Online Metric Matching on the Line

Senior Research Thesis

Kevin Lewi
April 2011

Abstract

Given a metric space, a set of points with distances satisfying the triangle inequality, a sequence of requests arrive in an online manner. Each request must be irrevocably assigned to a unique server before future requests are seen. The goal is to minimize the sum of the distances between the requests and the servers to which they are matched. We study this problem under the framework of competitive analysis.

We give two $O(\log k)$-competitive randomized algorithms, where k is the number of servers. These improve on the best previously known $O(\log^2 k)$-competitive algorithm for this problem. Our technique is to embed the line into a distribution of trees in a distance-preserving fashion, and give algorithms that solve the problem on these trees. Our results are focused on settings for the line, but these results can also be extended to all constant-dimensional metric spaces.

Contents

Chapter 1

Introduction

In the online metric matching problem, requests appear one at a time on the metric space. As a new request arrives, it must be assigned to a unique server. After a request has been assigned, it cannot be modified. The goal is to find a matching for each request such that the total sum of distances between request-server pairs is minimized. In the offline version of this problem, all requests arrive "simultaneously", so the problem is equivalent to the minimum weighted bipartite matching problem (but on a metric), and can be solved using traditional network flow algorithms.

The heuristic we use to measure the quality of an online algorithm is its competitive ratio. Given a fixed setting of servers on a metric space, the competitive ratio of an online algorithm is the ratio between the online algorithm's matching's cost and the offline optimal cost. We are interested in bounding the worst-case competitive ratio across all settings of servers on metric spaces.

The online metric matching problem was first introduced by Kalyanasundaram and Pruhs in [KP93], and also independently by Khuller, Mitchell, and Vazirani in [KMV94]. The problem is known as the online weighted bipartite matching problem in [KMV94] and simply online weighted matching in [KP93]. Since then, there has been a considerable amount of work on attempting to bridge the gap between the upper and lower bounds for this problem. A $2k-1$ competitive deterministic algorithm is presented in both [KP93] and [KMV94], establishing the initial upper bound for general metrics. Both sources also note a lower bound of $2k-1$ through the star graph, where a simple example can be constructed such that no algorithm can perform with competitive ratio less than $2k-1$.

Although these results close the discussion for deterministic algorithms on general metrics, study has been performed on the effects of allowing randomization. For the star graph (or uniform metric), one can show that any randomized algorithm must be $\Omega(\log k)$ competitive, and that the simple randomized greedy algorithm of choosing the closest server to assign to, breaking ties at random, is $O(\log k)$ competitive. In 2006,

Meyerson, Nanavati, and Poplawski presented a randomized $O(\log^3 k)$ competitive algorithm on general metrics in [MNP06], representing the first randomized algorithm to achieve a sublinear competitive ratio. In this paper, we will refer to this algorithm as the MNP algorithm. This algorithm applies the main results of [FRT03] by first converting the general metric space into a tree metric with certain properties on the lengths of the edges, running a simple algorithm on the tree, and then mapping the resulting assignments back to the original metric. The simple algorithm for the tree metric is $O(\log k)$ competitive, and an $O(\log^2 k)$ term is incurred in the conversion back to the original metric, resulting in $O(\log^3 k)$ competitiveness. In just a year later, Bansal, Buchbinder, Gupta, and Naor improve upon the MNP algorithm in [BBGN07] through a similar approach with an $O(\log k)$ competitive algorithm on a special tree metric, but this time only incurring another $O(\log k)$ term in the conversion to the original metric, thus resulting in a $O(\log^2 k)$ competitive algorithm. Since the star graph example mentioned earlier gives a lower bound of $\Omega(\log k)$ for the competitive ratio, the natural question to ask is whether there exists an $O(\log k)$ competitive algorithm for general metrics.

Several attempts have been made at reducing the upper bound for the randomized version of the online metric matching problem on special cases of metric spaces such as the line, or 1-dimensional Euclidean space. In [KP98], a deterministic lower bound of 9 was shown for the line in a reduction to a problem known as the cow-path problem. The cow-path problem is an example of a search game in an unknown environment, where a cow begins at the origin of the real number line and must find a bridge located somewhere on the line. The fact that the lower bound on the competitive ratio for the deterministic cow-path problem is 9 implies a lower-bound on the competitive ratio for deterministic algorithms of online metric matching on the line. It was conjectured in [KP98] that there exists a 9-competitive online algorithm for this problem, which was later disproved in [FHK05]. Also conjectured in [KP98] was that the Work Function Algorithm presented for the k-server problem by Koutsoupias and Papadimitriou in [KP95] obtains a constant competitive ratio on the line for the online metric matching problem. This conjecture was also disproven by Koutsoupias and Nanavati in [KN03], who acknowledge that the case of the line is the simplest non-trivial instance of minimum online metric matching.

1.1 Motivation

Instances of the online metric matching problem occur in numerous real-world situations apart from the evident application of the problem in network design involving a series of requests and fixed servers. For example, imagine a setting of k fire stations in the city, each of which can handle a single fire. As each fire arrives, a fire station must be assigned to extinguish it. Also, variations of online metric matching can be

applied to the algorithms behind electronic markets which must deal with effectively matching buyers to sellers, each of which name their prices.

Online metric matching on the line also garners theoretical interest due to the large disparity between the best known bounds for its competitive ratio. The tightest bounds known for online metric matching on the line are $9 + \epsilon$ for the deterministic lower bound and $O(k)$ for the deterministic upper bound, and $O(\log^2 k)$ for the randomized upper bound. Note that both types of upper bounds are derived from results on general metrics; no competitive algorithms have been developed specifically for the line. For general metrics, the $O(\log^2 k)$ competitive randomized algorithm in [BBGN07] is the tightest known upper bound. The relatively large gap between the upper and lower bounds provides a motivation for the study of this problem.

1.2 Problem Statement

An instance of the online metric matching problem (V, d, R, S, k) is defined by a metric space (V, d), with the sequence of requests $R = r_1, \cdots, r_k$ and each $r_i \in V$. The set of servers $S = \{s_1, \cdots, s_m\}$ is such that $m \geq k$ and each $s_i \in V$. The integer k represents the number of requests to arrive.

A solution to an instance of this problem is a permutation $\pi = \pi_1, \pi_2, \cdots, \pi_k$ of a subset of the set of servers S, also called a matching. The cost of such a solution is defined as $\sum_{i=1}^{k} d(r_i, \pi_i)$, where d is the metric used in the metric space (V, d).

The added restriction of the online version of this problem (as opposed to the offline version) is that the sequence R of requests is revealed one element at a time, rather than all at once. Thus, an online algorithm is forced to make assignments one-by-one, as each request arrives. If the online algorithm has made $i < k$ assignments so far, then only the first $i + 1$ requests $r_1, \cdots, r_{i+1}$ are revealed. When the algorithm has made $i = k$ assignments, then all requests have been assigned to unique servers, and there is no more work left to do.

We call a deterministic online algorithm c-competitive on a specific instance of this problem if the algorithm achieves a solution whose cost is c times the offline optimal solution. For randomized algorithms, c-competitiveness on an instance means that the solution achieves an expected cost that is c times the offline optimal solution. The competitive ratio of an algorithm is defined as the maximum over all instances of the ratio between the algorithm's (expected) cost and the offline optimal's cost for each instance. An algorithm is called c-competitive if its competitive ratio is c.

1.3 Our Results

We present several algorithms that perform competitively for the online metric matching problem. The techniques described here extend upon the results of [FRT03] to produce algorithms that perform $O(\log k)$-competitively on the line metric.

The first result is a combinatorial proof that the simple, deterministic greedy algorithm on a hierarchically well-separated tree (HST) is hd competitive, where h and d are the height and degree of a tree constructed from the line. Since we also give HST constructions for the line that allow $d = O(1)$ and $h = O(\log k)$ while changing expected distances by an $O(\log k)$ factor, so this gives rise to another $O(\log^2 k)$-competitive algorithm for the line.

Next, we show that a modification of the randomized algorithm given in [MNP06] is in fact $O(\log k \log^2 d)$-competitive on the line, where $d = 2$ for the line. The proof is based on defining a suitable potential function, and is also inspired by the proof in [MNP06]. This result also extends to metric spaces with low doubling dimension, since they also admit embeddings into HSTs where d remains a constant.

Finally, we give a new algorithm (which we call HST-greedy), and show that it is $O(\log k)$-competitive on the line. The approach used for this algorithm is relatively different from the previous proofs: the algorithm itself uses both the HST structure and the fact that the HST is constructed from the line. Moreover, the proof directly considers the cost incurred on the line, and compares that to an optimal solution's cost on the HST. We also show that our analysis is tight.

Chapter 2

Preliminaries

This section will fix various facts that will prove useful in the main results for the later chapters. First, we will fix some definitions and the notation that will be used throughout the paper. One of these definitions is of the hierarchically well-separated tree (HST), first considered in [FRT03]. Next, we will show how to construct an HST from any line while ensuring that the HST does not stretch the distances between any two points by more than an $O(\log k)$ factor from their original distances on the line. Although the results in [FRT03] already imply the existence of such a construction for any metric space, the actual algorithm for the consruction itself is non-trivial and so it is included here for completeness.

We also include the proofs of validity for two assumptions that we make on the underlying metric space of an instance of the problem. The first assumption that we would like to make is that all requests arrive at the location of servers. Thus, no request ever appears at a point in the metric that is not occupied by a server. We can qualify this assumption by incurring a multiplicative factor of 2 (plus an additive of 1) to the competitive ratio. In other words, for every instance I of the online metric matching problem, we can create an instance I' such that I' has colocated requests and servers, and any algorithm that is c-competitive on I' is $2c + 1$ competitive on I. The second assumption we make is that the ratio of the maximum distance between any two points and the minimum distance between any two points is polynomial in the number of servers. This assumption incurs a multiplicative factor of 6.

2.1 Definitions and Notation

In this paper, we will denote $c(M)$ to be the cost of a matching M. Sometimes, for an algorithm A, we will use $c(A)$ to denote the cost of the matching produced by A as shorthand. When the context is clear, we will occasionally use the matching M to actually represent the cost of the matching, $c(M)$. Furthermore, Opt will always

be used to denote the optimal offline matching for an instance of the problem. Also, the results that involve $\log k$ terms are derived from the harmonic series denoted by $H_k = \sum_{i=1}^{k} 1/i$, using the fact that $H_k \sim \log_2 k$. So, all $\log k$ terms are base 2 unless otherwise noted.

Given a tree T rooted at r, define the depth of the root as 0, and the depth of each other node as one more than the depth of its parent. An edge has depth i if its endpoint closer to the root has depth i. For $\alpha > 1$, a tree is called an α-HST if the length of each edge at depth i is c/α^i for some constant c. Moreover, we will assume that all the leaves have the same depth; it is easy to ensure this while changing interpoint distances by at most a constant factor (depending on α). Furthermore, the algorithms we present later will automatically maintain this property.

We can define the height of the leaves in T to be zero, and the height of a node to be one greater than the height of its children—all leaves having the same depth means the height of every node is well-defined, and is just the depth of the leaves minus the depth of that node. The "arity" of the HST is the maximum number of children that any node has; binary trees have an arity of 2.

The level of vertices in T is defined as follows: the level of the root r is defined to be 1, and the level of any other node v is defined to be one more than the level of its parent p_v. Hence, $L(v) = L(p_v) + 1$.

We will also talk about degree-d α-HSTs when the degree is important. All of the algorithms discussed will involve taking the line L in the instance of the problem and constructing an α-HST T such that for all pairs of points x, y in the metric space,

1. $d_L(x, y) \leq d_T(x, y)$, and
2. $\mathbf{E}[d_T(x, y)] \leq O(\log k) d_L(x, y)$

Here, we use k to represent the number of servers in the instance of the problem. Next, we show how we can construct an appropriate HST given any instance of the problem on the line metric, where the HST we construct satisfies the above two properties.

2.2 Constructing an HST from the Line

We are interested in embedding the integer line in the interval $[1, k]$ into a binary HST T such that for any integers $1 \leq x \leq y \leq k$, $d_L(x, y) \leq d_T(x, y)$ and $\mathbf{E}[d_T(x, y)] \leq O(\log k) d_L(x, y)$. For simplicity, we will assume that k a power of 2. In all of the schemes that we are investigating, we are superimposing T on top of the line, where each leaf of T occupies some integer point on the line within the interval $[1, k]$. Since no point on the line corresponds to more than one leaf, the adjacent points i and $i + 1$ for $1 \leq i \leq k - 1$ are always separated by the structure of the tree. Consequently, the non-leaf nodes with two children of T (of which there are $2^k - 1$) lie in the divisions

between i and $i+1$ for $1 \leq i \leq k-1$. We will say that they lie in the interval $[i, i+1]$. Furthermore, we will call these points "cuts". For a cut x, define $l(x)$ to be the location of x. If x lies in $[i, i+1]$, then we say that $l(x) = i$. Also, define the depth $d(x)$ of a cut x to be the length of the path from the root to the node associated with x.

Note that the precise structure of a binary tree can be completely characterized by its cuts. Thus, a binary HST construction algorithm simply decides the location $l(x)$, and depth $d(x)$ of each of these cuts x, with the binary restriction that there cannot be more than 2^i cuts of depth i for $0 \leq i \leq h-1$.

Also, for the edge lengths of the α-HST, we will adopt the convention that edge lengths starting from the leaves of the tree upwards are: $c, c\alpha, c\alpha^2, \cdots$, where c is some constant. In fact, we will only be considering the case where $\alpha = 2$ and $c = 1$. We will occasionally refer to the property of an algorithm producing a tree T being such that $\mathbf{E}[d_T(x,y)] \leq O(\log k) d_L(x,y)$ as the algorithm having "$O(\log k)$ stretch".

As mentioned previously, any algorithm can be formulated in terms of the properties of the $k-1$ cuts. We give the algorithm in terms of how it arranges the cuts, and then a proof for why it achieves an $O(\log k)$ stretch for any two points in the interval.

First, we show the following lemma. Let $A_{i,j}$ denote the event that there exists a cut of depth i in the interval $[j, j+1]$ for some $1 \leq j < k$.

Lemma 2.2.1. *If the height h of T is $\log_2(k) + c_1$ for some small constant c_1, then for all $1 \leq x < y \leq k$, $\boldsymbol{E}[d_T(x,y)] \leq O(k) d_L(x,y)$. If, in addition, $\boldsymbol{Pr}[A_{i,j}] \leq 2^i/(k-1)$ for all $1 \leq j < k$ and $0 \leq i < h$, then for all $1 \leq x < y \leq k$, $\boldsymbol{E}[d_T(x,y)] \leq O(\log k) d_L(x,y)$.*

Proof. First, note that if $A_{i,j}$ occurs, then

$$d_T(j, j+1) = 2c(1 + \alpha + \cdots + \alpha^{h-1}) \leq 4c \cdot \alpha^{h-i}$$

by the structure of the HST. Thus, we get that

$$\mathbf{E}[d_T(j,j+1)] = \sum_{i=0}^{h} \mathbf{E}[d_T(j,j+1)|A_{i,j}] \cdot \mathbf{Pr}[A_{i,j}] \leq \sum_{i=0}^{\log_2(k)+c_1} 4c \cdot \alpha^{\log_2(k)+c_1-i} \cdot \mathbf{Pr}[A_{i,j}]$$

Now, we know that $\mathbf{Pr}[A_{i,j}] \leq 1$, so $4c \cdot 2^{\log_2(k)+c_1-i} = O(k)/2^i$, so

$$\mathbf{E}[d_t(j,j+1)] = \sum_{i=0}^{\log_2(k)+c_1} O(k)/2^i \leq O(k) \cdot \sum_{i=0}^{\infty} 1/2^i = O(k).$$

Now, suppose we can safely assume that $\mathbf{Pr}[A_{i,j}] \leq 2^i/(k-1)$. Since $\alpha = 2$, the product $4c \cdot 2^{\log_2(k)+c_1-i} \cdot (2)^i/(k-1) = 4c \cdot 2^{c_1} \cdot k/(k-1)$ for all i, and so the

sum is $(\log_2(k) + c_1) \cdot 4c \cdot 2^{c_1} k/(k-1) = O(\log_2(k))$. Thus, we can conclude that $\mathbf{E}[d_T(j, j+1)] \leq O(\log k)$.

Now, for arbitrary x and y such that $1 \leq x < y \leq k$, we can break up the distance on the line into segments of length 1. Thus, without the assumption that $\mathbf{Pr}[A_{i,j}] \leq 2^i/(k-1)$, we conclude that

$$\mathbf{E}[d_T(x, y)] = \sum_{i=x}^{y-1} \mathbf{E}[d_T(i, i+1)] = O(k)(y - x) = O(k)d_L(x, y).$$

But, with the assumption we get a slightly better result:

$$\mathbf{E}[d_T(x, y)] = \sum_{i=x}^{y-1} \mathbf{E}[d_T(i, i+1)] = O(\log k)(y - x) = O(\log k)d_L(x, y).$$

□

Now, consider the following algorithm to construct an HST from the interval $[1, k]$ on the line:

Algorithm 1 HST-Construct(L)

$T \leftarrow$ a perfect binary HST with $2k$ leaves
Randomly place the root of T within the integers of $[1, k]$.

Since the HST has $2k$ leaves (although half are trimmed off because they hang outside of the interval), and the root is placed within the interval, this root has height $\log(2k) = \log(k) + 1$.

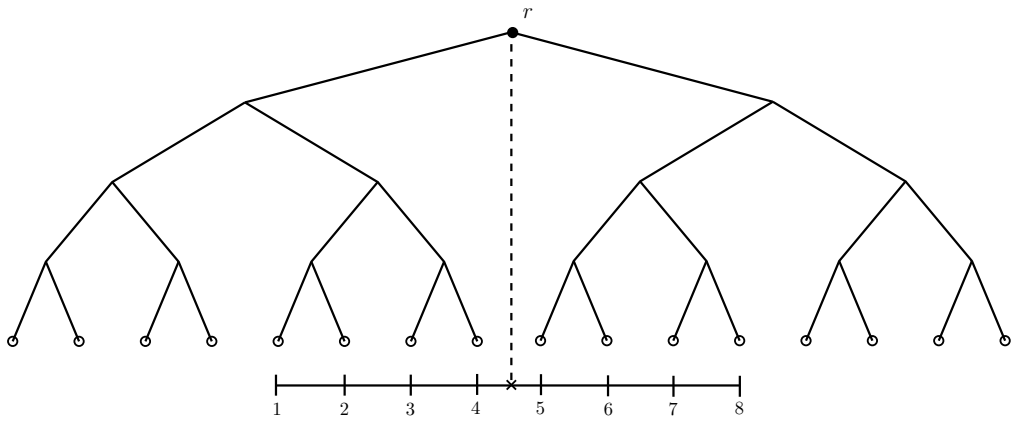

Figure 2.1: Here, using $k = 8$, we have randomly placed the root r of the perfect binary tree of $2k$ leaves on the interval $[1, k]$. We want the leaves to line up with the integers in $[1, k]$ as in the diagram.

Remark 2.2.2. Here is an alternate view of the algorithm that will make analysis easier later on. Rather than fixing the interval $[1, k]$ and picking a random location for the root cut within this interval, we can instead fix the binary tree of $2k$ leaves on the interval $[1, 2k]$, and then randomly select some $j \in [2, k + 1]$ so that the interval we are interested in is $[j, k + j - 1]$.

This next remark simply states that the cuts created by HST-Construct in the left half of the tree of $2k$ leaves are repeated in the right half of the tree. This just follows from the structure of the tree, and so the proof is omitted.

Remark 2.2.3. There is a cut of depth i in $[j, j + 1]$ if and only if there is a cut of depth i in $[j + k, j + k + 1]$, for all $1 \leq j < k$.

Now, we are able to show the main lemma for this section. This lemma allows us to bound the probability of an small interval being cut by a cut of large depth.

Lemma 2.2.4. *For* $0 \leq i < h$,

$$\boldsymbol{Pr}[A_{i,j}] \leq 2^i/(k-1).$$

Proof. We go by induction on i. At depth 0, the algorithm is defined to place the root cut uniformly at random at the $k - 1$ possible places for cuts, so for all j, $\mathbf{Pr}[A_{0,j}] \leq 1/(k-1)$. Assume that for some $i \geq 1$, $\mathbf{Pr}[A_{i-1,j}] \leq 2^{i-1}/(k-1)$.

Consider the tree of $2k$ leaves on the interval $[1, 2k]$, first. The node of a cut at depth $i - 1$ has two children, who are cuts at depth i. Also, any cut at depth i must be either the left child or right child of some cut at depth $i - 1$.

Now, we redefine the notion of a child cut on the interval $[j, j+k-1]$. For some cut x in $[j, j+k-1]$, if both children (in the tree) of the the node occupied by x lie within $[j, j + k - 1]$, then these two children are still called the children of x on the interval. Note that it is not possible for neither child to lie within $[j, j + k - 1]$, since x lies in between both children, and would thus not be within $[j, j + k - 1]$, either. But now, suppose that x has one child that does not lie in $[j, j+k-1]$. Let this child be called y. If y lies in the interval $[m, m + 1]$ for some $m < j$, then by Remark 2.2.3, there exists a child y' that lies in $[m + k, m + k + 1]$. Since $m + k \leq j + k - 1$, y' must lie within the interval, so we call y' the left child of x on the interval. Likewise, if $m > j + k - 1$, then by Remark 2.2.3, there exists a child y' that lies in $[m - k, m - k + 1]$, and since $m - k \geq j$, y' lies within the interval, so we call y' the right child of x on the interval.

We now show that under this definition of children in the interval, every cut of depth i in the interval has a parent cut of depth $i - 1$ in the interval. Let y be a cut of depth i in the interval. Of course, in the tree, y has a parent, and suppose it is located in $[m, m + 1]$. If $m < j$, then $m + k \leq j + k - 1$, so the cut of depth $i - 1$ in $[m + k, m + k + 1]$ would be the parent cut of y. If $m > j + k - 1$, then $m - k \geq j$, so the cut of depth $i - 1$ in $[m - k, m - k + 1]$ would be the parent cut of y. Thus, in both cases y has a parent in the interval.

We have shown that every cut of depth i belongs to a parent cut of depth $i-1$. Furthermore, a parent cut of depth $i-1$ cannot have more than two children. Thus, there is a cut of depth i within the interval $[m, m+1]$ if and only if there is either a cut of depth $i-1$ in one of two possible positions for cuts within $[j, j+k-1]$. By the induction hypothesis, the probability of either of these happening is at most $2^{i-1}/(k-1)$, so the probability that such a cut of depth i exists in $[m, m+1]$ can be at most $2^i/(k-1)$, which completes the induction step. □

Since the height of the tree is $\log_2(k)+1$, we can apply the above lemma to Lemma 2.2.1 to get that the algorithm HST-Construct builds a tree that stretches the expected distance of two points x and y on the tree by at most $O(\log k)d_L(x,y)$. To verify that the tree T outputted from HST-Construct is such that $d_L(x,y) \leq d_T(x,y)$, note that for any two points x and y, a cut of depth at least $\lfloor \log_2 d_L(x,y) \rfloor$ must lie within the interval $[x,y]$, and so $d_T(x,y) \geq 2 \cdot 2^{\lfloor \log_2 d_L(x,y) \rfloor} \geq d_L(x,y)$, since there are two edges in the tree adjacent to this cut, each of length α^i for i being the depth of the cut.

From now on, the topic of constructing an HST from the line is not discussed in further detail, and it is assumed that for any line L, any matching produced on an HST T outputted from HST-Construct(L) will be such that the cost of the matching on T will be on expectation at most $O(\log k)$ times the cost of the same matching on the line. It follows that an algorithm which is c-competitive on the HST T associated with L is $O(c \log k)$-comepetitive on L.

2.3 Assumptions

In this section, we present a series of assumptions that we make for any given instance of this problem. For each of these assumptions, we make the metric space slightly easier to deal with, while incurring at most a constant multiplicative factor in the competitive ratio of any algorithm. First, we show that if we assume that requests appear only at the locations of servers in the metric, then the competitive ratio on the actual instance will only rise by at most 3 times the competitive ratio on the instance with colocated requests and servers. This assumption turns out to be quite useful. For example, in the HST construction, since we already assume that servers appear only at the leaves of the tree, we can now assume safely that requests also appear only at the leaves. This simplifies matters considerably, since the analysis of the recursive structure of the HST is much cleaner.

The second assumption that we justify in this section is more than just a convenience. Fix an instance I, and let Δ be the ratio of the maximum distance between

any two points over the minimum distance between any two points. Formally,

$$\Delta = \frac{\max_{x,y} d(x,y)}{\min_{x,y} d(x,y)}.$$

Some of the algorithms that we present in later chapters yield $O(\log \Delta)$-competitiveness. However, since Δ can be exponential (or worse) in an untreated metric space, the upper bound of $O(\log \Delta)$ can be rather weak. However, if we can somehow construct an instance I' where $\Delta = \text{poly}(k)$ on I', and show that a c-competitive algorithm on I' is still competitive on I, then these algorithms would be upper-bounded by $O(\log k)$. The analysis provided at the end of this chapter shows that we can indeed construct such an instance I' for all I such that we incur at most a multiplicative factor of 6 in the competitive ratio when translating our matching from I' back to I.

2.3.1 Colocated Requests and Servers

We'd like to make the following assumption on the underlying metric: every request is colocated with some server. Consider the following algorithm Colocate_A, defined for a fixed algorithm A.

Algorithm 2 $\text{Colocate}_A(R)$

for all requests $r_i \in R$ **do**
  Move r_i to a closest server s_i^*.
  Assign r_i (now colocated with s_i^*) using A.
end for

In the next lemma, we show that the competitive ratio for an algorithm between an instance without colocated requests and servers and the output of Colocate on that instance changes by at most a constant multiplicative factor.

Lemma 2.3.1. *For any algorithm A that is c-competitive,* Colocate_A *is at most $(2c+1)$-competitive.*

Proof. For any r_i, let s_i^* be a closest server to r_i. Let $\hat{s}_i$ be the server that algorithm A assigns r_i to. Note that Colocate_A also assigns r_i to $\hat{s}_i$. Define Opt_1 to be the optimal matching under the original instance I_1 for which we want to show Colocate_A is $2c$-competitive, and Opt_2 the optimal matching under the modified I_1 where every request is colocated with a server, where we know that A is c-competitive. We know that $A(I_2) \leq c \cdot \text{Opt}_2(I_2)$, and we want to show that $\text{Colocate}_A(I_1) \leq (2c+1) \cdot \text{Opt}_1(I_1)$.

Let $\bar{s}_i$ be the server that a request r_i is assigned to in $\text{Opt}_2(I_2)$. First, note that for each $r_i \in R$, $d(r_i, \bar{s}_i) \geq d(r_i, s_i^*)$ by the definition of s_i^*. Thus, if we construct the

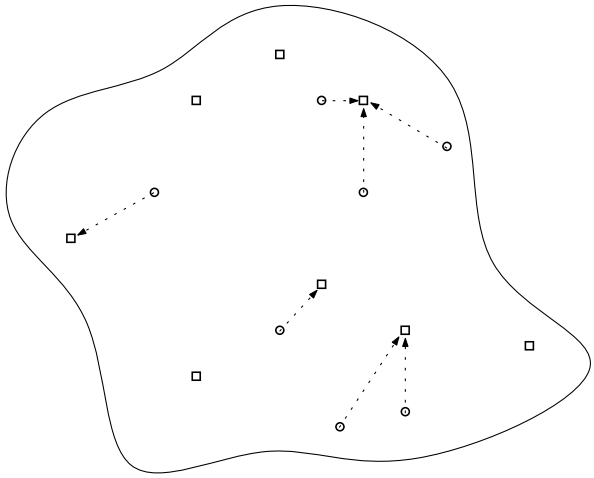

Figure 2.2: As each request arrives, Colocate$_A$ first moves each request to its nearest server, and then runs A. Note that Colocate$_A$ may end up initially moving multiple requests to the same server, as in the example.

matching $\mathrm{Opt}_1(I_2)$, the cost of this matching is at most $2\,\mathrm{Opt}_1(I_1)$. Therefore, we have that $\mathrm{Opt}_2(I_2) \leq \mathrm{Opt}_1(I_2) \leq 2\,\mathrm{Opt}_1(I_1)$. This also means that $A(I_2) \leq 2c\,\mathrm{Opt}_1(I_1)$.

Now, consider Colocate$_A(I_1)$. We have by the triangle inequality that

$$\mathrm{Colocate}_A(I_1) = \sum_i d(r_i, \hat{s}_i) \leq \sum_i d(r_i, \bar{s}_i) + \sum_i d(\bar{s}_i, \hat{s}_i)$$

But note that $\mathrm{Opt}_2(I_2) = \sum_i d(r_i, \bar{s}_i)$ and $\sum_i d(\bar{s}_i, \hat{s}_i) \leq A(I_2)$, since A assumes each request r_i appears at $\bar{s}_i$, and the cost incurred by A of r_i must be at least the distance from r_i to its closest server, for each r_i. This allows us to conclude that $\mathrm{Colocate}_A(I_i) \leq \mathrm{Opt}_2(I_2) + 2c\,\mathrm{Opt}_1(I_1) \leq (2c+1)\,\mathrm{Opt}_1(I_1)$. □

The above lemma allows us to take any instance of the online metric matching problem, move each request to its closest server, and then run a standard algorithm that assumes colocation of requests and servers. The gaurantee is that the competitive ratio of the original instance (where all requests may not be colocated with servers) is no more than 3 times the competitive ratio of the dummy instance that we have created, where all requests are colocated with servers.

2.3.2 Distances Polynomial in k

Suppose that we know $c(\mathrm{Opt})$ for some instance of the problem on the line. Let A be an algorithm that does not match any request r to a server s such that $d(r, s) > c(\mathrm{Opt})$.

Consider the following algorithm Divide_A^ρ, which breaks the line up into smaller instances $I_1, \cdots, I_m$ based on ρ.

Algorithm 3 $\text{Divide}_A^\rho(I)$

Break the line of I up into smaller instances $I_1, \cdots, I_\ell$ such that for each instance I_j, the largest distance between two servers in I_j is less than ρ.
for all requests $r_i \in R$ **do**
  Let I_j be the instance containing r_i.
  if there are no more servers in I_j **then**
    output FAIL and halt
  else
    Use $A(I_j)$ to determine which server r_i assigns to.
  end if
end for

We now show that if we somehow knew the value of $c(\text{Opt}(I))$ for a fixed instance I, then we could divide up the line into smaller segments and recursively run our algorithm A on these segments.

Lemma 2.3.2. *Denote* $\text{Opt}(I)$ *as the optimal matching on the original instance, and* $\text{Opt}(I_j)$ *as the optimal matching on the segment* I_j *produced from* $\text{Divide}_A^{c(\text{Opt}(I))}$. *Then,*

$$c(\text{Opt}(I)) = \sum_{I_j \in I} c(\text{Opt}(I_j))$$

Proof. To obtain the segments $I_1, \cdots, I_\ell$ from the original instance I, only the edges of length greater than $c(\text{Opt})$ are removed. Note however that these edges cannot be used in the matching by $\text{Opt}(I)$, and so the lemma follows. □

Remark 2.3.3. Combining the previous two lemmas, we get that if A is c-competitive for every instance I, then $\text{Divide}_A^{c(\text{Opt}(I))}$ is also c-competitive for every instance I.

Now, using m as the total number of servers, consider:

Algorithm 4 $\text{Contract}_A^\rho(I)$

while there is pair of adjacent servers s_i and s_j such that $d(s_i, s_j) < \rho/m^2$ **do**
  Contract the edge between s_i and s_j.
end while
Call this new instance I_2, and the original instance I_1.
for all requests $r_i \in R$ **do**
  Use $A(I_2)$ to determine which server r_i assigns to.
end for

The idea behind Contract_A is that for servers in the metric that are placed very closely together, we can "combine" them to form a pseudo-server, which, in the new instance, represents a cluster of servers. If a request chooses to assign to this pseudo-server, then we can actually match the request to an arbitrary member of that pseudo-server on the original instance. We just need to show that such a modified instance does not affect the competitive ratio by much.

Lemma 2.3.4.

$$c(\text{Contract}_A^\rho(I_1)) \leq c(A(I_2)) + \rho$$

Proof. Let $d_{I_1}(s_i, s_j)$ be the distance between two servers s_i and s_j on the original instance I_1, and $d_{I_2}(s_i, s_j)$ their distance on I_2. Then, since the procedure can only contract at most $m-1$ edges (every edge between each pair of adjacent servers), $d_{I_1}(s_i, s_j) \leq d_{I_2}(s_i, s_j) + ((m-1)/m^2) \cdot \rho$. Thus, for each of the $k \leq m$ requests r_i, the cost of r_i's assignment by A on I_2 can only increase by at most $(m-1)/m^2 \leq 1/m$ times ρ. Thus, the total cost of $\text{Contract}_A^\rho(I_1)$ can only be $(k/m) \cdot \rho \leq \rho$ more than the cost of A on I_2. □

Since we are interested in multiplicative approximation guarantees, we can scale distances so that the minimum non-zero distance is 1. Define

$$\text{Simplify}_A^\rho(I) = \text{Contract}^\rho_{\text{Divide}_A^\rho(I)}(I'),$$

where I' is the instance of the problem that $\text{Divide}_A^\rho(I)$ generates, and $\text{Simplify}_A^\rho(I)$ fails if $\text{Divide}_A^\rho(I)$ fails. Now, consider the following algorithm:

Algorithm 5 $\text{Guess\&Verify}_A(I)$

```
ρ ← 1
for all requests r_i ∈ R do
  while Simplify_A^ρ(I) fails do
    ρ ← 2 · ρ
  end while
  Use Simplify_A^ρ(I) to determine which server r_i assigns to.
end for
```

This algorithm combines the two previous algorithms to form a single procedure that we can invoke for an arbitrary instance to form a new instance that has its max distance over min distance ratio to be in $\text{poly}(k)$. However, we must address the problem that we do not know $c(\text{Opt})$ in hindsight, as was assumed in the lemmas regarding Divide_A. To account for this, we try to "guess" the value of $c(\text{Opt})$, and we repeatedly double our guess until the algorithm does not fail.

Lemma 2.3.5. *If algorithm A is c-competitive, and ρ_i is the ρ-value used in the algorithm on r_i, then $c(\text{Simplify}_A^{\rho_i}(I)) \leq 3c\rho_i$.*

Proof. First, note that for all i, $\text{Opt}(I, i) \leq \rho_i \leq 2 \cdot \text{Opt}(I, i)$. To see this, note that $\text{Divide}_A^{\rho}(I)$ can only fail if $\rho < \text{Opt}(I, i)$, since this means that there exists some sub-instance where an edge that was intended to be used by $\text{Opt}(I, i)$ was cut by Divide_A^{ρ}. For the upper bound, we stop increasing ρ as soon as the algorithm does not fail, so ρ_i is in fact the smallest ρ-value that is a power of 2 and is such that $\rho_i \geq \text{Opt}(I, i)$.

For the main proof, we go by induction on i, the number of requests that have so far appeared, using $\text{Alg}(I, i)$ to represent the cost of algorithm Alg on instance I for the first i requests. For the base case, the claim holds since $\text{Simplify}_A^{\rho_1}(I, 1) = A(I, 1) \leq c \cdot \text{Opt}(I, 1)$. Assume inductively that

$$\text{Simplify}_A^{\rho_{i-1}}(I, i-1) \leq 3c\rho_{i-1}.$$

There are two cases to consider on ρ_i. Let's denote I' as the modified instance that we run A on in $\text{Guess\&Verify}_A(I)$. If $\rho_i = \rho_{i-1}$, then $\text{Simplify}_A^{\rho_i}(I, i) \leq A(I', i) + \rho_i$ by Lemma 2.3.4. Now, since A is c-competitive, $A(I', i) \leq c \cdot \text{Opt}(I', i)$, and $\text{Opt}(I', i) \leq \text{Opt}(I, i)$, since I' is a version of I where all distances have either stayed the same or shrunk. Since $\rho_i \geq \text{Opt}(I, i)$, we get that

$$\text{Simplify}_A^{\rho_i}(I, i) \leq 2\rho_i \leq 3c\rho_i,$$

since $c \geq 1$ by the definition of competitive ratio.

Now, consider the case where $\rho_i \geq 2 \cdot \rho_{i-1}$, since this is the only other possibility if $\rho_i \neq \rho_{i-1}$. Here is one way to upper-bound $\text{Simplify}_A^{\rho_i}(I, i)$, conceptually. Given the assignment of requests we have made so far from $\text{Simplify}_A^{\rho_{i-1}}(I, i-1)$, we can imagine sending the requests back to their original location, and then assigning them according to the matching dictated by $A(I', i)$, since by the triangle inequality this is an upper bound on the cost of the actual matching made. Thus, we have by Lemma 2.3.4 again that

$$\text{Simplify}_A^{\rho_i}(I, i) \leq \text{Simplify}_A^{\rho_{i-1}}(I, i-1) + A(I', i) + \rho_i$$

Inductively, $\text{Simplify}_A^{\rho_{i-1}}(I, i-1) \leq 3c\rho_{i-1} \leq (3c/2)\rho_i$. Again, $A(I', i) \leq c \cdot \text{Opt}(I', i) \leq c \cdot \text{Opt}(I, i) \leq c\rho_i$, so we have that

$$\text{Simplify}_A^{\rho_i}(I, i) \leq c\rho_i(3/2 + 1 + 1/c) \leq 3c\rho_i$$

as desired. □

Using the lemma we have just proved, we can deduce the following corollary that ensures that running our algorithm on the result of our procedure will still attain small competitive ratio.

Corollary 2.3.6. Guess&Verify$_A$ *is* $6c$*-competitive if* A *is* c*-competitive.*

Proof. Note that Guess&Verify$_A$ uses Simplify$_A^{\rho_k}(I)$, where ρ_k is the final value of the ρ used in the algorithm. This is at most $2 \cdot \text{Opt}(I)$ as shown in the previous lemma, so Simplify$_A^{\rho_k}(I) \leq 3c\rho_k \leq 6c \cdot \text{Opt}(I)$, which means that Guess&Verify$_A \leq 6c \cdot \text{Opt}(I)$. □

The algorithms presented in the next few chapters always make the assumption that the instance I for which it must run on has colocated requests and servers and all interpoint distances are polynomial in k. Since the online metric matching problem considers all instances, it may not be the case that I has these properties. Thus, we implicitly run the procedures Colocate and Guess&Verify to an instance I given by the adversary in order to create an instance I' with the two desired properties. Then, the gaurantees provided in this section show us that if an algorithm on I' is c-competitive, then it must be $O(c)$-competitive on I. Such a result is quite appealing, since it allows us to make assumptions that simplify techniques that would otherwise be more intricate while only incurring a constant multiplicative factor into the actual competitive ratio.

Chapter 3

An hd-competitive Algorithm

Suppose that on an HST with k servers, an algorithm is faced with k requests that appear in an online manner. The algorithm then makes some matching of the k requests to the servers. Now, since we are assuming that all requests and servers appear at the leaves, we can imagine drawing directed arrows from each request to the server to which it was matched, where an arrowhead is placed on each edge in the path from the request to its matching server. In this diagram, there are probably some edges of the tree that have been labeled with arrows pointing in opposing directions (unless the matching is optimal).

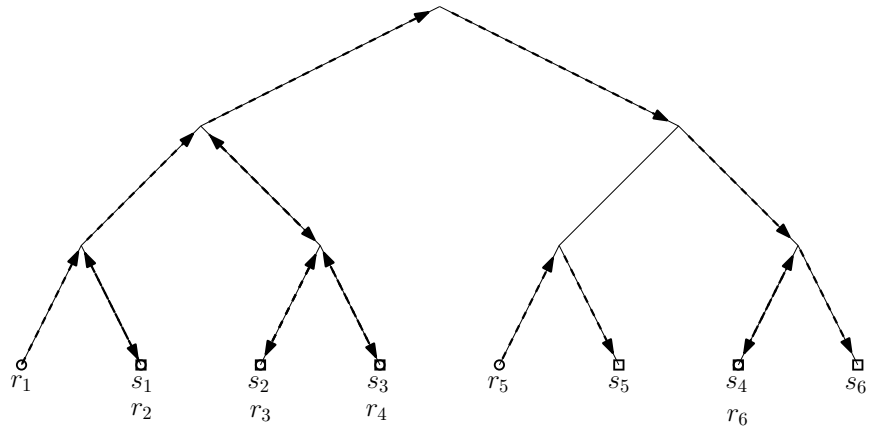

Figure 3.1: The diagram of assignments for the matching $M = \{(r_i, s_i)\}$ for all $1 \leq i \leq 6$. One can create such a diagram by imagining the path that each request travels along when being moved to its matching server.

Let's consider some edge e where two arrowheads that lie on e happen to point in opposite directions. It turns out that if we delete this pair of arrowheads but

change nothing else, then the resulting diagram describes a new matching that has cost strictly less than the original matching created by the algorithm. To see this, note that the only difference in the new diagram is that the new matching crosses e two less times than the old matching did (one for each arrowhead).

This conceptual procedure of deleting pairs of arrowheads is the primary motivation for the analysis of this chapter. Intuitively, the number and level of pairs of arrowheads that point in opposite directions can be used as a measure of the performance of an algorithm. We will later show that, for all algorithms that match in a somewhat greedy fashion, one can always look at the matching produced by the algorithm and repeatedly delete pairs of opposing arrowheads until no more exist. Interestingly, the resulting diagram represents an optimal matching. So, in order to measure the "distance" that an algorithm's matching is from an optimal matching, we attempt to form sequences of these arrowhead pairs, and then bounding the total amount of cost incurred by a worst-case such sequence.

In the final section of this chapter, we then relate these sets of sequences that we have created to the actual cost of a matching. We then analyze the deterministic greedy algorithm's performance on the tree, and we are able to show that on each level of the tree, deterministic greedy will pay at most d times the amount that the optimal matching pays, for d being the degree of the tree. Since there are h levels of the tree, this implies a competitive ratio of hd.

3.1 Matching Definitions on an HST

For an edge e on the tree T, define the parent node of e as its endpoint that is closer to the root of the tree, and the child node as the endpoint further from the root of the tree. Now, the level of an edge $e = \{v_1, v_2\}$ is the level of its parent node: in other words, $L(e) = \min(L(v_1), L(v_2))$.

Recall that an α-HST is a rooted tree $T = (V, E)$ such that for any edge e, the weight of e is

$$w(e) = \frac{c}{\alpha^{L(e)-1}}.$$

From now on, T will be used to represent an α-HST where all requests and servers occur at the leaves.

Definition 3.1.1 (Matching)**.** A matching $M : R \to S$ is a map from a set of requests R to the set of servers S.

Definition 3.1.2 (Matching Paths)**.** Let (r_i, s_i) be an element of some matching M, with $r_i \in R$ and $s_i \in S$. Suppose P_i represents the unique path from r_i to s_i, such that P_i has the following form:

$$P_i = \langle r_i, v_1, v_2, \cdots, v_l, s_i \rangle.$$

We can define a matching path $\vec{P}_i$ corresponding to the matched pair r_i, s_i thus:

$$\vec{P}_i = \{(r_i, v_1), (v_1, v_2), \cdots, (v_l, s_i)\},$$

where the direction associated with each arc is oriented towards s_i and away from r_i in P.

So far, it has been established that a matching M, which is a map from requests to servers, can be described as a set of (directed) matching paths from each request to its corresponding server. These matching paths consist of a collection of arcs, and let $A(M)$ be the multiset of all arcs in used in matching M. For each arc $a \in A(M)$, let $t(a) = i$ if $a \in \vec{P}_i$. Moreover, let $e(a)$ be the edge associated with a; i.e., if $a = (x, y)$ then $e(a) = \{x, y\}$. An up arc is one that is directed towards the root, whereas a down arc points away from the root.

Fix a matching M, which defines the matching paths $\vec{P}_i$, and hence the multiset of arcs $A(M)$. For some edge e in T, let Λ_e be the multiset of arcs associated with e in M. Moreover, let Λ_e^u be the multiset of all up-arcs at e, and Λ_e^d be the set of all down-arcs. It follows that $\Lambda_e = \Lambda_e^u \cup \Lambda_e^d$. Observe that for any arc a, it holds that $a \in \Lambda_{e(a)}$; moreover, $A(M) = \cup_{e \in E} \Lambda_e$.

Definition 3.1.3 (Cost of Matching)**.** For a matching M, the cost of the matching, denoted by $c(M)$, is defined as the sum of the weights of all arcs associated with M; i.e.,

$$c(M) = \sum_{\vec{P}_i} \sum_{a \in \vec{P}_i} w(e(a)) = \sum_{e \in E} w(e) \cdot |\Lambda_e|.$$

An optimal matching is one that achieves the minimum cost.

3.1.1 Characterizing Optimal Matchings

The following definition and the next lemma introduces an important characteristic of matchings of optimal cost.

Definition 3.1.4. A conflicted edge for some matching M is an edge such that both Λ_e^u and Λ_e^d are non-empty.

Lemma 3.1.5. *Let $M : R \to S$ be a matching from the set of requests to the set of servers on an α-HST, where all servers and requests are at the leaves. Assume that for every request r, if $M(r) = s$, then there does not exist some unassigned s' such that $d(r, s') < d(r, s)$. If M has no conflicted edges, then M is optimal.*

Before we prove this lemma, we present a useful definition. For some edge e, let $T_e = (V_e, E_e)$ denote the tree rooted at the child node of e. Also, let R_{T_e} denote the set of all requests in T_e, with S_{T_e} the set of all servers in T_e; in other words, $R_{T_e} = R \cap V_e$ and $S_{T_e} = S \cap V_e$.

Proof. Every request must be matched to a server in any matching—thus, at least $\beta_e = |R_{T_e}| - |S_{T_e}|$ requests from R_{T_e} must be matched to servers not in T_e. Thus, for any matching M, the number of up-arcs on e is $|\Lambda_e^u| \geq \beta_e$. Similarly, at least $\gamma_e = (|R| - |R_{T_e}|) - (|S| - |S_{T_e}|)$ requests outside of T_e must match to servers within T_e. Consequently, it must also be the case that $|\Lambda_e^d| \geq \gamma_e$.

For a contradiction, suppose the matching M is not optimal. Then there exists some edge e such that $|\Lambda_e^u| > \beta_e$ or $|\Lambda_e^d| > \gamma_e$.

If $|\Lambda_e^u| > \beta_e$, then at least one request r in T_e must have assigned to a server s outside of T_e. Also, we have that there exists some server s' in T_e that is not assigned by a request in T_e. Let r' be the request that is matched to s'. Then r' is not in T_e. The matching paths used by the assignments (r, s) and (r', s') then cause e to be conflicted.

The proof is similar for when $|\Lambda_e^d| > \gamma_e$. In both cases, we get a contradiction, implying that M is an optimal matching. □

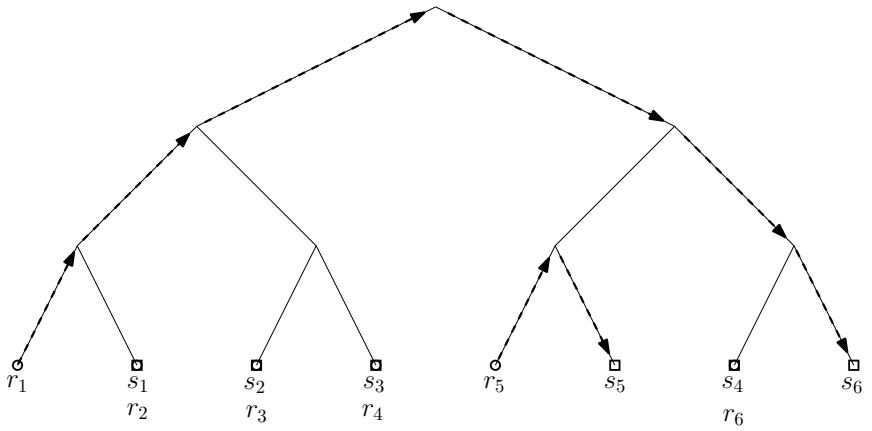

Figure 3.2: An example of how removing all conflict edges of a matching results in an optimal matching. The original matching from Figure 3.1 is represented here after deleting all edges with pairs of arrowheads pointing in opposite directions. The new matching we can infer from this diagram is $M^* = \{(r_1, s_6), (r_5, s_5)\}$ and then all other requests assigned to their colocated servers.

3.2 A Deterministic Algorithm

Consider the following algorithm $\mathcal{A}_{det}$ for matching requests to servers on T:

Algorithm 6 $\mathcal{A}_{det}$

for all requests $r_i \in R$ **do**
  Assign r_i to a closest unassigned server.
end for

Let M be the matching produced by $\mathcal{A}_{det}$.

Lemma 3.2.1. *For every request r, if $M(r) = s$, then there does not exist any unassigned server s' such that $d(r, s') < d(r, s)$.*

Proof. For some request r, with $M(r) = s$, suppose for the sake of contradiction that some s' remained unassigned and $d(r, s') < d(r, s)$. Then, when the algorithm chose to assign r, it chose a closest server. By definition, this means that the server it chooses, s, must be such that all other servers have distance at least $d(r, s)$. The existence of s' provides the necessary contradiction to establish this lemma. □

The above lemma must be established in order to apply Lemma 3.1.5 to M, which would allow the cost of M to be directly compared to the cost of the optimal matching. Let us now look closer at the nature of conflicted edges.

An arc a_1 is said to be added before an arc a_2 if $t(a_1) < t(a_2)$. Likewise, a_1 is added after a_2 if $t(a_1) > t(a_2)$.

Lemma 3.2.2. *As algorithm $\mathcal{A}_{det}$ is being run, all down arcs are added before any up arc is added. In other words, for any edge e covered by the matching,*

$$\max_{a \in \Lambda_e^d}(t(a)) < \min_{a \in \Lambda_e^u}(t(a)).$$

Proof. When a request r traverses an edge e upwards, this means that all servers in T_e have been assigned. When a request r' traverses e downwards, r' is matched to some unassigned server within T_e. Thus, if a down arc occurs after an up arc, then a contradiction forms. More formally, it is a contradiction if there exists some $a \in \Lambda_e^u$ and $a' \in \Lambda_e^d$ where $t(a) < t(a')$. □

At this point, more precision is needed for the categorization of conflicted edges. Rather than simply labeling an edge as conflicted or not, the following definition can be used to establish the number of conflicts at an edge.

Definition 3.2.3. A digon is a pair of arcs (a_1, a_2) such that a_1 in Λ_e^u and a_2 in Λ_e^d for some edge e. A set of digons is said to be distinct if no two digons of the set share an arc.

Lemma 3.2.4. *Let M be a matching produced by $\mathcal{A}_{det}$. Consider only the arcs in $A(M)$ corresponding to level i edges, and fix any maximal set of digons among these arcs. If the set of arcs is non-empty, then there exist at least two arcs a_1 and a_2 that do not belong to a digon, where a_1 is an up arc, and a_2 is a down arc. Also, these two arcs lie on non-conflicted edges.*

Proof. Let a_1 be the arc on some edge e_1 at level i, such that over all arcs at level i, $t(a_1)$ is minimal. Intuitively, this arc represents the first request that traversed level i. We know by Lemma 3.2.2 that no more down arcs can occur on e_1. Since a_1 was also the first request to traverse this level, there could not have been any down arcs before a_1. Thus, $\Lambda^d_{e_1}$ is empty, and so a_1 cannot be paired into a digon, and e_1 is non-conflicted.

Similarly, let a_2 be the arc on edge e_2 such that over all arcs at level i, $t(a_2)$ is maximal. Intuitively, this arc represents the last request that traversed level i. We know by Lemma 3.2.2 that no up arcs can occurred on e_1 before a_2. Since a_2 was also the last request to traverse this level, there could not have been any up arcs after a_2. Thus, $\Lambda^u_{e_2}$ is empty, and so a_2 cannot be paired into a digon, and e_2 is non-conflicted. □

The next lemma shows how to algorithmically obtain an optimal matching from any matching produced by $\mathcal{A}_{det}$ by repeatedly removing digons.

Lemma 3.2.5. *Let M be some matching produced by $\mathcal{A}_{det}$. Suppose two arcs a_1 and a_2 from $A(M)$ are chosen such that $\{a_1, a_2\}$ form a digon. Then there exists a matching M' such that $A(M') = A(M) \setminus \{a_1, a_2\}$. Also, $c(M') \leq c(M)$.*

Proof. Suppose that for all k, $M(r_k) = s_k$. Let $\vec{P}_i$ (connecting r_i to s_i) be the matching path containing a_1, and $\vec{P}_j$ (connecting r_j to s_j) be the matching path containing a_2. Define M' as follows:

$$M'(r_k) = \begin{cases} s_j, & \text{if } k = i \\ s_i, & \text{if } k = j \\ s_k, & \text{otherwise} \end{cases}$$

By switching the matching paths $\vec{P}_i$ with $\vec{P}_j$ to not use $e(a_1)$ in M', a valid matching is still maintained. Since all other arcs in M' also exist in M, we see that $c(M') = c(M) - 2w(e(a_1)) \leq c(M)$, with the equality being strict if $w(e(a_1)) > 0$. □

The above lemma, along with Lemmas 3.1.5 and 3.2.1 immediately imply the following.

Corollary 3.2.6. *If digons are repeatedly removed from a matching M produced by $\mathcal{A}_{det}$ until there are no more digons, the resulting matching must be optimal.*

If the number of digons removed to transform M into the optimal matching is small, we get a bound on the competitive ratio. To do this, we next consider ways to bound the number of digons.

3.3 Conflict Sequences

Loosely speaking, a conflict sequence is an ordered sequence of arcs that begins with an up arc, followed by 0 or more digons, and is terminated by a down arc. The digons are, however, obtained by pairing arcs in a specific fashion, as we describe next.

A sequence $\sigma = a_0, a_1, \cdots, a_{|\sigma_i|-1}$ of arcs at level i is a conflict sequence if

- a_j is an up arc if j is odd, and a down arc if j is even,
- a_{2j-1} and a_{2j} must belong to the same matching path (i.e., $t(a_{2j-1}) = t(a_{2j})$), and
- a_{2j+1} belongs to a matching path added after a_{2j} (i.e., $t(a_{2j}) < t(a_{2j+1})$), but these two arcs form a digon (i.e., $e(a_{2j}) = e(a_{2j+1})$).

Additionally, a conflict sequence is maximal if it is not a proper subset of any other conflict sequence.

Let the function $\#(\sigma)$ represent the number of distinct edges covered by σ, and $|\sigma|$ denote the number of arcs in σ. Then the following facts establish a relationship between $\#(\sigma)$ and $|\sigma|$.

Fact 3.3.1. *For any conflict sequence $\sigma = \langle a_0, a_1, \cdots, a_{|\sigma|-1}\rangle$, we have $e(a_{2j}) \neq e(a_{2k})$ for all $k \neq j$.*

Proof. We already know that $e(a_{2j}) = e(a_{2j+1})$, and that a_{2j} is a down arc, whereas a_{2j+1} is an up arc. But by Lemma 3.2.2, if an arc a' in σ is such that $e(a_{2j}) = e(a')$, then $t(a_{2j}) < t(a') < t(a_{2j+1})$. However, since a_{2j} and a_{2j+1} follow consecutively in σ, no such arc can exist. □

Fact 3.3.2. *For any conflict sequence $\sigma = \langle a_0, a_1, \cdots, a_{|\sigma_i|-1}\rangle$, it holds that $\#(\sigma) = \frac{1}{2}|\sigma| + 1$.*

Proof. Since $e(a_{2j}) = e(a_{2j+1})$, every arc can be paired with the next arc in the sequence, except for the last arc of σ. Also, $e(a_{2j}) = e(a_{2k})$ only when $j = k$. The lemma follows. □

3.3.1 Canonical Conflict Sequences

We want to have some notion of a fixed set of conflict sequences for a matching, such that every arc of the matching belongs to a conflict sequence, no arc belongs to two conflict sequences. We will denote $\mathcal{S}$ as the set of these canonical conflict sequences, and $\mathcal{S}_i$ will be used to represent the set of canonical conflict sequences at level i of T.

Lemma 3.3.3. *Let $\mathcal{R}_{\mathcal{S}} : A \to A$ be the relation between two arcs a_1 and a_2 such that $a_1 \mathcal{R}_{\mathcal{S}} a_2$ if and only if a_1 and a_2 belong to the same canonical conflict sequence. Then, there exists a setting of canonical conflict sequences $\mathcal{S}$ such that $\mathcal{R}_{\mathcal{S}}$ is an equivalence relation.*

Proof. We first define several functions for constructing $\mathcal{S}$.

For the remainder of this proof, σ will be used to represent a conflict sequence with arcs $a_1, a_2, \cdots, a_n$.

Algorithm 7 merge($S = \{\sigma_1, \sigma_2, \cdots, \sigma_t\}$)

```
S* ← ∅
for all pairs of conflict sequences (σ_i, σ_j) ∈ S × S do
  a_i ← the last arc of σ_i
  a_j ← the first arc of σ_j
  if t(a_i) < t(a_j) and e(a_i) = e(a_j) then
    S ← S \ {σ_i, σ_j}
    S* ← S* ∪ {σ_j ∘ σ_i}
  end if
end for
return S*
```

Remark 3.3.4. Given a set of conflict sequences S, merge(S) will extend these sequences such that the sequences in the set returned will be maximal.

The next function we define allows us to create a conflict sequence at level $i-1$ based on a conflict sequence at level i. This sequence at level $i-1$ thus acts as the parent.

Algorithm 8 image(σ)

```
E ← the set of edges adjacent to edges covered by the arcs of σ at one level higher
if |E| > 1 then
  return a conflict sequence out of the arcs in E
else
  return ⊥
end if
```

Fact 3.3.5. *In* image(σ)*, the arcs of* E *are enough to form a valid conflict sequence.*

Proof. It remains to prove that the arcs lying on edges in E can be organized as a conflict sequence at level $i-1$. This can be done by showing that all properties of being a conflicted sequence hold on some sequence involving the arcs at E in some order.

For each edge $e \in E$, let $S(e)$ be the set of arcs of σ_i that are adjacent to e. Define

$$f(e) = \min_{a \in S(e)} t(a)$$

Create the sequence $S = e_0, e_1, \cdots, e_{|E|-1}$, where $f(e_0) < f(e_1) < \cdots < f(e_{|E|-1})$.

Define e_i^u as an up arc and e_i^d as a down arc on edge e_i. We want to show that

$$\sigma_{i-1} = e_0^u, e_1^d, e_1^u, e_2^d, e_2^u, e_3^d, \cdots, e_{|E|-2}^d, e_{|E|-2}^u, e_{|E|-1}^d$$

is a conflict sequence at level $i-1$.

By our definition, σ_{i-1} consists only of arcs found at level $i-1$.

By how we constructed σ_{i-1}, we have that each even term is an up arc. Also note that the request that traversed up on e_i must have been the same request that traversed down on e_{i+1}. Thus, we have that $t(e_i) = t(e_{i+1})$.

By how we constructed σ_{i-1}, we have that each odd term is repeated once again, establishing that $e(a_j) = e(a_{j+1})$. Also note that the request that traversed down on e_i must have come before the request that traversed up on e_i. Thus, we have that $t(e_i) = t(e_{i+1})$.

Therefore, σ_{i-1} is a conflict sequence at level $i-1$. □

Fact 3.3.6. *Let* σ_1 *and* σ_2 *be two conflict sequences at level* i*. Then* image(σ_1) *and* image(σ_2) *are disjoint.*

Proof. Suppose an arc a at level $i-1$ belongs to image(σ_1) and image(σ_2). Then there exists an arc a' at level i such that $t(a') = t(a)$.

By the way the parent sequences are formed from image, it must be the case that a' is also a member of two sequences σ_1' and σ_2' at level i. Thus, by induction, it only remains to show that an arc cannot belong to two sequences at the last level of the tree.

Suppose for the sake of contradiction that an arc $\hat{a}$ at the last level belongs to two sequences $\hat{\sigma_1}$ and $\hat{\sigma_2}$. Then there exist arcs $\hat{a_1}$ in $\hat{\sigma_1}$ and $\hat{a_2}$ in $\hat{\sigma_2}$ such that $t(\hat{a_1}) = t(a) = t(\hat{a_2})$. However, there cannot be three arcs that have the same t-values, a contradiction. □

This next function will be used to generate a set of canonical sequences at level i. It requires the canonical sequences at level $i+1$. Let $\mathcal{S}_i$ be the set of canonical sequences at level i.

Algorithm 9 canonicalize$(i, \mathcal{S}_{i+1})$

$\mathcal{S}_i \leftarrow \emptyset$
if $i = h$ (so we are canonicalizing the bottom level of the tree) **then**
  $\mathcal{S}_i \leftarrow \{(a_1, a_2) \in A_i : t(a_1) = t(a_2)\}$
else
  for all $s \in \mathcal{S}_{i+1}$ **do**
    $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup \{\text{image}(s)\}$
  end for
end if
return merge$(\mathcal{S}_i)$

Thus, we now have a way of generating the set of canonical sequences for the entire tree. It is trivial to show that $\mathcal{R}_{\mathcal{S}}$ is reflexive and symmetric. To show that $\mathcal{R}_{\mathcal{S}}$ is also transitive, it is enough to show that no arc can belong to more than one canonical sequence. Since we remove the arcs that form a canonical sequence before adding them into the set of canonical sequences, it cannot be the case that one arc is used twice to form two canonical sequences. Thus, we have shown that $\mathcal{R}_{\mathcal{S}}$ is an equivalence relation. □

Now that we have fully defined a complete procedure for determining the canonical sequences of a matching for any given level on an HST, we can define a function image which relates canonincal sequences between levels of the HST. We then show that certain desirable properties hold between a canonical sequence at level i and its image at level $i - 1$.

Lemma 3.3.7. *Let $\mathcal{F}$ be the map defined by the* image *function. Then,*

$$\mathcal{F} : \mathcal{S}_i \to \mathcal{S}_{i-1} \cup \{\perp\}$$

is such that:

1. *If $\mathcal{F}(\sigma) = \perp$ then $|\sigma| \leq 2d - 2$.*
2. *if $\mathcal{F}^{-1}(\sigma') = \{\sigma_1, \sigma_2, \cdots, \sigma_t\}$, then $\sum_{k=1}^{t} \left[\frac{|\sigma_k|}{\tau} - 2\right] \leq |\sigma'|$, where $\tau = d$.*

Proof. For property 1, we show the contrapositive. Let E be the set of edges adjacent to arcs in σ, and suppose that σ is a conflict sequence at level i. It is enough to show that if $|\sigma| > 2d - 2$, then $|E| \geq 1$.

Suppose, to get a contradiction, that all arcs of σ are adjacent to exactly one edge e at level $i - 1$. We know that for any edge e' at level i, at most two arcs from σ can lie on e'. Since the maximum degree of the tree is d, we also know that at most d

edges at level i can be adjacent to e. But by Fact 3.3.2, $\#(\sigma) > \frac{1}{2}(2d-2)+1 = d$, which is a contradiction.

For property 2, look at any arbitrary σ_k, and let E_k be the set of edges adjacent to arcs in σ_k. Note that $\#(\sigma_k) = \frac{1}{2}|\sigma_k| + 1$, and so the Pigeonhole Principle gives us that

$$|E_k| \geq \left\lceil \frac{\#(\sigma_k)}{d} \right\rceil = \left\lceil \frac{\frac{1}{2}|\sigma_k|+1}{d} \right\rceil \geq \frac{1}{2d}|\sigma_k|$$

We know by Fact 3.3.5 that we can pull arcs from the edges of E_k to form a conflict sequence. Let σ'_k be this new sequence. Fact 3.3.2 then gives us that

$$|\sigma'_k| = 2|E_k| - 2 \geq \frac{|\sigma_k|}{d} - 2.$$

Since k was arbitrarily chosen, we can then apply this to all such σ_k for $1 \leq k \leq t$. By Fact 3.3.6, the arcs in the image of σ_k are disjoint from the arcs of any other σ_j for $j \neq k$. Thus, we can simply take the sum over all σ_k to establish a lower bound on the union of the images of each σ_k. Note that the union of the images of each σ_k make up σ', and so property 2 follows. □

Now that we have fully specified the definition of the canonical conflict sequences for matchings on the tree, we can move onto the relation between canonical sequences and the cost incurred by them in the next section.

3.4 Competitive Ratio

From now on, we will use M to denote the matching produced by our algorithm, and M^* the optimal matching. Note that for a canonical sequence $\sigma \in \mathcal{S}_i$, our algorithm pays $|\sigma|$, hence

$$c(M) = \sum_{i=1}^{h} \left[\sum_{\sigma \in \mathcal{S}_i} |\sigma| \right] \frac{c}{\alpha^{i-1}}$$

If we define $s_i = \sum_{\sigma \in \mathcal{S}_i} |\sigma|$, we get

$$c(M) = c \sum_{i=1}^{h} \frac{s_i}{\alpha^{i-1}}.$$

We can also give a lower bound on the cost of the optimal matching; note that every canonical sequence is associated with exactly two non-conflicted edges. Recall from

Corollary 3.2.6 that removing the digons in each the canonical sequences for M gives us an optimal matching. Hence, if $a_i = |\mathcal{S}_i|$, it follows that

$$c(\text{Opt}) = 2c \sum_{i=1}^{h} \frac{a_i}{\alpha^{i-1}}.$$

Let us now relate the s_i's to a_i's as follows:

Lemma 3.4.1.

$$s_i \leq \tau(s_{i-1} + 2a_i).$$

Proof. Let σ' be a canonical sequence at level $i-1$, and suppose $\mathcal{F}(\sigma') = \{\sigma^1, \sigma^2, \cdots, \sigma^t\}$ be the canonical sequences at level i that map to σ'. Lemma 3.3.7(2) implies that $|\sigma'| \geq \sum_{\sigma^k \in \mathcal{F}(\sigma)} (\frac{|\sigma^k|}{\tau} - 2)$.

Furthermore, we know that each σ' at level i either maps to some sequence at level $i-1$, or maps to $\perp$—in which case its length is at most $2\tau - 2$, and hence satisfies $\frac{2|\sigma'|}{\tau} - 2 \leq 0$.

Using these facts, we can sum over all σ's in $\mathcal{S}_i$ to get

$$\sum_{\sigma \in \mathcal{S}_i} \left(\frac{|\sigma|}{\tau} - 2 \right) \leq \sum_{\sigma' \in \mathcal{S}_{i-1}} |\sigma'|.$$

Now using the definitions of s_i and a_i, we get

$$\frac{s_i}{\tau} - 2a_i \leq s_{i-1},$$

or $s_i \leq (s_{i-1} + 2a_i) \cdot \tau$, which proves the claim. □

Now, we would like to relate the right-hand side of Lemma 3.4.1 to $c(\text{Opt})$.

Lemma 3.4.2.

$$s_i \leq \tau^i \left(\frac{d}{\tau} a_1 + 2 \sum_{k=2}^{i} \frac{a_k}{\tau^{k-1}} \right) \leq \frac{d^i}{c} \cdot c(\text{Opt}).$$

Proof. Let us use Lemma 3.4.1 repeatedly to represent s_i in terms of s_1 thus:

$$s_i \leq s_1 \cdot \tau^{i-1} + \tau^i \cdot 2 \sum_{k=2}^{i} \frac{a_k}{\tau^{k-1}}.$$

We also know that $a_i \cdot d^i \geq s_i$, since the maximum length of any canonical sequence at level i is d^i. Thus, $s_1 \leq a_1 \cdot d$, and so we can infer that

$$s_i \leq \tau^i \left(\frac{d}{\tau} a_1 + 2 \sum_{k=2}^{i} \frac{a_k}{\tau^{k-1}} \right).$$

Recall that $\tau = d$; plugging this in, we get

$$s_i \leq d^i \left(a_1 + 2 \sum_{k=2}^{i} \frac{a_k}{d^{k-1}} \right) \leq d^i \cdot 2 \sum_{k=1}^{i} \frac{a_k}{d^{k-1}} \leq \frac{d^i}{c} \cdot c(\text{Opt}).$$

This completes the proof of the lemma. □

Plugging this into the definition of $c(M)$, we get

$$c(M) = c \sum_{i=1}^{h} \frac{s_i}{\alpha^{i-1}} \leq c \sum_{i=1}^{h} \frac{d^i}{c \cdot \alpha^{i-1}} \cdot c(\text{Opt}) \leq c(\text{Opt}) \cdot \sum_{i=1}^{h} \frac{d^i}{\alpha^{i-1}},$$

which implies the main result for deterministic algorithms:

Theorem 3.4.3. *The competitive ratio of the deterministic greedy algorithm on α-HSTs with maximum degree d is at most*

$$\sum_{i=1}^{h} \frac{d^i}{\alpha^{i-1}}.$$

In particular, if $d \leq \alpha$, then the competitive ratio is at most hd.

The crucial idea in this analysis of the deterministic greedy algorithm on HSTs was the formulation of the canonical sequences and the conclusions drawn in Lemma 3.3.7. We were able to relate the canonical sequences of levels i and $i-1$ in the tree with one another. Without the ability to infer information about the canonical sequence level $i-1$ given the canonical sequence at level i, a bound of hd would not have been possible. One natural question to ask is whether or not the randomized greedy algorithm improves upon the competitive ratio of the deterministic greedy algorithm, under the same framework of analysis (the use of canonical sequences). Unfortunately, it appears to be more difficult to relate the canonical sequences at adjacent levels, and so the analysis would require a modified approach.

If we consider the HST construction algorithm described in Chapter 2, then we note that the $d = \alpha = 2$, and $h = \log k$. Thus, a competitive ratio of hd on the tree implies a competitive ratio of $O(\log^2 k)$ on the line (since an extra $O(\log k)$ term is incurred when converting back from the tree to the line). An $O(\log^2 k)$ competitive ratio for general metrics has already been attained in [BBGN07]—though, their algorithm requires the computation of an offline optimal solution on the HST as each new request appears, whereas the deterministic greedy algorithm is much simpler. However, in the next chapter, we will see an example of an algorithm that exhibits $O(\log k)$ on the line.

Chapter 4

The Random-Subtree Algorithm

The randomized greedy algorithm on HSTs, analyzed by Meyerson, Nanavati and Poplawski [MNP06], achieves an $O(\log^3 k)$ by showing that randomized greedy has competitive ratio $O(\log k)$ on α-HSTs, where $\alpha = \Omega(\log k)$. However, in the case of the line metric, we know that we are able to construct degree-2 2-HSTs with $O(\log k)$ expected stretch. Somehow, we want to factor in the advantage of the HSTs having small degree into the analysis of the randomized greedy algorithm. In this chapter, we show that by making a modification to the way a request chooses a random closest server to break ties, we can replace the $\log k$ terms with $\log d$ terms, where d is the degree of the HST. In the case of the line and low-dimensional spaces where constant-degree HST constructions are known, this new dependency on $\log d$ rather than $\log k$ proves to be quite advantageous.

4.1 The Algorithm

Our algorithm, called Random-Subtree, for online metric matching on HSTs is the following: when a request l comes in, consider its lowest ancestor node a that contains a free server. Now choose a path down from a to a free server by going to a (uniformly) random subtree that contains a free server. We imagine that each leaf of the HST contains at most one server, and so when we reach a server/leaf s, we map the request l to this server s. Note that this does not bias towards subtrees that contain more servers (as the randomized greedy algorithm of [MNP06] does).

The performance of this algorithm is given by the following theorem:

Theorem 4.1.1. *The algorithm* Random-Subtree *is a* $2(1 + 1/\epsilon)H_d$*-competitive algorithm for online metric matching on degree-*d α*-HSTs, as long as* $\alpha \geq \max((1+\epsilon)H_d, 2)$.

Using the fact that we can approximate a k-point line metric by degree-2 2-HSTs with distortion $O(\log k)$ [FRT03], we immediately get an $O(\log k) \cdot 8H_2$-competitive

randomized algorithm for online metric matching on a line. This appears to be the first $O(\log k)$-competitive algorithm for this problem. Throughout the proof, we use H_d to represent the dth harmonic number, so $H_d = \sum_{i=1}^{d} 1/i$.

4.2 Proof of the One-Level Lemma

To prove the theorem, we first show a one-level lemma (Lemma 4.2.2) that accounts for the expected number of times an edge adjacent to the root in the HST is used by the algorithm: we show that this number is at most H_d times the number of times the optimal algorithm uses those edges. We then use this result for every level to show that the total cost still remains at most $O(H_d)$ times the optimum, as long as the parameter α for the HST is sufficiently large compared to H_d. This proof appears in Lemma 4.3.1, and immediately implies Theorem 4.1.1.

Consider an HST T with a set of requests $S \cup S'$ such that the requests in S originate at the leaves of T, and those in S' originate at the root. Assume that the number of servers in T is at least $|S \cup S'|$. Occasionally we will use T_i to represent the set of requests that originate in a subtree T_i of T (rather than the subtree itself) when the context makes this clear. Let n_i be the number of servers in the i^{th} subtree T_i of T, and let $m^* = \sum_i \max(|S \cap T_i| - n_i, 0)$.

Fact 4.2.1. *In any assignment of requests in $S \cup S'$ to servers, at least $m^* + |S'|$ requests use top-level edges.*

Proof. The number of requests that originate in a subtree T_i is $|S \cap T_i|$, so $|S \cap T_i| - n_i$ represents the number of requests that originate in T_i and must assign to servers outside of T_i, and hence, must use a top-level edge. The maximum of this quantity over all subtrees is therefore a lower bound on the number of requests that use top-level edges. □

Let M be the random variable denoting the number of requests in $S \cup S'$ that use a top-level edge when assigned by the algorithm Random-Subtree.

Lemma 4.2.2 (One-Level Lemma).

$$E[M] \leq H_d \cdot (m^* + |S'|).$$

Proof. Let the k requests $S \cup S'$ be labeled $r_1, r_2, \cdots, r_k$, where r_1 is the earliest request and r_k is the last request.

We define time t to be the instant just before request r_t is assigned, so that $t = 1$ represents the time before any request assignments have been made, and $t = k + 1$ represents the the time after all request assignments have been made. Let $R_t = \{r_t, r_{t+1}, \cdots, r_k\}$, the set of requests at time t that have yet to arrive. At time t,

let $n_{i,t}$ be the number of available servers in tree T_i. A subtree T_i is said to be open at time t if $n_{i,t} > 0$ (there are available servers at time t in T_i). Let α_t be the number of open subtrees of T at time t. Define the first $\min(n_{i,t}, |R_t \cap T_i|)$ requests to be the local requests of T_i at time t (these are the ones in T_i that have the highest numbered indices), and the remaining requests in T_i to be the global requests of T_i at time t; let $\mathcal{L}_{i,t}$ and $\mathcal{G}_{i,t}$ be the set of local and global requests in T_i at time t, and let $\mathcal{L}_t = \cup_i \mathcal{L}_{i,t}$ and $\mathcal{G}_t = \cup_i \mathcal{G}_{i,t}$. All requests in $R_t \cap S'$ are called root requests of T at time t, and form the set $\mathcal{R}_t$.

Remark 4.2.3. At time $t = 1$, $R_1 = S \cup S'$, $n_{i,1} = n_i$, and the number of global requests in T_i is $|\mathcal{G}_{i,1}| = \max(|S \cap T_i| - n_i, 0)$, and the number of root requests is $|\mathcal{R}_{i,1}| = |S'|$. The total number of global requests at time 1 is m^*.

4.2.1 Cost Functions F_t

We will maintain functions $F_t : R_t \to \mathbb{Z}_{\geq 0}$; such a function F_t is called well-formed if it satisfies the following properties:

- $F_t(r_j) = 0$ if and only if $r_j \in \mathcal{L}_t$ (i.e., it is a local request at time t), and
- for all global and root requests $r_j \in \mathcal{G}_t \cup \mathcal{R}_t$, $F_t(r_j)$ is an upper bound on α_j (the number of open subtrees at time instant j) with probability 1.

To ensure that our functions are well-formed initially at time 1, we set $F_1(r_j) = d$ (the degree of the tree) for all $r_j \in \mathcal{G}_1 \cup \mathcal{R}_1$ (global and root requests at time 1), and $F_1(r_j) = 0$ for all $r_j \in \mathcal{L}_1$ (local requests at time 1). It is immediate that the map F_1 is well-formed. We will define a (well-formed) map F_t for every time instant t, as described in the following discussion.

Consider time instant t, and suppose that the well-formed map F_t has been defined. If $r_t \in \mathcal{L}_t$, then define $F_{t+1}(r_j) = F_t(r_j)$ for all $r_j \in R_t$. Note that if a request at time t is a local/global/root request, then it is still a local/global/root request at time $t+1$, so it follows that F_{t+1} is still well-formed.

Now suppose $r_t \in \mathcal{G}_t \cup \mathcal{R}_t$—it is a global or root request. For convenience, we say that a request r_j "becomes global" at time t if r_j is local at time $t-1$, but r_j is global at time t. Recall there are α_t open subtrees at time t; moreover, since r_t is a global request, its own subtree is not open at time t. For each open subtree T_i, there are $|R_t \cap T_i|$ requests and $n_{i,t}$ free servers in it, so if $|R_t \cap T_i| \geq n_{i,t}$ then assigning r_t to a server in this subtree would cause some request r_j in $R_t \cap T_i$ that is local at time t to become global at time $t+1$ (because $n_{i,t-1}$ would become $n_{i,t} - 1$). Let $a_t(T_i) = j$, so that $a_t(T_i)$ is the index of the request r_j that turns global in subtree T_i; if there is no such request, set $a_t(T_i) = k + i$. Let $A_t = \{a_t(T_i) \mid T_i \text{ open at time } t\}$; note that $|A_t| = \alpha_t$. Now denote the elements of A_t by $\{p_j\}_{j=1}^{\alpha_t}$ such that $p_1 < p_2 < \cdots < p_{\alpha_t}$. Note that each p_j corresponds to $a_t(T_i)$ for some subtree T_i.

Remark 4.2.4. Note that $p_{\alpha_t} > k$; indeed, since the total number of requests in $S \cup S'$ is at most the total number of servers. If r_t is global, then the subtree containing r_t has no available servers but has at least one request (namely r_t), which must assign to some open subtree T_i which has more available servers than requests. Alternatively, if r_t is a root request, then there must exist some there must be at least one open subtree T_i which has strictly more available servers than requests—for this subtree T_i, the corresponding $a_t(T_i)$ is greater than k.

Now, let r_t get randomly assigned to a server in one of the open subtrees, say in subtree T_i. We now need to define the map F_{t+1}. There are two cases to consider:

- If $a_t(T_i) > k$ (i.e., none of the requests in $T_i \cap R_{t+1}$ become global at time t), then we set $F_{t+1}(r) = F_t(r)$ for all requests $r \in R_{t+1}$.
- If $a_t(T_i) \leq k$, then say $a_t(T_i) = p_{\alpha(t)-q+1}$ in the ordering given above (i.e., $a_t(T_i)$ was the q^{th} largest value in A_t). Now assign $F_{t+1}(r) = F_t(r)$ for all $r \in R_{t+1} \setminus \{r_{a_t(T_i)}\}$, and $F_{t+1}(r_{a_t(T_i)}) = q - 1$.

Lemma 4.2.5. *The map F_{t+1} is well-formed.*

Proof. By induction, the map F_t was well-formed. In the first case when r_t is local, since the map remains unchanged on R_{t+1}, and so do the set of local/global/root requests in R_{t+1}, the claim follows.

When r_t is a global or root request and mapped into T_i, if none of the requests in $T_i \cap L_{t-1}$ become global due to this change, the well-formedness of F_{t+1} follows again. So, let's consider the case where the request $r_j \in T_i$ becomes global because of r_t. We previously defined that $j = a_t(T_i)$, and that j is the q^{th} largest of the sequence of A_t. Moreover, since r_j is mapped by F_{t+1} to an integer $q \leq k$, it suffices to show that at most $q - 1$ subtrees will be open at time j. Indeed, we claim that for any subtree T_h with $a_t(T_h) < a_t(T_i) = j$, there will be no servers available in T_h at time j. To see this, note that since $a_t(T_h) < k$, there must be some request that becomes global if r_t assigns in T_h. Thus, the number of requests in T_h that had indices smaller than j (and hence arrive before r_j) was equal at time t to the number of available servers in T_h, and hence these requests alone would cause T_h to be closed. Moreover, for subtree T_i, the fact that r_j becomes global at time t means that T_i will also be closed at time j. Hence, the only open subtrees at time j would be the subtrees T_ℓ which were open at time t, and which had $a_t(T_\ell) > j$. There are at most $q - 1$ of such subtrees T_ℓ, since j is the q^{th} largest of the sequence. This shows that F_{t+1} is well-formed. □

Observe that the changes to the map F_t over time are very simple: maps F_t and F_{t+1} differ in at most one request from R_{t+1}. Moreover, this difference is when some local request r_j becomes global at time t, and hence is mapped to some positive integer value instead of to zero. The value $F_{t'}(l_j)$ then remains unchanged thenceforth (for times $t' = t + 1, t + 2, \ldots, j$).

4.2.2 Potential Function Analysis

Finally, let us define a potential function:

$$\Phi_t = \sum_{r \in R_t} H_{F_t(r)},$$

where we consider $H_0 = 0$. Also, define ρ_t to be the number of requests that our algorithm has matched outside their subtrees at time t.

Lemma 4.2.6. *For all* $1 \leq t \leq k+1$, $E[\Phi_t + \rho_t] \leq H_d \cdot (m^* + |S'|)$.

Proof. We prove this by induction on time t. The base case is when $t = 1$. Then $\rho_1 = 0$; the number of global/root requests is $m^* + |S'|$, and since each such request r has $F_1(r) = d$, we get that $\Phi_1 = H_d \cdot (m^* + |S'|)$.

Inductively assume the claim is true at time t. Thus, $E[\Phi_t + \rho_t] \leq H_d \cdot (m^* + |S'|)$. We want to show the same at time $t+1$, right after r_t has been assigned. We claim that

$$E[\Phi_{t+1} + \rho_{t+1}] \leq \Phi_t + \rho_t,$$

which will complete the proof. There are two cases:

- Suppose r_t is a local request: its subtree contains an unassigned server, so $\rho_{t+1} = \rho_t$. Moreover, $F_t(r_t) = 0$ by the well-formed property, so $\Phi_{t+1} = \Phi_t$.
- Suppose r_t is a global request, and gets assigned to subtree T_i. In this case, $\rho_{t+1} = \rho_t + 1$. Now consider $E[\Phi_t - \Phi_{t+1}]$. This is

  $$H_{F_t(l_t)} - \frac{1}{\alpha_t} \sum_{j=0}^{\alpha_t - 1} H_j \geq 1$$

  since $F_t(r_t) \geq \alpha_t$ by the well-formedness of map F_t, and $H_m - \frac{1}{m} \sum_{j=0}^{m-1} H_j = 1$.

Hence, in both cases, conditioned on everything that happened before time t, the value $E[\Phi_{t+1} + \rho_{t+1}] \leq \Phi_t + \rho_t$, where the expectation is taken over the random choices of l_t. This completes the induction, and the proof of the lemma. □

Since $\rho_{k+1} = M$, and $\Phi_{k+1} = 0$, this finishes the proof of the one-level lemma. □

4.3 Bounding the Total Cost

Given the one-level lemma with parameter γ, we can now show the following result for α-HSTs:

Lemma 4.3.1. *Suppose T is an α-HST rooted at r. For any set S of requests at the leaves of T, and requests S' at the root of T, such that $|S \cup S'|$ is at most the number of servers in T. If we let* $\mathrm{Alg}(R,T)$ *be the cost of* Random-Subtree *on the set of requests R on the tree T, and* $\mathrm{Opt}(R,T)$ *the cost of the optimal solution, it holds that*

$$E[\mathrm{Alg}(S \cup S', T)] \leq c\gamma \,\mathrm{Opt}(S \cup S', T)$$

for $c = 2(1 + 1/\epsilon)$ as long as $\alpha \geq \max(2, (1+\epsilon)\gamma)$.

Proof. We prove this by induction on the depth of the HST. The base case of this problem is implied by the one-level lemma on a star (say of unit edge lengths); note that the algorithm incurs a cost of $2M$, whereas $\mathrm{Opt}(S \cup S', T) = |S'| + 2m^* \geq |S'| + m^*$. Hence we get that $E[\mathrm{Alg}(S \cup S', T)] = E[2M] \leq 2H_d \cdot \mathrm{Opt}(S \cup S', T) \leq c\gamma \,\mathrm{Opt}(S \cup S', T)$. Recall that M is the number of requests in $S \cup S'$ that in our algorithm's matching use top-level edges, and m^* is the number of requests in S that use top-level edges in the optimal matching.

For the inductive step, let us prove the claim for an α-HST T under the assumption it inductively holds for all the α-HSTs T_i that are the subtrees of the root. Let the length of edges incident to the root be 1, the length of their children be $1/\alpha$, etc. Let the length of the path from the root to a leaf in T be $(1 + \beta)$, which implies that $\beta \leq \frac{1}{\alpha - 1}$. Let n_i be the number of servers in a subtree T_i of T.

Consider the optimal matching Opt, and define the following quantities:

- Let Γ_i^* be the requests originating in T_i that Opt matches outside T_i (call these the Opt-global clients), and let $m_i^* = |\Gamma_i^*|$,
- Let Λ_i^* be the requests in T_i that Opt satisfies with servers in T_i (these are the Opt-local clients)
- Let $S_i = \Lambda_i^* \cup \Gamma_i^*$, and note that $S = \cup_i S_i$.

Fact 4.3.2 (Optimal Cost)**.**

$$\mathrm{Opt}(S \cup S', T) = \sum_i \mathrm{Opt}(\Lambda_i^*, T_i) + \sum_i m_i^* \cdot 2(1+\beta) + |S'| \cdot (1+\beta).$$

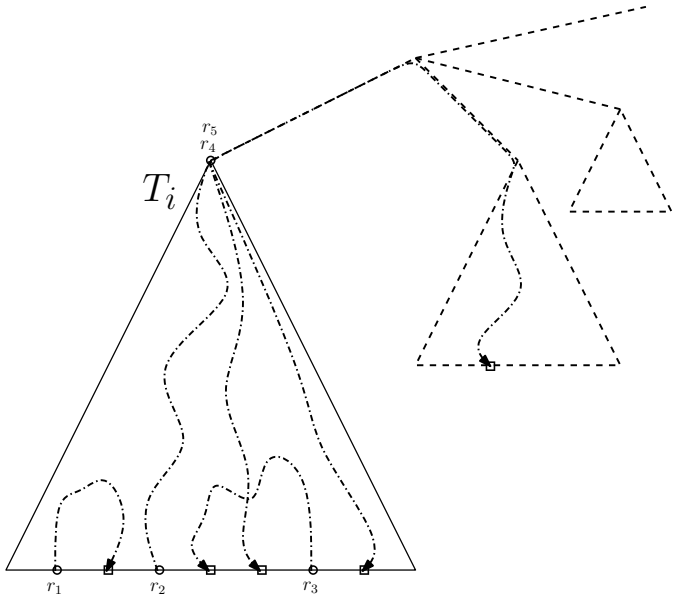

Figure 4.1: If the dotted lines represent the assignments that Opt makes within T_i, then $\Lambda_i^* = \{r_1, r_3\}$, $\Gamma_i^* = \{r_2\}$, and the set S' of T_i is $\{r_4, r_5\}$.

Proof. We can partition the set $S \cup S'$ into the three types of subsets Λ_i^*, Γ_i^*, and S'. For each request r in Λ_i^* we note that the optimal cost of assigning r is determined by $\mathrm{Opt}(\Lambda_i^*, T_i)$, since r's server must be in T_i. For every request in Γ_i^*, we must assign from some subtree T_i to T_j by using a top-level edge. Thus, we simply pay twice the length from the root to a leaf for each request in Γ_i^*, which can be expressed as $2(1+\beta)m_i^*$ for each subtree T_i. Finally, the requests that begin at the root (of which there are $|S'|$ many) will pay the length of the root to a leaf, which is exactly $1+\beta$. □

Now, let M_i be the set of requests originating outside T_i (but possibly at the root of T_i) that the algorithm satisfies by assigning into T_i. Look at $S_i \cup M_i$—these are all the requests that the subtree T_i encounters, and let $X_i = \widehat{S_i} \cup \widehat{M_i}$ be the first n_i of these requests which can be satisfied within the subtree T_i. (Note that the sets M_i, X_i, $\widehat{S_i}$, and $\widehat{M_i}$ are all random variables.)

Fact 4.3.3.

$$E[\mathrm{Alg}(S \cup S', T)] = \sum_i E[\mathrm{Alg}(\widehat{S_i} \cup \widehat{M_i}, T_i)] + \sum_i E[|M_i|] \cdot (2+\beta).$$

Proof. Suppose r is some request in T_i that Alg assigned to some server in $T_j \neq T_i$. We account for this assignment's cost by breaking the path from r to s into two parts.

The initial part, accounted for by the latter term of the equation, includes the edges used from r to the root along with both edges incident to the root. The path from r to the root is of length $\beta + 1$, and the additional edge incident to the root is of length 1, giving us $\beta + 2$. Since there are $|M_i|$ such requests for each subtree T_i, we see that the second term covers all of the initial parts of the paths of each global request.

The reason for the above convention is that now that we have covered all outgoing requests, we can imagine all global incoming requests as having originated at the root of the tree, since their inital parts have already been accounted for. Therefore, this quantity can be described as $\mathrm{Alg}(\widehat{S}_i \cup \widehat{M}_i, T_i)$ for each subtree T_i. □

By our inductive assumption we know that for any $\widehat{S}_i$ and $\widehat{M}_i$ defined for a tree T_i,

$$E[\mathrm{Alg}(\widehat{S}_i \cup \widehat{M}_i, T_i)] \leq c\,\gamma\ \mathrm{Opt}(\widehat{S}_i \cup \widehat{M}_i, T_i). \tag{4.1}$$

Fact 4.3.4.

$$\mathrm{Opt}(\widehat{S}_i \cup \widehat{M}_i, T_i) \leq \mathrm{Opt}(\Lambda_i^*, T_i) + m_i^* \cdot 2\beta + |M_i| \cdot \beta.$$

Proof. To bound Opt's cost on $\widehat{S}_i \cup \widehat{M}_i$, we imagine the requests in $\widehat{S}_i \cap \Lambda_i^*$ being sent according to where $\mathrm{Opt}(\Lambda_i^*, T_i)$ sent them, and the remaining requests being assigned arbitrarily to the remaining servers. The former cost is upper bounded by $\mathrm{Opt}(\Lambda_i^*, T_i)$. For the latter term, there are $|\widehat{S}_i \cap \Gamma_i^*| \leq |\Gamma_i^*| = m_i^*$ requests which incur a cost of at most 2β (since they go from some leaf to another within the fixed subtree T_i), and the remaining requests—at most M_i of them—incur a cost of at most β (since they go from the root of T_i to a leaf). □

Using Facts 4.3.3 and 4.3.4 with (4.1), we get

$$E[\mathrm{Alg}(S \cup S', T)] = c\,\gamma\ \sum_i \big(\mathrm{Opt}(\Lambda_i^*, T_i) + m_i^* \cdot 2\beta + E[|M_i|] \cdot \beta\big) + \sum_i E[|M_i|] \cdot (2+\beta).$$

Comparing this expression with Fact 4.3.2 (and cancelling the $\mathrm{Opt}(\Gamma_i^*, T_i)$ terms), it suffices to show that

$$\sum_i c\,\gamma\ \big(m_i^* \cdot 2\beta + E[|M_i|] \cdot \beta\big) + E[|M_i|] \cdot (2+\beta) \leq \sum_i c\,\gamma\ (2m_i^* + |S'|)(1+\beta).$$

Finally, we can use the one-level lemma to claim that

$$E[|M_i|] \leq \gamma \cdot (\sum_i m_i^* + |S'|).$$

Using this, abbreviating $m^* = \sum_i m_i^*$ and $s' = |S'|$, and cancelling γ throughout, it suffices to show that

$$c\,m^*\,2\beta + (m^* + s')(c\gamma\beta + (2+\beta)) \leq c\ (2m^* + s')(1+\beta).$$

Or equivalently, it suffices to choose c such that

$$c \geq \frac{(m^* + s')(2+\beta)}{(2m^* + s')(1+\beta) - 2\beta\, m^* - (m^* + s')\gamma\beta}$$

as long as the expression in the denominator is positive. But the expression on the right is

$$\frac{m^*(2/\beta + 1) + s'(2/\beta + 1)}{m^*(2/\beta - \gamma) + s'(1/\beta + 1 - \gamma)} \leq \max\left(\frac{2/\beta+1}{2/\beta-\gamma}, \frac{2/\beta+1}{1/\beta+1-\gamma}\right),$$

so as long the greater of $\frac{2/\beta+1}{2/\beta-\gamma}$ and $\frac{2/\beta+1}{1/\beta+1-\gamma}$ is bounded above by c, we are in good shape. If $\alpha \geq 2$, then $1/\beta \geq \alpha - 1 \geq 1$ and the latter expression is the larger one, so we can focus on that. But that expression is $\frac{2\alpha-1}{\alpha-\gamma}$, which is bounded above by 4 if $\alpha \geq 2\gamma$. In general, we could set $\alpha = (1+\epsilon)\gamma$, in which case we could set $c = 2(1 + \frac{1}{\epsilon})$. The following is a proof of this algebraic manipulation for completeness:

Lemma 4.3.5. *For $1/\beta \geq \alpha - 1$, we have that*

$$\frac{2/\beta+1}{1/\beta+1-\gamma} \leq c.$$

Proof. Plugging in $c = 2(1 + \frac{1}{\epsilon})$ and $\alpha = (1+\epsilon)\gamma$ we get:

$$\begin{aligned} c &\geq 2/\epsilon + 2 - 1/(\epsilon\gamma) \\ c &\geq \frac{2\gamma + 2\epsilon\gamma - 1}{\epsilon\gamma} \\ 2(1+\epsilon)\gamma - 1 &\leq c\epsilon\gamma \\ (2\alpha - 1) &\leq c(\alpha - \gamma) \\ \frac{2\alpha-1}{\alpha-\gamma} &\leq c \end{aligned}$$

Now, note that $f(x) = \frac{2x-1}{x-\gamma}$ is a decreasing function, so we have that $\alpha \leq 1/\beta + 1$ implies that $f(\alpha) \geq f(1/\beta + 1)$. Since $c \geq f(\alpha)$, then $c \geq f(1/\beta + 1)$, and so

$$\frac{2(1/\beta+1)-1}{1/\beta+1-\gamma} \leq c.$$

□

This completes the proof of the bound on the expected cost of Random-Subtree. □

Note that in the worst case, the HST is just the star graph, in which case $d = k$. Thus, we have that $d \leq k$. Since our results imply $O(\log^2 d \log k)$, then Random-Subtree must be at least as good as the MNP algorithm for general metric spaces. The analysis here allows us to conclude that Random-Subtree achieves $O(\log k)$ competitiveness on the line, and $O(\log k)$ competitiveness for any metric space for which there exists a constant-degree HST construction with $O(\log k)$ stretch. Now, any improvement upon $O(\log k)$ competitiveness for the line would require an algorithm that does not construct an HST, since the inevitable conversion back to the line would incur an $O(\log k)$ term already. However, in the next chapter we see a new, deterministic algorithm on HSTs derived from the line that achieve $O(1)$ competitiveness on the HST.

Chapter 5

An $O(\log k)$ Algorithm for the Line

The two algorithms in the previous chapters that we proposed for the online metric matching problem, as well as the poly$(\log k)$ algorithms presented in [MNP06] and [BBGN07], are algorithms that perform an HST construction, run a simple procedure on the HST, and map the matching back to their original metrics while incurring an additional $O(\log k)$ cost, obliviously. The problem with this approach is that we compare the algorithm's cost on the HST with the optimal cost on the HST, and then we apply the results of [FRT03] immediately to get a competitive ratio for the original metric. In the following analysis, we choose to ignore the cost of the algorithm's matching on the HST entirely. Rather, we compare the cost of our matching using the original metric's distance function with the cost of our matching with the HST's distance function.

However, the fact that we are now comparing costs between two different metric spaces makes the analysis more difficult. Our proof technique involves the analysis of a hybrid algorithm—one that makes arbitrary assignments for the first i requests, and then runs our competitive algorithm for the remaining requests. We compare this to a hybrid algorithm that makes the same arbitrary assignments for the first $i-1$ requests, and then runs our competitive algorithm on the remaining requests. We show through a careful analysis of the changes that occur between these two hybrid algorithms that the difference in cost is small enough. This allows us to conclude that the "hybrid" algorithm that makes arbitrary assignments on the first 0 requests and then runs our competitive algorithm on the remaining (all) of the requests performs well. In this chapter we also include an instance of the online metric matching problem on the line that shows that our analysis for this algorithm is tight.

5.1 The HST-greedy Algorithm

Consider the following algorithm for online minimum matching on the line. This deterministic algorithm takes a line L (on the set of servers S, with $|S| = k$), and a binary 2-HST T superimposed on it in the natural fashion as in the figure, such that distances d_T in the HST dominate the distances d_L along the line. Given a sequence of requests $\sigma = r_1, r_2, \ldots, r_k$ appearing online, the algorithm matches each request r_i to a distinct server $f(r_i)$ as follows: for the request r_i, let a_i denote the lowest ancestor of r_i in the tree such that the subtree $T(a_i)$ rooted at x_i contains a free server. Assign r_i to the free server in $T(a_i)$ that is closest to r_i along the line; this server is called $f(r_i)$. We call this the HST-greedy algorithm, and denote the matching produced by it on a request sequence σ as G_σ.

Theorem 5.1.1. *The cost (along the line) of this matching f is at most a constant times the cost (along the tree) incurred by any other matching f^*. I.e.,*

$$\sum_i d_L(r_i, f(r_i)) \leq O(1) \cdot \sum_i d_T(r_i, f^*(r_i)), \tag{5.1}$$

Combining this with the fact that we can choose this tree T from a probability distribution such that the expected distances in the tree are greater only by a factor of $O(\log k)$, we get:

Corollary 5.1.2. *The randomized algorithm that picks a random binary 2-HST for the line and runs the HST-greedy on this tree is an $O(\log k)$-competitive randomized online algorithm for metric matching on the line.*

In the subsequent discussion, we will find it useful to generalize the discussion to the case where we approximate the line using α-HSTs of maximum degree D. In this case we assume that the children of each node are numbered $1, 2, \ldots, D$ in the natural left-to-right order. In this case, we need to generalize the HST-greedy algorithm above "break ties consistently"; let us call this the generalized HST-greedy algorithm. In particular, on request r_i, let a_i again be the lowest node such that $T(a_i)$ contains a free server. Now let b_i be the lowest-numbered child of a_i such that $T(b_i)$ contains a free server; if the leaves of $T(b_i)$ lie to the left of r_i, send r_i to the closest free server to its left, else send it to the closest free server to its right. In the binary case, $T(b_i)$ is the subtree that does not contain r_i, and hence following the generalized algorithm just gives us the original HST-greedy algorithm for the binary case.

5.1.1 Analysis via a "Hybrid" Algorithm

To prove Theorem 5.1.1, first consider a "hybrid" algorithm that matches the request r_1 to an arbitrary server s_1, and then runs the HST-greedy algorithm on the remaining

requests in σ. Denote the matching produced to be H_σ; note that this matching is a function of the choice of s_1.

Lemma 5.1.3 (Hybrid Lemma)**.** *There is a $\lambda = O(D)$ such that for any set of servers S on the line, for any request sequence $\sigma = r_1, \cdots, r_k$, and for any choice of assignment $r_1 \to s_1$,*

$$\sum_i d_L(r_i, G_\sigma(r_i)) \leq \sum_i d_L(r_i, H_\sigma(r_i)) + \lambda\, d_T(r_1, s_1), \tag{5.2}$$

Note that if G were the optimal matching on the line, and H would match $r_1 \to s_1$ and then find the optimal matching on the remaining requests, such a claim is easily seen to be true with additive error $2d_L(r_1, s_1)$. Here we show that even the HST-greedy algorithm satisfies such a property with $O(1)\, d_T(r_1, s_1)$.

Also, before we prove this lemma, let us use it to prove the theorem. Given any request sequence σ and matching f^*, we can define a sequence of hybrid algorithms $\{H^t\}_{t=0}^k$, where H^t matches the first t requests r_i in σ to $f^*(r_i)$, and then runs the HST-greedy algorithm on the remaining requests. Note that H^0 is just the HST-greedy algorithm, and H^k produces the matching f^*. Moreover, by ignoring the servers in $\{f^*(r_i) \mid i \leq t\}$, just considering the request subsequence $r_{t+1}, \ldots, r_k$, we can use the lemma to claim that

$$\sum_{i=t+1}^{k} d_L(r_i, H^t(r_i)) \leq \sum_{i=t+1}^{k} d_L(r_i, H^{t+1}(r_i)) + \lambda \cdot d_T(r_t, f^*(r_t)),$$

or by adding $\sum_{i=1}^{t} d_L(r_i, f^*(r_i))$ to both sides,

$$\sum_{i=1}^{k} d_L(r_i, H^t(r_i)) \leq \sum_{i=1}^{k} d_L(r_i, H^{t+1}(r_i)) + \lambda \cdot d_T(r_t, f^*(r_t)).$$

Now, this summing this for all values of t, and using that $H^0 = G$ and $H^k = f^*$, we get

$$\sum_{i=1}^{k} d_L(r_i, G(r_i)) \leq \sum_{i=1}^{k} d_L(r_i, f^*(r_i)) + \lambda \cdot \sum_i d_T(r_t, f^*(r_t)).$$

Finally, since $d_L \leq d_T$, we get that

$$\sum_{i=1}^{k} d_L(r_i, G(r_i)) \leq (\lambda + 1) \cdot \sum_i d_T(r_t, f^*(r_t)),$$

and hence Theorem 5.1.1 follows.

Let us now proceed with the proof of Lemma 5.1.3. We first make a few simple claims relating the behavior of the algorithms G and H given any initial set of servers S and a request sequence σ.

Firstly, if there are any requests r_i such that $G_\sigma(r_i) = H_\sigma(r_i)$, we can delete the request r_i from σ, and delete the server $G_\sigma(r_i)$ from S, to get another server set and sequence with the same behavior; hence we will assume for the rest of the section that for each $r_i \in \sigma$, $G_\sigma(r_i) \neq H_\sigma(r_i)$.

5.1.2 Defining the Cavities

Lemma 5.1.4. *If the set of available servers in G's run and H's run just after request r_t has been satisfied is denoted by $A_G(t)$ and $A_H(t)$ respectively, then either $A_G(t) = A_H(t)$, or $|A_G(t) \setminus A_H(t)| = 1 = |A_H(t) \setminus A_G(t)|$.*

Proof. Suppose $G_\sigma(r_1) = x_1$; recall that $H_\sigma(r_1) = s_1$, and by the above observation, $x_1 \neq s_1$. Hence, $A_G(1) \setminus A_H(1) = \{s_1\}$, whereas $A_H(1) \setminus A_G(1) = \{x_1\}$. Let us call the former a "G-cavity" and the latter an "H-cavity". Now, inductively assume the claim is true just before assigning r_t. If $A_G(t-1) = A_H(t-1)$, then the claim is trivially true from then on, so assume there is a unique G-cavity g_{t-1} and H-cavity h_{t-1}. Let $H_\sigma(r_t) = s_t$ and $G_\sigma(r_t) = x_t$. There are some cases to consider:

1. If $x_t = \mathsf{g}_{t-1}$ and $s_t = \mathsf{h}_{t-1}$, then $A_G(t) = A_H(t)$.

2. If $x_t \neq \mathsf{g}_{t-1}$ and $s_t = \mathsf{h}_{t-1}$, then it follows that $A_H(t) \setminus A_G(t) = \{x_t\}$, whereas $A_G(t) \setminus A_H(t) = \{\mathsf{g}_{t-1}\}$—i.e., $\mathsf{g}_t = \mathsf{g}_{t-1}$ but $\mathsf{h}_t = x_t$.

3. If $x_t = \mathsf{g}_{t-1}$ and $s_t \neq \mathsf{h}_{t-1}$, then it follows that $A_H(t) \setminus A_G(t) = \{\mathsf{h}_{t-1}\}$, whereas $A_G(t) \setminus A_H(t) = \{s_t\}$—i.e., $\mathsf{h}_t = \mathsf{h}_{t-1}$ but $\mathsf{g}_t = s_t$.

4. Finally, we claim that the case $x_t \neq \mathsf{g}_{t-1}$ and $s_t \neq \mathsf{h}_{t-1}$ must imply that $x_t = s_t$. Indeed, say that the lowest ancestors considered by HST-greedy when assigning r_t in the two runs are a^G and a^H respectively. If these are not identical, say a^G is lower: then $T(a^G)$ contains a server in G's run but not in H's, but since the only additional free server in $A_G(t-1)$ is g_{t-1}, we would get $x_t = \mathsf{g}_{t-1}$ and a contradiction; a similar analysis shows that a^H cannot be lower. Hence $a^G = a^H = a$, say. Now if r_t is assigned to the first free server it encounters within $T(a)$ (by first scanning to the left of r_t for a free server in $T(a)$, then scanning to the right) in both runs, and neither is assigned to the G-cavity or the H-cavity, then $x_t = s_t$, which contradicts the assumption that $G(r_t) \neq H(r_t)$. Hence, $\mathsf{g}_t = \mathsf{g}_{t-1}$ and $\mathsf{h}_t = \mathsf{h}_{t-1}$, and so $A_G(t) \setminus A_H(t) = \{\mathsf{g}_t\}$ and $A_H(t) \setminus A_G(t) = \{\mathsf{h}_t\}$.

Another way to view the above lemma is to consider the symmetric difference $G_\sigma \Delta H_\sigma$ of the two matchings, and to claim that this is a single path or cycle. We

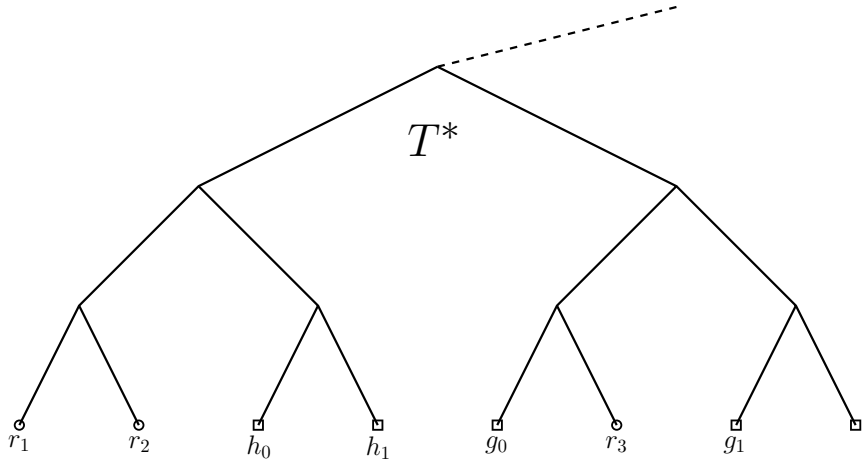

Figure 5.1: In this small example, suppose H is the matching $(r_1, \mathsf{g}_0), (r_2, \mathsf{h}_0), (r_3, \mathsf{g}_1)$, and G initially matches r_1 to h_0. Then, as r_2 arrives in G's run, it will be assigned to h_1, the new H-cavity, and when r_3 arrives, it will be assigned to g_0, causing g_1 to be the new G-cavity.

start off with two edges $(r_1, s_1), (r_1, x_1)$; each subsequent time we place down two edges adjacent to r_t, and these extend the path (in cases 2 and 3) until we close a cycle (as in case 1, when both the G-cavity and H-cavity disappear), at which time the process stops. □

In the rest of the argument, we define g_t to be the unique G-cavity in $A_G(t) \setminus A_H(t)$, and h_t to be the unique H-cavity in $A_H(t) \setminus A_G(t)$. Moreover, since we are interested in the difference between costs incurred by G and H respectively, we can assume that $A_G(t) \neq A_H(t)$ for all time $t < k$, and hence that g_t and h_t are defined for all times $t \in \{1, 2, \ldots, k-1\}$.

Moreover we will be relating the runs of G and H, so some jargon will be useful to avoid confusion. When we refer to server(s) in G's run, we call them G-servers. The request r_t is assigned at time t, and we refer to the situation just before this assignment as being at time t^-, and just after this assignment as being at time t^+; note that $(t-1)^+ = t^-$.

Lemma 5.1.5. *Suppose a^* is the least common ancestor of (r_1, s_1) in T. Then, for all times $t < k$, $\{\mathsf{g}_t, \mathsf{h}_t\} \subseteq T(a^*)$, the subtree rooted at a^*. Henceforth, let us denote the subtree $T(a^*)$ as T^*.*

Proof. To begin, $\mathsf{g}_1 = s_1$, and hence in $T(a^*)$. Also, $\mathsf{h}_1 = x_1$ is chosen by the HST-greedy, and must lie in the lowest subtree containing both r_1 and a free server; hence this is a (not necessarily proper) subtree of $T(a^*)$. Let t be such that $\{\mathsf{g}_t, \mathsf{h}_t\} \not\subseteq T(a^*)$

but $\{\mathsf{g}_{t-1}, \mathsf{h}_{t-1}\} \subseteq T(a^*)$, and r_t the request at time t. Since r_t assigns to a unique server, it cannot move both g_t and h_t out of $T(a^*)$ at the same time. Suppose $\mathsf{g}_t \in T(a^*)$ and $\mathsf{h}_t \notin T(a^*)$. Then the number of available servers in $T(a^*)$ at time t^- in G's run is not equal to the number of available servers at time t^- in H's run. The same holds true if $\mathsf{g}_t \notin T(a^*)$ and $\mathsf{h}_t \in T(a^*)$.

By the Induction Hypothesis, $\{\mathsf{g}_i, \mathsf{h}_i\} \in T(a^*)$ for all $i < t$. Therefore, all assignments have been made within $T(a^*)$, and so the number of free servers is the same between G's run and H's run at time t^-. This contradiction establishes the desired claim. □

5.1.3 An Accounting Scheme

The "hybrid lemma" Lemma 5.1.3 showed that it suffices to bound the difference in cost incurred by HST-greedy G on a sequence, and the cost incurred by the hybrid algorithm that assigned $r_1 \to s_1$ and used HST-greedy from that point onwards. Now we show that this difference is costs can be measured merely in terms of the line distances traveled by the cavities.

Lemma 5.1.6 (Accounting Lemma)**.**

$$\sum_{t \le k} d_L(r_t, G_\sigma(r_t)) - \sum_{t \le k} d_L(r_t, H_\sigma(r_t)) \le 2 \sum_{t=2}^{k-1} d_L(\mathsf{g}_{t-1}, \mathsf{g}_t) + 2 \sum_{t=2}^{k-1} d_L(\mathsf{h}_{t-1}, \mathsf{h}_t) + 2\, d_T(r_1, s_1),$$

i.e., twice the distance traveled by the G-cavities and H-cavities, plus twice the distance $r_1 \to s_1$.

Proof. First, we consider the cases where $t > 1$. By the triangle inequality, we get that for any t,

$$d_L(r_t, G_\sigma(r_t)) - d_L(r_t, H_\sigma(r_t)) \le d_L(G_\sigma(r_t), H_\sigma(r_t)).$$

If $G_\sigma(r_t) = H_\sigma(r_t)$, then $d_L(r_t, G_\sigma(r_t)) - d_L(r_t, H_\sigma(r_t)) = 0$, so we assume that $G_\sigma(r_t) \neq H_\sigma(r_t)$. There are three cases: either $G_\sigma(r_t)$ is not available for request r_t in H's run, or $H_\sigma(r_t)$ is not available for r_t in G's run, or both.

- If $G_\sigma(r_t)$ is not available for request r_t in H's run, then by Lemma 5.1.4, $G_\sigma(r_t) = \mathsf{g}_{t-1}$. But then $\mathsf{g}_t = H_\sigma(r_t)$, and so we have that $d_L(G_\sigma(r_t), H_\sigma(r_t)) = d_L(\mathsf{g}_{t-1}, \mathsf{g}_t)$.
- If $H_\sigma(r_t)$ is not available for request r_t in G's run, then again Lemma 5.1.4 implies $H_\sigma(r_t) = \mathsf{h}_{t-1}$. But now $\mathsf{h}_t = G_\sigma(r_t)$, and so we get $d_L(G_\sigma(r_t), H_\sigma(r_t)) = d_L(\mathsf{h}_{t-1}, \mathsf{h}_t)$.

- Finally, if both happen, then $H_\sigma(r_t) = \mathsf{h}_{t-1}$ and $G_\sigma(r_t) = \mathsf{g}_{t-1}$. Thus $A_G(t) = A_H(t)$, so this must be time k. Now, $d_L(G_\sigma(r_k), H_\sigma(r_k)) = d_L(\mathsf{g}_{k-1}, \mathsf{h}_{k-1})$, which in turn is at most

$$\sum_{t<k} \big(d_L(\mathsf{g}_{t-1}, \mathsf{g}_t) + d_L(\mathsf{h}_{t-1}, \mathsf{h}_t)\big) + d_L(r_1, \mathsf{g}_1) + d_L(r_1, \mathsf{h}_1)$$

  Note that $\mathsf{h}_1 = G_\sigma(r_1)$ and $\mathsf{g}_1 = H_\sigma(r_1)$.

Adding all these cases for $t > 1$, we get that $\sum_{t=2}^{k} d_L(r_t, G_\sigma(r_t)) - d_L(r_t, H_\sigma(r_t))$ is at most

$$\sum_{t=2}^{k-1} 2\big(d_L(\mathsf{g}_{t-1}, \mathsf{g}_t) + d_L(\mathsf{h}_{t-1}, \mathsf{h}_t)\big) + d_L(r_1, H_\sigma(r_1)) + d_L(r_1, G_\sigma(r_1))$$

Finally, adding in $d_L(r_1, G_\sigma(r_1)) - d_L(r_1, H_\sigma(r_1))$ to both sides, and noting that

$$2d_L(r_1, G_\sigma(r_1)) \le 2d_T(r_1, G_\sigma(r_1)) \le 2d_T(r_1, H_\sigma(r_1)),$$

we get

$$\sum_{t=1}^{k} d_L(r_t, G_\sigma(r_t)) - d_L(r_t, H_\sigma(r_t)) \le \sum_{t=2}^{k-1} 2\big(d_L(\mathsf{g}_{t-1}, \mathsf{g}_t) + d_L(\mathsf{h}_{t-1}, \mathsf{h}_t)\big) + 2\, d_T(r_1, H_\sigma(r_1)),$$

which completes the proof. □

5.1.4 Distance Traveled by the Cavities

Given the "accounting lemma" Lemma 5.1.6, it suffices to merely bound the total distance traveled by the cavities; in this section we show that this distance is proportional to the maximum distance between any two nodes of T^*, which is $O(2^{level(a^*)}) = O(d_T(r_1, s_1))$ and completes the proof. To do this, first let us define some useful concepts:

Definition 5.1.7 (Direction)**.** We define $\mathsf{dir}_\mathsf{g}(t)$, the direction of the G-cavity g_t, to be either L or R as follows: initially, we say that $\mathsf{dir}_\mathsf{g}(1)$ is L. For $t > 1$, if g_t is located to the left (respectively, right) of g_{t-1} on the line, then $\mathsf{dir}_\mathsf{g}(t) := L$ (respectively, R), and if $\mathsf{g}_t = \mathsf{g}_{t-1}$ then $\mathsf{dir}_\mathsf{g}(t) := \mathsf{dir}_\mathsf{g}(t-1)$.

Definition 5.1.8. Let $a_\mathsf{g}(t)$ be the least common ancestor of $\mathsf{g}_1, \cdots, \mathsf{g}_t$, and define $T_\mathsf{g}(t) = T(a_\mathsf{g}(t))$. Similarly, if $a_\mathsf{h}(t)$ is the least common ancestor of $\mathsf{h}_1, \cdots, \mathsf{h}_t$, then $T_\mathsf{h}(t) = T(a_\mathsf{h}(t))$.

Remark 5.1.9. If $u < t$, then $T_\mathsf{g}(u) \subseteq T_\mathsf{g}(t)$.

We now need more information on the servers that lie between g_{t-1} and g_t, and between h_{t-1} and h_t.

Lemma 5.1.10. *For $t > 1$, there are no free G-servers between g_{t-1} and g_t at time t^-. Likewise, there are no free H-servers between h_{t-1} and h_t at time t^-.*

Proof. Let r_t be the request that assigns to g_t in H's run and g_{t-1} in G's run. (If the G-cavity does not move, then the claim continues to hold, as no free servers are created and the direction does not change.) There are certainly no free G-servers between r_t and g_{t-1} at time t^- by the algorithm's run on G. Also, we know that there are no free H-servers between r_t and g_t at time t^-. The only free G-server that might be between r_t and g_t is g_{t-1}. If g_{t-1} is not between r_t and g_t, then the claim holds. Otherwise, even if g_{t-1} is between r_t and g_t, there are still no free G-servers between g_{t-1} and g_t at time t^-. An analogous proof holds for the H-servers.

□

Now, we will incorporate the "direction" in which g_t or h_t is moving into our proof. We are able to obtain some useful facts specifically at the points in time where the direction changes, as seen in the next lemma.

Lemma 5.1.11. *For $t > 1$, if $\mathsf{dir}_\mathsf{g}(t-1) = L$, $\mathsf{dir}_\mathsf{g}(t) = R$, and b is the child subtree of $a_\mathsf{g}(t)$ that contains g_t, then b does not contain g_{t-1}. Similarly, if $\mathsf{dir}_\mathsf{h}(t-1) = L$, $\mathsf{dir}_\mathsf{h}(t) = R$, and b is the child subtree of $a_\mathsf{h}(t)$ that contains h_t, then b does not contain h_{t-1}.*

Proof. We first show that for all $1 \le j \le t-2$, g_j must lie in between g_{t-1} and g_t. Assume for the sake of contradiction that there exists some j such that g_j is to the left of g_{t-1}. We know that g_{t-2} is to the right of g_{t-1} since $\mathsf{dir}_\mathsf{g}(t-1) = L$. Thus, by repeated applications of Lemma 5.1.10, there are no free G-servers between g_j and g_{t-2} at time $(t-2)^+$. However, this contradicts the fact that g_{t-1} is a free G-server at time $(t-2)^+$.

Assume for the sake of contradiction that there exists some j such that g_j is to the right of g_t. We know that g_{t-1} is to the left of g_t since $\mathsf{dir}_\mathsf{g}(t) = R$. Thus, by repeated applications of Lemma 5.1.10, there are no free G-servers between g_j and g_{t-1} at time $(t-1)^+$. However, this contradicts the fact that g_t is a free G-server at time $(t-1)^+$.

Thus, we have just shown that for all $1 \le j \le t-2$, g_j must lie in between g_{t-1} and g_t.

Now, let b be a child subtree of $a_\mathsf{g}(t)$ that contains g_t, and assume for the sake of contradiction that it also contains g_{t-1}. Then it must also contain g_j for all $1 \le j \le t-2$, and so b is a common ancestor of g_i for all $1 \le i \le t$. This contradicts the fact that $a_\mathsf{g}(t)$, the root of $T_\mathsf{g}(t)$, is supposed to be the least common ancestor of g_i for all $1 \le i \le t$, thus establishing the claim. An analogous proof holds for h_{t-1} under the assumptions involving h_t, a_h, and dir_h.

□

The previous lemma, which shows that a change in direction implies a change in the number of subtrees within the "convex hull" of the set of g_ts or h_ts, is now put to use in the next lemma.

Lemma 5.1.12. *If* $\mathsf{dir}_\mathsf{g}(t-1) = L$ *and* $\mathsf{dir}_\mathsf{g}(t) = R$*, then there are no free* G*-servers to the left of* g_t *in* $T_\mathsf{g}(t)$*. Likewise, if* $\mathsf{dir}_\mathsf{h}(t-1) = L$ *and* $\mathsf{dir}_\mathsf{h}(t) = R$*, then there are no free* G*-servers to the left of* h_t *in* $T_\mathsf{h}(t)$*.*

Proof. Let r_t be a request which causes the G-cavity to move from g_{t-1} to g_t. As the arguments in Lemma 5.1.4 show, the G-cavity moves from g_{t-1} to g_t because H assigns r_t to s_t, but G assigns r_t to g_{t-1}, which means that $\mathsf{g}_t := s_t$.

Because H runs the HST-greedy procedure after the first step, there are no free H-servers between r_t and $s_t = \mathsf{g}_t$ at time t^-, else r_t would be assigned to it. Hence, by Lemma 5.1.4, the only free G-server between r_t and $s_t = \mathsf{g}_t$ at this time t^- can be g_{t-1}. But $r_t \to \mathsf{g}_{t-1}$ in G, so there are also no free G-servers between r_t and g_t at time t^+.

Since $\mathsf{dir}_\mathsf{g}(t) = R$, we know that g_{t-1} is to the left of g_t. Let b be the child subtree of $a_\mathsf{g}(t)$ that contains g_t. By Lemma 5.1.11, b cannot contain g_{t-1}. Now, there are several cases to consider based upon the location of r_t with respect to g_t:

1. r_t is to the left of every point in $T_\mathsf{g}(t)$. Then we have already established that there are no free G-servers between r_t and g_t and time t^+. Since r_t is to the left of g_t, then there are no free G-servers within $T_\mathsf{g}(t)$ that are to the left of g_t at time t^+.

2. r_t is to the left of every point in b but within $T_\mathsf{g}(t)$. Since $r_t \to \mathsf{g}_t$ in H, r_t must have turned at $a_\mathsf{g}(t)$ to make this assignment in H, since r_t is not within b. By our algorithm, r_t first considered all H-servers within $T_\mathsf{g}(t)$ that are to the left of r_t. But since r_t assigned to g_t in H, there are no free H-servers to the left of r_t within $T_\mathsf{g}(t)$ at time t^-. Consequently, the only free G-server that might be to the left of r_t within $T_\mathsf{g}(t)$ is g_{t-1}, but then at time t^+, there are no free G-servers to the left of r_t within $T_\mathsf{g}(t)$. We have already established that there are no free G-servers between r_t and g_t, and so we can conclude that there are no free G-servers within $T_\mathsf{g}(t)$ to the left of g_t at time t^+.

3. r_t is within b, or r_t is to the right of g_t. If r_t is within b, then using Lemma 5.1.11, g_{t-1} cannot be within b, and so r_t prefers g_t over g_{t-1}. If r_t is to the right of g_t, then we can also assert that r_t prefers g_t over g_{t-1}. Since g_t is free at time t^- in G's run, r_t cannot assign to g_{t-1} in G's run, a contradiction.

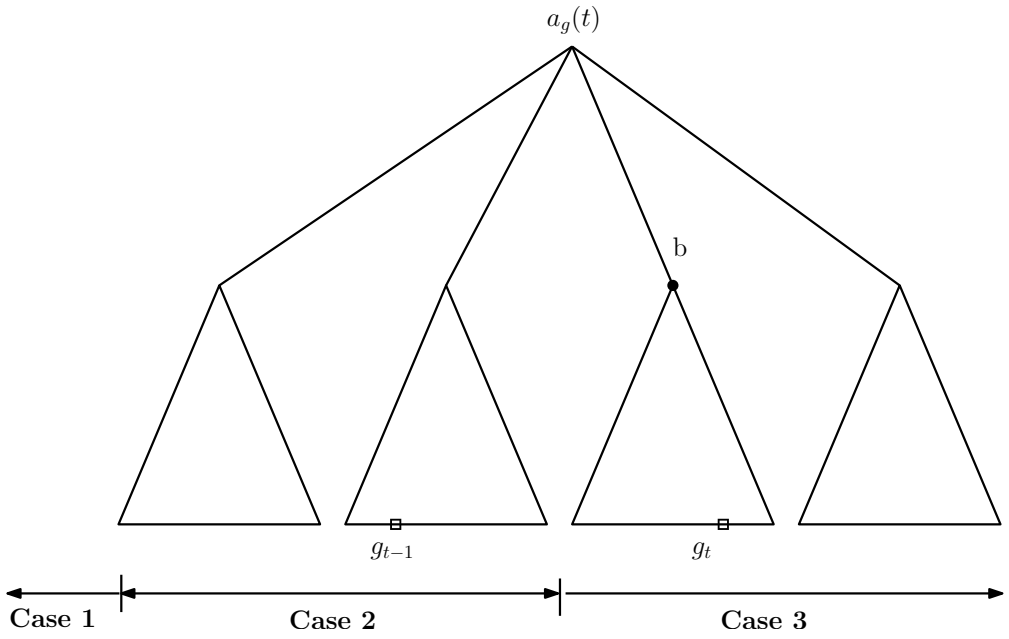

Figure 5.2: The three cases for the location of r_t.

Thus, in all possible cases, we conclude that there are no free G-servers to the left of g_t at time t^+ within $T_{\mathsf{g}}(t)$, which maintains the claim. An analogous argument can be used to show that this property holds for all h_t.

□

The following lemma follows from the above proof:

Lemma 5.1.13. *Suppose* $\mathsf{dir}_{\mathsf{g}}(t-1) = R$ *and* $\mathsf{dir}_{\mathsf{g}}(t) = L$*, and let* $u < t$ *be the largest integer such that* $\mathsf{dir}_{\mathsf{g}}(u-1) = R$ *and* $\mathsf{dir}_{\mathsf{g}}(u) = L$*, if any. Then the level of the tree* $T_{\mathsf{g}}(t^+)$ *is strictly greater than the level of* $T_{\mathsf{g}}(u^+)$*. Similarly, if* $\mathsf{dir}_{\mathsf{h}}(t-1) = R$ *and* $\mathsf{dir}_{\mathsf{h}}(t) = L$*, where* $u < t$ *is the largest integer such that* $\mathsf{dir}_{\mathsf{h}}(u-1) = R$ *and* $\mathsf{dir}_{\mathsf{h}}(u) = L$*, if any, then the level of* $T_{\mathsf{h}}(t^+)$ *is strictly greater than the level of* $T_{\mathsf{h}}(u^+)$*.*

Proof. Assume for the sake of contradiction that $T_{\mathsf{g}}(u^+) = T_{\mathsf{g}}(t^+)$ for some choice of u and t. From Remark 5.1.9, we have that for all $u \leq w \leq t$, $T_{\mathsf{g}}(w^+) = T_{\mathsf{g}}(u^+)$. Let v be such that $u < v < t$ and $\mathsf{dir}_{\mathsf{g}}(v-1) = L$ and $\mathsf{dir}_{\mathsf{g}}(v) = R$. Lemma 5.1.12 gives us that there are no free G-servers within $T_{\mathsf{g}}(v^+) = T_{\mathsf{g}}(t^-)$ and to the left of g_v. Also, for all $v \leq w \leq t$, $\mathsf{dir}_{\mathsf{g}}(v) = R$ since t is the first time after v that the direction can change. By repeated applications of Lemma 5.1.10, there can be no free servers at time t^+ between g_v and g_{t-1}, either. Thus, at time t^+, there are no free G-servers to the left of g_{t-1} that are within $T_{\mathsf{g}}(t^-)$.

Now, note that g_t is a free G-server at time t^+ that is to the left of g_{t-1}. Thus, it cannot be within $T_{\mathsf{g}}(t^-)$, and so $T_{\mathsf{g}}(t^+)$, which must include g_t by definition, is not equal to $T_{\mathsf{g}}(t^-)$, a contradiction.

□

Remark 5.1.14. For all t, $T_{\mathsf{g}}(t), T_{\mathsf{h}}(t) \subseteq T^*$, the subtree rooted at the least common ancestor of r_1 and s_1.

Lemma 5.1.15. *For the general D-ary α-HST, the total distance traveled by either the G-cavities or H-cavities is $O(d_T(r_1, s_1))$.*

Proof. As t increases, T_{g} may change, and define ρ_t such that $T(\rho_t) = T_{\mathsf{g}}(t)$; note that $level(\rho_t)$ is non-decreasing by Remark 5.1.9. Moreover, as long as the scope stays fixed at some subtree $T(\rho_t)$, the G-cavity g_t can only change direction once, and hence the total distance it travels is at most twice the width of $T(\rho_t)$, which is $2D \cdot O(2^{level(\rho_t)})$. Finally, by Remark 5.1.9, each of the ρ_i's is a descendent of a^*, the root of T^*. Hence the total distance traveled by the G-cavity is at most a $2D$ times $1 + \alpha + \alpha^2 + \ldots + \alpha^{level(a^*)}$, which is $O(D \cdot \alpha^{level(a^*)}) = O(D \cdot d_T(r_1, s_1))$. A similar argument holds for the distance traveled by the H-cavity. □

Now plugging this into Lemma 5.1.6 (the "accounting lemma"), we get

$$\sum_{t \le k} d_L(r_t, G_\sigma(r_t)) - \sum_{t \le k} d_L(r_t, H_\sigma(r_t)) \le O(D \cdot d_T(r_1, s_1)) + 2\, d_T(r_1, s_1).$$

Note that $d_T(r_1, s_1) \ge d_L(r_1, s_1)$, and hence the expression on the right is at most $\lambda\, d_T(r_1, s_1)$ for some $\lambda = O(D)$. This completes the proof of Lemma 5.1.3, the "hybrid lemma" and hence the proof of Theorem 5.1.1. Next, we show that this analysis is tight.

5.2 A Tight Example for HST-greedy

Consider the setting of k servers and requests on the real line where a server s_i is placed at point i for every $i \in \mathbb{Z}$ such that $1 \le i \le k$, and r_1 is placed at point 2, and then for all $1 < i \le k$, r_i is placed at point i. Thus, each adjacent server is at distance 1 from each other, there are no requests sitting on s_1, there are two requests sitting on s_2, and there is one request sitting on every other server. The optimal matching assigns $r_1 \to s_2$, $r_2 \to s_1$, and then $r_i \to s_i$ for all $i > 2$. The cost of this matching is 1. We want to show that the HST-greedy algorithm has an expected cost of $\Theta(\log k)$.

First, note that the binary tree of $2k$ leaves leaves $2k-1$ cuts on the interval $[1, 2k]$. The depths of these cuts, going from left to right, can be expressed as the sequence

$$s = 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, \cdots .$$

The sequence s can also be defined recursively. Let $t_1 = 1$ and $t_i = \langle t_{i-1} \rangle\ i\ \langle t_{i-1} \rangle$. Then, $s = t_{\log_2(k)+1}$.

Now, suppose we choose some $j \in [2, k+1]$ so that $[j, j+k-1]$ is our interval under consideration. Let r_i be the request that assigns to s_1. Then, the cost of the matching is simply $2d_L(s_1, r_i) - 1 = 2i - 3$. It remains to figure out, for a fixed j, which request ends up assigning to s_1.

Remark 5.2.1. The sequence s is such that for any $j \geq 1$, if $i = j + 2^{s_j - 1}$, then i is the smallest integer such that $s_i > s_j$.

Now, suppose the interval under consideration is $[j, j+k-1]$ for $1 \leq j \leq k+1$. Then, the cut between s_1 and s_2 is the cut placed in $[j, j+1]$, which has depth s_j. By Remark 5.2.1, all cuts between s_j and $s_{j+2^{s_j-1}}$ have depth smaller than s_j, so this means that the first $2^{s_j-1} - 1$ requests will not assign to the s_1, but the next request will. Thus, the r_i that assigns to s_1 is such that $i = 2^{s_j-1}$. Consequently, the cost of the matching for such a j is $\Theta(2^{s_j})$.

Remark 5.2.2. For all $j \in [1, k+1]$, there is exactly one j such that $s_j = \log_2(k) + 1$, and then there are exactly $2^{\log_2(k)-i}$ integers j such that $s_j = i$ for $1 \leq i \leq \log_2(k)$.

Since each j is chosen uniformly at random, we can now compute the expected cost of the matching M:

$$\mathbf{E}[M] = \sum_{i=0}^{\log_2(k)} \mathbf{Pr}[s_j = i] \cdot \mathbf{E}[M | s_j = i]$$

We have shown through Remark 5.2.2 that $\mathbf{Pr}[s_j = i]$ is $1/k$ for $i = \log_2(k) + 1$ and $2^{\log_2(k)-i}/k$ for $i > 0$, and $\mathbf{E}[M | s_j = i]$ is $\Theta(2^{s_j}) = \Theta(2^i)$. Thus, we get that

$$\mathbf{E}[M] = \sum_{i=0}^{\log_2(k)} 2^{\log_2(k)-i}/k \cdot \Theta(2^i) = \sum_{i=0}^{\log_2(k)} \Theta(k/k \cdot 2^i/2^i) = \sum_{i=0}^{\log_2(k)} \Theta(1) = \Theta(\log k)$$

Thus, this specific setting of servers and requests forces HST-greedy to perform with expected competitive ratio $\Theta(\log k)$.

Chapter 6

Conclusion and Open Problems

We have presented three algorithms that perform competitively on HSTs, two of which imply an $O(\log k)$ competitive ratio on the line. For each algorithm we present, the analysis for its competitive ratio is unique. In Chapter 3, we take a combinatorial approach to the problem. We show how if we can carefully count and make bounds for the number of digons in the matching, then we can upper-bound the cost of the matching. For the line metric, this analysis yielded an $O(\log^2 k)$ competitive ratio for a very simple algorithm: deterministic greedy on the HST. In Chapter 4, we argue via a carefully chosen potential function that our modification to the MNP algorithm yields $O(\log^2 d \log k)$ for general metrics, which implies $O(\log k)$ for the line and low-dimensional spaces. In Chapter 5, we show how if we ignore the cost of our algorithm on the HST and instead measure the cost of our algorithm on the original metric, we can obtain a tight competitive ratio of $O(\log k)$ on the line, where the algorithm that we run on the HST is deterministic.

The new approaches and algorithms that we have introduced give rise to potential opportunities for improvement in the upper bound of online metric matching, and they are mentioned in the following section.

6.1 Future Work

Listed here are several avenues that were not fully explored in this thesis which seem to hold some promise in improvements for the upper bound on the competitive ratio of online metric matching.

6.1.1 Randomized Greedy with the Combinatorial Approach

In Chapter 3, we explore the deterministic greedy algorithm and conclude that it yields a competitive ratio of hd. However, our worst-case analysis was quite harsh,

and it seems as if randomization could help. If we use a similar definition of the canonical sequences as in the proof of the deterministic greedy algorithm's competitive ratio, perhaps one could show that randomized greedy yields a competitive ratio of $O(h \log d)$—or better yet, without the dependence on h.

Also, even for deterministic greedy, one could use the fact that the HST construction from the line is a randomized procedure, and could possibly make the worst-case on the HST unlikely. In our analysis, we pretended as if our adversary could choose the worst-case HST for deterministic greedy to run on. However, in reality this is not the case, and could possibly be taken advantage of when assuming that the HST that deterministic/randomized greedy runs on is from the line.

6.1.2 Low-degree HST Constructions for Metric Spaces

In our analysis of the MNP algorithm, we show how the modified algorithm we propose has competitive ratio $O(\log^2 d \log k)$ on the original metric. In general, the best we can assume is that $d \leq k$, but do there exist any classes of metric spaces where d is still relatively small, say, $d = O(\log k)$? If this were the case, then running our modification to the MNP algorithm would result in an $O((\log \log k)^2 \log k)$ algorithm, which represents an improvement over $O(\log^2 k)$.

6.1.3 Extension of Ideas in Chapter 5

A small trick that we performed for the analysis of the HST-greedy algorithm allowed us to obtain tight bounds on its competitive ratio. However, we noted that this trick assumes that the HST was constructed from the line. It would be interesting to find other metric spaces for which this trick still applies. An even more interesting result would be if this kind of analysis applies to other algorithms, such as those discussed in Chapters 3 and 4, or from previous works.

6.2 Open Problems

The following is a more broad list of open problems whose answers are desirable, and would represent significant progress towards tightening the bounds on online metric matching.

1. Do there exist deterministic or randomized lower bounds for online metric matching on the line that do not reduce from results on the cow-path problem?

2. Does there exist a "simple" randomized algorithm that does not involve HSTs and still performs competitively (say, poly$(\log k)$)?

3. Can we improve upon the upper bound of $2k - 1$ for deterministic online metric matching on the line, or on general metric spaces?

4. Does there exist a derandomized version of the HST construction from the line that can be used to show a competitive deterministic algorithm?

Bibliography

[BBGN07] N. Bansal, N. Buchbinder, A. Gupta, and J. S. Naor. An o(log2k)-competitive algorithm for metric bipartite matching. In Proceedings of the 15th annual European conference on Algorithms, pages 522–533, 2007.

[FHK05] B. Fuchs, W. Hochstattler, and W. Kern. Online matching on a line. Theoretical Computer Science, 332:251–264, 2005.

[FRT03] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, pages 448–455, 2003.

[KMV94] S. Khuller, S. G. Mitchell, and V. V. Vazirani. On-line algorithms for weighted bipartite matching and stable marriages. Theor. Comput. Sci., 127(2):255–267, 1994.

[KN03] E. Koutsoupias and A. Nanavati. The online matching problem on a line. In WAOA03, pages 179–191, 2003.

[KP93] B. Kalyanasundaram and K. Pruhs. Online weighted matching. J. Algorithms, 14(3):478–488, 1993.

[KP95] E. Koutsoupias and C. H. Papadimitriou. On the k-server conjecture. J. ACM, 42:971–983, September 1995.

[KP98] B. Kalyanasundaram and K. Pruhs. Online network optimization problems, 1998. Online Algorithms: The State of the Art , eds. A. Fiat and G. Woeginger, Lecture Notes in Computer Science 1442, Springer-Verlag.

[MNP06] A. Meyerson, A. Nanavati, and L. Poplawski. Randomized online algorithms for minimum metric bipartite matching. In SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pages 954–959, 2006.