Semi-Matchings for Bipartite Graphs and Load Balancing

Nicholas J. A. Harvey[1], Richard E. Ladner[2], László Lovász[1], and Tami Tamir[2]

[1] Microsoft Research, Redmond, WA, USA,
{nickhar, lovasz}@microsoft.com
[2] Department of Computer Science and Engineering,
University of Washington, Seattle, WA, USA,
{ladner, tami}@cs.washington.edu

Abstract. We consider the problem of fairly matching the left-hand vertices of a bipartite graph to the right-hand vertices. We refer to this problem as the *semi-matching* problem; it is a relaxation of the known bipartite matching problem. We present a way to evaluate the quality of a given semi-matching and show that, under this measure, an optimal semi-matching balances the load on the right hand vertices with respect to any L_p-norm. In particular, when modeling a job assignment system, an optimal semi-matching achieves the minimal makespan and the minimal flow time for the system.

The problem of finding optimal semi-matchings is a special case of certain scheduling problems for which known solutions exist. However, these known solutions are based on general network optimization algorithms, and are not the most efficient way to solve the optimal semi-matching problem. To compute optimal semi-matchings efficiently, we present and analyze two new algorithms. The first algorithm generalizes the Hungarian method for computing maximum bipartite matchings, while the second, more efficient algorithm is based on a new notion of *cost-reducing paths*. Our experimental results demonstrate that the second algorithm is vastly superior to using known network optimization algorithms to solve the optimal semi-matching problem. Furthermore, this same algorithm can also be used to find maximum bipartite matchings and is shown to be roughly as efficient as the best known algorithms for this goal.

1 Introduction

One of the classical combinatorial optimization problems is finding a maximum matching in a bipartite graph. The bipartite matching problem enjoys numerous practical applications [2, Section 12.2], and many efficient, polynomial time algorithms for computing solutions [8] [12] [14]. Formally, a bipartite graph is a graph $G = (U \cup V, E)$ in which $E \subseteq U \times V$. A matching in G is a set of edges, $M \subseteq E$, such that each vertex in $U \cup V$ is an endpoint of at most one edge in M; that is, each vertex in U is matched with at most one vertex in V and vice-versa.

In this paper we consider a relaxation of the maximum bipartite matching problem. We define a *semi-matching* to be a set of edges, $M \subseteq E$, such that each vertex in U is an endpoint of exactly one edge in M. Clearly a semi-matching does not exist if there are isolated U-vertices, and so we require that each U-vertex in G have degree at least 1. Note that it is trivial to find a semi-matching — simply match each U-vertex with an arbitrary V-neighbor. Our objective is to find semi-matchings that match U-vertices with V-vertices as *fairly* as possible, that is, minimizing the variance of the matching edges at each V-vertex.

Our work is motivated by the following load balancing problem: We are given a set of tasks and a set of machines, each of which can process a subset of the tasks. Each task requires one unit of processing time, and must be assigned to some machine that can process it. The tasks are to be assigned to machines in a manner that minimizes some optimization objective. One possible objective is to minimize the *makespan* of the schedule, which is the maximal number of tasks assigned to any given machine. Another possible goal is to minimize the average completion time, or *flow time*, of the tasks. A third possible goal is to maximize the fairness of the assignment from the machines' point of view, i.e., to minimize the variance of the loads on the machines.

These load balancing problems have received intense study in the online setting, in which tasks arrive and leave over time [4]. In this paper we consider the offline setting, in which all tasks are known in advance. Problems from the online setting may be solved using an offline algorithm if the algorithm's runtime is significantly faster than the tasks' arrival/departure rate, and tasks may be reassigned from one machine to another without expense. In particular, the second algorithm we present can incrementally update an existing assignment after task arrivals or departures.

One example of an online load balancing problem that can be efficiently solved by an offline solution comes from the Microsoft Active Directory system [1], which is a distributed directory service. Corporate deployments of this system commonly connect thousands of servers in geographically distributed branch offices to servers in a central "hub" data center; the branch office servers periodically replicate with the hub servers to maintain database consistency. Partitioning the database according to corporate divisions creates constraints on which hub servers a given branch server may replicate with. Thus, the assignment of branch servers to hub servers for the purpose of replication is a load balancing problem: the branch servers are the "tasks", and the hub servers are the "machines". Since servers are only rarely added or removed, and servers can be efficiently reassigned to replicate with another server, this load balancing problem is amenable to the offline solutions that we present herein.

Load balancing problems of the form described above can be represented as instances of the semi-matching problem as follows: Each task is represented by a vertex $u \in U$, and each machine is represented by a vertex $v \in V$. There is an edge $\{u, v\}$ if task u can be processed by machine v. Any semi-matching in the graph determines an assignment of the tasks to the machines. Furthermore, we show that a semi-matching that is as fair as possible gives an assignment of tasks to machines that simultaneously minimizes the makespan and the flow time.

The primary contributions of this paper are: (1) the semi-matching model for solving load balancing problems of the form described above, (2) two efficient algorithms for computing optimal semi-matchings, and (3) a new algorithmic approach for the bipartite matching problem. We also discuss in Section 2 representations of the semi-matching problem as network optimization problems, based on known solutions to scheduling problems. Section 3 presents several important properties of optimal semi-matchings. One of these properties provides a necessary and sufficient condition for a semi-matching to be optimal. Specifically, we define a *cost-reducing path*, and show that a semi-matching is optimal if and only if no cost reducing path exists. Sections 4 and 5 present two algorithms for computing optimal semi-matchings; the latter algorithm uses the approach of identifying and removing cost-reducing paths. Finally, Section 6 describes an experimental evaluation of our algorithms against known algorithms for computing optimal semi-matchings and maximum bipartite matchings. Due to space limitations this paper omits proofs for some of the theorems.

2 Preliminaries

Let $G = (U \cup V, E)$ be a simple bipartite graph with U the set of left-hand vertices, V the set of right-hand vertices, and edge set $E \subseteq U \times V$. We denote by n and m the sizes of the left-hand and the right-hand sides of G respectively (i.e., $n = |U|$ and $m = |V|$). Since our work is motivated by a load balancing problem, we often call the U-vertices "tasks" and the V-vertices "machines".

We define a set $M \subseteq E$ to be a *semi-matching* if each vertex $u \in U$ is incident with exactly one edge in M. We assume that all of the vertices in U have degree at least 1 since isolated U-vertices can not participate in the matching. A semi-matching gives an assignment of each task to a machine that can process it.

For $v \in V$, let $deg(v)$ denote the degree of vertex v; in load balancing terms, $deg(v)$ is the number of tasks that machine v is capable of executing. Let $deg_M(v)$ denote the number of edges in M that are incident with v; in load balancing terms, $deg_M(v)$ is the number of tasks assigned to machine v. We frequently refer to $deg_M(v)$ as the *load* on vertex v. Note that if several tasks are assigned to a machine then one task completes its execution after one time unit, the next task after two time units, etc. However, semi-matchings do not specify the order in which the tasks are to be executed.

We define $cost_M(v)$ for a vertex $v \in V$ to be

$$\sum_{i=1}^{deg_M(v)} i = \frac{deg_M(v) \cdot (deg_M(v) + 1)}{2}.$$

This expression gives the total latency experienced by all tasks assigned to machine v. The *total cost* of a semi-matching, M, is defined to be $T(M) = \sum_{i=1}^{m} cost_M(v_i)$. A semi-matching with minimal total cost is called an *optimal semi-matching*. We show in Section 3 that an optimal semi-matching is also optimal with respect to other optimization objectives, such as maximizing the load balance on the machines (by minimizing, for any p, the L_p-norm of the load vector), minimizing the variance of the machines' load, and minimizing the maximum load on any machine.

For a given semi-matching M in G, define an *alternating path* to be a sequence of edges $P = (\{v_1, u_1\}, \{u_1, v_2\}, \ldots, \{u_{k-1}, v_k\})$ with $v_i \in V$, $u_i \in U$, and $\{v_i, u_i\} \in M$ for each i. Without the possibility of confusion, we sometimes treat paths as though they were a sequence of vertices $(v_1, u_1, \ldots, u_{k-1}, v_k)$. The notation $A \oplus B$ denotes the symmetric difference of sets A and B; that is, $A \oplus B = (A \setminus B) \cup (B \setminus A)$. Note that if P is an alternating path relative to a semi-matching M then $P \oplus M$ is also a semi-matching, derived from M by switching matching and non-matching edges along P. If $deg_M(v_1) > deg_M(v_k)+1$ then P is called a *cost-reducing path* relative to M. Cost-reducing paths are so named because switching matching and non-matching edges along P yields a semi-matching $P \oplus M$ whose cost is less than the cost of M. Specifically,

$$T(P \oplus M) = T(M) - (deg_M(v_1) - deg_M(v_k) - 1).$$

2.1 Related Work

The maximum bipartite matching problem is known to be solvable in polynomial time using a reduction from maximum flow [2] [9] or by the Hungarian method [14] [15, Section 5.5]. Push-relabel algorithms are widely considered to be the fastest algorithms in practice for this problem [8].

The load balancing problems we consider in this paper can be represented as restricted cases of *scheduling on unrelated machines*. These scheduling problems specify for each job j and machine i the value $p_{i,j}$, which is the time it takes machine i to process job j. When $p_{i,j} \in \{1, \infty\}\ \forall i, j$, this yields an instance of the semi-matching problem, as described in Section 2.2. In standard scheduling notation [11], this problem is known as $R \mid p_{i,j} \in \{1, \infty\} \mid \sum_j C_j$. Algorithms are known for minimizing the flow time of jobs on unrelated machines [2, Application 12.9] [7] [13]; these algorithms are based on network flow formulations.

The online version of this problem, in which the jobs arrive sequentially and must be assigned upon arrival, has been studied extensively in recent years [3] [5] [6]. A comprehensive survey of the field is given in [4].

2.2 Representation as Known Optimization Problems

The optimal semi-matching problem can be represented as special instances of two well-known optimization problems: weighted assignment and min-cost max-flow. However, Section 6 shows that the performance of the resulting algorithms is inferior to the performance of our algorithms presented in sections 4 and 5.

Recall that the scheduling problem $R \mid\mid \sum_j C_j$, and in particular the case in which $p_{i,j} \in \{1, \infty\}$, can be reduced to a weighted assignment problem [7] [13].

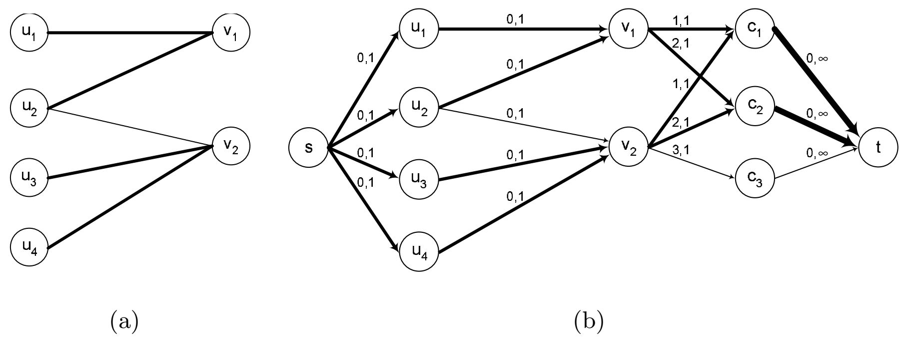

Fig. 1. *(a) shows a graph in which the bold edges form an optimal semi-matching. (b) shows the corresponding min-cost max-flow problem. Each edge is labeled with two numbers: a cost, and a capacity constraint. Bold edges carry one unit of flow and doubly-bold edges carry two units of flow.*

A semi-matching instance can be represented as an $R \mid p_{i,j} \in \{1, \infty\} \mid \sum_j C_j$ instance as follows: Each U-vertex represents a job, and each V-vertex represents a machine. For any job j and machine i, we set $p_{i,j} = 1$ if the edge $\{u_j, v_i\}$ exists, and otherwise $p_{i,j} = \infty$. Clearly, any finite schedule for the scheduling problem determines a feasible semi-matching. In particular, a schedule that minimizes the flow time determines an optimal semi-matching. Thus, algorithms for the weighted assignment problem can solve the optimal semi-matching problem.

The min-cost max-flow problem is one of the most important combinatorial optimization problems; its objective is to find a minimum-cost maximum-flow in a network [2]. Indeed, the weighted assignment problem can be reduced to min-cost max-flow problem. Thus, from the above discussion, it should be clear that a semi-matching problem instance can be recast as a min-cost max-flow problem. We now describe an alternative, more compact, transformation of the optimal semi-matching problem to a min-cost max-flow problem.

Given $G = (U \cup V, E)$, a bipartite graph giving an instance of a semi-matching problem, we show how to construct a network N such that a min-cost max-flow in N determines an optimal semi-matching in G. The network N is constructed from G by adding at most $|U| + 2$ vertices and $2|U| + |E|$ edges (see Figure 1). The additional vertices are a source, s, a sink, t, and a set of "cost centers" $C = \{c_1, \dots, c_\Delta\}$, where $\Delta \leq |U|$ is the maximal degree of any V-vertex. Edges with cost 0 and capacity 1 connect s to each of the vertices in U. The original edges connecting U and V are directed from U to V and are given cost 0 and capacity 1. For each $v \in V$, v is connected to cost centers $c_1, \dots, c_{deg(v)}$ with edges of capacity 1 and costs $1, 2, \dots, deg(v)$ respectively. Edges with cost 0 and infinite capacity connect each of the cost centers to the sink, t.

3 Properties of Optimal Semi-Matchings

This section presents various important properties of optimal semi-matchings. Section 3.1 characterizes when a semi-matching is optimal. Section 3.2 states that an optimal semi-matching always contains a maximum matching and discusses various consequences. Section 3.3 states that an optimal semi-matching is also optimal with respect to any L_p-norm and the L_∞-norm.

3.1 Characterization of Optimal Semi-Matchings

An important theorem from network flow theory is that a maximum flow has minimum cost if and only if no negative-cost cycle exists [2, Theorem 3.8]. We now prove an analogous result for semi-matchings. In Section 5 we describe the Algorithm $\mathcal{A}_{SM2}$ which is based on this property.

Theorem 1. *A semi-matching M is optimal if and only if no cost-reducing path relative to M exists.*

Proof. Let G be an instance of a semi-matching problem, and let M be a semi-matching in G. Clearly, if M is optimal then no cost-reducing path can exist. We show that a cost-reducing path must exist if M is not optimal.

Let O be an optimal semi-matching in G, chosen such that the symmetric difference $O \oplus M = (O \setminus M) \cup (M \setminus O)$ is minimized. Assume that M is not optimal, implying that M has greater total cost than O: i.e., $T(O) < T(M)$. Recall that $deg_O(v)$ and $deg_M(v)$ denote the number of U-vertices matched with v by O and M respectively. Let G_d be the subgraph of G induced by the edges of $O \oplus M$. Color with green the edges of $O \setminus M$ and with red the edges of $M \setminus O$. Direct the green edges from U to V and the red edges from V to U. We will use the following property of G_d (proof omitted).

Claim 1 *The graph G_d is acyclic, and for every directed path P in G_d from $v_1 \in V$ to $v_2 \in V$, we have $deg_O(v_2) \leq deg_O(v_1)$.*

Both O and M are semi-matchings, implying that $\sum_v deg_O(v) = \sum_v deg_M(v) = |U|$. Since $T(O) < T(M)$, there must exist $v_1 \in V$ such that $deg_M(v_1) > deg_O(v_1)$. Starting from v_1, we build an alternating red-green path, P', as follows. (1) From an arbitrary vertex $v \in V$, if $deg_{M \setminus O}(v) \geq 1$ and $deg_M(v) \geq deg_M(v_1) - 1$, we build P' by following an arbitrary red edge directed out from v. (2) From an arbitrary vertex $u \in U$, we build P' by following the single green edge directed out from u. (3) Otherwise, we stop.

By Claim 1, G_d is acyclic and therefore P' is well-defined and finite. Let $v_2 \in V$ be the final vertex on the path. There are two cases.

(1) $deg_M(v_2) < deg_M(v_1) - 1$: Thus P' is a cost-reducing path relative to M.

(2) $deg_{M \setminus O}(v_2) = 0$. In this case, we know that $deg_M(v_2) < deg_O(v_2)$ since P' arrived at v_2 via a green edge. By Claim 1, we must also have that $deg_O(v_2) \leq deg_O(v_1)$. Finally, recall that v_1 was chosen such that $deg_O(v_1) < deg_M(v_1)$. Combining these three inequities yields: $deg_M(v_2) < deg_O(v_2) \leq deg_O(v_1) < deg_M(v_1)$. This implies that $deg_M(v_2) < deg_M(v_1) - 1$, and so P' is a cost-reducing path relative to M.

Since P' is a cost-reducing path relative to M in both cases, the proof is complete.

3.2 Optimal Semi-Matchings Contain Maximum Matchings

In this section, we state, omitting the proof, that every optimal semi-matching must contain a maximum bipartite matching; furthermore, it is a simple process to find these maximum matchings. Thus, the problem of finding optimal semi-matchings indeed generalizes the problem of finding maximum matchings.

Theorem 2. *Let M be an optimal semi-matching in G. Then there exists $S \subseteq M$ such that S is a maximum matching in G.*

We note that the converse of this theorem is not true: Not every maximum matching can be extended to an optimal semi-matching.

Corollary 1. *Let M be an optimal semi-matching in G. Define $f(M)$ to be the number of right-hand vertices in G that are incident with at least one edge in M. Then the size of a maximum matching in G is $f(M)$.*

In particular, if G has a perfect matching and M is an optimal semi-matching in G then M is a perfect matching. Corollary 1 yields a simple algorithm for computing a maximum matching from an optimal semi-matching, M: For each $v \in V$, if $deg_M(v) > 1$, select one arbitrary edge from M that is incident with v.

3.3 Optimality with Respect to L_p- and L_∞- norm

Let $x_i = deg_M(v_i)$ denote the load on machine i (i.e., the number of tasks assigned to machine i). The L_p-norm of the vector $X = (x_1, \ldots, x_{|V|})$ is $||X||_p = (\sum_i x_i^p)^{1/p}$. The following theorem states that an optimal semi-matching is optimal with respect to the L_p-norm of the vector X for any finite p; in other words, optimal semi-matchings minimize $||X||_p$. (Note that $||X||_1 = |U|$ for all semi-matchings, so all semi-matchings are optimal with respect to the L_1-norm).

Theorem 3. *Let $2 \leq p < \infty$. A semi-matching has optimal total cost if and only if it is optimal with respect to the L_p-norm of its load vector.*

Another important optimization objective in practice is minimizing the maximal load on any machine; this is achieved by minimizing the L_∞-norm of the machines' load vector X. The following theorem states that optimal semi-matchings do minimize the L_∞-norm of X, and thus are an "ultimate" solution that simultaneously minimizes both the variance of the machines' load (from the L_2-norm) and the maximal machine load (given by the L_∞-norm).

Theorem 4. *An optimal semi-matching is also optimal with respect to L_∞.*

The converse of Theorem 4 is not valid; that is, minimizing the L_∞-norm does not imply minimization of other L_p-norms.

4 $\mathcal{A}_{SM1}$: An $O(|U||E|)$ Algorithm for Optimal Semi-Matchings

In this section we present our first algorithm, $\mathcal{A}_{SM1}$, for finding an optimal semi-matching. The time complexity of $\mathcal{A}_{SM1}$ is $O(|U||E|)$, which is identical to that of the Hungarian algorithm [14] [15, Section 5.5] for finding maximum bipartite matchings. Indeed, $\mathcal{A}_{SM1}$ is merely a simple modification of the Hungarian algorithm, as we explain below.

The Hungarian algorithm for finding maximum bipartite matchings considers each left-hand vertex u in turn and builds an alternating search tree, rooted at u, in order to find an unmatched right-hand vertex (i.e., a vertex $v \in V$ with $deg_M(v) = 0$). If such a vertex v is found, the matching and non-matching edges along the u-v path are switched so that u and v are no longer unmatched.

Similarly, $\mathcal{A}_{SM1}$ maintains a partial semi-matching M, starting with the empty set. In each iteration, it considers a left-hand vertex u and builds an alternating search tree rooted at u, looking for a right-hand vertex v such that $deg_M(v)$ is as small as possible. To build the tree rooted at u we perform a directed breadth-first search in G starting from u, where edges in M are directed from V to U and edges not in M are directed from U to V. We select in this tree a path P from u to a least loaded V-vertex reachable from u. We increase the size of M by forming $P \oplus M$; that is, we add to the matching the first edge in this path, and switch matching and non-matching edges along the remainder of the path. As a result, u is no longer unmatched and $deg_M(v)$ increases by 1.

We repeat this procedure of building a tree and extending the matching accordingly for all of the vertices in U. Since each iteration matches a vertex in U with a single vertex in V and does not change $deg_M(u)$ for any other $u \in U$, the resulting selection of edges is indeed a semi-matching.

Theorem 5. *Algorithm $\mathcal{A}_{SM1}$ produces an optimal semi-matching.*

Proof. We show that no cost-reducing path is created during the execution of the algorithm. In particular, no cost reducing path exists at the end of the execution; thus, by Theorem 1, the resulting matching is optimal.

Assume the opposite and let $P^* = (v_1, u_1, \ldots, v_{k-1}, u_{k-1}, v_k)$, be the *first* cost-reducing path created by $\mathcal{A}_{SM1}$. Let M be the partial semi-matching after the iteration in which P^* is created. Thus, $deg_M(v_1) > deg_M(v_k) + 1$. Without loss of generality (by taking a sub-path of P^*), we can assume that there exists some x such that $deg_M(v_1) \geq x + 1$, $deg_M(v_i) = x \; \forall i \in \{2, \ldots, k-1\}$, and $deg_M(v_k) \leq x - 1$. Let u' be the U-vertex added to the assignment during the previous iteration in which the load on v_1 increased. The algorithm gives that v_1 is a least-loaded V-vertex reachable from u'; thus, the search tree built for u' includes only V-vertices with load at least x; thus v_k is not reachable from u'.

Given that the path P^* exists, at some iteration occurring after the one in which u' is added, all the edges (u_i, v_i) of P^* are in the matching. Let u^* be the U-vertex, added after u', whose addition to the assignment creates P^*. The following claims yield a contradiction in the way u^* is assigned.

Claim 2 *When adding u^*, the load on v_k is at most $x - 1$ and v_k is in the tree rooted at u^*.*

Claim 3 *When adding u^*, the load on some vertex with load at least x increases.*

Claims 2 and 3 contradict the execution of $\mathcal{A}_{SM1}$, and therefore P^* cannot exist.

To bound the runtime of $\mathcal{A}_{SM1}$, observe that there are exactly $|U|$ iterations. Each iteration requires at most $O(|E|)$ time to build the alternating search tree and at most $O(\min\{|U|, |V|\})$ time to switch edges along the alternating path. Thus the total time required is at most $O(|U||E|)$.

5 $\mathcal{A}_{SM2}$: An Efficient, Practical Algorithm

We present $\mathcal{A}_{SM2}$, our second algorithm for finding optimal semi-matchings. Our analysis of its runtime gives an upper bound of $O(\min\{|U|^{3/2}, |U||V|\} \cdot |E|)$, which is worse than the bound of $O(|U||E|)$ for algorithm $\mathcal{A}_{SM1}$. However, our analysis for $\mathcal{A}_{SM2}$ is loose; in practice, $\mathcal{A}_{SM2}$ performs much better than $\mathcal{A}_{SM1}$, as our experiments in Section 6 show.

Theorem 1 proves that a semi-matching is optimal if and only if the graph does not contain a cost-reducing path. $\mathcal{A}_{SM2}$ uses that result to find an optimal semi-matching as follows:

Overview of $\mathcal{A}_{SM2}$

```
Find an initial semi-matching, M.
While there exists a cost-reducing path, P
   Use P to reduce the cost of M.
```

Since the cost can only be reduced a finite number of times, this algorithm must terminate. Moreover, if the initial assignment is nearly optimal, the algorithm terminates after few iterations.

Finding an Initial Semi-Matching: The first step of algorithm $\mathcal{A}_{SM2}$ is to determine an initial semi-matching, M. Our experiments have shown that the following greedy algorithm works well in practice. First, the U-vertices are sorted by increasing degree. Each U-vertex is then considered in turn, and assigned to a V-neighbor with least load. In the case of a tie, a V-neighbor with least degree is chosen. The purpose of considering vertices with lower degree earlier is to allow more constrained vertices (i.e., ones with fewer neighbors) to "choose" their matching vertices first. The same rule of choosing the least loaded V-vertex is also commonly used in the online case [3]. However, in the online case it is not possible to sort the U-vertices or to know the degree of the V-vertices in advance.

The total time required to find this initial matching is $O(|E|)$, since every edge is examined exactly once, and the sorting can be done using bucket sort.

Finding Cost-Reducing Paths: The key operation of the $\mathcal{A}_{SM2}$ algorithm is the method for finding cost-reducing paths. As a simple approach, one may determine if a particular vertex $v \in V$ is the ending vertex of a cost-reducing path simply by growing a tree of alternating paths rooted at v. As a better approach, one may determine if *any* $v \in V$ is the ending vertex of a cost-reducing path in $O(|E|)$ time. To do this, simply grow a depth-first search (DFS) forest of alternating paths where each tree root is chosen to be an unused V-vertex with lowest load. To find such a vertex, the V-vertices are maintained sorted by their load in an array of $|U| + 1$ buckets.

Analysis of $\mathcal{A}_{SM2}$: As argued earlier, the initial matching can be found in $O(|E|)$ time. Following this initial step, we iteratively find and remove cost-reducing paths. Identifying a cost-reducing path or lack thereof requires $O(|E|)$ time since it performs a depth-first search over all of G. If a cost-reducing path has been identified, then we switch matching and non-matching edges along that path, requiring $O(\min\{|U|, |V|\}) = O(|E|)$ time. Thus, the runtime of $\mathcal{A}_{SM2}$ is $O(I \cdot |E|)$, where I is the number of iterations needed to achieve optimality.

It remains to determine how many iterations are required. A simple bound of $I = O(|U|^2)$ may be obtained by observing that the worst possible initial matching has cost at most $O(|U|^2)$ and that each iteration reduces the cost by at least 1. The following theorem gives an improved bound.

Theorem 6. *$\mathcal{A}_{SM2}$ requires at most $O(\min\{|U|^{3/2}, |U||V|\})$ iterations.*

Remark 1. For graphs in which the optimal semi-matching cost is $O(|U|)$, the running time of $\mathcal{A}_{SM2}$ is $O(|U||E|)$. This bound holds since Awerbuch et al. [3] show that the cost of the greedy initial assignment is at most $4 \cdot T(M_{OPT})$; thus $\mathcal{A}_{SM2}$ needs at most $O(|U|)$ iterations to achieve optimality.

Practical Considerations: The description of $\mathcal{A}_{SM2}$ given above suggests that each iteration builds a depth-first search forest and finds a single cost-reducing path. In practice, a single DFS forest often contains numerous vertex-disjoint cost-reducing paths. Thus, our implementation repeatedly performs linear-time *scans* of the graph, growing the forest and removing cost-reducing paths. We repeatedly scan the graph until a scan finds no cost-reducing path, indicating that optimality has been achieved.

Our bound of $O(\min\{|U|^{3/2}, |U||V|\})$ iterations is loose: experiments show that much fewer iterations are required in practice. We were able to create "bad" graphs, in which the number of iterations needed is $\Omega(|U|^{3/2})$; however, most of the cost-reducing paths in these graphs are very short, thus each iteration takes roughly constant time. While our bound for $\mathcal{A}_{SM2}$ is worse than our bound for $\mathcal{A}_{SM1}$, we believe that the choice of $\mathcal{A}_{SM2}$ as the best algorithm is justified already by its actual performance, as described in the next section.

Variants of $\mathcal{A}_{SM2}$, in which each iteration seeks a cost-reducing path with some property (such as "maximal difference in load between first and last vertex"), will also result in an optimal semi-matching. It is unknown whether such algorithms yield a better analysis than $\mathcal{A}_{SM2}$, or whether each iteration of such algorithms can be performed quickly in practice.

6 Experimental Evaluation

We implemented a program to execute $\mathcal{A}_{SM1}$, $\mathcal{A}_{SM2}$ and various known algorithms on a variety of "benchmark" input graphs. All input graphs were created by the bipartite graph generators used in [8]. Our simulation program was implemented in C and run on a Compaq Evo D500 machine with a 2.2GHz Pentium 4 CPU and 512MB of RAM.

First, we compared $\mathcal{A}_{SM1}$ and $\mathcal{A}_{SM2}$ with known techniques for computing optimal semi-matchings based on the transformation to the assignment problem. To solve the assignment problem, we used two available algorithms: CSA [10], and LEDA [16]. For the CSA algorithm, the transformed graph was augmented with additional vertices and edges to satisfy CSA's requirement that a perfect assignment exist[1]. Table 1(a) shows the results of these experiments on graphs with 2^{16} vertices. The Zipf graphs (after being transformed to the assignment problem) exceeded the memory on our test machine, and no reasonable results could be recorded. Table 1(a) reports the elapsed execution time of these algorithms, excluding the time to load the input data. The reported value is the mean over five execution runs, each using a different seed to generate the input graph. These results show that $\mathcal{A}_{SM2}$ is much more efficient than assignment algorithms for the optimal semi-matching problem on a variety of input graphs.

Next, we compared $\mathcal{A}_{SM2}$ with two algorithms for computing maximum bipartite matchings from [8]: BFS, their fastest implementation based on augmenting paths, and LO, their fastest implementation based on the push-relabel method. For this series of experiments, we consider only graphs with 2^{19} vertices. As before, the reported value is the mean of the execution time over five runs; these results are shown in Table 1(b). These results show that $\mathcal{A}_{SM2}$ is roughly as efficient as the best known algorithm for the maximum bipartite matching problem on a variety of input graphs.

References

1. Active Directory. `http://www.microsoft.com/windowsserver2003/technologies`.
2. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.
3. B. Awerbuch, Y. Azar, E. Grove, M. Y. Kao, P. Krishnan, and J. S. Vitter. Load Balancing in the L_p Norm. In *Proceedings of FOCS*, 1995.
4. Y. Azar. On-line Load Balancing. In A. Fiat and G. Woeginger, editors, *Online Algorithms: The State of the Art (LNCS 1442)*, chapter 8. Springer-Verlag, 1998.
5. Y. Azar, A. Z. Broder, and A. R. Karlin. On-line load balancing. *Theoretical Computer Science*, 130(1):73–84, 1994.
6. Y. Azar, J. Naor, and R. Rom. The Competitiveness of On-line Assignments. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1992.
7. J. L. Bruno, E. G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[1] We acknowledge Andrew Goldberg's assistance in finding such a transformation with a linear number of additional vertices and edges.

Graph	$\mathcal{A}_{SM1}$	$\mathcal{A}_{SM2}$	LEDA	CSA
FewG	1.834	0.337	30.625	1.274
Grid	0.672	0.131	6.850	1.310
Hexa	1.521	0.319	28.349	2.131
Hilo	0.650	0.299	11.141	2.968
ManyG	1.669	0.200	18.388	1.238
Rope	0.269	0.188	7.588	1.330
Zipf	6.134	0.156	—	—
Total	12.749	1.630	>102.941	> 10.251

(a)

Graph	$\mathcal{A}_{SM2}$	BFS	LO
FewG	3.563	15.018	2.085
Grid	0.545	4.182	1.140
Hexa	3.569	13.990	1.755
Hilo	2.942	3.047	6.559
ManyG	3.607	13.640	2.199
Rope	1.308	2.459	1.400
Zipf	1.105	0.375	0.938
Total	16.639	52.711	16.076

(b)

Table 1. *(a) gives the execution time in seconds of four algorithms for the optimal semi-matching problem, on a variety of graphs with 65,536 vertices. "—" indicates that no results could be recorded since the graph exceeded the memory of our test machine. (b) gives the execution time in seconds of three algorithms for the maximum bipartite matching problem, on a variety of graphs with 524,288 vertices.*

8. 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. *ACM J. Exp. Algorithmics*, 3(8), 1998.
   Source code available at `http://www.avglab.com/andrew/soft.html`.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
10. A. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Prog.*, 71:153–178, 1995.
    Source code available at `http://www.avglab.com/andrew/soft.html`.
11. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Math*, 5:287–326, 1979.
12. J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 2:225–231, 1973.
13. W. A. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21:846–847, 1973.
14. H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.
15. E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover, 2001.
16. LEDA. `http://www.algorithmic-solutions.com/`.