The Generalized Stable Allocation Problem

Brian C. Dean and Namrata Swar

School of Computing, Clemson University
Clemson, SC, USA.
{bcdean,nswar}@cs.clemson.edu

Abstract. We introduce and study the generalized stable allocation problem, an "ordinal" variant of the well-studied generalized assignment problem in which we seek a fractional assignment that is stable (in the same sense as in the classical stable marriage problem) with respect to a two-sided set of ranked preference lists. We develop an $O(m \log n)$ algorithm for solving this problem in a bipartite assignment graph with n nodes and m edges. When edge costs are present, we show that it is NP-hard to compute a stable assignment of minimum cost, a result that stands in stark contrast with most other stable matching problems (e.g., the stable marriage and stable allocation problems) for which we can efficiently optimize over the set of all stable assignments.

1 Introduction

The generalized assignment problem (GAP) is a well-studied classical optimization problem that commonly appears in areas of application such as scheduling and load balancing. Using scheduling notation for convenience, consider a set of I jobs indexed by $[I] := \{1, \ldots, n\}$ and J machines indexed by $[J] = \{1, \ldots, J\}$, and let us form a bipartite assignment graph $G = ([I] \cup [J], E)$ with $n = I + J$ nodes and $m = |E|$ edges, where each edge $(i, j) \in E$ has an associated cost $c(i, j) \geq 0$, capacity $u(i, j) > 0$, and multiplier $\mu(i, j) > 0$. We can then state the GAP as the following linear program:

$$\begin{aligned}
\text{Minimize } & \sum_{(i,j)\in E} c(i,j)x(i,j) & \\
\text{Subject to } & \sum_{j\in[J]} x(i,j) = 1 & \forall i \in [I] \\
& \sum_{i\in[I]} x(i,j)\mu(i,j) \leq 1 & \forall j \in [J] \\
& 0 \leq x(i,j) \leq u(i,j) & \forall (i,j) \in E.
\end{aligned}$$

The multiplier $\mu(i, j)$ tells us that one unit of job i uses up $\mu(i, j)$ units of capacity when assigned to machine j. It is these multipliers that differentiate a *generalized* assignment problem from a traditional linear assignment problem, and these give us quite a bit of flexibility in modeling a variety of different scenarios. Multipliers allow us to convert between different units of measurement or currency, and *lossy* multipliers (less than one) allow us to account for spoilage of goods or other forms of leakage or loss during transportation. In the scheduling context above, we can scale our multipliers appropriately to assign non-uniform processing times to jobs and non-uniform capacities to machines.

In this paper, we introduce and study the *generalized stable allocation problem* (GSAP), an ordinal analog of the GAP where we express the quality of an assignment in terms of ranked preference lists submitted by the jobs and machines, rather than by numeric costs.

1.1 Previous Work

The fundamental underlying problem in our domain of interest is the classical bipartite stable matching (or stable marriage) problem, where we seek to match n men with n women, and each (man, woman) submits a ranked preference list over all of the (women, men). Here, we seek a matching M between the men and women that is *stable*, in the sense that it there is no man-woman pair $(m, w) \notin M$ who both prefer each-other to their partners in M. Gale and Shapley showed in 1962 how to solve any instance of the stable matching problem with a simple $O(m)$ algorithm [6]. Building on this work, a series of "high multiplicity" variants of the stable matching problem have been considered in the past few decades:

- Roth [9] studied the many-to-one bipartite *stable admission* problem, where we wish to match unit-sized entities with non-unit-sized entities. The most prominent application of this problem is the *National Residency Matching Program* (NRMP), a centralized program in the USA used to assign medical school graduates to hospitals, where each hospital has a quota governing the number of graduates it can accept.
- Baiou and Balinski [1] introduced what we call the bipartite *stable b-matching problem*, where we are matching non-unit-sized elements with non-unit-sized elements, and each element i in our biparite assignment graph has a specified quota $b(i)$ governing the number of elements on the other side of the graph to which it should be matched.
- A more recent paper of Baiou and Balinski [2] introduced the bipartite *stable allocation problem*, which is similar to the bipartite stable b-matching problem except that the quotas $b(i)$ can now be arbitrary real numbers. This problem is also known as the *ordinal transportation problem*, since it is a direct analog of the classical cost-based transportation problem. The generalized stable allocation problem is an extension of the stable allocation problem to allow for edge multipliers.

In terms of algorithmic complexity, the stable admissions and stable b-matching problems are not much different from the simpler stable matching problem; they all can be solved using the Gale-Shapley algorithm in $O(m)$ time by first expanding each element i of non-unit size $b(i)$ into $b(i)$ unit elements, each with the same preference list as i. However, the stable allocation problem is somewhat harder: the Gale-Shapley algorithm solves the problem but with exponential worst-case time, an algorithm of Baiou and Balinski [2] runs in $O(mn)$ time, and a more recent algorithm of Dean and Munshi [5] unifies these techniques and employs sophisticated data structures to achieve an $O(m \log n)$ running time.

1.2 Our Results

In this paper we discuss the algorithmic ramifications of adding edge multipliers to the stable allocation problem. Our first result is a positive one: we show that the Dean-Munshi algorithm can be appropriately enhanced to solve the generalized stable allocation problem while maintaining its fast $O(m \log n)$ running time. Our second result, however, is negative: we show that the if edges are also given associated costs, then the problem of finding a generalized stable assignment of minimum cost is NP-hard. This highlights a strong distinction between the generalized and non-generalized stable allocation problems, since with the non-generalized stable allocation problem, it is possible to optimize over the set of all stable assignments in an efficient fashion.

2 Preliminaries

Let $N(i)$ denote the set of machines adjacent to job i in our bipartite assignment graph, and let $N(j)$ be all the jobs adjacent to machine j. Each job i submits a ranked preference list over machines in $N(i)$ and each machine j submits a ranked preference list over jobs in $N(j)$. If job i prefers machine $j \in N(i)$ to machine $j' \in N(i)$, then we write $j >_i j'$. If machine j prefers job i to job i' in $N(j)$, then we similarly write $i >_j i'$. Preference lists are strict, containing no ties. As with the GAP, each edge $(i, j) \in E$ has an associated multiplier $\mu(i, j) > 0$ and an upper capacity $u(i, j) > 0$. Later on, when we consider the "optimal" variant of the GSAP, we will also associate a cost $c(i, j)$ with each edge. Let $x(i, j)$ denote the amount of assignment from job i to machine j, and let $x \in \mathbf{R}^m$ denote the vector representing an entire assignment. We use set notation in the standard way to describe aggregate quantities; for example, $x(i, N(i)) = \sum_{j \in N(i)} x(i, j)$.

In order for an assignment x to be feasible, every job i must be fully assigned (i.e., $x(i, N(i)) = 1$). It is typically not necessary for every machine to be fully assigned. However, it will simplify our model considerably to make this assumption as well, which we can do by introducing a "dummy" job (which we assign index 1), such that the dummy assignment $x(1, j)$ to machine j indicates the amount of machine j that is in reality unassigned. We set $\mu(1, j) = J$ and $u(1, j) = 1/J$ for each $j \in [J]$, so that the dummy job can simultaneously provide any amount of "slack" assignment required at each machine j. To balance out the dummy job, we also introduce a dummy machine (also assigned index 1). The dummy job and machine appear last in the preference lists of the remaining jobs and machines, including the dummies themselves. The remaining entries in the preference lists of the dummy job and machine are arbitrarily ordered.

We now say that an assignment x is *feasible* if

$$\begin{aligned}
&\sum_{j \in [J]} x(i, j) = 1 && \forall i \in [I] \\
&\sum_{i \in [I]} x(i, j)\mu(i, j) = 1 && \forall j \in [J] - \{1\} \\
&0 \leq x(i, j) \leq u(i, j) && \forall (i, j) \in E.
\end{aligned}$$

Note that the dummy machine is the only machine whose incoming assignment is unconstrained; all other machines are constrained to be fully assigned in any feasible assignment.

An assignment x is *stable* if it is feasible and contains no *blocking pair*, where a blocking pair is a pair $(i, j) \in E$ such that $x(i, j) < u(i, j)$, there exists a machine $j' < j$ for which $x(i, j') > 0$, and there exists a job $i' < i$ for which $x(i', j) > 0$. Intuitively, (i, j) is a blocking pair if i and j would both be happier switching some of their less-preferred assignments to the edge (i, j), and if there is room to increase $x(i, j)$. Note that traditionally, stable matching problems are defined with complete preference lists, with $E = [I] \times [J]$. In our case, we can conceive of extending every preference list so it is complete — for example, by appending the entries in $[J] - N(i)$ to the end of job i's preference list in arbitrary order. However, we note that this is unnecessary, since any assignment that utilizes one of these additional edges will be unstable due to a blocking pair formed with the dummy machine. Hence, we can think of our instance has having complete preference lists, but for all practical purposes we can ignore the extra edges not in E.

An assignment x is *job-optimal* if, for every job i, the vector describing i's assignments (ordered by i's preference list) is lexicographically maximal over all stable assignments. As we will see shortly, a stable, job-optimal assignment exists for every instance of the GSAP.

3 Algorithmic Results

In this section, we develop our $O(m \log n)$ algorithm for the GSAP. We begin with the much simpler Gale-Shapley (GS) algorithm, which solves the problem with worst-case exponential time, and whose analysis provides many of the underlying theoretical properties we need for our later analysis. We then describe the Dean-Munshi (DM) algorithm for the stable allocation problem, on which our approach is based.

3.1 The Gale-Shapley Algorithm

To generalize the classical Gale-Shapley (GS) algorithm so it solves the GSAP, we introduce a small amount of additional notation. Fixing an assignment x, we define r_j to be the job $i \in N(j)$ with $x(i, j) > 0$ that is least-preferred by j. We refer to r_j as the "rejection pointer" of j, since it points to the job that j would prefer to reject first, given a choice amongst its current assignments. We also define the "proposal pointer" of job i, q_i[1], to be the machine j most preferred by i such that $x(i, j) < u(i, j)$ and $i >_j r_j$. Informally, q_i points to the first machine on i's preference list that would be willing to receive additional assignment from i.

[1] We use q_i rather than p_i for i's proposal pointer to maintain consistency with the notation in [5].

The GS algorithm begins with an empty assignment. In our case, we start with all machines being assigned to the dummy job: $x(1, j) = 1$ for all $j \in [J]$. The algorithm will terminate when all jobs are fully assigned: $x(i, N(i)) = 1$ for all $i \in [I]$. In each iteration, we select a job i that is not fully assigned and have it propose all $b = 1 - x(i, N(i))$ units of unassigned load to machine $j = q_i$. Machine j accepts, thereby sending it over capacity, after which it proceeds to reject load from machine r_j until j's incoming assignment once again satisfies $\sum_i x(i, j)\mu(i, j) = 1$. Note that the rejection pointer r_j can move up j's preference list during this process, as edges become emptied out due to rejection, and that this in turn can cause proposal pointers to advance. More precisely, whenever $x(r_j, j)$ drops to zero due to rejection, we advance r_j up j's preference list until $x(r_j, j) > 0$. Any time a proposal pointer q_i becomes invalid due to either $x(i, j)$ reaching $u(i, j)$ or $i \leq_j r_{q_i}$, we advance q_i down i's preference list until it becomes valid. In total, since the proposal and rejection pointers move in a monotonic fashion, their maintenance requires only $O(m)$ time over the entire algorithm (note that this is exactly the same pointer management as used with the DM algorithm). Unfortunately, the GS algorithm can take exponential time in the worst case, as originally shown in [2].

Just as the GS algorithm itself generalizes readily to solve the GSAP, most of its well-known properties also generalize in a straightforward fashion. Proofs of the following lemmas are omitted in the interest of brevity, as they are straightforward to obtain by generalizing the corresponding results from the stable allocation problem (see [3, 5]).

Lemma 1. *Irrespective of proposal order, the GS algorithm for the GSAP always terminates in finite time (even with irrational problem data), and it does so with a stable job-optimal assignment.*

Lemma 2. *For each edge (i, j), as the GS algorithm executes, $x(i, j)$ will never increase again after it experiences a decrease.*

Corollary 1. *During the execution of the GS algorithm, each edge (i, j) becomes saturated at most once, and it also becomes empty at most once.*

We also note that the "integral" variant of the GS algorithm developed in [4] for the *unsplittable* variant of the stable allocation problem also generalizes in a straightforward manner to the generalized case with edge multipliers.

3.2 The (Generalized) Dean-Munshi Algorithm

Baiou and Balinski were the first to propose a polynomial-time ($O(mn)$ time) algorithm for the stable allocation problem; one can think of their algorithm as an "end-to-end" variant of the GS algorithm, which makes a series of proposals and rejections along "augmenting paths". The Dean-Munshi (DM) algorithm is an enhancement of this approach that uses additional structural insight coupled with dynamic tree data structures to improve the running time for each augmentation from $O(n)$ to $O(\log n)$, thereby resulting in an overall running time of $O(m \log n)$. We now describe how this algorithm should be modified in the context of the GSAP; the reader is referred to [5] for a more detailed discussion of the DM algorithm.

For any assignment x, we define $G(x)$ to be a bipartite graph on the same set of nodes as our original instance, having "proposal" edges (i, q_i) for all $i \in [I]$ and "rejection" edges (r_j, j) for all $j \in [J] - \{1\}$ (note that the dummy machine is the only node that lacks a proposal or rejection pointer). The graph $G(x)$ has a relatively simple structure: each of its connected components is either a tree or a tree plus one edge, forming a single cycle with paths branching off it. Moreover, there is only one "tree" component — the one containing the dummy machine; all remaining components are "cycle" components.

The DM algorithm maintains the structure of each component in $G(x)$, along with a list of the jobs in each component that are not yet fully assigned. In each iteration, a component C with some unassigned job i is selected, and an augmentation is performed within C. If C is the tree component, then the augmentation is straightforward: job i proposes ε units to machine $j = q_i$ (which arrive as $\varepsilon\mu(i, j)$ units at machine j), then machine j accordingly rejects $\varepsilon\mu(i, j)$ units from job $i' = r_j$ (which arrive as $\varepsilon\mu(i, j)/\mu(i', j)$ units at i'), after which i' proposes to machine $j' = q_{i'}$, and so on, until some job eventually proposes to the dummy machine, which accepts and does not issue a subsequent rejection. The quantities being proposed and rejected along this path vary in accordance with the initial amount being proposed, ε, and the aggregate product of the multipliers along the path (treating the multiplier of a rejection edge as the reciprocal of its forward multiplier). By traversing the path and accumulating this multiplier product, we can easily compute the maximum amount ε that job i can initially propose so that either a proposal edge along the path becomes saturated, some rejection edge becomes empty, or i becomes fully assigned. To be somewhat more precise, suppose C is the tree component of $G(x)$, and let $\pi(s) = \pi^+(s) \cup \pi^-(s)$ denote the unique augmenting path through C from source job s to machine 1, where $\pi^+(s)$ is the set of proposal edges along the path and $\pi^-(s)$ is the set of rejection edges. Let us define the *potential* of job s with respect to machine 1 as

$$\alpha(s) = \frac{\prod\limits_{(i,j) \in \pi^+(s)} \mu(i, j)}{\prod\limits_{(i,j) \in \pi^-(s)} \mu(i, j)}.$$

For any source machine $t \neq 1$ in C, we similarly define its potential as $\beta(t) = \alpha(r_t)/\mu(r_t, t)$, and we define $\beta(1) = 1$. Note that if we initially propose ε units from job $s \in C$, then $\varepsilon\alpha(s)$ units will eventually reach machine 1, assuming no edge on $\pi(i)$ becomes saturated or empty in the process. Similarly, a rejection of ε units from machine $t \in C$ will eventually arrive at machine 1 as $\varepsilon\beta(t)$ units.

Consider now some edge (i, j) along the augmenting path $\pi(s)$ in C, and suppose we initially propose $\varepsilon/\alpha(s)$ units from job s. If (i, j) is a proposal edge, then $\varepsilon/\alpha(i)$ units arrive at job i and are proposed along (i, j), and if (i, j) is a rejection edge, then $\varepsilon/\beta(j)$ units arrive at j and are then rejected along (i, j), so $\varepsilon/\alpha(i)$ rejected units arrive at i. We define the *scaled* assignments and capacities of edge (i,j) as $x'(i,j) = x(i,j)\alpha(i)$ and $u'(i,j) = u(i,j)\alpha(i)$, and we define the *scaled residual capacity* of a proposal edge (i,q_i) in C as $r(i,q_i) = u'(i,q_i) - x'(i,q_i)$, and of a rejection edge (r_j,j) in C as $r(r_j,j) = x'(r_j,j)$. Finally, we define the residual capacity of the path $\pi(s)$ as $r(\pi(s)) = \min\{r(i,j) : (i,j) \in \pi(s)\}/\alpha(s)$. The quantity $r(\pi(s))$ tells us exactly how large we can set ε so that some edge along $\pi(s)$ becomes saturated or empty. We therefore augment $\min(1-x(s,N(s)), r(\pi(s)))$ units from s, since this is sufficient to either saturate or empty an edge, or to make s fully assigned. We call such an augmentation within the tree component a type I augmentation.

If our algorithm chooses to augment within a cycle component C, the details are slightly more involved. Let π_C denote the unique cycle in C, and let $\mu(\pi_C)$ denote the product of the multipliers on the proposal edges in π_C divided by the product of the multipliers on the rejection edges in π_C. As with other generalized assignment problems, we can classify π_C as being *gainy* if $\mu(\pi_C) > 1$, *lossy* if $\mu(\pi_C) < 1$ or *break-even* if $\mu(\pi_C) = 1$. Suppose some job $i \in \pi_C$ is not fully assigned, and that we augment around π_C by initially proposing ε units from job i, so $\varepsilon\mu(\pi_C)$ rejected units eventually arrive back at job i after propagating the augmentation all the way around π_C. If π_C is gainy, then job i will end up seeing more than ε units returned to it, so augmenting from job i around a gainy cycle results in a net *decrease* in i's total assignment. Similarly, augmenting around a lossy cycle results in a net *increase* in i's assignment. In either case, we augment, as before, until some edge along π_C becomes either saturated or empty, or until job i is fully assigned. We call this a type II augmentation.

The final case to consider is the type III augmentation — when we augment a cycle component C, but where all jobs along π_C are fully assigned, so in this case we are starting our augmentation from a source job $s \notin \pi_C$. Let s' be the first job in π_C we reach when we follow an augmenting path from s. Here, we augment along the path only from s to s', again proposing the maximum possible amount from s so that some edge on our path becomes either saturated or empty, or so that s becomes fully assigned. As a result of this augmentation, s' will now no longer be fully assigned, so this triggers a subsequent type II augmentation within π_C.

It is easy to justify termination and correctness of our algorithm. Since it is performing a series of proposals and rejections (in a batch fashion) that the GS algorithm could have performed, Lemma 1 asserts that we must terminate with a stable job-optimal assignment. Running time analysis is fairly straightforward. Without using any sophisticated machinery, we can perform each augmentation in $O(n)$ time (we show how to reduce this to $O(\log n)$ time in a moment). To bound the number of augmentations, note that each one either saturates an edge, empties out an edge, or fully assigns some job. Corollary 1 tells us that each edge can be saturated or emptied out at most once, so there are at most $O(m)$ augmentations that saturate or empty an edge. We must take slightly more care with the augmentations that end up fully assigning a job. For the simpler stable allocation problem, the DM algorithm ensures that a job, once fully assigned, remains fully assigned. In the generalized case, however, this is no longer necessarily true. Note that we never perform type-II augmentations in a cycle whose jobs are all fully-assigned, so a type-II augmentation can never cause a fully-assigned job to become not fully assigned (and of course, a type-I augmentation cannot cause this to happen either). Only a type III augmentation can result in a fully-assigned job becoming not fully assigned. However, every type III augmentation that causes some job i to become not fully assigned is followed by a subsequent type II augmentation that either saturates/empties an edge or fully assigns job i. Hence a type III followed by type II augmentation either saturates/empties an edge, or results in a net increase in the number of fully-assigned jobs. We therefore have $O(m+n)$ total augmentations.

3.3 Fast Augmentation with Dynamic Trees

In order to augment in $O(\log n)$ time, we use dynamic trees [10, 11]. Each component of $G(x)$ is stored in a single dynamic tree, and each cycle component is stored as a dynamic tree plus one additional edge, arbitrarily chosen. A dynamic tree data structure maintains an edge-weighted and/or node-weighted free tree and supports a host of useful operations all in $O(\log n)$ amortized time. The main operations we need are the following:

- *Split/join:* Split a tree into two trees by deleting an edge, or join two trees by adding an edge.
- *Global-update:* Modify the weight on every edge or node in a tree by a given additive or multiplicative factor.
- *Path-min:* Find the minimum edge or node weight along the unique tree path joining two specified nodes i and j.
- *Path-update:* Update the edge or node weights along the unique tree path from i to j by a given additive offset.

For simplicity, we describe augmentation in the context of the tree component. The same general approach also works for cycle components, except there we must also account for the extra edge separately (i.e., augmentation on a cycle is equivalent to augmentation on a path through a tree, plus augmentation on a single edge).

Along with every edge (i,j) in our dynamic trees we store two weights: the scaled assignment $x'(i,j)$ and the scaled residual capacity $r(i,j)$. Every job node i in one of our dynamic trees is weighted with its potential $\alpha(i)$, and each machine node j is weighted with its potential $\beta(j)$. For the tree component, recall that these potentials are defined relative to the dummy machine. For a cycle component, we define potentials relative to an arbitrarily-chosen "reference" node within the component. Note that we can change the reference node in only $O(\log n)$ time; for example, we can change from reference node s to reference node t by applying a *global-update* to multiply all node potentials and divide all edge weights in the component by the ratio of the potential of t to that of s.

Augmentation from some source job s uses the *path-min* operation to compute $r(\pi(s)) = \min\{r(i,j) : (i,j) \in \pi(i)\}\alpha(s)$, followed by the *path-update* operation to decrease the residual capacity of every edge in $\pi(s)$ by $\varepsilon = \min(1 -$

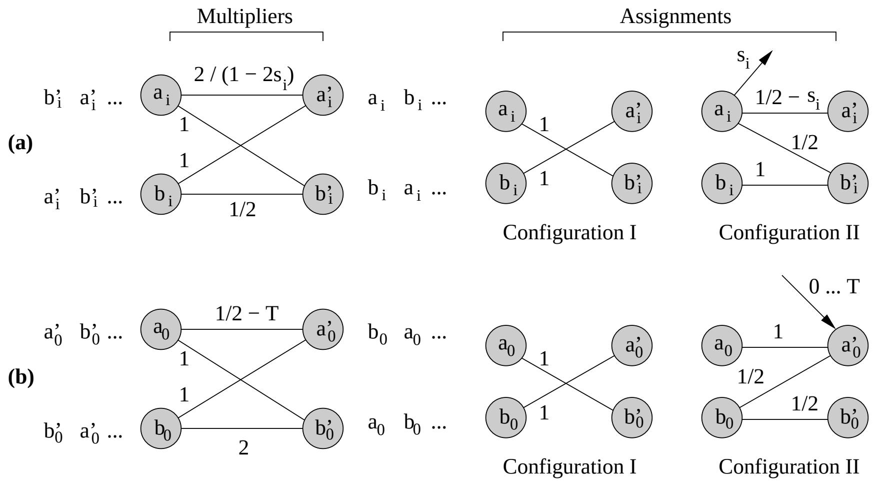

Fig. 1. Illustrations of (a) a "gainy" component C_i (left, with edge multipliers indicated) along with its two stable configurations (right, with edge assignments indicated), and (b) the "lossy" component C_0 along with its two stable configurations. In configuration II, the external edge incoming to a'_0 can carry any amount of assignment in the range $0 \ldots T$.

$x(s, N(s)), r(\pi(s)))/\alpha(s)$, and to adjust the scaled assignment $x'(i,j)$ of every edge $(i,j) \in \pi(s)$ by ε. Note that proposal edges and rejection edges must be treated slightly differently during the path update, since $x'(i,j)$ must increase by ε for a proposal edge and decrease by ε for a rejection edge; however, this is easy to accommodate by introducing some relatively standard extra machinery into the dynamic tree infrastructure. The fact that we use scaled residual capacity, where the weight of each edge (i,j) is scaled relative to the potential $\alpha(i)$, is absolutely crucial, since this "normalizes" all of the edge weights along a path so they are effectively in the same unit of measurement (relative to an endpoint of the path), allowing us to apply the *path-min* and *path-update* operations in a uniform fashion across a path.

Augmentations often cause some subset of edges to become saturated or empty. These edges leave $G(x)$, while other edges (due to advancement of proposal or rejection pointers) may enter, thereby changing the structure of the connected components of $G(x)$. Each leaving edge results in a *split* operation, and each entering edge result in a *join.* Since each edge leaves or enters at most a constant number of times, this accounts for at most $O(m \log n)$ work throughout the entire algorithm. Finally, whenever we perform a *split* or *join*, we must call *global-update* to update node potentials appropriately; for example, when a component with reference node s is split into two components by removing edge (i,j), we need to re-weight the potentials in the newly-created component not containing s relative to an arbitrarily-chosen reference node in that component.

4 The Optimal Generalized Stable Allocation Problem

If the edges in our bipartite assignment instance have associated costs, we can consider the *optimal* generalized stable assignment problem: among all stable assignments, find one having minimum total cost. In this section, we show that this problem is NP-hard. This result highlights a major distinction between the generalized stable assignment problem and its simpler variants, since the "optimal" variants of the both the stable matching and stable allocation problems can be solved in polynomial time [8, 5]. For both of these simpler variants, there exists a convenient combinatorial description of the set of all stable assignments in terms of closed subsets of a carefully crafted "rotation DAG" (and for the stable matching problem, one can also write a linear program whose extreme points correspond to stable solutions). However, since the optimal generalized stable assignment problem is NP-hard, this strongly suggests that there is no convenient mathematical characterization of the set of all stable solutions.

Theorem 1. *The optimal generalized stable allocation problem is NP-hard.*

The proof of this theorem uses a reduction from the well-known NP-complete SUBSET-SUM problem [7], which takes as input a set of positive numbers $s_1 \ldots s_n$ and a target T, and asks whether or not there exists a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} s_i = T$. Given an instance I of SUBSET-SUM, we show how to construct an instance I' of the optimal generalized stable allocation problem such that I has a YES answer if and only if I' has a stable solution of cost at most $\sum_i s_i - T$. Without loss of generality, we assume by rescaling that $s_1 \ldots s_n$ and T all lie in the range $(0, 1/2)$.

To build I', we construct a bipartite assignment graph with $2n + 2$ elements on each side, comprised of n "gainy" components $C_1 \ldots C_n$ and one "lossy" component C_0. Each gainy components C_i is constructed as shown in Figure 1(a), with two jobs a_i and b_i and two machines a'_i and b'_i. Preference lists are as shown in the figure, with elements external to C_i placed last on each preference list in arbitrary order. Edge multipliers are given by $\mu(a_i, a'_i) = 2/(1 - 2s_i)$, $\mu(b_i, b'_i) = 1/2$, and $\mu(a_i, b'_i) = \mu(b_i, a'_i) = 1$. The lossy component is similarly constructed, as shown in Figure 1(b). All remaining edges crossing between components are assigned unit multipliers. For each gainy component $i = 1 \ldots n$, we assign the cost of edge (b_i, a'_i) to s_i. All other edges have zero cost. For simplicity, we do not explicitly include the dummy job and machine in our construction, since these were introduced primarily to simplify our algorithms; rather, we simply allow each machine to be only partially assigned (which is equivalent to its assignment to a dummy job).

For the gainy component C_i, we define two specific assignment *configurations*, also shown in Figure 1(a). Configuration I has $x(a_i, a'_i) = x(b_i, b'_i) = 0$ and $x(a_i, b'_i) = x(b_i, a'_i) = 1$, and configuration II has $x(a_i, a'_i) = 1/2 - s_i$, $x(a_i, b'_i) = 1/2$, $x(b_i, b'_i) = 1$, with a_i having s_i units assigned to some machine external to C_i. We similarly define two assignment configurations for the lossy component C_0, shown in Figure 1(b).

Lemma 3. *For each gainy component C_i, any assignment pattern involving the edges incident to C_i other than configuration I and II is unstable.*

Proof. Let us say that an element is *externally assigned* if it has some positive allocation to an element outside C_i; otherwise, we say it is *internally assigned.* We note that job b_i and machines a'_i and b'_i cannot be externally assigned in any stable assignment: if job b_i were externally assigned, then (b_i, b'_i) would be a blocking pair, if machine a'_i were externally assigned, (b_i, a'_i) would be a blocking pair, and if machine b'_i were externally assigned, then (a_i, b'_i) would be a blocking pair. Both a_i and b_i must be fully assigned, since this is a requirement of any feasible assignment. Although a'_i and b'_i are not explicitly required to be fully assigned, this must still be the case in any stable solution: if a'_i is not fully assigned, then (b_i, a'_i) is a blocking pair, and if b'_i is not fully assigned, then (a_i, b'_i) is a blocking pair.

Now consider the following two cases: (i) if job a_i is internally assigned, then all of the elements in C_i are assigned internally, and it is easy to show that as a consequence of our multipliers (and the fact that every element is fully assigned), the only possible stable feasible assignment is configuration I. On the other hand (ii) if job a_i is externally assigned, then (a_i, a'_i) will be a blocking pair unless a'_i is assigned only to a_i, so $x(a_i, a'_i) = 1/2 - s_i$. Moreover, since b_i can now only be assigned to b'_i, we have $x(b_i, b'_i) = 1$, and since b'_i needs exactly one unit of incoming assignment we must have $x(a_i, b'_i) = 1/2$. Therefore, a_i must have s_i of its units assigned externally in order for it to have exactly one unit of outgoing assignment. This is configuration II.

Lemma 4. *For the lossy component C_0, any assignment pattern involving the edges incident to C_0 other than configuration I and II shown in Figure 1(b) is unstable.*

Proof. The argument in this case is quite similar to the preceding proof. We begin by arguing that in any stable assignment, jobs a_0 and b_0 and machine b'_0 must be internally assigned: if job a_0 is externally assigned, then (a_0, b'_0) is a blocking pair, if job b_0 is externally assigned, then (b_0, a'_0) is a blocking pair, and if machine b'_0 is externally assigned, then (b_0, b'_0) is a blocking pair. Moreover, a_0 and b_0 are explicitly constrained to be fully assigned, and b'_0 must be fully assigned or else (b_0, b'_0) is a blocking pair. We now argue, just as before, that if machine a'_0 is internally assigned, then we must be in configuration I, otherwise we must be in configuration II.

Lemma 5. *Given an instance I of SUBSET-SUM, the corresponding instance I' of the optimal generalized stable allocation instance I' has a stable allocation of cost at most $\sum_i s_i - T$ if and only if I has a YES answer.*

Proof. If I admits a set S such that $\sum_{i \in S} s_i = T$, then we can obtain a stable solution for I' by placing C_0 and each C_i for $i \in S$ in configuration II, and the remaining components in configuration I. For each $i \in S$, the external assignment of s_i units from job a_i is directed to machine a'_0, thereby exactly matching the

T units of external assignment needed at a'_0. The total cost of this solution is $\sum_i s_i - T$. On the other hand, if I' has a stable solution of cost at most $\sum_i s_i - T$, then we must have in aggregate at least T units of external assignment from $a_1 \ldots a_n$, and a'_0 can accept at most T units. Hence, we must have exactly T units of external assignment, and these T units must come from a subset of gainy components (in configuration II) corresponding to a solution for the SUBSET-SUM instance I.

References

1. M. Baiou and M. Balinski. Many-to-many matching: Stable polyandrous polygamy (or polygamous polyandry). *Discrete Applied Mathematics*, 101:1–12, 2000.
2. M. Baiou and M. Balinski. Erratum: The stable allocation (or ordinal transportation) problem. *Mathematics of Operations Research*, 27(4):662–680, 2002.
3. B.C. Dean, N. Immorlica, and M.X. Goemans. Finite termination of "augmenting path" algorithms in the presence of irrational problem data. In *Proceedings of the 14th annual European Symposium on Algorithms (ESA)*, pages 268–279, 2006.
4. B.C. Dean, N. Immorlica, and M.X. Goemans. The unsplittable stable marriage problem. In *Proceedings of the 4th IFIP International Conference on Theoretical Computer Science*, 2006.
5. B.C. Dean and S. Munshi. Faster algorithms for stable allocation problems. In *Proceedings of the 1st MATCH-UP (Matching Under Preferences) Workshop*, pages 133–144, 2008.
6. D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–14, 1962.
7. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completenes.* W. H. Freeman, 1979.
8. R.W. Irving, P. Leather, and D. Gusfield. An efficient algorithm for the "optimal" stable marriage. *Journal of the ACM*, 34(3):532–543, 1987.
9. A.E. Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of Political Economy*, 92:991–1016, 1984.
10. D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
11. D..D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.