JOURNAL OF ALGORITHMS 6, 577–595 (1985)

An Efficient Algorithm for the "Stable Roommates" Problem

ROBERT W. IRVING*

Department of Mathematics, University of Salford, Salford M5 4WT, United Kingdom

Received January 31, 1984; accepted May 1, 1984

The stable marriage problem is that of matching n men and n women, each of whom has ranked the members of the opposite sex in order of preference, so that no unmatched couple both prefer each other to their partners under the matching. At least one stable matching exists for every stable marriage instance, and efficient algorithms for finding such a matching are well known. The stable roommates problem involves a single set of even cardinality n, each member of which ranks all the others in order of preference. A stable matching is now a partition of this single set into $n/2$ pairs so that no two unmatched members both prefer each other to their partners under the matching. In this case, there are problem instances for which no stable matching exists. However, the present paper describes an $O(n^2)$ algorithm that will determine, for any instance of the problem, whether a stable matching exists, and if so, will find such a matching. © 1985 Academic Press, Inc.

1. INTRODUCTION AND HISTORY

The Stable Marriage Problem

The stable marriage assignment problem was introduced by Gale and Shapley [1] in the context of assigning applicants to colleges, taking into account the preferences of both the applicants and the colleges.

In its most familiar form, a problem instance involves two disjoint sets of cardinality n, the men and the women, with each individual having ranked the n members of the opposite sex in order of preference. A stable matching is defined as a one-to-one correspondence between the men and women with the property that there is no couple both of whom prefer each other to their actual partners.

Gale and Shapley demonstrated that at least one stable matching exists for every problem instance, and described an algorithm that would yield one such solution. McVitie and Wilson [4] proposed an alternative recursive

*Present address: Department of Computer Science, University of Glasgow, Glasgow G12 8QQ, United Kingdom.

0196-6774/85 $3.00
Copyright © 1985 by Academic Press, Inc.
All rights of reproduction in any form reserved.

algorithm for generating a stable matching, and extended this approach to an algorithm yielding all stable matchings for a given problem instance. Implementations of these algorithms in ALGOL60 were provided [5]. Both the Gale/Shapley and the McVitie/Wilson algorithms have been subjected to a variety of analyses (see, e.g., [2, 6, 3]). For each, the worst-case time-complexity is $O(n^2)$, but it is also known that, on average, the "core" of each algorithm (as opposed to setting up the data structures) involves a number of operations that is $O(n \log n)$.

The problem became more widely known as a result of two later works. In [8], Wirth developed an alternative algorithm for generating all stable matchings, as an illustration of the technique of backtracking. Though more transparent than McVitie and Wilson's approach, tests have revealed that Wirth's algorithm is dramatically less efficient. Then, in an eminently readable account [3], Knuth investigated a wide range of issues thrown up by the stable marriage problem, including connections with a number of other combinatorial problems, detailed analysis of the average behavior of McVitie and Wilson's algorithm, and a number of extensions of this "fundamental" algorithm. Finally, Knuth presented a list of twelve research problems related to stable marriage.

The Roommates Problem

One of the research problems mentioned by Knuth, the so-called roommates problem, had originally been described in the paper of Gale and Shapley [1], and has also been raised by Wilson [7]. The roommates problem is essentially a version of the stable marriage problem involving just one set. Each person in the set, of even cardinality n, ranks the $n - 1$ others in order of preference. The object is to find a stable matching, which is a partition of the set into $n/2$ pairs of roommates such that no two persons who are not roommates both prefer each other to their actual partners.

Gale and Shapley had already demonstrated that, contrary to the case of the stable marriage problem, roommates instances exist for which no stable matching is possible. They gave the instance of size 4 described by the preference lists below, in which anyone paired with person 4 will cause instability.

Person	Preference list
1	2 3 4
2	3 1 4
3	1 2 4
4	arbitrary

Knuth [3] demonstrated that multiple solutions could exist, giving the

instance of size 8 below, for which there are exactly 3 stable matchings, namely $\{1/2, 3/4, 5/8, 6/7\}$, $\{1/4, 2/3, 5/6, 7/8\}$, and $\{1/5, 2/6, 3/7, 4/8\}$.

1	2	5	4	6	7	8	3
2	3	6	1	7	8	5	4
3	4	7	2	8	5	6	1
4	1	8	3	5	6	7	2
5	6	1	8	2	3	4	7
6	7	2	5	3	4	1	8
7	8	3	6	4	1	2	5
8	5	4	7	1	2	3	6

Knuth asked for an efficient (polynomial-time in the worst case) algorithm to generate a solution, if one exists, to any given instance of the roommates problem, and suggested that it may be possible to prove the problem NP-complete.

It is the primary purpose of this note to present such an algorithm, to prove its effectiveness, and to demonstrate its polynomial-time worst-case behaviour. In addition, an implementation of the algorithm (in Pascal) is given, and the results of some computational experience are presented.

2. The Roommates Algorithm

First Phase

The algorithm breaks down naturally into two phases, the first of which is not unlike the McVitie/Wilson algorithm for the stable marriage problem. The first phase of the algorithm is based on a sequence of "proposals" made by one person to another. This sequence of proposals proceeds with each individual pursuing the following strategies:

(i) If x receives a proposal from y, then

(a) he rejects it at once if he already holds a better proposal (i.e., a proposal from someone higher than y in his preference list);

(b) he holds it for consideration otherwise, simultaneously rejecting any poorer proposal that he currently holds.

(ii) An individual x proposes to the others in the order in which they appear in his preference list, stopping when a promise of consideration is received; any subsequent rejection causes x to continue immediately his sequence of proposals.

The individuals begin their proposal sequences one at a time, and each person's first proposal will, in general, lead to a chain of proposals and rejections that terminates only when some person receives his first proposal.

EXAMPLE. Consider the problem instance of size 6 specified by the preference matrix below. (The rows of the preference matrix are the preference lists.)

1	4 6 2 5 3
2	6 3 5 1 4
3	4 5 1 6 2
4	2 6 5 1 3
5	4 2 3 6 1
6	5 1 4 2 3

According to the above rules, the following sequence of proposals and rejections takes place:

1 proposes to 4;	4 holds 1;
2 proposes to 6;	6 holds 2;
3 proposes to 4;	4 rejects 3;
3 proposes to 5;	5 holds 3;
4 proposes to 2;	2 holds 4;
5 proposes to 4;	4 holds 5 and rejects 1;
1 proposes to 6;	6 holds 1 and rejects 2;
2 proposes to 3;	3 holds 2;
6 proposes to 5;	5 rejects 6;
6 proposes to 1;	1 holds 6.

This phase of the algorithm will terminate either

(i) with every person holding a proposal (as in the example above), or

(ii) with one person rejected by everyone (as would happen, for example, in the problem instance of size 4 mentioned in Sect. 1).

In case (ii) it is not hard to see that everyone but the rejected person holds a proposal, because they have all rejected him. Hence everyone else has also made a proposal. The general outline of this phase of the algorithm is as follows:

```
set_proposed_to := [ ];
for person := 1 to n do
```

```
begin
    proposer := person;
    repeat
        proposer proposes to his next_choice;
        if not rejected
        then if next_choice in set_proposed_to
                then proposer := next_choice's reject
    until not (next_choice in set_proposed_to);
    set_proposed_to := set_proposed_to + [next_choice]
end
```

Note that possibility (ii) above can be detected by setting a sentinel nth column in the preference matrix so that each person's nth choice is himself. Exit from the for loop with next_choice equal to proposer is then the signal that this individual has been rejected by all of the others.

A consequence of the above proposal strategies is the following crucial lemma.

LEMMA 1. *If y rejects x in the proposal sequence described above, then x and y cannot be partners in a stable matching.*

Proof. Suppose that, of all the rejections that involve two individuals who are partners in some stable matching, the rejection of x by y is, chronologically, the first. Denote by M a stable matching in which x and y are partners.

Now, y rejected x because he either already held, or received later, a better proposal, say from z. But if y prefers z to x, then, for the stability of M, z must prefer his own partner in M, say w, to y. Now, before z could propose to y, he must have been rejected by w, and this rejection must have preceded the rejection of x by y, contradicting our initial assumption.

From this lemma we deduce a number of useful corollaries.

COROLLARY 1.1. *If, at any stage of the proposal process, x proposes to y, then, in a stable matching*

(i) *x cannot have a better partner than y;*

(ii) *y cannot have a worse partner than x.*

Proof. If x proposes to y then he has been rejected by everyone better than y, and (i) follows from the lemma.

If y partners z in a stable matching, and y prefers x to z, then since by (i), x prefers y to his own partner, stability is violated. Hence (ii) is established.

COROLLARY 1.2. *If the first phase of the algorithm terminates with one person having been rejected by all of the others, then no stable matching exists.*

Proof. By the lemma, the rejected person has no possible partner.

COROLLARY 1.3. *If the first phase of the algorithm terminates with every person holding a proposal, then the preference list of possible partners for y, who holds a proposal from x, can be "reduced" by deleting from it*

(i) *all those to whom y prefers x;*

(ii) *all those who hold a proposal from a person whom they prefer to y, (including all those who have rejected y);*

In the resulting reduced preference lists,

(iii) *y is first on x's list and x last on y's;*

(iv) *in general, b appears on a's list if and only if a appears on b's.*

Proof. (i) and (ii) follow directly from Corollary 1.1 (ii); (iii) follows from (i) and the parenthesised part of (ii); and (iv) is self-evident.

EXAMPLE. In our example problem instance of size 6, we reproduce the original preference matrix. The symbol * or † against an entry indicates that the entry should not appear in the reduced lists by virtue of Corollary 3 part (i) or (ii), respectively.

1	4†	6	2*	5*	3*
2	6†	3	5	1†	4
3	4†	5	1†	6†	2
4	2	6†	5	1*	3*
5	4	2	3	6*	1*
6	5†	1	4*	2*	3*

Hence, in this case, the reduced preference lists may be displayed as follows:

1	6		
2	3	5	4
3	5	2	
4	2	5	
5	4	2	3
6	1		

LEMMA 2. *If, in the reduced preference lists, every list contains just one person, then the lists specify a stable matching.*

Proof. That the lists specify a matching is a consequence of Corollary 1.3 (iv).

Suppose that x prefers y to the sole person on his list. Then x was rejected by y, presumably because y obtained a better proposal. But the

final proposal held by y is from the sole person on y's reduced list and so y clearly prefers this person to x. Hence there can be no instability in the matching specified by the reduced preference lists.

Second Phase

It remains to describe how we deal with the reduced preference lists when, as in our illustrative example, some of the lists contain more than one person. This brings us to the second phase of the algorithm.

The second phase involves further reduction of the preference lists, and this phase 2 reduction may have to be repeated several times until one of the same two terminating conditions is recognised—either one person runs out of people to propose to, in which case no stable matching exists, or all the preference lists shrink to a single person, in which case they specify a stable matching.

We shall formally describe the set of preference lists for a problem instance as *reduced* if

(i) they have been subjected to a phase 1 reduction, as described in Corollary 1.3, and

(ii) they have been subjected to zero or more phase 2 reductions, as described below.

The key to a phase 2 reduction is the recognition of a cyclic sequence $a_1, \ldots, a_r$ of distinct persons such that

(i) for $i = 1, \ldots, r-1$, the second person in a_i's current reduced preference list is the first person in a_{i+1}'s; we shall denote this person by b_{i+1};

(ii) the second person in a_r's current reduced preference list is the first in a_1's; we shall denote this person by b_1.

For want of a better term, and for a reason that will emerge in Lemma 3 below, we call such a sequence $a_1, \ldots, a_r$ an *all-or-nothing cycle* relative to the current reduced preference lists.

It is very easy to find an all-or-nothing cycle. Let p_1 be an arbitrary individual whose current reduced preference list contains more than one person. Generate the sequences

q_i = second person in p_i's current reduced list

p_{i+1} = last person in q_i's current reduced list (so that, as we shall see, q_i is first in p_{i+1}'s)

until the p sequence cycles, as it eventually must, and let

$$a_i = p_{s+i-1} \qquad (i = 1, 2, \ldots),$$

where p_s is the first element in the p sequence to be repeated. We refer to $p_1, p_2, \ldots, p_{s-1}$ as the "tail" of the cycle.

EXAMPLE. In our illustrative instance of size 6, we may take $p_1 = 2$, which gives

$$\begin{array}{ll} q_1 = 5 & p_2 = 3 \\ q_2 = 2 & p_3 = 4 \\ q_3 = 5 & p_4 = 3 \end{array}$$

so that $s = 2$, and we obtain an all-or-nothing cycle of length 2 with $a_1 = 3$, $a_2 = 4$, and a tail of length 1.

A phase 2 reduction, applied to a given set of reduced lists, and for a particular all-or-nothing cycle $a_1, \ldots, a_r$, involves forcing each b_i $(1 \le i \le r)$ to reject the proposal that he holds from a_i, thereby forcing each a_i to propose to b_{i+1} (modulo r), the second person in his current reduced list.

As a result, just as in Corollary 1.3, all successors of a_i in b_{i+1}'s reduced list can be deleted, and b_{i+1} can be deleted from their lists. For if a_i achieves no better partner than b_{i+1}, then, in the interests of stability, b_{i+1} can settle for no worse partner than a_i. It follows that parts (iii) and (iv) of Corollary 1.3 apply also to these more general reduced preference lists.

The essential significance of a reduced set of preference lists is the following: if the original problem instance admits a stable matching, then there is a stable matching in which every person is partnered by someone on his reduced list. We say that such a matching is *contained* in the reduced lists. This crucial result is a consequence of our next lemma.

LEMMA 3. *Let $a_1, \ldots, a_r$ be an all-or-nothing cycle relative to a set of reduced preference lists, and denote by b_i the first person in a_i's reduced list $(1 \le i \le r)$. Then*

(i) *in any stable matching contained in these reduced lists, either a_i and b_i are partners for all values of i or for no value of i;*

(ii) *if there is such a stable matching in which a_i and b_i are partners, then there is another in which they are not.*

Proof. (i) Considering subscripts modulo r, suppose that, for some fixed i, a_i and b_i are partners in a particular stable matching that is contained in the reduced lists. Since a_i is last on b_i's reduced list, and b_i is second on a_{i-1}'s with the consequence that a_{i-1} is at least present in b_i's reduced list, it follows that b_i prefers a_{i-1} to a_i. So, for stability, a_{i-1} must be partnered by someone he prefers to b_i, and the only such person in his reduced list is b_{i-1}. Repeating this argument shows that a_i and b_i must be partners for all values of i.

(ii) Let $A = \{a_1, \ldots, a_r\}$, $B = \{b_1, \ldots, b_r\}$. If $A \cap B$ is non-empty, say $a_j = b_k$, then it is impossible for all the a_i to have their first remaining preference, since b_k has his last when a_k has his first. Hence, by (i), $A \cap B \neq \varnothing$ implies that none of the a_i and b_i can be partners, so we may as well assume $A \cap B = \varnothing$.

Suppose that M is a stable matching, contained in the reduced lists, in which a_i and b_i are partners for all i $(1 \leq i \leq r)$. Denote by M' the matching in which each a_i is partnered by b_{i+1}, and any person not in $A \cup B$ has the same partner as in M. We claim that M' is stable.

Clearly, each member of B obtains a better partner in M', from his point of view, than the one he had in M. The only individuals who fare worse in M' than in M are the members of A, so that any instability in M', not present in M, must involve some a_i.

If a_i prefers x, say, to b_{i+1} (his partner in M'), then there are just three cases to consider:

(a) a_i and x were partners in M (i.e., $x = b_i$); but in this case, x undoubtedly prefers his new partner a_{i-1} to a_i.

(b) a_i also prefers x to b_i, in which case x is not in a_i's reduced preference list, where he would precede a_i's known first remaining choice b_i. So x has willingly rejected a_i, or has been forced to reject a_i. In the first case, x must have received a proposal from someone better than a_i, while in the second case, the forced rejection must have led to the receipt by x of a better proposal. In either case, x must therefore prefer even his last surviving choice, and so certainly his partner in M', to a_i.

(c) a_i prefers b_i to x, in which case x lies between b_i and b_{i+1} in a_i's original preference list, but again is absent from a_i's current reduced list. This absence must be the result of x obtaining a proposal from someone better than a_i, so that, just as in case (b), x must prefer his partner in M' to a_i.

Lemma 3 has the following two immediate corollaries.

COROLLARY 3.1. *If the original problem instance admits a stable matching, then there is a stable matching contained in any reduced set of preference lists.*

COROLLARY 3.2. *If one or more among a reduced set of preference lists is empty, then the original problem instance admits no stable matching.*

Our final lemma, an extension of Lemma 2, justifies the circumstances under which the algorithm leads to a positive conclusion.

LEMMA 4. *If in a reduced set of preference lists, every list contains just one person, then the lists specify a stable matching.*

Proof. That the lists specify a matching is a consequence of the extension of Corollary 1.3 (iv).

Suppose that y prefers x to the sole person on his reduced list. Then, exactly as in the proof of case (b) of Lemma 3(ii), we can demonstrate that x must prefer the sole person on his reduced list to y. Hence there can be no instability in the specified matching.

EXAMPLE. Pursuing our illustrative instance of size 6, we force 5 to reject 3, and 2 to reject 4, causing 3 to propose to 2, and 4 to propose to 5. As a result, all the preference lists shrink to a single person, and we obtain the stable matching $1/6, 2/3, 4/5$.

Of course, in general, several phase 2 reductions may have to be carried through before a definite conclusion is reached. So, in summary, the second phase of the algorithm can be expressed as follows:

```
while (some reduced preference list has length > 1)
   and (no reduced preference list has length < 1) do
begin
    locate an all-or-nothing cycle;
    carry out a phase 2 reduction
end;
```

Exit from the loop with the second condition false indicates, in view of Corollary 3.2, that no stable matching exists. Otherwise, according to Lemma 4, the final reduced preference lists specify a stable matching.

We observe in passing that the proof of Lemma 3(i) also reveals that, if the a_i and b_i are paired, then the "tail" items $p_0, p_1, \ldots, p_{s-1}$, if any, encountered in locating the all-or-nothing cycle, must also be partnered by their first surviving choices, and in some cases, this may lead to a contradiction. However, in its present form, the algorithm does not exploit this additional information.

EXAMPLE. We conclude this section with an illustrative example, again of size 6, in which we are led to conclude that no solution exists. The original preference lists are

1	2 6 4 3 5
2	3 5 1 6 4
3	1 6 2 5 4
4	5 2 3 6 1
5	6 1 3 4 2
6	4 2 5 1 3

The first phase of the algorithm leads to the reduced lists

1	2 3
2	3 1
3	1 2
4	5 6
5	6 4
6	4 5

The first cycle detected in the second phase of the algorithm involves persons 1, 2, and 3 and forcing each of them to be rejected by his first surviving choice causes the preference list of each of them to become empty.

3. Implementation of the Algorithm

The Appendix contains an implementation of the algorithm in the form of a Pascal procedure. A problem instance of size n is specified by a preference matrix whose rows constitute the preference lists of the n individuals. Each person appears in position n in his own list to act as a sentinel during the reduction process. The procedure returns a boolean value indicating whether a solution exists, and if so, a vector giving each person's partner in a stable matching.

In order to improve efficiency, a ranking array is set up to permit rapid comparisons between alternative choices for an individual. More precisely, the entry in row i and column j specifies the position of person j in the preference list of person i. In other words,

$$\text{preference}[i, k] = j \Leftrightarrow \text{ranking}[i, j] = k.$$

The reduction of the preference lists is recorded by the values in two vectors, called leftmost and rightmost, that contain, for each person, the positions in his original preference list occupied by his current best and worst potential partners. There turns out to be no need to delete explicitly persons on x's list between his leftmost and rightmost markers, even if such a person holds a proposal from someone better than x, and is therefore unobtainable as a partner for x. This impossibility will manifest itself in due course as a result of a comparison of rankings.

The reduction process, both in phase 1 and in phase 2, causes corresponding elements of the leftmost and rightmost vectors to move towards each other; when leftmost[i] and rightmost[i] coincide, then a partner is specified for person i, while leftmost[i] > rightmost[i] indicates that i has run out of potential partners, and therefore that no stable matching is possible.

A further vector "second" is used to facilitate the isolation of an all-or-nothing cycle in phase 2 of the algorithm. This marks the positions in an individual's original preference list of the person who is second in his current reduced list, but is updated only when needed in order to locate a cycle.

The array "cycle" is itself used to hold the sequence $p_1, \ldots, p_{s+r-1}$ described prior to Lemma 3 above, with markers first_in_cycle and last_in_cycle to delineate the actual extent within the array of the sequence $a_1, \ldots, a_r$.

If more than one phase 2 reduction is necessary, then, in the interests of efficiency, we wish to avoid, in searching for all-or-nothing cycles, a succession of long "tails" $p_1, \ldots, p_{s-1}$ which do not contribute to the reduction of the preference lists. This can be handled successfully by "remembering" the tail, if any, from the previous phase 2 reduction, and starting the next cycle search from the element p_{s-1}.

In order to keep parameter lists short and improve readability, the array variables are used globally in the various procedures nested within procedure roommates.

For further details of the implementation, see the annotated listing in the Appendix.

4. Analysis

The worst-case analysis of our implementation of the algorithm can be expressed largely in terms of "eliminations" from the preference lists. An elimination takes place either when an element of the leftmost vector is incremented or when an element of the rightmost vector is decremented. The total number of eliminations for any problem instance of size n cannot exceed n^2.

Procedure Phase_1_reduce

Within the main loop of this procedure, the total number of operations carried out is proportional to the number of changes made to elements of leftmost plus the number of changes made to elements of rightmost. This is bounded by a constant times the number of eliminations from the preference lists during this phase of the algorithm.

Procedure Find

The total number of incrementations of the variable first_unmatched is at most n.

Procedure Seek_cycle

Suppose that seek_cycle is called m times, and that the ith call yields a tail of length t_i and a cycle of length r_i. The number of operations during the ith call, excluding those in the inner repeat loop when an element of array second changes in value, is

$$\alpha_i + \beta(r_i + t_i - t_{i-1} + 1) + (\gamma r_i + \delta)$$

where $\alpha_i, \beta, \gamma, \delta$ are constants and $t_0 = 0$. Here, the three terms account for operations before, during and after the main repeat loop, respectively.

Now, the total number of operations involved in changing second choices, in all calls of the procedure, is $O(n^2)$, since there are n elements in the array and none can be changed as many as n times. Hence, the total number of operations in all calls of procedure seek_cycle is

$$O(n^2) + \sum_{i=1}^{m} \{\alpha_i + \beta(r_i + t_i - t_{i-1} + 1) + (\gamma r_i + \delta)\}.$$

But, the ith call results in at least $2r_i$ eliminations from the preference lists, and $r_i \geq 2$ for all i, so that

$$4m \leq 2\sum_{i=1}^{m} r_i \leq n^2.$$

Therefore, the total number of operations is bounded by

$$\begin{aligned} &O(n^2) + \sum_{i=1}^{m} \alpha_i + (\beta + \gamma)\sum_{i=1}^{m} r_i + \beta t_m + (\beta + \delta)m \\ &\leq O(n^2) + \lambda m + \mu n^2 + \beta n \\ &\qquad\qquad \text{where } \lambda = \beta + \delta + \max_i \alpha_i,\ \mu = \tfrac{1}{2}(\beta + \gamma), \\ &\leq O(n^2) + \frac{\lambda}{4} n^2 + \mu n^2 + \beta n \\ &= O(n^2). \end{aligned}$$

Procedure Phase_2_reduce

If, as above, the ith call involves a cycle of length r_i, then the number of operations is bounded by a constant times r_i. Hence the total number of operations in all calls of the procedure is

$$O\left(\sum_{i=1}^{m} r_i\right) = O(n^2).$$

Procedure Roommates

The various initialisations occupy $O(n^2)$ steps, as does the call of procedure phase_1_reduce.

Within the while loop, the total number of operations carried out in procedure calls is $O(n^2)$, as justified above for each procedure, so that the loop itself involves $O(n^2)$ steps. Hence the roommates algorithm, as implemented, has a worst-case time complexity that is $O(n^2)$.

Of course, the above analysis is of theoretical interest, but from a practical point of view, the memory requirements for the arrays, rather than

TABLE 1
Times in Seconds on PDP11/44 Running under UNIX in the Department of Mathematics at the University of Salford

Problem size	No. of instances	No. with solution	Proportion with solution	Average cpu time
4	1000	961	0.961	0.053
6	1000	931	0.931	0.077
8	1000	909	0.909	0.108
10	1000	868	0.868	0.142
12	1000	871	0.871	0.179
14	1000	867	0.867	0.222
16	1000	857	0.857	0.274
18	1000	830	0.830	0.323
20	1000	815	0.815	0.374
22	1000	808	0.808	0.441
24	1000	816	0.816	0.505
26	500	394	0.788	0.575
28	500	375	0.750	0.645
30	500	383	0.766	0.727
32	200	151	0.755	0.812
34	200	154	0.770	0.921
36	200	148	0.740	0.984
38	200	145	0.725	1.078
40	200	149	0.745	1.183
42	200	155	0.775	1.279
44	200	148	0.740	1.383
46	200	148	0.740	1.499
48	200	146	0.730	1.616
50	200	142	0.710	1.760
60	200	145	0.725	2.376
70	200	134	0.670	3.110
80	200	135	0.675	3.964
90	200	138	0.690	4.875

the processing time of the procedure, are likely to be the limiting factor on the size of roommates instances to which the algorithm can be successfully applied. (see the figures in Sect. 5).

5. Computational Experience

The algorithm implementation listed in the Appendix has been run, using as data a number of randomly generated problem instances of various sizes up to 90, beyond which memory demands became too great for the machine in use. The results of these experiments, in terms of the proportion of problem instances for which a stable matching exists, and the average cpu time per problem instance, are presented in Table 1.

The proportion of problem instances of a given size n for which a stable matching exists is clearly a matter of some interest. The computational evidence suggests that this proportion decreases as n increases, but it is not clear whether this proportion tends to a positive limit as n grows large. Any theoretical results that could be obtained in this respect would be of considerable interest.

Appendix: Pascal Implementation of the Algorithm

```
const SIZE = 91;        {FOR PROBLEM INSTANCES OF SIZE <= 90,
                                        ALLOWING FOR SENTINELS}
type person_type = 0..SIZE; rank_type = 0..SIZE;
     matrix = array[person_type,rank_type] of person_type;
     vector = array[person_type] of person_type;
     set_type = set of person_type;

procedure room_mates(var preference : matrix; n : integer;
                              var partner : vector; var soln_found : boolean);
var ranking : array[person_type,person_type] of rank_type;
     leftmost,second,rightmost : array[person_type] of rank_type;
     cycle : array[rank_type] of person_type;
     person,first_unmatched : person_type;
     rank,first_in_cycle,last_in_cycle : rank_type;
     soln_possible : boolean;
     tail : set_type;

procedure phase_1_reduce(var soln_possible : boolean);
var set_proposed_to : set_type;
     person,proposer,next_choice,current : person_type;
```

```
begin
  set_proposed_to := [];
  for person := 1 to n do
  begin
     proposer := person;
     repeat
       next_choice := preference[proposer,leftmost[proposer]];
                                        {BEST POTENTIAL PARTNER}
       current := preference[next_choice,rightmost[next_choice]];
                                     {NEXT_CHOICE HOLDS CURRENT}
       while  ranking[next_choice,proposer] > ranking[next_choice,current]
       do
       begin {PROPOSER IS REJECTED BY NEXT_CHOICE}
         leftmost[proposer] := leftmost[proposer] + 1;
         next_choice := preference[proposer,leftmost[proposer]];
         current := preference[next_choice,rightmost[next_choice]]
       end;
       rightmost[next_choice] := ranking[next_choice,proposer];
                                     {NEXT_CHOICE HOLDS PROPOSER}
       proposer := current
                                        {AND REJECTS CURRENT}
     until not (next_choice in set_proposed_to);
     set_proposed_to := set_proposed_to + [next_choice]
  end;
  soln_possible := proposer = next_choice
end; {phase_1_reduce}

procedure find(var first_unmatched : person_type);
begin {FINDS FIRST PERSON WITH > 1 POTENTIAL PARTNER}
  while leftmost[first_unmatched] = rightmost[first_unmatched] do
     first_unmatched := first_unmatched + 1
end; {find}


procedure seek_cycle(var first_in_cycle,last_in_cycle : rank_type;
                     first_unmatched : person_type; var tail : set_type);
var cycle_set : set_type;
    person,next_choice : person_type;
    posn_in_cycle,pos_in_list : rank_type;
begin
  if first_in_cycle > 1
  then begin
         person := cycle[first_in_cycle-1];          {LAST PERSON IN
                                                      PREVIOUS TAIL}
```

```
        posn_in_cycle := first_in_cycle-1;{HIS SECOND CHOICE MAY
                                            HAVE TO BE UPDATED}
        cycle_set := tail
      end
  else begin
        cycle_set := [];
        posn_in_cycle := 1;
        person := first_unmatched
      end;
  repeat {GENERATE SEQUENCE}
     cycle_set := cycle_set + [person];
     cycle[posn_in_cycle] := person;
     posn_in_cycle := posn_in_cycle + 1;
     pos_in_list := second[person];
     repeat {UPDATE SECOND CHOICE FOR CURRENT PERSON}
       next_choice := preference[person,pos_in_list];
       pos_in_list := pos_in_list + 1
     until ranking[next_choice,person] < = rightmost[next_choice];
     second[person] := pos_in_list - 1;
     person := preference[next_choice,rightmost[next_choice]]
  until person in cycle_set; {SEQUENCE STARTS TO CYCLE}
  last_in_cycle := posn_in_cycle - 1;
  tail := cycle_set;
  repeat {WORK BACK TO BEGINNING OF CYCLE}
     posn_in_cycle := posn_in_cycle - 1;
     tail := tail - [cycle[posn_in_cycle]]
  until cycle[posn_in_cycle] = person;
  first_in_cycle := posn_in_cycle
end; {seek_cycle}

procedure phase_2_reduce(first_in_cycle,last_in_cycle : rank_type;
                                          var soln_possible : boolean);
var proposer,next_choice : person_type;
     rank : rank_type;
begin
  for rank := first_in_cycle to last_in_cycle do
  begin {ALLOW NEXT PERSON IN CYCLE TO BE REJECTED}
     proposer := cycle[rank];
     leftmost[proposer] := second[proposer];
     second[proposer] := leftmost[proposer] + 1;      {PROPER UPDATE
                                          UNNECESSARY AT THIS STAGE}
     next_choice := preference[proposer,leftmost[proposer]];
```

```
      rightmost[next_choice] := ranking[next_choice,proposer]
                                        {NEXT_CHOICE HOLDS PROPOSER}
   end;
   rank := first_in_cycle;
   while (rank < = last_in_cycle) and soln_possible do
   begin                                {CHECK NO-ONE HAS RUN OUT
                                              OF POTENTIAL PARTNERS}
      proposer := cycle[rank];
      soln_possible := leftmost[proposer] < = rightmost[proposer];
      rank := rank + 1
   end
end; {phase_2_reduce}

begin
   soln_found := false;
   first_unmatched := 1;
   first_in_cycle := 1;
   for person := 1 to n do
   begin
      preference[person,n] := person; {SENTINEL}
      for rank := 1 to n do
         ranking[person,preference[person,rank]] := rank;
      leftmost[person] := 1;
      rightmost[person] := n
   end;
   leftmost[n + 1] := 1; rightmost[n + 1] := n;        {SENTINELS FOR
                                                        PROCEDURE FIND}
   phase_1_reduce(soln_possible);
   for person := 1 to n do
      second[person] := leftmost[person] + 1;      {PROPER INITIALISA-
                                                    TION UNNECESSARY}
   while soln_possible and not soln_found do
   begin
      find(first_unmatched);
      if first_unmatched > n
      then soln_found := true
      else begin
            seek_cycle(first_in_cycle,last_in_cycle,first_unmatched,tail);
            phase_2_reduce(first_in_cycle,last_in_cycle,soln_possible)
         end
   end;
   if soln_found
```

```
  then for person := 1 to n do
        partner[person] := preference[person,leftmost[person]]
end; {room_mates}
```

ACKNOWLEDGMENT

The author is grateful to Professor D. E. Knuth for a number of helpful comments, particularly that which led to a more rigorous presentation of phase two of the algorithm, and that which improved the worst-case performance of procedure seek-cycle, and thereby of the whole algorithm, by an order of magnitude.

REFERENCES

1. D. GALE AND L. S. SHAPLEY, College admissions and the stability of marriage, *Amer. Math. Monthly* **69** (1962), 9–15.
2. S. Y. ITOGA, The upper bound for the stable marriage problem, *J. Oper. Res. Soc.* **29** (1978), 811–814.
3. D. E. KNUTH, "Mariages Stables," Presses Univ. de Montréal, Montreal, 1976.
4. D.G. MCVITIE AND L. B. WILSON, The stable marriage problem, *Comm. ACM* **14** (1971), 486–490.
5. D. G. MCVITIE AND L. B. WILSON, Three procedures for the stable marriage problem, ACM Algorithm 411, *Comm. ACM* **14** (1971), 491–492.
6. L. B. WILSON, An analysis of the stable marriage assignment problem, *BIT* **12** (1972), 569–575.
7. L. B. WILSON, "A Survey of the Stable Marriage Assignment Problem," Department of Computing Science Technical Report 5, Univ. of Stirling, 1981.
8. N. WIRTH, "Algorithms + Data Structures = Programs," Prentice–Hall, Englewood Cliffs, N.J., 1976.