Stable Roommates and Constraint Programming

Patrick Prosser

School of Computing Science, University of Glasgow, Glasgow, Scotland
pat@dcs.gla.ac.uk

Abstract. In the stable roommates (SR) problem we have n agents, where each agent ranks all $n-1$ other agents. The problem is then to match agents into pairs such that no two agents prefer each other to their matched partners. A remarkably simple constraint encoding is presented that uses $O(n^2)$ binary constraints, and in which arc-consistency (the phase-1 table) is established in $O(n^3)$ time. This leads us to a specialized n-ary constraint that uses $O(n)$ additional space and establishes arc-consistency in $O(n^2)$ time. This can model stable roommates with incomplete lists (SRI), consequently it can also model stable marriage (SM) problems with complete and incomplete lists (SMI). That is, one model suffices. An empirical study is performed and it is observed that the n-ary constraint model can read in, model and output all matchings for instances with $n = 1,000$ in about 2 seconds on current hardware platforms. Enumerating all matchings is a crude solution to the egalitarian SR problem, and the empirical results suggest that although NP-hard, egalitarian SR is practically easy.

1 Introduction

In the **S**table **R**oommates problem (SR) [8,7] we have an even number of agents to be matched together as couples, where each agent strictly ranks all other agents. The problem is then to match pairs of agents together such that the matching is stable, i.e. there doesn't exist a pair of agents in the matching such that $agent_i$ prefers $agent_j$ to his matched partner and $agent_j$ prefers $agent_i$ to his matched partner[1].

The **S**table **M**arriage problem (SM) [4,15,5,7,16,11] is a specialized instance of stable roommates where agents have gender, such that we have two sets of agents m (men) and w (women). Each man has to be *married* to a woman and each woman to a man such that in the matching there does not exist a man m_i and a woman w_j where m_i prefers w_j to his matched partner and w_j prefers m_i to her matched partner i.e. there is no incentive for agents to divorce and elope.

Constraint programming has been applied to the stable marriage problem, probably the first efficient model being reported in 2001 [6], a 4-valued model in [12], a specialized binary constraint in [18] and an efficient n-ary constraint in [17]. This raises an obvious question: if there is an efficient constraint model for stable marriage, is there one for the more general stable roommates problem?

[1] For sake of brevity I assume agents are male, and hope this offends no one.

H. Simonis (Ed.): CPAIOR 2014, LNCS 8451, pp. 15–28, 2014.

In this paper I partially answer this question. I present a remarkably simple constraint model for SR, using $O(n^2)$ constraints. This model addresses SR with incomplete lists and consequently SM with incomplete lists. A more compact and computationally efficient encoding is then proposed. An empirical study is presented, comparing models and investigating the problem.

2 The Stable Roommates Problem (SR)

An example of a stable roommates instance is given in Figure 1, for $n = 10$, and this instance is taken from [7] (and we will refer to this as sr10). We have agents 1 to 10 each with a preference list, ranking the other agents. For example, $agent_1$'s first choice is for $agent_8$, then $agent_2$, followed by $agent_9$ and so on to last (9^{th}) choice $agent_{10}$.

```
 1 : 8 2 9 3 6 4 5 7 10
 2 : 4 3 8 9 5 1 10 6 7
 3 : 5 6 8 2 1 7 10 4 9
 4 : 10 7 9 3 1 6 2 5 8
 5 : 7 4 10 8 2 6 3 1 9
 6 : 2 8 7 3 4 10 1 5 9
 7 : 2 1 8 3 5 10 4 6 9
 8 : 10 4 2 5 6 7 1 3 9
 9 : 6 7 2 5 10 3 4 8 1
10 : 3 1 6 5 2 9 8 4 7
```

```
 1: 8 2 3 6 4 7
 2: 4 3 8 9 5 1 10 6
 3: 5 6 2 1 7 10
 4: 9 1 6 2
 5: 7 10 8 2 6 3
 6: 2 8 3 4 10 1 5 9
 7: 1 8 3 5
 8: 10 2 5 6 7 1
 9: 6 2 10 4
10: 3 6 5 2 9 8
```

```
(1,7) (2,3) (4,9) (5,10) (6,8)
(1,7) (2,8) (3,5) (4,9) (6,10)
(1,7) (2,8) (3,6) (4,9) (5,10)
(1,4) (2,8) (3,6) (5,7) (9,10)
(1,4) (2,9) (3,6) (5,7) (8,10)
(1,4) (2,3) (5,7) (6,8) (9,10)
(1,3) (2,4) (5,7) (6,8) (9,10)
```

Fig. 1. Stable roommates instance sr10 with $n = 10$ (on the left) phase-1 table (middle) and the 7 stable matchings (on the right). Instance taken from [7].

A quadratic time algorithm, essentially linear in the input size, was proposed in [8]. The algorithm has two phases. The first phase is a sequence of proposals, similar to that in the Gale Shapley algorithm [4], that results in the *phase-1 table*. The phase-1 table for sr10 is shown as the middle table in Figure 1. A sequence of *rotations* are then performed for agents with reduced preference lists that contain more than one agent. On the right hand side of Figure 1 we show the 7 stable matching that can result from this process.

3 A Simple Constraint Model

We assume that we have two dimensional integer arrays $pref$ and $rank$. Vector $pref_i$ is the preference list for $agent_i$ such that if $pref_{i,k} = j$ then $agent_j$ is $agent_i$'s k^{th} choice and $rank_{i,j} = k$. It is also assumed that if $agent_i$ finds $agent_j$ acceptable then $agent_j$ finds $agent_i$ acceptable (i.e. j appears in $pref_i$ if and only if i appears in $pref_j$). We also have the length of each agent's preference list l_i, and this allows us to model SRI instances, i.e. **S**table **R**oommates with **I**ncomplete lists.

Using sr10 as our example $pref_{3,1} = 5$ and $rank_{3,5} = 1$ ($agent_5$ is $agent_3$'s first choice) and $pref_{3,2} = 6$ and $rank_{3,6} = 2$ ($agent_6$ is $agent_3$'s second choice). Note that in sr10 $l_i = 10$ for all i, i.e. sr10 is an SR instance with complete preference lists.

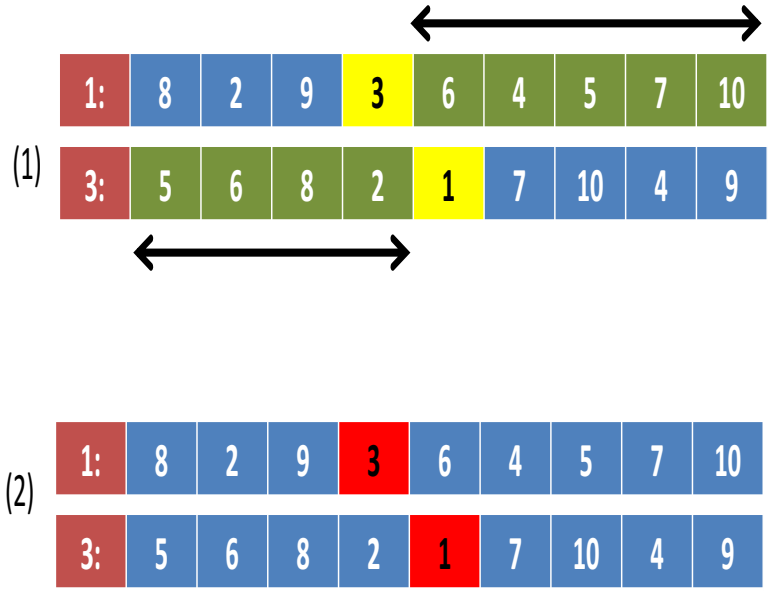

Fig. 2. A pictorial representation of the two constraints acting between $agent_1$ and $agent_3$

We have constrained integer variables a_1 to a_n, each with a domain of *ranks* $\{1..l_i+1\}$. When $a_i \leftarrow k$ this means that the corresponding agent is allocated his k^{th} choice, and that is $agent_j$ where $j = pref_{i,k}$. Furthermore, when $a_i \leftarrow l_i + 1$ the corresponding agent is matched to itself and is considered unallocated.

We can now make a declarative statement of the properties that a stable matching must have and we do this with two constraints. Given two agents, $agent_i$ and $agent_j$ who find each other acceptable, if $agent_i$ is matched to an agent he prefers less than $agent_j$ then $agent_j$ must match up with an agent that he prefers to $agent_i$ otherwise the matching will be unstable. This property must hold between every pair of agents that find each other acceptable and is expressed by constraint (2) below. Furthermore, when $agent_i$ is matched to $agent_j$ then $agent_j$ is matched to $agent_i$, and this is expressed by constraint (3).

$$\forall_{i\in[1..n]}\ a_i \in \{1..l_i + 1\} \tag{1}$$

$$\forall_{i\in[1..n]}\forall_{j\in pref_i}\ a_i > rank_{i,j} \implies a_j < rank_{j,i} \tag{2}$$

$$\forall_{i\in[1..n]}\forall_{j\in pref_i}\ a_i = rank_{i,j} \implies a_j = rank_{j,i} \tag{3}$$

This constraint is shown pictorially in Figure 2. The two constraints are shown for agents 1 and 3 in sr10. The brown box is the agent's identification number and the remaining boxes are the preference lists (a list of agents). In the top picture (1) we have the situation where $agent_1$ is matched to an agent he prefers

less than $agent_3$, i.e. $agent_1$ is matched to an agent in the green part of his preference list. Consequently $agent_3$ must be matched in the green region of its preference list. The bottom picture is for constraint (3) where $agent_1$ is matched to $agent_3$, both taking the pair of red values.

A similar constraint model was proposed for SM [12,17]. Establishing arc-consistency [10,19] in that simple SM constraint model has been shown to be $O(n^3)$ although at least three $O(n^2)$ encodings have been proposed: one using boolean variables [6], one using 4-valued variables [12] and one using a specialized n-ary constraint [17].

When sr10 is made arc-consistent the phase-1 table is produced. As we can see from Figure 1 the first agent in the phase-1 table for $agent_1$ is $agent_8$ yet none of the seven solutions have a matching that contains the pair $(1, 8)$. Therefore our constraint program must backtrack, i.e. after producing the phase-1 table via propagation, search instantiates $a_1 \leftarrow 1$ (assigned 1^{st} preference), attempts to make the model arc-consistent and fails, forcing a backtrack. To find a first solution to sr10 (a first matching) the constraint program makes 3 decisions, at least one of which results in a backtrack. To find all 7 solutions, 12 decisions are made.

```
1  public class StableRoommates {
2
3   public static void main(String[] args) throws IOException {
4
5      BufferedReader fin = new BufferedReader(new FileReader(args[0]));
6      int n = Integer.parseInt(fin.readLine());
7      int[][] pref = new int[n][n];
8      int[][] rank = new int[n][n];
9      int[] length = new int[n];
10     for (int i=0;i<n;i++){
11     StringTokenizer st = new StringTokenizer(fin.readLine()," ");
12     int k = 0;
13     length[i] = 0;
14     while (st.hasMoreTokens()){
15          int j = Integer.parseInt(st.nextToken()) - 1;
16          rank[i][j] = k;
17          pref[i][k] = j;
18          length[i]  = length[i] + 1;
19          k = k + 1;
20     }
21     rank[i][i] = k;
22     pref[i][k] = i;
23     }
24     fin.close();
25     Model model = new CPModel();
26     IntegerVariable[] a = new IntegerVariable[n];
27     for (int i=0;i<n;i++) a[i] = makeIntVar("a_"+ i,0,length[i],"cp:enum");
28     for (int i=0;i<n;i++)
29     for (int j=0;j<length[i];j++){
30          int k = pref[i][j];
31          model.addConstraint(implies(gt(a[i],rank[i][k]),lt(a[k],rank[k][i])));
32          model.addConstraint(implies(eq(a[i],rank[i][k]),eq(a[k],rank[k][i])));
33     }
34     Solver solver = new CPSolver();
35     solver.read(model);
36     if (solver.solve().booleanValue())
37       for (int i=0;i<n;i++){
38      int j = pref[i][solver.getVar(a[i]).getVal()];
39      if (i<j) System.out.print("("+ (i+1) +","+ (j+1) +") ");
40        }
41     System.out.println();
42   }
43 }
```

Listing 1. A simple encoding for SRI, StableRoommates.java

The model was implemented in the choco constraint programming toolkit [1] using Java and the code is shown in Listing 1. The first thing to note is that everything is zero-based, such that the first agent is a_0 and the last a_{n-1} (lines 26

```
 1: 8 2 9 3 6 4 5 7
 2: 4 3 8 9 5 1 10 6
 3: 5 6 8 2 1 7 10
 4: 9 3 1 6 2
 5: 7 4 10 8 2 6 3
 6: 2 8 7 3 4 10 1 5 9
 7: 1 8 3 5
 8: 10 4 2 5 6 7 1
 9: 6 7 2 5 10 3 4
10: 3 1 6 5 2 9 8
```

Fig. 3. *Bound* phase-1 table for sr10 using bound integer variables

and 27). Lines 5 to 24 read in the problem instance, building the arrays $pref$ and $rank$. To address SR with incomplete lists we add i to the end of a_i's preference list (lines 21 and 22) such that an unmatched agent is matched to itself. The constraint model is produced in lines 25 to 35 with constraint (2) posted in line 31 and constraint (3) in line 32. In lines 36 to 41 the choco toolkit searches for a first solution and prints it out.

The choco toolkit also supports bound integer variables, where only the upper and lower bounds on domains are maintained and removal of values between those bounds are performed lazily. In line 27 of Listing 1 adding the option "cp:bound" to the constructor $makeIntVarArray$ changes the model so that it uses bound integer variables. When the model is made arc-consistent we then get the *bound* phase-1 table shown in Figure 3. Comparing this to Figure 1 we see that the upper and lower bounds agree with the phase-1 table but there are values between those bounds that are omitted from the enumerated domains, in particular we see that $agent_1$ has $agent_9$ in its domain yet $agent_9$ does not have $agent_1$ in its domain. Nevertheless, the constraint program maintains the desired stable roommates properties and produces the same 7 solutions as in Figure 1 and does so in less time.

The constraint model also address SRI instances (SR with incomplete lists). Figure 4 shows instance sri6, with $n = 6$. This has one stable matching $\{(1,4),(2,6)\}$ with agents 3 and 5 unmatched.

```
1: 2 4 5
2: 6 1 3
3: 2 4
4: 1 6 3
5: 6 1
6: 2 5 4
```

Fig. 4. SRI instance sri6. This has one solution $\{(1,4),(2,6)\}$.

The model also addresses stable marriage problems with complete and incomplete lists (i.e. SM and SMI). As an example consider Figure 5, a stable marriage instance with 6 men and 6 women (taken from [6]). This is shown on the left of Figure 5 with a stable matching in bold font. On the right we have the same problem represented as an SRI instance. The men are represented as agents 1 to 6 and women as agents 7 to 12. Agents 1 to 6 (the men) only find agents 7 to 12 acceptable (the women) and agents 7 to 12 (the women) find only agents 1 to 6 (the men) acceptable. To read off the SRI matching we subtract 6 from the agent matched to agents 1 to 6. Therefore our simple constraint model addresses SR, SRI, SM and SMI.

1 : **1** 3 6 2 4 5	1 : **1** 5 6 3 2 4	1 : **7** 9 12 8 10 11
2 : 4 6 1 **2** 5 3	2 : **2** 4 6 1 3 5	2 : 10 12 7 **8** 11 9
3 : 1 **4** 5 3 6 2	3 : 4 3 **6** 2 5 1	3 : 7 **10** 11 9 12 8
4 : **6** 5 3 4 2 1	4 : 1 **3** 5 4 2 6	4 : **12** 11 9 10 8 7
5 : 2 3 1 4 **5** 6	5 : 3 2 6 1 4 **5**	5 : 8 9 7 10 **11** 12
6 : **3** 1 2 6 5 4	6 : 5 1 3 6 **4** 2	6 : **9** 7 8 12 11 10
		7 : **1** 5 6 3 2 4
		8 : **2** 4 6 1 3 5
		9 : 4 3 **6** 2 5 1
		10 1 **3** 5 4 2 6
		11 3 2 6 1 4 **5**
		12 5 1 3 6 **4** 2

Fig. 5. Stable marriage instance sm6. On the left, the familiar SM and on the right sm6 recast as an SRI instance. Problem is taken from [6].

4 A More Efficient Model

Our constraint model can be made more computationally efficient by adopting and modifying the models in [6,12]. However, these models are bulky and quickly exhaust memory on relatively modest sized instances of SM [17]. Therefore we propose an n-ary SR constraint (SMN), similar to that proposed in [17], that can establish arc-consistency in $O(n^2)$ and takes $O(n)$ additional space (assuming we are given the arrays $pref$ and $rank$ read in on lines 5 to 24 of Listing 1). The means of reducing the computational cost is by eliminating the redundancies brought about by the arc-consistency algorithm: when a variable's domain is altered all constraints involving that variable are revised. Therefore, if a value is removed from the domain of a_i, $O(n)$ constraints will be revised. This can occur n times for an agent, and since there are n agents this results in $O(n^3)$ complexity, assuming it takes $O(1)$ time to revise a constraint as above.

With a specialized n-ary constraint we can improve upon this. We can eliminate the above redundancy by revising only the domains of agents that must be affected by a change in another variable's domain. There are five possible changes that can occur to the domain of an agent and these are:

- the upper bound of a variable decreases (Algorithm 1)
- the lower bound of a variable increases (Algorithm 2)
- a variable looses a value (Algorithm 3)
- a variable is instantiated (Algorithm 4)
- the constraint is initially posted (Algorithm 5)

Presented below are the algorithms that address these five cases and the actual choco/Java implementation (Listing 2, with imports removed for brevity). The algorithms again assume that we have constrained integer variables a_1 to a_n, each with a domain of ranks $\{1 \ldots l_i + 1\}$, and that we have the preference and rank arrays $pref$ and $rank$. In addition we require *reversible* variables lwb_i and upb_i, where lwb_i is used to store the smallest value in the domain of a_i and upb_i the largest value. By *reversible* we mean that on backtracking the values of these variables are restored. The choco toolkit provides this as class $StoredInt$ (see lines 19 and 20 of Listing 2). In the complexity arguments we assume that the toolkit primitives $getMin(v)$ (get the smallest value in domain of variable v), $setMax(v, x)$ (set the upper bound of variable v's domain to be $min(max(v), x)$), $getMax(v)$ (get largest value in domain of v), $remove(v, x)$ (remove the value x from the domain of v if that value exists) and $getValue(v)$ (get the value v is instantiated to) each have a cost of $O(1)$.

deltaMin(i) (Algorithm 1). The lower bound of a_i has increased (and is now the value x, line 3). Consequently, the corresponding agent now at the top of $agent_i$'s preference list ($agent_j$ where $j = pref_{i,x}$, line 4)) can be matched to no one that he prefers less than $agent_i$ (line 5). For the corresponding agents that have been removed from $agent_i$'s preference list, and that $agent_i$ preferred to his current most preferred partner, those agents can do no worse than match up with agents that they prefer to $agent_i$ (lines 6 to 8). The new lower bound for a_i is saved in the reversible variable lwb_i. **Complexity:** This method can be called at most n times for an agent (the number of values in an agent's domain). Each time it is called the loop bound (line 6) is reduced (via line 9 on previous calls). Consequently this can reduce the maximum domain value of other agents (line 5 and line 8) at most n times. Therefore over all agents the cost of $deltaMin$ is $O(n^2)$.

deltaMax(i) (Algorithm 2). The upper bound of a_i has decreased (and now has the value x, line 3). For all corresponding agents removed from $agent_i$'s preference list we remove $agent_i$ from that agent's preference list as they can no longer be matched together (lines 4 to 6). The new upper bound is then saved in the reversible variable upb_i (line 7). **Complexity:** For an agent, this method can be called at most n times, each time with a reduced bound on the iteration in lines 4 to 6. Therefore lines 3 and 6 can be executed at most n times. Consequently the cost over all n agents is $O(n^2)$.

removeValue(i,x) (Algorithm 3). The value x has been removed from the domain of a_i consequently the corresponding agent ($agent_j$ where $j = pref_{i,x}$)

Algorithm 1. deltaMin (awakeOnInf in Listing 2).

```
1 deltaMin(int i)
2 begin
3     x ← getMin(a_i)
4     j ← pref_{i,x}
5     setMax(a_j, rank_{j,i})
6     for w ← lwb_i to x − 1 do
7         h ← pref_{i,w}
8         setMax(a_h, rank_{h,i} − 1)
9     lwb_i ← x
```

Algorithm 2. deltaMax (awakeOnSup in Listing 2).

```
1 deltaMax(int i)
2 begin
3     x ← getMax(a_i)
4     for y ← x + 1 to upb_i do
5         j ← pref_{i,y}
6         remove(a_j, rank_{j,i})
7     upb_i ← x
```

can no longer be matched to $agent_i$ (lines 3 and 4). **Complexity:** An execution is $O(1)$ cost and this can happen at most $O(n^2)$ times, i.e. n times for each of the n agents.

Algorithm 3. removeValue (awakeOnRem in Listing 2).

```
1 removeValue(int i, int x)
2 begin
3     j ← pref_{i,x}
4     remove(a_j, rank_{j,i})
```

instantiate(i) (Algorithm 4). The variable a_i has been assigned the value y (line 3) and corresponds to being matched to $agent_j$ where $j = pref_{i,y}$. All agents that $agent_i$ preferred to $agent_j$ can only be matched to agents that they prefer to $agent_i$ (lines 4 to 6). Furthermore, all agents that $agent_i$ preferred less than $agent_j$ can no longer consider $agent_i$ as a possible partner (lines 7 to 9). Finally we update the upper and lower bounds for the domain (lines 10 and 11). **Complexity:** An execution has a cost of $O(n)$ as we respond to the (at most $n-1$) removals from the domain of the variable (lines 4 to 9). An agent can be

Algorithm 4. instantiate (awakeOnInst in Listing 2).

```
1  instantiate(int i)
2  begin
3      y ← getValue(a_i)
4      for x ← lwb_i to y − 1 do
5          j ← pref_{i,x}
6          setMin(a_j, rank_{j,i} − 1)
7      for z ← y + 1 to upb_i do
8          j ← pref_{i,z}
9          remove(a_j, rank_{j,i})
10     lwb_i ← y
11     upb_i ← y
```

instantiated with a value at most once during propagation. Consequently, over all n agents this has a cost of $O(n^2)$.

init() (Algorithm 5). This is called at the top of search, when the model is made arc-consistent by revising all the constraints. First, the upper and lower bounds for each agent are initialized (lines 2 to 4) and then propagation kicks off by making all agents consistent with respect to their most preferred partner, and this is similar to the proposal stage in [8]. **Complexity:** Line 6 is called n times and each individual call to $deltaMin(i)$ has cost $O(n)$, consequently we have an $O(n^2)$ cost in total.

Algorithm 5. init (class constructor and awake in Listing 2).

```
1  init()
2  begin
3      for i ← 1 to n do
4          lwb_i ← 1
5          upb_i ← length_i + 1
6      for i ← 1 to n do
7          deltaMin(i)
```

5 Empirical Study

Experiments were performed over random SR instances with complete preference lists, on a 2.4GHz Intel Xeon E5645 processor with 97 GBytes of RAM, using java version 1.6.0_26 and choco-2.1.0. We start by investigating the three models:

```
1
2  public class SRN extends AbstractLargeIntSConstraint {
3
4      private int n;
5      private int[][] rank;
6      private int[][] pref;
7      private int[] length;
8      private IStateInt[] upb;
9      private IStateInt[] lwb;
10     private IntDomainVar[] a;
11
12     public SRN(Solver s,IntDomainVar[] a,int[][] pref,int[][] rank,int[] length){
13         super(a);
14         n            = a.length;
15         this.a       = a;
16         this.pref    = pref;
17         this.rank    = rank;
18         this.length  = length;
19         upb          = new StoredInt[n];
20         lwb          = new StoredInt[n];
21         for (int i=0;i<n;i++){
22             upb[i]  = s.getEnvironment().makeInt(length[i]);
23             lwb[i]  = s.getEnvironment().makeInt(0);
24     }
25     }
26
27     public void awake() throws ContradictionException {
28         for (int i=0;i<n;i++) awakeOnInf(i);
29     }
30
31     public void propagate() throws ContradictionException {}
32
33     public void awakeOnInf(int i) throws ContradictionException {
34         int x = a[i].getInf(); // best (lowest) rank for a i
35         int j = pref[i][x];
36         a[j].setSup(rank[j][i]);
37         for (int w=lwb[i].get();w<x;w++){
38             int h = pref[i][w];
39             a[h].setSup(rank[h][i]-1);
40     }
41     lwb[i].set(x);
42     }
43
44     public void awakeOnSup(int i) throws ContradictionException {
45         int x = a[i].getSup(); // worst (largest) preference for a[i]
46         for (int y=x+1;y<=upb[i].get();y++){
47             int j = pref[i][y];
48             a[j].remVal(rank[j][i]);
49     }
50     upb[i].set(x);
51     }
52
53     public void awakeOnRem(int i,int x) throws ContradictionException {
54         int j = pref[i][x];
55         a[j].remVal(rank[j][i]);
56     }
57
58     public void awakeOnInst(int i) throws ContradictionException {
59         int y = a[i].getVal();
60         for (int x = lwb[i].get();x<y;x++){
61             int j = pref[i][x];
62             a[j].setSup(rank[j][i]-1);
63     }
64     for (int z=y+1;z<=upb[i].get();z++){
65         int j = pref[i][z];
66         a[j].remVal(rank[j][i]);
67     }
68     lwb[i].set(y);
69     upb[i].set(y);
70     }
71 }
```

Listing 2. SRN.java

(a) SR, the simple constraint model, (b) SRB, the simple model using bound integer variables and (c) SRN, the n-ary constraint model. In all cases a sample size of 100 is used, unless stated otherwise.

Figure 6 presents two scatter plots of total run time against problem size. The plot on the left is for $10 \leq n \leq 100$ and on the right $100 \leq n \leq 1,000$ (and for $n > 100$ we omit SR). The total run time includes time to read in the problem, build the model and then find all stable matchings for that instance. Time is measured in milliseconds. The total run times shows that SR does not scale beyond $n = 100$ (plot on the left) and at $n = 1,000$ (plot on the right) SRB typically takes 4 minutes whereas SRN takes 2 seconds, i.e. SRN is two orders of magnitude faster.

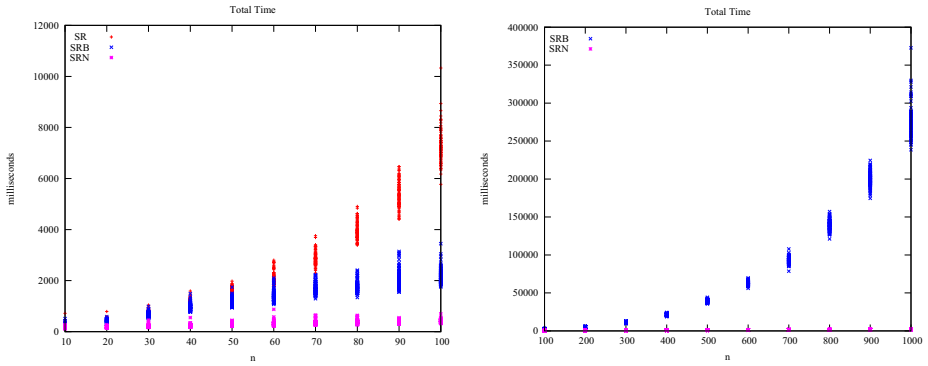

Fig. 6. Performance of the models: scatter of total time in milliseconds to find all matchings against problem size

We now investigate the problems, i.e. given n what proportion of instances have matchings? Shown in Table 1 are the proportion of instances with matchings, for $n \in \{10 \ldots 90\}$ with a sample size of 1,000. The column on the right are those reported in [8], with a sample size of 1,000 for n equal to 10 and 20, sample size 500 for $n = 30$, and sample size 200 for $40 \leq n \leq 90$.

In Table 2 we give the average total cpu time in seconds (i.e. time to read in the instance, produce the model, enumerate all solutions and output run time statistics) for $100 \leq n \leq 1,000$ using our best model (SRN). Also tabulated is the average number of nodes reported by the choco toolkit (and maximum in brackets) where a node is a decision made, and that decision might be one that leads to a failure and a backtrack. The second last column is the proportion of instances that had matchings. The last column is the maximum number of stable matchings found in an instance of size n. In all cases sample size is 100.

5.1 Discussion

Clearly (Figure 6) the n-ary encoding is orders of magnitude faster than the toolkit constraints. Although not presented, it is also more space efficient, i.e.

Table 1. Proportion of instances with solutions. Column on the right from [8].

n	Prosser	Irving
10	0.889	0.868
20	0.834	0.815
30	0.781	0.766
40	0.736	0.745
50	0.727	0.710
60	0.704	0.725
70	0.706	0.670
80	0.670	0.675
90	0.670	0.690

Table 2. Average total run times in seconds to enumerate all matchings using SRN, the average number of decisions (nodes) made by choco (maximum in brackets), the proportion of instances with stable matchings and the maximum number of matchings in an instance. Sample size is 100.

n	cpu time	nodes	matched	max matchings
100	0.423	4 (17)	0.63	9
200	0.511	6 (34)	0.52	16
300	0.645	7 (33)	0.53	16
400	0.768	7 (25)	0.38	10
500	0.950	7 (35)	0.45	16
600	1.094	7 (27)	0.41	14
700	1.290	7 (31)	0.42	12
800	1.555	8 (50)	0.44	24
900	1.786	8 (29)	0.39	12
1,000	2.046	8 (85)	0.40	40

it has a more compact model and this can be quickly constructed. Therefore it wins on two fronts: space and time.

The proportion of SR instances with matching was first investigated in [8] and later in [14] and [13]. Empirical evidence has been based on translations of the Pascal code given in the appendix of Irving's paper. Unfortunately that code has a bug and on occasion fails to find a matching when one exists. This has been observed by Stephan Mertens and independently by Ciaran McCreesh. Consequently, earlier reported results may be incorrect. The results in Table 1 use a sample size of 1,000 and might be assumed to be more accurate than those in Rob Irving's original study.

In Table 2 we have the average (and maximum) number of nodes required to find all matchings. This number is always low, and always less than 3 times the number of maximum matchings. More to the point, the model never exhibited exponential behaviour. As yet I have no explanation of why this is so, i.e. why the constraint model is so well behaved.

There are hard variants of SR. One example is egalitarian SR where a matching is to be found that minimizes the sum of the ranks, and this has been shown to be NP-hard [9]. In our constraint model an egalitarian matching is one that minimizes $\sum a_i$. Therefore we can model this problem by adding one more variable ($totalCost$), one more constraint ($totalCost = \sum a_i$) and a change from solving to minimization (line 36 of Listing 1). Naively, to find an egalitarian matching we could consider all matchings. As we see from Table 2 no instance had more than 40 matchings, no search took more than 85 nodes and the longest run time (not tabulated) was 2.6 seconds. Therefore, although NP-hard we would fail to encounter a hard instance in the problems sampled. So, (as Cheeseman, Kanefsky and Taylor famously asked [3]) where are the hard problems? As yet I do not know.

6 Conclusion

It has been demonstrated that there is a simple constraint model for the stable roommates problem. It was demonstrated that arc-consistency on this model produces the phase-1 table in $O(n^3)$ time. A backtracking search that maintains arc-consistency on each decision allows us to enumerate all matching. However, it was shown that the search process can make decisions that lead to failure. The simple model was enhanced by using bound, rather than enumerated constrained integer variables and arc-consistency delivers a *bound* phase-1 table. Nevertheless, this results in a substantial improvement in performance but the complexity of producing the phase-1 table remains $O(n^3)$. This lead to a specialized n-ary constraint with $O(n^2)$ cost for arc-consistency. Empirical study showed that this model can enumerate all matching to problems with 1,000 agents in about 2 seconds, orders of magnitude faster than the simple model.

It has also been shown that since our constraint model addresses incomplete preference lists it can also model stable marriage problems with complete and incomplete preference lists. That is, one model suffices.

Our model behaved well, never exhibiting exponential behaviour. Therefore there is work to do, to prove that the amount of backtracking is in some sense bounded by a polynomial, and this proof might be similar to that of failure-free enumeration in SM [6].

One of the first hard variants is egalitarian SR. This can be easily modeled and explored. However, it appears that it might be uninteresting. For $n \leq 1,000$ the number of matchings that need to be explored appears to be small. Furthermore, as n increases we expect that the number of instances with matchings will also fall [14,13]. Combined, this suggests that although NP-hard, egalitarian SR is easy.

All the code used in this study is available at [2].

Acknowledgements. I would like to thank Augustine Kwanashie, Ciaran McCreesh, David Manlove, Rob Irving and Ian Gent.

References

1. choco constraint programming system, http://choco.sourceforge.net/
2. Stable Roommates, http://www.dcs.gla.ac.uk/~pat/roommates/distribution
3. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are, pp. 331–337. Morgan Kaufmann (1991)
4. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. American Mathematical Monthly 69, 9–15 (1962)
5. Gale, D., Sotomayor, M.: Some remarks on the stable matching problem. Discrete Applied Mathematics 11, 223–232 (1985)
6. Gent, I.P., Irving, R.W., Manlove, D.F., Prosser, P., Smith, B.M.: A constraint programming approach to the stable marriage problem. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 225–239. Springer, Heidelberg (2001)
7. Gusfield, D., Irving, R.W.: The Stable Marriage Problem: Structure and Algorithms. The MIT Press (1989)
8. Irving, R.W.: An efficient algorithm for the "stable roommates" problem. J. Algorithms 6(4), 577–595 (1985)
9. Irving, R.W.: Optimal Stable Marriage. In: Encyclopedia of Algorithms. Springer (2008)
10. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 8, 99–118 (1977)
11. Manlove, D.: Algorithmics of Matching under Preferences. Theoretical Computer Science, vol. 2. World Scientific (2013)
12. Manlove, D.F., O'Malley, G.: Modelling and solving the stable marriage problem using constraint programming. In: Proceedings of the Fifth Workshop on Modelling and Solving Problems with Constraints, held at the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 10–17 (2005)
13. Mertens, S.: Random stable matchings. In: Journal of Statistical Mechanics: Theory and Experiments (2005)
14. Pittel, B., Irving, R.W.: An upper bound for the solvability of a random stable roommates instance. Random Struct. Algorithms 5(3), 465–487 (1994)
15. Roth, A.E.: The evolution of the labor market for medical interns and residents: a case study in game theory. Journal of Political Economy 92(6), 991–1016 (1984)
16. Roth, A.E., Sotomayor, M.A.O.: Two-sided matching: a study in game-theoretic modeling and analysis. Econometric Society Monographs, vol. 18. Cambridge University Press (1990)
17. Unsworth, C., Prosser, P.: An n-ary constraint for the stable marriage problem. In: Proceedings of the Fifth Workshop on Modelling and Solving Problems with Constraints, held at the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005) (2005)
18. Unsworth, C., Prosser, P.: A specialised binary constraint for the stable marriage problem. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 218–233. Springer, Heidelberg (2005)
19. van Hentenryck, P., Deville, Y., Teng, C.-M.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57, 291–321 (1992)