Decentralized Pairwise Bandwidth Prediction

Sukhyun Song
Department of Computer Science, University of Maryland
shsong@cs.umd.edu

Abstract

The distributed nature of modern computing makes end-to-end prediction of network behavior increasingly important. However, traditional network coordinate systems predict latencies only. The single extant system for bandwidth prediction is centralized, and suffers from load imbalance and low accuracy when driven from imprecise (real-world) network measurements. In this paper, we present an efficient, accurate, and decentralized system that predicts pairwise bandwidths. We first describe a decentralized node join algorithm that balances measurement workload across the system. Second, we prove the correctness of our bandwidth prediction when driven by precise (ideal world) measurements. Finally, we describe three novel heuristics that achieve high accuracy even with imprecise datasets. Simulation experiments with a real-world dataset confirm that our approach provides better load balance with less overhead than prior approaches.

Keywords: bandwidth prediction, decentralized, tree embedding

1 Introduction

Peer-to-peer (P2P) applications operate in the real Internet, and many can therefore benefit from knowing latencies to other hosts. For example, in P2P storage systems such as CFS [7] and PAST [17], a client can obtain a file quickly by selecting a replica from the closest node, in network terms. Network coordinate systems such as GNP [13] and Vivaldi [6] provide applications with the ability to predict end-to-end latency so that P2P applications can increase their overall performance.

As application bandwidth requirements increase, bandwidth prediction is also becoming increasingly important. Unlike latency, however, bandwidth measurements are generally expensive, and often need to be done in a pairwise fashion for accuracy. For example, a P2P computational grid system [10, 9, 11, 12] could benefit from bandwidth prediction when searching for high-bandwidth nodes (and links) to store a large scientific dataset. Based on the intuition that the Internet is tree-like, the Sequoia system embeds bandwidth prediction measurements as distances on tree graphs, and is highly successful at bandwidth prediction with idealized datasets [14, 15]. However, Sequoia has several limitations when applied to real-world P2P applications. First, Sequoia's centralized join algorithm can lead to significant load imbalance. More importantly, Sequoia's accuracy suffers when presented with real-world datasets. Datasets derived from actual bandwidth measurements can contain highly inconsistent data because measurements are accumulated over time, and can be greatly affected by dynamic network conditions.

We feel that a fully decentralized bandwidth prediction system must consider five requirements all at once: i) there must exist no central server, ii) bandwidth should be predicted accurately, iii) measurement workloads should be balanced across nodes, iv) measurement traffic should be scalable, and v) the system should be able to adapt to network changes.

The overall contribution of this paper is the design and evaluation of the first decentralized bandwidth prediction system satisfying all of the above requirements. The system matches Sequoia's 100% prediction accuracy on idealized datasets, and we provide the first proof of a bandwidth prediction algorithm's correctness. The system also uses a decentralized node join algorithm that balances measurement load. We provide three novel heuristics to achieve prediction accuracy on imprecise (real world) datasets. Finally, we present extensive simulation results validating high accuracy, low cost, and balanced measurement traffic.

The rest of the paper has the following organization. We first discuss related work in Section 2. Section 3 discusses the underlying theory and the *treeness* of network bandwidth. Section 4 describes the details of system design, and provides a proof of correctness. Section 5 presents the techniques used to support real world, imprecise, datasets. Finally, Section 6 evaluates our approach, and we conclude and discuss future work in Section 7.

2 Related Work

GNP [13], Vivaldi [6], and PIC [5], provide predictions of end-to-end latency. These systems assign each node synthetic coordinates in an almost Euclidean space. End-to-end latencies are then computed as the distance between the sets of coordinates of the two nodes. However, these systems are not designed to support bandwidth prediction. Attempts to use Vivaldi for bandwidth prediction, for example, result in poor accuracy [14, 15].

The original theoretical underpinnings of tree metric spaces were provided by Buneman [4], including the first constructive algorithm to induce tree metric spaces. However, Buneman's algorithm does not allow nodes to be incrementally added to existing weighted trees. Since the result weighted tree would not be expandable, we cannot directly apply the algorithm in practice to a dynamically changing real-world system.

This research is inspired by the Sequoia system [15], which uses a weighted tree embedding model for bandwidth prediction. Our design differs from Sequoia's primarily in three aspects. First, we resolve Sequoia's load imbalance problem that is caused by a single measurement starting point. Second, we prove the correctness of the prediction tree construction algorithm related to its 100% accuracy and fast end node searching. This proof can also be used to prove the correctness of the Sequoia algorithm since our algorithm is a generalized version of Sequoia's. Third, we provide three heuristics that increase the prediction accuracy in the real world, the former two without imposing any additional costs.

Our overall research on P2P desktop grid systems [10, 9, 11, 12] motivates this work. A scientific job running in a P2P desktop grid may need to store large-scale scientific output data into the grid, for example. Such stored data may then need to be retrieved as an input dataset by another job. If the application or grid system can select a high-bandwidth storage node with the help of our prediction algorithm, overall job turnaround time will be reduced, and the overall throughput of the grid can be maximized.

3 Background

Definitions: A *weighted tree* is a connected graph without cycles and with non-negative edge weights. The distance between two nodes u and v on a weighted tree T, denoted $d_T(u, v)$, is defined by the sum of weights of edges on the path $u \sim v$. A weighted tree T *induces* a metric space (V, d) if and only if T contains all nodes in V and $\forall u, v \in V,\ d(u, v) = d_T(u, v)$. The *four-point condition (4PC)* on a metric space (V, d) states that for any set of four points $w, x, y, z \in V$, $d(w, x) + d(y, z) \leq d(w, y) + d(x, z) \leq d(w, z) + d(x, y) \Rightarrow d(w, y) + d(x, z) = d(w, z) + d(x, y)$. A metric space (V, d) satisfies 4PC if and only if there exists a weighted tree that induces (V, d) [4]. A metric space satisfying 4PC is called a *tree metric space*.

Bandwidth and Metric Space: Higher is considered better for bandwidth while closer is generally better for distance in a metric space. So, the *linear transform function* $d(u, v) = C_{large} - BW(u, v)$ can be used to represent bandwidth as a metric where $BW(u, v)$ is the bandwidth between nodes u and v, $d(u, v)$ is the distance in a metric space, and C_{large} is a large constant that keeps $d(u, v)$ from becoming negative. For example, C_{large} can be set to the expected maximum bandwidth between any two nodes in the system.

Treeness of the Internet: There is an experimental result [15] that says the Internet bandwidth is an approximate tree metric space by showing small 4PC-ϵ values for a real-world dataset. A 4PC-ϵ parameter quantifies the deviation from 4PC [2], and can be computed for each set of four nodes in a metric space. All the 4PC-ϵs for any four nodes in a perfect tree metric space are zero. Large 4PC-ϵs mean the metric space is far from a tree metric space. The HP-PlanetLab bandwidth dataset used in [14, 15] was created by collecting available bandwidth measurements between PlanetLab nodes. They show 80% of the 4PC-ϵ values for the dataset are less than 0.2. This dataset will also be used in Section 6 to evalute our system.

Bottleneck Models: We consider two possible network topologies that differ in where bottlenecks usually occur. Both can be represented as perfect tree metric spaces. Bandwidth between two nodes is bottlenecked at the access link of one of the two nodes in the *access bottleneck model*. This model accurately describes a system of end users attached to the Internet via low-bandwidth links. It has been proved that a metric space for this model satisfies 4PC [14]. The *backbone bottleneck model* describes a system where a link in a network hierarchy is more likely to be a bottleneck due to aggregated traffic. This model is often appropropriate for hierarchical clusters, such as large-scale data centers, or clusters of such data centers, following conventional tree structuring in network switches and servers [1, 3]. For simplicity, we use a balanced k-ary tree structure for a hierarchical network topology in this model. A weighted tree can be generated by assigning proper edge weights so that the distance between two leaf nodes corresponds to bandwidth. Therefore the metric space for this model also satisfies 4PC.

4 Design

This section first provides a design overview. Details are then described, starting from a basic centralized algorithm that contains the key idea about load balancing. The algorithm is extended to a second centralized algorithm that reduces measurement traffic. Finally, we discuss how to build a decentralized algorithm. The overall algorithm guarantees perfect accuracy for an ideal world that can be represented as a tree metric space

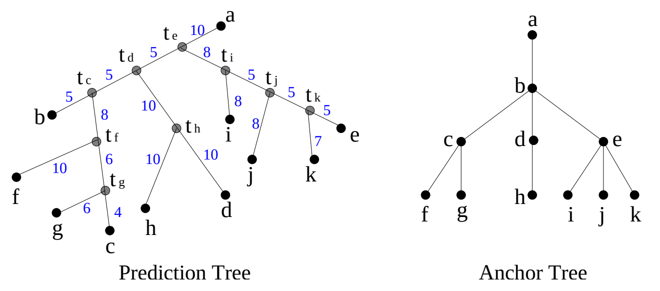


Figure 1: A Prediction Tree and a Corresponding Anchor Tree

satisfying 4PC. We also show proofs of correctness for the algorithms. Some discussion about how to support the real world, where 4PC does not hold completely, is provided in Section 5.

Overview: A logical *prediction tree* is constructed in a distributed way and is used to predict pairwise bandwidth between nodes. A prediction tree is a weighted tree that embeds bandwidth information. Figure 1 shows an example prediction tree. The number on each edge represents the weight of the edge. A leaf node in a prediction tree has degree one and represents each participating host in the system. An inner node with degree two or more is called a *virtual anchor node.* As described in Section 3, the linear transform function $d(u, v) = C_{large} - BW(u, v)$ is used to embed bandwidth into a metric space, and $BW_T(u, v) = C_{large} - d_T(u, v)$ for bandwidth prediction. For example, in Figure 1 if $C_{large} = 100$, the predicted bandwidth value $BW_T(b, c)$ is 77 because $d_T(b, c) = 23$. Physically, our system is an overlay network on the nodes, and the overlay structure is called an *anchor tree.* It is a hierarchical structure constructed based on a special relationship between nodes. We use the anchor tree to accelerate the construction of the prediction tree, by reducing the measurement traffic needed to add a new node. As an example, Figure 1 shows an anchor tree corresponding to the prediction tree to its left. Each node is assigned a *distance label* that contains information for a partial prediction tree. The distance labels are used to compute bandwidth without constructing a complete prediction tree.

4.1 Centralized Algorithm to Construct a Prediction Tree

Algorithm 1: **ConstructTree(T, x)**: It adds a node x to a prediction tree T without a fixed base node.

```
1 Pick a base node z from any leaf nodes in T
2 Measure $d(x,p)$ for all leaf nodes $p$ in $T$     // $d(x,p) = C_{large} - BW(x,p)$
3 Find an end node $y$ that maximizes $(x|y)_z$     // $(x|y)_z = \frac{1}{2} \times \{d(z,x) + d(z,y) - d(x,y)\}$
4 Put an inner node $t_x$ on the path $z \sim y$ where $d_T(t_x, z) = (x|y)_z$
5 Add $x$ by adding edge $(t_x, x)$ with weight $(y|z)_x$
```

We describe a centralized algorithm to construct a prediction tree, shown as Algorithm 1. The algorithm constructs a prediction tree by adding nodes one by one. To add a new node x to a prediction tree, the algorithm chooses a node z called the *base node,* which can be any leaf node, and selects another node y called the *end node* that maximizes *Gromov product* $(x|y)_z$. The Gromov product of x and y at z, denoted $(x|y)_z$, is defined as $(x|y)_z = \frac{1}{2} \times \{d(z, x) + d(z, y) - d(x, y)\}$. How x is positioned between z and y is shown in lines 4∼5 of Algorithm 1. From the proof for Theorem 4.1 provided in Appendix, we know that the prediction tree constructed by Algorithm 1 correctly embeds bandwidth information for a tree metric space.

Theorem 4.1 *Let T be a weighted tree that induces a tree metric space (V, d). Let x be a new node that is added to (V, d), so that the new (V, d) still satisfies 4PC. If we add x to T using Algorithm 1, then the new T will induce the new (V, d).*

Unlike Sequoia's tree construction algorithm [15] with a fixed base node, Algorithm 1 starts measurement at a random base node. This provides the theoretical underpinnings for our decentralized system. Since Algorithm 1 is a generalized version of Sequoia's algorithm, the proof of Theorem 4.1 can also be used to prove the correctness of Sequoia's fixed-base algorithm, which was not provided in their papers.

4.2 Reducing Measurement Traffic with the Anchor Tree

Lines 2∼3 of Algorithm 1 show a naive way to find an end node. A new node has to measure the bandwidth to all the hosts in the system, which takes $O(N)$ time. By constructing an anchor tree, as we now describe, the search time for an end node can be decreased.

Before describing the anchor tree, we must discuss a special relationship between two nodes called an *anchor relationship*. An anchor relationship is established when a new node is added to a prediction tree. When adding a new node x at lines 4∼5 of Algorithm 1, we define the inner node t_x as the *virtual anchor node* of node x and x as the *edge-owner node* of edge (t_x, x). If t_x splits edge e on the path from base node z and end node y, the edge-owner node a of edge e is defined as the *anchor node* of x. (If t_x is put at the same position as another virtual anchor node of node w, w's anchor node will be x's anchor node.) In this way, x establishes an anchor relationship with its anchor node a. In Figure 1, when adding h to a prediction tree, h's virtual anchor node t_h splits edge (t_d, d) whose edge-owner node is h's anchor node d, h becomes the edge-owner node of (t_h, h).

An anchor tree is a rooted tree where each node represents a host in the system. Each time a new node x is added to a prediction tree, x is also added to an anchor tree. If x is assigned its anchor node a while constructing a prediction tree, x becomes a's child node in an anchor tree. The first added node in the system becomes the root node of the anchor tree.

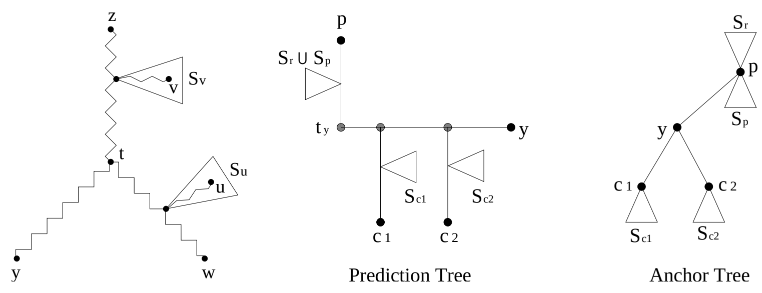
Algorithm 2: **FastConstructTree(T, A, x)**: It adds a node x to a prediction tree T and an anchor tree A, using a fast end node searching technique.

```
1  Pick a base node $z$ from any leaf nodes in $T$
2  $y \leftarrow z$
3  while true do
4      Scand $\leftarrow \{y, y$'s parent and child nodes in $A\}$    // a set of end node candidates
5      Measure $d(x,s)\ \forall s \in$ Scand
6      Gmax $\leftarrow \text{argmax}_{s \in \text{Scand}} (x|s)_z$
7      if $y \in$ Gmax then break else $y \leftarrow$ one node in Gmax
8  Put an inner node $t_x$ in $T$ on the path $z \sim y$ where $d_T(t_x, z) = (x|y)_z$
9  Add $x$ to $T$ by adding edge $(t_x, x)$ with weight $(y|z)_x$
10 $x$ becomes the edge-owner node of $(t_x, x)$
11 $a \leftarrow$ the edge-owner node of the edge that $t_x$ is located on
12 $a$ becomes the edge-owner node of two edges split by $t_x$ and becomes $x$'s anchor node
13 Add $x$ to $A$ as $a$'s child node
```

We now describe Algorithm 2, which is an improved version of Algorithm 1 with respect to end node search time. To find an end node, the algorithm moves up and down the anchor tree in the direction that locally maximizes the Gromov product until reaching a global maximizer (lines 2∼7). After adding a new node to a prediction tree, the algorithm also adds the node to an anchor tree by defining an anchor relationship (lines 10∼13). Theorem 4.2 and Theorem 4.3 are provided to prove correctness for Algorithm 2. These theorems deal with several cases where part of a prediction tree can be excluded from end node searching. You can see Figure 2(a) to understand these theorems and the proofs more easily. The proofs are provided in Appendix.

Theorem 4.2 *In the setting of Theorem 4.1, for three leaf nodes $z, y, w \in T$, let $t \in T$ be an inner node on the path $z \sim y$ at distance $(y|w)_z$ from z. Let S_u be a set of nodes in all the subgraphs connected to the inner nodes on the path $t \sim w$. (When the degree of t is greater than 3 and t has extra paths other than $t \sim z$, $t \sim y$, and $t \sim w$, S_u also includes the nodes associated with those extra paths from t.) Let S_v be a set of nodes in all the subgraphs connected to the inner nodes on the path $t \sim z$. If $(y|x)_z > (w|x)_z$, then $\forall u \in S_u, (y|x)_z > (u|x)_z$, and $\forall v \in S_v, (y|x)_z > (v|x)_z$.*

Theorem 4.3 *In the setting of Theorem 4.1 and 4.2, if $(y|x)_z = (w|x)_z$, then $\forall u \in S_u, (y|x)_z = (u|x)_z$.*



(a) Theorem 4.2 and 4.3 (b) Algorithm 2

Figure 2: Prediction Tree and Anchor Tree Used in Theorems and the Proof of Algorithm Correctness

Correctness of Algorithm 2: Since the correctness of Algorithm 1 has already been proved, we focus on showing that the while loop (lines 2∼7) correctly finds a global maximizer by moving around an anchor tree. Figure 2(b) shows a prediction tree and an anchor tree, from the perspective of node y. y has p as its parent in the anchor tree. c_1 and c_2 are y's child nodes. S_p, S_{c1}, and S_{c2} are a set of p's child nodes, c_1's, and c_2's, respectively. S_r is a set of remaining nodes in the system on p's parent side. t_y is y's virtual anchor node. Suppose Algorithm 2 is currently visiting y. We consider four cases, in terms of the location of base node z, and determine whether the algorithm makes a correct decision to find an end node. Each case is divided into four sub-cases according to where the computed local maximizer is. The notation $G(a) = (x|a)_z$ is used for simplicity. Since all cases can be proved using the same proof pattern, we only provide the proof for Case 2.

- **Case 1:** $z = y$ (This case can be proved similarly to Case 2.)
- **Case 2:** $z \in \{p\} \cup S_r \cup S_p$
  The algorithm must have visited p before coming to y. This means $G(y) > G(s)\ \forall s \in \{p\} \cup S_r \cup S_p$.
  - Case 2a: $y \in \mathsf{Gmax}$
    $G(y) \geq G(c_1) \Rightarrow G(y) \geq G(s)\ \forall s \in S_{c1}$ by Theorem 4.2 and 4.3
    $G(y) \geq G(c_2) \Rightarrow G(y) \geq G(s)\ \forall s \in S_{c2}$ by Theorem 4.2 and 4.3
    So, y must be an end node, and the algorithm correctly exits the while loop.
  - Case 2b: $y \notin \mathsf{Gmax} \land p \in \mathsf{Gmax} \Rightarrow G(p) > G(y)$ (impossible because $G(y) > G(p)$)
  - Case 2c: $y \notin \mathsf{Gmax} \land c_1 \in \mathsf{Gmax} \Rightarrow G(c_1) > G(y)$
    $G(c_1) \geq G(c_2) \Rightarrow G(y) \geq G(s)\ \forall s \in S_{c2}$ by Theorem 4.2 and 4.3
    So, the global maximizer must exists in $S_{c1} \cup \{c_1\}$, and the algorithm correctly moves to c_1.
  - Case 2d: $y \notin \mathsf{Gmax} \land c_2 \in \mathsf{Gmax}$ (similar to Case 2c)
- **Case 3:** $z \in \{c_1\} \cup S_{c1}$ (This case can be proved similarly to Case 2.)
- **Case 4:** $z \in \{c_2\} \cup S_{c2}$ (This case can be proved similarly to Case 2.)

In this way, Algorithm 2 correctly moves in the direction towards where a global maximizer exists, by skipping measurements for nodes in the other directions. Since the algorithm never moves back to an already visited node, like in Case 2b, the while loop (lines 2∼7) will terminate once it finds a global maximizer.

Performance: The number of measurements needed to add a node determines the performance of Algorithm 2. That depends on the number of visited nodes and the degree (the number of neighbors) of each visited node, which are both dependent on the shape of the anchor tree and the locations of the base node and end node. Also, for a given tree metric space, an anchor tree can have different shapes from different node addition orderings.

We can consider tree metric spaces derived from two theoretical models as described in Section 3. For both models, the best case is when the measurement starts at a leaf node and ends at its parent node that has no other child nodes, so takes $O(1)$ measurements. In the worst case, the measurement starts at a leaf node, passes the only child node of the root, and ends at another leaf node. That will take $O(maxdepth \times maxdegree)$ measurements where $maxdepth$ is the maximum depth of any node and $maxdegree$ is the maximum degree of any node. For some poor orderings for adding nodes in the access bottleneck model, the algorithm can produce a long chain-style tree of $O(N)$ depth or a shallow tree with a node of $O(N)$ degree, so that the worst case number of measurements taken is $O(N)$. On the other hand, the backbone bottleneck model involves $O(log^2 N)$ measurements in the worst case. *maxdepth* and *maxdegree* in an anchor tree are determined by the maximum number of inner nodes of all the paths between two leaf nodes in a prediction tree. For example, say path $r \sim a$ in a prediction tree has the maximum number of inner nodes, I. Let the inner nodes be denoted by $t_1, t_2, ..., t_I$, starting from the one closest to r. One node in the subgraph associated with t_i is denoted by n_i. An anchor tree can have *maxdepth* $(I + 1)$ if we add r first, then $n_1, n_2, ..., n_I$, and a last. In this case, n_1 becomes r's child, n_i becomes n_{i-1}'s child, and a becomes n_I's child. *maxdegree* is I when we add r first, a second, and then $n_1, n_2, ..., n_I$. r becomes the root, a becomes r's child node, and all $n_1, n_2, ..., n_I$ will be a's child nodes. For the balanced k-ary prediction tree of the backbone bottleneck model, the maximum number of inner nodes is $2log_k N$, when we consider the path from one leaf node to another passing through the top-level node. So, both *maxdegree* and *maxdepth* are $O(logN)$, and the algorithm will take $O(log^2 N)$ measurements.

Load Balance: Sequoia's fixed-base algorithm [15] starts measurement at the root node for every newly added node. On the other hand, our random-base algorithm does not have any fixed starting point, which enables better balanced measurement traffic. Say we add a new node to the anchor tree, for example to the tree shown in Figure 1. Assume that a base node and an end node are chosen in a uniformly random way. In the fixed-base algorithm, a and b suffer the maximum expected number of measurements, which is 1, and h has the minimum expected number of measurements, which is $\frac{2}{11} = 0.18$. In the random-base algorithm, b has maximum value $\frac{107}{121} = 0.88$, and h has minimum value $\frac{40}{121} = 0.33$. The standard deviation is 0.325 for the fixed-base algorithm while it is 0.197 for the random-base algorithm, which shows that our approach has relatively good load balance.

4.3 Decentralization

A *distance label* is assigned to each node, so that we can construct a prediction tree in a decentralized fashion. Node x's distance label contains all anchor nodes on the path from the root node to x in the anchor tree. The label also contains the corresponding distance values between anchor nodes and virtual anchor nodes. In other words, a distance label contains a partial prediction tree. A whole prediction tree can be constructed using the distance labels from all nodes in the system. If there are k anchor nodes in x's distance label, from x's anchor node a_1 to the root node a_k, x's distance label is denoted by:

$$(a_k \xrightarrow[d_T(t_{a_{k-1}}, a_{k-1})]{d_T(a_k, t_{a_{k-1}})} a_{k-1} \xrightarrow{\hspace{2em}} ... \xrightarrow{\hspace{2em}} a_2 \xrightarrow[d_T(t_{a_1}, a_1)]{d_T(a_2, t_{a_1})} a_1 \xrightarrow[d_T(t_x, x)]{d_T(a_1, t_x)} x)$$

(a_{i+1} is a_i's anchor node, and t_{a_i} is a_i's virtual anchor node for $1 \leq i \leq k-1$.)

For example, node d in Figure 1 has $(a \xrightarrow[25]{0} b \xrightarrow[20]{10} d)$ as its distance label, because $d_T(a, t_b) = 0$, $d_T(t_b, b) = 25$, $d_T(b, t_d) = 10$, and $d_T(t_d, d) = 20$. By introducing distance labels, the distance between two nodes can be estimated by Algorithm 3, shown in Appendix without constructing a complete prediction tree.

Decentralized Node Join Algorithm: We next discuss the decentralized node join algorithm. Since we have eliminated a single bottleneck measurement starting point, designed an anchor tree structure, and reduced node join cost using the anchor tree as previously described, we see that Algorithm 2 already has the potential to be decentalized. We provide a brief description here, and the detailed decentralized node join algorithm is provided in Appendix. Suppose there is an overlay network following the anchor tree structure, and each node has a distance label containing a partial prediction tree. A newly joining node x first chooses an arbitrary base node z from the existing system. z can be chosen in the same way as a bootstrapping node is identified in a general peer-to-peer system, for example, via a DNS service that does not have to have perfect information. x measures bandwidth to some existing nodes while moving around the overlay network until finding a global maximizer y. x determines its anchor node a and assigns it a distance label based on the distance labels of z and y and measured values $d(z, x)$ and $d(y, x)$. This is possible because the distance labels of z and y have the necessary information from the partial prediction trees to compute x's distance label. x becomes a's child node by notifying a of its join event. The performance of this node join algorithm, in terms of the number measurements, is the same as for Algorithm 2.

Maintenance: Our system maintains and restructures the overlay network in response to a changing network environment by having each node send periodic heartbeat messages to its neighbor nodes. We first discuss dealing with failover. When a node m fails, one of m's child nodes c takes over the role of the failed node. m's other child nodes become c's child nodes. c should be chosen as c's virtual anchor node t_c is the closest to m among the inner nodes on path from m to m's virtual anchor node t_m. This is because path $t_m \sim c$ must be long enough to contain all the other inner nodes that were originally put on path $t_m \sim m$. c becomes a new owner of the edges that were owned by m before, and c's new virtual anchor node is set to be t_m. c must adjust its distance label to reflect this change. c's takeover event is propagated down to the nodes in c's new subtree, so that they can update their distance labels. This activity updates $O(1)$ nodes in the best case when a leaf node fails and $O(N)$ nodes in the worst case when the root node fails. On the $O(\log N)$-depth anchor tree in the backbone bottleneck model, however, the update time is $O(\log N)$ hops in the worst case because the event can be propagated to child nodes in parallel. For example, in Figure 1, if e fails, k takes over its role because t_k is closer to e than any other inner nodes on path $e \sim t_e$. i and j become k's child nodes.

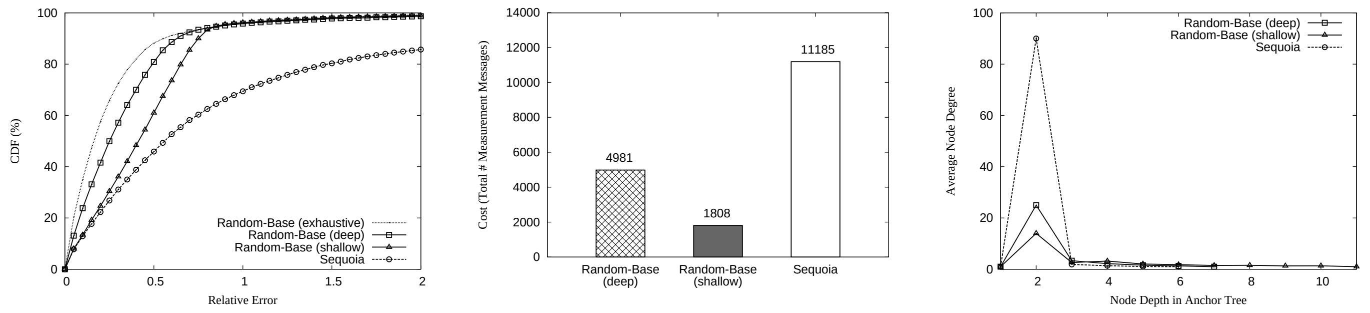
In a real network environment, bandwidths can change dynamically over time. Reconstructing the entire prediction tree to adapt to dynamic changes would have a high cost. We restructure only the part of the system where bandwidth changes occur, by periodic adjustment of the trees. Each node x periodically checks if $d(x,y) = d_T(x,y)$ for some y in the system. When x detects $d(x,y) \neq d_T(x,y)$, it leaves the system, then joins again using y as its bootstrap node. x's rejoin process is normally faster than an initial join process, because x can utilize bandwidth measurements it has already made.

5 Tolerating Imperfect Data

The previous sections considered an ideal world that can be represented as a tree metric space. The real world Internet does not perfectly satisfy 4PC as shown with non-zero 4PC-ϵs for a sample dataset [15]. Even though the Internet closely approximates a tree metric space, directly applying the previously described node join algorithm to the real nodes on the Internet will result in low accuracy for bandwidth prediction. This section describes useful heuristics to improve prediction accuracy.

Error Minimization for End Node Choice: The join algorithm chooses an end node y that maximizes the Gromov product $(x|y)_z$. In an ideal world, this guarantees perfect prediction accuracy, as shown by Theorem 4.1. In the real Internet, however, there exists uncertainty because some tree metric space assumptions do not hold everywhere, so finding an end node that maximizes the Gromov product does not always result in the best accuracy. Also, following the direction of the local maximizer does not always guarantee reaching the global maximizer. We modify the algorithm to choose an end node that minimizes the prediction error, rather than finding the Gromov product maximizer. Each time x visits a node m before finding y, x computes its temporary distance labels for m and all of m's neighbor nodes, that is, all candidate nodes for an end node. Based on each computed distance label for each candidate, x estimates relative errors for bandwidth prediction, using the formula $\frac{|BW(x,t)-BW_T(x,t)|}{BW(x,t)}$, for all nodes t that x has measured so far. Suppose n_t is the total number of nodes that x has measured since x started measurement at z. x now has a set of n_t relative error values for each candidate node. Then x moves to the candidate node that minimizes the average of the n_t prediction errors. When x reaches a local error minimizer, the local minimizer is chosen as the end node. Even though this heuristic does not guarantee minimizing the global error, we have found that it shows better accuracy on a real-world bandwidth dataset than moving to the local Gromov product maximizer. Since x uses the bandwidth data it has already collected, this heuristic does not cause any additional measurements to be made. The heuristic also does not affect accuracy in the ideal world scenario, because using the Gromov product maximizer in that scenario is exactly the same as using the error minimizer.

Rational Transform Function: A linear transform function $BW_T(u,v) = C_{large} - d_T(u,v)$ is used for bandwidth prediction in the base system. Unlike the ideal world scenario, $d_T(u,v)$ might not be equal to



(a) Accuracy (b) Cost (c) Average Node Degree at Each Depth

Figure 3: Overall Result (Our random-base approach shows higher accuracy and lower cost than Sequoia.)

$d(u, v)$ in the real world scenario. If $d_T(u, v)$ is much larger than $d(u, v)$, that can result in predicting a negative bandwidth value and will decrease overall prediction accuracy. We introduce a *rational transform function* to overcome this problem. By using $d(u, v) = \frac{C_{large}}{BW(u,v)}$ and $BW_T(u, v) = \frac{C_{large}}{d_T(u,v)}$ to transform between bandwidth and distance, instead of the linear transform functions, the predicted bandwidth will always be positive even when d_T is overestimated in a real world scenario. As does the linear function, the rational transform function also inverts ordering of bandwidth after performing the transformation. Thus, the rational transform function also works correctly for representing network bandwidth as a metric space where closer distance is considered better. An additional benefit of this change is that it does not affect accuracy in the ideal world scenario and adds no extra cost into the system, similar to the error minimization heuristic.

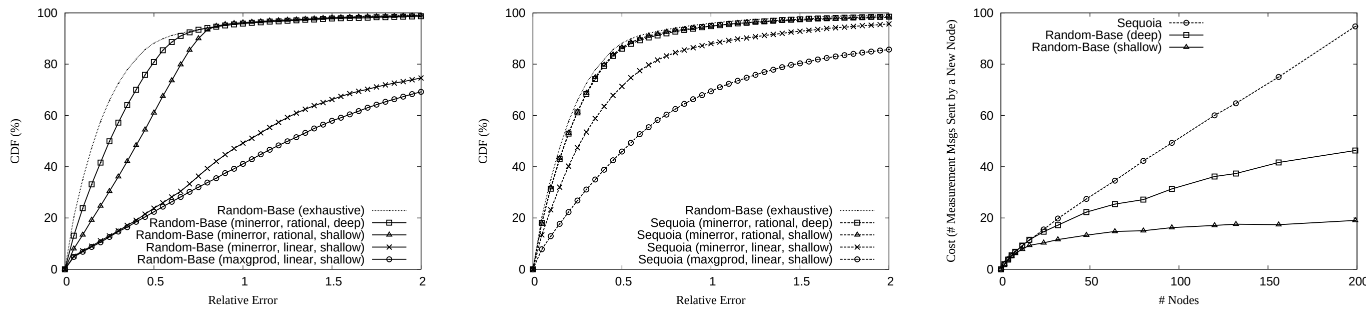
Deep Search for End Node Choice: The previously discussed node join algorithm considers only direct neighbors as candidate nodes to be an end node. By using the deep search heuristic, we can take advantage of additional candidates at each hop when searching for an error minimizer. Such candidates can be found by using indirect neighbors, including parents' parents, parents' children, and children's children. The larger pool of candidates should result in higher prediction accuracy when using error minimization, but will have little effect when using the global Gromov product maximizer, because the maximizer performs poorly in real world scenarios. Unlike the other two heuristics, however, deep search does add overhead because more candidates will be involved in the process of adding a new node to the system.

6 Evaluation

This section evaluates our approach by examining i) overall accuracy and cost, ii) heuristic efficacy, iii) scalability of measurement traffic, and iv) load balance with respect to measurement traffic. Our simulations are based on the HP-PlanetLab dataset described in [14, 15]. This dataset contains PlanetLab bandwidth measurements evaluated at HP using pathChirp [16]. Since the raw dataset is incomplete and has many unmeasured pairs of nodes, we extracted measurements for the 200 nodes (out of 459) that give a full *n-to-n* set of measurements. We simulated the algorithms in Java, using the PeerSim [8] P2P simulator as a starting point.

Overall performance is shown in Figure 3. All the experiments are performed by serially adding 200 nodes in the same order. We show results for four different approaches. **Random-Base (exhaustive)** uses a perfect oracle to exhaustively search for a base node and end node pair that maximizes prediction accuracy for each joining node. While this approach is only theoretical, not practical, it provides an upper bound on the potential results for the real algorithms. **Random-Base (deep)** shows results from our system using all three heuristics: error minimization, rational transformation, and deep searching. **Random-Base (shallow)** shows results from our system with only two heuristics: error minimization and rational transformation. **Sequoia** refers to our simulation of the centralized Sequoia algorithm.

We evaluate the prediction accuracy by computing a relative bandwidth prediction error for each pair of nodes. As described in Section 5, the relative bandwidth prediction error between node u and v is defined as $\frac{|BW(u,v)-BW_T(u,v)|}{BW(u,v)}$. Figure 3(a) shows the CDF of the relative errors for all possible pairs among the 200 nodes.



(a) Heuristics to Random-Base Algorithm (b) Heuristics to Fixed-Base Algorithm (c) Node Join Cost

Figure 4: The Effect of Heuristics on Accuracy and Scalable Node Join

Random-Base (deep) is highly accurate, with bandwidth prediction for more than 80% of the node pairs having a relative error less than 0.5, and closely tracks Random-Base (exhaustive). Both Random-Base (deep) and Random-Base (shallow) show higher accuracy than Sequoia because of the heuristics discussed in Section 5. Figure 3(b) shows the cost of each system. We use the total number of measurements caused by all 200 nodes joining the system to measure the cost. As expected, there is an accuracy versus overhead trade-off between Random-Base (deep) and Random-Base (shallow). The deep search heuristic decreases prediction errors, but also increases measurement traffic.

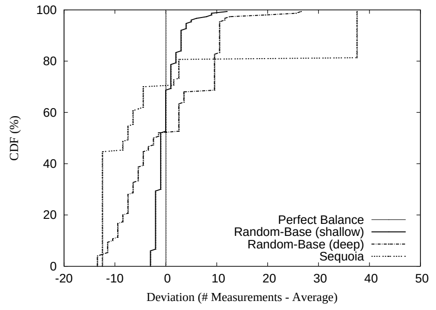
Regardless of whether deep search is used, our system requires many fewer measurements than Sequoia, and achieves higher accuracy. This result is surprising, as both systems move vertically on an anchor tree, and we therefore expected the cost for our system to be similar to that of Sequoia.

Sequoia's high costs come from an imbalance in the out-degree of tree nodes. As shown in Figure 3(c), high-level nodes in the Sequoia anchor tree have many more child nodes than in our system. This imbalance occurs because the node join algorithm is more likely to stop near the starting point when finding an end node than to stop at a distant node. And unlike the random base node in our system, the search for each new node in Sequoia must start at the root node, resulting in high degrees for Sequoia's high-level nodes. When a new node is added to Sequoia, it always passes through high-degree high-level nodes in a top-down way, and that causes heavy traffic for those nodes. On the other hand, our algorithm has much less imbalance because the base node is not fixed. As a result, the system exhibits lower measurement traffic than Sequoia.

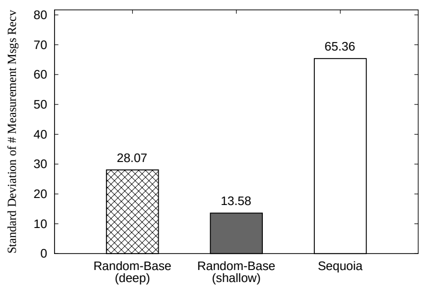
Figure 3(c) shows that, even with our approaches, high-level nodes have higher average degree than low-level nodes. This results from the hierarchical nature of the anchor tree. When the join algorithm moves around an anchor tree, a high-level node is more likely to be contacted as a *bridge* connecting low-level nodes, implying that a new node is more likely to stop at a high-level node. This phenomenon explains why Random-Base (shallow) nodes are slightly more balanced in degree than in Random-Base (deep). High-level nodes are contacted more times with than without the deep search heuristic. Accordingly, high-level nodes in Random-Base (deep) are more likely to have high degree than in Random-Base (shallow).

The Effect of Heuristics: Figure 4(a) shows how effectively the three heuristics discussed in Section 5 raise the accuracy of our system. **Random-Base (maxgprod, linear, shallow)** represents results for our system without any heuristics. **Random-Base (minerror, rational, deep)** represents results for our system with three heuristics: error minimizing, rational transforming, and deep searching, and the other curves show how accuracy improves as the heuristics are applied one by one. These heuristics can also be applied to Sequoia, and the result is shown in Figure 4(b). For both our system and Sequoia, use of the heuristics allows accuracy to approach that of the exhaustive test case. Note that neither error minimization nor rational transformation adds any additional overhead. Only deep searching adds overhead, as shown in Figure 3(b).

Scalable Node Join: Figure 4(c) presents the scalability in terms of the measurement traffic caused by each joining node. For each experiment, we constructed an anchor tree 200 times with a different order of node joining. Then we collected the average number of measurements sent by a joining node for different number of nodes in the system. As shown in Figure 4(c), the cost for Sequoia and Random-Base (deep)



(a) CDF of (#Measurements - Average)



(b) Standard Deviation of #Measurements

Figure 5: Load Balance in terms of Measurement Traffic

increases sub-linearly while the cost for Random-Base (shallow) increases almost logarithmically. The results can be explained by a combination of out-degree imbalance for Sequoia, and the extra overhead incurred by the deep search heuristic for our system. This distinction points to a trade-off between Random-Base (deep) and Random-Base (shallow) in terms of scalability versus accuracy.

Load Balance: We last present how well measurement traffic is balanced among nodes in our approach. In Figure 5, the experiment starts by constructing an initial system with 150 nodes, adds one node, and collects the number of messages received by each of the original 150 nodes. Based on the same initial structure, we run the experiment 50 times by adding a different node each time, and obtain the total number of messages received by each of the 150 nodes. Figure 5(a) shows the CDF of the total number of measurements for each node, subtracting the average across the 150 nodes. The case where all 150 nodes have the same value is considered as the standard for perfect balance (Perfect Balance in the key). Figure 5(b) shows the standard deviation of the total number of measurements for each node in the initial system. The CDFs for both Random-Base (deep) and Random-Base (shallow) are closer to Perfect Balance than is Sequoia's, and the standard deviations are also much smaller. This is because a node joins the system starting at a random base node while all Sequoia joins start at the root. Despite its random base node, the CDF for our system still has a tail, as seen in Figure 5(a). Also, Random-Base (deep) has a longer tail than Random-Base (shallow). We explain those by the hierarchical property of the anchor tree in exactly the same way we explained the degree imbalance in the overall result with Figure 3(c).

7 Conclusions and Future Work

This paper has presented a decentralized and scalable system that accurately predicts pairwise bandwidths on real-world datasets with low, balanced overheads. We provide a proof of correctness of the key ideas, as well as supporting theorems. The system uses a decentralized node join algorithm, rather than the more centralized (single measurement starting point) method used by prior work. It also uses three novel heuristics to achieve accuracy on imprecise (real-world) datasets. Simulation results show high accuracy, scalability, and balanced workload.

We are currently extending this work in several ways. First, we are working on a new node search algorithm. Currently our system is only able to predict bandwidth between a pair of given nodes. We are investigating approaches to predicting the highest-bandwidth node in the system, from a given node. We are also attempting to acquire larger real-world datasets. Finally, we intend to use our system as an underlying technology for resource discovery in P2P desktop grids [10, 9, 11, 12].

Acknowledgments

We thank Venugopalan Ramasubramanian and the other developers of Sequoia for providing us with the HP-PlanetLab real-world dataset used to evaluate our algorithm in Section 6.

References

[1] Cisco data center infrastructure 2.5 design guide. `http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5_book.html`.

[2] I. Abraham, M. Balakrishnan, F. Kuhn, D. Malkhi, V. Ramasubramanian, and K. Talwar. Reconstructing approximate tree metrics. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 43–52, New York, NY, USA, 2007. ACM.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 63–74, New York, NY, USA, 2008. ACM.

[4] P. Buneman. A note on the metric properties of trees. *Journal of Combinatorial Theory, Ser. B*, 17:48–50, 1974.

[5] M. Costa, M. Castro, A. Rowstron, and P. Key. Pic: Practical internet coordinates for distance estimation. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 178–187, Washington, DC, USA, 2004. IEEE Computer Society.

[6] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[8] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[9] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using content-addressable networks for load balancing in desktop grids. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC-16)*. IEEE Computer Society Press, June 2007.

[10] J.-S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Resource discovery techniques in distributed desktop grid environments. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing - GRID 2006*. IEEE Computer Society Press, Sept. 2006.

[11] J.-S. Kim, B. Nam, M. Marsh, P. Keleher, B. Bhattacharjee, and A. Sussman. Integrating categorical resource types into a P2P desktop grid system. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing - GRID 2008*. IEEE Computer Society Press, Sept. 2008.

[12] J. Lee, P. Keleher, and A. Sussman. Decentralized resource management for multi-core desktop grids. In *Proceedings of the 24th International Parallel & Distributed Processing Symposium*. IEEE Computer Society Press, Apr. 2010.

[13] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings of the IEEE INFOCOM*, pages 170–179, 2001.

[14] V. Ramasubramanian, D. Malkhi, F. Kuhn, I. Abraham, M. Balakrishnan, A. Gupta, and A. Akella. A unified network coordinate system for bandwidth and latency. Technical Report MSR-TR-2008-124, Microsoft Research, 2008.

[15] V. Ramasubramanian, D. Malkhi, F. Kuhn, M. Balakrishnan, A. Gupta, and A. Akella. On the treeness of internet latency and bandwidth. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 61–72, New York, NY, USA, 2009. ACM.

[16] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell. pathchirp: Efficient available bandwidth estimation for network paths. In *In Passive and Active Measurement Workshop*, 2003.

[17] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, Oct. 2001.

Appendix

Proof of Theorem 4.1

We can prove this by showing $d(x,p) = d_T(x,p)\ \forall p \in V$. For a leaf node $p \in T$, let $t_p \in T$ be an inner node located on the path $z \sim y$ where $d_T(z,t_p) = (y|p)_z$. Two cases can be considered by the location of t_p and p as shown in Figure 6.

Case 1: for p such that $d_T(z,t_x) \geq d_T(z,t_p)$

$d(z,x) + d(z,y) - d(x,y) \geq d(z,x) + d(z,p) - d(x,p) \quad \because y$ maximizes $(x|y)_z \Rightarrow (x|y)_z \geq (x|p)_z$

$d(z,x) + d(z,y) - d(x,y) \geq d(z,y) + d(z,p) - d(y,p) \quad \because d_T(z,t_x) \geq d_T(z,t_p) \Rightarrow (x|y)_z \geq (y|p)_z$

$d(x,y) + d(z,p) \leq d(z,y) + d(x,p) = d(z,x) + d(y,p) \quad \because (V,d)$ satisfies $4PC$ for x,y,z,p. **(4PC-xyzp)**

$$\begin{aligned}
d(x,p) &= d(z,x) + d(y,p) - d(z,y) \quad \because \text{(4PC-xyzp)} \\
&= d_T(z,x) + d_T(y,p) - d_T(z,y) \quad \because d(z,x) = d_T(z,x), \text{ and the old } T \text{ induces the old } (V,d). \\
&= (d_T(z,t_x) + d_T(t_x,x)) + (d_T(y,t_x) + d_T(t_x,p)) - (d_T(z,t_x) + d_T(t_x,y)) \\
&= d_T(t_x,x) + d_T(t_x,p) \quad = \quad d_T(x,p)
\end{aligned}$$

Case 2: for p such that $d_T(z,t_x) < d_T(z,t_p)$

$d(z,x) + d(z,y) - d(x,y) \geq d(z,x) + d(z,p) - d(x,p) \quad \because y$ maximizes $(x|y)_z \Rightarrow (x|y)_z \geq (x|p)_z$

$d(z,x) + d(z,y) - d(x,y) < d(z,y) + d(z,p) - d(y,p) \quad \because d_T(z,t_x) < d_T(z,t_p) \Rightarrow (x|y)_z < (y|p)_z$

$d(z,x) + d(y,p) \leq d(z,y) + d(x,p) = d(x,y) + d(z,p) \quad \because (V,d)$ satisfies $4PC$ for z,x,y,p. **(4PC-zxyp)**

$$\begin{aligned}
d(x,p) &= d(x,y) + d(z,p) - d(z,y) \quad \because \text{(4PC-zxyp)} \\
&= d_T(x,y) + d_T(z,p) - d_T(z,y) \quad \because d(x,y) = d_T(x,y), \text{ and the old } T \text{ induces the old } (V,d) \\
&= (d_T(x,t_x) + d_T(t_x,y)) + (d_T(z,t_x) + d_T(t_x,p)) - (d_T(z,t_x) + d_T(t_x,y)) \\
&= d_T(x,t_x) + d_T(t_x,p) \quad = \quad d_T(x,p)
\end{aligned}$$

Thus, T induces (V,d).

█

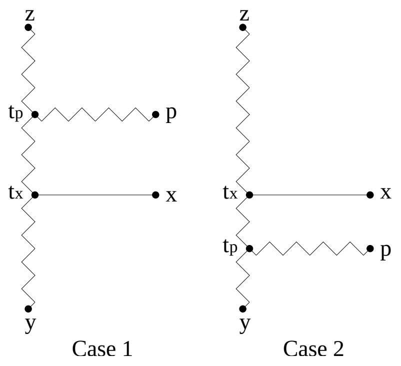


Figure 6: Two Cases of Prediction Tree Used in the Proof of Theorem 4.1.

Proof of Theorem 4.2

You can refer to Figure 2(a) for better understanding.

$d(z,y)+d(w,u) < d(z,w)+d(y,u) = d(z,u)+d(y,w)$ $\because$ T's structure **(T-struct-zywu)**

$d(z,y)+d(z,x)-d(y,x) > d(z,w)+d(z,x)-d(w,x)$ $\because (y|x)_z > (w|x)_z$

$d(y,x)+d(z,w) < d(z,y)+d(w,x) = d(y,w)+d(z,x)$ $\because (V,d)$ satisfies 4PC for z,w,y,x **(4PC-zwyx)**

$d(z,v)+d(y,w) < d(z,y)+d(w,v) = d(z,w)+d(y,v)$ $\because$ T's structure **(T-struct-zywv)**

Part 1: Prove $(y|x)_z > (u|x)_z$

$$\begin{aligned}
d(y,x)+d(z,u) &= d(y,x)+d(z,w)+d(z,u)-d(z,w) \\
&< d(y,w)+d(z,x)+d(z,u)-d(z,w) \quad \text{by (4PC-zwyx)} \\
&= d(y,u)+d(z,x)+d(z,u)+d(y,w)-(d(z,w)+d(y,u)) \\
&= d(y,u)+d(z,x) \quad \text{by (T-struct-zywu)}
\end{aligned}$$

$d(y,x)+d(z,u) < d(y,u)+d(z,x) = d(z,y)+d(u,x)$ $\because (V,d)$ satisfies 4PC for z,u,y,x

$d(z,y)+d(u,x) > d(z,u)+d(y,x) \Rightarrow d(z,y)+d(z,x)-d(y,x) > d(z,u)+d(z,x)-d(u,x)$

Thus, $(y|x)_z > (u|x)_z$

Part 2: Prove $(y|x)_z > (v|x)_z$

$$\begin{aligned}
d(y,x)+d(z,v) &= d(y,x)+d(z,w)+d(z,v)-d(z,w) \\
&< d(y,w)+d(z,x)+d(z,v)-d(z,w) \quad \text{by (4PC-zwyx)} \\
&= d(y,v)+d(z,x)+d(z,v)+d(y,w)-(d(z,w)+d(y,v)) \\
&< d(y,v)+d(z,x) \quad \text{by (T-struct-zywv)}
\end{aligned}$$

$d(y,x)+d(z,v) < d(y,v)+d(z,x) = d(z,y)+d(v,x)$ $\because (V,d)$ satisfies 4PC for z,v,y,x

$d(z,y)+d(v,x) > d(z,v)+d(y,x) \Rightarrow d(z,y)+d(z,x)-d(y,x) > d(z,v)+d(z,x)-d(v,x)$

Thus, $(y|x)_z > (v|x)_z$

∎

Proof of Theorem 4.3

You can refer to Figure 2(a) for better understanding.

$d(z,y)+d(w,u)<d(z,w)+d(y,u)=d(z,u)+d(y,w) \quad \because T\text{'s structure} \quad$ **(T-struct-zywu)**

$d(z,y)+d(z,x)-d(y,x)=d(z,w)+d(z,x)-d(w,x) \quad \because (y|x)_z=(w|x)_z$

$d(y,w)+d(z,x)\leq d(y,x)+d(z,w)=d(z,y)+d(w,x) \quad \because (V,d)$ satisfies 4PC for y,w,z,x **(4PC-ywzx)**

Case 1: $d(y,w)+d(z,x)=d(y,x)+d(z,w)=d(z,y)+d(w,x)$

$$\begin{aligned} d(z,x)+d(w,u) &= d(y,w)+d(z,x)+d(w,u)-d(y,w) \\ &= d(z,y)+d(w,x)+d(w,u)-d(y,w) \quad \text{by (Case 1)} \\ &= d(z,u)+d(w,x)+d(z,y)+d(w,u)-(d(z,u)+d(y,w)) \\ &< d(z,u)+d(w,x) \quad \text{by (T-struct-zywu)} \end{aligned}$$

$d(z,x)+d(w,u)<d(z,u)+d(w,x)=d(z,w)+d(u,x) \quad \because (V,d)$ satisfies 4PC for w,u,z,x **(4PC-wuzx)**

$$\begin{aligned} d(y,x)+d(z,u) &= d(y,x)+d(z,w)+d(z,u)-d(z,w) \\ &= d(z,y)+d(w,x)+d(z,u)-d(z,w) \quad \text{by (Case 1)} \\ &= d(z,u)+d(w,x)+d(z,y)-d(z,w) \\ &= d(z,w)+d(u,x)+d(z,y)-d(z,w) \quad \text{by (4PC-wuzx)} \\ &= d(z,y)+d(u,x) \end{aligned}$$

$d(z,y)+d(z,x)-d(y,x)=d(z,u)+d(z,x)-d(u,x)$
Thus, $(y|x)_z=(u|x)_z$

Case 2: $d(y,w)+d(z,x)<d(y,x)+d(z,w)=d(z,y)+d(w,x)$

$$\begin{aligned} d(y,x)+d(z,u) &= d(y,x)+d(z,w)+d(z,u)-d(z,w) \\ &> d(y,w)+d(z,x)+d(z,u)-d(z,w) \quad \text{by (Case 2)} \\ &= d(y,u)+d(z,x)+d(z,u)+d(y,w)-(d(z,w)+d(y,u)) \\ &= d(y,u)+d(z,x) \quad \text{by (T-struct-zywu)} \end{aligned}$$

$d(y,u)+d(z,x)<d(y,x)+d(z,u)=d(z,y)+d(u,x) \quad \because (V,d)$ satisfies 4PC for y,u,z,x

$d(z,y)+d(z,x)-d(y,x)=d(z,u)+d(z,x)-d(u,x)$
Thus, $(y|x)_z=(u|x)_z$

∎

Distance Computation with Distance Label

Algorithm 3 shows how to compute the distance between two nodes using their distance labels. The point is that we can compute a distance without having the whole information of prediction tree. Figure 7 shows how Algorithm 3 computes $d_T(p,q)$. Line 6 of Algorithm 3 computes the sum of thick solid lines in Figure 7. Line 11 computes the sum of thin solid lines and subtract the sum of thin dashed lines from it.

Algorithm 3: **GetDistance(p, q)**: It computes $d_T(p,q)$ with distance labels.

```
1  $z \leftarrow$ the lowest-level common anchor node in $p$.label and $q$.label
2  $x \leftarrow$ the node whose anchor node is $z$ in $p$.label      // $p$.label = (root node $r \to ... \to z \to x \to ... \to p$)
3  $y \leftarrow$ the node whose anchor node is $z$ in $q$.label      // $q$.label = (root node $r \to ... \to z \to y \to ... \to q$)
4  $t_x \leftarrow x$'s virtual anchor node
5  $t_y \leftarrow y$'s virtual anchor node
6  distance $\leftarrow |\ d_T(z,t_x) - d_T(z,t_y)\ | + (d_T(t_x,x) + d_T(t_y,y))$
7  foreach node $a$ in sub-labels $(x \to ... \to p)$ and $(y \to ... \to q)$ do
8      if $(a \neq x) \wedge (a \neq y)$ then
9          $a_2 \leftarrow a$'s anchor node
10         $t_a \leftarrow a$'s virtual anchor node
11         distance $\leftarrow$ distance $- d_T(a_2,t_a) + d_T(t_a,a)$
12 return distance
```

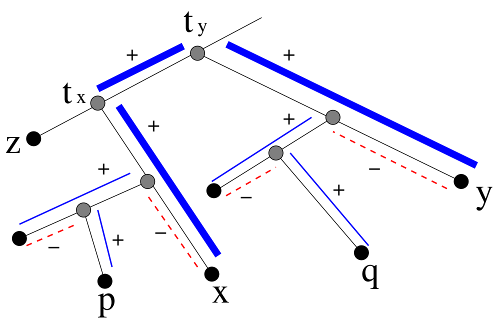


Figure 7: Distance Computation of $d_T(p,q)$ without a Complete Prediction Tree in Algorithm 3

Decentralized Node Join Algorithm

Algorithm 4: **x.Join(z)**: It is a decentralized node join algorithm that makes x join the system through a random bootstrap node z and assigns x's distance label.

```
1  if $z = x$ then  $x$.label ← $x$    // $x$ becomes the root node in an anchor tree.
2  else
3      $x$ measures $d(x, z)$;
4      $x$.JoinSub ($z, z$)    // find an end node with base node $z$.

   // $x$.JoinSub($z, y$):  $x$ contacts $y$ with base node $z$ to find an end node.
5  $x$.SetLabel ($z, y$)
6  Scand ← $\{y, y$'s parent , $y$'s children $\}$
7  $x$ measures $d(x, s)$ $\forall s \in$ Scand
8  Gmax ← $\text{argmax}_{s \in \text{Scand}} (x|s)_z$
9  if $y \notin$ Gmax then
10     $y$ ← one node in Gmax
11     $x$.JoinSub ($z, y$)    // move to new $y$

   // $x$.SetLabel($z, y$):  $x$ sets its distance label using base node $z$ and end node $y$.
12 $d_T(t_x, x) \leftarrow (y|z)_x$    // $t_x$ is $x$'s virtual anchor node.
13 $a$ ← the edge-owner node of the edge that $t_x$ is located on
14 if $d_T(a, t_y) < d_T(a, t_z)$ then  $d_T(a, t_x) \leftarrow d_T(t_a, a) - d_T(t_a, t_z) - \{(x|y)_z - d_T(t_z, z)\}$
15 else  $d_T(a, t_x) \leftarrow d_T(t_a, a) - d_T(t_a, t_z) + \{(x|y)_z - d_T(t_z, z)\}$
16 $x$.label ← ($a$.label $\xrightarrow[d_T(t_x, x)]{d_T(a, t_x)}$ $x$)
17 $x$ becomes $a$'s child node.
```

Algorithm 4 is a decentralized node join algorithm. The algorithm extends Algorithm 2 by introducing distance labels and constructing a prediction tree in a distributed way. A new node x joins the system by contacting any base node z (line 4). Then x moves up and down an anchor tree overlay network (line 11). When x contacts y, x updates its distance label by setting y as an end node with base node z (line 5). And x becomes y's child node in the anchor tree (line 17). This is repeated until x reaches the correct end node that maximizes the Gromov product.