8-2007

FASTER ALGORITHMS FOR STABLE ALLOCATION PROBLEMS

Siddharth Munshi
Clemson University, smunshi@clemson.edu

FASTER ALGORITHMS FOR STABLE ALLOCATION PROBLEMS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

by
Siddharth Munshi
August 2007

Accepted by:
Dr. Brian C. Dean, Committee Chair
Dr. David Jacobs
Dr. James Wang

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 an bipartite instance with n nodes 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 that introduces a new and useful transformation from non-bipartite to bipartite instances. We also 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. Finally, we highlight some important connections between the stable allocation problem and the maximum flow problem.

ACKNOWLEDGMENTS

This thesis is based on a paper I wrote with my adviser Dr. Brian Dean, submitted to "Symposium On Discreet Algorithms 2007". I am indebted to him for his invaluable insights, guidance and encouragement without which I would never have been able to accomplish this work. I would like to thank Dr. Jacobs and Dr. Wang for gracing my committee with their presence. I would also like to thank my parents, and my girlfriend, for their constant support, encouragement and love, throughout the course of this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION

The classical stable matching (marriage) problem has been extensively studied since its introduction by Gale and Shapley in 1962 [11]. 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 [12; 17]).

In this thesis 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 [20] 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 may 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 [6]. Baiou and Balinksi (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 running time $O(mn)$ on a bipartite instance with n vertices and m edges, and this is tight. In this thesis 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. Our algorithm bears a strong resemblance to "augmenting path" algorithms for the maximum flow problem based on dynamic trees, and in the final chapter we show that indeed the stable allocation problem and the maximum flow problem share much in common. 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.

Using our bipartite algorithm as a building block, 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 [12], 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 (note that "integer" versus "fractional" solutions are not an issue for the more general non-bipartite stable allocation problem, since it is inherently a real-valued problem).

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. [14] 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.

The remainder of this thesis is structured as follows. In Chapter 2, we introduce notation and definitions and summarize the GS algorithm along with some useful properties it satisfies. We then describe our new algorithm for the stable allocation problem in Chapter 3, and in the process show how it builds upon the BB algorithm. The optimal stable allocation problem is discussed in Chapter 4, and the non-bipartite problem in Chapter 5. Finally, we highlight some useful connections between the stable allocation problem and the maximum flow problem in Chapter 6.

Chapter 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 $[I] = \{1, \ldots, I\}$ to J machines indexed by $[J] = \{1, \ldots, J\}$. Each job i has an associated processing time $p(i)$, each machine j has a capacity $c(j)$. The jobs and machines comprise the left and right sides of a bipartite graph with $I + J = n$ nodes 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 [6] 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. Figure 2.1 gives an instance of the stable allocation problem with five jobs and four machines.

Letting $x(i, j)$ denote the amount of job i assigned to job 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

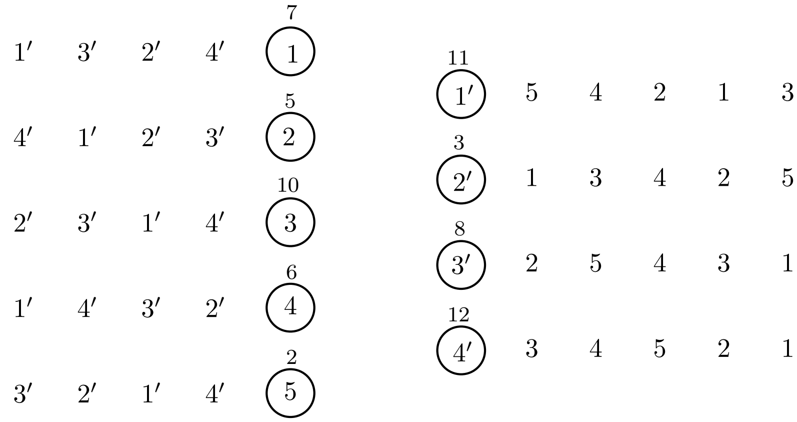


Figure 2.1: An instance of the stable allocation problem with five jobs and four machines. This figure does not include the “dummy" nodes, which can be added to ensure *feasible* assignment.

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 $p(1) \geq c([J] - \{1\})$ and $p([I]) = c([J])$. Job 1 should appear last on every machine’s preference list, and machine 1 should appear last on every job’s preference list. The preference list of job 1 consists of machine 1 followed by the remaining machines in arbitrary order, and likewise the preference list of machine 1 starts with job 1 and then contains the remaining jobs in arbitrary order. 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.

One can show that a stable assignment exists for any problem instance. Moreover, there always exists a 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. We can similarly define a *machine-optimal* assignment, which by symmetry 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

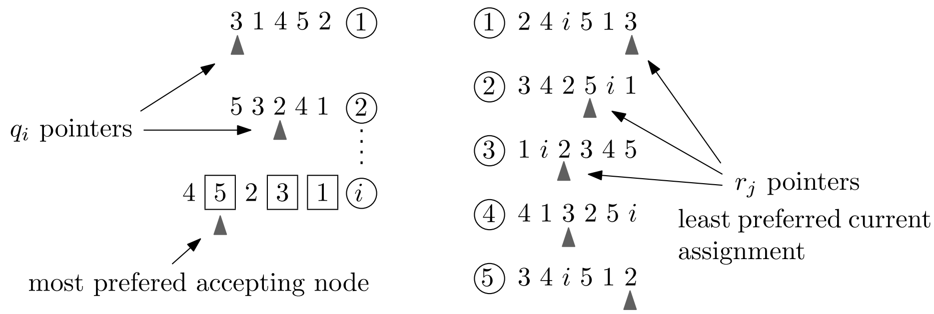


Figure 2.2: For a job i, the machines in boxes are willing to accept allocation from i. q_i is the most preferred machine among them. For a machine j, r_j is the least preferred job, j is currently assigned to.

fact that the dummy allocations $x(1, j)$ and $x(i, 1)$ are the same 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. Figure 2.2 gives a visual picture of the q_i and r_j pointers.

2.1 The Gale-Shapley (GS) Algorithm

The Gale-Shapley (GS) algorithm for the stable allocation problem is a natural generalization of the well-studied GS 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.

The GS algorithm typically starts with an empty assignment. In our case, we start with an assignment x where every machine $j > 1$ 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 remains fully assigned henceforth. 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.

ADVANCE-Q(i):
While $x_{ij} = u_{ij}$ or q_i is 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 2.1: Pseudo code to advance Q and R pointers.

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 ($x_{ij} = u_{ij}$). 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., [12]) 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 [6]. 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. $\square$

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, as we see in the Appendix.

Chapter 3

AN IMPROVED “AUGMENTING PATH” ALGORITHM

In this chapter, we describe our $O(m \log n)$ algorithm for the stable allocation problem. Along the way, we also introduce the algorithm of Baiou and Balinski (BB) on which our approach is loosely based, and we also show that our approach can be viewed as an end-to-end variant of the GS algorithm that is highly similar to an “augmenting path” algorithm for the maximum flow problem. We begin with some simple structural properties, and then show how to build our algorithm on top of the existing GS algorithm.

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 [6], 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. □

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.

At any given point in time during the execution of the GS algorithm (say, where we have built up some 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 “fractionally-assigned” edges (i, j) for which $0 < x(i, j) < u(i, j)$ must belong to $G(x)$. Initially $G(x)$ is a tree, containing n nodes, $n-1$ 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 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 $G(x)$ along with a flag for each component indicating whether or not it is fully assigned.

Suppose now we select an arbitrary job i in the tree component of $G(x)$ that is not fully assigned and consider the entire chain of events that will transpire as a result of job i proposing to machine $j = q_i$. First, machine j will reject job $i' = r_j$. Later, job i' will propose to machine $j' = q_{i'}$, which will reject job $i'' = r_{j'}$, and so on. The q_i and r_j pointers basically trace out what we could call an augmenting path, which ends with a proposal to machine 1 (the only machine that accepts without issuing a subsequent rejection). In terms of $G(x)$, we can describe this augmenting path very simply as the unique path from i to the root through the tree component. In a cycle component of $G(x)$, the unique cycle also defines what we could call an augmenting cycle, around which we can modify our assignment by performing an alternating series of proposals (on (i, q_i) edges) and rejections (on (r_j, j) edges).

We define the *residual capacity* of an edge 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

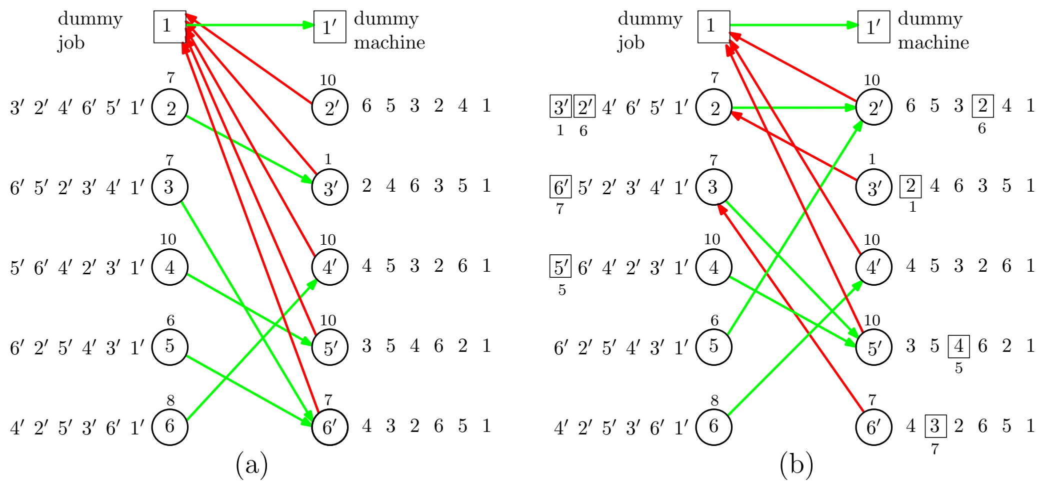


Figure 3.1: The graph $G(x)$ which consists of the q_i and r_j pointers. (a) At the beginning $G(x)$ is a tree as all r_js point to the dummy job. (b) The Graph $G(x)$ after job 2, job 3 and 5 units of job 4 have been introduced.

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 advancement infrastructure above, resulting in a change to the structure of $G(x)$ any time one of our 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 each iteration of our algorithm, we select an arbitrary component C of $G(x)$ that is not fully assigned and perform an augmentation. If C is a cycle component, then we augment along the unique cycle in C, and if C is the tree component, we augment starting from a job i in C that is not fully assigned. We terminate when every component is fully assigned.

Figure 3.1(a) shows the state of the graph $G(x)$ at the beginning of the algorithm. All the r_j pointers (the "rejecting" pointers) point to the dummy job (job 1), because the dummy job is assigned to all the machines. The q_i pointers (the “proposing" pointers) point to the first machines on their preference lists. Since every machine is fully assigned to the dummy job (which is last on it’s preference list), it is accepting for any job that proposes to it.

In order to augment efficiently, we store each component of $G(x)$ in a dynamic tree data structure (see [27; 28]). 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 to 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 $O(m \log n)$.

Correctness of our algorithm is quite simple to argue since we are performing in an aggregate fashion a set of proposals and rejections that the original GS algorithm *could* have performed. Since proposal order is irrelevant for the GS algorithm, Lemma 1 tells us that our algorithm must terminate with a stable assignment that is job-optimal and machine-pessimal.

Finally, we note that one can interpret the algorithm of Baiou and Balinski [2] as a variant of our approach where each augmentation is performed in $O(n)$ time without the use of sophisticated data structures. However, we stress that the original exposition in [2] is considerably different from ours, and it introduces none of the simplifying insight above involving our graph $G(x)$ and how its structure changes over the course of the algorithm (in fact, no mention of the exact $O(mn)$ running time of the BB algorithm appears in [2]; the authors simply claim the running time is polynomial). In Appendix A, we show that the $O(mn)$ running time bound for the BB algorithm is tight.

Chapter 4

THE OPTIMAL STABLE ALLOCATION PROBLEM

Once our algorithm from the previous chapter 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. Figure 4.1 shows the state of $G(x)$ after a job-optimal assignment is reached for the example in Figure 3.1. Note that there are two disjoint cycle components, hence two rotations are exposed.

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 augment any amount in the range $[0, r(\pi_C)]$ around π_C.

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

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.

When $r(\pi_C)$ units are augmented on the rotation π_C, hence saturating or emptying out one of the edges, at-least one of the pointers (q_i or r_j) on the rotation nodes moves ahead

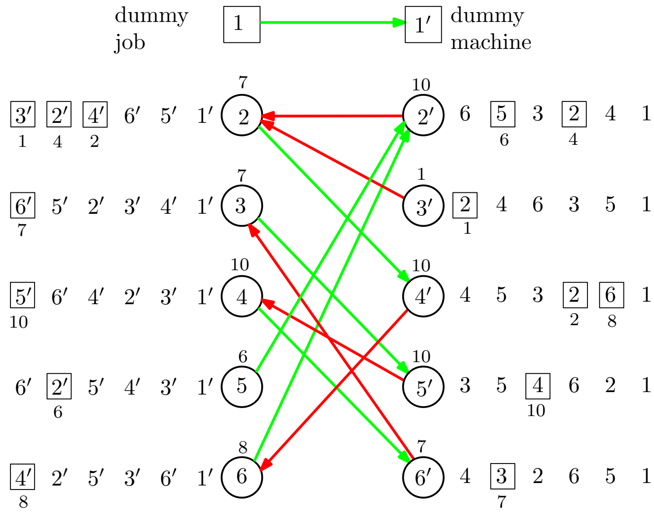


Figure 4.1: The Graph $G(x)$ after the job optimal solution has been reached. It is divided into three disjoint components - tree component: [1, 1′], cycle component 1: [2, 2′, 3′,4′,5,6] and cycle component 2: [3, 4, 5′, 6′].

and points to some other node. This breaks the cycle, and may create one or more new edges, which may now form cycles with other nodes.

Suppose we start with a job-optimal assignment and continue running the same algorithm from the previous chapter to eliminate all rotations we encounter, in some arbitrary order, until we finally reach 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 that we encounter in some rotation elimination ordering, staring with a job-optimal assignment, and ending with a machine-optimal assignment. Then π appears with the same residual capacity r in *every* such elimination ordering.

Since each elimination saturates or empties an edge, we conclude that there are at most $2m$ combinatorially-distinct rotations that can ever exist 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 rotations are vertex disjoint it must be the case that $\pi_i \prec \pi_j$ or $\pi_j \prec \pi_i$. In fact, we can extend this argument to show that for any job i (machine j) we must have $\pi_1 \prec \pi_2 \prec \ldots \prec \pi_k$, where $\pi_1 \ldots \pi_k$ are the rotations containing job i (machine j), appropriately ordered. Clearly, Π contains no cycle $\pi_1 \prec \pi_2 \prec \ldots \prec \pi_k \prec \pi_1$, since otherwise none of the rotations $\pi_1 \ldots \pi_k$ could ever be exposed. Let us therefore construct a directed acyclic graph $D = (\Pi, E)$ where $(\pi_i, \pi_j) \in E$ if $\pi_i \prec \pi_j$.

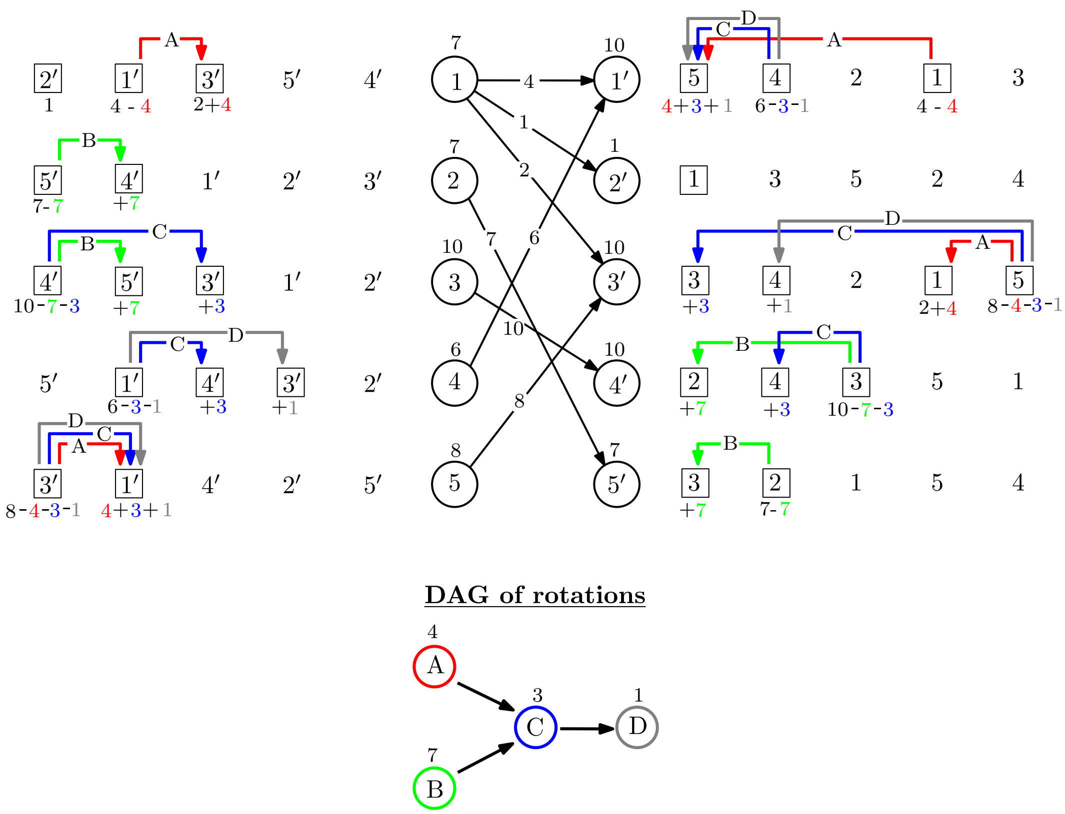


Figure 4.2: An example bipartite instance and its rotations. No dummy job or machine is shown, since once we reach the job-optimal assignment neither of these takes part in any further augmenting cycles (rotations). The initial job-optimal assignment is shown in the bipartite region, and rotations $A \ldots D$ and their associated multiplicities are shown in the "reduced" DAG D' at the bottom. The net effect of each rotation on our assignment is shown with a set of arrows over the preference lists.

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 (An example of this reduced *rotation DAG* D' is shown in Figure 4.2). To do this, we run our algorithm from Chapter 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. Then, we 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 $\pi_1 \ldots \pi_k$ containing i (j), ordered according to the order in which they were eliminated. We then add the $k-1$ edges $(\pi_1, \pi_2) \ldots (\pi_{k-1}, \pi_k)$ to E'.

The (reduced) rotation DAG has been instrumental in the unit case (see, e.g., [12]) 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 $y \in R^{|\Pi|}$ *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.

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 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 which minimizes $\sum_{ij} x(i, j)c(i, j)$. 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 $\pi \in \Pi$ a 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', which in turn is easily transformed into a maximum flow problem; for further details, see [14].

Chapter 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 $v \in V$, 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 amount of each vertex ending up unassigned must be the same in every stable assignment.

In response to an open question posed by Gusfield and Irving [12] 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 the construction of algorithms not only for stable roommates but also for the non-bipartite stable allocation problem. Suppose we construct a symmetric bipartite

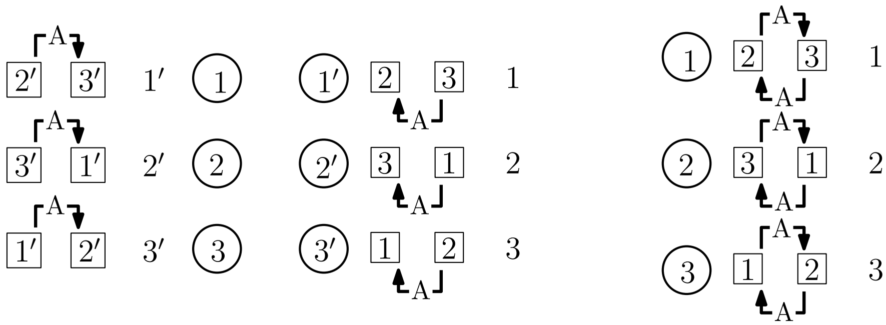


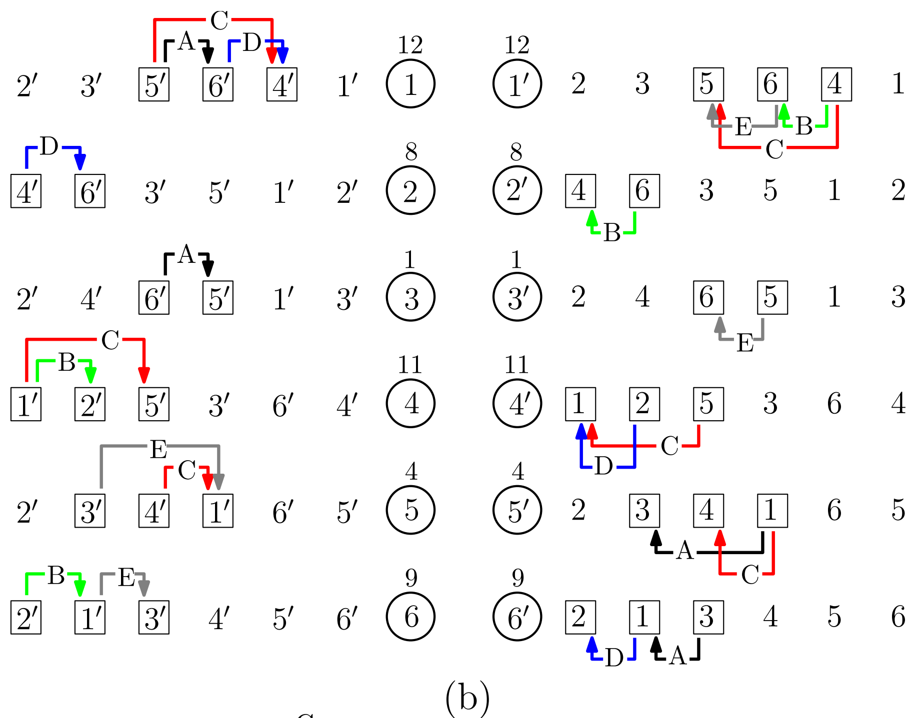
Figure 5.1: Unit case problem instance with no solution. Rotation A is its own dual.

instance by replicating a non-bipartite instance, as shown in Figure 5.2. 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 $x(e) = 2x(u,v) = 2x(v,u)$ we will obtain a stable solution to the non-bipartite instance. 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. As it turns out, one can always do this by carefully choosing the right combination of rotations to apply, starting from a job-optimal assignment.

Due to symmetry, rotations in our bipartite instance now tend to come in pairs. In the example shown in Figure 5.2(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 E are mirror images, or *duals*, of each-other, as are rotations B and D. 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 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 $O(m \log n)$



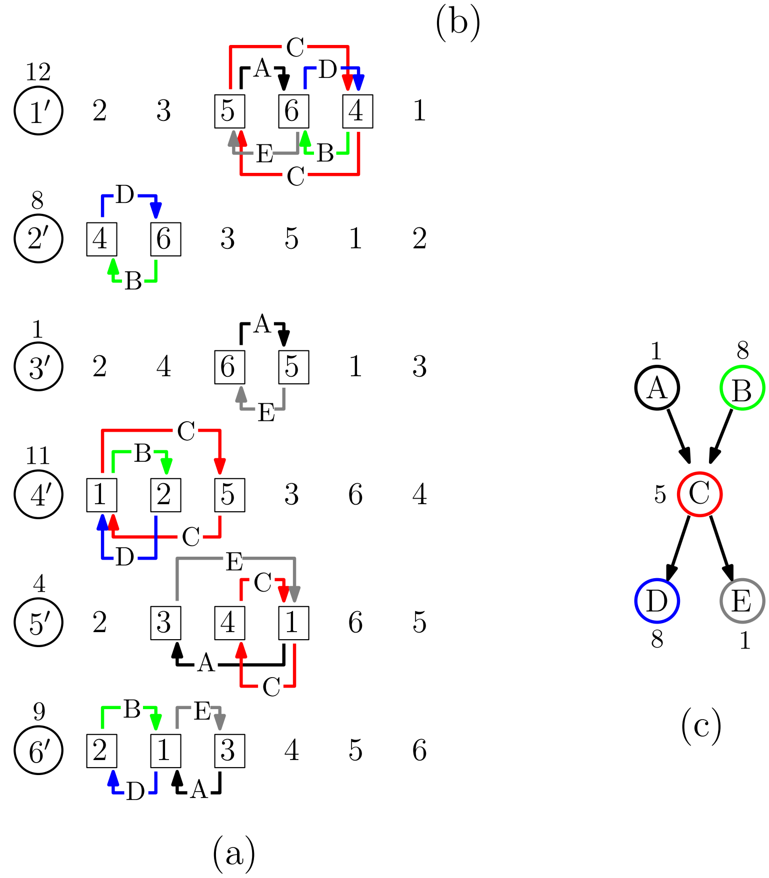


Figure 5.2: Transforming a non-bipartite instance (a) into a symmetric bipartite instance (b) with rotation DAG (c). Rotations in the resulting bipartite instance are shown overlaid on the non-bipartite instance. A symmetric stable assignment for the bipartite instance is obtained by eliminating A, B, and half of C.

algorithm for the non-bipartite stable allocation problem: transform into a symmetric bipartite instance, compute a job-optimal stable allocation, then eliminate rotations, taking care not to eliminate the dual of any rotation previously eliminated. Finally, eliminate half of the remaining self-dual rotations, leaving a symmetric stable assignment.

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 problem, a 2-approximation algorithm can be obtained via a reduction to a weighted 2SAT problem [12], so the same technique also gives us a 2-approximation for the optimal bipartite stable allocation problem.

Chapter 6

STABLE ALLOCATIONS AND MAXIMUM FLOWS

Stable matching problems can surface in unexpected places. One particularly surprising yet elegant application is given by Conforti et al. [5], who show that in any graph, if we are given a set of k_1 edge-disjoint s-r paths and a set of k_2 edge-disjoint r-t paths, we can "splice" these paths together by solving a bipartite stable matching problem to obtain $\min(k_1, k_2)$ edge-disjoint s-t paths.

Figure 6.1 illustrates the transformation of the flow problem into a corresponding stable marriage problem. Note that the s-r paths $(p_1 \ldots p_3)$ are edge disjoint from each other but may have common edges with the r-t paths $(q_1 \ldots q_3)$. The stable marriage instance constructed has the s-r paths on the left hand side with the preference lists constructed in the order in which the r-t paths are encountered when walking from s to r. Similarly it has the r-t paths on the right hand side. Here the preference lists are constructed in the order in which the s-r paths are encountered when walking from t to r. By solving the corresponding stable marriage problem we get 3 edge disjoint s-t paths. A resulting s-t path starts on the s-r path and transitions over to the r-t path when it encounters the common edge to which the s-r path is assigned to in the solution (As shown in the figure

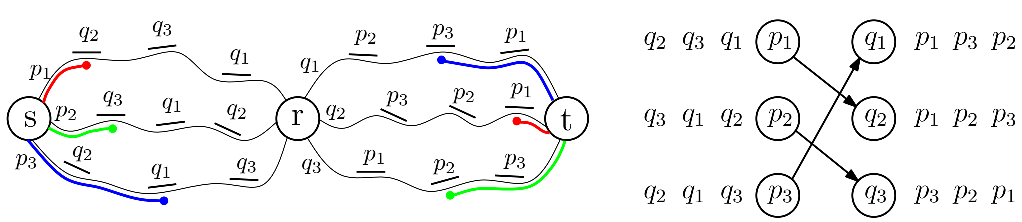


Figure 6.1: Transforming a max flow problem into a stable matching problem. Left hand side preference lists are constructed in the order in which common edges are encountered from s to r, while the right hand side preference lists are constructed in the order from t to r.

with colored paths).

This elegant transformation leads to a data structure that implicitly encodes a maximal set of edge-disjoint paths between every pair of vertices in a graph using $O(mn)$ space, such that the maximal set of edge-disjoint paths between two given vertices can be reconstructed in $O(m\alpha(n,n))$ time.

It is straightforward to generalize this result as follows: given a decomposition of an s-r flow of value f_1 into s-r path flows as well as a decomposition of an r-t flow of value f_2 into r-t path flows, we can splice these paths together to obtain an s-t flow of value min(f_1, f_2) by solving a bipartite stable allocation problem. Unfortunately, the analogous data structure we would obtain for storing all pairs of s-t maximum flows in a graph turns out ineffective. This is because the decomposition of flow may result in m path flows. This gives us a bipartite stable allocation problem with m elements on both sides, where each preference list can have length up to n (since a preference list is at most the length of a path). The stable allocation instance constructed, would solve in $O(mn \log n)$ worst case time.

Using the analogous data structure to calculate the flow between any given two nodes would take $O(\alpha(n,n)mn \log n)$ worst case time, which is larger than the time required to compute an s-t maximum flow from scratch.

APPENDIX

HARD INSTANCES FOR THE GS AND BB ALGORITHMS

Gale-Shapley: The worst-case running time of the Gale-Shapley algorithm can be exponential. This fact can be illustrated by a simple instance. Figure A.1 shows a bipartite arrangement with two jobs a and b (left hand side), both having processing time C; and three machines a', b' and c' (right hand side) having capacities $C-1$, C and 1 respectively. Here C can be any arbitrarily large value. Job a proposes C units to machine a', which accepts $C-1$ units and rejects one unit. Job a then proposes one unit to machine b', which accepts it (Figure A.1(a)). Now job b proposes C units to machine b', which accepts $C-1$ units and rejects 1 unit (Since machine b' already has 1 unit of assignment from the more preferred job - a). Job b proposes the rejected 1 unit to machine a' (Figure A.1(b)). This starts a series of proposals and rejections where 1 unit is pushed around in a loop between jobs a and b, and machines a' and b'. In every iteration of the loop the assignments $x(a, a')$ and $x(b, b')$ get decreased by 1 and assignments $x(a, b')$ and $x(b, a')$ get increased by 1. When the loop ends we have C units of a assigned to b' and $C-1$ units of b assigned to a'. The remaining 1 unit of b is proposed to c' and accepted, hence terminating the algorithm (Figure A.1(c)). The running time for this instance is $O(nC)$. Note that for this instance the algorithm will always be caught in the same loop, irrespective of the proposal order.

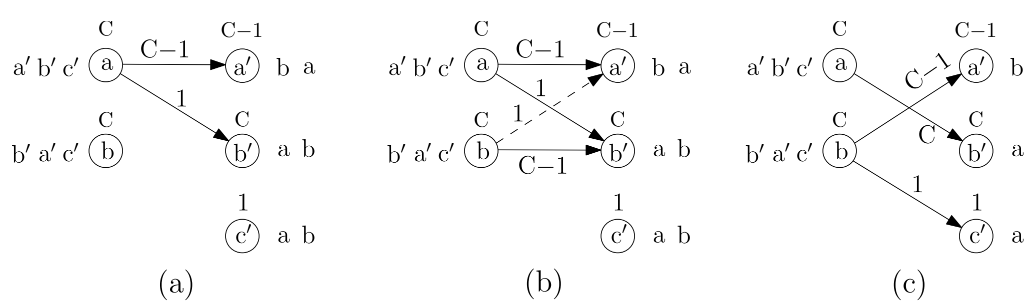


Figure A.1: A bad instance where the Gale-Shapley algorithm runs in exponential time irrespective of the proposal order.

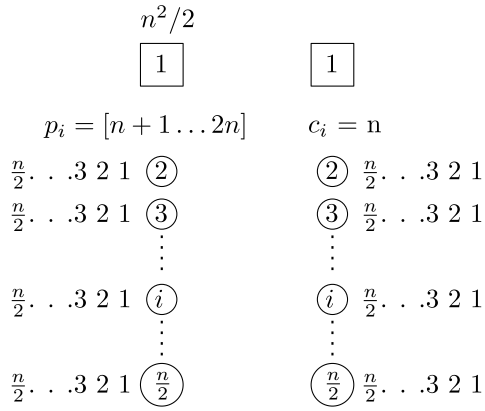


Figure A.2: A bad instance where the Baiou-Balinski algorithm runs in $\Omega(n^3)$ runtime with high probability.

Baiou-Balinski: 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 $\{n+1,\ldots,2n\}$ 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 $p([n/2]) = c([n/2])$. Each job ranks the machines in order $n/2, n/2-1, \ldots, 1$, and each machine ranks the jobs in order $n/2, n/2-1, \ldots, 1$. 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 $1 \ldots i_0 - 1$ 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 $j_0 \ldots n/2$. 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 $E[r(\pi)] \le 5$.

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 $E[X] \le n^2/4$. Since $\Pr[X \le n^2/4] > 0$, the probabilistic method tells us that there must be *some* instance for which $X \le n^2/4$. 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, $r(\pi) \le x(i, j'_i)$, and moreover $r(\pi) \le Z$ 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,

$$\begin{aligned} E[r(\pi)] \le E[Z] &= \sum_{k=1}^{\infty} \Pr[Z \ge k] \\ &= \sum_{k=0}^{n-1} \left(1 - \frac{k}{n}\right)^{n/4} \\ &\le \sum_{k=0}^{n-1} e^{-k/4} \\ &\le 1 + \int_0^{\infty} e^{-x/4} dx = 5. \end{aligned}$$

□

BIBLIOGRAPHY

[1] M. Baiou and M. Balinski, *Many-to-many matching: Stable polyandrous polygamy (or polygamous polyandry)*, Discrete Applied Mathematics **101** (2000), 1–12.

[2] ______, *Erratum: The stable allocation (or ordinal transportation) problem*, Mathematics of Operations Research **27** (2002), no. 4, 662–680.

[3] V. Bansal, A. Agrawal, and V. S. Malhotra, *Polynomial time algorithm for an optimal stable assignment with multiple partners.*, Theoretical Computer Science **379** (2007), no. 3, 317–328.

[4] C. Chekuri and S. Khanna, *A PTAS for the multiple knapsack problem.*, Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2000, pp. 213–222.

[5] M. Conforti, R. Hassin, and R. Ravi, *Reconstructing flow paths*, Operations Research Letters **31** (2003), no. 4, 273–276.

[6] B.C. Dean, N. Immorlica, and M.X. Goemans, *Finite termination of "augmenting path" algorithms in the presence of irrational problem data*, Proceedings of the 14th annual European Symposium on Algorithms (ESA), 2006, pp. 268–279.

[7] E.A. Dinic, *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Mathematics Doklady **11** (1970), 1277–1280.

[8] Y. Dinitz, N. Garg, and M.X. Goemans, *On the single-source unsplittable flow problem*, Combinatorica **19** (1999), 17–41.

[9] L.R. Ford and D.R. Fulkerson, *Maximal flow through a network*, Can. Journal of Math. **8** (1956), 339–404.

[10] ______, *Flows in networks*, Princeton University Press, 1962.

[11] D. Gale and L.S. Shapley, *College admissions and the stability of marriage*, American Mathematical Monthly **69** (1962), no. 1, 9–14.

[12] D. Gusfield and R. Irving, *The stable marriage problem: Structure and algorithms*, MIT Press, 1989.

[13] N. Immorlica and M. Mahdian, *Marriage, honesty, and stability*, Proceedings of 16th ACM Symposium on Discrete Algorithms, 2005, pp. 53–62.

[14] R.W. Irving, P. Leather, and D. Gusfield, *An efficient algorithm for the "optimal" stable marriage*, Journal of the ACM **34** (1987), no. 3, 532–543.

[15] B. Klaus and F. Klijn, *Stable matchings and preferences of couples*, Journal of Economic Theory **121** (2005), 75–106.

[16] J.M. Kleinberg, *Approximation algorithms for disjoint paths problems*, Ph.D. thesis, M.I.T., 1996.

[17] D.E. Knuth, *Stable marriage and its relation to other combinatorial problems*, 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.

[18] C.H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: Algorithms and complexity*, Prentice-Hall, 1982.

[19] E. Ronn, *NP-complete stable matching problems*, Journal of Algorithms **11** (1990), 285–304.

[20] 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** (1984), 991–1016.

[21] ______, *On the allocation of residents to rural hospitals: a general property of two-sided matching markets*, Econometrica **54** (1986), 425–427.

[22] ______, *The national residency matching program as a labor market*, Journal of the American Medical Association **275** (1996), no. 13, 1054–1056.

[23] A.E. Roth and E. Peranson, *The redesign of the matching market for american physicians: Some engineering aspects of economic design*, American Economic Review **89** (1999), 748–780.

[24] A.E. Roth and M. Sotomayor, *Two-sided matching: A study in game-theoretic modeling and analysis*, Cambridge University Press, 1990.

[25] D.B. Shmoys and É. Tardos, *Scheduling unrelated machines with costs*, Proceedings of the 4th annual ACM-SIAM Symposium on Discrete algorithms (SODA), 1993, pp. 448–454.

[26] M. Skutella, *Approximating the single source unsplittable min-cost flow problem*, Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS), 2000, pp. 136–145.

[27] D.D. Sleator and R.E. Tarjan, *A data structure for dynamic trees*, Journal of Computer and System Sciences **26** (1983), no. 3, 362–391.

[28] ______, *Self-adjusting binary search trees*, Journal of the ACM **32** (1985), no. 3, 652–686.

[29] U. Zwick, *The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate*, Theoretical Computer Science **148** (1995), 165–170.