Competitive Algorithms for Distributed Data Management

Yair Bartal　　Amos Fiat　　Yuval Rabani

Department of Computer Science
School of Mathematics
Tel-Aviv University
Tel-Aviv 69978, Israel.

Abstract

We deal with the competitive analysis of algorithms for managing data in a distributed environment. We deal with the file allocation problem ([DF], [ML]), where copies of a file may be be stored in the local storage of some subset of processors. Copies may be replicated and discarded over time so as to optimize communication costs, but multiple copies must be kept consistent and at least one copy must be stored somewhere in the network at all times. We deal with competitive algorithms for minimizing communication costs, over arbitrary sequences of reads and writes, and arbitrary network topologies. We define the constrained file allocation problem to be the solution of many individual file allocation problems simultaneously, subject to the constraints of local memory size. We give competitive algorithms for this problem on the uniform network topology. We then introduce distributed competitive algorithms for on-line data tracking (a generalization of mobile user tracking [AP1, AP3]) to transform our competitive data management algorithms into distributed algorithms themselves.

1 Introduction

The management of data in a multiprocessing environment has been extensively studied. The 1981 survey paper by Dowdy and Foster [DF], dealing with the file allocation (or assignment) problem, cites close to a hundred references.

The file allocation problem has a plethora of models, with differing design goals and assumptions. [DF] compares studies on fourteen different models, and mentions several others. We deal with dynamic self-adjusting algorithms, in the context of two basic file allocation problems, and primarily address issues of communications efficiency. We define the *file allocation problem* and the more complex *constrained file allocation problem*, but these names may conflict with other usage.

We consider the competitive performance [ST, KMRS, MMS, BLS, BBKTW] of algorithms for these problems, and present algorithms with an optimal or nearly optimal competitive ratio. Black and Sleator [BS] consider competitive algorithms for two partial components of the file allocation family of problems. Our file allocation problem may be viewed as the combined solution to the two subproblems defined in [BS].

Another issue is that of global versus distributed management. The question of file allocation is quite different in the context of disk management in a small network of large mainframes versus local cache management in a large scale multiprocessing computer. We show that our competitive data management algorithms can be run in a distributed environment, at the cost of a small increase of the competitive ratio.

1.1 Competitive Basics

Informally, an on-line game consists of a sequence of interleaved events and responses. Events are produced by one player, the adversary, whereas responses are produced by the other player, the on-line algorithm. Each response is produced without knowing what future events will be. A sequence of events and responses has a fixed cost.

The competitive ratio [ST, KMRS] is defined as the ratio between the cost associated with an on-line algorithm to deal with a sequence of events versus the cost expended by an optimal (off-line) algorithm. The competitive ratio is c if for all event sequences, (online cost) $\leq c \times$ (off-line cost) + some additive constant. A competitive algorithm with a competitive ratio of c is called *strictly* competitive if the additive constant is zero. Models for on-line problems are presented in [BLS], [MMS], [BBKTW]. Competitive analysis of distributed data management algorithms begins with Karlin *et. al.* in [KMRS] who analyze competitive algorithms for snoopy caching on a bus connected PRAM.

If the on-line algorithm may use randomization to process events then the competitive ratio is defined as an expectation and one must make precise the power given to the adversary. Ben-David et. al. [BBKTW] define oblivious and adaptive adversaries and show various relationships between the competitive ratios achievable against different adversaries. An oblivious adversary must commit to the sequence of events while knowing neither the coin tosses nor the actions taken by the on-line algorithm. An adaptive adversary may decide upon the next event after seeing all previous on-line responses. An adaptive on-line adversary must respond to events when it decides upon them and may not later change previous actions. An adaptive off-line adversary may decide upon all its responses after seeing the entire sequence, [BBKTW] show that randomization does not help against such an adversary.

The distinction between adaptive and oblivious adversaries is not relevant for deterministic algorithms. We distinguish between the adversary types by adding the qualification "(oblivious)" or "(adaptive)" when referring to a competitive ratio.

[BBKTW] also show how to transform a randomized c-competitive algorithm against an adaptive on-line adversary into a c^2-competitive deterministic algorithm if a certain *augmented potential function* they define is computable.

1.2 The File Allocation Problem

A *network* is a weighted graph where processors are represented by vertices P, and edges weights represent the length or cost of the link between the two adjacent processors. The weighted graph need not obey the triangle inequality, but a natural metric space can be defined where the points are processors and the distance between two points is equal to the length of the shortest path between the processors in the weighted graph. We use the terms network, weighted graph, and metric space as called for by the discussion, but they refer to the same underlying interconnection network.

The *file allocation problem* assumes that data is organized in indivisible blocks such as files (or pages). Data can be accessed via communication links by paying a charge equal to the data transfer size times the distance traversed. Words or records can be accessed or updated over communication links, but a file cannot be split among processors. Files may be replicated in various processors throughout the network, but consistency must be maintained. Copies may also be discarded but at least one copy of every file must be stored somewhere in the network. This problem can be formalized as follows:

Initially, a subset $Q \subseteq P$ of processors is each assigned a copy of the file. The algorithm receives a sequence of requests initiated by processors in P. Each request is either a *read* request or a *write* request. A read request at processor r is served by the closest processor p holding a copy of the file. The cost associated with this transmission is the distance between p and r. In response to a *write* request initiated at processor w, the algorithm must transmit an update to all currently held copies of the file – the subset $Q \subseteq P$. It pays a cost equal to the minimum Steiner tree spanning $Q \cup \{w\}$. In between requests, the algorithm may re-arrange the copies of the database. A processor may delete the copy it is holding, unless it is the last copy in the network, at no cost. The file may also be *replicated* from a processor p, which holds a copy, to a subset $Q' \subset P$. The cost of replicating is equal to D times the minimum Steiner tree spanning $Q' \cup \{p\}$. D represents the ratio between the size of the entire file and the size of the minimal data unit being read or updated. A new current subset Q of processors holding copies of the file is determined as a result of delete and replicate steps. A combination of a replicate step from a processor p to a processor q, followed by a delete at p, is sometimes called a *migration* step. The subset Q is called the configuration of the algorithm.

While the costs above are certainly a lower bound on the communication costs for any algorithm in a given configuration, it is an upper bound for on-line algorithms only if they have global knowledge of the current configuration and can solve hard minimum Steiner tree problems. In fact, we can charge the on-line algorithm the real communication costs and obtain competitive algorithms without either assumption.

If many read requests to a specific file are issued by some processor, it may be advisable to copy the relevant file to, or near, that processor. However, this should be balanced by the relatively greater cost of moving an entire file versus the smaller cost of transferring only the data being read. If a processor issues a write request, it now seems advisable to move *all* copies of the file to, or near, the issuing processor. *I.e.*, move some copy near the processor and discard others. These conflicting heuristics must somehow be balanced.

One way to limit the concerns of data consistency is to assume that only one processor may store a copy of a file at any given time. Thus, read and write requests issued by other processors in the network must all access the processor that holds the copy. [BS] call this problem the *file migration problem*. [BS] give an optimal 3-competitive ratio for this problem on the uniform network topology and for trees. Westbrook ([W]) gives a randomized 3-competitive algorithm against an adaptive on-line adversary for any network, and a $1+\phi$-competitive randomized algorithm against an oblivious adversary. The data migration problem can also be considered as a special case of the 1-server with excursion problem defined in [MMS].

Black and Sleator also consider the *file replication problem*, which is the file allocation problem with writes disallowed. Here, copies need never be discarded. They give an optimal 2-competitive algorithm for the replication problem when the network is a tree, or a uniform graph.

We give a randomized $O(\log n)$-competitive algorithm against an adaptive on-line adversary for the file allocation problem on any network with n processors. We also prove that $\Omega(\log n)$ is the best competitive ratio one can obtain for general networks, even for randomized algorithms against an oblivious adversary. Our algorithm is also memoryless [RS] (I.e., its decisions depend only on its current configuration and the current request). We give an optimal deterministic 3-competitive algorithm for the uniform architecture (*e.g.*, bus based). Westbrook and Yan [WY1] have obtained an optimal deterministic 3-competitive algorithm for tree networks. In this paper we present time and memory efficient competitive algorithms for tree networks.

The proof of our $O(\log n)$-competitive algorithm uses a construct we call the "*natural potential function*." This is a modification of the [BBKTW] "augmented potential function." We prove general theorems relating a large class of *configuration problems* and the natural potential function. This is useful in *proving* the correctness of competitive algorithms for complex problems by concatenating competitive algorithms for simpler subproblems. Our analysis of the competitive file allocation algorithm is based upon the natural potential function for on-line Steiner tree algorithms. Similar ideas appear in [CL] in the context of deterministic algorithms, our natural potential function is defined for both deterministic and randomized algorithms.

1.3 The Constrained File Allocation Problem

If it is not true that every processor can accommodate all files, then copying a file into a processor's local memory may be impossible as that memory is full. Possibly, some other file in local memory should be dropped. However, if this candidate is the last copy in the network, it must be stored somewhere else. Thus, it may be dumped to some other processor that has space for it, or that will have space for it after it too drops a file currently in its memory. Clearly, this game of hot potato may continue.

The *constrained file allocation problem* attempts to solve many individual file allocation problems simultaneously, while considering the actual memory capacity of the processors. The point is that the different file allocation problems may interfere with each other if there is insufficient memory. Similarly, we could define the *constrained file migration problem* if holding multiple copies of the same file is disallowed.

For the file allocation problem, different files may have different sizes as every file allocation problem is solved independently. For the constrained file allocation problem, we only deal with files equal in size (D). One case where this makes perfect sense is in the context of distributed virtual memory, where the entire network is viewed as one large address space, and pages (of various multiplicities) are stored throughout the network so as to minimize communication costs.

Given that processor i can accommodate k_i files, all files equal in size, let $m = \sum_{i=1}^{n} k_i$. We give an $O(m)$ competitive deterministic algorithm for the constrained file allocation problem on uniform networks. We also give a lower bound of $\Omega(m)$ on the competitive ratio for any network.

1.4 Distributed Execution

Our algorithms above assume that some centralized power keeps track of the migrating, replicating, and dying populations of files in the network, and tells processors how to go about finding the closest current copy of every file. To justify this assumption in the distributed setting for arbitrary architectures, we present a generalization of the Awerbuch and Peleg [AP1, AP3] mobile user algorithm called *distributed data tracking*.

Disallowing ESP, if two processors have a copy of the same file then it must have a common source and must have reached these processors through communications links. We seek to access a copy of a file, while passing through a path of length not much larger than the shortest path to a copy of the file. We manage a distributed data structure that allows fast access to the closest copy of a file, while the cost of managing the data structure is amortized against the cost of the data movement itself.

[AP1] solve a similar problem, they allow a *move* operator to be applied to a mobile user, but do not efficiently support birth and death. We allow *insert* and *delete* operations. The competitive ratio is polylogarithmic in n. The total cost for a sequence of inserts and deletes is $O(\log^2 n / \log^2 D)$ times the inherent cost for these operations, where D represents the file size. The path length traversed per find is $O(\log^2 n / \log^2 D)$ times the length of the shortest path to a copy of the file, and the copy of the data found is at distance at most $O(\log n / \log D)$ times the length of the shortest path to a copy.

We use distributed data tracking to we present a randomized distributed algorithm for the file allocation problem, with a competitive ratio of $O(\log^4 n / log^3 D)$ against adaptive on-line adversaries.

Our major omission in this paper is that we do not consider problems of concurrency and effectively assume that all read and write commands are serialized. We note that some aspects of our algorithms do not require this assumption, but do not claim a complete solution at present.

2 Preliminaries

3 Configuration Problems and Potential Functions

We define on-line configuration problems. As a class of problems it is equivalent to the request-answer games of [BBKTW]. Most of the previously studied on-line problems (including server problems and metrical task systems, and including the problems dealt with in this paper) are naturally described in the context of this model.

Definition. An *on-line configuration problem* consists of a set of configurations Con, a set of requests Req, and cost function cost : $Con \times Con \times Req \mapsto \mathbb{R} \cup \{\infty\}$.

An *algorithm* for an on-line configuration problem gets a sequence of requests drawn from Req and an initial configuration drawn from Con. For each request r in the input sequence, the algorithm selects a configuration from Con. If C_1 is the configuration selected for the previous request (or the initial configuration, if r is the first request in the sequence) and C_2 is the configuration selected for r, then the algorithm's *cost for serving* r is $\mathrm{cost}(C_1, C_2, r)$. The cost of the algorithm over the entire sequence is the sum of costs for serving the individual requests.

A randomized algorithm tosses coins to select configurations. Its cost is the expectation taken over its own coin tosses. An *on-line* algorithm selects the configuration for a request r independent of the suffix of the sequence after r.

The index of a request in an input sequence is called the *stage* or the *time*.

A *task system* (see [BLS]) is an on-line configuration problem where the cost function has the following structure. Define the cost of a move between configurations in Con, denoted $\mathrm{dist}(C_1, C_2)$ (where $C_1, C_2 \in Con$) (this is the *move cost*). Associate with every request r and every configuration C the cost of serving r in configuration C, denoted $\mathrm{task}(C, r)$ (this is the *task cost*). The cost function of a task system is defined by: $\mathrm{cost}(C_1, C_2, r) = \mathrm{dist}(C_1, C_2) + \mathrm{task}(C_2, r)$. For a task system, input requests are usually called *tasks*. If the move cost function dist forms a metric space over Con, then the task system is called *metrical*.

The *history* of an algorithm at a given stage is defined by the corresponding prefix of the sequence of requests and the algorithm's coin tosses so far.

The *memory* of an algorithm is a subset of its history such that the way the algorithm serves future requests is a function of its memory.

For the competitive analysis of on-line algorithms, request sequences are assumed to be generated by an *adversary* that has to serve them as well. The competitive ratio of an on-line algorithm is the ratio of costs maximized over all adversaries of a certain type. In this paper, we are specifically interested in analysis against the *adaptive on-line* adversary. This type of adversary can generate the sequence of requests on-line as the algorithm serves them, and can adapt to the on-line algorithm's coin tosses after each request is served, as long as the adversary serves each request before the on-line algorithm does so. So, if the on-line algorithm is randomized, the sequence generated by this type of adversary is randomized as well. Ben-David et al. [BBKTW] elaborate on this and other types of adversaries.

Notation. Fix a time n. The request sequence at time n, is denoted $\sigma_n = r_1 r_2 \cdots r_n$.

Let Alg be an on-line algorithm, and let Adv be an adversary. Alg's history at stage n is denoted h_n, its memory is denoted m_n, and the adversary's configuration is denoted A_n. A_0 and h_0 are Adv's initial configuration, and Alg's initial history respectively.

$\mathrm{Cost}_{\mathrm{Alg}}(\sigma_n)$, and $\mathrm{Cost}_{\mathrm{Adv}}(\sigma_n)$ denote Alg's cost and the adversary's cost for serving σ_n respectively. For a randomized algorithm $\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(\sigma_n))$ denotes its expected cost over the request sequence.

Let τ be a sequence of requests, then $\mathrm{E}_\tau(\mathrm{Cost}_{\mathrm{Alg}}(h_n, \tau))$ denotes Alg's expected cost for serving τ after serving σ_n, conditioned upon the fact that Alg's history after serving σ_n is h_n (i.e., toss coins as to reach this history).

The notation $\mathrm{E}_\tau(\cdot)$, where τ is a sequence of requests, means that expectation is taken over the algorithm's coin tosses while serving τ. (The subscript τ in the notation is often omitted when the meaning is clear.)

Since an on-line algorithm's future behavior depends on its memory alone, the algorithm's memory is often used instead of its history. Similarly $\mathrm{Cost}_{\mathrm{Adv}}(A_n, \tau)$ denotes the adversary's cost for serving τ starting with configuration A_n.

Finally let the history space $\mathcal{H}$ of the problem be the set of all possible pairs of request sequences and coin tosses.

We define potential functions for on-line algorithms:

Definition. A *potential function* Φ for a (possibly randomized) algorithm Alg and some constant c is a function $\Phi : \mathcal{H} \times Con \mapsto \mathbb{R}$, having the following properties:

1. For every history h_n and configuration A_n, $\Phi(h_n, A_n) \geq 0$.

2. For every $n \geq 1$, let Alg's history at time n be h_n, and let Adv's final configuration be A_n.

Then,

$$\mathrm{E}(\Phi(h_n, A_n)) - \Phi(h_0, A_0) \;\; \leq \;\; \mathrm{E}(c \cdot \mathrm{Cost}_{\mathrm{Adv}}(\sigma_n) - \mathrm{Cost}_{\mathrm{Alg}}(\sigma_n)),$$

A potential function Φ is called *strict* iff $\Phi_0 = \Phi(h_0, A_0) = 0$.

Potential functions are useful in the competitive analysis of on-line algorithms, as shown in the following theorem (see [ST]):

Theorem 1 *If there exists a potential function for Alg (and c), then Alg is c-competitive (against adversaries for which property 2 above holds).*

The following types of potential functions are commonly used for competitive analysis against an adaptive on-line adversary. We name these types of potential functions according to the number of steps in the game on which the analysis proceeds.

Definition. A *two-step potential function* has property 1 of a potential function, and, instead of property 2, the following stronger property: Let $\sigma_{n+1} = \sigma_n r$, h_{n+1} and A_{n+1} be Alg's history and Adv's configuration after σ_{n+1}, respectively. Then,

$$\mathrm{E}(\Phi(h_{n+1}, A_{n+1})) - \Phi(h_n, A_n) \;\; \leq \;\; c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A_n, r) - \mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(h_n, r)).$$

A *one-step potential function* for a task system algorithm has property 1 of a potential function and the following properties:

$$\Phi(h_n, A_{n+1}) - \Phi(h_n, A_n) \;\; \leq \;\; c \cdot \mathrm{dist}(A_n, A_{n+1}) \tag{1}$$

$$\mathrm{E}(\Phi(h_{n+1}, A_{n+1})) - \Phi(h_n, A_{n+1}) \;\; \leq \;\; c \cdot \mathrm{task}(A_{n+1}, r) - \mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(h_n, r)). \tag{2}$$

We will use the term *global potential function* to refer to any potential function that satisfies the first definition in order to distinguish between the first definition of a potential function and the last two definitions. Obviously, one or two-step potential functions are also global potential functions.

Remark. We use here the usual definition of a task system in which the task cost depends on the new configuration. All results stated in the next section regarding task systems also hold if the task cost depends on the configuration before receiving the request. The data management problems are formalized in the latter manner.

4 The Natural Potential Function

Fix some on-line configuration problem $\mathcal{P}$, let Alg be an on-line algorithm for $\mathcal{P}$, and let $c > 0$. Let the adversary be Adv_0. Let σ_n be the previous request sequence it has produced, and let m_n be the current on-line memory configuration. We define the *natural potential function* for Alg as follows:

$$\Upsilon(m_n, A) \quad = \quad \sup_{\mathrm{Adv}}\{\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, \tau) - c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A, \tau))\},$$

where Adv ranges over all possible adaptive on-line adversaries that reach configuration A, and τ is a random variable that represents the request sequence generated by Adv.

Theorem 2 *An algorithm Alg is c-competitive for $\mathcal{P}$ against adaptive on-line adversaries iff Alg has a two-step potential function (for c). Alg is strictly competitive iff the potential function is strict.*

Proof. The if direction follows immediately from the fact that any two-step potential function is also global.

We will prove that Υ is indeed a two-step potential function for Alg and c.

We first show that Υ is well-defined, that is finite, for all memory values m_n and all configurations A. Assume that Alg is c-competitive against adaptive on-line adversaries, then there exist a constant a_0 s.t. for every on-line adversary Adv

$$\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(\rho)) \quad \leq \quad c \cdot \mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}}(\rho)) + a_0$$

(where ρ is a random variable representing the request sequence generated by Adv). Now, let Adv be the adaptive on-line adversary that produces the random sequence τ. Define adversary Adv_1 as follows: Adv_1 produces the request sequence σ_n, and serves it the same as Adv_0. He then continues to generate a random sequence τ only if the Alg's memory is m_n, and serves it the same as Adv would (otherwise, Adv_1 terminates the sequence). Let p be the probability that Alg's memory after serving σ_n is m_n. Since m_n is a valid memory configuration for Alg after serving σ_n, $p > 0$. The expected cost for Alg against Adv_1 satisfies

$$\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(\sigma_n \tau)) \quad \geq \quad p\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, \tau)).$$

The cost of the adversary is

$$\mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}_1}(\sigma_n \tau)) \quad = \quad \mathrm{Cost}_{\mathrm{Adv}_0}(\sigma_n) + p\mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}}(A, \tau)).$$

From the competitiveness of Alg we have

$$\begin{aligned} p\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, \tau)) &\leq \mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(\sigma_n \tau)) \\ &\leq c \cdot \mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}_1}(\sigma_n \tau)) + a_0 \\ &= c \cdot \{p\mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}}(A, \tau)) + \mathrm{Cost}_{\mathrm{Adv}_0}(\sigma_n)\} + a_0. \end{aligned}$$

Set $a_1 = \frac{1}{p}(\mathrm{Cost}_{\mathrm{Adv}_0}(\sigma_n) + a_0)$. Then for any adaptive on-line adversary Adv,

$$\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, \tau)) \leq c \cdot \mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}}(A, \tau)) + a_1.$$

We therefore conclude that the natural potential function it is finite.

We now show it is a two-step potential function for Alg.

Clearly for all n and A, Υ is nonnegative since for $\tau = \epsilon$ the empty sequence, $\mathrm{Cost}_{\mathrm{Alg}} = \mathrm{Cost}_{\mathrm{OPT}} = 0$. Consider a new request r generated by Adv_0. Adversary Adv_0 serves the request by moving from configuration A_n to configuration A_{n+1}. Then m_{n+1} is chosen by the on-line algorithm according to its coin tosses on r.

Consider the expected change in the potential function after serving a new request r. The potential function value before the request may only decrease if we limit the adversary to request sequences that start with a request at r. Thus we obtain

$$\begin{aligned} \mathrm{E}(\Delta\Upsilon) &= \mathrm{E}_r(\Upsilon(m_{n+1}, A_{n+1})) - \Upsilon(m_n, A_n) \\ &\leq \mathrm{E}_r[\sup_{\mathrm{Adv}}\{\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_{n+1}, \tau) - c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A_{n+1}, \tau))\}] \\ &- \sup_{\mathrm{Adv}}\{\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, r\tau) - c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A_n, r\tau))\}. \end{aligned}$$

If an adversary Adv starts on configuration A_{n+1} then we can bound the previous potential function from below, by using an adversary which first serves the request the same as Adv_0, moving from configuration A_n to A_{n+1}, and then continues the same as Adv. Hence,

$$\begin{aligned} \mathrm{E}(\Delta\Upsilon) &\leq \sup_{\mathrm{Adv}}\{\mathrm{E}_r[\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_{n+1}, \tau) - c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A_{n+1}, \tau))] \\ &- \{\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, r\tau)) - c \cdot \mathrm{Cost}_{\mathrm{Adv}_0}(A_n, r) - c \cdot \mathrm{E}_r[\mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}}(A_{n+1}, \tau))]\}\} \\ &= \sup_{\mathrm{Adv}}\{\mathrm{E}_r[\mathrm{E}_\tau(\mathrm{Cost}_{\mathrm{Alg}}(m_{n+1}, \tau))] - \mathrm{E}_{r\tau}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, r\tau))\} + c \cdot \mathrm{Cost}_{\mathrm{Adv}_0}(A_n, r) \\ &= c \cdot \mathrm{Cost}_{\mathrm{Adv}_0}(A_n, r) - \mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n, r)). \end{aligned}$$

If Alg is strictly competitive then for every on-line adversary Adv, there holds $\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(\rho)) \leq c \cdot \mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}}(\rho))$, by the definition of Υ, it follows that $\Upsilon_0 = 0$, and hence Υ is a strict potential function. ■

Theorem 3 *An algorithm Alg for a task system is c-competitive against adaptive on-line adversaries iff it has a one-step potential function (for c).*

Proof. The if direction follows from the fact that any one-step potential function is also global.

We shall show Υ is a one-step potential function for task systems.

Consider an adversary move from configuration A_n to configuration A_{n+1} to serve the request r. By the triangle inequality the cost of an adversary to serve a request sequence starting at configuration A_n, is at most the cost of first moving to A_{n+1} and then serving the request sequence there. Therefore,

$$\begin{aligned}
\Delta\Upsilon &= \Upsilon(m_n, A_{n+1}) - \Upsilon(m_n, A_n) \\
&\le \sup_{\mathrm{Adv}}\{\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n,\tau) - c\cdot \mathrm{Cost}_{\mathrm{Adv}}(A_{n+1},\tau)) \\
&- \mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n,\tau) - c\cdot \mathrm{Cost}_{\mathrm{Adv}}(A_n,\tau))\} \\
&= \sup_{\mathrm{Adv}}\{c\cdot \mathrm{Cost}_{\mathrm{Adv}}(A_n,\tau) - c\cdot \mathrm{Cost}_{\mathrm{Adv}}(A_{n+1},\tau)\} \\
&\le c\cdot \mathrm{dist}(A_n, A_{n+1}).
\end{aligned}$$

We now consider the change in the potential due to Alg's move. The cost for an adversary to serve the request sequence $r\tau$ in some configuration, differs by at most the cost of the task r in that configuration from the cost of serving just τ. Therefore,

$$\begin{aligned}
\mathrm{E}(\Delta\Upsilon) &= \mathrm{E}_r(\Upsilon(m_{n+1}, A_{n+1})) - \Upsilon(m_n, A_{n+1}) \\
&\le \sup_{\mathrm{Adv}}\{\mathrm{E}_r[\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_{n+1},\tau) - c\cdot \mathrm{Cost}_{\mathrm{Adv}}(A_{n+1},\tau))] \\
&- \mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n,r\tau) - c\cdot \mathrm{Cost}_{\mathrm{Adv}}(A_{n+1},r\tau))\} \\
&= \sup_{\mathrm{Adv}}\{\mathrm{E}_r[\mathrm{E}_\tau(\mathrm{Cost}_{\mathrm{Alg}}(m_{n+1},\tau))] - \mathrm{E}_{r\tau}(\mathrm{Cost}_{\mathrm{Alg}}(m_n,r\tau))\} + c\cdot \mathrm{task}(A_{n+1},r) \\
&= c\cdot \mathrm{task}(A_{n+1},r) - \mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(m_n,r)).
\end{aligned}$$

■

4.1 The On-line Steiner Tree Problem

Let G be a weighted graph. An *on-line Steiner tree algorithm* obtains a sequence of vertices $\sigma = v_1, v_2, \ldots, v_\ell$ of the graph G.

In response the Steiner tree algorithm computes subtrees $T_1, T_2, \ldots, T_\ell$ of G, such that for every i, $i = 1, 2, \ldots, \ell$, T_i spans all vertices v_j, $j = 1, 2, \ldots, i$ (and possibly other vertices as well). The subtree T_i must include T_{i-1} as a subgraph. The algorithm must compute T_i independently of vertices v_j, $j > i$.

The configuration of a Steiner tree algorithm is the tree it currently holds. The cost for changing from configuration T_i to configuration T_{i+1} is is defined to be equal to the sum of weights of the edges added to T_i as to obtain T_{i+1}. This is also referred to as the distance between the configurations $\mathrm{dist}(T_i, T_{i+1})$.

For a Steiner tree algorithm Alg, the over an input request sequence σ, denoted $\mathrm{cost}_{\mathrm{Alg}}(\sigma)$, is defined to be the sum of the individual request costs: $\sum_i \mathrm{dist}(T_i, T_{i+1})$. It follows from the definition that this cost is equal to the weight of T_ℓ.

The cost of an optimal adversary is the weight of a minimum Steiner tree spanning all vertices in σ.

Since we are interested in strictly competitive on-line Steiner tree algorithms the word "strictly" is often omitted when discussing the on-line Steiner tree problem.

When required, the superscript St is used to distinguish between Steiner tree dist and cost functions and other dist and cost functions.

Notation. For a weighted graph G, $d(p,q)$ denotes the weight of the minimum weighted path between vertices p and q of G . Where Q is a subset of vertices and p is a vertex of G, $d(Q,p) = \min_{q \in Q}\{d(p,q)\}$.

The k-neighborhood of a vertex v is the set of all vertices u, s.t., $d(u,v) \leq k$. This set is denoted $N_v(k)$.

$T(Q)$ denotes the weight of a minimum Steiner tree spanning the vertices in Q. $T(Q)$ is also used to denote the Steiner tree itself, and the meaning should be clear from the context.

Where S is a tree, $T(S)$ simply denotes the weight of the tree.

Where S is a tree and Q is a subset of vertices in G, $T(S,Q)$ denotes the minimum Steiner expansion of S spanning Q; i.e., the minimum-weighted tree T, such that S is a subtree of T and T spans Q.

The on-line Steiner tree problem is equivalent to a special case of the file allocation problem, where $D = 1$, only read requests are issued, and the algorithm is forced to replicate upon a read request.

Imaze and Waxman [IW] have defined this problem and gave upper and lower bounds for it.

They have shown that the greedy on-line Steiner tree algorithm is $\lceil \log n \rceil$ competitive.

For completeness of the discussion we give here an alternative very simple proof of this claim. Analysis of the greedy Steiner tree algorithm was also independently made by [AA], [ABF1], [CV] and [WY2] giving similar bounds.

The Greedy Steiner Tree Algorithm. The greedy Steiner tree algorithm connects a new point to the closest point already in the tree.

Theorem 4 *The greedy Steiner tree algorithm is strictly* $\lceil \log n \rceil$*-competitive for any weighted graph over* n *vertices.*

Proof. Let $\sigma = v_1, v_2, \ldots, v_\ell$ be the request sequence of vertices. Let A be the minimum Steiner tree spanning all vertices in σ.

Let H be a minimal cycle for the vertices in σ. The weight of H is bounded above by twice the weight of A.

Consider any two adjacent vertices v and u along the cycle H. W.l.o.g let u be the one requested after v, then the cost for serving the request for u is at most $d(u, v)$ which is at most the weight of the path between them in H.

Now divide the vertices into $\lfloor n/2 \rfloor$ pairs of adjacent vertices along the cycle H. This can be done so that the sum of weights of paths between these adjacent vertices will be at most half the total weight of H, and therefore at most the weight of A.

It follows that the cost greedy incurs on the $\lfloor n/2 \rfloor$ requests for a vertex in each of these pairs is at most the weight of A.

The result follows by removing these $\lfloor n/2 \rfloor$ vertices and repeating the process described until one vertex is left. Thus we get that greedy's cost is at most $\lceil \log n \rceil$ times the weight of the minimum Steiner tree A. ∎

5 A File Allocation Algorithm

We present a randomized algorithm for the file allocation problem on all networks, which is competitive against an adaptive on-line adversary.

Let $\mathcal{N}$ be an arbitrary network. Let Alg be a strictly c-competitive *Steiner tree* algorithm on $\mathcal{N}$. We show that Alg can be used to give a competitive randomized file allocation algorithm on $\mathcal{N}$. We assume that the initial configuration consists of one copy of the file at a processor p of $\mathcal{N}$. If that is not the case we start by deleting all copies of the file except one, incurring no cost.

Algorithm **Steiner Based** (SB).
Algorithm SB simulates a version of the Steiner tree algorithm Alg starting with p as the initial configuration. At all times, the set of processors in which SB keeps copies of the file is equal to the set of processors covered by Alg's Steiner tree.

Upon receiving a *read* request initiated at node r, the algorithm serves it, and then with probability $1/D$ feeds Alg with a new request at vertex r . In response Alg computes a new Steiner tree T' in place of its previous tree T. SB *replicates* new copies of the file at the processors corresponding to the vertices that Alg added to its tree.

Upon receiving a *write* request initiated at node w, the algorithm serves it, and with probability $1/\alpha D$ *deletes* all copies of the file, leaving only one copy at the processor closest to w, and then migrates the file to w, initializing a new version of Alg starting at vertex w as its initial configuration.

SB achieves best performance for $\alpha = \sqrt{3}$.

Theorem 5 *If Alg is a strictly c-competitive Steiner tree algorithm against adaptive on-line adversaries on a network $\mathcal{N}$, then* SB *is a* $(2+\sqrt{3})c$*-competitive algorithm for the file allocation problem on $\mathcal{N}$ against adaptive on-line adversaries.*

Proof. Let Υ be the natural potential function for Alg. From Theorem 3, we have that Υ is a strict one-step potential function. We use it to define a new one-step potential function Φ for the Steiner Based algorithm as follows: Let h_n be the history of SB. This history explicitly defines the history of the current version of Alg that SB simulates, denoted $\widehat{h}_n$.

Let σ_n be the sequence of requested vertices already fed to Alg since the last initialization. (we use σ_n to denote the set of these vertices as well). Finally let A denote the adversary's current configuration, let B denote the on-line algorithm's current configuration, and let $\widehat{B}$ denote the on-line Steiner tree algorithm's configuration. The potential function for SB is:

$$\Phi(h_n, A) \quad = \quad \{(\alpha+2)\cdot\overline{\Upsilon}(\widehat{h}_n, A) + \alpha\cdot\Theta(\widehat{B})\}\cdot D,$$

where $\overline{\Upsilon}$ and Θ defined by

$$\begin{aligned} \Theta(\widehat{B}) &= T(\widehat{B}) \\ \overline{\Upsilon}(\widehat{h}_n, A) &= \inf_T\{\Upsilon(\widehat{h}_n, T)\}, \end{aligned}$$

where T ranges over all subtrees of $\mathcal{N}$ such that $\sigma_n \cup A \subseteq T$. (Notice that $\Upsilon(\widehat{h}_n, T)$ is defined for all trees T.)

Clearly Φ is nonnegative as Υ is a potential function, and the weight of a Steiner tree is always nonnegative.

Our proof proceeds by analyzing separately the change in $\overline{\Upsilon}$ due to an adversary change of configuration (an adversary *move*) and the change in $\overline{\Upsilon}$ due to the service of a request by both the adversary and SB, assuming that the adversary (but not SB) does *not* change its configuration, thus accounting for both the on-line and the adversary work following a request.

Throughout, let T_0 denote the subtree that minimizes $\Upsilon(\widehat{h}_n, T)$ before an analyzed event takes place. We think of Alg as playing against a Steiner tree adversary Adv_1 that maintains T_0 as its configuration. We shall bound the change in the potential by extracting a new configuration T_1 for the Steiner tree adversary in the range over which the infimum in $\overline{\Upsilon}$ is taken. The new value of $\overline{\Upsilon}$ will only be less than or equal to the value of Υ on that new configuration.

The following fact, an application of Theorem 3 to the on-line Steiner tree problem, is useful:

Fact 6 *Let T_0, T_1 be trees, such that T_0 is a subtree of T_1. Then, for every history $\widehat{h}_n$ of Alg*

$$\Upsilon(\widehat{h}_n, T_1) - \Upsilon(\widehat{h}_n, T_0) \;\; \leq \;\; c \cdot dist^{\mathrm{St}}(T_0, T_1).$$

Adversary Move.

The adversary replicates or deletes copies of the file changing its configuration from A to A'. The change in potential is

$$\Delta\Phi \;\; = \;\; (\alpha+2)D \cdot \Delta\overline{\Upsilon},$$

since there is no change in the on-line algorithm's configuration, and thus $\Delta\Theta = 0$.

We proceed with the analysis according to the management operation initiated by the adversary:

Replication. Consider a replication initiated by the adversary from processor $p \in A$ to a subset of processors Q (i.e., $A' = A \cup Q$). The cost incurred is $\mathrm{dist}(A, A') = D \cdot T(Q \cup \{p\})$. The Steiner tree adversary,having configuration $T_0 \supseteq \sigma_n \cup A$, can also add to its tree the vertices in Q, ending with a Steiner tree of $\sigma_n \cup A'$, by letting $T_1 = T(T_0, Q)$. Since $p \in T_0$, $dist^{\mathrm{St}}(T_0, T_1) \leq T(Q \cup \{p\})$. Therefore, we can bound the change in potential by

$$\begin{aligned}
\Delta\Phi &= (\alpha+2)D \cdot \Delta\overline{\Upsilon} \\
&\leq (\alpha+2)D \cdot \{\Upsilon(\widehat{h}_n, T_1) - \Upsilon(\widehat{h}_n, T_0)\} \\
&\leq (\alpha+2)D \cdot c \cdot \mathrm{dist}^{\mathrm{St}}(T_0, T_1) \\
&\leq (\alpha+2)c \cdot D \cdot T(Q \cup \{p\}) \\
&= (\alpha+2)c \cdot \mathrm{dist}(A, A').
\end{aligned}$$

Deletion. If the adversary deletes copies of the file, incurring no cost, then $A' \subset A$. Thus we may choose $T_1 = T_0$, so that $T_1 \supseteq \sigma_n \cup A \supseteq \sigma_n \cup A'$. Therefore,

$$\Delta\Phi \;=\; (\alpha+2)D \cdot \Delta\overline{\Upsilon} \;\leq\; (\alpha+2)D \cdot \{\Upsilon(\widehat{h}_n, T_1) - \Upsilon(\widehat{h}_n, T_0)\} \;=\; 0.$$

Request Analysis.

We analyze different request types separately. For any request the change in the potential is bounded above by a constant times the cost of the adversary to serve the request (not including its move cost) minus the expected work done by SB for serving the request and for changing configuration.

Read Request.

The cost of algorithm SB on a read request ρ initiated at a processor r is $d(B, r)$. In case it replicates, its replication cost is exactly D times the cost of Alg on the request at vertex r. Thus its expected cost for ρ is

$$\begin{aligned} \mathrm{E}(\mathrm{Cost}_{\mathrm{SB}}(h_n, \rho)) &= d(B, r) + \frac{1}{D} \cdot D \cdot \mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)) \\ &\leq 2\mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)). \end{aligned}$$

The last inequality follows because the cost of any Steiner tree algorithm, whose current configuration, $\widehat{B}$, is a tree spanning the vertex set B, to serve a request at r, is at least the cost of adding some path from a vertex in B to r, bounded below by $d(B, r)$ (Note that the expectation of Alg's cost is taken only over its own coin tosses).

The probability that SB's configuration is changed is $\frac{1}{D}$. Therefore, with probability $1 - \frac{1}{D}$ the potential function does not change. Therefore,

$$\begin{aligned} \mathrm{E}(\Delta\Phi) &= (1 - \frac{1}{D}) \cdot 0 + \frac{1}{D} \cdot D \cdot \mathrm{E}((\alpha+2) \cdot \Delta\overline{\Upsilon} + \alpha \cdot \Delta\Theta) \\ &= (\alpha+2) \cdot \mathrm{E}(\Delta\overline{\Upsilon}) + \alpha \cdot \mathrm{E}(\Delta\Theta), \end{aligned}$$

where the expected change in $\overline{\Upsilon}$ and Θ is the conditional expected change, in the case that SB decides to replicate, taken only over the coin tosses of Alg.

We proceed with analyzing each of the potential terms. Suppose that the Steiner tree algorithm Alg with the current subtree $\widehat{B}$ changes to a (possibly random) configuration $\widehat{B}'$, following the new request at r. The change in Θ is

$$\mathrm{E}(\Delta\Theta) \;=\; \mathrm{E}(T(\widehat{B}')) - T(\widehat{B}) \;=\; \mathrm{dist}^{\mathrm{St}}(\widehat{B}, \widehat{B}') \;=\; \mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)).$$

We now analyze the change in $\overline{\Upsilon}$ when SB decides to replicate. SB feeds Alg with a new request at vertex r, and therefore the new history $\widehat{h}_{n+1}$ of Alg consists of the request sequence $\sigma_{n+1} = \sigma_n r$. Let the Steiner tree adversary, having current configuration $T_0 \supseteq \sigma_n \cup A$ add to its tree the minimal path from the closest vertex to r in A, incurring cost $d(A,r)$ and ending with a Steiner tree $T_1 = T(T_0, \{r\}) \supseteq \sigma_{n+1} \cup A$. Using that Υ is a one-step (and therefore also a two-step) potential function for Alg, we obtain that

$$
\begin{aligned}
\mathrm{E}(\Delta\overline{\Upsilon}) &\leq \mathrm{E}[\Upsilon(\widehat{h}_{n+1}, T_1)] - \Upsilon(\widehat{h}_n, T_0) \\
&\leq c \cdot \mathrm{Cost}^{\mathrm{St}}_{\mathrm{Adv}}(T_0, r) - \mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)) \\
&= c \cdot d(A,r) - \mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)).
\end{aligned}
$$

As $\mathrm{Cost}_{\mathrm{Adv}}(A,\rho) = d(A,r)$, we conclude that the expected change in Φ is:

$$
\begin{aligned}
\mathrm{E}(\Delta\Phi) &= (\alpha+2)\cdot \mathrm{E}(\Delta\overline{\Upsilon}) + \alpha \cdot \mathrm{E}(\Delta\Theta) \\
&\leq (\alpha+2)\cdot \{c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A,\rho) - \mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)\} + \alpha \cdot \mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)) \\
&= (\alpha+2)c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A,\rho) - 2\mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, r)) \\
&\leq (\alpha+2)c \cdot \mathrm{Cost}_{\mathrm{Adv}}(A,\rho) - \mathrm{E}(\mathrm{Cost}_{\mathrm{SB}}(h_n,\rho)).
\end{aligned}
$$

Write Request.

We follow the same steps as in the analysis of the read request. The cost of SB on a write request ω initiated at processor w consists of the cost of the write $T(B \cup \{w\})$, and in case SB decides to delete, it also pays the cost of the migration. Therefore SB's expected cost is

$$
\begin{aligned}
\mathrm{E}(\mathrm{Cost}_{\mathrm{SB}}(h_n,\omega)) &= T(B \cup \{w\}) + \frac{1}{\alpha D} \cdot D \cdot d(B,w) \\
&\leq T(B) + \tfrac{\alpha+1}{\alpha} \cdot d(B,w).
\end{aligned}
$$

As $\widehat{B}$ spans B, $T(B)$ is a lower bound on $T(\widehat{B})$, it now follows that

$$
\mathrm{E}(\mathrm{Cost}_{\mathrm{SB}}(h_n,\omega)) \leq T(\widehat{B}) + \tfrac{\alpha+1}{\alpha} \cdot d(B,w).
$$

Since SB changes its configuration only with probability $\frac{1}{\alpha D}$, we have that the potential function does not change with probability $1 - \frac{1}{\alpha D}$. Therefore,

$$
\begin{aligned}
\mathrm{E}(\Delta\Phi) &= (1 - \frac{1}{\alpha D}) \cdot 0 + \frac{1}{\alpha D} \cdot D \cdot \{(\alpha+2)\cdot \Delta\overline{\Upsilon} + \alpha \cdot \Delta\Theta\} \\
&= \tfrac{\alpha+2}{\alpha} \cdot \Delta\overline{\Upsilon} + \Delta\Theta.
\end{aligned}
$$

As before, the change in Υ and Θ is the conditional change, in case that SB deletes.

Since with probability $1/\alpha D$ the new configuration of Alg is $\{w\}$ and its new history, denoted $\widehat{h}_w$, consists of a single request at w, we have

$$\begin{aligned}\Delta\Theta &= T(\{w\}) - T(\widehat{B}) = -T(\widehat{B});\\ \Delta\overline{\Upsilon} &\leq \Upsilon(\widehat{h}_w, T(A\cup\{w\})) - \Upsilon(\widehat{h}_n, T_0).\end{aligned}$$

This follows because a new version of Alg is initialized, and the new Steiner tree adversary can obviously choose $T_1 = T(A\cup\{w\})$ as its configuration in order to cover the vertices in A and w.

Suppose that instead of initializing a new version of Alg with initial configuration w, Alg were to receive a new request at w, resulting with the fictitious history $\widehat{h}_{n+1}$. Following the read request analysis, the Steiner tree adversary can choose the configuration $T_1' = T(T_0, \{w\})$ so that

$$\begin{aligned}\mathrm{E}[\Upsilon(\widehat{h}_{n+1}, T_1')] - \Upsilon(\widehat{h}_n, T_0) &\leq c\cdot \mathrm{Cost}^{\mathrm{St}}_{\mathrm{Adv}}(T_0, w) - \mathrm{E}(\mathrm{Cost}^{\mathrm{St}}_{\mathrm{Alg}}(\widehat{h}_n, w))\\ &\leq c\cdot d(A,w) - d(B,w).\end{aligned}$$

Alg is strictly competitive and, hence by Theorem 3, $\Upsilon(\widehat{h}_w, \{w\}) = \Upsilon_0 = 0$. Therefore,

$$\begin{aligned}\Upsilon(\widehat{h}_w, T(A\cup\{w\})) &= \Upsilon(\widehat{h}_w, T(A\cup\{w\})) - \Upsilon(\widehat{h}_w, \{w\})\\ &\leq c\cdot \mathrm{dist}^{\mathrm{St}}(\{w\}, T(A\cup\{w\})) = c\cdot T(A\cup\{w\}).\end{aligned}$$

Since Υ is nonnegative we obtain

$$\begin{aligned}\Delta\overline{\Upsilon} &\leq \Upsilon(\widehat{h}_w, T(A\cup\{w\})) + \tfrac{\alpha+1}{\alpha+2}\cdot\{\mathrm{E}[\Upsilon(\widehat{h}_{n+1}, T_0')] - \Upsilon(\widehat{h}_n, T_0)\}\\ &\leq c\cdot T(A\cup\{w\}) + \tfrac{\alpha+1}{\alpha+2}\cdot\{c\cdot d(A,w) - d(B,w)\}.\end{aligned}$$

Clearly, $d(A,w) \leq T(A\cup\{w\})$, since a Steiner tree spanning $A\cup\{w\}$ includes some path from a vertex in A to w. Hence,

$$\Delta\overline{\Upsilon} \leq \tfrac{2\alpha+3}{\alpha+2}c\cdot T(A\cup\{w\}) - \tfrac{\alpha+1}{\alpha+2}\cdot d(B,w).$$

The cost of the write request to the adversary is $\mathrm{Cost}_{\mathrm{Adv}}(A,\omega) = T(A\cup\{w\})$. We conclude that

$$\begin{aligned}\mathrm{E}(\Delta\Phi) &= \tfrac{\alpha+2}{\alpha}\cdot\Delta\overline{\Upsilon} + \Delta\Theta\\ &\leq \tfrac{2\alpha+3}{\alpha}c\cdot T(A\cup\{w\}) - \tfrac{\alpha+1}{\alpha}\cdot d(B,w) - T(\widehat{B})\\ &\leq \tfrac{2\alpha+3}{\alpha}c\cdot \mathrm{Cost}_{\mathrm{Adv}}(A,\omega) - \mathrm{E}(\mathrm{Cost}_{\mathrm{SB}}(h_n,\omega)).\end{aligned}$$

Summarizing the above case analysis, SB is $\max\{\frac{2\alpha+3}{\alpha}, \alpha+2\}\cdot c$-competitive against the adaptive on-line adversary; $\max\{\frac{2\alpha+3}{\alpha}, \alpha+2\}$ has its minimum at $\alpha = \sqrt{3}$. ∎

The competitive ratio in Theorem 5 is best possible up to a constant factor for any network as follows from the lower bound given in Theorem 23.

Note that, although the cost incurred by the Steiner-Based algorithm for serving a write request initiated at w is assumed to be the optimal inherent cost; i.e., the weight of a minimum Steiner tree spanning w and all processors holding a copy of the file (i.e., $T(B \cup \{w\})$), the proof holds even if we assume that the on-line cost is the minimum path length from w to the current configuration plus the weight of the on-line Steiner tree that Alg maintains (i.e., $T(\widehat{B}) + d(B, w)$). This variation is required for the analysis of the distributed version of the algorithm in Section 9.4.

In order to explicitly characterize the competitive ratio for the file allocation problem on a variety of networks, we present the following results on the competitive ratio of the on-line Steiner tree problem.

Define the greedy on-line Steiner tree algorithm as follows: Given a request at vertex v, greedy adds to its current subtree the shortest path in G from a vertex in its subtree to v.

As stated in Section 4.1 Imase and Waxman [IW] prove that the greedy steiner tree algorithm is $\lceil \log n \rceil$-competitive. In fact in [ABF1, WY2] are different proof proving the following

Theorem 7 *For any weighted graph G on n nodes, the greedy Steiner tree algorithm is $O(\min\{\log n, \log(Diam$ competitive.*

We also have the following easy to verify facts.

Fact 8 *The greedy Steiner tree algorithm is 1-competitive for trees and for uniform complete graphs.*

Fact 9 *The greedy Steiner tree algorithm is 2-competitive for the ring.*

Applying Theorem 5 we conclude

Theorem 10 *For every network on n processors,* SB *using greedy as a Steiner tree algorithm is an $O(\min\{\log n, \log(Diam)\})$-competitive file allocation algorithm against adaptive on-line adversaries. It is $O(1)$-competitive for processors on a ring, for trees, and for uniform networks.*

Finally, the result of [BBKTW] implies the following corollary.

Corollary 11 *For every network on n processors, if there exists a strictly c-competitive Steiner tree algorithm against adaptive on-line adversaries on $\mathcal{N}$, then there exists a computable deterministic $O(c^2)$-competitive algorithm for the file allocation problem. In particular there exists a computable deterministic $O(1)$-competitive algorithm for processors on a ring.*

Proof. The corollary follows from the fact that for any finite network $\mathcal{N}$, the natural potential function for any Steiner tree algorithm for $\mathcal{N}$ is computable. ■

6 Uniform Networks

A uniform network topology is one where the underlying graph is the complete graph with all edge weights equal to 1.

The file allocation problem in a uniform network defines the following costs for an algorithm:

The cost of a read at processor p is 0 if p contains a copy of the file and 1 otherwise.

The cost of a write at processor p equals to the number of file copies in the network if p does not contain a copy of th file. If p contains a copy of the file then the write's cost is equal to the number of file copies in the network minus 1.

The cost for a file replication over an edge is D.

We give an optimal deterministic algorithm for file allocation on uniform network topologies. We present an optimal 3-competitive, deterministic file allocation algorithm for uniform networks.

Let P denote the set of processors in the network.

Algorithm **Count.**
Count is defined for each processor $p \in P$ separately. It maintains a counter c, and performs the following algorithm. We say that Count is *waiting*, if there is a single copy of the file and the processor holding the file is performing step 4 of the algorithm. Initially, set $c := 0$. If p holds a copy of the file, begin at step 4.

1. While $c < D$, if a *read* is initiated by p, or if a *write* is initiated by p, and Count is waiting, increase c by 1.
2. *Replicate* a copy of the file to p.
3. While $c > 0$, if a *write* is initiated by any other processor, decrease c by 1.
4. If p holds the last copy of the file, wait until it is replicated by some other processor.
5. *Delete* the copy held by p.
6. Repeat from step 1.

Theorem 12 *Algorithm* Count *is 3-competitive for uniform networks.*

Proof. Fix a processor p. One iteration of steps 1–6 at p is named a *phase*. Note that if Count is waiting then it is executing step 4 in the single processor holding a copy of the file, and it is executing step 1 in all the other processors. Count's cost is charged on individual processors as follows:

1. A processor initiating a read is charged the cost of the read.
2. If Count is waiting, a processor initiating a write is charged the cost of the write.
3. If Count is not waiting, and a write is initiated, the cost of 1 is charged at each processor holding a copy, except for the initiating processor. Note that the sum of costs charged here is exactly the cost of that write.
4. The cost D of replicating is charged at the processor receiving the copy.

The adversary's cost is charged the same, except that a replication is not charged. Rather, it registers a debit of D at the processor receiving the copy. That debit is paid (and a cost of D is charged) when the copy is deleted. Debits are initially set to 0 for processors *not* holding copies and to D for processors that initially do hold a copy. Note that the charging of the adversary's cost minus the sum of initial debits is a lower bound on its actual cost, because at the end of the sequence some processors may have positive debit.

At the beginning Count is waiting after all but one copy are deleted, so that no cost is incurred. Now, during a phase of a processor p, Count's cost charged to p is at most $3D$. Steps 1 and 2 cause a charge of D each. Step 3 causes a charge of D. The total cost of Count is the sum of costs over all phases of all processors. There can be at most n partial phases (which are not over).

The adversary's cost during a full phase (note that the duration of a phase is determined by Count) is at least D. If the adversary ever deletes a copy from the processor during a phase, it is charged D. Otherwise, it either holds a copy at that processor when Count begins step 3 (and therefore not waiting), so it pays D during that step; or, it does not hold a copy at the end of step 1, and since it could not delete during that step, it must have been charged at least D for the requests of step 1. The reason is that during step 1 the processor initiated a total of D requests, counting read requests and write requests initiated while Count was waiting. ■

7 Tree Networks

A tree network topology is one where the underlying graph is a tree.

We give an optimal randomized memoryless algorithm against adaptive on-line adversaries for file allocation on tree network topologies.

We then give an optimal deterministic memoryless algorithm for continuous trees, and finally show how to use the ideas in that algorithm to obtain a nearly optimal deterministic algorithm for discrete trees.

For the proofs of the algorithms we need the following definitions.

Definition. Let x, y, and z be points on a tree.

The *path* between x and y is denoted $P(x, y)$.

The *slack* of x, y, and z is define by

$$s(x, y, z) \;=\; \tfrac{1}{2}(d(x, y) + d(x, z) - d(y, z)).$$

The subsequent lemma is a simple consequence of the slack definition.

Lemma 13 *The slack of x, y, and z has the following properties:*

$$s(x, y, z) \;=\; \begin{cases} d(x, y) & \textit{if } y \in P(x, z) \\ d(x, z) & \textit{if } z \in P(x, y) \\ 0 & \textit{if } x \in P(y, z) \end{cases} \tag{3}$$

$$s(x, y, z) + d(y, z) \;\geq\; \max\{d(x, y), d(x, z), d(y, z)\} \tag{4}$$

$$s(x, y, z) - s(y, x, z) \;=\; d(x, z) - d(y, z). \tag{5}$$

7.1 A 3-Competitive Randomized Memoryless Algorithm

Randomized Trees Algorithm (RT).
At all times the algorithm maintains a subtree of the processors holding copies of the file.

Upon receiving a *read* request initiated at node r, the algorithm serves it. Let b be the closest processor to r holding a copy of the file. With probability $1/D$ *replicates* copies of the file along the edge from b to r.

Upon receiving a *write* request initiated at node w, the algorithm serves it, and with probability $1/D$ *deletes* all copies of the file, leaving only one copy at the processor closest to w, and then with probability $1/2$, *migrates* the file to w.

Theorem 14 *Algorithm* RT *is* 3*-competitive for file-allocation on trees against adaptive on-line adversaries.*

Proof. We give a two-step potential function proof. Let B denote the on-line configuration (i.e., the set of processors holding a copy of the file). Let A denote the adversary configuration. The potential function for RT is

$$\Phi \;=\; D \cdot \{3 \cdot (T(A \cup B) - T(B)) + T(B)\}.$$

We will analyze separately the potential components:

$$\begin{aligned}\Psi &= D\cdot(T(A\cup B)-T(B));\\ \Theta &= D\cdot T(B),\end{aligned}$$

so that $\Phi = 3\Psi + \Theta$.

Our proof proceeds by analyzing separately the change in Φ due to an adversary configuration change and the change in Φ due to the service of a request by both the adversary and the algorithm, assuming that the adversary does *not* change its configuration, thus accounting for both the on-line and the adversary work following a request.

Φ is clearly nonnegative since $T(B) \le T(A\cup B)$.

We start with analyzing configuration changes made by the adversary.

Adversary Move.

Clearly the only potential component that changes is $T(A\cup B)$.

A deletion made by the adversary can only decrease it.

If a replication is made by the adversary from some node $p \in A$ to q, resulting with a new configuration A', at a cost $D\cdot d(p,q)$, then since $T(A'\cup B) \le T(A\cup B) + d(p,q)$ we get

$$\Delta\Phi = 3D\cdot(T(A'\cup B) - T(A\cup B)) \le 3D\cdot d(p,q).$$

Request Analysis.

We analyze reads and writes separately. For any request we show that the change in the potential function is bounded above by 3 times the the adversary's cost for serving the request (not including its move cost) minus the expected cost RT incurs for both serving the request and for changing configuration.

Read Request.

Consider a read request ρ initiated at a processor r. Let b be the closest processor to r in B. Let a be the closest processor to r in $T(A)$.

The cost incurred by algorithm RT on the request at r is $d(b,r)$. In case it replicates, its replication cost is $D\cdot d(b,r)$. Thus its expected cost for ρ is

$$\begin{aligned}\mathrm{E}(\mathrm{Cost}_{\mathrm{RT}}(\rho)) &= d(b,r) + \frac{1}{D}\cdot D\cdot d(b,r)\\ &= 2d(b,r).\end{aligned}$$

The probability that RT replicates is $\frac{1}{D}$. Therefore, with probability $1-\frac{1}{D}$ the potential function does not change.

The increase in Θ in the case RT replicates is equal to the length of the replication path:

$$\begin{aligned} \mathrm{E}(\Delta\Theta) &= (1-\frac{1}{D})\cdot 0 + \frac{1}{D}\cdot D \cdot d(b,r) \\ &= d(b,r). \end{aligned}$$

We now turn to analyzing the change in Ψ.

Claim 15 *Given an arbitrary point r, let b be the closest point to r in B, and let a be the closest point to r in $T(A)$. The change in Ψ for a replication from b to r is*

$$\Delta\Psi = -D\cdot s(b,a,r).$$

Proof of Claim 15. If $a \in P(b,r)$ then $T(A\cup B)$, but not $T(B)$, has included the path from b to a before the replication. After the replication $T(B)$ includes this path and therefore Ψ decreases by $D\cdot d(b,a)$.

Similarly if $r \in P(b,a)$ then $T(A\cup B)$, but not $T(B)$, has included the path from b to r before the replication. After the replication $T(B)$ includes this path and therefore Ψ decreases by $D\cdot d(b,r)$.

Finally if $b \in P(a,r)$ then both $T(B)$ and $T(A\cup B)$ increase by the same distance, and thus $\Delta\Psi = 0$.

The claim follows from property (3) of Lemma 13. ■

Using Claim 15 and property (4) of Lemma 13 we get

$$\begin{aligned} \mathrm{E}(\Delta\Psi) &= (1-\frac{1}{D})\cdot 0 + \frac{1}{D}\cdot(-D\cdot s(b,a,r)) \\ &= -s(b,a,r) \\ &\le d(a,r) - d(b,r). \end{aligned}$$

Therefore the expected change in Φ is

$$\begin{aligned} \mathrm{E}(\Delta\Phi) &= \mathrm{E}(3\Delta\Psi + \Delta\Theta) \\ &\le 3(d(a,r)-d(b,r)) + d(b,r) \\ &= 3d(a,r) - 2d(b,r) \\ &= 3\cdot \mathrm{Cost}_{\mathrm{Adv}}(\rho) - \mathrm{E}(\mathrm{Cost}_{\mathrm{RT}}(\rho)). \end{aligned}$$

Write Request.

Consider a write request ω initiated at a processor w. Let b be the closest processor to w in B. Let a be the closest processor to w in $T(A)$.

The cost incurred by algorithm RT on the write request at w is $T(B \cup \{w\})$. In case RT decides to delete all copies and to migrate the last file copy, it incurs an additional cost for the migration of $D \cdot d(b, w)$. Thus its expected cost for ω is

$$\begin{aligned} \mathrm{E}(\mathrm{Cost}_{\mathrm{RT}}(\omega)) &= T(B \cup \{w\}) + \frac{1}{2D} \cdot D \cdot d(b, w) \\ &\leq (T(B) + d(b, w)) + \tfrac{1}{2} d(b, w) \\ &= T(B) + \tfrac{3}{2} d(b, w). \end{aligned}$$

Since RT deletes all file copies but one with probability $1/D$ we have

$$\begin{aligned} \mathrm{E}(\Delta\Theta) &= (1 - \frac{1}{D}) \cdot 0 + \frac{1}{D} \cdot (-D \cdot T(B)) \\ &= -T(B). \end{aligned}$$

We analyze the change in Ψ separately for the change in the case that RT decides to delete all copies but not to migrate the last file copy, denoted $\Delta\Psi^1$, and the change in the case that RT decides to delete all copies and to migrate the last file copy to w, denoted $\Delta\Psi^2$.

We need another claim.

Claim 16 *Given an arbitrary point w, let b be the closest point to w in B. The change in Ψ for a deletion of all copies in B except for one at b is*

$$\Delta\Psi^1 \leq D \cdot (T(A) + s(a, b, w)).$$

Proof of Claim 16. For any edge, Ψ may increase by D times its weight if that edge belongs to $T(A \cup \{b\})$ but not to $T(A \cup B) \setminus T(B)$.

Consider the set of edges in $S = T(A \cup \{b\}) \setminus T(A)$.

If $a \in P(b, w)$ then S is empty. Otherwise S includes the path from a to b, $P(a, b)$.

But if $w \in P(a, b)$ and $w \neq a, w \neq b$, then $P(a, b)$ belongs to $T(A \cup B) \setminus T(B)$ as well.

Therefore Ψ increases by at most $D \cdot T(A)$ plus $D \cdot d(a, b)$ when $b \in P(a, w)$ and otherwise Ψ increases by at most $D \cdot T(A)$, and thus by at most $D \cdot (T(A) + s(a, b, w))$. ■

To analyze the change in Ψ in the case RT deletes all copies and migrates the last file copy to w, we look at the configuration change as obtained by first replicating new file copies along the path from b to w and then deleting all file copies except the one at w.

Claim 15 implies the change in Ψ for the imaginary replication from b to w is $-D \cdot s(b, a, w)$. For the deletion the increase in Ψ can be at most its final value $D \cdot T(A \cup \{w\})$. Thus we get

$$\Delta\Psi^2 \leq D \cdot (T(A \cup \{w\}) - s(b, a, w)).$$

Therefore, using property 5 of Lemma 13 we obtain that the total expected change in Ψ is

$$\begin{aligned}
\mathrm{E}(\Delta\Psi) &= (1 - \frac{1}{D}) \cdot 0 + \frac{1}{2D} \cdot (\Delta\Psi^1) + \frac{1}{2D} \cdot (\Delta\Psi^2) \\
&\leq \tfrac{1}{2}(T(A) + s(a, b, w)) + \tfrac{1}{2}(T(A \cup \{w\}) - s(b, a, w)) \\
&= T(A) + \tfrac{1}{2}d(a, w) + \tfrac{1}{2}(s(a, b, w) - s(b, a, w)) \\
&= T(A) + \tfrac{1}{2}d(a, w) + \tfrac{1}{2}(d(a, w) - d(b, w)) \\
&= T(A \cup \{w\}) - \tfrac{1}{2}d(b, w).
\end{aligned}$$

Therefore the expected change in Φ for the write request ω is

$$\begin{aligned}
\mathrm{E}(\Delta\Phi) &= \mathrm{E}(3\Psi + \Theta) \\
&\leq 3(T(A \cup \{w\}) - \tfrac{1}{2}d(b, w)) - T(B) \\
&= 3 \cdot T(A \cup \{w\}) - (T(B) + \tfrac{3}{2}d(b, w)) \\
&= 3 \cdot \mathrm{Cost}_{\mathrm{Adv}}(\omega) - \mathrm{E}(\mathrm{Cost}_{\mathrm{RT}}(\omega)).
\end{aligned}$$

This concludes the proof of the theorem. ■

7.2 A 3-Competitive Determistic Algorithm for Continueous Trees

We next give a 3-competitive algorithm for *continuous trees*. The proof uses the same potential function we have used to prove the randomized algorithm in section 7.1. Our deterministic algorithm makes actions that affect the potential function at most as much as in the randomized case.

At all times our algorithm maintains a subtree B of the processors holding copies of the file. We describe how this subtree is changed in response to each request.

In our algorithm we use a *deleting procedure* to perform two deletion operations:

1. A *fraction-deletion* of an α-fraction of the weight of a subtree from the leaves towards some node in the subtree.

2. A *fixed-deletion* of some fixed quantity from the subtree from the leaves towards some leaf of the subtree.

This procedure will be described following the description of the algorithm.

Continuous Trees Algorithm (CT).
Upon receiving a *read* request initiated at node r, serve it, then let b be the nearest point in B to the request position r. *Enlarge* the subtree B along the path from b towards r a distance of $\frac{1}{D}d(b,r)$.

Upon receiving a *write* request initiated at processor w, serve it, use a fraction-deletion to *delete* an overall of $1/D$-fraction of the weight of the subtree B from the leaves towards w. The perform a *tree migration* as follows: Let b be the nearest point in B to the request position w. *Enlarge* the subtree B along the path from b towards w a distance of $\frac{1}{2D}d(b,w)$, Let B' denote the new CT configuration. Let b' denote the nearest point in B' to w. Use a fixed-deletion to *delete* an overall weight of $d(b',b)$ from the subtree B' from the leaves towards b'.

The deleting procedure :

1. *Fixed-deletion* of an overall weight of x from an edge (v,w) towards some node u. Assume w.l.o.g that $v \in P(u,w)$. This is defined by deleting from w towards v a weight of $\min\{x, d(v,w)\}$. If $x > d(v,w)$ we say the deletion modulo equals $x - d(v,w)$, otherwise it is 0.

2. *Fixed-deletion* of an overall weight of x from a subtree S towards some leaf, u, of S. This is defined recursively as follows: If S contains a single edge use operation (1) of this procedure. Otherwise let v be the node adjacent to u. While $x > 0$, repeat for each branch rooted at v, recursively deleting an overall of x from the branch towards v, and set x to be equal to the deletion modulo. Finally if $x > 0$ use operation (1) of this procedure to delete a weight of x from (u,v).

3. *Fraction-deletion* of an overall of an α-fraction ($\alpha < 1$) of the weight of a subtree S from the leaves towards a node u in S. This is defined recursively as follows: If u is not a leaf of S then for each branch rooted at u, recursively delete an overall α-fraction of the weight of that branch towards u. Otherwise, u is a leaf of S. If S contains a single edge (u,v) use operation (1) of this procedure to delete $\alpha \cdot d(u,v)$ from the edge weight towards u. Otherwise let v be the node adjacent to u. For each branch rooted at v, recursively delete an overall α-fraction of the weight of that branch towards v. Finally use operation (2) of this procedure to delete $\alpha \cdot d(u,v)$ from the subtree weight towards u.

Theorem 17 *Algorithm* CT *is* 3*-competitive for file-allocation on continuous trees.*

Proof. Let B denote the on-line configuration. Let A denote the adversary configuration. We use the same potential function function as in the proof for the randomized algorithm in the proof of Theorem 14.

$$\Phi \;=\; D \cdot \{3 \cdot (T(A \cup B) - T(B)) + T(B)\}.$$

As before we will analyze separately the potential components:

$$\begin{aligned} \Psi &= D \cdot (T(A \cup B) - T(B)) \\ \Theta &= D \cdot T(B), \end{aligned}$$

so that $\Phi = 3\Psi + \Theta$.

From the proof of Theorem 14 we have that Φ is nonnegative and that the change in Φ over the configuration changes made by the adversary is at most 3 times the adversary cost for those changes.

Thus we turn to the analysis of the read and write requests.

Request Analysis.

Read Request.

Consider a read request ρ initiated at a processor r. Let b be the closest processor to r in B. Let a be the closest processor to r in $T(A)$.

The cost incurred by algorithm CT on the request at r is

$$\begin{aligned} \mathrm{Cost}_{\mathrm{CT}}(\rho) &= d(b,r) + D \cdot \frac{1}{D} d(b,r) \\ &= 2d(b,r). \end{aligned}$$

The increase in Θ is equal to the length of the replication path:

$$\Delta\Theta \;=\; D \cdot \tfrac{1}{D} d(b,r) \;=\; d(b,r).$$

We now turn to analyzing the change in Ψ.

Let c be the point at distance $\frac{1}{D}d(b,r)$ from b towards r. Claim 15 implies that if a_c is the closest point to c in $T(A)$ then:

$$\Delta\Psi \;=\; -D \cdot s(b, a_c, c).$$

We have the following claim:

Claim 18 *Given an arbitrary point r, let b be the closest point to r in B, and let a_r be the closest point to r in $T(A)$. Let c be some point in $P(b,r)$, and let a_c be the closest point to c in $T(A)$. Let $\lambda = d(b,c)/d(b,r)$.*

$$s(b, a_c, c) \;\; \geq \;\; \lambda \cdot s(b, a_r, r).$$

Proof of Claim 18. If $b \in P(a_c, c)$ then $a_c = a_r$ and hence also $b \in P(a_r, r)$ and therefore $s(b, a_c, c) = s(b, a_r, r) = 0$.

If $c \in P(b, a_c)$ then we have

$$s(b, a_c, c) \;\; = \;\; d(b,c) \;\; = \;\; \lambda \cdot d(b,r) \;\; \geq \;\; \lambda \cdot s(b, a_r, r).$$

Finally, if $a_c \in P(b,c)$ then $a_c = a_r$ and therefore

$$s(b, a_c, c) \;\; = \;\; d(b, a_c) \;\; \geq \;\; \lambda \cdot d(b, a_c) \;\; = \;\; \lambda \cdot s(b, a_r, r).$$

■

Thus using Claim 18 we conclude that the change in Ψ is at most as in the proof of Theorem 14:

$$\Delta\Psi \;\; = \;\; -D \cdot s(b, a_c, c) \;\; \leq \;\; -s(b, a, r).$$

The rest of the read request analysis is the same as in the proof of Theorem 14.

Write Request.

Consider a write request ω initiated at a processor w. Let b be the closest processor to w in B. Let a be the closest processor to w in $T(A)$.

The cost incurred by algorithm CT for the write request at w consists for the cost of the write plus an additional cost for the replication during the *tree migration.*

$$\begin{aligned} \mathrm{Cost}_{\mathrm{CT}}(\omega) &= T(B \cup \{w\}) + D \cdot \frac{1}{2D} d(b,w) \\ &\leq (T(B) + d(b,w)) + \tfrac{1}{2} d(b,w) \\ &= T(B) + \tfrac{3}{2} d(b,w). \end{aligned}$$

Algorithm CT deletes an overall $1/D$-fraction of the weight of the subtree B. Then in the tree migration it first enlarges the subtree a distance of $\frac{1}{2D} d(b,w)$ and then deletes the same weight of the weight of the new subtree. Thus overall the tree migration does not change the weight of B. Therefore

$$\Delta\Theta \;\; = \;\; -D \cdot \tfrac{1}{D} T(B) \;\; = \;\; -T(B)$$

as in the proof of Theorem 14.

We now turn to analyze the change in Ψ over the configuration changes made by CT. Again, we would like to show that this change is at most the expected change for algorithm RT, analyzed in the previous section.

We have the following claims:

Claim 19 *Consider a deletion made by the algorithm from a leaf v of B to a node u in B. Let a_v denote the nearest point to v in $T(A)$. Then*

$$\Delta\Psi \quad = \quad s(u, a_v, v).$$

Proof of Claim 19. If $v \in P(u, a_v)$ then $T(A \cup B)$ does not change while $T(B)$ increases by exactly $d(u, v)$, and thus $\Delta\Psi = d(u, v)$.

If $u \in P(a_v, v)$ then both $T(B)$ and $T(A \cup B)$ decrease by $d(u, v)$ so that $\Delta\Psi = 0$.

Finally, if $a_v \in P(u, v)$ then while $T(B)$ decreases by $d(u, v)$, $T(A \cup B)$ decreases by only $d(a_v, v)$ and therefore $\Delta\Psi = d(u, a_v)$.

Therefore the claim follows from property 3 of Lemma 13. ■

Claim 20 *Consider some edge (u, v) of B. Let y be point in B such that $v \in P(u, y)$ and let x be a node at distance $\lambda \cdot d(u, v)$ from y towards u ($0 \le \lambda \le 1$). Let a_v denote the nearest point to v in $T(A)$, and let a_y denote the nearest point to y in $T(A)$. Then*

$$s(x, a_y, y) \quad \le \quad \lambda \cdot s(u, a_v, v).$$

Proof of Claim 20. If $v \in P(u, a_v)$ then

$$s(u, a_v, v) \quad = \quad d(u, v) \quad = \quad \tfrac{1}{\lambda} d(x, y) \quad = \quad \tfrac{1}{\lambda} s(x, a_y, y).$$

Otherwise, since $v \in P(u, y)$, we have that $a_v = a_y$.

If $x \in P(a_v, v)$ then since $v \in P(u, y)$ we get that also $x \in P(a_y, y)$ Thus in this case $s(x, a_y, y) = 0$.

Finally, if $a_v \in P(x, v)$ then since x is at distance $\lambda d(u, v) \le d(u, v)$ from y towards u we get that also $a_v \in P(u, v)$. Therefore

$$\begin{aligned} s(x, a_y, y) &\le d(x, a_y) \\ &= d(x, y) - d(a_y, y) \end{aligned}$$

$$
\begin{aligned}
&\leq \lambda \cdot d(u,v) - d(a_v,v)\\
&\leq \lambda \cdot (d(u,v) - d(a_v,v))\\
&= \lambda \cdot d(v,a_v)\\
&= \lambda \cdot s(u,a_v,v).
\end{aligned}
$$

■

We now would like to bound the change in Ψ by comparing it to the appropriate expected change for RT analyzed in the proof of Theorem 14.

Consider an algorithm C_1 with configuration B that deletes all file copies leaving one at b. Viewing the deletion of copies as done edge by edge starting with leaves towards b, CT performs a fraction deletion that is defined so that for every edge (u,v) of B deleted by C_1 from v to u, CT deletes an exact weight of $\frac{1}{D}d(u,v)$ from some point y in B such that $v \in P(u,y)$ towards u.

Consider only a deletion of $\frac{1}{2D}d(u,v)$ made by CT for each edge, it follows from Claim 19 and Claim 20 that the change in Ψ for the deletions by CT is at most $\frac{1}{2D}$ times the change for the deletions by C_1.

From Claim 16 we get that the change in Ψ for these deletion operations is

$$
\begin{aligned}
\Delta\Psi^1 &\leq \frac{1}{2D} \cdot D \cdot (T(A) + s(a,b,w))\\
&= \tfrac{1}{2}(T(A) + s(a,b,w)).
\end{aligned}
$$

Consider now an algorithm C_2 that first replicates from b along the path to w. The appropriate operation made by CT is the tree enlargement from b towards w a distance of $\frac{1}{2D}d(b,w)$.

Using Claim 18 we have that if c is the nearest point to w after CT's replication, and a_c the nearest point in A to c then

$$
\Delta\Psi^2 = -\tfrac{1}{2D}D \cdot s(b,a_c,c) \leq -\tfrac{1}{2}s(b,a,w).
$$

Now let C_2 delete all file copies except the one at w. Again, viewing the deletion as done from the leaves through b and upto w, CT performs a fraction deletion that is defined so that for every edge (u,v) in B deleted by C_2 from v to u, CT deletes an exact weight of $\frac{1}{D}d(u,v)$ from some point y in B such that $v \in P(u,y)$ towards u, and for every edge (u,v) in $P(b,w)$ deleted by C_2 from v to u, CT deletes an exact weight of $\frac{1}{2D}d(u,v)$ from some point y in B such that $v \in P(u,y)$ towards u.

Again considering a deletion of only $\frac{1}{2D}d(u,v)$ for each edge made by CT, we obtain from Claim 19 and Claim 20 that the change in Ψ for the deletions by CT is at most $\frac{1}{2D}$ times the change for the deletions by C_2, therefore

$$\Delta\Psi^3 \;\leq\; \tfrac{1}{2D} \cdot D \cdot T(A \cup \{w\}) \;=\; \tfrac{1}{2}T(A \cup \{w\}).$$

Since all configuration changes made by CT have been accounted for we get that the total change in Ψ is

$$\begin{aligned}\Delta\Psi &= \Delta\Psi^1 + \Delta\Psi^2 + \Delta\Psi^3 \\ &\leq \tfrac{1}{2}(T(A) + s(a,b,w)) + \tfrac{1}{2}(T(A \cup \{w\}) - s(b,a,w))\end{aligned}$$

which is at most the expected change for RT.

The rest of the write request analysis is the same as in the proof of Theorem 14. ■

We now show that the continuous trees algorithm CT yields a very simple deterministic algorithm for discrete trees.

Given a continuous trees algorithm CO we define discrete trees algorithm DI as follows: DI simulates CO on the input request sequence. We may assume CO always keeps a tree configuration, T, since it only reduces its cost. DI holds file copies at all processors included in T and may also have copies at the nodes adjacent to the leaves of T. For any edge (u,v) such that $u \in T$ and $v \notin T$, let t be the location of the leaf of T in (u,v), DI works as follows:

1. The first time that $d(u,t) \geq 2/3 \cdot d(u,v)$, DI replicates file onto v.

2. The first time that $d(u,t) \leq 1/3 \cdot d(u,v)$, DI deletes copy at v.

Theorem 21 *Given a file allocation algorithm CO for continuous trees, for any sequence of requests the cost of the discrete trees algorithm DI is at most* 3 *times CO's cost.*

Proof. It follows from algorithm DI definition that for any read request the distance from that read to the closest leaf in T is at least $1/3$ the distance to the closest leaf in DI's configuration.

The cost for DI over a write request may be larger than that of CO by the weight of edges (u,v) including leaves of T but since both u and v hold file copies we infer that T includes at least $1/3$ of the edge weight.

For an edge (u,v), every deletion from v and the next replication into v can be amortized against a replication over distance at least $1/3 \cdot d(u,v)$ by CO. ■

Applying Theorem 21 on algorithm CT we obtain a 9-competitive algorithm for discrete trees.

8 Lower Bounds for File Allocation

8.1 A Lower Bound of 3 for 2-Processors

The following theorem gives a lower bound on the competitive ratio of any file allocation (or file migration) algorithm in any network topology.

Theorem 22 *Let $\mathcal{N}$ be any network over a set of at least two processors. The competitive ratio of any randomized on-line file allocation/migration algorithm for $\mathcal{N}$ against adaptive on-line adversaries is at least 3.*

Proof. Let two different processors in the network be p and q, and assume the distance between p and q is 1.

Assume the on-line algorithm holds a copy of the file at processor p. We define 3 different adaptive on-line adversaries as follows:

- The *p-adversary*: holds a single copy of the file at p.
- The *q-adversary*: holds a single copy of the file at q.
- The *jumping-adversary*: holds a single copy of the file at a processor not holding a copy of the file by the on-line algorithm (if such exists and otherwise the adversary's configuration remains unchanged) ;I.e., at processor q.

Now a write request is generated at q.

The cost for this request for both the q-adversary and the jumping-adversary is 0. The cost for the request for the p-adversary as well as the on-line algorithm is 1.

Consider next the cost incurred for configuration changes. The p-adversary and the q-adversary never change their configuration. On-line replications do not change the adversaries configurations. An on-line deletion (or migration) may cause the jumping-adversary to change its configuration incurring a cost of D. Except for the first deletion, the cost for file copies deletions can be charged D, against the cost for the replication.

We therefore conclude that over the entire sequence of events the online cost is equal to the cost of the 3 adversaries up to an additive term, implying the lower bound. ■

Black and Sleator [BS] used a result of Karlin et al. [KMRS] to get a lower bound of 3 for deterministic data migration algorithms. If requests are limited to write requests only, the file allocation problem collapses to the data migration problem, and therefore for deterministic algorithms the result in [BS] can be used to get the lower bound above.

8.2 An $\Omega(\log n)$ Lower Bound on Arbitrary Network Topologies

We now proceed to show, that in certain networks, the lower bound can be as bad as $\Omega(\log n)$, where n is the number of processors in the network. The following theorem relates file allocation lower bounds to Steiner tree lower bounds.

Theorem 23 *For every network $\mathcal{N}$, if there exists a c-competitive on-line file allocation algorithm for $\mathcal{N}$, then there exists a strictly c-competitive on-line Steiner tree algorithm for $\mathcal{N}$.*

The theorem holds for any type of adversary. However the proof of Theorem 23 is stated in terms of competitive randomized algorithms against oblivious adversaries. The proof for adaptive adversaries is similar.

In the proof of Theorem 23 we use the following definition and lemma.

Definition. Let A be a c-competitive randomized on-line file allocation algorithm in a network $\mathcal{N}$. Let the initial configuration be a single copy at a vertex v_1 of $\mathcal{N}$. Let σ be a sequence of requests to A. A (σ, v, ϵ)-*replicate forcing* sequence τ is a sequence of read requests at v, such that an optimal algorithm serving $\sigma\tau$ must have a copy at v at the end, and A has a copy at v at the end with probability $1 - \epsilon$. A (σ, ϵ)-*delete forcing* sequence τ is a sequence of write requests at v_1, such that an optimal algorithm serving $\sigma\tau$ must end in the configuration $\{v_1\}$, and A ends in that configuration with probability $1 - \epsilon$.

Notice that if A is c-competitive then for every σ, v, ϵ there must be a (σ, v, ϵ)-replicate forcing sequence. This is because each read request at v incurs an expected cost of at least the minimum distance in the network times ϵ, unless A replicates to v with probability greater than $1-\epsilon$, whereas the adversary's cost is at most D times the maximum distance in the network (for replicating to v). A similar argument shows that for every σ, ϵ, there is a (σ, ϵ)-delete forcing sequence.

Lemma 24 *Let $\mathcal{N}$ be a network over a set P of processors. Let A be a randomized c-competitive on-line file allocation algorithm in $\mathcal{N}$. Let σ be an arbitrary request sequence for A. Then, there exists a randomized on-line Steiner tree algorithm B for $\mathcal{N}$ with the following property: Let $\nu = v_1, v_2 \ldots, v_n$ be a sequence of vertices input to B. Let the initial configuration for A be $\{v_1\}$. Let $1 > \epsilon > 0$. Let τ be a (σ, ϵ)-delete forcing sequence for A. Let $\varrho = \varrho_2\varrho_3 \cdots \varrho_n$ be the following sequence. ϱ_2 is a $(\sigma\tau, v_2, \epsilon)$-replicate forcing sequence. ϱ_3 is a $(\sigma\tau\varrho_2, v_3, \epsilon)$-replicate forcing sequence. In general, ϱ_i, $2 \leq i \leq n$, is a $(\sigma\tau\varrho_2 \cdots \varrho_{i-1}, v_i, \epsilon)$-replicate forcing sequence.*

Then, B's expected cost to serve ν is at most $\frac{1}{D}$ times the expected cost incurred by A to serve ϱ after serving $\sigma\tau$, plus δ, where $\delta = \epsilon|P|W$, W being the sum of weights of all edges in $\mathcal{N}$.

Proof. We construct an on-line Steiner tree algorithm B for $\mathcal{N}$ as follows. Given input ν, we define the trees $T_1, T_2, \ldots, T_n$ chosen by B in response to ν as follows. we simulate A on $\sigma\tau$. If A's configuration is $\{v_1\}$ (this happens with probability $1-\epsilon$), then $T_1 = (\{v_1\}, \emptyset)$. Otherwise $T_1 = T_2 = T_3 = \cdots = T_n$ is an arbitrary spanning tree of $\mathcal{N}$. Now, in the first case, we give A ϱ_2, and execute the following procedure, with $j = 2$.

Let $T_j := T_{j-1}$. Repeat for $r = 1, \ldots, j$: give the rth request of ϱ_j, as input to A. For each processor, p, that A replicated from in response to the input request sequence, let Q be the set of processors that the file was replicated to from p. Set $T_j := T(T_j, Q)$.

If A does not have a copy at v_2 after it serves ϱ_2, then extend T_2 to an arbitrary spanning tree of $\mathcal{N}$, and set $T_3 = \cdots = T_n := T_2$. In general, if the first $j-1$ requests of ν are served, and B's tree does not span $\mathcal{N}$, then execute the above procedure, and if A does not have a copy at v_j in the end, then extend T_j to an arbitrary spanning tree of $\mathcal{N}$, and set $T_{j+1} = \cdots = T_n := T_j$.

It is obvious from the construction, that the edges added to the Steiner tree in the above procedure are exactly the edges along which A replicates. Therefore, in all executions of A in which A replicates to the vertices $v_1, v_2, v_3, \ldots, v_n$, B's cost is at most $1/D$ times the cost A incurs on ϱ. The probability that this does not happen is at most $n\epsilon \leq |P|\epsilon$. In this case, B pays the size of an arbitrary spanning tree of $\mathcal{N}$, which is at most W. ∎

Proof of Theorem 23. Assume the opposite. So, let A be an on-line file allocation algorithm such that $\mathrm{E}(\mathrm{cost}_A(\sigma)) \leq c \cdot \mathrm{Cost}_{\mathrm{OPT}}(\sigma) + a$ for every request sequence σ, while no on-line Steiner tree algorithm in $\mathcal{N}$ can achieve a ratio better than $b > c$. Let $b > b' > c$. Let B_0 be the on-line Steiner tree algorithm constructed from A by Lemma 24, taking some $\epsilon = \epsilon_0$, $\sigma = \sigma_0 = \emptyset$. There exists a sequence ν_0, such that B_0's expected cost on ν_0 is at least $b' \cdot T(\nu_0)$. By Lemma 24, the expected cost incurred by A on the sequence $\varrho = \varrho_0$ is at least D times B_0's cost on ν_0 minus $D\delta_0$, where $\delta_0 = \epsilon_0|P|W$. Since before serving ϱ_0 the optimal algorithm is forced to the configuration v_1, the cost incurred by the optimal algorithm to serve ϱ_0 is at most $D \cdot T(\nu_0)$, as it can replicate immediately to all vertices of ν_0. Note, that the cost incurred by the optimal algorithm to serve ϱ_0 is at least the minimum distance ℓ in $\mathcal{N}$. Let τ_0 be the $(\emptyset, \epsilon_0)$-delete forcing sequence used by Lemma 24.

Now, use Lemma 24 again with ϵ_1, $\sigma = \sigma_1 = \tau_0\varrho_0$. We can define sequences ν_1, τ_1 and ϱ_1, and a Steiner tree algorithm B_1, such that B_1's cost on ν_1 is at least $b' \cdot T(\nu_1)$, and A's cost on ϱ_1 is at least D times B_1's cost on ν_1 minus $D\delta_1$, where $\delta_1 = \epsilon_1|P|W$, and the optimal cost on ϱ_1 is at most $D \cdot T(\nu_1)$. Now, repeat this process infinitely many times. Let $\sigma_i = \tau_0\varrho_0\tau_1\varrho_1 \cdots \tau_{i-1}\varrho_{i-1}$ be the sequence given to A after i repetitions of this process. We get:

$$\frac{\mathrm{cost}_A(\sigma_i) - a}{\mathrm{Cost}_{\mathrm{OPT}}(\sigma_i)} \geq b' - \frac{a}{i\ell} - \frac{\sum_{j=0}^{i-1} \delta_j}{i\ell}.$$

Choose ϵ_j such that $\sum_{j=0}^{\infty} \delta_j$ converges, and get that the right-hand side of the inequality converges to b' as i goes to infinity, a contradiction. ∎

Imase and Waxman [IW] prove the following theorem.

Theorem 25 *For all n, there exist graphs G_n over n nodes, such that the competitive ratio for on-line Steiner tree for those graphs is in $\Omega(\log n)$.*

We note that this result applies to randomized algorithms against the oblivious adversary.

Theorems 23 and 25 give

Theorem 26 *For all n, there exist networks over n processors $\mathcal{N}_n$, such that the competitive ratio of any randomized algorithm against the oblivious adversary on those networks is in $\Omega(\log n)$.*

9 Distributed Algorithms

In the previous sections (Sections 6, 7.1, 7.2, and 5, we have assumed some "global intelligence," that knows the configuration of the entire network, and makes decisions for the single processors. In this section, we remove this assumption and give distributed competitive file allocation algorithms.

The model assumes that in a network over n processors, sending a message of size $O(\max\{\log n, \log(Diam)\}$ over a communication link of weight w costs w. We assume, that the size of data, which a read or write request use, is a single word of size $\log n$ bits. The size of the file is D words, each of size $\log n$ bits.

Definition. A distributed on-line file allocation algorithm has to serve sequences of read and write requests that processors in the network initiate. The cost of a distributed on-line file allocation algorithm to serve a sequence of requests is the total cost of messages it sends to serve the sequence.

Definition. A distributed on-line algorithm is c-competitive iff there exists a constant a, such that for any global-control adversary Adv,

$$\mathrm{E}(\mathrm{Cost}_{\mathrm{Alg}}(\sigma)) \leq c \cdot \mathrm{E}(\mathrm{Cost}_{\mathrm{Adv}}(\sigma)) + a$$

where σ is the request sequence generated by Adv.

9.1 Preliminaries and Distributed Data Structures

9.2 The Cover Problem

The Cover Problem Definition

The *on-line cover problem* is the problem of maintaining a covering of small number of small diameter subsets of a dynamically changing set in a weighted graph.

Let G be a weighted graph. Let Q be a subset of nodes of G.

For integers $r, s > 0$, a set $C = \{C_1, C_2, \ldots, C_s\}$ of mutually exclusive subsets of nodes, and a choice of nodes $p_1, p_2, \ldots, p_s$, $p_i \in C_i$, is called an *r-cover* of Q iff for every i, $i = 1, 2, \ldots, s$, $Q \cap C_i \neq \emptyset$, and $Q \subset \cup_{i=1}^{s} C_i$, and for every C_i, $i = 1, 2, \ldots, s$, the distance between any node in C_i and p_i is at most r.

Each of the sets C_i is called a *cover set*. The chosen nodes, $p_1, p_2, \ldots, p_s$, are called *covering nodes*.

Initially the set contains a single node $Q = \{q_0\}$ and the cover contains one covering set $C_1 = \{q_0\}$, and a single covering node $p_1 = q_0$.

The on-line cover problem for some fixed parameter r is the problem of maintenance of an r-covering for a dynamic set Q, where insertions into Q and deletions from Q are allowed (but Q is never allowed to be empty), where s changes with Q.

The distributed on-line cover problem is that of maintaining a covering of a dynamic set of processors Q in the network allowing insertions, which are initiated at processors in Q, and deletions, which are initiated at the deleted processor.

Define the optimal cost of an insertion to be the distance between the origin node p and the inserted node q, and the optimal cost for a deletion is 0.

For every integer $k > 0$, We give in Section 9.6 an algorithm for the on-line cover problem that maintains a $2(k-1)$-covering such that for every sequence of insertions and deletions, if the optimal cost for the sequence is UPD, the final value of s is at most $1 + \frac{1}{k} \cdot \mathrm{UPD}$.

In fact we give a distributed algorithm that maintains the covering for a set Q of processors with the following additional properties:

1. The algorithm is $O(1)$-competitive. I.e., its communication cost is at most $O(1) \cdot \mathrm{UPD}$
2. The algorithm maintains a distributed data structure of the cover sets, so that reaching a processor in Q from a covering processor costs $O(k)$.

The solution for the distributed on-line cover problem and its analysis is described in section 9.6.

When all operations are insertions we can define a tree S over the set of nodes Q with edges between every origin and destination nodes for each insertion. In this case the covering of Q can be viewed as a *tree cover*, i.e., a partition of the dynamically growing subtree S in a graph into $O(T(S)/k)$ subtrees of diameter at most k each, where $k > 0$ is some integer.

The Hierarchical Cover Problem

In most of the cover problem applications (see Section 9.5) we need to simultaneously solve (r, s)-cover problems with $r = O(2^i)$ for all $0 \leq i \leq \log(Diam)$.

We would like to minimized the final value of s as well as the costs of the distributed covers algorithm.

To obtain the hierarchy of covers define i-level cover algorithms for solving the $2(k-1)$-cover problem for Q for $k = 2^i$, i.e., give all levels algorithms the entire sequence of insertions and deletions. Naively, this will result in $\log(Diam)$ factor in the competitive ratio of the resulting hierarchical algorithm, since every i-level algorithm is $O(1)$ competitive.

We have stronger claims on the performance of the hierarchical algorithm.

Let the number of of covering sets maintained by the i-level cover algorithm be denoted $s(i)$. Let the total number of covering sets creations made by the i-level cover algorithm be denoted $c(i)$. Then $s(i) \leq c(i)$.

Let the communication cost expended by the distributed i-level algorithm be denoted $\mathrm{Cost}_{\mathrm{CP_i}}$.

Theorem 27 *The hierarchical cover algorithm has the following properties:*

- *The total cost expended by the hierarchical cover algorithm is*

$$\sum_{i=0}^{\log(Diam)} \mathrm{Cost}_{\mathrm{CP_i}} \leq O(\min\{\log n, \log(Diam)\}) \cdot \mathrm{UPD}.$$

- *The total sum of diameters of covering sets maintained by the algorithm obeys*

$$\begin{aligned}\sum_{i=0}^{\log(Diam)} 2^i \cdot (s(i) - 1) &\leq \sum_{i=0}^{\log(Diam)} 2^i \cdot (c(i) - 1) \\ &\leq O(\min\{\log n, \log(Diam)\}) \cdot \mathrm{UPD}.\end{aligned}$$

- *For every i, the algorithm maintains a distributed data structure of the i-level cover sets, so that reaching a processor in Q from a covering processor at the i-level cover costs $O(2^i)$.*

9.2.1 Data Tracking

The *data tracking* mechanism of is a generalization of the mobile user tracking mechanism of [AP1, AP3].

In a network over a set P of n processors, the data tracking problem allows to maintain a subset Q of processors holding copies of the file with the following operations on Q:

Insert(u,v), initiated at $u \in Q$, inserts v to the set Q.

Delete(v), initiated at v, removes v from the set Q.

Find(u), initiated at u, returns the address of a processor $v \in Q$.

Definition. A distributed on-line data tracking algorithm serves sequences of Insert, Delete and Find operations initiated at processors of the network. The cost of a distributed on-line data tracking algorithm for a sequence of operations is the total cost of messages it sends to conduct those operations.

Definition. The *approximation factor* for an on-line data tracking algorithm, α, is the maximum over all Find operations, of the ratio $d(u,v)/d(u,Q)$, where u is the node initiating the Find, and $v \in Q$ is the returned node.

Definition. The optimal cost of Insert(u,v) is the cost of transmitting the file from u to v alone; i.e., $D \cdot d(u,v)$.

The optimal cost of Delete(v) is 0.

The optimal cost of Find(u) is the cost of sending a message from u to the closest processor in Q; i.e., $d(u,Q)$

In Section 9.5 we present a distributed on-line data tracking algorithm, named TRACK, dealing with arbitrary sequences of Insert, Delete and Find operations, such that the following theorem holds.

Theorem 28 *For every n-processor network, for every sequence of operations σ,*

1. *TRACK's total cost for conducting Insert and Delete in σ is $O(\min\{\log^2 n, \log n \log(Diam)\}/\log^2 D)$ times the total optimal cost of those operations.*
2. *TRACK's cost on each Find in σ is $O(\log^2 n/\log^2 D)$ times the optimal cost of that Find.*

3. TRACK's approximation factor is $O(\log n/\log D)$.

(Where the value of D *is truncated to* $[2, n]$*).*

The data tracking competitive ratio over request sequences of of updates and searches is denoted C_{TRACK}. *We have that* $C_{\mathsf{TRACK}} = O(\log^2 n/\log^2 D)$.

The data tracking approximation factor is denoted by α.

The memory needed for the algorithm is at most $O(\log^2(Diam))$ *per processor.*

If no memory considerations are made then the approximation factor can be in fact reduced to $\alpha = O(1)$.

9.2.2 Scanning Mechanism

Another simple but useful tool (see Section 9) is a distributed data structure that enables scanning through a subset of processors.

The scanning mechanism is usually used together with the data tracking mechanism, as to enable performing tracking operations over all processors, such as a Delete-All operation.

In a network over a set P of n processors, let Q be a subset of processors in P. Initially Q includes a single processor v_0.

Given a sequence of insertions for processors $v_1, v_2, \ldots, v_t$, we maintain a distributed data structure.

The processors $v_1, v_2, \ldots, v_t$ are connected in a tree structure $\mathcal{T}$, by using an adjacency list at each processor in $\mathcal{T}$.

When an insertion for v_i arrives: i.e., Insert(v_j,v_i) , $j < i$ is made, we add v_i to the tree t, by leaving a pointer in v_j to v_i, we also leave a back-tracking pointer in v_i to v_j.

This procedure enables scanning through all processors in the tree structure starting at every one of the processors in $\mathcal{T}$, at a cost equal to the weight of $\mathcal{T}$.

The above description requires a considerable amount ($\Theta(n)$) of memory per processor, for keeping the list of its children. In the next section we describe how this can be reduced to $O(\log(Diam))$.

9.2.3 Lists Manipulation

We now turn to the question of memory needs of our distributed algorithms. In our algorithms processors, p, maintain lists of pointers $L(p)$ to other processors.

Assume that the address of a processor appears in the lists of at most x processors.

Keeping every list at the processor maintaining it requires $\Theta(x \cdot n)$ pointers per processor.

To overcome this difficulty, for every processor we translate the list $L(p)$, into $\log(Diam)$ lists. List $L_i(p)$, $0 \leq i \leq \log(Diam)$, is the list of processors at distance between 2^i and $2^{i+1}-1$ from p.

Now, instead of keeping the entire list in p, pointers are kept to only one of the processors in each of the $\log(Diam)$ lists, and within a list $L_i(p)$, each processor in the list keeps a pointer to the next. The distance between any two processors u and v in $L_i(p)$ is at most $d(u,p)+d(p,v) \leq 2(2^{i+1}-1)$, and therefore every operation on the list $L_i(p)$, is proportional to 2^i.

It follows that every processor needs only $\log(Diam)$ pointers for maintaining the information for its own list, and since it may appear in the lists of at most x processors it may need x more pointers. Therefore the total memory requirements is reduced to $O(x+\log(Diam))$ pointers per processors.

The operations provided by this data structures, for the list $L(p)$, are:

- Insert(v) — insert the address of processor v into $L(p)$, this is done by updating the pointers in the required level, at a cost of $O(d(p,v))$.
- Delete(v) — delete the address of processor v into $L(p)$, this is done by updating the pointers in the required level, at a cost of $O(d(p,v))$.
- Find — Find a processor in $L(p)$, by using the pointer to a processor in the nonempty list $L_i(p)$ with minimal i.
- Scan — Scan the entire list, at a cost proportional to the communication cost between p and all processors in $L(p)$.

9.3 Uniform, Tree and Ring Network Topologies

We start with some simple examples of competitive distributed algorithms, by showing that the our file allocation algorithms for uniform, tree and ring networks in the previous sections can be adapted to obtain distributed competitive algorithms keeping the competitive ratio $O(1)$.

Uniform Network

This is the simplest case, since for any deterministic algorithm, requests that do not cost the algorithm can be ignored without increasing the adversary cost, and every request of cost 1, can be informed of to a single chosen leader in the network that will simulate the algorithm and invoke the required operations. The cost of communicating to and from the leader is at most $O(1)$ times the cost of the request and hence the increase in the competitive ratio is $O(1)$.

In particular we have that the algorithm Count in Section 6 can be made distributed with only $O(1)$ overhead.

Tree Networks

We give a distributed implementation of the algorithms for given in Section 7.1 and 7.2.

Both algorithms maintain the set of file copies in a subtree of the network, denoted B, and the operations needed to implement the algorithm are the following:

- Find(v) — Find the nearest processor to v, in the tree of processors holding copies B, to enable reading and replicating the file. The cost expended for this operation should be proportional to $d(B, v)$.
- Scan the tree of copies B in order to enable writing to the file. The cost for this operations should be proportional to $T(B)$.
- Growing and Shrinking the tree of copies B at its leaves. The cost associated with growing the tree should be proportional to the length of the path added. The cost associated with shrinking the tree is 0.

In order to do these tasks we maintain two distributed data structure: one is the *tree scanning* mechanism described in Section 9.2.1, that keeps the processors in B, in a tree structure maintaining pointers between adjacent processors. This structure enables writing the file by scanning the tree structure $\mathcal{T}$ at a cost proportional to $T(B)$. Since both algorithms shrink the tree only after a write is invoked then when scanning the tree is over a scan-back operations begins from the leaves back, shrinking the tree. The shrinking is done as described in the global-control algorithms. The cost of this process is still proportional to $T(B)$.

The other data structure is a very simple data tracking mechanism for trees, which is just a pointer in each node not holding a copy of the file, indicating in the direction of which of its edges is the tree of copies B. This structure is updated when a replication or deletion of a file copy is made, without incurring further cost. Therefore a Find(v) operation is performed by following the pointers from v towards a processor in B, at a cost of $d(B,v)$.

Ring Networks

Similarly to the case of trees, the ring randomized algorithm in Section 5 maintains the file copies in processors along an interval B.

Again, we maintain pointers at processors of B that enable scanning B at a cost proportional to $T(B)$.

The only difference from trees is the implementation of a Find(v). This is implemented using a search algorithm that searches from v, in phases. In phase i, it searches in the two possible directions from v to a distance 2^i. It easily follows that the cost expended until a file copy is found is proportional to $d(B,v)$.

9.4 A Randomized Algorithm for Arbitrary Network Topologies

We demonstrate the implementation of our randomized general topology file allocation algorithm SB as a distributed algorithm in a network of processors. SB is defined with respect to some on-line Steiner tree algorithm. We use a version of SB, that runs a variant of the greedy on-line Steiner tree algorithm of [IW]. Given a new input vertex p, this greedy algorithm attaches it to the closest vertex in the existing tree. No other vertices are added. In a network over n processors, the greedy algorithm is strictly $O(\log n)$-competitive. The proof is identical to the one given in [IW].

We give a distributed on-line file allocation algorithm, named distributed-SB, for any network. We measure distributed-SB's messages cost for any sequence of reads and write and show that it is within polylogarithmic factors of the off-line cost.

Note, that distributed-SB's cost is measured against the cost of an optimal *non-distributed* algorithm.

Distributed-SB uses the distributed data tracking mechanism (Section 9.2.1

The Distributed On-line Data Tracking Problem.

In a network over a set P of n processors, maintain a subset Q of processors holding copies of the file with the following operations on Q:

Insert(u,v), initiated at $u \in Q$, inserts v to the set Q.

Delete(v), initiated at v, removes v from the set Q.

Find(u), initiated at u, returns the address of a processor $v \in Q$.

In Section 9.5 we present a distributed on-line algorithm, named TRACK, that solves the data tracking problem. The properties of this algorithm are given in Theorem 28 in Section 9.2.1.

Let $C_{\mathsf{TRACK}} = O(\log^2 n / \log^2 D)$ be the data tracking competitive ratio, and let α be the approximation factor of the data tracking mechanism.

We prove the following result:

Theorem 29 *For every n-processor network $\mathcal{N}$, distributed*-SB *is $O(\alpha \cdot C_{\mathsf{TRACK}} \cdot \min\{\log n, \log(Diam)\})$ competitive.*

Corollary 30 *For every n-processor network $\mathcal{N}$, distributed*-SB *is $O(\alpha \log^3 n / \log^2 D)$ competitive.*

Distributed-SB.

Distributed-SB works as follows. It maintains the set of processors holding copies of the file in a tree structure $\mathcal{T}$, by using an adjacency list at each processor in $\mathcal{T}$, enabling scanning the set of processors, as described in Section 9.2.2. Vertices of $\mathcal{T}$ are also maintained by the distributed on-line data tracking algorithm. If a processor r initiates a *read* request, invoke Find(r), which returns the address of a processor q holding a copy. Get the required data. With probability $1/D$ do the following. Simulate the greedy Steiner tree algorithm by adding r to $\mathcal{T}$ (connected to q). *Replicate* to r using Insert(q,r).

If a processor w initiates a *write* request, invoke Find(w), which returns the address of a processor q holding a copy. Send the required data from w to q, then from q to the rest of the processors holding copies via the tree structure $\mathcal{T}$. With probability $1/\sqrt{3}D$ do the following. Add w to $\mathcal{T}$, and *replicate* to w using Insert(q,w). Then, scan $\mathcal{T}$ post-order, starting at q, and at each visited processor $p \neq w$, Delete(p). Then, collapse $\mathcal{T}$ to the single vertex w.

Our analysis of distributed-SB proceeds as follows. We compare its cost with the cost of SB on the same sequence and with the same outcome of coin tosses. We divide SB's cost and distributed-SB's cost into three categories:

1. Find cost, which is the cost of reads, excluding the replication cost.

2. Scan cost, which is the cost of writes, excluding the migration cost.

3. Update cost, which is the cost of replications and deletions.

In the following claims, we assume that SB and distributed-SB use the same sequence of random bits.

Fact 31 *At all times, the set of processors holding copies of the file that distributed-*SB *maintains (vertices of $\mathcal{T}$) equals the set of processors in which* SB *holds copies of the file.*

Lemma 32 *At all times, the total length of edges of the tree $\mathcal{T}$ maintained by distributed-*SB *is α times the total length of of edges of the greedy Steiner tree maintained by* SB*.*

Proof. Follows from Fact 31 and statement 3 of Theorem 28. ■

Lemma 33 *For every sequence of requests σ, for every read request in σ, the find cost of distributed-*SB *for that read is C_{TRACK} times the find cost of* SB *for the same read.*

Proof. Follows from Fact 31 and statement 2 of Theorem 28. ■

Lemma 34 *For every sequence of requests σ, for every write request in σ, the scan cost of distributed-*SB *for that write is $C_{\mathsf{TRACK}} + \alpha$ times the scan cost of* SB *for the same write.*

Proof. The scan cost includes the cost of finding a processor holding a copy of the file and the cost of scanning the tree of processors holding copies. By Fact 31 and statement 2 of Theorem 28, the first task costs distributed-SB C_{TRACK} times SB's cost for the same task. By Lemma 32, the second task costs distributed-SB α times SB's cost for the same task. ■

Lemma 35 *For every sequence of requests σ, the update cost of distributed-*SB *for σ is $O(\alpha \cdot C_{\mathsf{TRACK}})$ times the update cost of* SB *for σ.*

Proof. The update cost of distributed-SB includes the data tracking cost and the cost for maintaining $\mathcal{T}$. SB and distributed-SB replicate and delete the same copies, but distributed-SB replicates along distances, which are α times the distances SB replicates along. Fact 31 and statement 1 of Theorem 28 give, that the data tracking cost of distributed-SB over σ is $O(C_{\mathsf{TRACK}})$ times the update cost of a non-distributed algorithm, that replicates and deletes exactly the same way that distributed-SB does. This is $O(\alpha \cdot C_{\mathsf{TRACK}})$ the update cost of SB, which is simply the cost of replications. Now, consider a subsequence of σ, where $\mathcal{T}$ grows monotonically (i.e., processors are added to $\mathcal{T}$) excluding the last request, where T might collapse. The maintenance cost of T over this subsequence is $O(1)$ times the maximal size of T, which by Lemma 32 is $O(\alpha)$ times the maximal size of the tree maintained by SB. SB's update cost to create that tree is D times its size. Therefore, we get that distributed-SB's cost for maintaining $\mathcal{T}$ is $O(\alpha/D)$ times SB's update cost. ■

Proof of Theorem 29. Follows from Lemmas 33, 34, 35, and the fact that SB is $O(\min\{\log n, \log(Diam)\})$-competitive (Theorem 10). ■

9.5 Distributed Data Tracking

In this section we describe the internals of the *data tracking* mechanism.

We recall the definitions and statements from Section 9.2.1.

In a network over a set P of n processors, the data tracking problem allows to maintain a subset Q of processors holding copies of the file with the following operations on Q:

Insert(u,v), initiated at $u \in Q$, inserts v to the set Q.

Delete(v), initiated at v, removes v from the set Q.

Find(u), initiated at u, returns the address of a processor $v \in Q$.

Definition. The *approximation factor* for an on-line data tracking algorithm, α, is the maximum over all Find operations, of the ratio $d(u,v)/d(u,Q)$, where u is the node initiating the Find, and $v \in Q$ is the returned node.

Definition. The optimal cost of Insert(u,v) is the cost of transmitting the file from u to v alone; i.e., $D \cdot d(u,v)$.

The optimal cost of Delete(v) is 0.

The optimal cost of Find(u) is the cost of sending a message from u to the closest processor in Q; i.e., $d(u,Q)$

We give a distributed on-line data tracking algorithm, named TRACK. We recall Theorem 28 stated in Section 9.2.1.

Theorem 28 For every n-processor network, for every sequence of operations σ,

1. TRACK's total cost for conducting Insert and Delete in σ is $O(\min\{\log^2 n, \log n \log(Diam)\}/\log^2 D)$ times the total optimal cost of those operations.
2. TRACK's cost on each Find in σ is $O(\log^2 n/\log^2 D)$ times the optimal cost of that Find.
3. TRACK's approximation factor is $O(\log n/\log D)$.

(Where the value of D is truncated to $[2, n]$).

The memory needed for the algorithm is at most $O(\log^2(Diam))$ per processor. If no memory considerations are made then the approximation factor can be in fact reduced to $O(1)$.

The Data Tracking Solution

In the solution to the data tracking problem (Section 9.2.1), we make use two tools.

One is a graph-theoretic structure of *regional matchings*, given by [AP1], an application of the sparse graph partitions [AP2].

An m-regional matching is an assignment of 2 sets of processors to each processor, a *read-set* and a *write-set*, such that for every two processors p and q that satisfy $d(p, q) \leq m$, the read-set of p and the write-set of q have a non-empty intersection.

The *radius* of a read-set or a write-set of p is the maximum distance between p and a processor in the set, divided by m.

The *degree* of a read-set or a write-set of p is the number of processors in the set.

The *read-radius*, *read-degree*, *write-radius* and *write-degree* of an m-regional matching are defined as the maximum over all processors p of the corresponding parameter for p.

[AP1] show how to construct for every m and ℓ, $2 \leq \ell \leq 2 \log n$, an m-regional matching with the following parameters: read-radius at most ℓ, read-degree at most $2\ell + 1$, write-radius at most $2\ell + 1$, and write-degree at most $n^{2/\ell}$.

For our purposes we take $\ell = 2 \log n / (\log D - \log\log D)$. Therefore, the read-radius, read-degree, and write-radius are all $O(\log n / \log D)$, and the write-degree is $D / \log D$.

The other tool, we use for the solution to the data tracking problem, is a solution to the *on-line cover problem.*

We recall the definitions of the cover-problem from Section 9.2.

Let Q be a subset of processors. For integers $r, s > 0$, a set $C = \{C_1, C_2, \ldots, C_s\}$ of mutually exclusive subsets of processors, and a choice of processors $p_1, p_2, \ldots, p_s$, $p_i \in C_i$, is called an *r-cover* of Q iff for every i, $i = 1, 2, \ldots, s$, $Q \cap C_i \neq \emptyset$, and $Q \subset \cup_{i=1}^{s} C_i$, and for every C_i, $i = 1, 2, \ldots, s$, the distance between any processor in C_i and p_i is at most r.

Initially the set contains a single processor $Q = \{q_0\}$ and the cover contains one covering set $C_1 = \{q_0\}$, and a single covering processor $p_1 = q_0$.

Each of the sets C_i is called a *cover set.* The chosen processors, $p_1, p_2, \ldots, p_s$, are called *covering processors.*

The on-line cover problem is the problem of maintenance of covering processors for a dynamic set Q, where insertions into Q and deletions from Q are allowed (but Q is never allowed to be empty).

The *hierarchical cover problem* is that of simultaneously maintaining (r, s)-cover problems with $r = 2 \cdot (2^i - 1)$ for all $0 \leq i \leq \log(Diam)$.

In Section 9.6 we give a competitive distributed algorithm that solves the hierarchical cover problem.

The properties of the hierarchical cover algorithm is given in Theorem 27 in Section 9.2.

Given solutions to the regional matching problem and the hierarchical cover problem, we solve the data tracking problem as follows. Compute m-regional matchings for $m = 2^i$, $i = 2, 3, 4, \ldots, \log(Diam)$. This is done once.

Insert(u,v) is performed by inserting v into Q by the hierarchical cover algorithm. If the i-level cover algorithm creates a new cover set with a new covering processor p, then p's write-set of the 2^{i+2}-regional matching is informed of p. Informing the write-set costs $O(2^i \cdot D \log n / \log^2 D)$.

Delete(u) is performed by deleting u from Q by the hierarchical cover algorithm. Each time an entire cover set of the i-level cover algorithm is removed, the corresponding covering processor informs its write-set of the 2^{i+2}-regional matching. Again, informing the write-set costs $O(2^i \cdot D \log n / \log^2 D)$.

Find(u) is performed by searching u's read-sets, starting with the 4-regional matching read-set, then the 8-regional matching read-set, etc. For the 2^i-regional matching read-set, u checks if there is a processor in the read-set, which is in the write-set of a covering processor (in the same regional matching). If such a processor q is found, u stops the search. Now, u can reach a processor holding a copy through q, the covering processor p that contains q in its write-set, and the data structure of the $(i-2)$-level cover algorithm that enables p to find a processor in Q.

The following claims are useful for the analysis:

Claim 36 *Let u be a processor. If there exists a processor v holding a copy, such that the distance between u and v is at most 2^i, then the read-set search in the Find(u) implementation does not go beyond the 2^{i+1} regional matching.*

Proof. Let w be v's covering processor in the $(i-1)$-level cover algorithm. The distance between v and w is at most $2 \cdot (2^{i-1} - 1) \leq 2^i$. By the triangle inequality, the distance between u and w is at most $2^i + 2^i = 2^{i+1}$. Therefore, in the 2^{i+1}-regional matching, the read-set of u and the write-set of w intersect. ■

Claim 37 *Let v be the processor returned by a Find(u) call whose read-set search terminated at the 2^i-regional matching. Then, the distance between u and v is in $O(2^i \log n / \log D)$.*

Proof. Let q denote the processor in the 2^i-regional matching at which the search ended successfully. Let p denote the covering processor that contains q in its write-set. $d(u, v)$ is at most the length of the path u–q–p–v, which is $d(u, q) + d(q, p) + d(p, v)$. $d(u, q)$ is bounded by the diameter of u's 2^i-regional matching read-set, which is in $O(2^i \log n / \log D)$. Similarly, $d(q, p)$ is bounded by the diameter of p's 2^i-regional matching write-set, which is in $O(2^i \log n / \log D)$. A bound on $d(p, v)$ is given by Property 3 of the $(i-2)$-level cover algorithm; i.e., $O(2^{i-2})$. ■

Proof of Theorem 28. Let σ be an arbitrary sequence of Insert, Delete and Find operations. We analyze the cost of the algorithm on σ.

Let the sum of optimal costs for Inserts and Deletes in σ be denoted UPD. The optimal cost of the sequence of insertions and deletions given to the hierarchical cover algorithm during the handling of σ is UPD$/D$.

Theorem 27 implies that the total cost of all cover algorithms to handle the sequence they are given during the handling of σ is

$$O(\mathrm{UPD} \cdot \min\{\log n, \log(Diam)\}/D). \tag{6}$$

In each of the $\log(Diam)$ cover algorithms, the number of cover sets removed is bounded by the number of cover sets created.

Let the total number of covering sets creations made by the i-level cover algorithm be denoted $cs(i)$. Then $s(i) \leq cs(i)$.

Theorem 27 states that

$$\sum_{i=0}^{\log(Diam)} 2^i \cdot (cs(i) - 1) \quad \leq \quad O(\min\{\log n, \log(Diam)\}) \cdot \mathrm{UPD/D}.$$

Therefore, the total cost of informing the write-sets of Inserts and Deletes is at most:

$$\begin{aligned} & \sum_{i=0}^{\log(Diam)} 2(cs(i) - 1) \cdot O(2^i D \log n / \log^2 D) \\ & \leq \; O(\min\{\log n, \log(Diam)\}) \cdot \frac{\mathrm{UPD}}{\mathrm{D}} D \log n / \log^2 D \\ & \leq \; O(\min\{\log^2 n, \log n \log(Diam)\} / \log^2 D) \cdot \mathrm{UPD}. \end{aligned} \tag{7}$$

The first statement of the theorem follows from Equations 6 and 7.

Now, examine the cost of a Find. Let the last read-set searched be that of the 2^j-regional matching. The communication cost of the last search is bounded by $O(2^j \cdot \log^2 n / \log^2 D)$. This also bounds the total search cost. Tracing the pointers to a processor holding a copy costs $O(2^j \cdot \log n / \log D) + O(2^j \cdot \log n / \log D) + O(2^j) = O(2^j \cdot \log n / \log D)$. The optimal cost of this Find operation is given by Claim 36. Therefore, we conclude that the cost of the on-line data tracking algorithm per Find is $O(\log^2 n / \log^2 D)$ times the optimal cost per the same Find. This shows the correctness of the second statement of the theorem.

The third statement of the theorem follows directly from Claim 37. ■

9.6 The Cover Problem Solution

We complete our discussion by showing a solution to the cover problem (Section 9.2). We repeat the formulation of the cover problem.

The Cover Problem

Given a network of processors, defined by a weighted graph G, and a dynamically changing set of processors among the network processors, we would like to construct an (r, s)-cover for Q. That is dynamically define a set of mutually exclusive subsets of processors $C = \{C_1, C_2, \ldots, C_s\}$, and a choice of processors $p_1, p_2, \ldots, p_s$, $p_i \in C_i$, so that for every i, $i = 1, 2, \ldots, s$, $Q \cap C_i \neq \emptyset$, and $Q \subset \cup_{i=1}^{s} C_i$, and for every C_i, $i = 1, 2, \ldots, s$, the distance between any node in C_i and p_i is at most r.

Initially the set contains a single node $Q = \{q_0\}$ and the cover contains one covering set $C_1 = \{q_0\}$, and a single covering node $p_1 = q_0$.

The on-line cover problem for some fixed parameter r is the problem of maintenance of an r-cover for a dynamic set Q, where insertions into Q and deletions from Q are allowed (but Q is never allowed to be empty), where s changes with Q.

Define the optimal cost of an insertion to be the minimum distance between a node of Q and the inserted node, and the optimal cost for a deletion is 0. The optimal cost for a sequence of inserts and deletes is the sum of optimal costs of the operations.

We now turn to describe the cover algorithm and the proof of Theorem 27.

The Basic Cover Algorithm — Unweighted Case

We describe an on-line algorithm for maintaining a $2(k-1)$-cover for any integer $k > 0$.

Assume at first, that the network of processors is defined by a weighted graph G in which all weights are 1, and that all insertions are to processors adjacent in this graph to processors already in Q. Therefore, the cost charged for an insertion is 1.

Each cover set is represented by a directed tree. The root is the corresponding covering processor, and all edges point towards the root. A processor contained in a cover set is marked as such. A processor in Q is marked as such.

The cover algorithm works as follows. Every processor p holds a counter c_p. Initially q_0's counter is 0. For all other processors the value is undefined.

Let the current cover sets be $C = \{C_1, C_2, \ldots, C_s\}$, let $\mathcal{C} = \cup_{i=1}^{s} C_i$, and let the covering processors be $p_1, p_2, \ldots, p_s$. The algorithm maintains the following invariants:

1. For every $p \in \mathcal{C}$, $0 \leq c_p \leq 2k - 2$.
2. For every i, $1 \leq i \leq s$, for every $p \in C_i$, c_p is an upper bound on $d(p, p_i)$.
3. For every i, $1 \leq i \leq s$, all processors in C_i form a directed tree rooted at p_i. This tree is a subtree of G.
4. For every i, $1 \leq i \leq s$, every path from p_i to a leaf in the tree representing C_i contains at least one processor of Q.
5. ($k\times$ the number of cover sets created) + (the number of processors p such that $c_p \geq k$) $\leq$ (the optimal cost for the sequence).

These invariants ensure the correctness of the algorithm. Invariants 1,2 and 4 imply that the cost of reaching a processor in Q from a covering processor is at most $2k - 2$.

We now describe the cover algorithm, by defining how it handles insertions and deletions.

Let p be an inserted processor. Let $q \in Q$ be the processor adjacent to p, that initiates the insertion. If p is already in $\mathcal{C}$, it simply marks itself as being in Q. Otherwise, the following update procedure is performed. First, c_p is set to $c_q + 1$, and p is added to q's tree by an edge pointing from p to q.

Now, if $c_p = 2k - 1$, a scan-back procedure is conducted, starting at p. Each processor scanned decreases its counter by k, and then the scan moves to its parent in the tree. The scan-back stops once a processor b with $c_b < k$ is encountered (the root has a counter 0, so the process must stop). This preserves invariant 1: $0 \leq c_p \leq 2k - 2$.

Let t be the processor scanned just before b. If the new value of c_t is 0, then a new cover set is created with t as the covering processor. This is done by detaching t and all successors of t from b's tree.

Let p be a deleted processor. p marks itself as not in Q. If $p \in \mathcal{C} \setminus Q$ has no successors in Q, remove p from $\mathcal{C}$. If p is the root of its tree then the appropriate cover set is removed, otherwise a message is sent to p's parent to inform of the its removal from the subtree.

This procedure preserves invariant 4.

The Basic Cover Problem — Analysis

The procedure described above clearly preserves properties 1 and 4, as mentioned above.

We turn to prove the other properties of the cover algorithm.

We show that the cover algorithm preserves a somewhat stronger property than invariant 2:

2'. For every i, $1 \leq i \leq s$, for every $p \in C_i$, c_p is an upper bound on $d(p, p_i)$. If $c_p \leq k$ then $c_p = d(p, p_i)$.

The proof that the invariant holds is by induction on the insertion steps in the algorithm, using the fact that when a processor is inserted it is firstly assigned its predecessor's counter plus one, and the fact that if a counter is ever decreased then it decreased from some value greater than k to some value smaller than k.

It is clearly true at the beginning where there is only one node in the tree with zero counter. Consider a new insertion of processor p from q. For a processor x, let c'_x denote the value of its counter after the insertion. If $c'_p \leq 2k-2$ then since $c'_p = c_q + 1$ the invariant follows from the induction hypothesis. Assume $q \in C_i$, since the distance to the covering processor p_i increases by at most 1. we have $d(p_i, p) \leq d(p_i, q) + 1 \leq c_q + 1 = c'_p$.

If $c'_p = 2k-1$ then let b and t be as in the procedure above. If $c'_t > 0$ then since when t was inserted from b, we had that t's counter was greater by 1 from b's counter. and before the insertion of p, $c_b < k$ while $c_t > k$ it follows that c_b was once decreased by k, and therefore $c'_t = c_t - k = c_b + 1$.

If $c_t = 0$ then the invariant holds since before the insertion all counters of processors in the subtree rooted at t where greater than k, and there distance to the covering processor only decreased as a result of the creation of a new cover set. The only counters decreased are those along the path from t to p. Obviously each counter is equal to the processors distance to the new covering processor t. It follows that the procedure preserves invariant 2.

To prove invariant 5, define $\Phi =$ the number of processors p, such that $c_p \geq k$. Obviously when a new processor is inserted $\Delta\Phi \leq 1$. A processor deletion may only reduce the value of Φ. If a new cover set is created, that is $c_t = 0$, it follows that the length of the path from t to p is $k-1$. The counter of every processor along the path has decreased from a value of at least k to a value below k. Since there are k such processors we get $\Delta\Phi \leq -k$. As $\Phi \geq 0$ the claim follows.

In the actual implementation of this process, between the time a processor p is inserted by q and the time p is deleted, if ever, there is a constant number of messages passed over the edge between p and q — one message for the insertion itself, at most once the scan-back passes over this edge, and at most once p is detached from q. The fact that a scan-back message passes only one through each edge follows from the fact that during the scan-back the counters are always decrease from a value at least k to a value less than k. This implies the competitiveness of the algorithm.

The Cover Algorithm — Weighted Case

Arbitray distances can be translated into integral distances while changing costs by no more than a constant factor. To deal with arbitrary integer distances between nodes, imagine that along an edge between two nodes there are virtual nodes that divide the distance into segments of length 1.

If q inserts p, let u_p be the length of the shortest path from q to p. The insertion is done by simulating insertions for all virtual processors along the shortest path from q in Q to p in that order. If one of the virtual processors is a covering processor then let p be a covering processor. This may only decrease distances between a covering processor and the actual processors in it's cover set. Following this, delete all virtual processors.

The Hierarchical Cover Algorithm

The hierarchical cover algorithm (Section 9.2) is defined by simultaneously running cover algorithms for $\log(Diam)$ levels. For level $0 \leq i \leq \log(Diam)$ we run a $2(k-1)$-cover algorithm for $k = 2^i$.

Let the number of of covering sets maintained by the i-level cover algorithm be denoted $s(i)$. Let the total number of covering sets creations made by the i-level cover algorithm be denoted $c(i)$. Then $s(i) \leq c(i)$.

Let the communication cost expended by the distributed i-level algorithm be denoted $\mathrm{Cost}_{\mathrm{CP_i}}$.

We recall Theorem 27 stated in Section 9.2.

Theorem 27 The hierarchical cover algorithm has the following properties:

- The total cost expended by the hierarchical cover algorithm is

$$\sum_{i=0}^{\log(Diam)} \mathrm{Cost}_{\mathrm{CP_i}} \leq O(\min\{\log n, \log(Diam)\}) \cdot \mathrm{UPD}.$$

- The total sum of diameters of covering sets maintained by the algorithm obeys

$$\begin{aligned}\sum_{i=0}^{\log(Diam)} 2^i \cdot (s(i)-1) &\leq \sum_{i=0}^{\log(Diam)} 2^i \cdot (c(i)-1) \\ &\leq O(\min\{\log n, \log(Diam)\}) \cdot \mathrm{UPD}.\end{aligned}$$

- For every i, the algorithm maintains a distributed data structure of the i-level cover sets, so that reaching a processor in Q from a covering processor at the i-level cover costs $O(2^i)$.

To prove the theorem we need to replace property 5 of the cover algorithm by the following stronger claim.

Let the optimal cost for insertions of nodes v, such that $u_v \geq (2^i-1)/2n$ be denoted UPD_i.

5'. For every level i,

$$(c(i)-1)\cdot(2^i-1) + \sum_{p;u_p \geq \frac{(k-1)}{2n}, c_p \geq k} u_p \leq 2 \cdot \mathrm{UPD}_i.$$

Thus the optimal cost, u_v, of an insertion of node v is accounted for only in levels i such that $2^i - 1 \leq 2n \cdot u_v$, and therefore in at most $O(\min\{\log n, \log(Diam)\})$ levels.

Summing up over all levels we get $\sum_{i=1}^{\log(Diam)}(c(i)-1)\cdot 2^i$ is bounded above by $O(\min\{\log n, \log(Diam)\})$ times the optimal cost, giving the bound in Theorem 27.

To prove invariant 5', let $k = 2^i$, and define

$$\Phi = \sum_{p;u_p \geq \frac{(k-1)}{2n}, c_p \geq k} u_p.$$

Obviously when a new node p such that $u_p \leq (k-1)/2n$ is inserted $\Delta\Phi \leq u_p$. A node deletion may only reduce the value of Φ. Let t be as defined in the cover algorithm. If a new cover set is created, that is $c_t = 0$, let P be the path from t to p. It follows that the length of the path P is $k-1$. Since $\sum_{p \in P; u_p < \frac{(k-1)}{2n}} u_p \leq (k-1)/2$, we have that $\sum_{p \in P; u_p \geq \frac{(k-1)}{2n}} u_p \geq (k-1)/2$. The counter of every node along the path has decreased from a value of at least k to a value below k. We therefore get that

$$\Delta\Phi \leq -\sum_{p \in P; u_p \geq \frac{(k-1)}{2n}} u_p \leq -(k-1)/2.$$

As $\Phi \geq 0$ the claim follows.

From the one-level cover problem analysis we can infer that the total communication cost incurred by the hierarchical cover algorithm is at most $O(\log(Diam))$ times the total updates cost.

To analyze the communication cost incurred by the hierarchical algorithm as to get the $O(\min\{\log n, \log(Di$ competitive ratio we need a more delicate distributed implementation of the hierarchy of cover algorithms while the actual output of algorithms stays the same as before.

The distributed implementation is based on the following properties of the cover algorithms:

1. Denote the i-level counter of processor p by c_p^i. Let $k = 2^i$. Then for every $j \geq i$, if $c_p^i < k$ then $c_p^i = c_p^j \bmod k$, and if $c_p^i \geq k$ then $c_p^i = k + c_p^j \bmod k$.
2. The total cost of scan-backs for a tree C_j in level i is at most the weight of C_j, at the time of the creation of C_j.
3. Consider a leaf p of a tree C_j in level i. Let $k = 2^i$. Then either p is a leaf in level $i+1$ as well or that from the time p was inserted p was the root of a subtree of weight at least $k-1$.

Property 1 is proved by induction on the algorithms executions. If an insertion is made then the counter of both i-level and j-level algorithms are increased by 1. The property holds since always $c_p^i < 2k$. Since during a scan-back processors' counters are always decreased by k in case were at least k the property still holds.

It follows that upon an insertion we need only send the highest level counter plus one bit per level indicating if $k = 2^i$ should be added. Then the i-level algorithm can compute the counter of the inserted processor using Property 1. This can be done using one message.

Property 2 follows from the fact that an edge from q to p is passed during a scan-back only if $c_p \geq k$ and then c_p is decreased to a value less then k.

It follows that the scan-back cost is bounded by the weight of the tree C_j at the time it was constructed. Recalling the bound on $c(i)$ above, this weight was at least $k-1$ at that time, and thus the optimal cost incurred of insertions of cost at least $(k-1)/2n$ is at least half the weight of C_j.

Property 3 is also proved by induction on the execution of the cover algorithms. When a new node p is inserted then it is a leaf in both levels. If during a scan-back the tree in level i is detached and p becomes a new leaf, then it follows from Property 1 that either p is a new leaf at level $i+1$ as well or there is a path from p to a leaf of the subtree rooted at p at level $i+1$ of length at least $k-1$. When a deletion for a leaf is made, one of its ancestors becomes a new leaf. It follows by induction that the property remain true for the new node.

Using Property 3 we have that when a leaf p is deleted at level i, it is either deleted at level $i+1$ as well, or a deletion of weight at least $k-1$ can be associated with the deletion of p at level $i+1$. Let q be p's parent. Then the cost of sending a message during a deletion of p at level $i+1$ can be associated with an optimal insertion cost of $k-1$ and thus a cost of at least $(k-1)/2$ for insertions of cost at least $(k-1)/2n$.

We conclude that the optimal cost for each insertion can be charged in at most $O(\min\{\log n, \log(Diam)\})$ levels as to account for the total communication cost of the hierarchical cover algorithm.

The above description requires a considerable amount ($\Theta(n)$) of memory per processor for each level, if every processor keeps the list of its children. Using the list maintenance mechanism described in Section 9.2.3 this can be reduced to $O(\log(Diam))$.

10 Constrained File Allocation

In this section we study the solution of multiple file allocation problems, constrained by the local memory of the processors. We assume all files are of the same size. Let $m = \sum_p k_p$, the total number of files that can be stored in the network, and $k = \max_p k_p$, the maximal number of files that can be stored in any one processor.

10.1 Lower Bound

In contrast to the situation for the file-allocation problem – the competitive ratio achievable for deterministic distributed paging algorithms is much higher.

Theorem 38 *The competitive ratio of any distributed paging algorithm, against an adaptive on-line adversary, is at least $2m-1$, in any network, when the memory capacity of all processors is equal.*

Proof. The lower bound is achieved even if only read requests are issued, for $k+1$ different files.

One of the files, called U, is special and receives no requests at all, but both algorithm and adversary must hold it somewhere in the network. Let the other k files be $R_1, R_2, \ldots, R_k$. Define the impossible configuration C, in which all processor p, hold files $R_1, R_2, \ldots, R_k$. This is not a legal configuration since file U does not reside anywhere in the network. Now, for all processors p, and $1 \leq i \leq k$, we define configuration $C_{p,i}$, derived from C by replacing file R_i with file U in processor p.

We say the algorithm is in state $C_{q,j}$ if processor q holds a copy of U, and does not hold a copy of R_j. Following [MMS], we define a set of $2m-1$ adversaries. If the on-line algorithm is in state $C_{q,j}$ then the configuration $C_{q,j}$ is associated with one of the adversaries, and the other $m-1$ configurations $C_{p,i}$, $p \neq q$ or $i \neq j$, are each associated with 2 adversaries. The adversary with configuration $C_{q,j}$ is said to coincide with the on-line algorithm.

The next read request is issued at processor q for file R_j. Since the algorithm does not hold a copy of R_j in q, it is charged at least the distance from q to q's nearest neighbor. All adversary algorithms, except the one that coincides with the on-line algorithm, have a copy of R_j in q, and thus incur no cost. The algorithm that coincides with the on-line algorithm has a copy of R_j at q's nearest neighbor, and therefore can read the data requested at a cost no larger than the on-line cost.

We can continue this procedure as long as the algorithm does not replicate R_j to q. If the algorithm replicates the file, overwriting file R_t, $(t \neq j)$, then the one of the two adversaries in configuration $C_{q,t}$ switches to configuration $C_{q,j}$ by replicating R_t instead of R_j, paying D times the distance to q's nearest neighbor, which is a lower bound on the on-line algorithm's replication cost.

If the algorithm replicates the file while overwriting U, then U must also be migrated to some other processor z overwriting some file R_l. The new on-line configuration is $C_{z,l}$, and the on-line cost is at least D times the distance from q to q's nearest neighbor, called the replication cost for the algorithm, plus the distance from q to z. One of the two adversaries in configuration $C_{z,l}$ migrates its copy of U to q, and replicate R_l to take U's place.

Thus, preserving the invariant that only one adversary coincides with the on-line algorithm, and every other configuration has two adversary algorithms associated with it.

The cost for this adversary algorithm is D times the distance from z to q, which is the same as the migration cost for the on-line algorithm, plus D times the distance from z's nearest neighbor to z. We call this cost the replication cost for the adversary. This concludes a phase of requests to processor q, and now a new phase of requests to processor z begins.

Thus the replication cost for the adversary in one phase is equal to the replication cost for the on-line algorithm in a subsequent phase.

Summing the costs of all adversaries over all phases is the same as the algorithm's cost over all phases, up to a constant additive term for the first and last phases. Since there are $2m-1$ different adversaries at all times, at least one of them must have been charged no more than a $1/(2m-1)$ fraction of the on-line algorithm's cost, giving the required lower bound. ∎

10.2 Uniform Networks

We present a deterministic competitive distributed paging algorithm for uniform networks. The algorithm is optimal up to a constant factor.

Our algorithm uses the following terminology. We say a processor p is *free* if it holds less than k_p different files. A copy of a file is called *single* if there are no other copies of that file currently in the network.

Our algorithm works in phases. Copies of files can be either marked or unmarked. At the beginning of a phase, all counters are zero and all copies are unmarked. Throughout, an unmarked copy is single, a marked copy may be not single.

Algorithm DFWF (Distributed Flush-When-Full).
The algorithm is defined for each processor p separately. Every processor maintains a counter c_F for every file F. Initially, or as a result of a *restart* operation, all counters are set to zero and all markings are erased. Arbitrarily, copies of files are deleted until there is exactly one copy of every file somewhere in the network.

Every processor p follows the following procedure simultaneously for all files F:

1. While $c_F < D$, if a *read*(F) request is initiated at p, or if a *write*(F) request is initiated at p and F is unmarked, increase c_F by 1, if p does not contain a copy of F.
2. (a) If p is free, *replicate* F to p and mark it. If F was unmarked, *delete* the unmarked copy.
   (b) Otherwise, if all file copies in p are marked then restart.
   (c) Otherwise, choose S to be an arbitrary unmarked copy in p.
      i. If F is unmarked, switch between S and F, and mark F in p.
      ii. Otherwise, if some free processor q is available, *dump* S to q, and *replicate* a copy of F to p, mark this copy.
      iii. Otherwise, restart.
3. While $c_F > 0$, if a *write*(F) request is initiated by any other processor, decrease c_F by 1.
4. Restart.

Theorem 39 *Algorithm* DFWF *is $3m$-competitive for distributed paging on uniform networks.*

Proof. We analyze the algorithm over a phase, between consecutive restarts. We compare the algorithm to an optimal algorithm for that phase, which may start at any initial configuration.

We measure the *modified optimal cost*, whereby deletes cost D, whereas replications cost 0. The sum over phases of the modified optimal cost is a lower bound on any adversary's cost, up to a constant additive term. Let W denote the total number of write requests dealt with in step 3, for all processors and all files. Let R denote the total number of read/write requests initiated by all processors while at step 1 except that every processor p excludes requests to the k_p files having the largest c_F counts.

Claim 40 DFWF*'s cost per phase is at most* $(3m-2)D+R+W$.

Claim 41 *The modified optimal cost per phase is at least* $\max\{D, R, W\}$.

The Theorem follows from these claims. ■

Proof of Claim 40. We denote by $\mathcal{M}_p$ the set of k_p files with the largest c_F counters in processor p. $\mathcal{L}_p$ denotes the set of files excluding those in $\mathcal{M}_p$. For every p, for every file $F \in \mathcal{M}_p$, let $C(p, F)$ be the cost of the algorithm for requests to F in step 1, the possible replication of F in step 2, and the possible dumping of an unmarked file as a result of replicating F in step 2c. Clearly, $C(p, F) \leq 3D$. We want to show that

$$\sum_{p, F \in \mathcal{M}_p} C(p, F) \quad \leq \quad (3m-2)D.$$

Case 1. At the end of the phase there were at most $m-1$ marked copies. For each such copy F in p, $C(p, F) \leq 3D$. For all other copies considered, there were at most D requests.

Case 2. At the end of the phase there were m marked copies. The last unmarked copy F was marked in p in step 2a. Therefore, there was no dump, and $C(p, F) \leq 2D$. Also, marking F left the processor q holding the unmarked copy of F free. Therefore, there exists $G \in \mathcal{M}_q$, such that $C(q, G) \leq 2D$, because, again, no dump occurs on behalf of G in q.

The only requests in step 1 not accounted for are those to files in $\mathcal{L}_p$, for all p. Each such request costs 1. The only write requests not accounted for are those in step 3. Each such request can be charged 1 in each processor not initiating the request, which holds a copy of the requested file. Note, that each such copy is marked. ■

Proof of Claim 41. We denote the modified optimal cost in a phase by OPT.

1. We do a case analysis to show that OPT$\geq D$.

(a) The phase ended in step 2b. Then, there is a processor p in which $k_p + 1$ distinct files received at least D requests in step 1 each. Therefore, OPT either includes the cost of D requests to some file not available at p, or the cost of deleting a file.

(b) The phase ended in step 2c. Then, no processor is free. Either OPT includes the cost of D requests at some processor p to some file not available at p, or OPT includes the cost of deleting a file (because at the end of the phase the number of copies unmarked (singles) plus the number of copies marked is exactly m, and there is a new copy requested D times, but unavailable at the requesting processor).

(c) The phase ended in step 4 by processor p and file F. Either OPT includes the cost of D requests to F in step 1, or the cost of deleting F, or the cost of D writes to F in step 3.

2. For each processor p, ignore the first k_p (or less) files that the optimal algorithm places in p. For any other file F requested in step 1 in p, either OPT includes the cost of the requests to F, or D for deleting some other file. Since F was requested at most D times, OPT includes the cost of the requests to F. Therefore, OPT$\geq R$.

3. Let the number of write requests in step 3 for file F in p be denoted by x. $x \leq D$. Either OPT includes the cost of D requests to F in step 1, or the cost of deleting F, or the cost of x writes to F in step 3. Therefore OPT$\geq W$.

■

11 Conjectures and Open Problems

The obvious open problems are to close the gaps between upper and lower bounds, and to give deterministic and/or randomized (oblivious) results where possible. A deterministic $O(\log n)$-competitive file-allocation algorithm, and a deterministic distributed algorithm are given in [ABF1], but the question of giving a deterministic counterpart to Theorem SBA is still open.

Motivated by the famous [MMS] conjecture, we conjecture that the constrained file allocation problem has a deterministic competitive ratio of $O(m)$ on arbitrary topologies. [ABF2] gives an $O(\log m)$-competitive randomized algorithm for the constrained file allocation problem on the uniform network. We hazard the guess that similar results can be obtained by randomized algorithms against oblivious adversaries for other network topologies as well.

The question of what competitive algorithms can be given distributed implementations, and at what cost, seems to extend beyond the distributed data management set of problems, and should be worth pursuing.

The models presented here can clearly be generalized in several directions and at least some of them seem to address real-life concerns. *E.g.*, issues regarding delay and congestion should be eventually addressed.

12 Acknowledgments

We thank Baruch Awerbuch, Howard Karloff, Dick Karp, David Peleg and Jeffery Westbrook for their very kind aid and comments.

References

[AA] N. Alon and Y. Azar. On-line Steiner Trees in the Euclidean Plane. In *Proc. 8th ACM Symp. on Computational Geometry*, pages 337-343, 1992.

[ABF1] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive Distributed File Allocation. To Appear in *Proc. of the 25th Ann. ACM Symp. on Theory of Computing*, pages 164-173, May 1993.

[ABF2] B. Awerbuch, Y. Bartal, and A. Fiat. Heat & Dump: Competitive Distributed Paging. In *Proc. of the 34th Ann. IEEE Symp. on Foundations of Computer Science*, pages 22-31, October 1993.

[AP1] B. Awerbuch and D. Peleg. Online Tracking of Mobile Users. Technical Report MIT/LCS/TM-410, Aug. 1989.

[AP2] B. Awerbuch and D. Peleg. Sparse Partitions. In *Proc. of the 31st Ann. Symp. on Foundations of Computer science*, pages 503–513, October 1990.

[AP3] B. Awerbuch and D. Peleg, Concurrent Online Tracking of Mobile Users, *Proc. SIGCOMM.* Zurich, Sept. 1991.

[BBKTW] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Wigderson. On the Power of Randomization in Online Algorithms. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 379–386, May 1990.

[BLS] A. Borodin, N. Linial, and M. Saks. An Optimal On-Line Algorithm for Metrical Task Systems. In *Proc. of the 19th Ann. ACM Symp on Theory of Computing*, pages 373–382, May 1987.

[BS] D.L. Black and D.D. Sleator. Competitive Algorithms for Replication and Migration Problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[C] W.W. Chu. Optimal File Allocation in a Multiple Computer System. IEEE Transactions of Computers, 18(10), October 1969.

[CL] M. Chrobak, L. Larmore, The Server Problem and On-line Games, in *On-Line Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 7, 1991, 11-64.

[CLRW] M. Chrobak, L. Larmore, N. Reingold, and J. Westbrook. Optimal Multiprocessor Migration Algorithms Using Work Functions. Unpublished.

[CV] B. Chandra and S. Vishwanathan. Constructing Reliable Communication Networks of Small Weight On-line. Journal of Algorithms.

[DF] D. Dowdy and D. Foster. Comparative Models of The File Assignment Problem. Computing Surveys, 14(2), June 1982.

[FKLMSY] A. Fiat, R.M. Karp, M. Luby, L.A. McGeoch, D.D. Sleator , and N.E. Young. Competitive Paging Algorithms. Journal of Algorithms, 12, pages 685–699, 1991.

[HP] J.L. Hennessy and D.A. Patterson. Computer Architecture: A Quantitative Aproach. Morgan Kaufmann Publishers, Inc. 1990.

[IW] M. Imase and B.M. Waxman. Dynamic Steiner Tree Problem. SIAM Journal on Discrete Mathematics, 4(3):369–384, August 1991.

[KMRS] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive Snoopy Caching. Algorithmica, 3(1):79–119, 1988.

[MMS] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive Algorithms for On-Line Problems. In *Proc. of the 20th Ann. ACM Symp. on Theory of Computing*, pages 322-333, May 1988.

[ML] H.L. Morgan and K.D. Levin. Optimal Program and Data Locations in Computer Networks. CACM, 20(5):124–130

[RS] P. Raghavan and M. Snir. Memory versus Randomization in On-Line Algorithms. In *Proc. 16th ICALP*, July 1989.

[ST] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communication of the ACM*, 28(2) pages 202–208, 1985.

[W] J. Westbrook. Randomized Algorithms for Multiprocessor Page Migration. Proc. of DIMACS Workshop on On-Line Algorithms, to appear.

[WY1] J. Westbrook. and D.K. Yan. personal communication.

[WY2] J. Westbrook. and D.K. Yan. Greedy On-Line Steiner Tree and Generalized Steiner Problems. In *Proc. of the 3rd Workshop in Algorithms and Data Structures*, Also *Lecture Notes in Computer Science*, vol. 709, pages 622-633, Montréal, Canada, 1993, Springer-Verlag.