1 The Stable Matching Problem

In the previous two notes, we discussed several proof techniques. In this note, we apply some of these techniques to analyze the solution to an important problem known as the *Stable Matching Problem*, which we now introduce. The Stable Matching Problem is one of the highlights of the field of algorithms.

Suppose you run an employment system[1], and your task is to match up n jobs and n candidates. Each job has an ordered *preference list* of the n candidates, and each candidate has a similar list of the n jobs. For example, consider the case of $n = 3$, i.e., three jobs at Approximation Inc., Basis Co., and Control Corp., and three candidates[2] Anita, Bridget, and Christine, with the following preference lists (lists are ordered from most favorable to least favorable):

Jobs	Candidates		
Approximation Inc.	Anita	Bridget	Christine
Basis Co.	Bridget	Anita	Christine
Control Corp.	Anita	Bridget	Christine

Candidates	Jobs		
Anita	Basis Co.	Approximation Inc.	Control Corp.
Bridget	Approximation Inc.	Basis Co.	Control Corp.
Christine	Approximation Inc.	Basis Co.	Control Corp.

What you would like to do as head of the employment system is to match each job with a candidate. For example, two possible matchings are {(Approximation Inc., Anita), (Basis Co., Bridget), (Control Corp., Christine)} and {(Approximation Inc., Bridget), (Basis Co., Christine), (Control Corp., Anita)}. However, you don't want just any old matching! In order for your employment system to be successful, you wish the matching to make "everyone happy", in that nobody can realistically hope to benefit by switching jobs. Can you do this efficiently?

It turns out that there is indeed an algorithm to achieve this; moreover, it's remarkably simple, fast, and widely-used. It's called the *Propose-and-Reject algorithm* (a.k.a. the Gale-Shapley algorithm), and we present it now.

2 The Propose-and-Reject Algorithm

We think of the algorithm as proceeding in "days" to have a clear unambiguous sense of discrete time.

[1] If you'd like, think of a hypothetical system that could be run by the EECS department for student internships.

[2] For clarity of exposition in written English, we will use female pronouns for candidates and neutral pronouns for jobs. If we were to use neutral pronouns for both, there might be confusion between inanimate jobs and living candidates.

Every Morning: Each job proposes (i.e. makes an offer) to the most preferred candidate on its list who has not yet rejected this job.

Every Afternoon: Each candidate collects all the offers she received in the morning; to the job offer she likes best among these, she responds "maybe" (she now has it *in hand* or *on a string*), and to the other offers she says "no" (i.e., she rejects them). (This is just a way for us to virtually model that there are no "exploding offers" and a job can't withdraw an offer once an offer is made.)

Every Evening: Each rejected job crosses off the candidate who rejected its offer from its list.

The above loop is repeated each successive day until there are no offers rejected. At that point, each candidate has a job offer in hand (i.e. on a string); and on this day, each candidate accepts their offered job (i.e. the job offer she has in hand) and the algorithm terminates.

Let's understand the algorithm by running it on our example above. The following table shows which jobs make offers to which candidates on the given day (the jobs in bold are "on a string"):

Days	Candidates	Offers
1	Anita	**Approximation Inc.**, Control Corp.
	Bridget	**Basis Co.**
	Christine	—
2	Anita	**Approximation Inc.**
	Bridget	**Basis Co.**, Control Corp.
	Christine	—
3	Anita	**Approximation Inc.**
	Bridget	**Basis Co.**
	Christine	**Control Corp.**

Thus, the algorithm outputs the matching: {(Approximation Inc., Anita), (Basis Co., Bridget), (Control Corp., Christine)}.

Before analyzing the algorithm's properties further, let's take a moment to ask one of the first questions you should always ask when confronted with a new concept: Why study stable matchings in the first place? What makes it and its solution so interesting? For this, we discuss the very real impact of the Gale-Shapley algorithm on the *Residency Match* program.

3 The Residency Match

Perhaps the most well-known application of the Stable Matching Algorithm is the residency match program, which pairs medical school graduates and residency slots (internships) at teaching hospitals. Graduates and hospitals submit their ordered preference lists, and the stable matching produced by a computer matches students with residency programs.

The road to the residency match program was long and twisted. Medical residency programs were first introduced about a century ago. Since interns offered a source of cheap labor for hospitals, soon the number

of residency slots exceeded the number of medical graduates, resulting in fierce competition. Hospitals tried to outdo each other by making their residency offers earlier and earlier. By the mid-40s, offers for residency were being made by the beginning of junior year of medical school, and some hospitals were contemplating even earlier offers — to sophomores[3]! The American Medical Association finally stepped in and prohibited medical schools from releasing student transcripts and reference letters until their senior year. This sparked a new problem, with hospitals now making "short fuse" offers to make sure that if their offer was rejected they could still find alternate interns to fill the slot. Once again the competition between hospitals led to an unacceptable situation, with students being given only a few hours to decide whether they would accept an offer.

Finally, in the early 1950's, this unsustainable situation led to the centralized system called the National Residency Matching Program (N.R.M.P.) in which the hospitals ranked the residents and the residents ranked the hospitals. The N.R.M.P. then produced a pairing between the applicants and the hospitals, though at first this pairing was not stable. It was not until 1952 that the N.R.M.P. switched to the Propose-and-Reject Algorithm, resulting in a stable matching.

Finally, if the above still hasn't convinced you of the worth of this algorithm, consider this: In 2012, Lloyd Shapley and Alvin Roth won the Nobel Prize in Economic Sciences by extending the Propose-and-Reject Algorithm. (The moral of the story? Careful modeling with appropriate abstractions pays off.)

4 Properties of the Propose-and-Reject Algorithm

There are two properties we wish to show about the propose-and-reject algorithm: First, that it doesn't run forever, i.e., it halts; and second, that it outputs a "good" (i.e., stable) matching. The former is easy to show; let us do it now.

Lemma 4.1. *The propose-and-reject algorithm always halts.*

Proof. The argument is simple: On each day that the algorithm does not halt, at least one job must eliminate some candidate from its list (otherwise the halting condition for the algorithm would be invoked). Since each list has n elements, and there are n lists, this means that the algorithm must terminate in at most n^2 iterations (days). □

Next, we'd like to show that the algorithm finds a "good" matching. Before we do so, let's clarify what we mean by "good".

4.1 Stability

What properties should a good matching have? Perhaps we'd like to maximize the number of first ranked choices? Alternatively, we could minimize the number of last ranked choices. Or maybe it would be ideal to minimize the sum of the ranks of the choices, which may be thought of as maximizing the average happiness.

[3] Any resemblance to the frustration experienced by Berkeley students seeking internships in the summer and having to interview in the fall is not coincidental.

In this lecture we will focus on a much more basic criterion that is rooted in the idea of autonomy, namely *stability*. A matching is **unstable**[4] if there is a job and a candidate who both prefer working with each other over their current matchings. We will call[5] such a pair a *rogue couple*. So a matching of n jobs and n candidates is **stable** if it has no rogue couples. Let us now recall our example from earlier:

Jobs	Candidates		
Approximation Inc.	Anita	Bridget	Christine
Basis Co.	Bridget	Anita	Christine
Control Corp.	Anita	Bridget	Christine

Candidates	Jobs		
Anita	Basis Co.	Approximation Inc.	Control Corp.
Bridget	Approximation Inc.	Basis Co.	Control Corp.
Christine	Approximation Inc.	Basis Co.	Control Corp.

Concept check! Consider the following matching for the example above: {(Approximation Inc., Christine), (Basis Co., Bridget), (Control Corp., Anita)}. Why is this matching unstable? (Hint: Approximation Inc. and Bridget are a rogue couple — why?)

Here is a stable matching for our example: {(Basis Co., Anita), (Approximation Inc., Bridget), (Control Corp., Christine)}. Why is (Approximation Inc., Anita) *not* a rogue couple here? It's certainly true that Approximation Inc. would rather work with Anita than its current employee Bridget. Unfortunately for it, however, Anita would rather be with her current employer (Basis Co.) than with Approximation Inc.. Note also that both Control Corp. and Christine are paired with their *least* favorite choice in this matching. Nevertheless, this does not violate stability since none of their more preferred choices would rather work with them than who they are matched with.

Before we discuss how to find a stable matching, let us ask a more basic question: Do stable matchings always exist? Surely the answer is yes, since we could start with any matching and seemingly make it more and more stable as follows: If there is a rogue couple, modify the current matching so that this couple is matched up. Repeat. Surely this procedure must result in a stable matching? Unfortunately this reasoning is not sound! Why? Although pairing up the rogue couple reduces the number of rogue couples by one, it may also *create new* rogue couples. So it is not at all clear that this procedure will terminate!

Let's further illustrate the fallacy of the reasoning above by applying it to a closely related scenario known as the *Roommates Problem*. In this problem, we have $2n$ people who must be paired up to be roommates (the

[4] Why is this called "unstable?" Because in such a situation, the rogue candidate could just renege on their official offer and the rogue job/employer could just fire the person that they officially hired to hire their preferred rogue candidate instead. Then one job is suddenly empty and one innocent person just got fired. This is not what we want to see happening. We want everyone to be happy enough that they all want to follow through on their final accepted offers.

[5] The choice of words for definitions is something that requires a balance of considerations. On the one hand, definitions in mathematical English have to be precise and unambiguous, and so to a computer, it doesn't matter what specific English words we use for the definition. But our definitions are going to be used by humans, and we would like to allow people to leverage their intuition, etc. At the same time, we want the definitions to not be too cumbersome to use. Here, there is often a balancing act that we must follow between over-triggering the incredibly nuanced set of human intuitions and having concise wording. In this case, human intuition would suggest *potentially rogue couple* instead of *rogue couple* to be more accurate, since they aren't actually matched up to each other. However, since "potentiality" isn't something we want to model more fully, we just go with the more concise wording.

difference being that unlike in our view of stable matching with asymmetric partners of different types, a person can be paired with any of the other $2n-1$ people, i.e., there is no asymmetry in types). Now, suppose our approach of iteratively matching up rogue couples indeed *did* work. Since this approach does not exploit the asymmetry, we can apply it to the roommates problem. Thus, by the (flawed) reasoning above, we could conclude that there must always exist a stable matching for roommates. Wouldn't you be surprised then to read the following counterexample, which gives an instance of the roommates problem which does *not* have a stable matching!

Roommates			
A	B	C	D
B	C	A	D
C	A	B	D
D	—	—	—

We claim that in this instance, there always exists a rogue couple for any matching. For example, the matching {(A,B), (C,D)} contains the rogue couple B and C. How about {(B,C), (A,D)}? This matching is unstable because now A and C are a rogue couple.

Concept check! Verify that in this example there is *no* stable matching!

We conclude that any proof that there always exists a stable matching in the stable matching problem *must* use the fact that there are two different types[6] in an essential way. In the next section we give such a proof, and a comforting one at that: We prove that there must exist a stable matching because the propose-and-reject algorithm always outputs one!

4.2 Analysis

We now prove that the propose-and-reject algorithm always outputs a stable matching. Why should this be the case? Consider the following intuitive observation:

Observation 4.1. *Each job begins the algorithm with its first choice being a possibility; as the algorithm proceeds, however, its best available option can only get* worse *over time. In contrast, each candidate's offers can only get* better *with time.*

Thus, as the algorithm progresses, each job's options get worse, while each candidate's improves; at some point, the jobs and the candidates must "meet" in the middle, and intuitively such a matching should be stable.

Let us formalize this intuition beginning with a formal statement of Observation 4.1 via the following lemma.

Lemma 4.2 (Improvement Lemma)**.** *If job J makes an offer to candidate C on the kth day, then on every subsequent day C has a job offer in hand (on a string) which she likes at least as much as J.*

[6]For historical reasons that relate to how cable connectors and fasteners are described in mechanical systems, as well as how linguistics denotes grammatical types on nouns, this category of types is sometimes referred to using the technical term *gender*. You might see references to gender in the literature if you look up stable matchings. However, what matters here is that we need to match up two different kinds of things. It doesn't have to be jobs and candidates. It could packets and network ports, computational jobs and servers, threads and cores, students and slots in classes, etc. As long as the two things are different in nature.

Proof. We proceed by induction on the day i, with $i \geq k$.

Base case $(i = k)$: On day k, C receives at least one offer (from J). At the end of day k, she will therefore have an offer in hand either from J or a job she likes more than J, since she chooses the best among her offers.

Inductive Hypothesis: Suppose the claim is true for some arbitrary $i \geq k$.

Inductive Step: We prove the claim for day $i+1$. By the Induction Hypothesis, on day i, C had an offer from job J' on a string which she likes at least as much as J. (Note that J' may be J.) According to the algorithm, J' proposes again to C again on day $i+1$ (since this job offer hasn't yet been rejected by her and is not allowed to explode). Therefore, at the end of day $i+1$, C will have on a string either J' or an offer from a job she likes more than J'; in both cases, she likes this job at least as much as J. Hence the claim holds for day $i+1$, completing the inductive step. □

Detour: The Well-Ordering Principle. Let's take a moment to consider an alternate proof of Lemma 4.2; in the process, we'll uncover a fundamental principle which is equivalent to induction.

Alternate proof of Lemma 4.2. As in our original proof, the claim certainly holds on day $i = k$. Suppose now, for the sake of contradiction, that the ith day for $i > k$ is the first counterexample where C has either no offer or an offer from some J^* inferior to J on a string. On day $i-1$, she had an offer from some job J' on a string and liked J' at least as much as J. According to the algorithm, J' still has an offer out to C on the ith day. (i.e. offers don't explode and can't be withdrawn) So C has the choice of at least one job (J') on the ith day; consequently, her best choice must be at least as good as J', and therefore certainly better than J^* or nothing. This contradicts our initial assumption. □

What proof technique did we use to prove this lemma? Is it contradiction? Or some other beast entirely? It turns out that this is *also* a proof by induction! Why? Well, we began by establishing the base case of $i = k$. Next, instead of proving that for all i, $P(i) \implies P(i+1)$, we showed that the *negation* of this statement is false, i.e., that "there exists i such that $\neg(P(i) \implies P(i+1))$" does not hold; thus, by the law of the excluded middle, it must indeed hold that for all i, $P(i) \implies P(i+1)$.

Concept check! Ensure you understand why these two proofs are equivalent.

Let us now be a bit more careful — what exactly allowed us to take this alternate approach? The answer lies in a very special property of the natural numbers known as the *Well-Ordering Principle*. This principle says that any non-empty set of natural numbers contains a "smallest" element. Formally:

Definition 4.1 (Well-ordering principle)**.** *If $S \subseteq \mathbb{N}$ and $S \neq \emptyset$, then S has a smallest element.*

Concept check! Consider the following subsets of the natural numbers: $S_1 = \{5, 2, 11, 7, 8\}$, $S_2 = \{n \in \mathbb{N} : n \text{ is odd}\}$, $S_3 = \{n \in \mathbb{N} : n \text{ is prime}\}$. Does each of these sets have a smallest element?

Where did we use the well-ordering principle in our proof? In the line "Suppose… that the ith day for $i > k$ is the first counterexample…" — specifically, if the natural numbers did not obey this principle, then the

notion of a *first* counterexample would not be valid. Note also that induction relies on this principle: The statement "for all i, $P(i) \implies P(i+1)$" only makes sense if there is a well-defined order imposed on the natural numbers (i.e., that 3 comes before 4, and 4 comes before 5, etc...). That the natural numbers obey this principle may be a fact you have long taken for granted; but there do exist sets of numbers which do *not* satisfy this property! In this sense, the natural numbers are indeed quite special.

Concept check! Do the integers satisfy the well-ordering principle? How about the reals? How about the non-negative reals?

Back to our analysis of the Propose-and-Reject Algorithm. Returning to our analysis, we now prove that when the algorithm terminates, all n candidates have job offers. Before reading the proof, see if you can convince yourself that this is true. The proof is remarkably short and elegant, and critically uses the Improvement Lemma.

Lemma 4.3. *The propose-and-reject algorithm always terminates with a matching.*

Proof. We proceed by contradiction. Suppose that there is a job J that is left unpaired when the algorithm terminates. It must have made an offer to all n of the candidates on its list and been rejected by all of them. By the Improvement Lemma, since its offer was rejected, each of these n candidates has had a better offer in hand since J made an offer to her. Thus, when the algorithm terminates, n candidates have n jobs on a string not including J. So there must be at least $n+1$ jobs. But this is a contradiction, since there are only n jobs by assumption. □

To complete our analysis of the propose-and-reject algorithm, we need to verify the key property that the matching produced by the algorithm is stable.

Theorem 4.1. *The matching produced by the algorithm is always stable.*

Proof. We give a direct proof that, in the matching output by the algorithm, no job can be involved in a rogue couple. Consider any couple (J,C) in the final matching and suppose that J prefers some candidate C^* to C. We will argue that C^* prefers her job to J, so that (J,C^*) cannot be a rogue couple. Since C^* occurs before C in J's list, J must have made an offer to C^* before it made an offer to C. Therefore, by the Improvement Lemma, C^* likes her final job at least as much as J, and therefore prefers it to J. Thus no job J can be involved in a rogue couple, and the matching is stable. □

Notice that we did this proof of stability from the perspective of the jobs and not from the perspective of the candidates.

5 Optimality

In Section 4.2, we showed that the propose-and-reject algorithm always outputs a stable matching. But is this so impressive? Will your employment system really be successful outputting matches which are just

good enough? To offer the best service (and to displace the current approach), you would ideally strive for some notion of *optimality* in the solutions you obtain.

Consider, for example, the case of 4 jobs and 4 candidates with the following preference lists (for simplicity, we use numbers and letters below to label the jobs and candidates):

Jobs	Candidates			
1	A	B	C	D
2	A	D	C	B
3	A	C	B	D
4	A	B	C	D

Candidates	Jobs			
A	1	3	2	4
B	4	3	2	1
C	2	3	1	4
D	3	4	2	1

For these lists, there are exactly two stable matchings: $S = \{(1,A), (2,D), (3,C), (4,B)\}$ and $T = \{(1,A), (2,C), (3,D), (4,B)\}$. The fact that there are *two* stable matchings raises the questions: What is the best possible job for each candidate? What is the best possible candidate for each job? For example, let us consider job 2. The trivial guess for defining the best partner for 2 is its first choice, candidate A; unfortunately, A is just not a realistic possibility for taking this job, as matching this job with A would not be stable, since this job is so low on her preference list and other jobs also prefer A. Indeed, there is no stable matching in which 2 is paired with A. Examining the two stable matchings, we find that the best possible *realistic* outcome for job 2 is to be paired with candidate D. This demonstrates that the notion of "best partner" can be a bit fuzzy if we are not careful.

So, let us be careful; inspired by the discussion above, let's make a more precise and tenable definition of optimality.

Definition 4.2 (Optimal candidate for a job)**.** *For a given job J, the* optimal *candidate for J is the highest rank candidate on J's preference list that J could be paired with in any stable matching.*

In other words, the optimal candidate is the best that a job can do in a matching, *given that the matching is stable*.

Concept check! Who are the optimal candidates for each job *A*, *B*, *C*, and *D* in our example above? (Hint: The optimal candidate for job 2, as discussed above, is *D*.)

By definition, the best that each job can hope for is to be paired with its optimal candidate. But can all jobs achieve this optimality *simultaneously*? In other words, is there a stable matching such that each and every job is paired with its optimal candidate? If such a matching exists, we will call it a *job/employer optimal* matching. Returning to the example above, *S* is a job/employer optimal matching since *A* is 1's optimal candidate, *D* is 2's optimal candidate, *C* is 3's optimal candidate, and *B* is 4's optimal candidate.

Similarly, we can define an optimal job for a candidate.

Definition 4.3 (Optimal job for a candidate)**.** *For a given candidate C, the* optimal *job for C is the highest-ranked job on C's preference list that C could be paired with in any stable matching.*

In other words, the optimal job is the best job that a candidate could get in a matching, *given that the matching is stable*.

We can define a *candidate optimal* matching, which is the matching in which each candidate is paired with her optimal job.

Concept check! Check that T is a candidate optimal matching.

We can also go in the opposite direction and define the *pessimal* candidate for a job to be the lowest ranked candidate that it is ever paired with in some stable matching. This leads naturally to the notion of a *employer pessimal* matching — can you define it, and also a candidate pessimal matching?

Now, we get to the heart of the matter: Who is better off in the Propose-and-Reject Algorithm — jobs/employers or candidates?

Theorem 4.2. *The matching output by the Propose-and-Reject algorithm is job/employer optimal.*

Proof. Suppose for sake of contradiction that the matching is *not* employer optimal. Then, there exists a day on which some job had its offer rejected by its optimal candidate; let day k be the first such day. On this day, suppose J was rejected by C^* (its optimal candidate) in favor of an offer from J^*. By the definition of optimal candidate, there must exist a stable matching T in which J and C^* are paired together. Suppose T looks like this: $\{\ldots,(J,C^*),\ldots,(J^*,C'),\ldots\}$. We will argue that (J^*,C^*) is a rogue couple in T, thus contradicting stability.

First, it is clear that C^* prefers J^* to J, since she rejected an offer from J in favor of an offer from J^* during the execution of the propose-and-reject algorithm. Moreover, since day k was the first day when some job had an offer rejected by its optimal candidate, before day k, job J^* had not had its offer rejected by its optimal candidate. Since J^* made an offer to C^* on day k, this implies that J^* likes C^* at least as much as its optimal candidate, and therefore at least as much as C' (its partner in the stable matching T). Therefore, (J^*,C^*) form a rogue couple in T, and so T is not stable. Thus, we have a contradiction, implying the matching output by the propose-and-reject algorithm must be employer/job[7] optimal. □

What proof techniques did we use to prove this Theorem 4.2? Again, it is a proof by induction, structured as an application of the well-ordering principle.

Concept check! Where did we use the well-ordering principle in the proof of Theorem 4.2?

Exercise. Can you rewrite the proof of Theorem 4.2 as a regular induction proof? (Hint: The proof is really showing by induction on k the following statement: For every k, no job gets rejected by its optimal candidate on the kth day.)

We conclude that employers would fare very well by following this algorithm. How about the candidates? The following theorem confirms a sad truth:

[7] What we have actually proved is that the propose-and-reject algorithm returns a matching that is optimal for the "proposers." In our case, the employers/jobs are the ones that make offers, so it is employer/job optimal. If we ran propose-and-reject with candidates proposing, then it would be candidate-optimal.

Theorem 4.3. *If a matching is employer/job optimal, then it is also candidate pessimal.*

Proof. We proceed by contradiction. Let $T = \{\ldots,(J,C),\ldots\}$ be the employer optimal matching (which we know from Theorem 4.2 is output by the algorithm). Suppose for the sake of contradiction that there exists a stable matching $S = \{\ldots,(J^*,C),\ldots,(J,C'),\ldots\}$ such that job J^* is lower-ranked on C's preference list than job J (i.e., J is not her pessimal job). We will argue that S cannot be stable by showing that (J,C) is a rogue couple in S. By assumption, C prefers job J to J^* since J^* is lower on her list. And J prefers candidate C to its partner C' in S because C is its partner in the employer optimal matching T. Contradiction. Therefore, the employer optimal matching is candidate pessimal. □

In light of these findings, what does the propose-and-reject algorithm teach us? Institutional structure matters greatly in the distributional quality of outcomes, and institutional structure reflects real-world power and its history.

Exercise. What simple modification would you make to the propose-and-reject algorithm so that it always outputs a *candidate optimal* matching?

Let us close with some final comments about the National Residency Matching Program. Until recently the propose-and-reject algorithm was run[8] with the hospitals doing the proposing, and so the matchings produced were hospital optimal. In the nineties, the roles were reversed[9] so that the medical students were proposing to the hospitals. More recently, there were other improvements made to the algorithm which the N.R.M.P. used. For example, the matching takes into account preferences for married students for positions at the same or nearby hospitals.

6 Further Reading (optional)

Though it was in use 10 years earlier, the propose-and-reject algorithm was first properly analyzed by Gale and Shapley, in a famous paper dating back to 1962 that still stands as one of the great achievements in the analysis of algorithms. The full reference is:

D. Gale and L.S. Shapley, "College Admissions and the Stability of Marriage," *American Mathematical Monthly* **69** (1962), pp. 9–14.

Stable matchings and related variants remains an active topic of research in EECS (and economics!). Although it is by now 30 years old, the following very readable book covers many of the interesting developments since Gale and Shapley's algorithm:

D. Gusfield and R.W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press, 1989.

[8] Everything happens inside a computer. None of the intermediate offers are actually made to humans, only the final stable offer.

[9] This role reversal was possible precisely because the algorithm was already running inside a computer. Switching who makes offers in the outside world would have been far harder to do credibly given the difference in real-world negotiating *power* between medical students and hospitals. This is an important lesson in the ordering of improvements as you try to work towards change in the real world. A non-hospital-optimal automated matching system would never have been adopted in the first place, and then an intern-optimal matching system would have always remained an idealistic fantasy.