Semi-Matchings for Bipartite Graphs and Load Balancing

Nicholas J. A. Harvey [a] Richard E. Ladner [b] László Lovász [c] Tami Tamir [d]

[a] *MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA. E-mail:* `nickh@mit.edu`

[b] *Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA. E-mail:* `ladner@cs.washington.edu`

[c] *Microsoft Research, Redmond, WA, USA. E-mail:* `lovasz@microsoft.com`

[d] *Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA. E-mail:* `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 *optimal 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.

Key words: bipartite graphs, load-balancing, matching algorithms, optimal algorithms, semi-matching

1 Introduction

One of the classical combinatorial optimization problems is finding a maximum matching in a bipartite graph. The bipartite matching problem has numerous practical applications [1, Section 12.2], and many efficient, polynomial time algorithms for computing solutions [2] [3] [4]. 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. In other words, 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, 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. Semi-matchings were previously considered in [5], with the objective of finding semi-matchings of maximum weight. We consider a different optimization objective: finding 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 [6]. 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 [7], 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. Servers in the branch offices must periodically *replicate* with the servers in the hub, which means to exchange modified data between the servers to ensure that all servers have a consistent copy of the database. However, there are constraints on which hub servers a given branch server may replicate with. The database is commonly partitioned according to corporate divisions, such as "American Division" and "European Division". Each branch server must replicate with any hub server that also stores the branch server's database partition. Thus, the assignment of branch servers to hub servers for the purpose of replication may be viewed as a constrained load balancing problem: the branch servers are the "tasks", the hub servers are the "machines", and the database partitions induce the constraints. Since servers are only rarely added or removed, and servers can be efficiently reassigned to replicate with another server, this load balancing problem is solvable by 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 proves 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.

2 Preliminaries

2.1 Definitions and Problem Statement

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 frequently refer to the U-vertices as "tasks" and to the V-vertices as "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. Note that isolated U-vertices can not be matched with any V-vertex, therefore we assume that all of the vertices in U have degree at least 1. A semi-matching gives an assignment of each task to a machine that is capable of processing 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

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

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 $T(M) = \sum_{i=1}^{m} cost_M(v_i)$; this expression gives the flow time of the tasks' schedule [8]. A semi-matching with minimum 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 the minimizing the maximal machine load.

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. For convenience, we sometimes treat paths as 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. This switching process decreases the load on v_1, increases the load on v_k, and does not affect the load on any other v_i. 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,

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

2.2 Related Work

The maximum bipartite matching problem is known to be solvable in polynomial time using a reduction from maximum flow [1] [9] or by the Hungarian method [4] [5, Section 5.5]. Push-relabel algorithms are widely considered to be the fastest algorithms in practice for this problem [2].

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 2.3. In standard scheduling notation [10], 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 [1, Application 12.9] [11] [8]; 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 [12] [13] [14]. A comprehensive survey of the various models, and different objective functions is given in [6].

Optimal semi-matchings were concurrently and independently studied by Abraham [15, Section 4], under the name "balanced matchings". His work contains a proof of our Theorem 13 and presents an algorithm for finding optimal semi-matchings. His algorithm differs from our algorithms $\mathcal{A}_{SM1}$ and $\mathcal{A}_{SM2}$, but has theoretical performance similar to that of $\mathcal{A}_{SM1}$.

2.3 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.

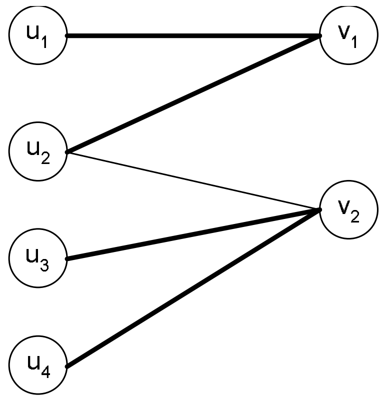
Recall that the scheduling problem $R \mid \mid \sum_j C_j$, and in particular the case in which $p_{i,j} \in \{1, \infty\}$, can be solved optimally [11] [8]. We now show that the optimal semi-matching problem can be reduced to the scheduling problem $R \mid p_{i,j} \in \{1, \infty\} \mid \sum_j C_j$. A semi-matching instance can be represented as a scheduling problem 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, an algorithms for $R \mid\mid \sum_j C_j$ can solve the optimal semi-matching problem. We note that current optimal algorithms for $R \mid\mid \sum_j C_j$ are using as a subroutine a weighted assignment algorithm for a bipartite graph having $n + nm$ vertices [11] [8].

The min-cost max-flow problem is one of the most important combinatorial optimization problems [1]. 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.

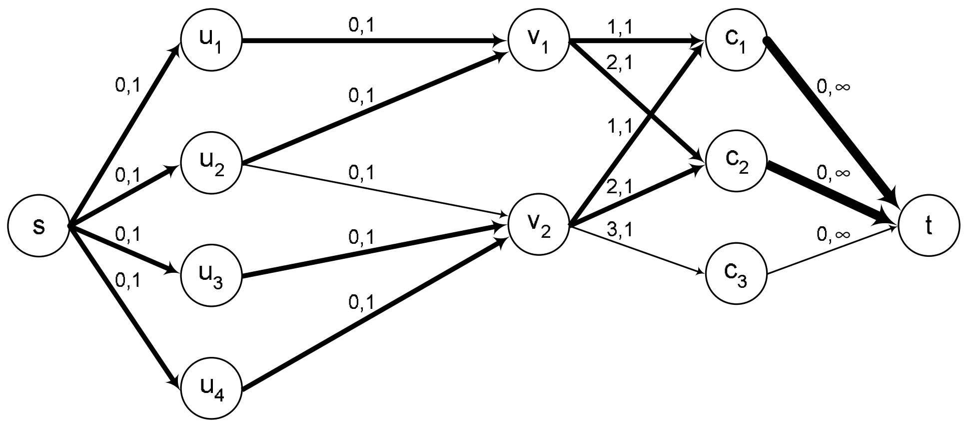
Given $G = (U \cup V, E)$, a bipartite graph giving an instance of a semi-matching problem, we show how to construct a network N such that a min-cost max-flow in N determines an optimal semi-matching in G. The network N is constructed from G by adding at most $|U| + 2$ vertices and $2|U| + |E|$ edges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

We now prove the validity of this reduction.

Lemma 1 *If f is a maximum flow in N then f determines a semi-matching*



(a)



(b)

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.*
in G.

PROOF. We may assume that f is integral since standard flow algorithms always return integral solutions. Since there are no isolated U-vertices, the n edges connecting s and U are a minimum cut in the network N. Since all edges from s to U have unit capacity, the total capacity of this cut is n. When this cut is saturated, each U-vertex must send one unit of flow to a single V-vertex, since the flow is integral. Thus the flow matches each U-vertex with a V-vertex. Since the value of the maximum flow is equal to the capacity of the minimum cut, this cut is saturated in any maximum flow, and therefore the flow in f induces a semi-matching.

Lemma 2 *If M is a semi-matching in G, there exists a corresponding maximum flow f in N with equal cost.*

PROOF. The flow f is defined as follows: for each $u \in U$, one flow-unit is sent from s to u and then to the V-vertex to which u is matched. For each $v \in V$, each flow unit arriving at v is sent to the available edge of least cost connecting v with a cost-center, and then to t. By definition of the edges from V to C, the total cost of the flow on those edges is $\sum_{i=1}^{m} \sum_{j=1}^{deg_M(v_i)} j$. Since all other edges have zero cost, this is also the cost of the whole flow.

In particular, if the optimal semi-matching has cost C_{opt} then the min-cost max-flow in N has cost C_{opt}. We now extend Lemma 1 to show that a min-cost max-flow in N determines a semi-matching with the same cost.

Lemma 3 *If f_{opt} is a minimum-cost maximum-flow in N and M is the corresponding semi-matching in G, then the cost of f_{opt} equals the total cost of M.*

PROOF. As before, we assume that f_{opt} is integral. In any minimum-cost flow, the flow-units leaving each $v \in V$ must be sent to the cost-centers using edges with least cost. That is, if $f(v_i)$ is the total amount of flow leaving v_i then the flow is sent to cost-centers $c_1, \ldots, c_{f(v_i)}$. Since all the other edges in N have cost zero, the total cost of the flow is $\sum_{i=1}^{m} \sum_{j=1}^{f(v_i)} j$. In the matching M induced by f_{opt}, each $u \in U$ is matched to a $v_i \in V$ if and only if there is one flow unit on the edge (u, v_i). Therefore, $f(v_i) = deg_M(v_i)$, and the total cost of f_{opt} is exactly $T(M)$.

3 Properties of Optimal Semi-Matchings

This section proves various important properties of optimal semi-matchings. Section 3.1 characterizes when a semi-matching is optimal. Section 3.2 generalizes the notion of optimality to other cost functions. Section 3.3 proves that an optimal semi-matching always contains a maximum matching and discusses various consequences. Section 3.4 proves 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 [1, 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 4 *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.

Claim 5 *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 $deg_O(v_2) \leq deg_O(v_1)$.*

PROOF. Let $P = (v_1, \ldots, v_2)$ be a directed path in G_d. By the choice of directions for the edges, P must consist of alternating red and green edges. If $deg_O(v_2) > deg_O(v_1) + 1$ then P (in reverse order) is a cost-reducing path for O, in contradiction to the optimality of O. If $deg_O(v_2) = deg_O(v_1) + 1$, then by alternating the assignment of the U-vertices along P we get an optimal assignment with smaller symmetric difference from M, again in contradiction to the choice of O. Therefore, we must have that $deg_O(v_2) \leq deg_O(v_1)$. A similar argument shows that G_d is also acyclic.

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. Such a green edge must always exist. (3) Otherwise, we stop.

By Claim 5, 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.

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

Case 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 5, 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 Optimality with Respect to Other Cost Functions

So far we have considered only the objective of minimizing $T(M) = \sum_{i=1}^{m} cost_M(v_i)$, where $cost_M(v) = \sum_{i=1}^{deg_M(v)} i = \frac{(deg_M(v)+1)deg_M(v)}{2}$. Recall that this expression gives the flow time of the tasks' schedule. In this section we show that in fact our results capture a wider set of objectives, given by different cost functions. We show that an optimal semi-matching is optimal with respect to any of these cost functions. In other words, an optimal semi-matching minimizes flow time as well as many other important properties of the system.

Definition 6 *A* cost function *for a matching M is a function $cost_f(M) = \sum_{i=1}^{m} f(deg_M(v_i))$, where $f : \mathbb{R}^+ \to \mathbb{R}$ is a strictly convex function. If f is weakly convex then $cost_f(M)$ is called a* weak cost function.

For example, $T(M)$ is the cost function induced by the function $f(x) = \sum_{j=1}^{x} j = x(x+1)/2$.

Lemma 7 *Let P be a cost-reducing path. For any cost function $cost_f$, switching matching and non-matching edges along P yields a semi-matching $M' = P \oplus M$ such that $cost_f(M') < cost_f(M)$. If $cost_f$ is a weak cost function then $cost_f(M') \leq cost_f(M)$.*

PROOF. First, we need the following property of convex functions:

Claim 8 *If f is strictly convex then $x > y$ implies $f(x+1) - f(x) > f(y+1) - f(y)$. If f is weakly convex then the inequality holds but is not strict.*

PROOF. A property of strictly convex functions is that $\frac{f(b)-f(a)}{b-a} < \frac{f(c)-f(a)}{c-a} < \frac{f(c)-f(b')}{c-b'}$ if $a < b < c$ and $a < b' < c$ (see, for example, [16]). Setting $a = y$, $b = y+1$, $b' = x$ and $c = x+1$ we obtain that $f(y+1) - f(y) < \frac{f(x+1)-f(y)}{x+1-y} < f(x+1) - f(x)$. If f is weakly convex then all stated inequalities hold but are not strict.

Next, consider the matching M and the cost-reducing path $P = (v_1, \ldots, v_k)$. Let $x = deg_M(v_1) - 1$ and $y = deg_M(v_k)$. The change in $cost_f$ at v_1 due to switching edges along path P is $f(x) - f(x+1)$. Similarly, the change in $cost_f$ at v_k is $f(y+1) - f(y)$. Therefore

$$cost_f(M') = cost_f(M) - \Big(f(x+1) - f(x)\Big) + \Big(f(y+1) - f(y)\Big).$$

Applying Claim 8 completes the proof.

Using Lemma 7 we can extend Theorem 4 as follows.

Theorem 9 *Let M be a semi-matching. If $cost_f$ is a strict cost function and if M is optimal with respect to $cost_f$ then no cost-reducing path relative to M exists. The converse holds even if $cost_f$ is a weak cost function: if no cost-reducing path relative to M exists then M is optimal with respect to $cost_f$.*

PROOF. Identical to the proof of Theorem 4.

Thus, if M is optimal with respect to a strict cost function then no cost-reducing path relative to M exists. Also, if M has no cost-reducing paths then M is optimal with respect to every cost function.

Corollary 10 *A semi-matching M that is optimal with respect to any strict cost function is optimal with respect to* every *cost function, strict or weak.*

3.3 Optimal Semi-Matchings Contain Maximum Matchings

In this section, we prove 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 11 *Let M be an optimal semi-matching in G. Then there exists $S \subseteq M$ such that S is a maximum matching in G.*

PROOF. Given a semi-matching M (not necessarily an optimal one), we may construct an ordinary matching (not necessarily a maximum one) by selecting one incident M-edge for each vertex $v \in V$. If there are ℓ vertices in V with $deg_M(v) > 0$, then the resulting matching is of size ℓ. Conversely, any matching can be extended to a semi-matching (not necessarily an optimal one) by adding one arbitrary incident edge for any unmatched u in U. Thus, any matching, and in particular any maximum one, is contained in some semi-matching. In order to show that an optimal semi-matching contains a maximum matching, we show that an optimal semi-matching minimizes the number of unmatched V-vertices (i.e., vertices $v \in V$ with $deg_M(v) = 0$).

Define $f : \mathbb{R}^+ \to \mathbb{R}$ such that $f(x) = 0$ for $x \leq 1$ and $f(x) = x - 1$ for $x > 1$. Note that f is weakly convex so $cost_f$ is a weak cost function. Let M be an optimal semi-matching (i.e., with respect to $T(M)$). M is therefore optimal with respect to $cost_f$ by Corollary 10. $cost_f$ is defined such that if $deg_M(v) > 0$ then v contributes $deg_M(v) - 1$ to the total cost. Summing over all $v \in V$, $cost_f(M) = n - \ell$, where ℓ is the number of vertices in V with at least one incident edge in M. Since M minimizes $cost_f$, it maximizes ℓ. By our previous remarks, M contains a matching of size ℓ, so M contains a maximum matching.

The converse of this theorem is not true. As demonstrated in Appendix C, not every maximum matching can be extended to an optimal semi-matching.

Corollary 12 *Let M be an optimal semi-matching in G. Define $\alpha(M)$ 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 $\alpha(M)$.*

In particular, if G has a perfect matching and M is an optimal semi-matching in G, then M is a perfect matching. Corollary 12 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.

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

Let $x_i = deg_M(v_i)$ denote the load on machine i, that is, the number of tasks assigned to machine i. The L_p-norm of the vector $X = (x_1, \ldots, x_m)$ is $\|X\|_p = (\sum_{i=1}^{m} x_i^p)^{1/p}$. In this section we show 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 $\|X\|_p$ of the load vector. (All semi-matchings have $\|X\|_1 = |U|$, and hence are optimal with respect to the L_1-norm).

Theorem 13 *Let $1 < p < \infty$. A semi-matching is optimal if and only if it is optimal with respect to the L_p-norm of its load vector.*

PROOF. Fix any $p > 1$ and define $f_p(x) = x^p$. Note that f_p is strictly convex so $cost_{f_p}$ is a strict cost function. Let X be the load vector for M. Since $\|X\|_p = cost_{f_p}(M)^{1/p}$, M optimizes $\|X\|_p$ if and only if it optimizes $cost_{f_p}$. By Corollary 10, M is optimal with respect to $cost_{f_p}$ if and only if M is an optimal semi-matching.

Another common measure for load balancing is the *variance of the loads* on the machines. The following theorem shows that optimal semi-matchings are also optimal for this objective.

Theorem 14 *A semi-matching M is optimal if and only if $\mathrm{Var}_{v \in V}\{deg_M(v)\}$ is minimized.*

PROOF. Let $\mu = n/m$ be the mean of $deg_M(v)$ where $v \in V$. Define $f_{\mathrm{Var}}(x) = (x - \mu)^2$ and note that $cost_{f_{\mathrm{Var}}}$ is a strict cost function. By Corollary 10, M is optimal with respect to $cost_{f_{\mathrm{Var}}}$ if and only if M is an optimal semi-matching.

Another important optimization objective in practice is minimizing the makespan, which is the maximal load on any machine. This is achieved by minimizing the L_∞-norm of the machines' load vector X. As we show below, optimal semi-matchings do minimize the L_∞-norm of X, and thus are an "ultimate" solution that simultaneously minimizes both the flow time and the makespan. Recall that $|V| = m$ and consider the function f_∞ defined by $f_\infty(x) = (m+1)^x$. Note that $cost_{f_\infty}$ is a strict cost function.

Claim 15 *A semi-matching that minimizes $cost_{f_\infty}$ minimizes the maximal load on a single V-vertex.*

PROOF. Let M be an assignment that minimizes $cost_{f_\infty}$. Assume that M does not minimize the maximal load on a single V-vertex, and that this objective is achieved by a different assignment M'. Let Δ and Δ' be the maximal load on a single V-vertex in M and M' respectively. Thus, $cost_{f_\infty}(M) \geq (m+1)^\Delta$ and $cost_{f_\infty}(M') \leq m(m+1)^{\Delta'}$ (at worst, all m of the V-vertices have load Δ'). However, since $\Delta' \leq \Delta - 1$, it follows that $cost_{f_\infty}(M') < cost_{f_\infty}(M)$, which is a contradiction.

Theorem 16 *An optimal semi-matching M is also optimal with respect to the L_∞-norm.*

PROOF. Since $cost_{f_\infty}$ is a cost function, Corollary 10 implies that M is optimal with respect to $cost_{f_\infty}$. By Claim 15, this implies that M minimizes the maximal load on a V-vertex. Therefore M is optimal with respect to the L_∞-norm.

The converse of Theorem 16 is not valid; that is, minimizing the L_∞-norm does not imply minimization of other cost functions. The converse of Theorem 16 fails to hold because the converse of Claim 15 does not hold: minimizing the makespan does not necessarily minimize $cost_{f_\infty}$. A semi-matching that minimizes *only* the L_∞-norm is given in Appendix C.

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 [4] [5, 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, looking for an unmatched right-hand vertex (i.e., a vertex $v \in V$ with $deg_M(v) = 0$). If an unmatched right-hand 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 $P \oplus M$; in other words, we add to the matching the first edge in this path, and switch the next 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. The pseudocode of this algorithm is given in Appendix A.

Interestingly, there is another way to characterize $\mathcal{A}_{SM1}$. Section 2.3 showed that G can be represented as a network N in which a min-cost max-flow corresponds to an optimal semi-matching. One way to compute a min-cost max-flow in N is with the successive shortest path algorithm [1, Section 9.7]. Regarding G as a compact representation of N, $\mathcal{A}_{SM1}$ is an efficient implementation of the successive shortest path algorithm on G.

Theorem 17 *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 4 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 this algorithm. Let M_1 be the matching after the iteration in which P^* is created. Thus, $deg_{M_1}(v_1) > deg_{M_1}(v_k) + 1$. Without loss of generality (by taking a sub-path of P^*), we can assume that there exists some x such that $deg_{M_1}(v_1) \geq x+1,\ deg_{M_1}(v_i) = x, \forall i \in \{2, \ldots, k-1\}$, and $deg_{M_1}(v_k) \leq x-1$. Consider the last iteration in which the load on v_1 is increased. Let u' be the U-vertex added to the assignment at this iteration. By the definition of the algorithm, 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; in particular, 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^*. We show a contradiction to the way u^* is assigned. Specifically, we show that when adding u^*, the algorithm increases the load on some vertex with load at least x, while v_k, whose load at that time is at most $x-1$, is also reachable from u^*.

Claim 18 *When u^* is added, the load on v_k is at most $x-1$ and v_k is in the tree rooted at u^*.*

PROOF. The load on any V-vertex can only increase during the algorithm. Since the load on v_k in M_1 is $x-1$, and u^* is added before P^* exists, $x-1$ is an upper bound on the load on v_k at the time u^* is added. To see that v_k is in T^*, the search tree built during the assignment of u^*, recall the assumption that the path P^* is created by the assignment of u^*. Let (u_i, v_i) be the last edge (i.e., farthest from v_1) of P^* that is added to the matching in this iteration.

Thus, v_i is reachable from u^*, and since the edge (u_j, v_j) is already in the matching $\forall j, i < j \leq k$, the suffix of P^* from v_i to v_k must also be in T^*.

Claim 19 *When u^* is added, the load on some vertex with load at least x is increased.*

PROOF. Suppose the opposite. We show that a cost-reducing path exists before u^* is added, contradicting our choice of P^* as the first cost-reducing path created by the algorithm. Once again, we use the fact that P^* is created when u^* is added. Let (u_i, v_i) be the first edge (i.e., closest to v_1) of P^* which is added to the matching at this iteration. Thus, before adding u^*, the vertex v_i is reachable from v_1. Let $P' = (u^*, \ldots, v^*)$ be the path from u^* to the least loaded vertex in T^*. Note that v_i must appear in path P'. Thus, the path from v_1 to v_i can be extended to reach v^* using the suffix of P' from v_i to v^*. All the edges of this suffix are available before u^* is added, since they were all available to T^*. The load on v^* must be at least x, otherwise the above path from v_1 to v^* is a cost-reducing path which exists before P^*.

Combining Claims 18 and 19 contradicts the execution of $\mathcal{A}_{SM1}$, and therefore P^* cannot exist.

To bound the running time 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 $O(|U||E|)$.

This $O(|U||E|)$ bound is the same upper-bound as for the Hungarian algorithm. However, in practice $\mathcal{A}_{SM1}$ is slower than the Hungarian algorithm because it tends to build a very large search tree in each iteration. The following section presents another algorithm which is much more efficient in practice.

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

This section describes $\mathcal{A}_{SM2}$, another algorithm for finding optimal semi-matchings. Our analysis of this algorithm's 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 4 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

```
Find an initial semi-matching, M
While there exists a cost-reducing path, P
    Reduce the cost by switching the matching and non-matching
    edges along path P
```

Fig. 2. *Overview of* $\mathcal{A}_{SM2}$

semi-matching. Figure 2 gives an overview of the $\mathcal{A}_{SM2}$ algorithm. Since each iteration reduces the total cost by an integral amount, the cost can only be reduced a finite number of times, so this algorithm must terminate. Moreover, if the initial assignment is nearly optimal, the algorithm terminates after few iterations.

As was the case for $\mathcal{A}_{SM1}$, network flows give an alternative characterization of algorithm $\mathcal{A}_{SM2}$. As described in Section 2.3, G can be represented as a network N. Regarding G as a compact representation of N, $\mathcal{A}_{SM2}$ is an efficient implementation of the generic cycle canceling algorithm [1, Section 9.6] on G.

Finding an Initial Semi-Matching: The first step of algorithm $\mathcal{A}_{SM2}$ is to determine an initial semi-matching, M. A simple approach would be to arbitrarily assign each left-hand vertex to a right-hand vertex. However, as a general rule, finding an initial matching with lower cost tends to reduce the number of iterations required to achieve optimal cost. 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 [12]. 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 pseudocode of this algorithm is given in Appendix B.

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. Our experiments have shown that sorting U is effective in practice at reducing the cost of the initial matching, and hence reducing the number of iterations required to achieve optimal cost.

Finding Cost-Reducing Paths: The critical operation of the $\mathcal{A}_{SM2}$ algorithm is the method for finding cost-reducing paths. As a simple approach, we can determine if a particular vertex $v \in V$ is the initial vertex of a cost-reducing path simply by growing a depth-first search tree of alternating paths rooted at v. To determine if G has any cost-reducing paths at all, it suffices to perform m depth-first searches, one from each vertex $v \in V$. This simple algorithm unfortunately performs much redundant work. The following claim leads to an approach for avoiding this redundant work.

Claim 20 *Let $v, w \in V$ be such that $deg_M(v) \geq deg_M(w)$ and there is an alternating path from v to w. If there is no cost-reducing path starting from v then there is no cost-reducing path starting from w.*

PROOF. We prove the contrapositive. Since there is an alternating path from v to w, any cost-reducing path starting from w can be extended to a cost-reducing path starting from v.

Our algorithm for finding cost-reducing paths works as follows. First we find a vertex $v \in V$ such that v has not been visited by any previous depth-first searches and $deg_M(v)$ is maximum over all such vertices. To find a such a vertex v quickly, the non-visited V-vertices are maintained sorted by their load in an array of $|U| + 1$ buckets. Next we build a depth-first search tree of alternating paths rooted at v in the usual manner. If this depth-first search tree does not contain a cost-reducing path then Claim 20 shows that there is no cost-reducing path starting from any of the vertices that were visited. Since this algorithm considers each edge in G at most once and there are only $|U|+1$ buckets, it finds a cost-reducing path or lack thereof in $O(|U| + |E|) = O(|E|)$ time.

After finding a cost-reducing path, we reduce the cost of the matching by switching matching and non-matching edges along the path. This step clearly takes $O(|E|)$ time as well. Recall that this switching process only affects the load on the first and last vertices on the path.

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 requires $O(|E|)$ time. 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 total cost at most $O(|U|^2)$ and that each iteration reduces the cost by at least 1. We now derive an improved bound.

Theorem 21 *$\mathcal{A}_{SM2}$ requires at most $O(\min\{|U|^{3/2}, |U||V|\})$ iterations to achieve optimality.*

PROOF. For a given initial semi-matching, M_0, assume that the V-vertices are sorted by load in non-increasing order, that is, $deg_{M_0}(v_1) \geq \ldots \geq deg_{M_0}(v_m)$. In each iteration, the algorithm identifies a cost-reducing path $P = (v_a, \ldots, v_b)$. By switching matching and non-matching edges along this path we remove one unit of load from v_a and add one unit to v_b. This load balancing process can be described as follows: Initially we have n towers of blocks, where the i^{th} tower consists of $deg_{M_0}(v_i)$ blocks. Reducing the cost using path P amounts to moving one block from the tower corresponding to v_a to the tower corresponding to v_b. We bound the number of iterations of $\mathcal{A}_{SM2}$ by bounding the total number of possible block moves. Our proof is based on the following properties:

(1) Since we always move load units to a less loaded vertex, operations may be ordered such that the V-vertices remain sorted in non-increasing order of their loads throughout the execution of $\mathcal{A}_{SM2}$. In other words, the towers are always sorted from highest to lowest.
(2) By the definition of cost-reducing paths, when moving a load unit from v_a to v_b, the load on v_a before the iteration is strictly greater than the load on v_b after the iteration.

For any value of m, the total number of moves is at most nm since by the above properties each block may be moved at most m times. If $m > \sqrt{n}$, we use a different argument. Consider the k^{th} tower, corresponding to v_k, where k is arbitrary. By the first property, the maximum possible load on v_k at any time is n/k. Thus, the height of the k^{th} tower is at most n/k. By the second property, any block that we keep moving from tower k can be moved at most n/k additional times before it arrives a tower with height 1 (and hence could never be moved again).

Thus, for any k, each block may be moved at most $k+n/k$ times in total. The tightest bound is obtained by choosing $k = \lceil \sqrt{n} \rceil$: Each block may be moved at most $2\sqrt{n}+1$ times. Since each move of a block corresponds to one iteration of $\mathcal{A}_{SM2}$, the number of iterations of $\mathcal{A}_{SM2}$ is $O(n^{3/2})$. In conclusion, combining the nm bound that holds for arbitrary m with the improved bound for $m > \sqrt{n}$, we may bound the total number of iterations by $O(\min\{|U|^{3/2}, |U||V|\})$.

Remark: 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. [12] 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 simplified pseudocode for $\mathcal{A}_{SM2}$ given in Appendix B 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 significantly 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 performance in practice, 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. Our simulation program was implemented in C and run on a 2.4GHz Pentium 4 machine with 512MB of RAM. Each input graph is a random graph from a particular family. The graph generators for each family, and the following descriptions of them, are from [2]. All graphs have $|U| = |V|$.

FewG and ManyG: The U-vertices and V-vertices are divided into k groups of equal size. The FewG family has $k = 32$ and the ManyG family has $k = 256$. Each vertex in the j^{th} group of U chooses y random neighbors from the $(i-1)^{th}$ through $(i+1)^{th}$ group of V, where y is binomially distributed with mean 5.

Grid: The graph is an approximate d-dimensional grid. d is chosen such that the average degree is roughly 6.

Hexa: The vertices on each side are partitioned into n/b blocks of size b. One random bipartite hexagon is added between each block i on one side and each of the blocks $i+k$ on the other side, with $|k| \leq K$ for some K. The parameters b and K are chosen by the program in such a way that an average degree of 6 is achieved (i.e., $3K/b = 6$), but few pairs of hexagons have more than one vertex in common.

Hilo: The i^{th} U-vertex is connected to the j^{th} V-vertex, for all positive j with

$i - 10 \leq j \leq i$.

Rope: The U-vertices and V-vertices are partitioned into $t = n/6$ blocks of size 6, denoted $U_0 \dots U_{t-1}$ and $V_0 \dots V_{t-1}$. Block i on one side is connected to block $i+1$ on the other side, for $i = 0 \dots t-2$; block U_{t-1} is connected to block V_{t-1}. Thus the graph is a "rope" that zigzags between the two sides of the graph. Consecutive pairs of blocks along the rope are connected alternately by perfect matchings and random bipartite graphs of average degree 5.

Zipf: The i^{th} U-vertex is connected to the j^{th} V-vertex with probability roughly proportional to $1/(ij)$. The constants are chosen such that the average degree is 6.

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 [17], and LEDA [18]. 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) summarizes the results of these experiments on graphs with 2^{16} vertices. Appendix D contains graphs of the experimental results over a wide range of graph sizes. In both cases we measured 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 random seed to generate the input graph. Large input graphs (after being transformed into an assignment problem) exceeded the memory of our test machine and no reasonable results could be recorded. Therefore we do not report CSA and LEDA results for all input sizes. 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 $\mathcal{A}_{SM1}$ and $\mathcal{A}_{SM2}$ with two algorithms for computing maximum bipartite matchings from [2]: BFS, their fastest implementation based on augmenting paths, and LO, their fastest implementation based on the push-relabel method. Table 1(b) summarizes the results of these experiments on graphs with 2^{19} vertices. Appendix D contains graphs of the experimental results over a wide range of graph sizes. As before, the reported value is the mean of the execution time over five graphs with different seeds. These results show that $\mathcal{A}_{SM2}$ is roughly as efficient as LO, the best known algorithm for the maximum bipartite matching problem, on a variety of input graphs.

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

(a)

Graph	$\mathcal{A}_{SM1}$	$\mathcal{A}_{SM2}$	CSA	LEDA
FewG	1.190	0.165	0.931	22.324
Grid	0.428	0.059	0.975	5.068
Hexa	0.969	0.187	1.531	20.906
Hilo	0.427	0.218	2.059	8.952
ManyG	1.096	0.109	0.897	13.603
Rope	0.175	0.109	0.978	5.890
Zipf	3.962	0.078	—	—
Total	8.247	0.925	>7.371	>76.744

(b)

Graph	$\mathcal{A}_{SM2}$	BFS	LO
FewG	2.696	11.406	1.531
Grid	0.428	3.075	0.850
Hexa	2.715	10.721	1.315
Hilo	2.237	2.247	4.728
ManyG	2.728	10.228	1.631
Rope	1.015	1.818	1.047
Zipf	0.841	0.287	0.693
Total	12.66	39.782	11.795

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.

7 Conclusions and Future Work

This paper has introduced the optimal semi-matching problem for bipartite graphs. We have seen how this problem can be reduced to known network optimization problems, and that solutions to this problem minimize all L_p-norms, including the L_∞-norm. We have presented two new algorithms for solving this problem, $\mathcal{A}_{SM1}$ and $\mathcal{A}_{SM2}$; the former has a good worst-case time bound, and the latter performs very well in practice. $\mathcal{A}_{SM2}$ also solves the maximum bipartite matching problem with roughly the same efficiency as the best known algorithms.

We feel that our analysis of $\mathcal{A}_{SM2}$ is not the best possible. As future work, we plan to investigate improved bounds for this algorithm. Other future work may include a generalization of the semi-matching problem to allow weighted edges in the graph. As in the unweighted case, one can adopt network optimization algorithms to compute optimal weighted semi-matchings; thus, the challenge is in developing algorithms for the weighted problem with similar efficiency to our current algorithms for the unweighted problem.

References

[1] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, Network Flows: Theory, Algorithms, and Applications, Prentice Hall, 1993.

[2] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, J. Stolfi, Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms, ACM J. Exp. Algorithmics 3 (8), Source code available at `http://www.avglab.com/andrew/soft.html`.

[3] J. Hopcroft, R. Karp, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, SIAM J. Computing 2 (1973) 225–231.

[4] H. W. Kuhn, The Hungarian method for the assignment problem, Naval Res. Logist. Quart. 2 (1955) 83–97.

[5] E. Lawler, Combinatorial Optimization: Networks and Matroids, Dover, 2001.

[6] Y. Azar, On-line Load Balancing, in: A. Fiat, G. Woeginger (Eds.), Online Algorithms: The State of the Art (LNCS 1442), Springer-Verlag, 1998, Ch. 8.

[7] Active Directory, `http://www.microsoft.com/windowsserver2003/technologies`.

[8] W. A. Horn, Minimizing average flow time with parallel machines, Operations Research 21 (1973) 846–847.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, 2001.

[10] R. L. Graham, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: A survey, Ann. Discrete Math 5 (1979) 287–326.

[11] J. L. Bruno, E. G. Coffman, R. Sethi, Scheduling independent tasks to reduce mean finishing time, Communications of the ACM 17 (1974) 382–387.

[12] B. Awerbuch, Y. Azar, E. Grove, M. Y. Kao, P. Krishnan, J. S. Vitter, Load Balancing in the L_p Norm, in: Proceedings of the IEEE Symposium on Foundations of Computer Science, 1995.

[13] Y. Azar, A. Z. Broder, A. R. Karlin, On-line load balancing, Theoretical Computer Science 130 (1) (1994) 73–84.

[14] Y. Azar, J. Naor, R. Rom, The Competitiveness of On-line Assignments, in: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 1992.

[15] D. Abraham, Algorithmics of two-sided matching problems, Master's thesis, Department of Computing Science, University of Glasgow (2003).

[16] W. Rudin, Principles of Mathematical Analysis, 3rd Edition, McGraw-Hill, 1976.

[17] A. Goldberg, R. Kennedy, An efficient cost scaling algorithm for the assignment problem, Math. Prog. 71 (1995) 153–178,
Source code available at `http://www.avglab.com/andrew/soft.html`.

[18] K. Melhorn, S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1999.

A Pseudocode of $\mathcal{A}_{SM1}$

Algorithm 1 *Find an optimal semi-matching via a modified Hungarian algorithm.*

```
// M is the semi-matching that is built by this algorithm.
// Q is a queue used for performing breadth-first search.
// Initially M and Q are empty.
For each root ∈ U
    // Perform breadth-first search from root
    Remove all entries from Q
    Enqueue(Q, root)

    // S is the set of vertices visited by this breadth-first search
    Let S = { root }

    // bestV is the V-vertex with least load found so far
    Let bestV = null

    While Q is not empty
        Let w = Dequeue(Q)
        If w ∈ U Then
            Let N = UnmatchedNeighbors(w)
        Else
            Let N = MatchedNeighbors(w)
            If bestV = null Or deg_M(w) < deg_M(bestV) Then
                Let bestV = w
            End
        End
        For each n ∈ N
            Let Parent(n) = w
            Enqueue(Q, n)
        End
        Let S = S ∪ N
    End

    // Switch edges along path from bestV to root,
    // thus increasing the size of M
    Let v = bestV
    Let u = Parent(v)
    Add {u,v} to M
    While u ≠ root
        Let v = Parent(u)
        Remove {u,v} from M
        Let u = Parent(v)
        Add {u,v} to M
    End
End
```

In this pseudocode, `MatchedNeighbors(w)` returns a set containing all neighbors of vertex w that are matched with vertex w. Similarly, `UnmatchedNeighbors(w)` returns all neighbors of vertex w that are not matched with vertex w.

B Pseudocode of $\mathcal{A}_{SM2}$

Algorithm 2 *Compute an initial semi-matching.*

```
// M is the semi-matching that is built by this algorithm.
// Initially M is empty.
Let U be the list of left-hand vertices
Sort U by increasing degree
For each u ∈ U
    Let S be the set of neighbors of u with minimum load
    Let v be a member of S such that deg(v) is minimum
    Add {u,v} to M
End
```

Algorithm 3 *Find a cost-reducing path.*

```
FindCostReducingPath()
    // S is the set of V-vertices that have not yet been visited
    Let S = V
    While S ≠ ∅
        Let root be a member of S such that deg_M(root) is maximum
        Remove root from S
        Let Parent(root) = null
        DoDFS(root)
    End
    Stop: No cost-reducing path exists
End

DoDFS(v)
    For each u ∈ MatchedNeighbors(v)
        Let Parent(u) = v

        For each w ∈ UnmatchedNeighbors(u)
            If w ∉ S Or deg_M(w) > deg_M(v) Then
                // w has already been visited,
                // or load on w worse than load on v
                Skip to next w
            End

            Remove w from S
            Let Parent(w) = u
            If deg_M(w) ≤ deg_M(root)-2 Then
                Stop: Path from w to root is a cost-reducing path
            End
```

```
                DoDFS(w)
            End
        End
    End
```

C Counterexamples to Converses of Theorems 11 and 16

Not every maximum matching is a subset of an optimal semi-matching. For example, the maximum matching in Figure C.1(a) cannot be extended to an optimal semi-matching in this graph—the only optimal semi-matching for this graph is given in Figure C.1(b).

A semi-matching that minimizes the maximal load but not the total cost is illustrated in Figure C.2(a), where the maximal load is 2 and $T(M) = 6$. The optimal semi-matching that minimizes both objectives is given in Figure C.2(b), where the maximal load is 2 and $T(M) = 5$.

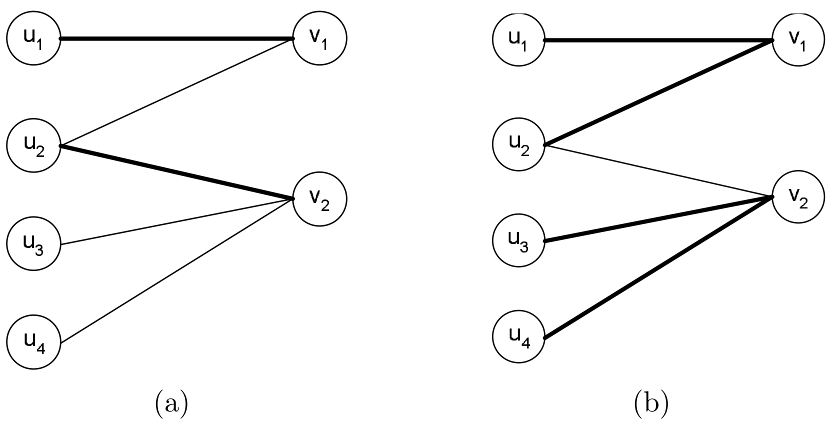


Fig. C.1. *The maximum matching in (a) cannot be extended to the unique optimal semi-matching, shown in (b).*

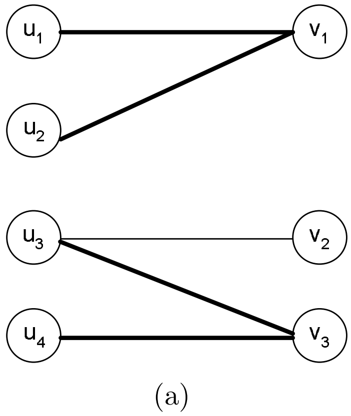


Fig. C.2. *(a) shows a semi-matching that is optimal with respect to the L_∞ norm of the load on the right-hand vertices. (b) shows the unique semi-matching that is optimal with respect to both the L_∞-norm and the L_2-norm of the load on the right-hand vertices.*

D Experimental Results

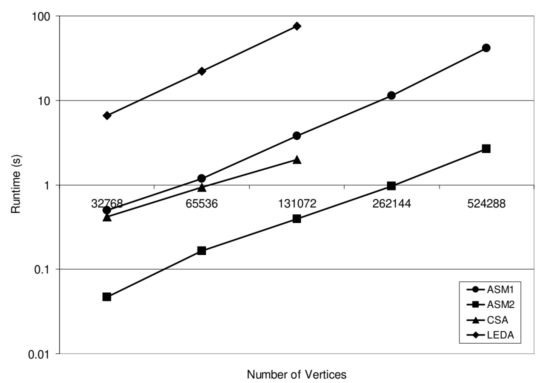


Fig. D.1. Computing optimal semi-matchings on the FewG graph family

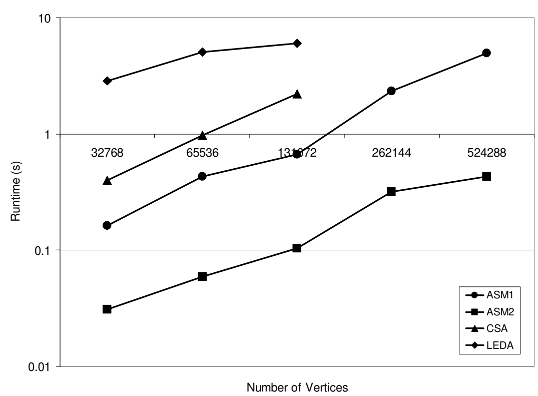


Fig. D.2. Computing optimal semi-matchings on the Grid graph family

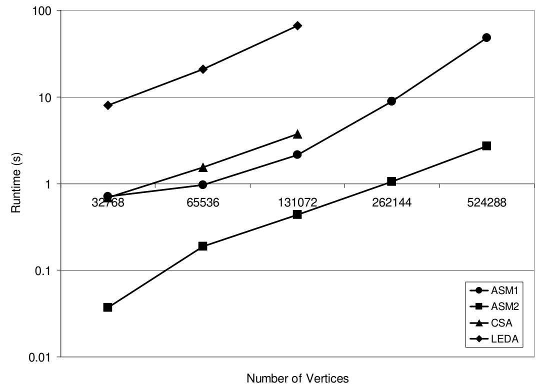


Fig. D.3. Computing optimal semi-matchings on the Hexa graph family

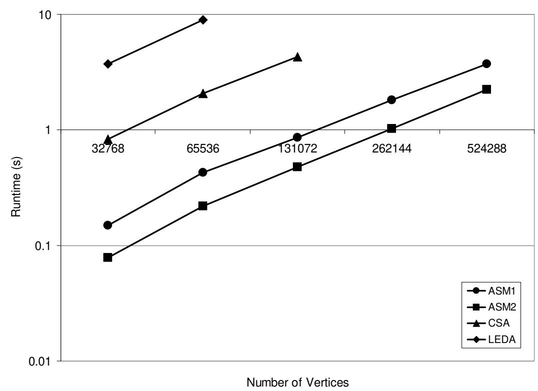


Fig. D.4. Computing optimal semi-matchings on the Hilo graph family

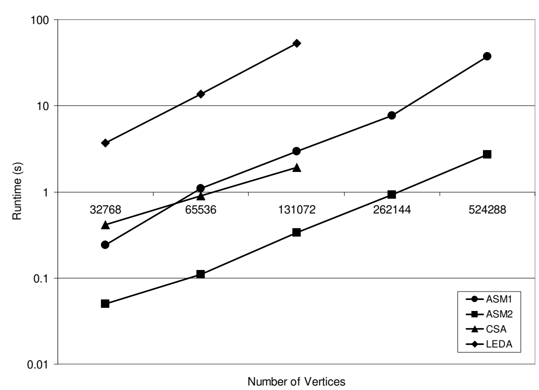


Fig. D.5. Computing optimal semi-matchings on the ManyG graph family

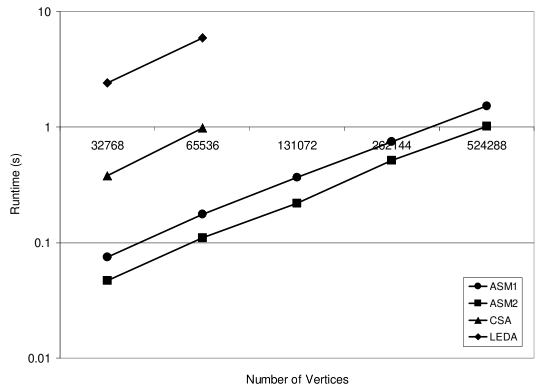


Fig. D.6. Computing optimal semi-matchings on the Rope graph family

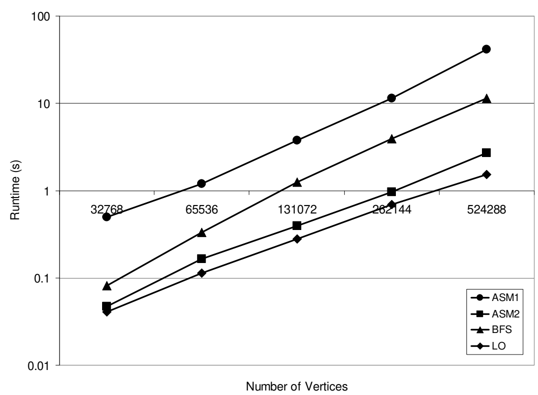


Fig. D.7. Computing maximum bipartite matchings on the FewG graph family

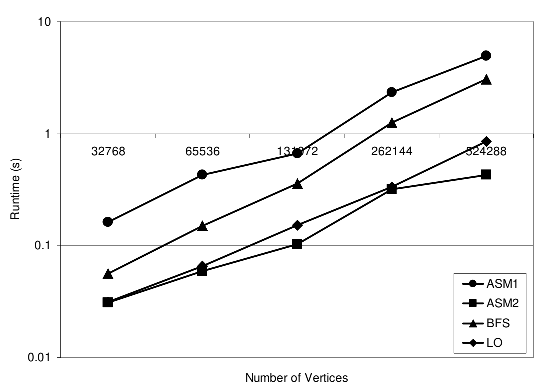


Fig. D.8. Computing maximum bipartite matchings on the Grid graph family

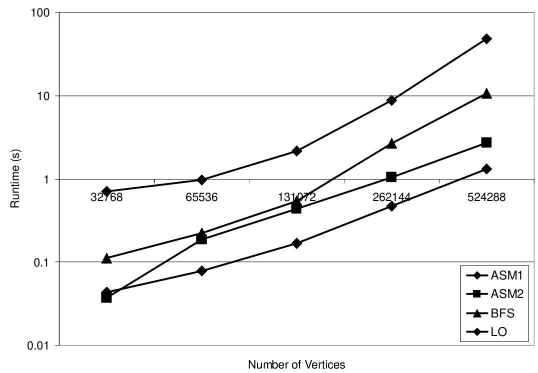


Fig. D.9. Computing maximum bipartite matchings on the Hexa graph family

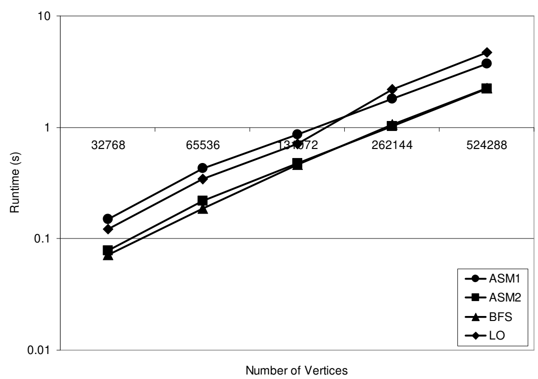


Fig. D.10. Computing maximum bipartite matchings on the Hilo graph family

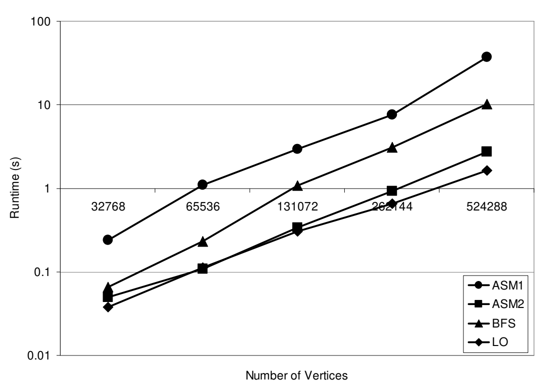


Fig. D.11. Computing maximum bipartite matchings on the ManyG graph family

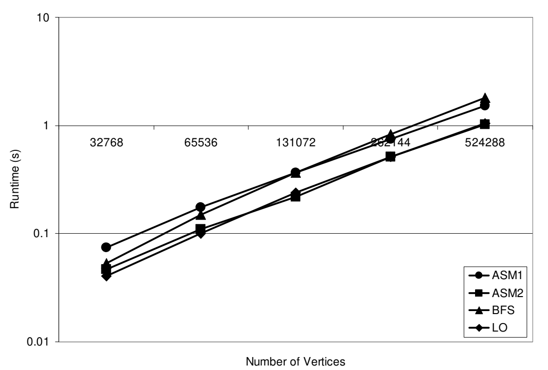


Fig. D.12. Computing maximum bipartite matchings on the Rope graph family