

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

May 17, 2010

Abstract

We consider a high-multiplicity generalization of the classical stable matching problem known as the *stable allocation problem*, introduced by Baïou 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 $O(mn)$. Building on this algorithm, we provide an algorithm for the non-bipartite stable allocation problem running in $O(m \log n)$ time with high probability. 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 [6]. 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 (i, j) (known as a *blocking pair* or a *rogue couple*) where both i and j 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 (more precisely, in $O(m)$ time on a bipartite instance with only m edges) 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 [8, 11]).

In this paper we study a high-multiplicity variant of the stable matching problem known as the *stable allocation problem*, introduced by Baïou 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 [12] 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, Baïou 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 introduced by Baïou and Balinski [2] that issues “batch” proposals and rejections (Baïou and Balinski call this the “row greedy” algorithm, although we prefer to characterize it as an extension of the well-known unit GS algorithm). Although this algorithm tends to run quite fast in practice, often even in sublinear time, its worst-case running time is exponential [5]. Baïou and Balinski also proposed the first strongly-polynomial algorithm for the stable allocation problem, with worst-case running time $\Theta(mn)$ on a bipartite instance with n vertices and m edges; we refer to this as the BB algorithm. In this paper, we develop an algorithm that builds on the BB approach, using 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.

Our algorithm is highly reminiscent of fast “augmenting path” algorithms for solving the maximum flow problem (e.g., [13, 7]), which also rely heavily on dynamic trees. Indeed, the reader familiar with flow algorithms will see many familiar ideas in our work, and we have tried to develop our exposition so as to highlight these similarities. It is worth noting, however, that the fastest known algorithms for solving high-multiplicity “flow-based” assignment problems run in $\Omega(mn)$ worst-case time, so 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 algorithm for the *non-bipartite* stable allocation problem, running in $O(m \log n)$ time with high probability, improving a weakly-polynomial bound of $O(m^3 \log B)$ due to Biró and Fleiner [4] (here, $B = \max_i b(i)$, with $b(i)$ describing the amount of allocation required for vertex i , just as in the stable bipartite b -matching problem). When $b(i) = 1$ for all i and an integer solution is sought, the non-bipartite stable allocation problem is known as the *stable roommates* problem, and this special case is solved by a linear-time algorithm of Irving [9]. Although an integer-valued solution may not always exist for the stable roommates problem, a half-integral solution does always exist; the problem of finding a such a half-integral solution is known as the *stable partition* problem, introduced and solved with a linear-time algorithm by Tan [15]. In the book of Gusfield and Irving on the stable marriage problem [8], 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 marriage problem. Although it remains open whether there exists a pure reduction from one problem to the other, we show that a transformation of this flavor does indeed exist, and that it simplifies the construction of algorithms not only for stable roommates/partition but also for the more general non-bipartite

stable allocation problem. It also provides a simple alternative proof of the fact that a half-integral solution always exists for the stable roommates/partition problem [15].

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. [10] 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 [5] 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 use set notation in the standard fashion to 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. Job 1 should be the final entry in each machine’s preference list, and machine 1 should be the final entry in each job’s preference list. We can regard a job or machine that ends up being assigned to a dummy as being unassigned

in our original instance. The addition of the dummy job and machine gives us a problem with $n' = n + 2$ total vertices and $m' = m + 2n$ total edges. This slight increase does not affect the $O(m' \log n') = O(m \log n)$ running time of our algorithm, so we henceforth use m and n in place of m' and n' for simplicity.

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.

Just as with the unit stable matching problem, a stable assignment always exists for any problem instance. Moreover, there always exists a unique stable assignment $x^<$ 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 $x^>$ always exists as well. As it turns out, a job-optimal assignment is always machine-pessimal and vice-versa, where a machine-pessimal assignment is such that the vector describing the allocation of each machine j (ordered according to the reversal of j 's preference list) is lexicographically maximal over all possible stable assignments. Proofs of these facts follow shortly. It is also easy to show that the dummy allocations $x(1, j)$ and $x(i, 1)$ are the same in every stable assignment; this property, and many other useful facts about stable allocation problems and their solutions, are discussed in substantial detail in [2].

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-Shapley (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. Baïou and Balinski first introduced this approach and called it the “row greedy” algorithm [2], since the jobs (left-hand elements) that propose greedily become the rows of a matrix encoding of the 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

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)
---	---

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 now 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 its output is $x^<$.*

It is well-known (see, e.g., [8]) 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 an instance I consisting of 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 I' we obtain when we split each job i into $p(i)$ unit jobs $i_1 \dots i_{p(i)}$ and each machine j into $c(j)$ unit machines $j_1 \dots j_{c(j)}$ (see also [2] for a proof along these lines, in the context of the equivalent ‘‘row-greedy’’ algorithm of Baiou and Balinski). Every appearance of machine j in a job's preference list is replaced by $j_1 \dots j_{c(j)}$, and every appearance of i in a machine's preference list is replaced by $i_1 \dots i_{p(i)}$. After computing a stable matching M in I' , we obtain an equivalent solution x in I by setting $x(i, j)$ to the number of edges in M between some job in $i_1 \dots i_{p(i)}$ and some machine in $j_1 \dots j_{c(j)}$. It is fairly easy to show that x is stable, job-optimal, and machine pessimal if M has these properties as well. Unfortunately, this reduction does not easily apply if we have irrational problem data or upper edge capacities, so different arguments are necessary in the following proof.

Proof of Lemma 1. Finite termination of the GS algorithm, even in the presence of irrational capacities, is shown in [5]. To show that our final assignment is stable, suppose at termination that (i, j) is a blocking pair. Recall that i must be fully-assigned, since this is necessary for the GS algorithm to terminate. Since $q_i <_i j$, we know j must have rejected i at some point; however, this implies that $r_j \geq_j i$, contradicting our assumption that j has any allocation it prefers less than i . We now show that our final assignment is indeed the unique job-optimal, machine-pessimal assignment $x^<$.

Suppose that the final assignment of the GS algorithm is not job-optimal, so there must be some other stable assignment x^* such that for some job i , the allocation of i in x^* is lexicographically greater than that of the final assignment reached by the GS algorithm. At some point during

execution of the GS algorithm, we must therefore encounter a rejection from some machine j to job i that results in an assignment x with $x(i, j) < x^*(i, j)$, where $x(i, j') \geq x^*(i, j')$ 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 , and this cannot be the case since $x(i', j) > x^*(i', j)$, so $x(i', j)$ must be positive. Since $x(i', j) > x^*(i', j)$ and $x(i', [J]) \leq x^*(i', [J]) = p(i')$, 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 x is machine-pessimal is symmetric to the one above. Suppose, for purposes of contradiction, that at some point during the GS algorithm machine j rejects sufficiently much from job i so that we now have $x(i, j) < x^*(i, j)$, where x^* is a stable assignment with $x(i', j) \leq x^*(i', j)$ for all $i' <_j i$. Consider the first point in time when such a rejection occurs. Since $x([I], j) = x^*([I], j)$, there must exist some job $i' >_j i$ with $x(i', j) > x^*(i', j)$. Since $x(i', [J]) \leq x^*(i', [J]) = p(i')$, there must exist some machine j' such that $x(i', j') < x^*(i', j')$. We claim that $j' <_{i'} j$. If not, then this implies that there was an earlier rejection (at machine j' , either of job i' or of some job j' prefers less than i') of the same type as above. It then follows that (i', j) is a blocking pair in x^* . Observe that our proof that the solution is job-optimal / machine pessimal remains valid even if the GS algorithm is less aggressive in its proposals — proposing less than the maximum it could ask for in certain steps. \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 [5], 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 [5].

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 algorithm of Baïou 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 $j \in [J] - \{1\}$. Note that any “slack” edges (i, j) for which $0 < x(i, j) < u(i, j)$ must belong to $G(x)$. This follows from the monotonic behavior of $x(i, j)$ during the course of the GS algorithm: $x(i, j) = 0$ initially, then $x(i, j)$ increases during the time that (i, j) is used for proposals (with $q_i = j$); this phase ends when either (i, j) becomes saturated (and no longer slack) or when j rejects i , so $i = r_j$. The value of $x(i, j)$ can then only decrease when (i, j) is used for rejections (with $i = r_j$), and this phase ends only when $x(i, j) = 0$.

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

The Main Algorithm. Our algorithm has a simple high-level description even though its low-level implementation requires attention to a number of somewhat messy details. As a result, we believe the clearest way to describe the algorithm is in prose rather than pseudocode. We say a component in $G(x)$ is *fully assigned* if $x(i, [J]) = p(i)$ for each job i in the component. In each step of our algorithm, we select an arbitrary component C of $G(x)$ that is not fully assigned, and we perform an *augmentation* within C . To store the components of $G(x)$, we use a dynamic tree data structure, described in more detail shortly. We also maintain a doubly-linked list L of references to all components that are not fully assigned, so that each iteration involves selecting an arbitrary component C from L for augmentation. As augmentations change the structure of $G(x)$ or make components fully-assigned, we will modify L accordingly. We terminate when L becomes empty, at the point when every component is fully assigned.

Augmentation. 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 $r(\pi) = \min\{r(i, j) \mid (i, j) \in \pi\}$. 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 an augmenting path/cycle π become saturated or empty, this automatically triggers any appropriate calls to our pointer management infrastructure above: any time an edge $(r_j, j) \in \pi$ becomes empty, this triggers a call to `ADVANCE-R(j)`, and any time an edge $(i, q_i) \in \pi$ becomes saturated, this triggers a call to `ADVANCE-Q(i)`. This results in a change to the structure of $G(x)$ due to advancement of one or more of the q_i or r_j pointers. 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 $G(x)$ 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. Any components we destroy are removed from L , and any components we create that are not fully assigned are appended to L . Once a component becomes fully-assigned, it is also removed from L .

Implementation with Dynamic Trees. In order to augment efficiently, we store each component of $G(x)$ in a dynamic tree data structure [13, 14]. For cycle components, we store a dynamic tree plus one arbitrary edge along the cycle. A dynamic tree is an extremely versatile data structure that maintains an edge-weighted free tree and supports the following key operations, all in $O(\log n)$ (amortized) time:

- *path-min*(T, u, v): return the minimum edge weight along the unique path through T specified by two endpoint vertices u and v .
- *path-update*(T, u, v, a): add a common value a to all edge weights along the unique path through T specified by two endpoint vertices u and v .
- *split*(T, u, v): separate one dynamic tree T into two dynamic trees by removing the edge uv .
- *join*(T_1, T_2, v_1, v_2): join two dynamic trees T_1 and T_2 by adding an edge between vertex v_1 (in T_1) and vertex v_2 (in T_2).

Our dynamic trees are also augmented so that each job vertex maintains a flag indicating if it is fully assigned, so each dynamic tree T can determine if its corresponding component is fully assigned, or alternatively produce a pointer to a non-fully-assigned job in T in $O(\log n)$ (amortized) time. We store residual capacities as edge weights in our dynamic trees. This allows us to compute the residual capacity of any augmenting path or cycle in $O(\log n)$ (amortized) time using the *path-min* operation, and to augment in $O(\log n)$ (amortized) time using the *path-update* operation. Each augmentation therefore takes $O(\log n)$ (amortized) 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, contributing a total of $O(m \log n)$ towards the overall running time.

After augmenting on a path, its residual capacity drops to zero. By calling *path-min* successively, we can identify all k of the edges along the path having zero residual capacity in only $O(k \log n)$ time. More precisely, after identifying the first edge e of zero residual capacity, we recursively call *path-min* on the two subpaths on either side of e , continuing to recurse in this manner as long as we keep discovering edges of zero residual cost. All the edges of zero residual cost have either

reached their lower or upper bounds, so we invoke appropriate calls to our pointer advancement functions ADVANCE-R and ADVANCE-Q for each such edge, as discussed previously. Any time an r_j or q_i pointer advances, recall that this modifies the structure of $G(x)$ by removing one edge and adding one edge. Using the *split* and *join* operations, we can make these modifications in $O(\log n)$ (amortized) time each, and we can also easily maintain the list L of not-fully-assigned components in the process. 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, for a grand total of $O(m \log n)$ for the entire algorithm.

Lemma 5. *The preceding algorithm outputs the solution $x = x^<$.*

Proof. Let x denote the solution our algorithm outputs at termination. We would ideally like to make the simple argument that our algorithm is performing, in an aggregate fashion, a set of proposals and rejections that the original GS algorithm *could* have performed (in which case Lemma 1 would establish that $x = x^<$). Unfortunately, this argument is somewhat problematic for the case where our algorithm augments in a cycle component whose cycle consists of fully-assigned jobs, since the GS algorithm is only capable of launching an augmentation by proposing from a *partially-assigned* job, so there is no sequence of GS operations that mimics the process of augmentation around the cycle. To fix this issue, let us modify our original instance I to construct an instance I' as follows: for each job i , let $last(i)$ denote the least-preferred machine in i 's preference list for which $x(i, last(i)) > 0$. For every $j \leq_i last(i)$ except the dummy machine, we set $u(i, j) = x(i, j)$ in I' . Finally, we increase $p(i)$ by one unit for all jobs i in I' . Observe that the job-optimal assignment for I' is identical to $x^<$, except with the one unit of extra processing time for each job being assigned to the dummy machine. Moreover, the GS algorithm applied to I' has the option of postponing the final proposal of this one unit to the dummy machine for each job, thereby converging the assignment $x^<$ while every job still has one unit of unassigned load. We therefore claim that the augmentations performed by our algorithm correspond to a set of aggregated proposals and rejections that the GS algorithm *could* have performed on I' (up until the final proposals to the dummy machine involving the last unit of load from each job in I'), so Lemma 1 implies that $x = x^<$. The difficulty with cycle components no longer manifests itself, since every job in I' is only partially assigned through the entire process of building $x^<$. \square

One might wish to think of our algorithm as a fast implementation of the BB algorithm, which performs a series of augmentations in essentially the same fashion as above (although they are described quite differently in [2], using an “inductive” approach, rather than in terms of our auxiliary graph $G(x)$). Each augmentation in the BB algorithm is implemented without sophisticated data structures, and hence requires $O(n)$ worst-case time, for a total running time of $O(mn)$ — the same running time our algorithm would have if we did not use dynamic trees (in this case, our algorithm would be more or less equivalent in structure to the BB approach). In Appendix A, we show that this running time bound without sophisticated data structures is tight, by describing an instance for which the BB algorithm runs in $\Omega(mn)$ time. The key to our running time improvement is the use of dynamic trees to augment quickly, owing to our new structural insight involving the decomposition of the $G(x)$ graph. Our approach also exposes a pleasing similarity between state-of-the-art approaches based on dynamic trees for solving our problem and the related maximum flow problem, since many of these augment using dynamic trees in a manner quite similar to our algorithm.

4 The Optimal Stable Allocation Problem

Once our algorithm from the previous section terminates with $x = x^<$, $G(x)$ 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 6. *Let x be a stable assignment with a cycle component C in $G(x)$, where π_C is the unique augmenting cycle in C . If we augment any amount in the range $[0, r(\pi_C)]$ around π_C , then this unilaterally decreases the happiness of all jobs in π_C and increases the happiness of all machines in π_C . Furthermore, each job in π_C experiences an increase in the allocation to the machine that after augmentation is least preferred among all machines with positive allocation in its preference list, and each machine in π_C experiences a decrease in the allocation to the job that before augmentation is the least-preferred job in its preference list with positive allocation.*

Proof. Consider any machine j in π_C . In terms of the edges adjacent to j in π_C , the augmentation process involves decreasing $x(r_j, j)$ and increasing $x(i, j)$ where i is the job in π_C with $q_i = j$ (note: we define the q_i 's and r_j 's with respect to x , rather than the assignment after augmentation). Since q_i can only point to a machine that is accepting of i , we have $i >_j r_j$, so our augmentation increases the happiness of j . The augmentation therefore also reduces the amount of j 's least-preferred allocation (from r_j). Now consider some job i in π_C , and let $j = q_i$ and j' be any machine other than j with $x(i, j) > 0$. We claim that $j <_i j'$, for otherwise (i, j) would be a blocking pair in x , since $x(i, j) < u(i, j)$, $i >_j r_j$ (since j is accepting of i), and $x(r_j, j) > 0$. As a consequence, i 's happiness decreases during augmentation. \square

Lemma 7. *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 .*

Proof. Suppose augmentation introduces a blocking pair (i, j) (which was not present in x , since x is stable). This can only happen if one of the following occurs during augmentation: (i) $x(i, j)$ decreases from $u(i, j)$, (ii) job i experiences an increase in its allocation to some machine $j' <_i j$, or (iii) machine j experiences an increase in its allocation to some job $i' <_j i$. In case (i), we know from Lemma 6 that $i = r_j$ (note: we define the q_i 's and r_j 's with respect to x , rather than the assignment after augmentation), so there is no job $i' <_j i$ with $x(i', j) > 0$, and hence (i, j) cannot be a blocking pair. In case (ii), we know that j is not accepting of i (otherwise we would have $q_i = j$, contradicting Lemma 6), so $r_j >_j i$ and again there cannot be a job $i' <_j i$ with $x(i', j) > 0$. Consider finally case (iii), in the event where cases (i) and (ii) do not apply. Since Lemma 6 tells us that j becomes happier during augmentation, we must have $x(r_j, j) > 0$ with $r_j <_j i' <_j i$; hence (i, j) would have been a blocking pair in x . \square

Lemma 8. *If x is a stable assignment, then $x = x^>$ if and only if $G(x)$ has no cycles (i.e., $G(x)$ consists of a single tree component).*

Proof. Clearly, $G(x^>)$ contains no cycles, since otherwise Lemma 6 would allow us to augment along such a cycle to obtain an assignment that is more machine-optimal than $x^>$. Now let x be any

stable assignment other than $x^>$, and consider some non-dummy job i that is assigned differently in x and $x^>$. Since $x^>$ is job-pessimal, we can locate some machine j such that $x(i, j') = x^>(i, j')$ for all $j' <_i j$ and $x(i, j) < x^>(i, j)$. Note that j is not the dummy machine, since all stable assignments share the same dummy allocation, so $x(i, 1) = x^>(i, 1)$. Now since $x([I], j) = x^>([I], j)$, there exists some job i' such that $x(i', j) > x^>(i', j)$. We claim that $i' <_j i$. If not, then since $x(i', [J]) = x^>(i', [J])$ and $x^>$ is job-pessimal, we can find job $j' <_{i'} j$ such that $x(i', j') < x^>(i', j')$, and this implies that (i', j) would be a blocking pair for $x^>$. Since $i' <_j i$, we know that j is accepting of i , so $q_i \geq_i j$ (note that the q_i 's and r_j 's are all defined with respect to x). If $q_i = j$ then we claim $x(r_j, j) \neq x^>(r_j, j)$. If not, then let $i'' = r_j <_j i'$, and observe that job i' is assigned differently in x and $x^>$, so we can find machine $j' <_{i'} j$ with $x(i', j') < x^>(i', j')$. From this, we conclude that (i', j) is a blocking pair in $x^>$, since $x^>(i', j') > 0$ and $x(i'', j) > 0$. Therefore, if $q_i = j$, then $x(r_j, j) \neq x^>(r_j, j)$. Now suppose $q_i >_i j$. Let $j' = q_i$ and $i'' = r_{j'} <_{j'} i$. Here, we must have $x^>(i'', j') = 0$, since otherwise (i, j') is a blocking pair in $x^>$. Hence, $x^>(i'', j) \neq x(i'', j) > 0$, so in all cases, we have $x(r_{j'}, j') \neq x^>(r_{j'}, j')$, where $j' = q_i$ is a non-dummy machine and $r_{j'}$ is a non-dummy job (since dummy allocations are always equal in different stable assignments). Stepping from i to $j' = q_i$ and then to $r_{j'}$, we repeat the argument above, noting that job $r_{j'}$ is assigned differently in x and $x^>$. Continuing in this fashion, we will eventually trace out a cycle in $G(x)$. \square

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 .

Lemma 9. *If two rotations π and π' can ever be simultaneously exposed, then π and π' must be vertex-disjoint.*

Proof. This follows easily from the fact that π and π' , when simultaneously exposed, constitute two vertex-disjoint cycles in $G(x)$. \square

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 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 10. *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.*

Proof. Let us say π is *elusive* with respect to an allocation x if, starting from x , π is either encountered by some rotation elimination orderings but not others, or if it appears with different residual capacities in different elimination orderings. For purposes of contradiction, suppose that some rotation π is elusive with respect to $x = x^<$. For any rotation π' exposed by x , we say π' is π -*mixed* if π remains elusive in all elimination orderings that start with π' ; otherwise, we say π' is π -*pure*. We repeatedly eliminate π -mixed rotations from x while still preserving property that π is

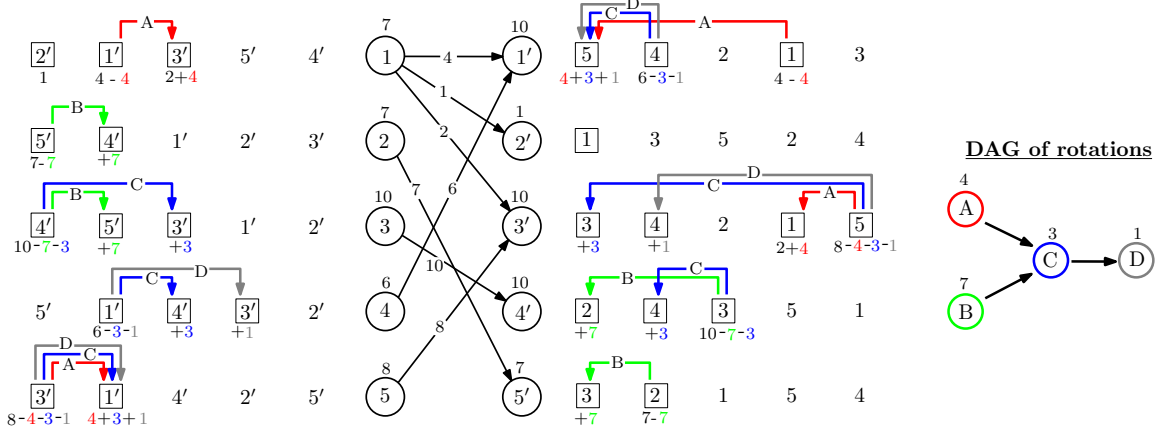


Figure 1: An example of a bipartite instance and its rotations. No dummy job or machine is shown, since once we reach the job-optimal assignment neither of these takes part in any further augmenting cycles (rotations). The initial job-optimal assignment $x^<$ is shown in the bipartite region, and rotations $A \dots D$ and their associated multiplicities are shown in the DAG on the right. A preference list entry is outlined if it takes part in some stable solution (these outlined entries are sometimes known as *stable partners* for the unit stable matching problem). Underneath the entries in the preference lists, we show how the original job-optimal assignment changes in response to each rotation; this is also indicated with arrows atop the preference lists. For example, the initial assignment of job 1 sends 4 units to machine $1'$ and 2 units to machine $3'$ (machines are indicated with primes to help distinguish them from jobs). Rotation A , when fully eliminated, decreases 4 units of assignment from job 1 to machine $1'$, and increases 4 units of assignment from job 1 to machine $3'$.

elusive with respect to x . After doing so, there must be two exposed π -pure rotations π_1 and π_2 such that rotation elimination orderings starting with π_1 and π_2 do not reach π in the same state. Note that π_1 and π_2 are vertex-disjoint according to Lemma 9. Hence, we arrive at the same stable solution if we apply π_1 then π_2 , or if we apply π_2 then π_1 . This implies that we must reach π in the same state irrespective of whether we start with π_1 or π_2 , a contradiction. \square

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 (Lemma 9), 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. In fact, we will shortly prove that non-vertex-disjointness is the basis for all precedence relationships between rotations in Π . 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 $x^<$, then we continue running it until we have generated the set of all rotations Π (transforming $x^<$ into $x^>$ in the process). 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' .

Lemma 11. *The transitive closure of D' is D .*

Proof. Clearly the transitive closure of D' is a subgraph of D . For the purposes of contradiction, suppose there are two rotations π_i and π_j such that $(\pi_i, \pi_j) \in E$ but there is no $\pi_i \rightsquigarrow \pi_j$ path in D' . Starting from $x^<$, let us eliminate exposed rotations according to an arbitrary ordering, except let us not eliminate π_i if it becomes exposed. Since $(\pi_i, \pi_j) \in E$, this process must terminate before reaching $x^>$, since π_j will never become exposed. At termination, π_i will be the only exposed rotation. We now eliminate π_i , exposing a set of rotations S , none of which are vertex-disjoint with π_i , so $(\pi_i, \hat{\pi}) \in E'$ for every $\hat{\pi} \in S$. Hence, $\pi_j \notin S$, since otherwise we would have $(\pi_i, \pi_j) \in E'$, implying the existence of a $\pi_i \rightsquigarrow \pi_j$ path in D' . However, since $(\pi_i, \pi_j) \in E$, there must be some $\hat{\pi} \in S$ with $(\hat{\pi}, \pi_j) \in E$ and no $\hat{\pi} \rightsquigarrow \pi_j$ path in D' . We now repeat our argument, by setting $\pi_i = \hat{\pi}$ (i.e., eliminate rotations until only π_i is exposed, then eliminate π_i , etc.) Eventually, we must reach a contradiction where $\pi_j \in S$. \square

The (reduced) rotation DAG has been instrumental in the unit case (see, e.g., [8]) 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.

Lemma 12. *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 .*

Proof. Given a *D-closed* vector y , we generate a stable assignment x by starting with $x^<$ and applying each rotation $\pi \in \Pi$ to the extent $y(\pi)$, according to a topological ordering of D . According to Lemma 7, we preserve stability during the entire process. Now consider any stable solution x . Transform this solution into $x^>$ by repeatedly eliminating exposed rotations (Lemma 8 ensures that these always exist as long as we have not yet reached $x^>$). Let $y' \in \mathbf{R}^{|\Pi|}$ describe the amount of each rotation $\pi \in \Pi$ we apply during the process. The application of the rotations described by $y = r - y'$ starting with $x^<$ gives us x , and y is *D-closed* since y' describes the complement of a *D-closed* set. \square

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 12, 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 always exists that corresponds with 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 (i.e., the sum of $r(\pi)c(i, q_i)$ over all (i, q_i) edges in π minus the sum of $r(\pi)c(r_j, j)$ over all (r_j, j) edges in π), we can transform the optimal stable allocation problem into an equivalent minimum-cost closure problem on the DAG D' . For further details, see [10]. The running time of our approach is the same as that in [10], $O(m^2 \log n)$, since the closure computation (the dominant part of the running time) takes place on a DAG with $O(m)$ vertices and $O(m)$ edges, just as with the simpler unit stable matching problem.

Finally, note that rotations never involve the dummy job or machine. This gives us a simple means of showing that the dummy allocation $x(i, 1)$ of each job i and the dummy allocation $x(1, j)$ of each machine j remain the same in all stable assignments. For further structural results pertaining to stable assignments, we refer the reader to [2].

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 $\delta(v)$ denote the set of edges incident to v , our goal is to compute an assignment $x \in \mathbf{R}^m$ with $x(\delta(v)) = b(v)$ for all $v \in V$ and $x(e) \in [0, u(e)]$ for all $e \in E$ 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 [9]. As a consequence of the integrality restriction, one can construct instances that have no stable integer-valued solution for the roommates problem. To remedy this, Tan [15] introduced the related *stable partition* problem, which allows for non-integral solutions (specifically, half-integer solutions, which do always exist for this problem). We view the non-bipartite stable allocation problem as being inherently a real-valued problem, so we do not require an integer-valued solution; 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 [8] on whether or not there exists a reduction 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 the construction of algorithms not only for stable roommates but also for the non-bipartite stable allocation problem. Our transformation is not a “perfect” reduction since it results in a stable matching problem with an extra side constraint, so it does not quite resolve the full conjecture of Gusfield and Irving.

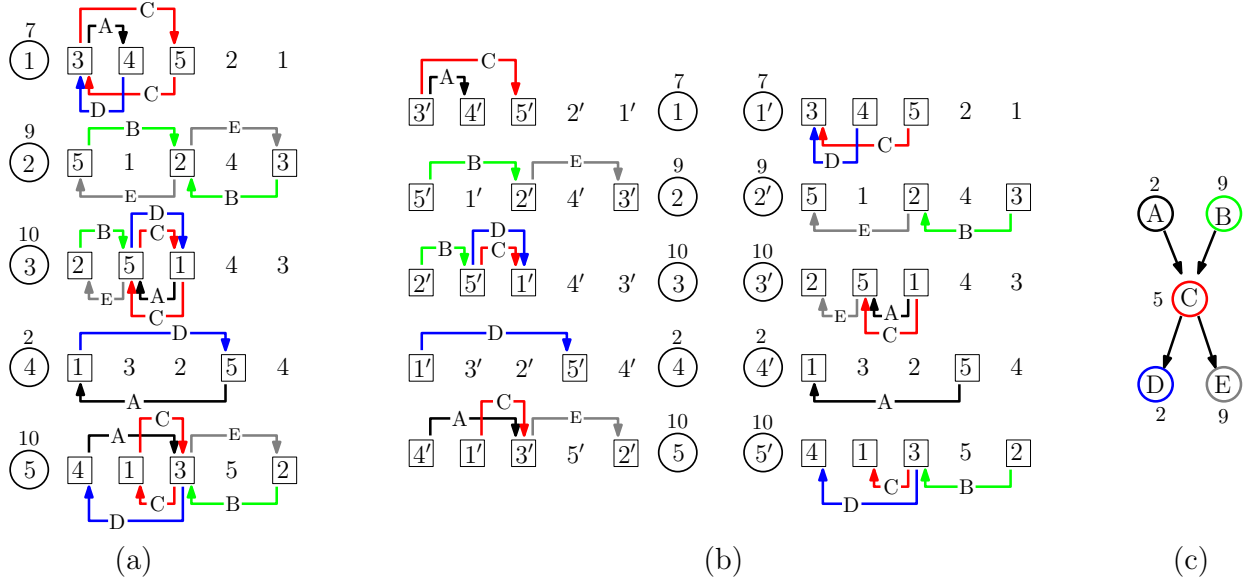


Figure 2: Transforming a non-bipartite instance, shown in (a), into a symmetric bipartite instance, shown in (b). In (a), we see the preference lists of 5 vertices, whose sizes (7, 9, 10, 2, 10) are also indicated. By essentially duplicating this instance, we obtain the symmetric bipartite in (b), drawn in the same style as Figure 1. Observe that each vertex has been split into both a job and a machine, each of the same size as the original vertex, and each having the same preference list as the original vertex. The rotation DAG for the instance in (b) is shown in (c), along with the multiplicity of each rotation. As in Figure 1, we also show with arrows above the preferences lists what is the effect of each rotation on the assignment in both the bipartite instance. We have also overlaid these arrows from both the right and left hand sides of the bipartite instance on top of the original non-bipartite instance, so that duality relationships are apparent. A symmetric stable assignment for the bipartite instance is obtained by eliminating A , B , and half of C .

However, we believe it may be the closest such result one can probably achieve in addressing the spirit of the conjecture. Suppose we construct a symmetric bipartite instance I' by replicating a non-bipartite instance I , as shown in Figure 2. More precisely, each vertex $v \in V$ in I is transformed into both a job i_v and a machine j_v in I' , both of whose preference lists are identical, and the same as the original preference list of v . For example, the preference list of v contains vertices 2, 5, and 3 in that order, then the preference list of i_v will be j_2, j_5 , and j_3 , and the preference list of j_v will be i_2, i_5 , and i_3 . We also set $p(i_v) = c(j_v) = b(v)$. Observe that if we reverse the roles of the jobs and machines, our instance remains unchanged. If we can now find a *symmetric* stable assignment x' for I' (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 I . 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.

Lemma 13. *Let I be an instance of the non-bipartite stable allocation problem, and let I' be the symmetric bipartite instance obtained by replicating I . There is a one-to-one correspondence between stable solutions x of I and symmetric stable solutions x' of I' .*

Proof. We transform between x and x' by setting $x'(u, v) = x'(v, u) = x(e)$ for every edge $e = uv$

in I' . Note that if x is feasible for I , then so is x' for I' , and vice-versa. To show that stability of x implies stability of x' , suppose x' admits a blocking pair (u, v) in I' on account of $a <_u v$ with $x'(u, a) = x'(a, u) > 0$ and $b <_v u$ with $x'(v, b) = x'(b, v) > 0$. In I , this implies that edge uv is a blocking edge, since $x(ua) > 0$ and $x(vb) > 0$, and u and v respectively prefer each-other to a and b . Likewise, if there is a blocking pair uv in x , this translates into a blocking pair in x' using exactly the same correspondence. \square

One can always find a symmetric solution to a symmetric bipartite instance by carefully choosing the right combination of rotations to apply to $x^<$, which is now the “mirror image” of $x^>$ in that $x^<(i, j) = x^>(j, i)$. 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. A rotation satisfying $\pi = \pi^*$, such as rotation C , is called a *self-dual* rotation.

Lemma 14. *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^*) . (in particular, this implies that $y(\pi) = r(\pi)/2$ for every self-dual rotation).*

Proof. For any (job, machine) pair (i, j) in our bipartite instance, let $\Pi^+(i, j)$ denote the set of rotations in Π whose application causes $x(i, j)$ to increase, and let $\Pi^-(i, j)$ denote the set of rotations whose application causes $x(i, j)$ to decrease. Note that if $\pi \in \Pi^+(i, j)$, then $\pi^* \in \Pi^-(j, i)$ due to symmetry. Now suppose $y(\pi) + y(\pi^*) = r(\pi)$ for all $\pi \in \Pi$. Letting x denote the assignment obtained by applying the rotations described by some D -closed vector y (so x must be stable), we have

$$\begin{aligned}
x(j, i) &= x^<(j, i) + \sum_{\pi \in \Pi^+(j, i)} y(\pi) - \sum_{\pi \in \Pi^-(j, i)} y(\pi) \\
&= x^<(j, i) + \sum_{\pi \in \Pi^-(i, j)} y(\pi^*) - \sum_{\pi \in \Pi^+(i, j)} y(\pi^*) \\
&= x^<(j, i) + \sum_{\pi \in \Pi^-(i, j)} (r(\pi) - y(\pi)) - \sum_{\pi \in \Pi^+(i, j)} (r(\pi) - y(\pi)) \\
&= x^<(j, i) + (x(i, j) - x^<(i, j)) + \sum_{\pi \in \Pi^-(i, j)} r(\pi) - \sum_{\pi \in \Pi^+(i, j)} r(\pi) \\
&= x(i, j),
\end{aligned}$$

where the penultimate step follows from

$$x(i, j) = x^<(i, j) + \sum_{\pi \in \Pi^+(i, j)} y(\pi) - \sum_{\pi \in \Pi^-(i, j)} y(\pi),$$

and the last step follows from the fact that

$$x^<(j, i) = x^>(i, j) = x^<(i, j) + \sum_{\pi \in \Pi^+(i, j)} r(\pi) - \sum_{\pi \in \Pi^-(i, j)} r(\pi).$$

Our assignment x is therefore symmetric. To prove the other direction, suppose x is a symmetric stable solution. Let y denote the D -closed vector describing the rotations needed to obtain x from $x^<$, and let us define $y^*(\pi) = y(\pi^*)$. Due to symmetry, it must be the case that $r - y = y^*$, since the application of the rotations described by y^* in reverse starting from $x^>$ should also give us x (since this is completely symmetric to the original forward process). Hence, $y + y^* = r$, so $y(\pi) + y(\pi^*) = r(\pi)$ for all $\pi \in \Pi$. \square

Lemma 15. *A symmetric stable assignment always exists for any symmetric stable allocation instance. Moreover, this solution corresponds to a D -closed (D' -closed) vector y with $y(\pi) = r(\pi)/2$ for every self-dual rotation π , and with $y(\pi) \in \{0, r(\pi)\}$ for all other rotations.*

Proof. Suppose we start with the assignment $x = x^<$ and begin eliminating rotations according to an arbitrary ordering, taking care to eliminate only half of any self-dual rotation and also not to eliminate the dual of any previously-eliminated rotation. We claim that this process yields the desired solution. For purposes of contradiction, suppose not. In this case, the preceding algorithm will terminate with some rotation π_i satisfying $y(\pi_i) = y(\pi_i^*) = 0$, since neither π_i nor π_i^* has been exposed yet. This implies that there exists another rotation π_j satisfying $y(\pi_j^*) > 0$ and $\pi_j \prec \pi_i$ (i.e., π_j blocks the exposure of π_i , and we cannot apply π_j since we have already applied π_j^*). However, since $\pi_j \prec \pi_i$, symmetry gives us $\pi_i^* \prec \pi_j^*$, meaning that we must have already fully applied π_i^* , contradicting $y(\pi_i^*) = 0$. \square

This lemma gives another simple proof of the (previously-known) fact that a half-integral solution always exists for the stable roommates problem. In fact, similar techniques are used in the combinatorial optimization folklore to prove half-integrality of other problems. For example, if we take a matching problem on a graph $G = (V, E)$, the polyhedron $\mathcal{P} = \{x \in \mathbf{R}^{|E|} \mid x(\delta(v)) = 1 \text{ for all } v \in V, x(e) \geq 0 \text{ for all } e \in E\}$ can be shown to have half-integral extreme points using this approach: since \mathcal{P} is known to be integral when G is bipartite, we can form a bipartite graph G' by replicating G (as we do above), find an integral solution x for G' , and then obtain a half-integral solution for G by averaging x with its mirror image.

Lemma 15 leads us to an efficient algorithm for the non-bipartite stable allocation problem. Starting with our symmetric bipartite instance, we compute $x^<$ and start eliminating exposed rotations, taking care not to eliminate the dual of any previously-eliminated rotation, and eliminating only half of each self-dual rotation. To do this effectively, we need to be able to quickly identify for each rotation π we encounter during elimination whether π is self-dual, and if not, whether we have already eliminated π^* . Self-duality of π can be checked in $O(n)$ time by verifying that the structure of π is symmetric, and we can check whether π^* has been previously eliminated by maintaining an explicit list of all previously-eliminated rotations. However, the need to check every new rotation we encounter against the entries in this list will dramatically increase our running time (although it will still be strongly polynomial). Checking these conditions efficiently seems to be somewhat of a challenge.

In order to obtain a total running time of $O(m \log n)$ with high probability, we use an approach based on hashing. Before running our algorithm, we assign a random integer *weight* $w(i, j)$ independently to every edge (i, j) with $i \leq j$, chosen uniformly from the range $[1, W]$, with $W = (2m)^{c+2}$, and $c \geq 1$ being an arbitrarily-chosen constant. For each edge (i, j) with $i > j$, we set $w(i, j) = w(j, i)$. It is a simple matter to augment our dynamic trees so that in $O(\log n)$ amortized time, the total

weight $w(\pi)$ of the edges in each rotation π can be computed when π is considered for elimination. Note that $w(\pi) = w(\pi^*)$ due to symmetry.

Lemma 16. *Let us call two rotations π_i and π_j unrelated if $\pi_i \neq \pi_j$ and $\pi_i^* \neq \pi_j$. Let E be the event that $w(\pi_i) \neq w(\pi_j)$ for all pairs (π_i, π_j) of unrelated rotations (i.e., that we have no hashing collisions). Then $\Pr[E] \geq 1 - 1/(2m)^c$.*

Otherwise stated, E occurs with high probability, since the standard notion of “with high probability” in the computer science literature requires that $1 - \Pr[E]$ be the reciprocal of a polynomial function of the problem size.

Proof. Consider a specific pair (π_i, π_j) of unrelated rotations, where e is an edge in $\pi_i - \pi_j$. Of all the W possible settings for $w(e)$, at most one can make $w(\pi_i) = w(\pi_j)$. Therefore, $\Pr[w(\pi_i) = w(\pi_j)] \leq 1/W$. Taking a union bound over all pairs of unrelated rotations (at most $(2m)^2$ such pairs), we see that $1 - \Pr[E] \leq (2m)^2/W = 1/(2m)^c$. \square

Since E occurs with high probability, we need only show that our running time will be $O(m \log n)$ in the event that E occurs. If E does not occur, and as a result we fail to compute a symmetric solution, then we can detect this fact (by checking for symmetry at termination) and re-run our algorithm accordingly. The remainder of the algorithm is as follows: starting from $x = x^<$, we eliminate all rotations according to an arbitrary ordering until we reach $x = x^>$. For each rotation π we eliminate, we insert its weight $w(\pi)$ into a balanced binary search tree T ; if this weight is already present, remove it from T instead. The total time required so far is $O(m \log n)$. If E occurs, then after this step T will contain the weights of precisely the self-dual rotations, since all other rotations will have been deleted upon encountering their duals. We now start again with $x = x^<$ and eliminate rotations using an arbitrary ordering, eliminating only half of a rotation π if we discover that $w(\pi)$ is present in T . We also insert the weights of the rotations eliminated thus far in a second balanced binary search tree T' , so that we can avoid eliminating any rotation π where $w(\pi)$ appears in T' (thereby preventing the elimination of the dual of any previously-eliminated rotation). This elimination step also requires a total of $O(m \log n)$ time. If our final solution ends up symmetric (as it will if E occurs), then we are done; otherwise, we restart the algorithm from scratch. The result is an algorithm for the non-bipartite stable allocation problem running in $O(m \log n)$ time with high probability.

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. Given an instance I of the optimal stable roommates problem, we can reduce it to an instance I' of the optimal non-bipartite stable allocation problem by applying an infinitesimal perturbation to its costs — this makes the cost of each non-self-dual rotation and its dual slightly different, thereby ensuring that if an integer-valued solution x is optimal for I , then x is the unique optimal solution for I' (otherwise, there could be infinitely-many real-valued optimal solutions for I' , by taking any rotation π not satisfying $\pi \prec \pi^*$, and by applying an α fraction of π and a $1 - \alpha$ fraction of π^* , where $\alpha \in [0, 1]$).

Although the optimal non-bipartite stable allocation problem is NP-hard, we can obtain a 2-approximation algorithm for it using exactly the same methods as have been used to obtain a 2-approximation algorithm for the optimal stable roommates problem. Note that any optimal solution to the non-bipartite stable allocation problem must use exactly half of each self-dual rotation; after we commit to this, the remainder of the solution can always be chosen so that

we apply all or none of each remaining rotation in order to obtain an optimal D -closed solution containing one rotation from each dual pair (and this is *exactly* the same problem we face when solving the optimal stable roommates problem). This problem can be reduced to a weighted 2SAT problem, from which a 2-approximation algorithm is then derived; see [8] for further details.

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] P. Biró and T. Fleiner. Integral stable allocation problem on graphs. Technical Report TR-2008-290, University of Glasgow, Department of Computing Science, 2008.
- [5] 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.
- [6] D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–14, 1962.
- [7] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
- [8] D. Gusfield and R.W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.
- [9] R.W. Irving. An efficient algorithm for the “stable roommates” problem. *Journal of Algorithms*, 6(4):577–595, 1985.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
- [14] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
- [15] J.J.M. Tan. A necessary and sufficient condition for the existence of a complete stable matching. *Journal of Algorithms*, 12:154–178, 1991.

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 17. *For the instance 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 17. 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 $x(i, j'_i) = n - ((p(\{i+1, \dots, i_0-1\}) + x(i_0, \lfloor n/2 \rfloor)) \bmod n)$, which we rearrange to obtain $n - x(i, j'_i) \equiv p(i+1) + K \pmod{n}$, where $K = p(\{i+2, \dots, i_0-1\}) + x(i_0, \lfloor n/2 \rfloor)$. 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,

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

□