Stable roommates problem with random preferences

Stephan Mertens

Institut für Theoretische Physik, Otto-von-Guericke Universität, PF 4120, 39016 Magdeburg, Germany

Santa Fe Institute, 1399 Hyde Park Rd, Santa Fe, NM 87501, USA

E-mail: mertens@ovgu.de

Abstract. The stable roommates problem with n agents has worst case complexity $O(n^2)$ in time and space. Random instances can be solved faster and with less memory, however. We introduce an algorithm that has average time and space complexity $O(n^{\frac{3}{2}})$ for random instances. We use this algorithm to simulate large instances of the stable roommates problem and to measure the probabilty p_n that a random instance of size n admits a stable matching. Our data support the conjecture that $p_n = \Theta(n^{-1/4})$.

Contents

1. Introduction

Matching under preferences is a topic of great practical importance, deep mathematical structure, and elegant algorithmics [1,2]. The most famous example is the stable marriage problem, where n men and n women compete with each other in the 'marriage market'. Each man ranks all the women according to his individual preferences, and each woman does the same with all men. Everybody wants to get married to someone at the top of his or her list, but mutual attraction is not symmetric and frustration and compromises are unavoidable. A minimum requirement is a matching of men and women such that no man and woman would agree to leave their assigned partners in order to marry each other. Such a matching is called stable since no individual has an incentive to break it. The problem then is to find such a stable matching.

The stable marriage problem was introduced by Gale and Shapley in 1962 [3]. In their seminal paper they proved that each instance of the marriage problem has at least one stable solution, and they presented an efficient algorithm to find it. Since then, the Gale–Shapley algorithm has been applied to many real-world problems, not by dating agencies but by central bodies that organize two-sided markets like the assignment of students to colleges or residents to hospitals [4]. Besides its practical relevance, the stable marriage problem has many interesting theoretical features that have attracted researchers from various field, including physics [5–9].

The salient feature of the stable marriage problem is its bipartite structure: the agents form two groups (men and women), and matchings are only allowed between these groups but not within a group. This is adequate for two-sided markets. But what about one-sided markets, like the formation of cockpit crews from a pool of pilots or the assignment

of students to the double bedrooms in a dormitory? The latter is known as the stable roommates problem. It is the paradigmatic example for matchings in one-sided markets.

The stable roommates problem was also introduced by Gale and Shapley [3]. They noted an intriguing difference between the marriage and the roommates problem: Whereas the former always has a solution, the latter may have none.

The Gale–Shapley algorithm for bipartite matching does not work for non-bipartite problems like the stable roommates problem. In fact some people believed that the roommates problem was NP-complete [10], but more than 20 years after the Gale–Shapley paper, Robert Irving presented a polynomial time algorithm for the stable roommates problem [11]. Irving's algorithm either yields a stable solution or 'No' if none exists.

An instance of the stable roommates problem consists of an even number n of persons (students, pilots), each of whom ranks all of the others in strict order of preference. Since each person has to keep a list of preferences for all $n - 1$ other persons, an instance of the stable roommates problem has size $\Theta(n^2)$. Irving's algorithm has time complexity $O(n^2)$. This is optimal if we assume that one has to look at the complete instance (or at least a finite fraction of it) in order to solve the problem.

In this paper we show that in random instances, Irving's algorithm only looks at $O(\sqrt{n})$ entries in each preference list, and we provide a modification of the algorithm that has average time and space complexity $O(n^{\frac{3}{2}})$. We use this algorithm to compute the probability p_n that a random instance of size n has a solution for systems that are more than 500 times larger than previously simulated systems [12].

The paper is organized as follows. We start with a review of Irving's algorithm. In section 3 we discuss the complexity of Irving's algorithm for random instances and our modification that reduces the average time and space complexity from $O(n^2)$ to $O(n^{\frac{3}{2}})$. Section 4 comprises the results of the simulations on p_n, obtained with the modified algorithm.

2. The algorithm

Irving's algorithm can be expressed as a sequence of 'proposals' from one person to another. If person x makes a proposal to person y (to share a room, to form a cockpit crew etc), y can accept or reject this proposal. If y accepts the proposal, x becomes *semiengaged* to y. If y later receives another proposal from someone he prefers to x, he will accept the new proposal and cancel the semiengagement from x, who will in turn look for someone else to propose to.

As the name suggests, semiengagement is not symmetric: if x is semiengaged to y, y can be semiangaged to $z \neq x$ or to no one. If all semiengagements are symmetric, they represent a matching.

Irving's algorithm proceeds in two phases. Phase I sets up semiengagements for everybody. In phase II, these semiengagements are modified by cyclically swapping partners until all semiengagements are symmetric, i.e. until they represent a matching. The corresponding sequence of proposals (and breakups) is organized such that the resulting matching is stable. If the instance admits no stable matching, this is recognized either in phase I or phase II by running out of partners to propose to.

For the time being, we assume that the preferences of all participants are stored in two 2D arrays:

- person$[x, i]$: person on position i in x's list,

- rank$[x, y]$: position of person y in x's list.

The two arrays are not independent, of course, but the redundancy allows us to look up persons and ranks in time $O(1)$.

For random instances we initialize the preference list of person x a by random permutation of all other persons (including x) and then move x to the very end of its own preference list. This means that we allow x being matched with himself as the worst choice. If this really happens, this means that x has no proper partner, i.e. that no stable matching exists for that instance.

In our implementation of the algorithm we will access the preference lists only through the function

function GETDATA(x, i)
    $y :=$ person$[x, i]$
    $r :=$ rank$[y, x]$
    return (y, r)
end function

which returns the pair (y, r) where y is the person with rank i in x's prefence list and r is rank of person x in y's preference list.

We will describe both phases of Irving's algorithm without proving their correctness. For the proofs we refer the reader to Irving's original paper [11].

2.1. Phase I

Phase I of the algorithm tries to establish semiengagements for every person. The general idea is that the first proposal of x goes to the first person on his preference list, and only if this proposal is rejected (immediately or subsequently), x proposes to the second person on his preference list and so on. On the receiving side, y accepts a proposal only if the proposing person ranks higher on his preference list than the person whose proposal he has currently accepted.

Imagine the list of preferences written horizontally left (most desired partner) to right (least desired partner). Then the proposals *made* move from left to right, while the proposals *accepted* move from right to left. In a matching, both types of proposals meet at the same position. This motivates the names for the following lists that hold the current set of proposals:

- leftperson$[x]$: the person whom x is currently proposing to,

- leftrank$[x]$: the rank of that person in x's preference list,

- rightperson$[x]$: the person from which x is currently holding a proposal

- rightrank[x]: the rank of that person in x's preference list.

Again these lists are not independent, but the redundancy allows a faster lookup especially in phase II.

With this lists, the semiengagement of x to y is expressed by the simultaneous validity of the identities $y = $ leftperson[x] and $x = $ rightperson[y].

Algorithm 1. Phase I of the stable roommates algorithm.

```
 1: procedure PHASE_I
 2:     for x := 1, . . . , n do                                    ▷ initialization
 3:         holds_proposal[x] := false          ▷ no one is holding a proper proposal...
 4:         rightperson[x] := x                          ▷ ...but a proposal from self
 5:         rightrank[x] := n                              ▷ self proposals are the worst
 6:         leftrank[x] := 1
 7:     end for
 8:     for x := 1 . . . n do
 9:         proposer := x
10:         repeat
11:             (next, rank) := GETDATA proposer, leftrank[proposer]
12:             while rank > rightrank[next] do
13:                 leftrank[proposer] := leftrank[proposer]+1
14:                 (next, rank) := GETDATA proposer, leftrank[proposer]
15:             end while
16:             previous := rightperson[next]
17:             rightrank[next] := rank
18:             rightperson[next] := proposer
19:             leftperson[proposer] = next
20:             proposer := previous
21:         until  holds_proposal[next] = false
22:         holds_proposal[next] := true
23:         if leftrank[proposer] = n then          ▷ proposer engaged to himself...
24:             return false                    ▷ ... means: no stable matching possible
25:         end if
26:     end for
27:     return true
28: end procedure
```

Algorithm 1 shows the pseudocode for phase I of Irving's algorithm. It stops, when every person *holds* a proposal, which implies that every person has also *made* a proposal that has been accepted, i.e. that every person is semiengaged. It returns **false** if someone has run out of partners (and is therefore engaged to himself), which means that there is no stable matching for this instance. If it returns **true** we can still hope to find a stable matching in phase II of the algorithm.

2.2. Phase II

Phase I usually ends with everybody semiengaged to someone, but with asymmetric engagements leftrank[x] < rightrank[x] for most persons x. Such persons have to give up their current proposal to leftperson[x] and find somebody down the list that would (temporarily) accept a proposal from x. We keep track of these second choices in the following lists

- secondperson[x]: x's next best person that would accept a proposal

- secondrank[x]: the rank of that person in x's preference list,

- secondrightrank[x]: the rank of x in the preference list of secondperson[x].

If x withdraws his proposal and proposes to y = secondperson[x], who temporarily accepts the proposal, the previous partners of x and y both loose their semiengagements and have to look themselves for their second best partners and so on. This avalanche of break-ups and new propsals is called a rotation. It reduces the difference rightrank[x]–leftrank[x] for several persons x and is a step towards a matching.

The key idea of phase II is to organize this rearrangement of semiengagements in a so called all-or-nothing cycle. This is a sequence $a_1 \ldots, a_r$ of persons such that a_i's current second choice is a_{i+1}'s current first choice for $i = 1 \ldots, r - 1$, and a_r's current second choice is a_1's current first choice. In terms of our lists, an all-or-nothing cycle is given by

$$\text{secondperson}[a_i] = \text{leftperson}[a_{i+1}] \quad i = 1, \ldots, r - 1$$
$$\text{secondperson}[a_r] = \text{leftperson}[a_1] \,.$$

In phase II of Irving's algorithm, an all-or-nothing cycle is identified and the corresponding rotation is executed. This process is iterated until there are no more all-or-nothing cycles (in which case we've found a stable matching) or until someone runs out of partners after a rotation (in which case this instance has no stable matching).

Algorithm 2 shows the pseudocode for a function that finds and returns an all-or-nothing cycle or an empty cycle. To compute a cycle we need to identify the person whose current first choice is y = secondperson[x]. But this person is simply given by rightperson[y], i.e. it can be found in time $O(1)$ (see line 20 of algorithm 2).

Algorithm 3 shows pseudocode for phase II which finds an all-or-nothing cycle, executes the corresponding rotation and iterates this until there are no more all-or-nothing cycles or a rotation has left a person without any partners to propose to.

The complete algorithm consists of an initialization phase (not shown), which generates a random preference list for each person, followed by calls to PHASE_I and PHASE_II.

3. Analysis and modification

Figure 1 shows the average running times of the different phases on random instances of varying size n. The only phase that scales like $\Theta(n^2)$ is the initialization, i.e. the generation of the random permutation of the preference lists. The time for the actual solution (phase I

Algorithm 2. Finding an all-or-nothing cycle for phase II of the stable roommates algorithm.

1: **procedure** SEEKCYCLE( )
2:     **for** $x := 1, \ldots, n$ **do**
3:         **if** leftrank$[x] <$ rightrank$[x]$ **then break** $\triangleright$ find unmatched person
4:     **end for**
5:     **if** leftrank$[x] \geqslant$ rightrank$[x]$ **then** $\triangleright$ no unmatched person found
6:         **return** $(0, 0, \emptyset)$ $\triangleright$ return empty cycle
7:     **else** $\triangleright$ unmatched person found
8:         last $:= 1$
9:         **repeat**
10:            cycle[last] $:= x$
11:            last $:=$ last $+ 1$
12:            $p :=$ leftrank$[x]$
13:            **repeat** $\triangleright$ find second choice of x
14:                $p := p + 1$
15:                $(y, r) :=$ GETDATAx, p
16:            **until** $r \leqslant$ rightrank$[y]$ $\triangleright$ y would accept a proposal
17:            secondrank$[x] := p$ $\triangleright$ store second choice
18:            secondperson$[x] := y$
19:            secondrightrank$[x] := r$
20:            $x :=$ rightperson[secondperson$[x]$] $\triangleright$ next element in cycle
21:         **until** $x \in$ cycle $\triangleright$ cycle closed
22:         last $:=$ last $- 1$
23:         first $:=$ last $- 1$ $\triangleright$ rewind to start of cycle
24:         **while** cycle[first] $\neq x$ **do** first $:=$ first $- 1$
25:         **return** (first, last, cycle)
26:     **end if**
27: **end procedure**

---

and phase II) grows significantly slower than n^2, which implies that the algorithm doesn't need to look at the complete preference table to solve the problem.

Figure 2 shows the average number of entries in the preference lists that are actually read by Irving's algorithm in order to find a stable matching or to report that no stable matching exists. For large values of n, this number is $2n^{\frac{3}{2}}$. This can be understood by the following simple, albeit non rigorous consideration. Let k be the average number of proposals that a person makes in the course of Irving's algorithm. Then k is also the average number of proposals that a person receives, and the total number of entries in the preference table involved is $2kn$. Now leftrank$[x]$ increases by one with each proposal made by x, hence leftrank$[x] = k$ on average. The value of rightrank$[x]$ is given by the *minimum* of k values drawn uniformly from $1, \ldots, n$. Hence the distribution of rightrank$[x] = \ell$ is

$$P(\ell) = \frac{\binom{n-\ell}{k-1}}{\binom{n}{k}},$$

Algorithm 3. Phase II of the stable roommates algorithm.

```
1: procedure PHASE_II
2:     solution_possible := true
3:     solution_found := false
4:     while solution_possible and not solution_found do
5:         (first, last, cycle) := SEEKCYCLE( )
6:         if cycle is empty then solution_found := true
7:         else
8:             for x := cycle[first], ..., cycle[last] do
9:                 leftrank[x] := secondrank[x]
10:                leftperson[x] := secondperson[x]
11:                rightrank[leftperson[x]] := secondrightrank[x]
12:                rightperson[leftperson[x]] = x
13:            end for
14:            for x := cycle[first], ..., cycle[last] do
15:                if leftrank[x] > rightrank[x] then sol_possible := false
16:            end for
17:        end if
18:    end while
19:    return solution_found
20: end procedure
```

---

with mean value $\frac{n+1}{k+1}$. The algorithm terminates if leftrank$[x] \simeq$ rightrank$[x]$ or $k \simeq \frac{n+1}{k+1}$. Hence $k \simeq \sqrt{n}$, and the total number of entries read by Irving's algorithm is $2n^{\frac{3}{2}}$. Note that this consideration ignores the fluctuations in rightrank$[x]$: if $k \simeq \sqrt{n}$, the standard deviation of $P(\ell)$ is $O(\sqrt{n})$, hence the number of proposals received by an individual person x can differ considerably from the mean value $\sqrt{n}$. We do have the strict equality between the total number of proposals made and total number of proposals received, however. But a person, who has *received more* proposals than average, at termination has *made less* proposals than the average (and vice versa). Hence the individual fluctuations cancel in the total number of proposals if we assume, that at termination leftrank$[x] \simeq$ rightrank$[x]$ for almost all persons x. This is not obvious, since in most cases (see next section) the algorithm terminates when the first person runs out of partners to propose to, i.e. the first time that leftrank$[x] >$ rightrank$[x]$ for some x. It could well be that at that moment the gap between leftrank and rightrank is still large for some or many other persons. Hence the crucial assumption that underlies our argument is the assumption of a certain uniformity of the decrease of rightrank$[x]$–leftrank$[x]$ over all persons x. The fact that the observed total number of proposal is in fact narrowly concentrated around the predicted value (figure 2) is an indication that this assumption is justified.

The number of elements read in phase I is even smaller. Phase I terminates if every person holds a proposal. Consider the sequence $x_1, x_2, \ldots$ of persons that receive a proposal. Phase I terminates if this sequence contains every person at least once. If we assume that the x_i's are independent random variables, uniformly drawn from $\{1, \ldots, n\}$,
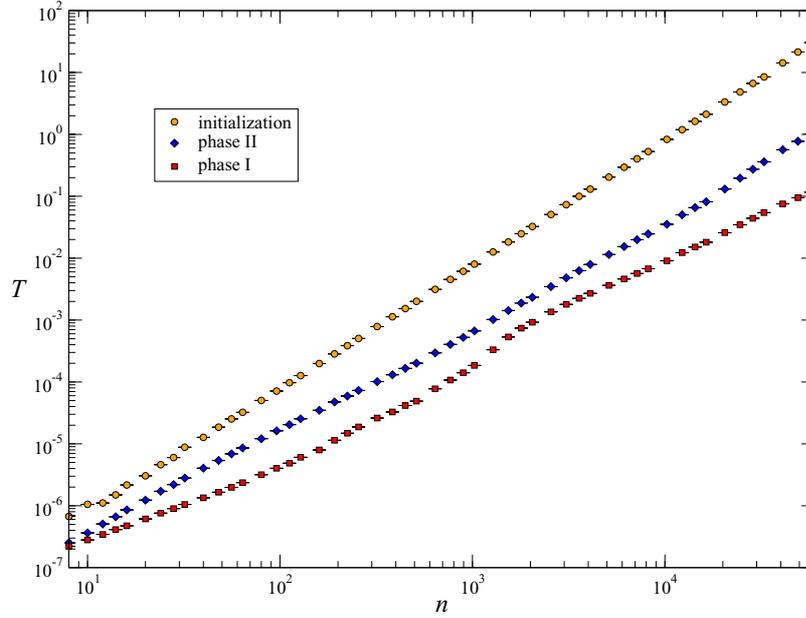
Figure 1. Running times of the various phases of Irving's algorithm for random instances of the stable roommates problem. The total running time is dominated by the time for initialization of the complete instance. This time scales like n^2, whereas the time for the actual solution (phases I and II) grows much slower. The time T is the average wallclock time in seconds on a single core of an Intel® Xeon® E5-1620 CPU running at 3.6 GHz. The 'bump' in the data for phase I is probably due to cache misses for larger systems.

this problem is known as the coupon collector's problem [10]: an urn contains n different coupons, and a collector draws coupons from that urn with replacement. How many coupons does the collector need to draw (on average), before he has drawn each coupon at least once? It is well known that the collector should expect to draw

$$n \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right) = nH_n$$

coupons in order to own at least one coupon of every kind. H_n is known as the nth harmonic number. Note that

$$nH_n = n \log n + \gamma n + \frac{1}{2} + O(n^{-1}), \tag{1}$$

where $\gamma = 0.577\,215\,6\ldots$ is the Euler–Mascheroni constant.

In our case, the n coupons are the proposals to the n different recipients, and phase I is the coupon collector. Hence the expected number of proposals in phase I is nH_n, and since each proposal implies two accesses to the preference lists, the number of elements read in phase I should be $2nH_n$. This is in fact the observed asymptotic scaling, as can be seen in figure 2.

We can exploit the fact that Irving's algorithm looks only at $O(n^{3/2})$ elements of preference table by generating and storing only the elements that are requested by the algorithm. This saves us the expensive initialization phase and reduces the memory consumption considerably. Algorithm 4 shows the corresponding version of the function GetData. It maintains two arrays (person and rank) of maps. A map (aka associative
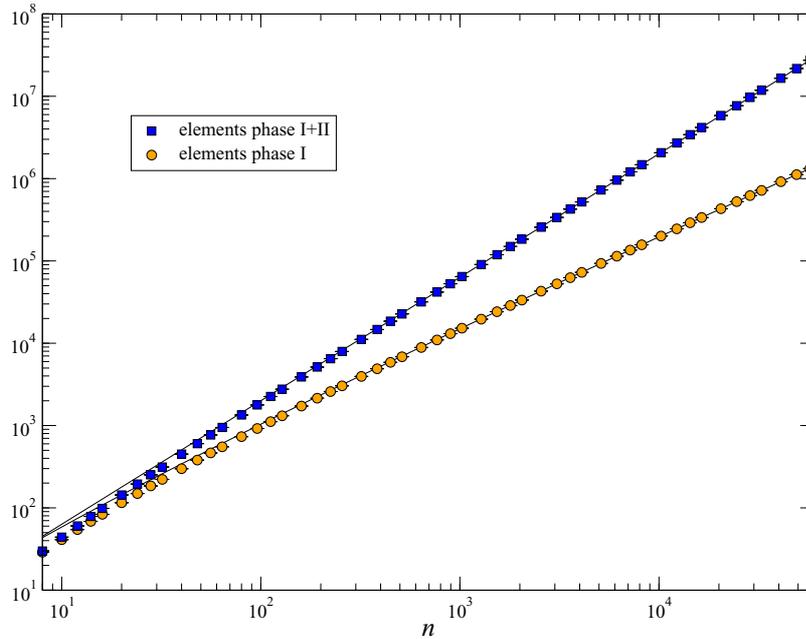
Figure 2. Number of elements in the preference tables that are actually read by Irving's algorithm. Each symbol represents an average over 10^4 random instances. The lines are $2n^{\frac{3}{2}}$ for the total number of elements and $2nH_n$ for phase I, where H_n is the nth harmonic number.

Algorithm 4. This version of the GetData function generates the entries of the preference lists 'on the fly' as they are requested. The preference list of person x is maintained in the associative arrays person$[x]$ and rank$[x]$

```
 1: function GETDATA(x, i)
 2:     if i ∈ rank[x] then                          ▷ element is known
 3:         y := person[x](i)
 4:         r := rank[y](x)
 5:     else                                         ▷ element is new
 6:         repeat
 7:             y := random                          ▷ generate random person
 8:         until y ∉ person[x]
 9:         repeat
10:             r := random                          ▷ generate random rank
11:         until r ∉ rank[y]
12:         person[x] := person[x] ∪ (y, i)          ▷ save new table entries
13:         rank[x] := rank[x] ∪ (i, y)
14:         person[y] := person[y] ∪ (x, r)
15:         rank[y] := rank[y] ∪ (r, x)
16:     end if
17:     return (y, r)
18: end function
```
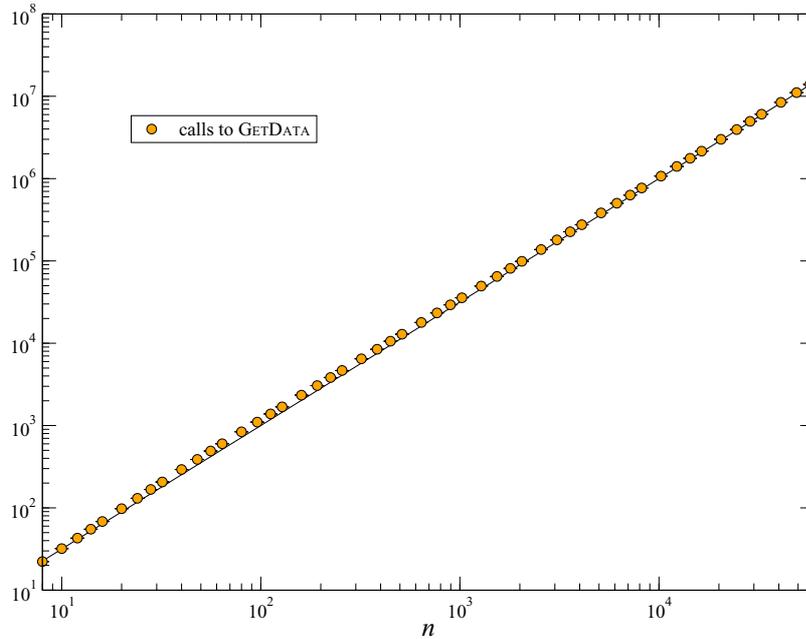

Figure 3. Number of calls of function GETDATA. Each symbol represents an average over 10^4 random instances. The line is $n^{\frac{3}{2}}$.

array) is a data structure that holds pairs (k, v) where k is the key and v is the value of the data element. In the map person[x], the key is the rank and the value is the person of that rank in x's preference list. The map rank[x] holds the same data elements but with the role of key and value reversed. The rationale behind this redundancy is again efficiency: using hash tables, a map can be implemented such that the lookup of a value given the key can be done in (expected) constant time, independent of the number of elements. Hence GETDATA has average time complexity $O(1)$ when the requested data element is already known. When the requested element is new, the generation of a new random element may take longer since we need to generate a random person y that is not contained in x's preference list so far and an unoccupied rank r for person x in y's list. Both are computed by a simple loop that generates random numbers until it hits a number not already contained in the list. In our case this is a reasonable approach since we know that the expected number of elements in person[x] and rank[y] is $O(\sqrt{n})$, hence the expected number of iterations in our loop is $1+O(n^{-1/2})$. Since inserting new elements in a map can also be done in constant time, the average time complexity of GETDATA is $O(1)$. Each map is initialized with the single entry person[x][n]=x and rank[x][x]=n, all other entries are only added as needed.

The function GETDATA is called unconditionally from within the innermost loop in phase I. In phase II it is called for each element in search for a cycle (including all cycle elements). Hence the number of calls of GETDATA is a good measure for the average time complexity of the algorithm. As can be seen in figure 3, the average time complexity is indeed $\Theta(n^{\frac{3}{2}})$.

The actual memory usage depends on the implementation of the map. We used the container class `unordered_map` from the C++ standard library provided by the GNU Compiler Collection (http://gcc.gnu.org). Figure 4 shows the total memory usage of the
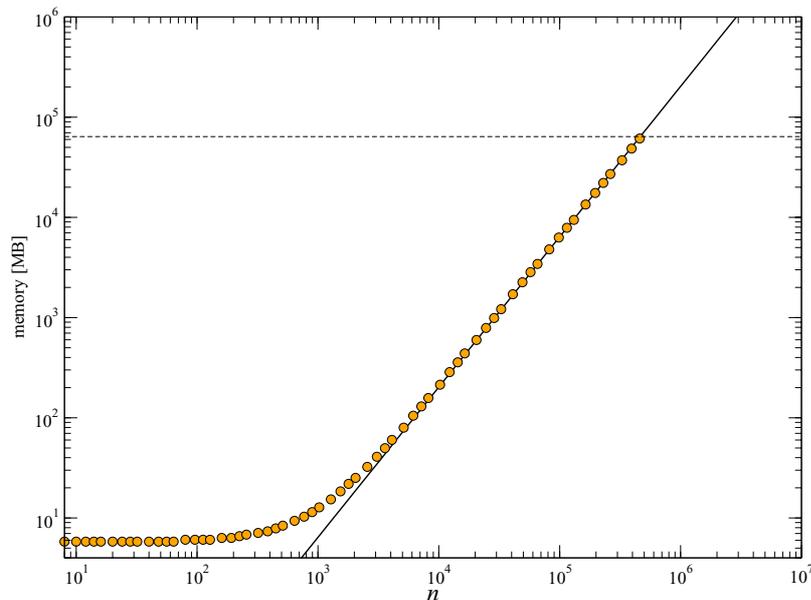
Figure 4. Actual memory consumption of Irving's algorithm for random instances, implemented with associative maps. The solid line is a numerical $\Theta(n^{3/2})$ fit, the dashed line marks the 64 GBytes limit of our hardware.

program. For large values of n it is dominated by the maps `person` and `rank` and scales like $n^{3/2}$. On a computer with 64 GBytes of memory our implementation allows us to run problems up to $n \simeq 470\,000$, which is more than 5 times larger than the maximum problem size allowed by a straightforward implementation with storage of the complete preference tables. The memory efficiency of our implementation is in fact so good that time becomes the bottleneck, as we will see in the next section.

4. Application

A long standing open problem is the computation of the probability p_n that a random instance of the stable roommates problem of size n has a stable matching [2, problem 8]. In particular one is interested in the asymptotic behavior of p_n as n grows large.

There is an integral representation for p_n that can be used to compute p_n exactly [13]. Unfortunately, the number of terms in the integral increases exponentially with n^2, which had limited the explicit evaluation of p_n to the case $n = 4$. Using a computer algebra system, we evaluated the integrals for $n \leqslant 10$ [14]:

$$p_4 = \frac{26}{27} = 0.962\,96\ldots$$

$$p_6 = \frac{181\,431\,847}{194\,400\,000} = 0.933\,29\ldots$$

$$p_8 = \frac{809\,419\,574\,956\,627}{889\,426\,440\,000\,000} = 0.910\,046\ldots$$

$$p_{10} = \frac{25\,365\,465\,754\,520\,943\,457\,921\,774\,207}{28\,460\,490\,127\,321\,448\,448\,000\,000\,000} = 0.891\,251\ldots.$$

Table 1. Values of p_n from simulations. Notation: $p_8 = 0.910\,048(5)$ means that $[0.910\,048 - 0.000\,005, 0.910\,048 + 0.000\,005]$ is the 95% confidence interval for p_8.

n	p_n	n	p_n	n	p_n	n	p_n
8	0.910\,048(5)	128	0.609\,86(1)	2048	0.324\,73(9)	32\,768	0.1650(4)
10	0.891\,247(6)	160	0.581\,83(1)	2560	0.307\,94(9)	40\,960	0.1563(4)
12	0.875\,525(6)	192	0.559\,46(1)	3072	0.294\,64(9)	49\,152	0.1494(3)
14	0.861\,952(6)	224	0.540\,99(1)	3584	0.283\,91(9)	57\,344	0.1440(5)
16	0.849\,958(7)	256	0.525\,36(1)	4096	0.274\,86(9)	65\,536	0.140(2)
20	0.829\,239(7)	320	0.499\,93(1)	5120	0.260\,42(9)	81\,920	0.132(2)
24	0.811\,499(7)	384	0.479\,87(1)	6144	0.2492(2)	98\,304	0.126(2)
28	0.795\,768(7)	448	0.463\,39(2)	7168	0.2402(2)	114\,688	0.120(2)
32	0.781\,542(8)	512	0.449\,49(2)	8192	0.2320(2)	131\,072	0.118(2)
40	0.756\,482(8)	640	0.427\,04(2)	10240	0.2200(2)	163\,840	0.111(2)
48	0.734\,851(8)	768	0.409\,39(7)	12\,288	0.2103(2)	196\,608	0.103(6)
56	0.715\,866(8)	896	0.394\,82(7)	14\,336	0.2024(3)	229\,376	0.104(6)
64	0.699\,044(9)	1024	0.382\,70(9)	16\,384	0.1961(3)	262\,144	0.098(6)
80	0.670\,377(9)	1280	0.363\,27(9)	20\,480	0.1854(4)	327\,680	0.097(6)
96	0.646\,797(9)	1536	0.347\,82(9)	24\,576	0.1774(4)	393\,216	0.089(6)
112	0.626\,92(1)	1792	0.335\,26(9)	28\,672	0.1709(4)	458\,752	0.085(5)

Computing p_n for larger values of n requires Monte Carlo simulations. These simulations indicate that p_n is a monotonically decreasing function of n, but early simulations up to $n = 2000$ [15] did not settle the question as to whether p_n converges to 0 or to some positive constant. The problem with simulations is that the decay of p_n is rather slow. In fact Pittel [13] proved the asymptotic lower bound

$$p_n \gtrsim \frac{2\mathrm{e}^{3/2}}{\sqrt{\pi n}} \tag{2}$$

by applying the second moment method to the number of stable matchings. Extended simulations [12] up to $n = 20\,000$ suggested an algebraic decay $p_n \simeq an^{-\delta}$. The numerical data from [12] were used to boldly conjecture the values of a and δ as

$$p_n \simeq \mathrm{e}\sqrt{\frac{2}{\pi}}\, n^{-1/4}. \tag{3}$$

Using our algorithm with reduced running time and memory consumption, we can check this conjecture against extended numerical data.

We simulated systems of size $n = n_0 2^k$, $k = 0, \ldots, k_{\max}$ and $n_0 \in \{8, 10, 12, 14\}$ where $k_{\max}$ is limited by the available memory. In our case this means $k_{\max} \leqslant 15$ (figure 4). The corresponding instances have an effective size that is more than 500 times larger than the largest systems investigated in [12].

To measure p_n, we generate and solve M independent random instances of size n and record the fraction $\hat{p}_n$ of samples that admit a stable matching. The 95% confidence interval for p_n is then $\hat{p}_n \pm 2\sigma_n$, where the standard deviation σ is given by

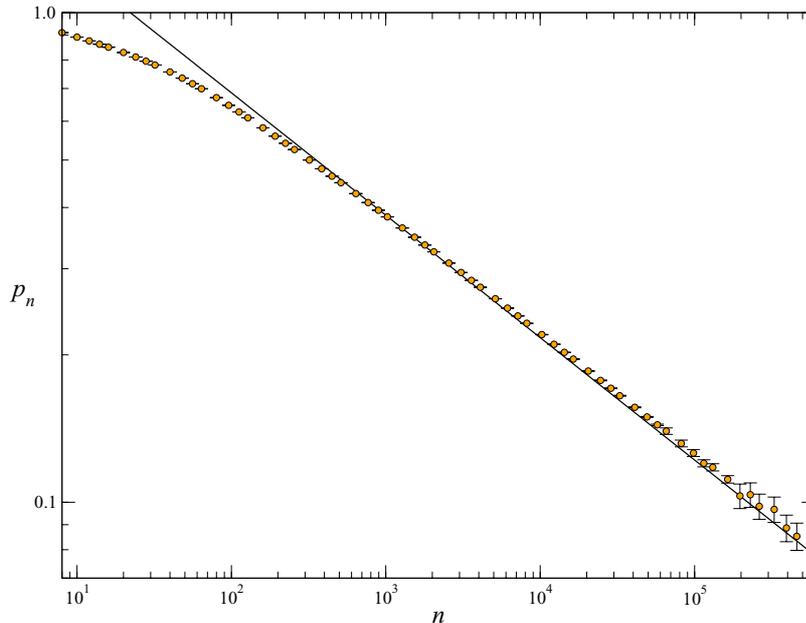$$\sigma_n = \sqrt{\frac{\hat{p}_n(1 - \hat{p}_n)}{M}}. \tag{4}$$

Figure 5. Simulation results for p_n in a log–log plot. Error bars denote 95% confidence intervals. The line represents the conjectured asymptotic behavior (3).

We vary the number of samples M with the system size n. We used values from $M = 10^{10}$ for small values of n down to $M = 10^4$ for the largest values of n. Table 1 shows the results.

We ran our simulation on a small cluster consisting of five nodes. Each node has 64 GByte of RAM and two Intel Xeon CPU E5-2630 running at 2.30 GHz. Each CPU has six real cores or (using hpyerthreading) twelve virtual cores. For smaller systems, we can use all $5 \times 2 \times 12 = 120$ virtual cores to solve instances in parallel, but for the larger sizes, the available memory per core limits the number of usable cores. The available memory allows us to compute problems up to $n = 2^{15} \times 14 = 458\,752$ using only one core per node. Solving a single instance of this size takes about 14 min, i.e. solving a sample of $M = 10^4$ instances as in table 1 takes about 20 d on five nodes in parallel.

Figure 5 shows p_n versus n in a log–log-plot. The data support an asymptotic algebraic decay $p_n \simeq a n^{-\delta}$ for some constants δ and a, in agreement with the conjecture (3), which is also displayed in figure 5. The visual impression suggests that $\delta = 1/4$ as claimed in (3), but that the true prefactor a is slightly larger than $\mathrm{e}\sqrt{2/\pi}$. In fact, a least squares fit of the one-parameter function $a n^{-1/4}$ to the data points for $n \geqslant 32\,765$ yields $a = 2.223(3)$, which is 3% larger than $\mathrm{e}\sqrt{2/\pi} = 2.1688\dots$.

The numerical value of the fit parameter varies with the choice of the data points used in the least square fit, however. As a more systematic way to estimate the asymptotic behavior we applied least squares fitting of the two-parameter function $a n^{-\delta}$ to sliding windows of w consecutive data points p_n. Figure 6 shows the results for $w = 10$. Larger values of w yield similar curves with smaller errorbars but fewer data points. This analysis shows that the available numerical data for p_n supports the conjecture (3) within the errorbars.
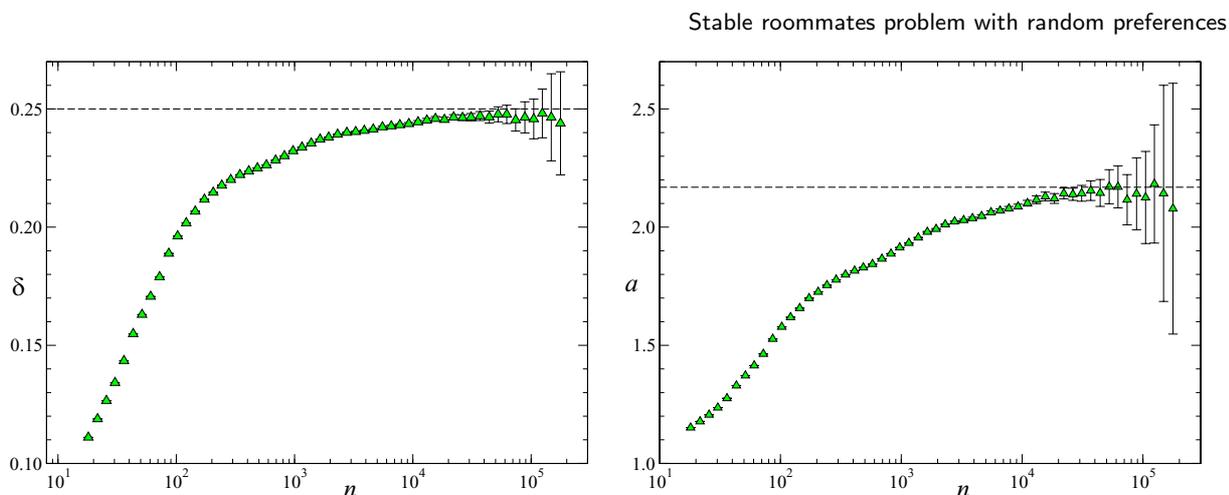
Figure 6. Results of a least-squares fit of $p_n = an^{-\delta}$ to a sliding window of $w = 10$ consecutive data points. The abscissa represents the geometric mean of the n-values contained in a window.

A more elementary question is whether $\lim_{n\to\infty} p_n$ is zero or non-zero. To address this question we applied the sliding window technique to fit the three parameter function $p_n = an^{-\delta} + b$. The result for b is a curve very similar to the curves shown in figure 6 with b converging to zero within the errorbars. This result supports the claim $\lim_{n\to\infty} p_n = 0$.

5. Conclusions and outlook

We have demonstrated that Irving's algorithm for the stable roommates problem can be organized such that the expected time and space complexity is $O(n^{3/2})$ on random instances. Our reasoning about the dynamics of the algorithm (approaching random walks of leftrank and rightrank, phase I as coupon collector's problem) is of course non-rigorous, but the results are well confirmed by the numerical simulations. Maybe this simplistic view on Irving's algorithm can help to derive the observed $n^{-1/4}$ decay of the probability p_n that a random instance of size n has a solution.

References

[1] Manlove D F 2013 *Algorithmics of Matching under Preferences* (*Series on Theoretical Computer Science* vol 2) (Singapore: World Scientific)
[2] Gusfield D and Irving R W 1989 *The Stable Marriage Problem: Structure, Algorithms* (Cambridge, MA: MIT Press)
[3] Gale D and Shapley L S 1962 College admissions and the stability of marriage *Am. Math. Mon.* **69** 9–15

[4] Roth A E and Oliveira Sotomayor M A 1990 *Two-Sided Matching: a Study in Game-Theoretic Modeling and Analysis* (*Economic Society Monographs*) (Cambridge: Cambridge University Press)

[5] Oméro M-J, Dzierzawa M, Marsili M and Zhang Y-C 1997 Scaling behavior in the stable marriage problem *J. Physique I* **7** 1723–32

[6] Nieuwenhuizen T M 1998 The marriage problem and the fate of bachelors *Physica* A **252** 178–98

[7] Dzierzawa M and Oméro M-J 2000 Statistics of stable marriages *Physica* A **287** 321–33

[8] Caldarelli G and Capocci A 2001 Beauty, distance in the stable marriage problem *Physica* A **300** 325–31

[9] Lage-Castellanos A and Mulet R 2006 The marriage problem: from the bar of appointments to the agency *Physica* A **364** 389–402

[10] Moore C and Mertens S 2011 *The Nature of Computation* (Oxford: Oxford University Press) (www.nature-of-computation.org)

[11] Irving R W 1985 An efficient algorithm for the stable roommates problem *J. Algorithms* **6** 577–95

[12] Mertens S 2005 Random stable matchings *J. Stat. Mech.* P10008

[13] Pittel B 1993 The 'stable roommates' problem with random preferences *Ann. Probab.* **21** 1441–77

[14] Mertens S 2015 Small random instances of the stable roommates problem, in preparation

[15] Pittel B and Irving R W 1994 An upper bound for the solvability of a random stable roommates instance *Random Struct. Algorithms* **5** 465–87