JOURNAL OF ALGORITHMS 11, 208–230 (1990)

# Competitive Algorithms for Server Problems

MARK S. MANASSE

*DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301*

LYLE A. MCGEOCH*

*Department of Mathematics and Computer Science, Amherst College, Amherst, Massachusetts 01002*

AND

DANIEL D. SLEATOR†

*School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213*

Received December 2, 1988; accepted April 13, 1989

The *k-server problem* is that of planning the motion of $k$ mobile servers on the vertices of a graph under a sequence of requests for service. Each request consists of the name of a vertex, and is satisfied by placing a server at the requested vertex. The requests must be satisfied in their order of occurrence. The cost of satisfying a sequence of requests is the distance moved by the servers. In this paper we study on-line algorithms for this problem from the *competitive* point of view. That is, we seek to develop on-line algorithms whose performance on any sequence of requests is as close as possible to the performance of the optimum off-line algorithm. We obtain optimally competitive algorithms for several important cases. Because of the flexibility in choosing the distances in the graph and the number of servers, the $k$-server problem can be used to model a number of important paging and caching problems. It can also be used as a building block for solving more general problems. We show how server algorithms can be used to solve a seemingly more general class of problems known as *task systems*. 

*Part of the work of the second author was done at Carnegie-Mellon University and was supported by an NSF Graduate Fellowship and by NSF Grants DCR-8352081 and MCS-8308805.

†Partial support provided by DARPA, ARPA order 4976, Amendment 20, monitored by the Air Force Avionics Laboratory under Contract F33615-87-C-1499, and by the National Science Foundation under Grant CCR-8658139.



0196-6774/90 $3.00

## 1. Introduction

Let $G$ be an $n$-vertex graph with positive edge lengths obeying the triangle inequality, and let $k$ mobile servers occupy vertices of $G$. Given a sequence of requests, each of which specifies a vertex that requires service, the *k-server problem* is to decide how to move the servers in response to each request. The initial locations of the servers are specified. If a requested vertex is unoccupied, then some server must be moved there. The requests must be satisfied in order of their occurrence in the request sequence. The cost of handling a sequence of requests is equal to the total distance moved by the servers. A server problem is *symmetric* if, for all vertices $i$ and $j$, the distance from $i$ to $j$ equals that from $j$ to $i$, and is *asymmetric* otherwise.

An *on-line* algorithm for solving the $k$-server problem operates under the additional constraint that it must decide which server to move to satisfy a given request without knowing what the future requests will be.

As we shall see later, the flexibility in choosing the distances between vertices and the number of servers allows us to use the $k$-server problem as a building block for solving other on-line problems. Furthermore, the $k$-server problem is a generalization of several important scheduling and caching problems. Here we mention three examples.

*Paging problems.* In a two-level memory system there are $k$ pages of fast memory and a total of $n$ pages of memory. A page fault occurs when a page which is not in fast memory is needed there. The paging problem is that of deciding which page to take out of fast memory when a page fault occurs. The goal is to minimize the number of page faults. This problem is a thinly disguised instance of the $k$-server problem in which all distances are one. The $n$ nodes of the graph correspond to the $n$ pages of address space, and the $k$ servers occupy the nodes corresponding to the pages currently in fast memory.

*Caching problems.* These are similar to paging problems, except that the costs of moving different items into the cache differ. An example is the caching of fonts in a printer or in the memory of a bitmap display. The parameter to be minimized is the transmission time. The display or printer can store the bitmaps of a fixed number of characters in its font memory. The number of bits required to transmit the bitmap of a character varies according to its complexity. These problems are instances of the asymmetric $k$-server problem where all move costs into a vertex are equal. Raghavan and Snir [10] note that they can be converted to instances of the symmetric $k$-server problem (see Section 8).

*Two-headed disks.* This is the problem of planning the motion of the heads of a two-headed disk drive along a linear track. Each request requires

that one of the heads be moved to a particular point along the line. The problem is to decide which head to move so that the total head movement is small. This is a 2-server problem in which the distance matrix is that of a set of points arranged in a line. This problem, as well as variants in which the servers are in a circle or on the surface of a sphere were considered by Calderbank, Coffman, and Flatto [3, 4]. All of these problems are instances of the 2-server problem.

We shall examine the $k$-server problem using an approach to on-line algorithms that was pioneered by Sleator and Tarjan [11]. They compared the move-to-front heuristic (MTF) for maintaining a linear search list to other list-maintenance heuristics. They showed that on any sequence of requests the performance of MTF is within a factor of four of the performance of any algorithm, even an *off-line* algorithm that can see all future requests.

Karlin, Manasse, Rudolph, and Sleator [6] applied this approach to multiprocessor caching problems, and Borodin, Linial, and Saks [2] applied it to a more general class of problems called task systems. The former group used the term *c-competitive* to refer to an on-line algorithm with performance that is within a factor of $c$ (plus a constant) of optimum on any sequence of requests. More formally, let $C_A(\sigma)$ denote the cost incurred by an algorithm $A$ in satisfying a request sequence $\sigma$. Algorithm $B$ is $c$-competitive if it is on-line and if there is a constant $a$ such that

$$\text{for all } A \text{ and } \sigma, \qquad C_B(\sigma) \leq c \cdot C_A(\sigma) + a. \tag{1.1}$$

An algorithm is *competitive* if it is $c$-competitive for some constant $c$. We will say that a competitive algorithm is efficient if the constant $c$ is small. The constant $c$ is called the *competitive factor*. An algorithm is *strongly competitive* if it achieves the smallest possible competitive factor.

In this paper we present efficient competitive algorithms for several classes of symmetric server problems. We also show how the server problems that we solve can be used as building blocks to obtain competitive algorithms for other problems. In Sections 3 and 4 we describe an $(n-1)$-competitive algorithm for the symmetric $(n-1)$-server problem and a 2-competitive algorithm for the symmetric 2-server problem, respectively. In Section 5 we show that the competitive factor of any algorithm for the symmetric $k$-server problem is at least $k$. Thus our algorithms for the $(n-1)$-server problem and the 2-server problem have the best possible competitive factors.

In Section 6 we introduce the problem of servers with excursions. This problem differs from the original server problem in that the algorithm has the option of moving a server to the requested vertex, but is not forced to.

It has the choice of making a less expensive "excursion" to the vertex, and returning home. We show how our competitive $(n-1)$-server algorithm can be used to develop a $(2n-1)$-competitive algorithm for the $(n-1)$-server problem with excursions.

There is a close relationship between our work on server problems and the work of Borodin, Linial, and Saks [2] on task systems. This connection is elucidated in Section 7. We show how our competitive algorithm for the $(n-1)$-server problem with excursions can be used to obtain a $(2n-1)$-competitive algorithm for a class of task systems that is almost (but not quite) as general as the class of Borodin *et al.* We also give an $(n-1)$-competitive algorithm for a slightly more restricted, but very natural, class of task systems called forced task systems.

Section 8 collects a number of open problems related to this work. The conjectures that we believe far outnumber the theorems that we have proven.

## 2. Preliminaries

We let $d_{uv}$ denote the length of edge $(u, v)$ in the graph $G$. We assume that the triangle inequality holds (for any three vertices $u$, $v$, and $w$, $d_{uw} \leq d_{uv} + d_{vw}$) and that the lengths are symmetric ($d_{uv} = d_{vu}$ for all $u$ and $v$).

An algorithm $B$ is called *lazy* if it moves a single server to handle each request at an unoccupied vertex, but does not move servers otherwise. The following lemma shows that we may restrict our attention to lazy algorithms.

Lemma 1. *For any algorithm $B$, there is a modified algorithm $B'$ that is lazy, does not cost more, and is on-line if $B$ is.*

*Proof.* This follows from the fact that the triangle inequality holds on edge lengths and is proven by induction. Suppose that on request sequence $\sigma$, $B$ is lazy until the $i$th request. We show that $B$'s response to that request can be changed to be lazy, without increasing its cost on $\sigma$.

Suppose $B$ moves a server $s$ from vertex $u$ to vertex $v$ on the $i$th request, but that the move is unnecessary because some other server is handling the request. Suppose that the next request that $s$ handles alone is at vertex $w$. Algorithm $B$'s cost for moving to $v$ and then to $w$ is $d_{uv} + d_{vw}$. If the unnecessary move is deleted, $s$ would eventually move directly from $u$ to $w$ at cost $d_{uw}$. By the triangle inequality $d_{uw} \leq d_{uv} + d_{vw}$. No other moves change, so the modified algorithm does not cost more. □

The algorithms described in this work will all be lazy. In order to prove that an algorithm is competitive within a certain factor, it will be sufficient to compare it to other lazy algorithms, since they outperform all others.

The on-line algorithms we describe have another property: they completely ignore requests for service at vertices that are already covered. These requests do not affect later decisions. It is easy to show that we only need to consider such algorithms.

Call a request sequence $\sigma$ *hard* for algorithm $B$ if $B$ must move some server in response to every request in $\sigma$. Algorithm $B$ is said to be $c$-competitive on all of its hard request sequences if there exists a constant $a$ such that $C_B(\sigma) \leq c \cdot C_A(\sigma) + a$ for all hard $\sigma$ and all algorithms $A$.

LEMMA 2. *If $B$ is a server algorithm that ignores all requests to covered vertices and is $c$-competitive on all of its hard request sequences, then $B$ is $c$-competitive.*

*Proof.* Let $B$ be such an algorithm and let $\sigma$ be a sequence of requests. When algorithm $B$ is run on $\sigma$, some of the requests force a server to move, and some do not. Let $\sigma'$ be the subsequence of $\sigma$ consisting of requests on which $B$ must move a server. Because $B$ ignores all requests in $\sigma$ to covered vertices, we know that $C_B(\sigma) = C_B(\sigma')$.

Let OPT be an off-line algorithm that handles every request sequence optimally. (It is obvious that such an algorithm exists because there are only a finite number of ways of handling each request sequence.) Clearly $C_{\text{OPT}}(\sigma') \leq C_{\text{OPT}}(\sigma)$ because it is impossible, by the triangle inequality, that the extra requests in $\sigma$ lead to a lower cost. Combining these bounds with the fact that $B$ is $c$-competitive on $\sigma'$, we get:

$$C_B(\sigma) = C_B(\sigma') \leq c \cdot C_{\text{OPT}}(\sigma') + a \leq c \cdot C_{\text{OPT}}(\sigma) + a.$$

This proves that $B$ is $c$-competitive on any sequence $\sigma$. □

Lemmas 1 and 2 together show that in order to prove that our algorithms are competitive, it is sufficient to compare them to lazy algorithms on hard sequences.

### 2.1. *An Optimal Off-line Algorithm*

Let $C_{\text{OPT}}(\sigma, S)$ be a function whose value is the cost of a minimum-cost algorithm (making lazy moves only) that handles request sequence $\sigma$ and ends up in state $S$ (covering a particular set of vertices). We can compute this function recursively as follows, assuming that the servers are initially

covering a set of vertices $S_0$:

$$C_{\mathrm{OPT}}(\varepsilon, S) = \begin{cases} 0, & \text{if } S = S_0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$C_{\mathrm{OPT}}(\sigma v, S) = \begin{cases} \min_T C_{\mathrm{OPT}}(\sigma, T) + d(T, S), & \text{if } v \text{ is covered in } S \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

where $d(T, S)$ is the cost of a transition (by a lazy move) from state $T$ to state $S$. This is a correct method to compute the function because the minimum-cost algorithm for reaching state $S$ at time $i$ must have been in some state $T$ at time $i - 1$.

A dynamic programming procedure can be used to compute the cost of an optimal algorithm handling request sequence $\sigma$. Build a table with $|\sigma| + 1$ rows (one for each prefix of $\sigma$) and $\binom{n}{k}$ columns (one for each possible state). The entry in row $i$ and column $j$ is $C_{\mathrm{OPT}}(\sigma_i, S_j)$, where $\sigma_i$ is the prefix of $\sigma$ of length $i$. Each row of the table can be built from the previous one within time proportional to $\binom{n}{k}^2$. Upon completion, the smallest entry in the last row is the cost of a minimum-cost algorithm that processes sequence $\sigma$. Furthermore, the table-building procedure can be modified to actually produce an optimal sequence of moves. Every entry in the table equals some entry on the previous row plus (perhaps) the cost of a move. By saving this information, it is easy to follow pointers back from any minimum entry on the last row and determine a sequence of moves that achieves that cost. It is possible for an on-line algorithm to maintain the current row of this table.

### 2.2. *Residues*

Based on $C_{\mathrm{OPT}}(\sigma, S)$ we can now define a new function which measures how well an on-line algorithm $B$ is doing compared to the optimal. Call $R_c(\sigma, S)$ the *c-residue function* and define it as

$$R_c(\sigma, S) = c \cdot C_{\mathrm{OPT}}(\sigma, S) - C_B(\sigma). \tag{2.1}$$

This function has a number of useful properties. First, it is easy to compute. Call $\langle R_c(\sigma, S_1), R_c(\sigma, S_2), \ldots \rangle$ (for some enumeration of states) the *c-residue vector* for sequence $\sigma$. Because of the dynamic programming method of computing $C_{\mathrm{OPT}}$, any on-line algorithm can compute its current vector of residues. The time required for this is bounded by a small polynomial if the number of servers is small.

Residues are also useful because they can be used to prove that algorithms are competitive. The minimal cost of handling sequence $\sigma$, as noted

before, is $\min_S \{C_{OPT}(\sigma, S)\}$. Therefore for any algorithm $A$,

$$c \cdot C_A(\sigma) - C_B(\sigma) \geq \min_S \{ R_c(\sigma, S) \}.$$

By Eq. (1.1), $B$ is then a $c$-competitive algorithm if there is some constant $a$ such that $R_c(\sigma, S) \geq -a$ for any $\sigma$ and $S$. Our proofs of competitiveness will use this approach.

The correctness of proofs using residues does not depend on the starting residues, which can be set to arbitrary constants. Suppose that, for some set of initial residues, there is a constant lower bound on all residues. That bound will hold (or be improved) by raising the initial residues for non-starting states to $\infty$, effectively making them impossible. Furthermore, the bound will vary by at most a constant if the residue for the starting state is set to any constant.

It should be noted that the definition of residues used here differs slightly from the definition appearing in [8]. The proofs in this paper are greatly simplified by allowing some freedom in the initial residue values and by defining $C_{OPT}(\sigma, S)$ in terms of opposing algorithms that make only lazy moves.

## 3. An $(n-1)$-Competitive Algorithm for $n-1$ Servers

The *balance algorithm* (BAL) for the $k$-server problem works as follows. For each server, the algorithm maintains the total distance it has moved since the start of the request sequence. If the server is currently at vertex $i$, this cumulative distance is denoted by $D_i$. Now consider a request at a vertex $j$. If $j$ is already covered by a server, then BAL does nothing. If $j$ is not covered, then BAL moves the server on vertex $i$ to vertex $j$, where $i$ is chosen to minimize the expression $D_i + d_{ij}$. In other words, BAL moves any server that would have the smallest cumulative cost after moving.

As indicated by its name, the balance algorithm tends to use all of its servers equally. The following lemma makes this precise.

Lemma 3. *Let* 1 *be the most recently covered vertex,* $n$ *be the uncovered vertex, and* $i$ *be any vertex occupied by a server. The bound*

$$-d_{1i} \leq D_i - D_1 \leq d_{in} - d_{1n}$$

*holds for any* $i$*, at any time after the first move of BAL.*

*Proof.* We shall prove by induction that at any time and for all $i$ and $j$ $D_i - D_j \leq d_{ij}$. This claim is clearly true initially. Suppose it is true up to some point and that a new request arrives for vertex $n$. Suppose BAL

moves from vertex $k$ in response to the request, implying that $D_k + d_{kn} \leq D_i + d_{in}$ for all $i$. Vertex $n$ becomes the new "most recently covered" vertex, and we therefore refer to it as $1'$. Vertex $k$ is the new vacant vertex, which we call $n'$. The new cumulative distance for the server at $1'$, denoted $D_{1'}$, equals $D_k + d_{kn}$. The other cumulative distances are unchanged. Only two cases need to be considered to prove that the claim holds inductively. First, for any $i$,

$$D'_{1'} - D'_i = D_k + d_{kn} - D_i \leq d_{in} = d_{1'i}.$$

(The inequality here follows from the decision rule.) Further,

$$D'_i - D'_{1'} = D_i - D_k - d_{kn} \leq d_{ik} - d_{kn} = d_{in'} - d_{1'n'} \leq d_{1'i}. \quad (3.1)$$

(The inequalities here follow from the inductive hypothesis and the triangle inequality.) This completes the inductive proof of the claim. The claim directly implies the left-side inequality of the lemma. The right-side inequality is proven in eq. 3.1. □

We can now prove the following theorem.

THEOREM 4. *Algorithm BAL is an* $(n-1)$*-competitive algorithm for the symmetric* $(n-1)$*-server problem on an n-vertex graph.*

*Proof.* We consider a vector of residues that compares BAL's cost to $(n-1)$ times the cost of an optimal algorithm. We show that there is a constant lower bound on these residues, proving the theorem.

There are $n-1$ ways an algorithm can use $n-1$ servers to simultaneously cover a requested vertex and $n-2$ other vertices. This means there are $n-1$ non-infinite residues at any step. Let $R_i$ denote a residue that compares BAL's cost to $(n-1)$ times the cost of an algorithm that leaves vertex $i$ uncovered. Let 1 be the most-recently requested vertex, let $n$ be the vertex BAL is not covering, and let $2, \ldots, n-1$ be the other vertices. Let the initial $R_n$ equal $-(n-1)d_{1n}$, and let $R_i$ $(1 < i < n)$ equal 0. $R_1$ is undefined because vertex 1 must be covered by any algorithm.

As shown in Lemma 2, it is only necessary to consider the request sequence that always hits $n$, BAL's vacant vertex. For this sequence, we show that the following invariant holds:

$$R_j = \begin{cases} (n-1)(D_1 - d_{1n}) - \Sigma_k D_k, & \text{if } j = n \\ (n-1)D_j - \Sigma_k D_k & \text{otherwise.} \end{cases} \quad (3.2)$$

In conjunction with Lemma 3 (which bounds the difference between $D_i$ and $D_j$ for all $i$ and $j$) this implies a constant lower bound on the residues. The

lemma also implies that

$$\text{for all } i \text{ and } j, \qquad R_i \le R_j + (n-1)\,d_{ij}. \tag{3.3}$$

The inductive step requires us to prove that if the invariants hold, then they will hold after a request at vertex $n$. If $i$ minimizes the expression $D_i + d_{in}$, then BAL handles the request by moving from $i$ to $n$. Vertex $n$ becomes vertex $1'$, and vertex $i$ becomes vertex $n'$. The new cumulative distance for the server now at $1'$, denoted $D'_{1'}$, equals $D_i + d_{in}$. For every other server, the cumulative distance is unchanged.

The new residues are described by the equation

$$R'_j = \begin{cases} R_n + (n-1)\,d_{1n} - d_{in}, & \text{if } j = 1 \\ \min\{R_j, R_n + (n-1)\,d_{jn}\} - d_{in}, & \text{otherwise.} \end{cases}$$

Using inequality (3.3), we obtain

$$R'_j = \begin{cases} R_n + (n-1)\,d_{1n} - d_{in} & \text{if } j = 1 \\ R_j - d_{in} & \text{otherwise.} \end{cases}$$

Using invariant (3.2), we obtain

$$R'_j = (n-1)\,D_j - \Sigma_k D_k - d_{in}.$$

Finally, using the update rule for cumulative distances, we obtain

$$R'_j = \begin{cases} (n-1)(D'_{1'} - d_{1'n'}) - \Sigma_k D'_k, & \text{if } j = n' \\ (n-1)\,D'_j - \Sigma_k D'_k, & \text{otherwise.} \end{cases} \tag{3.4}$$

This completes the inductive step. □

## 4. A 2-Competitive Algorithm for 2 Servers

Unfortunately, the balance algorithm described in Section 3 is not $k$-competitive when the number of servers is less than $n - 1$. In attempting to find a 2-competitive algorithm for the 2-server problem, we ruled out BAL as well as many other simple approaches. We have obtained an algorithm RES that maintains certain invariants on its residues and chooses which server to move by comparing the residues.

Algorithm RES always keeps its two servers on two different vertices. Let vertex 1 be the vertex that was last requested, and let vertex 2 be the other covered vertex. An off-line algorithm must also cover vertex 1, but its other server could be anywhere. This means there are again $(n-1)$ non-infinite residues. Algorithm RES maintains residues that compare the cost incurred by RES to twice the cost of an optimal algorithm. Let $R_{1i}$ denote the residue that compares RES to an off-line algorithm occupying vertices 1 and $i$.

Algorithm RES begins with $R_{1i} = d_{12} + 2d_{2i}$ for each $i$. In response to a request at vertex $i$, RES moves from vertex 1 if

$$\min_k \{ R_{1k} + 2d_{ki} \} \geq 2d_{1i} + d_{12}$$

and from vertex 2 otherwise.

To prove that RES is 2-competitive, we need the function,

$$\gamma(v, w, x, y) = 2 \cdot \max\{ d_{vw} + d_{xy}, d_{vx} + d_{wy}, d_{vy} + d_{wx} \}. \quad (4.1)$$

This function has two important properties. First, its value does not change if its arguments are permuted. Second, the triangle inequality ensures that

$$\gamma(u, v, w, x) + 2d_{uy} \geq \gamma(y, v, w, x). \quad (4.2)$$

THEOREM 5. *Algorithm RES is a 2-competitive algorithm for the symmetric 2-server problem.*

*Proof.* Induction and a case analysis suffice to prove that RES maintains the following bound for every pair of vertices $i$ and $j$ (where possibly $i = j$):

$$R_{1i} + R_{1j} \geq \gamma(1, 2, i, j). \quad (4.3)$$

These bounds guarantee the 2-competitiveness of the algorithm.

Initially, $R_{1i} = d_{12} + 2d_{2i}$. For any $i$ and $j$,

$$R_{1i} + R_{1j} = 2(d_{12} + d_{2i} + d_{2j}).$$

By the triangle inequality, this is at least $2(d_{12} + d_{ij})$, $2(d_{1i} + d_{2j})$, and $2(d_{1j} + d_{2i})$, and therefore is at least $\gamma(1, 2, i, j)$.

We now inductively assume that the invariant holds after some number of requests and that a new request arrives for an unoccupied vertex $i$. There are two cases. Suppose that $\min_k\{ R_{1k} - 2d_{ki} \} \geq 2d_{1i} + d_{12}$ and that RES

therefore moves from 1. By the naming convention, $i$ becomes the new vertex 1, denoted 1′. The location of RES's other server, 2, is unchanged. The updated residues are

$$R'_{1'j} = \begin{cases} \min_k\{ R_{1k} + 2d_{ki}\} - d_{1i} & \text{if } j = 1 \\ R_{1j} + d_{1i} & \text{otherwise.} \end{cases}$$

The following cases show that bound (4.3) holds after RES moves. Let $k$ be such that $\min_k\{R_{1k} + 2d_{ki}\}$ is minimized:

$$\begin{aligned} & R'_{1'1} + R'_{1'1} = 2(R_{1k} + 2d_{ki} - d_{1i}) \\ & \qquad \geq 2((2d_{1i} + d_{12}) - d_{1i}) && \text{decision rule} \\ & \qquad = 2(d_{1i} + d_{12}) \\ & \qquad = 2(d_{11'} + d_{12'}) \\ & \qquad = \gamma(1', 2', 1, 1) \\ (j \neq 1) \qquad & R'_{1'1} + R'_{1'j} = R_{1k} + 2d_{ki} + R_{1j} \\ & \qquad \geq \gamma(1, 2, k, j) + 2d_{ki} && (4.3) \\ & \qquad \geq \gamma(1, 2, i, j) && (4.2) \\ & \qquad = \gamma(1, 2', 1', j) \\ (j, l \neq 1) \qquad & R'_{1'j} + R'_{1'l} = R_{1j} + R_{1l} + 2d_{1i} \\ & \qquad \geq \gamma(1, 2, j, l) + 2d_{1i} && (4.3) \\ & \qquad \geq \gamma(i, 2, j, l) && (4.2) \\ & \qquad = \gamma(1', 2', j, l). \end{aligned}$$

Alternatively if $\min_k\{R_{1k} + 2d_{ki}\} < 2d_{1i} + d_{12}$, algorithm RES moves from 2. Again, $i$ becomes 1′. In this case 1 also becomes 2′. The new residues are:

$$R'_{1'j} = \begin{cases} \min_k\{ R_{1k} + 2d_{ki}\} - d_{2i}, & \text{if } j = 2' \\ R_{1j} + 2d_{1i} - d_{2i}, & \text{otherwise.} \end{cases}$$

A case analysis shows that bound (4.3) holds when RES moves. Again, let $k$ be such that $\min_k\{R_{1k} + 2d_{ki}\}$ is minimized. The first two cases are

straightforward:

$$\begin{aligned} R'_{1'2'} + R'_{1'2'} &= 2(R_{1k} + 2d_{ki} - d_{2i}) \\ &\geq 2(d_{1k} + d_{2k} + 2d_{ki} - d_{2i}) \qquad (4.3) \\ &\geq 2(d_{1i} + d_{2i} - d_{2i}) \qquad \text{triangle inequality} \\ &= 2d_{1i} \\ &= 2d_{1'2'} \\ &= \gamma(1',2',2',2') \end{aligned}$$

$$\begin{aligned} (j \neq 2') \qquad R'_{1'2'} + R'_{1'j} &= R_{1k} + 2d_{ki} + R_{1j} + 2d_{1i} - 2d_{2i} \\ &\geq \gamma(1,2,j,k) + 2d_{ki} + 2d_{1i} - 2d_{2i} \qquad (4.3) \\ &\geq \gamma(1,2,j,i) + 2d_{1i} - 2d_{2i} \qquad (4.2) \\ &\geq 2d_{2i} + 2d_{1j} + 2d_{1i} - 2d_{2i} \qquad (4.1) \\ &= 2d_{1i} + 2d_{1j} \\ &= \gamma(i,1,1,j) \\ &= \gamma(1',2',2',j) \end{aligned}$$

In order to prove the final case, $R'_{1'j} + R'_{1'l} \geq \gamma(1',2',j,l)$, for $j, l \neq 2'$, it is necessary to compare the sum of residues to each of the three terms in $\gamma$:

$$\begin{aligned} R'_{1'j} + R'_{1'l} &= R_{1j} + R_{1l} + 4d_{1i} - 2d_{2i} \\ &= (R_{1j} + R_{1k}) + (R_{1l} + R_{1k}) - 2R_{1k} + 4d_{1i} - 2d_{2i} \\ &\geq \gamma(1,2,j,k) + \gamma(1,2,k,l) - 2R_{1k} + 4d_{1i} - 2d_{2i} \qquad (4.3) \\ &\geq 2(d_{1j} + d_{2k} + d_{12} + d_{kl} - R_{1k} + 2d_{1i} - d_{2i}) \qquad (4.1) \\ &> 2(d_{1j} + d_{2k} + d_{12} + d_{kl} - (2d_{1i} + d_{12} - 2d_{ki}) \\ &\qquad + 2d_{1i} - d_{2i}) \qquad \text{decision rule} \\ &\geq 2(d_{1j} + d_{il}) \qquad \text{triangle inequality} \\ &= 2(d_{2'j} + d_{1'l}) \end{aligned}$$

$$R'_{1'j} + R'_{1'l} > 2(d_{2'l} + d_{1'j}) \qquad \text{symmetric to first case}$$

$$\begin{aligned} R'_{1'j} + R'_{1'l} &= R_{1j} + R_{1l} + 4d_{1i} - 2d_{2i} \\ &\geq \gamma(1,2,j,l) + 4d_{1i} - 2d_{2i} \qquad (4.3) \\ &\geq 2(d_{12} + d_{jl} + 2d_{1i} - d_{2i}) \qquad (4.1) \\ &\geq 2(d_{1i} + d_{jl}) \qquad \text{triangle inequality.} \\ &= 2(d_{1'2'} + d_{jl}) \end{aligned}$$

This completes the proof that the bounds hold inductively and that Theorem 5 holds. □

## 5. A Lower Bound

In this section we prove that any general algorithm for the symmetric $k$-server problem must have a competitive factor of at least $k$. This implies that algorithms BAL and RES have the best possible competitive factors for the symmetric $(n-1)$-server and 2-server problems, respectively.

We have actually proven a slightly more general lower bound on the competitive factor. Suppose we wish to compare an on-line algorithm with $k$ servers to off-line algorithms with $h \leq k$ servers. Naturally, giving the on-line algorithm more servers than the off-line algorithm decreases the "competitive factor." We prove a lower bound of $k/(k-h+1)$ on this factor. A similar approach was taken in [11], where this lower bound and a matching upper bound are given for the paging problem.

THEOREM 6. *Let $A$ be an on-line algorithm for the symmetric $k$-server problem on a graph $G$ with at least $k+1$ nodes. Then, for any $1 \leq h \leq k$, there exist request sequences $\sigma_1, \sigma_2, \ldots$ such that:*

1. *For all $i$, $\sigma_i$ is an initial subsequence of $\sigma_{i+1}$, and $C_A(\sigma_i) < C_A(\sigma_{i+1})$.*
2. *There exists an $h$-server algorithm $B$ (which may start with its servers anywhere) such that for all $i$,*

$$C_A(\sigma_i) > \frac{k}{k-h+1} \cdot C_B(\sigma_i).$$

*Proof.* Without loss of generality, assume $A$ is lazy and that the $k$ servers start out at different nodes. Let $H$ be a subgraph of $G$ of size $k+1$ induced by the $k$ initial positions of $A$'s servers and one other vertex.

Define $\sigma$, $A$'s nemesis sequence on $H$, such that $\sigma(i)$ is the unique vertex in $H$ not covered by $A$ at time $i$, for all $i \geq 1$. Then

$$C_A(\sigma, t) = \sum_{i=1}^{t} d_{\sigma(i+1),\sigma(i)},$$

because at each step $\sigma$ requests the node just vacated by $A$.

Let $S$ be any $h$-element subset of $H$ containing $\sigma(1)$. We can define an off-line $h$-server algorithm $A(S)$ as follows: The servers initially occupy the

vertices in the set $S$. To process a request $\sigma(i)$, the following rule is applied:

> If $S$ contains $\sigma(i)$ do nothing. Otherwise, move the server at node $\sigma(i-1)$ to $\sigma(i)$, and update $S$ to reflect this change.

It is easy to see that for all $i > 1$, the set $S$ contains $\sigma(i-1)$ when step $i$ begins.

The following observation is the key to the rest of the proof: if we run the above algorithm starting with distinct equal-sized sets $S$ and $T$, then $S$ and $T$ never become equal, for the reason described in the following paragraph.

Suppose that $S$ and $T$ differ before $\sigma(i)$ is processed. We shall show that the versions of $S$ and $T$ created by processing $\sigma(i)$ as described above also differ. If both $S$ and $T$ contain $\sigma(i)$, neither is changed, and there is nothing more to prove. Otherwise, we are not processing $\sigma(1)$, so both $S$ and $T$ contain $\sigma(i-1)$. If exactly one of $S$ or $T$ contains $\sigma(i)$, then after the request exactly one of them contains $\sigma(i-1)$, so they still differ. If neither of them contains $\sigma(i)$, then both change by dropping $\sigma(i-1)$ and adding $\sigma(i)$, so the symmetric difference of $S$ and $T$ remains the same (non-empty).

Let us consider simultaneously running an ensemble of algorithms $A(S)$, starting from each $h$-element subset $S$ of $H$ containing $\sigma(1)$. There are $\binom{k}{h-1}$ such sets. Since no two sets ever become equal, the number of sets remains constant. After processing $\sigma(i)$, the collection of subsets consists of all the $h$ element subsets of $H$ which contain $\sigma(i)$.

By our choice of starting configuration, step 1 never costs anything. At step $i+1$ (for $i \geq 1$), each of these algorithms either does nothing (at no cost) or it moves a server from $\sigma(i)$ to $\sigma(i+1)$, at cost $d_{\sigma(i)\sigma(i+1)}$. Of the $\binom{k}{h-1}$ algorithms being run, $\binom{k-1}{h-1}$ of them (the ones which contain $\sigma(i)$ but do not contain $\sigma(i-1)$) incur this cost, and the rest incur no cost. So for step $i+1$ the total cost incurred by all of the algorithms is $\binom{k-1}{h-1} \cdot d_{\sigma(i)\sigma(i+1)}$. The total cost of running all of these algorithms up to and including $\sigma(t)$ is

$$\sum_{i=1}^{t-1} \binom{k-1}{h-1} \cdot d_{\sigma(i)\sigma(i+1)}.$$

Thus the expected cost of one of these algorithms chosen at random is

$$\frac{\binom{k-1}{h-1}}{\binom{k}{h-1}} \cdot \sum_{i=1}^{t-1} d_{\sigma(i)\sigma(i+1)}.$$

Recall that the cost to $A$ for the same steps was

$$\sum_{i=1}^{t} d_{\sigma(i+1)\sigma(i)}.$$

Because the distances are symmetric, the two summations are identical, except that the second one includes one extra term.

By expanding the binomial coefficients we see that

$$\frac{\binom{k-1}{h-1}}{\binom{k}{h-1}} = \frac{k-h+1}{k}.$$

Finally, there must be some initial set that has the property that infinitely often its performance is no worse than the average of the costs. Let $S$ be this set, and $A(S)$ be the algorithm starting from this set. Let $\sigma_i$ be all the initial subsequences of $\sigma$ for which $A(S)$ does no worse than average. □

This theorem gives a lower bound of $k/(k-h+1)$ on the competitive factor, for even if we require our off-line algorithm to start with its servers in particular locations, we can move the servers wherever we choose at the cost of an additive constant.

COROLLARY 7. *For any symmetric k-server problem, there is no c-competitive algorithm for* $c < k$.

COROLLARY 8. *BAL is a strongly competitive algorithm for the symmetric* $(n-1)$*-server problem.*

COROLLARY 9. *RES is a strongly competitive algorithm for the symmetric* 2*-server problem.*

## 6. Server Problems with Excursions

Suppose we allow the servers to satisfy a request without actually moving to the requested vertex. We call this type of response an *excursion* because it is natural to think of the server sending off an assistant to make an excursion to the requested vertex. The assistant satisfies the request, then returns to the starting point. Let $r_{ij}$ be the cost for a server at vertex $i$ to make an excursion to vertex $j$.

A natural example of a server problem with excursions is that of where to locate $k$ firehouses. In this case $d_{ij}$ is the cost of moving the firehouse from $i$ to $j$, and $r_{ij}$ is the cost for the firehouse at $i$ to put out a fire at location $j$. In order to obtain a competitive algorithm for this problem, it is clear that

if there are many fires at a particular location, then it will be necessary to move a firehouse there, even if moving a firehouse is very expensive.

Very little is known about the general problem of servers with excursions. Special cases that have been considered are 1-server on a graph that is a tree [1], and one server on a real line.[1] The first result assumes that the cost of an excursion between vertices $i$ and $j$ is proportional to the distance between them and obtains a competitive factor of three (the best possible). The second result assumes that the excursion cost is the move cost and obtains a competitive factor of $\frac{3}{2}$ (the best possible).

In this section we address the problem of $(n-1)$ servers with excursions on a graph of $n$ vertices. As shown in Section 7, the task systems of Borodin, Linial, and Saks [2] are essentially equivalent to these server problems. A lower bound on the competitive factor for the $(n-1)$-server problem with excursions gives a lower bound on the competitive factor for task systems. Similarly, a $c$-competitive algorithm for the $(n-1)$-server problem with excursions gives a $c$-competitive algorithm for task systems. (The latter statement actually only applies to a slightly restricted form of task systems, those with discrete tasks.) Furthermore, the $(2n-1)$-competitive algorithm of Borodin *et al.* for task systems can be adapted to give a $(2n-1)$-competitive algorithm for the $(n-1)$-server problem with excursions.

For a particular problem, let $r_u$ be the ratio of the largest excursion cost to the smallest move cost, and let $r_l$ be the ratio of the smallest excursion cost to the largest move cost. As a corollary of the lower bound theorem of [2], the competitive factor of $2n-1$ is the best possible as $r_u$ goes to zero. As $r_l$ goes to two, the problem is reduced to an ordinary server problem, because it is no longer ever useful to do an excursion. As we have shown, in this case we can obtain a competitive factor of $n-1$. For problems in which the values of $r_l$ and $r_u$ are not so extreme, no exact results are known except that the competitive factor lies between $n-1$ and $2n-1$. Perhaps in these cases the competitive factor actually depends on the distances and excursion costs.

We have developed a construction that allows us to apply our results on the $(n-1)$-server problem to the $(n-1)$-server problem with excursions. Using this technique and the ideas of the proof of Theorem 6 we have obtained a lower bound of $(2n-1)(1+r_u)/(1+2r_u)$ (assuming $r_u \leq 2$) on the competitive factor for the $(n-1)$-server problem with excursions. This slightly improves the lower bound of $(2n-1)/(1+r_u)$ of Borodin *et al.* [2].

This construction allows us to apply algorithm BAL to give a new $(2n-1)$-competitive algorithm for the $(n-1)$-server problem with excur-

[1] Unpublished notes by L. McGeoch.

sions. The remainder of this section is devoted to describing and analyzing this algorithm.

In the $(n-1)$-server problem with excursions, all vertices but one are covered by a server, so an excursion to a particular vertex always costs the same amount. Let the excursion cost for vertex $i$ be $r_i$, where $r_i = \min_j r_{ji}$.

Our algorithm for the $(n-1)$-server problem with excursions is called BALE. It works by mapping the $(n-1)$-server problem on a graph $G$ of $n$ vertices onto a $(2n-1)$-server problem (without excursions) in a graph $G'$ of $2n$ vertices. It then applies algorithm BAL to the $2n$-vertex problem and maps the resulting moves back to the $n$-vertex problem.

We obtain $G'$ from $G$ by making two copies of each vertex of $G$. The distances in $G'$ are defined as follows: If $i$ and $j$ are two vertices of $G'$ that came from the same vertex of $G$ (namely $k$), then the distance between them is $r_k$. If $i$ and $j$ came from different vertices of $G$, then the distance between them in $G'$ is the same as the corresponding distance in $G$. Two vertices of $G'$ that come from the same vertex of $G$ will be called *siblings*. (If $r_k$ is more than twice the distance between $k$ and some other vertex, then $G'$ will violate the triangle inequality. In this case, the excursion is so expensive that it will never be used, and we can replace $r_k$ by twice the distance to the nearest neighbor of $k$ without changing the problem.)

Algorithm BALE maintains the following *server invariant*:

> If vertex $i$ of $G$ is the one not occupied by a server, then one of the vertices of $G'$ corresponding to $i$ is not occupied by a server.

If the requested vertex $i$ is already covered, then BALE does nothing. If the requested vertex is unoccupied, then BALE issues a request for the uncovered vertex of $G'$ in its simulation of BAL. If BAL responds to the request by moving the server from a vertex to its sibling, then BALE satisfies its request by doing an excursion. If BAL responds to the request by moving a server from some other vertex, then BALE responds by moving a server from the corresponding vertex in $G$. The cost incurred by BALE is exactly that incurred by BAL in the simulation.

THEOREM 10. *Algorithm BALE is a* $(2n-1)$*-competitive algorithm for the symmetric* $(n-1)$*-server problem with excursions.*

*Proof.* Let $\sigma$ be the sequence of requests in the $(n-1)$-server problem with excursions. Let $\sigma'$ be the sequence of requests issued by BALE to its simulation of BAL. Given any algorithm $A$ for satisfying $\sigma$ with $(n-1)$ servers in $G$ there is an algorithm $A'$ for satisfying $\sigma'$ in the $(2n-1)$-server problem in $G'$ such that $A$ and $A'$ maintain the server invariant. The cost incurred by algorithm $A'$ is at most that incurred by $A$ for the following reason: if the request is to a covered vertex in $G$, then it also to a covered

vertex in $G'$, if the request is processed by an excursion in $G$, then it is either free in $G'$ or costs as much as a move from one vertex to its sibling, if the request is processed by moving a vertex in $G$, then the corresponding move can be made in $G'$ at the same cost. That is, for any algorithm $A$ there is an algorithm $A'$ such that

$$C_{A'}(\sigma') \leq C_A(\sigma).$$

Because BAL is $(2n-1)$-competitive (for $2n-1$ servers on $2n$ vertices), we know that for any algorithm $A'$,

$$C_{\mathrm{BAL}}(\sigma') \leq (2n-1)C_{A'}(\sigma') + a.$$

Furthermore from the definition of BALE it follows that

$$C_{\mathrm{BALE}}(\sigma) = C_{\mathrm{BAL}}(\sigma').$$

These three inequalities show that BALE is $(2n-1)$-competitive. □

## 7. Task Systems

In [2], Borodin, Linial, and Saks considered *task systems*, a class of on-line problems more general than server problems. They proved upper and lower bounds on the best competitive factors that can be achieved for these problems. There is a very close relationship between their work and server problems. In this section we describe this relationship and extend their results.

A task system is specified by an integer $n$, and an $n$ by $n$ positive real matrix $D$ satisfying the triangle inequality. The system has $n$ *states*, labeled $1, 2, \ldots, n$. The entry of $d_{ij}$ of $D$ is the cost of changing from state $i$ to state $j$. A sequence of tasks $T(1), T(2), \ldots, T(N)$ is to be accomplished in order by an on-line algorithm. Each task is an $n$-dimensional vector of non-negative real numbers which specify the cost of doing this task in each of the $n$ states. As each task is received, the on-line algorithm has the option of changing from its current state $i$ to any other state $j$ at a cost of $d_{ij}$. It then does the required task in the new state at a cost specified in the task vector. Of course, the algorithm is not allowed to see the future tasks while it is making its decision about how to process the current task.

A symmetric, or metrical, task system is one in which the matrix $D$ is symmetric. Borodin *et al.* [2] give an on-line algorithm that is $(2n-1)$-competitive for any metrical task system of $n$ states. They also show that

for any task system and on-line algorithm there are sequences of tasks which force the algorithm to reach a competitive factor of at least $2n - 1$.

Here we give a different algorithm which obtains the same competitive factor of $2n - 1$ for almost arbitrary sequences of tasks. Our algorithm works as long as each task is an integer combination of a set of $n$ basis vectors $b_1, \ldots, b_n$, where vector $b_i$ is zero in all components except the $i$th. The basis set is fixed and known in advance. (For example, if all the task vectors are known to have integral costs, then we let $b_i$ to be the unit vector with a one in position $i$ and zeros elsewhere.) A set of tasks having this structure will be called *discrete tasks*.

Our $(2n - 1)$-competitive algorithm for discrete tasks is called GBALE, because it is a generalized form of the BALE algorithm given in Section 6. Our algorithm will use a simulation of an instance of BALE running on an $(n - 1)$-server problem with excursions. The size of the server problem is the same as the number of states of the task system, and the distance matrix for the server problem, $D$, is the same as the distance matrix for the task system. Furthermore, the cost of an excursion for vertex $i$, $r_i$ will be the non-zero component of $b_i$, the $i$th basis vector for the discrete tasks.

To process a sequence of tasks $T = T(1), T(2), \ldots$ algorithm GBALE simulates algorithm BALE, on a new sequence of requests $\sigma = \sigma(1), \sigma(2), \ldots$. For each task $T(i)$, a sequence of zero or more requests of $\sigma$ are generated. Before each task is processed (as well as after) the following invariant is maintained relating the state of BALE and GBALE.

> The vertex that BALE has decided not to occupy with a server is the same as the state of the task system chosen by GBALE.

It remains to describe the sequence of requests corresponding to a task $T(i)$. First expand $T(i)$ into a non-negative integer combination of the basis vectors as follows:

$$T(i) = m_1 b_1 + m_2 b_2 + \cdots + m_n b_n.$$

The requests generated are $m_1$ requests to vertex 1, $m_2$ requests to vertex 2, and so on, ending with $m_n$ requests to vertex $n$. We shall let $\tau(i)$ denote this subsequence of $\sigma$.

In order to process $T(i)$, algorithm GBALE gives $\tau(i)$ to BALE. The state reached by BALE at the end of $\tau(i)$ is the state chosen by GBALE to process request $T(i)$. The invariant is thus maintained.

THEOREM 11. *Algorithm GBALE is an* $(2n - 1)$*-competitive algorithm for any metrical task system with discrete tasks.*

*Proof.* First we shall prove the claim

$$C_{\text{GBALE}}(T(i)) \leq C_{\text{BALE}}(\tau(i)).$$

Observe that the move cost incurred by GBALE is at most that of BALE. This is because GBALE and BALE start and end in the same state. By the triangle inequality, the cheapest way to effect this change is to move there directly, which is what GBALE does.

Let $j$ be the state (the vertex that is unoccupied by a server) in which BALE chooses to process the last request of $\tau(i)$. (This is the state chosen by GBALE to process $T(i)$.) The task cost incurred by GBALE in processing $T(i)$ is $m_j r_j$. The excursion costs incurred by BALE are at least this much for the following reason. We know that during the last request to vertex $j$ in $\tau(i)$, algorithm BALE is in state $j$. This is because BALE will never change its state in response to a free request, such as those after the last request to $j$. Furthermore, BALE must have been in state $j$ during all of the requests to $j$, because BALE will never move into state $j$ in response to a request to $j$. Therefore, BALE must have been in state $j$ during all of the requests to $j$, and therefore incurs a cost of at least $m_j r_j$. This completes the proof of the claim.

Next, we shall relate the costs of the optimal algorithms for processing $T$ and $\sigma$. Given an off-line algorithm $A$ for processing $T$ there is an off-line algorithm $A'$ for processing $\sigma$ at exactly the same cost. Algorithm $A'$ stays in the same state during all of the requests of $\tau(i)$, and this is the state used by $A$ to process $T(i)$. From this, we can conclude:

$$C_{\text{OPT}}(\sigma) \leq C_{\text{OPT}}(T).$$

Combining the above two inequalities with the fact that BALE is $(2n - 1)$-competitive proves the theorem. □

There is a natural class of tasks which was not considered by Borodin *et al.* [2], for which we have obtained an algorithm with a better competitive factor. This is the case where each component of each task vector is either 0 or $\infty$. (This means that to process a task, the algorithm must change to a state in which the task cost is 0.) We call such a task a *forcing task*. In this case we obtain an $(n - 1)$-competitive algorithm.

Given a forcing task system $S$ in which there are state transition costs that are zero, we can transform it to an equivalent task system $S'$ in which there are no such zero-cost transitions. We do this by dividing the states of $S$ into equivalence classes, where two states are equivalent if and only if the transition cost between them is zero. We define $S'$ to have one state for each of these equivalence classes. It is easy to see that any competitive on-line algorithm for $S'$ gives one with the same competitive factor for $S$. A

task $t$ given to $S$ is translated into $t'$, a task for $S'$ as follows: A component of $t'$ is $\infty$ if all the states in the equivalence class of that component are $\infty$ in $t$, and 0 otherwise.

Our algorithm, GBAL, is built upon BAL, the $(n-1)$-competitive algorithm for the $(n-1)$-server problem. The construction is very similar (but not identical) to that used above in constructing GBALE.

For a task sequence $T$, GBAL generates a request sequence $\sigma$ which it applies to BAL. For task $T(i)$ in $T$ there is a subsequence $\tau(i)$ in $\sigma$. The invariant is maintained that before and each after $T(i)$, the state of GBAL and BAL are the same. (The state of BAL is the name of the vertex not covered by a server.)

The subsequence $\tau(i)$ is generated as follows. Let $S$ be the set of states for which the task component in $T(i)$ is infinite. Let $s$ be the current state of BAL. As long as $s \in S$, a request to $s$ is generated. Eventually, since BAL is competitive, and there are no zero-cost transitions, a point must be reached where $s \notin S$. This marks the end of $\tau(i)$. Algorithm GBAL can now use this state $s$ to process the task $T(i)$, since it is one of the zero-cost states of $T(i)$.

THEOREM 12. *Algorithm GBAL is an* $(n-1)$*-competitive algorithm for any metrical task system with forcing tasks.*

*Proof.* We have $C_{\text{GBAL}}(T(i)) \leq C_{\text{BAL}}(\tau(i))$, because of the triangle inequality, and the fact that BAL and GBAL start and end in the same state.

Furthermore, any off-line algorithm $A$ for $T$ gives an off-line algorithm $A'$ for $\sigma$ which costs no more. Algorithm $A'$ stays in the same state during all of the requests of $\tau(i)$, and this is the state used by $A$ to process $T(i)$. From this, we can conclude that $C_{\text{OPT}}(\sigma) \leq C_{\text{OPT}}(T)$. Combining these inequalities with the fact that BAL is $(n-1)$-competitive proves the theorem. □

## 8. OPEN PROBLEMS

The most obvious open problem is to devise a $k$-competitive algorithm for the symmetric $k$-server problem. We conjecture that such an algorithm exists, but have been unable even to extend our solution to the 2-server problem to three or more servers. A computer search has revealed that there are 3-competitive algorithms for certain instances of the 3-server problem. Further evidence for this conjecture was supplied by Chrobak, Karloff, Payne, and Vishwanathan,[2] who have recently devised a simple $k$-competi-

[2] Personal communication.

tive algorithm for $k$ servers on a line (a graph in which the distances are consistent with the Euclidean distances between points on a line).

We do know that there is an algorithm for the symmetric $k$-server problem in which the competitive factor depends only on $k$ and $n$. Because the $k$-server problem is a forced task system with $\binom{n}{k}$ states, there is an $\left(\binom{n}{k} - 1\right)$-competitive algorithm. Although this result is very weak, it is the best bound we know which is independent of the distances.

An even more difficult problem is to find an algorithm for the $k$-server problem that matches the lower bound of Theorem 6 when compared to an optimal $h$-server algorithm. It would be particularly interesting if there was a single algorithm (independent of $h$) that achieved this bound for every $h$. The LRU algorithm for the uniform server problem (where all move costs equal one) has this property [11].

Nothing is known about server problems with excursions beyond the results described in Section 6. We conjecture that there is a 3-competitive algorithm for any one-server problem with excursions when the cost to move from $i$ to $j$ is a constant times the excursion cost. In [8], a procedure is described that will compute (in principle) the minimum competitive factor for any instance of a server problem. We have used this procedure to verify this conjecture in several small special cases.

The definition of competitiveness carries over in a natural way to randomized on-line algorithms [7]. In this case we want the expected cost of the randomized on-line algorithm (taken over all possible outcomes of its coin flips) to be within a constant factor of the optimum off-line algorithm on any sequence of requests. In [5], a randomized algorithm for the uniform $k$-server problem (the paging problem) is given. This algorithm is roughly $2\ln(k)$. In [9], the competitive factor is reduced to $\ln(k)$, which is optimal. A natural problem is to consider the non-uniform case.

In [10], Raghavan and Snir describe an even more restricted class of randomized on-line algorithms for server problems. These algorithms are memoryless in the sense that the only state information they maintain from one request to the next is the current arrangement of the servers. They show that their *harmonic algorithm* is $k$-competitive for the uniform problem, as well as the caching problem (all moves into a vertex have the same cost). They leave as an open problem that of finding a memoryless algorithm that is $k$-competitive for any $k$-server problem.

We know very little about asymmetric server problems. For a particular asymmetric $k$-server problem, let $\Delta$ be the maximum over all cycles in the graph of the ratio of the cost of moving around the cycle in one direction to that in the other direction. Any $c$-competitive algorithm for a symmetric server problem can be used to give a $c\Delta$-competitive algorithm for the asymmetric problem. This is done by letting $d'_{ij} = \frac{1}{2}(d_{ij} + d_{ji})$, and run-

ning the $c$-competitive algorithm on the new problem. Similarly, a lower bound of $c$ on the competitive factor in the symmetric problem gives a lower bound of $c/\Delta$ in the asymmetric problem.

## REFERENCES


1. D. BLACK AND D. D. SLEATOR, Algorithms for the 1-server problem with excursions, in preparation.
2. A. BORODIN, N. LINIAL, AND M. SAKS, An optimal online algorithm for metrical task systems, *in* "Proceedings, 19th Annual ACM Symposium on Theory of Computing, New York, 1987," pp. 373–382.
3. A. R. CALDERBANK, E. G. COFFMAN, JR., AND L. FLATTO, Sequencing problems in two-server systems, *Math. Oper. Res.* **10**, No. 4 (1985), 585–598.
4. A. R. CALDERBANK, E. G. COFFMAN, JR., AND L. FLATTO, Sequencing two servers on a sphere, *Commun. Statist.-Stochastic Models* **1**, No. 1 (1985), 17–28.
5. A. FIAT, R. M. KARP, M. LUBY, L. A. MCGEOCH, D. D. SLEATOR, AND N. E. YOUNG, "Competitive Paging Algorithms," Carnegie Mellon University Computer Science technical report CMU-CS-88-196, 1988.
6. A. R. KARLIN, M. S. MANASSE, L. RUDOLPH, AND D. D. SLEATOR, Competitive snoopy caching, *Algorithmica* **3**, No. 1 (1988), 79–119.
7. M. S. MANASSE, L. A. MCGEOCH, AND D. D. SLEATOR, Competitive algorithms for on-line problems, *in* "Proceedings, 20th Annual ACM Symposium on Theory of Computing, Chicago, 1988," pp. 322–333.
8. L. A. MCGEOCH, D. D. SLEATOR, AND C. TOMASI, Decision procedures for competitive algorithms, in preparation.
9. L. A. MCGEOCH AND D. D. SLEATOR, "A Strongly Competitive Randomized Paging Algorithm," Carnegie Mellon University Computer Science technical report CMU-CS-89-122, 1989; *Algorithmica*, in press.
10. P. RAGHAVAN AND M. SNIR, Memory versus randomization in on-line algorithms, *in* "*Automata, Languages, and Programming*, Lecture Notes in Computer Science," Vol. 372, pp. 687–703, Springer-Verlag, New York, 1988.
11. D. D. SLEATOR AND R. E. TARJAN, Amortized efficiency of list update and paging rules, *Comm. ACM* **28**, No. 2 (1985), 202–208.