Online Load Balancing and Network Flow

Steven Phillips
Computer Science Department
Stanford University *

Jeffery Westbrook
Computer Science Department
Yale University †

Abstract

In this paper we study two problems that can be viewed as on-line games on a dynamic bipartite graph. The first problem is on-line load balancing with preemption. A centralized scheduler must assign tasks to servers, processing online a sequence of task arrivals and departures. Each task is restricted to run on some subset of the servers. The scheduler attempts to keep the load well-balanced. If preemptive reassignments are dissallowed, Azar, Broder and Karlin [3] proved a lower bound of $\Omega(\sqrt{n})$ on the ratio between the maximum load achieved by an on-line algorithm and the optimum off-line maximum load. We show that this ratio can be greatly reduced by an efficient scheduler using only a small amount of rescheduling.

We then apply these ideas to network flow. Cheriyan and Hagerup [6] introduced an on-line game on a bipartite graph as a fundamental step in improving algorithms for computing the maximum flow in networks. They described a randomized strategy to play the game. King, Rao and Tarjan [11] studied a modified version of this game, called "node kill", and gave a deterministic strategy. We obtain an improved deterministic algorithm for the node kill game (and hence for maximum flow) in all but the sparsest graphs. The running time achieved is $O(mn \log_{m/n} n + n^2 \log^{2+\epsilon} n)$, compared to the King, Rao and Tarjan's $O(mn + n^{2+\epsilon})$.

These problems combine a demand for good competitive ratios with more traditional requirements of implementation efficiency. Our solutions deal with the tradeoffs between these measures.

1 Introduction

1.1 Load Balancing with Preemption

In the dynamic load balancing problem we have a fixed set V of n servers and a dynamic set of tasks U that arrive and depart on-line. Each task has edges to a *subset* S of the servers and at all times it must be matched to some server in S. A server may have multiple tasks assigned to it simultaneously. The *load* on a server is the number of tasks currently assigned to it.

Azar, Karlin and Broder [3] proposed the dynamic load balancing problem as an abstraction of scheduling problems that can occur in heterogenous networks containing workstations, I/O devices, etc. Servers correspond to communication channels and tasks to requests for communication links between devices. A network controller must coordinate the channels so that no channel is too heavily loaded. The request sequence is unknown in advance.

Let $Load(A, \sigma)$ be the maximum load on any server after scheduler A processes sequence σ. Following the conventions of competitive analysis, we say the *competitive ratio* of A is the supremum over sequences σ of $Load(A, \sigma)/Load(OPT, \sigma)$, where $Load(OPT, \sigma)$ is the minimum maximum load after σ.

*Partially supported by NSF Grant CCR-9010517 and grants from Mitsubishi and OTL.

†This research partially done while the author was visiting DEC Systems Research Center. Partially supported by NSF Grant 9009753.

Azar, Naor and Rom [4] studied dynamic load balancing under two restrictions: first, that no tasks ever depart; and second, that tasks cannot be preempted. Azar *et al.* showed that the greedy strategy has a competitive ratio of $\log n$ and that this is optimal within constant factors among both deterministic and randomized algorithms. If for each task the eligible subset is all servers, however, then the competitive ratio falls to $2 - \epsilon$ for some small constant ϵ [10, 5].

Azar, Broder and Karlin [3] studied the dynamic load balancing problem when tasks both arrive and depart, but retained the restriction that tasks cannot be preempted. In this version, the picture is worse: using our definition of $Load(A, \sigma)$, the best possible competitive ratio is n. Azar *et al.* defined $Load(A, \sigma)$ to be the maximum load at *any* time during σ and showed under this definition that the greedy strategy has a ratio of $\Theta(n^{2/3})$. They also proved a bound of $\Omega(\sqrt{n})$ on the competitive ratio of any randomized or deterministic strategy. (Again, under this definition the ratio is $2 - \epsilon$ if the eligible subset is all servers.) In both [3, 4] the same bounds are shown for an extension where tasks have weights. These strong bounds indicate that the performance of an online scheduler may be severely impacted by the lack of information about future tasks. The $O(\sqrt{n})$ lower bound has recently been shown to be tight [2].

We address the problem of the huge lower bound on the achievable competitive ratio by giving the scheduler a little more power. We study *load balancing with task preemption.* That is, the scheduler is allowed to reassign tasks to servers as new jobs arrive, in order to keep the load well-balanced. In this problem we cannot simply measure an on-line algorithm by a competitive ratio. Since the scheduler is free at any point to reassign all tasks, it can always guarantee a competitive ratio of 1. On the other hand, this approach may be undesirable in practice. Preempting a task can be an expensive process and one may be willing to accept an imbalance in the load in order to avoid the expense of reassignments. Similarly, one may wish to limit the time the scheduler spends processing task arrivals and departures, in which case computing an optimal assignment may be too expensive.

With this in mind, we measure the quality of an on-line scheduler for preemptive load-balancing in three ways: the *competitive ratio* it achieves; the *number of preemptive reassignments* performed during a run; and the overall *computation time* during a run.

In Section 2 we show that the competitive ratio can be greatly reduced by a small amount of preemptive rescheduling. We consider two algorithms, Balance-A and Balance-B. The Balance-A algorithm keeps the maximum on-line load within $12(\log n)/\rho$ times the optimum maximum load while performing ρm preemptive reassignments on a sequence of m task arrivals and departures. Here n is the number of servers, m the total number of tasks, and ρ an arbitrary parameter less than 1 that may be a function of n or m. Balance-B is a version of Balance-A that incurs a small increase in the maximum load to keep computation costs low (linear in the total degree of the arriving tasks plus logarithmic overhead per task arrival or departure).

For comparison, consider two other preemptive scheduling algorithms. First, we may never preempt and use the algorithm of Azar *et al.* [3]. This gives a ratio of $\Theta(n^{2/3})$ and optimal computation time linear in the total degree. At the other extreme, we may rebuild an optimum assignment each time a task arrives or departs. An optimum assignment can be computed in $\Theta(n^3 \log n)$ time using binary search and a b-matching algorithm [12]. This results in a ratio of 1, $\Theta(m^2)$ reassignments, and $\Theta(mn^3 \log n)$ total computation.

Our algorithms extend to more general versions of the problem where both tasks and servers may arrive and depart, and where tasks have an associated weight and the maximum weighted load must be minimized.

1.2 Network Flow

The node kill game, introduced by King *et al.* [11], is played on a bipartite graph $G = (U, V, E)$ where $|U| = |V| = n$ and $|E| = m$. We refer to the nodes in U as items and the nodes in V as columns. The rules of the game are as follows.

1. At the start of each turn, player A (the algorithm) must designate a *current edge* $\{u, v\}$ for each item u. We say u is assigned to v.

2. Player B (the adversary) begins a turn by either removing a single edge (an *edge kill*) or removing all edges incident to a single column (a *node kill*). If an item u has no remaining edges it is out of the game.

3. Player A completes the turn by designating a new current edge for any each item u whose former current edge was removed. In addition, Player A may *redesignate* the current edge of any other item y.

4. Player B scores one point for each current edge it removes in a node kill. It scores zero points for each edge removed by an edge kill. It scores one point each time player A redesignates the current edge of an item.

The similarity to the online load balancing problem is evident. Clearly Player A should keep the columns balanced in some way, to prevent Player B from gaining too many points during node kills. Edge kills are closely related to task departures.

The number of points scored by B is the number of dynamic tree operations performed by the Goldberg-Tarjan algorithm for maximum flow [9]. The correspondence between the game and the running time for network flow is given by the following theorems from King *et al.*

Theorem 1 *Given a strategy to play the node kill game, scoring at most $P(n, m)$ points with an implementation cost of $C(n, m)$, maximum flow in a graph with N nodes and M edges can be computed in time $C(N^2, NM) + O(N^{3/2}M^{1/2} + P(N^2, NM) \log N)$.*

Theorem 2 *When the node kill game is used for network flows, if at most $P(n, m)$ points are scored in the node kill game, then the number of edge kills is at most $O(\sqrt{nm} + P(n, m))$.*

Cheriyan and Hagerup [6, 7] first introduced a restricted form of this game, in which player A may not redesignate edges, and described a randomized strategy. Alon [1] derandomized this strategy. King *et al.* extended the game to the form described above and found an improved deterministic strategy.

We apply ideas from online load balancing to play the node-kill game well. The *ratio algorithm* for the node kill game is presented in Section 3. It provides a speedup for computing maximum flow, for m between $n \log^{1+\epsilon} n$ and $n^{1+\epsilon}$. The results achieved are summarized in table 1.2. King, Tarjan and Rao have recently extended their algorithm, and now achieve the same running time as our algorithm.

In addition to the running time of a maximum flow algorithm, Cheriyan *et al.* studied the number of real number operations, referred to as *flow* operations. Their randomized algorithm gives the first bound of $o(nm)$ on the number of flow operations — in particular, an expected bound of $O(n^{3/2}m^{1/2} \log n + n^2(\log n)^2/\log(2 + n(\log n)^2/m))$. King *et al.* have shown that their extended deterministic algorithm performs only $O(m^{1/2+\epsilon}n^{3/2-\epsilon}log_{m/n}^{1/2} n \log n)$ flow operations. [1] We show that our algorithm also achieves this number of flow operations.

[1] King *et al.* also achieve a slightly smaller number of flow operations at the cost of a small increase in running time.

Algorithm	Runtime	Determinstic?
Goldberg, Tarjan 1988	$O(mn \log n^2/m)$	Yes
Cheriyan, Hagerup Mehlhorn 1990	$O(mn + n^2 \log^2 n)$	No
Alon 1989	$O(mn + n^{8/3} \log n)$	Yes
King, Rao, Tarjan 1991	$O(mn + n^{2+\epsilon})$	Yes
This paper	$O(mn \log_{m/n} n + n^2 \log^{2+\epsilon} n)$	Yes

Table 1: Running time for Network Flow

2 Dynamic Load Balancing with Preemption

Our algorithms combine the greedy strategy of assigning an arriving task to an adjacent server of minimum load with an efficient rebalancing technique that occasionally redistributes the work of heavily loaded servers.

2.1 Balance-A

This section describes a simple and efficient algorithm, Balance-A. There is a lot of flexibility in the details of the algorithm and its implementation, and its robustness and simplicity indicate that it may be useful in other scheduling and load balancing applications.

The algorithm takes a parameter ρ which is the desired ratio of preemptive reassignments to tasks leaving the system. We constrain $1 \geq \rho > 0$. For the moment we assume that ρ is a fixed constant, although the algorithm will work with ρ a function of n or even s, where s is the number of tasks currently in the system.

A *greedy assignment* of a task means that the task is assigned to the adjacent server of minimum load. Balance-A works as follows. When a new task u arrives, it is assigned greedily to some server v. After each task arrival or departure, Balance-A executes a load balancing procedure described below. The procedure first makes an estimate of the optimum maximum load. It then performs a limited number of reassignments designed to bring the loads into approximate balance, based on that estimate.

If we order the tasks on a server by the time they were last assigned there then we can think of each server supporting a column of tasks, with the earliest assigned tasks at the bottom and the latest assigned on top. The *height* of task u, $h(u)$, is the number of tasks currently below it. The height of server v, $h(v)$, is the load on v. We define a third quantity, $m(v)$, which roughly corresponds to the maximum height v ever had. When task u is assigned to server v, set $h(u) = h(v)$, $h(v) = h(v) + 1$, and $m(v) = \max\{m(v), h(v)\}$. When a task departs from server v, decrement $h(v)$ but do not change $m(v)$.

For a given choice of ℓ, we can partition the load on each server into levels of size ℓ. We make the following definitions:

$$\begin{aligned} \mathcal{W}_j^{(\ell)} &= \{u \in U \mid j\ell \leq h(u) < (j+1)\ell\} \\ \mathcal{R}_j^{(\ell)} &= \{u \in U \mid h(u) \geq j\ell\} \\ \mathcal{M}_j^{(\ell)} &= \{v \in V \mid m(v) \geq j\ell\}. \end{aligned}$$

Let $W_j^{(\ell)} = |\mathcal{W}_j^{(\ell)}|$, $R_j^{(\ell)} = |\mathcal{R}_j^{(\ell)}|$ and $M_j^{(\ell)} = |\mathcal{M}_j^{(\ell)}|$. We will omit the superscript ℓ when the value

of ℓ is clear from the context. Remember that s is the number of tasks currently in the system. The rebalance algorithm is described by the following four steps.

1. Set $\ell = \max\{1, 2^{\lfloor \log s/n \rfloor}\}$.
2. Let j be minimum such that $W_j < R_{j+2}$. If there is no such j, then terminate.
3. If $R_{j+2}/M_{j+2} \geq \rho\ell/3$ then double ℓ and go to Step 2.
4. Set $m(v) = \ell(j+1) \ \ \forall v \in \mathcal{M}_{j+1}$. Greedily reassign all tasks in $\mathcal{R}_{j+1}$ and terminate.

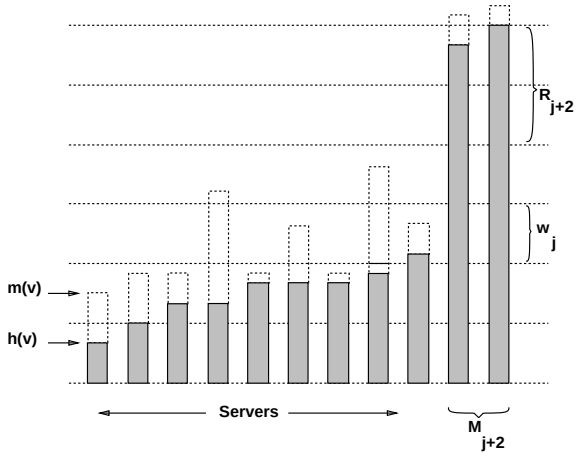

Figure 1: In Step 3 $W_j < R_{j+2}$, and $R_{j+2}/M_{j+2} \geq \rho\ell/3$, so ℓ is doubled.

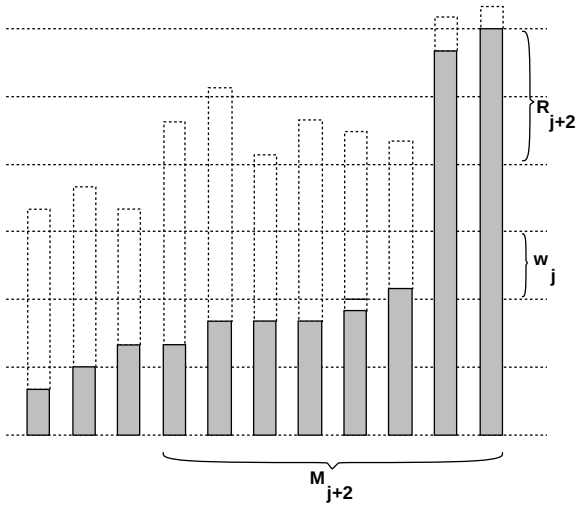

Figure 2: In Step 4 $W_j < R_{j+2}$, and $R_{j+2}/M_{j+2} < \rho\ell/3$, so $\mathcal{R}_{j+1}$ reassigned.

The reasoning behind the algorithm as follows.

- The variable ℓ is always at most $6/\rho$ times OPT, the optimum maximum load. In Step 1, ℓ is initialized to a trivial lower bound on OPT. We only increase ℓ in Step 3, and then only if we find a large number of set ($\mathcal{R}_{j+2}$) that can only be assigned to a small set of servers ($\mathcal{M}_{j+2}$) — see Figure 1. This gives a lower bound of R_{j+2}/M_{j+2} on OPT.

- If the number of levels is $O(\log n)$, we immediately have a competitive ratio of $O(\log n)$. If the number of levels is larger, then there is some level $j = O(\log n)$ where $W_j < R_{j+2}$.
- If R_{j+2}/M_{j+2} is small, then the number of tasks that have departed from servers in $\mathcal{M}_{j+2}$ is enough to pay for reassigning all tasks in $\mathcal{R}_{j+1}$. See Figure 2.
- With unlimited time, we could calculate an optimal reassignment of $\mathcal{R}_{j+1}$. However, we don't need to. The theorem of Azar, Naor and Rom shows that greedily reassigning $\mathcal{R}_{j+1}$ only increases the maximum load by at most $OPT \log n$. Hence the maximum load is $O(j\ell + OPT \log n) = O(OPT \log n/\rho)$.

The following lemmas prove that the rebalance procedure halts correctly. Regarding termination, observe that the algorithm must halt in Step 2 once $\ell \geq s$.

To prove correctness, we begin by bounding the number of reassignments that occur during the sequence of task arrivals and departures. We will define an *operation* to be either a task arrival and greedy assignment, a task departure, or an execution of the load-balancing procedure. We define a *potential function* Φ and say that the amortized number of reassignments for an operation is the actual number of reassignments plus the change in the potential.

Lemma 3 *The amortized number of reassignments per departing task is ρ. The amortized number of reassignments per task arrival or load-balancing operation is at most zero.*

Let $\Phi = \sum_v \rho(m(v) - h(v))$. The assignment of a task to a server cannot increase the value of Φ, and the deletion of a task increases Φ by at most ρ.

Consider a reassignment done in Step 4 of the load balancing procedure with value ℓ. We show that the decrease in potential is at least R_{j+1}, the number of tasks reassigned.

Let $F = \{v \in V \mid h(v) \geq (j+1)\ell\}$. We have

$$|F| \leq W_j/\ell \leq R_{j+2}/\ell.$$

Let $E = \mathcal{M}_{j+2} \setminus F$. For each server $e \in E$, $h(e) < (j+1)\ell$. Upon resetting $m(e)$ to $(j+1)\ell$, the potential changes as follows:

$$\begin{aligned} \Delta\Phi(e) &= \rho[(j+1)\ell - h(e)] - \rho[m(e) - h(e)] \\ &\leq \rho[(j+1)\ell - (j+2)\ell] \\ &\leq -\rho\ell \end{aligned}$$

The potential associated with the servers outside of E is non-increasing. The reassignment of the tasks above level j cannot increase the potential. Hence

$$\begin{aligned} -\Delta\Phi &\geq |E|(\rho\ell) \\ &\geq (\rho\ell)\,(M_{j+2} - |F|) \\ &\geq (\rho\ell)\left(\frac{3R_{j+2}}{\rho\ell} - \frac{R_{j+2}}{\ell}\right) \\ &\geq 2R_{j+2}. \end{aligned}$$

Since $R_{j+2} = R_{j+1} - W_{j+1}$ and $W_{j+1} \leq W_j < R_{j+2}$, we conclude $-\Delta\Phi \geq 2R_{j+2} \geq R_{j+1}$. □

Lemma 4 *If task u has an edge to server v, then $m(v) \geq h(u)$.*

The lemma is trivially true if u is assigned to server v. Suppose u is assigned to server w. When u was greedily assigned to w, column v must have had at least the same height as w, and hence $m(v)$ was at least that height. The only way for $m(v)$ to decrease is during a rebalancing, but since u has not been reassigned by assumption, that rebalancing must have left $m(v) \geq h(u)$. □

Theorem 5 *The Balance-A algorithm keeps the maximum on-line load within $13 \log n/\rho$ times the optimum maximum load while performing ρm preemptive reassignments over a sequence of m task departures.*

The bound on reassignments follows from Lemma 3 and standard arguments of amortized analysis.

Let OPT be the optimum maximum load. Observe that if there are a tasks that can only be placed on a given subset of b servers, then in any assignment of tasks to servers some server must have load $\lceil a/b \rceil$. Suppose that in Step 3 $R_{j+2}/M_{j+2} \geq \ell\rho/3$. By Lemma 4, the tasks in $\mathcal{R}_{j+2}$ can only be placed on servers in $\mathcal{M}_{j+2}$, so $OPT \geq R_{j+2}/M_{j+2}$, and therefore $\ell \leq 6\ OPT/\rho$.

Now assume that the rebalance algorithm terminates without reassigning tasks, so for all j, $W_j < R_{j+2}$. Then the recurrence $R_{j+1} = R_j - W_j$ implies $R_{j+2} \leq R_j/2$ for each j. We have $R_0 = s$, and if $R_j \leq s/n$ then $R_{j+1} = 0$, since all remaining tasks can be placed on one machine in level j. Hence there are at most $2 \log n$ levels, so the maximum load is at most $12\ OPT \log n/\rho$.

On the other hand, assume a reassignment is done. Since j is chosen to be minimal, we have $j + 1 \leq 2 \log n$. Consider the set of tasks that are in $\mathcal{R}_{j+1}$ after the reassignment: these tasks were assigned greedily, without any task departures, so by the $\log n$-competitiveness of the greedy algorithm for *static* load balancing [4], the maximum server load is at most $(j+1)\ell + OPT \log n \leq 13\ OPT \log n/\rho$ after the reassignment. □

Note that our algorithm belongs to a class of algorithms that does no preemptive rescheduling unless tasks depart the system. Hence we cannot hope to achieve a bound on the competitive ratio better than $\Omega(\log n)$, since the lower bound of Azar *et al.* [4] applies in the case of permanent tasks. This lower bound also applies to randomized algorithms in the same class.

2.2 Running Time of Balance-A

If a task arrives with a list of d edges to eligible servers, greedily assigning it takes $\Theta(d)$ time.

To implement the balance condition tests in Steps 2 and 3, the algorithm keeps $O(\log n)$ arrays, one for each estimate ℓ that may be tested. Each array stores the first $O(\log n)$ values W_j^ℓ. Any tasks with height greater than the highest level are included in that level. As s doubles or halves we add new arrays for the minimum or maximum values of ℓ. The amortized cost is $O(1)$ per task. To check the balance condition each successive value R_j is computed using the recurrence $R_{j+1} = R_j - W_j$, $R_0 = s$. An analogous scheme is used for computing the values of M_j.

We show that the total time checking for violations of the balance condition is $O(\log n)$. Suppose we begin Step 3 with some value of j and ℓ. Since $W_{j'}^{(\ell)} \geq R_{j'+2}^{(\ell)}$ for $j' < j$, we know that $W_{j'}^{(2\ell)} \geq R_{j'+2}^{(2\ell)}$ for all $j' < \lfloor j/2 \rfloor$. Therefore if ℓ is doubled and the procedure loops back to Step 2, it need only check the balance condition for $j' \geq \lfloor j/2 \rfloor$. The value $R_{\lfloor j/2 \rfloor}^{(2\ell)}$ can be computed in $O(1)$ time from $R_j^{(\ell)}$ and the W values. Furthermore, $R_{j+1}^{(\ell)} \geq R_{\lfloor j/2 \rfloor + 1}^{(2\ell)}$, and every two times through the calculation loop of Step 2 the value of R decreases by 2. Combining these observations, we

conclude that the total number of balance condition tests in Step 2 is $O(\log n)$, and the total time in Steps 2 and 3 is $O(\log n)$.

To implement rebalancing we need additional data structures. For each server we maintain a stack of the tasks on that server. Then the set $\mathcal{R}_{j+1}$ can be found easily given the set $\mathcal{M}_{j+1}$. The set $\mathcal{M}_{j+1}$ can be computed in $O(\log n + M_{j+1})$ time by storing the set of servers in a balanced binary tree ordered by $m(v)$.

The cost of assigning a task to a server includes the basic cost of finding the minimum adjacent server plus the overhead of updating the above data structures. The overhead is $O(\log n)$ worst-case. One entry must be updated in each of the $O(\log n)$ arrays storing the values $W_j^{(\ell)}$. The balanced tree of servers can be appropriately modified in time $O(\log n)$ following updates to $m(v)$. By Theorem 5 there are $O(m)$ reassignments over m task arrivals and departures. Hence the amortized overhead per task arrival or departure is $O(\log n)$.

The basic cost of greedy reassignments per task arrival or departure, is $O(d)$, where d is sum of the degrees of all tasks currently in U.

Theorem 6 *The Balance-A algorithm runs in $O(d + \log n)$ amortized time per task arrival and departure.*

2.3 Balance-B

The Balance-B algorithm improves on the running time of Balance-A at the expense of a worse ratio in the load. It is based on the following observation. Suppose all tasks have identical degree b. Then the cost of computing a greedy reassignment of tasks can be amortized against the tasks that have departed by defining a second potential function $\hat{\Phi} = b\Phi$ and duplicating the proofs for the number of reassignments. It is easily seen that the amortized computation cost of an insertion remains $O(b)$ while the amortized cost of a rebalancing is 0.

In general, tasks $u \in U$ have arbitrary degree $d(u)$. The Balance-B algorithm makes up for this by maintaining $\log_\delta n$ different classes of tasks, where δ is a parameter between 2 and n. The tasks in class i have degree between δ^i and δ^{i+1}. Within each class independently we run the Balance-A algorithm.

Theorem 7 *The Balance-B algorithm keeps the maximum on-line load within $13(\log n)(\log_\delta n)/\rho$ times the optimum maximum load, performs ρm preemptive reassignments over a sequence σ of m task arrivals and departures, and runs in total time $O(m \log n + D(1 + \rho\delta))$, where $D = \sum_{u \in \sigma} d(u)$.*

The increase in the maximum load by a factor of $\log_\delta n$ follows from there being $O(\log_\delta n)$ classes. To prove the claimed running time, define a potential $\Phi_i = \delta^{i+1}\Phi$ for each class i. When a reassignment is done, the drop in potential is at least the total degree of the reassigned tasks. When a task departs, the increase in potential is at most $\rho\delta$ times its degree. Hence the total time spent on reassignments is $O(D\rho\delta)$. After including $O(m \log n)$ time for updating the arrays $W_j^{(\ell)}$ during task arrivals and departures, and $O(D)$ time for initial (nonpreemptive) assignments of tasks, the total execution time is $O(m \log n + D(1 + \rho\delta))$. □

Note that no algorithm can improve on $\Theta(D)$ time unless it ignores some edges on tasks.

2.4 Weighted Load-Balancing

Our algorithms are also applicable when each task u has a real size $s(u) \geq 1$ and the load on a server is the sum of the sizes of the tasks assigned to that server. Let the height of a task be

the sum of the sizes of tasks under it on its server. The load balancing procedure is essentially the same, although there is the small complication that tasks can lie across the boundary between levels. We redefine $\mathcal{W}_j$, $\mathcal{R}_j$, W_j, and R_j to deal with weighted tasks as follows:

$$\begin{aligned}
\mathcal{W}_j^{(\ell)} &= \{u \in U \mid h(u) < (j+1)\ell \ \&\ h(u) + s(u) > j\ell\} \\
\mathcal{R}_j^{(\ell)} &= \bigcup_{j' \geq j} \mathcal{W}_{j'}^{(\ell)} \\
W_j^{(\ell)} &= \sum_{u \in \mathcal{W}_j^{\ell}} \min\{h(u) + s(u), (j+1)\ell\} - \max\{h(u), j\ell\} \\
R_j^{(\ell)} &= \sum_{j' \geq j} W_{j'}^{(\ell)}
\end{aligned}$$

The definitions of $\mathcal{M}_j^{(\ell)}$ and $M_j^{(\ell)}$ remain the same. We use the following modified load balancing algorithm. Let B be the maximum of the largest job size and the sum of the all job sizes divided by n.

1. Set $\ell = B$.
2. Let j be minimum such that $W_j < R_{j+3}$. If there is no such j, then terminate.
3. If $R_{j+3}/M_{j+2} \geq \rho\ell/4$ then double ℓ and go to Step 2.
4. Set $m(v) = \ell(j+1) \ \ \forall v \in \mathcal{M}_{j+1}$. Greedily reassign all tasks in $\mathcal{R}_{j+1} \setminus \mathcal{W}_j$ and terminate.

Note that in this algorithm, we use R_{j+3} in Steps 2 and 3, rather than R_{j+2} as in the unweighted case. Let u be a task in $\mathcal{R}_{j+3}$ and let v be adjacent to u. In the unweighted case, we had $m(v) \geq (j+3)\ell$, but now we know only that $m(v) \geq (j+2)\ell$, since $s(u) \leq B \leq \ell$.

Using the potential function $\Phi = \sum_{v \in V} \rho(m(v) - h(v))$ we can repeat the proof of Lemma 3 (with R_{j+3} in place of R_{j+2}) and show that the amortized weighted cost of reassignments charged to a departing task u is $\rho s(u)$. That is, the total size of all tasks reassigned over a sequence of m task departures is ρS, where S is the total size of all departing tasks in the request sequence. If a is a lower bound on the size of a task, this bound implies that the total *number* of reassignments is at most $\rho S/a$. In addition, we can argue that the value of ℓ is always at least $8OPT/\rho$. Following the reasoning of Theorem 5 we deduce the following theorem.

Theorem 8 *The modified Balance-A algorithm achieves a competitive ratio of* $25 \log n/\rho$ *while the total size of jobs reassigned is at most* ρS*, where* S *is the sum of sizes of all tasks that ever arrive.*

To improve the running time, we can use the same partitioning scheme as before and easily derive an algorithm that achieves a ratio of $25 \log n \log_\delta n/\rho$ and runs in time $O((m + \rho S/a) \log n + (1 + \delta\rho S/a)D)$.

If bounds on the sizes of the tasks are known, however, then we can improve the running time as follows. Suppose $a \leq s(u) \leq b$ for all tasks u. Partition the input tasks into $O(\log_\delta(bn/a))$ classes such that task u is in class i if

$$\frac{\delta^i}{b} \leq \frac{d(u)}{s(u)} < \frac{\delta^{i+1}}{b}.$$

In other words, partition the tasks based on number of edges per unit weight. The maximum value of the ratio is n/a, giving the bound on the number of classes.

For each class i define a potential function $\hat{\Phi}_i = \frac{\delta^{i+1}}{b}\Phi$. Now if, while rebalancing class i, a collection of tasks having total size s is reassigned, then at most $\frac{\delta^{i+1}}{b}s$ edges must be examined. The potential Φ decreases by s, however, to pay for the weighted reassignments, and hence $\hat{\Phi}_i$ decreases sufficiently to pay for examining the edges. The increase in $\hat{\Phi}_i$ when task u departs class i is given by $\rho s(u)\frac{\delta^{i+1}}{b} \leq \delta\rho d(u)$. We have the following.

Theorem 9 *The modified Balance-B algorithm achieves a ratio of* $25(\log n)(\log_\delta bn/a)/\rho$*, runs in total time* $O(m\log n + D(1+\rho\delta) + \ min\{\rho S/a, \rho\delta D\}\log n)$*, and the total size of jobs reassigned is at most* ρS*, where* S *is the sum of sizes of all tasks that ever arrive.*

3 Network Flow

3.1 Intuition

Consider the *EXACT-LOAD* algorithm for the node kill game: whenever an item's current edge is killed, reassign that item to the neighbouring column with minimum load. Development of our algorithm for the node kill game was motivated by the proof of the following theorem.

Theorem 10 *If there are no edge kills, EXACT-LOAD has a point score of* $O(n\log^2 n)$.

Proof: Consider the WATER-LEVEL algorithm, that is allowed to divide an item into pieces of arbitrary size, assigning the pieces to neighbours of the original item as desired. The height of a column is then the sum of the sizes of the pieces assigned to it. WATER-LEVEL redivides and reassigns items as frequently as necessary to maintain the following condition: If there is an item x with a piece on a column c and an edge to column c', then $h(c') \geq h(c)$. (The WATER-LEVEL Algorithm is actually the optimal offline algorithm for minimizing the maximum load at all times).

We compare EXACT-LOAD to the WATER-LEVEL algorithm, showing the following two points:

1. Let c be any column. The EXACT-LOAD height of c is always at most a factor $O(\log n)$ larger than $\lceil h_W(c)\rceil$, where $h_W(c)$ is the WATER-LEVEL height of c.
2. WATER-LEVEL incurs $O(n\log n)$ node kill points (where the point score of a column kill to c is $\lceil h_W(c)\rceil$).

For point 2, fix a sequence of column kills. Assume there is some time t when the column killed does not have minimal height (under WATER-LEVEL's assignment). Let c be the first column later in the sequence that has minimal height. The height of c does not increase after time t when the columns up till c in the sequence are killed, since by the definition of WATER-LEVEL, when the item pieces on those columns are reassigned, nothing can trickle down to c. Hence moving the kill of column c forward to time t does not decrease the point score. Hence there is a column kill sequence that achieves a maximum point score and always kills the current column of minimum height. For any such sequence WATER-LEVEL achieves a point score of $O(n\log n)$, since the ith column killed has height at most n/i.

For point 1, let C be the set of columns whose WATER-LEVEL height is no more than $h_W(c)$, and consider the set I of items EXACT-LOAD has on columns in C. Note that WATER-LEVEL assigns all pieces of each item in I to columns in C. Any $x \in I$ has at least one edge to a column

in $c_1 \in C$, and so if a piece of x were placed on a column $c_2 \notin C$ it would violate the required condition $h(c_1) \geq h(c_2)$.

The EXACT-LOAD assignment of items in I to columns in C is a greedy assignment, and the following analysis is similar to the proof that greedy algorithm for online assignments is $\log n$-competitive [4]. Order the items on columns of C under EXACT-LOAD's assignment according to the time they were assigned. Partition each column into levels of $\lceil h_W(c) \rceil$. Let R_j be the set of items in level j and above, and let W_j be the set of items in level j. At least $k \geq |R_j|/h_W(c)$ columns must have at least one neighbor in R_j, since WATER-LEVEL can fit all items in R_j into one level of height $h_W(c)$. By the definition of EXACT-LOAD, any such column must reach level j. This implies $W_{j-1} \geq k \lceil h_W(c) \rceil \geq R_j$. Hence the number of levels is $O(\log n)$. $\square$

3.2 The Ratio Algorithm

The disadvantage with EXACT-LOAD is that every time an item is reassigned, the entire edge set of that item must be scanned to find the lowest incident column. This is too much work in the case of network flow. To avoid this problem we use a different but related approach.

If a column v has height (or load) $h(v)$ and initial degree d_v, define $r(v) = h(v)/d_v$ as the *ratio* of v. There are two main differences between our algorithm for load balancing and our algorithm for the node kill game. The first is that we perform greedy reassignments based on ratio rather than load.

The second difference is that it is too costly for items to monitor the precise ratio of neighbouring columns. Define $r = 1/\log n$. If a column v has ratio $r(v)$, define the *level* $l(v) = \lfloor r(v)/r \rfloor$. The algorithm maintains an *estimated level* $el(v)$ which is known by all neighbours of v. The estimated levels are updated as lazily as possible, subject to maintaining the relationship $el(v) = l(v)$ or $l(v) + 1$. This way many assignments or edge kills must occur on a column before its estimated level changes. Greedy assignments are based on estimated level. The algorithm of King *et al.* [11] also used greedy assignments by estimated level.

The Ratio Algorithm sometimes makes some columns *inactive* Initially, any column v with degree $d_v \leq \log n$ is made inactive. When all the remaining neighbours of an item have become inactive, the item is also inactive. From that time, whenever the item is subjected to an edge kill or node kill, it is reassigned to an arbitrary neighbour.

From now on we ignore inactive columns and items in the presentation of the algorithm. We can now describe the Ratio Algorithm. It uses a Rebalance routine (described below), which is closely related to the rebalance routine of Section 2.

- A *greedy assignment* of an item u is as follows:
  1. Assign u to the adjacent column w of minimum estimated level.
  2. Increment $h(w)$. If $r(w) > (el(w) + 1)r$ then increment $el(w)$.
  3. Perform a rebalance.
- Suppose the adversary kills the designated edge from $u \in U$ to $v \in V$, forcing a reassignment of u. Proceed as follows.
  1. Decrement $h(v)$. If $r(v) < (el(v) - 1)r$ then decrement $el(v)$.
  2. Greedily assign u.
- To handle a column kill, process each item on the column as in the case of an edge kill.
- To initialize, set $el(v) = 0$ for each v. Then greedily assign each $u \in U$.

3.3 Rebalancing

If an item on column v has k items under it, we say its level is $\left\lfloor \frac{k}{d_v r} \right\rfloor$. Define the *work* at level j, W_j, to be the number of items in level j. The *residual work* above level j, R_j, is $\sum_{k>j} W_k$.

We say the *balance condition* holds at level j if $R_{j+1} \leq \delta W_j$. We will ensure the balance condition always holds at all levels.

Claim 1 *If the balance conditions all hold, each column has level* $O(\log_{1+1/\delta} n)$.

We set the parameter $\delta = (n/m)^\epsilon$ for small constant $\epsilon < 1/2$. Clearly a small value for δ means the maximum ratio that can be attained is small.

Let $\gamma = 2 + 1/\delta$. The Rebalance routine is as follows:

1. Let j be the minimum level such that $R_{j+1} > \delta W_j$. If there is no such level, stop.
2. Let X be the set of items with level $> j+1$. Let E be the set of columns v such that there is an edge from an item in X to v. Let F be the set of columns at levels $\geq j+1$. Finally, let $Y = E \cup F$ and $d_Y = \sum_{v \in Y} d_v$.
3. If $|X|/d_Y < r/\gamma$, greedily reassign all items in X and stop.
4. If $|X|/d_Y \geq r/\gamma$, make all columns in Y inactive. Greedily reassign all items that were on those columns.

In steps 3 and 4, the items to be reassigned are first removed from their columns, then assigned one by one, rebalancing after each assignment. Hence the balance conditions always hold, except perhaps for the effect of the last item assigned.

The complete analysis of the Ratio algorithm appears in the next section. Here we outline the intuition for the above balance steps:

Step 3. The columns in Y have had many adversary edge kills, and though many were originally too high for the items in X to be assigned to them, they aren't any longer. The cost of the reassignments is charged to the edge kills.

Step 4. Many items can only go onto Y, so will become inactive when Y becomes inactive. At most n items can ever become inactive, so the total degree of inactive columns is quite small. This keeps the point score on inactive columns low.

4 Analysis of the Ratio Algorithm

The analysis of the ratio algorithm is quite similar to that of the Balance-A algorithm. In particular, rebalancing work is charged to edge kills here, and to task departures there, in the same way.

Consider a rebalance at level j. Let F be the set of columns at levels $\geq j+1$. These columns are all full in level j, so we have $\sum_{v \in F} r d_v \leq W_j < R_{j+1}/\delta = |X|/\delta$.

Define a potential Φ as follows. For each column v let $m(v)$ be the level of the highest adjacent item x. Then

$$\begin{aligned} \Phi &= \frac{1}{2} \sum_{v \in V} r d_v \max\{0, m(v) - el(v)\} \\ &\quad + \frac{1}{2} \sum_{v \in V} d_v \max\{0, r \cdot el(v) - r(v)\}. \end{aligned}$$

Note that Φ is non-negative, increases only when there is an adversary edge kill, and then by at most $\frac{1}{2}$.

Lemma 11 *If the rebalance routine greedily reassigns in Step 3, the decrease in potential is at least $|X|$, so the amortized cost of the reassignments is ≤ 0.*

The columns in Y that are not in F initially are at level j or lower, hence have estimate level at most $j+1$, and are all adjacent to at least one item in X. Since an item in X is at least at level $j+2$, the potential associated with each such column can be written as $\phi \geq \phi_0 + \frac{1}{2} r d_v$. After the rebalancing, every item at level $j+2$ or higher has been reassigned. Thus the new potential for each such column is at most ϕ_0 and the decrease is at least $\frac{1}{2} r d_v$.

$$\begin{aligned} -\Delta\Phi &\geq \sum_{v \in Y, v \notin F} \frac{1}{2} r d_v \\ &\geq \frac{1}{2}\left(\sum_{v \in Y} r d_v - \sum_{v \in F} r d_v\right) \\ &\geq \frac{1}{2}\left(|X|\gamma - W_j\right) \\ &\geq \frac{1}{2}\left(|X|(2 + 1/\delta) - |X|/\delta\right) \\ &\geq |X| \end{aligned}$$

□

A point is scored each time the Ratio Algorithm has to reassign an item because of a column kill or a rebalance. No points are scored for edge kills, however.

Theorem 12 *The total number of points scored is $O(mr \log_{1+1/\delta} n + n\gamma/r + \sqrt{mn})$.*

First we consider points from column kills on active columns, and from reassignments in Step 4 when columns become inactive. When an active column v is killed or becomes inactive, at most $O(d_v r \log_{1+1/\delta} n)$ items can be reassigned. Hence the total number of these points is $O(\sum_{v \in V} d_v r \log_{1+1/\delta} n) = O(mr \log_{1+1/\delta} n)$.

Secondly we consider points scored by node kills to inactive columns. The number of these points is at most d_I, the total degree of columns that ever become inactive. Initially $d_I \leq n \log n$. By Step 4 of the Rebalance routine, the number of items that become inactive is $\Omega(r d_I/\gamma)$. At most n items can become inactive, so $d_I = O(n\gamma/r)$.

Thirdly we consider points scored in Step 3 of the Rebalance routine. By Lemma 11, the number of these points is at most $\frac{1}{2}$#edge kills. Finally, from Theorem 2 we know that #edge kills $\leq O(\sqrt{mn})$ + #points scored, and the theorem follows. □

The following corollary gives parameters for the best performance of the Ratio Algorithm in terms of points scored in the game, if implementation cost were not an issue.

Corollary 1 *If we use the values $r = \sqrt{\frac{n}{m \log n}}$ and $\delta = 1$, then the total number of game points is $O(\sqrt{mn \log n})$.*

In fact, it is possible to handle the inactive columns more carefully and achieve a point score of $O(n \log n \log(m/n)) + \frac{1}{2}$#edge kills $= O(n \log n \log(m/n) + \sqrt{mn})$.

4.1 Implementation of the Algorithm

To detect when to balance, we need a simple data structure for the columns. Keep arrays $L[\cdot]$ and $R[\cdot]$ of size $\log_{1+1/\delta} n$. Entry $R[j]$ contains the value of R_j and $L[j]$ contains a linked list of the columns v with level j.

For each $u \in U$ we maintain a set of A-lists and a B-list. The active columns adjacent to u are partitioned among $O(\log n)$ linked lists (the A-lists), one for each possible estimated level. List B is a list of active columns adjacent to u such that $el(v)$ has decreased since u was last reassigned. Whenever the estimated level of column v changes, the new value is broadcast to all items u to which v has edges. The A and B lists of those items u are then updated, with columns moving between lists as appropriate. A column can be moved between lists in time $O(1)$. When an item u is greedily assigned, it can find the adjacent column with smallest estimated level in $O(\log n)$ time by scanning to find the first non-empty list.

For each $v \in V$ we maintain a stack of the items assigned to v.

In the Rebalance Algorithm, the set X can be computed by popping items off the stacks of those columns in $L[j']$ for $j' \geq j+2$. This takes time $O(\log n + |X|)$. Given the set X, the set of columns Y can be found as follows. Set F is easily found in time $O(|F|)$ using the $L[\cdot]$ lists. Since $|F| \leq |X|$ from the analysis in Section 4, this takes time $O(\log n + |X|)$. Then examine each item $x \in X$ in turn. Scan the B list of x, and for each column v on the B list of x let $Y = Y \cup \{v\}$. It is easy to verify that the set Y contains all columns that any element in X can be placed on.

Theorem 13 *The total cost of implementing the Ratio Algorithm is $O(m \log_{m/n} n + n^2 \log^{2+\epsilon} n)$ for any constant ϵ.*

We charge the implementation cost of the Ratio algorithm to reassignment points at a rate of $O(\log n)$ time units per reassignment. The theorem follows from substituting the values $r = 1/\log n$ and $\delta = (n/m)^\epsilon$ into the reassignment bound of Theorem 12.

Whenever an item u is reassigned, its B list is reset to NIL, and otherwise it only grows, so B lists can be maintained in $O(1)$ time per modification.

The cost of the Rebalance Algorithm is at most the cost of reassignments plus the sum of the sizes of the B lists of the elements. The latter size can be amortized against the operations that placed the columns on the B lists at $O(1)$ per edge. So the total cost is simply the cost of reassignments.

There must be at least $d_v r$ reassignments or edge kills involving column v to cause a change in $el(v)$. The cost of the subsequent broadcast is $O(d_v)$. Hence the cost of updating lists is $O(1/r)$ per reassignment.

When the height of a column changes, we update the values of $R[\cdot]$ as appropriate, and test for violations of the Balance Invariant. In addition, the column is moved between linked lists as needed. The total time for this is $O(\log n)$. A column changes its height only if a reassignment or kill has happened on it. Hence the work in updating $R[\cdot]$ and $L[\cdot]$ can be amortized against reassignments and edge kills at a rate of at most $O(\log n)$ time units per reassignment. When an item is reassigned, it uses $O(\log n)$ time to make its greedy choice. □

Note that there is a lot of slack in the choice of r: although smaller values of r decrease the point count, they don't affect the implementation cost (as long as $mr \log_{1+1/\delta} n$ remains the dominant term in Theorem 12).

Theorem 14 *Network flow in a graph with N nodes and M edges can be computed in time $O(MN \log_{M/N} N + N^2 \log^{2+\epsilon} N)$.*

Follows from Theorems 12 and 13, with $r = 1/\log n$ and $\delta = (n/m)^{\epsilon}$, and from Theorem 1. □

4.2 Flow operations

The number of flow operations is just $O(\log n)$ times the number of points scored [8]. To determine the minimum number of flow operations our algorithm can perform, while maintaining the running time given in Theorem 14, we use the slack in the choice of r, mentioned above. The smallest r can be made without increasing the running time is

$$r = \frac{(n/m)^{1/2-\epsilon/2}}{\sqrt{log_{m/n} n}},$$

which makes $mr \log_{1+1/\delta} n = n\gamma/r$ in Theorem 12 (the $\sqrt{mn}$ term is insignificant). Under this setting of r, the number of points is

$$(m/n)^{1/2+\epsilon/2} n \sqrt{log_{m/n} n}.$$

Substituting $n = N^2$ and $m = MN$ gives the following expression for the number of flow operations of our algorithm on an N-node, M-edge graph:

$$\#\text{ flow operations } = M^{1/2+\epsilon/2} N^{3/2-\epsilon/2} \sqrt{log_{M/N} N} \log N.$$

This matches the number of flow operations for the extended algorithm of King, Rao and Tarjan.

Acknowledgements

The authors gratefully acknowledge the contribution of Anna Karlin in the early phases of this work.

References

[1] N. Alon. Generating pseudo-random permutations and maximum flow algorithms. *Inf. Process. Lett.*, 1990.

[2] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. Online load balancing of temporary tasks. In *Proc. Workshop on Algorithms and Data Structures*, August 1993. To appear.

[3] Y. Azar, A. Karlin, and A. Broder. On-line load balancing. In *Proc. 33nd Symp. of Foundations of Computer Science*, pages 218–225, 1992.

[4] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 203–210, 1992.

[5] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24nd ACM Symp. on Theory of Computing*, 1992.

[6] J. Cheriyan and T. Hagerup. A randomized maximum flow algorithm. In *Proceedings IEEE Symposium on Foundations of Computer Science*, pages 118–123, 1989.

[7] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a maximum flow be computed in o(mn) time? In *Int. Colloq. on Automata, Languages, and Programming (ICALP 90)*, 1990.

[8] J. Cheriyan, T. Hagerup, and K. Mehlhorn. A $o(n^3)$-time maximum flow algorithm. Technical report, MP11, Saarbruecken, Germany, 1990.

[9] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *J. ACM*, 35:921–940, 1988.

[10] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[11] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 157–164, 1992.

[12] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization.* Prentice-Hall, Englewood Cliffs, NJ., 1982.