set $X_{h+i} \subseteq X_{h+i-1}$ to be a 2^{h+i}-net covering X_{h+i-1}, for $i = 1$ up to a value j s.t. $|X_{h+j}| = 1$ (Note that $j = \log\frac{1}{\epsilon} + O(1)$). In the meanwhile, we add cluster header sets $Y_{h+i} = X_{h+i}$ for $i = 1, \cdots, j$, the cluster tree and the search tree for each newly added cluster header. Then we update $h = h + j$. Note that by Lemma 3.6, $(1/\epsilon)^{O(\alpha)}h^6$ amortized messages per node in the current network suffice for the operation.

Whenever we delete a vertex in X_h, we check whether Invariant 5.1 (ii) is preserved. If not, we drop all X_{h-k} and Y_{h-k} for $k = 0, \cdots, \log\frac{1}{\epsilon} - 1$, and update $h = h - \log\frac{1}{\epsilon}$. Note that a constant amortized number of messages per node in the current network suffice for the operation.

Thus we achieve the result in Theorem 1.3.

6 Discussion and Future work

There are a number of directions that should be investigated further. The delay in dynamic protocols results from two basic operations: the creation of new search trees and load balancing. For search tree creation, merge sort may help. We maintain a search tree for each descendant tree $T(x)$, to store the keys of nodes. Thus for a new cluster tree $CT(y)$, instead of inserting keys for all nodes in $CT(y)$, we just merge all sorted lists in the search trees of all descendant trees $T(x)$, $\forall x \in N(y)$, into a single list for $ST(y)$. For load-balancing, a divide-and-conquer approach may be useful. Note that the load-balancing procedure distributes a sorted list across a search tree. With the help of the search tree, it is easy to divide the sorted list into equal-size segments and then let the node of each segment send the segment to the destination in parallel.

Another issue is the control of hot-spot nodes in the case where the bandwidth of each link is limited. For example, roots of search trees may suffer from congestion. Further, in this paper, we only consider one topology change at a time. Can our routing schemes be improved to deal with multiple concurrent changes of the topology? Finally, it would be interesting to generalize the compact routing problems to dynamic graphs.

References

[1] I. Abraham, C. Gavoille, A. V. Goldberg, and D. Malkhi. Routing in networks with low doubling dimension. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 75, 2006.

[2] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and Systems Sciences*, 18:143–154, 1979.

[3] H. T.-H. Chan, A. Gupta, B. Maggs, and S. Zhou. On hierarchical routing in doubling metrics. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 762–771, 2005.

[4] L.-A. Gottlieb and L. Roditty. Improved algorithms for fully dynamic geometric spanners and geometric routing. In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*, 2008.

[5] A. Gupta, R. Krauthgamer, and J.R.Lee. Bounded geometries, fractals and low-distortion embeddings. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science*, pages 534–543, 2003.

[6] G. Konjevod, A. Richa, and D. Xia. Optimal-stretch name-independent compact routing in doubling metrics. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, pages 198–207, 2006.

[7] G. Konjevod, A. Richa, and D. Xia. Optimal scale-free compact routing schemes in networks of low doubling dimension. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 939–948, 2007.

[8] G. Konjevod, A. Richa, D. Xia, and H. Yu. Compact routing with slack in low doubling dimension. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, 2007.

[9] A. Slivkins. Distance estimation and object location via rings of neighbors. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 41–50, 2005.

[10] A. Slivkins. Towards fast decentralized construction of locality-aware overlay networks. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, 2007.

[11] K. Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, pages 281–290, 2004.

[12] D. Tschopp, S. Diggavi, and M. Grossglauser. Hierarchical routing in dynamic α-doubling networks. Technical report, 2007. http://licos.epfl.ch/Papers/TDGdoubling.pdf.

Appendices

A Notation List

- (M,d): A metric space, where M is a set of points, $d : M \times M \to R^+$ is a distance function, and assume $min_{x \neq y \in M} d(x,y) = 1$
- $B_x(r)$ for a point $x \in M$ and $r \in R^+$: A set of points in M within distance r to x
- α: The doubling dimension of M, i.e. the least value α such that any ball $B_x(r)$ in M can be covered by at most 2^α balls with half the radius
- V: The dynamic set of nodes
- Δ: The maximum diameter of the node set V over all time
- n: The total number of nodes over all time
- $h = \log \Delta$: The highest level of the hierarchies
- ϵ: The small addition to the stretch (The labeled routing scheme has stretch $1 + O(\epsilon)$, while the name-independent routing scheme has stretch $9 + O(\epsilon)$)
- X_i: A parent set of vertices at level i, which is a 2^i-net, for $i \in [h]$
- Y_i: A cluster header set, which is a 2^i-net covering X_i, for $i \in [h]$
- $p_i : V \to X_i$: A parent mapping at level i that maps each node u in V to a parent vertex $p_i(u)$ in X_i
- $f : X_i \to Y_i$: A header mapping that maps any vertex x in X_i to a cluster header vertex $f(x)$ in Y_i s.t. $d(x, f(x)) \leq 2^i$
- $N : Y_i \to X_i$: A neighborhood mapping that maps any cluster header $y \in Y_i$ to a set of parent vertices in X_i within distance $2^i/\epsilon$ to y, i.e. $N(y) = X_i \cap B_y(2^i/\epsilon)$
- $CT(y)$ for each $y \in Y_i$: A cluster tree over nodes whose parents at level i are in the neighborhood $N(y)$
- $X = \cup_{i=0}^h X_i$
- $Y = \cup_{i=0}^h Y_i$
- $V' = X \cup Y \cup \cup_{y \in Y} V(CT(y))$: The set of virtual vertices
- $E' = E'_1 \cup E'_2 \cup E'_3 \cup \cup_{y \in Y} E(CT(y))$: The set of virtual edges
- $\phi : V' \to V$: A host mapping that maps any virtual vertex in V' to a real node in V
- $E = \{(\phi(x), \phi(y)) \ : \ (x,y) \in E'\}$: The set of links such that the two end nodes of any link are able to communicate directly
- $G' = (V', E')$: The virtual graph
- $G = (V, E)$: The real dynamic graph
- $pnt : V \cup V' \to M$: A point mapping that associates any node in V and any virtual vertex in V' to a point in M
- *Points*: elements in M; *Nodes*: elements in V; *Vertices*: elements in V'; *Links*: elements in E; *Edges*: elements in E'
- *Restricted vs. General Models*: In the restricted model, nodes are allowed to join and leave the network, while in the general model, nodes are allowed to join, leave, *and* move.
- r in the restricted model: the root vertex such that $p_{h+1}(u) = r$, $\forall u \in V$

- $c : X \to constant^{O(\alpha)}$: a color function s.t. no two siblings share a color, where we say two vertices $x, x' \in X_i$ are sibling if $\exists z \in X_{i+1}$ s.t. edges (z, x), (z, x') are in E'_1
- $\ell(u)$: the label of u, that is $\ell(u) = \langle c(p_h(u)), c(p_{h-1}(u)), \cdots, c(p_0(u)) \rangle$ in the restricted model, and $\ell(u) = \langle pnt(p_h(u)), pnt(p_{h-1}(u)), \cdots, pnt(p_0(u)) \rangle$ in the general model
- $n(u)$ for all $u \in V$: the name of u

A.1 The notation list with regard to search trees

- $ST(y)$ for each $y \in Y_i$: A skeletal tree of the cluster tree $CT(y)$, which is also referred to as the *search tree* for $CT(y)$
- $CT_y(x)$ for $y \in Y_i$, $i \in [h]$ and $x \in CT(y)$: the subtree rooted at x of the cluster tree $CT(y)$
- $ST_y(x)$ for $y \in Y_i$, $i \in [h]$ and $x \in ST(y)$: the subtree rooted at x of the skeletal tree $ST(y)$
- $s_y(x)$ for $y \in Y$ and $x \in CT(y)$: the number of leaves in the subtree $CT_y(x)$
- $Range_y(x)$ for $y \in Y$ and $x \in ST(y)$: the minimal interval that contains all keys stored in $ST_y(x)$
- $List_y(x)$ for $y \in Y$ and $x \in ST(y)$: the sorted list of all keys stored in $ST_y(x)$
- $t_y(x)$ for $y \in Y$ and $x \in ST(y)$: A counter at x that increases by one whenever a key is inserted to $ST_y(x)$, and triggers a load balancing when $t_y(x) = s_y(x)$