Maximizing Throughput in Minimum Rounds in an Application-Level Relay Service[*]

Fred Annexstein Kenneth A. Berman Svetlana Strunjaš Chad Yoshikawa

Abstract

Application-level network relays possess many desirable properties, including support for communication between disconnected clients, increasing bandwidth between distant clients, and enabling routing around Internet failures. One problem not considered by existing systems is how to assign client load to relay servers in order to maximize throughput of the relay-system. In this paper, we are interested in the particular case where network conditions change frequently so that the ability of clients to adapt flow is restricted and each round of activity is critical. To this end, we present an algorithm, called Aggressive Increase, $\mathcal{A}_{AI}$ which improves its competitive ratio in each time round that the network conditions persists. Given a relay network where a client connects to at most N servers, if network conditions persist for $log(N)$ rounds then the algorithm's throughput becomes constant competitive. Our results improve upon the competitive ratio of previous work (of Awerbuch, Hajiaghayi, Kleinberg, and Leighton [2]). In addition we show that the $\mathcal{A}_{AI}$ algorithm performs well in simulation studies as compared with the algorithm of [2] and an adaptation of the multiplicative increase algorithm of [8]. On a variety of input graphs, we show that the $\mathcal{A}_{AI}$ algorithm typically reaches close to peak bandwidth levels within only a small constant (< 10) number of rounds.

1 Introduction

We consider the problem of maximizing throughput in a network relay service where each sender must tunnel data to its receiver via a set of permissible servers. For example, the set of permissible servers for a sending client could be its k-closest servers where 'closest' is determined by network latency measurements. When a sender communicates it must fractionally assign each successive block of data in its data stream to its set of permissible servers. These servers then forward the data on to the receiver. This problem has practical implications in today's Internet since there are many devices, e.g. computers behind firewalls or network-address-translation (NAT) boxes, which cannot communicate with one another directly without the use of public third parties or waypoints [7]. In addition, it has been shown that application-level relays can even benefit connected clients by increasing bandwidth [11] and routing around Internet failures [1].

In this paper we focus on the the problem of deriving a method of optimizing throughput in the case of limited periods of activity. The goal is to determine how each sending client can aggressively assign data to its set of permissible servers (in each round) in order to maximize total system throughput. This problem is related to purely oblivious (one-shot) routing; however, we consider the case where sending clients can adjust their flow based on local load information from their waypoint servers in the previous round. While we assume that network streams have some measure of persistence, we do not assume *a priori* the duration of each stream. Therefore, any solution to this problem should attempt to maximize throughput during each additional round of persistence but cannot make any assumption about the length of persistence of client demands. Furthermore, given the nature of the streams, e.g., video or voice-over-IP, extensive buffering of the data is not considered.

In this paper we present a simple algorithm, called Aggressive Increase, $\mathcal{A}_{AI}$ which adjusts and improves its throughput in each time round that the set of sending clients persists. In each round the *competitive ratio*, that is the ratio with regard to the optimal throughput, is improved. We show that if the set of senders persist for a *run* of $\rho \in \Omega(\log(\Delta))$ consecutive time rounds then the algorithm becomes constant competitive with the optimal, where Δ is the maximum out degree of the set of active senders. Our results compare favorably to previous works. We show that the $\mathcal{A}_{AI}$ algorithm has slightly improved competitive ratio, compared to previously known results, for all runs of length at least $2\lceil\log\Delta\rceil + 4$ rounds. In addition we show that the $\mathcal{A}_{AI}$ algorithm tends to perform better in simulation studies as compared with the previously studied algorithms in [2, 8].

[*] Department of Computer Science, University of Cincinnati, Cincinnati, OH, 45221-0030. Supported in part by NSF Grant 0521189 and an Ohio Board of Recents Ph.D. Fellowship

2 Comparisons with Related Work

Previous work on algorithms maximizing throughput in few rounds have tended to have one or more of the following problems: (1) slow convergence to the final solution after polylog (or more) rounds, (2) failure to reach constant-competitiveness, or (3) dependence on a priori knowledge of the length of persistence. Of course this throughput problem can be modeled as a flow problem, and several researchers have considered distributed primal-dual approaches to approximating flow solutions; see for example, [12, 4, 8] which yield $(1+\epsilon)$ approximations within polylog distributed rounds. However, as noted in [2], these algorithms converge to a final solution which is to say that the performance of the algorithms before convergence is not necessarily known. Furthermore, the convergence rates of these algorithms typically have large constants (on the order of $\frac{1}{\epsilon^3}$). If some clients stop sending and/or new clients start sending before convergence is reached then it is not known how much throughput has been achieved.

Chattopadhyay, Higham, and Seyffarth [5] have described a distributed maximum matching algorithm which operates by repeatedly finding and removing augmenting paths. However, this algorithm converges to an optimal solution in $O(Nn)$ rounds where n is the actual number of processors in the system and N is an upperbound on n. The algorithm's performance before convergence is not known.

Oblivious routing schemes are also well-suited to distributed load balancing since local knowledge alone is used to route network traffic. A polynomial time algorithm for constructing a polylog-competitive (w.r.t edge congestion) oblivious routing scheme is given by Azar, Cohen, Fiat, Kaplan and Räcke in [3]. However, this algorithm is designed for one-shot oblivious routing and does not make use of demand persistence to improve its performance. A recent semi-oblivious routing algorithm of Awerbuch, Hajiaghayi, Kleinberg, and Leighton [2] is closely related to our work. Their algorithm is shown to be constant competitive as long as one can guarantee *a priori* that clients' demands always persist for $R \in \Omega(\log(\Delta))$ rounds where Δ is an upperbound on the maximum client out degree. Throughout the rest of the paper, we refer to this algorithm as the $\mathcal{A}_{RA}$ or 'restricted adversary' algorithm. In general, for a given value of R, the algorithm is shown to be $18\frac{\lceil R/2\rceil}{R}(2\Delta)^{6/R}$ competitive. This algorithm operates under a restricted adversary model where the adversary may specify when each client becomes active but, once active, a client must always send for R consecutive rounds where R is a system-wide, well-known constant. In practice, it may be difficult to know how long client demands will persist, i.e., a particular network stream could begin or terminate at any time. If a too small value of R is used then the throughput may suffer since the competitive ratio of the algorithm is inversely related to R. (See Section 6 which discusses the $\mathcal{A}_{RA}$ algorithm dependence on R.)

In this paper we show that if the set of network streams and Δ have persisted for the last $i > 1$ rounds then the algorithm has been $4/(1-g)$-competitive during that i-round period where $g = \frac{l}{l+\frac{1-l}{\Delta}}$, and $l = \frac{1}{2^{\lfloor i/2\rfloor}}$. Note the use of the past tense, meaning that the algorithm's competitiveness depends on how persistent demands *have been* versus how persistent the demands *will be* and thus it does not require a priori knowledge of client demands or client demand distribution. We call an i-round period where the state of the system has been persistent a 'run' of length i. Specifically, for any run of length $i = 2*(\lceil\log(\Delta)\rceil + \lceil\log(1/\epsilon)\rceil)$ rounds, we show that the $\mathcal{A}_{AI}$ algorithm is $4/(1-\epsilon)$ competitive with the optimal algorithm over the length of the run for any desired ϵ, $0 < \epsilon < 1$. For example, for any run of length $i = 2(\lceil\log(\Delta)\rceil + 1)$ rounds the $\mathcal{A}_{AI}$ algorithm is 8-competitive. Furthermore, we show that given the same restricted adversary model of the $\mathcal{A}_{RA}$ algorithm, the $\mathcal{A}_{AI}$ algorithm's competitive ratio is always smaller than $\mathcal{A}_{RA}$ given that $R \geq 4(\lceil\log(\Delta)\rceil+2)$ where Δ is the max client out degree. Our main result is the following:

Theorem 2.1. *Over $i > 1$ rounds of persistent demands, the $\mathcal{A}_{AI}$ algorithm throughput is $4/(1-g(\lfloor i/2\rfloor+1))$-competitive with the optimal algorithm where $g(i) = \frac{l(i)\Delta}{l(i)\Delta+(1-l(i))}$ and $l(i) = \frac{1}{2^{i-1}}$. In particular, for a run of length $i = 2*(\lceil\log(\Delta)\rceil + \lceil\log(1/\epsilon)\rceil)$ rounds, the $\mathcal{A}_{AI}$ algorithm is $4/(1-\epsilon)$ competitive with the optimal algorithm for any desired ϵ, $0 < \epsilon < 1$.*

Finally, we present a series of simulation studies which suggest that the $\mathcal{A}_{AI}$ algorithm will perform quite well in practice. We compare throughput results of the $\mathcal{A}_{AI}$ with the algorithm of [2] and the $\mathcal{A}_{MI}$ $(1+\epsilon)$-competitive multiplicative increase algorithm of Garg and Young [8].

The rest of the paper is organized as follows. The graph model is given in Section 3, the $\mathcal{A}_{AI}$ algorithm is described in Section 4, performance and convergence of the algorithm is contained in Section 5, and experimental results are given in Section 6.

3 Model

Clients and servers operate synchronously in a series of time rounds, where in each round an active client can fractionally assign at most one block to its set of permissible servers and each server can process any fractional number of blocks up to its specified capacity. The sum of the blocks processed by all servers per round is called 'bandwidth' and the sum of the processed blocks over any run is called the run's 'throughput'. We assume that each sender and receiver is a part of at most one simultaneous communication and that the receiver's bandwidth is not the bottleneck. Thus, we can model the problem as fractional assignment in a bipartite graph $\{U, V, E\}$ where each sender-receiver pair is a node in U, each waypoint is a node in V, and each edge $(u, v) \in E$ indicates that the sender-receiver pair u can communicate via waypoint v. From this point forward, we will refer to a client-pair as a single client which indicates the sender of the pair. During each round, clients decide how much data to send to each server based on a server's feedback from the previous round. If a client did not send in the previous round, then the client can choose how much to send arbitrarily. (The first round flow does not impact the behavior or analysis of the algorithm.) For ease of exposition, we declare the client's first round flow to be zero when describing the algorithm in Section 4. We make use of the following definitions and notation in subsequent sections:

1. The fractional assignment, or flow, on the edge e is denoted $f(e)$.
2. The load on a server v, L_v, is the sum of the fractional assignments on its incoming edges, $L_v = \sum_{e=(u,v)} f(e)$. The offered load of a client u, Ω_u, is the sum of the fractional assignments on its outgoing edges, $\Omega_u = \sum_{e=(u,v)} f(e)$.
3. A server v is saturated if $\frac{L_v}{C_v} \geq 1$ and unsaturated otherwise, where C_v is the capacity of server v.
4. A client u is saturated if $\Omega_u = 1$, unsaturated otherwise.
5. The set of a server v's clients is denoted $Clients_v$. This set is partitioned into saturated clients and unsaturated clients, $Clients_v^{sat}$ and $Clients_v^{unsat}$, respectively.
6. The remaining capacity of a server v, Ψ_v, is defined to be the capacity of a server minus its load, i.e., $\Psi_v = C_v - L_v$.
7. Client demands for round t are defined by the bit-vector D_t of size N. If $D_t(u) = 1$, this means that client u is sending (active) during round t, not sending (inactive) otherwise.

4 Algorithm

The $\mathcal{A}_{AI}$ algorithm closely models max-min fair algorithms used to compute flow rates for available-bit rate traffic in ATM networks (see, for example, [13, 10]). These algorithms compute max-min fair rate allocations in $O(N)$ distributed rounds where N is the number of unique rates. We show that, by computing the max-min fair rate allocation at each server, the algorithm reaches constant-competitive throughput in a logarithmic number of rounds. (Note that $\mathcal{A}_{AI}$ algorithm, like the other algorithms benchmarked in this paper, are not TCP friendly. However, the $\mathcal{A}_{AI}$ algorithm is intended for use in an environment where the relay nodes are the bottleneck and would typically be implemented on top of TCP or a TCP-friendly algorithm.)

4.1 Client Algorithm Initially, upon becoming active after a period of inactivity, each client's flow assignment is set to zero for each permissible server. The client-side algorithm is described in terms of one client; the algorithm is identical on each client. In the beginning of each round, the client sends flow to each of its permissible servers based on its current flow assignment. (This means that the client will send zero units of flow in the first round it becomes active.) At the conclusion of the round, the client receives a message from its permissible servers indicating a request for flow increase. The client handles these flow increase requests in any particular order and increases its flow assignments accordingly. The client stops handling requests if it runs out of its one unit of flow to assign. If $\Omega = 1$ (the client has assigned 1 unit of its flow), the client declares itself 'saturated' to all of its permissible servers and will maintain its current flow assignment to each outgoing edge while it remains active. Otherwise, the client declares itself 'unsaturated' to all of its servers, meaning that it can increase its flow to some server(s) in the next round. Notice that for every active client u, $f(e)$ is monotonically increasing for all edges out of u, and so Ω is also monotonically increasing.

4.2 Server Algorithm The server-side algorithm is described in terms of a single server; the algorithm is the same on each server. Each server needs to keep a count of its unsaturated clients and its current load. During each round, the server processes flow from its clients (the server is never overloaded) and updates its count of unsaturated clients and its current load value. At the end of each round, the server v asks each unsaturated client to increase its flow value by the following amount:

$$f_{inc}(v) = \frac{\Psi_v}{|Clients_v^{unsat}|}$$

Intuitively, the server is asking the unsaturated clients to fill it to capacity in the very next round by having each unsaturated client increase its flow by an equal

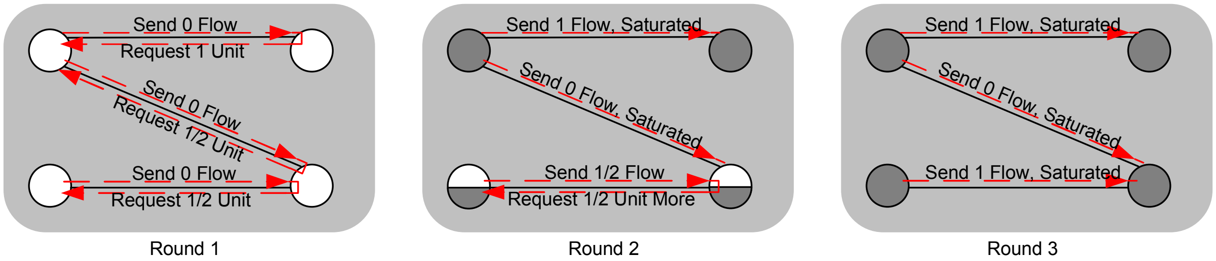


Figure 1: An example run of the $\mathcal{A}_{AI}$ algorithm for a simple bipartite graph over three rounds. At the start of the second round, the top-most client has chosen to satisfy the request for one unit of flow increase instead of splitting its flow among its two servers.

amount. (The saturated clients maintain flow by definition, so the message is not sent to them.)

An instance of the algorithm is shown in Figure 1 for a simple graph and three rounds.

5 Analysis

We prove two general lemmas regarding flow induced generalized vertex covers of bipartite graphs.

Suppose we are given a bipartite graph G of clients U and servers V, where each client can send at most one unit of flow and each server $v \in V$ has capacity C_v. For a given feasible flow assignment f, let $X_f \subseteq U$ be a set of saturated clients and let $Y_f \subseteq V$ be a set of saturated servers induced by f. We say the the pair (X_f, Y_f) is a *saturated cover* if all the edges of G have at least one endpoint in $X \cup Y$.

Lemma 5.1. *If (X_f, Y_f) is a saturated cover then the value of the flow f is 2-competitive with the optimal flow.*

Proof. Since (X_f, Y_f) covers every edge in G we have that the value of any feasible flow is no larger than $|X| + \sum_{\forall v \in Y} C_v$. The Lemma follow, since the the flow f is at least $\max(|X|, \sum_{\forall v \in Y} C_v)$.

We extend the notion of a saturated cover as follows. For a given feasible flow assignment f, pair (X_f, Y_f) is an *(r, s)-cover* (where $0 < r, s \le 1$) if (i) $X_f \cup Y_f$ is a vertex cover (covering all edges of G), (ii) X_f is a set of clients for which at least the fraction r are saturated by f, and (iii) Y_f is a set of servers so that each $v \in Y$ is loaded to at least an s fraction of its capacity. Note that a saturated cover is equivalent to a (1,1)-cover. We have the following lemma.

Lemma 5.2. *If f is a feasible flow for which (X_f, Y_f) is a (r, s)-cover, then the value of the flow f is $2/\min\{r, s\}$-competitive with the optimal flow.*

Proof. Again, the optimal flow is upper-bounded by the expression $|X| + \sum_{\forall v \in Y} C_v$. The flow f associated of the (r, s)-cover is at least $\max(r|X|, s\sum_{\forall v \in Y} C_v) \ge \min\{r, s\} \max\{|X|, \sum_{\forall v \in Y} C_v\}$. The Lemma follows.

See Figure 2 which illustrates a saturated cover and (r, s)-cover for $r = 2/3$ and $s = 1/2$.

These two lemmas will be used in the following analysis of the $\mathcal{A}_{AI}$ algorithm. Specifically, we will show that if Δ is the maximum degree of any client, then after $\Omega(\log(\Delta))$ rounds the $\mathcal{A}_{AI}$ algorithm produces a flow f that has an associated (r, s)-cover (X_f, Y_f) where r and s are constants bounded away from 0, and thus f is constant competitive with the optimal flow.

Now, we formalize the definition of a run:

Definition 5.1. *A run is a sequence of consecutive time rounds in which the system state has persisted, where system state consists of: the set of active clients U, set of servers V, maximum degree Δ, server capacities C, and the bipartite-connectivity E.*

In this section, for a particular run of length $\rho > 1$, we will determine the competitive ratio of the $\mathcal{A}_{AI}$ algorithm's throughput w.r.t the optimal algorithm's throughput over the same run. The rounds of a run are numbered $1, \ldots, \rho$ where ρ is the last round before the state of the system changes. We are given a bipartite graph $\{U, V, E\}$ of the clients, servers, and edges respectively. Initially, we will focus our analysis on a single server (right-hand vertex) v and its client neighborhood, $Clients = \{u \in U | (u, v) \in E\}$. Then, we will extend our result to include the entire bipartite graph.

5.1 One Server

First, we define a 'local-approximation' fraction $l(i)$ for round $i > 1$ as:

$$l(i) \le \frac{1}{2^{i-1}} \tag{5.1}$$

Now we give the following lemma:

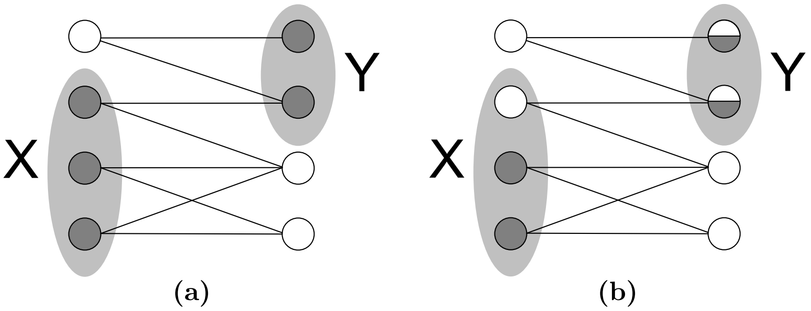


Figure 2: (a) An example saturated cover and (b) (2/3,1/2)-cover.

LEMMA 5.3. *After i rounds, either a server v will be loaded to at least $1 - l(i)$ fraction of its capacity or at least $1 - l(i)$ fraction of v's clients will be saturated.*

Proof. Let $P_v(i)$ denote the fraction of v's clients that are unsaturated at the end of round i, i.e., $P_v(i) = \frac{|Clients_v^{unsat}|}{|Clients_v|}$. Let $P_{\Psi_v}(i)$ denote v's fraction of remaining capacity at the end of round i, $1 - \frac{L_v}{C_v}$.

We define a potential function $\Phi_v(i)$ equal to the product of these values, $\Phi_v(i) = P_v(i) * P_{\Psi_v}(i)$. We have that $\Phi_v(1) \leq 1$, since $P_v(1) \leq 1$ and $P_{\Psi_v}(1) \leq 1$. We indicate less than or equal to, rather than equal to, since at the first round there may be active clients which were also active during the last run. By definition of the algorithm, the flow of these clients in the first round of the current run would be greater than or equal to their flow in the last round of the last run. In other words, not all clients start out with zero flow in the first round of run. Only those clients who were inactive in the previous run will start out with zero flow in the first round of the current run. Now, we will show that Φ_v decreases by at least a factor of 4 after every subsequent round, i.e., $\Phi_v(i+1) \leq \frac{\Phi_v(i)}{4}, \forall i \in 1, \ldots, \rho - 1$. First, we note that P_{Ψ_v} and P_v are monotonically decreasing functions. P_{Ψ_v} can never increase since each client maintains or increases flow to its servers and clients cannot be removed during a run. Neither can P_v increase, since the algorithm states that once a client declares itself saturated it remains saturated and clients cannot be added nor removed during a run. There are three cases to consider, based on the change to P_v from round i to round $i+1$, for $\rho > i \geq 1$. The first two cases are special cases of the third, but we include them for clarity.

1. P_v does not change. This indicates that all unsaturated clients from the previous round remained unsaturated, which implies that the server v's entire flow request is satisfied in the current round and thus P_{Ψ_v}, and consequently $\Phi_v(i+1)$, go to zero.
2. P_v becomes zero. This means that all clients have become saturated in the current round and are sending their entire unit of flow. In this case, clearly $\Phi_v(i+1)$ becomes zero by definition.
3. P_v decreases by the fraction ϵ, $0 < \epsilon < 1$, i.e., $P_v(i+1) = (1-\epsilon) * P_v(i)$. In this case, P_{Ψ_v} decreases by $(1-\epsilon)$, i.e., $P_{\Psi_v}(i+1) = (\epsilon) * P_{\Psi_v}(i)$. To see this, consider that in the $\mathcal{A}_{AI}$ algorithm the server asks each of its unsaturated clients for a flow increase equal to $\Psi_v/|Clients_v^{unsat}|$. The clients that remain unsaturated increase their flow to match the server's request. So, if P_v goes down by ϵ fraction then there will be $1-\epsilon$ fraction of the unsaturated clients from the previous round remaining unsaturated. This $1-\epsilon$ fraction of the clients will send additional flow equal to $\Psi_v * (1-\epsilon)$. Thus, P_{Ψ_v} goes down by at least $(1-\epsilon)$ fraction when P_v decreases by ϵ fraction. We say 'at least', because it's possible that the newly saturated clients send some nonzero flow less than what the server asked for. Thus, $\Phi_v(i+1) \leq \Phi_v(i)\epsilon(1-\epsilon)$. This factor, $\epsilon * (\epsilon - 1)$, is maximized when $\epsilon = 1/2$ so Φ_v goes down by at least a factor of 4 at every round.

Recall that $\Phi_v(1) \leq 1$. With every additional round, Φ_v reduces by at least a factor of 4 so $\Phi_v(i+1) \leq \frac{1}{4^i}$ which implies that at round i one of either P_{Ψ_v} or P_v is smaller than or equal to $l(i)$ given by Equation 5.1. So either the server will be loaded to at least $1 - l(i)$ fraction of its capacity or at least $1 - l(i)$ fraction of its clients will be saturated. The proof of the Lemma follows.

5.2 All Servers The problem we now have to solve is that Lemma 5.3 does not hold globally for multiple servers. For example, given a situation where two out of every four clients connected to any server are saturated (so that $P_v = 0.5$), there may be only two saturated clients in the entire bipartite graph, i.e., we could be counting the same saturated clients multiple times. Fortunately, we can use Δ as a bound on how many times we can multiply-count the same saturated client. Since each client has maximum degree of Δ, this means that the same saturated client can be multiply counted at most Δ times. We will use this Δ bound to show how many additional rounds are needed to achieve the conditions in Lemma 5.3 globally. Note that we use knowledge of Δ in the analysis of the algorithm's competitive ratio, but Δ is not used by the algorithm itself. After round $i > 1$, we will call a server with a load-to-capacity ratio less than $(1 - l(i))$ 'underloaded', otherwise we call it a 'loaded' server. At the conclusion of round i, let X_i be the set of all clients connected to underloaded servers and Y_i be the set of all loaded servers. Clearly $X_i \cup Y_i$ forms a vertex cover for the bipartite graph.

Using Equation 5.1, we first define a 'global' fraction $g(i)$ for round $i > 1$ which will be used in the Lemma below.

$$g(i) \leq \frac{l(i)}{l(i) + \frac{1-l(i)}{\Delta}} \tag{5.2}$$

Lemma 5.4. *Let X_i be the set of all clients connected to underloaded servers after round i. After $i > 1$ rounds, at least $1 - g(i)$ fraction of the clients in X_i will be saturated.*

Proof. Given an unsaturated client connected to an underloaded server, let us call both the unsaturated client and the edge connecting it to the underloaded server 'uncovered'. At round $i > 1$, we know from Lemma 5.3 that $l(i)$ is the maximum fraction of uncovered clients incident to any single underloaded server relative to the server's in-degree. Since there is a one-to-one correspondence between in-edges and clients, i.e., we do not consider multiedges in the graph, this means that there is no more than $l(i)$ fraction of *edges* uncovered relative to any single underloaded server.

Let E'_i denote the set of all edges incident to the set of underloaded servers at round i. Clearly no more than $l(i)$ fraction of these edges are uncovered since $l(i)$ is the maximum fraction of uncovered edges for any single server. We now can find an upper bound $g(i)$ on the fraction of *clients* in X_i which are unsaturated, using the value $l(i)$ and the maximum client out-degree Δ.

At round i there is at most $l(i)E'_i$ uncovered edges and at least $(1 - l(i))E'_i$ covered edges. These $l(i)E'_i$ uncovered edges could be originating from at most $l(i)E'_i$ clients, since there is at least one edge per client. However, the $(1 - l(i))E'_i$ covered edges could be, in the worst case, originating from only $(1 - l(i))E'_i/\Delta$ clients (since each saturated client may have up to Δ edges). Thus, the global fraction of uncovered clients is bounded by $g(i)$ given by the following:

$$g(i) \leq \frac{l(i)}{l(i) + \frac{1-l(i)}{\Delta}}$$

The proof of Lemma 5.4 follows.

Now we present the main theorem of the paper:

Theorem 5.1. *After $i > 1$ rounds of any run, the $\mathcal{A}_{AI}$ algorithm produces a flow f which is $4/(1-g(\lfloor i/2 \rfloor+1))$-competitive with the optimal algorithm.*

Proof. Let Y_i denote the set of loaded servers, and let X_i denote the set of clients connected to underloaded servers after round $i > 1$. Combining Lemmas 5.3 and Lemma 5.4, we have that Y_i are each $(1-l(i))$ loaded and $(1 - g(i))$ fraction of the clients X_i are saturated. Thus, at round $i > 1$ we have arrived at a $((1-g(i)), (1-l(i)))$-saturated cover as defined in Lemma 5.2. Given that $g(i) \geq l(i)$ for any $\Delta \geq 1$, then by Lemma 5.2 the $\mathcal{A}_{AI}$ algorithm's bandwidth is $2/(1 - g(i))$-competitive with the optimal algorithm.

Now we will use this bandwidth bound to prove the competitiveness of the throughput. Recall that bandwidth is measured in units per round while throughput is measured in total units over a time period. Since the bandwidth is monotonically increasing in the $\mathcal{A}_{AI}$ algorithm, we know that after i rounds we have been sending at a rate of at least the sending rate at round $\lfloor i/2 \rfloor + 1$, and this rate has persisted for $\lceil i/2 \rceil$ rounds. Thus, after i rounds, the $\mathcal{A}_{AI}$ algorithm's throughput is $4/(1 - g(\lfloor i/2 \rfloor + 1))$-competitive with the optimal algorithm.

Lemma 5.5. *After $i = 2 * (\lceil \log(\Delta) \rceil + \lceil \log(1/\epsilon) \rceil)$ rounds, the $\mathcal{A}_{AI}$ algorithm is $4/(1 - \epsilon)$ competitive with the optimal algorithm for any ϵ, $0 < \epsilon < 1$.*

Proof. This lemma follows immediately by setting i and using Theorem 5.1.

Beyond the competitive ratio bound above, we also note also that the $\mathcal{A}_{AI}$ algorithm reaches a steady state that is 2-competitive with the optimal flow, after at most Δ_{in} steps, where Δ_{in} is the maximum in-degree of a server. This can be seen from the fact that in each round a given server v becomes (or is) fully loaded, or otherwise at least one client of v must become saturated. Since, if no client becomes saturated then the server must have received its entire (flow increase) request and therefore becomes saturated. In the following result, we have a more precise calculation of the cumulative throughput achieved by the $\mathcal{A}_{AI}$ algorithm before this steady state is achieved.

LEMMA 5.6. *For $I < \Delta_{in}$, an I-round run, the $\mathcal{A}_{AI}$ algorithm has throughput at least:*

$$\frac{Opt}{2}\left(I - \frac{\Delta}{(\Delta-1)}\frac{1}{ln(2)}ln\left(\frac{2^I*(\Delta-1/2)}{2^{I-1}+\Delta-1}\right)\right) \, if \, \Delta > 1,$$

$$\frac{Opt}{2}\left(I - \frac{2}{ln(2)}\left(1-\frac{1}{2^I}\right)\right) \, if \, \Delta = 1.$$

where Opt is the optimal throughput over the same I-round run.

Proof. First, note that the optimal throughput after I rounds is $I * Opt$ where Opt is the optimal throughput for one round. We will now calculate the throughput of the $\mathcal{A}_{AI}$ algorithm after I rounds. Since we can lower-bound the bandwidth of the $\mathcal{A}_{AI}$ algorithm for every round i, and since clients' demands do not change over the course of a run, we can simply sum up the bandwidths for each round. Recall that during round i the AI algorithm's bandwidth is $2/(1-g(i))$-competitive with the optimal algorithm where $g(i) = \frac{l(i)}{l(i)+\frac{1-l(i)}{\Delta}}$ and $l(i) = \frac{1}{2^{(i-1)}}$. Summing over all i in a run of I rounds, we have that the bandwidth of the $\mathcal{A}_{AI}$ algorithm is at least:

$$\sum_{i=1}^{I}(1-g(i))/2 * Opt = \frac{Opt}{2} * \sum_{i=1}^{I}(1-g(i))$$

Since the bandwidth function is increasing, it is well-known that we can bound this summation with an integral over the bounds $[0, I]$:

$$\frac{Opt}{2} * \sum_{i=1}^{I}(1-g(i)) \geq \frac{Opt}{2} * \int_{0}^{I}(1-g(i))di$$

Evaluating this integral we get that after I rounds of a run the throughput of the $\mathcal{A}_{AI}$ algorithm is at least:

$$\frac{Opt}{2}\left(I - \frac{\Delta}{(\Delta-1)}\frac{1}{ln(2)}ln\left(\frac{2^I*(\Delta-1/2)}{2^{I-1}+\Delta-1}\right)\right) \text{ if } \Delta > 1,$$

$$\frac{Opt}{2}\left(I - \frac{2}{ln(2)}\left(1-\frac{1}{2^I}\right)\right) \text{ if } \Delta = 1.$$

5.3 Competitive Ratio Comparisons We conclude this section with a comparison of the competitive ratios of the $\mathcal{A}_{AI}$ algorithm with the $\mathcal{A}_{RA}$ algorithm. To do this we will use the same restricted adversary model described in [2]. We briefly describe the model here. In this model, an adversary can specify when each client starts sending, but once a client starts sending it must send for R consecutive rounds. Time is divided into a series of equal sized windows w defined as $w = \lceil \frac{R}{2} \rceil$. Clients are called eligible if they are scheduled to send during every round of a window, ineligible otherwise. Instead of operating on the original demands D, the $\mathcal{A}_{RA}$ algorithm operates on a 'modified sequence of demands' D' which is defined as follows: $D'_t(i) = 1$ iff client i is eligible during the window containing round 't'. Lemma 4 of [2] shows that the optimal throughput on the modified demands is no more than a factor of three worse than the optimal throughput on the original sequence of demands. Thus, we will focus on a single window of size w where client demands are guaranteed to persist but we will lose a factor of 3 in the competitive ratios of both algorithms.

THEOREM 5.2. *If $R \geq 4(\lceil \log(\Delta) \rceil + 2)$, the $\mathcal{A}_{AI}$ algorithm is 16-competitive with the optimal algorithm under the restricted adversary model.*

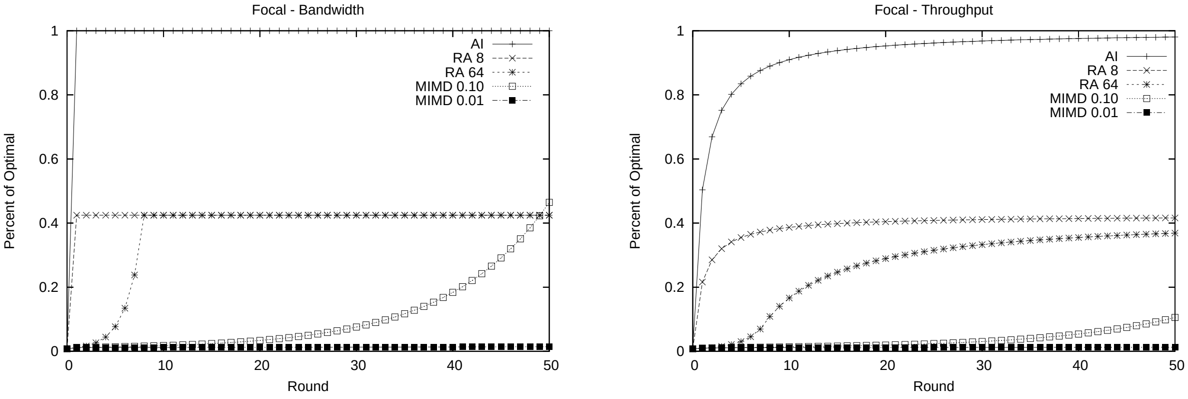
Proof. The proof follows directly by applying Lemma 5.5 with $\epsilon = 1/4$ and $i = R/2$ and using the fact that we lose a factor of three by modifying the original demands.

By analyzing the $\mathcal{A}_{RA}$ algorithm, we have determined that this algorithm has a competitive ratio of at least 18 under the restricted adversary model regardless of the value of R and Δ. So, from Theorem 5.2 and the fact that the $\mathcal{A}_{AI}$ algorithm's bandwidth is monotonically increasing, we see that as long as $R \geq 4(\lceil \log(\Delta) \rceil + 2)$ the competitive ratio of the $\mathcal{A}_{AI}$ algorithm is strictly less than the competitive ratio of the $\mathcal{A}_{RA}$ algorithm.

As compared to the $\mathcal{A}_{MI}$ algorithm, we note that the $\mathcal{A}_{MI}$ algorithm becomes $(1+\epsilon)$-competitive when run per connection, while we are running the algorithm per client and arbitrating between a client's multiple connections by sorting the flows and satisfying them in order. While this performs well in practice, there is no formal proof of the competitiveness of the algorithm when used in this manner. It remains future work to compare these algorithms against $(1+\epsilon)$-competitive algorithms such as that presented in [4].

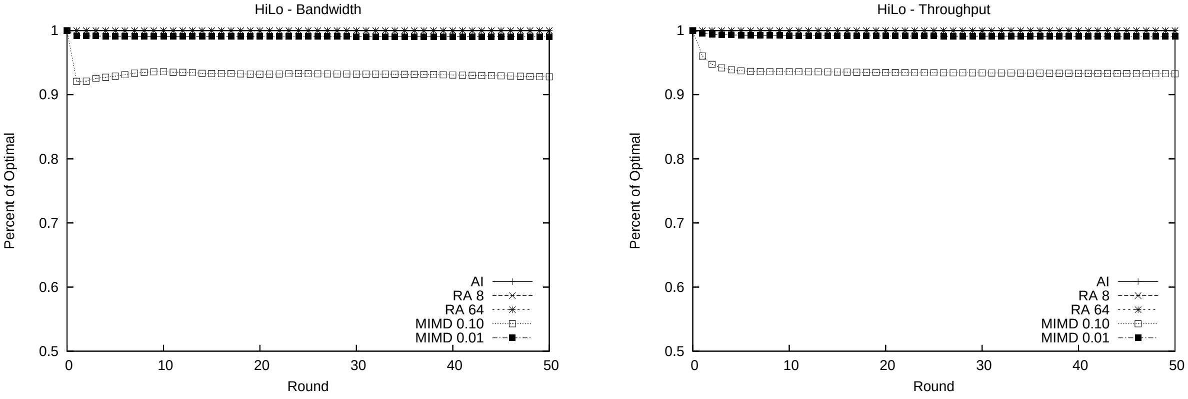
6 Simulation Results

In this section, we report on results of simulations which test maximum throughput algorithms against different bipartite graphs. We compare the $\mathcal{A}_{AI}$ algorithm given in this paper to the optimal algorithm, the $\mathcal{A}_{RA}$ (restricted adversary) algorithm, and to the $\mathcal{A}_{MI}$ $(1+\epsilon)$ multiplicative increase algorithm. Empirical results which are important since comparing competitive ratios



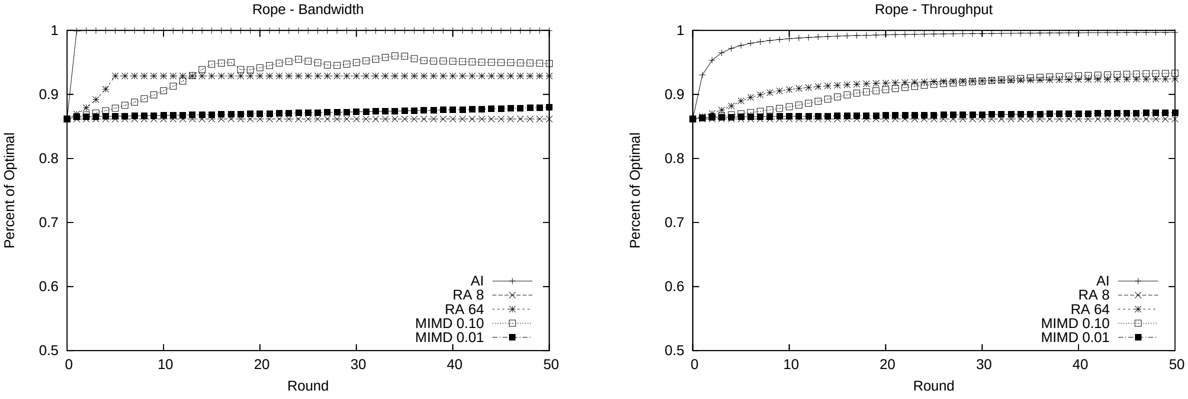
(a) **(b)**

Figure 3: (a) Per-round Bandwidth and (b) Cumulative Throughput on a graph with a single 128-Focal matching.



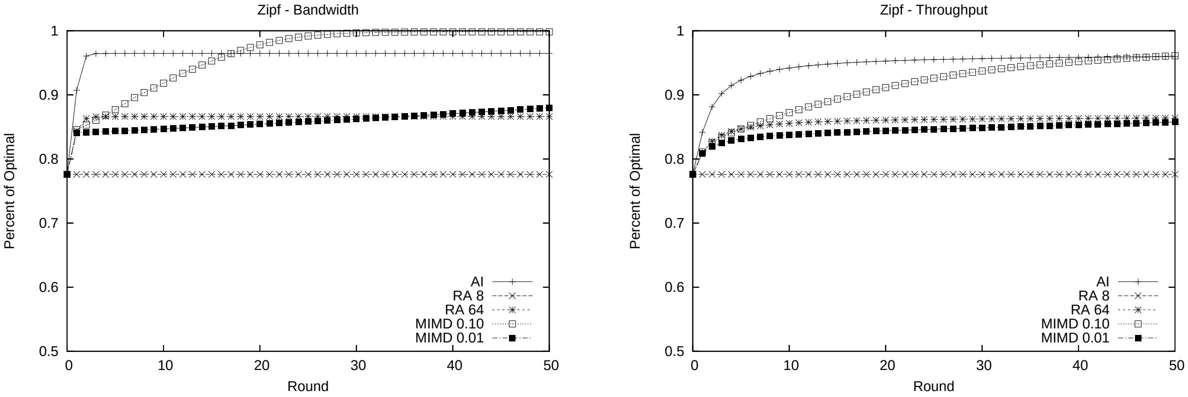
(a) **(b)**

Figure 4: (a) Per-round Bandwidth and (b) Cumulative Throughput on graph HiLo.



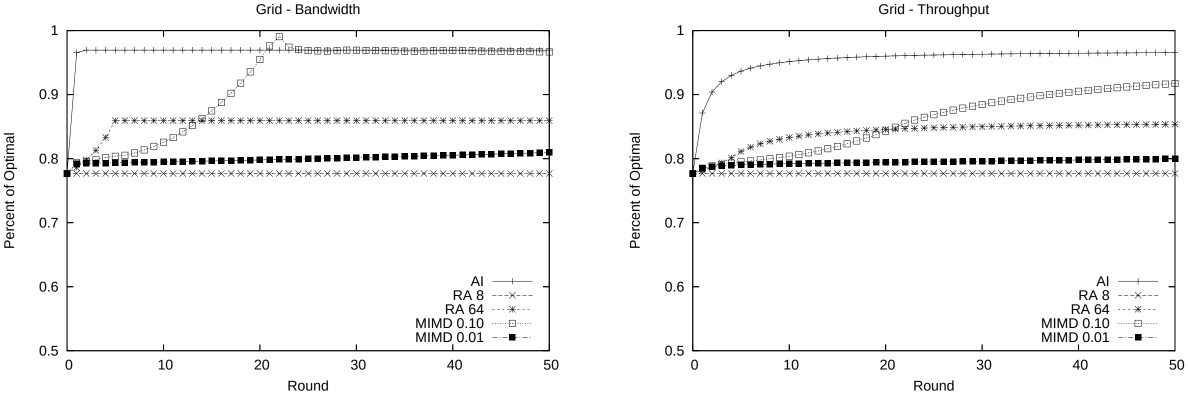
(a) **(b)**

Figure 5: (a) Per-round Bandwidth and (b) Cumulative Throughput on graph Rope.



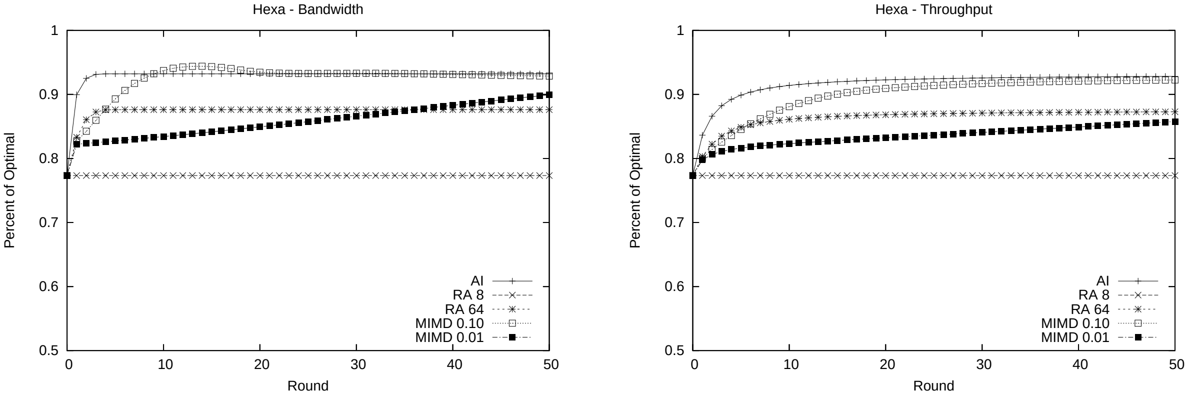
(a) **(b)**

Figure 6: (a) Per-round Bandwidth and (b)Cumulative Throughput on graph Zipf.



(a) **(b)**

Figure 7: (a)Per-round Bandwidth and (b) Cumulative Throughput on graph Grid.



(a) **(b)**

Figure 8: (a) Per-round Bandwidth and (b)Cumulative Throughput on graph Hexa.

of two algorithms may say little about their relative real-world performance. We present results of a series of experiments over a variety of network structures. For each we compute the optimal throughput, through standard transformations of the bipartite graph into a flow network, from which we compute a maximum flow using LEDA [14] library.

In these tests, all clients send for $\rho = 64$ rounds and servers have unit capacity. For our bipartite graph workload, we use a total of six different bipartite graphs. Five of these graphs, HiLo, Hexa, Rope, Zipf, Grid, are described in [6]. The other graph in our workload, Focal, is taken from [2]. All graphs have roughly 2^{16} U vertices and 2^{16} V vertices. The values of Δ for each graph are: Hexa (28), Zipf (24576), Rope(7), HiLo(10), Grid (8), Focal (256).

The algorithms compared are $\mathcal{A}_{AI}$ (Aggressive Increase), $\mathcal{A}_{RA}$ (Restricted Adversary of Awerbuch, et. al [2]), and $\mathcal{A}_{MI}$ a multiplicative-increase multiplicative decrease algorithm adapted from Narg and Young [8]. For the $\mathcal{A}_{RA}$ algorithm, we have used 8 and 64 as values of R. Although the clients are sending for 64 rounds, we have observed that in practice the $\mathcal{A}_{RA}$ algorithm tends to perform better with smaller values of R. However, choosing very small choices of R, e.g. 2 or 4, tend to result in sub-par performance. In addition, for $\mathcal{A}_{MI}$ we have implemented the 'low packet loss rate' version of the algorithm [8], where the new round's sending rate is adjusted based on the previous round's receiving rate and a parameter ϵ, i.e., the algorithm sends $1 + \epsilon$ packets for each packet received. In our test we have used ϵ values of 0.10 and 0.01. Tests using $\epsilon = 1$ had sub-par performance. It is worth noting that our $\mathcal{A}_{AI}$ algorithm is parameterless.

6.1 Heuristics In our implementations of both $\mathcal{A}_{AI}$ and $\mathcal{A}_{MI}$ for clients with multiple outgoing edges, we sort the edges according to the sending rate and satisfy larger sending rates first. This is similar to the heuristic used in [9], and satisfies servers which have fewer clients first. This does not change the analysis in Section 5 since the $\mathcal{A}_{AI}$ algorithm permits the server requests to be satisfied in any particular order. In addition, when a client has 'leftover' flow, it divides this flow equally among all of its connected servers. This makes for a fairer comparison, since clients in the $\mathcal{A}_{RA}$ algorithm always send 1 unit of flow per round. We have chosen to split the extra flow equally since in [2] this is shown to be the best assignment for one-shot oblivious routing on general graphs.

6.2 Experimental Results For each graph, we present per-round bandwidth and cumulative throughput results. Per-round bandwidth is measured as the fraction (of optimal) flow units sent per round. Cumulative throughput is measured as the fraction of optimal flow units sent so far, i.e., throughput at round i is the sum of the bandwidth for rounds $0..i$. Results are presented only for the first 50 rounds so that the behavior of the algorithms during the first few rounds is visible. We are interested mainly in the first few (less than 20) rounds, since we are assuming that the network conditions are dynamic. We note that, eventually, after > 100 rounds the $\mathcal{A}_{MI}$ algorithm with small ϵ typically outperforms all of the algorithms which is to be expected.

The qualitative result of our experiments is that the $\mathcal{A}_{AI}$ algorithm throughput outperforms the throughput of the other algorithms over the first 50 rounds for every graph. This is despite the fact that the bandwidth of the other algorithms does occasionally outperform the $\mathcal{A}_{AI}$ algorithm. For example, in Figure 6 we see that the $\mathcal{A}_{MI}$ algorithm with $\epsilon = 0.1$ has the best bandwidth after roughly 17 rounds. The reason is that the $\mathcal{A}_{AI}$ algorithm's bandwidth is close-to-peak after only a couple of rounds; the lower bandwidth of the other algorithms during this time is difficult to make up.

Moreover, it is apparent that one cannot statically pick a single parameter for $\mathcal{A}_{MI}$ or $\mathcal{A}_{RA}\dot{}$In Figure 3, we see that RA_8 outperforms RA_{64}. However, picking a value of $R = 8$ for the Rope graph in Figure 5 would result in lower performance as compared to $R = 64$. Likewise, $\mathcal{A}_{MI}$ with an $\epsilon = 0.10$ typically outperforms $\epsilon = 0.01$. However, in Figure 4, $\mathcal{A}_{MI}$ with $\epsilon = 0.10$ is the worst-performing algorithm. Thus, choosing parameters for $\mathcal{A}_{MI}$ algorithms is difficult. The advantage of the $\mathcal{A}_{AI}$ algorithm is that it is parameterless and essentially adjusts to the parameters of the graph and attempts to maximize throughput in every additional round.

The Focal graph is a special case and deserves some explanation. In this graph (Figure 3), all but one of each client's edges is focused on an overloaded portion of the graph and there is only 1 perfect matching in the graph. The $\mathcal{A}_{RA}$ algorithm requires each edge to carry a trickle bandwidth of $\frac{1}{2\Delta}$, so roughly half of each client's bandwidth is wasted on this overloaded focal point and the two variants of the $\mathcal{A}_{RA}$ algorithms both achieve less than 50% optimal throughput. ($\mathcal{A}_{RA}$ parameterized with larger values of R will eventually reach 50% throughput.) In the case of the $\mathcal{A}_{MI}$ algorithm, each client's single matching edge initially has only a small amount of flow since the client outdegree is high. Thus, while the $\mathcal{A}_{MI}$ algorithms will eventually do well, they will take many more rounds to do so. (In this case, it might be advantageous to run

$\mathcal{A}_{MI}$ with a much higher ϵ value but we did not do so.)

6.3 Discussion The simulations have shown that, for the algorithms that we have tested, the $\mathcal{A}_{AI}$ algorithm reaches close-to-peak bandwidth in very few rounds, reaching 90% or greater of the peak bandwidth in only 3 rounds for the graphs we have tested. This bandwidth-advantage for the initial rounds is difficult to make up for other algorithms, even over tens of subsequent rounds. However, the advantage of the $\mathcal{A}_{AI}$ algorithm dissipates as the number of rounds increase. For most of the graphs, we have observed that the $\mathcal{A}_{MI}$ algorithm with $\epsilon = 0.10$ has the best throughput when the clients send for > 100 rounds. Thus, for dynamic networks, where the network state persists for only a few time rounds, $\mathcal{A}_{AI}$ will outperform the other algorithms. If, however, the network is static on the order of hundreds of rounds, the $\mathcal{A}_{MI}$ algorithm (and other algorithms such as [4]) will eventually prevail.

7 Conclusion

In this paper we have described a simple semi-oblivious online routing algorithm designed to maximize throughput in a network of clients and servers. We have shown that the competitive ratio of this algorithm improves upon a previous result [2] provided that client demands persist for at least $2(\lceil \log(\Delta) \rceil + 2)$ rounds. Most importantly, the $\mathcal{A}_{AI}$ algorithm runs oblivious to the value of Δ and to the length of client demand persistence, ρ. This enables the algorithm to be run on a variety of demand inputs, deployed more easily, and in general have better throughput than the previous semi-oblivious algorithm on a variety of bipartite graphs. An interesting extension of this work would be to determine a competitive ratio of the algorithm over a sequence of time rounds which have *approximate* demand persistence, e.g. client demands which do not change 'too much' from round to round. The intuition is that the competitive ratio would be inversely proportional to the similarity of demands from round to round, so that a run with highly similar demands would result in a low competitive ratio and vice versa.

References

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[2] B. Awerbuch, M. T. Hajiaghayi, R. D. Kleinberg, and T. Leighton. Online Client-Server Load Balancing without Global Information. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 197–206, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[3] Y. Azar, E. Cohen, A. Fiat, H. Kaplan, and H. Racke. Optimal Oblivious Routing in Polynomial Time. In *In Proc. of ACM Symposium on the Theory of Computation*, 2003.

[4] Y. Bartal, J. W. Byers, and D. Raz. Global Optimization Using Local Information with Applications to Flow Control. In *IEEE Symposium on Foundations of Computer Science*, pages 303–312, 1997.

[5] S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and Self-Stabilizing Distributed Matching. In *PODC '02: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, pages 290–297, New York, NY, USA, 2002. ACM Press.

[6] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi. Augment or Push: A Computational Study of Bipartite Matching and Unit-Capacity Flow Algorithms. *J. Exp. Algorithmics*, 3:8, 1998.

[7] Y.-H. Chu and A. Ganjam. Early Experience with an Internet Broadcast System Based on Overlay Multicast. In *USENIX Annual Technical Conference*, pages 155–170, 2004.

[8] N. Garg and N. E. Young. On-Line End-to-End Congestion Control. In *FOCS '02: Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 303–312, Washington, DC, USA, 2002. IEEE Computer Society.

[9] N. Harvey, R. Ladner, L. Lovász, and T. Tamir. Semi-matchings for Bipartite Graphs and Load Balancing. In *Proc. 8th WADS*, pages 294–306, 2003.

[10] Y. T. Hou, S. S. Panwar, and H. H.-Y. Tzeng. On Generalized Max-Min Rate Allocation and Distributed Convergence Algorithm for Packet Networks. *IEEE Transactions on Parallel and Distributed Systems*, 15(5):401–416, 2004.

[11] Y. Liu, Y. Gu, H. Zhang, W. Gong, and D. Towsley. Application Level Relay for High-bandwidth Data Transport. In *The First Workshop on Networks for Grid Applications (GridNets)*, San Jose, CA, October 2004.

[12] M. Luby and N. Nisan. A Parallel Approximation Algorithm for Positive Linear Programming. In *STOC '93: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 448–457, New York, NY, USA, 1993. ACM Press.

[13] Q. Ma, P. Steenkiste, and H. Zhang. Routing High-Bandwidth Traffic in Max-Min Fair Share Networks. In *SIGCOMM*, pages 206–217, 1996.

[14] S. Naher. LEDA — A Library of Efficient Data Types and Algorithms. *Lecture Notes in Computer Science*, 665, 1993.