

Faster Algorithms For Stable Allocation Problems

Brian C. Dean
School of Computing
Clemson University
bcdean@cs.clemson.edu

Siddharth Munshi
School of Computing
Clemson University
smunshi@cs.clemson.edu

June 10, 2008

Abstract

We consider a high-multiplicity generalization of the classical stable matching problem known as the *stable allocation problem*, introduced by Baiou and Balinski in 2002. By leveraging new structural properties and sophisticated data structures, we show how to solve this problem in $O(m \log n)$ time on a bipartite instance with n vertices and m edges, improving the best known running time of $O(mn)$. Our approach simplifies the algorithmic landscape for this problem by providing a common generalization of two different approaches from the literature — the classical Gale-Shapley algorithm, and a recent algorithm of Baiou and Balinski. Building on this algorithm, we provide an $O(m \log n)$ algorithm for the non-bipartite stable allocation problem. Finally, we give a polynomial-time algorithm for solving the “optimal” variant of the bipartite stable allocation problem, as well as a 2-approximation algorithm for the NP-hard “optimal” variant of the non-bipartite stable allocation problem.

1 Introduction

The classical stable matching (marriage) problem has been extensively studied since its introduction by Gale and Shapley in 1962 [5]. Given n men and n women, each of whom submits an ordered preference list over all members of the opposite sex, we seek a matching between the men and women that is *stable* — having no man-woman pair (m, w) (known as a *blocking pair* or a *rogue couple*) where both m and w would both be happier if they were matched with each-other instead of their current partners. Gale and Shapley showed how to solve the problem optimally in $O(n^2)$ time using a simple and natural “propose and reject” algorithm, and over the years we have come to understand a great deal about the rich mathematical and algorithmic structure of this problem and its many variants (e.g., see [6, 8]).

In this paper we study a high-multiplicity variant of the stable matching problem known as the *stable allocation problem*, introduced by Baiou and Balinski in 2002 [2]. This problem follows in a long line of “many-to-many” generalizations of the classical stable matching problem. The many-to-one *stable admission* problem [9] has been used since the 1950s in a centralized national program in the USA known as the *National Residency Matching Program* (NRMP) to assign medical school graduates to residencies at hospitals; here, we have a bipartite instance with unit-sized elements (residents) on one side and capacitated non-unit-sized elements (hospitals) on the other. In 2000,

Baiou and Balinski [1] studied what one could call the stable bipartite b -matching problem, where both sides of our bipartite graph contain elements of non-unit size, and each element i has a specified quota $b(i)$ governing the number of elements on the other side of the graph to which it should be matched. The stable allocation problem is a further generalization of this problem where the amount of assignment between two elements i and j is no longer zero or one, but a nonnegative real number (we will give a precise definition of the problem in a moment). The stable allocation problem is also known as the *ordinal transportation problem* since it can be viewed as a variant of the classical transportation problem where the quality of an assignment is specified in terms of ranked preference lists and stability instead of absolute numeric costs. This can be a useful model in practice since in many applications, ranked preference lists are often easy to obtain while there may not be any reasonable way to specify exact numeric assignment costs; for example, it may be obvious that it is preferable to process a certain job on machine A rather than machine B , even though there is no natural way to assign specific numeric costs to each of these alternatives.

In the literature, there are two prominent algorithms for solving the stable allocation problem. The first is a natural generalization of the Gale-Shapley (GS) algorithm that issues “batch” proposals and rejections. Although this algorithm tends to run quite fast in practice, often even in sublinear time, its worst-case running time is exponential [4]. Baiou and Balinski (BB) propose what one could view as an “end-to-end” variant of the GS algorithm (we will describe both algorithms in detail in a moment), with worst-case running time $\Theta(mn)$ on a bipartite instance with n vertices and m edges. In this paper we develop an algorithm that generalizes both the GS and BB approaches and uses additional structural properties as well as dynamic tree data structures to achieve a worst-case running time of $O(m \log n)$, which is only a factor of $O(\log n)$ worse than the optimal linear running time we can achieve for the much simpler unit stable matching problem. Note that since the fastest known algorithms for solving high-multiplicity “flow-based” assignment problems run in $\Omega(mn)$ worst-case time, our new results now provide a significant algorithmic incentive to model assignment problems as stable allocation problems rather than flow problems.

Building on our new algorithm, we also provide an $O(m \log n)$ algorithm for the *non-bipartite* stable allocation problem, a natural generalization of the non-bipartite unit stable matching problem (commonly called the *stable roommates* problem). In the book of Gusfield and Irving on the stable marriage problem [6], one of the open questions posed by the authors is whether or not there exists a convenient transformation from the non-bipartite stable roommates problem to the simpler bipartite stable matching problem. We show that a transformation of this flavor does indeed exist, and that it simplifies the construction of algorithms not only for stable roommates but also for the non-bipartite stable allocation problem. It also provides a simple proof of the well-known fact that although an integer-valued solution may not always exist for the stable roommates problem, a half-integral solution does always exist.

The Gale-Shapley algorithm for the unit stable matching problem finds a stable solution that is “man-optimal, woman-pessimal”, where each man ends up paired with the best partner he could possibly have in any stable matching, and each woman ends up with the worst partner she could possibly have in any stable assignment (by symmetry, we obtain a “woman-optimal, man-pessimal” matching if the women propose instead of the men). In order to rectify this asymmetry, Gusfield et al. [7] developed a polynomial-time algorithm for the *optimal* stable matching problem, where we associate a cost with each (man, woman) pairing and ask for a stable matching of minimum total cost (costs are typically designed so that the resulting solution tends to be “fair” to both sexes). Bansal et al. [3] extended this approach to the optimal stable bipartite b -matching problem, and we show how to extend it further to solve the optimal stable allocation problem in polynomial time.

As a consequence, we also obtain a 2-approximation algorithm for the NP-hard “optimal” variant of the non-bipartite stable allocation problem by generalizing a similar 2-approximation algorithm for the optimal stable roommates problem.

2 Preliminaries

In order to eliminate any awkwardness associated with multiple-partner matchings involving men and women, let us assume we are matching I jobs indexed by $[I] = \{1, \dots, I\}$ to J machines indexed by $[J] = \{1, \dots, J\}$. Each job i has an associated processing time $p(i)$, and each machine j has a capacity $c(j)$. The jobs and machines comprise the left and right sides of a bipartite graph with $n = I + J$ vertices and m edges. Let $N(i)$ denote the set of machines to which job i is adjacent in this graph, and similarly let $N(j)$ denote the set of jobs that are neighbors of machine j . For each edge (i, j) we associate an upper capacity $u(i, j) \leq \min(p(i), c(j))$ governing the maximum amount of job i that can be assigned to machine j . Later on, we will also associate a cost $c(i, j)$ with edge (i, j) . Problem data is not assumed to be integral (see [4] for further notes on the issue of integrality in stable allocation problems).

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 $j' \in N(i)$ or if $j \in N(i)$ and $j' \notin N(i)$, then we write $j >_i j'$; similarly, we say $i >_j i'$ if machine j prefers job i to job i' . Preference lists are strict, containing no ties. Letting $x(i, j)$ denote the amount of job i assigned to machine j , we say the entire assignment $x \in \mathbf{R}^m$ is *feasible* if it satisfies

$$\begin{aligned} x(i, [J]) &= p(i) && \forall i \in [I] \\ x([I], j) &= c(j) && \forall j \in [J] \\ 0 \leq x(i, j) &\leq u(i, j) && \forall \text{ edges } (i, j), \end{aligned}$$

where we denote by $x(S, T)$ the sum of $x(i, j)$ over all $i \in S$ and $j \in T$. In order to ensure that a feasible solution always exists, we assume job 1 and machine 1 are both “dummy” elements with very large respective processing times and capacities, which we set so that $p(1) = c([J] - \{1\})$ and $p([I]) = c([J])$. The preference list of job 1 should contain all machines in arbitrary order, ending with machine 1, and the preference list of machine 1 should contain all jobs in an arbitrary order, ending with job 1. We can regard a job or machine that ends up being assigned to a dummy as being unassigned in our original instance.

An edge (i, j) is said to be a *blocking pair* for assignment x if $x(i, j) < u(i, j)$, there exists a machine $j' <_i j$ for which $x(i, j') > 0$, and there exists a job $i' <_j i$ for which $x(i', j) > 0$. Informally, (i, j) is a blocking pair if $x(i, j)$ has room to increase, and both i and j can be made happier by increasing $x(i, j)$ in exchange for decreasing some of their current lesser-preferred allocations. An assignment x is said to be *stable* if it is feasible and admits no blocking pairs. Note that the dummy job can never be part of a blocking pair, and neither can the dummy machine.

One can show that a stable assignment exists for any problem instance. Moreover, there always exists a unique stable assignment that is *job-optimal*, where an assignment is job-optimal if the vector describing the allocation of each job i (ordered by i 's preference list) is lexicographically maximal over all possible stable assignments. By symmetry, a unique *machine-optimal* assignment always exists as well. As it turns out, a job-optimal assignment is always machine-pessimal and vice-versa. It is also a well-known fact that the dummy allocations $x(1, j)$ and $x(i, 1)$ are the same

ADVANCE-Q(I): While $x(i, j) = u(i, j)$ or q_i not accepting of i : Step q_i downward in i 's preference list.	ADVANCE-R(J): While $x(r_j, j) = 0$: Step r_j upward in j 's preference list. ADVANCE-Q(r_j)
---	---

in every stable assignment.

Given any 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 . Job r_j is the job that j would logically choose to reject first if it were offered an allocation from a more highly-preferred job. If $i >_j r_j$, then we say machine j is *accepting* for job i , since j would be willing to accept some additional allocation from i in exchange for rejecting some of its current allocation from r_j . For each job i , we let q_i be the machine j most preferred by i such that $x(i, j) < u(i, j)$ and j is accepting for i . If i wishes to increase its allocation, q_i is the first machine it should logically ask.

The Gale-Shapely (GS) Algorithm. The GS algorithm for the stable allocation problem is a natural generalization of the well-studied GS “propose and reject” algorithm for the unit stable matching problem. The analysis of this algorithm will help us to analyze the correctness and running time of our new algorithm to follow.

Although the GS algorithm typically starts with an empty assignment, we start with an assignment x where every machine j is fully assigned to the dummy job ($x(1, j) = c(j)$), and the remaining jobs are unassigned — this simplifies matters somewhat since every machine except the dummy henceforth remains fully assigned. In each iteration of the algorithm, we select a arbitrary job i that is not yet fully assigned; let $T = p(i) - x(i, [J])$ be the amount of i 's processing time that is currently unassigned. Job i “proposes” $T' = \min(T, u(i, j) - x(i, j))$ units of processing time to machine $j = q_i$, which accepts. However, if j is any machine except the dummy, then it is now overfilled by T' units beyond its capacity, so it proceeds to reject T' units, starting with job r_j . During the process, $x(r_j, j)$ may decrease to zero, in which case r_j becomes a new job higher on j 's preference list and rejection continues until j is once again assigned exactly $c(j)$ units. The algorithm terminates when all jobs are fully assigned, and successful termination is ensured by the fact that each job can send all of its processing time to the dummy machine as a last resort.

Consider briefly the behavior of the q_i 's and r_j 's during the GS algorithm. We regard q_i as a pointer into job i 's preference list that starts out pointing at i 's first choice and over time scans monotonically down i 's preference list according to the ADVANCE-Q procedure above, which is automatically called any time an edge (i, q_i) becomes saturated ($x(i, j) = u(i, j)$). Similarly, r_j is a pointer into machine j 's preference list that starts at job 1 (the dummy, which is the least-preferred job on j 's list), and over time advances up the list according to the ADVANCE-R procedure, which is automatically called any time an edge (r_j, j) becomes empty. Note that all of this “pointer management” takes only $O(m)$ total time over the entire GS algorithm. We use exactly the same pointer management infrastructure in our new algorithm.

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

Proof. It is well-known (see, e.g., [6]) that for the classical unit stable matching problem, the

GS algorithm always terminates with a man-optimal (job-optimal) stable assignment. This result easily extends to the stable allocation problem if we have integral problem data and no upper edge capacities, since in this case the GS algorithm can be viewed as a “batch” version of the classical GS algorithm executed on the unit instance we obtain when we split each job i into $p(i)$ unit jobs and each machine j into $c(j)$ unit machines. However, this reduction no longer applies if we have irrational problem data or upper edge capacities. In this case, finite termination is shown in [4]. To show that our final assignment is stable, suppose at termination that (i, j) is a blocking pair. Since $q_i <_i j$, we know j must have rejected i at some point; however, this implies that $r_j \leq_j i$, contradicting our assumption that j has any allocation it prefers less than i . To show that our assignment is job-optimal, suppose it is not. At some point during execution, there must have been a rejection from some machine j to some job i that resulted in an assignment x with $x(i, j) < x^*(i, j)$, where x^* is a stable assignment satisfying $x(i, j') \geq x^*(i, j')$ for all $j' >_i j$. Consider the first point in time when such a rejection occurs, and let x denote our assignment right after this rejection. Since $x(i, j) < x^*(i, j)$ and since j is fully assigned in both x and x^* , there must be some i' for which $x(i', j) > x^*(i', j)$. Note that $i' >_j i$, since otherwise j would have rejected i' fully before rejecting i . Since $x(i', j) > x^*(i', j)$ and $x(i', [J]) \leq x^*(i', [J])$, there must be some machine j' such that $x(i', j') < x^*(i', j')$; let j' be the first such machine in the preference list of i' . We know $j >_i j'$ since otherwise i' would have already been rejected by j' , contradicting the fact that (i, j) is the earliest instance of a rejection of the type considered above. Since $x^*(i', j) < u(i', j)$, this implies that (i', j) is a blocking pair in x^* , contradicting our assumption that x^* was stable. The argument showing that our final assignment is machine-pessimal is analogous and completely symmetric to this job-optimality argument. \square

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

Proof. This is also shown in [4], and it follows easily as a consequence of the monotonic behavior of the q_i and r_j pointers: $x(i, j)$ increases as long as $q_i = j$, stopping when q_i advances past j , which happens either when (i, j) becomes saturated, or when r_j advances to i . From this point on, $x(i, j)$ decreases until r_j advances past i , after which $x(i, j) = 0$ forever. \square

Corollary 3. *During the execution of the GS algorithm, each edge (i, j) becomes saturated at most once, and it also becomes empty at most once.*

In practice, the GS algorithm often runs quite fast; for example, in the common case where all jobs get one of their top choices, the algorithm usually runs in sublinear time. Unfortunately, the worst-case running time can be exponential even on relatively simple problem instances [4].

3 An Improved “Augmenting Path” Algorithm

In this section, we describe our $O(m \log n)$ algorithm for the stable allocation problem and show how it generalizes and improves upon the GS algorithm and the algorithm of Baiou and Balinski (BB), which we describe shortly. Just like the GS algorithm, our algorithm starts with an assignment x in which every job but the dummy is unassigned, and every machine is fully assigned to the dummy job. As the algorithm progresses, the machines remain fully assigned and the jobs become progressively more assigned. The algorithm terminates when every job is fully assigned.

At any given point in time during the execution of the GS algorithm (say, where we have built up some partial assignment x), we define $G(x)$ to be a bipartite graph on the same set of vertices as our original instance, having edges (i, q_i) for all $i \in [I]$ and (r_j, j) for all $j \in [J] - \{1\}$. Initially $G(x)$ is a tree, containing n vertices, $n - 1$ edges, and no cycles; we regard the dummy machine (the only machine j without an incident (r_j, j) edge) as the root of this tree.

Lemma 4. *For every assignment x we obtain during the course of the GS algorithm, $G(x)$ consists of a collection of disjoint components, the one containing the root vertex (the dummy machine) being a tree and each of the others containing one unique cycle.*

Proof. Consider any connected component C of $G(x)$ spanning job set I' and machine set J' . If $1 \in J'$ (i.e., if C contains the root), then C has $|I'| + |J'| - 1$ edges and must therefore be a tree. Otherwise, C has $|I'| + |J'|$ edges, so it consists of a tree plus one additional cycle-forming edge. \square

We say a component in $G(x)$ is *fully assigned* if $x(i, [J]) = p(i)$ for each job i in the component. As we run our algorithm, we maintain the structure of the tree and cycle components in $G(x)$ along with a list of jobs in each component that are not yet fully assigned. In each iteration of our algorithm, we select an arbitrary component C of $G(x)$ that is not fully assigned and perform an *augmentation* within C . We terminate when every component is fully assigned.

An augmentation consists of a simultaneously-enacted series of proposals and rejections along a path or cycle that can be viewed as the “end to end” execution of a series of GS operations. In the tree component, an augmentation starts from any job i that is not yet fully assigned, and follows the unique path from i to the root (i.e., i proposes to $j = q_i$, which rejects $i' = r_j$, which proposes to $j' = p_{i'}$, and so on, just as the GS algorithm would operate, until we reach machine 1, which is the only machine that accepts a proposal without issuing a subsequent rejection). Along our augmenting path from i to the root, we increase the assignment of each (i, q_i) edge and decrease the assignment of each (r_j, j) edge by the same amount. For a cycle component, we augment along the unique cycle within the component, increasing the assignment on (i, q_i) edges and decreasing the assignment on (r_j, j) edges by the same amount.

We define the *residual capacity* of an edge (i, j) in $G(x)$ as $r(i, j) = u(i, j) - x(i, j)$ for an (i, q_i) edge, and $r(i, j) = x(i, j)$ for an (r_j, j) edge. The residual capacity $r(\pi)$ of an augmenting path/cycle π is defined as $r(\pi) = \min\{r(i, j) : (i, j) \in \pi\}$. When we augment along an augmenting path π starting from job i , we push exactly $\min(r(\pi), p(i) - x(i, [J]))$ units of assignment along π , since this is just enough to either make i fully assigned, or to saturate or make empty one of the edges along π . When we augment along a cycle π , we push exactly $r(\pi)$ units of assignment, since this suffices to saturate or empty out some edge along π , thereby “breaking” the cycle π . When one or more edges along π become saturated or empty, this triggers any appropriate calls to our pointer management infrastructure above, resulting in a change to the structure of $G(x)$ because one or more of the q_i or r_j pointers advances. In general, any time one of these pointers advances, one edge leaves $G(x)$ and another enters: if some pointer q_i advances to q'_i , then (i, q_i) leaves G and (i, q'_i) enters, and if r_j advances to r'_j then (r_j, j) leaves and (r'_j, j) enters. The net impact of each of these modifications is either (i) the tree component splits into a tree and a cycle component, (ii) the tree component and some cycle component merge into a tree component, (iii) one cycle component splits into two cycle components, or (iv) two cycle components merge into one cycle component.

In order to augment efficiently, we store each component of $G(x)$ in a dynamic tree data structure (see [10, 11]). For cycle components, we store a dynamic tree plus one arbitrary edge along the cycle.

This allows us to find the residual capacity along an augmenting path/cycle as well as augment on the path/cycle in $O(\log n)$ time (amortized time is also fine), in much the same way dynamic trees are used push flow along augmenting paths when solving maximum flow problems. Since dynamic trees can handle *split* and *join* operations in $O(\log n)$ time, we can also efficiently maintain the structure of the components of $G(x)$ as edges are removed and added. In total, we spend $O(\log n)$ time for each edge removal (split) and edge addition (join), and since edges are removed at most once (when saturated or emptied) and added at most once, this contributes $O(m \log n)$ to our total running time. Each augmentation takes $O(\log n)$ time and either saturates an edge, empties out an edge, or fully assigns some job, all three of which can only happen once per edge/job. We therefore perform at most $2m + n = O(m)$ augmentations, for a total running time of $O(m \log n)$. Since we are performing in an aggregate fashion a set of proposals and rejections that the original GS algorithm *could* have performed, Lemma 1 tells us that our algorithm must terminate with a stable assignment that is job-optimal and machine-pessimal.

One might wish to think of our algorithm as either an “end to end” variant of the GS algorithm, or as a more sophisticated implementation of the algorithm of Baiou and Balinski [2], which performs augmentations in a similar but much slower fashion ($O(n)$ time per augmentation, leading to a worst-case running time of $\Omega(mn)$, as shown in Appendix A). The key to our approach is the use of dynamic trees to augment quickly, owing to our new structural insight involving the decomposition of the $G(x)$ graph. In addition to unifying the algorithmic landscape for the stable allocation problem, our approach also exposes a remarkable similarity between state-of-the-art approaches based on dynamic trees for solving our problem and the related maximum flow problem.

4 The Optimal Stable Allocation Problem

Once our algorithm from the previous section terminates with a stable, job-optimal assignment x , the graph $G(x)$ may still contain cycle components. The augmenting cycles in these components are known in the unit stable matching literature as *rotations*, and they generalize readily to the case of stable allocation. Rotations lie at the heart of a rich mathematical structure underlying the stable allocation problem, and they give us a means of describing and moving between all different stable assignments for an instance.

Lemma 5. *Let x be a stable assignment with a cycle component C in $G(x)$, where π_C is the unique augmenting cycle in C . We obtain another stable assignment when we augment any amount in the range $[0, r(\pi_C)]$ around π_C .*

Lemma 6. *If x is a stable assignment, then x is the machine-optimal (and job-pessimal) assignment if and only if $G(x)$ has no cycles (i.e., $G(x)$ consists of a single tree component).*

For space considerations, we leave the (fairly mechanical) proofs of these lemmas for the full version of this paper. If we augment $r(\pi_C)$ units around the rotation π_C , we say that we *eliminate* π_C , since this causes one of the edges along π_C to saturate or become empty, thereby eliminating π_C permanently from $G(x)$. The resulting structural change to $G(x)$ might *expose* new rotations that were not initially present in $G(x)$. Note that the structure of $G(x)$ only changes when we eliminate (fully apply) π_C , and not when we push less than $r(\pi_C)$ units around π_C .

Suppose we start with the job-optimal assignment and continue running the same algorithm from the previous section to eliminate all rotations we encounter, in some arbitrary order, until we finally

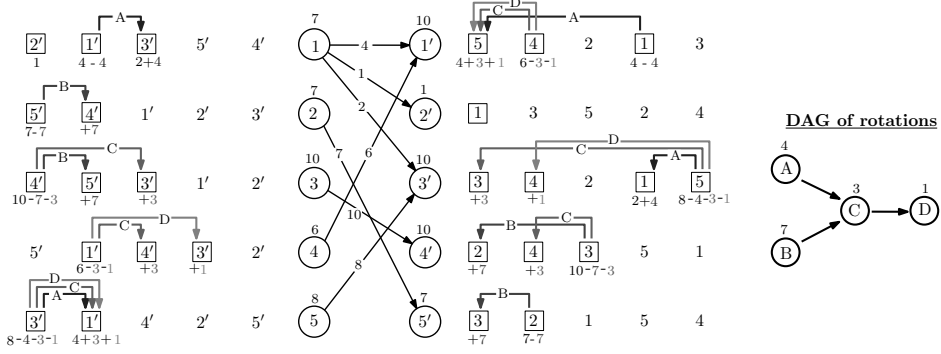


Figure 1: An example bipartite instance and its rotations. No dummy job or machine is shown, since once we reach the job-optimal assignment neither of these takes part in any further augmenting cycles (rotations). The initial job-optimal assignment is shown in the bipartite region, and rotations $A \dots D$ and their associated multiplicities are shown in the DAG on the right. The net effect of each rotation on our assignment is shown with a set of arrows over the preference lists.

reach the machine-optimal assignment (the only stable assignment with no further exposed rotations). We call this a *rotation elimination ordering*. Somewhat surprisingly, as with the unit case, one can show that irrespective of the order in which we eliminate rotations, we always encounter *exactly the same set* of rotations along the way.

Lemma 7. *Let π be a rotation with initial residual capacity r encountered in some rotation elimination ordering. Then π appears with initial residual capacity r in every elimination ordering.*

Since each elimination saturates or empties an edge, we conclude that there are at most $2m$ combinatorially-distinct rotations that can ever appear in $G(x)$, where each such rotation π has a well-defined initial residual capacity $r(\pi)$ (we will also call this the *multiplicity* of π). Let Π denote the set of these rotations. For any $\pi_i, \pi_j \in \Pi$, we say $\pi_i \prec \pi_j$ if π_j cannot be exposed unless π_i is fully applied at an earlier point in time; that is, $\pi_i \prec \pi_j$ if π_i precedes π_j in every rotation elimination ordering. For example, if π_i and π_j share any job or machine in common, then since simultaneously-exposed rotations are vertex disjoint it must be the case that $\pi_i \prec \pi_j$ or $\pi_j \prec \pi_i$; otherwise, we could find a rotation elimination ordering in which π_i and π_j are both exposed at some point in time. We can extend this argument to show that for any job i (machine j) we must have $\pi_1 \prec \pi_2 \prec \dots \prec \pi_k$, where $\pi_1 \dots \pi_k$ are the rotations containing job i (machine j), appropriately ordered. Clearly, Π also contains no cycle $\pi_1 \prec \pi_2 \prec \dots \prec \pi_k \prec \pi_1$, since otherwise none of the rotations $\pi_1 \dots \pi_k$ could ever be exposed. Let us therefore construct a directed acyclic graph $D = (\Pi, E)$ where $(\pi_i, \pi_j) \in E$ if $\pi_i \prec \pi_j$. An example of this *rotation DAG* is shown in Figure 1.

For our purposes, it will be sufficient to compute a “reduced” rotation DAG $D' = (\Pi, E')$ with $E' \subseteq E$ whose transitive closure is D . To do this, we run our algorithm from Section 3 to obtain a job-optimal assignment, then we continue running it until we have generated the set of all rotations Π . This takes $O(m \log n)$ time, although $O(mn)$ time is needed if we actually wish to write down the structure of each rotation along the way. We then use the observation above to generate the $O(mn)$ edges in E' in $O(mn)$ time as follows: for each job i (machine j), compute the set of rotations $\pi_1 \dots \pi_k$ containing i (j), ordered according to the order in which they were eliminated.

We then add the $k - 1$ edges $(\pi_1, \pi_2) \dots (\pi_{k-1}, \pi_k)$ to E' .

The (reduced) rotation DAG has been instrumental in the unit case (see, e.g., [6]) in characterizing the set of all stable matchings for an instance. Conveniently, we can generalize this to the stable allocation problem. Let us call the vector $y \in \mathbf{R}^{|\Pi|}$ *D-closed* if $y(\pi) \in [0, r(\pi)]$ for each rotation $\pi \in \Pi$, and $y(\pi_i) = r(\pi_i)$ if there is an edge $(\pi_i, \pi_j) \in E$ with $y(\pi_j) > 0$. The vector y tells us the extent to which we should apply each rotation in an elimination ordering that follows a topological ordering of D . The *D-closed* property ensures that we fully apply any rotation π_i upon which another rotation π_j depends. Note that *D'*-closed means the same thing as *D*-closed, since D and D' share the same transitive closure (we will give a more detailed discussion of this fact in the full version of this paper).

Lemma 8. *For any instance of the stable allocation problem, there is a one-to-one correspondence between all stable assignments x and all *D*-closed (*D'*-closed) vectors y .*

Consider now the *optimal* stable allocation problem: given a cost $c(i, j)$ on each edge (i, j) in our original instance, we wish to find a stable assignment x minimizing $\sum_{ij} x(i, j)c(i, j)$. Using Lemma 8, we can solve this problem in polynomial time the same way we can solve the optimal variant of the unit stable matching problem. Note that an optimal stable assignment corresponds to a subset of fully-applied rotations — that is, a *D'*-closed vector y with $y(\pi) \in \{0, r(\pi)\}$ for each $\pi \in \Pi$. By assigning each rotation $\pi \in \Pi$ cost indicating the net cost of fully applying π , we can transform the optimal stable allocation problem into an equivalent minimum-cost closure problem on the DAG D' ; for further details, see [7].

5 The Non-Bipartite Stable Allocation Problem

In the *non-bipartite stable allocation problem*, we are given an n -vertex, m -edge graph $G = (V, E)$ where every vertex $v \in V$ has an associated size $b(v)$ and a ranked preference list over its neighbors, and every edge $e \in E$ has an associated upper capacity $u(e)$. Letting $I(v)$ denote the set of edges incident to v , our goal is to compute an assignment $x \in \mathbf{R}^m$ with $\sum_{e \in I(v)} x(e) = b(v)$ for all $v \in V$ that is stable in that it admits no blocking pair. Here, a blocking pair is an edge $e = uv \in E$ such that $x(e) < u(e)$ and both u and v would prefer to increase $x(e)$ while decreasing some of their other allocations.

In the unit case (with $b(v) = 1$ for all $v \in V$), with the added restriction that x must be integer-valued, this is known as the *stable roommates* problem, and it can be solved in $O(m)$ time. As a consequence of the integrality restriction, one can construct instances that have no stable integer-valued solution for the roommates problem. However, no such difficulties arise with the non-bipartite stable allocation problem since it is inherently a “real-valued” problem; we also ensure a solution always exists by adding uncapacitated self-loops to all vertices, and by placing each vertex last on its own preference list. Just as with the dummy job and machine in the bipartite case, we can regard a vertex assigned to itself as actually being unassigned in the original instance, and one can show that the extent to which each vertex is unassigned must be the same in every stable assignment.

In response to an open question posed by Gusfield and Irving [6] on whether or not there exists a convenient transformation from the stable roommates problem to the simpler stable matching problem, we show that a transformation of this flavor does indeed exist, and that it simplifies

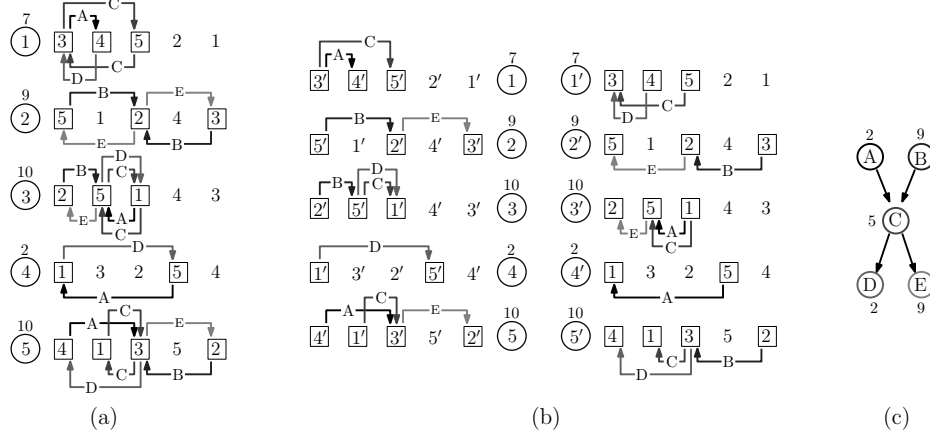


Figure 2: Transforming a non-bipartite instance (a) into a symmetric bipartite instance (b) with rotation DAG (c). Rotations in the resulting bipartite instance are shown overlaid on the non-bipartite instance. A symmetric stable assignment for the bipartite instance is obtained by eliminating A , B , and half of C .

the construction of algorithms not only for stable roommates but also for the non-bipartite stable allocation problem. Suppose we construct a symmetric bipartite instance by replicating a non-bipartite instance, as shown in Figure 2. If we can find a *symmetric* stable assignment x for this symmetric instance (with $x(u, v) = x(v, u)$ for each edge $e = uv$), then by setting $x(e) = x(u, v) = x(v, u)$ we will obtain a stable solution to the non-bipartite instance (since if there was a blocking pair in the non-bipartite solution, this would imply an analogous blocking pair in the bipartite solution). Hence, to solve the non-bipartite problem, we need only consider how to find a *symmetric solution* to a *symmetric instance* of the bipartite problem. One can always do this by carefully choosing the right combination of rotations to apply, starting from the job-optimal assignment.

Due to symmetry, rotations in our bipartite instance now tend to come in pairs. In the example shown in Figure 2(a) we have taken the left-hand-side and right-hand-side effect of each rotation and overlaid these on the original non-bipartite instance. From this, we can see that rotations A and D are mirror images, or *duals*, of each-other, as are rotations B and E . More precisely, rotations π and π' are duals if π is the symmetric analog of π' when we reverse the roles of the left-hand and right-hand sides of our bipartite instance. Note that $r(\pi) = r(\pi')$ if π and π' are duals. The rotation C is its own dual, so we call it a *self-dual* rotation.

Lemma 9. *Consider any symmetric bipartite instance. There is a one-to-one correspondence between symmetric stable assignments x and D -closed (D' -closed) vectors y where $y(\pi) + y(\pi') = r(\pi)$ for every dual pair of rotations (π, π') and $y(\pi) = r(\pi)/2$ for every self-dual rotation.*

This lemma gives another simple proof of the (previously-known) fact that a $1/2$ -integral solution always exists for the stable roommates problem, and it also leads us to an $O(m \log n)$ algorithm for the non-bipartite stable allocation problem: transform into a symmetric bipartite instance, compute the job-optimal stable assignment, then eliminate rotations starting from the job-optimal assignment, taking care not to eliminate the dual of any rotation previously eliminated (there are several ways to accomplish this; for example, we can store a hash of each eliminated rotation).

Finally, eliminate half of the remaining self-dual rotations, leaving a symmetric stable assignment. Complete implementation details will appear in the full version of this paper.

The “optimal” (i.e., minimum-cost) version of the non-bipartite stable allocation problem is NP-hard since it generalizes the NP-hard optimal stable roommates problem. The only difference between the two lies in the self-dual rotations. For the stable roommates problem, the existence of a self-dual rotation is precisely what prevents the existence of an integral stable solution, so we must assume there are no self-dual rotations in our instance. For the non-bipartite stable allocation problem, we are forced to take half of each self-dual rotation, thereby removing them from consideration as well. The remaining problem now looks the same in both cases: find an optimal D -closed set of rotations containing one of each dual pair. For this NP-hard problem, a 2-approximation algorithm can be obtained via a reduction to a weighted 2SAT problem [6], so the same technique also gives us a 2-approximation for the optimal bipartite stable allocation problem.

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] V. Bansal, A. Agrawal, and V. S. Malhotra. Polynomial time algorithm for an optimal stable assignment with multiple partners. *Theoretical Computer Science*, 379(3):317–328, 2007.
- [4] 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.
- [5] D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–14, 1962.
- [6] D. Gusfield and R. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.
- [7] 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.
- [8] D.E. Knuth. Stable marriage and its relation to other combinatorial problems. In *CRM Proceedings and Lecture Notes, vol. 10, American Mathematical Society, Providence, RI. (English translation of Marriages Stables, Les Presses de L’Université de Montréal, 1976)*, 1997.
- [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.

A A Hard Instance for the BB Algorithm

We describe here an n -vertex bipartite instance that causes the BB algorithm to run in $\Omega(n^3)$ time. Suppose we have $n/2$ jobs, each of whose processing time is an integer chosen independently at random from $\{n+1, \dots, 2n\}$ except for job 1 (the dummy), with $p(1) = n^2/2$. We also have $n/2$ machines, each with capacity n except machine 1 (the dummy), whose capacity is set so that $p(\lfloor n/2 \rfloor) = c(\lfloor n/2 \rfloor)$. Each job ranks the machines in order $n/2, n/2-1, \dots, 1$, and each machine ranks the jobs in order $n/2, n/2-1, \dots, 1$. There are no upper capacities $u(i, j)$. When applying the BB algorithm to this instance, we repeatedly augment starting from job 2 until it is fully assigned, then from job 3, and so on (recall that every machine starts out assigned to the dummy job 1).

Due to the order of the preference lists and the order in which we augment, the structure of every intermediate assignment x generated during the execution of the BB algorithm is as follows: a contiguous range of jobs $1 \dots i_0 - 1$ will be fully assigned, with job i_0 (the job from which augmentations are currently issued) partially assigned. These jobs will be assigned to a suffix of the machines $j_0 \dots n/2$. The graph $G(x)$ will be a tree, and the path through $G(x)$ from i_0 to the root (machine 1) visits every job from i_0 down to 1 in sequence. Intuitively, each augmentation starting from i_0 causes the entire assignment to “shift up” from the perspective of the machines.

Let us focus on execution of the BB algorithm from $i_0 = n/4 + 1$ onward. In this regime, there are at least $n^2/4$ units of processing time still to assign, and each augmentation takes $\Omega(n)$ time since each augmenting path has length $\Omega(n)$.

Lemma 10. *For the instance described above, with $i_0 > n/4$, each augmenting path π satisfies $\mathbf{E}[r(\pi)] \leq 5$.*

Suppose we perform $n^2/20$ augmentations (starting from $i_0 = n/4 + 1$). Letting X denote the number of units of processing time assigned during this process, we have $\mathbf{E}[X] \leq n^2/4$. Since $\Pr[X \leq n^2/4] > 0$, the probabilistic method tells us that there must be *some* instance for which $X \leq n^2/4$. For this instance, the BB algorithm performs at least $n^2/20 = \Omega(n^2)$ augmentations, each taking $\Omega(n)$ time.

Proof of Lemma 10. Consider a particular augmenting path π with $i_0 > n/4$, where x denotes the assignment immediately before augmentation on π . Consider any job $i \in [n/4]$. Note job i is assigned in x to a contiguous range of machines $j_i \dots j'_i$, and that augmenting on π will increase $x(i, j_i)$ while decreasing $x(i, j'_i)$. Since we can decrease $x(i, j'_i)$ to no less than zero, $r(\pi) \leq x(i, j'_i)$, and moreover $r(\pi) \leq Z$ where $Z = \min\{x(i, j'_i) : i \in [n/4]\}$.

Due to the uniform machine capacities and the fact that jobs $i+1 \dots i_0$ are assigned to a contiguous suffix of the machines, we can write $x(i, j'_i) = n - ((p(\{i+1, \dots, i_0-1\}) + x(i_0, [n/2])) \bmod n)$, which we rearrange to obtain $n - x(i, j'_i) \equiv p(i+1) + K \pmod{n}$, where $K = p(\{i+2, \dots, i_0-1\}) + x(i_0, [n/2])$. Irrespective of K , we see that $p(i+1) \bmod n$ is uniform in $\{0, \dots, n-1\}$, so $x(i, j'_i)$ is a uniform random number in $[n]$. Moreover, since each $p(i)$ is chosen independently, the $x(i, j'_i)$'s are also independent. Using this fact, we see that Z is the minimum of a set of independent random variables each uniformly chosen from $[n]$. Hence,

$$\mathbf{E}[r(\pi)] \leq \mathbf{E}[Z] = \sum_{k=1}^{\infty} \Pr[Z \geq k] = \sum_{k=0}^{n-1} \left(1 - \frac{k}{n}\right)^{n/4} \leq \sum_{k=0}^{n-1} e^{-k/4} \leq 5.$$