New algorithms for matching problems

Jacky Lo and Mark C. Wilson

March 14, 2017

Abstract

The standard two-sided and one-sided matching problems, and the closely related school choice problem, have been widely studied from an axiomatic viewpoint. A small number of algorithms dominate the literature. For two-sided matching, the Gale-Shapley algorithm; for one-sided matching, (random) Serial Dictatorship and Probabilistic Serial rule; for school choice, Gale-Shapley and the Boston mechanisms.

The main reason for the dominance of these algorithms is their good (worst-case) axiomatic behaviour with respect to notions of efficiency and strategyproofness. However if we shift the focus to fairness, social welfare, tradeoffs between incompatible axioms, and average-case analysis, it is far from clear that these algorithms are optimal.

We investigate new algorithms several of which have not appeared (to our knowledge) in the literature before. We give a unified presentation in which algorithms for 2-sided matching yield 1-sided matching algorithms in a systematic way. In addition to axiomatic properties, we investigate agent welfare using both theoretical and computational approaches. We find that some of the new algorithms are worthy of consideration for certain applications. In particular, when considering welfare under truthful preferences, some of the new algorithms outperform the classic ones.

1 Introduction

The standard two-sided and one-sided matching problems, and the closely related school choice problem, have been widely studied from an axiomatic viewpoint. A small number of algorithms dominate the literature. For two-sided matching, the Gale-Shapley algorithm; for one-sided matching, (random) Serial Dictatorship and Probabilistic Serial rule; for school choice, Gale-Shapley and the Boston mechanisms.

The main reason for the dominance of these algorithms is their good axiomatic behaviour with respect to notions of efficiency and strategyproofness. However if we shift the focus to fairness, social welfare, or tradeoffs between incompatible axioms, it is far from clear that these algorithms are optimal.

1.1 Our contribution

In Section 3 we introduce several (in our opinion) natural algorithms for one-sided matching, several of which have not appeared (to our knowledge) in the literature before. We give a consistent derivation using specializations of the Gale-Shapley algorithm [] for two-sided matching, which includes the well-known algorithms Serial Dictatorship [] and Naive Boston [] in a unified framework. In Section 4, in addition to axiomatic properties such as efficiency and strategyproofness, we investigate welfare loss using a computational approach. We find that under truthful preferences, some of the new algorithms clearly outperform the classic ones. In particular, we recommend some new algorithms for some applications.

2 Definitions and terminology

Let $\mathscr{A} = \{a_1, \dots, a_n\}$ be a finite set of ***agents*** and $\mathscr{O} = \{o_1, \dots, o_m\}$ a finite set of ***items***.

In general, the number of items and agents may not be equal. We focus on the case $m = n$ in the present article. The more general case involves substantial complications: different ways of assigning preferences over subsets of $\mathscr{O}$ or $\mathscr{A}$ lead to different notions of strategyproofness, for example. However all our algorithms can be modified trivially in order to work in the general case.

In the standard two-sided matching problem, each element of $\mathscr{O}$ has a complete strict preference order for elements of $\mathscr{A}$, and vice versa, while for 1-sided matching only the latter information is required. For school choice the order of preference of items over agents depends on the preferences of agents over items — the two sides are definitely not independent (schools typically must admit students who are qualified as long as there is capacity, and use their own preferences only when a tie must be broken). We aim to unify these three cases, and restrict to the case of strict linear orders.

Let $L(\mathscr{O})$ (respectively $L(\mathscr{A})$) denote the set of all strict linear orders on $\mathscr{O}$ (resp. agents). A preference ***profile*** is a pair of functions (π_A, π_I) where $\pi_A : A \to L(\mathscr{O})$ and $\pi_I : \mathscr{O} \to L(\mathscr{A})$. A ***matching*** or ***discrete assignment*** is a function $f : \mathscr{O} \leftrightarrow \mathscr{A}$. Let S be the set of all doubly stochastic $n \times n$ matrices with rows indexed by agents and columns by items. A ***random assignment*** is an element of S. The ***matching problem*** is simply to output a matching given an input profile. The ***proportional*** assignment is the random assignment in which each matrix entry equals $1/n$.

Of course, there are $n!$ discrete assignments and finding one is trivial. The point is to find one with desirable properties. A discrete assignment is ***efficient*** if there is no other assignment which improves the outcome for some agent and does not worsen it for any agent. An algorithm for randomized assignment is ***ex-post efficient*** if every matching occurring with positive probability is an efficient assignment.

We now review some standard algorithms from the literature.

Two algorithms for the same matching problem are ***equivalent*** if they each produce the same output for every input. All the algorithms under study are ***anonymous***, meaning that a permutation of the players leads to the same permutation of the assignment. In other words, only the preferences matter, not the agents' identities. When discrete assignments are used, this means that whenever we have two agents with identical preferences, one must envy the other's assignment. A stronger condition is ***symmetry*** (also called "equal treatment of equals") which says that agents with the same preferences receive the same assignment. Clearly, this can only be satisfied in the framework of random assignments.

Every algorithm that uses a fixed initial order of agents and produces a matching can yield an algorithm that produces a random matching, simply by randomizing over the initial order. This is usually done according to the uniform distribution, in order to preserve symmetry between agents. Thus every algorithm discussed in Section 3 has a randomized version, which we denote by prefixing "R" to its name.

2.1 One-Sided Algorithms

One-sided matching refers to the situation where the items' preferences over agents are ignored.

The most commonly discussed algorithm is Serial Dictatorship [].

Example 2.1. *(Serial Dictatorship)*

*Fix an arbitrary linear ordering on A. The **Serial Dictatorship** algorithm (SD) with respect to this ordering assigns items to agents as follows: at step i, allocate to agent ith its most preferred item that has not already been allocated to a previous agent.*

The randomized version is denoted RSD, as mentioned above. Serial Dictatorship satisfies important axiomatic properties such as ***ex-post efficiency*** and ***strategyproofness*** (all axiomatic properties are defined and discussed in Section 4).

The next algorithm was proposed by Bogolmanaia and Moulin [BM01]. It is ***ordinally efficient*** in addition to being ex-post efficient, and has a weak strategyproofness property. It can be described using a cake-eating analogy.

Example 2.2. *(Probabilistic Serial) The **Probabilistic Serial** rule is inherently randomized, and generates a random assignment as follows. We interpret an assignment of fraction x of item j to agent i to mean that i receives fraction x of j (in other words we pretend that the items are infinitely divisible). All agents simultaneously begin "eating" at unit speed, each agent at each instant eating from its most preferred item among those that have not been completely consumed. On termination we have a random assignment.*

A closely related problem to one-sided matching is the ***housing market*** problem [SS74]. The difference is that every agent is assumed to have an initial allocated item, and we seek a method for finding an allocation that is optimal in some way.

Example 2.3. *(Top Trading Cycle) When each agent is considered to be initially assigned an entire item, the agents may trade amongst themselves as follows. Each agent i points to the agent currently owning the item on the top of i's preference list. By finiteness and since every node has outdegree 1, this directed graph must contain a cycle. Reallocate items according to the arcs in the cycle, and remove these agents and items from further consideration. Repeat (using pointers to the next level preference if necessary) until no items/agents remain. This **TTC algorithm** [SS74], attributed by Shapley and Scarf to David Gale, always yields a discrete assignment that is efficient, and the mechanism is strategyproof and individually rational. Furthermore the algorithm runs in polynomial time.*

Example 2.4. *(sample execution of TTC) Consider the profile where agents* 1 *and* 2 *have preferences* $a > b > c$ *and agent* 3 *has preference* $b > a > c$. *The assignment* $1 : c, 2 : b, 3 : a$ *is not ex-post efficient, because* 2 *and* 3 *can trade to their mutual benefit. In the first round of TTC, agents* 1 *and* 2 *points to agent* 3*, while agent* 3 *points to agent* 2*. There is a cycle between agents* 2 *and* 3*. The agents swap along the cycle and are removed from consideration. Agent* 1 *then points to itself in the next round, swaps along the cycle, remains with item* c*, and is removed from consideration. With no agents left, the TTC algorithm halts. The output is the ex-post efficient assignment* $1 : c, 2 : a, 3 : b$.

When using TTC we have freedom in the choice of initial assignment. For example, choosing this uniformly at random and running TTC yields an algorithm equivalent to RSD [AS98], and an adaptation of TTC to trade unit shares yields an algorithm equivalent to PS when run on the proportional endowment [Kes09].

We find it useful to run TTC on the output of some of our algorithms in Section 3 (algorithms which satisfy ex-post efficiency gain no benefit from running TTC, which terminates immediately because there is no cycle). The resulting combined algorithms, denoted XG where X is the name of the basic algorithm, are ex-post efficient and appear to have considerably better overall performance than the original algorithms.

2.2 Two-sided algorithms

In this case agents have complete strict preferences over items, and vice versa. The most well-known algorithm belongs to the class of ***deferred acceptance*** algorithms. Items and agents are tentatively matched, but these "engagements" may be broken. In fact each item may attach to up to n agents in the course of the algorithm.

In the ***Gale-Shapley*** algorithm [GS62], agents in turn approach previously unapproached items that they prefer to their currently assigned item (every agent prefers each item to not having an item, and every item prefers every agent to not being held by an agent). If the currently proposing agent is preferable to the agent currently matched with the item, the item will reject its current partner for the proposing agent. No agent may approach an item that has already rejected it (such an approach would lead to another rejection by the above rules). This ensures termination after at most n^2 proposals.

Example 2.5. *This is adapted from Example 2 in [GS62].*

Suppose that the proposers' preferences are as follows:

$$\begin{aligned} 1 &: \quad a > b > c > d \\ 2 &: \quad a > d > c > b \\ 3 &: \quad b > a > c > d \\ 4 &: \quad d > b > c > a \end{aligned}$$

and the proposees' preferences are given by

$$\begin{aligned} a &: \quad 4 > 3 > 1 > 2 \\ b &: \quad 2 > 4 > 1 > 3 \\ c &: \quad 4 > 1 > 2 > 3 \\ d &: \quad 3 > 2 > 1 > 4 \end{aligned}$$

The final matching is $1 : c, 2 : d, 3 : a, 4 : b$. *There are 9 proposals made during the execution of the algorithm.*

The Gale-Shapley algorithm has the well-known property that the output matching is ***stable***, meaning that there is no unmatched (agent, item) pair who each prefer each other to their current partner. Also, the output matching is optimal for proposers, meaning that each proposer receives the best possible item it can receive in a stable matching. It follows that the output of the algorithm does not depend on the order of proposals made by agents. Note that, by contrast, the output of the algorithms in Section 3 will depend strongly on the order of proposals.

We can produce one-sided matching algorithms from two-sided ones by forcing the items to have specific (fictitious) preferences. We use this idea systematically in Section 3.

2.3 School choice algorithms

A case intermediate between 1-sided and 2-sided matching, which is important for later, is that of ***school choice***. The Gale-Shapley algorithm is applicable to the case where the number of agents exceeds the number of items, provided items (schools) have capacity for some number of agents (students). A school accepts a student's proposal provisionally provided there is capacity remaining, or the student is preferable to an already tentatively accepted student. The case where each school has capacity 1 and the numbers of schools and students are equal is the case described in Section 2.2.

There are other algorithms for school choice that use ***immediate*** acceptance. In this case, each student first applies to her first choice school. Each school ranks applicants and chooses as many as it can, subject to capacity. Students not accepted already then apply to their second choice school, etc. This description implicitly uses simultaneous proposing by all unmatched agents and is called the ***Boston mechanism*** [AS03, MS14]. There is also a sequential version in which proposals are made one agent at a time. In that case, the order of proposals clearly changes the final allocation, since no engagement can ever be broken.

The special case of the Boston mechanism in which each school has capacity 1 and there are equal numbers of agents and items is a 2-sided matching algorithm as defined above. For the same input as in Example 2.5, the final allocation using sequential offers by $1,2,3,4$ in that order is $1:a, 2:d, 3:b, 4:c$. The final allocation using simultaneous offers is $1:a, 2:c, 3:b, 4:d$.

3 New algorithms

For the rest of the analysis, we construct one-sided matching algorithms by relaxing 2-sided algorithms. The Gale-Shapley algorithm generates a stable matching using a series of proposals, based on fixed preferences of the agents and items. Given the agent preferences over items as input to a 1-sided matching problem, we construct fictitious preferences for the items over the agents. For example, we can assume that all items have a fixed common preference.

Proposition 3.1. *SD is equivalent to a special case of GS.*

Proof. Suppose that all items have the same preference order over agents, which without loss of generality we write $1 > 2 > \cdots > n$. We claim that Gale-Shapley will output an assignment that is the same as the output of Serial Dictatorship with the agent order $1,2,\dots,n$.

The proof is inductive. The base case is that agent 1 will get his first choice with GS. It is trivially true as agent 1 will propose to its most preferred item, and since every item prefers agent 1 to any other agent, they cannot be rejected later. Therefore agent 1 will be allocated the same item under GS or SD.

Stability of GS implies that for every item that agent i wants more than the item they are allocated, the item must be held by an agent ranked higher by the item. If every agent before i gets its choice as per SD, agent i will eventually propose to its choice under SD. As every agent after i is ranked below i by all items, they cannot cause that item to reject i. Therefore agent i will have the same item under GS or SD. □

We can do the same thing with the (simultaneous) Naive Boston algorithm. Given an instance of 1-sided matching, we create fictitious preferences in which each item has the same preference, say $1 > 2 > \cdots > n$. The resulting algorithm we call the 1-sided Naive Boston algorithm. Note that we can also interpret this algorithm sequentially if we ensure that agent order is $1,2,\dots,n$, but otherwise the sequential and simultaneous forms will differ in general.

Below, we generalize this fictitious preference approach by allowing each item to build its fictitious preferences dynamically, using some fixed rule, based only on the sequence of proposals that it receives from agents. Recall that any order of proposals gives the same result for the Gale-Shapley algorithm with fixed preferences, but as we see below, this is not the case in our relaxed setup.

Definition 3.2. *Throughout the rest of this article, for the purposes of illustration and comparison between algorithms we use what we call the* ***standard profile*** *in which agents* $1,2$ *and* 3 *have preferences* $a > b > c > d$, *and agent* 4 *has preferences* $b > a > c > d$.

3.1 The "permanent memory" case

As the order of the agents' proposals affects the items' preferences, the order of proposals affects the final allocation. This has two consequences. The first is that the treatment of rejected agents matters. After an agent is rejected, either because the item prefers its current agent or breaks its tentative engagement, it may not necessarily be the next agent to propose. We consider two possibilities: a stack (rejected agents go to the top of the stack) or a queue (rejected agents go to the back of the queue). The other consequence is, as noted above, that the initial order of agents has an impact on the final allocation. For the purpose of the analysis, the algorithms will fix an arbitrary initial order.

We consider two rules for building preferences dynamically. These are ***early-proposal preference*** (or ***Accept-First***) and ***late-proposal preference*** (or ***Accept-Last***). Accept-First means that the first agent to approach an item is accepted, and subsequent proposals are rejected. Using Accept-Last, an item always breaks an engagement in favour of a new proposer, if it has not yet been held by the new proposer. In terms of the marriage interpretation often used to describe the Gale-Shapley algorithm, for Accept-First algorithms the proposees stick faithfully to their first suitor, whereas for Accept-Last algorithms the proposees are always more satisfied with a new suitor than their current fiancé.

These two dichotomies (stack/queue, Accept-First/Accept-Last) when combined with the distinction between permanent and temporary memory (explained in Section 3.2) yield 8 algorithms. We denote them by three-letter abbreviations. For example, PFS refers to permanent memory, Accept-First, stack.

Sample executions on the standard profile of all 8 algorithms introduced below can be found in Appendix A. The results are summarized in Table 1.

We consider the Accept-First algorithms. The algorithm PFS is equivalent to Serial Dictatorship. Interestingly, switching the stack to a queue results in an algorithm that is equivalent to the Naive Boston algorithm, so we obtain no new inequivalent algorithms in this case, just a unified presentation of old ones. We give the details below.

Proposition 3.3. *PFS with order of agents* $1, 2, \dots, n$ *is equivalent to Serial Dictatorship with the same order of agents.*

Proof. By definition PFS is a special case of GS in which there is a common preference order $1 > 2 > \cdots > n$ for items over agents, because engagements are never broken (by Accept-First and the permanent memory). By Proposition 3.1 the latter is equivalent to SD with the agent order $1, 2, \dots, n$. □

Proposition 3.4. *PFQ with order of agents* $1, 2, \dots, n$ *is equivalent to the* 1*-sided Naive Boston algorithm where the common preference order of items is* $1 > 2 > \cdots > n$.

Proof. It is useful to consider the above algorithm as occurring in rounds. Round i ends precisely when all the remaining agents have proposed to their ith choice. We show by induction that during round i:

- the agents proposing are precisely those who have not been matched previously;
- each such agent makes exactly one proposal, to its ith choice.

Since this is exactly the behaviour of the (simultaneous) 1-sided Boston algorithm and since assignments are never changed in either algorithm, the proposition follows.

When $i = 1$, all agents have proposed to their first choice. Thus each agent has made exactly one proposal, because any agent that is accepted never makes another proposal, and any agent that is rejected must go to the back of the queue and wait until all other agents have made their first proposal.

Assuming the result holds for all rounds before i, then all remaining agents have proposed to and been rejected by all items down to rank $i - 1$. In round i they must then propose to their ith choice. This happens exactly once because of the queue discipline. □

Example 3.5. *(Accept-Last permanent memory algorithms versus accept-first)*

For the standard profile, the final assignment using PFS is $1 : a, 2 : b, 3 : c, 4 : d$, *while using PLS it is* $1 : d, 2 : c, 3 : a, 4 : b$. *If we change the stack to a queue, the final assignment is* $1 : a, 2 : c, 3 : d, 4 : b$ *for PFQ and* $1 : d, 2 : c, 3 : b, 4 : a$ *for PLQ. Note that with these preferences, some agent must receive its 4th choice and no more than two agents can receive their 1st choice. Only PFQ achieves the latter condition.*

Although the Accept-Last algorithms are not ex-post efficient, empirical results show that (among other things) their output, when used as an initial endowment for TTC, leads to better welfare performance than the ex-post efficient Accept-First algorithms. We discuss this in details in Section 4.

Table 1: Behaviour of algorithms on standard profile where $n = 4$

Algorithm	Output matching	Number of proposals
PFS	1:a, 2:b, 3:c, 4:d	10
PFQ	1:a, 2:c, 3:d, 4:b	9
PLS	1:d, 2:c, 3:a, 4:b	9
PLQ	1:d, 2:c, 3:b, 4:a	10
TFS	1:d, 2:a, 3:c, 4:b	18
TFQ	1:a, 2:b, 3:d, 4:c	33
TLS	1:b, 2:a, 3:d, 4:c	18
TLQ	1:a, 2:b, 3:d, 4:c	21

3.2 The “temporary memory" case

The previous algorithms assume that each item retains its preferences throughout the execution of the algorithms. Other interesting algorithms can be generated by relaxing that requirement. Whenever the item resets its memory, it make sense for the agents to propose to items that rejected them before. Any rules that uses temporary memory for items must ensure that the algorithm will halt with a matching. To ensure the algorithm halts, we only allow the item to reset its memory when the number of tentative matchings has increased. As items do not go from matched to unmatched, this happens precisely when a new item is matched.

Whenever an agent proposes to an unmatched item, all items, including the new item, lose their memory of preferences. When an agent proposes to a matched item with no preferences, the item prefers the proposing agent instead of the matched agent. As the number of tentative matchings increases throughout the execution of the algorithm, there can only be n resets of the preferences, and thus the number of proposals is bounded by n^3.

Using the above rule, four new algorithms analogous to those in Section 3.1 can be constructed.

We first present the Accept-First algorithms. The temporary memory analogue of Serial Dictatorship, namely TFS, is interesting. Each round of this algorithm operates like a Serial Dictatorship in which a subset of the agents repeatedly “steal" items in chains until some agent chooses an unmatched item, whereupon a new round begins with a new agent beginning the stealing. The order of agent choices in a round is not fixed, but determined by the stealing process. We suggest the alternative name “Iterative Dictatorship” for this algorithm. The algorithm TFQ is the temporary memory analogue of the 1-sided Boston algorithm. Like all queue-based algorithms it is harder to interpret than a stack-based algorithm.

The Accept-Last algorithms are harder to understand (but see the party interpretation below, which was the inspiration for our entire research program).

Example 3.6. *(Temporary memory, Accept-First versus Accept-Last) For the standard profile, the final assignment under TFS is* $1:d, 2:a, 3:c, 4:b$ *and the final assignment using TFQ is* $1:a, 2:b, 3:d, 4:c$*. By contrast, the final assignment using TLS is* $1:b, 2:a, 3:d, 4:c$ *while the final assignment under TLQ is* $1:a, 2:b, 3:d, 4:c$*.*

3.3 Further comments

We have presented 8 algorithms in a unified framework, corresponding to the dichotomies memory/no memory, stack/queue, Accept First/Accept Last. Basic description of their behaviour on the standard profile is shown in Table 1. Note that all give different outputs on this input, except TFQ and TLQ, which are of course different in general. A stronger statement, namely that all randomized versions are inequivalent algorithms, is shown by example in Appendix A.9. Also note that on this input, only PLQ fails to give an efficient allocation. Running TTC on the output of PLQ yields the allocation $1:a, 2:c, 3:b, 4:d$.

All algorithms can be interpreted in terms of a party game, with the host providing a stash of presents. As each person arrives at the party (say though a narrow door), they take a present from the stash or (in some cases) from another person. Permanent memory algorithms have a single round, and temporary memory algorithms begin a new round every time a new present is taken. For Accept-First algorithms, in each round each present can be taken at most once. For Accept-Last algorithms, in each round each (person, present) pair can occur at most once. The queue or stack discipline determines what happens to a partygoer when it loses its present: choose a replacement present immediately, or go to the back of the queue. The Accept-First queue-based algorithms would be uninteresting, as would PFS, but the others seem to us worth trying.

The TLS algorithm is closely related in this interpretation to the party game ***Yankee Swap*** or White Elephant [Con17], which was the inspiration for our research program. In the real game, presents are contributed by partygoers and are wrapped, so no person has full information on their own preference. We are not aware of any other real-life party games based on the other algorithms.

4 Properties of the algorithms

Most of our algorithms fail to satisfy any of the common axiomatic properties. However, some have good average-case behaviour. Interestingly, they behave quite differently from each other.

4.1 Ex-post Efficiency

Proposition 4.1. *All Accept-First algorithms are ex-post efficient.*

Proof. We show by induction on the round number that the partial allocation constructed so far is ex-post efficient (in each case a round ends every time a previously unmatched item is chosen — note that this is a different usage of "round" to that in Section 3.3, where permanent memory algorithms have a single round). The first round always terminates with the first agent taking its top choice, and this is obviously an ex-post efficient outcome. Suppose that the result holds for all rounds before i and consider round i. The entering agent chooses an item and retains it throughout the round (by Accept-First policy). Each agent taking an unmatched item or stealing an item during this round (an "active agent") chooses the best item available. Such an agent j would only wish to trade with an agent k who has chosen since the last memory reset (in the permanent memory case, since the beginning of the algorithm; in the temporary memory case, since the beginning of the round). However in this case k will not wish to trade with j. Thus there can be no mutually beneficially trading cycle within the group of active agents. By inductive hypothesis there is also no such cycle within the group of inactive agents. Each agent in the active group prefers its current item to everything held by the inactive group because such items were available to steal. The result follows because TTC will terminate with no trades. □

Example 4.2. *All Accept-Last algorithms fail ex-post efficiency. To see this, consider the profile where agents 1,2,3 have respective preference orders $a > b > c, a > b > c, b > a > c$. Direct computation shows that for each algorithm X, there is some initial agent order such that algorithm XG formed by running TTC on the output of X gives a different result. Thus X cannot be ex-post efficient.*

4.2 Ordinal Efficiency

A random allocation S is ordinally efficient if there is not another random allocation, S' that each agent i SD-prefers S'_i to S_i, with at least 1 agent strictly SD-preferring their allocation under S'. Ordinal efficiency implies ex-post efficiency, but not vice versa [BM01].

Example 4.3. *All Accept-First algorithms fail ordinal efficiency. A counterexample (details omitted): suppose that agents 1, 2 prefer $a > b > c > d$, while agents 3, 4 have preference $a > b > d > c$. This same counterexample also works for the ex-post efficient algorithms PLSG and PLQG. A different counterexample with $n = 4$ works for TLSG and TLQG, and ordinal efficiency seems to be violated less often for these two algorithms.*

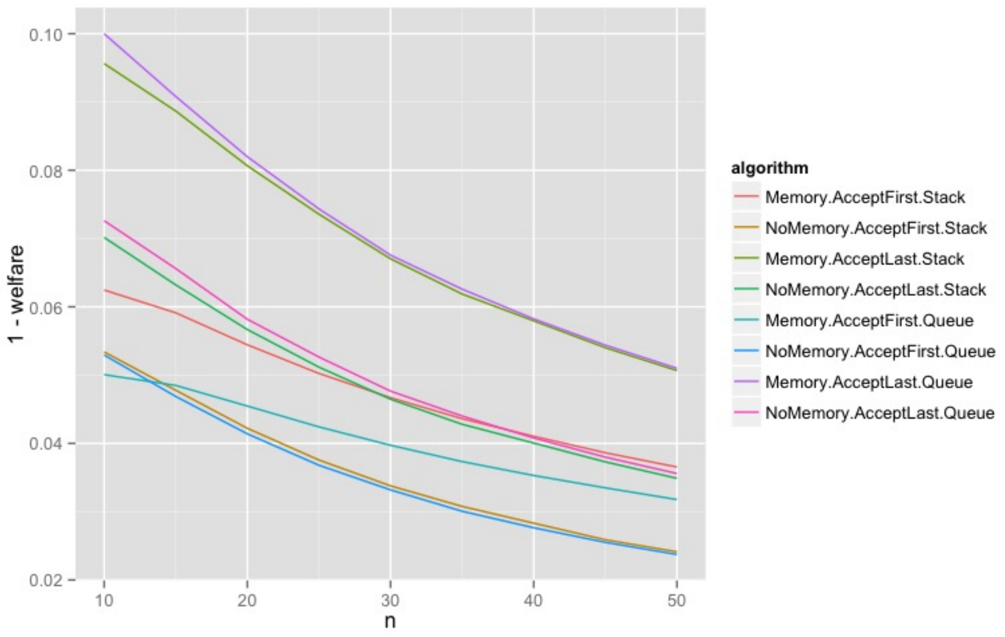
4.3 Strategyproofness

An assignment algorithm is ***strategyproof*** if no agent has incentive to misreport its preferences, irrespective of what other agents do. In other words, truthful reporting is a strictly dominant strategy for each agent. This condition is of course rather strong, but is known to be satisfied by Serial Dictatorship and hence by PFS. All other algorithms discussed here fail to satisfy it as we show below. For random allocation algorithms, the incentive is understood to be expressed in terms of expected utility in the usual way.

The algorithm Probabilistic Serial satisfies ***weak strategyproofness*** [BM01], which says that no agent has incentive to deviate, where the incentive is expressed in terms of first-order stochastic dominance. In other words, for each profile and each agent, there is ***some*** consistent choice of utility function for that agent for which deviation from truthfulness is unprofitable. Note that this is weaker than strategyproofness, which says that for each profile and agent, and for ***every*** consistent choice of utility function for that agent, deviation from truthfulness is unprofitable.

Clearly, if algorithm X is strategyproof for every initial order of agents, so is RX.

Figure 1: Utilitarian welfare loss of our 8 basic algorithms



We show by examples that none of the randomized versions of our algorithms are weakly strategyproof. Consider the case where agents 1, 2, 3, 4 all have preference $a > b > c > d$. If some agent submits instead $a > c > b > d$ while the others remain truthful, algorithms TLS, TLG, PLS, PLQ, TLQ, TLQG obtain a preferable outcome for that agent.

Similar examples show that PLQG, PLSG, TFS and TFQ, and PFQ all fail weak strategyproofness.

4.4 Utilitarian welfare

We give a basic analysis here, and refer the reader to our more extensive analysis [LW16]. We analyse here only the average-case performance of the randomized versions of the algorithms under truthful behaviour. We impute utility values of agents by requiring them to all have the same (Borda) utility function, whereby the ith choice corresponds to utility $n - 1 + i$. We consider the ***utilitarian social welfare***, which is the sum over all agents of the utility of their allocation. The optimal value of the utilitarian social welfare is efficiently computable, for example using the ***Hungarian algorithm*** []. This allows us to quantify the fraction of the maximum possible social welfare that is lost, on average, by each algorithm. We use a Java implementation by K. Stern [Ste].

Results are shown in Figure 1. They show that RSD is outperformed substantially by PFQ, TFQ and TFS, but is better than the Accept-Last algorithms. In Figure 2 we show how the welfare improves substantially when TTC is run on the output of our Accept-Last algorithms. In Figure 3 we show how our best-performing algorithms, namely TLQG and TLQS, compare with the standard algorithms RSD, PS and Naive Boston. In fact all the Accept-Last algorithms with TTC outperform those standard algorithms.

4.5 Egalitarian welfare

We also consider the egalitarian welfare, namely the welfare of the worst-off agent. Since exact computation of the optimum is difficult [], we scale by n instead of the exact optimum.

Results are similar to the utilitarian case and show the non-competitiveness of RSD and Naive Boston, the positive effect of GTTC, and the overall superiority of TLSG and TLQG. In Figure 4 the lines for TLSG and TLQG are indistinguishable. We do not compare with PS because it is inherently random and so comparison would be unfair to the other algorithms — the expectation of the minimum welfare is less than the same as the minimum of the expectations. Note that welfare increases with n for the new algorithms, but decreases for the old ones.

Figure 2: Utilitarian welfare improvement for Accept-Last with TTC

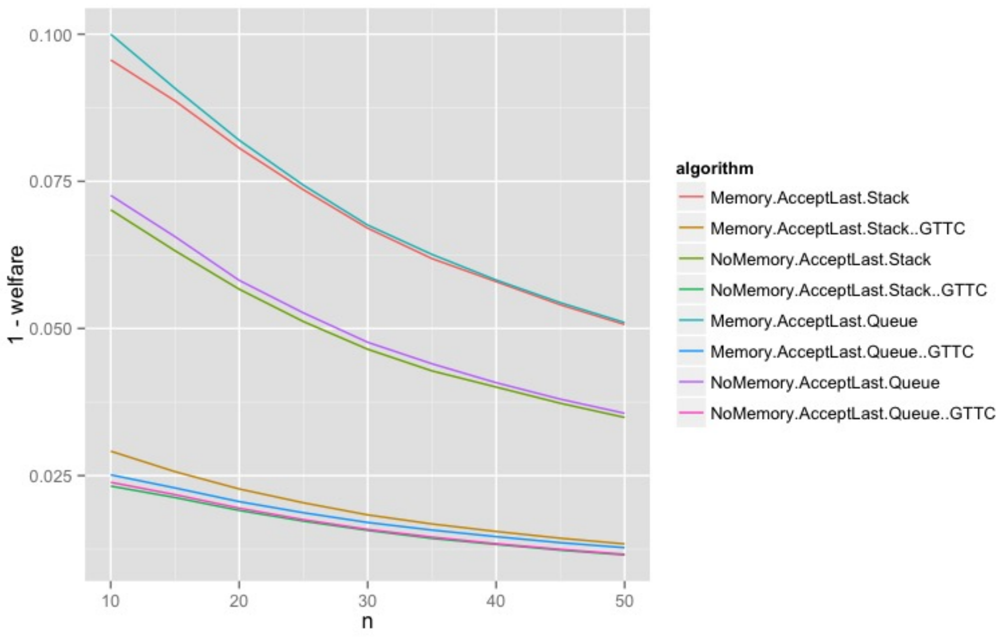


Figure 3: Utilitarian welfare loss comparison

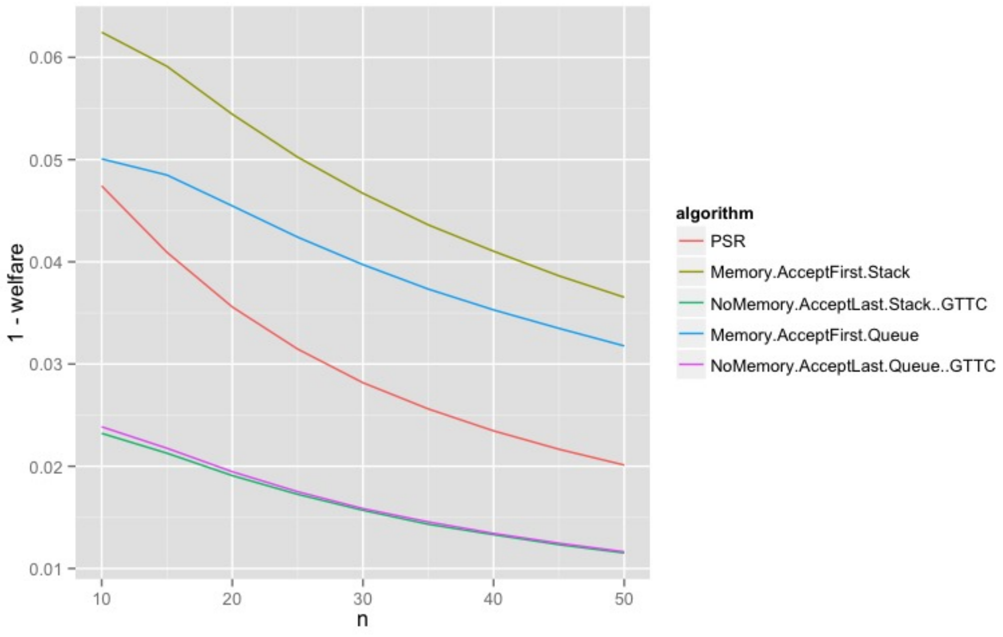
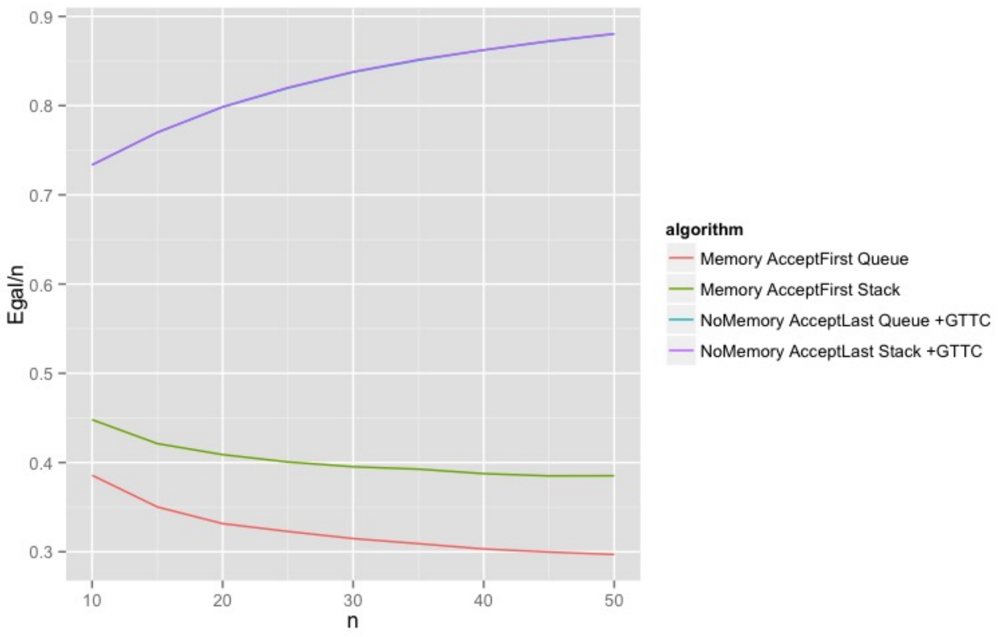


Figure 4: Normalized egalitarian welfare comparison



4.5.1 Egalitarian welfare bounds

Say that an algorithm satisfies a ***conditional egalitarian welfare bound*** of k if every agent receives one of its top k choices, ***whenever that is possible under some allocation***. If k happens to be the minimal possible value, this says that the algorithm yields optimal egalitarian Borda welfare on that input. All reasonable algorithms (including all those in this paper) satisfy a bound of 1, because if all agents have different top choices, each receives its top choice. We investigated the case $k = 2$ and found that none of our algorithms satisfy it in general (details omitted). However, TLS satisfies the bound with $k = 2$ when $n = 3$, as does its queue-based counterpart TLQ, while none of the other algorithms does. Thus for $n = 3$ these algorithms are egalitarian-optimal.

4.6 Order bias

In the case where all agents have the same preferences over items, some algorithms (such as serial dictatorship) are clearly biased toward the first agent while others (accept-last algorithms) are clearly biased toward the last agent. We define the ***order bias*** of an algorithm to be the maximum over all pairs of agents of the difference of the expected (under the uniform distribution on preferences) Borda welfare gained, and normalize by n.

Results show that the order bias of queue-based algorithms is markedly smaller than that for stack-based algorithms. Our best welfare algorithms, namely TLSG and TLQG, have almost zero order bias (PS has zero order bias by definition), but the randomized Serial Dictatorship and Naive Boston algorithms have substantial order bias, with the former being clearly more biased than all other algorithms. Adding TTC to the Accept-Last algorithms substantially reduces order bias (not shown).

5 Conclusion

5.1 Summary of results

We have introduced 10 new algorithms for 1-sided matching, none of them equivalent to each other or to any algorithms in the literature, to our knowledge. Their derivation using the Gale-Shapley framework gives a unified description. Each algorithm runs in worst-case time of order n^2 (permanent memory) or n^3 (temporary memory). Although they lack strong axiomatic foundations, several of the algorithms perform well on criteria such as egalitarian welfare, utilitarian welfare, and order bias, in sharp contrast to the standard

Figure 5: Normalized order bias of our 8 basic algorithms

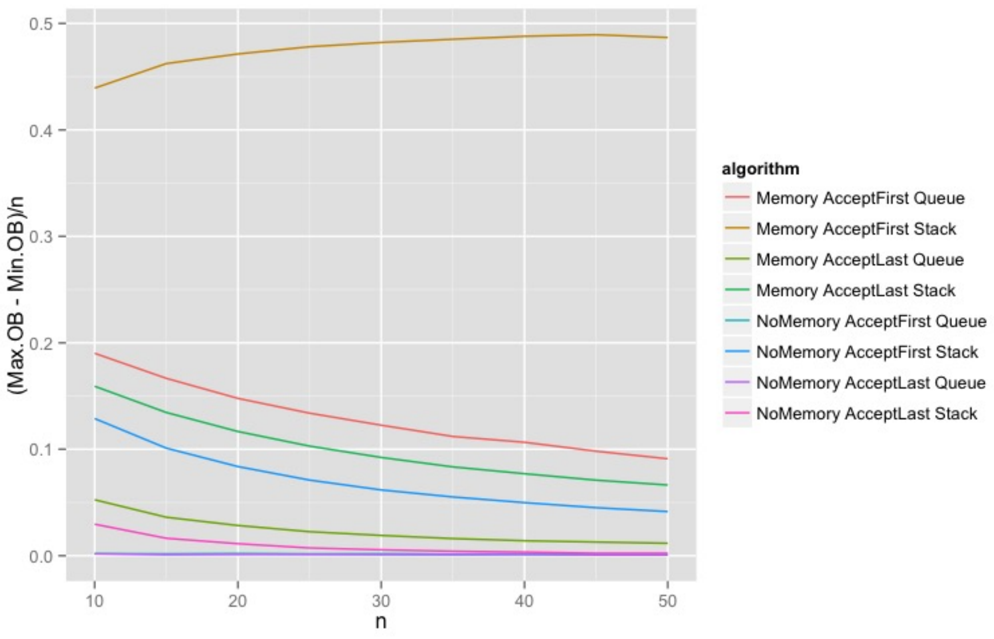
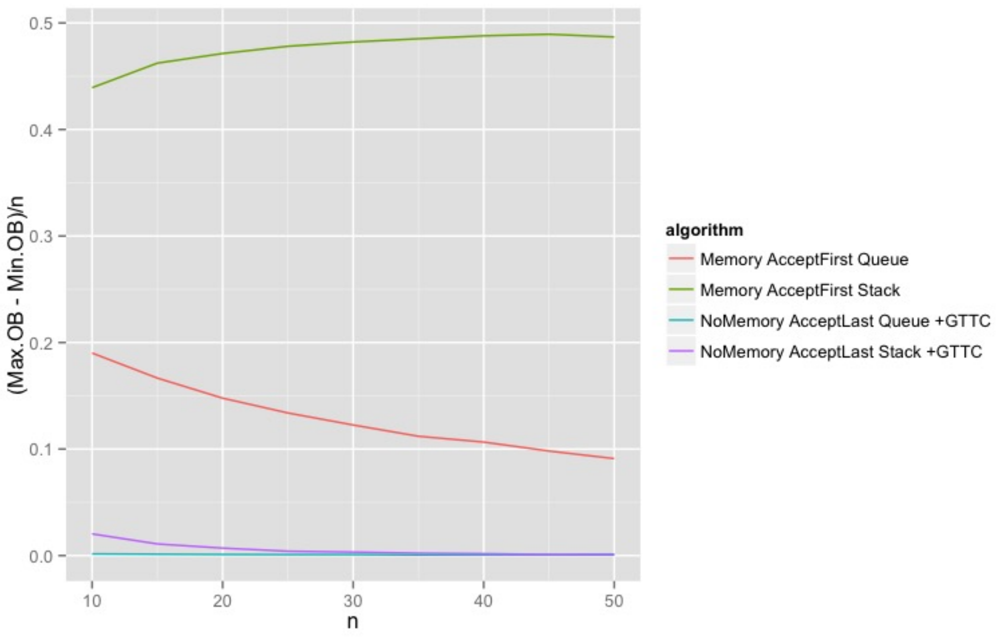


Figure 6: Normalized order bias comparison



Serial Dictatorship or Boston algorithms. The algorithms TLSG and TLQG perform better overall than the standard algorithms Serial Dictatorship, Naive Boston, and Probabilistic Serial on such measures.

In this article we have introduced a new (to our knowledge) performance criterion related to symmetry, namely order bias. The queue-based algorithms perform better overall than the stack-based ones by this measure. The algorithm TLQ (and TLQG) has remarkably small order bias. Our intuition is that the queue simulates randomization of the order of choosing by agents.

The approach we use of considering algorithms that are not ex-post efficient, and then running TTC on their output, seems new to us. Ex-post efficiency is, roughly speaking, a local optimum criterion. By avoiding prematurely locking in efficiency, our new Accept-Last + TTC algorithms seem to be able to achieve efficient outcomes with higher global welfare.

The benefits of using temporary versus permanent memory are relatively small compared to the gains made by using TTC, for example. However they are real and for situations where solution quality is substantially more important than runtime, we recommend its use. The best overall algorithms in terms of solution quality are arguably TLSG and TLQG. The latter has lower order bias and the former higher welfare, although the differences between the algorithms are small.

Some of our new algorithms may be useful for specialized situations (in addition to party games). For example, TFS seems to treat all agents equally in welfare except the last, who has a definite advantage — this may be useful, for example, when one agent is a small child. Having (almost) zero order bias is a strong fairness condition that may be very important in some applications. Our 1-sided Naive Boston algorithm PFQ maximizes the number of agents receiving their first choice.

Enlarging the stock of basic algorithms has more benefits than simply allowing us 10 more algorithms to choose from. The concept of ***hybridization*** has been used by Mennle & Seuken [MS13]. This simply forms a new algorithm $(1-p)A + pB$ from random allocation algorithms A and B and a fixed $p \in [0,1]$ by taking the convex combination $(1-p)M_A + pM_B$ of the stochastic matrices output A and B. This allows us to trade off desirable properties such as strategyproofness and efficiency in a controlled way. Mennle & Seuken considered only RSD, PS, Naive Boston and the algorithm maximizing utilitarian welfare as their basic algorithms. We believe that our new algorithms will prove useful as building blocks for hybrid algorithms with good overall behaviour.

5.2 Future work

The Adaptive Boston school choice algorithm improves over Naive Boston by allowing agents to skip proposals that will obviously be rejected because a school has reached capacity. It satisfies a property intermediate between weak strategyproofness and strategyproofness, called ***partial strategyproofness*** by Mennle & Seuken [MS14]. Although we found a sequential interpretation of Naive Boston that avoided discussion of simultaneous proposals, we have not yet done this for Adaptive Boston.

Each of our algorithms extends to the school choice situation. The “memory" component and the “data structure" component translate directly with no changes required. The aceptance policy if more complicated. In school choice, schools accept applicants provisionally until capacity is reached, and then each new temporary enrolment requires an existing enrolment to be cancelled. Our Accept-Last and Accept-First policies in the capacity 1 case described above could also be termed “Reject-Current” or “Reject-New". In the general school choice situation we would need to create fictitious prefernces for schools to enable them to decide which current student to reject. Some obvious ways to do that include FIFO or LIFO.

The case where the number of items exceeds the number of agents, and agents receive bundles of items chosen one at a time, is complicated. The order in which agents should choose in each round (here a round ends when all agents have incremented their previous total of items by 1) must be specified. For example, when we have 2 agents with preferences $a > b > c > d$ over 4 items, and the picking sequence 1221, Serial Dictatorship awards a, d to agent 1 and b, c to agent 2. However under PFQ agent 2 attempts to get item a and fails, going to the back of the queue and hence missing that turn. In the next turn it is allocated b, then agent 1 tries for b and fails. Thus the picking sequence must be extended. Accept-Last algorithms work better in this situation.

We leave further exploration of these interesting cases with $m \neq n$ to future work.

References

[AS98] Atila Abdulkadiroğlu and Tayfun Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.

[AS03] Atila Abdulkadiroğlu and Tayfun Sönmez. School choice: A mechanism design approach. *The American Economic Review*, 93(3):729–747, 2003.

[BM01] Anna Bogomolnaia and Hervé Moulin. A new solution to the random assignment problem. *Journal of Economic theory*, 100(2):295–328, 2001.

[Con17] Wikipedia Contributors. White elephant gift exchange — Wikipedia, the free encyclopedia, 2017. [Online; accessed 2017-03-13].

[GS62] David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[Kes09] Onur Kesten. Why do popular mechanisms lack efficiency in random environments? *Journal of Economic Theory*, 144(5):2209–2226, 2009.

[LW16] Jacky Lo and Mark C. Wilson. Average-case analysis of random assignment algorithms. *Preprint*, 2016.

[MS13] T. Mennle and S. Seuken. Hybrid Mechanisms: Trading off Efficiency and Strategyproofness in One-Sided Matching. *ArXiv e-prints*, March 2013.

[MS14] Timo Mennle and Sven Seuken. The Naive versus the Adaptive Boston Mechanism. *arXiv preprint arXiv:1406.3327*, 2014.

[SS74] Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical Economics*, 1(1):23–37, 1974.

[Ste] Kevin Stern. Hungarianalgorithm.java. `https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/optimization/HungarianAlgorithm.java`.

Appendices

A Sample executions of algorithms

For a profile where agent 1, 2 and 3 have preferences $a > b > c > d$, and agent 4 have preferences $b > a > c > d$, and the initial ordering of agents being in ascending numerical order, the sequence of proposal and rejection using the various algorithms will be as follows.

A.1 Permanent Memory, Early-Proposal preferred, Stack

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a
a_2 => a	a_2 rejected	a_2, a_3, a_4	a_1:a
a_2 => b	tentatively matched	a_3, a_4	a_1:a, a_2:b
a_3 => a	a_3 rejected	a_3, a_4	a_1:a, a_2:b
a_3 => b	a_3 rejected	a_3, a_4	a_1:a, a_2:b
a_3 => c	tentatively matched	a_4	a_1:a, a_2:b, a_3:c
a_4 => b	a_4 rejected	a_4	a_1:a, a_2:b, a_3:c
a_4 => a	a_4 rejected	a_4	a_1:a, a_2:b, a_3:c
a_4 => c	a_4 rejected	a_4	a_1:a, a_2:b, a_3:c
a_4 => d	tentatively matched		a_1:a, a_2:b, a_3:c, a_4:d

A.2 Permanent Memory, Early-Proposal preferred, Queue

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a
a_2 => a	a_2 rejected	a_3, a_4, a_2	a_1:a
a_3 => a	a_3 rejected	a_4, a_2, a_3	a_1:a
a_4 => b	tentatively matched	a_2, a_3	a_1:a, a_4:b
a_2 => b	a_2 rejected	a_3, a_2	a_1:a, a_4:b
a_3 => b	a_3 rejected	a_2, a_3	a_1:a, a_4:b
a_2 => c	tentatively matched	a_3	a_1:a, a_2:c, a_4:b
a_3 => c	a_3 rejected	a_3	a_1:a, a_2:c, a_4:b
a_3 => d	tentatively matched		a_1:a, a_2:c, a_3:d, a_4:b

A.3 Permanent Memory, Last-Proposal preferred, Stack

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a
a_2 => a	a_1 rejected	a_1, a_3, a_4	a_2:a
a_1 => b	tentatively matched	a_3, a_4	a_1:b, a_2:a
a_3 => a	a_2 rejected	a_2, a_4	a_1:b, a_3:a
a_2 => b	a_1 rejected	a_1, a_4	a_2:b, a_3:a
a_1 => c	tentatively matched	a_4	a_1:c, a_2:b, a_3:a
a_4 => b	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b
a_2 => c	a_1 rejected	a_1	a_2:c, a_3:a, a_4:b
a_1 => d	tentatively matched		a_1:d, a_2:c, a_3:a, a_4:b

A.4 Permanent Memory, Last-Proposal preferred, Queue

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a
a_2 => a	a_1 rejected	a_3, a_4, a_1	a_2:a
a_3 => a	a_2 rejected	a_4, a_1, a_2	a_3:a
a_4 => b	tentatively matched	a_1, a_2	a_3:a, a_4:b
a_1 => b	a_4 rejected	a_2, a_4	a_1:b, a_3:a
a_2 => b	a_1 rejected	a_4, a_1	a_2:b, a_3:a
a_4 => a	a_3 rejected	a_1, a_3	a_2:b, a_4:a
a_1 => c	tentatively matched	a_3	a_1:c, a_2:b, a_4:a
a_3 => c	a_1 rejected	a_1	a_2:b, a_3:c, a_4:a
a_1 => d	tentatively matched		a_1:d, a_2:c, a_3:b, a_4:a

A.5 Temporary Memory, Early-Proposal preferred, Stack

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching	Item Preferences
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a	none
a_2 => a	a_1 rejected	a_1, a_3, a_4	a_2:a	i_a:$a_2 > a_1$
a_1 => b	tentatively matched	a_3, a_4	a_1:b, a_2:a	none
a_3 => a	a_2 rejected	a_2, a_4	a_1:b, a_3:a	i_a:$a_3 > a_2$
a_2 => a	a_2 rejected	a_2, a_4	a_1:b, a_3:a	i_a:$a_3 > a_2$
a_2 => b	a_1 rejected	a_1, a_4	a_2:b, a_3:a	i_a:$a_3 > a_2$
				i_b:$a_2 > a_1$
a_1 => a	a_1 rejected	a_1, a_4	a_2:b, a_3:a	i_a:$a_3 > a_2 > a_1$
				i_b:$a_2 > a_1$
a_1 => b	a_1 rejected	a_1, a_4	a_2:b, a_3:a	i_a:$a_3 > a_2 > a_1$
				i_b:$a_2 > a_1$
a_1 => c	tentatively matched	a_4	a_1:c, a_2:b, a_3:a	none
a_4 => b	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_b:$a_4 > a_2$
a_2 => a	a_3 rejected	a_3	a_1:c, a_2:a, a_4:b	i_a:$a_2 > a_3$
				i_b:$a_4 > a_2$
a_3 => a	a_3 rejected	a_3	a_1:c, a_2:a, a_4:b	i_a:$a_2 > a_3$
				i_b:$a_4 > a_2$
a_3 => b	a_3 rejected	a_4	a_1:c, a_2:a, a_4:b	i_a:$a_2 > a_3$
				i_b:$a_4 > a_2 > a_3$
a_3 => c	a_1 rejected	a_1	a_2:a, a_3:c, a_4:b	i_a:$a_2 > a_3$
				i_b:$a_4 > a_2$
				i_c:$a_3 > a_1$
a_1 => a	a_1 rejected	a_1	a_2:a, a_3:c, a_4:b	i_a:$a_2 > a_3 > a_1$
				i_b:$a_4 > a_2$
				i_c:$a_3 > a_1$
a_1 => b	a_1 rejected	a_1	a_2:a, a_3:c, a_4:b	i_a:$a_2 > a_3 > a_1$
				i_b:$a_4 > a_2 > a_1$
				i_c:$a_3 > a_1$
a_1 => c	a_1 rejected	a_1	a_2:a, a_3:c, a_4:b	i_a:$a_2 > a_3 > a_1$
				i_b:$a_4 > a_2 > a_1$
				i_c:$a_3 > a_1$
a_1 => d	tentatively matched		a_1:d, a_2:a, a_3:c, a_4:b	none

A.6 Temporary Memory, Early-Proposal preferred, Queue

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching	Item Preferences
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a	none
a_2 => a	a_1 rejected	a_3, a_4, a_1	a_2:a	i_a:$a_2 > a_1$
a_3 => a	a_3 rejected	a_4, a_1, a_3	a_2:a	i_a:$a_2 > a_1 > a_3$
a_4 => b	tentatively matched	a_1, a_3	a_2:a, a_4:b	none
a_1 => a	a_2 rejected	a_3, a_2	a_1:a, a_4:b	i_a:$a_1 > a_2$
a_3 => a	a_3 rejected	a_2, a_3	a_1:a, a_4:b	i_a:$a_1 > a_2 > a_3$
a_2 => a	a_2 rejected	a_3, a_2	a_1:a, a_4:b	i_a:$a_1 > a_2 > a_3$
a_3 => b	a_4 rejected	a_2, a_4	a_1:a, a_3:b	i_a:$a_2 > a_1 > a_3$
				i_b:$a_3 > a_4$
a_2 => b	a_2 rejected	a_4, a_2	a_1:a, a_3:b	i_a:$a_2 > a_1 > a_3$
				i_b:$a_3 > a_4 > a_2$
a_4 => a	a_4 rejected	a_2, a_4	a_1:a, a_3:b	i_a:$a_2 > a_1 > a_3 > a_4$
				i_b:$a_3 > a_4 > a_2$
a_2 => c	tentatively matched	a_4	a_1:a, a_2:c, a_3:b	none
a_4 => b	a_3 rejected	a_3	a_1:a, a_2:c, a_4:b	i_b:$a_4 > a_3$
a_3 => a	a_1 rejected	a_1	a_2:c, a_3:a, a_4:b	i_a:$a_3 > a_1$
				i_b:$a_4 > a_3$
a_1 => a	a_1 rejected	a_1	a_2:c, a_3:a, a_4:b	i_a:$a_3 > a_1$

				i_b:$a_4 > a_3$
a_1 => b	a_1 rejected	a_1	a_2:c, a_3:a, a_4:b	i_a:$a_3 > a_1$ i_b:$a_4 > a_3 > a_1$
a_1 => c	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_1$ i_b:$a_4 > a_3 > a_1$ i_c:$a_1 > a_2$
a_2 => a	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_1 > a_2$ i_b:$a_4 > a_3 > a_1$ i_c:$a_1 > a_2$
a_2 => b	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_1 > a_2$ i_b:$a_4 > a_3 > a_1 > a_2$ i_c:$a_1 > a_2$
a_2 => c	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_1 > a_2$ i_b:$a_4 > a_3 > a_1 > a_2$ i_c:$a_1 > a_2$
a_2 => d	tentatively matched	a_1:c, a_2:d, a_3:a, a_4:b	none	
a_3 => b	a_1 rejected	a_4, a_1	a_2:a, a_3:b	i_a:$a_2 > a_1 > a_3$ i_b:$a_3 > a_1 > a_4$
a_4 => b	a_4 rejected	a_1, a_4	a_2:a, a_3:b	i_a:$a_2 > a_1 > a_3$ i_b:$a_3 > a_1 > a_4$
a_1 => c	tentatively matched	a_4	a_1:c, a_3:b, a_4:a	none
a_4 => b	a_3 rejected	a_3	a_1:c, a_2:a, a_4:b	i_b:$a_4 > a_3$
a_3 => a	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_2$ i_b:$a_4 > a_3$
a_2 => a	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_2$ i_b:$a_4 > a_3$
a_2 => b	a_4 rejected	a_4	a_1:c, a_2:b, a_3:a	i_a:$a_3 > a_2$ i_b:$a_2 > a_4 > a_3$
a_4 => a	a_3 rejected	a_4	a_1:c, a_2:b, a_4:a	i_a:$a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$
a_3 => b	a_3 rejected	a_3	a_1:c, a_2:b, a_4:a	i_a:$a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$
a_3 => c	a_1 rejected	a_1	a_2:b, a_3:c, a_4:a	i_a:$a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$ i_c:$a_3 > a_1$
a_1 => a	a_4 rejected	a_4	a_1:a, a_2:b, a_3:c	i_a:$a_1 > a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$ i_c:$a_3 > a_1$
a_4 => c	a_3 rejected	a_3	a_1:a, a_2:b, a_4:c	i_a:$a_1 > a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$ i_c:$a_4 > a_3 > a_1$
a_3 => d	tentatively matched		a_1:a, a_2:b, a_3:d, a_4:c	none

A.7 Temporary Memory, Last-Proposal preferred, Stack

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching	Item Preferences
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a	none
a_2 => a	a_1 rejected	a_1, a_3, a_4	a_2:a	i_a:$a_2 > a_1$
a_1 => b	tentatively matched	a_3, a_4	a_1:b, a_2:a	none
a_3 => a	a_2 rejected	a_2, a_4	a_1:b, a_3:a	i_a:$a_3 > a_2$
a_2 => b	a_1 rejected	a_1, a_4	a_2:b, a_3:a	i_a:$a_3 > a_2$
				i_b:$a_2 > a_1$
a_1 => a	a_3 rejected	a_3, a_4	a_1:a, a_2:b	i_a:$a_1 > a_3 > a_2$
				i_b:$a_2 > a_1$
a_3 => b	a_2 rejected	a_2, a_4	a_1:a, a_3:b	i_a:$a_1 > a_3 > a_2$
				i_b:$a_3 > a_2 > a_1$
a_2 => c	tentatively matched	a_4	a_1:a, a_2:c, a_3:b	none
a_4 => b	a_3 rejected	a_3	a_1:a, a_2:c, a_4:b	i_b:$a_4 > a_3$
a_3 => a	a_1 rejected	a_1	a_2:c, a_3:a, a_4:b	i_a:$a_3 > a_1$
				i_b:$a_4 > a_3$
a_1 => a	a_1 rejected	a_1	a_2:c, a_3:a, a_4:b	i_a:$a_3 > a_1$
				i_b:$a_4 > a_3$
a_1 => b	a_4 rejected	a_4	a_1:b, a_2:c, a_3:a	i_a:$a_3 > a_1$
				i_b:$a_1 > a_4 > a_3$
a_4 => a	a_3 rejected	a_3	a_1:b, a_2:c, a_4:a	i_a:$a_4 > a_3 > a_1$
				i_b:$a_1 > a_4 > a_3$
a_3 => b	a_3 rejected	a_3	a_1:b, a_2:c, a_4:a	i_a:$a_4 > a_3 > a_1$
				i_b:$a_1 > a_4 > a_3$
a_3 => c	a_2 rejected	a_2	a_1:b, a_3:c, a_4:a	i_a:$a_4 > a_3 > a_1$
				i_b:$a_1 > a_4 > a_3$
				i_c:$a_3 > a_2$
a_2 => a	a_4 rejected	a_4	a_1:b, a_2:a, a_3:c	i_a:$a_2 > a_4 > a_3 > a_1$
				i_b:$a_1 > a_4 > a_3$
				i_c:$a_3 > a_2$
a_4 => c	a_3 rejected	a_3	a_1:b, a_2:a, a_4:c	i_a:$a_2 > a_4 > a_3 > a_1$
				i_b:$a_1 > a_4 > a_3$
				i_c:$a_3 > a_2$
a_3 => d	tentatively matched		a_1:b, a_2:a, a_3:d, a_4:c	none

A.8 Temporary Memory, Last-Proposal preferred, Queue

Proposal	Outcome	Order of Remaining Agents	Current Partial Matching	Item Preferences
a_1 => a	tentatively matched	a_2, a_3, a_4	a_1:a	none
a_2 => a	a_1 rejected	a_3, a_4, a_1	a_2:a	i_a:$a_2 > a_1$
a_3 => a	a_2 rejected	a_4, a_1, a_2	a_3:a	i_a:$a_3 > a_2 > a_1$
a_4 => b	tentatively matched	a_1, a_2	a_3:a, a_4:b	none
a_1 => a	a_3 rejected	a_2, a_3	a_1:a, a_4:b	i_a:$a_1 > a_3$
a_2 => a	a_1 rejected	a_3, a_1	a_2:a, a_4:b	i_a:$a_2 > a_1 > a_3$
a_3 => a	a_3 rejected	a_1, a_3	a_2:a, a_4:b	i_a:$a_2 > a_1 > a_3$
a_1 => b	a_4 rejected	a_3, a_4	a_1:b, a_2:a	i_a:$a_2 > a_1 > a_3$ i_b:$a_1 > a_4$
a_3 => b	a_1 rejected	a_4, a_1	a_2:a, a_3:b	i_a:$a_2 > a_1 > a_3$ i_b:$a_3 > a_1 > a_4$
a_4 => b	a_4 rejected	a_1, a_4	a_2:a, a_3:b	i_a:$a_2 > a_1 > a_3$ i_b:$a_3 > a_1 > a_4$
a_1 => c	tentatively matched	a_4	a_1:c, a_3:b, a_4:a	none
a_4 => b	a_3 rejected	a_3	a_1:c, a_2:a, a_4:b	i_b:$a_4 > a_3$
a_3 => a	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_2$ i_b:$a_4 > a_3$
a_2 => a	a_2 rejected	a_2	a_1:c, a_3:a, a_4:b	i_a:$a_3 > a_2$ i_b:$a_4 > a_3$
a_2 => b	a_4 rejected	a_4	a_1:c, a_2:b, a_3:a	i_a:$a_3 > a_2$ i_b:$a_2 > a_4 > a_3$
a_4 => a	a_3 rejected	a_4	a_1:c, a_2:b, a_4:a	i_a:$a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$
a_3 => b	a_3 rejected	a_3	a_1:c, a_2:b, a_4:a	i_a:$a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$
a_3 => c	a_1 rejected	a_1	a_2:b, a_3:c, a_4:a	i_a:$a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$ i_c:$a_3 > a_1$
a_1 => a	a_4 rejected	a_4	a_1:a, a_2:b, a_3:c	i_a:$a_1 > a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$ i_c:$a_3 > a_1$
a_4 => c	a_3 rejected	a_3	a_1:a, a_2:b, a_4:c	i_a:$a_1 > a_4 > a_3 > a_2$ i_b:$a_2 > a_4 > a_3$ i_c:$a_4 > a_3 > a_1$
a_3 => d	tentatively matched		a_1:a, a_2:b, a_3:d, a_4:c	none

A.9

Proposition A.1. *The randomized versions of each of the 8 algorithms introduced in this article are all inequivalent.*

Proof. For a profile where agents 1, 2 and 3 have preferences $a > b > c > d$, and agent 4 has preferences $a > c > d > b$, the algorithms make the following allocations.

Alias	Memory	Item Preference	Data Structure	Probabilistic Allocation
Random Serial	Permanent	Accept-First	Stack	1/4, 1/3, 1/6, 1/4
Dictatorship				1/4, 1/3, 1/6, 1/4
				1/4, 1/3, 1/6, 1/4
				1/4, 0 , 1/2, 1/4
Naive Boston	Permanent	Accept-First	Queue	1/4, 1/3, 1/12, 1/3
				1/4, 1/3, 1/12, 1/3
				1/4, 1/3, 1/12, 1/3
				1/4, 0 , 3/4, 0
	Memory	Accept-Last	Stack	1/4, 1/3, 1/4, 1/6
				1/4, 1/3, 1/4, 1/6
				1/4, 1/3, 1/4, 1/6
				1/4, 0 , 1/4, 1/2
	Memory	Accept-Last	Queue	1/4, 1/3, 1/3, 1/12
				1/4, 1/3, 1/3, 1/12
				1/4, 1/3, 1/3, 1/12
				1/4, 0 , 0 , 3/4
Iterative Dictatorship	Temporary	Accept-First	Stack	1/4, 1/3, 1/4, 1/6
				1/4, 1/3, 1/4, 1/6
				1/4, 1/3, 1/4, 1/6
				1/4, 0 , 1/4, 1/2
	Temporary	Accept-First	Queue	1/3, 1/3, 1/4, 1/12
				1/3, 1/3, 1/4, 1/12
				1/3, 1/3, 1/4, 1/12
				0 , 0 , 1/4, 3/4
Yankee Swap	Temporary	Accept-Last	Stack	1/12, 1/3, 1/3, 1/4
				1/12, 1/3, 1/3, 1/4
				1/12, 1/3, 1/3, 1/4
				3/4, 0 , 0 , 1/4
	Temporary	Accept-Last	Queue	1/12, 1/3, 1/3, 1/4
				1/12, 1/3, 1/3, 1/4
				1/12, 1/3, 1/3, 1/4
				3/4, 0 , 0 , 1/4

□